Meltzer
& Michie

Machine Intelligence 7

Edinburgh

MACHINE INTELLIGENCE 7

MACHINE INTELLIGENCE 7

edited by
BERNARD MELTZER
Department of Computational Logic
University of Edinburgh
and
DONALD MICHIE
Department of Machine Intelligence
University of Edinburgh

with a preface by
LORD BOWDEN

EDINBURGH at the University Press

© 1972 Edinburgh University Press
22 George Square, Edinburgh

ISBN 0 85224 234 4

Printed in Great Britain at
The Aberdeen University Press

Library of Congress
Catalog Card Number
72-7980

PREFACE

'Machine Intelligence' has now been actively studied for several years and has attracted the interest and support of some extremely able men. The problem I wish to pose to you is this. At this particular moment in time, is it reasonable to divert intellectual resource into an enterprise which, if I may presume to say so, has been disappointingly slow to produce significant results? There is no doubt at all of the rapid development of computers and of their associated software, but is this enterprise of yours part of the main stream development of computers, or is it merely an interesting, but irrelevant, side line?

Perhaps I may begin by speaking briefly about the developments which have taken place in the machines themselves during the last few years. I think it is worth reminding ourselves that thirty-five years ago, at the beginning of the last war, virtually the whole of engineering, of commerce and of administration depended on the use of hand-operated calculating machines and very complicated manual operations performed with simple instruments (like pens) which had not changed significantly since the time of classical Athens. Nevertheless, a few punch card machines, a few desk calculators, and a few very good mathematical tables had speeded up ordinary computation by a factor of four or five hundred times as compared with the speed which could be achieved by the ancient astronomers or even by Kepler himself. May I remind you that he had to work out all his original calculations by brute force and that it was relatively late in his life that he was able to exploit the new mathematical tables which had just been invented by Briggs. Society as we knew it forty years ago depended entirely on the achievement of a couple of millenia in increasing the speed of ordinary arithmetical operations.

The first computers to be installed in Cambridge, Manchester and Toronto some twenty years ago were about five hundred times as fast as an ordinary operator using calculating machines and log tables. In other words, the achievements of a millenium were paralleled by the achievements of a decade.

Since then we have had Mercury and Atlas. No one can dispute that Atlas was a thousand times as fast as the very old original Mark One machine which we were so proud of twenty years ago. We are about to instal a machine in Manchester which will be twenty times as powerful as Atlas. How much further this process can go is far from clear, because, of course,

the speed of calculation is now limited by the speed of propagation of pulses down wires, in other words, by the speed of light. But the point I wish to make is that we can now calculate many thousands of times as fast as we could in 1953 and at least a million times as fast as we could three hundred years ago.

Now this change is quite extraordinary, if one compares it for example with the increase in the speed of travel. A satellite orbiting the earth or moving towards the planets is unlikely to go much faster than twenty-five or thirty thousand miles an hour. An ordinary man can usually do two and a half or three, so that the satellite is perhaps ten thousand times as fast as a walking man. The enormous increase in speed of travel has changed our world and our ideas of the potentially possible. We don't use satellites to go from Manchester to Edinburgh in a few minutes, but we hope to explore the solar system. Few people have made similarly realistic and visionary forecasts of the effect of our modern speed of computation.

The memory of machines has increased many thousand fold and so has the speed by which one can get access to the information in the stores; at the same time the cost of storing a binary digit has come down at least a hundred to one, while the reliability of the machines has increased even more dramatically!

It is in fact now quite possible to put the whole of the contents of the *Encyclopaedia Brittanica* on a largish spool of tape (this is about a thousand million binary digits), and one can imagine a computing installation which had immediately available to it a very significant part of all the information in the world in so far as it has ever been produced in print.

May I say again that few people appreciate just how powerful these machines can be when they undertake the tasks for which they were designed.

On the other hand, as I said myself about twenty years ago, a machine is unlikely to be able to answer satisfactorily such a question as this. 'If a man of twenty can gather ten pounds of blackberries in a day and a girl of eighteen can gather nine, how many will they gather if they go out together?' And it is problems like this which dominate the thinking of ordinary human beings.

But astonishing though the achievements of the engineers have been, I believe that the achievements of the mathematicians who have made it possible for ordinary human beings to use them have been even more dramatic and are even more astonishing.

When I first interested myself in the subject, there were very few programmers in the world, many of whom were outstandingly good mathematicians, and I think I knew most of them. Today there are hundreds of thousands of them and any intelligent schoolboy can learn to program a machine and solve problems which would have defeated some of the ablest men of their generation twenty years ago.

I think that there is a very close parallel between the achievements of the men who have created the languages and the compilers which made com-

puters available to ordinary people, and the achievements of a few great mathematicians in the eighteenth and nineteenth centuries who made Newton's work accessible to ordinary mortals. That very great mathematician Euler spent many years of his life in simplifying the calculus so that it could be used by schoolboys. When Newton left the subject it was obscure and almost universally misunderstood; but Euler created the A-level mathematics which schoolboys learned when I was young, and I have always thought that his achievements were among the most important and underestimated in the history of mathematics. I believe, furthermore, that the achievements of a dozen or so men who have invented and developed the languages of computers have been equally remarkable and must rank with the most extraordinary achievements of which human beings can boast.

We have enormously powerful machines and extremely ingenious languages. What then are the problems which make us need these complicated machines?

In the first place, of course, they have revolutionised engineering calculations and have made it possible to introduce analytical mathematical techniques in the design of vastly complex structures such as aeroplanes and atomic power stations which could never have been built without them. Furthermore, it had become obvious twenty years ago that clerical work was becoming more and more expensive, more and more difficult to do and more and more necessary in modern society. Industry, banking, commerce and government depend more and more on numbers which computers can handle and which ordinary clerks no longer need to study or to process. It is no longer true to say, as it once was, that the number of clerks is increasing more rapidly than that of any other category known to the Census. I think there are more hairdressers today than there are steel makers! – but that is by the way.

However, we still have many unsolved problems. For example, the fast increasing rate of accumulating and publishing knowledge is beginning to destroy the very mechanism which was designed to foster it. There was a time when a few people wrote books and papers, and thousands of people read them and knowledge was disseminated thereby. Today more people seem to be engaged in writing books than there are who read them; the whole process of publication is getting more and more expensive and it is producing ever diminishing returns. If six people write a paper and only one man reads it, it can hardly be considered to be a profitable enterprise on the part of authors or readers, or even publishers.

The number of books in university libraries increases all the time – it seems to double every eight years or so – but the number of books which are actually read depends on the speed and devotion of individual readers. In other words, the fraction of books in a library which remains unread on the shelves increases all the time, and in some libraries it must already be well over ninety per cent. In many libraries, including our own, it would be cheaper to give away the books that the students actually read instead of keeping

them on the shelves until they want them, but librarians seem strangely reluctant to accept the implications of this argument. It is analogous to arguments which suggest that it would be cheaper to *give* cars to students instead of providing them with parking spaces. The point I want to make is that the classical techniques for storing and disseminating information are collapsing as we watch them and no one has ever faced the basic problems of serendipity as a technique for uncoordinated research. Here then are problems in search of solution; in the computers and their extraordinary software we may have solutions in search of problems.

May I indulge myself in a personal reminiscence? I shall never forget one interview I had some forty years ago with that great man Professor Lord Rutherford for whom I worked in the Cavendish for three very exciting years. He came into the lab, hung his coat on the main switch and gave himself a minor electric shock, and then lit his pipe. He always dried his tobacco on a radiator before he used it, so his pipe burned with occasional explosions and bright blue flames, furthermore it emitted showers of sparks from time to time. His students were always intimidated and watched this operation very anxiously; they wondered if he would set fire to his moustache. He began in that great booming voice of his: 'The time has come, my boy, to decide if the results you are getting from your researches are really worth the price of the liquid air they are costing the Cavendish'! Rather unwillingly he decided that they might be, but ever since then I have displayed a certain scepticism about all research projects, and not only my own.

I have always hoped that your Society would concern itself with the marriage, or the liaison if you prefer it, between unsolved problems and unused solutions. How far can computers be used to study the apparently insoluble intellectual problems of the day? I have probably misunderstood your work, but I have been disheartened by some of the research in which your members are engaged. It is pressing against the limits of knowledge and this is splendid, but are you right to be so worried about problems which appear to me to be semantically significant rather than technically important, and philosophically interesting rather than economically useful? It is some years since people attempted to improve the speed of aircraft by improving the thermodynamic efficiency of horses, but I hope that our new expensive motorways will be suitable for horse-drawn traffic in a hundred years' time, when the oil runs out!

There is no doubt at all that it is extraordinarily difficult to understand the very complicated interactions and interlinkages in our own brains. It may be even that one can draw some analogy between the structure of a human cortex and that of a computer. If so it will be great fun, but will it help the design of machines or will it help to relieve the otherwise unendurable burden which scholars and administrators have to carry in these days of more and more complicated numerical analysis and more and more detailed recording of the details of our lives?

The concept of *gestalt* is hard enough to understand – why and how do humans recognise the squareness of squares, the straightness of lines, the 'e'-ness of a letter 'e', the beauty of a sunset or the charm of a melody, which was written for an orchestra and is whistled off-key by a butcher's boy? Heaven knows these problems defy solution – why therefore should one expect that they can be studied or solved or practised by computers? When Atlas was closed down last year after nearly a decade of use, it had no more intelligence than it had when first it was switched on – but it had done some astonishingly useful work in the meantime. In this life one should, I believe, play from strength and reinforce success whenever possible – I commend the idea to all of you.

LORD BOWDEN OF CHESTERFIELD
June 1972

CONTENTS

INTRODUCTION xiii

PREHISTORY

1 On Alan Turing and the origins of digital computers. B. RANDELL 3

PROGRAM PROOF AND MANIPULATION

2 Some techniques for proving correctness of programs which alter data structures. R. M. BURSTALL 23
3 Proving compiler correctness in a mechanized logic. R. MILNER and R. WEYHRAUCH 51

COMPUTATIONAL LOGIC

4 Building-in equational theories. G. D. PLOTKIN 73
5 Theorem proving in arithmetic without multiplication. D. C. COOPER 91
6 The sharing of structure in theorem-proving programs. R. S. BOYER and J S. MOORE 101
7 Some special purpose resolution systems. D. KUEHNER 117
8 Deductive plan formation in higher-order logic. J. L. DARLINGTON 129

INFERENTIAL AND HEURISTIC SEARCH

9 G-deduction. D. MICHIE, R. ROSS and G. J. SHANNAN 141
10 And-or graphs, theorem-proving graphs and bi-directional search. R. KOWALSKI 167
11 An approach to the frame problem, and its implementation. E. SANDEWALL 195
12 A heuristic solution to the tangram puzzle. E. S. DEUTSCH and K. C. HAYES, Jr. 205
13 A man-machine approach for creative solutions to urban problems. P. D. KROLAK and J. H. NELSON 241
14 Heuristic theory formation: data interpretation and rule formation. B. G. BUCHANAN, E. A. FEIGENBAUM, and N. S. SRIDHARAN 267

PERCEPTUAL AND LINGUISTIC MODELS

15 Mathematical and computational models of transformational grammar. JOYCE FRIEDMAN 293
16 Web automata and web grammars. A. ROSENFELD and D. L. MILGRAM 307
17 Utterances as programs. D. J. M. DAVIES and S. D. ISARD 325
18 The syntactic inference problem applied to biological systems. G. T. HERMAN and A. D. WALKER 341
19 Parallel and serial methods of pattern matching. D. J. WILLSHAW and O. P. BUNEMAN 357
20 Approximate error bounds in pattern recognition. T. ITO 369
21 A look at biological and machine perception. R. L. GREGORY 377

PROBLEM-SOLVING AUTOMATA

22 Some effects in the collective behaviour of automata. V.I. VARSHAVSKY 389
23 Some new directions in robot problem solving. R.E. FIKES, P.E. HART and N.J. NILSSON 405
24 The MIT robot. P.H. WINSTON 431
25 The Mark 1.5 Edinburgh robot facility. H.G. BARROW and G.F. CRAWFORD 465

INDEX 481

INTRODUCTION

Among the many properties ascribed to the magical number seven is that of marking out the significant epochs of a human life-span – seven years from birth to departure from the kindergarten, another seven to puberty, another seven to majority. The occasion of the Seventh International Machine Intelligence Workshop is perhaps an appropriate moment to take stock.

Views differ as to exactly which of the successive thresholds is the one on which machine intelligence research is now poised. But there is no mistaking the sense of transition, felt both by its practitioners, who claim that their field is at last attaining maturity, and in a rather different way by its interested spectators. The latter rightly point out that if maturation brings opportunity and new powers it also brings the obligation to earn a living. In an invited Address of bracing candour, Lord Bowden called on our profession to examine itself with this criterion in mind. We reproduce his Address as the Preface to this volume.

The seventh Workshop was also marked by a new and unexpected addition to the study of origins. Volume 5 contained A.M.Turing's previously unpublished essay 'Intelligent Machinery'; we here make available a recent work of scholarship by Professor Brian Randell which has cast entirely new light on the technical *milieu* in which Turing first arrived at his vision of intelligence in machines. In the course of his pertinacious study, Professor Randell approached the Prime Minister for access to papers still held under security classification. Although the request was not granted, Mr Heath and his advisers did not let the matter rest. In due course Professor Randell was informed that the Prime Minister had decided that an official record of the early electronic computer developed during the Second World War should be compiled in consultation with those who were concerned in the project. It will surely be a source of widespread gratification that Professor Randell's initiative has catalysed action to preserve the technical facts for the record.

Mention was made earlier of a sense of transition. Not only is there a new awareness of the obligation to earn a living, but there is an even keener awareness that this is far from being the whole point. A man may design a program to carry out a particular cognitive activity with the idea of making it as efficient as possible at this task; or another man may write a program to carry out the same cognitive task, with the idea that it may be suggestive for understanding the way human beings carry out that task. It is important

that the desirability of interaction with social needs and with other sciences should not obscure the obvious fact that *both men are engaged in the same activity*. In the last decade or two a fundamental new discipline has emerged as a result of the creation of the digital computer, namely the experimental study of cognitive activity in the abstract.

The success of the Workshops is dependent upon benefits cheerfully volunteered, too numerous to acknowledge individually. Mention must, however, be made of the University of Edinburgh's hospitality in arranging the Workshop Lunch, presided over by its Principal, Sir Michael Swann; of the combination of *panache* and *punctilio* with which Mr Archie Turnbull, Secretary to the University Press, has once more officiated over the annual miracle of fast publication; and of the welcome financial support received from the Science Research Council.

B. MELTZER
D. MICHIE
October 1972

PREHISTORY

1

On Alan Turing and the Origins of Digital Computers

B. Randell

Computing Laboratory
University of Newcastle upon Tyne

Abstract

This paper documents an investigation into the role that the late Alan Turing played in the development of electronic computers. Evidence is presented that during the war he was associated with a group that designed and built a series of special purpose electronic computers, which were in at least a limited sense 'program controlled', and that the origins of several post-war general purpose computer projects in Britain can be traced back to these wartime computers.

INTRODUCTION

During my amateur investigations into computer history, I grew intrigued by the lack of information concerning the rôle played by the late Alan Turing. I knew that he was credited with much of the design of the ACE computer at the National Physical Laboratory starting in 1945. His historic paper on computability, and the notion of a universal automaton, had been published in 1936, but there was no obvious connection between these two activities. The mystery was deepened by the fact that, though it was well known that his war-time work for the Foreign Office, for which he was awarded an OBE, concerned code-breaking, all details of this remained classified. As my bibliography on the origins of computers grew, I came across differing reports about the design of the ACE, and about Turing, and I decided to try to solve the mystery. The purpose of this paper is to document the surprising (and tantalizing) results that I have obtained to date.

This paper is in the main composed of direct quotations from printed articles and reports, or from written replies to enquiries that I have made. It does not make explicit use of any of the many conversations that I have had. The layout of the paper follows the sequence of my investigation, but does not attempt to give a complete record of the unfruitful leads that I pursued. Furthermore, for reasons that will become clear to the reader, the

discussion of the later stages of my investigations has, deliberately, been left somewhat undetailed. A final section of the paper contains a brief discussion of the origins of the stored program concept.

THE START OF THE SEARCH

Official credit was given to Alan Turing as the originator of the design of ACE, in a Government press release (Anon 1946). The biography of Turing, written by his mother (Turing 1959) stated that he had proposed a computer design to the British Government, and that it was on the basis of this proposal that he joined NPL in 1945. The obituary notice for Turing (Newman 1955), written by Professor M. H. A. Newman, who was associated with the post-war computer developments at Manchester University, stated that:

> At the end of the war many circumstances combined to turn his attention to the new automatic computing machines. They were in principle realisations of the 'universal machine' which he had described in the 1937 paper for the purpose of a logical argument, though their designers did not yet know of Turing's work.

On the other hand an obituary notice (Menzler 1968) for E. W. Phillips (who before the war (Phillips 1936) had demonstrated a mechanical binary multiplier, now in the Science Museum, and proposed the building of a version based on the use of 'light-rays') claimed that Phillips and John Womersley of NPL had started to design ACE in 1943.

Faced with these statements, I realized that the question of where the ACE fitted into the chronology of electronic computers was not at all clear. Was it in fact the case that ENIAC, and the plans for EDVAC, were the sole source of the idea of first a program-controlled electronic digital computer, and then the stored-program concept?

At this stage I came across the following statement by Lord Halsbury (1949):

> [One of the most important events in the evolution of the modern computer was] a meeting of two minds which cross-fertilised one another at a critical epoch in the technological development which they exploited. I refer of course to the meeting of the late Doctors Turing and von Neumann during the war, and all that came thereof (von Neumann 1945; Burks, Goldstine and von Neumann 1946). In a sense, computers are the peace-time legacy of war-time radar, for it was the pulse techniques developed in the one that were applied so readily in the other.

I wrote to Lord Halsbury, but he could not recollect the source of his information (Halsbury 1971):

> I am afraid I cannot tell you more about the meeting between Turing and von Neumann except that they met and sparked one another off. Each had, as it were, half the picture in his head and the two halves came together during the course of their meeting. I believe both were working on the mathematics of the atomic bomb project.

From Dr Herman H. Goldstine, who was of course closely associated with the ENIAC and EDVAC projects, came the following comments (1971):

> The question about von Neumann and Turing as I remember it is like this: Turing came to the United States the first time in 1936. He went to Princeton and studied under Church from 1936–1938. There he was in Fine Hall, which is where the Institute for Advanced Study was housed at the time, so he met von Neumann. Von Neumann, as early as the mid-twenties, had been profoundly interested in formal logics and watched Turing's work with interest. In fact he invited Turing to be his assistant for one year. Turing turned this down and preferred to return to England and the war. There he spent the war period working for the Foreign Office. While I only know this in a very second-hand way, I believe he was working on cryptographic problems. . . . Womersley was heading up the Math work at the National Physical Laboratory during the latter part of the war when I first met him (I think he also was in the same Foreign Office group as Turing early in the war and therefore knew of Turing's capabilities). After Womersley saw the machine developments in America, he got much interested and started the project at NPL and persuaded Turing to join him. Hence Turing's work on computers post-dates that of von Neumann.

Inquiries of those of Turing's colleagues who are still at NPL proved fruitless, but through the kind offices of Mr D. W. Davies, who is now Superintendent of the Division of Computing Science at NPL, it was arranged for me to visit Mrs Sara Turing. Mrs Turing was very helpful and furnished me with several further leads, but was not really able to add much to the very brief, and unspecific, comments in her book that her son had, before the war, as part of a calculating machine project, got interested in the problem of cutting gears, and had begun to build a computer. The gear-cutting episode has been described by Professor D. G. Champernowne (1971):

> In about 1938 Alan was interested in looking for 'roots of the zeta function' or some much mathematical abstractions: I think he expected them to lie on 'the imaginary axis' or on 'the line $R(z)=1$' or some such place; and to find them he had devised a machine for summing a series involving terms such as $a_r \cos 2\pi t/p_r$ where p_r is the rth prime. All I remember is that the machine included a set of gear wheels the numbers of whose teeth were prime numbers, and I liked to fancy that as soon as the machine had found a root of the zeta function its centre of gravity would pass over the edge of the table and it would fall off uttering a swansong. I went once or twice with Alan to the engineering laboratory to assist in cutting the gear wheels, but the second world war interrupted this ambitious project and it was never completed. During and immediately after the war I often went for long walks with Alan and he tried to teach me a good deal about computers based on 'mercury delay-lines'.

Various other leads proved fruitless, and my enthusiasm for the search was

beginning to wane. I eventually had the opportunity to inspect a copy of Turing's report giving detailed plans for the ACE (Turing 1945). This proved to postdate, and even contain a reference to, von Neumann's Report on the EDVAC. However, Turing's report did allude to the fact that he had obtained much experience of electronic circuits.

But then my investigation took a dramatic turn.

A SECRET COMPUTER

One of my enquiries elicited the following response (Michie 1972a):

> I believe that Lord Halsbury is right about the von Neumann–Turing meeting.... The implication of Newman's obituary notice, as you quote it, is quite misleading; but it depends a bit on what one means by a 'computer'. If we restrict this to mean a stored-program digital machine, then Newman's implication is fair, because no-one conceived this device (apart from Babbage) until Eckert and Mauchly (sometimes attributed to von Neumann). But if one just means high-speed electronic digital computers, then Turing among others was thoroughly familiar during the war with such equipment, which predated ENIAC (itself not a stored-program machine) by a matter of years.
>
> However, it is at this stage that problems of security begin to close in. In the summer of 1970 I attempted to gain clearance, through the Cabinet Office, for release of some of this information. The matter was passed to the security people, who replied in the negative – incomprehensible after so many years.... It *can* at least be said that any Anglo-American leakage of ideas is likely to have been in the opposite direction to your suggestion [that Turing had known of, and been influenced by, the ENIAC and EDVAC projects].
>
> It may surprise you and interest you to know that during the war Turing was already not only thinking about digital computation, but was even speculating, in what we now see to have been a prophetic way, about 'thinking machines'. Many of the ideas in his 1947 essay... were vigorous discussion topics during the war. Some of his younger associates were fired by this, although most regarded it as cranky.

It turns out that at least three references have been made in the open literature to the work with which Turing was associated.

In a brief account of the origins of computers, McCarthy (1966) states:

> During World War II, J. Presper Eckert and John W. Mauchly of the University of Pennsylvania developed ENIAC, an electronic calculator. As early as 1943 a British group had an electronic computer working on a war-time assignment. Strictly speaking, however, the term 'computer' now designates a universal machine capable of carrying out any arbitrary calculation, as propounded by Turing in 1936. The possibility of such a machine was apparently guessed by Babbage; his collaborator Lady Lovelace, daughter of the poet Lord Byron, may have been the first to

propose a changeable program to be stored in the machine. Curiously it does not seem that the work of either Turing or Babbage played any direct role in the labours of the men who made the computer a reality. The first practical proposal for universal computers that stored their programs in their memory came from Eckert and Mauchly during the war.

A more direct comment has been published by Michie (1968):

During the war I was a member of Max Newman's group at Bletchley, working on prototypes of what we now call the electronic digital computer. One of the people I came most in contact with was Alan Turing, a founder of the mathematical theory of computation. . . .

(According to the book *The Code Breakers* (Kahn 1967), Bletchley Park was the wartime home of what the Foreign Office 'euphemistically called its Department of Communications'.)

Neither of these compare, however, to the amount of detail contained in a paper by Good (1970), which includes the wartime machine in a listing of successive generations of general purpose computers:

Cryptanalytic (British): classified, electronic, calculated complicated Boolean functions involving up to about 100 symbols, binary circuitry, electronic clock, plugged and switched programs, punched paper tape for data input, typewriter output, pulse repetition frequency 10^5, about 1000 gas-filled tubes; 1943 (M. H. A. Newman, D. Michie, I. J. Good and M. Flowers. Newman was inspired by his knowledge of Turing's 1936 paper).

. . . .

MADM (Manchester), Williams Tube and Magnetic drum: 1950 (F. C. Williams, T. Kilburn and others, with early influence from M. H. A. Newman, I. J. Good and David Rees).

. . . .

M. Flowers was a high rank telephone engineer and his experience with the cryptanalytic machine enabled him later to design an electronic telephone exchange. Another telephone engineer, Dr A. W. M. Coombs, who worked with the machine, later designed the time-shared trans-Atlantic multi-channel voice-communication cable system. . . . In Britain there was a causal chain leading from Turing's paper through [the wartime cryptanalytic machine, and MADM] to the giant Atlas machine (1964), although the main influence was from the IAS plans. [Flower's initials are in fact 'T. H.' rather than 'M.'.]

Confirmation of the fact that the machine was program-controlled has come from another of Turing's war-time colleagues (Anon 1972):

. . . Turing did visit America and may well have seen von Neumann there. Turing was an exceptionally creative man with a very far-ranging mind, and I would have expected a meeting between him and von Neumann to be very productive.

> I don't think that I can, or should, try to answer your questions on the reason for Turing's interest except as follows in the most general terms. The nature of our work was such that we needed to be able to carry out a series of logical steps at high speeds; this led in the first instance to a need for special purpose high-speed devices and then – because of the need to carry out a variety of different operations and to deal quickly with new problems – to the idea of the general purpose and thus the programmable equipment.
>
> In our field I would guess that in relation to USA, Turing gave a good deal more than he received – he was of a higher calibre than any of his opposite numbers.

However, Professor Good has indicated that this war-time computer was preceded by at least one other computer, although he too was unable to give an explicit confirmation of the reported Turing/von Neumann meeting (Good 1972):

> [Early in the Second World War Turing] made a visit to the States where I understand he met a number of top scientists including Claude Shannon. He was quite secretive about his visit but I had some reason to suspect that a small part of it was related to the atom bomb. It would not surprise me if he had contact with von Neumann during that visit. . . . Turing was very interested in the logic of machines even well before World War II and he was one of the main people involved in the design of a large-scale special purpose electromagnetic computer during the war. If he met von Neumann at that time I think it is certain that he would have discussed this machine with him.
>
> There were two very large-scale machines which we designed in Britain, built in 1941–43. One of them was the machine I just mentioned, which was mainly electromagnetic, and the second one was much more electronic and much less special purpose. . . . The second machine was closer to a modern digital binary general-purpose electronic computer than the first one, but the first also might very well have suggested both to Turing and von Neumann that the time had come to make a general-purpose electronic computer.
>
> . . . .
>
> Returning to Turing, in 1941 he designed at least one other small special-purpose calculator which was built and used, and later he worked on various speech devices. It is extremely difficult to estimate his total influence since, apart from all his classified work, he also had many conversations with many people about the automation of thought processes. In addition to this he anticipated, in classified work, a number of statistical techniques which are usually attributed to other people. I think that in order to obtain really hard facts, it would certainly be interesting to find out precisely whom he met in his war-time visit to the States. One problem is that the people with whom he spoke would have

been sworn to secrecy. Thus, for example, von Neumann might never have told H. H. Goldstine of the details of Turing's visit.

Professor Newman's role in post-war computer developments at Manchester, referred to in an earlier quotation, has been clarified by Professor Williams (1972):

> About the middle of the year [1946] the possibility of an appointment at Manchester University arose and I had a talk with Professor Newman who was already interested in the possibility of developing computers and had acquired a grant from the Royal Society of £30,000 for this purpose. Since he understood computers and I understood electronics the possibilities of fruitful collaboration were obvious. I remember Newman giving us a few lectures in which he outlined the organisation of a computer in terms of numbers being identified by the address of the house in which they were placed and in terms of numbers being transferred from this address, one at a time, to an accumulator where each entering number was added to what was already there. At any time the number in the accumulator could be transferred back to an assigned address in the store and the accumulator cleared for further use. The transfers were to be effected by a stored program in which a list of instructions was obeyed sequentially. Ordered progress through the list could be interrupted by a test instruction which examined the sign of the number in the accumulator. Thereafter operation started from a new point in the list of instructions.
>
> This was the first information I received about the organisation of computers. It may have derived from America through Newman but if it did it was pure Babbage anyway. Our first computer was the simplest embodiment of these principles, with the sole difference that it used a subtracting rather than an adding accumulator.
>
> ....
>
> Our first machine had no input mechanism except for a technique for inserting single digits into the store at chosen places. It had no output mechanism, the answer was read directly from the cathode ray tube monitoring the store. At this point Turing made his, from my point of view, major contribution. He specified simple minimum input facilities that we must provide so that he could organise input to the machine from five hole paper tape and output from the machine in similar form. Simultaneously with the assistance of J. C. West the synchronous magnetic backing up store was produced.
>
> These remarks must not be interpreted as suggesting that these were the only contributions to computers made by Newman and Turing, but only that they were the contributions of greatest importance to a particular group of engineers whose sole concern was to make a working machine.

(Turing had moved to Manchester from NPL, where he had been somewhat

discouraged by the slowness of the progress being made towards the building of the full ACE computer.)

Another person whom I had contacted in an effort to check the story of the Turing/von Neumann meeting was Dr S. Frankel, who had known von Neumann whilst working at Los Alamos. Although unable to help in this matter, he provided further evidence of the influence of Turing's pre-war work (Frankel 1972):

> I know that in or about 1943 or '44 von Neumann was well aware of the fundamental importance of Turing's paper of 1936 'On computable numbers . . .' which describes in principle the 'Universal Computer' of which every modern computer (perhaps not ENIAC as first completed but certainly all later ones) is a realization. Von Neumann introduced me to that paper and at his urging I studied it with care. Many people have acclaimed von Neumann as the 'father of the computer' (in a modern sense of the term) but I am sure that he would never have made that mistake himself. He might well be called the midwife, perhaps, but he firmly emphasized to me, and to others I am sure, that the fundamental conception is owing to Turing – insofar as not anticipated by Babbage, Lovelace, and others. In my view von Neumann's essential role was in making the world aware of these fundamental concepts introduced by Turing and of the development work carried out in the Moore school and elsewhere. Certainly I am indebted to him for my introduction to these ideas and actions. Both Turing and von Neumann, of course, also made substantial contributions to the 'reduction to practice' of these concepts but I would not regard these as comparable in importance with the introduction and explication of the concept of a computer able to store in its memory its program of activities and of modifying that program in the course of these activities.

A fourth of Turing's war-time colleagues tended to discount the story of a Turing/von Neumann meeting, but gave further details of Turing's role at Bletchley (Flowers 1972):

> In our war-time association, Turing and others provided the requirements for machines which were top secret and have never been declassified. What I can say about them is that they were electronic (which at that time was unique and anticipated the ENIAC), with electromechanical input and output. They were digital machines with wired programs. Wires on tags were used for semi-permanent memories, and thermionic valve bi-stable circuits for temporary memory. For one purpose we did in fact provide for variable programming by means of lever keys which controlled gates which could be connected in series and parallel as required, but of course the scope of the programming was very limited. The value of the work I am sure to engineers like myself and possibly to mathematicians like Alan Turing, was that we acquired a new understanding of and familiarity with logical switching and processing

because of the enhanced possibilities brought about by electronic technologies which we ourselves developed. Thus when stored program computers became known to us we were able to go right ahead with their development. It was lack of funds which finally stopped us, not lack of know-how.

THE SECOND PHASE OF THE INVESTIGATION

At about this stage I prepared a first draft account of my investigation, for use in the furtherance of my enquiries. One of the first reactions I obtained came from Professor Knuth (1972), who has spent much time investigating wartime computer developments in the United States:

> Some years ago when I had the opportunity to study the classified literature I looked at the early history of computers in cryptanalysis, and everything I found derived from ENIAC–EDVAC. An influential memorandum written by one of the attendees at the Moore summer school session in 1946 was republished (still classified) three years ago, and it seems that memo was (in America) what led to government support for computer development.

Then Professor Michie, first quoted earlier, amplified his comments considerably (Michie 1972b):

> 1. Turing was not directly involved in the design of the Bletchley electronic machines, although he was in touch with what was going on. He was, however, concerned in the design of electromagnetic devices used for another cryptanalytic purpose; the Post Office engineer responsible for the hardware side of this work was Bill Chandler.... Chandler also worked subsequently on the electronic machines (see below).
>
> 2. Good's statement quoting 10^5 as the pulse repetition frequency is wrong. The fastest machines (the 'Colossi') had 5000 clock pulses per second, and these were driven photo-electrically from the sprocket holes of the paper tape being scanned.
>
> 3. *First machines:* The 'Heath Robinson' was designed by Wynn Williams [who had published an important paper (Wynn Williams 1931) on the design of electronic counting devices well before the war] at the Telecommunications Research Establishment at Malvern, and installed in 1942/1943. All machines, whether 'Robinsons' or 'Colossi', were entirely automatic in operation, once started. They could only be stopped manually! Two five-channel paper tape loops, typically of more than 1000 characters length, were driven by pulley-drive (aluminium pulleys) at 2000 characters/sec. A rigid shaft, with two sprocket wheels, engaged the sprocket-holes of the two tapes, keeping the two in alignment. The five main channels of each tape were scanned by separate photo-cells. One of the tapes was 'data', in current terminology, and can be compared with a modern rotating backing store, such as a disk drive. The other

carried some fixed pattern, which was stepped systematically relative to the data tape, by differing in length by e.g. one unit. Counts were made of any desired Boolean functions of the two inputs. Fast counting was done by valves, and slow operations (e.g. control of on-line teleprinter) by relays.

Various improved Robinsons were installed – the 'Peter Robinson', the 'Robinson and Cleaver' – about four or five in number.

4. *Second crop:* The 'Colossi' were commissioned from the Post Office, and the first installation was made in December 1943 (the Mark 1). This was so successful that by great exertions the first of three more orders (for a Mark 2 version) was installed before D-day (June 6th 1944). The project was under the direction of T. H. Flowers, and on Flowers' promotion, A. W. M. Coombs took over the responsibility of coordinating the work. The design was jointly done by Flowers, Coombs, S. W. Broadbent and Chandler.

The main new features incorporated in the Colossus series, as compared with the Robinsons, were:

(1) Almost all switching functions were performed by hard valves, which totalled about 2000.

(2) There was only one pulley-driven tape, the data tape. Any pre-set patterns which were to be stepped through these data were generated internally from stored component-patterns. These components were stored as ring registers made of thyrotrons and could be set manually by plug-in pins. The data tape was driven at 5000 characters/sec, but (for the Mark 2) by a combination of parallel operations with short-term memory an effective speed of 25,000/sec was obtained.

(3) Boolean functions of all five channels of pairs of successive characters could be set up by plug-board, and counts accumulated in five bi-quinary counters.

(4) On-line typewriter in place of teleprinter.

The total number of Colossi installed and on order was about a dozen by the end of the war, of which about 10 had actually been installed.

5. A 'Super-Robinson' was designed by Flowers in 1944. Four tapes were driven in parallel. Photo-signals from the four sprocket holes were combined to construct a master sprocket pulse which drove the counting. . . .

6. Two or three members of the US armed services were seconded at various times to work with the project for periods of a year or more. The first of these arrivals was well after the first machines were operational.

From this description it is possible to attempt an assessment of the Bletchley machines, and particularly the Colossi, with respect to the modern digital computer. Clearly the arithmetical, as opposed to logical, capabilities were minimal, involving only counting, rather than general addition or other operations. They did, however, have a certain amount of electronic storage, as well as the paper-tape 'backing storage'. Although fully automatic, even

to the extent of providing printed output, they were very much special purpose machines, but within their field of specialization the facilities provided by plug-boards and banks of switches afforded a considerable degree of flexibility, by at least a rudimentary form of programming. Their importance as cryptanalytic machines, which must have been immense, can only be inferred from the number of machines that were made and the honours bestowed on various members of the team after the end of the war; however, their importance with respect to the development of computers was two-fold. They demonstrated the practicality of large-scale electronic digital equipment, just as ENIAC did, on an even grander scale, approximately two years later. Furthermore, they were also a major source of the designers of the first post-war British computers (the links to the ACE and Manchester University computers have been described already; in addition there is a direct link to the MOSAIC computer (Coombs 1954, Chandler 1954), a three-address serial machine whose design was started at NPL after the war, and completed at the Post Office Research Station).

There is, however, no question of the Colossi being stored program computers, and the exact sequence of developments, and patterns of influence, that led to the first post-war British stored program computer projects remains very unclear. The years 1943 to 1945 would appear to be particularly important, and for this reason I returned briefly to the investigation of E.W.Phillips.

The claim, made in the obituary notice (Menzler 1968) for Phillips, that he started work on ACE in 1943 with John Womersley turned out to be a repetition of claims that Phillips had made himself:

> Team work on the Pilot ACE commenced to be pedantically precise, at 10.30 am on Friday 15th January 1943. Turing did not join the team until the autumn of 1945. (Phillips 1965a)
>
> On 12th March 1943 John Womersley came down to Staple Inn to speak to the Students' Society, and one of the members of the Committee gave him a copy of the paper on binary notation. Thus after seven years I acquired at last – and at once – a very sterling collaborator. (Phillips 1965b)

This second remark was accompanied by again typically cryptic comments by Phillips, indicating that his 1936 paper had been based on a memorandum which he had prepared for the Department of Scientific Research, and that due to the insistence by the government that certain patents be filed, all references to thermionic valves had to be deleted. He did indeed file two patent applications relating to a calculating apparatus on 24 December 1935. However, no patents were ever granted, and there is no record of the contents of the original applications.

In a letter to the Editor of the Sunday Times (Phillips 1965c), Phillips revealed the debt he owed to Babbage, but made no further mention of events following his 1936 paper:

> After completing actuarial examinations in 1913, I turned back to a boyhood interest in Babbage's 1834 dream of an Analytical Engine, a self-operating, self-recording calculating machine – and during the 1914–18 war I was still thinking in terms of gear wheels.
>
> ....
>
> Still thinking in terms of mechanism until in 1928 or 1929 learned that in 1919 Eccles and Jordan had demonstrated that a thermionic valve could be in either of two stable states, and made to change with the speed of light.
>
> 1931: Wynn-Williams produced designs for electronic valves as counting units – all the binary electronic computer needed was a battery of such 'counters'.
>
> 1934: plan of electronic computer working in binary, but with octonary (digits 0 to 7) input and output completed to make the human operator's task easier. Babbage's 1834 sleeping beauty had awakened – after the proverbial hundred years.

Doubt has been cast on Phillips' claim to have collaborated with Womersley by people who knew both persons concerned. For example, Dr Goodwin has stated (1972):

> John Womersley was appointed Superintendent of the newly formed Mathematics Division at the NPL in early 1945 but the division did not really get under way until the autumn and it was then that Turing, in common with a number of us, joined the Division. I understand that it was Max Newman who suggested to Womersley that Turing should join us. He of course knew of Turing's development of the logical theory of a universal machine... and also of his war work. I imagine that Newman would have known of the aims of the new division from Hartree, who had played a big part in its setting up and who also knew of the work then going on in the USA on computing machines such as the ENIAC.
>
> I understand that Womersley did have a meeting with Phillips during the latter part of the war but certainly they did not do any work together. So far as I know, Phillips merely talked to Womersley about the importance of the binary scale in the context of automatic computing. While Phillips was a man of great talent, with an innovative turn of mind, I do not believe he was directly involved in the development of modern computers.

Mr J.G.L.Michel, who had attended the 1936 meeting at which Phillips had extolled the virtues of the binary system (Phillips 1936), and who had known Womersley from early in 1943, confirmed Dr Goodwin's comments, adding (Michel 1972):

> My impression was that it was I who acquainted both Womersley and Hartree with Phillips' paper... however since Comrie was present at the original paper in 1936 and was known by both Hartree and Womers-

> ley, I may be wrong... Despite these remarks (Phillips 1965b) by Phillips, my impression from weekly lunches with Womersley was that he himself took no direct interest in digital computers until his appointment as Superintendent of the new Mathematics Division – about January 1945. Electronic computers were in the air – ENIAC had been built, a decimal machine – Eckert, Mauchly and von Neumann were discussing EDVAC, probably the first binary machine. Nevertheless in general the idea of universality of a general purpose digital computer took some grasping, and until the idea of the 'boot strapping' operation of a machine doing its own binary to decimal conversion, the furthest most people went beyond a purely decimal machine was a biquinary representation; hence one of the reasons for the delay in recognising Phillips' contribution.

Clearly, my hope that a brief investigation of Phillips' career would explain how the early British computer projects came into being has proved over-optimistic. It is, however, to be hoped that, perhaps stimulated by the present paper, a more complete investigation of the work of the British pioneers during and immediately after the war will be undertaken.

THE STORED PROGRAM CONCEPT

The suggestion that Babbage had conceived the idea of a stored-program computer apparently rests on one brief passage in Lady Lovelace's notes on the analytical engine (1843):

> Figures, the symbols of numerical magnitude, are frequently *also* the symbols of *operations*, as when they are the indices of powers... whenever numbers meaning *operations* and not *quantities* (such as indices of powers) are inscribed on any column or set of columns, those columns immediately act in a wholly separate and independent manner, becoming connected with the *operating mechanism* exclusively, and reacting upon this. They never come into combination with numbers upon any other columns meaning quantities; though, of course, if there are numbers meaning *operations* upon *n* columns, these may combine amongst each other, and will often be required to do so, just as numbers meaning *quantities* combine with each other in any variety. It might have been arranged that all numbers meaning *operations* should have appeared on some separate portion of the engine from that which presents numerical *quantities*; but the present mode is in some cases more simple, and offers in reality quite as much distinctness when understood.

This rather obscure passage is somewhat contradicted by other statements in Lady Lovelace's notes. I do not know of any of Babbage's own writings which would throw any further light on this question, other than perhaps a brief passage in one of his sketchbooks dated 9 July 1836 which has been drawn to my attention by Mr M. Trask. In this passage Babbage's comments on the possibility of using the analytical engine to compute and punch out

modified program cards, which would later be used to control a further computation. However, Professor Wilkes' investigations of Babbage's notebooks (1971) have led him to the view that Babbage did not have a really clear idea of the notion of a program, rather than to any confirmation that Babbage had thought of the idea of a stored program. My own opinion is that, considering its date, Lady Lovelace's account of programming (which even includes an allusion to what we would now call indexed addressing) shows a remarkable understanding of the concept of a program – the question of the extent to which she, rather than Babbage, was responsible for the contents of her notes is not at all clear.

Other than perhaps this vague paragraph by Lady Lovelace and of course the implications of Turing's 1936 paper the earliest suggestion that instructions be stored in the main computer memory, that I know of, is contained in the 1945 report by von Neumann. This describes the various purposes for which memory capacity was needed – intermediate results, instructions, tables of numerical constants – ending:

> The device requires a considerable memory. While it appeared that various parts of this memory have to perform functions which differ somewhat in their nature and considerably in their purpose, it is nevertheless tempting to treat the entire memory as one organ, and to have its parts even as interchangeable as possible for the various functions enumerated above.

On the other hand, a later report by Eckert and Mauchly (1945) claims that in early 1944, prior to von Neumann's association with the project, they had designed a 'magnetic calculating machine' in which the program would 'be stored in exactly the same sort of memory device as that used for numbers'.

The earliest accounts imply that the idea of storing the program in the same memory as that used for numerical values arose from considerations of efficient resource utilization. The question of who first had the idea of, and an understanding of the fundamental importance of, the full stored program concept, that is of an extensive internal memory, used for both instructions and numerical qualities, together with the ability to program the modification of stored instructions, has been for years a very vexed one.

The earliest published discussions of the stored program concept and its impact on computer design and programming that I have seen include those by Goldstine and von Neumann (1963), Eckert (1947), Mauchly (1948), and, interestingly enough, Newman (1948). The IBM Selective Sequence Electronic Calculator (Eckert, W.J. 1948), which was started in 1945, first worked in 1947, and was publicly demonstrated in January 1948, was almost certainly the first machine which could modify and execute stored instructions (Phelps 1971). However, this machine was basically a tape-controlled machine, which had much more in common with the Harvard Mark 1 than with modern electronic computers. The earliest fully electronic stored pro-

gram computer to operate was probably a very small experimental machine, referred to earlier, that was built at Manchester in 1948 primarily to test the Williams tube type of storage (1948). It was apparently not until 1949 that any practical electronic stored program computers, incorporating input/output devices, became operational. The first of these was, I believe, the Cambridge EDSAC (Wilkes and Renwick 1950), whose design had been strongly influenced by the plans, described at the 1946 Moore School lectures, for the EDVAC.

I do not wish to enter this controversy, but am instead content to attribute the idea to Eckert, Mauchly, von Neumann and their colleagues collectively. Certainly the various papers and reports emanating from this group, from 1945 onwards, were the source of inspiration of computer designers in many different countries, and played a vital part in the rapid development of the modern computer.

CONCLUDING REMARKS

The initial major goal of this little investigation, which was to check out the story of a decisive war-time meeting of von Neumann and Turing, has not been achieved. Instead, and perhaps more importantly, I have to my own surprise accumulated evidence that in 1943, that is, in the year that work started on ENIAC, and 2–3 years before it was operational, a group of people directed by M.H.A.Newman, and with which Alan Turing was associated, had a working special purpose electronic digital computer. This machine, and its successors, were in at least a limited sense 'program-controlled', and it has been claimed that Turing's classic pre-war paper on computability, a paper which is usually regarded as being of 'merely' theoretical importance, was a direct influence on the British machine's designers, and also on von Neumann, at a time when he was becoming involved in American computer developments. Furthermore, at least three post-war British computer projects, namely those at Manchester University, at NPL and at the Post Office Research Station, can be seen to have direct links to the wartime project. Unfortunately, from the computer historian's point of view, the technical documentation of these war-time British computers is, nearly thirty years later, still classified. One wonders what other war-time machines, either British or American, have yet to be revealed!

Acknowledgements

It is a pleasure to thank the many people who have helped me in this investigation, and especially those whose letters I wished to quote in this paper and who, without exception, readily agreed to my request. I would, however, like to make explicit mention of the help I have received from Professor I.J.Good, Lord Halsbury, and Professor D.Michie.

REFERENCES

Champernowne, D.G. (1971) Private communication

Chandler, W.W. (1954) Gates and trigger circuits. *Automatic Digital Computation*, pp. 181–6. London: HMSO.

Coombs, A.W.M. (1954) MOSAIC – the 'Ministry of Supply Automatic Computer'. *Automatic Digital Computation*, pp. 38–42. London: HMSO.

Eckert, J.P. (1947) A preview of a digital computing machine. *Theory and Techniques for the Design of Electronic Digital Computers. Lectures delivered 8th July–31st August* 1946, pp. 10.1–10.26 (ed. Chambers, C.C.). Philadelphia: Moore School of Electrical Engineering, University of Pennsylvania.

Eckert, J.P. and Mauchly, J.W. (1945) *Automatic High Speed Computing: a progress report on the EDVAC*. Philadelphia: Moore School of Electrical Engineering, University of Pennsylvania.

Eckert, W.J. (1948) Electrons and computation. *The Scientific Monthly* **67, 5,** 315–23.

Flowers, T.H. (1972) Private communication.

Frankel, S. (1972) Private communication.

Goldstine, H.H. (1971) Private communication.

Goldstine, H.H. and von Neumann, J. (1963) On the principles of large scale computing machines. *John von Neumann: Collected Works*, vol. 5, pp. 1–32 (ed. Taub, A.H.). Oxford: Pergamon Press (previously unpublished, based on various lectures given by von Neumann, in particular one on 15 May 1946).

Good, I.J. (1970) Some future social repercussions of computers. *Intern. J Environmental Studies*, **1,** 67–69.

Good, I.J. (1972) Private communication.

Goodwin, E.T. (1972) Private communication.

Earl of Halsbury (1959) Ten years of computer development. *Comp. J*, **1,** 153–9.

Earl of Halsbury (1971) Private communication.

Kahn, D. (1967) *The Code-Breakers*. New York: MacMillan.

Knuth, D.E. (1972) Private communication.

[Ada Augusta, Countess of Lovelace] (1843) Sketch of the Analytical Engine invented by Charles Babbage, by L.F. Menabrea of Turin, Officer of the Military Engineers, with notes upon the Memoir by the Translator. *Taylor's Scientific Memoirs*, **3,** 666–731.

Mauchly, J.W. (1948) Preparation of problems for EDVAC-type machines. *Proc. of a Symp. on Large Scale Digital Calculating Machinery, 7th–10th January 1947. Annals of the Computation Laboratory of Harvard University*, **16,** pp. 203–7. Cambridge: Harvard University Press.

McCarthy, J. (1966) Information. *Scientific American*, **215, 3,** 65–72.

Menzler, F.A.A. (1968) William Phillips. *J. Inst. of Actuaries* **94,** 269–71.

Michel, J.G.L. (1972) Private communication.

Michie, D. (1968) Machines that play and plan. *Science Journal*, 83–8.

Michie, D. (1972a) Private communication.

Michie, D. (1972b) Private communication.

Newman, M.H.A. (1948) General principles of the design of all-purpose computing machines. *Proc. Roy. Soc. London A*, **195,** 271–4.

Newman, M.H.A. (1955) Alan Mathison Turing 1912–1954. *Biographical Memoirs of Fellows of the Royal Society*, **1,** 253–63.

Phelps, B.E. (1971) *The Beginnings of Electronic Computation*. Report TR 00.2259. Poughkeepsie, New York: Systems Development Division, IBM Corporation.

Phillips, E.W. (1936) Binary Calculation. *J. Inst. of Actuaries*, **67,** 187–221.

Phillips, E.W. (1965a) Irascible genius (Letter to the Editor). *Comp. J*, **8,** 56.

Phillips, E.W. (1965b) Presentation of Institute Gold Medals to Mr Wilfred Perks and Mr William Phillips, 23rd November 1964. *J. Inst. of Actuaries*, **91**, 19–21.
Phillips, E.W. (1965c) Birth of the computer (Letter to the Editor). *Sunday Times*, 5 September 1965.
Turing, A.M. (1936) On computable numbers, with an application to the entscheidungs-problem. *Proc. London Math. Soc.* (2), **42**, 230–67.
Turing, A.M. [1945] *Proposals for Development in the Mathematics Division of an Automatic Computing Engine (ACE)*. Teddington: Report E.882, Executive Committee, National Physical Laboratory.
Turing, S.(1959) *Alan M.Turing*. Cambridge: W.Heffer and Sons.
von Neumann, J. (1945) *First Draft of a Report on the EDVAC*. Philadelphia: Contract No. W–670–ORD–492, Moore School of Electrical Engineering, University of Pennsylvania.
Wilkes, M.V. (1971) Babbage as a computer pioneer. *Report of Proceedings, Babbage Memorial Meeting, London, 18th October 1971*, pp. 1–18. London: British Computer Society.
Wilkes, M.V. and Renwick, W. (1950) The EDSAC (Electronic Delay Storage Automatic Calculator). MTAC, **4**, 62–5.
Williams, F.C. (1972) Private communication.
Williams, F.C. and Kilburn, T. (1948) Experimental machine using electrostatic storage. *Nature*, **162**, 487.
Wynn-Williams, C.E. (1931) The use of thyratrons for high speed automatic counting of physical phenomena. *Proc. Roy. Soc. London A*, **132**, 295–310.
Anon (1946) Calculations and electronics: automatic computator [sic] designed by the National Physical Laboratory. *Electrician*, **137**, 1279–80.
Anon (1972) Private communication.

Note added in proof

Since this paper was written I have obtained from Professor M.Lehman a two page account that Phillips prepared, probably in 1963, with the view to using it as an introduction to a reprinted edition of his 1936 paper. The account describes his early work, and gives further details of his claims to be 'the earliest of the pioneers'. The statements it makes include:

(i) Womersley, after seeing a copy of the Binary Calculation paper in 1943, wrote to Phillips saying that, stimulated by Turing's 1936 paper, he had started to build a decimal telephone-relay computer in 1937, assisted by G.L.Norfolk from 1938.

(ii) Phillips and Womersley had war-time offices in Baker Street, London, which were opposite one another, and they began to collaborate. Womersley already knew that after the war he would be joining NPL, and he resolved that he would try to convince NPL to construct a binary electronic computer.

(iii) In November 1944 Womersley read a paper embodying the ideas so far formulated to the Executive Committee of the NPL.

(iv) Before Womersley joined NPL in April 1945 he spent three months in the States, where he learnt about the still secret Harvard Mk. 1 and the ENIAC, and 'reminded his hosts of Phillips' 1936 advocacy' of binary notation.

(v) After his return to England he resumed his joint efforts with Phillips,

and they were joined in August 1945 by Turing, at Womersley's invitation.

Prompted by these more detailed claims Mr J. G. L. Michel kindly investigated the files of the NPL Executive Committee. These showed that the proposals for a Mathematics Division were not agreed until May 1944 and that Womersley was not selected as Superintendent until September of that year. However, it is recorded that Womersley in his report to the Executive Committee (in December rather than November 1944) suggested that one of the sections of the Mathematics Division should be concerned with 'analytical engines and computing development'. He recommended the building of a differential analyzer and a machine using megacycle electronic counters, stating that 'all the processes of arithmetic can be performed and by suitable inter-connections operated by uniselectors a machine can be made to perform certain cycles of operations mechanically'. (Intriguingly, Dr Southwell, a member of the Committee, is recorded as having mentioned that 'some three years ago Professor Wiener in the U.S.A. was considering the development of analytical contrivances on the lines now advocated by Mr Womersley'.) Thus although we now have no reason to doubt the existence of a factual basis to Phillips' claims it would appear that the early plans for an NPL 'analytical engine' were very rudimentary indeed compared to Turing's 1945 proposal for the ACE.

PROGRAM PROOF AND MANIPULATION

Some Techniques for Proving Correctness of Programs which Alter Data Structures

R. M. Burstall

Department of Machine Intelligence
University of Edinburgh

1. INTRODUCTION

Consider the following sequence of instructions in a list-processing language with roughly ALGOL 60 syntax and LISP semantics (*hd* (*head*) is LISP CAR and *tl* (*tail*) is LISP CDR).

$x := cons(1, nil)$;
$y := cons(2, x)$;
$hd(x) := 3$; (compare LISP RPLACA)
$print(x)$; $print(y)$;

The intention is as follows.

x becomes the list (1)
y becomes the list (2, 1)
The head (first element) of the list x becomes 3.

Since y was manufactured from x it 'shares' a list cell with x, and hence is side-effected by the assignment to $hd(x)$.

When x is printed it is (3) and y when printed is (2, 3) rather than (2, 1) as it would have been had the last assignment left it undisturbed.

How are we to prove assertions about such programs? Figure 1 traces the course of events in the traditional picture language of boxes and arrows. Our task will be to obtain a more formal means of making inferences, which, unlike the picture language, will deal with general propositions about lists. We will extend Floyd's proof system for flow diagrams to handle commands which process lists. The principles which apply to lists would generalise in a straightforward way to multi-component data structures with sharing and circularities.

Although this technique permits proofs, they are rather imperspicuous and fiddling for lack of appropriate higher level concepts. Investigating the special case of linear lists in more depth we define 'the list from x to y' and consider systems of such lists (or perhaps we should say list fragments) which do not

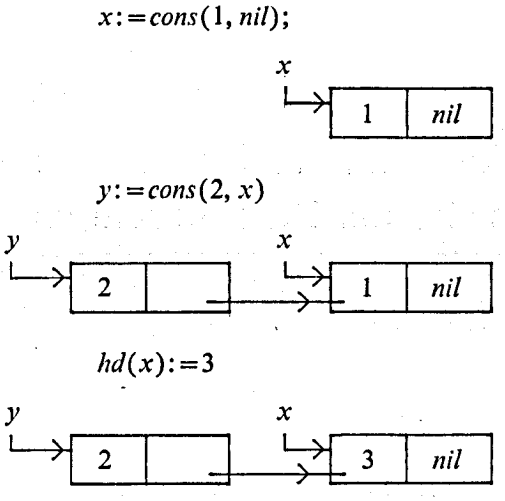


Figure 1.

share with each other or within themselves. (By a linear list we mean one which either terminates with *nil*, such as LISP $(A, (B, C), D)$, or is circular; by a tree we mean a list structure which terminates with atoms rather than with nil, such as LISP $((A . B) . (C . D))$). We thus get a rather natural way of describing the states of the machine and the transformations on them and hence obtain easy proofs for programs. Some ideas from the application of category theory to tree automata help us to extend this treatment from lists to trees: fragments of lists or trees turn out to be morphisms in an appropriate category. Acquaintance with category-theoretic notions is not however needed to follow the argument. Our aim has been to obtain proofs which correspond with the programmer's intuitive ideas about lists and trees. Extension to other kinds of data structures awaits further investigation.

2. PREVIOUS WORK

Since McCarthy (1963) raised the problem a number of techniques for proving properties of programs have been proposed. A convenient and natural method is due to Floyd (1967) and it has formed the basis of applications to non-trivial programs, for example by London (1970) and Hoare (1971). Floyd's technique as originally proposed dealt with assignments to numerical variables, for example, $x:=x+1$, but did not cater for assignments to arrays, for example, $a[i]:=a[j]+1$, or to lists, such as $tl(x):=cons(hd(x), tl(tl(x)))$. McCarthy and Painter (1967) deal with arrays by introducing 'change' and 'access' functions so as to write $a[i]:=a[j]+1$ as $a:=change(a, i, access$

$(a, j)+1$), treating arrays as objects rather than functions. King (1969) in mechanising Floyd's technique gives a method for such assignments which, however, introduces case analysis that sometimes becomes unwieldy. Good (1970) suggests another method which distinguishes by subscripts the various versions of the array. We will explain below how Good's method can be adapted to list processing. Although the proofs mentioned above by London (1970) and Hoare (1971) involve arrays they do not give rigorous justification of the inferences involving array assignments, which are rather straightforward.

List processing programs in the form of recursive functions have received attention from McCarthy (1963), Burstall (1969) and others, but quite different problems arise when assignments are made to components of lists. This was discussed in Burstall (1970) as an extension to the axiomatic semantics of ALGOL, but the emphasis there was on semantic definition rather than program proof.

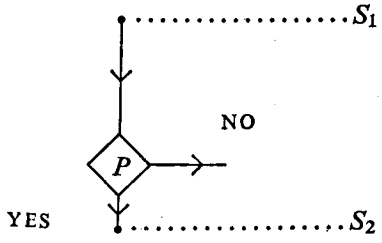
Hewitt (1970), Chapter 7, touches on proofs for list processing programs with assignments. J. Morris of Berkeley has also done some unpublished work, so have B.Wegbreit and J.Poupon of Harvard (Ph.D. thesis, forthcoming).

3. FLOYD'S TECHNIQUE

Let us recall briefly the technique of Floyd (1967) for proving correctness of programs in flow diagram form. We attach assertions to the points in the flow diagram and then verify that the assertion at each point follows from those at all the immediately preceding points in the light of the intervening commands. Floyd shows, by induction on the length of the execution path, that if this verification has been carried out whenever the program is entered with a state satisfying the assertion at the entry point it will exit, if at all, only with a state satisfying the assertion at the exit point.

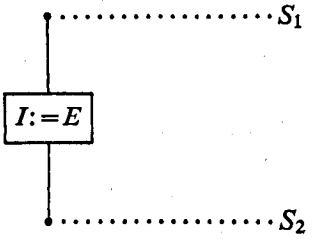
The rules for verification distinguish two cases: tests and assignments.

(1) A triple consisting of an assertion, a test and another assertion, thus



is said to be *verified* if S_1, $P \vdash_A S_2$, that is, assertion S_2 is deducible from S_1 and test P using some axioms A, say the axioms for integer arithmetic. If S_2 is attached to the NO branch the triple is verified if S_1, $\neg P \vdash_A S_2$.

(2) A triple consisting of an assertion, an assignment and another assertion, thus

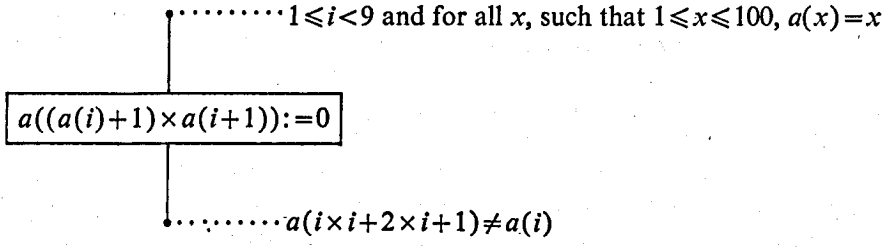


where I is some identifier and E some expression, is said to be *verified* if $S_1 \vdash_A [S_2]_I^E$ where $[S_2]_I^E$ means the statement S_2 with E substituted for I throughout. (This is called backward verification by King (1969); it is a variant of Floyd's original method, which introduces an existentially quantified variable.)

We will here retain the inductive method of Floyd for dealing with flow diagrams containing loops, but give methods for coping with more complex kinds of assignment command.

4. EXTENSION OF THE VERIFICATION PROCEDURE TO ARRAY ASSIGNMENTS

Consider the following command and assertions



The second assertion does hold if the first one does, but the verification rule given above for assignments to numeric variables, such as $j:=2\times j$, is inadequate for array assignments such as this. Thus attempts to substitute 0 for $a((a(i)+1)\times a(i+1))$ in $a(i\times i+2\times i+1)\neq a(i)$ merely leave it unchanged, but the unchanged assertion does not follow from the first assertion. (Floyd's version of the rule, using an existential quantifier is equally inapplicable.)

Following Good (1970), with a slightly different notation, we can overcome the difficulty by distinguishing the new version of the array from the old one by giving it a distinct symbol, say a'. We also make explicit the fact that other elements have not changed. We thus attempt to show that

$$1 \leqslant i \leqslant 9, (\forall x)(1 \leqslant x \leqslant 100 \Rightarrow a(x) = x), a'((a(i)+1)\times a(i+1)) = 0,$$
$$(\forall y)(y \neq (a(i)+1)\times a(i+1) \Rightarrow a'(y) = a(y)) \vdash_{\text{arith}} a'(i\times i+2\times i+1) \neq a'(i).$$

Once we note that $(a(i)+1)\times a(i+1)$ is $(i+1)^2$, as is $(i\times i+2\times i+1)$ and that, since $1 \leqslant i$, $(i+1)^2 \neq i$, we have $a'(i\times i+2\times i+1)=0$ and $a'(i)=i$, so

$a'(i)>0$. The distinction between a and a' and the condition that all elements which are not assigned to are unchanged reduce the problem to elementary algebra.

5. COMMANDS, CHANGE SETS AND TRANSFORMATION SENTENCES

The following technique is a variant of the one given by Good (1970). He uses it for assignments to arrays but not for list processing.

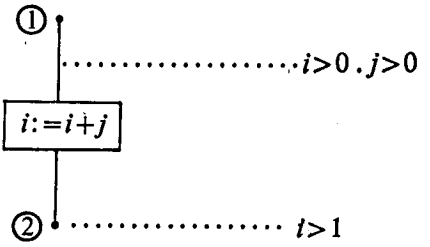
With each assignment command we associate a set of identifiers called the *change set* and a set of sentences called the *transformation sentences.* (Rules for specifying these are given below.) The basic rule of inference is:

If a set S of sentences are written before a command C, and C has a set T of transformation sentences, then we may write a sentence U after C iff S, $T \vdash U'$, where U' is U with each identifier i in the change set of C replaced by i'.

(1) *Simple assignment*

Command: $I:=E$	e.g. $i:=i+j$
Change set: $\{I\}$	$\{i\}$
Transformation sentences: $I'=E$	$i'=i+j$

Example:



$i>1$ is legitimate at ② because $i>0, j>0, i'=i+j \vdash i'>1$.

Note. By $A \vdash B$ we mean B is provable from A using axioms of arithmetic or other axioms about the operations of the language.

Note. Variables (identifiers) in the program become *constants* in the logic.

(2) *Array assignment*

Command: $A[E_1]:=E_2$	e.g. $a[a[i]]:=a[i]+a[j]$
Change set: $\{A\}$	$\{a\}$
Transformation sentences:	
$A'(E_1)=E_2$	$a'(a(i))=a(i)+a(j)$
$(\forall x)(x\neq E_1 \Rightarrow A'(x)=A(x))$	$(\forall x)(x\neq a(i)\Rightarrow a'(x)=a(x))$

(3) *List processing*

(a) Command: $hd(E_1):=E_2$	e.g. $hd(tl(hd(i))):=hd(i)$
Change set: $\{hd\}$	$\{hd\}$
Transformation sentences:	
$hd'(E_1)=E_2$	$hd'(tl(hd(i)))=hd(i)$
$(\forall x)(x\neq E_1\Rightarrow hd'(x)=hd(x))$	$(\forall x)(x\neq tl(hd(i))\Rightarrow hd'(x)=hd(x))$

(b) Command: $tl(E_1):=E_2$ . . . as for hd

(c) Command: $I:=cons(E_1, E_2)$	e.g. $i:=cons(2, j)$
Change set: $\{I, used, new\}$	$\{i, used, new\}$
Transformation sentences:	
$new' \notin used$	$new' \notin used$
$used' = used \cup \{new'\}$	$used' = used \cup \{new'\}$
$hd(new') = E_1$	$hd(new') = 2$
$tl(new') = E_2$	$tl(new') = j$
$I' = new'$	$i' = new'$

Note. We assume E_1 and E_2 do not contain *cons*. Complex *cons* expressions must be decomposed.

It should be clear that the choice of two-element list cells is quite arbitrary, and exactly analogous rules could be given for a language allowing 'records' or 'plexes', that is a variety of multi-component cells.

6. CONCEPTS FOR LIST PROCESSING

We could now proceed to give examples of proofs for simple list processing programs, but with our present limited concepts it would be difficult to express the theorems and assertions except in a very *ad hoc* way. We want to talk about the list pointed to by an identifier i and say that this is distinct from some other list. We want to be able to define conveniently such concepts as reversing or concatenating (appending) lists.

To do this we now specialise our discussion to the case where *cons*, *hd* and *tl* are used to represent linear lists. Such lists terminate in *nil* or else in a cycle, and we do not allow atoms other than *nil* in the tail position. Thus we exclude binary trees and more complicated multi-component record structures.

First we define the possible states of a list processing machine.

A *machine* is characterised by:

C, a denumerable set of cells

nil, a special element

A, a set of atoms (C, $\{nil\}$ and A are disjoint)

σ, a function from finite subsets of C to C, such that $\sigma(X) \notin X$, a function for producing a new cell.

It is convenient to write X^0 for $X \cup \{nil\}$ where $X \subseteq C$.

A *state* is characterised by:

$U \subseteq C$, the cells in use

hd: $U \to A \cup U^0$

tl: $U \to U^0$ (thus we are talking about lists rather than binary trees)

Let X^* be the set of all strings over a set X, including the empty string which we write as 1. Also let $T = \{true, false\}$.

It is convenient to define unit strings over $A \cup U^0$ thus

$Unit$: $(A \cup U^0)^* \to T$

$Unit(\sigma) \Leftrightarrow \sigma \in (A \cup U^0)$

We can now associate a set $\mathscr{L}$ of triples with a state. $(\alpha, u, v) \in \mathscr{L}$ means

that α is a list from u to v (or perhaps we should say a list fragment from u to v). Thus

$$\mathscr{L} \subseteq (A \cup U^0)^* \times U^0 \times U^0$$

We shall write $u \overset{\alpha}{\rightarrow} v$ as a mnemonic abbreviation for (α, u, v), $hd\,u$ for $hd(u)$ and $tl\,u$ for $tl(u)$. We define $\mathscr{L}$ inductively, thus:

(i) $u \overset{1}{\rightarrow} u \in \mathscr{L}$ for each $u \in U^0$

(ii) if $u \overset{\alpha}{\rightarrow} v \in \mathscr{L}$ and $v \overset{\beta}{\rightarrow} w \in \mathscr{L}$ then $u \overset{\alpha\beta}{\rightarrow} w \in \mathscr{L}$

(iii) $u \overset{hd\,u}{\rightarrow} tl\,u \in \mathscr{L}$ for each $u \in U$.

For example, in figure 2, $x \overset{a_1a_2a_3}{\longrightarrow} y$, $y \overset{a_4a_5a_3}{\longrightarrow} y$, $y \overset{1}{\rightarrow} y$ and $z \overset{a_6a_7}{\rightarrow} nil$ are all in $\mathscr{L}$.

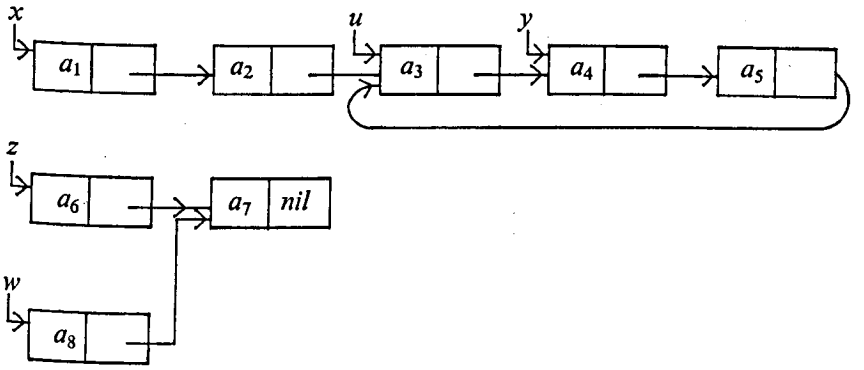


Figure 2.

Identity lists are defined thus

$$Identity: \mathscr{L} \rightarrow T$$

$$Identity(u \overset{\alpha}{\rightarrow} v) \Leftrightarrow \alpha = 1$$

A partial operation of composition $(\cdot)$ between lists is defined thus

$$\cdot : \mathscr{L} \times \mathscr{L} \rightarrow T$$

$$(u \overset{\alpha}{\rightarrow} v) \cdot (v \overset{\beta}{\rightarrow} w) = u \overset{\alpha\beta}{\rightarrow} w$$

Thus a list from u to v can be composed with one from v' to w if and only if $v = v'$.

The reader familiar with category theory will notice that $\mathscr{L}$ forms a category with U as objects and the lists (triples in $\mathscr{L}$) as morphisms. (A definition of 'category' is given in the Appendix.) Indeed it is the free category generated by the graph whose arrows are the lists $u \overset{hd\,u}{\longrightarrow} tl\,u$ for each u. There is a forgetful functor from $\mathscr{L}$ to $(A \cup U^0)^*$, where the latter is regarded as a category with one object. This functor gives the string represented by a list fragment (cf. Wegbreit and Poupon's notion of covering function in the unpublished work mentioned on p. 25).

We define the number of occurrences of a cell in a list by induction

$$\delta: U \times \mathscr{L} \to N$$

(i) $\delta_u(v \xrightarrow{1} v) = 0$

(ii) $\delta_u(x \xrightarrow{a} tl\, x \xrightarrow{\alpha} w) = \delta_u(tl\, x \xrightarrow{\alpha} w) + 1$ if $x = u$

$= \delta_u(tl\, x \xrightarrow{\alpha} w)$ if $x \neq u$

It follows by an obvious induction that $\delta_u(\lambda \cdot \mu) = \delta_u(\lambda) + \delta_u(\mu)$.

To keep track of the effects of assignments we now define a relation of distinctness between lists, that is, they have no cells in common.

$$Distinct: \mathscr{L} \times \mathscr{L} \to T$$

$Distinct(\lambda, \mu) \Leftrightarrow \delta_u(\lambda) = 0$ or $\delta_u(\mu) = 0$ for all $u \in U$.

We also define a property of non-repetition for lists

$$Nonrep: \mathscr{L} \to T$$

$Nonrep(\lambda) \Leftrightarrow \delta_u(\lambda) \leqslant 1$ for all $u \in U$

Lemma 1

(i) $Distinct(\lambda, u \xrightarrow{1} u)$ for all λ and u

(ii) $Nonrep(u \xrightarrow{1} u)$ for all u

(iii) $Distinct(\lambda, \mu)$ and $Nonrep(\lambda)$ and $Nonrep(\mu) \Leftrightarrow Nonrep(\lambda \cdot \mu)$, if $\lambda \cdot \mu$ is defined.

Proof. Immediate.

We are now equipped to state the correctness criterion for a simple list processing program and to supply and prove the assertions. Consider the problem of reversing a list by altering the pointers without using any new space (figure 3). We first need to define an auxiliary function to reverse a string

$$rev: X^* \to X^*$$

$$rev(1) = 1$$

$$rev(x\alpha) = rev(\alpha)x \qquad \text{for } x \in X, \alpha \in X^*$$

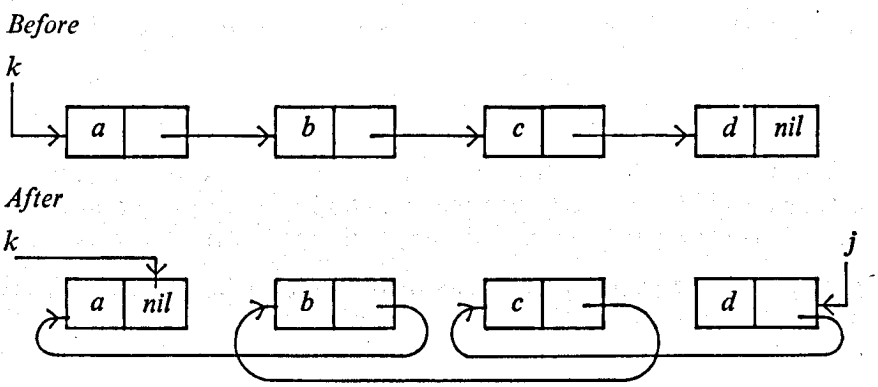


Figure 3.

$j = REVERSE(k)$

Assume $rev: U^* \to U^*$ reverses strings.

K is a constant string.

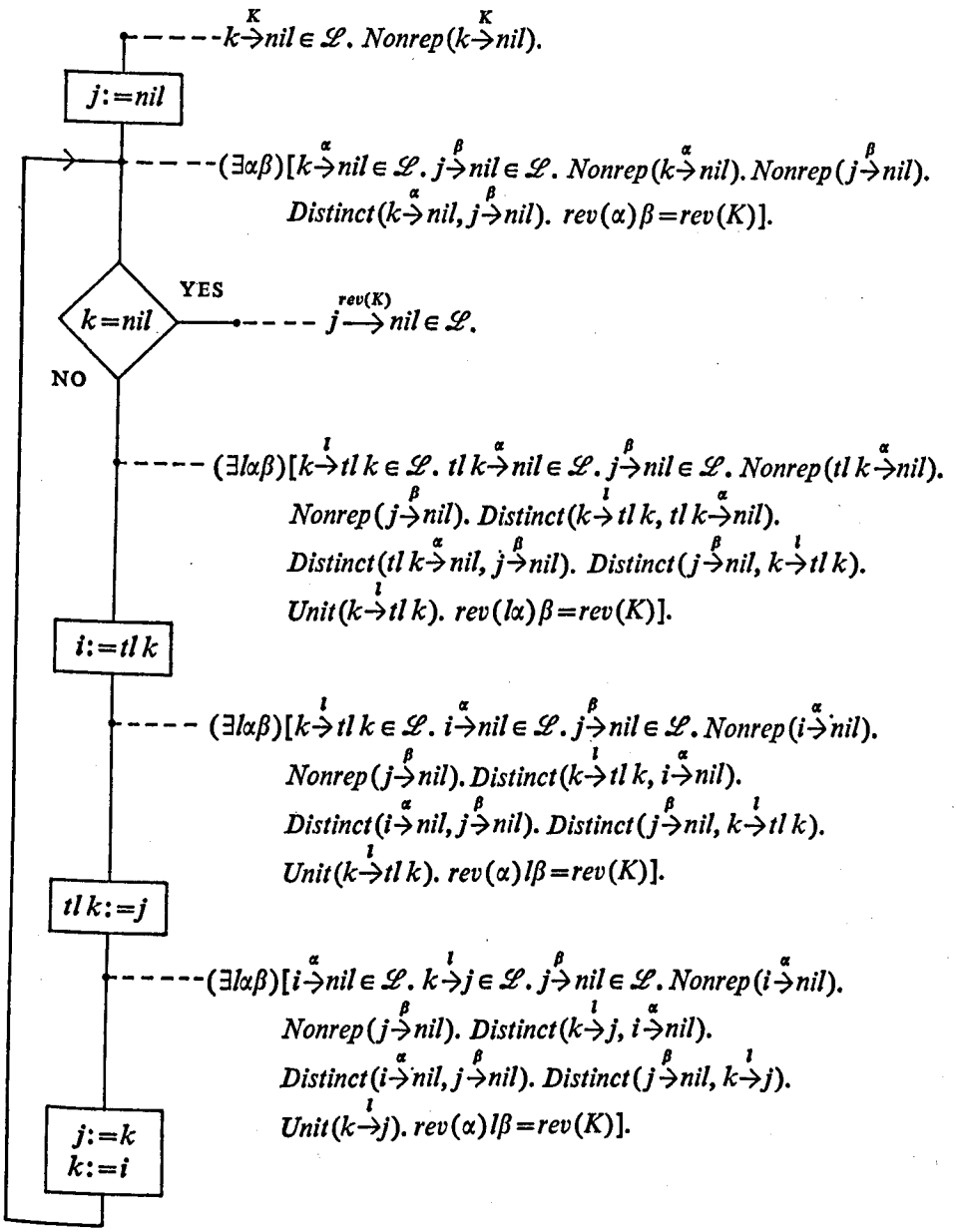


Figure 4. Reversing a list.

By well-known methods of induction on the length of the string (structural induction) we can prove simple lemmas such as

$$rev(\alpha\beta) = rev(\beta)\, rev(\alpha) \qquad \text{for } \alpha, \beta \in X^*$$

Notice the distinction between the function *rev* which works on strings and has easily proved properties and the function or procedure *REVERSE* to reverse lists using assignment, which we are about to define. The latter is essentially a function from machine states to machine states, where a machine state is characterised by the two functions *hd* and *tl*.

The flow diagram for *REVERSE* with assertions is given in figure 4. Notice that the assertions are long and tedious. We can verify the assertions by using the techniques given above, distinguishing between *tl* and *tl'* and consequently between *List*, *Distinct* and *List'*, *Distinct'*. The verification proofs are quite long for such a simple matter and very boring. We will not weary the reader with them; instead we will try to do better.

7. DISTINCT NON-REPEATING LIST SYSTEMS

We will now gather together the concepts introduced so far into a single notion, that of a Distinct Non-repeating List System (DNRL System). Using a suitable abbreviated notation we can render the assertions brief and perspicuous. To make the verification proofs equally attractive we show how the various kinds of commands, namely assignment to head or tail and *cons* commands, correspond to simple transformations of these systems. We can prove this once and for all using our previous technique and then use the results on a variety of programs.

We define a Distinct Non-repeating List System as an n-tuple of triples λ_i, $i = 1, \ldots, n$, such that

(i) $\lambda_i \in \mathscr{L}$ for each $i = 1, \ldots, n$

(ii) $Nonrep(\lambda_i)$ for each $i = 1, \ldots, n$

(iii) If $j \neq i$ then $Distinct(\lambda_i, \lambda_j)$ for each $i, j = 1, \ldots, n$

It is clear that if S is a DNRL System then so is any permutation of S. Thus the ordering of the sequence is immaterial. We should not think of S merely as a *set* of triples, however, since it is important whether S contains a triple $x \overset{\alpha}{\rightarrow} y$ once only or more than once (in the latter case it fails to be a DNRL System unless $\alpha = 1$).

Abbreviation. We will write $u_1 \overset{\alpha_1}{\rightarrow} u_2 \overset{\alpha_2}{\rightarrow} u_3 \ldots u_{k-1} \overset{\alpha_{k-1}}{\rightarrow} u_k$, for $u_1 \overset{\alpha_1}{\rightarrow} u_2, u_2 \overset{\alpha_2}{\rightarrow} u_3, \ldots, u_{k-1} \overset{\alpha_{k-1}}{\rightarrow} u_k$.

We also write $*S$ for 'S is a DNRL System'.

For example, an abbreviated state description for the state shown in figure 2 is

$$*(x \overset{a_1 a_2}{\rightarrow} u \overset{a_3}{\rightarrow} y \overset{a_4 a_5}{\rightarrow} u,\ z \overset{a_6}{\rightarrow} tl\, z \overset{a_7}{\rightarrow} nil,\ w \overset{a_8}{\rightarrow} tl\, z)$$

or a less explicit description

$$(\exists \alpha a \beta \gamma \delta)(*(x \overset{\alpha}{\rightarrow} u \overset{a}{\rightarrow} y \overset{\beta}{\rightarrow} u,\ z \overset{\gamma}{\rightarrow} nil) \text{ and } *(w \overset{\delta}{\rightarrow} nil) \text{ and } Atom(a))$$

$j=REVERSE(k)$

Assume *rev*: $(A \cup U)^* \rightarrow (A \cup U)^*$ reverses strings.

K is a constant string. Assume $a, b, c \ldots, \alpha, \beta, \gamma$ are existentially quantified before each assertion.

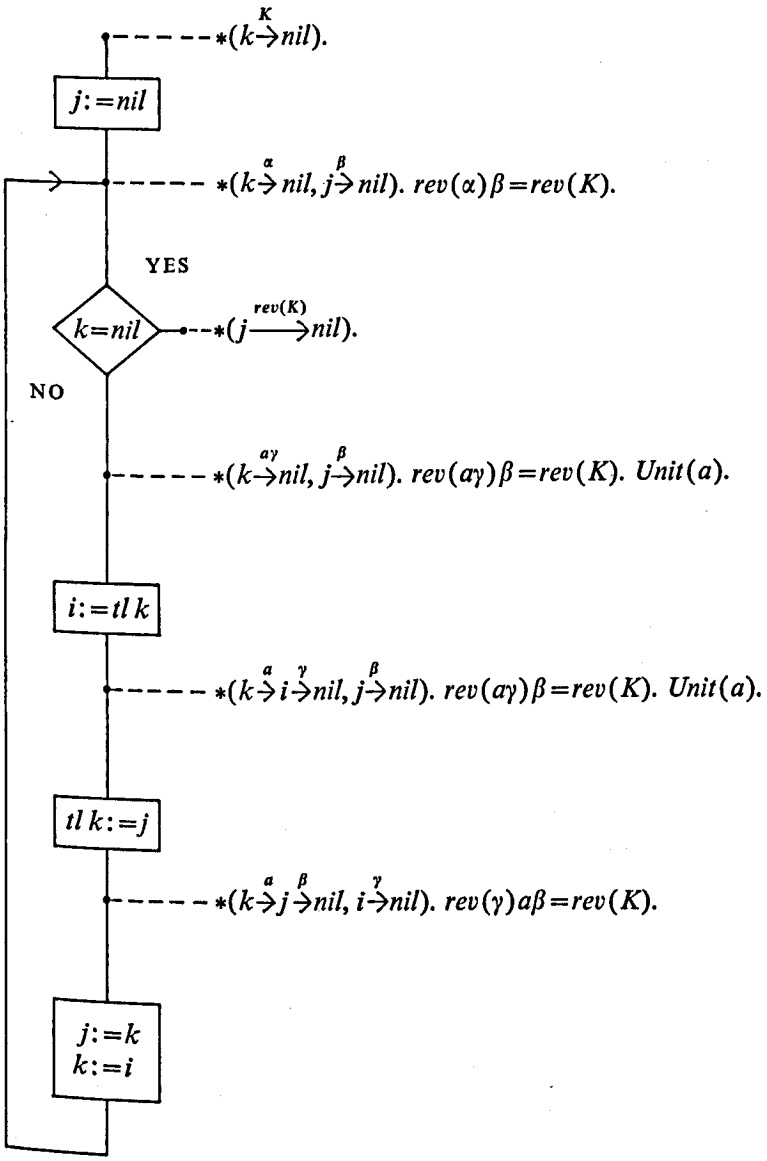


Figure 5. Reversing a list.

Figure 5 shows the *REVERSE* program again with much pithier assertions. We will now consider how to verify assertions such as these painlessly.

The following proposition which enables us to manipulate DNRL Systems follows easily from the definitions above.

Proposition 1

(i) *Permutation*
$*S \Rightarrow *S'$ if S' is a permutation of S.

(ii) *Deletion*
$*(\lambda_1, \ldots, \lambda_n) \Rightarrow *(\lambda_2, \ldots, \lambda_n)$

(iii) *Identity*
$*(\lambda_1, \ldots, \lambda_n) \Rightarrow *(\overset{1}{u \rightarrow} u, \lambda_1, \ldots, \lambda_n)$

(iv) *Composition*
$*(\overset{\alpha}{u \rightarrow} v \overset{\beta}{\rightarrow} w, \lambda_1, \ldots, \lambda_n) \Rightarrow *(\overset{\alpha\beta}{u \rightarrow} w, \lambda_1, \ldots, \lambda_n)$
and conversely
$*(\overset{\alpha\beta}{u \rightarrow} w, \lambda_1, \ldots, \lambda_n) \Rightarrow (\exists v)\, *(\overset{\alpha}{u \rightarrow} v \overset{\beta}{\rightarrow} w, \lambda_1, \ldots, \lambda_n)$

(v) *Distinctness*
$*(\overset{\alpha}{u \rightarrow} v, \overset{\beta}{u \rightarrow} w) \Rightarrow \alpha = 1$ or $\beta = 1$

(vi) *Inequality*
$*(\overset{\alpha}{u \rightarrow} v)$ and $u \neq v \Rightarrow (\exists b\beta)(\mathit{Unit}(b)$ and $\alpha = b\beta)$.

Proof. (i) and (ii) Immediate from the definition of *.

(iii) By Lemma 1 (i) and (ii).

(iv) By Lemma 1 (iii).

(v) If $\alpha \neq 1$ and $\beta \neq 1$ then $\delta_u(u \rightarrow v) = 1$ and $\delta_u(\overset{\beta}{u \rightarrow} w) = 1$ so they are not distinct.

(vi) By definition of $\mathscr{L}$.

We are now able to give the transformation sentences associated with the various commands, in terms of * rather than in terms of *hd* and *tl*. They enable us to verify the assertions in figure 5 very easily. The transformation sentences all involve replacing or inserting a component of a *-expression, leaving the other components unchanged. They are displayed in figure 6. We will make some comments on these transformation sentences and how to use them. Their correctness may be proved using the transformation sentences given earlier for *hd* and *tl* and the following lemma.

Lemma 2. Suppose for all $y \neq x$, $hd'y = hd\, y$ and $tl'y = tl\, y$. Then for all $\lambda \in \mathscr{L}$ such that $\delta_x(\lambda) = 0$ we have $\lambda \in \mathscr{L}'$ and $\delta'_y(\lambda) = \delta_y(\lambda)$ for all y.

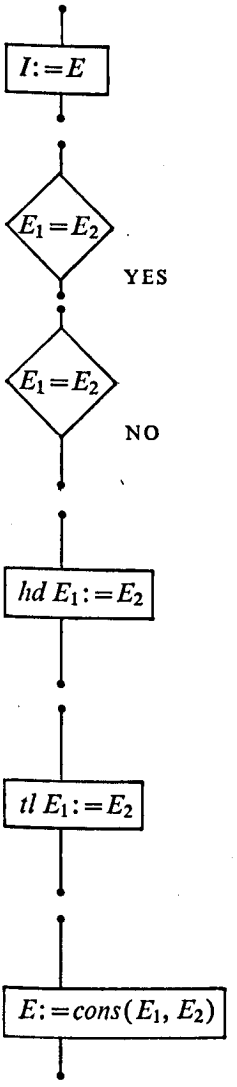
Corollary. Under the same suppostion if $\lambda, \mu \in \mathscr{L}$ and $\delta_x(\lambda) = 0$ and $\delta_x(\mu) = 0$ then $\mathit{Nonrep}(\lambda) \Rightarrow \mathit{Nonrep}'(\lambda)$ and $\mathit{Distinct}(\lambda, \mu) \Rightarrow \mathit{Distinct}'(\lambda, \mu)$.

Proof. By an obvious induction on $\mathscr{L}$.

The rule for assignment to a simple identifier is as before.

Transformation sentences are given for the YES and NO branches of a test, even though these do not alter the state (their change sets are empty).

Assume $n, \lambda_1, \ldots, \lambda_n, x, y, a$ are *universally* quantified. The transformation sentences are shown to the right of the command. We assume that E, E_1 and E_2 do not contain *cons*.



Change set $=\{I\}$
$I' = E$

Change set $=\{\ \}$
$E_1 = E_2$

Change set $=\{\ \}$
$E_1 \neq E_2$

Change set $=\{hd, *\}$
$*(E_1 \xrightarrow{a} y, \lambda_1, \ldots, \lambda_n)$ and $Unit(a) \Rightarrow$
$*'(E_1 \xrightarrow{E_2} y, \lambda_1, \ldots, \lambda_n)$

Change set $=\{tl, *\}$
$*(E_1 \xrightarrow{a} y, \lambda_1, \ldots, \lambda_n)$ and $Unit(a) \Rightarrow$
$*'(E_1 \xrightarrow{a} E_2, \lambda_1, \ldots, \lambda_n)$

Change set $=\{*, I\}$
$*(\lambda_1, \ldots, \lambda_n) \Rightarrow$
$*'(E' \xrightarrow{E_1} E_2, \lambda_1, \ldots, \lambda_n)$

Figure 6. Transformation sentences for state descriptions.

Consider for example the NO branch of the test '$k=nil$' in the *REVERSE* program of figure 5.

Before the command we have

$$(\exists\alpha\beta)[*(k\overset{\alpha}{\rightarrow}nil, j\overset{\beta}{\rightarrow}nil).\ rev(\alpha)\beta=rev(K)] \tag{1}$$

The transformation sentence, putting $n=1$, is

$$k\neq nil. \tag{2}$$

After the command, we have

$$(\exists a\gamma\beta)[*(k\overset{a\gamma}{\rightarrow}nil, j\overset{\beta}{\rightarrow}nil).\ rev(a\gamma)\beta=rev(K).\ Unit(a)] \tag{3}$$

But (3) follows immediately from (1) and (2) using Proposition 1 (vi).

In general, Proposition 1 will be used to reorder or otherwise manipulate the *-expressions and if E_1 or E_2 contain references to *hd* or *tl* these will need to be removed by replacing them by $\hat{E}_1$ and $\hat{E}_2$ using Proposition 2. Still, the verification is quite trivial.

Consider another example from the *REVERSE* program, the command '*tl* $k:=j$'.

Before the command, we have

$$(\exists a\gamma\beta)[*(k\overset{a}{\rightarrow}i\overset{\gamma}{\rightarrow}nil, j\overset{\beta}{\rightarrow}nil).\ rev(a\gamma)\beta=rev(K).\ Unit(a)] \tag{1}$$

The transformation sentence, putting $n=2$, is

$$*(k\overset{a}{\rightarrow}y, \lambda_1, \lambda_2)\Rightarrow$$
$$*'(k\overset{a}{\rightarrow}j, \lambda_1, \lambda_2), \text{ for all } a, y, \lambda_1, \lambda_2 \tag{2}$$

Rewriting the statement after the command with *' for *, we have

$$(\exists a\gamma\beta)[*'(k\overset{a}{\rightarrow}j\overset{\beta}{\rightarrow}nil, i\overset{\gamma}{\rightarrow}nil).\ rev(\gamma)a\beta=rev(K)] \tag{3}$$

We must prove this from (1) and (2).

Combining (1) with (2) we get

$$(\exists a\gamma\beta)[*'(k\overset{a}{\rightarrow}j, i\overset{\gamma}{\rightarrow}nil, j\overset{\beta}{\rightarrow}nil).\ rev(a\gamma)\beta=rev(K).\ Unit(a)]$$

But permuting the *' expression (Proposition 1 (*i*)) and using obvious properties of *rev* we get (3).

The sentences for *hd* and *cons* are used in an analogous way.

Because their meaning changes in a relatively complex way it is advisable to debar *hd* and *tl* from appearing in state descriptions and work in terms of * alone. We now consider how to reduce an expression involving *hd* or *tl* with respect to a state description so as to eliminate references to these. For example the expression $hd(tl(tl(i)))$ with respect to the state description $*(i\overset{a}{\rightarrow}x\overset{b}{\rightarrow}j\overset{c}{\rightarrow}y, \ldots)$ with $Unit(a)$, $Unit(b)$, $Unit(c)$ reduces to c. If such an expression occurs in a transformation sentence we reduce it to obtain an equivalent transformation sentence not involving *hd* or *tl*.

The reduction of an expression E with respect to a state description D, written $\hat{E}$, is defined recursively by

(i) If E is an identifier, a constant or a variable then $\hat{E}=E$

(ii) If E is $hd\,E_1$ and D contains $*(\hat{E}_1\overset{a}{\rightarrow}x, \ldots)$ and $Unit(a)$ then $\hat{E}=a$

(iii) If E is $tl\,E_1$ and D contains $*(\hat{E}_1 \overset{a}{\rightarrow} x, \ldots)$ and $Unit(a)$ then $\hat{E}=x$.

Proposition 2. $D \vdash \hat{E}=E$.

The proof is straightforward by induction on the structure of E, using the definition of *.

8. EXAMPLES OF PROOFS FOR LIST PROCESSING

The transformation sentences can best be understood by seeing how we use them in proving some simple list processing programs which alter the list structures. Although these programs are short their mode of operation, and hence their correctness, is not immediately obvious. We have already looked at the program *REVERSE*.

Another example, involving *cons*, is concatenation of two lists, copying the first and terminating it with a pointer to the second, see figure 7. Notice that the input lists need not necessarily be distinct. The other, destructive, method of concatenation is to overwrite the final *nil* of the first list with a pointer to the second. In this case our initial condition would have to be $*(k \overset{K}{\rightarrow} nil, l \overset{L}{\rightarrow} nil)$, ensuring distinctness.

Our next example, figure 8, is reversing a cyclic list, where instead of terminating with *nil* the list eventually reaches the starting cell again.

The next example, figure 9, involves a list with sub-lists. A list of lists are to be concatenated together and the *REVERSE* and *CONCAT* routines already defined are used. The suspicious reader may notice that we are making free with '. . .' in the assertions, but we can always replace $(s_1, \ldots, s_n)$ by, say, $(s_i)_{i=1}^{n}$ or $sequence(s, 1, n)$, so nothing mysterious is involved.

Some of the attractions of such proofs seems to come from the fact that the form of the assertions is graph-like and so (at least in figures 6, 7 and 8) strictly analogous to the structures they describe.

9. TREES

We will now pass from lists to trees. We will consider general 'record' or 'plex' structures with various kinds of cell each having a specified number of components, as for example in Wirth and Hoare (1966). A particular case is that of binary trees. Thus, instead of

$$hd\colon U \rightarrow A \cup U^0$$
$$tl\colon U \rightarrow U^0$$

we now allow

$$hd\colon U \rightarrow A \cup U$$
$$tl\colon U \rightarrow A \cup U$$

(*nil* has no special place in binary trees; it can be considered an atom like any other).

More generally we may replace trees built using *cons* with trees (terms or expressions) built using any number of operators with arbitrary numbers of arguments, corresponding to different kinds of records.

$m = CONCAT(k, l)$
K and L are constant strings of atoms, a and b units.

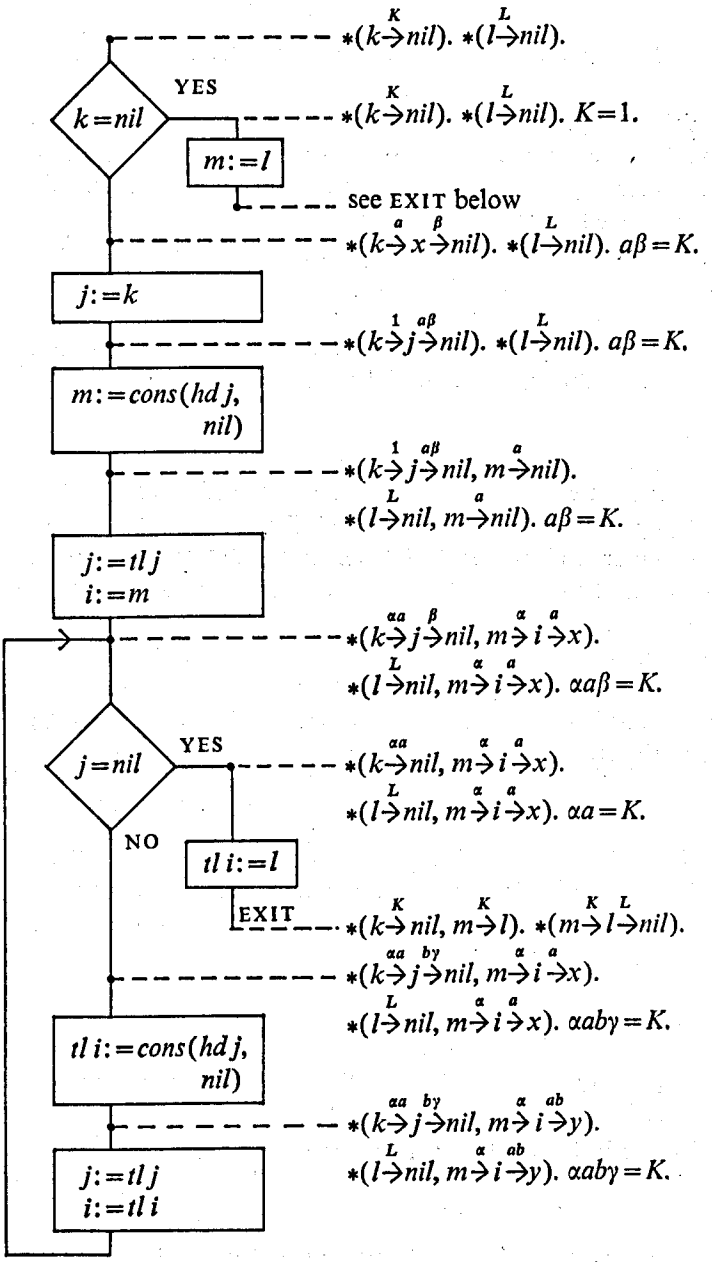


Figure 7. Concatenation.

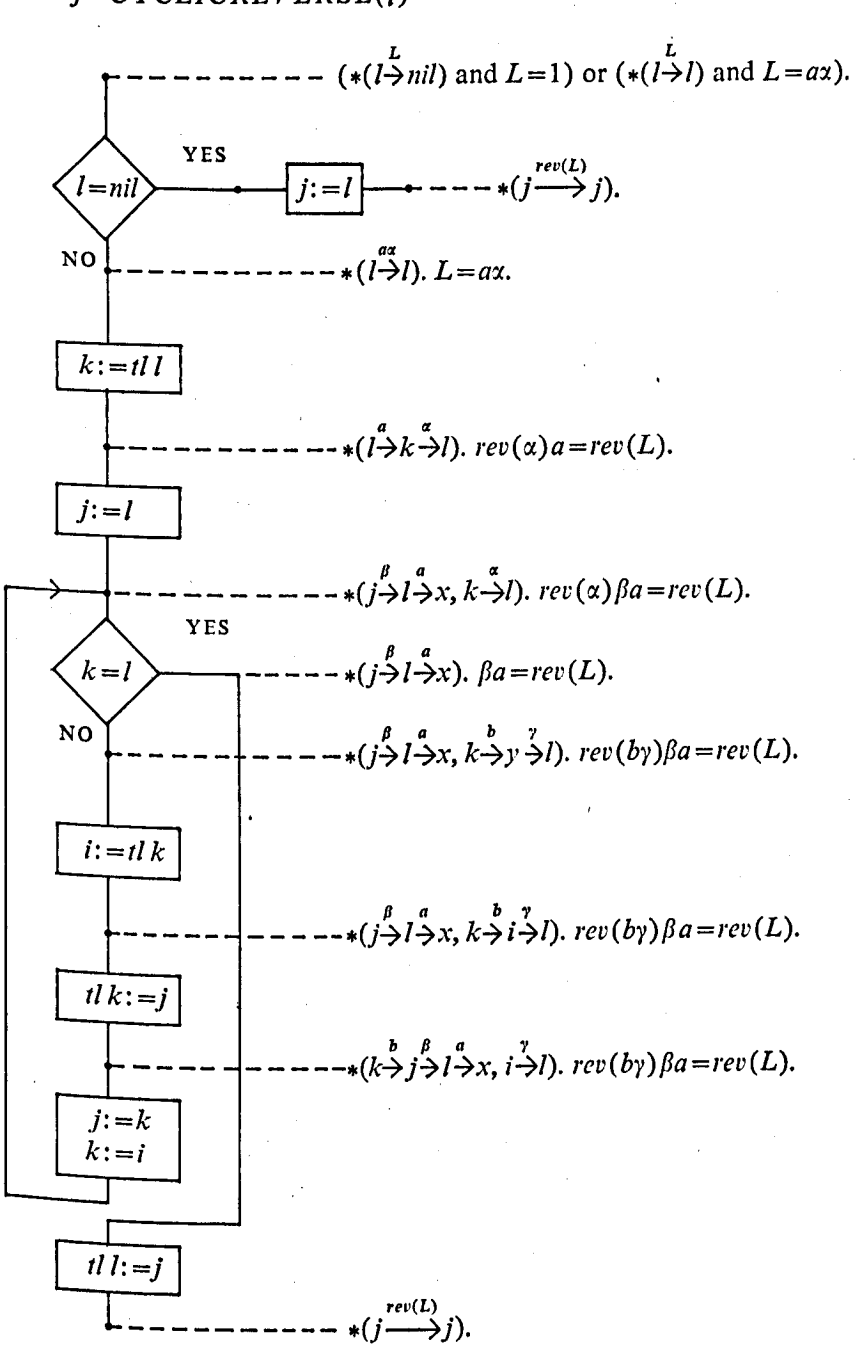


Figure 8. Reversing a cyclic list.

$l = MULTICONCAT(i)$

Assume $N, C_1, \ldots, C_N, A_1, \ldots, A_N$ are constants.

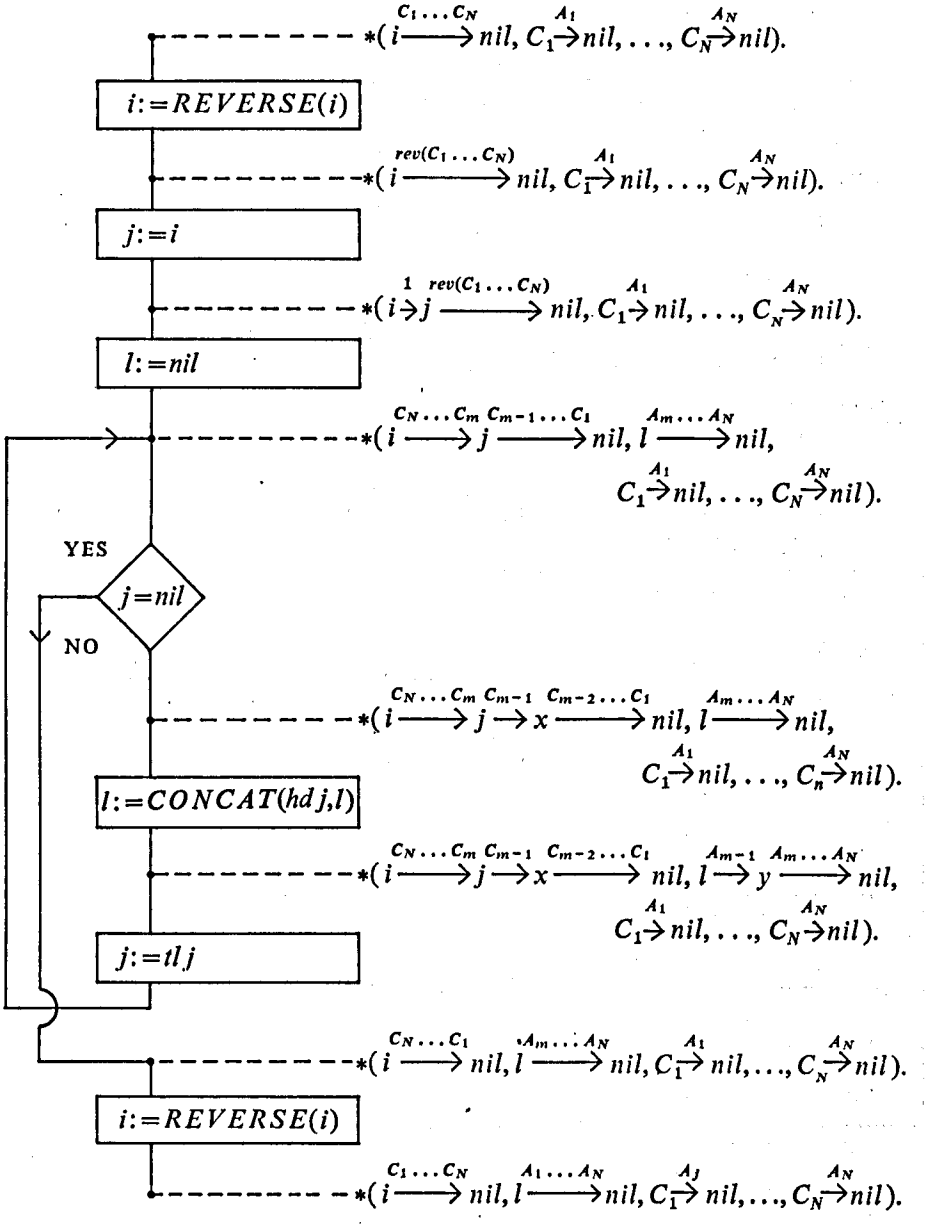


Figure 9. Multiple concatenation.

Our previous discussion depended rather heavily on the concatenation operation for lists. Fortunately Lawvere (1963) with his notion of a 'Free Theory' has shown how concatenation can be extended from strings to trees (see also Eilenberg and Wright 1967). His category theory approach is convenient in view of the remark in an earlier section that $\mathscr{L}$ forms a category. We will not present our ideas in category-theoretic language, since this may not be familiar to some readers, but will be content to point out where the concept of category is appropriate.

Our first task is to define a system of trees or expressions with a composition operation, analogous to the strings used above. We will then apply them to tree-like data structures under assignment commands.

Let $\Omega=\{\Omega_n\}$, $n=0, 1, \ldots$ be a sequence of sets of operators, $\{x_i\}$, $i=1, 2, \ldots$ a sequence of distinct variables, and let $X_m=\{x_1, \ldots, x_m\}$.

We define a sequence of sets of terms (or trees) $T=\{T_\Omega(X_m)\}$, $m=0, 1, \ldots$. $T_\Omega(X_m)$ means all terms, in the usual sense of logic, in the operators Ω and variables X_m. We can regard $T_\Omega(X_m)$ as a subset of the strings $(\bigcup\Omega\cup X_m)^*$ and define it inductively thus

(i) $x\in T_\Omega(X_m)$ if $x\in X_m$

(ii) $\omega t_1 \ldots t_n\in T_\Omega(X_m)$ if $\omega\in\Omega_n$ for some n and $t_1, \ldots, t_n\in T_\Omega(X_m)$.

(Algebraically speaking $T_\Omega(X_m)$ forms the word algebra or generic algebra over Ω with generating set X_m.)

If $t\in T_\Omega(X_m)$ and $(s_1, \ldots, s_m)\in T_\Omega(X_n)^m$, $m\geqslant 0$, by the composition $t\cdot(s_1, \ldots, s_m)$ we mean the term in $T_\Omega(X_n)$ obtained by substituting s_i for x_i in t, $i=1, \ldots, m$.

To preserve the analogy with our previous treatment of strings we would like to make composition associative. For this we consider n-tuples of trees. Thus, more generally, if $(t_1, \ldots, t_l)\in T_\Omega(X_m)^l$ and $(s_1, \ldots, s_m)\in T_\Omega(X_n)^m$, we define the composition by

$$(t_1, \ldots, t_l)\cdot(s_1, \ldots, s_m)=(t_1\cdot(s_1, \ldots, s_m), \ldots, t_l\cdot(s_1, \ldots, s_m)).$$

This composition is in $T_\Omega(S_m)^l$.

For example if $\Omega_2=\{cons\}$, $\Omega_0=\{a, b, c, \ldots\}$, and we allow ourselves to write parentheses in the terms for readability

$$(cons(x_1, a), cons(x_2, x_1), b)\in T_\Omega(X_2)^3$$

$$(c, x_1)\in T_\Omega(X_1)^2$$

and their composition is

$$(cons(c, a), cons(x_1, c), b)\in T_\Omega(X_1)^3.$$

The composition is now associative and $(x_1, \ldots, x_n)\in T_\Omega(X_n)^n$ is an identity for each n. It is, however, a partial operation (compare matrix multiplication, which is only defined for matrices of matching sizes).

We see now that the disjoint union of the $T_\Omega(X_n)$ forms a category, T_Ω, with as objects the integers $n=0, 1, \ldots$ and with a set of morphisms $T_\Omega(X_n)^m$ from m to n for each m, n. Indeed, this is just what Lawvere calls the Free Theory on Ω. Eilenberg and Wright (1967) use it to extend automata theory to trees as well as strings, giving a category theory presentation (see

also Arbib and Giv'eon (1968)). The category T_Ω replaces the monoid $(A \cup U)^*$ used in our treatment of lists, the main difference being that the composition operation, unlike string concatenation, is only partial. The strings over an alphabet Σ can be regarded as a special case by taking Ω_1 to be Σ and Ω_0 to be $\{nil\}$. In the Appendix we summarise the abstract definition of a Free Theory as given by Eilenberg and Wright. We will not use any category theory here, but it is perhaps comforting to know what kind of structure one is dealing with.

We will consider a fixed Ω and X for the time being and write T_{mn} for $T_\Omega(X_n)^m$, the set of m-tuples of terms in n variables. If $\omega \in \Omega_n$ we will take the liberty of writing ω for the term $\omega x_1 \ldots x_n$ in T_{1n}. This abbreviation may appear more natural if we think of an element τ of T_{1n} as corresponding to the function $\lambda x_1 \ldots x_n \cdot t$ from $T_\Omega(\varnothing)^n$ to $T_\Omega(\varnothing)$. For example *cons* is short for $\lambda x_1, x_2 \cdot cons(x_1, x_2)$, but not of course for $\lambda x_1, x_2 \,.\, cons(x_2, x_1)$.

We will write 1 for the identity $(x_1, \ldots, x_n) \in T_{nn}$ and 0 for the 0-tuple $() \in T_{0n}$. We will sometimes identify the 1-tuple (x) with x.

10. STATE DESCRIPTIONS USING TREES

We can now use the m-tuples of terms, T_{mn}, just as we previously used strings to provide state descriptions.

Suppose now that each $\omega \in \Omega_n$, $n=0, 1, \ldots$, corresponds to a class of cells U_ω (records) with n-components and that these components can be selected by functions

$$\delta_i^\omega : U_\omega \to \bigcup \{U_\omega : \omega \in \bigcup \Omega\}, \quad i=1, \ldots, n$$

We put $U = \bigcup \{U_\omega : \omega \in \bigcup \Omega\}$.

For example if $\Omega_2 = \{cons\}$, $\Omega_0 = A$ and $\Omega_i = \varnothing$ for $\omega \neq 0$ or 2 then

δ_1^{cons} is hd: $U_{cons} \to U_{cons} \bigcup A$

δ_2^{cons} is tl: $U_{cons} \to U_{cons} \bigcup A$

(here we have put $U_a = \{a\}$ for each $a \in A$, and U_{cons} is just our previous U, the set of list cells).

A state is defined by the U_ω and the δ_i^ω and we associate with it a set $\mathscr{T}$ of triples, just as the set $\mathscr{L}$ was previously associated with a state.

If a triple (τ, u, v) is in $\mathscr{T}$ then for some m, n, $u \in U^m$, $v \in U^n$ and $\tau \in T_{mn}$. As before we write $u \overset{\tau}{\to} v$ for (τ, u, v). We define $\mathscr{T}$ inductively thus:

(i) If $\tau \in T_{mn}$ is $(x_{i_1}, \ldots, x_{i_m})$ and $i_j \in \{1, \ldots, n\}$ for $j=1, \ldots, m$ (that is, τ involves only variables and not operators) then $(u_{i_1}, \ldots, u_{i_m}) \overset{\tau}{\to} (u_1, \ldots, u_n)$ is in $\mathscr{T}$.

(ii) If $u \overset{\tau}{\to} v \in \mathscr{T}$ and $v \overset{\sigma}{\to} w \in \mathscr{T}$ then their composition $(u \overset{\tau}{\to} v) \cdot (v \overset{\sigma}{\to} w)$ is defined as $u \overset{\tau \cdot \sigma}{\to} w$, and it is in $\mathscr{T}$.

(iii) If $(u_1) \overset{(t_1)}{\to} v, \ldots, (u_n) \overset{(t_n)}{\to} v$ are in $\mathscr{T}$ then $(u_1, \ldots, u_n) \overset{(t_1, \ldots, t_n)}{\longrightarrow} v$ is in $\mathscr{T}$.

(iv) If $\omega \in \Omega_n$ and $\delta_i^\omega(u) = v_i$ for $i=1, \ldots, n$ then $(u) \overset{\omega}{\to} (v_1, \ldots, v_n)$ is in $\mathscr{T}$.

Taking the case where Ω consists of *cons* and atoms, (iv) means that $(u) \overset{cons}{\rightarrow} (v, w) \in \mathscr{T}$ if $hd(u)=v$ and $tl(u)=w$, also that $(a) \overset{a}{\rightarrow} ()$ is in $\mathscr{T}$ for each atom $a \in A$.

An example will make this clearer. In the state pictured in figure 10 $\mathscr{T}$ contains the following triples, amongst others,

(i) $(i) \xrightarrow{cons(cons(cons(c,d),a),cons(cons(c,d),b))} ()$

(ii) $(i) \xrightarrow{cons(cons(x_1,a),cons(x_1,b))} (j)$

(iii) $(j) \xrightarrow{cons(c,d)} ()$

(iv) $(i) \xrightarrow{cons(cons(x_1,a),x_2)} (j, k)$

(v) $(l) \xrightarrow{cons(x_1,x_1)} (l)$

The first of these is the composition of the second and third.

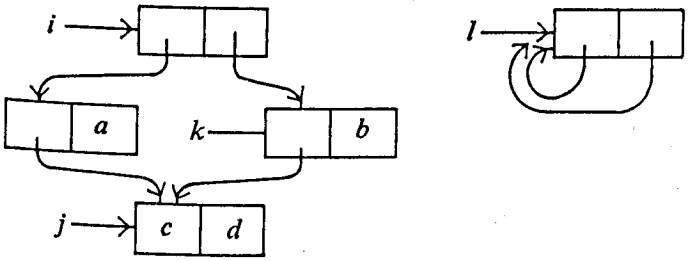


Figure 10.

It will be noticed that $\mathscr{T}$ forms a category with objects u for each $u \in U^n$, $n=0, 1, \ldots$. The triples (τ, u, v) in $\mathscr{T}$ are the morphisms from u to v. There is a forgetful functor *represents*: $\mathscr{T} \rightarrow T$.

We can now define a Distinct Non-repeating Tree System analogous to a Distinct Non-repeating List System. We take over the definitions of Distinctness and Non-repetition almost unchanged except that they now apply to $\mathscr{T}$ rather than to $\mathscr{L}$.

We call $(\tau, u, v) \in \mathscr{T}$ *elementary* if τ involves variables and operators in Ω_0 but none in Ω_n, $n \geqslant 1$ (for example, trees which share only elementary constituents can share atoms but not list cells).

We now define, as for lists, the number of occurrences of a cell in an n-tuple of trees, thus

$$\delta: U \times \mathscr{T} \rightarrow N$$

(i) $\delta_u(v \overset{\tau}{\rightarrow} w)=0$ if u is elementary

(ii) $\left.\begin{aligned} \delta_u((x) \overset{\omega}{\rightarrow} v \overset{\tau}{\rightarrow} w) &= \delta_u(v \overset{\tau}{\rightarrow} w)+1 \text{ if } x=u \\ &= \delta_u(v \overset{\tau}{\rightarrow} w) \quad \text{if } x \neq u \end{aligned}\right\}$ for $\omega \in \Omega_n$, $n>0$

(iii) $\delta_u((u_1, \ldots, u_n) \xrightarrow{(\tau_1, \ldots, \tau_n)} v) = \delta_u((u_1) \xrightarrow{\tau_1} v) + \cdots + \delta_u((u_n) \xrightarrow{\tau_n} v)$

Notice that for $a \in \Omega_0$ we have $\delta_a((a) \xrightarrow{a} ()) = 0$, since we do not wish to notice sharing of atoms, which is innocuous.

The definitions of *Distinct* and *Nonrep* are as before replacing $\mathscr{L}$ by $\mathscr{T}$. Lemma 1 holds unchanged with the obvious addition that Distinctness and Non-repetition are preserved by formation of n-tuples in the same way as by composition.

The definition of a Distinct Non-repeating Tree System is, as before, a k-tuple of elements of $\mathscr{T}$, λ_i, $i = 1, \ldots, k$, such that

(i) λ_i is in $\mathscr{T}$ for $i = 1, \ldots, k$

(ii) $Nonrep(\lambda_i)$ for each $i = 1, \ldots, n$

(iii) If $j \neq i$ then $Distinct(\lambda_i, \lambda_j)$ for each $i, j \in 1, \ldots, k$.

We employ the same abbreviation as before, writing $*S$ for 'S is a Distinct Non-repeating Tree System'.

We can adapt Proposition 1 and specify some useful properties of such systems.

Proposition 3.

(i) *Permutation*

$*S \Rightarrow *S'$ if S' is a permutation of S.

(ii) *Deletion*

$*(\lambda_1, \ldots, \lambda_k) \Rightarrow *(\lambda_2, \ldots, \lambda_k)$.

(iii) *Rearrangement of variables*

$$*(\lambda_1, \ldots, \lambda_k) \Rightarrow *((u_{i_1}, \ldots, u_{i_m}) \xrightarrow{(x_{i_1}, \ldots, x_{i_m})} (u_1, \ldots, u_n), \lambda_1, \ldots, \lambda_k)$$

if $i_j \in \{1, \ldots, n\}$ for $j = 1, \ldots, m$.

(iv) *Composition*

$$*(u \xrightarrow{\tau} v \xrightarrow{\sigma} w, \lambda_1, \ldots, \lambda_k) \Leftrightarrow *(u \xrightarrow{\tau \cdot \sigma} w, \lambda_1, \ldots, \lambda_k)$$

and conversely

$$*(u \xrightarrow{\tau \cdot \sigma} w, \lambda_1, \ldots, \lambda_k) \Rightarrow (\exists v)\ *(u \xrightarrow{\tau} v \xrightarrow{\sigma} w, \lambda_1, \ldots, \lambda_k)$$

(v) *Tupling*

$$*((u_1) \xrightarrow{\tau_1} v, \ldots, (u_m) \xrightarrow{\tau_m} v, \lambda_1, \ldots, \lambda_k)$$
$$\Leftrightarrow *((u_1, \ldots, u_m) \xrightarrow{(\tau_1, \ldots, \tau_m)} v, \lambda_1, \ldots, \lambda_k) \quad (m \geqslant 0).$$

(vi) *Distinctness*

$*(u \xrightarrow{\tau} v, u \xrightarrow{\sigma} w) \Rightarrow u \xrightarrow{\tau} v$ is elementary

or $u \xrightarrow{\sigma} w$ is elementary.

(vii) *Inequality*

$*((u) \xrightarrow{\tau} (v_1, \ldots, v_n))$ and $u \neq v_i$, $i = 1, \ldots, n$
and $u \in U_\omega \Rightarrow (\exists \sigma)(\tau = \omega \cdot \sigma)$.

Proof. Obvious using Lemma 1 adapted for trees.

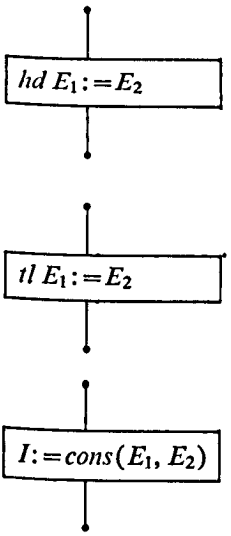
In the case of *cons* and atoms we assume a predicate *Atom*,

$Atom(u) \Leftrightarrow u \in A$, that is, $u \in U_a$ for some $a \in A$,

and (vii) yields

$*((u) \overset{\tau}{\rightarrow} (v_1, \ldots, v_n))$ and $u \neq v_i$, $i=1, \ldots, n$ and $Atom(u) \Rightarrow$
$(\exists \sigma)\tau = u \cdot \sigma$
and $\neg Atom(u) \Rightarrow (\exists \sigma)\tau = cons \cdot \sigma$.

In figure 11 we give the transformation sentences for this case. Their correctness is provable easily in just the same manner as those for lists. The extension to a general Ω should be obvious. The sentences for $I:=E$ and $E_1=E_2$ are unchanged and for the test $Atom(E)$ we just add the sentence $Atom(E)$ or its negation. Proposition 2 and the definition of E go through unchanged.



Change set $=\{hd, *\}$
$*((E_1) \overset{cons}{\rightarrow} (u, v), \lambda_1, \ldots, \lambda_n) \Rightarrow$
$*'((E_1) \overset{cons}{\rightarrow} (E_2, v), \lambda_1, \ldots, \lambda_n)$

Change set $=\{tl, *\}$
$*((E_1) \overset{cons}{\rightarrow} (u, v), \lambda_1, \ldots, \lambda_n) \Rightarrow$
$*'((E_1) \overset{cons}{\rightarrow} (u, E_2), \lambda_1, \ldots, \lambda_n)$

Change set $=\{*, I\}$
$*(\lambda_1, \ldots, \lambda_n) \Rightarrow$
$*'((I') \overset{cons}{\rightarrow} (E_1, E_2), \lambda_1, \ldots, \lambda_n)$

Figure 11. Transformation sentences for tree processing.

We now give a couple of examples of tree processing (or perhaps we should say 'expression processing'). Figure 12 gives a program in the form of a recursive subroutine for reversing a tree; for example $cons(a, cons(b, c))$ becomes $cons(cons(c, b), a)$. We have identified the one-tuple (τ) with τ to save on parentheses. The proof is by induction on the structure of τ. Strictly we are doing induction on the free theory T. We define a tree reversing function recursively by

$rev(a) = a$ if a is an atom
$rev(cons(\sigma, \tau)) = (cons(rev(\tau), rev(\sigma))$.

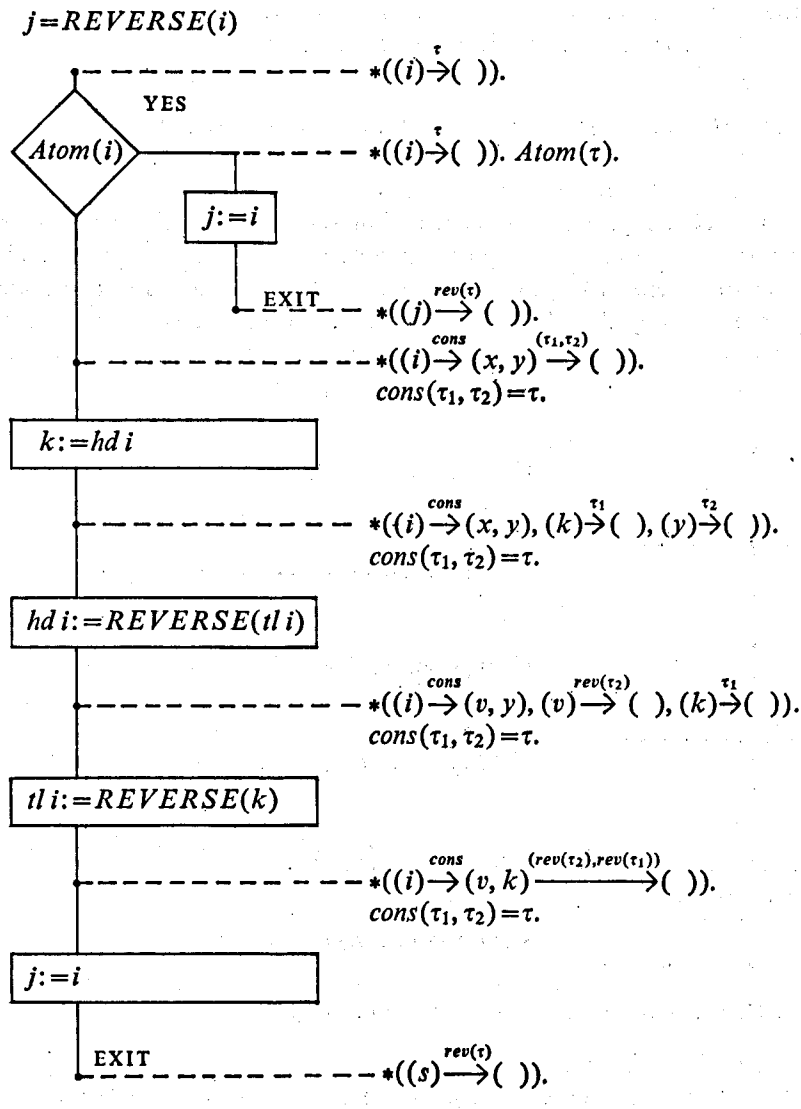


Figure 12. Reversing a tree.

$k = SUBST(i, a, j)$. Substitute i for a in j. a is an atom.

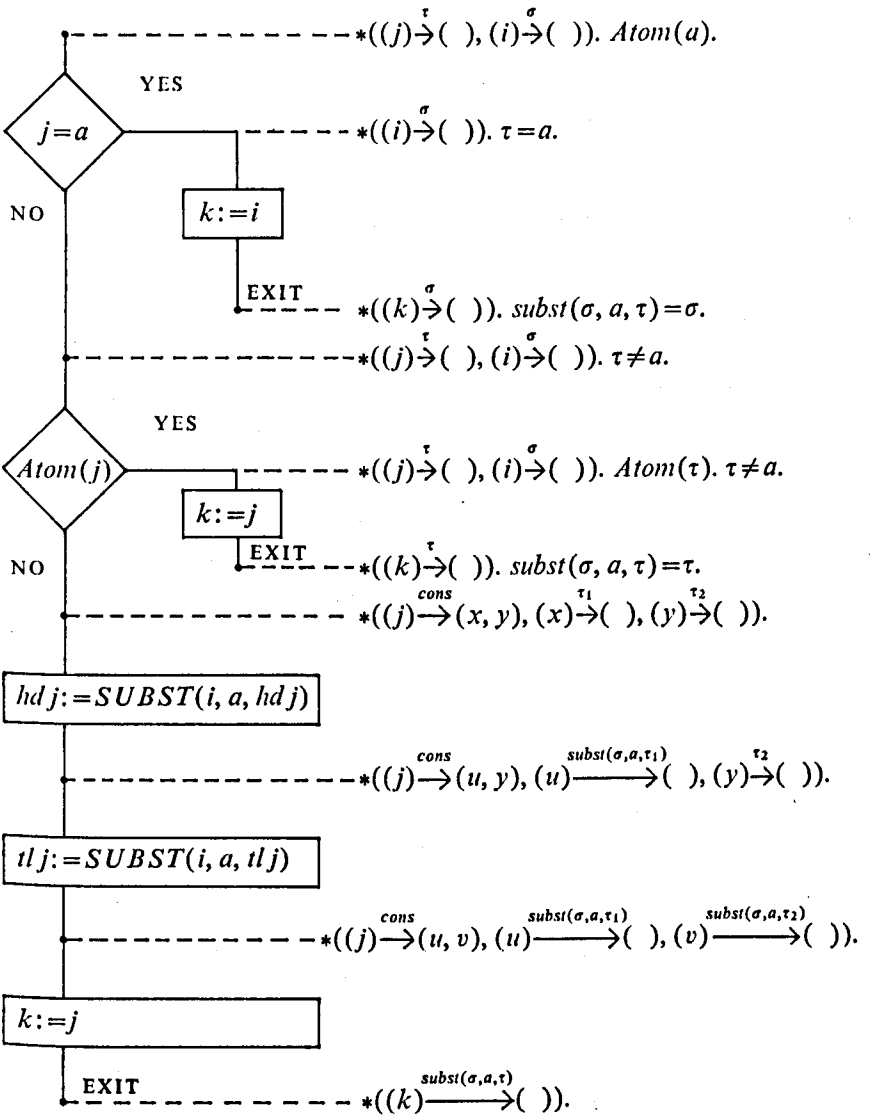


Figure 13. Substitution in a tree.

Figure 13 gives a recursive subroutine for substituting one tree into another wherever a given atom appears. Again the proof is by induction on the structure of τ. We define substitution for morphisms of T by

$subst(\sigma, a, a) = \sigma$

$subst(\sigma, a, b) = b$ if $b \neq a$ and b is an atom

$subst(\sigma, a, cons(\tau_1, \tau_2)) = cons(subst(\sigma, a, \tau_1), subst(\sigma, a, \tau_2))$.

We should really have included some arbitrary extra morphisms λ_i in the * expressions at entry and exit so as to carry through the induction, since the recursive calls act in the presence of other distinct morphisms; but this is clearly admissible and we have omitted it.

In our examples we have used the LISP case of *cons* and atoms, but even for LISP programs it might be useful to consider a more general Ω. List structures are often used to represent expressions such as arithmetic expressions, using $cons('PLUS', cons(x, cons(y, nil)))$ for '$x+y$', similarly for '$x \times y$'. We can then allow T to have binary operators $+$ and $\times$ defining $(u) \overset{+}{\rightarrow} (v, w)$ if $hd(u) = 'PLUS'$ and $hd(tl(u)) = v$ and $hd(tl^2(u)) = w$. This enables us to write the assertions intelligibly in terms of $+$ and $\times$. In fact this representation of expressions corresponds to an injective functor between the category $T_{+,\times}$ and the category $T_{cons,A}$.

Free Theories as above seem appropriate where the trees have internal sharing only at known points. Data structures other than lists and trees remain to be investigated. More general categories of graphs with inputs and outputs might be worth considering. In general the choice of a suitable category analogous to T would seem to depend on the subject matter of the computation, since we wish the vocabulary of the assertions to be meaningful to the programmer.

Acknowledgements

I would like to thank John Darlington and Gordon Plotkin for helpful discussions and Michael Gordon for pointing out some errors in a draft, also Pat Hayes and Erik Sandewall for helpful criticism during the Workshop discussion. I am grateful to Miss Eleanor Kerse for patient and expert typing. The work was carried out with the support of the Science Research Council. A considerable part of it was done with the aid of a Visiting Professorship at Syracuse University, for which I would like to express my appreciation.

REFERENCES

Arbib, M.A. & Give'on, Y. (1968) Algebra automata I: parallel programming as a prolegomena to the categorical approach. *Information and Control*, **12**, 331–45. Algebra automata II: the categorical framework for dynamic analysis. *Information and Control*, **12**, 346–70. Academic Press.

Burstall, R.M. (1969) Proving properties of programs by structural induction. *Comput. Jl.*, **12**, 41–8.

Burstall, R.M. (1970) Formal description of program structure and semantics in first order logic. *Machine Intelligence 5*, pp. 79–98 (eds Meltzer, B. & Michie, D.). Edinburgh: Edinburgh University Press.

Eilenberg, S. & Wright, J.B. (1967) Automata in general algebras. *Information and Control*, **11**, 4, 452–70.
Floyd, R.W. (1967) Assigning meanings to programs. *Mathematical Aspects of Computer Science*, 19–32. Providence, Rhode Island: Amer. Math. Soc.
Good, D.I. (1970) Toward a man-machine system for proving program correctness. Ph.D. thesis. University of Wisconsin.
Hewitt, C. (1970) PLANNER: a language for manipulating models and proving theorems in a robot. *A.I. Memo. 168*. Project MAC. Cambridge, Mass.: MIT.
Hoare, C.A.R. (1971) Proof of a program: FIND. *Comm. Ass. comput. Mach.*, **14**, 39–45.
King, J.C. (1969) A program verifier. Ph.D. thesis. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania.
Lawvere, F.W. (1963) Functorial semantics of algebraic theories. *Proc. Natl. Acad. Sci. U.S.A.*, **50**, 869–72.
London, R.L. (1970) Certification of Algorithm 245 – TREESORT 3. *Comm. Ass. comput. Mach.* **13**, 371–3.
MacLane, S. (1971) *Categories for the working mathematician*. Graduate Texts in Mathematics 5. Springer-Verlag.
McCarthy, J. (1963) A basis for a mathematical theory of computation. *Computer Programming and Formal Systems*, pp. 33–70. (eds Braffort, P. & Hirschberg, D.). Amsterdam: North Holland Publishing Co.
McCarthy, J. & Painter, J.A. (1967) Correctness of a compiler for arithmetic expressions. *Mathematical Aspects of Computer Science*, pp. 33–41. Providence, Rhode Island: Amer. Math. Soc.
Wirth, N. & Hoare, C.A.R. (1966) A contribution to the development of ALGOL. *Comm. Ass. comput. Mach.*, **9**, 413–32.

APPENDIX: FREE THEORIES

Categories

By a category C we mean a set O of objects and a set M of morphisms, together with a pair of functions $\partial_0, \partial_1 : M \to O$ (domain and co-domain), a partial operation of composition, $\cdot : M \to M$, and an identity operation $1 : O \to M$ such that

(i) $f \cdot g$ is defined iff $\partial_1 f = \partial_0 g$
(ii) If $f \cdot g$ and $g \cdot h$ are both defined then $(f \cdot g) \cdot h = f \cdot (g \cdot h)$
(iii) $\partial_0(1_a) = \partial_1(1_a) = a$
(iv) If $\partial_0 f = a$ and $\partial_1 f = b$ then $1_a \cdot f = f = f \cdot 1_b$

We write $f : a \to b$ as an abbreviation for $\partial_0 f = a$ and $\partial_1 f = b$ and write $C(a, b)$ for the set of all f such that $f : a \to b$ in C. (For further development see, for example, MacLane 1971.)

Functions over finite sets

For each integer $n \geqslant 0$, we write $[n]$ for the set $\{1, \ldots, n\}$.

We write $\varnothing$ for $[0]$ and I for $[1]$ and notice that there are unique functions $\varnothing \to [n]$ and $[n] \to I$ for any n. A set $[n]$ and an integer $i \in [n]$ determine a unique function $I \to [n]$ which we will denote by i.

The free theory on Ω

Let $\Omega=\{\Omega_n\}$, $n=0, 1, \ldots$ be a sequence of sets of operators.

We define the *free theory* on Ω inductively as the smallest category T such that

(i) The objects of T are the sets $[n]$, $n=0, 1, \ldots$

(ii) There is a function d from the morphisms of T to the non-negative integers. We call $d(f)$ the degree of the morphism f, and write $T_j([m], [n])$ for the set of all morphisms of degree j from m to n.

(iii) $T_0([n], [m])$ is the set of all functions from the set $[n]$ to the set $[m]$. Composition and identity are defined as usual for functions.

(iv) There is an operation of 'tupling' on morphisms from I (as well as composition and identity). Thus for each n-tuple of morphisms $f_i: I\to[k]$, $i=1, \ldots, n$ there is a unique morphism f, written $\langle f_1, \ldots, f_n\rangle$ such that $i\cdot f=f_i$ for each $i=1, \ldots, n$. (Recall that $i: 1\to[n]$, is the function taking 1 to i in $[n]$.) From the uniqueness we see that for any $f: [n]\to[k]$, $f=\langle 1\cdot f, \ldots, n\cdot f\rangle$.

The degree of $\langle f_1, \ldots, f_n\rangle$ is $d(f_1)+\ldots+d(f_n)$.

(v) $\Omega_n\subseteq T_1(I,[n])$.

(vi) If $\omega\in\Omega_m$ and $f\in T_j([m], [n])$ then $\omega\cdot f\in T_{1+j}(I, [n])$, and conversely if $g\in T_{1+j}(I, [n])$ for $j\geqslant 0$ then there is a unique m, a unique $\omega\in\Omega_m$ and a unique f such that $g=\omega\cdot f$.

Now T_{mn} in the body of the paper may be equated (to within an isomorphism) with $\bigcup_j T_j([m], [n])$ here.

3

Proving Compiler Correctness in a Mechanized Logic

R. Milner and R. Weyhrauch
Computer Science Department
Stanford University

Abstract

We discuss the task of machine-checking the proof of a simple compiling algorithm. The proof-checking program is LCF, an implementation of a logic for computable functions due to Dana Scott, in which the abstract syntax and extensional semantics of programming languages can be naturally expressed. The source language in our example is a simple ALGOL-like language with assignments, conditionals, whiles and compound statements. The target language is an assembly language for a machine with a pushdown store. Algebraic methods are used to give structure to the proof, which is presented only in outline. However, we present in full the expression-compiling part of the algorithm. More than half of the complete proof has been machine checked, and we anticipate no difficulty with the remainder. We discuss our experience in conducting the proof, which indicates that a large part of it may be automated to reduce the human contribution.

INTRODUCTION

In this paper we describe our experiences in trying to use a mechanized version of a logic for computable functions, LCF (Milner 1972a,b; Weyhrauch and Milner 1972), to express and formally prove the correctness of a compiler. This logic is based on the theory of the typed lambda calculus, augmented by a powerful induction rule, suggested in private communications with Dana Scott. More particularly: (1) We show how to define in LCF an extensional semantics for our target language T which contains an unrestricted jump instruction. This definition provides, in a direct manner, a single recursive definition MT for the semantics of a program. This contrasts with the approach of McCarthy (1963) where each program is assigned a set of mutually recursive function definitions as its semantics. (2) We give a description, using algebraic methods, of the proof of the correctness of a

compiling algorithm for a simple ALGOL-like source language *S*. (3) We present in its entirety a machine-checked proof of the correctness of an algorithm for compiling expressions. We call this the McCarthy-Painter lemma, as it is essentially the algorithm proved correct by them (McCarthy and Painter 1967).

The question of rigorous verification of compilers has already been the subject of considerable research. As we mentioned, McCarthy and Painter have given a proof for an expression-compiling algorithm. Kaplan (1967) and Painter (1967) have verified compiling algorithms for source languages of about the same complexity as ours; in both cases the source language contains jump instructions, whereas our source language with conditional and while statements is in the spirit of the 'goto-less' programming advocated by Djikstra (1968) and others. Burstall and Landin (1969) first explored the power of algebraic methods in verifying a compiler for expressions, and in pursuing this exploration with a more powerful language we have been helped by discussions with Lockwood Morris, whose forthcoming doctoral thesis (1972) is concerned with this topic. London (1971, 1972) has given a rigorous proof of two compilers for a LISP subset. All these authors have looked forward to the possibility of machine-checked compiler proofs, and Diffie (1972) has successfully carried out such a proof for the expression compiler of McCarthy and Painter, using a proof-checking program for the First Order Predicate Calculus written by him. We believe that LCF has advantages over First Order Predicate Calculus for the expression and proof of compiler correctness, and our current paper is in part an attempt to justify this belief. Briefly, the advantages of LCF consist in its orientation towards partial computable functions, and functions of higher type. For example, we consider that the meaning of a program is a partial computable function from states to states, where a state is conveniently represented (at least for the simple source language which we consider) as a function from names to values.

THE SOURCE LANGUAGE *S*

Expressions in *S* are built up from names by the application of binary operators. The collection of such operators is an entirely arbitrary but fixed set. Thus expressions are defined in LCF by the equations

$$iswfse \equiv \alpha f.\ wfsefun(f),$$
$$wfsefun \equiv \lambda f\ e.\ type(e) = _N \rightarrow TT,$$
$$type(e) = _E \rightarrow isop(opof(e)) \wedge f(arg1of(e)) \wedge f(arg2of(e)),$$
$$UU,$$

that is, well-formed source expressions are just those individuals on which *iswfse* (which is meant to abbreviate 'is well-formed source expression') takes the value TT. Here '$\alpha f.\ F(f)$' denotes the least fixed point of the functional F, and 'TT', 'UU' denote the truth-values true and undefined respectively. The '$\rightarrow$' denotes the McCarthy conditional operator; $(p \rightarrow q, r)$ means if p

and q, else r, and in LCF is undefined if p is undefined. A more detailed description of the terms of LCF can be found in Milner (1972a). The 'type' of an object is determined axiomatically, for example $type(e) = _N$ is true just in case e is a name.

There are assignment, conditional, and while statements as well as compound statements formed by pairing any two statements. A well-formed source program is just any statement, that is,

$$iswfs \equiv \alpha f.\ wfsfun(f),$$
$$wfsfun \equiv \lambda fp.\ type(p) = _A \rightarrow type(lhsof(p)) = _N \wedge iswfse(rhsof(p)),$$
$$type(p) = _C \rightarrow iswfse(ifof(p)) \wedge f(thenof(p)) \wedge f(elseof(p)),$$
$$type(p) = _W \rightarrow iswfse(testof(p)) \wedge f(bodyof(p)),$$
$$type(p) = _CM \rightarrow f(firstof(p)) \wedge f(secondof(p)),$$
$$UU.$$

Of course in LCF programs are expressed by means of an abstract syntax for S, using appropriate constructors and selectors, some of which appear in the equation above. A complete list of the axioms for the abstract syntax is found below in Appendix 1.

We consider that the meaning of a program in S is a statefunction, that is, a function that maps states on to states, where a state is a function from the set of names to the set of values. Thus the meaning function MSE for expressions is a function which 'evaluates' an expression in a state.

$$MSE \equiv \alpha M.\ \lambda e\ sv.$$
$$type(e) = _N \rightarrow sv(e),$$
$$type(e) = _E \rightarrow$$
$$(MOP(opof(e)))(M(arg1of(e), sv), M(arg2of(e), sv)),$$
$$UU.$$

That is, to give the meaning of an expression e in a state sv, we compute as follows: if e is a name, look up e in the state, that is, evaluate $sv(e)$; if e is a compound expression, $opof(e)$ is the selector which computes from e its operator symbol, and MOP is a function which maps an operator symbol onto a binary function, which is then applied to the meanings of the subexpressions of e.

The following combinators are used in defining the semantics MS of S:

$$ID \equiv \lambda x.\ x,$$
$$WHILE \equiv \alpha g.\ \lambda q f.\ COND(q, f \otimes g(q,f), ID),$$
$$COND \equiv \lambda q f g\ s.\ (q(s) \rightarrow f(s), g(s)),$$
$$\otimes \equiv \lambda f g\ x.\ g(f(x)).$$
$$SCMPND \equiv \lambda f g.\ f \otimes g,$$
$$SCOND \equiv \lambda e f g.\ COND(MSE(e) \otimes true, f, g).$$
$$SWHILE \equiv \lambda e f.\ WHILE(MSE(e) \otimes true, f).$$

MS is now defined as

$$MS \equiv \alpha M.\ \lambda p.$$
$$type(p) = _A \rightarrow [\lambda sv\ m.\ m = lhsof(p) \rightarrow MSE(rhsof(p), sv), sv(m)],$$
$$type(p) = _C \rightarrow SCOND(ifof(p))(M(thenof(p)), M(elseof(p))),$$

$type(p) = _W \rightarrow SWHILE(testof(p))(M(bodyof(p))),$
$type(p) = _CM \rightarrow SCMPND(M(firstof(p)), M(secondof(p))),$
$UU.$

Several things should be noted about this definition. First of all there are no boolean expressions *per se*. Their place is taken by the function '*true*' which yields 'TT' just on those values which we wish to let represent true. This corresponds to the LISP convention, for example, where NIL is false and any other expression is considered true. Secondly, if the program p is an assignment (that is, $type(p) = _A$) it has been treated asymmetrically from the others. The reason for this will appear below.

THE TARGET LANGUAGE T

Our target language T is an elementary assembly language which contains unrestricted jumps and manipulates a stack. We consider the meaning of a program p in T to be a storefunction, that is, a function from stores to stores. A store sp is a pair consisting of a state sv and a list pd called the pushdown. We use '|' as the constructor for pairing a state and a pushdown, and '*svof*', '*pdof*' as the respective selectors. The following axioms hold:

$\forall sv\, pd.\quad svof(sv|pd) \equiv sv,$
$\forall sv\, pd.\quad pdof(sv|pd) \equiv pd,$
$svof(UU) \equiv UU,$
$pdof(UU) \equiv UU,$
$UU|UU \equiv UU.$

In T an instruction is a pair, whose head is the type of the instruction and whose tail is either a name m, an operator symbol o, or a natural number i. We assume that the set of operator symbols of T contains that of S. The instructions are:

Instruction (head)	(tail)	*Meaning*
JF	i	If head of pd is false, jump to label i, otherwise proceed to next instruction. In either case delete head of pd.
J	i	Jump to label i.
$FETCH$	m	Look up value of m in sv, and place it on top of pd.
$STORE$	m	Assign head of pd to m in sv, and delete head of pd.
$LABEL$	i	Serves only to label next instruction.
DO	o	Apply $MOP(o)$ – that is the binary function denoted by o – to the top two elements of pd, and replace them by the result.

We use '&' for the pairing operation and '*hd*', '*tl*' as the selectors for pairs. These, together with *null*, *NIL* and @ (*append*) are the conventional list-processing operations. Thus the pair whose members are JF and i is formally written $(JF\&i)$. We give the axioms for lists in Appendix 1.

By a program we mean a list of instructions. Unfortunately the existence of labels in T allows the meaning of such a program to be undetermined; for example, there might be two instructions $(LABEL\&6)$ in the list. To which one is $(JF\&6)$ to go? Although there are many alternatives we have chosen the following. A program p is well-formed if (i) the set L of numbers appearing in label statements forms an initial segment of the natural numbers; (ii) for each $n \in L$, $(LABEL\&n)$ occurs only once in p, and (iii) the set of numbers occurring in J and JF instructions is a subset of L, that is, there is no instruction which tries to jump to a non-existent label. These properties are guaranteed by the following definition of *iswft*:

$$iswft \equiv \lambda p.\ iswft1(count(p), p),$$
$$iswft1 \equiv \alpha w.\ \lambda n\, p.\ n=\emptyset \rightarrow TT,\ occurs1(n-1, p) \rightarrow w(n-1, p),\ FF,$$
$$count \equiv \alpha c.\ \lambda p.\ null(p) \rightarrow \emptyset,$$
$$(hd(hd(p))=J) \vee (hd(hd(p))=JF) \vee (hd(hd(p))=LABEL) \rightarrow$$
$$\max(tl(hd(p))+1, c(tl(p))), c(tl(p)),$$
$$occurs \equiv \alpha oc.\ \lambda n\, p.\ (count(p) \leqslant n) \rightarrow FF,$$
$$hd(hd(p))=LABEL \rightarrow tl(hd(p))=n \rightarrow TT,\ oc(n, tl(p)),\ oc(n, tl(p)),$$
$$occurs1 \equiv \alpha oc1.\ \lambda n\, p.\ count(p) \leqslant n \rightarrow FF,$$
$$hd(hd(p))=LABEL \rightarrow tl(hd(p))=n \rightarrow$$
$$\neg(occurs(n, tl(p))),\ oc1(n, tl(p)),\ oc1(n, tl(p)).$$

$count(p)$ computes the least natural number not appearing in a program p. $occurs(n, p)$ yield true if n occurs in p, false otherwise, and $occurs1(n, p)$ checks that a label occurs exactly once. $iswft1(n, p)$ checks that for every natural number m, $\emptyset \leqslant m < n$, the instruction $(LABEL\&m)$ occurs exactly once. Thus $iswft(p)$ is as described above.

We can now define MT.

$$MT \equiv \lambda p.\ MT1(p, p),$$
$$MT1 \equiv [\alpha f.\ [\lambda p\, q.\ null(q) \rightarrow [\lambda sp.\ svof(sp) | pdof(sp)],$$
$$hd(hd(q))=JF \rightarrow [\lambda sp.\ (true(hd(pdof(sp))) \rightarrow f(p, tl(q)),$$
$$f(p, find(p, tl(hd(q))))))(svof(sp) | tl(pdof(sp)))],$$
$$hd(hd(q))=J \rightarrow f(p, find(p, tl(hd(q)))),$$
$$hd(hd(q))=FETCH \rightarrow [\lambda sp.\ svof(sp) | (svof(sp)(tl(hd(q))) \&$$
$$pdof(sp))] \otimes f(p, tl(q)),$$
$$hd(hd(q))=STORE \rightarrow [\lambda sp.[\lambda m.\, m=tl(hd(q)) \rightarrow hd(pdof(sp)),$$
$$svof(sp)(m)] | tl(pdof(sp))] \otimes f(p, tl(q)),$$
$$hd(hd(q))=DO \rightarrow [\lambda sp.\ svof(sp) |$$
$$(MOP(tl(hd(q))(hd(tl(pdof(sp))), hd(pdof(sp)))$$
$$\&\, tl(tl(pdof(sp))))] \otimes f(p, tl(q)),$$
$$hd(hd(q))=LABEL \rightarrow f(p, tl(q)),$$
$$UU]],$$

$$find \equiv [\alpha f.\,[\lambda p\ n.\ null(p) \rightarrow UU,\ hd(hd(p)) = LABEL \rightarrow tl(hd(p)) = n \rightarrow tl(p), f(tl(p), n), f(tl(p), n)]].$$

The auxiliary function *find* has as arguments a program p and a label n and if $(LABEL\&n)$ occurs in p it yields that terminal sublist of p immediately following $(LABEL\&n)$, otherwise it yields undefined. One should note that the definition of $MT1$ could be parameterized by a variable in place of find thus allowing any computable method of 'finding' the appropriate instruction to jump to. This corresponds to choosing different notions of the semantics of a program. For example, if one allowed jumps to nonexistent labels '*find*' might simply compute the program NIL when such a jump was attempted. This amounts to choosing the convention that a jump to a nonexistent label terminates the program. Many such conventions can be mimicked by an appropriate find function. For the other instructions the definition of $MT1$ follows their informal description quite closely.

THE COMPILER

Strictly speaking we do not prove the correctness of a compiler in this paper. What we prove is the correctness of a compiling algorithm, which we call '*comp*'. That is, a compiler is a syntactic object written in some programming language; we have not started with such an object and shown that its meaning (semantics) is '*comp*', but rather we have assumed that '*comp*' is indeed the meaning of some suitably chosen compiler.

Expressions are compiled by

$$compe \equiv \alpha f.\ compefun(f),$$
$$compefun \equiv \lambda f\ e.$$
$$type(e) = _N \rightarrow (FETCH\&e)\&NIL,$$
$$type(e) = _E \rightarrow f(arg1of(e))\ @\ f(arg2of(e))\ @\ ((DO\&opof(e))\&NIL),$$
$$UU.$$

In order to define *comp* we use the following auxiliary functions:

$$shift = \alpha sh.\ \lambda n\ p.\ count(p) = \emptyset \rightarrow p,$$
$$(hd(hd(p)) = J) \vee (hd(hd(p)) = JF) \vee (hd(hd(p)) = LABEL) \rightarrow (hd(hd(p))\&(tl(hd(p)) + n))\&sh(n, tl(p)),$$
$$hd(p)\&sh(n, tl(p)),$$
$$mktcmpnd \equiv \lambda p\ q.\ p\ @\ shift(count(p), q)),$$
$$mktcond \equiv \lambda e.\lambda p\ q.\ compe(e)\ @\ ((JF\&count(p))\&NIL)\ @\ p\ @\ ((J\&(count(p)+1))\&NIL)\ @\ ((LABEL\&count(p))\&NIL)\ @\ shift(count(p)+2, q)\ @\ ((LABEL\&(count(p)+1))\&NIL,$$
$$mktwhile \equiv \lambda e.\lambda p.\ ((LABEL\&count(p))\&NIL)\ @\ compe(e)\ @$$

$((JF\&(count(p)+1))\&NIL)@$
$p@$
$((J\&count(p))\&NIL)@$
$((LABEL\&(count(p)+1))\&NIL).$

shift(n, p) adds n to the integer in each label and jump instruction occurring in p. Using *shift* in the definitions of the other combinators then guarantees that when applied to well-formed objects, *mktcmpnd*, *mktcond* and *mktwhile* generate well-formed target programs. *Comp* is defined as

$comp \equiv \alpha f.\ compfun(f),$
$compfun \equiv \lambda f\, p.$
$type(p) = _A \rightarrow compe(rhsof(p))@((STORE\&lhsof(p))\&NIL),$
$type(p) = _C \rightarrow mktcond(ifof(p))(f(thenof(p)), f(elseof(p))),$
$type(p) = _W \rightarrow mktwhile(testof(p))(f(bodyof(p))),$
$type(p) = _CM \rightarrow mktcmpnd(f(firstof(p)), f(secondof(p))),$
$UU.$

For well-formed source programs p the correctness of this compiler can be expressed as

$$(MS(p))(sv) \equiv svof((MT(comp(p)))(sv|NIL)).$$

This equation simply states that the result of executing a source program p on a state sv is the same as the state component of the store resulting from the execution of the compiled program $comp(p)$ on the store $sv|NIL$.

OUTLINE OF THE PROOF

Once we had defined the source and target languages and the compiling algorithm, and formulated the statement of compiler correctness, we proceeded to tackle the proof with the help of our proof-checking program LCF. The natural approach is to use structural induction on source programs. However, it was soon clear that the proof would be long and uninformative, and we became concerned not merely with carrying it out but also with giving it enough structure to make it intelligible. Observe that if we define

$SIMUL$: storefunctions$\rightarrow$statefunctions

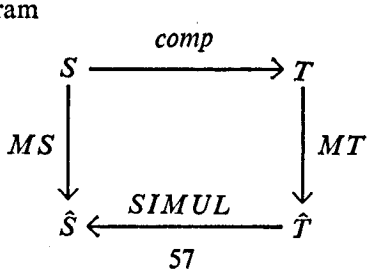
by

$$SIMUL \equiv \lambda g.\ \lambda sv.\ svof(g(sv|NIL))$$

then the compiler correctness is equivalently stated by

$$MS \equiv comp \otimes MT \otimes SIMUL \qquad \text{(G1)}$$

where it is understood that both sides are restricted to the domain of well-formed source programs. Now this equation is equivalent to the commutativity of the diagram



where $\hat{S}$ and $\hat{T}$ are the sets of statefunctions and storefunctions respectively. This diagram suggests an algebraic approach; in fact, by defining appropriate operations on the sets S, $\hat{S}$, T, $\hat{T}$ we will show that with respect to these operations the mappings in the diagram are actually homomorphisms. Then our result will follow as an instance of a fundamental theorem of universal algebra, as we shall explain below. This algebraic approach gives our proof the desired structure; it is an open question whether a similar approach will extend to more complex languages.

We now introduce those few concepts of universal algebra that we need. These may be found in Cohn (1965); we take the liberty of giving somewhat less formal definitions than his, since our needs are simple. For a clear exposition of some of the concepts of universal algebra written for computer scientists, see also Lloyd (1972).

An operator domain Ω is a set of operator symbols each with an associated integer $n \geqslant \emptyset$, called its *arity*, which is the number of arguments taken by the operation that the symbol will denote in an algebra.

An Ω-algebra **A** is a set A, called the carrier of **A**, together with an n-ary operation for each member of Ω with arity n. An Ω-algebra **B** is a subalgebra of **A** if its carrier B is a subset of A, its operations are the restrictions to B of **A**'s operations, and B is closed under these operations.

Given any set of terms X, the Ω-word algebra $\mathbf{W}_\Omega(X)$ has as carrier the smallest set of terms containing X and such that if $a \in \Omega$ and a has arity n, and if $w1, w2, \ldots, wn$ are in $\mathbf{W}_\Omega(X)$, then the term $a(w1, w2, \ldots, wn)$ is in $\mathbf{W}_\Omega(X)$. This term-building operation is the operation corresponding to a in $\mathbf{W}_\Omega(X)$. The members of X are called the generators of $\mathbf{W}_\Omega(X)$.

We are concerned only with algebras for a certain fixed Ω. We need not trouble to name the members of Ω; Ω is only a device for defining a 1–1 correspondence between the operations of different algebras, and this correspondence will be clear from the way we define our algebras.

The fundamental theorem that we need – see Cohn (1965), p. 120, Theorem 2.6 – states that if **W** is a word algebra, then any mapping from the generators of **W** into the carrier of an algebra **A** extends in only one way to a homomorphism from **W** to **A**. In our case the word algebra **W** is the algebra **S** of well-formed source programs, whose generators are the assignment statements and whose denumerably many operations are as follows:

(i) The binary operation *mkscmpnd*

(ii) For each well-formed source expression e, the binary operation *mkscond*(e)

(iii) For each such e, the unary operation *mkswhile*(e).

The second algebra **A** is the algebra $\hat{\mathbf{S}}$ of statefunctions, with operations as follows:

(i) The binary operation $SCMPND$

(ii) For each well formed source expression e, the binary operation $SCOND(e)$

(iii) For each such e, the unary operation $SWHILE(e)$.

Our main goal is (G1). We proceed to set up a tree of subgoals to attain this goal, and we will first state each of the subgoals in algebraic terms and then later list the formal statements of the subgoals as sentences of the logic *LCF*.

Our first level of subgoaling is justified by the fundamental theorem; to achieve (G1) it is sufficient to prove that

$MS: \mathbf{S} \to \hat{\mathbf{S}}$ is a homomorphism (G1.1)

$comp \otimes MT \otimes SIMUL: \mathbf{S} \to \hat{\mathbf{S}}$ is a homomorphism (G1.2)

and that

$MS \equiv comp \otimes MT \otimes SIMUL$,

when both sides are restricted to the generators of S. (G1.3)

Before going further, we must mention that in proving the formal statement of (G1) from the formal statements of (G1.1), (G1.2) and (G1.3) we do not rely on a formal statement in LCF of this fundamental theorem (though we believe that a restricted version of the theorem is indeed expressible and provable in LCF); rather we prove in LCF the relevant instance of that theorem. Thus we are using algebra as a guide to structuring our proof, not as a formal basis for the proof.

Now (G1.1) is a ready consequence of the definitions of MS, as the reader might suspect if he considers the operators of the algebras S and $\hat{\mathbf{S}}$. To achieve (G1.3), remember that the generators of S are the assignment statements, so we need a lemma which states that expressions – in particular, the right hand sides of assignments – compile correctly. This is expressed by:

$MT(compe(e)) \equiv \lambda sp.\ (svof(sp) | (MSE(e, sp)\ \&\ pdof(sp)))$,

whenever e is a well-formed source expression. (G1.3.1)

This says that the target program for an expression places the value of the expression on top of the stack and leaves the store otherwise unchanged.

In order to prove (G1.2), it is helpful to introduce some further algebras. First, the algebra **T** of well-formed target programs whose operations are as follows:

(i) The binary operation *mktcmpnd*

(ii) For each well formed source expression e, the binary operation $mktcond(e)$

(iii) For each such e, the unary operation $mktwhile(e)$.

We have defined *mktcmpnd*, *mktcond* and *mktwhile* in the previous section. Second, we need the algebra $\hat{\mathbf{T}}$ of storefunctions whose operations are as follows:

(i) The binary operation $TCMPND$

(ii) For each well-formed source expression e, the binary operation $TCOND(e)$

(iii) For each such e, the unary operation $TWHILE(e)$,

where we define

$TCMPND \equiv \otimes$,

$TCOND \equiv \lambda e.\lambda f g.\ MT(compe(e)) \otimes COND(GET, POP \otimes f, POP \otimes g)$,

$TWHILE \equiv \lambda e.\lambda f.\ \alpha g.MT(compe(e)) \otimes COND(GET, POP \otimes f \otimes g, POP)$,

which in turn require the definitions

$GET \equiv pdof \otimes hd \otimes true$,

$POP \equiv \lambda sp.\ (svof(sp) \mid tl(pdof(sp)))$.

Consider these definitions for a moment. POP is a storefunction which just deletes the top stack element. $GET(sp)$ yields the truth-value represented by the top stack element in the store sp. $MT(compe(e))$ is a storefunction which simply places the value of the expression e on top of the stack. $COND(GET, POP \otimes f, POP \otimes g)$ is a storefunction which examines the top stack element and then, after deleting this element, performs either the storefunction f or the storefunction g, according to the truth-value represented by it.

To achieve (G1.2) it is sufficient to prove

$comp: \mathbf{S} \to \mathbf{T}$ is a homomorphism (G1.2.1)

$MT: \mathbf{T} \to \hat{\mathbf{T}}$ is a homomorphism (G1.2.2)

$SIMUL: \hat{\mathbf{T}}' \to \hat{\mathbf{S}}$ is a homomorphism (G1.2.3)

where $\hat{\mathbf{T}}'$ is the subalgebra of $\hat{\mathbf{T}}$ induced by the homomorphism $comp \otimes MT: \mathbf{S} \to \hat{\mathbf{T}}$. (G1.2.1) is an immediate consequence of the definition of *comp*, provided that *comp* does indeed generate well-formed target programs from well-formed source programs. This is a consequence of the following two subgoals

comp takes the generators of **S** onto well-formed target programs (G1.2.1.1)

The operations of **T** preserve well-formedness of target programs (G1.2.1.2)

(G1.2.2) uses following general lemma about target programs, which we call the context-free lemma for MT, since it states that under certain conditions the execution of a sub-program is independent of its environment:

$MT1(p @ q' @ r, q' @ r) \equiv MT(q) \otimes MT1(p @ q' @ r, r)$, provided that q is well-formed, $q' = shift(n, q)$ for some n, and $p @ q' @ r$ is also well-formed. (G1.2.2.1)

(G1.2.3) depends critically on the property of storefunctions g in $\hat{\mathbf{T}}'$ that

$svof(g(sv \mid pd)) \equiv svof(g(sv \mid NIL))$,

that is, the left hand side is independent of pd. This of course is not true for an arbitrary storefunction. Stated algebraically,

For all g in $\hat{\mathbf{T}}'$, $g \otimes svof \equiv svof \otimes SIMUL(g)$. (G1.2.3.1)

This concludes our attempt to structure the proof of the correctness of the compiler using algebraic methods. We have given eleven subgoals, most of which have a simple algebraic interpretation and therefore contribute significantly to the understanding of the proof as a whole.

We now list the goals as they are represented formally by sentences of LCF.

We have abbreviated $comp \otimes MT \otimes SIMUL$ by H throughout.

(G1) (Compiler correctness)
$$iswfs(p) \equiv TT \vdash MS(p) \equiv SIMUL(MT(comp(p)))$$

(G1.1) (MS is a homomorphism)
$$iswfse(e) \equiv TT,\ iswfs(p) \equiv TT,\ iswfs(q) \equiv TT \vdash$$
$$MS(mkscmpnd(p, q)) \equiv SCMPND(MS(p), MS(q)),$$
$$MS(mkscond(e)(p, q)) \equiv SCOND(e)(MS(p), MS(q)),$$
$$MS(mkswhile(e)(p)) \equiv SWHILE(e)(MS(p))$$

(G1.2) (H is a homomorphism)
$$iswfse(e) \equiv TT,\ iswfs(p) \equiv TT,\ iswfs(q) \equiv TT \vdash$$
$$H(mkscmpnd(p, q)) \equiv SCMPND(H(p), H(q)),$$
$$H(mkscond(e)(p, q)) \equiv SCOND(e)(H(p), H(q),$$
$$H(mkswhile(e)(p)) \equiv SWHILE(e)(H(p))$$

(G1.3) (MS and H agree on the generators of S)
$$isname(n) \equiv TT,\ iswfse(e) \equiv TT \vdash$$
$$MS(mkassn(n, e)) \equiv H(mkassn(n, e))$$

(G1.2.1) ($comp$ is a homomorphism)
$$iswfse(e) \equiv TT,\ iswfs(p) \equiv TT,\ iswfs(q) \equiv TT \vdash$$
$$comp(mkscompnd(p, q)) \equiv mktcmpnd(comp(p), comp(q)),$$
$$comp(mkscond(e)(p, q)) \equiv mktcond(e)(comp(p), comp(q)),$$
$$comp(mkswhile(e)(p)) \equiv mktwhile(e)(comp(p))$$

(G1.2.2) (MT is a homomorphism)
$$iswfse(e) \equiv TT,\ iswft(p) \equiv TT,\ iswft(q) \equiv TT \vdash$$
$$MT(mktcmpnd(p, q)) \equiv TCMPND(MT(p), MT(q)),$$
$$MT(mktcond(e)(p, q)) \equiv TCOND(e)(MT(p), MT(q)),$$
$$MT(mktwhile(e)(p)) \equiv TWHILE(e)(MT(p))$$

(G1.2.3) ($SIMUL$ is a homomorphism)
$$iswfse(e) \equiv TT,\ iswfs(p) \equiv TT,\ iswfs(q) \equiv TT \vdash$$
$$SIMUL(TCMPND(MT(comp(p)), MT(comp(q)))) \equiv$$
$$SCMPND(SIMUL(MT(comp(p))), SIMUL(MT(comp(q)))),$$
$$SIMUL(TCOND(e)(MT(comp(p)), MT(comp(q)))) \equiv$$
$$SCOND(e)(SIMUL(MT(comp(p))), SIMUL(MT(comp(q)))),$$
$$SIMUL(TWHILE(e)(MT(comp(p)))) \equiv$$
$$SWHILE(e)(SIMUL(MT(comp(p))))$$

(G1.3.1) (well-formed expressions compile correctly)
$$iswfse(e) \equiv TT \vdash$$
$$MT(compe(e)) \equiv \lambda sp.(svof(sp) \mid (MSE(e) \,\&\, pdof(sp)))$$

(G1.2.1.1) (assignment statements compile into well-formed target programs)
$$isname(n) \equiv TT,\ iswfse(e) \equiv TT \vdash iswft(comp(mkassn(n, e))) \equiv TT$$

(G1.2.1.2) (the operations of T preserve well-formedness)
$$iswfse(e) \equiv TT,\ iswft(p) \equiv TT,\ iswft(q) \equiv TT \vdash iswft(mktcmpnd(p, q)) \equiv TT,\ iswft(mktcond(e)(p, q)) \equiv TT,\ iswft(mktwhile(e)(p)) \equiv TT$$

(G1.2.2.1) (Context-free lemma for MT)
$$iswft(q) \equiv TT,\ isnat(n) \equiv TT,\ q' \equiv shift(n, q),\ iswft(p\,@q'\,@r) \equiv TT \vdash MT1(p\,@q'\,@r, q'\,@r) \equiv MT(q) \otimes MT1(p\,@q'\,@r, r)$$

(G1.2.3.1) (For $g \in \hat{\mathbf{T}}'$, $svof(g(sv|pd))$ is independent of pd)
$$iswfs(p) \equiv TT \vdash MT(comp(p)) \otimes svof \equiv svof \otimes SIMUL(MT(comp(p)))$$

In Appendix 1 we give, in a form acceptable to the proof-checker, those axioms and definitions required for the proof which do not appear above. The only omission is the axioms for natural numbers. In Appendix 3 we give in full the machine printout of the proof of (G1.3.1), the McCarthy-Painter lemma, together with some notes as an aid to understanding it. This theorem has a somewhat independent status, as it states the correctness of that part of our compiler, *compe*, which compiles expressions. Our machine-checked proof therefore parallels the informal proof of essentially the same theorem given by McCarthy and Painter (1967). In Appendix 2 we give the sequence of commands typed by the user in generating the proof of the McCarthy-Painter lemma. We do not explain these commands; we give them merely to indicate that although the proof generated is quite long, the user does not have very much to type.

DISCUSSION OF THE PROOF

In this section we discuss the machine proof, and what we have learnt from carrying it out.

As is apparent from the details we have presented, the proof is lengthy but not profound. We have in fact not checked the whole proof on the machine – (G1.2.2), (G1.2.2.1) and parts of (G1.2.3) and (G1.2.1.2) remain to be done – so at present we cannot claim to have completely proved the correctness of a compiler on the machine. However, the aims were rather (i) to demonstrate that the proof is feasible, (ii) to explore the use of algebraic methods to give structure to the proof, and (iii) to obtain a case study which, in conjunction with those in our previous work (Milner 1972b, Weyhrauch and Milner 1972),

give us a feeling for how to enhance our implementation to diminish the human contribution to a proof. We have no significant doubt that the remainder of the proof can be done on the machine.

We have already discussed the value of algebraic methods, at least for this example of a simple compiler. It remains to be seen whether more complex compilers and semantics will fall naturally into the algebraic framework, or whether they may be coerced into the framework – and if so whether the advantages will justify the effort of coercion. But what is certain is that for machine-checked compiler proofs some way of structuring the proof is desirable.

Concerning feasibility; one measure of this is the number of proof steps required. The part of the proof that we have executed took about 600 steps, and we estimate that this is more than half of the total, although (G1.2.2) is not a trivial task. This measure does not take into account the considerable human effort in planning the proof, but – at least if the algebraic method can be applied in more complex cases – some part of this effort will be common to many compiler proofs.

This case study and the others referenced above have convinced us that the formal proofs were indeed feasible, but would not have been so without two features of our proof-checker, namely its subgoaling facility and its simplification mechanism. Usually the most creative contribution that the human makes is the decision as to what instance of the induction rule to apply (we do not discuss the induction rule, but many forms of structural induction are instances of it; for example the goal (G1), once the other goals have been proved, merely requires an induction on the structure of well-formed source programs). Once this decision is made, the remainder of the proof, if it requires no further inductions, follows a pattern which is sufficiently pronounced to give us hope for automation.

Acknowledgements

This research was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defence Under Contract SD–183 and in part by the National Aeronautics and Space Administration under Contract NSR 05–020–500.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency, the National Aeronautics and Space Administration, or the U.S. Government.

We would like to thank Malcolm Newey for his work on the implementation of LCF, Lockwood Morris for valuable discussions concerning algebraic methods, and Henri Ajenstat for carrying out most of the proof of (G1.2.1.2) on the machine.

REFERENCES

Burstall, R.M. & Landin, P.J. (1969) Programs and their proofs: an algebraic approach. *Machine Intelligence 4*, pp. 17–43 (eds Meltzer, B. & Michie, D.). Edinburgh: Edinburgh University Press.

Cohn, P.M. (1965) *Universal Algebra*. New York: Harper and Row.

Diffie, W. (1972) Mechanical verification of a compiler correctness proof. Forthcoming *A.I. Memo*, Stanford University.

Dijkstra, E.W. (1968) Goto statement considered harmful. Letter to Editor, *Comm. Ass. Comp. Mach.*, **11**, 147–8.

Kaplan, D.M. (1967) Correctness of a compiler for ALGOL-like programs. *A. I. Memo 48*. Stanford University.

Lloyd, C. (1972) Some concepts of universal algebra and their application to computer science. CSWP–1, Computing Centre, University of Essex.

London, R.L. (1971) Correctness of two compilers for a LISP subset. *A. I. Memo 151*. Stanford University.

London, R.L. (1972) Correctness of a compiler for a LISP subset. *Proc. Conf. on Proving Assertions about Programs*. New Mexico State University.

McCarthy, J. (1963) Towards a mathematical science of computation. Information processing; *Proc. IFIP Congress 62*, pp. 21–8 (ed. Popplewell, C.M.). Amsterdam: North Holland.

McCarthy, J. & Painter, J.A. (1967) Correctness of a compiler for arithmetic expressions. *Proceedings of a Symposium in Applied Mathematics*, **19**, *Mathematical Aspects of Computer Science*, pp. 33–41 (ed. Schwartz, J.T.). Providence, Rhode Island: American Mathematical Society.

Milner, R. (1972a) Logic for computable functions; description of a machine implementation. *A.I. Memo 169*. Stanford University.

Milner, R. (1972b) Implementation and applications of Scott's logic for computable functions. *Proc. Conf. on Proving Assertions about Programs*. New Mexico State University.

Morris, L. (1972) Forthcoming Ph.D. Thesis, Stanford University.

Painter, J.A. (1967) Semantic correctness of a compiler for an ALGOL-like language. *A.I. Memo 44*. Stanford University; also Ph.D. Thesis. Stanford University.

Weyhrauch, R.W. and Milner, R. (1972) Program semantics and correctness in a mechanized logic. *Proc. USA-Japan Computer Conference*, Tokyo.

APPENDIX 1: Some of the axioms

```
AXIOM LOGAX:
 !^ ≡ [λP Q .¬(¬(P)v¬(Q))], ¬ ≡ [λP .(P=FF,TT)], [λP Q .PvQ] ≡ [λP Q .QvP],
 [λP .PvTT] ≡ [λP .TT], [λP .PvFF] ≡ [λP .P];

AXIOM EQUAX:
 ∂ ≡ [λx .x=x], ∀x y .(x=y)!! x ≡ y,
 ∀x y .(∂ x)^(∂ y) ≡ (x=y)v¬(x=y);

AXIOM LISAX:
 ∀x y. hd(x&y) ≡ x,    ∀x y. tl(x&y) ≡ y,    ∀x y. null(x&y) ≡ FF,
 null NIL ≡ TT,    ∀x. null x !! x ≡ NIL,    null UU ≡ UU,
 hd UU ≡ UU,    tl UU ≡ UU,    ∀x. ¬(null x) !! (hd x)&(tl x) ≡ x;

AXIOM LISFN:
 !∂ ≡ ∂f. λl m. null l → m. (hd l)&f(tl l,m);

AXIOM SYNAXS:
 ∀e e1 e2. type (mkse e e1 e2) ≡ _E,        _N=_N ≡ TT,
 ∀e e1 e2. opof (mkse e e1 e2) ≡ e ,        _E=_N ≡ FF,
 ∀e e1 e2. arg1of(mkse e e1 e2) ≡ e1,       _N=_E ≡ FF,
 ∀e e1 e2. arg2of(mkse e e1 e2) ≡ e2,       _E=_E ≡ TT,

 ∀n e. type(mksasgn n e) ≡ _A,              _A=_A ≡ TT,
 ∀n e. lhsof(mksasgn n e) ≡ n ,             _C=_A ≡ FF,
 ∀n e. rhsof(mksasgn n e) ≡ e ,             _W=_A ≡ FF,
                                            _CM=_A ≡ FF,
 ∀e p1 p2. type (mkscond e p1 p2) ≡ _C,     _A =_C  ≡ FF,
 ∀e p1 p2. ifof (mkscond e p1 p2) ≡ e,      _C =_C  ≡ TT,
 ∀e p1 p2. thenof(mkscond e p1 p2) ≡ p1,    _W =_C  ≡ FF,
 ∀e p1 p2. elseof(mkscond e p1 p2) ≡ p2,    _CM=_C  ≡ FF,
                                            _A =_W  ≡ FF,
 ∀e p. type(mkswhile e p) ≡ _W,             _C =_W  ≡ FF,
 ∀e p. testof (mkswhile e p) ≡ e ,          _W =_W  ≡ TT,
 ∀e p. bodyof (mkswhile e p) ≡ p,           _CM=_W  ≡ FF,
                                            _A =_CM ≡ FF,
 ∀p1 p2. type(mkscmpnd p1 p2) ≡ _CM,        _C =_CM ≡ FF,
 ∀p1 p2. firstof(mkscmpnd p1 p2) ≡ p1,      _W =_CM ≡ FF,
 ∀p1 p2.secondof(mkscmpnd p1 p2) ≡ p2,      _CM=_CM ≡ TT,

AXIOM SYNAXT:
 JF    =JF     ≡ TT,    JF    =J     ≡ FF,    JF    =FETCH ≡ FF,
 J     =JF     ≡ FF,    J     =J     ≡ TT,    J     =FETCH ≡ FF,
 FETCH=JF      ≡ FF,    FETCH=J      ≡ FF,    FETCH=FETCH ≡ TT,
 STORE=JF      ≡ FF,    STORE=J      ≡ FF,    STORE=FETCH ≡ FF,
 DO    =JF     ≡ FF,    DO    =J     ≡ FF,    DO    =FETCH ≡ FF,
 LABEL=JF      ≡ FF,    LABEL=J      ≡ FF,    LABEL=FETCH ≡ FF,
 JF    =STORE  ≡ FF,    JF    =DO    ≡ FF,    JF    =LABEL ≡ FF,
 J     =STORE  ≡ FF,    J     =DO    ≡ FF,    J     =LABEL ≡ FF,
 FETCH=STORE   ≡ FF,    FETCH=DO     ≡ FF,    FETCH=LABEL ≡ FF,
 STORE=STORE   ≡ TT,    STORE=DO     ≡ FF,    STORE=LABEL ≡ FF,
 DO    =STORE  ≡ FF,    DO    =DO    ≡ TT,    DO    =LABEL ≡ FF,
 LABEL=STORE   ≡ FF,    LABEL=DO     ≡ FF,    LABEL=LABEL ≡ TT,
```

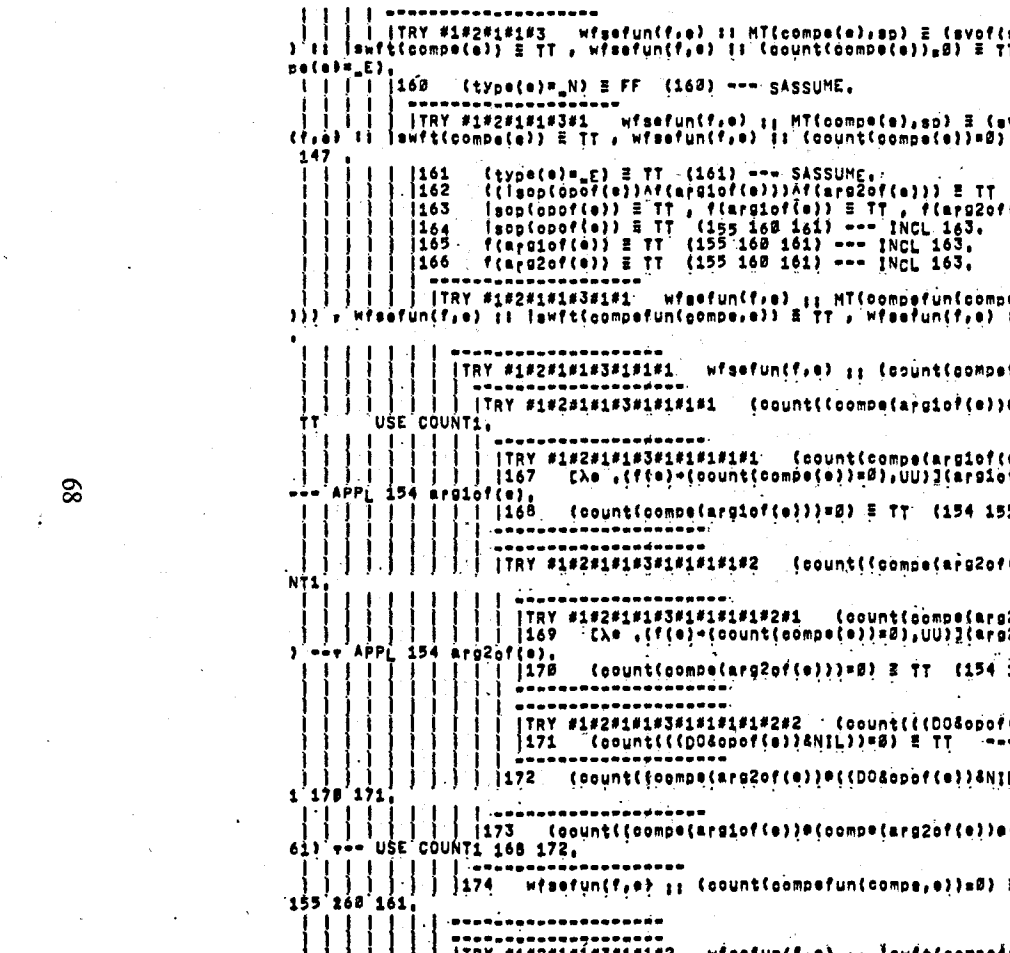
APPENDIX 2: command sequence for McCarthy-Painter lemma

```
GOAL ∀e sp,¬swfse e⊃¬MT(compe e,sp)∧svof(sp)∧¬(MSE(e,svof sp))∧pdof(sp)),
     ∀e,¬swfse e⊃¬swft(compe e)≡TT,
     ∀e,¬swfse e⊃¬(count(compe e)=0)≡TT;

TRY 1 INDUCT 56;
 TRY 1 SIMPL;
 LABEL INDHYP;
 TRY 2 ABSTR;
  TRY 1 CASES wfse(fun(f,e));
  LABEL TT;
   TRY 1 CASES type e=N;
    TRY 1 SIMPL BY ,FMT1,,FMSE,,FCOMPE,,FISWFT1,,FCOUNT;
    TRY 2;SS-,TT;SIMPL,TT;QED;
    TRY 3 CASES type e=E;
     TRY 1 SUBST ,FCOMPE;
     SS-,TT;SIMPL,TT;USE BOTH3 -;SS+,TT;
     INCL-,1;SS+-;INCL--,2;SS+-;INCL---,3;SS+-;
      TRY 1 CONJ;
       TRY 1 SIMPL;
        TRY 1 USE COUNT1;
         TRY 1;
         APPL ,INDHYP+2,arg1of e;
         LABEL CARG1;
         SIMPL-;QED;
         TRY 2 USE COUNT1;
          TRY 1;
          APPL ,INDHYP+2,arg2of e;
          LABEL CARG2;
          SIMPL-;QED;
          LABEL CDO;
          TRY 2 SIMPL BY ,FCOUNT;
         TRY 2 SIMPL BY ,FISWFT1,--;
         SIMPL ¬swft(compe(arg1of e)) BY ,FISWFT1,,CARG1;
         SIMPL ¬swft(compe(arg2of e)) BY ,FISWFT1,,CARG2;
         USE THM3 -,,CDO;SS+-;
         USE COUNT1 ,CARG2,,CDO;
         USE THM3 ----,-;SS+-;
       APPL ,INDHYP,arg1of e;SIMPL-;SS+-;
       APPL ,INDHYP,arg2of e;SIMPL-;SS+-;
       TRY 3 SUBST ,FMSE OCC 1;
       SS-100;
       TRY 1 SIMPL;
       SS+100;
       TRY 1 SIMPL BY ,FMT1;
   TRY 2;SS-,TT;SIMPL,TT;QED;
   TRY 3;SS-,TT;SIMPL,TT;QED;
   TRY 2 SIMPL;
   TRY 3 SIMPL;
```

APPENDIX 3: proof of the McCarthy-Painter lemma

```
 ---+---------------
 |TRY #1   Ve sp , |swfse(e) ;; MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , Ve , |swfse(e) ;; |swf~
t(compe(e)) ≡ TT , Ve , |swfse(e) ;; (count(compe(e))=0) ≡ TT     INDUCT 56 ,
  | -+-----------------
  | |TRY #1#1   Ve sp , UU(e) ;; MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , Ve , UU(e) ;; |swft(co~
mpe(e)) ≡ TT , Ve , UU(e) ;; (count(compe(e))=0) ≡ TT     SIMPL ,
  | |151   Ve sp , UU(e) ;; MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , Ve , UU(e) ;; |swft(compe(e~
)) ≡ TT , Ve , UU(e) ;; (count(compe(e))=0) ≡ TT   --- SIMPL.
  | -+-----------------
  | -+-----------------
  | |TRY #1#2   Ve sp , wfsefun(f,e) ;; MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , Ve , wfsefun(f,~
e) ;; |swft(compe(e)) ≡ TT , Ve , wfsefun(f,e) ;; (count(compe(e))=0) ≡ TT : ASSUME   Ve sp , f(e) ;; MT(compe(e~
),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , Ve , f(e) ;; |swft(compe(e)) ≡ TT , Ve , f(e) ;; (count(compe(e)~
)=0) ≡ TT     ABSTR,
  | |152   Ve sp , f(e) ;; MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp)))  (152) --- ASSUME,
  | |153   Ve , f(e) ;; |swft(compe(e)) ≡ TT  (153) --- ASSUME,
  | |154   Ve , f(e) ;; (count(compe(e))=0) ≡ TT  (154) --- ASSUME,
  | | ------------------
  | | |TRY #1#2#1   wfsefun(f,e) ;; MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun(f,e) ;; |sw~
ft(compe(e)) ≡ TT , wfsefun(f,e) ;; (count(compe(e))=0) ≡ TT     CASES wfsefun(f,e),
  | | | -----------------
  | | | |TRY #1#2#1#1   wfsefun(f,e) ;; MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun(f,e) ;;~
 |swft(compe(e)) ≡ TT , wfsefun(f,e) ;; (count(compe(e))=0) ≡ TT : SASSUME   wfsefun(f,e) ≡ TT     CASES (type(e~
)=_N),
  | | | |155   wfsefun(f,e) ≡ TT  (155) --- SASSUME,
  | | | | -----------------
  | | | | |TRY #1#2#1#1#1   wfsefun(f,e) ;; MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun(f,e~
) ;; |swft(compe(e)) ≡ TT , wfsefun(f,e) ;; (count(compe(e))=0) ≡ TT : SASSUME   (type(e)=_N) ≡ TT     SIMPL BY ~
140 138 147 146 142,
  | | | | |156   (type(e)=_N) ≡ TT  (156) --- SASSUME,
  | | | | |157   wfsefun(f,e) ;; MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun(f,e) ;; |swft(~
compe(e)) ≡ TT , wfsefun(f,e) ;; (count(compe(e))=0) ≡ TT  (155 156) --- SIMPL BY 5 9 10 11 15 63 71 77 83 101 1~
08 111 112 116 127 130 138 140 142 146 147 150 155 156,
  | | | | -----------------
  | | | | -----------------
  | | | | |TRY #1#2#1#1#2   wfsefun(f,e) ;; MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun(f,e~
) ;; |swft(compe(e)) ≡ TT , wfsefun(f,e) ;; (count(compe(e))=0) ≡ TT : SASSUME   (type(e)=_N) ≡ UU     ,
  | | | | |158   (type(e)=_N) ≡ UU  (158) --- SASSUME,
  | | | | |159   UU ≡ TT  (155 158) --- SIMPL 155 BY 57 158,
  | | | | -----------------
```


|TRY #1#2#1#1#3 wfsefun(f,e) ;; MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun(f,e) ;; swft(compe(e)) ≡ TT , wfsefun(f,e) ;; (count(compe(e))=0) ≡ TT ! SASSUME (type(e)=_N) ≡ FF CASES (type(e)=_E),
|160 (type(e)=_N) ≡ FF (160) --- SASSUME.
|TRY #1#2#1#1#3#1 wfsefun(f,e) ;; MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun(f,e) ;; swft(compe(e)) ≡ TT , wfsefun(f,e) ;; (count(compe(e))=0) ≡ TT ! SASSUME (type(e)=_E) ≡ TT SUBST 147 ,
|161 (type(e)=_E) ≡ TT (161) --- SASSUME.
|162 (((sop(opof(e)))Af(arg1of(e)))Af(arg2of(e))) ≡ TT (155 160 161) --- SIMPL 155 BY 57 160 161,
|163 |sop(opof(e)) ≡ TT , f(arg1of(e)) ≡ TT , f(arg2of(e)) ≡ TT (155 160 161) --- USE BOTH3 162,
|164 |sop(opof(e)) ≡ TT (155 160 161) --- INCL 163.
|165 f(arg1of(e)) ≡ TT (155 160 161) --- INCL 163.
|166 f(arg2of(e)) ≡ TT (155 160 161) --- INCL 163.
|TRY #1#2#1#1#3#1#1 wfsefun(f,e) ;; MT(compefun(compe,e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun(f,e) ;; |swft(compefun(compe,e)) ≡ TT , wfsefun(f,e) ;; (count(compefun(compe,e))=0) ≡ TT CONJ ,
|TRY #1#2#1#1#3#1#1#1 wfsefun(f,e) ;; (count(compefun(compe,e))=0) ≡ TT SIMPL .
|TRY #1#2#1#1#3#1#1#1#1 (count((compe(arg1of(e))@(compe(arg2of(e))@((DO&opof(e))&NIL))))=0) ≡ TT USE COUNT1,
|TRY #1#2#1#1#3#1#1#1#1#1 (count(compe(arg1of(e)))=0) ≡ TT ,
|167 [λe ,(f(e)→(count(compe(e))=0),UU)](arg1of(e)) ≡ [λe ,(f(e)→TT,UU)](arg1of(e)) (154) --- APPL 154 arg1of(e),
|168 (count(compe(arg1of(e)))=0) ≡ TT (154 155 160 161) --- SIMPL 167 BY 165.
|TRY #1#2#1#1#3#1#1#1#2 (count((compe(arg2of(e))@((DO&opof(e))&NIL)))=0) ≡ TT USE COUNT1,
|TRY #1#2#1#1#3#1#1#1#1#2#1 (count(compe(arg2of(e)))=0) ≡ TT ,
|169 [λe ,(f(e)→(count(compe(e))=0),UU)](arg2of(e)) ≡ [λe ,(f(e)→TT,UU)](arg2of(e)) (154) --- APPL 154 arg2of(e),
|170 (count(compe(arg2of(e)))=0) ≡ TT (154 155 160 161) --- SIMPL 169 BY 166.
|TRY #1#2#1#1#3#1#1#1#1#2#2 (count(((DO&opof(e))&NIL))=0) ≡ TT SIMPL BY 142,
|171 (count(((DO&opof(e))&NIL))=0) ≡ TT --- SIMPL BY 5 9 10 11 15 73 79 103 142 150,
|172 (count((compe(arg2of(e))@((DO&opof(e))&NIL)))=0) ≡ TT (154 155 160 161) --- USE COUNT1 170 171,
|173 (count((compe(arg1of(e))@(compe(arg2of(e))@((DO&opof(e))&NIL))))=0) ≡ TT (154 155 160 161) --- USE COUNT1 168 172,
|174 wfsefun(f,e) ;; (count(compefun(compe,e))=0) ≡ TT (154 155 160 161) --- SIMPL 173 BY 130 155 160 161,
|TRY #1#2#1#1#3#1#1#2 wfsefun(f,e) ;; |swft(compefun(compe,e)) ≡ TT SIMPL BY 146 173,


|175 wfsefun(f,e) ;; |swft(compefun(compe,e)) ≡ TT (154 155 160 161) --- SIMPL BY 127 130 146 155 160 161 173,
|176 |swft(compe(arg1of(e))) ≡ TT (154 155 160 161) --- SIMPL |swft(compe(arg1of(e))) BY 127 146 168,
|177 |swft(compe(arg2of(e))) ≡ TT (154 155 160 161) --- SIMPL |swft(compe(arg2of(e))) BY 127 146 178,
|178 MT((compe(arg2of(e))#((DO&opof(e))&NIL))) ≡ (MT(compe(arg2of(e)))•MT(((DO&opof(e))&NIL))) (154 155 160 161) --- USE THM3 177 171,
|179 (count((compe(arg2of(e))#((DO&opof(e))&NIL)))=0) ≡ TT (154 155 160 161) --- USE COUNT1 170 171,
|180 MT((compe(arg1of(e))#(compe(arg2of(e))#((DO&opof(e))&NIL)))) ≡ (MT(compe(arg1of(e)))•MT((compe(arg2of(e))#((DO&opof(e))&NIL)))) (154 155 160 161) --- USE THM3 176 179,
|181 [λe sp .(f(e)•MT(compe(e),sp),UU)](arg1of(e)) ≡ [λe sp .(f(e)•(svof(sp)|(MSE(e,svof(sp))&pdof(sp))),UU)](arg1of(e)) (152) --- APPL 152 arg1of(e),
|182 Vsp . MT1(compe(arg1of(e)),compe(arg1of(e)),sp) ≡ (svof(sp)|(MSE(arg1of(e),svof(sp))&pdof(sp))) (152 155 160 161) --- SIMPL 181 BY 108 165,
|183 [λe sp .(f(e)•MT(compe(e),sp),UU)](arg2of(e)) ≡ [λe sp .(f(e)•(svof(sp)|(MSE(e,svof(sp))&pdof(sp))),UU)](arg2of(e)) (152) --- APPL 152 arg2of(e),
|184 Vsp . MT1(compe(arg2of(e)),compe(arg2of(e)),sp) ≡ (svof(sp)|(MSE(arg2of(e),svof(sp))&pdof(sp))) (152 155 160 161) --- SIMPL 183 BY 108 166,
|TRY #1#2#1#1#3#1#1#3 wfsefun(f,e) ;; MT(compefun(compe,e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) SUBST 138 OCC 1,
|TRY #1#2#1#1#3#1#1#3#1 wfsefun(f,e) ;; MT(compefun(compe,e),sp) ≡ (svof(sp)|([λe s .((type(e)=_N)•s(e),((type(e)=_E)•MOP(opof(e),MSE(arg1of(e),e),MSE(arg2of(e),s)),UU))](e,svof(sp))&pdof(sp))) SIMPL ,
|TRY #1#2#1#1#3#1#1#3#1#1 MT(((DO&opof(e))&NIL),MT(compe(arg2of(e)),MT(compe(arg1of(e)),sp)) ≡ (svof(sp)|(MOP(opof(e),MSE(arg1of(e),svof(sp)),MSE(arg2of(e),svof(sp)))&pdof(sp))) SIMPL BY 140,
|185 MT(((DO&opof(e))&NIL),MT(compe(arg2of(e)),MT(compe(arg1of(e)),sp))) ≡ (svof(sp)|(MOP(opof(e),MSE(arg1of(e),svof(sp)),MSE(arg2of(e),svof(sp)))&pdof(sp))) (152 155 160 161) --- SIMPL BY 9 10 11 15 63 73 79 85 91 97 108 111 112 116 140 182 184,
|186 wfsefun(f,e) ;; MT(compefun(compe,e),sp) ≡ (svof(sp)|([λe s .((type(e)=_N)•s(e),((type(e)=_E)•MOP(opof(e),MSE(arg1of(e),s),MSE(arg2of(e),s)),UU))](e,svof(sp))&pdof(sp))) (152 154 155 160 161) --- SIMPL 185 BY 63 130 155 160 161 178 180,
|187 wfsefun(f,e) ;; MT(compefun(compe,e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) (152 154 155 160 161) --- SUBST 186 USING 138 OCC 1,
|188 wfsefun(f,e) ;; MT(compefun(compe,e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun(f,e) ;; |swft(compefun(compe,e)) ≡ TT , wfsefun(f,e) ;; (count(compefun(compe,e))=0) ≡ TT (152 154 155 160 161) --- CON, 174 175 187,
|189 wfsefun(f,e) ;; MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun(f,e) ;; |swft(compe(e)) ≡ TT , wfsefun(f,e) ;; (count(compe(e))=0) ≡ TT (152 154 155 160 161) --- SUBST 188 USING 147 ,
|TRY #1#2#1#1#3#2 wfsefun(f,e) ;; MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun(f,e) ;; |swft(compe(e)) ≡ TT , wfsefun(f,e) ;; (count(compe(e))=0) ≡ TT ; SASSUME (type(e)=_E) ≡ UU ;
|190 (type(e)=_E) ≡ UU (190) --- SASSUME,
|191 UU ≡ TT (155 160 190) --- SIMPL 155 BY 57 160 190,

```
| | | | | ---------------------
| | | | | |TRY #1#2#1#1#3#3   wfsefun(f,e) :: MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun
(f,e) :: |swft(compe(e)) ≡ TT , wfsefun(f,e) :: (count(compe(e))=0) ≡ TT ! SASSUME   (type(e)=_E) ≡ FF    ,
| | | | | |192   (type(e)=_E) ≡ FF  (192) --- SASSUME.
| | | | | |193   UU ≡ TT  (155 160 192) --- SIMPL 155 BY 57 160 192.
| | | | | ---------------------
| | | | |194   wfsefun(f,e) :: MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun(f,e) :: |swft(
compe(e)) ≡ TT , wfsefun(f,e) :: (count(compe(e))=0) ≡ TT  (152 154 155 160) --- CASES (type(e)=_E) 189 191 193.
| | | | ---------------------
| | | |195   wfsefun(f,e) :: MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun(f,e) :: |swft(co
mpe(e)) ≡ TT , wfsefun(f,e) :: (count(compe(e))=0) ≡ TT  (152 154 155) --- CASES (type(e)=_N) 157 159 194.
| | | ---------------------
| | | ---------------------
| | | |TRY #1#2#1#2   wfsefun(f,e) :: MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun(f,e) ::
|swft(compe(e)) ≡ TT , wfsefun(f,e) :: (count(compe(e))=0) ≡ TT ! SASSUME   wfsefun(f,e) ≡ UU     SIMPL ,
| | | |196   wfsefun(f,e) ≡ UU  (196) --- SASSUME.
| | | |197   wfsefun(f,e) :: MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun(f,e) :: |swft(co
mpe(e)) ≡ TT , wfsefun(f,e) :: (count(compe(e))=0) ≡ TT  (196) --- SIMPL BY 196.
| | | ---------------------
| | | ---------------------
| | | |TRY #1#2#1#3   wfsefun(f,e) :: MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun(f,e) ::
|swft(compe(e)) ≡ TT , wfsefun(f,e) :: (count(compe(e))=0) ≡ TT ! SASSUME   wfsefun(f,e) ≡ FF     SIMPL ,
| | | |198   wfsefun(f,e) ≡ FF  (198) --- SASSUME.
| | | |199   wfsefun(f,e) :: MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun(f,e) :: |swft(co
mpe(e)) ≡ TT , wfsefun(f,e) :: (count(compe(e))=0) ≡ TT  (198) --- SIMPL BY 198.
| | | ---------------------
| | |200   wfsefun(f,e) :: MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun(f,e) :: |swft(comp
e(e)) ≡ TT , wfsefun(f,e) :: (count(compe(e))=0) ≡ TT  (152 154) --- CASES wfsefun(f,e) 195 197 199.
| | ---------------------
| |201   Ve sp , wfsefun(f,e) :: MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , Ve , wfsefun(f,e) ::
|swft(compe(e)) ≡ TT , Ve , wfsefun(f,e) :: (count(compe(e))=0) ≡ TT  (152 154) --- ABSTR 200.
| ---------------------
|202   Ve sp , |swfse(e) :: MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , Ve , |swfse(e) :: |swft(c
ompe(e)) ≡ TT , Ve , |swfse(e) :: (count(compe(e))=0) ≡ TT   --- INDUCT 151 201.
---------------------
```

The proof presented here actually proves more than just the McCarthy-Painter lemma. It also shows that the *count* of a compiled expression is 0 and thus *compe*(*e*) of a well-formed source expression *e* is a well-formed target program. These three facts are proved simultaneously by a structural induction on well-formed source programs. The main goal is stated at $TRY\#1$, and the base case of the induction at $TRY\#1\#1$. In the machine print-out of a proof the successive subgoals are nested in their natural way by using boxes. The main step of the induction is stated at $TRY\#1\#2$. The '::' in sentences of the form $A::B\equiv C$ can be interpreted as meaning if $A\equiv TT$ then $B\equiv C$. Actually it is an abbreviaton for $(A\rightarrow B, UU)\equiv(A\rightarrow C, UU)$. The induction hypotheses are stated at steps 152, 153, 154. Assuming *wfsefun*(*f*, *e*) is true, the proof then proceeds in a straightforward way by cases on *type*(*e*); first if *e* is a name and then if *e* is a compound expression. The numbered steps of the printout can be read as a formal proof of the lemma. Each step is followed by its list of dependencies in parentheses, for example, line 174 depends on steps 154, 155, 160, 161.

COMPUTATIONAL LOGIC

4

Building-in Equational Theories

G. D. Plotkin
Department of Machine Intelligence
University of Edinburgh

INTRODUCTION

If let loose, resolution theorem-provers can waste time in many ways. They can continually rearrange the multiplication brackets of an associative multiplication operation or replace terms t by ones like $f(f(f(t, e), e), e)$ where f is a multiplication function and e is its identity. Generally they continually discover and misapply trivial lemmas. Global heuristics using term complexity do not help much and *ad hoc* devices seem suspicious.

On the other hand, one would like to evaluate terms when possible, for example we would want to replace $5+4$ by 9. More generally one would like to have liberty to simplify, to factorise and to rearrange terms. The obvious way to deal with an associative multiplication would be to imitate people, and just drop the multiplication brackets. However used or abused the basic facts involved in such manipulations form an equational theory, T, that is, a theory all of whose sentences are universal closures of equations.

Under certain conditions, we will be able to build the equational theory into the rules of inference. The resulting method will be resolution-like, the difference being that concepts are defined using provable equality between terms rather than literal identity. Therefore the set of clauses expressing the theory will not be among the input clauses, so no time will be wasted in the misapplication of trivial lemmas, since the rules will not waste time in this way.

Such devices as evaluation, factorisation and simplification consist of the application of a recursive function, N, from terms to terms with the property:

$$\vdash_T t = N(t).$$

For example, given an associative function f in t, N might rearrange t with all the f's as far to the right as possible; for an arithmetic expression t, $N(t)$ might be the value.

The simplification function can be extended to one from literals and clauses to, respectively, literals and clauses by:

$$N((\pm)P(t_1, \ldots, t_n)) = (\pm)P(N(t_1), \ldots, N(t_n))$$
$$N(C) = \{N(L) | L \in C\}$$

and similarly to other expressions.

We shall look for a complete set of rules $r_1, \ldots, r_m$ such that $r_1 \circ N, \ldots, r_m \circ N$ is also a complete set. (If r is a rule, $r \circ N$ is the rule which outputs $N(C)$ iff r outputs, with the same inputs, C). So the required facilities can be used, but incorporated in a theoretical framework.

While one could just drop the brackets of an associative operation, and make appropriate changes, such a procedure could not be systematic. Instead note that the set of terms in right associative normal form is in a natural 1–1 correspondence with the set of 'terms' in which the multiplication brackets and symbols, and the commas are dropped. So the bracketless terms will be considered simply as an abbreviation for the normal form. This situation seems to be quite common. Another example: if T is the theory of Boolean algebra, terms can be represented as sets of sets of literals, being in 1–1 correspondence, according to convention, with terms in either a suitable disjunctive or conjunctive normal form.

Normal forms will be considered to be given by a simplification function, N, which in addition satisfies the postulate:

If $\vdash_T t = u$ then $N(t)$ is identical to $N(u)$.

A theory T has a normal form iff it is decidable.

This does not cover all uses of the phrase. For example in the untyped λ-calculus, not all terms have one and N is only partial recursive. And of course things other than terms can have normal forms. As defined, normal form functions give less redundancy than any other simplification function.

The special characteristic of resolution-like methods is the unification algorithm. In our case, unification is defined relative to the theory T. A substitution, σ, T-unifies two terms t and u iff $\vdash_T t\sigma = u\sigma$. The exact generalisation of the notion of most general unifier will be given later; in general rather than a single most general unifier, there will be an infinite set of maximally general ones.

Define an N-application and composition operator, $*$, by:

$$t * \sigma = N(t\sigma)$$
$$\sigma * \mu = N(\sigma\mu)$$

It can be shown that $*$ possesses properties analogous to ordinary application and composition independently of the particular T and N chosen (see Robinson 1965).

For each equational theory one must invent a special unification algorithm with these equations built in. If we know a normalising function for the theory we are likely to get a more efficient algorithm. The purpose of this paper is not to display such algorithms, although we will give a couple of examples to fix ideas. Our purpose is to show how, given any such unification algorithm satisfying certain conditions, the usual resolution techniques can

be carried through using the more sophisticated algorithm, and without losing completeness.

Now as an example, consider the case where T is the theory of an associative function, f, whose language may include other function symbols, and $N(t)$ is the right associative form of t.

How can we unify $f(x, y)$ with $f(a, f(b, c))$ which is equal to $f(f(a, b), c)$? There are two T-unifications: $x=a$, $y=f(b, c)$ and $x=f(a, b)$, $y=c$. The first is straightforward. The second can be obtained in two steps:

(1) Put $x=f(a, x')$ where x' is a new variable.

(2) Unify $f(f(a, x'), y)$ with $f(a, f(b, c))$.

In step (2) we should normalise both expressions before unifying, so we actually unify $f(a, f(x', y))$ with $f(a, f(b, c))$. Thus $x'=b$ and $y=c$, giving us the second unification.

In practice it is simpler to reuse the variable x instead of introducing a new variable x'.

The unification algorithm is non-deterministic; a substitution is in the maximally general set of T-unifiers it defines iff the substitution can be output by the algorithm. That this set has the correct properties, and the correctness of the other algorithm given in this introduction is proved in Plotkin (1972). Terms are, temporarily, regarded as strings of function symbols, brackets, commas and variables. The disagreement pair of two terms t and u is the rightmost pair of distinct terms t' and u', such that t and u have the forms $Xt'Y$ and $Xu'Z$ respectively, for suitable strings X, Y and Z.

(1) Set σ equal to ε.

(2) If t is identical to u, stop with success and output σ.

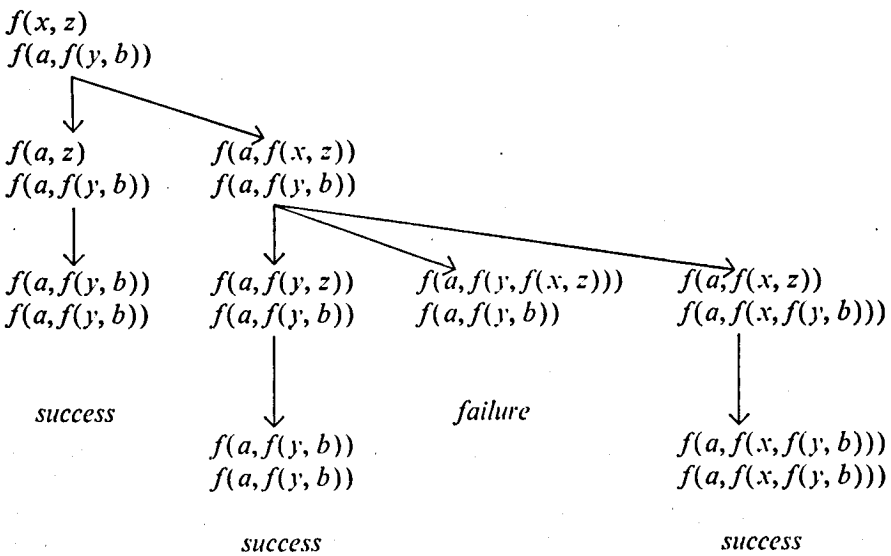


Figure 1

(3) Find the disagreement pair, t', u' of t and u.
(4) If t' and u' begin with different function symbols, stop with failure.
(5) If t' and u' are both variables, change t, u and σ to $t * \lambda$, $u * \lambda$ and $\sigma * \lambda$ where λ is either $\{u'/t'\}$, $\{f(u', t')/t'\}$ or $\{f(t', u')/u'\}$.
(6) If t' is a variable and u' is not, then, if t' occurs in u', stop with failure; otherwise change t, u and σ to $t * \lambda$, $u * \lambda$ and $\sigma * \lambda$ where λ is either $\{u'/t'\}$ or $\{f(u', t')/t'\}$.
(7) If u' is a variable and t' is not, then, if u' occurs in t', stop with failure; otherwise change t, u and σ to $t * \lambda$, $u * \lambda$ and $\sigma * \lambda$ where λ is either $\{t'/u'\}$, or $\{f(t', u')/u'\}$.
(8) Go to 2.

Figure 1 gives an example which traces the values of t and u through the execution tree of the algorithm when $t=f(x, z)$ and $u=f(a, f(y, b))$, giving all the successful and some of the unsuccessful searches.

The set of unifiers is $\{\{a/x, f(y, b)/z\}, \{f(a, y)/x, b/z\}, \{f(a, x)/x, f(x, y)/y, f(y, b)/z\}\}$.

One can have infinitely many successes, as illustrated in figure 2, where we are now using the abbreviation mechanism, writing xy for $f(x, y)$; g is some other function. In this case, the set of unifiers produced is

$$\{\{a^n/x, a^n/y\} \mid n>0\}.$$

Infinite failures are also possible: for example, take

$t=g(x, xa)$ and $u=g(y, by)$.

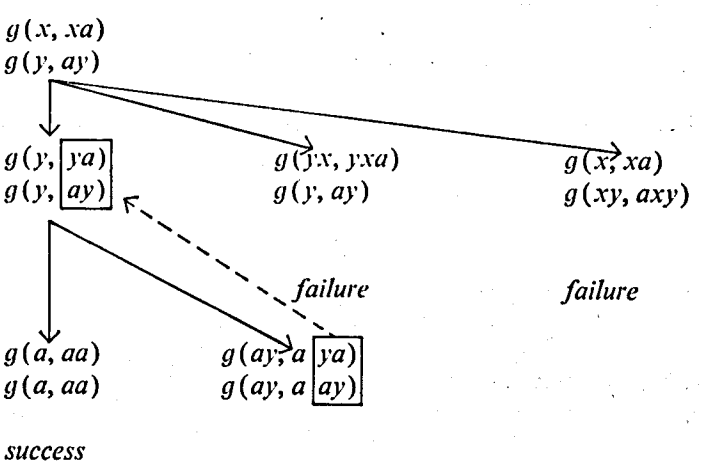


Figure 2

We conjecture that it is possible to decide whether two terms have a unifier – this is essentially the same problem as deciding whether a set of simultaneous equations in a free semigroup has a solution.

As a second example, we give an 'arithmetical evaluation' theory. Among its function symbols are two binary ones + and × and infinitely many

constants – the numerals $\Delta_n (n \geqslant 0)$. The axioms are:
(1) $+(\Delta_m, \Delta_n) = \Delta_{m+n} \qquad (m, n \geqslant 0)$
(2) $\times(\Delta_m, \Delta_n) = \Delta_{m \times n} \qquad (m, n \geqslant 0)$.
These are just the addition and multiplication tables.

$N(t)$ is obtained from t by continual application of the reduction rules:
(1) Replace an occurrence of $+(\Delta_m, \Delta_n)$ by one of $\Delta_{m+n} \qquad (m, n \geqslant 0)$
(2) Replace an occurrence of $\times(\Delta_m, \Delta_n)$ by one of $\Delta_{m \times n} \qquad (m, n \geqslant 0)$

A term, t, is a *number* term iff the only function symbols occurring in t are $+$, $\times$ or numerals.

Suppose Eq is a set of equations and Ineq is a set of sets of inequations and all terms in Eq or Ineq are number terms. A substitution, σ, *solves* Eq and Ineq iff:
(1) If x is a variable in Eq and Ineq then $x\sigma$ is a numeral and otherwise $x\sigma$ is x.
(2) If $t = u$ is in Eq then $t * \sigma$ is identical to $u * \sigma$.
(3) Every member of Ineq has a member, $t \neq u$, such that $t * \sigma$ is distinct from $u * \sigma$.

For example, $\{3/x\}$ solves $(\{x+2=5\}, \{\{x \neq 0\}\})$.

In general $\{\sigma \mid \sigma$ solves Eq and Ineq$\}$ is a partial recursive set, being the set of solutions of a certain set of Diophantine equations and inequations, effectively obtainable from Eq and Ineq.

The algorithm for producing a maximally general set of unifiers for two normal terms t and u is:
(1) Set Unify to $\{t = u\}$, Eq and Ineq to $\varnothing$ and σ to ε.
(2) Remove equations of the form $t = t$ from Unify.
(3) If Unify is empty, let μ be a substitution which solves Eq and Ineq, change σ to $\sigma * \mu$ and stop with success.
(4) Remove an equation, $t = u$, from Unify.
(5) If t and u are variables, replace Unify, Eq, Ineq and σ by Unify $* \lambda$, Eq $* \lambda$, Ineq $* \lambda$ and $\sigma * \lambda$ respectively, where $\lambda = \{t/u\}$.
(6) Suppose t is a variable and u is a number term. If t occurs in u, put $t = u$ in Eq; otherwise replace Unify, Eq, Ineq and σ by Unify $* \lambda$, Eq $* \lambda$, Ineq $* \lambda$ and $\sigma * \lambda$ respectively where $\lambda = \{u/t\}$.
(7) Suppose t is a variable and u is a term, but not a number one. If t occurs in u, stop with failure; otherwise replace Unify, Eq, Ineq and σ by Unify $* \lambda$, Eq $* \lambda$, Ineq $* \lambda$ and $\sigma * \lambda$ respectively, where $\lambda = \{u/t\}$.
(8) If u is a variable and t is not, proceed as in steps 6 and 7, but reverse the roles of t and u.
(9) If t and u begin with distinct function symbols then if either of them are not number terms stop with failure, otherwise put $t = u$ in Eq.
(10) If t and u begin with the same function symbol, f, say, and one of them is not a number term, then add

$$t_1 = u_1, \ldots, t_n = u_n \text{ to Unify where } t = f(t_1, \ldots, t_n) \text{ and } u = f(u_1, \ldots, u_n)$$

(11) If t and u have the forms $g(t_1, t_2)$ and $g(u_1, u_2)$ respectively and both

are number terms, where g is either $+$ or $\times$, then either add $t=u$ to Eq and $\{t_1 \neq u_1, t_2 \neq u_2\}$ to Ineq or else add $t_1=u_1$ and $t_2=u_2$ to Unify.
(12) Go to 2.

In the above, steps 3 and 11 are non-deterministic. The equation removed in step 4 is to be chosen in some fixed way. The algorithm always terminates, and can be implemented efficiently (apart from the difficulty of having to solve arbitrary Diophantine equations!). It uses no special properties of $+$ or $\times$ and so can be adjusted to suit any similar evaluation theory.

For example, suppose t is $\times(x, x)$ and u is $\times(\Delta_4, \times(y, y))$, then, at the beginning of the algorithm, we have Unify$=\{\times(x, x)=\times(\Delta_4, \times(y, y))\}$, Eq$=$Ineq$=\varnothing$ and $\sigma=\varepsilon$. Next, the execution sequence splits at step (11). Either Unify becomes $\varnothing$, Eq becomes $\{\times(x, x)=\times(\Delta_4, \times(y, y))\}$, Ineq becomes $\{\{x \neq \Delta_4, x \neq \times(y, y)\}\}$ and σ still has the value ε, or else Unify becomes $\{x=\Delta_4, x=\times(y, y)\}$, Eq and Ineq remain at $\varnothing$ and σ at ε. In the first case, the algorithm stops successfully at step (3), via a μ of the form $\{\Delta_{2n}/x, \Delta_n/y\}$, where $n \geqslant 0$ and $n \neq 2$, and then $\sigma=\mu$. In the second case, supposing $x=\Delta_4$ to be chosen at step (2), after step (6) we have Unify$=\{\Delta_4=\times(y, y)\}$, Eq$=$Ineq$=\varnothing$ and $\sigma=\{\Delta_4/x\}$. After steps (9) and (3), this execution path terminates with $\sigma=\{\Delta_4/x, \Delta_2/y\}$. If the commutativity and associativity of $\times$ were also built-in, one would expect here the single result, $x=\times(\Delta_2, y)$.

As a final example, the theory of an associative, commutative, idempotent binary function also has a unification algorithm – although we only know an extremely inefficient one. In this theory every pair of terms has a finite maximally general set of unifiers. This can be considered as building in sets, to some extent.

We believe most common algebraic systems admit unification algorithms. In particular we believe one can build in bags, lists and tuples in this way (Rulifson 1970). Group theory seems difficult. In general, the problem of finding a maximally general set of unifiers resembles, but is easier than the problem of solving in a closed form, equations in the corresponding free algebra. Other people have designed unification algorithms (Bennett, Easton, Guard and Settle 1967, Gould 1966, Nevins 1971, Pietrzykowski 1971, Pietrzykowski and Jensen 1972) but have not demonstrated that their methods satisfy the independence condition or, with the exception of Pietrzykowski *et al.*, the completeness one (see later). Cook (1965) seems to be the first person to notice the usefulness of normal forms in the theorem-proving context.

FORMAL PRELIMINARIES

The formalism is that of Robinson (1965) and Schoenfield (1967). The two are compatible with some minor adjustments. Clauses should be regarded as abbreviations for a corresponding sentence. Schoenfield omits brackets in his terms; these should be put back.

We use the equality symbol in infix mode and regard $t=u$ and $u=t$ as indistinguishable, thus building-in symmetry. Although the logic is not sorted, only minor adjustments are necessary to accommodate disjoint sorts.

The letters $t, u, \ldots$ range over terms. An occurrence of t in v is indicated by $v(t)$; $v(u/t)$ indicates the term obtained from $v(t)$ by replacing that distinguished occurrence of t in v by one of u.

The letters $L, M, \ldots$ range over literals. The general form of a literal is $(\pm)\, P(t_1, \ldots, t_n)$. $\bar{L}$ is like L, but has opposite sign.

The letters $C, D, \ldots$ range over clauses. By convention, $C \vee L$ represents the clause $C \cup \{L\}$ and implies L is not in C. Similarly $C \vee D$ represents $C \cup D$, and implies $C \cap D = \varnothing$ and $D \neq \varnothing$. Equations are unit clauses, $\{t=u\}$; $\{t \neq u\}$ is the general form of an inequation. The clause $\bar{C}$ is $\{\bar{L} \mid L \text{ in } C\}$.

The letter S varies over sets of clauses and Eq varies over sets of equations. The letter $\mathscr{E}$ stands for a set of sets, each member of which is either a finite set of terms or a finite set of literals. Greek letters $\sigma, \mu, \ldots$ range over substitutions: ε is the empty substitution. If $\xi = \{y_1/x_1, \ldots, y_n/x_n\}$ and the y_i are all distinct then ξ is *invertible* and ξ^{-1} is defined to be $\{x_1/y_1, \ldots, x_n/y_n\}$. The letter ξ is reserved for invertible substitutions. With each pair of clauses C and D, an invertible substitution $\xi_{C,D}$ is associated in some standard way, and is such that $C\xi_{C,D}$ and D have no common variables and if x does not occur in C, $x\xi_{C,D}$ is x.

The letter V ranges over finite sets of variables. Var(- - -) is the set of variables in the syntactic entity - - -, which can be a term, literal, clause or sets of sets of ... such; substitutions may be applied to such entities in the natural fashion. The *restriction* of σ to V, $\sigma \restriction V$ is the substitution μ such that if x is in V, $x\mu = x\sigma$ and otherwise $x\mu = x$.

From now on we discuss a fixed equational theory T with language L. In order to allow for Skolem function symbols we make:

Assumption 1. L contains infinitely many function symbols of each degree, none of which occur in T.

Suppose L has the form $(\pm)\, P(t_1, \ldots, t_n)$ and M has the form $(\pm)\, Q(u_1, \ldots, u_m)$. Then if P is not the equality symbol, $\vdash_T L \equiv M$ iff P and Q are identical, L and M have the same sign and, $\vdash_T t_i = u_i$, $(1 \leqslant i \leqslant n)$.

Simplification and normal-form functions have been defined in the introduction. They can be applied in the natural way to literals, clauses, and sets of sets of ... such.

GROUND LEVEL

We will formulate a set of rules, complete at ground level.

Define an equivalence relation, $\sim$, between literals by:

If the equality symbol does not occur in L, then $L \sim M$ iff $\vdash_T L \equiv M$. If it occurs with the same sign in both, L is $(\pm)\, t=u$ and M is $(\pm)\, v=w$, then $L \sim M$ iff $\vdash_T (t=v) \wedge (u=w)$ or $\vdash_T (t=w) \wedge (u=v)$. Otherwise, $L \nsim M$.

A set of literals is a T-*unit* iff $L \sim M$ for any L, M in the set.

If $C'' \cup \bar{D}''$ is a T-unit containing no occurrence of the equality symbol, then $C' \cup D'$ is a *ground* T-*resolvent* of $C = C' \vee C''$ and $D = D' \vee D''$.

If $C = C' \vee t_1 = u_1 \vee \ldots \vee t_n = u_n (n > 0)$, $D = D' \vee D''$, $(\pm) P(v_1, \ldots, v_{j_0}, \ldots, v_m)$ is in D'', $\{t_i = u_i | i = 1, n\}$ and D'' are T-units and $\vdash_T v_{j_0} = w(t_1)$ then

$$C' \cup D' \cup \{(\pm) P(v_0, \ldots, v_{j_0-1}, w(u_1/t_1), v_{j_0+1}, \ldots, v_m)\}$$

is a *ground* T-*paramodulant* of C and D.

If $C = C' \vee t_1 \neq u_1 \vee \ldots \vee t_n \neq u_n (n > 0)$ and $\{t_i \neq u_i | i = 1, n\}$ is a T-unit and $\vdash_T t_1 = u_1$, then C' is a *ground* T-*trivialisation* of C.

These rules are, in general, semi-effective; they are clearly consistent. If, in an application of one of the above, $\#(C'') = \#(D'') = n = 1$ ($\#$ is the cardinality function) the application has *degree* 1.

Given a set, Eq, of ground equations, write $t \approx_1 u$ iff there is a term $w(v)$ and another v' such that $\vdash_T t = w(v)$, $\{v = v'\}$ is in Eq and $u = w(v'/v)$. Note that if $t \approx_1 u$ and $\vdash_T u = v$ then for some w, $v \approx_1 w$ and $\vdash_T w = t$, that if $\vdash_T t = u$ and $u_1 \approx_1 w$ then $t \approx_1 w$, and if $t_j \approx_1 u$ then $f(t_1, \ldots, t_n) \approx_1 f(t_1, \ldots, t_{j-1}, u, t_{j+1}, \ldots, t_n)$.

Now define $\approx$ by: $t \approx u$ iff there are $t_i (n \geqslant 1$ and $1 \leqslant i \leqslant n)$ such that

$$t = t_1 \approx_1 \ldots \approx_1 t_n \text{ and } \vdash t_n = u.$$

Lemma 1. $t \approx u$ iff $\vdash_{T \cup Eq} t = u$, if t and u are ground terms.

Proof. Evidently $t \approx u$ implies $\vdash_{T \cup Eq} t = u$. Conversely, the properties of $\approx_1$ ensure that $\approx$ is a congruence relation. With the usual square bracket notation for congruence classes, define an interpretation, $\mathscr{A}$, whose domain is the set of congruence classes and whose operations are given unambiguously by:

$$\mathscr{A}(f)([t_1], \ldots, [t_n]) = [f(t_1, \ldots, t_n)].$$

Every equation in T $\cup$ Eq is true under this interpretation, so as $\vdash_{T \cup Eq} t = u$, then $[t] = \mathscr{A}(t) = \mathscr{A}(u) = [u]$; that is, $t \approx u$.

Theorem 1. Suppose S is a non-empty set of non-empty ground clauses such that $S \cup$ T has no model. Then there is a derivation of the null clause from S using the rules given above, in which all applications of the rules have degree 1.

Proof. The proof is by induction on the excess literal parameter

$$l(S) = \sum_{C \in S} (\#(C) - 1) \geqslant 0.$$

When $l(S) = 0$, S is a set of unit clauses. Let Eq $\subseteq S$ be the set of equations in S. If every set of the form Eq $\cup \{t \neq u\} \cup$ T ($\{t \neq u\}$ in S) or the form Eq $\cup \{\{L\}\{M\}\} \cup$ T ($\{L\}, \{M\}$ in S) is satisfiable, then it can be shown that $S \cup$ T is satisfiable. Consequently a set with one such form is unsatisfiable.

In the first case, $\vdash_{Eq \cup T} t = u$ and so, with $\approx$ as in lemma 1, $t \approx u$. So, by successive ground T-paramodulations from Eq into $t \neq u$, one can obtain a clause $t' \neq u$ such that $\vdash_T t' = u$. From this the null clause is obtained by a ground T-trivialisation of degree 1.

The second case is similar but uses a ground T-resolution.

If $l(S)>0$, there is a literal L in a non-unit clause C in S. Applying the induction hypothesis to $S'=(S\setminus\{C\})\cup(C\setminus\{L\})$, we obtain a derivation of the null clause, by the above rules, from S'. So either there is a derivation of the null clause from S by the rules or there is one of $\{L\}$ by them, from S. In the second case, we obtain, by induction, a refutation of $S''=(S\cup\{L\})\setminus\{C\}$ and the proof is concluded.

Notice that in the above if S contains no occurrences of the equality symbol, then a refutation of S can use only ground T-resolution; similar results hold for other grammatical possibilities.

A clause C, is a *ground N-T-resolvent of D and E* iff $C=N(C')$ for some ground T-resolvent of D and E. *Ground N-T-paramodulation* and *trivialisation* are similarly defined.

A partial order $\precsim$ is defined between clauses by:

$C\precsim D$ iff there is a function ϕ: $C\to D$ such that $L\sim\phi(L)$ for every L in C.

Note that $N(C)\precsim C$, for any clause C

One can check that if $C\precsim C'$, $D\precsim D'$ and E' is a ground T-resolvent of C' and D' then either $C\precsim E'$, $D\precsim E'$, or $E\precsim E'$ for some ground T-resolvent E of C and D; similar results hold for the other two rules.

From these remarks, it follows that if S has a refutation by the T rules, it has one by the N-T ones; further remarks analogous to those made immediately after theorem 1 also apply. (Actually only degree 1 applications of the rules are necessary).

Greater freedom in the application of N is permissible. One can allow N to depend on the derivation of the clause to which it is being applied; perhaps N could even depend on the state of execution of the algorithm searching for a refutation.

UNIFICATION

A substitution, σ, is a T-*unifier* of $\mathscr{E}$ iff when t, u are terms in the same set in $\mathscr{E}$, $\vdash_T t\sigma=u\sigma$ and if L, M are literals in the same set in $\mathscr{E}$, $L\sigma\sim M\sigma$.

An equivalence relation between substitutions is defined by:

$\sigma\sim\mu\ (V)$ iff, for all variables, x, in $V\vdash_T x\sigma=x\mu$.

Note that, if $\sigma\sim\mu(V)$ then $\sigma\nu\sim\mu\nu(V)$ and if $\sigma\sim\mu\,(\mathrm{Var}(V\nu))$ then $\nu\sigma\sim\nu\mu(V)$ for any ν.

A set, Γ, of substitutions is a *maximally general set of* T-*unifiers* (MGSU) of $\mathscr{E}$ *away from* V iff:

(1) (Correctness). Every σ in Γ T-unifies $\mathscr{E}$.

(2) (Completeness). If σ' T-unifies $\mathscr{E}$, then there is a σ in Γ and a λ such that $\sigma'\sim\sigma\lambda\,(\mathrm{Var}(\mathscr{E}))$.

(3) (Independence). If σ and σ' are distinct members of Γ, then for no λ does $\sigma'\sim\sigma\lambda\,(\mathrm{Var}(\mathscr{E}))$.

(4) For every σ in Γ and x not in $\mathrm{Var}(\mathscr{E})$, $x\sigma=x$.

(5) For every σ in Γ, $\mathrm{Var}(\mathscr{E}\sigma)\cap V=\varnothing$.

Conditions 4 and 5 are technical: from every set satisfying the first three conditions one can effectively and, indeed, efficiently construct one satisfying them all. Note that conditions 2 and 3 use relative equivalence rather than equality. These conditions can be satisfied in cases where the corresponding ones with equality cannot, and we know of no example of the opposite situation.

We also know of no example of a theory T and an $\mathscr{E}$ and V for which there is no such Γ, although we expect that one exists.

However we make:

Assumption 2. There is a partial recursive predicate $P(\mathscr{E}, \sigma, V)$ such that $\Gamma(\mathscr{E}, V) = \{\sigma | P(\mathscr{E}, \sigma, V)\}$ is a MGSU for $\mathscr{E}$ apart from V.

On this basis, semi-effective proof procedures can be developed. Of course in each case we should look for efficient algorithms for generating Γ.

We conjecture that if L_1 and L_2 are any two languages, not necessarily satisfying assumption 1 and L_1 is the language of T then if there is a (partial recursive, recursive, effectively obtainable, efficiently obtainable) MGSU for any $\mathscr{E}$, whose vocabulary is in L_1 and any V then there is a (partial recursive, recursive, effectively obtainable, efficiently obtainable) MGSU for any whose vocabulary is in $L_1 \cup L_2$. This would eliminate the necessity for assumption 1 and would as a special case, yield unification algorithms for evaluation systems like the arithmetical one in the introduction. Indeed if we consider theories T_1, T_2 whose languages are L_1, L_2 respectively, such that $L_1 \cap L_2 = \varnothing$, it may be the case that MGSU's for $T_1 \cup T_2$ always exist (and are partial recursive, etc.) if the same holds for T_1 and T_2 individually.

The algorithms given in the introduction do not quite conform to the specifications in that they do not give unifiers for an arbitrary $\mathscr{E}$ and, further, violate condition 5, in general. However, this is easily corrected. Interestingly, instead of condition 5, they satisfy:

(5′) For every σ in Γ, $\mathrm{Var}(\mathscr{E}\sigma) \subseteq \mathrm{Var}(\mathscr{E})$.

This condition allows the subsequent theory to proceed with some modifications, but can't be fulfilled for some T's for which 5 can.

MGSU's are essentially unique. If Γ, Γ' are MGSU's for $\mathscr{E}$ away from V and V', say, there is a bijection $\phi: \Gamma \to \Gamma'$ such that for all σ in Γ there are λ, λ' such that $\sigma \sim \phi(\sigma)\lambda(\mathrm{Var}(\mathscr{E}))$ and $\phi(\sigma) \sim \sigma\lambda'$ $(\mathrm{Var}(\mathscr{E}))$.

Again, if $\mathscr{E}$ and $\mathscr{E}'$ have the same set of T-unifiers and variables, Γ is a MGSU for $\mathscr{E}$ away from V iff it is for $\mathscr{E}'$. This holds, in particular, if $\mathscr{E}' = N(\mathscr{E})$.

A change of variables in $\mathscr{E}$ causes a corresponding one in Γ. If Γ is a MGSU for $\mathscr{E}$ away from V then if ξ maps distinct variables of $\mathscr{E}$ to distinct variables, $\{(\xi^{-1}\sigma) | \mathrm{Var}(\mathscr{E}\xi) | \sigma \in \Gamma\}$ is a MGSU for $\mathscr{E}\xi$ away from V.

Before defining suitable generalisations of resolution and so on, one should decide between formulations with or without factoring. Usually the difference is one, merely, of efficiency; the justification lies in a theorem of Hart (1965) (see also Kowalski (1970)), that if σ is a m.g.u. of $\mathscr{E}$ and σ' is of $\mathscr{E}'\sigma$ then $\sigma\sigma'$ is of $\mathscr{E} \cup \mathscr{E}'$. Unfortunately the generalisation of this theorem fails. We can prove:

Theorem 2 (*Weak Refinement Theorem*). Suppose Γ is a MGSU for $\mathscr{E}_1$ apart

from $\mathrm{Var}(\mathscr{E}_2) \cup V$ and for each σ in Γ, Γ_σ is a MGSU for $\mathscr{E}_2\sigma$, apart from $\mathrm{Var}(\mathscr{E}_1\sigma) \cup V$.

Then, $\Gamma' = \{\sigma\mu \restriction \mathrm{Var}(\mathscr{E}_1 \cup \mathscr{E}_2) \mid \sigma \in \Gamma, \mu \in \Gamma_\sigma\}$ satisfies conditions 1, 2, 4 and 5 for being an MGSU for $\mathscr{E}_1 \cup \mathscr{E}_2$ away from V.

Proof. Conditions 1, 4 and 5 may be straightforwardly verified. To establish condition 2, suppose θ T-unifies $\mathscr{E}_1 \cup \mathscr{E}_2$. Then by condition 2 on Γ, there is a σ in Γ and a λ such that:

$$\theta \sim \sigma\lambda(\mathrm{Var}(\mathscr{E}_1)).$$

It can be assumed, without loss of generality, that $\lambda = \lambda \restriction \mathrm{Var}(\mathscr{E}_1\sigma)$. Now $\mathrm{Var}(\mathscr{E}_2) \cap \mathrm{Var}(\mathscr{E}_1\sigma) = \varnothing$ by hypothesis and condition 5 on Γ. So $\lambda' = \lambda \cup (\theta \restriction \mathrm{Var}(\mathscr{E}_2))$ is a substitution. Now:

$$\theta \sim \sigma\lambda' (\mathrm{Var}(\mathscr{E}_1 \cup \mathscr{E}_2)).$$

For if x is in $\mathrm{Var}(\mathscr{E}_1)$ then $\mathrm{Var}(x\sigma) \cap \mathrm{Var}(\mathscr{E}_2) = \varnothing$, so $x\sigma\lambda' = x\sigma\lambda$ and $\vdash_T x\sigma\lambda = x\theta$. If x is in $\mathrm{Var}(\mathscr{E}_2)$ but not in $\mathrm{Var}(\mathscr{E}_1)$ then $x\sigma = x$, by condition 4 on Γ and $x\lambda = x$ by assumption. So $x\lambda' = x(\theta \restriction \mathrm{Var}(\mathscr{E}_2)) = x\theta$.

Therefore $\sigma\lambda$ T-unifies $\mathscr{E}_1 \cup \mathscr{E}_2$, λ T-unifies $\mathscr{E}_2\sigma$ and so by condition 2 on Γ_σ, there is a μ in Γ_σ and a δ such that:

$$\lambda' \sim \mu\delta \, (\mathrm{Var}(\mathscr{E}_2\sigma))$$

It can be assumed without loss of generality that $\delta \restriction \mathrm{Var}(\mathscr{E}_2\sigma\mu) = \delta$. Now, $\mathrm{Var}(\mathscr{E}_1\sigma) \cap \mathrm{Var}(\mathscr{E}_2\sigma\mu) = \varnothing$, by hypothesis and condition 5 on Γ_σ. So $\delta' = \delta \cup (\lambda' \restriction \mathrm{Var}(\mathscr{E}_1\sigma))$ is a substitution and one can show much as above that:

$$\lambda' \sim \mu\delta' \quad (\mathrm{Var}((\mathscr{E}_1 \cup \mathscr{E}_2)\sigma))$$

Then $\sigma\lambda' \sim \sigma\mu\delta \quad (\mathrm{Var}(\mathscr{E}_1 \cup \mathscr{E}_2))$ and:

$$\theta \sim \sigma\lambda' \sim \sigma(\mu\delta') \sim (\sigma\mu \restriction \mathrm{Var}(\mathscr{E}_1 \cup \mathscr{E}_2))\delta' \quad (\mathrm{Var}(\mathscr{E}_1 \cup \mathscr{E}_2)),$$

concluding the proof.

When T is the theory of an associative, commutative idempotent function, one can find examples where condition 3 fails. We will therefore formulate the rules without using factoring. (Factoring would still result in completeness, but would cause redundancy which although eliminable by a subsumption-type check would probably cause more trouble than it was worth.) Condition 3 does hold, however, in the associative and evaluation examples given in the introduction.

Gould (1966) has defined, in the context of ω-order logic, the concept of a general matching set.

In our terms, a literal, M, is a T-*unification* of L and L' iff for some σ, $L\sigma \sim M \sim L'\sigma$.

Then, Δ is a *general matching set of literals* for L and L' away from V iff:

(1) Every member of Δ is a T-unification of L and L'.

(2) If M' is a T-unification of L and L', then there is a λ and an M in Δ such that $M\lambda \sim M'$.

(3) If M, M' are distinct members of Δ then for every λ, $M\lambda \nsim M'$.

(4) If M is in Δ, $\mathrm{Var}(M) \cap V = \varnothing$.

In our opinion, this concept cannot, in general, be made the basis of a complete inference system; a counterexample will be given later.

GENERAL LEVEL

We begin with the rules.

Suppose σ is in $\Gamma(\{C''\xi_{C,D} \cup \bar{D}''\}, \mathrm{Var}(C\xi_{C,D} \cup D))$ where $C=C' \vee C''$, $D=D' \vee D''$ and the equality symbol has no occurrence in C''. Then,

$C'\xi_{C,D}\sigma \cup D'\sigma$ is a T-*resolvent* of C and D.

A variable-term pair is *special* iff it is of the form $\langle x, x\rangle$ or else $\langle x, f(x_1, \ldots, x_{i-1}, t, x_{i+1}, \ldots, x_n)\rangle (n>0)$ and $\langle x, t\rangle$ is special and the x_i are distinct and not in t.

Given an occurrence of a term u in another v there is a unique, to within alphabetic variance, special pair $\langle x, t\rangle$ and a substitution η such that:

(1) $v=t\{u/x\}\eta$,

(2) $\{u/x\}\eta=\eta\{u/x\}$,

(3) x has the same occurrence in t as u in v, and

(4) $\mathrm{Var}(t) \cap \mathrm{Var}(v)=\varnothing$.

We assume available a function, *Sp*, from finite sets of variables to special pairs such that:

(1) There is an alphabetic variant of every special pair in $Sp(V)$.

(2) No two distinct members of $Sp(V)$ are alphabetic variants.

(3) If $\langle x, t\rangle$ is in $Sp(V)$, $\mathrm{Var}(t) \cap V=\varnothing$.

Suppose $C=C' \vee t_1=u_n \vee \ldots \vee t_n=u_n (n>0)$, $D=D' \vee D''$, $(\pm)\ P(v_1, \ldots, v_{j_0}, \ldots, v_m)$ is in D'', $\langle x, t\rangle$ is in $Sp(\mathrm{Var}(C\xi_{C,D} \cup D))$ and that $\mathscr{E}=\{\{t_i\xi_{C,D} | i=1, n\}, \{u_i\xi_{C,D} | i=1, n\}, D'', \{v_{j_0}, t\{t_1\xi_{C,D}/x\}\}\}$.

Then, if σ is in $\Gamma(\mathscr{E}, \mathrm{Var}(C\xi_{C,D} \cup D))$,

$C'\xi_{C,D}\sigma \cup D'\sigma \cup \{(\pm)P(v_1\sigma, \ldots, v_{j_0-1}\sigma, t\{u_1\xi_{C,D}/x\}\sigma, v_{j_0+1}\sigma, \ldots, v_m\sigma)$ is an *assisted primary* T-*paramodulant from C into D*.

Suppose that $C=C' \vee t_1 \neq u_1 \ldots \vee t_n \neq u_n (n>0)$ and σ is in $\Gamma(\{\{t_i | i=1, n\} \cup \{v_i | i=1, n\}\}, \mathrm{Var}(C))$; then, $C'\sigma$ is a T-*trivialisation* of C.

These three rules are evidently consistent and semi-effective.

For completeness a lifting lemma is needed.

Lemma 2

(1) Suppose E' is a ground T-resolvent of ground clauses $C\mu$ and Dv. Then there is a T-resolvent, E of C and D and a λ such that $E\lambda \subseteq E'$.

(2) Suppose E' is a ground T-paramodulant from the ground clause $C\mu$ into the ground clause Dv. Then there is an assisted primary T-paramodulant, E, from C into D and a λ such that $E\lambda \subseteq E'$.

(3) Suppose E' is a ground T-trivialisation of the ground clause $C\mu$. Then there is a T-trivialisation, E, of C and a λ such that $E\lambda \subseteq E'$.

Proof. Only the proof of part 2, the hardest one, will be given.

Let $\sigma'=(\xi_{C,D}^{-1}\mu \restriction \mathrm{Var}(C\xi_{C,D})) \cup (v \restriction \mathrm{Var}(D))$.

Then $C\xi_{C,D}\sigma'=C\mu$ and $D\sigma'=Dv$. So, as E' is a ground T-paramodulant of $(C\xi_{C,D})\sigma'$ and $D\sigma'$ there are subsets C', D' and D'' of C and D, respectively,

and terms $t_i, u_i\ (i=1, n)$, $t, u, v'_{j_0}, v'(t)$ and a literal $(\pm)P(v'_1, \ldots, v'_{j_0}, \ldots, v'_m)$ in $D''\sigma'$ such that:

$C\xi_{C,D}\sigma' = C'\xi_{C,D}\sigma' \vee \{(t_i = u_i)\xi_{C,D}\sigma' \mid i=1, n\}$,
$D\sigma' = D'\sigma' \vee D''\sigma'$,
$t_1\xi_{C,D}\sigma' = t$
$u_1\xi_{C,D}\sigma' = u$
$\vdash_T v'_{j_0} = v'(t)$,
$E' = C'\xi_{C,D}\sigma' \vee D'\sigma' \vee (\pm)P(v'_1, \ldots, v'_{j_0-1}, v'(u/t), v'_{j_0+1}, \ldots, v'_m)$,
$\{(t_i = u_i)\xi_{C,D}\sigma' \mid i=1, n\}$ and $D''\sigma'$ are T-units,
$C = C' \vee t_1 = u_1 \vee \ldots \vee t_n = u_n$,
$D = D' \vee D''$,
$\vdash_T t_i\xi_{C,D}\sigma' = t \quad (i=1, n)$ and
$\vdash_T u_i\xi_{C,D}\sigma' = u \quad (i=1, n)$.

As t is a subterm of v', there is a unique $\langle x, w\rangle$ in $Sp(\mathrm{Var}((C\xi_{C,D} \cup D))$ and an η such that:

$v' = w\{t/x\}\eta$, $\{t/x\}\eta = \eta\{t/x\}$ and x has the same occurrence in w as the distinguished occurrence of t in v'.

All the above equations hold if σ' is replaced by $\sigma'' = \sigma'\eta$, as $C\mu$ and Dv are ground.

Now, let $(\pm)P(v_1, \ldots, v_m)$ be a literal in D'' such that $(\pm)P(v_1, \ldots, v_m)\sigma' = (\pm)P(v'_1, \ldots, v'_m)$. We have,

$v_{j_0}\sigma'' = v'_{j_0}$,
$\vdash_T v'_{j_0} = v'(t)$ and
$v'(t) = w\{t/x\}\eta = w\{t_1\xi_{C,D}\sigma'/x\}\eta$
$= w\{t_1\xi_{C,D}/x\}\sigma'\eta \qquad (\text{as } \sigma' \restriction \mathrm{Var}(w) = \varepsilon)$
$= w\{t_1\xi_{C,D}/x\}\sigma''$.

Therefore, $\vdash_T v_{j_0}\sigma'' = w\{t_1\xi_{C,D}/x\}\sigma''$, and we have proved that σ'' T-unifies $\mathscr{E} = \{\{t_i\xi_{C,D} \mid i=1, n\}, \{u_i\xi_{C,D} \mid i=1, n\}, D'', \{v_{j_0}, w\{t_1\xi_{C,D}/x\}\}\}$. So there is a σ in $\Gamma(\mathscr{E}, \mathrm{Var}((C\xi_{C,D} \cup D))$ and a λ' such that:

$\sigma'' \sim \sigma\lambda' \ (\mathrm{Var}(\mathscr{E}))$.

One then finds, as in the proof of theorem 2, a λ such that

$\sigma'' \sim \sigma\lambda \ (\mathrm{Var}((C\xi_{C,D} \cup D))$.

Now $E = C'\xi_{C,D}\sigma \cup D'\sigma \cup \{(\pm)P(v_1\sigma, \ldots, v_{j_0-1}\sigma,\ w\{u_1\xi_{C,D}/x\}\sigma, v_{j_0+1}\sigma, \ldots, v_m\sigma)\}$, is an assisted primary T-paramodulant from C into D.

Since $w\{u_1\xi_{C,D}/x\}\sigma'' = w\{u_1\xi_{C,D}/x\}\sigma'\eta$
$= w\{u_1\xi_{C,D}\sigma'/x\}\eta \qquad (\text{as } \sigma' \restriction \mathrm{Var}(w) = \varepsilon)$
$= w\{u/x\}\eta$
$= v'(u/t)$,

$E\lambda = C'\xi_{C,D}\sigma\lambda \cup D'\sigma\lambda \cup \{(\pm)P(v_1\sigma\lambda, \ldots, v_{j_0-1}\sigma\lambda, w\{u_1\xi_{C,D}/x\}\sigma\lambda, v_{j_0+1}\sigma\lambda, \ldots, v_m\sigma\lambda)\}$
$\subseteq C'\xi_{C,D}\sigma'' \cup D'\sigma'' \cup \{(\pm)P(v_1\sigma'', \ldots, v_{j_0-1}\sigma'', w\{u_1\xi_{C,D}/x\}\sigma'', v_{j_0+1}\sigma'', \ldots, v_m\sigma'')\}$
$= C'\xi_{C,D}\sigma'' \cup D'\sigma'' \cup \{(\pm)P(v'_1, \ldots, v'_{j_0-1}, v'(u/t), v'_{j_0+1}, \ldots, v'_m)\}$
$= E'$, concluding the proof.

Theorem 3. If S is a non-empty set of non-empty clauses such that $S \cup T$ has no model, there is a derivation of the null clause from S using the rules given above.

Proof. First it is shown by induction on derivation size (that is, the number of applications of rules) that if there is a derivation of a clause E' from a set of ground clauses of the form $\bigcup_i (S\sigma_i)$, using the ground rules, then there is a general derivation of a clause E from S and a λ such that $E\lambda \subseteq E'$.

When the derivation has size zero this is immediate.

Otherwise either there is a derivation of a ground clause C' from a set of clauses of the form $\bigcup_i (S\sigma_i)$ and another of a ground clause D' from a set of the form $\bigcup_j (S\sigma'_j)$, each derivation having strictly smaller size than the main derivation under consideration and E' is a ground T-resolvent of C' and D', or a corresponding statement involving one of the other two ground rules is true.

Let us suppose the first case holds. Then, by induction there are general derivations of clauses C and D from S and, substitutions μ and ν such that $C\mu \subseteq C'$ and $D\nu \subseteq D'$. By a remark made after theorem 1 either $C\mu \subseteq E'$, $D\nu \subseteq E'$ or there is a ground T-resolvent E'' of $C\mu$ and $D\nu$ such that $E'' \subseteq E'$. In the first two subcases, we are finished with this case. In the third, by lemma 2.1 there is a T-resolvent, E, of C and D and a λ such that $E\lambda \subseteq E''$, concluding this case.

A similar proof works in the other two cases.

Now, as $S \cup T$ has no model, there are, by Herbrand's Theorem, substitutions σ_i, such that $\bigcup_i (S\sigma_i)$ is ground and $\bigcup_i (S\sigma_i) \cup T$ has no model. Hence by theorem 1, there is a ground derivation of the null-clause from $\bigcup_i (S\sigma_i)$. By the above, there is a general derivation of a clause E from S and a λ such that $E\lambda \subseteq \varnothing$. Then $E = \varnothing$, concluding the proof.

Kowalski (1968) proved that, when $T = \varnothing$ one need only paramodulate into the functional reflexivity axioms, not from them, and all other paramodulations can be primary. Theorem 3 strengthens this result: special pairs take the place of the functional reflexivity axioms. One can also insist on $P1$ and A-ordering restrictions, analogous to Kowalski's, without losing completeness.

Other refinements are possible. For example one can define an E-T-resolution rule and a corresponding E-T-trivialisation one analogous to E-resolution (Morris, 1969). These two rules form a complete set. If one regards an E-T-resolution as a many-input rule, rather than a deduction using several T-paramodulations and a T-resolution (and similarly for E-T-trivialisation), and says that a clause has *support* if it is in the support set or is obtained by a rule application with a clause with support in its input set, the resulting set of support strategy is complete. (Anderson (1970) showed that a stronger one is incomplete). Presumably, although we have

not checked, one can obtain systems analogous to the other ones for equality.

As in the ground case, when there is no occurrence of the equality symbol in S, there is a refutation using only T-resolution, with similar things holding for the other grammatical possibilities. When there are no equality symbols occurring in S, refinements analogous to all the usual refinements of the resolution principle hold.

Again, using a simplification function N, one can define N-T-resolution and so on and obtain N-T-versions of all the above, by methods like those used in the ground case.

We do not know how to formulate the paramodulation conjecture, as the obvious way fails.

If $C=C' \vee C''$ and σ is in $\Gamma(\{C\}, \mathrm{Var}(C))$, then if L is in C'', $C'\sigma \cup \{L\sigma\}$ is a T-*factor* of C, with *distinguished literal* $L\sigma$.

If $C'=C'' \vee t=u$ is a T-factor of C with distinguished literal $t=u$, $D'=D'' \vee (\pm)P(v_1, \ldots, v_{j_0}, \ldots, v_m)$ is a T-factor of D with distinguished literal $(\pm)P(v_1, \ldots, v_m)$, $w(t')$ is a term such that $\vdash_T v_{j_0}=w(t')$ and σ is in $\Gamma(\{\{t, t'\}\}, \mathrm{Var}(C'\xi_{C',D'} \cup D'))$ then

$$C''\xi_{C',D'}\sigma \cup D''\sigma \cup \{(\pm)P(v_1, \ldots, w(u\xi_{C,D}/t'), \ldots, v_m)\sigma\}$$

is a T-*paramodulant from C into D*.

However, suppose T is the theory of an associative function, then if $S=\{f(bx, x)\neq f(xa, caa), bc=ca\}$, T is unsatisfiable but has no refutation by T-resolution, T-trivialisation and T-paramodulation.

A clause C, is a T-tautology iff $\vdash_T C$. We don't know if any deletion strategies work, even when $T=\varnothing$. If there are non-equality literals L, M in C such that $L\sim M$, or a literal $t=u$ in C such that $\vdash_T t=u$, then C is a *weak* T-*tautology*. The usual deletion strategies altered to take account of the extra rules work in this case.

A clause $C=\{L_i\}$ T-*subsumes* another $D=\{M_j\}$ iff there is a σ such that $\vdash_T (\bigvee_i L_i\sigma) \supset (\bigvee_j M_j)$. We do not know if any deletion strategies work, even when $T=\varnothing$. If $C\sigma \subseteq D$ for some σ then C *weakly* T-*subsumes* D. Appropriate alterations of the usual deletion strategies work in this case.

Here is an example showing why we do not consider that the concept of a general matching set of literals (or whatever) can be made the basis of a complete system of inference.

Consider the rule:

Let $C'=C'' \vee L$ and $D'=D'' \vee L'$ be T-factors of clauses C and D, with distinguished literals L and L' respectively. If M is in a general matching set of literals for $L\xi_{C',D'}$, and $\bar{L}'$ away from $\mathrm{Var}(C'\xi_{C',D'} \cup D')$, σ is such that $L\xi_{C',D'}\sigma \sim M \sim \bar{L}'\sigma$, and the equality symbol does not occur in L, then $C''\xi_{C',D'}\sigma \cup D''\sigma$ can be deduced from C and D by the rule.

But if T is the theory of an associative binary function, recursive effectively obtainable general matching sets for L and L' away from V exist, for all L, L'

and V, and furthermore the required σ's in the definition of the rule can be calculated from $L\xi_{C',D'}$, M and L'.

However, if $S=\{\bar{P}(ax, aa), \bar{Q}(aa, ax), P(yz, x) \vee Q(y, xw)\}$ then $S \cup T$ is unsatisfiable but S has no refutation by the above rule.

EFFICIENCY

The most obvious difficulty is that there can be infinitely many T-resolvents, etc., of two clauses. This does not cause any difficulty as far as the Hart-Nilsson *theory* is concerned (Kowalski 1972). It may be possible in some cases to represent the infinite set by using terms with parameters. These have already been used informally in the introduction.

Sometimes, as when T is the theory of an associative function, the unifiers are generated as successful nodes in a search tree. In this case the unification search tree can be incorporated in the general search space and one can save the unused parts.

However, we would like to advance the thesis that in general these large numbers of unifiers are present in the usual search spaces – we have just brought them to the surface. Indeed we believe that our method can have advantages over the standard resolution one precisely analogous to those obtained by the resolution method over the Davis-Putnam one. We will defend our method in the case where the theory is that of an associative function; the T-search space will be compared to that generated by resolution, paramodulation and trivialisation.

It can be shown that the associative unification of two literals, say, may be simulated by successive paramodulations of the axiom of associativity into these literals followed by an ordinary unification. The complexity of the simulation is about the same as that of the unification, using a symbol count. So the T-search space is included in the usual one.

The resolution method has smaller redundancy and avoids more irrelevancies than the Davis-Putnam method. For example, if L and M are unifiable then in general $\{\bar{L}, M\}$ has exactly one resolution refutation, but infinitely many Davis-Putnam ones. The crudest resolution search procedure produces no irrelevancies but the Davis-Putnam will, in general. If L, M are not unifiable, this will be detected by the resolution method, but not by the Davis-Putnam one. These phenomena are manifestations of the most general unifier method.

Similarly, if L and M have T-unifications, $\{\bar{L}, M\}$ can have many fewer T-resolution refutations than the comparison system. For example $\{\bar{P}(ax), P(yb)\}$ have two T-unifiers ($\{a/y, b/x\}$ and $\{ay/y, yb/x\}$ but there are infinitely many standard refutations of $\{\bar{P}(ax), P(yb)\}$, which essentially produce the T-unifiers, $\{ay_1 \ldots y_n/y, y_1 \ldots y_m b/x \mid m \geqslant 0\}$. Each of these unifiers can be produced in many different ways, involving arbitrarily long detours of bracket swapping.

As another example, consider $\{\bar{P}(x, y, xy), P(z, w, wz)\}$ which has an

infinite set of refutations involving the T-unifiers $\{\{w^m/z, w^m/x, w^n/y, w^n/w\}|$ *m and n positive integers with greatest common divisor unity*$\}$. The standard method essentially produces the denser set:

$$\{\{w^m/z, w^m/x, w^n/y, w^n/w\}|m, n>0\}.$$

Sometimes if L and M have no T-unifier, the T-unification algorithm will stop, when the standard procedure does not – for example, if $L=P(xa)$ and $M=\bar{P}(yb)$; but generally it will also generate useless structures (though not so many).

We believe that these informal remarks can be converted into a rigorous proof of increased efficiency.

On the other hand, the associative unification procedure can certainly be greatly improved, and we have no practical experience at the moment. It is surely not the case that these methods will by themselves make a practical theorem-prover, of course. We have only removed one of many exponential explosions.

PROBLEMS

There are many obvious problems concerning particular axiom systems and how to combine different unification procedures.

However, what does one do with only partial knowledge? Suppose one has a simplification method, but no unification algorithm as is the case, at the moment, with group theory or with integration algorithms (Moses 1967)? Or, what use can be made of a one-way unification procedure? How and when does one simplify (Moses 1971)? Answering such questions might produce more efficient systems closer to normal human practice.

Acknowledgements

I have had many helpful discussions with Rod Burstall. Jerry Schwarz found some mistakes in a draft. The work was supported by the Science Research Council.

REFERENCES

Anderson, R. (1970) Completeness results for E-resolution. *Proc. AFIPS 1970 Spring Joint Comp. Conf.* Washington, DC.

Bennett, J.H., Easton, W.B., Guard, J.R. and Settle, L.G. (1967) C.R.T.-aided semi-automated mathematics. Final report. AFCRL-67-0167. Princeton: Applied Logic Corporation.

Cook, S.A. (1965) Algebraic techniques and the mechanisation of number theory. RM-4319-PR. California, Santa Monica: RAND Corporation.

Gould, W.E. (1966) A matching procedure for ω-order logic. *Sci. Rep. No. 4.* AFCRL 66-781. Princeton, New Jersey: Applied Logic Corporation.

Hart, T.P. (1965) A useful property of Robinson's unification algorithm. *A.I. Memo 91.* Project MAC. Cambridge, Mass.: MIT.

Kowalski, R.A. (1968) The case for using equality axioms in automatic demonstration. *Lecture Notes in Mathematics*, vol. 125. (eds Laudet, M., Nolin, L. and Schützenberger, M.) Berlin: Springer-Verlag.

Kowalski, R. (1970) Studies in the completeness and efficiency of theorem-proving by resolution. Ph.D. Thesis. Department of Computational Logic, University of Edinburgh.

Kowalski, R.K. (1972) And-or graphs, theorem-proving graphs and bi-directional search. Machine Intelligence 7, paper 10 (eds Meltzer, B. & Michie, D.). Edinburgh: Edinburgh University Press.

Morris, J.B. (1969) E-resolution: extension of resolution to include the equality relation. *Proc. First Int. Joint Conf. on Art. Int.* Washington, DC.

Moses, J. (1967) Symbolic integration. *Report* MAC-TR-47. Project MAC. Cambridge, Mass.: MIT.

Moses, J. (1971) Algebraic simplification: a guide for the perplexed. *Proc. Second Symp. on Symbolic and Algebraic Manipulation*, pp. 282–303. (ed. Petrich, S.R.). New York: Association for Computation Machinery.

Nevins, A.J. (1971) A human-oriented logic for automatic theorem proving. TM-62789. George Washington University.

Pietrzykowski, T. (1971) A complete mechanisation of second-order logic. Science Research Report (CSRR 2038). Department of Analysis and Computer Science, University of Waterloo.

Pietrzykowski, T. & Jensen, D.C. (1972) A complete mechanisation of ω-order logic. Science Research Report CSRR 2060. Department of Analysis and Computer Science, University of Waterloo.

Plotkin, G.D. (1972) Some unification algorithms. Research Memorandum (forthcoming). School of Artificial Intelligence, University of Edinburgh.

Robinson, J.A. (1965) A machine-oriented logic based on the resolution principle. *J. Ass. comput. Mach.*, **12**, 23–41.

Rulifson, J.F. (1970) Preliminary specification of the QA4 language. Artificial Intelligence Group. Technical Note 50. Stanford Research Institute.

Schoenfield, J.R. (1967) *Mathematical logic.* Reading, Mass.: Addison-Wesley.

5

Theorem Proving in Arithmetic without Multiplication

D. C. Cooper

Department of Computer Science
University College of Swansea

INTRODUCTION

A considerable amount of effort has been expended in the last ten years in the field known as 'mechanical theorem proving'. Original motivation in this area was primarily the attempt to prove interesting theorems in mathematics although applications in other areas were considered – see, for example, McCarthy (1958), where application to more general problem solving is made, and W.S.Cooper (1964), where the need for a theorem proving system in information retrieval was considered. More recently other areas have arisen such as robotology, automatic program construction and proofs of correctness of programs. Most work has been done in the first order predicate calculus, but at various times pleas have been made not only for general theorem provers but also for efficient provers in restricted areas. One such is presented in this paper.

In Cooper (1971) a set of programs was described to aid the proving of the convergence and correctness of programs. Part of this package was a theorem prover for the logical system consisting of the integers, integer variables, addition, the usual arithmetical relations and the usual first order logical connectives. This system is commonly referred to as Presburger arithmetic – see Presburger (1929), where a similar system involving only the equality relation was proved decidable. Whilst in such a system one can only state rather simple theorems, yet an efficient algorithm to test for the validity of such formulas is useful in that it can quickly dispose of a host of the simpler formulas arising in some application, leaving only the more complex to be dealt with by some more general theorem prover or by the human. However, the present known algorithm proved impractical even on the formulas produced by the simple program in that paper, due to exponential growth. In this paper we investigate this growth and give a new algorithm which completely eliminates one factor in the growth and considerably mitigates

another. It is possible that decision procedures in other areas could be related to that for Presburger arithmetic, thus providing another outlet for the algorithm. One such area is the equivalence of certain special program schemas – see section 4 of Paterson (1972).

THE LOGICAL SYSTEM

A formula of the system is formed from algebraic expressions (only allowing variables, integer constants and addition), the binary relation $<$, the propositional calculus logical connectives and quantification. The domain is that of positive and negative integers, and formulas of the system take their usual meaning. The other arithmetic relations may easily be added – for example $a \geqslant b$ can be defined as $b<a+1$ and $a=b$ as $a<b+1 \wedge b<a+1$ – although it could well lead to a more efficient algorithm if equality were included in the basic system. Subtraction can be allowed: $a-b>c$ is $a>c+b$. The major omission is multiplication, although multiplication by a constant can be included: $3*a$ is $a+a+a$. Such a system is known to be decidable. Examples of simple formulas in the system are:

$$(\forall a)(\forall b)(\exists x)(a<20x \wedge 20x<b)$$

and $$(\forall a)(\exists b)(a<4b+3a \vee (\neg a<b \wedge a>b+1))$$

For the technique to be used in the decision procedure it is essential to introduce another relation, even though that relation is definable in the system. This relation can be taken to be $\delta \mid \alpha$ (α is exactly divisible by δ) where α is a term but δ must be an integer. This relation can be defined as $(\exists x)(\alpha=\delta * x)$, assuming x does not occur free in α.

THE PRESENT ALGORITHM

Decision algorithms for this system are well known – see, for example, Hilbert and Bernays (1968), or Kreisel and Krivine (1967). These are based on the method known as 'elimination of quantifiers'. Clearly if, given a formula $(\exists x)F$ where F is quantifier free, we can construct an equivalent quantifier-free formula G then we have a decision algorithm: first, close the formula by universally quantifying all free variables, then eliminate quantifiers from the inside thus obtaining a formula without variables which may be evaluated to *true* or *false*. Universal quantifiers may be replaced by existential quantifiers – use $(\forall x)F \equiv \neg(\exists x)\neg F$ – and our basic task is therefore the elimination of a quantifier from a formula of the form $(\exists x)F$ where F is quantifier free but may involve x and other variables. This may be accomplished as follows:

Step 1

Transform F to disjunctive normal form and distribute the quantifier over the disjuncts. As a result we have a number of separate eliminations to perform in each of which F is a conjunct of relations or the negation of relations.

Step 2

Eliminate negation by using $\neg\alpha<\beta$ is $\beta<\alpha+1$ and $\neg(\delta \mid \alpha)$ is $\bigvee_{i=1}^{\delta-1} \delta \mid \alpha+i$.

Step 3

Simplify each relation by collecting the x-terms so that each relation either does not involve x (in which case take it outside the quantifier) or is one of

$$\lambda_i x < \alpha_i$$
$$\beta_i < \mu_i x$$
$$\delta_i | \nu_i x + \gamma_i,$$

where λ_i, μ_i, ν_i and δ_i are positive integers and α_i, β_i, γ_i are expressions not involving x.

Step 4

Let δ be the L.C.M. of all the λ_i, μ_i and ν_i. By multiplying both sides of all relations by appropriate constants the coefficient of all x's may be made δ. Now replace $(\exists x)F(\delta x)$ by $(\exists x)(F(x) \wedge \delta | x)$. The result will be a conjunct of terms as at the end of Step 3 except that now the coefficients of all x's are unity.

Step 5

The elimination may now be performed by using a generalisation of the equivalence:

$$(\exists x)(\alpha < x \wedge x < \beta \wedge \delta | x) \equiv \bigvee_{j=1}^{\delta} (\alpha + j < \beta \wedge \delta | \alpha + j).$$

This generalisation is given in full in Cooper (1971) but it will not be given here as the new algorithm is both more efficient and more succinct.

FORMULA EXPANSIONS

There are two main sources of expansion which make the above algorithm unworkable on any but the simplest formulas. First, there is the initial transformation to disjunctive normal form. In some applications this might not be troublesome; however, in the application of Cooper (1971) the formula produced for the algorithm tended to be long but repetitious, with many common sub-expressions. The simplest formula (about 100 characters long) had 55 disjuncts. As the formula was not valid every separate disjunct had to be proved invalid; the variations were primarily in parts of the disjuncts not contributing to the inconsistency, so much work was repeated. A computer program for the algorithm was able to solve this problem, but could not tackle the other formula (up to about 1000 characters long) in any reasonable time because of the very large number of disjuncts. The major feature of the new algorithm is that it is not necessary to have the formula in disjunctive normal form and a program for this new algorithm quickly disposed of all cases.

The second source of expansion occurs in Steps 2 and 5 where new disjuncts are produced whose number of terms depends on the integers occurring in the formula. This can clearly be disastrous, yet in the old algorithm seems to be inevitable. The expansion in Step 2 is avoided altogether in the new algorithm; that in Step 5 remains, but remarks will be made later showing

how the effect can be greatly reduced. It should also be noted that in an interesting sub-class of formulas (essentially those in which one only has the successor function, not full addition) the δ of the previous formula is always 1 and this expansion does not occur – this was the case in the formulas occurring in Cooper (1971).

THE NEW ALGORITHM

A new process will now be described for the elimination of an existential quantifier which does not assume the formula is in disjunctive normal form.

Steps 2, 3 and 4 of the previous algorithm did not use the fact that the formula was in disjunctive normal form, and these are again performed. They will not produce any major expansion, apart from the replacement of $\neg(\delta \mid \alpha)$ in Step 2, and to avoid this we introduce $\nmid$ as a basic relation – $\delta \nmid \alpha$ means that α is not exactly divisible by δ.

The formula is now of the form $(\exists x)F(x)$ where $F(x)$ is formed by conjunction and disjunction (note no negation) of basic relations each of which is one of the forms:

(A) $x < a_i$
(B) $b_i < x$
(C) $\delta_i \mid x + c_i$
(D) $\varepsilon_i \nmid x + d_i$,

where a_i, b_i, c_i and d_i are expressions not involving x, and δ_i, ε_i are positive integers.

Let δ be the L.C.M. of all δ_i, ε_i and define $F_{-\infty}(x)$ to be $F(x)$ with *true* substituted for all formulas of type (A) and *false* substituted for all formulas of type (B). Clearly, we have:

Lemma. For x sufficiently small $F(x) \equiv F_{-\infty}(x)$.

The following theorem then enables the elimination of the quantifier:

Theorem. With notation and restrictions as above

$$(\exists x)F(x) \equiv \bigvee_{j=1}^{\delta} F_{-\infty}(j) \vee \bigvee_{j=1}^{\delta} \bigvee_{b_i} F(b_i + j)$$

(the second term on the right hand side is omitted if there are no formulas of type (B)).

Proof of theorem. Assume the right hand side is true. Then one of the disjuncts must be true; if it is one of the second term then an x has been found; if $F_{-\infty}(j)$ is true for some j then, by the definition of $F_{-\infty}$, $F_{-\infty}(j) \equiv F_{-\infty}(j - \lambda\delta)$ for any integer λ, and if λ is sufficiently large then $F_{-\infty}(j - \lambda\delta) \equiv F(j - \lambda\delta)$ by the lemma. In either case the left-hand side is true.

Assume the left-hand side is true and let x_0 be such that $F(x_0)$ is true.

Suppose x_0 is of the form $b_i + j$ for some b_i and for some j with $1 \leqslant j \leqslant \delta$. Then one of the disjuncts of the second term on the right-hand side is true, hence the right-hand side is true.

Suppose x_0 is not of that form, consider $F(x_0 - \delta)$. If $F(x_0 - \delta)$ is false,

but $F(x_0)$ is true, then at least one basic relation in F of one of the four types must change from true to false as x_0 is changed to $x_0-\delta$ (remember that F does not involve the negation operator). But this is clearly impossible for relations of types (A), (C) and (D), and moreover could only happen in a relation of type (B) if $b_i<x_0$ and $\neg b_i<x_0-\delta$. This latter implies x_0 is of the form b_i+j with $1\leqslant j\leqslant\delta$, contrary to hypothesis for this case.

We can therefore assume $F(x_0-\delta)$ is true. This argument can be repeated with $x_0-\delta$ replacing x_0 until either we find an x of the form b_i+j (making one of the second set of disjuncts true) or until we have an x so small that $F(x)\equiv F_{-\infty}(x)$. By adding a multiple of δ we will prove one of the first set of disjuncts true.

The theorem is now proved and may be used in order to eliminate the quantifier.

PRACTICAL REMARKS ON THE ALGORITHM

If arithmetic relations other than $<$ occur in the formula it is not necessary to formally eliminate them. The only point of interest in applying the theorem is to determine what 'b_i' would occur if the elimination were to be performed. Similarly it is not necessary to eliminate negation: for each relation we determine how many negations control the relation, and if this is an odd number consider the negation of the relation in determining the b_i. Further, instead of decreasing x by δ in the proof, we could increase it, thus obtaining the theorem

$$(\exists x)F(x)\equiv\bigvee_{j=1}^{\delta}F_{\infty}(-j)\vee\bigvee_{j=1}^{\delta}\bigvee_{a_i}F(a_i-j)\quad,$$

where $F_{\infty}(x)$ is $F(x)$ with *false* substituted for formulas of type (A) and *true* for formulas of type (B). Clearly, if there are less 'a_i' than 'b_i' one should use this second approach. These remarks can be summed up as in table 1. The 'A set' and the 'B set' are merely the 'a_i' and the 'b_i' from the formulas of type (A) and type (B), respectively.

Table 1

		positive		*negative*	
	relation	A set	B set	A set	B set
1	$x<\alpha$	α	–	–	$\alpha-1$
2	$\alpha<x$	–	α	$\alpha+1$	–
3	$x\leqslant\alpha$	$\alpha+1$	–	–	α
4	$\alpha\leqslant x$	–	$\alpha-1$	α	–
5	$x=\alpha$	$\alpha+1$	$\alpha-1$	α	α
6	$x\neq\alpha$	α	α	$\alpha+1$	$\alpha-1$
7	$\delta\mid x+\alpha$	–	–	–	–
8	$\delta\nmid x+\alpha$	–	–	–	–

The enlarged algorithm to eliminate the existential quantifier from $(\exists x)F$ becomes

1. Perform Steps 3 and 4 of the old algorithm.
2. For each relation occurring in the new F determine whether it occurs positively or negatively (i.e. whether if the formula were to be put into disjunctive normal form it would occur with a negation sign). Add the appropriate terms to the A set and the B set as indicated in table 1, a dash indicates no addition has to be made.
3. Depending on whether the A set or the B set is smaller use the appropriate formula:

$$(\exists x)F(x) \equiv \bigvee_{j=1}^{\delta} F_{-\infty}(j) \vee \bigvee_{j=1}^{\delta} \bigvee_{\beta \in B} F(\beta + j)$$

or

$$(\exists x)F(x) \equiv \bigvee_{j=1}^{\delta} F_{\infty}(-j) \vee \bigvee_{j=1}^{\delta} \bigvee_{\alpha \in A} F(\alpha - j) \quad .$$

If the arithmetic relations are not eliminated the definitions of $F_{-\infty}$ and F_{∞} must be extended in the obvious way – $F_{-\infty}$ is obtained from F by substituting *true, false, true, false, false, true* for formulas of type 1 to 6 in table 1, F_{∞} is obtained from F by substituting *false, true, false, true, false, true* for formulas of type 1 to 6.

Two further remarks. Whilst it can be disastrous to transform the formula into disjunctive normal form, yet it is advantageous to distribute existential quantifiers over disjuncts wherever possible, as this can lead to sub-problems with smaller values of δ and smaller sizes of A sets and B sets. Also, advantage should be taken of equalities by substituting $x = \alpha \wedge P(\alpha)$ for $x = \alpha \wedge P(x)$ wherever it occurs, assuming x is the bound variable being eliminated.

THE δ EXPANSION

The effect of this can be considerably lessened, at least in an important subset of formulas. Assume that the closed formula being tested for validity is in prenex normal form and all the quantifiers are the same, either all existential or all universal. This includes the interesting case of a formula with free variables but no quantifiers, as its closure will have all universal quantifiers at the front. If the quantifiers are universal, the transformation to existential quantifiers will produce a string of existential quantifiers (preceded by a negation), and so in either case the quantifiers have to be eliminated from a formula of the type

$$(\exists x_1)(\exists x_2) \ldots (\exists x_n)F(x_1, x_2, \ldots, x_n) \quad .$$

The steps in the algorithm depend only on the form of the basic relations (assuming negations have been eliminated) and not at all on the form of F. Let us therefore rewrite the formula in the form

$$(\exists x_1)(\exists x_2) \ldots (\exists x_n)F(R_1, R_2, \ldots, R_m) \quad ,$$

where $R_1, \ldots, R_m$ are arithmetical relations involving the variables $x_1, \ldots, x_n$ and F contains no negations. Each elimination produces an operator of the form $\bigvee_{i=1}^{\delta}$, and as this commutes with the existential quantifier we may carry on the procedure without formally expanding this disjunct by merely preserving i as a variable. Let us illustrate this process on a simple example with $n=2$ and $m=3$. *True* and *false* will be denoted by t and f. Starting with:

$$(\exists y)(\exists x)F(x+5y>1,\ 13x-y>1,\ x+2<0) \quad ,$$

replace $13x$ by x:

$$(\exists y)(\exists x)[F(x>13-65y,\ x>y+1,\ x<-26)\wedge 13|x] \quad .$$

Use the A set to eliminate x:

$$\bigvee_{i=1}^{13}(\exists y)[F(t,t,f)\wedge 13|-i] \quad \vee$$

$$\bigvee_{i=1}^{13}(\exists y)[F(65y>39+i,\ y<-27-i,\ i>0)\wedge 13|-26-i] \quad .$$

Replace $65y$ by y in the second disjunct:

$$\bigvee_{i=1}^{13}(\exists y)[F(t,t,f)\wedge 13|-i] \quad \vee$$

$$\bigvee_{i=1}^{13}(\exists y)[F(y>39+i,\ y<-1755-65i,\ i>0)\wedge 13|-26-i\wedge 65|y].$$

Trivially eliminate the quantifier from the first disjunct and use the B set in the second disjunct to obtain finally:

$$\bigvee_{i=1}^{13}[F(t,t,f)\wedge 13|-i] \quad \vee$$

$$\bigvee_{i=1}^{13}\bigvee_{j=1}^{65}[F(f,t,i>0)\wedge 13|-26-i\wedge 65|j] \quad \vee$$

$$\bigvee_{i=1}^{13}\bigvee_{j=1}^{65}[F(j>0{,}66i+j<-1794,i>0)\wedge 13|-26-i\wedge 65|39+i+j].$$

This process is quite general and, starting with

$$(\exists x_1)(\exists x_2)\ldots(\exists x_n)F(R_1, R_2, \ldots, R_m) \quad ,$$

we shall obtain the disjunction of a number of terms each of which is of the form:

$$\bigvee_{i_1=1}^{\delta_1}\bigvee_{i_2=1}^{\delta_2}\ldots\bigvee_{i_n=1}^{\delta_n}[F(S_1, S_2, \ldots, S_m)\wedge\lambda_1|a_1\wedge\lambda_2|a_2\wedge\ldots\wedge\lambda_n|a_n] \quad ,$$

where $S_1, \ldots, S_m$ are arithmetical relations, $a_1, \ldots, a_n$ are algebraic expressions, and $\lambda_1, \ldots, \lambda_n$ are positive integers – all relations and expressions

being within Presburger arithmetic and involving only integers and the variables $i_1, \ldots, i_n$.

This process only involves a moderate expansion which depends on the size of the A and B sets.

In order to evaluate each of these logical expressions it is not necessary to expand the δ disjuncts. In the appendix we show how to find the sets of values of $i_1, \ldots, i_n$ which satisfy the expression $\lambda_1|a_1 \wedge \ldots \wedge \lambda_n|a_n$; the expansion is then only done over these sets.

Applying this process to the previous examples (in this simple case the answer can be seen immediately) we get our final form:

$$F(t,t,f) \vee F(f,t,t) \vee F(t,f,t) \quad .$$

The original expression is therefore equivalent to this one, only assuming F contains no negations; moreover no drastic expansions occur in obtaining this result. Further simplification is often possible, although not in this example. As F contains no negations we have

$$F(\ldots,f,\ldots) \rightarrow F(\ldots,t,\ldots) \quad ,$$

and therefore, for example,

$$F(f,f,t) \vee F(f,t,t) \quad ,$$

can be replaced by

$$F(f,t,t) \quad .$$

In particular, if $F(t,t,t)$ ever occurs, the process may be stopped with $F(t,t,t)$ as the answer.

If the prenex normal form contains mixed quantifiers then the process is more complicated as, for example, $(\forall y)(\exists x)F$ will become, after elimination of x and transformation of the first quantifier, the disjunct of terms of the form $\neg(\exists y) \bigwedge_{i=1}^{\delta} \neg F$.

There is now no commuting of operators and the A and B sets will in general contain a number of members with i as a variable thus making the size of the sets depend on δ. Hand calculation on simple examples suggests that the method may well still work, but as yet we do not have a simple way to express the algorithm.

CONCLUDING REMARKS

One way to compare the new algorithm with the old is to regard the original transformation to disjunctive normal form method as first finding which sets of truth values of the relations make F true and then testing these required sets of values to determine if they are consistent; on the other hand, the new transformation first finds by direct calculation which sets of truth values for the relations are consistent, and then evaluates F with these consistent values to see if any make F itself true. With examples of the kind which occurred in Cooper (1971) the second method proves much more efficient, as well as being simpler to state and understand. Moreover, the independence of the process on the form of F could lead to an advantage if we had ex-

amples with the same relations but different F's, as the above process need only be performed once.

APPENDIX

We wish to find the sets of values of $i_1, \ldots, i_n$ within certain bounds which make true:

$$\bigwedge_{j=1}^{n} \lambda_j \Big| \sum_{k=1}^{n} \mu_{jk} i_k + \nu_j \quad ,$$

where μ_{jk}, ν_j are integers and λ_j are positive integers. This can be done by a method similar to Gaussian elimination using the two easily proved theorems from the theory of numbers:

Theorem 1

$m|ax+b \wedge n|\alpha x+\beta$ if and only if
$mn|dx+bpn+\beta qm \wedge d|\alpha b-a\beta$
where $d=GCD(an, \alpha m)$ and $pan+q\alpha m=d$

(p and q are found as a by-product if Euclid's algorithm is used to find the GCD).

Theorem 2

$m|ax+b$ has solutions for x if and only if
$d|b$, the solutions are $x=-p(b/d)+t(m/d)$ for all integers t
where $d=GCD(a, m)$ and $d=pa+qm$.

The sets of solutions are found by first using Theorem 1 to reduce the set of divisibility relations to triangular form and then using Theorem 2 and the known bounds on the variables to obtain all the solutions by a back substitution process.

REFERENCES

Cooper, D.C. (1971) Programs for Mechanical Program Verification. *Machine Intelligence* 6, pp. 43–59 (eds Meltzer, B. & Michie, D.). Edinburgh: Edinburgh University Press.

Cooper, W.S. (1964) Fact Retrieval and Deductive Question-answering information retrieval systems. *J. Ass. Comput. Mach.*, **11**, pp. 117–37.

Hilbert, D. & Bernays, P. (1968) *Grundlagen der Mathematik I* (Zweite Auflage) pp. 366–75. Berlin, Heidelberg, New York: Springer-Verlag.

Kreisel, G. & Krivine, J. L. (1967). *Elements of Mathematical Logic*, pp. 54–7. Amsterdam: North-Holland.

McCarthy, J. (1958) Programs with common sense. *Mechanization of Thought Processes vol. I*, pp. 77–84. Proc. Symp. Nat. Phys. Lab., London.

Paterson, M. (1972) Decision problems in computational models. *Proc. ACM conference on proving assertions about programs*, pp. 74–82. Los Cruces, New Mexico.

Presburger, M. (1929) Uber die Vollstandigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. *Comptes-Rendus du I Congres de Mathematiciens des pays Slaves*, pp. 92–101, 395. Warsaw.

6

The Sharing of Structure in Theorem-proving Programs

R. S. Boyer and J S. Moore
Department of Computational Logic
School of Artificial Intelligence, Edinburgh

Abstract

We describe how clauses in resolution programs can be represented and used without applying substitutions or cons-ing lists of literals. The amount of space required by our representation of a clause is independent of the number of literals in the clause and the depth of function nesting. We introduce the concept of the value of an expression in a binding environment which we use to standardize clauses apart and share the structure of parents in representing the resolvent. We present unification and resolution algorithms for our representation. Some data comparing our representation to more conventional ones is given.

1. INTRODUCTION

In this paper we are concerned with representing literals and clauses in computers. Lists provide the most obvious and natural representation of literals because lists perfectly reflect function nesting structure. A list is also a reasonable representation of a set, in particular of a clause. Lists, however, can consume large amounts of space, and cause frequent garbage collections. We shall present in this paper a representation of clauses and literals which is as natural as lists but far more compact. We achieve this economy by sharing the structure of the parents of a resolvent in our representation of the resolvent.

A clause is a set of literals; but throughout this paper we shall speak of the literals of a clause as having an order. That is, we shall speak of the first, second, etc., literal of a clause.

Suppose C and D are clauses and K is the ith literal of C and L is the jth literal of D. Suppose further that the signs of K and L are opposite. Finally, suppose that the substitution σ most generally unifies the atoms of K and L.

Under these hypotheses, we may resolve C and D on K and L to obtain

the resolvent R=((C−{K})∪(D−{L}))σ. We think of the literals in R that come from C as being 'before' the literals that come from D. (We ignore merging and factoring until section 8.)

The tuple T=⟨C, *i*, D, *j*, σ⟩ contains sufficient information to enable one to construct the resolvent R. Therefore, in some sense, T represents R. At first sight T does not appear to be a very good way to represent R; for it seems that the only way to use T is to construct a list consisting of all but the *i*th literal of C and *j*th literal of D and then to apply σ to the list.

We shall show in this paper that it is possible, in fact easy, to use a tuple like T without constructing any lists or applying any substitutions. Actually, the tuples we shall use have the form

⟨C, *i*, D, *j*, NL, MI, σ⟩.

NL is simply the number of literals in the resolvent, that is, the sum of the number of literals in C and D minus 2. MI is a number which helps us standardize clauses apart. The MI of an input clause is 1; the MI of a resolvent is the sum of the MI's of the parents. ('MI' is mnemonic for 'maximum index'.)

2. TERMS AND SUBSTITUTIONS

To understand how to avoid applying substitutions, it is first necessary to understand the concept of the value of a term in the context of a substitution. First, an example:

The value of the term

$$(P\,x\,(f\,y\,(g\,z\,x)))$$

in the context of the substitution

$$((y\,.\,(f\,v\,w))\,(z\,.\,(g\,x\,u))\,(u\,.\,(h\,x)))$$

is the term

$$(P\,x\,(f\,(f\,v\,w)\,(g\,(g\,x\,(h\,x))\,x))).$$

By a term, we mean either a variable (e.g. *x*,*y*,*z*) or a list whose first member is a symbol (e.g. *f*,*g*,*P*,*Q*) and whose other members are terms.*

By a substitution we mean a collection of pairs; the first member of each pair is a variable and the second member is a term. If (VAR . TERMB) is a member of a substitution *S*, we say that VAR is bound in *S* to TERMB.

By the value of a term T in the context of a substitution *S*, we mean the result of replacing each variable in T that is bound in *S* to a term TERMB by the value of TERMB in *S*.

Our definitions are not those standard to the theorem-proving literature. For example, we do not need to distinguish between predicate and function symbols. Furthermore, there exist substitutions *S* such that some terms have no value in the context of *S*. We take precautions never to generate such substitutions. Roughly speaking, a variable ought not be bound twice or bound to something whose value contains that variable.

It is possible to determine anything about the value of a term in the context

* We think of a term such as (*a*) as a constant.

of a substitution S without physically creating the value. The only thing one must do is:

> Whenever one encounters a variable VAR, check whether VAR is bound to some term TERMB. If VAR is bound, proceed as if one had encountered TERMB instead of VAR.

For example, suppose we wish to determine whether some variable V occurs in the value of a term TERM in the context of a substitution S. We define the recursive function OCCUR:

Definition of OCCUR(V,TERM)

If TERM is a variable, then

If TERM is bound to TERMB in S, return(OCCUR(V,TERMB))

Otherwise, if V=TERM, return(true)

Otherwise return(false)

Otherwise TERM is not a variable and has the form $(f\ T_1 \ldots T_n)$.

If any call of OCCUR(V,T_i) returns true, then return(true)

Otherwise return(false)

End of definition.

Notice that we check to see if we have encountered a variable that is bound to some term TERMB in S (S is global to OCCUR). If it is, we proceed as if we had encountered TERMB instead of the variable by returning the result of the recursive call OCCUR(V,TERMB).

By avoiding the application of substitutions to terms it is possible to achieve a dramatic saving in space, which, of course, one pays for by looking-up the bindings of variables. That this is worth while is demonstrated by the successful use of similar methods to 'substitute' values for the formal parameters in LISP and ALGOL function calls.

3. EXPRESSIONS AND BINDINGS

The key to our representation of clauses is the avoidance of physically creating the value of a term in the context of a substitution. This idea is at least as old as the first LISP. Terms and substitutions, however, are not quite sufficient for our purposes because we often need to refer to different versions of a term at one time. Therefore, we introduce the concepts of an expression, a binding environment, and the value of an expression in a binding environment. First some examples:

The value of the expression

$$(P\,x\,(f\,y\,(g\,z\,x))),10$$

in the empty binding environment is the term

$$(P\,x_{10}\,(f\,y_{10}\,(g\,z_{10}\,x_{10}))).$$

The value of the expression

$$(P\,x\,(f\,y\,(g\,z\,x))),5$$

in the empty binding environment is the term

$$(P\,x_5\,(f\,y_5\,(g\,z_5\,x_5))).$$

Notice that these two values have no variables in common.

The value of the expression

$$(P\,x\,(f\,y\,(g\,z\,x))),5$$

in the binding environment

$$((y,5\,.\,(f\,x\,y),4)\;(z,5\,.\,(g\,x\,u),5)\;(u,5\,.\,(h\,x),5))$$

is the term

$$(P\,x_5\,(f\,(f\,x_4\,y_4)\,(g\,(g\,x_5\,(h\,x_5))\,x_5))).$$

By an index we mean a positive integer. By an expression we mean a term together with an index. If we denote an expression by T,I then T is a term and I is an index.

By a binding we mean a pair (VAR,INDEX . TERMB,INDEXB) where VAR is a variable, TERMB is a term, and INDEX and INDEXB are indices.

By a binding environment we mean a collection of bindings. If (VAR, INDEX . TERMB,INDEXB) is a member of the binding environment BNDEV, we say that VAR,INDEX is bound in BNDEV to TERMB,INDEXB.

The value of an expression T,I in a binding environment BNDEV is the result of replacing each variable V in T by the value of V,I in BNDEV. If V,I is not bound in BNDEV, its value is the variable V_I (i.e. V subscript I). If V,I is bound to TERMB,INDEXB in BNDEV, then its value is the value of TERMB, INDEXB in BNDEV.

It is possible to determine anything about the value of an expression in a binding environment without physically creating the value.

Throughout this paper we shall use two procedures, ISBOUND and BIND, to facilitate the handling of binding environments. ISBOUND(VAR,INDEX, BNDEV) returns true if VAR,INDEX is bound in BNDEV, and false otherwise. If true is returned, then the global variables TERMB and INDEXB will have been so set that VAR,INDEX is bound to TERMB,INDEXB in BNDEV. BIND(V,I,T,J,BNDEV) so alters the binding environment BNDEV that V,I is then bound to T,J in BNDEV.

In the next three sections we shall display binding environments as lists of bindings. We do this to help introduce our representation of clauses intuitively. The actual structure of a binding environment is made precise in section 7. The only essential feature of a binding environment is that one can discover bindings with ISBOUND and add bindings with BIND.

Suppose we wish to determine whether some variable V_I occurs in the value of the expression TERM,J in the binding environment BNDEV. We define the recursive function OCCUR as below and call OCCUR(V,I,TERM,J):

Definition of OCCUR(V,I,TERM,J)

If TERM is a variable, then

If ISBOUND(TERM,J,BNDEV) then return(OCCUR(V,I,TERMB, INDEXB))

Otherwise if V=TERM and I=J, return(true)

Otherwise return(false)

Otherwise TERM is not a variable and has the form $(f\ T_1 \ldots T_n)$.
If any call of OCCUR(V,I,T_i,J) returns true, then return(true)
Otherwise return(false)
End of definition.

Observe the similarity between this definition and the previous definition of OCCUR. BNDEV is global to OCCUR.

4. UNIFY: OUR UNIFICATION ALGORITHM

Suppose that VAL1 is the value of the expression TERM1,INDEX1 in the binding environment BNDEV. Suppose further that VAL2 is the value of TERM2,INDEX2 in BNDEV. Finally suppose that VAL is the most general common instance of VAL1 and VAL2. If we call UNIFY(TERM1,INDEX1, TERM2,INDEX2) then BNDEV will be altered during the call so that the value of TERM1,INDEX1 in BNDEV and the value of TERM2,INDEX2 in BNDEV are both equal to VAL. If VAL1 and VAL2 have no common instance, then the call will return false. Our procedure UNIFY, like the procedure OCCUR of the previous section which UNIFY uses, applies no substitutions. We write $x=y$ if x and y are the same atom or number. By EQUAL we mean the LISP EQUAL.

Definition of UNIFY(TERM1,INDEX1,TERM2,INDEX2)
If EQUAL(TERM1,TERM2) and INDEX1 =INDEX2 then return(true)
Otherwise if TERM1 is a variable, then
If ISBOUND(TERM1,INDEX1,BNDEV) then return(UNIFY (TERMB,INDEXB,TERM2,INDEX2))
Otherwise if OCCUR(TERM1,INDEX1,TERM2,INDEX2) then return(false)
Otherwise BIND(TERM1,INDEX1,TERM2,INDEX2,BNDEV) and return(true)
Otherwise if TERM2 is a variable, then return(UNIFY(TERM2,INDEX2, TERM1,INDEX1))
Otherwise, since neither TERM1 nor TERM2 is a variable, TERM1 has the form $(f\ T_1 \ldots T_n)$ and TERM2 has the form $(g\ S_1 \ldots S_m)$.
If $f \neq g$, then return(false)
Otherwise if every call of UNIFY(T_i,INDEX1,S_i,INDEX2) returns true, return(true)
Otherwise return(false)
End of definition.

Here is an example of unification. Let TERM1 be $(P\ x\ y)$. Let TERM2 be $(P\ (g\ x)\ z)$. Let BNDEV be

$((x, 2\ .\ x, 3)$
$(y, 2\ .\ (f\ x\ y), 4)$
$(y, 4\ .\ x, 3)$
$(z, 7\ .\ (f\ x\ y), 8)$
$(x, 8\ .\ x, 7)$
$(y, 8\ .\ (g\ y), 5))$.

The value of TERM1,2 in BNDEV is $(P\ x_3\ (f\ x_4\ x_3))$.
The value of TERM2,7 in BNDEV is $(P\ (g\ x_7)\ (f\ x_7\ (g\ y_5)))$.

After a call of UNIFY(TERM1,2,TERM2,7),BNDEV is

$$\left.\begin{array}{l}((x, 4\ .\ y, 5)\\ (x, 7\ .\ x, 4)\\ (x, 3\ .\ (g\ x), 7)\end{array}\right\}\ \text{added by UNIFY}$$

$$\left.\begin{array}{l}(x, 2\ .\ x, 3)\\ (y, 2\ .\ (f\ x\ y), 4)\\ (y, 4\ .\ x, 3)\\ (z, 7\ .\ (f\ x\ y), 8)\\ (x, 8\ .\ x, 7)\\ (y, 8\ .\ (g\ y), 5)).\end{array}\right\}\ \text{the old BNDEV}$$

The value of TERM1,2 in the new BNDEV is $(P\ (g\ y_5)\ (f\ y_5\ (g\ y_5)))$.
The value of TERM2,7 in the new BNDEV is $(P\ (g\ y_5)\ (f\ y_5\ (g\ y_5)))$.

5. INCREMENTING INDICES: HOW TO STANDARDIZE EXPRESSIONS APART

Let T be the term $(Q\ (f x(a))\ (g\ y\ z))$. The value of the expression T,5 in the binding environment

BNDEV1: $((x, 5\ .\ (g\ y\ z), 5)$
$(z, 5\ .\ (f(a)\ u), 6)$
$(u, 6\ .\ x, 3))$

is $(Q\ (f(g\ y_5\ (f(a)\ x_3))\ (a))\ (g\ y_5\ (f(a)\ x_3)))$. The value of the expression T,11 in the binding environment

BNDEV2: $((x, 11\ .\ (g\ y\ z), 11)$
$(z, 11\ .\ (f(a)\ u), 12)$
$(u, 12\ .\ x, 9))$

is $(Q\ (f(g\ y_{11}\ (f(a)\ x_9))\ (a))\ (g\ y_{11}\ (f(a)\ x_9)))$.

Notice that the value of T,5 in BNDEV1 is a variant of the value of T,11 in BNDEV2; furthermore, the values have no variable in common, that is, they have been standardized apart. Notice that BNDEV2 is obtained from BNDEV1 by adding the increment 6 to every index in BNDEV1.

Suppose that T is a term, BNDEV1 is a binding environment, and BNDEV2 is obtained from BNDEV1 by adding the increment INC to every index in BNDEV1. Then the value of T,J in BNDEV1 is a variant of the value of T,J+INC in BNDEV2. If INC is greater than any index in BNDEV1 then the two values have no variable in common.

6. RESOLVING CLAUSES USING EXPRESSIONS AND BINDINGS

In this section we describe by example how expressions, incrementing indices, and our unification procedure work together in resolution. We use in this section a simple representation of clauses, namely a list of expressions in a

binding environment. After we have performed one resolution using this representation, we come to the main point of the paper.

The list

C1: $((+(Q\,y\,y)),2$
$(+(P\,x\,y)),2$
$(-(P\,x\,(f\,y\,z))),4)$

in the binding environment

B1: $((x,2\,.\,x,3)$
$(y,2\,.\,(f\,x\,y),4)$
$(y,4\,.\,x,3)$
$(z,4\,.\,(f\,x\,y),2))$

represents the clause

$\mathscr{C}1$: $(Q\,(f\,x_4\,x_3)\,(f\,x_4\,x_3))\,(P\,x_3\,(f\,x_4\,x_3))-(P\,x_4\,(f\,x_3\,(f\,x_3\,(f\,x_4\,x_3))))$.

in an obvious way.

Similarly, the list

C2: $((-(Q\,x\,y)),1$
$(-(P\,(g\,x)\,z)),3$
$(+(R\,x\,(f\,x\,y)),4)$

in the binding environment

B2: $((z,3\,.\,(f\,x\,y),4)$
$(x,4\,.\,x,3)$
$(y,4\,.\,(g\,y),1)$
$(y,2\,.\,(g\,z),3))$

represents the clause

$\mathscr{C}2$: $-(Q\,x_1\,y_1)-(P\,(g\,x_3)\,(f\,x_3\,(g\,y_1)))\,(R\,x_3\,(f\,x_3\,(g\,y_1)))$.

To resolve $\mathscr{C}1$ and $\mathscr{C}2$ on their second literals, we first standardize $\mathscr{C}1$ and $\mathscr{C}2$ apart. We do this by adding 4 to every index in C2 (we denote the result as C2′) and by adding 4 to every index in B2 (we denote the result by B2′). We add 4 because it is the maximum index in C1 or B1. C2′ in the binding environment B2′ represents the clause

$\mathscr{C}2'$: $-(Q\,x_5\,y_5)-(P\,(g\,x_7)\,(f\,x_7\,(g\,y_5)))\,(R\,x_7\,(f\,x_7\,(g\,y_5)))$.

$\mathscr{C}2'$ is a variant of $\mathscr{C}2$ and has no variables in common with $\mathscr{C}1$.

We obtain the second expression $(+(P\,x\,y)),2$ of C1 and the second expression $(-(P\,(g\,x)\,z)),7$ of C2′. We check that their signs are opposite. We then call UNIFY$((P\,x\,y),2,(P\,(g\,x)\,z),7)$. Of course UNIFY requires a binding environment. In this case BNDEV is originally set to B1∪B2′. The call to UNIFY returns true and BNDEV has been modified so that it is

$(x,4\,.\,y,5)$
$(x,7\,.\,x,4)$
$(x,3\,.\,(g\,x),7)$
} added by UNIFY

$(x,2\,.\,x,3)$
$(y,2\,.\,(f\,x\,y),4)$
$(y,4\,.\,x,3)$
$(z,4\,.\,(f\,x\,y),2)$
} B1

$$\left.\begin{array}{l}(z, 7 . (f\,x\,y), 8)\\ (x, 8 . x, 7)\\ (y, 8 . (g\,y), 5)\\ (y, 6 . (g\,z), 7)).\end{array}\right\} \text{B2}'$$

The resolvent, $\mathscr{R}$, could be represented by a list of expressions R in the binding environment BNDEV above. R is obtained by appending C1 and C2′ after removing their second literals.

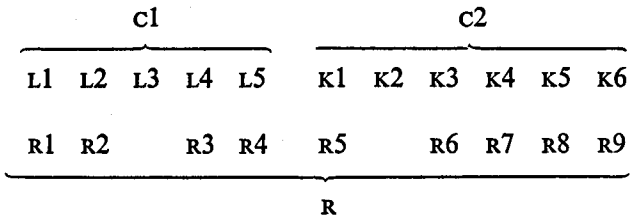
R: $((+(Q\,y\,y)), 2$
$(-(P\,x\,(f\,y\,z))), 4$
$(-(Q\,x\,y)), 5$
$(+(R\,x\,(f\,x\,y))), 8).$

R in the binding environment BNDEV represents the resolvent

$\mathscr{R}$: $(Q\,(f\,y_5\,(g\,y_5))\,(f\,y_5\,(g\,y_5)))$
$-(P\,y_5\,(f\,(g\,y_5)\,(f\,(g\,y_5)\,(f\,y_5\,(g\,y_5)))))$
$-(Q\,x_5\,y_5)$
$(R\,y_5\,(f\,y_5\,(g\,y_5))).$

We now come to the central part of this paper. It should be obvious that it is exceedingly wasteful to create physically the lists C2′ and R and the binding environments B2′ and BNDEV, given their definitions in terms of C1, B1, C2, and B2. We certainly do not physically create any of C2′, R, B2′, or BNDEV. We do represent a clause so that we can easily retrieve (1) the expression for the *n*th literal and (2) the binding of V,I (if it is bound). Under the hypothesis that we can retrieve (1) and (2) for C1 and C2, our clause record contains precisely enough information to retrieve easily (1) and (2) for R.

(1) Assume, inductively, that it is possible to retrieve the expression for the *n*th literal of either parent, C1 or C2, of a resolvent R. The expression for the *n*th literal of R is either the expression for the *j*th literal of C1 or the expression for the *j*th literal of C2 with its index incremented by the maximum index (MI) of C1. *j* depends only upon the number of literals in C1 and the numbers of the literals in C1 and C2 upon which we resolved. The following picture should make it obvious how to compute *j*. We are resolving on L3 and K2.



(2) Assume, again inductively, that it is possible to determine whether V,I is bound to TERMB,INDEXB in the binding environments of either parent. If in the representation of a resolvent we include the bindings added by the unification of the resolution, it is possible to determine if V,I is bound to

TERMB,INDEXB in the binding environment of R. In particular, V,I is bound to TERMB,INDEXB in R if and only if

I≤MI and V,I is bound to TERMB,INDEXB in the binding environment of C1, or

I>MI and V,I−MI is bound to TERMB,INDEXB−MI in the binding environment of C2, or

V,I was bound to TERMB,INDEXB in the unification made for the resolvent.

The main point of this paper is:

If we can compute the expressions and binding environments for input clauses, then we can compute them for derived clauses if we include only the following information in the record of such a resolvent:

1. the record of the left parent, C1
2. the number of the literal resolved upon in C1
3. the record of the right parent, C2
4. the number of the literal resolved upon in C2
5. the number of literals in the clause resolvent
6. the maximum index of the resolvent
7. the bindings added during the unification for the resolvent.

This is precisely the information we include in our clause representation described in detail in the next section.

7. THE DETAILS OF OUR REPRESENTATION

By a clause record we mean either an input record or a resolvent record. By an input record, we mean a list of literals. A literal has a sign which may be + or − followed by an atomic formula which is a term in the sense of section 2. Here are two input records:

((+(*P x* (*f y*))))

((−(*Q x y*)) (+(*P* (*f x*) *y*)) (+(*R* (*a*) *z*)))

If IP is an input record, then NUMBEROFLITERALS(IP)IN is the length of IP, and MAXIMUMINDEX(IP) is 1.

By a resolvent record, R, we mean a structure of 7 components:

1. a clause record which we access as LEFTPARENT(R)
2. an integer which we access as LEFTLITERALNUMBER(R)
3. a clause record which we access as RIGHTPARENT(R)
4. an integer which we access as RIGHTLITERALNUMBER(R)
5. an integer which we access as NUMBEROFLITERALSIN(R)
6. an integer which we access as MAXIMUMINDEX(R)
7. a list of bindings which we access as BINDINGS(R). The function ISBOUND accesses this component. The function BIND accesses and alters this component.

These components represent, in order, the items enumerated at the end of the previous section.

To obtain the expression for the Kth literal of a clause record CL we call

GETLIT(CL,K). GETLIT sets a global variable LITG to the input literal and a global variable INDEXG to the index such that the expression LITG,INDEXG represents the Kth literal of CL (in the binding environment of CL).

Definition of GETLIT(CL,K)

If CL is an input record, then
set LITG to the Kth member of CL
set INDEXG to 1
Otherwise if K<LEFTLITERALNUMBER(CL) then
call GETLIT(LEFTPARENT(CL),K)
Otherwise if K<NUMBEROFLITERALSIN(LEFTPARENT(CL)) then
call GETLIT(LEFTPARENT(CL),K+1)
Otherwise if K<NUMBEROFLITERALSIN(LEFTPARENT(CL))−1
+RIGHTLITERALNUMBER(CL) then
call GETLIT(RIGHTPARENT(CL),
K−NUMBEROFLITERALSIN(LEFTPARENT(CL))+1)
set INDEXG to INDEXG+MAXIMUMINDEX(LEFTPARENT(CL))
Otherwise
call GETLIT(RIGHTPARENT(CL),
K−NUMBEROFLITERALSIN(LEFTPARENT(CL))+2)
set INDEXG to INDEXG+MAXIMUMINDEX(LEFTPARENT(CL))

End of definition.

To determine the binding, if any, of a variable VAR and an index INDEX in a binding environment BNDEV we call ISBOUND(VAR,INDEX,BNDEV). The call returns true if VAR,INDEX is bound in BNDEV. Otherwise, the call returns false. If the call returns true, then ISBOUND has set a global variable TERMB to the term and a global variable INDEXB to the index such that VAR,INDEX is bound to TERMB,INDEXB in BNDEV. BNDEV is always a clause record. ISBOUND looks in the BINDINGS(BNDEV) for a binding of VAR,INDEX. If none exists, ISBOUND is called recursively on the appropriate parent of BNDEV.

Definition of ISBOUND(VAR,INDEX,BNDEV)

If BNDEV is an input clause, return(false)
Otherwise if there is some binding of the form (VAR,INDEX . T, I) in
BINDINGS(BNDEV) then
set TERMB to T
set INDEXB to I
return(true)
Otherwise if INDEX≤MAXIMUMINDEX(LEFTPARENT(BNDEV)) then
return(ISBOUND(VAR,INDEX,LEFTPARENT(BNDEV)))
Otherwise
call ISBOUND(VAR,INDEX−MAXIMUMINDEX
(LEFTPARENT(BNDEV)),RIGHTPARENT(BNDEV))
If the call returns false, then return(false)
Otherwise

set INDEXB to INDEXB+MAXIMUMINDEX(LEFTPARENT (BNDEV))
return(true)

End of definition.

To add a binding (V,I . T,J) to BNDEV we call BIND(V,I,T,J,BNDEV).

Definition of BIND(V,I,T,J,BNDEV)

set BINDINGS(BNDEV) to CONS((the binding (V,I . T,J)), BINDINGS(BNDEV))

End of definition.

To resolve two clause records CL1 and CL2 on their Ith and Jth literals respectively, we call RESOLVE(CL1,I,CL2,J). RESOLVE uses the local variables: LEFTLIT,RIGHTLIT,LEFTINDEX,RIGHTINDEX,BNDEV.

Definition of RESOLVE(CL1,I,CL2,J)

Call GETLIT(CL1,I)
Set LEFTLIT to LITG
Set LEFTINDEX to INDEXG
Call GETLIT(CL2,J)
Set RIGHTLIT to LITG
Set RIGHTINDEX to INDEXG+MAXIMUMINDEX(CL1)
Set BNDEV to the new resolvent record
⟨CL1,I,CL2,J,NUMBEROFLITERALSIN(CL1)+ NUMBEROFLITERALSIN(CL2)−2, MAXIMUMINDEX(CL1)+MAXIMUMINDEX(CL2), the empty list⟩
Check to see that the signs of RIGHTLIT and LEFTLIT are opposite.
If not, return(FAIL)
Call UNIFY(theatomof(LEFTLIT),LEFTINDEX,theatomof (RIGHTLIT),RIGHTINDEX)
If UNIFY returns true, then return(BNDEV)
Otherwise return(FAIL)

End of definition.

Note that the most time consuming function in RESOLVE is the unification step. In particular, notice that standardizing the clauses apart is accomplished entirely by incrementing indices, and that except for the unifying substitution, the work involved in the creation of the resolvent is independent of the complexity of the two clauses represented by the parents. If the unification is successful, BNDEV is the clause record of the resolvent, and is returned. Otherwise, RESOLVE returns FAIL. The functions UNIFY and OCCUR are exactly as in sections 4 and 3 (except that they now use the definitions of ISBOUND and BIND of this section).

With minor alterations one can avoid constructing the new BNDEV unless the unification succeeds. In section 8 we mention other instances in which we have sacrificed efficiency for clarity in our definitions.

Figure 1 exhibits a derivation involving four resolutions. Figure 1(a) of

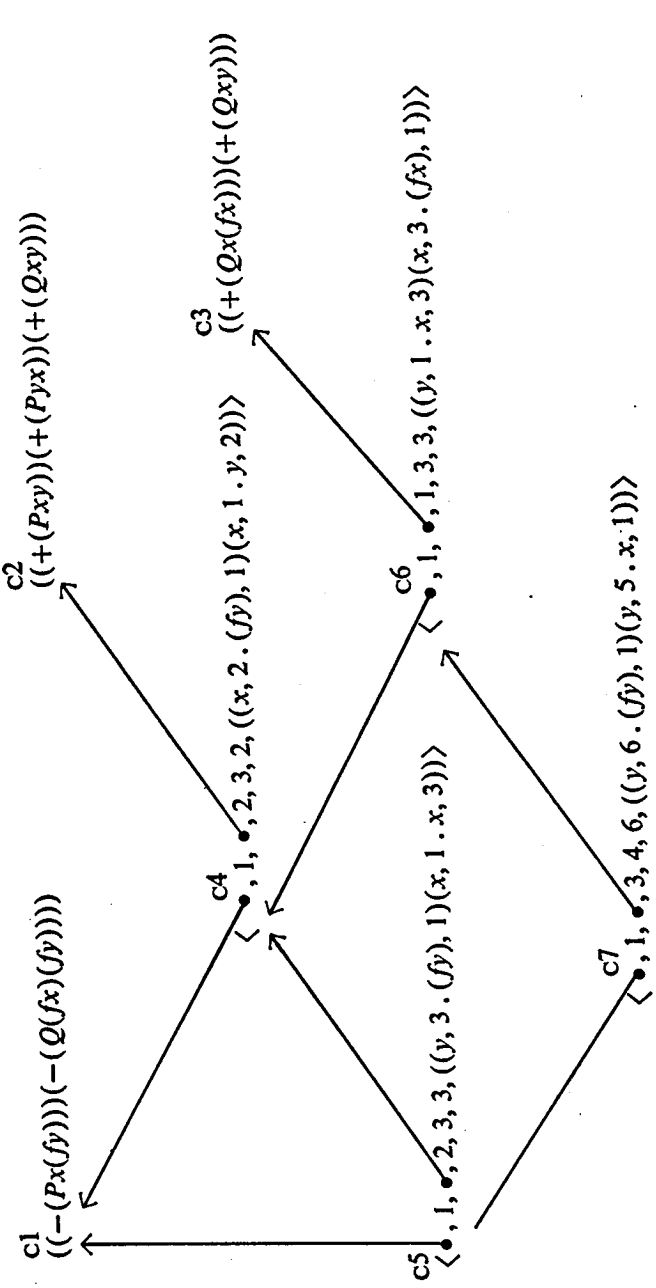

c1
((−(Px(fy)))(−(Q(fx)(fy))))
c2
((+(Pxy))(+(Pyx))(+(Qxy)))
c3
((+(Qx(fx)))(+(Qxy)))
c4
⟨, 1, •, 2, 3, 2, ((x, 2 . (fy), 1)(x, 1 . y, 2))⟩
c5
⟨• , 1, •, 2, 3, 3, ((y, 3 . (fy), 1)(x, 1 . x, 3))⟩
c6
⟨• , 1, •, 1, 3, 3, ((y, 1 . x, 3)(x, 3 . (fx), 1))⟩
c7
⟨• , 1, •, 3, 4, 6, ((y, 6 . (fy), 1)(y, 5 . x, 1))⟩

Figure 1(a).

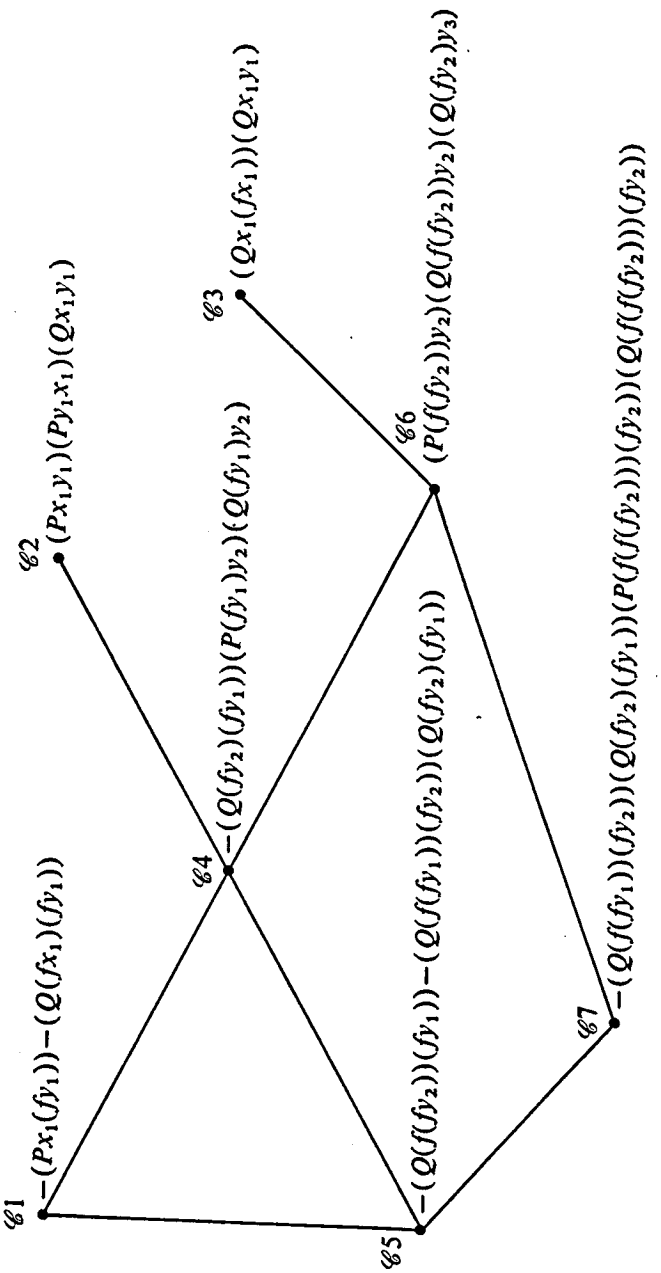

¢1
−(Px1(fy1))−(Q(fx1)(fy1))
¢2
(Px1y1)(Py1x1)(Qx1y1)
¢3
(Qx1(fx1))(Qx1y1)
¢4
−(Q(fy2)(fy1))(P(fy1)y2)(Q(fy1)y2)
¢5
−(Q(f(fy2))(fy1))−(Q(f(fy1))(fy2))(Q(fy2)(fy1))
¢6
(P(f(fy2))y2)(Q(f(fy2))y2)(Q(fy2)y3)
¢7
−(Q(f(fy1))(fy2))(Q(fy2)(fy1))(P(f(f(fy2)))(fy2))(Q(f(f(fy2)))(fy2))

Figure 1(b).

the figure exhibits the tree of clause records, while 1(b) shows the clauses represented at each node.

The clauses labeled C1, C2, and C3 are input clause records. The four remaining clause records are generated by RESOLVE as follows:

C4=RESOLVE(C1,1,C2,2)
C5=RESOLVE(C1,1,C4,2)
C6=RESOLVE(C4,1,C3,1)
C7=RESOLVE(C5,1,C6,3).

It is useful to trace the descent of the 3rd literal of C2 through the tree. In C2 it is represented by the expression T,1 where T is $(+(Q\,x\,y))$. In the binding environment C2 this expression has the value $(Q\,x_1\,y_1)$.

The descendant of this literal in clause C4 is the 3rd literal of that clause. There the expression is T,2 which has value $(Q(f\,y_1)\,y_2)$ in C4 due to the binding $(x,2\,.\,(f\,y),1)$ in BINDINGS(C4).

In C5, the term has index 3 and represents the 3rd literal of C5. T,3 in C5 has the value $(Q(f\,y_2)\;(f\,y_1))$.

Of course, the value of this expression in C5 in no way affects its value in C4. Thus, when we resolve C4 and C3 to form C6, the 3rd literal of C4 descends to become the 2nd literal of C6, where it is represented by T,2. The value of T,2 in C6 is $(Q(f(f\,y_2))\,y_2)$, due to bindings at C6 and C4.

Finally, we can trace the term, T, to C7 where it has index 3 as the 2nd literal and index 5 as the 4th literal of C7. T,3 in C7 has value $(Q(f\,y_2)\;(f\,y_1))$. T,5 in C7 has value $(Q(f(f(f\,y_2)))\;(f\,y_2))$.

Note that the double use of C4 in the tree introduces no confusion of bindings. In using both T,3 and T,5 ISBOUND finds relevant bindings at C4. However, in one case it returns to C4 via the branch through C5, and in the other via the branch through C6.

As an example of how bindings on one side of the tree can affect values of terms from the other, the reader should calculate the value of y,4 in the binding environment C7. It is found to be $(f(f\,y_2))$ after using bindings found at C6, C4, C7, and C5.

8. NOTES

We have completely ignored merging and factoring up to this point; however, they present no difficulty for our representation. All that is required to represent a merge or factor is an indication of which literal is to be deleted and the substitution used. In one of our programs we represent a merge as a resolvent in which one of the parents is a dummy (with one literal and maximum index zero), and the other is the clause containing the literal to be merged. We pretend to be resolving on that literal.

Subsumption and variant checking are also possible. Because the indices can be used to note from which clause a given variable has come, it is easy to redefine the function OCCUR in such a way that UNIFY succeeds only when one term subsumes (or is a variant of) the other. Thus the code for

UNIFY can be made to do several different jobs in this representation.

The recursion in the functions ISBOUND and GETLIT can be replaced by loops. The first recursion in OCCUR and the first two in UNIFY can also be replaced by loops. The resulting code is more efficient and opaque.

A more interesting increase in efficiency can be obtained by eliminating the search ISBOUND makes through the tree of clause records. We set up a two dimensional array we call VALUE of VARIABLES×INDICES. When we expect to use a clause record CL for any length of time (e.g., repeated resolutions, factoring, subsumption), we load VALUE with the binding environment of CL. Then to find if VAR,INDEX is bound we simply check VALUE(VAR, INDEX).

In the context of the VALUE array, BIND takes only four arguments, and it inserts TERMB,INDEXB into VALUE(VAR,INDEX). In addition, it pushes a pointer to the VALUE cell thus modified so that we can recover the substitution produced by UNIFY and later remove the bindings inserted. This allows recursive code for factoring, subsumption checking, and depth-first (backtracking) search. At each level of the recursion, VALUE contains the current BNDEV. Successful unifications add bindings to the array so that it contains the correct binding environment for the resolvent or factor produced. Upon exiting from recursion (in the search or factoring functions, for example), the stack is used to restore VALUE to its configuration upon entry (by removing the bindings inserted since entry).

It was because of the extensive use of VALUE that we chose integers as indices. Actually, all that an index must do is specify a unique branch up the binary tree of clause records. In one of our programs we use logical words treated as bit strings in place of indices. Each bit tells ISBOUND whether to branch to the right or left parent at the current node. Instead of incrementing indices, one shifts them.

Our 7-tuple representation can probably be improved for many restrictions of resolution. For example, in our implementation of SL-resolution (Kowalski and Kuehner 1971) we take advantage of the fact that in SL one parent of each resolvent is an input clause. Furthermore, the literals last to enter a clause are the first resolved upon. Thus SL-derivations have an attractive stack structure in which each stack entry is the residue of an input clause. Instead of keeping the number of literals in a clause, we keep a bit mask for each stack entry to tell us which literals from the input clause are still around. In this representation merging involves merely turning off a bit and storing a substitution.

Since many of the components of our records contain small integers, it is possible to pack these so that a record requires very few machine words. Our general non-linear implementation in POP-2 requires $(7+2n)$ 24-bit words per clause, where n is the number of bindings made. This includes the overhead for the POP-2 structures involved. Our SL implementation requires $(6+2n)$ 24-bit words. A machine code implementation of the general

structure sharing on a 36-bit word machine would require $(2+n)$ words per clause. Of course, the beauty of these expressions is that the space required to represent a clause is independent of its length or the depth of its function nesting.

We have obtained some rough statistics comparing our representation with two others, namely the most obvious list representation and the most compact character array imaginable. The latter is extremely slow to use since one spends almost all of one's time parsing. We assumed a 36-bit word machine was being used. On the basis of 3,000 randomly generated clauses, our representation is 10 times more compact than character arrays (at 5 characters per word) and 50 to 100 times more compact than lists (at 1 cons per word).

This data was generated using a general purpose implementation of structure sharing in POP-2 on an ICL 4130. Each clause was generated using structure sharing and the space required to represent it under the various schemes was then calculated. The VALUE array was not used. On terms whose average function nesting depth was 5, the program required 160 milliseconds per unification. The average longest branch in the derivations searched by ISBOUND was 8.3. For comparison purposes it should be pointed out that POP-2 on the 4130 requires 160 microseconds to execute '1+2' in a compiled function.

Our SL-resolution implementation is written as efficiently as possible in POP-2 and generates 9 clauses per second on the 4130. This includes tautology checking (but not subsumption) for each clause generated. The compiled POP-2 code requires 10K of 24-bit words and the program causes no garbage collection.

J.A.Robinson describes (1971) how unification of terms in the context of a substitution is possible without applying the substitutions. We believe that R.Yates wrote for QA3 the first such algorithm. The idea also appears in Hoffman and Veenker (1971).

Acknowledgements

Our thanks to J.A.Robinson, B.Meltzer, Pat Hayes, Robert Kowalski, and to the Science Research Council for financial support.

REFERENCES

Hoffman, G.R. & Veenker, G. (1971) The unit-clause proof procedure with equality. *Computing*, **7**, 91–105.

Kowalski, R. & Kuehner, D. (1971) Linear resolution with selection function. *Artificial Intelligence*, **2**, 227–60.

Robinson, J.A. (1971) Computational logic: the unification algorithm. *Machine Intelligence 6*, pp. 63–72 (eds Meltzer, B. & Michie, D.) Edinburgh: Edinburgh University Press.

7

Some Special Purpose Resolution Systems

D. Kuehner
Computer Science Department
University of Western Ontario

Abstract

Any input set with a purely linear (input) or unit refutation is shown to be renameable as a set of Horn clauses. Purely linear and unit resolution are refined to selective negative linear (SNL) resolution, and selective positive unit (SPU) resolution. For input sets of Horn clauses, SPU-resolution, SNL-resolution, and bi-directional SPU/SNL-resolution are shown to be complete. An efficient bi-directional search strategy is examined.

PURELY LINEAR AND UNIT REFUTATIONS

In automatic theorem proving, proofs in which each line is deducible from the line preceding it are considered to be conceptually simpler than other types of proofs. This paper characterizes the theorems which have such proofs, and examines alternative inference rules for them.

In the following it is assumed that the reader is familiar with the basic terminology of resolution theory (J.A.Robinson 1967).

A derivation from a set S of input clauses is *purely linear* if each resolvent in the derivation has an input clause as one parent. In such derivations each line is deducible from the line preceding it. In a *unit* derivation, each resolvent has a unit clause as one parent. (A unit clause is a one literal clause.) Let the substitution θ be a most general unifier of a partition of the literals in the clause C. Then $C\theta$ is a *factor* of C, and is a *unit factor* if $C\theta$ is a unit clause.

Chang (1970) shows that if S contains all of its unit factors, then S has a purely linear refutation iff it has a unit refutation.

HORN CLAUSES AND THE RENAMING OF SETS

Not all unsatisfiable sets of clauses have purely linear or unit refutations. If the unsatisfiable sets having such refutations can be characterized, then purely linear or unit resolution may be used to search for their refutations.

A clause is a *Horn clause* (Horn 1951) iff it has at most one positive literal. (A positive literal is an unnegated atom.) Problems expressible using Horn clauses are characterized by Cohn (1965) and include many theorems of group theory.

Let E be a literal, a clause, a set of clauses, or a derivation. Let Q be a set of atoms occurring in E. Let $r(E)$ be identical to E except that for each A in Q, each occurrence of A in E is replaced by $\bar{A}$ and each occurrence of $\bar{A}$ in E is replaced by A. Then r is a *renaming* and $r(E)$ is the *renamed version* of E.

A *positive unit* derivation is a derivation in which every resolvent has a positive unit as one parent.

Theorem 1. The minimally unsatisfiable set S of ground clauses has a unit refutation iff there is a renaming r such that $r(S)$ is a set of Horn clauses.

Proof. Case 1. Let D be a positive unit refutation of S. Let $C_1 \cup C_2$ be a clause in S such that every literal in C_1 is negative and every literal in C_2 is positive. Since S is minimally unsatisfiable, $C_1 \cup C_2$ occurs at a tip of D. It is not difficult to see that C_2 must be a resolvent occurring in D. If C_2 contains two or more positive literals, then it must resolve with a negative unit. This is contrary to the assumption that D is a positive unit refutation. Thus C_2 contains at most one literal. It follows that S must be a set of Horn clauses. That is, any minimally unsatisfiable set of ground clauses which has a positive unit refutation is a set of Horn clauses.

Case 2. Let D be a unit refutation but not a positive unit refutation of S. If $r(D)$ is a renamed version of D such that $r(D)$ is a positive unit refutation, then clearly $r(S)$ is a set of Horn clauses.

Let D be a unit refutation of S such that no other unit refutation of S has fewer resolutions than occur in D.

In D let $\{L\}$ and $\{\bar{L}\}$ be the unit clauses whose resolvent is the null clause. Let U^+ be the set of all positive units, except L, which occur in D. Let U^- be the set of all negative units, except $\bar{L}$, which occur in D. Since no refutation of S has fewer resolutions than occur in D, it follows that no literal in U^+ has the same atom as a literal in U^-. Let r be the renaming which changes the sign of the literals in U^-. Note that $r(U^+) = U^+$. Then $r(D)$ is a positive unit refutation and so $r(S)$ is a set of Horn clauses.

To prove the converse, assume that S is an unsatisfiable set of Horn clauses. Then there is a P_1-refutation of S (J. A. Robinson 1965). It is easy to verify that a P_1-refutation of a set of Horn clauses is a positive unit refutation. Q.E.D.

Unfortunately there is no straightforward lifting of theorem 1 to the general case. However, the existence of an appropriate but extended form of renaming can be deduced from the examination of the ground image of a general level unit refutation.

Let D be a unit refutation of a set S of general level clauses. Let C be any

resolvent in D. Let C_1 and C_2 be the parents of C, and let θ be the substitution used to obtain C from C_1 and C_2.

Let ϕ be the substitution such that $C\phi$ is the ground image of C. If C is the null clause then ϕ is the null substitution. Then $C_i\,\theta\,\phi$ is the *ground image* of C_i in D for $i = 1, 2$. If D' is isomorphic to D, and if for each C in D, the corresponding C' in D' is the ground image of C in D, then D' is the *ground image* of D. Let S' be the corresponding ground image of S.

From theorem 1 it is clear that there is a renaming r such that $r(S')$ is a set of Horn clauses. Obviously, some set of replicas of S can be renamed in a way corresponding to the way r renames S'.

In order to effectively determine whether a set of clauses can be renamed as a set of Horn clauses, a renaming procedure must be specified which preserves unsatisfiability and allows the independent renaming of different occurrences of the same predicate letter, and the independent renaming of different copies of the same clause. The existence of procedures of this kind are suggested by the following example.

$$S=\{Pax\,Pxa, \bar{P}ax\,\bar{P}xf(x), \bar{P}ax\,\bar{P}f(x)\,x\}.$$

Note that $Pax\,Pxa$ is an abbreviated form of $\{P(a,x), P(x,a)\}$, etc. After factoring and numbering the occurrences of literals S becomes

$$S_1=\{P_1ax\,P_2xa, P_3aa, \bar{P}_4ax\,\bar{P}_5xf(x), \bar{P}_6ax\,\bar{P}_7f(x)x\}$$

By noting which literals are not unifiable, and avoiding refactoring of $P_1ax\,P_2xa$, the following unifiable groups are formed

P_2xa	P_1ax	P_3aa
$\bar{P}_7f(x)x$	$\bar{P}_5xf(x)$	$\bar{P}_4ax$
	$\bar{P}_4ax$	$\bar{P}_6ax$
	$\bar{P}_6ax$	

The following is a predicate letter renaming and clause recopying corresponding to this grouping:

$$S_2=\{Qax\,Pxa, Raa, \bar{Q}ax\,\bar{Q}xf(x), \bar{R}ax\,\bar{Q}xf(x),$$
$$\bar{Q}ax\,\bar{P}f(x)x, \bar{R}ax\,\bar{P}f(x)x\}.$$

Finally the literals with predicate letter P are renamed (re-signed) to form a set of Horn clauses.

It has been shown that any input set has a purely linear refutation iff it has a unit refutation, and that any set has a unit refutation iff it can be renamed as a set of Horn clauses. Thus, in order to refine purely linear resolution, it is sufficient to look at refinements which are complete for input sets of Horn clauses.

SNL-RESOLUTION

It is possible to impose two further restrictions on purely linear resolution without losing completeness relative to Horn clauses. The first restriction is that one parent must be negative. That is, each literal in one parent is a negated atom. The other restriction is that the non-input parent, C, of a

resolution have a selected literal, L, and that all resolvents with C as one parent have L as the literal resolved upon. As has been amply pointed out by Kowalski and Kuehner (1971), this causes a remarkable reduction in the branching rate of search trees. The selection of a literal may be anticipated with the input clauses, so that all clauses of the search are ordered.

For notational convenience and for efficient computer implementation, it is useful to treat resolution as a sequence of two operations, factoring followed by resolution of factored clauses. If C is a clause and E a unifiable partition of the literals of C, having most general unifier (m.g.u.) θ, then $C\theta$ is a *factor* of C. If exactly one component of E contains two literals and every other component exactly one, then $C\theta$ is a *basic factor* of C, and $C\theta$ is obtained from C by one factoring operation. The resolution of factored clauses unifies one literal from one parent with the complement of one literal from the other parent. Although other factoring methods are compatible with SNL-resolution, only Kowalski's m-factoring (1970) is considered in the following discussion. The method of implementing m-factoring is to factor input clauses in all possible ways, and to factor resolvents in all possible ways provided that the literals which are unified in the factoring descend from different input parents. This last restriction ensures against redundant factoring. This implementation of factoring is built into the definition of SNL-resolution.

An *ordered clause* C^* is a sequence of literals, possibly containing duplications. An *ordered Horn clause* contains at most one positive literal, which must be the leftmost literal if it occurs. If C is a Horn clause then C^* is an ordered Horn clause containing just the literals of C. For any set S of Horn clauses, the set S^* is the set of all ordered Horn clauses obtainable from factors of clauses of S. For example, if

$$S=\{\bar{P}ax\,\bar{P}xy\} \quad \text{then} \quad S^*=\{\bar{P}ax\,\bar{P}xy,\ \bar{P}xy\,\bar{P}ax,\ \bar{P}aa\}$$

For any ordered Horn clauses C_1^* and C_2^*, let $C_1^*C_2^*$ be the ordered clause beginning with the literals in C_1^* in the order they appear in C_1^*, followed on the right by the literals of C_2^* in the order that they appear in C_2^*. $C^*\bar{L}$ is the ordered Horn clause whose rightmost literal is $\bar{L}$.

An *ordered negative resolution* has as one parent an ordered negative Horn clause of the form $C_1^*\bar{L}$ and as the other parent an ordered mixed or unit Horn clause of the form KC_2^* where C_2^* may be empty. Neither parent contains two literals which have the same atom. The two parents do not share variables. If L and K are unifiable with m.g.u. θ, then the *ordered negative resolvent*, or ordered N_1-resolvent, is $C_1^*C_2^*\theta$. Note that $C_1^*C_2^*\theta$ is an ordered negative Horn clause. The *literal resolved upon* in $C_1^*\bar{L}$ is $\bar{L}$, and in KC_2^* it is K. If C_2^* is not empty, then the literals of C_2^* are the *new* literals of $C_1^*C_2^*\theta$.

Let C^* be an ordered negative resolvent, and let $\bar{K}$ occur as a new literal of C^*. If there is a non-new literal $\bar{L}$ occurring in C^* such that L and K are unifiable with m.g.u. θ, then an *ordered factor* of C^* is $C_0^*\theta$, where C_0^* is

obtained by deleting the given occurrence of L from C^*. The literal L is *factored out* of C^*. The *factoring* operation is said to have been applied to C^*. If n rightmost literals of C^* are new in C^*, then define the n rightmost literals of $C_0^*\theta$ to be new in $C_0^*\theta$. If the mgu θ is the null substitution, then $L=\bar{K}$. In this case L and $\bar{K}$ are said to *merge*, and L is *merged out* of C^*. The *merging* operation is an instance of the factoring operation.

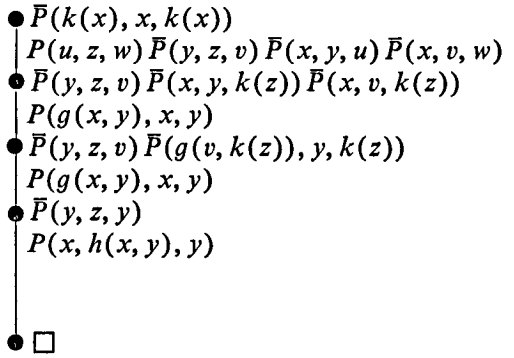
An *SNL-derivation* from a set S of Horn clauses is a sequence $(C_1^*, \ldots, C_n^*)$ of ordered clauses satisfying the following conditions:

(1) The *initial* clause C_1^* is in S^* and is negative.

(2) C_{i+1}^* is an ordered N_1-resolvent of C_i^* (the *near parent*) and an ordered clause (the *input parent*) from S^*, or C_{i+1}^* is an ordered factor of C_i^*.

It should be noted that in an SNL-derivation factoring precedes resolution. That is, if a new literal is to be factored out of a clause, this factoring must be done before any other new literal is resolved upon.

SNL-derivations are not represented in the standard derivation format. Together with other linear derivations, SNL-derivations are conceived of as being vine-like trees consisting of an initial node together with a sequence of its descendants. The input parent of an ordered clause is attached to the arc joining the node of a resolvent with the node of its near parent. This representation simplifies the appearance of SNL-search spaces. Using this representation, the following example is an illustration of an SNL-refutation of a familiar group theory problem. The sequence of ordered clauses at the nodes is the SNL-refutation.



The implementation of SNL-resolution is more efficient if there is a retroactive ordering of factors of input clauses. Ordered negative resolution could be redefined so that the literal resolved on in the negative parent is any literal selected from those literals most recently introduced into the deduction. In order to use this dynamic ordering, some marker should be inserted between the residues of the two parents when forming a resolvent. Thus, resolvents become sequences of cells of literals separated by markers. Such sequences of cells correspond to the chains of SL-resolution (Kowalski and Kuehner 1971) and model elimination (Loveland 1969). However, the use of ordered clauses rather than a selection function simplifies the following discussion.

It should be noted that since every SNL-resolvent is negative, no resolvent can be a tautology. Thus, if tautologies are deleted from the factors of the input set, no further deletion of tautologies need be done.

A subset S' of a set S of clauses is a *support set* (Wos, Carson, and Robinson 1965) for S iff $S-S'$ is satisfiable. In common with other linear resolution systems, the initial clause of an SNL-derivation may be restricted to belong to a given support set of the input set S. In this case, the support set must be a subset of the set of all factors of negative clauses in S.

COMPLETENESS THEOREMS FOR SPU- AND SNL-RESOLUTION

The existence of an SNL-refutation for any unsatisfiable set of Horn clauses is proved by permuting the resolutions of a positive unit refutation of S. In order to do this, it is necessary to examine the structure of a P_1-refutation of a set of Horn clauses.

Let S by any set of Horn clauses. Clearly, any P_1-resolution between members of S must have a Horn clause as a resolvent. Thus, in any P_1-derivation from S, one parent of each resolution must be a positive unit. Thus P_1-resolution with Horn clauses is positive unit resolution. If the other parent is a mixed Horn clause, then the resolvent is either a shorter mixed Horn clause or a positive unit. If the other parent is a negative Horn clause, then the resolvent is either a shorter negative clause or a null clause. It follows that any positive unit refutation of a set of Horn clauses has one and only one negative input clause.

Since every positive unit resolution has a positive unit clause as one parent, all resolvents are instances of subsets of input clauses. It follows that in every positive unit derivation, the input clauses may be replaced by appropriately ordered Horn clauses, with the positive literal on the left, and with the rightmost literal the literal resolved upon. A positive unit derivation, all of whose clauses are ordered Horn clauses, is a selective positive unit (SPU) derivation. Clearly, there is a one-one correspondence between positive unit derivations and isomorphic SPU-derivations. The following theorem needs no further proof.

Theorem 2. If S is an unsatisfiable set of Horn clauses then there is an SPU-refutation of S^*.

In order to compare the complexity of SPU-derivations and SNL-derivations, the complexity to be used is the *size* of the derivation, the number of resolutions in the derivation tree.

In calculating the size of either an SPU-derivation or an SNL-derivation, it is assumed that the input clauses are from S^* not from S. Since all SPU-resolvents are instances of subsets of ordered input clauses, SPU-resolution for Horn clauses is complete with no factoring other than the factoring of input clauses.

Theorem 3. Let S be any unsatisfiable set of ground-level Horn clauses. Let

D be any SPU-refutation of S^*. Then there exists an SNL-refutation D^* of S^* which is at least as simple as D.

Proof (by induction on the size of D). The proof is for the stronger theorem which also proves that the ordered negative input clause of D is the initial ordered clause of D^*.

If D has size one, then D^* and D are the same derivation, so in this case the theorem is trivially true.

Otherwise, let D have size $n>1$, and assume that the theorem holds for all ordered SPU-refutations of size less than n. Let $\bar{L}C^*$ be the ordered negative input clause of D. Then the immediate sub-derivations of D derive $\bar{L}$ and L. Let D_1 be obtained by removing $\bar{L}$ from $\bar{L}C^*$, and its descendants, in the immediate sub-derivation which derives $\bar{L}$. Then D_1 is an ordered SPU-refutation of the set S_1^* of ordered input clauses of D_1. Also, D_1 is isomorphic to the derivation of $\bar{L}$ and it has C^* as its negative input clause. Since the size of D_1 is less than n, then by the induction hypothesis there exists an SNL-refutation D_1^* of S_1^* which is at least as simple as D_1, and whose initial ordered clause is C^*.

If D_1^* is $(C^*, C_1^*, \ldots, C_k^*)$ and C_i^* is the first ordered clause of D_1^* which contains $\bar{L}$, then let D^* be obtained from D_1^* by concatenating $\bar{L}$ onto the left of each of $C^*, C_1^*, \ldots, C_{i-1}^*$, and inserting $\bar{L}C^*$ between $\bar{L}C_{i-1}^*$ and C_i^*. Then C_i^* is obtained from $\bar{L}C_i^*$ by merging out $\bar{L}$. ($\bar{L}$ must be new in the first clause of D_1^* in which it occurs.) Then D^* is an SNL-refutation of S with size less than n, and initial ordered clause $\bar{L}C^*$.

Otherwise, none of the ordered clauses in D_1^* contain $\bar{L}$. Let D_{11}^* be obtained from D_1^* by concatenating $\bar{L}$ onto the left of each ordered clause of D_1^*. Then D_{11}^* is an SNL-derivation of $\bar{L}$, of the same size as D_1^*, and with initial ordered clause $\bar{L}C^*$.

Let D' be the immediate sub-derivation of D which derives L. Let LC'^* be the ordered input clause of D' from which L descends. (That is, L occurs in all descendants of LC'^* which are ancestors of L.) Let D_2 be obtained by replacing LC'^* in D' by C'^*. Then D_2 is an SPU-refutation of the set S_2^* of ordered input clauses of D_2. Also, D_2 is isomorphic to the derivation of L and has C'^* as its negative input clause. Since the size of D_2 is less than n, then by the induction hypothesis there exists an SNL-refutation D_2^* of S_2^* which is at least as simple as D_2, and whose initial ordered clause is C'^*.

Let D_{21}^* be obtained from D_2^* by adding $\bar{L}$ onto the beginning of the refutation D_1^*. Then the second ordered clause C'^* of D_{21}^* is obtained from $\bar{L}$ by the ordered negative resolution with parents $\bar{L}$ and LC'^*.

Let D^* be obtained by identifying the last ordered clause of D_{11}^* with the initial clause of D_{12}^* to form a refutation of S^*, which is at least as simple as D and which has initial ordered clause $\bar{L}C^*$. Q.E.D.

It should be noted that if there is no merging in D^*, or if the use of the merging operation is suppressed in constructing D^*, then D and D^* have exactly the same size.

Since both SPU-resolution and SNL-resolution use ordered clauses, the concept of lifting must be slightly modified. For any ordered clause $C=L_1 \ldots L_n$, and for any substitution θ, the ordered clause $C\theta=L_1\theta \ldots L_n\theta$ is an *ordered instance* of C. $C\theta$ may contain identical literals even though C does not.

The derivation D *lifts* the derivation D' iff

(1) D is isomorphic to D',

(2) for any ordered clause C of D, the corresponding ordered clause C' of D' is an ordered instance of C, and

(3) the literal resolved upon in C' is an instance of the literal resolved upon in C and is in the same position in both clauses.

Theorem 4. Let D' be an SNL-refutation of a set of ordered ground instances of clauses in the set S of Horn clauses. Then there exists an SNL-refutation D^* of S^* which lifts D' and has the same size as D'.

Proof. Let S' be the set of ordered input clauses of D', and let C_1' be the ordered negative clause of S'. Let S^* be the set of ordered factors of clauses of S such that C^* is in S^* iff S' contains an ordered instance C' of C^*.

The initial ordered clause C_1^* of D^* is the ordered clause in S^* which has C_1' as an ordered instance. Assume that the SNL-derivation $(C_1^*, \ldots, C_i^*)$ lifts the sub-derivation $(C_1', \ldots, C_i')$ of $D'=(C_1', \ldots, C_i', \ldots, C_n')$. Then C_i^* is an ordered instance of C_i'.

If C_{i+1}' is obtained from C_i' by merging the ith and jth literals of C_i', then there exists a most general unifier σ which unites the ith and jth literals of C_i^* to produce C_{i+1}^*. If $C_i'=C_i^*\theta$, then $\theta=\sigma\lambda$ for some λ. If the ith and jth literals are the only identical literals of C_i', then they are the only identical literals of $C_i^*\sigma$. Let C_{i+1}^* be the factor of C_i^* obtainable by deleting the leftmost of the ith or jth literals of $C_i^*\sigma$. Clearly C_{i+1}^* is an ordered instance of C_{i+1}^*.

Otherwise, let C_{i+1}' be obtained from C_i' by SNL-resolution with $C' \in S'$. Then there exists C^* in S^* such that C' is an ordered instance of C^*. By using the lifting lemma of J. A. Robinson (1965), it is easily seen that there exists an SNL-resolvent C_{i+1}^* such that C_{i+1}' is an ordered instance of C_{i+1}^*.

In either case $(C_1^*, \ldots, C_i^*, C_{i+1}^*)$ is an SNL-derivation which lifts $(C_1', \ldots, C_i', C_{i+1}')$. It follows that there exists an SNL-refutation D^* which lifts D' and has the same size as D'. Q.E.D.

Theorem 5. For any unsatisfiable set S of Horn clauses, there is an SNL-refutation D^* of S^* such that D^* is at least as simple as the simplest SPU-refutation of S^*.

Proof. By Lemma 2 of Kowalski and Kuehner (1971), there is a simplest SPU-refutation D of S which lifts and has the same size as a ground SPU-refutation D' of a set S' of ordered ground instances of clauses in S. By Theorem 3, there is an SNL-refutation D'^* of S^* which is at least as simple as D'. By Theorem 4 there is an SNL-refutation D^* which lifts D'^* and which has the same size as D'^*. Therefore D^* is an SNL-refutation of S^* which is at least as simple as the simplest SPU-refutation of S^*. Q.E.D.

BI-DIRECTIONAL SPU/SNL RESOLUTION

Although bi-directional search has been investigated as a general problem solving technique by Pohl (1971) and others, there are difficulties in applying it to theorem proving. However, for input sets of Horn clauses, SPU- and SNL-resolution can be combined to perform a bi-direction search for a refutation.

In the following, ordered negative resolution is combined with factoring. Thus, if $D^*=(C_1^*, \ldots, C_m^*)$ is an SNL-derivation, then for $i=1, \ldots, m-1$ each C_{i+1}^* is obtained from C_i^* by ordered negative resolution with an input clause, together with factoring of the resolvent.

Let S be an unsatisfiable set of Horn clauses, and let $D^*=(C_1^*, \ldots, C_k^*, \ldots, C_m^*)$ be an SNL-derivation from S^*. Let $D_k^*=(C_1^*, \ldots, C_k^*)$ be the kth sub-derivation of D^*, and let $C_k^*=\bar{L}_1 \ldots \bar{L}_n$. Let D be an SPU-derivation from S^* with sub-derivations $D_1, \ldots, D_n$ of $K_1, \ldots, K_n$ respectively. If $\{L_1, K_1\}, \ldots, \{L_n, K_n\}$ are simultaneously unifiable then there is a *stage k meeting* between D^* and D, where C_k^* is said to *meet* $K_1, \ldots, K_n$. If there is such a stage k meeting, then D_k^* can be combined with the derivations $D_1, \ldots, D_n$ to form a stage k *bi-directional refutation* of S^*.

Note that any SPU-refutation is a stage 1 bi-directional refutation, and that any SNL-refutation of size k is a stage k bi-directional refutation.

Theorem 6. Let D be any SPU-refutation of the unsatisfiable set S^* of ordered ground-level Horn clauses, and let D have size s. Then there exists an SNL-refutation D^* of S^* such that, for every k less than or equal to the size of D^*, there is a stage k meeting between D and D^*. The corresponding stage k bi-directional refutation of S has size less than or equal to s.

Proof. The theorem will be proved by presenting a recursive method for the construction of D^* from D. That is, the theorem to be proved is that for every positive integer i, either (1) there is an SNL-refutation $D_j^*=(C_1, \ldots, C_j)$ of S^* for some $j \leqslant i$ and there is a stage k meeting between D_j^* and D for all $k \leqslant j$, or (2) there is an SNL-derivation $D_i^*=(C_1, \ldots, C_i)$ from S^* and there is a stage k meeting between D_i^* and D for all $k \leqslant i$.

If $C_1=\bar{L}_1 \ldots \bar{L}_n$ is the ordered negative input clause of D, then $D_1^*=(C_1)$. Since D is a refutation, it has subderivations $D_1, \ldots, D_n$ of $L_1, \ldots, L_n$. If these sub-derivations have size $m_1, \ldots, m_n$ then $m_1+\ldots+m_n+n=s$. Thus there is a stage 1 meeting between D_1^* and D. The resulting bi-directional refutation is D itself.

Let i be any positive integer and assume that the lemma holds for i. If (1) holds, then the lemma is proved.

Otherwise, assume that there is a derivation $D_i^*=(C_1, \ldots, C_i)$ such that there is a stage k meeting between D_i^* and D for every $k \leqslant i$. If C_i is the null clause, then (1) holds, and the lemma is proved.

Otherwise, let $C_i=\bar{L}_1 \ldots \bar{L}_n$. Then D has sub-derivations $D_1, \ldots, D_n$ of $L_1, \ldots, L_n$ such that if these subderivations have size $m_1, \ldots, m_n$ then, by

assumption, $m+m_1+,\ldots+m_n+n\leqslant s$, where m is the size of D_i^*.

If L_n is an ordered input clause, then the only clause of D_n is L_n. In this case, let $D_{i+1}^*=(C_i,\ldots,C_i,C_{i+1})$ where $C_{i+1}=\bar{L}_1\ldots\bar{L}_{n-1}$ is obtained from C_i by SNL-resolution with L_n.

Otherwise, let $L_n\bar{K}_1\ldots\bar{K}_r$ be the ordered input clause of D_n from which L_n descends. Then D_n has sub-derivations $D_1',\ldots,D_r'$ of $K_1,\ldots,K_r$ which have sizes $m_1',\ldots,m_r'$ where $m_1'+\ldots+m_r'+r=m_n$. Let $D_{i+1}^*=(C_1,\ldots,C_i,C_{i+1})$ where $C_{i+1}=\bar{L}_1\ldots\bar{L}_{n-1}\,\bar{K}_1\ldots\bar{K}_r$ is obtained from C_i by SNL-resolution with $L_n\bar{K}_1\ldots\bar{K}_r$. But D contains sub-derivations $D_1,\ldots,D_{n-1},D_1',\ldots,D_r'$ of $L_1,\ldots,L_{n-1},K_1,\ldots,K_r$ whose sizes are $m_1,\ldots,m_{n-1},m_1',\ldots,m_r'$. Since $m+m_1+\ldots+m_n+n\leqslant s$ and $m_i'+\ldots+m_r'+r-m_n$, the size of the stage $i+1$ bi-directional refutation obtainable from $D_{i+1}^*,D_1,\ldots,D_{n-1},D_1,\ldots,D_r'$ is less than or equal to s.

Since it has been shown that D_{i+1}^* can be constructed for any i when D_i^* is not a refutation, it follows that D^* can be constructed and that for every k less than or equal to the size of D^*, there is a stage k meeting between D^* and D, and that the size of the corresponding stage k bi-directional refutation is less than s. Q.E.D.

The following theorems follow easily from Theorem 6, using the methods for proving Theorem 2 and Theorem 5.

Theorem 7. Let D' be a bi-directional refutation of a set S^* of ordered ground instances of clauses in the set S of Horn clauses. Then there exists a bi-directional refutation D^* of S^* which lifts D' and has the same size as D'.

Theorem 8. For any unsatisfiable set S of Horn clauses, there is a bi-directional refutation of S^* which is at least as simple as the simplest SPU-refutation of S^*.

A BI-DIRECTIONAL SEARCH STRATEGY

In the following discussion it is assumed that a search strategy should exhaust the possibility of finding a refutation of size k before trying to find one of size $k+1$.

Let the *length* of a clause be the number of literals in the clause. Let the *cost* of an SPU- or SNL-resolvent be the size of the smallest SPU- or SNL-derivation of the resolvent from the given set of input clauses. There is a bi-directional refutation of size k if an SNL-resolvent of cost g and length h meets h unit SPU-resolvents the sum m of whose sizes satisfies $g+h+m=k$. To ensure that all such units have been generated, it is necessary that all SPU-resolvent units of cost m or smaller have been generated. It is easy to verify that all ancestors of a unit of cost m have cost g' and length h' where $g'+h'\leqslant m+1$. Let the *merit* of an ordered clause be defined to be the sum of its cost and its length. If an SNL-resolvent of merit $g+h$ has been generated and if the search is attempting to generate a refutation of size k, then the search should generate all SPU-resolvents of merit $g'+h'\leqslant k+1-(g+h)$. If all such SPU-resolvents have been generated, then to exhaust the possibility

of finding a refutation of size k it is necessary and sufficient to generate all SNL-resolvents of merit $g+h$.

Defining merit in this way layers the SNL- and SPU-search spaces according to the strategy of diagonal search (Kowalski 1970).

The meeting of SNL- and SPU-derivations has been defined to use only positive unit SPU-resolvents, while any SNL-resolvent may be used to meet these units. There are three searches, the SPU-search for units, the SNL-search, and the search for a meeting between SPU-resolution units and SNL-resolvents. It seems most natural to combine the last two of these searches by using the SPU-resolvent units to augment the SNL-search. Call such units the *imported* units of the SNL-search. Since the imported units are intended to be used immediately if they are to be used at all, they are treated as input clauses by the SNL-search. Because the purpose of the SPU-search is to produce positive units, no negative clauses should be used as input clauses for the SPU-search.

For any SPU-refutation D of a set S^* of ordered Horn clauses there is an SNL-refutation D^* of S^* such that D and D^* meet at any stage. Since at least one clause of such a D^* occurs on each merit level up to the size of D^*, each merit level contains at least one clause which meets units of D. Because of this, the SNL-search may be stopped at any merit level, and a meeting will be obtained by generating SPU-resolvents. By generating all SPU-resolvents up to a merit level such that the sum of the merit levels is the size of the refutation, a meeting is generated. A similar argument holds for stopping the SPU-search at any merit level. Thus the number of merit levels saturated by each search is immaterial as long as the sum of the merit level reaches the size of the refutation.

A bi-directional search strategy is most efficient if it saturates as many merit levels as possible while generating as few clauses as possible. The implementation of a bi-directional search requires an alternation between searching an SPU-search space and searching an SNL-search space. To be most efficient, the alternation should be controlled by the relative number of clauses on the merit levels of the two searches.

On each merit level, the SPU-search and the SNL-search counts the number of clauses generated on that level. One search generates clauses until its count exceeds that of the other search. Then the other search begins generating clauses. If search A saturates a merit level before its count exceeds that of search B, then search A begins generating clauses on its next merit level, and its count becomes the number of clauses generated on the new level. In this way, one search may generate all the clauses on one merit level while the other search is inactive. In doing this, the bi-directional search is saturating as many merit levels as possible while generating as few clauses as possible. This also ensures that the difficulty of the bi-directional search is less than or equal to the difficulty of either search by itself. This method resembles the bi-directional search procedure suggested by Pohl (1971).

This alternating procedure is interrupted whenever a unit positive clause is

generated by the SPU-search and imported to the SNL-search. The SNL-search uses such a newly imported clause in all possible ways up to its current merit level. This interruption is necessary since the bi-directional search can be terminated only by finding a refutation during the SNL-search.

Apart from the efficiency gained by alternating between search spaces, the efficiency of bi-directional search is based on the assumption that the number of clauses on a merit level increases as the merit increases. It follows that the number of clauses generated in the search up to merit level n is less than half the number generated in the search up to merit level $2n$. If an SPU-search and an SNL-search produce a meeting after searching m and n levels respectively, then the bi-directional search generates fewer clauses than if either search were to go to level $m+n$.

NOTE

Two earlier versions of most of these results have had limited previous circulation (Kuehner 1969, Kuehner 1971).

REFERENCES

Chang, C. L. (1970) The unit proof and the input proof in theorem proving. *J. Ass. Comput. Mach.*, **17**, 698–708.

Cohn, P.M. (1965) *Universal Algebra*. New York & London: Harper and Row.

Horn, A. (1951) On sentences which are true of direct unions of algebras. *J. Symb. Logic*, **16**, 14–21.

Kowalski, R.A. (1970) Search strategies for theorem proving. *Machine Intelligence 5*, pp. 181–201 (eds Meltzer, B. & Michie, D.) Edinburgh: Edinburgh University Press.

Kowalski, R.A. and Kuehner, D.G. (1971) Linear resolution with selection function. *Artificial Intelligence*, **2**, 227–60.

Kuehner, D.G. (1969) Bi-directional search with Horn clauses. Metamathematics Unit Memo, University of Edinburgh.

Kuehner, D.G. (1971) Strategies for improving the efficiency of automatic theorem-proving. Ph.D. thesis, University of Edinburgh.

Loveland, D.W. (1969) A simplified format for the model-elimination theorem-proving procedure. *J. Ass. Comput. Mach.*, **16**, 349–63.

Pohl, I. (1971) Bi-directional search. *Machine Intelligence 6*, pp. 127–40 (eds Meltzer, B. & Michie, D.) Edinburgh: Edinburgh University Press.

Robinson, J.A. (1965) Automatic deduction with hyper-resolution. *Int. J. Comput. Math.*, **1**.

Robinson, J.A. (1967) A review of automatic theorem-proving. *Proc. Symp. Appl. Math.*, **19**, 1–18.

Wos, L.T., Carson, D.F. & Robinson, G.A. (1965) Efficiency and completeness of the set of support strategy in theorem-proving. *J. Ass. Comput. Mach.*, **12**, 687–97.

8

Deductive Plan Formation in Higher-order Logic

J. L. Darlington
Gesellschaft für Mathematik und
Datenverarbeitung Bonn

Abstract
A resolution-based theorem prover has been applied to problems in automatic plan formation. The program incorporates some recent advances in higher-order logic which enable it to produce shorter proofs than those achieved solely by first-order resolution methods, and to generate plans and programs with elementary loops with the aid of an axiom based on mathematical induction.

Much of the recent work in answering questions and solving problems by formal deductive methods has been inspired by the discovery by Green (1969) and others that a resolution-based theorem prover can be easily adapted to the generation of constructive answers to questions from data formulated in first-order predicate logic. The resolution method proves a theorem T from a set of axioms S by deducing a contradiction from the conjunction of S and $\overline{T}$ (the negation of T), but if '$\overline{T}$' is replaced by the disjunction '$\overline{T} \vee \text{ANS}(\ldots)$', where '$\overline{T}$' expresses the theorem that there exists a positive answer to a given question and where the 'answer literal' contains the subset of the variables in T that express the subject of the question, then deducing a contradiction from S and $\overline{T}$ causes the generation of a set of unit answer clauses that contains the name(s) of the individual(s) that answer the question positively. This method has been employed in question-answering systems for medicine and for physics (see Coles 1972), and has also been extended to the area of 'problem solving' or 'plan formation', where a question takes the form 'How can an initial state A be transformed into a goal state B?' and an answer consists of a sequence of steps or actions, selected from a given repertoire, which a person or a machine can execute in order to effect the desired transformation. An example is the much-discussed 'blind hand' problem of Popplestone (1970), in one version of which there

are two (not necessarily distinct) places termed '*here*' and '*there*' and a mechanical hand with three actions, namely '*pickup*', which causes a randomly selected object at the place of the hand to be held; '*letgo*', which results in the hand being empty; and '*go*', which moves the hand to a place. In the initial state s_0 there are red things and only red things '*here*', and the goal is a state in which at least one red thing is '*there*'. The most straightforward solution is first to '*letgo*' so that the hand will be empty and nothing will be carried '*here*', then '*go here*', then '*pickup*' and finally '*go there*'. If '*here*' and '*there*' are the same place, or if the hand is already '*here*' in s_0, then the solution is still valid though redundant in whole or in part. The problem has proved to be difficult to formulate and solve in first-order predicate calculus, partly because it requires a large number of 'frame axioms' saying that certain states are not changed by certain actions: for example, 'if thing t_1 is at place x_1 in state s_1 and if t_1 is not held in s_1 then t_1 is still at x_1 if the hand goes to x_1 in s_1'. Such axioms greatly increase the computational load on the theorem prover in any attempt to solve this type of problem by machine, but by following a suggestion of Popplestone and expressing the actions of the hand in terms of their effects on the set of '*things at*' a place and on the set of '*things held*', our SNOBOL-4 program obtained a comparatively simple 'higher order' solution in about four minutes of machine time on the IBM 360/50 at the GMD in Bonn:

Machine solution of 'blind hand' problem

(1) $((c_1 \cup c_2) \cap c_3) \neq 0 \vee . (c_1 \cap c_3) = 0 \wedge (c_2 \cap c_3) = 0$

(2) $\neg f(thingsat(x_1, go(x_1, s_1))) \vee f(thingsat(x_1, s_1) \cup thingsheld(s_1))$

(3) $\neg f(thingsheld(pickup(go(x_1, s_1)))) \vee f(anyof(thingsat(x_1, go(x_1, s_1))))$

(4) $\neg f(thingsheld(letgo(s_1))) \vee f(0)$

(5) $(anyof(thingsat(here, s_0)) \cap redthings) \neq 0$

(6) $(thingsat(there, s_1) \cap redthings) = 0 \vee \text{ANS}(s_1)$

(7) $((thingsat(there, s_1) \cup thingsheld(s_1)) \cap redthings) = 0 \vee$
$\text{ANS}(go(there, s_1))$ From (2) & (6)

(8) $(thingsheld(s_1) \cap redthings) = 0 \vee \text{ANS}(go(there, s_1))$ From (1) & (7)

(9) $(anyof(thingsat(x_1, go(x_1, s_1)))) = 0 \vee$
$\text{ANS}(go(there, pickup(go(x_1, s_1))))$ From (3) & (8)

(10) $((anyof(thingsat(x_1, s_1)) \cup thingsheld(s_1)) \cap redthings) = 0 \vee$
$\text{ANS}(go(there, pickup(go(x_1, s_1))))$ From (2) & (9)

(11) $((anyof(thingsat(x_1, letgo(s_1))) \cup 0) \cap redthings) = 0 \vee$
$\text{ANS}(go(there, pickup(go(x_1, letgo(s_1)))))$ From (4) & (10)

(12) $((anyof(thingsat(x_1, s_1)) \cup 0) \cap redthings) = 0 \vee$
$\text{ANS}(go(there, pickup(go(x_1, letgo(s_1)))))$ From (11) & Rule 1

(13) $(anyof(thingsat(x_1, s_1)) \cap redthings) = 0 \vee$
$\text{ANS}(go(there, pickup(go(x_1, letgo(s_1)))))$ From (12) & Rule 2

(14) $\text{ANS}(go(there, pickup(go(here, letgo(s_0)))))$ From (5) & (13)

In the preceding proof, 'x_1' is a place variable; 's_1' is a state variable; 'c_1', 'c_2'

and 'c_3' are set variables; 'f' is a function variable ranging over properties of sets; '*anyof*(c_1)' is the unit set of a thing selected from a set c_1 by a choice function; '*thingsheld*(s_1)' is a set-valued variable always taking as positive value a unit set, and for each x_1 and s_1 there is a set '*thingsat*(x_1, s_1)'. As there is no special predicate '*at*', the information that the hand is at x_1 in s_1 is expressed by the state '*go*(x_1,s_1)', an action which will be redundant if the hand is already at x_1. The required operations on sets are handled by axiom (1) and by Rule 2 which reduces $c_1 \cup 0$ to c_1. The 'frame axioms' do not appear explicitly but only in the form of Rule 1 which reduces '*thingsat*$(x_1,$ *pickup*$(s_1))$' and '*thingsat*$(x_1,$ *letgo*$(s_1))$' to '*thingsat*(x_1,s_1)', saying in effect that '*pickup*' and '*letgo*' do not change the number of things at a place. Axioms (2), (3) and (4) result from resolving the set equalities

$$thingsat(x_1, go(x_1, s_1)) = thingsat(x_1, s_1) \cup thingsheld(s_1)$$
$$thingsheld(pickup(go(x_1,s_1))) = anyof(thingsat(x_1, go(x_1,s_1)))$$
$$thingsheld(letgo(s_1)) = 0$$

which express the properties of the three actions, with the general equality axiom (Darlington 1968)

$$x \neq y \vee \neg f(x) \vee f(y)$$

The 'blind hand' problem itself is presented in the form of axiom (5), which is a concise way of saying that there are red things and only red things '*here*' in s_0, and axiom (6), which says that if there is a state s_1 in which at least one red thing is '*there*' then s_1 is an answer. The program resolves only on first literals of clauses, applying a restrictive strategy shown by Kowalski and Hayes (1969) to be complete for first-order logic, and makes essential use also of the 'set of support' strategy of Wos, Robinson, and Carson (1965), according to which the axioms that do not formulate the negation of the theorem to be proved (or the statement of the problem) are not resolved with each other. This means in terms of our example that (5) and (6), which formulate the problem, are resolved with (1)–(4) but that these initial four axioms are not resolved among themselves. This strategy becomes particularly important if one is applying theorem-proving methods to answer questions or retrieve information from large data bases (see Darlington 1969).

The higher-order aspect of the solution is illustrated by the derivation of (7) from (2) and (6), whose first literals do not match according to the first-order unification algorithm (Robinson 1965) but which can be made to match by doing a property abstraction on the first literal of (6). To begin with, the 's_1' in (6) is replaced by a different variable, namely 's_5', so that the two clauses initially have no variables in common, a condition required by the unification algorithm. Next, the first literal of (2) is negated, again in accordance with the resolution method. The program is therefore trying to match (or 'unify') the two strings

$$(thingsat(there, s_5) \cap redthings) = 0 \qquad L_1$$
$$f(thingsat(x_1, go(x_1, s_1))) \qquad L_2$$

Since these two strings fail to match in the ordinary way, the algorithm enters the '*f*-matching' mode (Darlington 1968, 1971) and the argument of the function variable '*f*' in L_2, namely '$thingsat(x_1, go(x_1,s_1))$', finds a piece of L_1 that it can match, namely '$thingsat(there,s_5)$', under the value assignments

x_1 = '*there*'

s_5 = '$go(there,s_1)$'

The program then replaces the matched substring by a marker '*u*' and converts L_1 into the equivalent form

$$[\lambda u(u \cap redthings)=0]\ thingsat(there,s_5) \qquad L_1'$$

which matches L_2 under the assignment

f = '$\lambda u(u \cap redthings)=0$'

plus those already given for x_1 and s_5. These value assignments are then applied to the disjunction of the remainders of (2) and (6), namely

$$f(thingsat(x_1,s_1) \cup thingsheld(s_1)) \vee \text{ANS}(s_5)$$

to obtain the resolvent (7). The sort of higher-order matching with lambda-conversion illustrated here has also been investigated by Gould (1966), and more recently by Andrews (1971) and Pietrzykowski and Jensen (1972) specifically in connexion with resolution-type theorem proving. The last two authors have established the completeness of their unification algorithm for all orders of the functional calculus, while Huet (1972) and Lucchesi (1972) have proved the undecidability of the unification problem for languages of order higher than two. These results are of course not contradictory: if two strings are unifiable the Pietrzykowski-Jensen algorithm will find all of the (possibly infinite) unifiers, while if they are not unifiable then this cannot invariably be known. The '*f*-matching' procedure employed by our program is not a complete algorithm. It attempts to generate matches between two strings, at least one of which contains a functional expression of the form '$f(x_1, \ldots, x_n)$', by matching the $x_1, \ldots, x_n$ in turn in all possible ways with the subexpressions of the other string, performing a lambda-abstraction whenever a match is successful. The unifications thereby produced are some of the more obvious ones, and are a subset of those generated by the 'imitation' and 'projection' rules of Pietrzykowski and Jensen.

It should be pointed out that, since the axioms in the preceding proof that contain function variables all result from equality statements, the proof could also be accomplished by a first-order theorem prover with an equality rule such as 'paramodulation' (Robinson & Wos 1969, also employed by Hoffmann & Veenker 1971). The advantages of higher-order logic become more apparent when theorem provers are applied to the generation of plans or programs with loops, primarily because of the close relation between loop-generation and a principle of mathematical induction (Manna & Waldinger 1971). A good starting point is Burstall's (1969) example of a program that computes 2^n since it contains a loop that is shown to succeed for any n, the proof being formulated in first-order logic with an appropriate instance of the induction axiom. The problem of generating such a program was

submitted to our theorem prover with appropriate axioms for exponentiation, assignment and loop generation, with the following result:

Generate a program to compute 2^n

(1) $\neg f_1(x_1^{x_2+1}) \vee f_1(x_1^{x_2} \,.\, x_1)$

(2) $\neg Hasval(x_1, x_3, s_1) \vee Hasval(x_2, f_1(x_3), ((x_2 := f_1(x_1))s_1))$

(3) $Hasval(x_1, f_1(x_2), ((Loop(f_2((J:=J+1)(J \neq x_2))))((x_1:=f_1(0))$
$((J:=0)s_1))))$
$\vee \,.\, Hasval(x_1, f_1(m), s_c) \wedge \neg Hasval(x_1, f_1(m+1), f_2(s_c))$

(4) $\neg Hasval(K, 2^n, s_2) \vee \text{ANS}(s_2)$

(5) $Hasval(K, 2^m, s_c) \vee$
$\text{ANS}((Loop(f_2((J:=J+1)(J \neq n))))((K:=2^0)((J:=0)s_1)))$
From (3) & (4)

(6) $\neg Hasval(K, 2^{m+1}, f_2(s_c)) \vee$
$\text{ANS}((Loop(f_2((J:=J+1)(J \neq n))))((K:=2^0)((J:=0)s_1)))$
$x_1 = K, x_2 = n, f_1 = \lambda u \,.\, 2^u,$ From (3) & (4)
$s_2 = ((Loop(f_2((J:=J+1)(J \neq n))))((K:=2^0)((J:=0)s_1)))$

(7) $Hasval(x_2, f_1(2^m), ((x_2:=f_1(K))s_c)) \vee$
$\text{ANS}((Loop(f_2((J:=J+1)(J \neq n))))((K:=2^0)((J:=0)s_1)))$
$x_1 = K, x_3 = 2^m, s_1 = s_c$ From (2) & (5)

(8) $\neg Hasval(K, 2^m \,.\, 2, f_2(s_c)) \vee$
$\text{ANS}((Loop(f_2((J:=J+1)(J \neq n))))((K:=2^0)((J:=0)s_1)))$
$x_1 = 2, x_2 = m,$ From (1) & (6)
$f_1 = \lambda u \,.\, \neg Hasval(K, u, f_2(s_c))$

(9) $\text{ANS}((Loop((K:=K \,.\, 2)((J:=J+1)(J \neq n))))((K:=1)((J:=0)s_1)))$
$x_2 = K, f_1 = \lambda u \,.\, (u \,.\, 2),$ From (7) & (8)
$f_2 = \lambda u \,.\, ((K:=K \,.\, 2)u)$

In the preceding proof, the variables 'x_i' range over SNOBOL-type character strings, where an integer string is handled in the same way as the corresponding integer; the variables 's_i' range over states; the variables 'f_1' and 'f_2' range over functions of strings and of states, respectively; 'J' and 'K' are particular strings that serve as program variables; ':=' denotes assignment; '$Hasval(x_1, x_2, s_1)$' means that the string denoted by 'x_1' has as value the string denoted by 'x_2' in state s_1, and '$((Loop(f_2((J:=J+1)(J \neq x_2))))$ $((x_1:=f_1(0))((J:=0)s_1)))$' denotes the state that results from executing the loop shown in figure 1.

Axiom (1) results from the equality

$$x_1^{x_2+1} = x_1^{x_2} \,.\, x_1$$

Axiom (2) is a general assignment axiom, (4) formulates the terms of the problem, and (3) is a loop axiom based on the inductive argument

$Hasval(K, f_1(J), ((J:=0)s_1))$

$Hasval(K, f_1(J), s_2)$ *implies* $Hasval(K, f_1(J), f_2((J:=J+1)s_2))$

$/\!\therefore Hasval(K, f_1(n), ((Loop(f_2((J:=J+1)(J \neq n))))((J:=0)s_1)))$

which states that if K has value $f_1(J)$ when J is assigned 0 in s_1, and if the

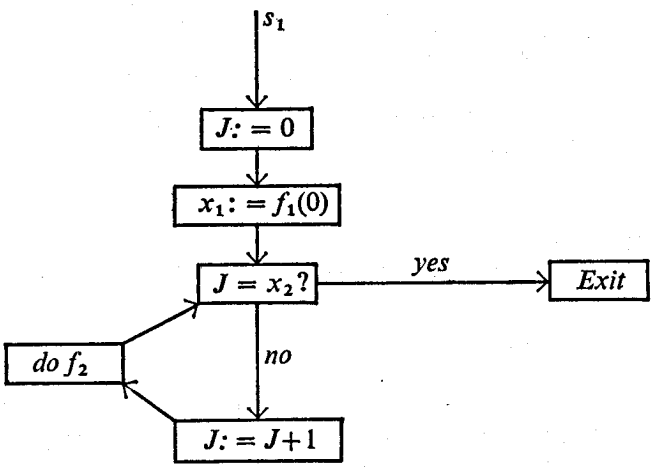


Figure 1. Loop introduced by Axiom (3).

assumption that K has value $f_1(J)$ in any state s_2 implies that K has value $f_1(J)$ when a string of operations denoted by f_2 is performed after augmenting J by 1 in s_2, then K has value $f_1(n)$ if the operations $f_2(J:=J+1)$ are performed n times, that is, until J has value n, starting in state $((J:=0)s_1)$ (strictly speaking, the test condition '$J\neq n$' should be expressed as '$\neg Hasval(J, n, s_i)$', where 's_i' denotes the state of having gone round the loop an arbitrary number of times). The first premiss may be made true merely by assignment, that is, by stipulating that the loop is to be entered in state $((K:=f_1(0))$ $((J:=0)s_1))$, transforming the argument into

$$Hasval(K,f_1(J),s_2)\ implies\ Hasval(K,f_1(J),f_2((J:=J+1)s_2))$$
$$/\therefore Hasval(K,f_1(n),((Loop(f_2((J:=J+1)(J\neq n))))$$
$$((K:=f_1(0))((J:=0)s_1))))$$

or in prenex form

$$\forall x_1, x_2, s_1, f_1, f_2 \exists s_2$$
$$[Hasval(x_1,f_1(J),s_2)\ implies\ Hasval(x_1,f_1(J),f_2((J:=J+1)s_2))]$$
$$implies$$
$$Hasval(x_1,f_1(x_2),((Loop(f_2((J:=J+1)(J\neq x_2))))$$
$$((x_1:=f_1(0))((J:=0)s_1))))$$

Before Skolemising, the antecedent of the implication is replaced by

$$Hasval(x_1,f_1(J-1),((J:=J+1)s_2))\ implies$$
$$Hasval(x_1,f_1(J),f_2((J:=J+1)s_2))$$

The state expression '$((J:=J+1)s_2)$', which contains an existentially quantified variable, is then replaced by an arbitrary constant 's_c', and 'J' is replaced by '$m+1$' (so if 'm' appears in the answer it must be replaced by '$J-1$'). The quantifiers are then dropped, and the resulting formula is put into the resolution-type notation required by the program.

Some further problems which the program has solved include constructing a program to compute n^2 by summing the first n odd integers, constructing a program (suggested by an example of Robinson 1969) to compute n^2+n by summing the doubles of the first n integers, and constructing a program (see Manna & Waldinger 1971) to express m as $J \,.\, n+K$ where $K<n$ (the last example required the substitution of '$\neg(K<n)$' for '$J\neq n$' as the test condition in the loop axiom). Because of the restrictive strategies employed and the involved nature of most of the unifications not many irrelevant clauses were generated and retained during these proofs, though the unifications themselves were time-consuming and, as pointed out by Pietrzykowski and Jensen (1972), there is a clear need for research into the heuristics of higher-order unification. Additional areas for research include increasing the number of input variables, generating loops within loops, and having the program automatically construct the test condition for each loop.

In related work, Manna and Waldinger (1971) also use inductive principles to aid in the generation of loops, though their axioms are mainly forms of the least number principle or 'strong induction' as opposed to our reliance on 'weak induction'. The question of higher-order logic is avoided, partly by not presupposing any particular method of mechanical theorem proving and partly by careful selection of the relevant instances of the inductive axioms, though if one works through some of their examples in detail it is clear that some form of higher-order unification is being done. In the domain of robot problem solving, Fikes, Hart and Nilsson (1972) have written a program called STRIPS in which a distinction is made between the production of new 'world models' by applying operators to existing models, and the search for a proof that a given model satisfies the goal condition or that a given operator is applicable. In Green's approach (and ours) both tasks are performed by the resolution-based theorem prover, while in STRIPS the first task is performed by a GPS-type search (Ernst and Newell 1969) and the theorem prover is applied only to the second; significant reductions in search time are foreseen where the number of operators is large, though one of the STRIPS examples with few operators was solved by our program in a manner similar to the 'blind hand' problem. In practice, of course, one must normally expect that many different operators will be applicable to a given node, so some method of selection such as that employed by STRIPS becomes essential; but an important question to consider is to what extent it is necessary to look outside the set of search strategies designed by theorem provers to improve the efficiency of resolution-based programs. Many of these strategies seem in fact to have reappeared in different guise in the PLANNER language for problem solving (Hewitt 1970, 1971), whose advocates nonetheless propose it as a radical alternative to the allegedly inefficient 'uniform proof procedures'. Of the three main special features of the language that are relevant to deductive problem solving, 'backtracking' has recently been de-emphasised by PLANNER advocates and 'pattern matching' is merely a different way of

talking about unification; but the third and perhaps most important feature is PLANNER's way of expressing an axiom such as 'all humans are fallible' as a 'procedure', that is

⟨CONSEQUENT (Y) (FALLIBLE ?Y)⟨GOAL (HUMAN ?Y)⟩⟩

instead of as a disjunctive clause in predicate calculus, that is

$\neg Human(y) \vee Fallible(y)$

the object being to allow such a premiss to be called into play only when it is desired to prove that some particular thing is fallible, and to avoid generating for every statement of the form '*y is human*' the consequence that *y* is fallible. But resolution theorem provers achieve the same restrictive end either by the 'set of support' strategy (Wos *et al.* 1965), according to which two statements in the data base, such as '*Human*(*Socrates*)' and the above axiom, are never resolved with each other, or by an ordering of literals, or by a 'selection function', as in the SL-resolution method of Kowalski and Kuehner (1970), that singles out a particular literal in a given clause to be resolved on. Thus, there seems no good reason for preferring PLANNER to a theorem-proving approach to plan formation, though much work remains to be done on the development of heuristic search strategies for both first- and higher-order resolution.

REFERENCES

Andrews, P.B. (1971) Resolution in type theory. *Journal of Symbolic Logic*, 36, 414–32.

Burstall, R.M. (1969) Formal description of program structure and semantics in first order logic. *Machine Intelligence 5*, pp. 79–98 (eds Meltzer, B. & Michie, D.). Edinburgh: Edinburgh University Press.

Coles, L.S. (1972) The application of theorem proving to information retrieval. *Proc. 5th Hawaii International Conference on System Sciences*, Honolulu, Jan. 12–14 1972.

Darlington, J.L. (1968) Automatic theorem proving with equality substitutions and mathematical induction. *Machine Intelligence 3*, pp. 113–27 (ed. Michie, D.). Edinburgh: Edinburgh University Press.

Darlington, J.L. (1969) Theorem proving and information retrieval. *Machine Intelligence 4*, pp. 173–81 (eds Meltzer, B. & Michie, D.). Edinburgh: Edinburgh University Press.

Darlington, J.L. (1971) A partial mechanization of second-order logic. *Machine Intelligence 6*, pp. 91–100 (eds Meltzer, B. & Michie, D.). Edinburgh: Edinburgh University Press.

Ernst, G.W. & Newell, A. (1969) *GPS: A Case Study in Generality and Problem Solving*. New York and London: Academic Press.

Fikes, R.E., Hart, P.E., & Nilsson, N.J. (1972) Some new directions in robot problem solving. *Machine Intelligence 7*, paper no. 23 (eds Meltzer, B. & Michie, D.). Edinburgh: Edinburgh University Press.

Gould, W.E. (1966) A matching procedure for omega-order logic. Sci. Rep. No. 4. AFRCL 66–781. Applied Logic Corp., Princeton.

Green, C.C. (1969) Theorem-proving by resolution as a basis for question-answering systems. *Machine Intelligence 4*, pp. 183–205 (eds Meltzer, B. & Michie, D.). Edinburgh: Edinburgh University Press.

Hewitt, C. (1970) *PLANNER: A language for manipulating models and proving theorems in a robot*. AI Memo 168, Project MAC, MIT.

Hewitt, C. (1971). Procedural embedding of knowledge in PLANNER. *Proc. 2nd IJCAI*, London. Portsmouth: Eyre & Spottiswoode Ltd.

Hoffmann, G.-R. & Veenker, G. (1971) The unit-clause proof procedure with equality. *Computing* **7**, 91–105.

Huet, G.P. The undecidability of the existence of a unifying substitution between two terms in the simple theory of types. Report 1120, Case Western Reserve University, February 1972.

Kowalski, R. & Hayes, P.J. (1969) Semantic trees in automatic theorem-proving. *Machine Intelligence 4*, pp. 87–101 (eds Meltzer, B. & Michie, D.). Edinburgh: Edinburgh University Press.

Kowalski, R. & Kuehner, D. (1970) Linear resolution with selection function. Memo 34, Metamathematics Unit, University of Edinburgh.

Lucchesi, C.L. (1972) The undecidability of the unification problem for 3rd order languages. Report CSRR–2059, University of Waterloo, February 1972.

Manna, Z. & Waldinger, R.J. (1971) Toward automatic program synthesis. *Comm. ACM* **14**, 151–65.

Pietrzykowski, T. & Jensen, D.C. (1972) A complete mechanization of omega-order logic. Report CSRR–2060, University of Waterloo, February 1972.

Popplestone, R.J. (1970) Using Floyd comments to make plans. *Experimental Programming* 1970. Department of Machine Intelligence, University of Edinburgh.

Robinson, G.A. and Wos, L. (1969) Paramodulation and theorem-proving in first-order theories with equality. *Machine Intelligence 4*, pp. 135–50 (eds Meltzer, B. & Michie, D.). Edinburgh: Edinburgh University Press.

Robinson, J.A. (1965) A machine-oriented logic based on the resolution principle. *JACM* **12**, 23–41.

Robinson, J.A. (1969) Mechanizing higher-order logic. *Machine Intelligence 4*, pp. 151–70 (eds Meltzer, B. & Michie, D.). Edinburgh: Edinburgh University Press.

Wos, L., Robinson, G.A. & Carson, D.F. (1965) Efficiency and completeness of the set of support strategy in theorem proving. *JACM* **12**, 536–41.

INFERENTIAL AND HEURISTIC SEARCH

9

G-deduction

D. Michie, R. Ross and G. J. Shannan
Department of Machine Intelligence
University of Edinburgh

INTRODUCTION

The search procedure to be described has been used for theorem proving in first order logic. But the motivation of the work is concerned with automatic problem solving more generally.

It is often convenient to subdivide a problem-solving procedure into (1) a *problem-representation*, defining the space to be searched, and (2) a *search strategy*. Kowalski (1969) has given formal expression to this subdivision in the special case of resolution theorem-proving. Progress with (1) promises the larger reward. But even at such future time that significant progress with (1) can be demonstrated, the need for good search strategies will remain.

In this paper we first define a strategy, GPST, which combines the sub-goaling principles of GPS (Ernst and Newell 1969) with the heuristic guidance mechanisms of GT4 (Michie and Ross 1969). We then propose an extension, the *G*-procedure. Finally we specialise it to the theorem-proving domain, using Robinson's (1965) P_1-resolution, to obtain *G*-deduction.

Consider binary derivation systems (Michie and Sibert 1973) comprising objects of just two types, operands and operators. There is an initial set of operands $X \subseteq \mathscr{X}$ and an initial set of operators $G \subseteq \mathscr{G}$. The result of *apply* (x, g) where $x \in \mathscr{X}$ and $g \in \mathscr{G}$ is either undefined or it is a member of $\mathscr{X}$ or it is a member of $\mathscr{G}$.

Classical graph-search algorithms, including GPS and GT4, demand that *apply* (x, g), if defined, be a member of $\mathscr{X}$, so that the initial repertoire of operators remains fixed. This restriction also characterises GPST. Moreover, GPST is identical with GT4 so long as *apply* (x, g) is always defined. But when an operator is selected for an operand to which it is not applicable, the various algorithms behave differently from each other, as follows.

GT4 aborts the application and selects a fresh operator.

GPS and GPST set up (x, g) as a sub-problem, the goal of which is to derive an x' such that *applicable* (x', g).

G-procedure sets up (x, g) as a sub-problem, the goal of which is to

derive *either* an x' such that *applicable* (x', g) *or* a g' such that *applicable* (x, g').

These distinctions appear in the following outline definitions, which have been framed recursively for perspicuousness. We denote by the symbol *gt* our schema for GT4, and we express the goal set as a predicate.

GT4. Define a set of 'developed' nodes X_{dev}; *developed* (x) is true iff all operators in G have been applied to x. X_{dev} is initialised to the unit set $\{undefined\}$. A set of 'candidate' nodes, X_{cand}, is initialised to $\{x_0\}$.

let *gt* (*node, apply, eval, select, goalpred*) **be**
if *goalpred* (*node*) **then** *node*
else assign *select* ($x_{\min}$, *operatorset, goalpred*) **to** *op*;
assign *apply* ($x_{\min}$, *op*) **to** *newnode*;
if *developed* ($x_{\min}$) **then** *move* $x_{\min}$ *from* X_{cand} *into* X_{dev};
assign $X_{\text{cand}} \cup \{newnode\}$ **to** X_{cand};
gt (*min* (X_{cand}, *eval*), *apply, eval, select, goalpred*);
and let *apply* (x, g) **be**
if *applicable* (x, g) **then** $g(x)$ **else** *undefined.*

eval estimates the expected cost of the path leading to the nearest member of *goalpred.*

GPS. The sub-problem mechanism can be illustrated schematically by defining a pair of mutually recursive procedures:

let *preprocess* (*node, operator*) **be**
if *applicable* (*node, operator*) **then** (*node, operator*)
else *preprocess* (*apply* (*node, select*(*node, operatorset, operator*)), *operator*)
and let *apply* (*node, operator*) **be**
if *applicable* (*node, operator*) **then** *operator* (*node*)
else *apply* (*preprocess* (*node, operator*))

A call of *preprocess* (*node, operator*) transforms the given node into one satisfying the applicability conditions of *operator.* At top level we supply as actual parameter a notional 'goal operator' applicable to a node x iff *goalpred* (x). Note that search only occurs via sub-problem formation. For it to be branching, we need a failure condition in *apply* for returning control to *select.*

GPST. By a small modification of the above we can combine the benefits of both approaches – on the assumption that when *applicable* is false, search should proceed by sub-problem formation, but that when *applicable* is true it should proceed by heuristically guided graph-traversing.

let *preprocess* (*node, operator*) **be**
if *applicable* (*node, operator*) **then** (*node, operator*)
else (*gt* (*node, apply, eval, select, domainof*(*operator*)),*operator*)
and (as before) **let** *apply* (*node, operator*) **be**
if *applicable* (*node, operator*) **then** *operator* (*node*)
else *apply* (*preprocess* (*node, operator*)).

Search occurs by sub-problem formation on those calls of *apply* at which *applicable* fails, and by graph-traversing whenever *applicable* holds. *domainof*(g) yields a predicate which when applied to x gives the value *true* iff *applicable* (x, g).

GPST, however, still suffers the limitation of a fixed operator set, so that the only response to 'g will not apply to x' is 'construct an x' to which g will apply'. For full generality we would like the option 'construct a g' which will apply to x'. Minor modification gives the required generalised procedure.

G-procedure

let *preprocess* (*node*, *operator*) **be**
if *applicable* (*node*, *operator*) **then** (*node*, *operator*)
else if *gpst* **then** (*gt* (*node*, *apply*, *eval*, *select*, *domainof*(*operator*)), *operator*)
else (*node*, *gt* (*node*, *apply*, *eval*, *select*, *opof*(*node*)))
and let *apply* (*node*, *operator*) **be**
if *applicable* (*node*, *operator*) **then** *operator* (*node*)
else *apply* (*preprocess* (*node*, *operator*))

opof(x) yields a predicate which when applied to g gives *true iff applicable* (x, g)

At a yet higher level of generality are the *H*-procedures of Michie and Sibert, which present the search as a single-level process without specifically invoking recursion. The conceptual advantage for some purposes of the recursive scheme lies in the fact that *eval* is defined with reference to a goal set; on each call of *gt*, *eval* is redefined with reference to the sub-goal associated with the given sub-problem. The global estimator relevant to the monkey's expected residual labour required to get the bananas is kept cleanly segregated from, say, the estimator local to the subgoal of finding the chair. The *H* evaluation function has the entire derivation tree as one of its arguments, so that any of the foregoing search procedures can be implemented as an *H*-procedure by suitably re-defining this function. But nothing would be gained in the present instance, and indeed we now turn to a specialisation of the *G*-procedure. Just as it yields GPST when the value of the switch variable *gpst* is fixed at *true*, so by fixing it at *false* we obtain the search strategy of *G*-deduction.

DESCRIPTION OF THE PROCEDURE

G-deduction is a strategy for automatic theorem proving in the first order predicate calculus based on P_1-resolution (Robinson, 1965). The P_1 inference system imposes the restriction that one of the two predecessors of each resolvent contains no negative literals. Therefore every P_1-resolvent contains fewer negative literals than does its non-positive predecessor, so that a process of 'shrinking' is involved. So long as only unit clauses are used as the positive predecessor shrinkage is absolute, and indefinite iteration of such a process

must lead to the generation of the empty clause. The general idea is thus to select initially a clause containing *only* negative literals (and if the initial set of clauses is unsatisfiable we are guaranteed that there will be at least one such clause in the set) and to use this as the root node from which to grow a search tree by repeated applications of the P_1-operation. If the initial set contains more than one wholly-negative clause, then make each the root of a separate tree to be grown by 'time-sharing' among the searches.

Search-tree form has obvious attractions, since a proof procedure might hope to benefit from established techniques of heuristic search. These techniques, however, base themselves on a graph model of problem solving in which directed arcs of the graph represent unary operators. At first sight, resolution seems difficult to adapt to this framework, since it is a binary operation. The approach adopted here is to regard the *non-positive* predecessor of a P_1-resolvent as playing the role of a problem state, corresponding to a node on the problem graph, and the *positive* predecessor as playing the role of a unary operator, and hence corresponding to a branch of the search-tree. So long as there exists some non-positive clause on the tree for which it is possible to find a positive clause with which it will resolve, growth of the tree can continue. The behaviour of the *G*-deduction algorithm when this is *not* possible is its crucial feature; we shall defer description of this for the moment, except to remark that when search for a suitable pre-existing positive clause fails it is not unreasonable to think of continuing it by seeking to *deduce* a new positive clause with the required properties. In other words, 'if you haven't got an operator, make one!'. There is an echo here of GPS and of STRIPS (Fikes and Nilsson 1971). In these cases the recursive search is for a state to fit an operator, rather than the other way round.

The illustrative problems selected in this paper for application of *G*-deduction include two cases where the base set of clauses are represented in a special form based upon that of Green (1969). There is, however, no lack of generality of the procedure; on the contrary, its domain of applicability is co-extensive with that of hyper-resolution, as will be shown.

We now briefly describe this special form, which imposes certain constraints on conjunctive normal form so as to express problems of reasoning about situations and actions in the context of robot planning. In *G*-deduction one clause, the negation of the conjectured theorem, is $\{\overline{ANS}(S)\}$, and we take this as a root clause. S is a 'situation' variable used in the axiomatic definitions of certain special functions referred to as 'actions'. These functions are such that expressions formed from them are substitutable for S. The remaining clauses constitute a consistent set, and one (or more) of them contains just one occurrence of the predicate letter ANS as a constituent of a positive literal, that is, is a clause which defines the 'goal' or 'answer' situation.

The object of the procedure to be described is to detect inconsistency in a set of clauses taking this form. *G*-deduction ensures that proof of the theorem

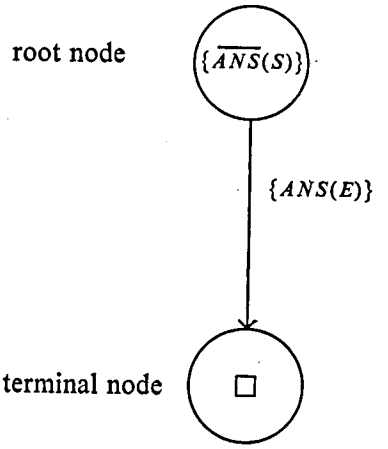


Figure 1. A search tree of two nodes only, corresponding to the root clause and the empty clause. The single arc is labelled with the corresponding 'operator', the positive clause $\{ANS(E)\}$.

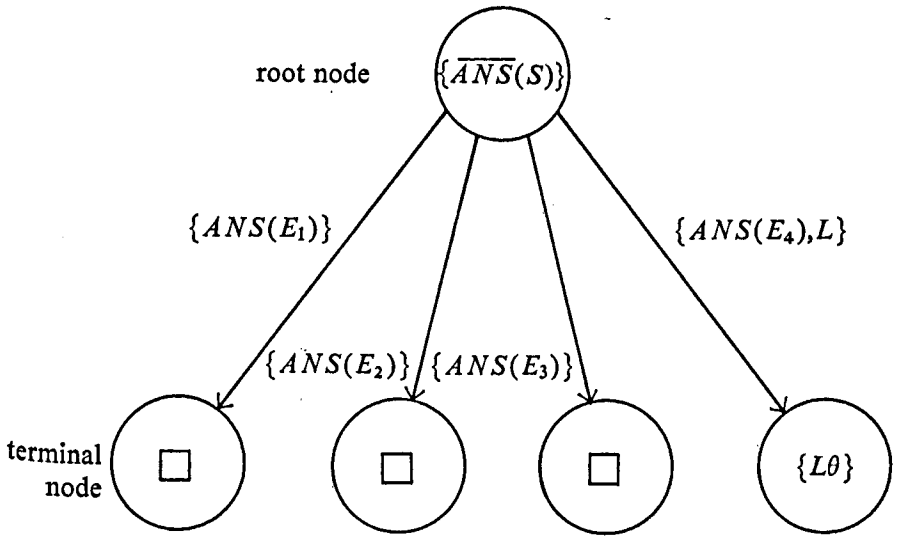


Figure 2. A search tree in which, in contrast to figure 1, the computation only halts when *all* possible distinct 'top-level' deductions of the empty clause have been made. These involve different instantiations of the variable S, that is, applications of different 'P_1-operators'. The corresponding arcs thus bear different labels and can be thought of as coloured. By 'top level' we mean 'possible by the P_1-application of positive clauses already present in the initial set of clauses'. L denotes a disjunction of one or more positive literals, and θ denotes a substitution.

by deduction of the empty clause must proceed by first deducing a unit clause $\{ANS(E)\}$ where E is some expression substituted for S during the deduction. The search tree mentioned above can be depicted, at 'top level', as in figure 1.

This representation oversimplifies the situation in two ways:
(1) Since more than one positive clause may resolve with a given negative clause, the search tree is essentially branching;
(2) non-unit clauses, used as P_1-operators, produce non-empty terminal successors of the root clause. These positive clauses are added to the base set.

Both the above features are illustrated in figure 2.

In the application considered by Green, E itself (or E_1, E_2, . . ., etc., in the case of multiple 'answers') is the main object of interest, and takes the form of a composition of action functions. Hence it can be interpreted as a 'plan of action'. An arrangement must be made to preserve it as an 'answer' constructed by the proof.

The algorithm

We shall briefly describe the constituents of Robinson's hyper-resolution as the basis from which the G-deduction algorithm will be developed. First we establish nomenclature for use in this paper which differs somewhat from that of Robinson's paper.

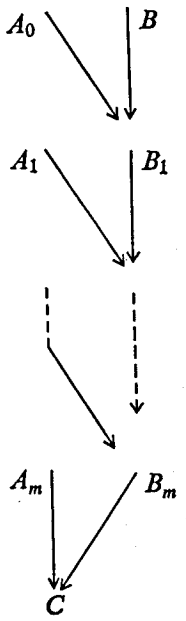


Figure 3. Robinson's 'figure 4' showing a P_2-deduction as a sequence of P_1-resolutions, the sequence having the input-output properties stated in the text.

Positive clause: clause containing only positive literals (such clauses will be referred to hereafter as '*A*-clauses');
negative clause: clause containing only negative literals;
mixed clause: clause containing both positive and negative literals;
non-positive clause: clause containing at least one negative literal (such clauses will be referred to hereafter as '*B*-clauses');
non-negative clause: clause containing at least one positive literal;
□: clause containing no literals.

As the key to an informal definition of hyper-resolution we reproduce in figure 3 Robinson's 'figure 4', in which a succession of P_1-resolutions is shown, starting with a positive and a non-positive clause (A_0 and B respectively in the diagram) as the two initial inputs and producing a single positive clause (C in the diagram) as the eventual output. Such a complete sequence is termed a 'P_2-deduction'. Note that the input clause B does not uniquely determine a single P_2-deduction: other choices of unifiable positive clauses may be possible. In such cases there will be alternative branches for the computa-

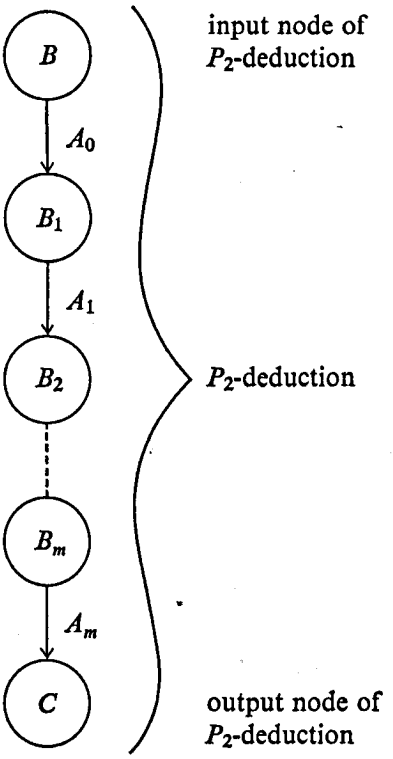


Figure 4. The relation between P_1-resolution and P_2-deduction. The *B*-clauses are all non-positive; the *A* clauses and *C* are positive. The scheme of figure 3 has been re-cast to express the 'states and operators' approach of this discussion.

tion, leading to alternative outputs. In order more easily to depict such branching structures, and to express P_1-deduction as a unary operation along the lines proposed earlier, we have re-drawn this scheme in figure 4 in accordance with the conventions of figure 1 and 2.

In figure 4, a single positive clause is shown as the output of P_2, but as stated previously, the output is in reality a *set* of clauses: (1) a given A and B may give more than one P_1-resolvent and (2) more than one A may be available to resolve against a given B. Associated with this is the idea of branching at each constituent P_1-deduction, so that the P_2-deduction takes the form not of a chain but of a tree. Such a tree, which we term a 'P_2-tree', is illustrated in figure 5, and can be related to figure 2 which depicts the special case where B is a unit negative clause and C is the empty clause.

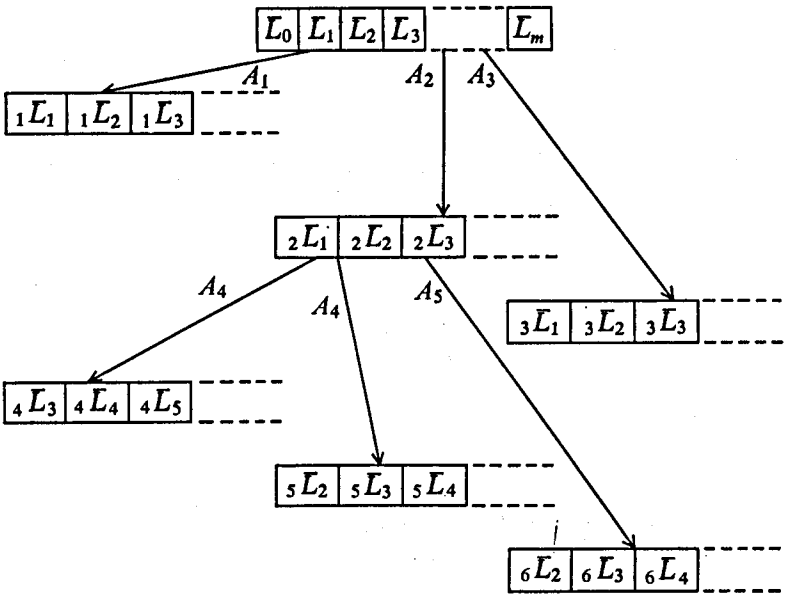


Figure 5. A P_2-tree. Left-hand subscripts distinguish different instantiations of a given negative literal. If the next node to be expanded is the one representing the clause with fewest negative literals, we get 'depth first' search; if it is always the one corresponding to the clause with most negative literals, then 'breadth first' search. Note that the number of negative literals lessens at each level down the tree. A case is shown (use of positive clause A_4) in which two literals are stripped in a single P_1-deduction, and we have also indicated the possibility (again A_4) that a single positive clause may generate more than one P_1-resolvent.

We now relate the scheme of figure 5 to the hyper-resolution algorithm presented by Robinson. To compute the hyper-resolution $\bar{R}(S)$ of a finite set of clauses, this algorithm computes the sets B_j and A_j as, for $j=0, 1, 2 \ldots$, respectively, the set of all non-positive P_1-resolvents of S_j, which are not

mere instances of members of S_j. S_{j+1} is formed as $S_j \cup B_j$, and the process is iterated until no new non-positive P_1-resolvents are produced, that is, B_j is empty and $S_{j+1}=S_j$. At this point, the positive P_1-resolvents collected from successive cycles, are available as the output of $\tilde{R}(S)$, namely as the set $A=A_0 \cup A_1 \ldots \cup A_j$. Now consider what structures exist on exit from $\tilde{R}$, in terms of figure 5 and the earlier diagram. The set S_j can be identified with the B-nodes of a *set of P_2-trees*. The set has N_B members where N_B is the number of B clauses in the initial set of clauses, S_0, since each such clause is the root of a P_2-tree. By a P_2-tree we mean a structure such as that of figure 5, if grown to completion so that all terminal nodes represent clauses which are either positive or null. Three forms are possible for a P_2-tree which cannot be further grown by application of operators from the set A:

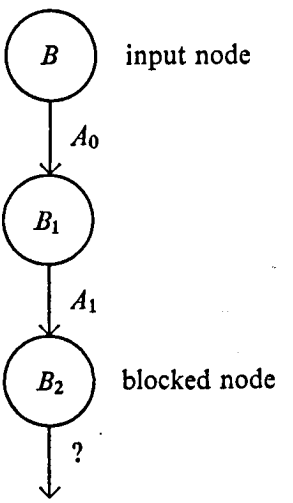


Figure 6. An incomplete P_2-deduction. The process shown in figure 4 is halted through lack of an A-clause of suitable form to resolve against B_2. The node marked B_2 is thus 'blocked'. In later cycles of hyper-resolution (see text) such an A-clause may be generated as the output of some other P_2-deduction, so that the halted process may be resumed. As with figure 4, the P_2-deduction is presented as a non-branching tree. A branching P_2-tree, comprising a set of incomplete P_2-deductions, is shown in figure 7.

(1) All branches lead to an A-clause (each of which is at a given stage transferred to the base set or to $\square$). This is a complete P_2-tree.
(2) Some terminal nodes represent A-clauses, while others represent B-clauses for which no unifiable A-clause yet exists. Such nodes are called 'blocked'. Again, at a given stage its positive nodes are added to the base set. Such a P_2-tree, as also (3) below, is called incomplete.
(3) All terminal nodes are 'blocked'. This form of incomplete P_2-tree

corresponds to a set of incomplete P_2-deductions; – all the P_1-chains have halted for lack of A-clauses suitable for operating on the terminal B-clause of each chain. The concept is illustrated in figures 6 and 7, and assumes importance in the later description of G-deduction. Every such halted deduction represents a potential point of resumption, should suitable A-clauses become available. In classical hyper-resolution, resumptions are in effect enabled if and when later applications of $\tilde{R}$ generate appropriate A-clauses. To complete our account of the classical scheme, the output of $\tilde{R}$ is added to the input set, thus: $\tilde{R}(S)=S \cup A$ and $\tilde{R}$ is iterated on the augmented set. Robinson's fundamental result states that if the finite set S_0 of clauses is unsatisfiable, then, for some $n \geqslant 0$ $\tilde{R}^n(S_0)$ contains $\square$.

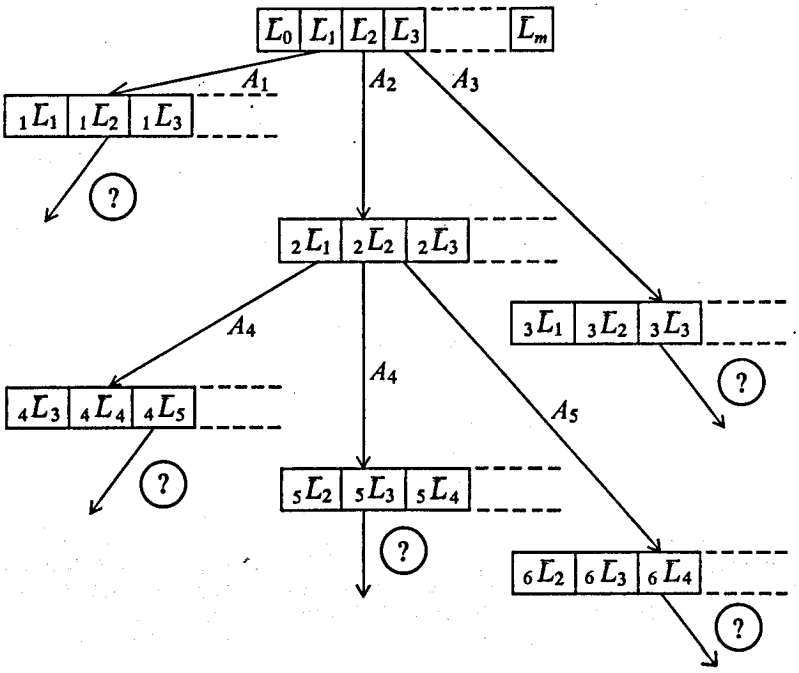


Figure 7. An incomplete P_2-tree. The arcs marked with queries correspond to the 'missing operators' for lack of which the growing of the tree is held up. The nodes from which these arcs originate are 'blocked' nodes.

The idea, which will now be described in more detail, is that for a given incomplete P_2-deduction it may be more advantageous actively to call for a sub-deduction to supply a missing operator rather than to wait passively for its eventual appearance as a by-product of some subsequent hyper-resolution; further, that there is no need to initiate any P_2-deduction until and unless it is called for.

G-deduction

A *G*-deduction is a *recursive* branching P_2-deduction. Recursion is used when the P_2 process requires an *A*-clause with which to perform its next operation of node-expansion and none is available. Associated with each *B*-node is a list of *potential* operators. These take the form of clauses, each of which contains a positive literal unifiable with the leftmost negative literal of the given *B*-clause. We shall call such a list 'the *A*-list of the *B*-clause', and take this to include actual *A*-clauses as a special case. If the potential *A*-clauses are ordered in increasing number of their negative literals, so that actual *A*-clauses head the list, then so long as the search for an operator does not exhaust this positive sequence matters will proceed as in figure 5. We are assuming for the moment that *A*-clauses are always *unit* positive clauses. Other *A*-clauses are given much lower priority on the list, but this complication is irrelevant to the present discussion. But if the positive sequence is exhausted, *G*-deduction sets up as a sub-problem the task of stripping the negative literals from the next clause on the list, so as to create a new positive clause for use by the main P_2 process. All positive clauses thus created are side-effected, at a stage to be defined later, into the base set. Whenever an *A*-list is consulted it is first updated from this pool.

The sub-problem evidently has identically the same form as the main problem, namely to remove all negative literals from its root clause, and is attacked by recursive application of *G*-deduction. The recursion can go to arbitrary depth, and can itself be controlled heuristically just as can the individual tree-growing processes at the various recursive levels. Indeed, it *must* be controlled, whether heuristically or not, with some element of depth-limitation, so as to ensure the completeness of *G*-deduction (see below). As to the control of the individual tree-growths, two terms of the heuristic function suggest themselves: (1) the number of literals in the clause; (2) the number of negative literals in the clause's next potential *A*-clause. The second relates to the amount of preliminary work to be done before a new descendant can be generated from the *B*-clause in question. It measures the maximum tree-depth of the node's immediate junior: the actual depth is of course only ascertainable by calling for the corresponding P_2-deduction.

Hyper-resolution as a growth-cycle of P_2-trees

A picture can thus be presented of the hyper-resolution algorithm as proceeding by successive growth bursts in a set of P_2-trees alternating with the collection of successive harvests of 'leaves' (*A*-clauses) from the new terminal nodes. In Robinson's algorithm, all trees die back to the root at the end of each cycle and are re-grown at each fresh cycle; in addition, successive harvests of *A*-clauses are pooled, and in each cycle every *A*-clause in the pool is tried for P_1-resolution against every new node as it appears. The tree-growing itself proceeds 'breadth-first' in every tree, so that first every node of depth 1 from its proper root is generated, then every node at depth 2, etc. This process

continues until all terminal nodes either represent A-clauses or are 'blocked'. The new A-clauses harvested then represent the output of the given application of $\tilde{R}$.

Revisions of this scheme aimed at efficient machine implementation might preserve the incomplete P_2-trees between cycles, and more generally would avoid the application in different cycles of the same A-operator to the same B-node. But a more fundamental source of wasteful computation is present in the $\tilde{R}^n$ algorithm, in that *all the P_2-trees are treated as having equal status.* In each cycle all growth that *can* be promoted in any of these trees *is* promoted. Recall that there are N_B such trees, each rooted in a different B-clause of the initial set S. The succession of $\tilde{R}$ cycles only terminates when a 'leaf' is produced which is $\square$. Yet it can be seen at the outset that only some trees are capable of producing such a leaf – the ones with wholly negative roots (at least one such must exist if S is unsatisfiable). Growth of other trees serves a subsidiary function: their outputs may serve as applicable operators for continuing the growth of negative-rooted trees; or for continuing the growth of trees whose outputs may serve for continuing the growth of negative-rooted trees, . . ., etc.

The spirit of G-deduction is to re-organise the sequencing of the computation so as to reflect this recursive structure in the process of growing P_2-trees. The incomplete P_2-trees at each level of the hierarchy are regarded as 'calling' those at lower levels. Growth of trees in this latter category is only initiated when 'called for' by an immediately calling P_2-process: specifically to supply a missing operator so as to unblock a blocked node. The called-for P_2-deduction may itself become blocked and call in turn for a subsidiary deduction, and so on.

Proof of completeness

It remains to show completeness of some form of G-deduction. To do this we must specify certain points in the strategy which have been left vague. We define a particular version of the algorithm, called 'conservative G-deduction', and show that it is complete. Completeness proofs for other versions can then be constructed, if so desired, by showing essential identity with the conservative version.

Conservative G-deduction represents a specialisation of the procedure described above in the following respects:

(1) no recursive application of G-deduction to any node of a P_2-tree is made unless every B-node of the tree is 'blocked';

(2) no recursive application of G-deduction at level $n+k$ is made until every possible recursive application of G-deduction at level n has been made, where k is some integer, for example, 1.

(3) when a recursive application of G-deduction succeeds, the A-operator which has been produced is applied to the calling node so as to unblock it, but is not added at this stage to the base set.

(4) addition to the base set of new A-clauses harvested from a P_2-tree only occurs when no further progress can be made without increasing the permitted depth of recursion. Also at this point the permitted depth of recursion, *rmax*, is increased to $rmax+k$.

Definition. A clause is said to be activated when it is selected as the root of a P_2-tree. From that moment, it and any of its descendent nodes may at any stage be operated on by new positive clauses accruing to S and thus acquire fresh descendants. A clause once activated remains so *in perpetuo*, since the possibility of new growth in the P_2-tree rooted in it can never be precluded.

Theorem. If S is a finite unsatisfiable set of clauses, there is a conservative G-deduction of $\square$ from S.

In hyper-resolution all non-positive clauses in S are activated at the start. When no further progress is possible in any of the P_2-trees, the base set is augmented by the positive clauses generated in the given growth cycle. In G-deduction by contrast only a subset of the non-positive clauses is activated at any stage, namely:

(1) the wholly negative clauses in S, and

(2) non-positive clauses which have been called for by blocked nodes of existing P_2-trees.

However, the algorithm ensures that all B-clauses that must be activated if $\square$ is to be produced will eventually be activated. This is because (1) the A-list of each B-node contains all B-clauses, the activation of which could possibly lead to unblocking it; and (2) the above restriction on recursion depth ensures that all the B-clauses on the A-list *will* be activated before the depth of recursion is incremented. Thus any hyper-resolvents (derived A-clauses) that are essential to the finding of a proof will be generated.

This informal account of G-deduction has assumed a fixed rule for choice of the next B-clause negative literal to be 'stripped' by P_1-resolution. For expository purposes the rule adopted has been to take the left-most. This restriction preserves completeness, as discussed by Kowalski and Hayes (1969).

Recapitulation

Figure 8 is offered as a graphical recapitulation of the leading features of the algorithm, which is further illustrated by two worked ground examples in figures 9 and 10.

The main ideas underlying G-deduction are the following:

(1) re-interpretation of P_1-resolution as a unary operation so as to express the deduction of the empty clause as a path problem in a states-and-operators graph;

(2) the concept of the 'blocked node' and the corresponding principle: 'if you haven't got a pre-fabricated operator ready, then make one';

(3) formulation of the process in such a way that the construction of the new operator is done by the same procedure as the higher-level process which called it. Hence the recursive structure of G-deduction;

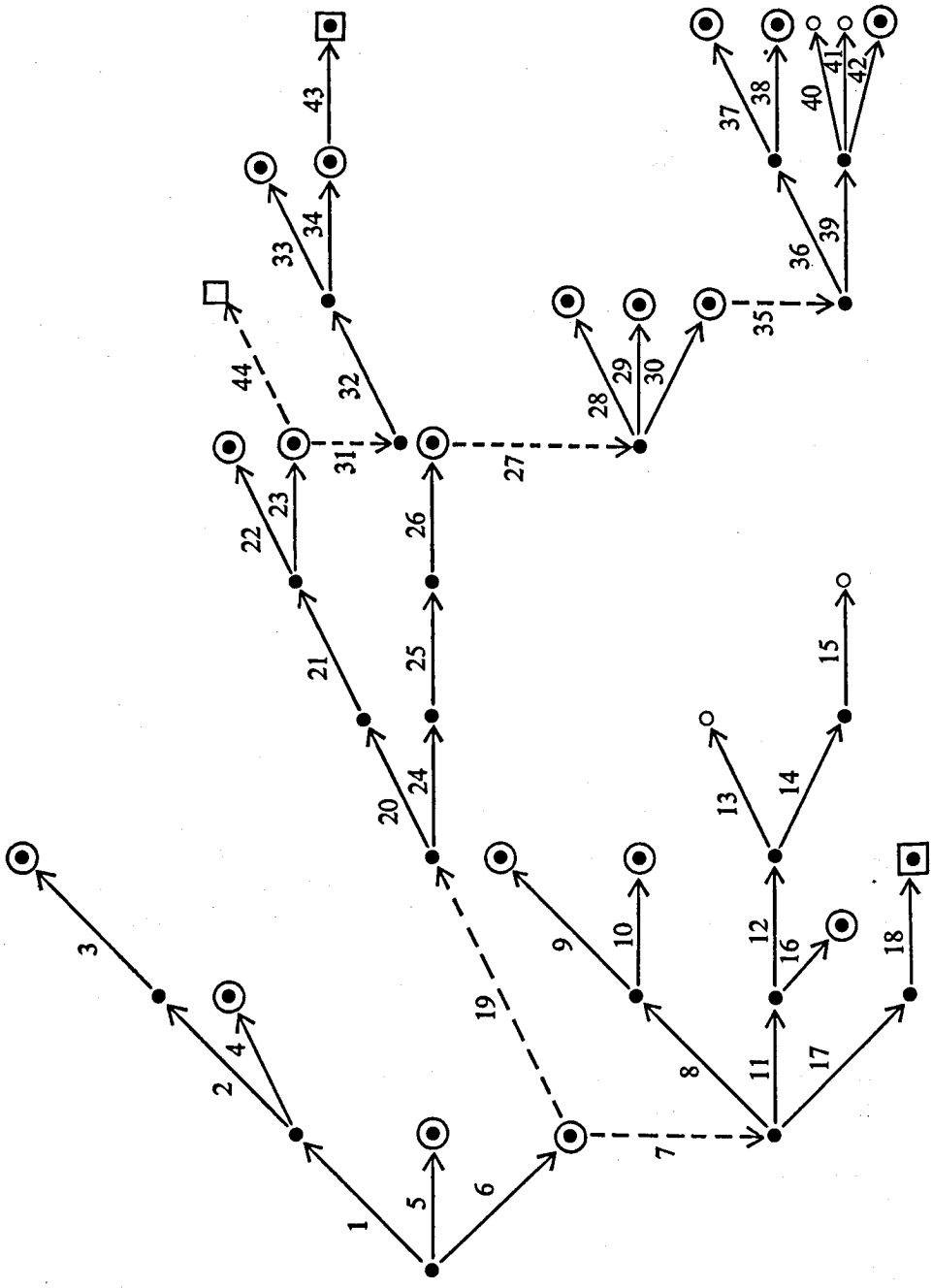


Figure 8. The structure of a *G*-deduction. Movement from left to right corresponds to increasing *depth* in the individual P_2-trees. Movement from top to bottom corresponds to increasing *level* of recursive call. ⊙ denotes blocked *B*-nodes; ○ denotes *A*-nodes; ⊡ denotes goal nodes; □ denotes the top-level goal node, representing the empty clause. ⟶ denotes *if* down pointing then the calls-for relation, *otherwise* a successor relation enabled by a successful call. Labels on the arcs denote temporal sequence.

(4) representation of *G*-deduction as a re-sequencing of the constituent operations of hyper-resolution in such a way as to guarantee completeness.

We now consider some experimental studies of the performance of the procedure when implemented as a computer program.

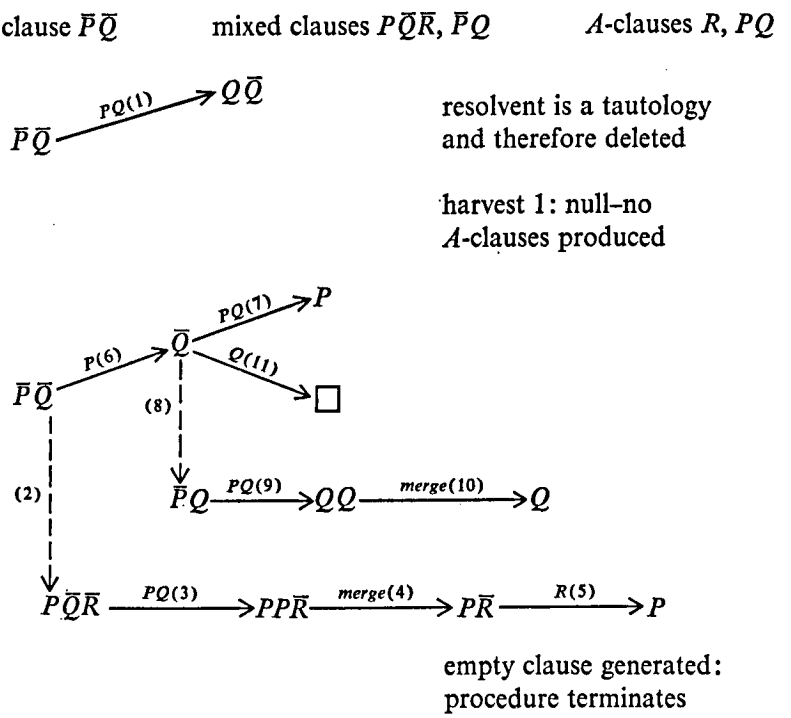


Figure 9. First worked example, EW 2. The clause-generation and binary merging are done as separate operations as evidenced in the sequence $PP\bar{R} \rightarrow P\bar{R}$.

EXPERIMENTAL MATERIALS AND METHODS

The *G*-deduction program

The *G*-deduction program implements an iterative version of the 'conservative' *G*-deduction procedure described recursively above. It is written in POP-2 (Burstall, Collins and Popplestone 1971) and when compiled occupies approximately 18,000 words of store on an ICL 4130 computer. The resolution routines were written by Bob Boyer and JMoore of the Department of Computational Logic, clauses being represented in the 'structure-sharing' format described by them elsewhere in this volume (p. 101). We leave to a later report detailed description of the implementation and confine our present comments to those aspects which affect implementation-independent performance measures, for example, the number of clauses generated.

Within the constraints imposed by conservative *G*-deduction to ensure

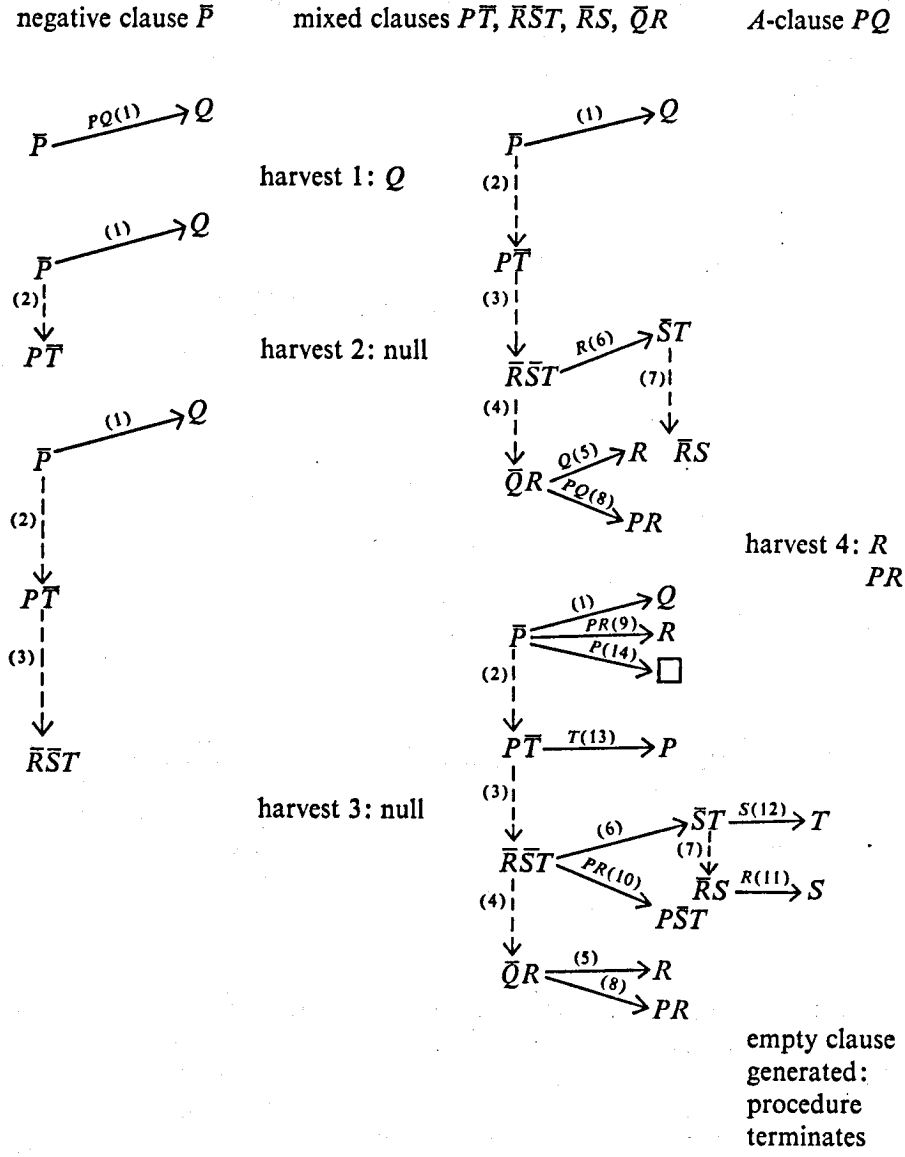


Figure 10. Second worked example, EW 1.

completeness there are several points at which it is possible to exercise heuristic control. In particular, it is possible to control heuristically the growth of P_2-trees and the order in which blocked nodes are selected for unblocking.

P_2-trees were grown by the Graph Traverser running in partial development mode. This corresponds to GT4. The evaluation function was

$$w_1 \cdot depth(x) + w_2 \cdot h_1(x) + w_3 \cdot h_2(x)$$

$depth(x)$ = the distance of clause x from the root of the P_2-tree in which it was generated.

$h_1(x) = n^-(x) + p \cdot n^+(x)$

where $n^-(x)$ = the number of negative literals in x

$n^+(x)$ = the number of positive literals in x

$p = \max_{s \in S_0} n^-(s) + 1$

$h_2(x)$ = the number of symbols in the positive literals of x, excluding brackets and commas.

w_1, w_2, w_3 are weighting parameters, set respectively to 1, 10, 1.

Note that it would be possible, and perhaps desirable, to define *eval* so as to take into account the 'merit' of the operator next to be applied to the given clause, x. But the present program does not do this.

Implementation of the 'A-list' concept (see p. 151) was effected by keeping a global list of operators ordered by the heuristic component of the above evaluation function, that is, by

$$w_1 \cdot h_1 + w_2 \cdot h_2 \qquad (w_1 = 10, w_2 = 1)$$

and this list was updated after each harvest of new A-clauses.

In conservative G-deduction, recursively defined, if all nodes at a given level of recursion become blocked the problem then arises of determining a sensible order in which to attempt to unblock them. In the present program the iterative analogue of this problem occurs and is handled by ordering the blocked nodes by their heuristic values, calculated as described above. When, however, the sub-problems of a particular blocked node are activated they are ordered arbitrarily: again, a heuristic ordering would presumably be an improvement.

All tautologies are deleted and only positive clauses are factored. In neither case is completeness lost. The program also incorporates facilities for a weak form of alphabetic variant checking (see below).

The SL program

The SL program is an experimental implementation of Kowalski and Kuehner's (1971) SL-resolution. Briefly, it employs a depth-first search strategy and a selection function which selects that literal which resolves with the fewest input operators. Tautologies, but not duplicate clauses, are deleted.

The program was written, in POP-2, by R. Boyer and J Moore. When compiled it occupies approximately 11,000 words of store on an ICL 4130 computer.

Motivation and method

It seems desirable to use some program as a basis for comparative tests of performance of the *G*-deduction program. The SL program was chosen for this purpose because its performance is believed to be reasonable in terms of the current state of the art. A second attraction of the comparison was that the two programs show marked complementarity: each is strong where the other is weak. Specifically, *G*-deduction has a sophisticated search strategy operating within a rather simple restriction of the resolution search space (P_1), while SL has a simple strategy operating within a highly sophisticated restriction. A characterisation of the behaviour of each over a range of examples might form a useful preliminary to some later attempt to design an improved theorem-prover by combining the good features of both.

Experimental runs were carried out using a sample of ten problems, of which two problems were selected to test the factoring routines of the *G*-deduction program. The problems are described in clause form in the Appendix. To test the sensitivity of each program to variations in the input order of the literals, two orderings were employed. In the first, the literals were ordered alphabetically by predicate letter with, in the case of ties between positive and negative literals, the negative literals having precedence. An arbitrary tie-breaking rule was employed in all other cases. The second ordering, the anti-alphabetic, was the reverse of the first.

For both the alphabetic and anti-alphabetic orderings two distinct runs were carried out, one without any form of duplicate checking, the other employing a weak form of alphabetic variant-checking in which equality between two positive variants was detected only if the literals of each clause were in the same order. The factors of a positive clause were, however, not checked. Duplicate checking of non-positive clauses was confined to the P_2-tree in which they were generated.

An attempt by *G*-deduction to detect unsatisfiability was terminated unsuccessfully after (a) 3 hours running time when duplicate checking was employed – 1 hour when it was not, or (b) in either mode when program workspace occupied the whole store.

It was our intention, originally, to employ similar termination conditions for the SL runs. However, condition (a) could not always be met in practice due to the frequent occurrence of spurious errors at the time the runs were being carried out. Because of the highly compact method of representing clauses condition (b) was unlikely to be met before (a). We therefore imposed the additional condition that an attempt be terminated when 1000 clauses had been generated.

RESULTS

The results of the runs are summarized in table 1. We have recorded the number of clauses generated by resolution, factoring and binary merging.

Table 1. Comparison of the number of clauses generated by *G*-deduction and SL.

	G-deduction with duplicate testing		*G*-deduction without duplicate testing		SL-resolution without duplicate testing	
theorem	literals ordered alphabetically	literals ordered anti-alphabetically	literals ordered alphabetically	literals ordered anti-alphabetically	literals ordered alphabetically	literals ordered anti-alphabetically
EW 1	10	7	10	7	11	9
EW 2	9	9	9	9	10	9
EW 3	24	21	24	40	20	21
ROB 1	6	6	6	6	13	13
DM	5	5	5	5	21	1000†
QW	55	47	55	47	7	8
MQW	16	18	16	18	8	8
ROB 2	242	246	729*	673*	1000†	1000†
DBA BHP	454	578	454	651	26	322‡
APA BHP	226*	270*	215‡	162‡	213‡	213‡

* indicates failure to find a proof owing to store overflow.
† indicates failure to find a proof through reaching the limit set on number of clauses generated.
‡ indicates failure to find a proof owing to expiry of time limit.

The same problems were then tackled by the SL program, the number of clauses generated being calculated and compared with those of the *G*-deduction program (table 1).

In table 2 we have compared the CPU time (in secs.) spent on each problem by the two programs. The results are expressed to an accuracy of 2 significant figures.

Table 2. Comparison of *G*-deduction and SL results using CPU time (secs.) as the performance measure. Numbers are expressed to an accuracy of two significant figures.

	G-deduction with duplicate testing		*G*-deduction without duplicate testing		SL-resolution without duplicate testing	
theorem	literals ordered alphabetically	literals ordered anti-alphabetically	literals ordered alphabetically	literals ordered anti-alphabetically	literals ordered alphabetically	literals ordered anti-alphabetically
EW 1	5·1	4·1	4·7	3·8	0·88	0·56
EW 2	1·6	1·4	1·3	1·2	0·44	0·38
EW 3	11	12	9·3	27	1·1	1·2
ROB 1	2·3	2·2	2·1	2·0	1·4	1·5
DM	2·4	2·2	2·1	2·1	3·4	890†
QW	20	15	16	12	0·63	0·81
MQW	4·9	6·8	4·1	5·1	0·69	0·69
ROB 2	1100	620	1800*	2000*	810†	1100†
DBA BHP	2000	2700	1100	2200	5·3	3800‡
APA BHP	3600*	8000*	3600‡	3600‡	3700‡	3700‡

* indicates failure to find a proof owing to store overflow.
† indicates failure to find a proof through reaching the limit set on number of clauses generated.
‡ indicates failure to find a proof owing to expiry of time limit.

DISCUSSION

In terms of the number of clauses generated in finding a proof the performance of *G*-deduction is, in general, not unsatisfactory. We must emphasise, however, that this performance is often achieved at a price. The overheads in core occupancy of the *G*-deduction control structure can be considerable. They are particularly acute when large numbers of sub-problems have to be generated. In table 3 we have recorded the number of sub-problems generated per problem for both the duplicate and no-duplicate checking runs. In the case of ROB 2 and APA BHP a bound was placed on the number of sub-problems that could be activated and in each instance this bound was attained. Without it program running time would have been much increased

and, moreover, it is unlikely that ROB 2 would have been solved. In one version of APA BHP it was only possible to generate 226 clauses before program workspace occupied all available core store. Changes, aimed at preventing the proliferation of sub-problems might with advantage be made to the conservative *G*-deduction algorithm.

Table 3. Number of sub-problems generated in *G*-deduction experiments.

	G-deduction with duplicate testing		*G*-deduction without duplicate testing	
theorem	literals ordered alphabetically	literals ordered anti-alphabetically	literals ordered alphabetically	literals ordered anti-alphabetically
EW 1	6	5	6	5
EW 1	2	2	2	2
EW 3	27	29	27	78
ROB 1	2	2	2	2
DM	2	2	2	2
QW	0	0	0	0
MQW	3	7	3	7
ROB 2	298	298	118*	120*
DBA BHP	88	256	88	495
APA BHP	1660*	1660*	1660‡	1660‡

* indicates failure to find a proof owing to store overflow.
‡ indicates failure to find a proof owing to expiry of time limit.

The SL results demonstrate how sensitive the program is to the input order of the literals. This sensitivity is a consequence of the particular depth-first search strategy which it employs. Thus if the program is lucky in the order in which the literals are input the search will be performed along a branch that leads quickly to a solution and it will do well, otherwise it may be misled, even on simple problems. This defect in its search strategy is one of the indications that substantial gains might accrue either from equipping the SL program with some of the features of *G*-deduction's search strategy, or from applying *G*-deduction to restrictions of the search space more refined than P_1.

Acknowledgements

The theoretical part of this work was done in response to an invitation from J.A. Robinson to one of us (D.M.) to investigate applications of heuristic search to the design of resolution strategies. Thanks are owed to him, and also to E.Sibert, for pointing out errors in the original version of the algorithm, and for helpful discussions. A special debt is owed to R.S.Boyer and J Moore for patiently taking us through the details of their SL program and for making available to us their POP-2 routines for resolution and factoring. Financial support was contributed by the Advanced Research Projects Agency of the U.S. Defense Department, by the Science Research Council (UK) and by the Telecommunications Headquarters of the General Post Office (UK).

REFERENCES

Burstall, R.M., Collins, J.S. & Popplestone, R.J. (1971) *Programming in POP-2.* Edinburgh: Edinburgh University Press.

Ernst, G.W. & Newell, A. (1969) GPS: *A Case Study in Generality and Problem Solving.* New York and London: Academic Press.

Fikes, R.E. & Nilsson, N.J. (1971) STRIPS: a new approach to the application of theorem proving to problem solving. *Art. Int.*, **2**, 189–208.

Green, C. (1969) Application of theorem proving to problem solving. *Proc. Int. Joint Conf. on Art. Int.*, pp. 219–39 (eds. Walker, D.E. & Norton, L.M.). Washington DC.

Kowalski, R. (1969) Search strategies for theorem-proving. *Machine Intelligence 5*, pp. 181–201 (eds. Meltzer, B. & Michie, D.). Edinburgh: Edinburgh University Press.

Kowalski, R. & Hayes, P.J. (1969) Semantic trees in automatic theorem-proving. *Machine Intelligence 4*, pp. 87–101 (eds. Meltzer, B. & Michie, D.). Edinburgh: Edinburgh University Press.

Kowalski, R. & Kuehner, D. (1971) Linear resolution with selection function. *Artificial Intelligence*, **2**, 227–60.

Michie, D. & Ross, R. (1969) Experiments with the adaptive Graph Traverser. *Machine Intelligence 5*, pp. 301–18 (eds. Meltzer, B. & Michie, D.). Edinburgh: Edinburgh University Press.

Michie, D. & Sibert, E. (1973) Some binary derivation systems. (forthcoming).

Popplestone, R.J. (1970) Freddy, things and sets. (unpublished).

Robinson, J.A. (1965) Automatic deduction with hyper-resolution. *Int. J. comput. Math.*, **1**, 227–34.

APPENDIX: TEST BATTERY OF PROBLEMS

Literals are ordered alphabetically

EW 1

$$\{\{\bar{P}\}, \{P, Q\}, \{\bar{Q}, R\}, \{\bar{R}, S\}, \{\bar{R}, \bar{S}, T\}, \{P, \bar{T}\}\}$$

EW 2

$$\{\{P, Q\}, \{\bar{P}, Q\}, \{\bar{P}, \bar{Q}\}\ \{P, \bar{Q}, \bar{R}\}, \{R\}\}$$

EW 3

$$\{\{\bar{P}, \bar{Q}, \bar{R}\}, \{P, \bar{S}, \bar{T}\}, \{\bar{R}, S\}, \{R, \bar{T}\}, \{\bar{Q}, \bar{R}, T\}, \{\bar{Q}, R\}, \{Q, R\}, \{Q, \bar{R}\}, \{\bar{P}, Q, \bar{S}\}\}$$

ROB 1

$$\{\{P(x, y)\}, \{\bar{P}(y, F(x, y)), \bar{P}(F(x, y), F(x, y)), \bar{Q}(x, F(x, y)), \bar{Q}(F(x, y), F(x, y))\}, \{\bar{P}(y, F(x, y)), \bar{P}(F(x, y), F(x, y)), Q(x, y)\}$$

DM

$\{\{\bar{P}(y, x, u), \bar{P}(x, v, z), \bar{P}(y, z, w), P(u, v, w)\}, \{P(G(x, y), x, y)\},$
$\{P(x, H(x, y), y)\}, \{\bar{P}(K(x), x, K(x))\}\}$

QW

$\{\{\bar{P}(x, A), \bar{P}(x, y), \bar{P}(y, x)\}, \{P(x, A), P(x, F(x))\},$
$\{P(x, A), P(F(x), x)\}\}$

MQW

$\{\{G(y, A), G(F(y), y)\}, \{G(y, A), G(y, F(y))\}, \{\bar{G}(w, y), G(F(y), y)\},$
$\{\bar{G}(w, y), G(y, F(y))\}, \{\bar{G}(w, y), \bar{G}(y, A)\}\}$

ROB 2

$\{\{P(A, B, C)\}, \{P(x, x, E)\}, \{P(x, E, x)\}, \{P(E, x, x)\}, \{\bar{P}(B, A, C)\},$
$\{\bar{P}(x, y, u), \bar{P}(y, z, v), \bar{P}(x, v, w), P(u, z, w)\},$
$\{\bar{P}(x, y, u), \bar{P}(y, z, v), \bar{P}(u, z, w), P(x, v, w)\}\}$

DBA BHP

$\{\{\bar{A}(x, z, y), \bar{H}(z, y), I(x, P(y))\}$
$\{\bar{H}(w, y), H(z, G(z, y))\}$
$\{A(x, z, G(z, y)), \bar{H}(w, y), \bar{I}(x, y)\}$
$\{\bar{H}(z, y), H(z, L(y))\}$
$\{A(S, E, N)\}$
$\{\bar{C}(y)\}$
$\{\bar{I}(x, L(y))\}$
$\{\bar{A}(x, E, y), R(x)\}$
$\{\bar{A}(x, z, y), A(x, z, L(y))\}$
$\{\bar{A}(x, z, y), A(x, z, P(y))\}$
$\{\bar{A}(x, z, y), B(x, P(G(z, L(y))))\}$
$\{C(y), \bar{Q}(x, T, y), \bar{R}(x)\}$
$\{\bar{A}(x, w, y), \bar{B}(x, y), Q(x, z, G(z, y))\}$
$\{\bar{A}(x, w, y), A(x, w, G(z, y)), I(x, y)\}\}$

This problem, like APA BHP following, is a version of Popplestone's (1970) 'Blind Hand Problem'.

SEMANTIC KEY

Functions and constants

Interpret

A	as	*AT*
B		*GRAB*
C		*ANSWER*
E		*HERE*
G		*GO*
H		*HANDAT*
I		*HELD*
L		*LETGO*
N		*NOW*
P		*PICKUP*

Q *PUT*
R *RED*
S a Skolem constant
T *THERE*

Variables

Interpret
x as a 'thing' variable
y as a 'situation' variable
z as a 'place' variable

APA BHP

$\{\{\bar{A}(x, E, y), \bar{A}(x, T, y)\}$
$\{\bar{I}(M(x), D(L, y))\}$
$\{\bar{R}(H)\}$
$\{\bar{C}(y)\}$
$\{A(H, z, D(G(z), y))\}$
$\{A(M(S), E, N)\}$
$\{\bar{A}(M(x), z, D(G(z), y)), A(M(x), z, y), I(M(x), y)\}$
$\{\bar{A}(M(x), z, y), \bar{A}(H, z, y), I(M(K(y)), D(P, y))\}$
$\{\bar{A}(H, z, y), A(M(x), z, y), \bar{I}(M(x), y)\}$
$\{\bar{A}(M(x), z, y), A(H, z, y), \bar{I}(M(x), y)\}$
$\{\bar{A}(x, T, y), C(y), \bar{R}(x)\}$
$\{\bar{A}(M(x), z, y), A(M(x), z, D(G(z), y))\}$
$\{\bar{I}(M(x), y), I(M(x), D(G(z), y))\}$
$\{\bar{A}(H, z, y), A(M(K(y)), z, y)\}$
$\{\bar{A}(x, z, y), A(x, z, D(P, y))\}$
$\{\bar{A}(x, z, y), A(x, z, D(L, y))\}$
$\{\bar{A}(x, z, D(L, y)), A(x, z, y)\}$
$\{\bar{A}(x, E, N), R(x)\}\}$

SEMANTIC KEY

Functions and constants

Interpret
A as *AT*
C *ANSWER*
D *DO*
E *HERE*
G *GO*
H *HAND*
I *HELD*
K *TAKEN*
L *LETGO*
M *THING*
N *NOW*
P *PICKUP*
R *RED*

S a Skolem constant
T *THERE*
Variables
Interpret
x as a 'thing' variable
y as a 'situation' variable
z as a 'place' variable.

10

And-or Graphs, Theorem-proving Graphs and Bi-directional Search

R. Kowalski
Department of Computational Logic
University of Edinburgh

Abstract

And-or graphs and theorem-proving graphs determine the same kind of search space and differ only in the direction of search: from axioms to goals, in the case of theorem-proving graphs, and in the opposite direction, from goals to axioms, in the case of and-or graphs. Bi-directional search strategies combine both directions of search. We investigate the construction of a single general algorithm which covers uni-directional search both for and-or graphs and for theorem-proving graphs, bi-directional search for path-finding problems and search for a simplest solution as well as search for any solution. We obtain a general theory of completeness which applies to search spaces with infinite or-branching. In the case of search for any solution, we argue against the application of strategies designed for finding simplest solutions, but argue for assigning a major role in guiding the search to the use of symbol complexity (the number of symbol occurrences in a derivation).

INTRODUCTION

This paper investigates some of the search spaces and search strategies which are applied to the representation and attempted solution of problems in artificial intelligence. Our main objective is to demonstrate that both and-or graphs and theorem-proving graphs determine the same kind of search space and that the difference lies only in the direction of search – forwards from axioms to goal in theorem-proving graphs and backwards from goal to axioms in and-or graphs. This initial observation suggests the construction of bi-directional algorithms which combine forward and backward searches within a given search space. The resulting bi-directional strategy generalises the one studied by Pohl (1969) for path-finding problems.

We investigate the notions of exhaustiveness and completeness for search strategies. For both of these properties it is unnecessary to insist that the

search space contain only finitely many axioms or alternative goals or that only finitely many inference steps apply to a fixed set of premises or to a given conclusion. We discuss the utility of constructing search spaces which violate such finiteness restrictions and show how these spaces can be searched completely by employing symbol complexity to guide the search. The removal of the finiteness restrictions makes it impossible to apply Pohl's (1969) cardinality comparison of candidate sets in order to determine which direction of search to choose at any given stage. The decision to choose instead the direction which has smallest set of candidates of best merit resolves this dilemma and can be effectively applied precisely in those circumstances when it is possible to search the space completely in at least one of two directions.

We distinguish between problems which require a simplest solution and problems which require any solution. For the first kind of problem we investigate the construction of search strategies which are guaranteed to find simplest solutions. These strategies involve the use of heuristic functions to estimate the additional complexity of a simplest solution containing a given derivation. Under suitable restrictions the search strategies find simplest solutions and do so by generating fewer intermediate derivations (both those which are relevant and irrelevant to the eventually found solution) than do the more straightforward complexity saturation strategies. The heuristic search strategies cover those previously investigated by Hart-Nilsson-Raphael (1968) for path-finding problems, by Nilsson (1968) for and-or trees, and by Kowalski (1969) for theorem-proving graphs. In the case of bi-directional path-finding problems, we propose a method of updating the heuristic function so that it reflects properly the progress of search in the opposite direction. The resulting search strategy overcomes much of the inefficiency, noted by Pohl (1969), of the original bi-directional strategy.

Having investigated search strategies which aim to find simplest solutions, we then argue against applying similar strategies in situations which require finding any solution, no matter what its complexity. We argue that such search strategies have the undesirable characteristic of examining and generating equally meritorious potential solutions in parallel. We propose no concrete alternative, but suggest that more appropriate search strategies will need to employ methods for 'looking-ahead' in the search space and will need to bear a greater resemblance to depth-first searches than do those which belong in the family with search strategies for simplest solutions. We argue moreover that symbol complexity should occupy a central role in guiding the search for a solution.

The problems investigated in this paper assume a solution to the representation problem: for an initial semantically defined problem, how to obtain an adequate syntactic formulation of the problem. In the case of theorem-proving, the semantically defined problem might have the form of showing that one set of sentences implies another. The original problem does not

specify a search space of axioms, goals and basic inference operators. Indeed, many different search spaces can represent the same original problem, as in the case of resolution systems where different refinements of the resolution rule determine different search spaces for a single problem of demonstrating the unsatisfiability of a given set of clauses. Viewed in this context, the name 'theorem-proving graph', for a particular direction of search in a particular kind of search space, can be misleading. A given theorem-proving problem can be represented by different search spaces, not all of which need be interpretable as theorem-proving graphs when searched from the direction which starts with axioms and works toward the goal.

Conversely, the abstract graph-theoretic structure of problems incorporated in and-or graphs and in theorem-proving graphs is not confined to the representation of theorem-proving problems. In the next section of this paper we demonstrate an application of such search spaces to the problem of enumerating all subsets of a given set of objects.

This paper assumes no prior acquaintance with the research literature about search spaces and search strategies. On the contrary, our intention is that it provide a readable, and not overly-biased, introduction to the subject. For the same purpose, the reader will probably find Nilsson's book (1971) useful.

DERIVATIONS, THEOREM-PROVING GRAPHS, AND-OR GRAPHS

We observe first that and-or graphs and theorem-proving graphs determine the same kind of search space (see figure 1). What distinguishes the two is the direction of search: forwards from initially given axiom nodes towards the goal node in theorem-proving graphs, and backwards from the goal node towards the initially given solved nodes in and-or graphs. Both directions of search can be combined in bi-directional search strategies which generalise those investigated for path-finding problems.

Typically, theorem-proving graphs/and-or graphs are used for problems of logical deduction where the task is one of finding some solution derivation of an initially given goal sentence from initially given axioms. With theorem-proving graphs, the initially given set of solved problems (axioms) is repeatedly augmented by the addition of deduced theorems until the goal sentence is derived and recognised as having been solved. With and-or graphs, the solution of the initially given goal problem is repeatedly reduced to the solution of alternative sets of subgoal problems, some of which might be immediately recognised as being solved, others of which might need further reduction. A solution is obtained when some such reduction set of subgoals consists entirely of completely solved subproblems (axioms) of the original problem. In both cases the search strategy can be regarded as selecting some candidate derivation and then generating it by generating all of its previously ungenerated sentences and connecting inference steps.

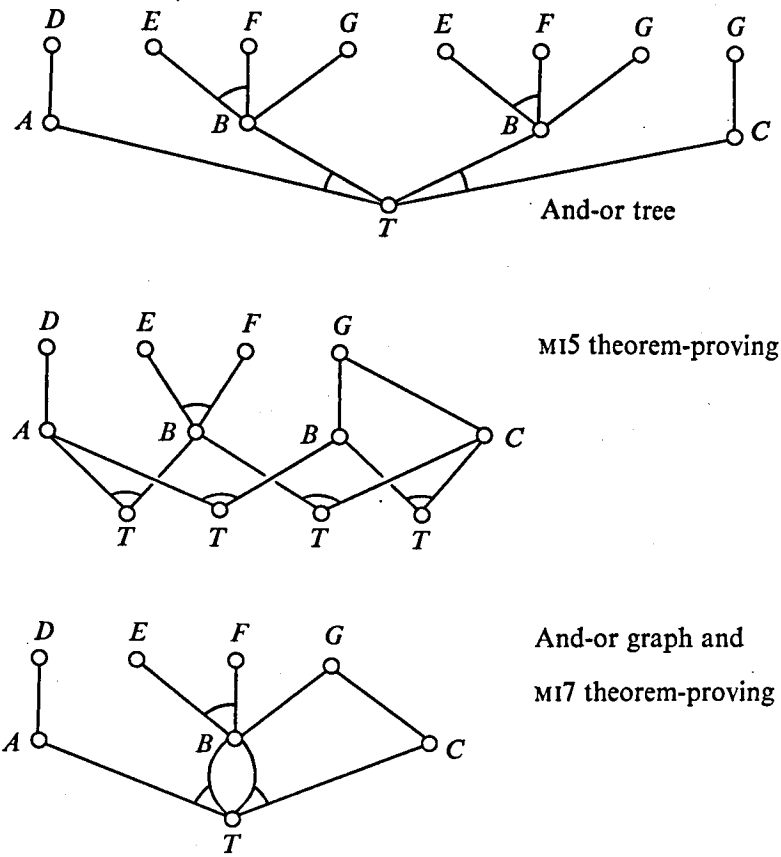
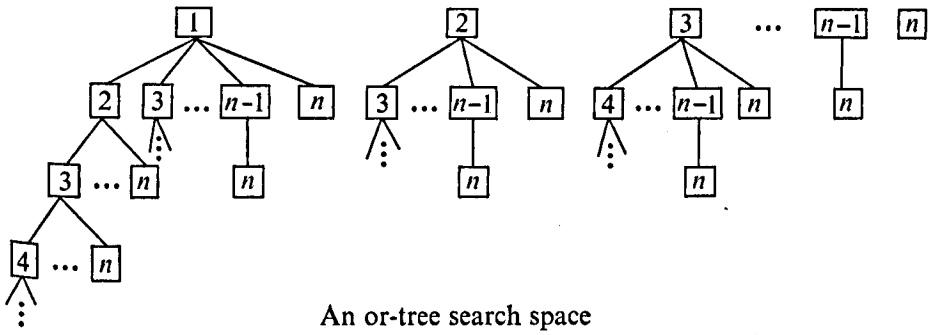


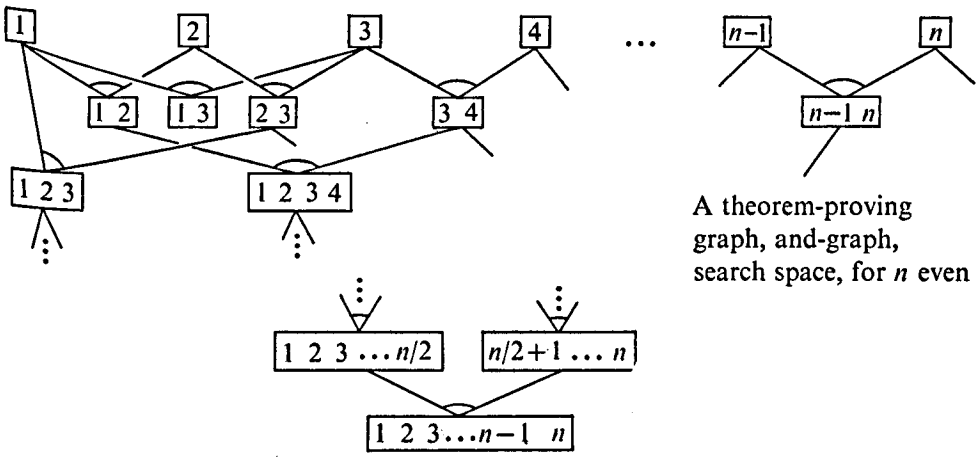
Figure 1. In the and-or tree, the identical problems *B* are encountered in different inference steps, working backwards from the goal *T*. Similarly *E*, *F* and *G* occur more than once in the and-or tree. All distinct occurrences of a sentence have distinct nodes associated with them. In the tree representation of theorem-proving graphs (used in *Machine Intelligence 5* (Kowalski 1969)), identical problems *T* are generated at different times, working forwards from the initial set of axioms {*D*, *E*, *F*, *G*}. Here too distinct occurrences of a sentence are associated with distinct nodes. In the and-or graph and in the graph representation of theorem-proving graphs (as defined in this paper) sentences are the nodes of the search space. Different ways of generating the same sentence are represented by different arcs connected to the sentence.

Theorem-proving graphs and and-or graphs can also be applied to the representation of search spaces in problems where the inference operation connecting premises with conclusion is not one of logical implication. Figure 2 illustrates just such an application where the 'inference operator' is a restricted form of disjoint union of sets. Figure 2 also illustrates a more conventional or-tree representation for the same problem of enumerating without redundancy all subsets of a given collection of *n* objects. Both search spaces contain the same number of nodes and achieve similar economies by

sharing a single copy of a subset when it is common to several supersets. For a particular application to a problem in urban planning of identifying a least costly collection of m housing sites out of a total of n possible, the use of the theorem-proving graph, and-graph, provides several advantages. In particular a certain bi-directional search strategy can be employed and



An or-tree search space



A theorem-proving graph, and-graph, search space, for n even

Figure 2. The or-tree and the theorem-proving graph provide two different ways of enumerating all subsets of n objects without redundancy. The theorem-proving graph is an and-graph, without any or-branching, because every subset can be derived in only one way; there are no alternatives. (Every subset is the conclusion of a single inference step. The two subsets which are premises of the inference step are disjoint. One of these subsets contains only elements which are less than all of the elements contained in the other premise subset. Either both premises have the same number of elements, or else the premise which contains the smaller elements contains one less element than the other premise.) Moving downwards in the space corresponds to generation of subsets in the theorem-proving graph direction, combining smaller subsets into larger ones. Moving upwards corresponds to generation in the and-graph direction, decomposing subsets into smaller ones.

interpreted as a model of the method of successive approximation. The search begins with aggregated collections of objects in the middle of the search space and works both forward combining smaller sets into larger ones and backwards splitting aggregates into smaller sets; this method of bi-directional search differs from the one which is discussed in this paper. Further details regarding the use of theorem-proving graphs and heuristic programming in the urban design problem are contained in the reports by Kowalski (1971) and du Feu (1971).

In the following definitions, the notions of sentence and inference operator should be interpreted liberally to include any kind of object and any operator which maps sets of objects onto single objects. A more formal and abstract treatment of these notions can be supplied along the lines detailed for MI 5 theorem-proving graphs (Kowalski 1969).

Let sentence N follow directly from the *finitely* many sentences $N_1, \ldots, N_n$ by means of one application of an inference operator. Then N is connected to each of N_i by a directed *arc* (from N_i to N), the collection of which is called an *and-bundle*. The *premises* $N_1, \ldots, N_n$, the *conclusion* N, and the connecting and-bundle constitute a single *inference step*. Corresponding to each axiom N is a single inference step having N as conclusion and an empty set of premises. (In the figures we have not drawn arcs associated with inference steps having axioms as conclusions.)

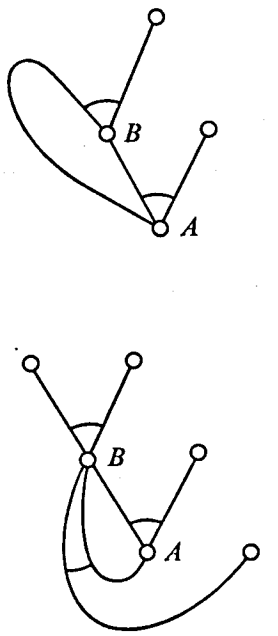


Figure 3. Some non-derivations.

A *derivation* D is a finite set of sentences and connecting inference steps such that:

(1) Every sentence in D belongs to at least one inference step in D.

(2) Exactly one sentence C in D is the conclusion of an inference step but premise of none.

(3) Every sentence in D is the conclusion of at most one inference step in D.

(4) D contains no infinite branches, where a *branch* in D is a sequence of sentences $S_1, S_2, \ldots, S_n, S_{n+1}, \ldots$, where S_1 is in D and S_{n+1} is a premise of an inference step in D which has S_n as conclusion. (This condition guarantees that the graph associated with D contains no cycles.)

Figure 3 illustrates some non-derivations.

The *premises* of D are those sentences in D which are conclusions of no inference steps belonging to D. The *conclusion* of D is C. D is *premise-free* if it has no premises. D is a *reduction* if the conclusion it derives is the initially given goal sentence. D is a *solution* if it is a premise-free reduction. All candidate derivations selected for generation in the forwards theorem-proving direction are premise-free. All candidates selected in the and-or direction are reductions. Every reduction derivation reduces the solution of the goal node to the solution of the premises of the derivation. Every premise-free derivation augments the initially given set of solved nodes by the conclusion it derives.

Path-finding problems constitute a special case of the derivation-finding problem: every inference step contains at most one premise. A derivation, therefore, consists of a single path connecting its premise with its conclusion.

THE ALGORITHM

The bi-directional algorithm for and-or graphs/theorem-proving graphs described in this section includes as special cases

(1) uni-directional algorithms for theorem-proving graphs and for and-or graphs;

(2) bi-directional algorithms for path-finding problems;

(3) algorithms for finding simplest solutions, relative to a given measure of complexity, as well as algorithms for finding any solution, regardless of its complexity; and

(4) complexity saturation algorithms which use no heuristic information, as well as algorithms which do use heuristic information.

We do not claim that the general algorithm produces the most efficient algorithm for any of these special cases; our objective is, instead, to formulate an algorithm which abstracts what is common to many different search problems and search strategies. The intention is that best algorithms for special cases might be obtained by special additions or minor modifications to the general algorithm.

The algorithm proceeds by first selecting a direction of generation, direction **F** for forward generation or direction **B** for backward generation. If there are

no candidate derivations (containing exactly one inference step as yet ungenerated) the algorithm terminates without solution; otherwise it chooses a candidate which is of best merit among all candidates in the selected direction and then generates all previously ungenerated sentences and the one previously ungenerated inference step belonging to the chosen derivation. If an appropriate solution has not been generated (depending on the problem, either any solution or else one which is simplest), then the algorithm begins another cycle of selecting a direction of generation and then choosing and generating a candidate derivation. The algorithm terminates when an appropriate solution is generated. If the objective is to generate a simplest solution then the algorithm continues until it finds a simplest solution and identifies it as being simplest.

At any given time, the set of sentences and inference steps so far generated is stored in two sets **F** and **B**. **F** contains all sentences and inference steps belonging to already generated premise-free derivations. (Thus all sentences in **F** are solved sentences, or theorems.) **B** contains all sentences and inference steps belonging to already generated reduction derivations. The intersection of **F** and **B** need not in general be empty. A derivation is regarded as *generated* provided all its sentences and inference steps have been generated, even though they might have been generated as part of some other candidate derivation explicitly selected for generation. Thus some derivations are deliberately selected and generated, while others, with which they share inference steps, may be generated adventitiously. The following definitions specify the algorithm more precisely.

Suppose that there exists an inference step which does not belong to **F**, but all of whose premises do belong to **F**. Then any premise-free derivation which consists of

(1) this inference step,
(2) its conclusion, and
(3) for every premise of the inference step, exactly one premise-free derivation, contained in **F**, whose conclusion is the premise,

is a *candidate* for generation in direction **F**.

Suppose that D_0 is a reduction derivation, contained in **B**, and every premise of D_0 is the conclusion of some inference step not belonging to **B**. Then any reduction derivation which consists of

(1) just one such inference step,
(2) its premises, and
(3) D_0,

is a *candidate* for generation in direction **B**.

Assume that a notion of *merit* has been defined on derivations, such that for any collection of derivations it is always possible to select effectively a derivation of best merit, in the sense that no other derivation belonging to the collection has better merit. Assume there is given just one goal problem. This, without loss of generality, covers cases where there are finitely many

goals to be solved simultaneously or denumerably many (finite or infinite) alternative goals.

(1) *Initialize* both **F** and **B** to the null set.

(2) *Select* one of **F** or **B** as the *direction* **X** of generation.

(3) *Terminate without solution* if there are no candidates for generation in direction **X**. Otherwise continue.

(4) *Select and generate a candidate* derivation D for **X** having best merit among candidates for **X**. Generate D by adding to **X** the single inference step and all sentences in D not already belonging to **X**.

(5) *Update* **F** and **B** by adding to both of them all inference steps and sentences belonging to any already generated premise-free derivation of a sentence in **B**.

(6) *Terminate with solution* if (a) some solution D is contained in **F** or **B**, and (b) no candidate for direction **F** or **B** has better merit than D. Otherwise go back to (2).

Figure 4 illustrates the operation of a single cycle of the algorithm.

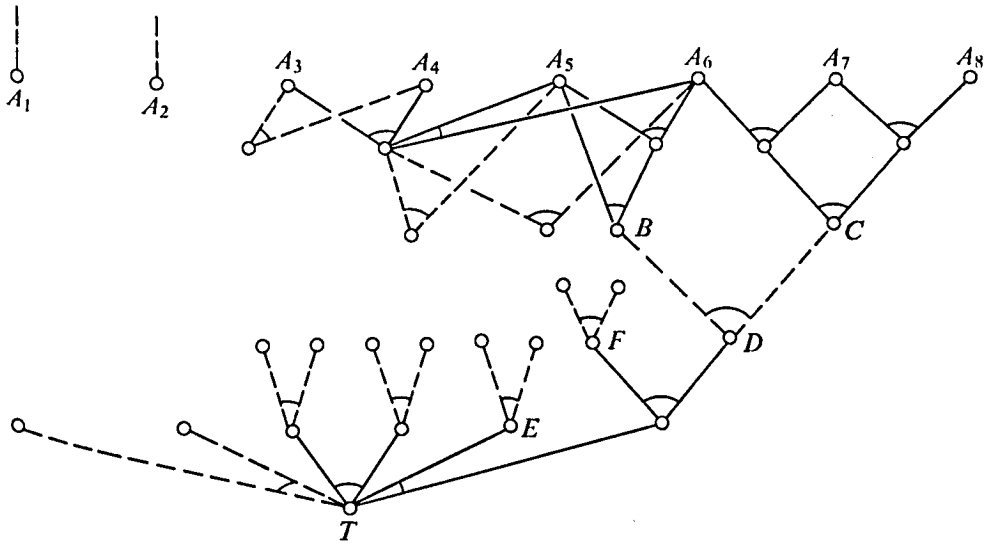


Figure 4. The state of the search space at the beginning of a cycle of the algorithm: Solid lines represent arcs which belong to inference steps already generated. Broken lines represent arcs which belong to inference steps which are candidates for generation. Inference steps which have not yet been generated and which are not candidates for generation are not represented in the figure. Of the axioms $A_1, \ldots, A_8$, the first two axioms have not yet been generated. The inference steps which have the remaining axioms as conclusions are not illustrated.

The inference step which has B and C as premises and D as conclusion is a candidate for generation in both direction **F** and direction **B**. The sets **F** and **B** do not yet have any elements in common.

Suppose now that **B** is chosen as direction of generation and that the reduction derivation having premises $\{E, F, B, C\}$ is the candidate selected for generation. Then in step (4), the sentences B and C are added to **B** along with the inference step having them as premises. In the update step, the same inference step is also added to **F** along with its conclusion D. At the same time, all sentences and inference steps in the derivations of B and C are added to **B**.

Remarks

(1) The most complicated uni-directional case of the algorithm is the case of and-or graphs (where the direction **X** is always chosen to be **B**). The set **F**, which is initially empty, is augmented whenever a sub-problem is solved. The updating step corresponds, in the usual algorithm for searching and-or trees, to the operation of labelling sub-problems as solved. The effect of labelling sub-problems as unsolvable is obtained by not counting as a candidate derivation one whose premises are not conclusions of some (as yet ungenerated) inference step.

(2) The algorithm can be implemented easily in a list processing system, by associating with every inference step backward pointers from the conclusion to the premises. Such pointers correspond to the directed arcs connecting conclusion with premises in the and-bundle of the inference step. Derivations already generated can be accessed by following the pointers backwards.

(3) Unless the problem is one of obtaining a simplest solution, condition 6(b) is unnecessary and can be disregarded. 6(b) is used only when searching for simplest solutions and when employing certain kinds of merit orderings. But even then 6(b) is unnecessary for finding simplest solutions in uni-directional and-or tree search or in uni-directional theorem-proving graph search. This condition is necessary, however, for all bi-directional searches as well as for uni-directional and-or graph search. We shall investigate these matters in greater detail in the section concerned with admissibility.

(4) Our algorithm differs from others reported elsewhere in the literature most importantly in one respect. Ordinarily a search strategy is regarded as selecting derivations which are candidates for expansion. For path-finding problems, a selected derivation is fully expanded by extending it in all possible ways by the addition of a single inference step (arc). For and-or trees, a derivation is selected, one of its premises is chosen and then all inference steps are generated which have the given premise as conclusion. In our algorithm, however, a search strategy is always regarded as selecting derivations which are candidates for generation. A selected derivation is generated by generating its single previously ungenerated inference step.

Our algorithm degenerates when it is impossible to predict the merit of an ungenerated candidate derivation without a detailed examination which requires that the derivation be fully generated. For **X** either **F** or **B**, let $\bar{\mathbf{X}}$ consist of those sentences and inference steps not in **X** but belonging to derivations which are candidates for generation in direction **X**. The algorithm selects and generates an inference step in $\bar{\mathbf{X}}$ belonging to a candidate derivation of best merit. In the degenerate case, it is necessary to store the sets $\bar{\mathbf{X}}$ explicitly inside the computer. The algorithm then behaves indistinguishably from one which fully extends candidate derivations for expansion. **X** can then be re-interpreted as containing the set of fully expanded nodes and $\bar{\mathbf{X}}$ as containing the candidates for expansion.

In several important cases, the sets $\bar{\mathbf{X}}$ need not be explicitly stored in order

to select and generate candidate derivations of best merit. Such cases are more closely related to partial expansion (Michie and Ross 1969) than they are to full expansion. The algorithm does not however, as in partial expansion, first select previously generated nodes of best merit and then extend them by applying the best applicable operator; nor does it first pick an operator of best merit and then apply it to nodes of best merit. For in neither case does the corresponding derivation, selected for generation, necessarily have best merit among all candidates for generation. Our algorithm, in contrast, selects and generates a candidate of best merit even when it cannot be obtained from nodes and operator either of which individually has best merit.

The chief advantage of regarding the sets $\tilde{\mathbf{X}}$ as containing candidates for generation is that then the sets $\tilde{\mathbf{X}}$ need not be finite in order to admit the construction of exhaustive search strategies. In particular, it is quite useful to construct spaces which have *infinite or-branching*, in the sense of having infinitely many axioms, infinitely many alternative goals, or infinitely many inference steps which apply to a fixed set of premises or to a given conclusion. In fact, it seems apparent that such search spaces are often more appropriate for representing certain problems and can be searched more efficiently than alternative search spaces which involve finite or-branching. We shall continue our discussion of this topic in the section concerned with completeness.

DELETION OF REDUNDANCIES

The use, in our algorithm, of and-or graphs, as opposed to and-or trees, eliminates the redundancy which occurs when identical sub-problems are generated, not identified, and consequently solved separately and redundantly as though they were distinct. This use of and-or graphs has been necessitated by wanting to regard generation of derivations in direction **B** as backward search in a theorem-proving graph. Similarly, in order to regard generation in direction **F** as 'backward' search in an and-or graph, we have had to abandon the tree representation of theorem-proving graphs (Kowalski 1969). The graph representation we use here can be obtained from the tree representation by identifying nodes having the same label. The graph representation eliminates the redundancy which arises when the same sentence is derived by distinct premise-free derivations, which can be used interchangeably in all super-derivations containing the given sentence. In both graph representations, and-or graph and theorem-proving graph, maximal use is made of sharing common sub-derivations. In practice the recognition and identification of regenerated sentences requires certain computational overheads. These overheads can be reduced by using the hash-coding methods discussed by Pohl (1971).

The present version of the algorithm admits another kind of redundancy: when a sentence in **F** is derived by more than one premise-free derivation contained in **F**. It is unnecessary to save all of these derivations by keeping all backward pointers associated with each of the inference steps which have

the given sentence as conclusion. It suffices instead to preserve at any given time only those pointers associated with some single distinguished inference step. In the case where one is interested in finding any solution no matter what its complexity, virtually any decision rule can be employed to select the distinguished inference step. However, completeness and efficiency are ordinarily served best by choosing the inference step which belongs to the simplest premise-free derivation of the sentence. If a search strategy is *admissible*, in the sense that it always finds a simplest solution whenever one exists, then the use of this decision rule, to remove redundant inference steps, preserves admissibility.

The analogous redundancy in **B** occurs when distinct derivations contained in **B** have the same set of premises. The different derivations represent different reductions of the original problem to the same set of sub-problems. It suffices to preserve only one such reduction at a time. Chang and Slagle (1971) incorporate into their algorithm for and-or graphs a test for just such redundancy. More generally, redundancy occurs when the premises of one derivation are a subset of those belonging to another. In such a case the first derivation represents a reduction of the original problem to a subset of the sub-problems associated with the second derivation. Unfortunately it seems virtually impossible to eliminate this redundancy in an efficient way, because of the complicated manner in which derivations in **B** share substructure. The situation does not improve even if consideration is limited to the case where distinct derivations have identical sets of premises. In general, virtually every inference step belonging to a derivation in **B** is shared with some other derivation belonging to **B**. For this reason a derivation cannot be removed from **B** by the simple removal of one of its inference steps, since this would almost certainly result in the unintended removal of other derivations. Perhaps the single case where redundant derivations can easily and efficiently be eliminated is the case of path-finding problems. Here, eliminating redundant derivations in **B** can be accomplished exactly as it is done in **F**.

COMPLETENESS

A search strategy is *complete* if it will find a solution whenever one exists. It is *exhaustive* for direction **X** if it will eventually, if allowed to ignore the termination-with-solution condition, generate all sentences and inference steps which can be generated in direction **X**. Clearly, *every exhaustive search strategy is complete.* That *a search strategy can be complete without being exhaustive* is illustrated in figure 5.

It is of practical value to find useful conditions under which search strategies can be certified to be complete. δ-finiteness is such a condition. As defined below, this condition is a straightforward extension of the one given in Kowalski (1969) and is more general than the one considered elsewhere in the literature. An important feature of our definition is that it applies to cases where or-branching is infinite. A derivation is considered

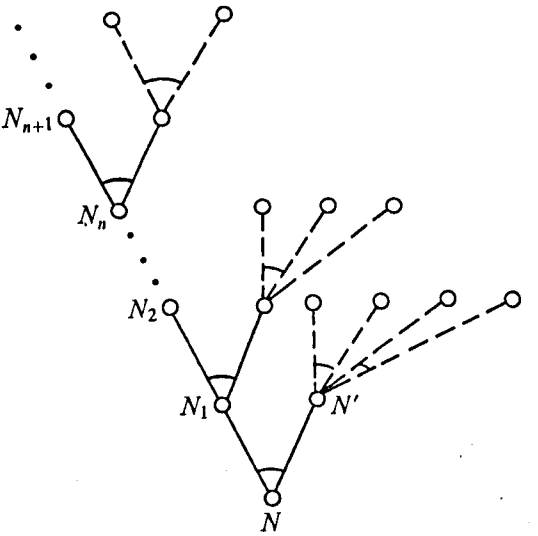


Figure 5. The search space contains an infinite chain of reductions: N can be solved only if N_1 can be solved; N_1 can be solved only if N_2 can; N_n can be solved only if N_{n+1} can. A search strategy which is depth-first along and-branches can be complete without ever generating any of the search space above node N'. In general, depth-first search along and-branches, although it sometimes violates exhaustiveness, in no way adversely affects completeness. (A strategy is *depth-first along and-branches* if from among any pair of premises in an inference step, it selects one premise and determines its solvability or unsolvability before deciding that of the other premise.)

generatable in direction **F** if it is premise-free and is considered *generatable* in direction **B** if it is a reduction derivation of the goal sentence. A merit ordering of derivations is *δ-finite* for direction **X** if for any derivation D generatable in that direction there exist only finitely many other derivations, generatable in the same direction which have merit better than or equal to that of D. A good example of a δ-finite merit ordering is the one measured by the total number of distinct occurrences of symbols in a derivation and called its *symbol complexity*. For denumerable languages constructed from finite alphabets, symbol complexity has the property that, for each natural number n, only a finite number of derivations has complexity n. It follows that merit measured by symbol complexity is δ-finite, even in search spaces which have infinite or-branching. Size and level are δ-finite for search spaces which have finite or-branching but are not δ-finite for spaces with infinite or-branching. A δ-finite search strategy need not be exhaustive but always is complete.

Theorem

Any bi-directional search strategy using a δ-finite merit ordering for some direction **X** is complete, provided that at no stage does it become uni-directional in the opposite direction $\mathbf{Y} \neq \mathbf{X}$.

Proof. Suppose that D^* is some solution which is not generated by the search strategy. Then there exists some sub-derivation D of D^* which at some stage

becomes a candidate for generation in direction X but at no stage ever gets generated. Unless the search strategy eventually terminates by finding some solution other than D^*, it must otherwise select and generate in direction X candidate derivations $D_1, \ldots, D_n, \ldots$ without termination. But then at some stage, because of δ-finiteness, one of the selected derivations D_n has merit worse than D; and this is impossible.//

It seems to be a common misconception that finite or-branching is a necessary condition either for completeness or for efficiency. In particular, it is sometimes believed that resolution theorem-proving inference systems have the advantage over alternative inference systems that only a finite number of inference steps can have a given set of sentences as premises. This belief appears to be without foundation. Similarly, the condition often required in pure logic that it be effectively decidable whether a given conclusion follows directly from a set of premises by one application of an inference rule, is yet another restraint which has no apparent relevance to the construction of complete or efficient proof procedures.

It is especially important to bear in mind that infinite or-branching poses no hindrance to efficient theorem-proving when evaluating proof procedures for higher-order logic or for inference systems like those investigated by Plotkin (1972). The practical utility of constructing search spaces with infinite or-branching and of employing exhaustive search strategies is well illustrated by the efficient theorem-proving programs written by Siklossy and Marinov (1971). It is interesting to note that the notion of merit which guides their search strategy is similar to that of symbol complexity.

Another relaxation of constraints which can usefully be employed to improve the performance of search strategies is that the relative merit of derivations be allowed to depend upon the state of computation, that is, upon the entire set of derivations generated up to a given cycle of the algorithm. This liberalized notion of merit, applied to evaluation functions, has been investigated by Michie and Sibert (1972). A useful application to bi-directional heuristic search is described in the section concerned with bi-directionality.

COMPLEXITY, DIAGONAL SEARCH, AND ADMISSIBILITY

Let f be a real-valued, non-negative function defined on derivations. f is an *evaluation function* if it is used to define the merit of derivations:

D_1 has *better merit* than D_2 if $f(D_1) < f(D_2)$,

D_1 and D_2 have *equal merit* if $f(D_1) = f(D_2)$.

f is *monotonic* if no derivation has merit better than any of its sub-derivations, that is, if

$f(D') \leqslant f(D)$ whenever $D' \subseteq D$.

A *complexity measure* (or cost measure) is any monotonic evaluation function. Various measures which have been used to compute the complexity of derivations are

(1) *size*, the number of sentences in a derivation,
(2) *level*, the largest number of sentences along any one branch of a derivation,
(3) *sum cost*, the sum of all costs, for all sentences in a derivation, where every sentence has an associated cost (or complexity), and
(4) *max cost*, the largest sum of all costs, for all sentences along any one branch of a derivation.

Symbol complexity is the special case of sum cost which is obtained when the complexity of an individual sentence is measured by the number of occurrences of distinct symbols in the sentence. Size is the special case of sum cost obtained when each sentence has associated with it a unit cost. Similarly level is max cost where individual sentences have unit costs. Useful variant definitions of complexity are obtained by associating costs with inference steps rather than with individual sentences. On the whole, most of the discussion of complexity below is independent of the exact details of the definition involved. The only important property which we need require of complexity measures is that they be monotonic.

It often occurs, especially in path-finding problems, that a particular measure of complexity is given and it is required to find a simplest solution (one having least complexity). Perhaps more often no such measure is given and it is required only to find any solution, no matter what its complexity. In the latter case it is plausible that some measure of the complexity of partial solutions can usefully be employed to improve the efficiency of the search. Such a measure might be employed in a depth-first search both to extend a given derivation along simpler lines in preference to ones more complex as well as to suspend the further extension of a derivation when it has become intolerably complex. Alternatively, it may be decided to replace the original problem of finding an arbitrary solution by the related, but different, problem of finding a simplest or most elegant one. We will delay a more thorough discussion of this problem until we have first dealt with the problem of finding a simplest solution.

Diagonal search strategies

These strategies (or more precisely the special case of bounded diagonal search) are equivalent to the branch-and-bound algorithms employed in operations research (Hall 1971), to the algorithms of Hart-Nilsson-Raphael (1968) for path-finding problems, to those of Nilsson (1968) for and-or trees, and to those of Kowalski (1969) for theorem-proving graphs. These algorithms are well suited for problems where it is required to find a simplest solution. Some authors, including Hall (1971), Kowalski (1969), Nilsson (1971) and Pohl (1970), have recommended using such algorithms (or minor variations) for problems requiring any solution, regardless of its complexity.

For a given complexity function g, a search strategy employing an evaluation function f is a *diagonal search strategy* if

$$h(D)=f(D)-g(D)\geqslant 0 \text{ for all derivations } D.$$

h is called the *heuristic function* and f is often written as the sum $g+h$. The evaluation function f of a diagonal search strategy can often be interpreted as providing an estimate $f(D)$ of the complexity of a simplest solution D^* containing the derivation D. The heuristic function h then estimates that part of the complexity of D^* which is additional to that of D. A diagonal search strategy is a *complexity saturation strategy* if the complexity measure g is used as evaluation function f, that is, if $f \equiv g$ and therefore $h \equiv 0$.

A diagonal search strategy is an *upward diagonal* strategy if, whenever two candidate derivations for the same direction have joint best merit but unequal complexity, the candidate derivation selected for generation is the one having greater complexity. The use of upward diagonal strategies helps to avoid the simultaneous exploration of equally meritorious partial solutions.

The geometric intuition underlying the terminology 'diagonal' and 'upward diagonal strategies', is illustrated in figure 6. Inference steps in the search space are treated as points in a two-dimensional space. An inference step has co-ordinates (g, h) if it is generated by the search strategy as part of a selected candidate derivation D with complexity $g(D)=$g and heuristic value $h(D)$

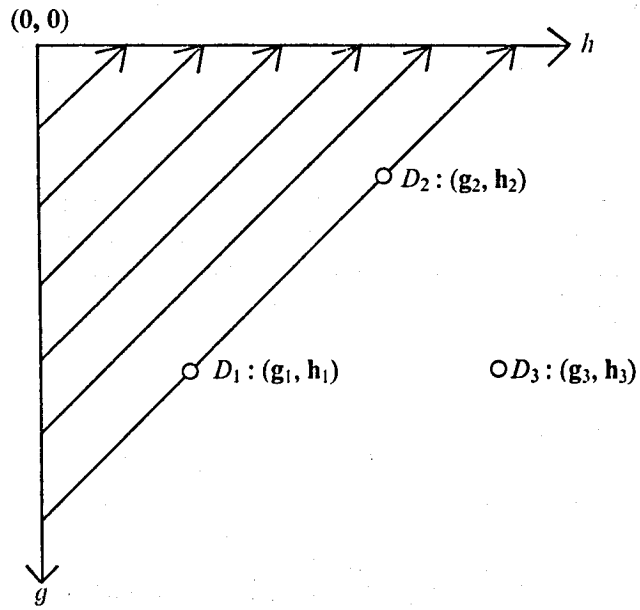


Figure 6. Whenever possible, diagonal search generates D_1 before D_2 and D_2 before D_3. It generates D_2 before D_1 only if D_2 is a sub-derivation of D_1 and generates D_3 before D_1 or D_2 only if D_3 is a sub-derivation of D_1 or D_2. If the evaluation function is monotonic then D_3 could not be a sub-derivation of D_1 or D_2, because, if it were, it would have larger evaluation. Therefore, for monotonic evaluation functions, diagonal search generates all derivations on a given diagonal before generating any on a longer diagonal. Notice that a diagonal search strategy is bounded if its evaluation function is monotonic and if $h(D^*)=0$ whenever D^* is a solution. All solution derivations lie on the g-axis.

$=\mathbf{h}$. A diagonal search strategy generates inference steps along diagonals, moving away from the origin $(\mathbf{0}, \mathbf{0})$, generating inference steps on shorter diagonals (having smaller $\mathbf{g}+\mathbf{h}$) in preference to inference steps on longer diagonals (having greater $\mathbf{g}+\mathbf{h}$). Within a diagonal, upward diagonal search moves up the diagonal whenever possible, generating inference steps lower on the diagonal (having greater $\mathbf{g}$) before inference steps higher on the diagonal (having smaller $\mathbf{g}$).

A diagonal strategy is *bounded* if for all candidate derivations D, $f(D)$ is less than or equal to the complexity of any solution containing D, that is, if

$$f(D) \leqslant g(D^*) \text{ whenever } D \subseteq D^* \text{ and } D^* \text{ is a solution.}$$

Notice that any complexity saturation strategy is bounded.

Admissibility

A search strategy is admissible if it terminates with a simplest solution, whenever the search space contains any solution.

Theorem

If a bounded diagonal strategy terminates with a solution derivation D^* then D^* is a simplest solution.

Proof. Suppose D_0 is a simpler solution than D^* and suppose that termination has occurred because no candidate in direction $\mathbf{X}$ has better merit than D^*. Then at time of termination some sub-derivation D_0' of D_0 is a candidate for generation in direction $\mathbf{X}$. Moreover D_0' has merit better than D^* because

$$f(D_0') \leqslant g(D_0) < g(D^*) = f(D^*).$$

But this is in violation of the termination condition.//

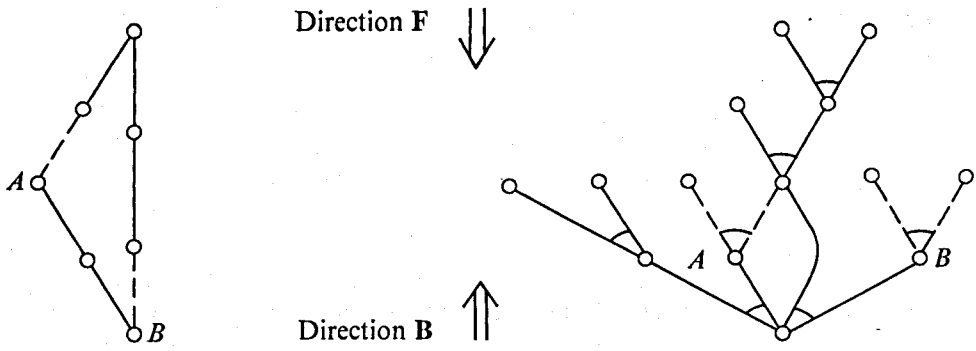
Corollary

If a bounded bi-directional diagonal search strategy is δ-finite for some direction $\mathbf{X}$, then it is admissible, provided that at no stage does it become uni-directional in direction $\mathbf{Y} \neq \mathbf{X}$.

In the case of uni-directional search either in and-or trees or in theorem-proving graphs, it can be shown that when a bounded diagonal strategy first generates a solution D^*, then no candidate for generation has merit better than D^*. Thus condition 6(b) is automatically satisfied and need not be tested explicitly. In these cases, it suffices to terminate, therefore, as soon as a first solution has been generated. Figure 7 illustrates the need for 6(b) when searching for simplest solutions in bi-directional path-finding problems and in uni-directional and-or graph search.

An interesting application of bounded diagonal search can be made in certain cases where the problem is one of enumerating all nodes in a finite search space and of verifying that they all have some property P. Under certain conditions it is possible to verify that all nodes have property P without generating all of the search space. Reformulate the problem as being that of finding some derivation of a node having property not-P. Assume that for some notion of complexity g, it is possible to construct a bounded diagonal search strategy. Assume also that the maximum complexity of any

solution is g. Then under these conditions, if a node having property not-P is not found before generating a candidate D with value $f(D)>g$, then it suffices to terminate the search and to conclude that no node in the search space has property not-P and that all nodes, therefore, have property P.



Bi-directional search for a path-finding problem

Uni-directional, and-or graph, search for a derivation-finding problem

Figure 7. For both of these problems, all inference steps in the search space are illustrated and either have already been generated or else are contained in candidates for generation. For the bi-directional search, F is the chosen direction of generation. In both cases complexity is measured by size. Assume that, at the stage of the algorithm illustrated here, both candidate derivations in each of the two search spaces have heuristic value zero. Then, in both spaces, the candidate derivation containing the node A is the candidate having best merit. When this candidate is selected and generated, a solution derivation containing A is adventitiously generated. In neither case is this solution a simplest one and its merit is worse than that of the remaining candidate derivation containing the node B.

OPTIMALITY

The Optimality Theorem of Hart, Nilsson and Raphael (1968) states that, unless it is better informed, no admissible algorithm generates fewer nodes before termination than does a bounded diagonal strategy. They define one admissible algorithm to be *better informed* than another if it admits the calculation of a better heuristic function, that is, if the heuristic information of the first strategy is represented by h_1 and if the heuristic information of the second is represented by h_2, then $h_1(D)\geqslant h_2(D)$ for all derivations and $h_1(D)>h_2(D)$ for some derivation implies that the first strategy is better informed than the second. With these definitions, Hart, Nilsson and Raphael prove the Optimality Theorem for uni-directional path-finding. Under the assumption that both admissible strategies are bounded diagonal search strategies, Kowalski (1969) proves the Optimality Theorem for uni-directional theorem-proving graph search, and Chang and Slagle (1971) prove the theorem for an uni-directional and-or graph search strategy similar to, but different from, diagonal search. Since the authors of the original Optimality

Theorem do not specify what it means for a non-diagonal strategy to employ a heuristic function, it is not clear that they have proved a stronger Optimality Theorem, which is free of the assumption that both search strategies are diagonal.

Even the weak version of the Optimality Theorem fails for bi-directional path-finding and for uni-directional and-or graph search. In both cases the attempted proof fails for the same reason. The proof requires it to be shown that if an inference step is generated by the better informed strategy then it is also generated by the other, worse informed strategy. The argument proceeds by showing that, for the latter strategy, the inference step in question belongs to a candidate derivation of merit better than that of a simplest solution and is therefore generated before the search strategy terminates. This proof works for uni-directional theorem-proving search and for the special case of uni-directional path-finding, because, in these cases, once a derivation becomes a candidate for generation it remains a candidate of unchanging merit until it is selected for generation. This is not the case, for instance, for and-or tree search. A derivation which is candidate for generation at one stage may cease to be a candidate without ever being generated at a later stage. Its ungenerated inference step may remain ungenerated as part of a new candidate derivation which contains the original one. The new derivation may have merit worse than the old. The ungenerated inference step, which once belonged to a candidate derivation having better merit than a simplest solution, may now belong only to candidates whose merit is worse than that of a solution.

Chang and Slagle rescue the proof of the Optimality Theorem by altering the definition of candidate reduction derivation. Initially, before the goal sentence has been generated, any single inference step which has the goal as conclusion is a *candidate* for generation. Afterwards, a derivation D is *candidate* for generation if some reduction derivation D_0 is a sub-derivation of D which has already been generated and D contains only generated inference steps belonging to D_0 and, for each premise of D_0, a single ungenerated inference step whose conclusion is the given premise. The effect of this definition is to assure that once an ungenerated inference step belongs to a derivation which is a candidate for generation then that derivation remains a candidate of constant merit until the inference step is generated. But it is just this property which was needed to save the proof of the Optimality Theorem.

The Chang-Slagle algorithm can be regarded as a special case of the general bi-directional algorithm and, for and-or tree search, as a special case of Nilsson's algorithm (1968). Viewed in this way, their algorithm amounts to a strategy of *breadth-first search along and-branches*, which given a reduction derivation investigates the solvability or unsolvability of all its premises in parallel. Such a strategy does not seem optimal in any intuitive sense and is demonstrably inefficient in many cases.

BI-DIRECTIONALITY

Various methods have been suggested for determining, in path-finding problems, which direction of generation should be selected in a given cycle of the algorithm. Among these are the method of alternating between **F** and **B** and of choosing that of **F** and **B** which possesses a candidate derivation of best merit. Pohl discusses these and related methods in his papers (1969, 1971). In particular, he provides theoretical and experimental arguments to support his method of *cardinality comparison* which chooses that of **F** or **B** for which the set of candidates has least cardinality. Since we are especially interested in the case where the sets of candidates are infinite, we need to formulate a different criterion for the choice of direction. Our criterion, in fact, can be regarded as a refinement of Pohl's:

Choose that of **F** *or* **B** *which has the fewest candidates of best merit.*

Notice that if a merit ordering is δ-finite for **F** or **B** then in that direction, although the number of candidates for generation may be infinite, the numbers of candidates of best merit is always finite.

A more serious problem with bi-directional searches, independent of the method for choosing direction of generation, is that the two directions of search may pass one another before an appropriate solution derivation is found and verified. The resulting bi-directional search may then generate more derivations than either of the corresponding uni-directional searches. Such a situation may occur with bounded diagonal strategies and even occurs with our present formulation of bi-directional complexity saturation strategies. As the algorithm is presently formulated a bi-directional complexity saturation strategy must both generate a simplest solution (of complexity **g**) and also verify that the solution is simplest by generating all candidates of complexity less than **g** for one of the directions **X**. Such a bi-directional strategy does all the work of a uni-directional strategy for direction **X** and, in addition, generates extra derivations in the opposite direction. The algorithm can be modified so that bi-directional complexity saturation behaves as it should: generating a simplest solution of complexity $\mathbf{g_F}+\mathbf{g_B}$ (where $\mathbf{g_X}$ is the complexity of that part of the solution which is contained in direction **X**) and generating all candidates in direction **X** of complexity less than $\mathbf{g_X}$. Most importantly, the modifications necessary to achieve this effect apply equally to the more general case of bi-directional bounded diagonal strategies. The resulting improvement in efficiency should significantly increase the utility of such search strategies. To minimize the complications involved in the following discussion, we limit ourselves to consideration of bounded diagonal search strategies for path-finding problems.

Suppose that every partial path maximally contained in direction **X** has complexity greater than or equal to **c**. (A derivation is *maximally contained* in **X** if the addition of some single ungenerated inference step makes it a candidate for generation in direction **X**.) Then no partial path *D* which is candidate for generation in the opposite direction **Y** can be contained in a

solution path whose additional complexity (in addition to that of D) is less than **c**. Therefore we may require that

$h(D) \geqslant \min(g(D'))$, for D' maximally contained in **X**.

If $h(D)$ is less than $\min(g(D'))$ then we should reset its value to $\min(g(D'))$.

Suppose that every partial path D' maximally contained in direction **X** has value $f(D')$ greater than or equal to **e**. Then **e** is a lower bound on the complexity of a simplest solution. We may require, therefore, that for all candidates D for generation in the opposite direction **Y**

$f(D) \geqslant \min(f(D'))$ for D' maximally contained in **X**.

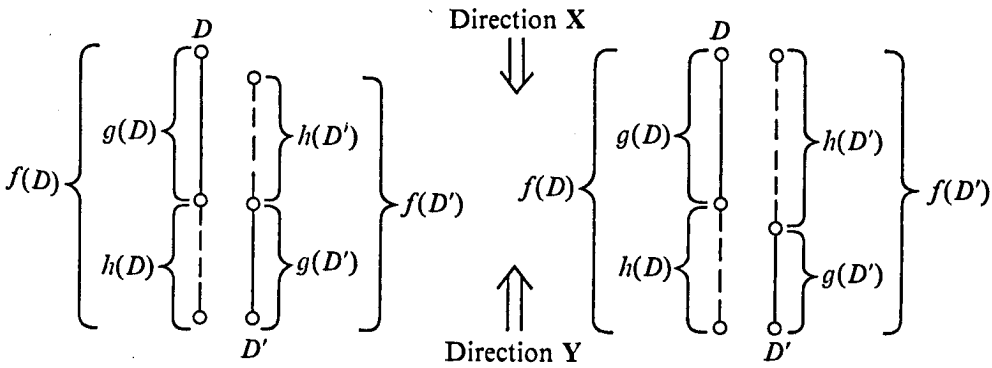
If $f(D)$ is less than the right-hand member of the inequality then we reset its value so that it is equal.

These two ways, of updating the heuristic function for direction **Y** by keeping track of h and f values for direction **X**, can be combined to obtain even better-informed heuristic functions. For D, candidate for generation in direction **Y**, we may insist that

$h(D) \geqslant \min[g(D') + (h(D') \dot{-} g(D))]$ for D' maximally contained in **X**, where

$$a \dot{-} b = \begin{cases} 0, \text{ if } a \leqslant b \\ a - b, \text{ otherwise.} \end{cases}$$

Figure 8 illustrates a proof that such a heuristic function always yields a bounded diagonal search strategy.



Case (1) $h(D') \dot{-} g(D) = 0$ implies $g(D') + (h(D') \dot{-} g(D)) = g(D')$.

Case (2) $h(D') \dot{-} g(D) > 0$ implies $g(D') + (h(D') \dot{-} g(D)) = f(D') - g(D)$.

Figure 8. Suppose that D is a candidate for generation in direction **X** and suppose that it is contained in some simplest solution D^*. Then D^* contains some partial path D' which is maximally contained in **Y**. There are two cases:
(1) if $h(D') < g(D)$ then $h(D)$ should not be less than $g(D')$.
(2) If $h(D') > g(D)$ then $f(D)$ should not be less than $f(D')$ (which is already less than or equal to $g(D^*)$). But this means that $h(D) = f(D) - g(D)$ should not be less than $f(D') - g(D)$.
In both cases $h(D)$ should not be less than $g(D') + (h(D') \dot{-} g(D))$.

FINDING ARBITRARY SOLUTIONS

It has sometimes been assumed that the problem of finding any solution can be adequately replaced by the problem of finding a simplest solution. Pohl (1970) and Nilsson (1971) adopt a more liberal position favouring the use of diagonal search strategies with an evaluation function of the form $g+\omega h$ where h is a heuristic function and $0 \leqslant \omega$. We shall argue here that such search strategies are not adequate to deal efficiently with the kind of search spaces that arise in artificial intelligence problems.

Our main objection to diagonal search strategies is that they investigate equally meritorious alternatives in parallel. For this reason, bounded diagonal strategies are even better suited for finding all simplest solutions than they are for finding any simplest solution. (Just change the termination condition so that termination takes place when all candidates for generation have merit *worse* than any solution found so far. By that time the algorithm will have generated all simplest solutions.) The point is that once a bounded diagonal strategy has found one simplest solution then it has either generated all other simplest solutions or it very nearly has. More generally when two candidates are tied for best merit, diagonal strategies typically generate one soon after the other.

Typical of the search spaces and of the heuristic functions that can be constructed for problem domains in artificial intelligence are ones where the eventually-found solution derivation is obtained from candidate sub-derivations which at some stage look worse (as viewed from the heuristic function) before looking better again. More precisely, if D^* is the eventually-found solution, then more often than not there exist sub-derivations D_1 and D_2 selected and generated before D^* (where $D_1 \subset D_2 \subset D^*$) which are such that the heuristic value of D_2 is not better than that of D_1, that is, $h(D_1) \leqslant h(D_2)$. In such a case (no matter how large an ω is used to weight the heuristic function), if no other extension of D_1 looks better than D_2, then a $g+\omega h$ search strategy will turn away from further exploration of extensions of D_1 to an exploration of any alternatives which look equally meritorious to D_1. The search strategy eventually returns to generate D_2 and then continues generating extensions of D_2 until some further extension has heuristic value no better than D_2. The upwards diagonal variant of diagonal search is useful for dealing with the case where $h(D_1)=h(D_2)$, but is unable to deal appropriately with the more usual case where $h(D_1)<h(D_2)$. The trouble with diagonal searches is that they have no persistence. They abandon a line of approach as soon as things look bad. They are short-sighted and incapable of bringing long-term objectives to bear on the evaluation of short-term alternatives.

Another more technical problem arises when we seek to apply diagonal strategies to the finding of arbitrary solutions in and-or/theorem-proving graph search spaces: what measure of complexity g should be employed by the evaluation function $g+\omega h$? The arguments in favour of using $g+\omega h$

carry no indication of how g should be measured. In the case of path-finding problems the alternatives are few in number. In the more general case we have to choose between level, size, symbol complexity and various kinds of sum costs and max costs. The choice, for instance, between level and size has an important effect on the resulting behaviour of the corresponding $g+\omega h$ strategy.

In fact, for resolution theorem-proving problems, when size is used to measure complexity and when number of literals in the derived clause is used to measure heuristic value, the resulting $g+\omega h$ strategies display intolerable inefficiency. The problem is most acute when a solution of least size contains subderivations of contradictory unit clauses of nearly equal size. Although the solution can be generated as soon as the two units have been generated, the search strategy must wait until the candidate solution becomes a candidate of best merit. In most cases this will not happen before the program has exceeded pre-assigned bounds of computer space and time. With the same heuristic function, the situation would seem to be more satisfactory when level is used to measure complexity. Whenever a unit clause is generated, an immediate attempt is made to resolve it with all other previously generated units, because any resulting solution derivation would automatically have the same merit as the unit clause just generated (or better merit in the case of upward diagonal searches). But other objections apply to using level as a measure of complexity. For a fixed amount of computer time and space a diagonal search strategy, treating level as complexity, generates fewer derivations per unit of time (or spends less time evaluating each clause it generates) than a diagonal strategy using size. By the same measure, symbol complexity is still more efficient than size. We shall further elaborate upon this theme in the following section.

It would be too lengthy to outline here some of the concrete alternatives we envisage for the construction of more efficient search strategies. We remark only that in the case of theorem-proving problems we favour strategies which seek to minimise the additional effort involved in generating a solution. Such strategies employ look-ahead and resemble depth-first strategies as much as they do diagonal ones.

SOME ARGUMENTS FOR SIMPLICITY PREFERENCE

Our arguments against the employment of diagonal search for finding arbitrary solutions are not arguments against the use of complexity for guiding the order in which alternatives are explored by an intelligent search strategy. On the contrary we believe that the complexity of candidate derivations is one of the most important factors which search strategies can employ in order to increase efficiency. For this purpose, level and max costs are misleading measures of complexity, size is more appropriate than level, and symbol complexity is more adequate than size. The arguments which support symbol complexity are both empirical and theoretical.

Empirical arguments

The increased efficiency contributed by the use of *ad hoc* heuristics in resolution theorem-proving programs has been an unmistakable, and unexplained, phenomenon. We shall argue that the most important of these heuristics are disguised first approximations to a more general principle of preference for small symbol complexity. Assuming that the argument succeeds we will have an *a priori* case for believing that the use of a single strategy of preference for small complexity will result in improved efficiency on the order of that contributed by the *ad hoc* heuristics. In fact, we will have grounds for believing more. If the use of symbol complexity unifies, subsumes and subjects to a common unit of measure otherwise diverse heuristics, and if such heuristics have a positive effect on efficiency, then we might expect an even greater improvement to result from an application of a unifying simplicity preference strategy. Such a theory could be tested by subjecting it to experiment.

It can be argued in fact that such an experiment has already been performed by Siklossy and Marinov (1971). Their program for proving theorems in rewriting systems employs a search strategy which is very nearly symbol complexity saturation. The statistics they have collected for a large number of problems substantiate the thesis that a general-purpose, exhaustive, symbol complexity preference strategy outperforms many special-purpose heuristic problem-solving programs. What is additionally impressive about their results is that they use a complexity saturation strategy which might be improved further by extending it to a bounded bi-directional diagonal search – this despite the arguments against diagonal search for finding arbitrary solutions. We conclude that the advantages obtained by generalising and extending the *ad hoc* heuristics compensate for the inefficiencies inherent in saturation and diagonal search. It remains for us now to argue our case that many of the *ad hoc* heuristics can be interpreted as first approximations to some variation of simplicity preference. We shall limit our attention to heuristics used in resolution theorem-proving programs.

(1) *Unit preference* (Wos *et al.* 1964). This heuristic gives preference to resolution operations which have a unit clause (one containing a single literal) as one of the premises. It behaves somewhat like the bounded diagonal search strategy which measures complexity by level and measures the heuristic value $h(D)$ of a derivation D by the number of literals in the derived clause. Until recently this seemed like an adequate, theoretically justified substitute for unit preference. The more radical preference for units involved in the original heuristic strategy seemed to be unjustified. However, if we measure complexity by symbol complexity, and heuristic value by the symbol complexity of the derived clause, then the resulting bounded diagonal strategy exhibits a behaviour significantly more like that of the unit preference strategy. The diagonal strategy has the advantage that it does not depend on the initial specification of an arbitrary level bound,

within which unit preference is employed, and outside of which it is not employed.

The argument here is not intended as a one-sided attack against unit preference. On the contrary, the utility of the unit preference strategy is widely recognised and is not in dispute. Indeed, we take the usefulness of unit preference as an argument for the use of complexity, and of symbol complexity more particularly, in search strategies for theorem-proving problems.

(2) *Function nesting bounds* (Wos *et al.* 1964). This heuristic preserves all clauses which have function nesting less than some initially prescribed bound and deletes those which have nesting greater than or equal to the bound. It is most effective when the user supplies a bound which is small enough to filter out a great number of useless possibilities and large enough to include all sentences in some tolerably simple solution. Unfortunately the kind of knowledge necessary for obtaining such bounds is usually nothing less than a knowledge of some simple solution and of the maximal function nesting depth of all clauses contained in the solution. In other cases, the reliability of user-prescribed function nesting bounds would seem to be highly precarious and without foundation.

Despite these reservations, the uniform application of consistently small nesting bounds has been exceptionally successful for obtaining solutions to a large number of problems (see, for some examples, Allen and Luckham (1969)). But Siklossy and Marinov (1971) make the same observation – that simple problems have simple solutions – and obtain efficient results with a graded simplicity preference strategy which does not involve the employment of some initial, arbitrary, function nesting bound. Since degree of function nesting can be regarded as an approximation to symbol complexity, we can regard the employment of function nesting bounds as an approximation to a strategy of simplicity preference which measures complexity by symbol complexity.

(3) *Preference for resolution operations involving functional agreement.* (This heuristic has been discovered independently by Isobel Smith, Donald Kuehner and Ed Wilson, at different times in the Department of Computational Logic. It has probably been noticed elsewhere with similar frequency.) Given a clause containing a literal $P(f(y))$, preference is given, for example, to resolving it with a clause containing $\bar{P}(f(t))$ rather than with one containing $\bar{P}(x)$. This heuristic generalises to one which prefers resolving pairs of literals which have many common function symbols in preference to others which have fewer common symbols. Such a heuristic favours the resolution operation which involves the least complicated unifying substitution and therefore the least complicated resolvent as well. The operational effect of this heuristic can be obtained by preference for small symbol complexity.

(4) *Other heuristics*, such as preference for equality substitutions which simplify an expression in strong preference to those which complicate it, are

obvious and direct applications of the principle of preference for least symbol complexity.

Theoretical arguments

Arguments for the use of symbol complexity can be obtained from entirely theoretical considerations.

(1) We consider, as an argument for the use of simplicity preference, the organising and simplifying effect which such search strategies have in a theory of efficient proof procedures. Thus, for example, we count in its favour the fact that symbol complexity provides a means for constructing complete search strategies for search spaces with infinite or-branching. The assumption that search strategies generate simple proofs in preference to more complex ones is necessary for a formal justification of the intuitive conviction that deletion of tautologies and subsumed clauses increases the efficiency of resolution theorem-proving programs (Kowalski 1970, Meltzer 1971). A greater improvement in efficiency can be demonstrated when complexity is measured by size rather than by level. Still greater improvement follows when symbol complexity is used instead of size. Similar assumptions about the employment of simplicity preference strategies are necessary for the proofs of increased efficiency applied to linear resolution and SL-resolution (Kowalski and Kuehner 1971). In all of these cases, the assumption that search strategies prefer simple proofs to ones more complex is a prerequisite for all proofs of increased efficiency. We regard the unavoidability of such an assumption in a theory of efficient proof procedures as an argument in favour of the implementation of such strategies.

(2) Assume that, within a given search space, all derivations have the same probability of being a solution derivation. Assume that, on the average, all symbol occurrences occupy the same storage space and require the same amount of processing time. Then, for a fixed finite quantity of space and time, symbol complexity saturation generates more derivations and therefore is more likely to generate some solution than is any other search strategy for the same search space. To the extent that not all derivations have equal probability of being a solution, and to the extent that such information can be made available to the search strategy, symbol complexity needs to play a role subordinate to such knowledge. None-the-less, the point of our argument stands. Other considerations being equal, a search strategy which generates derivations containing few symbol occurrences, in preference to others which contain more, maximises the probability of finding a solution derivation within fixed bounds on computer time and space.

(3) Symbol complexity is seemingly a most obvious example of the kind of purely syntactic concept which has come under increasingly more vigorous attack in the community of artificial intelligence workers. The argument seems to be that emphasis upon uniform, general-purpose, syntactic notions is detrimental to, and perhaps even incompatible with, progress in com-

municating special-purpose, problem-dependent semantic and pragmatic information to problem-solving programs. It is not our intention to wage here a lengthy counter-attack against this position. We shall instead outline a counter-proposal which aims to reconcile the opposing sides of this dispute. We shall argue that special-purpose, domain-dependent semantic and pragmatic information is inevitably reflected in the syntactic properties of sets of sentences. Among the most important of these properties is symbol complexity.

Suppose that in a given language, some definable concept occupies a semantically-important and pragmatically-useful role for solving problems posed in the language. Unless the concept is given a name, its definition in terms of other named concepts may be exceedingly complex. The data processing of propositions about the concept will be similarly complicated and consuming both of space and time. In order that data processing be made as efficient as is necessary to reflect the importance and utility of the concept, the concept is given a name, which serves the effect of lowering its symbol complexity as well as the complexity of propositions concerned with the concept.

What can be argued about concepts can also be argued about propositions. Suppose that some derivable proposition occupies a centrally-important semantic and pragmatic role for solving problems posed in the language. Then the more important this role, the more accessible the proposition needs to be made to the problem-solver. It will not do if a useful proposition needs to be re-derived in a complicated and time-consuming way every time it needs to be applied. In order that data processing be done efficiently, axioms are chosen and readjusted so that important and useful facts are accessed by simple derivations.

To summarise, our argument has been that, in order for important and useful concepts and propositions to be stored and processed efficiently, it is necessary that they be associated with small symbol complexity. We do not claim that symbol complexity is the only way of communicating useful, special purpose information to a problem-solving system. We do maintain, however, that it is one of the most important ways. This is not to argue that symbol complexity has been used as we suggest it be used. Indeed practice has been totally unconcerned with specifying languages whose syntax reflects their pragmatics. Perhaps it is an argument against this practice that is one of the more important and useful contributions of those who, otherwise, seem to be arguing against reliance on general-purpose, syntactic methods in problem-solving systems.

Acknowledgements

This research was supported by a Science Research Council grant to Professor B. Meltzer. Additional support came from an ARPA grant to Professor J. A. Robinson, during a visit to Syracuse University, as well as from a French Government grant to Dr A.

Colmerauer, during a visit to the University of Aix-Marseille. Thanks are due, for their useful criticism, to my colleagues, Bob Boyer, Pat Hayes and Ed Wilson, in the Department of Computational Logic, and to Mike Gordon and Gordon Plotkin in the Department of Machine Intelligence.

REFERENCES

Allen, J.R. & Luckham, D. (1969) An interactive theorem-proving program. *Machine Intelligence 5*, pp. 321–36 (eds Meltzer, B. & Michie, D.) Edinburgh: Edinburgh University Press.

Chang, C.L. & Slagle, J.R. (1971) An admissible and optimal algorithm for searching and-or graphs. *Art. Int.*, **2**, 117–28.

du Feu, D. (1971) An application of heuristic programming to the planning of new residential development. *Department of Computational Logic Memo No. 49*, University of Edinburgh.

Hall, P.A.V. (1971) Branch-and-bound and beyond. *Proc. Second Int. Joint Conf. on Art. Int.*, pp. 941–50. The British Computer Society.

Hart, P.E., Nilsson, N.J. & Raphael, B. (1968) A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. on Sys. Sci. & Cyber.*, **4**, 100–7.

Kowalski, R. (1969) Search strategies for theorem-proving. *Machine Intelligence 5*, pp. 181–201 (eds Meltzer, B. & Michie, D.) Edinburgh: Edinburgh University Press.

Kowalski, R. (1970) Studies in the completeness and efficiency of theorem-proving by resolution. Ph.D. Thesis, University of Edinburgh.

Kowalski, R. & Kuehner, D. (1971) Linear resolution with selection function. *Art. Int.*, **2**, 227–60.

Kowalski, R. (1971) An application of heuristic programming to physical planning. *Department of Computational Logic Memo No. 41*, University of Edinburgh.

Meltzer, B. (1971) Prolegomena to a theory of efficiency of proof procedures. *Artificial Intelligence and Heuristic Programming*, pp. 15–33. Edinburgh: Edinburgh University Press.

Michie, D. & Ross, R. (1969) Experiments with the adaptive Graph Traverser. *Machine Intelligence 5*, pp. 301–18 (eds Meltzer, B. & Michie, D.) Edinburgh: Edinburgh University Press.

Michie, D. & Sibert, E.E. (1972) Some binary derivation systems. *Department of Machine Intelligence Internal Report*, University of Edinburgh.

Nilsson, N.J. (1968) Searching problem-solving and game-playing trees for minimal cost solutions. *IFIPS Congress preprints*, H125–H130.

Nilsson, N.J. (1971) *Problem-solving Methods in Artificial Intelligence*. New York: McGraw-Hill.

Plotkin, G.D. (1972) Building-in equational theories. *Machine Intelligence 7*, paper no. 4 (eds Meltzer, B. & Michie, D.). Edinburgh: Edinburgh University Press.

Pohl, I. (1969) Bi-directional and heuristic search in path problems. *SLAC Report No. 104*, Stanford, California.

Pohl, I. (1970) Heuristic search viewed as path-finding in a graph. *Art. Int.*, **1**, 193–204.

Pohl, I. (1971) Bi-directional search. *Machine Intelligence 6*, pp. 127–40 (eds Meltzer, B. & Michie, D.). Edinburgh: Edinburgh University Press.

Siklossy, L. & Marinov, V. (1971) Heuristic search vs. exhaustive search. *Proc. Second Int. Joint Conf. on Art. Int.*, pp. 601–7. The British Computer Society.

Wos, L.T., Carson, D.F. & Robinson, G.A. (1964). The unit preference strategy in theorem-proving. *Proc. AFIPS 1964 Fall Joint Comp. Conf.*, **25**, 615–21. Washington DC: Spartan Books.

11

An Approach to the Frame Problem, and its Implementation

E. Sandewall

Computer Sciences Department
Uppsala University

Abstract

The frame problem in representing natural-language information is discussed. It is argued that the problem is not restricted to problem-solving-type situations, in which it has mostly been studied so far, but also has a broader significance. A new solution to the frame problem, which arose within a larger system for representing natural-language information, is described. The basic idea is to extend the predicate calculus notation with a special operator, *Unless*, with peculiar properties. Some difficulties with *Unless* are described.

THE FRAME PROBLEM

This paper proposes a method for handling the *frame problem* in representing conceptual, or natural-language-type information. The method is part of a larger calculus for expressing conceptual information, called PCF-2, which is described in Sandewall (1972), and which is a modification and extension of Sandewall (1971a). The method proposed here is identical to the approach of mine which B. Raphael mentions at the end of his paper on the frame problem (Raphael 1971).

Previous work on the frame problem

The frame problem is the following: suppose we have a sequence of situations (in the sense of McCarthy 1968) in an ordinary-world-type environment, and that we know all 'free-will' actions that have been performed between these situations, that is, all actions which we do not wish to explain purely by laws of cause and effect within the environment. This information enables us to make deductions about properties of the environment in each situation. In practice, most properties stay set unless something changes them. For example, the color of an object usually stays the same when the object is

moved, and of course when it is left alone. However, it may occasionally change, and we might be able to realize this by a trivial deduction (for example, if the object was re-painted) or by a more complex one (for example, if the object was coated with fluorescent color and somebody blew a fuse so that the UV lamp went out). The problem then is to find a deduction mechanism which changes properties when they have to change, which retains them when they do not change, and which (this is perhaps the most difficult part) does this with a moderate number of axioms and moderate amounts of processing time.

This problem may seem odd at first, but it is a very real problem in the design of systems that reason about sequences of actions. This includes problem-solving systems (which are supposed to find a sequence of actions that leads to a given goal), but also some other cases. For example, an intelligent CAI system along the lines of Carbonell's (1970) system, but for history instead of geography, would certainly need to reason about sequences of actions, and therefore would need a solution to the frame problem. In this paper we are concerned with the frame problem for general use, as exemplified by the history application, but not with short-cut methods for problem-solving in simple environments.

Several previous authors have discussed the frame problem and proposed solutions for it, particularly Raphael (1971), Fikes and Nilsson (1971), and McCarthy and Hayes (1969, 1971a, 1971b). The work of Hewitt (1968–70) and the descendant work of Sussman and Winograd (1970) is also quite relevant to the frame problem, as has been pointed out, for example, by Hayes (1971b).

Let us very briefly comment on the proposed approaches from our viewpoint of 'general use'.

1. *The general frame axiom approaches* of McCarthy and Hayes (1969) and Hayes (1971a). In these approaches one attempts to write very general rules for when properties are retained, and their retention then has to be re-proved for every new situation. We doubt if it is possible to get a set of frame axioms with sufficient coverage, and we therefore reject these approaches.

2. *Consistency-based methods.* This approach is discussed by Hayes (1971b) and is based on work by Rescher (1964) and Simon and Rescher (1966). The basic idea is to remove properties only when they cause an inconsistency. Some conventions are needed for deciding which property is responsible for a detected inconsistency. We believe that this method could be useful, but that it will be very costly in computation time, so that a more immediate method is desirable.

3. *The* STRIPS *approach* of Raphael (1971) and Fikes and Nilsson (1971). Here facts are classified (on syntactic grounds) into 'primitive' and 'non-primitive'. Every action is characterized by a schema which specifies which primitive facts in the situation are added and deleted by the action. Other primitive facts are retained automatically. Non-primitive facts must in

principle be re-proved from the primitive facts in every new situation, although in practice one can design the program so that the deduction need not be re-performed.

One can think of cases for which this scheme is not sufficient, for example, Hayes' example (1971a): if a cup is standing on a saucer and the saucer is moved, the cup usually comes with it, but if only the cup is moved, the saucer (usually) does not come with it. We would like to add another example of a different nature. Let us make the reasonable assumption that friendship assumes that you are alive, that is, it is impossible for a dead person to have friends, or to have a dead person as a friend. We then want the 'friendship' property to be 'turned off' when a person dies. However, we do not wish to burden the schemas associated with actions like 'die', 'kill', and so on, with information regarding the cessation of friendship, and all other things that change at such a time. On the other hand, if 'to be a friend of' is to be a non-primitive, then we do not see what the supporting primitive property (-ies) could be. From such examples, we conclude that the STRIPS approach is probably limited to the problem-solving-type situations for which it is presently being used.

4. *The* PLANNER *approach.* Since PLANNER is (among many other things) a proposed programming language, any other approach would be a PLANNER approach in the sense that it could be implemented in PLANNER. However, there is one way of handling the frame problem, using the PLANNER primitives in a straightforward fashion, which we shall discuss here.

PLANNER enables the user to write rules which specify 'things that can be done'. Such a rule might, for example, correspond to a STRIPS schema. When the STRIPS schema adds a fact, PLANNER would add the corresponding fact to the data base using the primitive *thassert*. When the STRIPS schema states that a fact should be removed, the corresponding PLANNER rule would contain a command to remove the fact from the data base, using the primitive *therase*. Some care must be taken; for example, it is necessary to do the *therase*s of a situation transformation before the *thasserts*, so that fresh facts are not erased.

Although the PLANNER approach has several points in common with the STRIPS approach, the above two examples (cup and saucer, and dying friend) do not present any difficulties in themselves. Both of them can be handled by doing forward deduction of the form

$$\textit{therase}\ (A) \supset \textit{therase}\ (B).$$

Moreover, the programming language aspect of PLANNER makes it possible to counter any proposed, single counterexample, in the worst case by writing a piece of code. However, we depart from the PLANNER approach for another reason, which will be discussed in the next section.

EPISTEMOLOGY VS. HEURISTICS

McCarthy and Hayes (1969) discussed possible criteria for adequacy for a proposed notation, and made the following distinction: 'A representation is called epistemologically adequate for a person or machine if it can be used practically to express the facts that one actually has about the aspect of the world .... A representation is called heuristically adequate if the reasoning processes actually gone through in solving a problem are expressible in the language'. In our case, the 'aspect of the world' is of course the natural-language aspect, rather than, for example, the quantum physical aspect. Natural language itself is one epistemologically adequate representation, but (as discussed in Sandewall 1971b) we want another one which is more suitable for the task at hand, that is, question-answering, problem-solving, and other reasoning by computer.

Notice that the word 'heuristic' is here used in a slightly broader sense than usual. If we have a system which performs its reasoning by breadth-first search (which is not 'heuristic' in the ordinary sense), and if this and other search strategies are describable within the language, then the language is here termed heuristically adequate. However, the major reason for having a heuristically adequate system is of course to be able to communicate non-trivial heuristic guidance to the system.

The PLANNER approach to computer reasoning makes it a virtue to integrate epistemological and heuristic information, and the PLANNER language is the notation for expressing these together. We believe that this is permissible for problem environments of moderate complexity. However, for real-life problem environments, the task of writing a reasoning program is so complex that it is necessary to divide it into sub-tasks, and one very reasonable division is to study the epistemological problems first, and to add the heuristics afterwards, rather than try to do everything at once. This is particularly attractive since a set of rules *without* heuristic information (or other control information) is much less connected, and therefore much more modular, than the rules with heuristic information.

In this context, by epistemological information we mean a notation together with a set of rules (for example, logical axioms) which describe permissible deductions. The predicate-calculus notation proposed in our previous reports (1971a, 1971b, 1972) is intended to serve in this fashion. However, we stress that such systems are intended as *a basis for* programming in some programming language, and that it is essential that heuristic information is later added. Attempts to feed such systems into uniform-strategy theorem-provers (that is, to use them as *a substitute for* programming) are bound to fail for efficiency reasons.

The argument has sometimes been raised, at least in informal discussions at conferences, that epistemology *cannot* be separated from heuristics, that is, that in order to express facts in a convenient fashion, one must utilize statements about or make assumptions about when and how these facts are to be

used in the reasoning process. The discussion of approaches to the frame problem above might seem to verify that argument, since the PLANNER approach seemed the most satisfactory one. If we want to argue the distinction between epistemology and heuristics, we should therefore provide a mechanism which handles the frame problem *without* using heuristic information.

The present paper intends to do this – using a mechanism, the *Unless* operator, which is an extension to predicate-calculus notation, and which adequately handles the frame problem. At the same time, the *Unless* operator is independent of the theorem-prover or other interpreter that executes the search (in the data base), and of the heuristic information that governs this interpreter. We shall outline several different execution strategies, all of which are compatible with the intended meaning of the *Unless* operator. In that sense, our notation stays on the epistemological level, and is neutral with respect to heuristics.

THE HOST NOTATION

As was previously mentioned, our approach to the frame problem is part of a larger system for expressing conceptual information, called PCF-2. The full PCF-2 notation is too extensive to be described here, but for the present purpose it is sufficient to describe a subset.

We utilize the following sorts:

objects (which may be physical objects, or persons)
properties (e.g. 'red', 'angry', 'male')
situations (in McCarthy's sense of the word)
actions (e.g. 'to walk', 'to smoke', 'to smoke cigarettes in bed', 'to push', 'the robot pushing', 'to push a box', 'to push the box called :box12')

The following relations and functions are needed:

IS: object × property × situation
states that the object has the property in the situation.

INN: action × situation
states that the action occurs in the situation.

Case functions, of which there are several, one for each Fillmoresque 'case'. Their sorts are usually

$$\begin{cases} \text{action} \times \text{object} \rightarrow \text{action} \\ \text{action} \times \text{property} \rightarrow \text{action} \end{cases}$$

They are used to construct composite actions (for example, 'to smoke cigarettes' from simpler actions (for example, 'to smoke')

Succ: situation × action → situation
Maps a situation into that successor situation which results if the action is taken.

Additional relations and functions are needed, for example, to characterize the hierarchy of properties (human – mammal – animate . . .) or the succession ordering of situations, but they are not of interest here.

THE APPROACH TO THE FRAME PROBLEM

Given the system above, we have two versions of the frame problem, one for properties and one for actions. In the former, we want an object to retain a property (for example, 'red') until the property is explicitly changed (for example, by repainting). In the latter, we want an action to last (for example, 'John sleeping') until the action is explicitly discontinued by another action (for example, 'Peter waking up John'). Let us outline the solution to the property version of the problem.

We introduce a ternary relation

ENDS: object × property × situation

and a *frame inference rule* which with some simplification may be written as

$$\frac{\text{IS}(o, p, s),\quad \textit{Unless}(\text{ENDS}(o, p, \textit{Succ}(s, a)))}{\text{IS}(o, p, \textit{Succ}(s, a))}$$

The *Unless* operator makes this rule peculiar. The rule is intended to mean: 'if it can be proved that IS(o, p, s), and if it can not be proved that ENDS$(o, p, \textit{Succ}(s, a))$, then it has been proved that IS$(o, p, \textit{Succ}(s, a))$'.

The solution to the other, action version to the frame problem is analoguous.

This approach to the frame problem gains its strength (as compared to, for example, STRIPS) from the fact that one can make deductions to any depth using the predicate ENDS. Thus in the dying friend example, one could have a rule to the effect 'if x ends being alive, then x ends being a friend of y'. In the cup and saucer example, one would have 'if x supports y, and x moves to l, then y moves to l', plus 'if x moves, then x ends being where it was'. (Writing these rules out in the formalism is a trivial task.) Furthermore, this approach also makes it possible to do deductions backwards in time, such as 'since A is the case now, B must have been the case in the previous situation', where B or conclusions from B may then be used in *Unless* clauses. Such deduction may appear in history-type reasoning, although probably not in problem-solving applications.

THE *UNLESS* OPERATOR

The introduction of the *Unless* operator is a drastic modification of the logic. It even violates the extension property of almost all logical systems, which says that if you add more axioms, everything that used to be a theorem is still a theorem. However, it is not obvious whether this is serious. Our reason for using predicate calculus in the first place was that we wanted a *notation* in which to express conceptual information, and which could serve as a basis for a computer program that is expected to become large and complex. In other words, predicate calculus is only used for its syntax. The *Unless* operator extends the notation, but it is still reasonably clear how it is implemented in practice in a 'backward-search' algorithm. If we have the above frame inference rule of the form

$$\frac{A \quad \textit{Unless } B}{C}$$

and if we wish to prove C, we first make it a sub-problem to prove A. If we succeed, we then make it a sub-problem to prove B. If we succeed in this proof, then the proof of C did not succeed, and vice versa. Programming-wise, it should not be a problem that the proof of B might involve another *Unless*-clause. The fact that the search for a proof of B might have to be interrupted should not bother us too much. In an advanced system, the search for a proof for B might later be resumed in a final check-out of a proposed plan, or line of reasoning. This implementation of the *Unless* operator is similar to the PLANNER *thnot*, and in fact PLANNER would be a convenient (although expensive) implementation language for this system.

However, the fact that the *Unless* operator has some dirty logical properties should not be completely ignored. In one form or another, these difficulties are bound to appear in any implementation of an *Unless* operator. They are intrinsic and cannot be evaded, for example, by the choice of programming language. In this paper, we shall merely draw attention to some of these problems, without attempting to provide a solution. In the sequel, we shall permit *Unless* in wffs, for example, in axioms, with the obvious intended meaning.

First, consider the axioms

$$A$$
$$A \wedge \textit{Unless}(B) \supset C$$
$$A \wedge \textit{Unless}(C) \supset B$$

Clearly we cannot permit both B and C to be theorems simultaneously. Which of them is a theorem, if any? One possible approach to resolving the question is to use a precedence ordering on the axioms. Another approach is to restrict the structure of permitted axioms so that such situations can not occur. A third possibility is to bite the sour apple and accept that theoremhood is three-valued: theorem, not a theorem, or undetermined.

If the above backward-search algorithm is asked to prove B, it will create the sub-questions B and C alternatingly while digging down into the recursion. Its final answer (theorem or not theorem) will depend on which of the two sub-questions is at the break-off point. This is of course not satisfactory. One would like to have a definition of theoremhood which uses relatively conventional logical methods, that is, by specifying a procedure which will generate all theorems and only them. The practically useful backward-search algorithm would then be modified so as to approximate the theorem generator. However, it is not easy to set up such a procedure which also satisfies our intuition about how the *Unless* operator would behave. Consider, for example, the following seemingly very natural definition:

We define a restricted wff(rwff) as a two-tuple $\langle e, s \rangle$, where e is a wff and s is a set of wff. We extend all inference rules for wff,

$$e_1, e_2, \ldots, e_n \vdash e$$

into corresponding inference rules for rwff of the form:

$$\langle e_1, s_1\rangle, \langle e_2, s_2\rangle, \ldots, \langle e_n, s_n\rangle \vdash \langle e, s_1 \cup s_2 \cup \ldots \cup s_n\rangle$$

Moreover, we add the inference rule

$$\langle \mathit{Unless}(B) \supset e, s\rangle \vdash \langle e, s \cup \{B\}\rangle$$

(The notation is sloppy, but the intention should be clear.) We then specify the following inference mechanism for rwff:

1. From a set $A=\{a_i\}$ of axioms which are wffs, we construct the set
$$X_0=\{\langle a_i, \varnothing\rangle\}$$
where $\varnothing$ is the empty set.
2. For every X_i, we construct Y_{i+1} by adding the results of all possible one-step applications of inference rules.
3. From Y_{i+1}, we construct X_{i+1} by deleting all $\langle e, s\rangle$ such that $e' \in s$ where $\langle e', s'\rangle \in Y_{i+1}$.
4. For a given e: if there exists some s and K such that $\langle e, s\rangle \in X_k$ for all $k>K$, then e is a theorem.

With this procedure, neither B nor C in the above example would be theorems. Unfortunately, in the following example,

$$A$$
$$A \wedge \mathit{Unless}(B) \supset C$$
$$A \wedge \mathit{Unless}(C) \supset D$$
$$A \wedge \mathit{Unless}(D) \supset E$$

our intuition would permit E to be a theorem, but the above algorithm would not. It seems that more work is needed on this problem.

TIME-SEQUENTIAL DEDUCTION

In the face of the peculiarities of the *Unless* operator, it is tempting to use the selected frame inference rule in a deduction which proceeds 'forward' in time. Such a deduction might be performed as follows:

1. Collect all information about the initial situation. Make deductions from it, as long as the conclusions refer to the initial situation. (In practice, it may of course be necessary to interrupt this deduction.)
2. Determine all unconditional information (not relying on *Unless* conditions) about the next situation, for example, by using information about the action that led to this new situation.
3. Add the frame inference rule, and make forward deduction as far as possible about the new situation.
4. Transfer properties from the old situation, except in those cases when it is blocked by an 'ENDS' relation.
5. Remove all 'ENDS' relations. Then go to 2.

This procedure belongs to the same class of methods as STRIPS and the PLANNER method, whereas the general *Unless* deduction that was mentioned above is more similar to the consistency-based methods.

In a problem-solving situation, where we search a tree of possible futures for a good plan, we could strongly limit the forward deduction in step 3

during the planning phase. In the checkout phase, when we have found a plan, we would then go back and continue the deduction in step 3 in order to detect whether something could go wrong in the plan.

This time-sequential algorithm is not completely general, since it assumes that all deduction is performed within a situation, or from a situation to its successors. It will therefore not be sufficient if we require deductions 'backwards' in time, saying 'since *A* is the case in the present situation, *B* must have been the case in the previous situation'. Such deductions do not need to be common, but they do exist. We therefore believe that the time-sequential deduction is satisfactory in a planning phase, where one can ignore the backward-time deductions, but that a more general procedure, which can handle the full power of the *Unless* operator, is needed in the analysis of a given history, and possibly also in the checkout of plans.

Finally, let us remark that the *Unless* operator may be useful in some other cases besides the frame problem. For example, in handling hypothetical situations ('suppose you had been born two years later') we wish to assume all facts of our ordinary situation *unless* they are explicitly or implicitly changed by the hypothesis. There is some discussion about this in section 13 of Sandewall (1972). Here, again, a comparison with the consistency-based method of Rescher is relevant: the *Unless* operator is less elegant, but closer to a practical implementation.

Acknowledgements

This research was supported by the Swedish Natural Science Research Council under grant Dnr 2654-006.

REFERENCES

Carbonell, J.R. (1970) Mixed-initiative man-computer instructional dialogues. *BBN Report 1971*, Job no. 11399. Cambridge, Mass.: Bolt, Beranek & Newman Inc.

Fikes, R.E. & Nilsson, N.J. (1971) STRIPS: a new approach to the application of theorem proving to problem solving. *Art. Int.*, **2**, 189–208.

Hayes, P.J. (1971a) A logic of actions. *Machine Intelligence 6*, pp. 495–520 (eds Meltzer, B. & Michie, D.). Edinburgh: Edinburgh University Press.

Hayes, P.F. (1971b) The frame problem and related problems in artificial intelligence. *A.I. Memo 153*, Stanford Artificial Intelligence Project. California: Stanford University.

Hewitt, C. (1968) PLANNER: a language for manipulating models and proving theorems in a robot. *A.I. Memo 168*, Artificial Intelligence Project MAC. Cambridge, Mass: MIT.

McCarthy, J. (1968) Situations, actions, and causal laws. *Semantic Information Processing*, pp. 410–17 (ed. Minsky, M.). Cambridge, Mass.: MIT Press.

McCarthy, J. & Hayes, P.J. (1969) Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence 4*, pp. 463–502 (eds Meltzer, B. & Michie, D.). Edinburgh: Edinburgh University Press.

Raphael, B. (1971) The frame problem in problem-solving systems. *Artificial Intelligence and Heuristic Programming*, pp. 159–69 (eds Findler, N.V. & Meltzer, B.). Edinburgh: Edinburgh University Press.

Rescher, N. (1964) *Hypothetical Reasoning*. Amsterdam: North Holland Press.

Sandewall, E. (1971a) Representing natural language information in predicate calculus. *Machine Intelligence 6*, pp. 255–77 (eds Meltzer, B. & Michie, D.). Edinburgh: Edinburgh University Press.

Sandewall, E. (1971b) Formal methods in the design of question-answering systems. *Art. Int.*, 2, 129–46.

Sandewall, E. (1972) PCF-2, a first-order calculus for expressing conceptual information. *Computer Science Report*. Uppsala: Computer Sciences Department, Uppsala University.

Simon, H.A. & Rescher, N. (1966) Cause and counterfactual. *Philosophy of Science*, 323–40.

Sussman, G.J., Winograd, T. & Charniak, E. (1970) Micro-Planner Reference Manual. *A.I. Memo 203*, Artificial Intelligence Project MAC. Cambridge, Mass: MIT.

12

A Heuristic Solution to the Tangram Puzzle

E. S. Deutsch and K. C. Hayes Jr.
Computer Science Center
University of Maryland

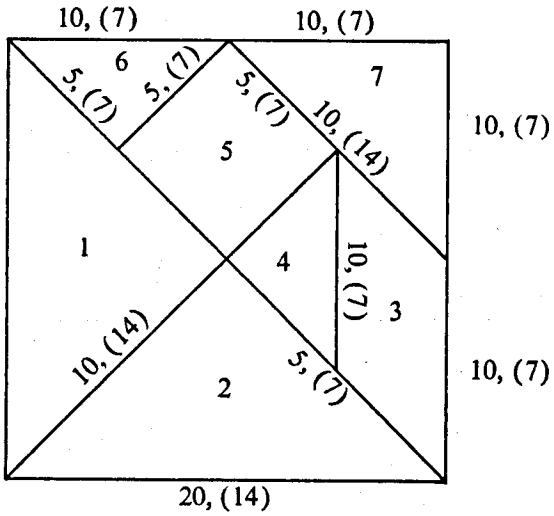
Abstract

A heuristic program leading to the solution of tangram puzzles is described. The program extracts puzzle pieces using a set of rules which search for piece-defining edges. The rules decrease in their rigor, and hence in their reliability, in the sense that the edge requirements become more lax. Such edges include those constructed during the solution process. Composites of puzzles are also formed and are treated like puzzle pieces. The solution procedure is such that the most reliable rules are applied recursively as often as possible. It is only when the solution process comes to a halt that the lower reliability rules are applied in order for the process to continue. Sometimes it is necessary to commence with one of the weaker rules after which a return to the more reliable rules is made.

1. INTRODUCTION

The tangram problem is essentially a puzzle problem but its solution has very little in common with the solution to the familiar jigsaw puzzle (Freeman and Garder 1964). For whereas the jigsaw is composed of a host of pieces of many different shapes, the tangram consists of seven pieces only, all of which have a very simple shape. Furthermore, the way in which the jigsaw puzzle pieces are combined is unique; with the tangram there is a great variety of ways in which the seven pieces can be combined so as to form different shapes.

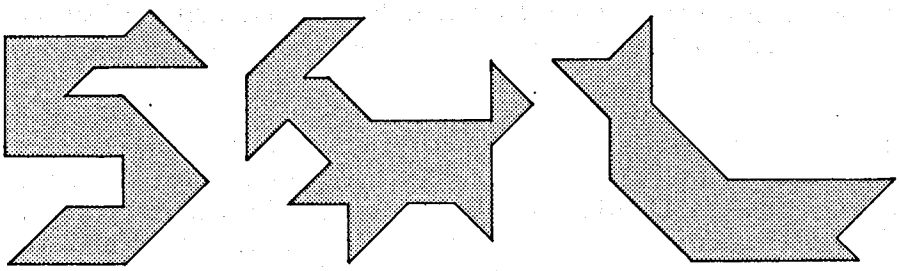
The tangram consists of the following seven pieces: two large right triangles, two small right triangles, one medium-size right triangle, a square and a rhomboid. They can all be combined to form the puzzle shown in figure 1(a). Two points of interest should be noted. Firstly, all the pieces have edges inclined to one another at an angle which is a multiple of 45°. Secondly, it should be observed that the pieces are either three or four sided. The first



(a)

Figure 1(a)

property facilitates the display of a tangram puzzle on a rectangular matrix of points for analysis purposes. This does not apply to the tangram globally, as some edges may be inclined at angles other than multiples of 45°. The second property reduces the number of common edges two or more pieces may share.



(b)

Figure 1(b)

Given a tangram shape, examples of which are given in figure 1(b), the problem consists of assembling the seven tangram pieces in such a manner so that the final assembly corresponds to the given shape. This report discusses a possible computer solution to this problem. It should be stated that the solution suggested is not a generalized one in the sense that it does not purport to solve every tangram puzzle available. Our source of tangram

puzzles is Read's book *Tangrams* (Read 1965), and it contains, for example, puzzles employing two tangram sets (fourteen pieces), and puzzles containing holes. These types of puzzles are more complicated in their structure and it was considered better to concentrate on a solution to simpler shapes first.

2. GENERAL APPROACH

At least two approaches come to mind with reference to this kind of problem. In the first instance one could resort to the use of combinatorics, whereby the computer program would attempt to combine the seven tangram pieces in a variety of ways until, by trial and error, a solution was reached. Some combination rules would clearly be incorporated in this program and the approach would definitely have a characteristic combinatorics flavor about it. A principal objection to this approach is the large amount of computation involved.

On the other hand, a solution could be attempted from the point of view of heuristic programming. With this approach one would develop a program which would first perform a series of tests and tentative partitioning operations on the given shape. These would then be evaluated somehow, or ordered, in such a way that the contribution of each test would be used to decide how an individual piece, or a set of pieces, should be placed in conjunction with the remaining pieces. Ideally, the program should incorporate back-tracking facilities by means of which the analysis procedure can resort to some information gleaned from a previously discarded test. Furthermore, the program should incorporate a facility whereby the analysis can be started afresh, using another set of tests, if no solution is obtained so far. This is the approach to be followed here.

The solution to the tangram problem to be described below hinges upon the need for the ranking of tests and rules. Examination of a number of tangram puzzles showed that some tests contributed more to the solution of the puzzle than did others. However, the latters' contribution becomes invaluable later on during the solution process, once the earlier valuable tests had little further effect. At the same time, some puzzles could not be solved without resorting to the less powerful tests first, merely in order to initiate the solution process. In these cases the usually more powerful tests or heuristics yielded nothing at all. Specific examples will be dealt with in a later section where this point will be amplified; however, the desirability of establishing an ordered list of heuristics should be clear.

Now, it could be argued that all the tests should be performed in parallel. It should, however, be realized that as the solution proceeds, there is a high degree of interdependence between tangram pieces that have already been taken up, even provisionally, to form a partial solution and the remaining, as yet unplaced, pieces. For example, it may be decided at a certain stage in the solution process that the square be placed in a particular position. Care must then be exercised by the program as to whether a similar effect would

not be realizable had the two smaller triangles been placed in that position instead. The square piece would then be used for a later insertion. Alternatively, the square piece may no longer be available at this stage, in which case the solution method must incorporate the possibility of substituting the two small triangular pieces for the square piece. Since there is considerable interdependence between the partial solution and the as-yet-unsolved remainder of the puzzle, the ranking of tests and back-tracking are quite necessary.

The following strategy was adopted for the mechanized solution of the tangram problem. Various *global* operations are to be applied to the entire given puzzle first. The purpose of these operations would be to split the puzzle into, hopefully, easier subpuzzles. In addition, these operations would yield some tentative guidelines as to the way in which some of the individual tangram pieces should be placed in the puzzle. A solution, entire or partial, is then attempted using the heuristic considered most powerful. The heuristic is applied to the subpuzzles too, if any, and will rely in part upon results obtained by the preliminary operations. In most instances only a partial solution will be obtained at this stage; the preliminary set of operations and the first heuristic are re-applied. As soon as the solution's progress is halted the remainder of the unsolved puzzle is analyzed by the heuristic next in rank. Should a further partial solution be derived, the remainder of the puzzle is re-investigated by the first heuristic. The heuristic ranking third is applied should the second heuristic yield nothing. Each time a heuristic places a puzzle piece, even tentatively, an immediate return to the most powerful heuristic is made. Should the puzzle be unsolved by the time all the heuristics have been applied, a fresh attempt is made: this time, however, the second-ranking heuristic is applied first, and so on. A return to the first heuristic is made as soon as a partial solution is available.

In summary, the reasoning behind the structure of the program is the following. Clearly the most reliable heuristics, in the sense that they are deemed to provide a correct solution more often than not, are the ones that should be used as frequently as possible. Experience with those used here has shown that when their criteria are met, the resulting placement of a puzzle piece is very reliable. The criteria of these high-ranking heuristics are rather stringent, and should the solution process terminate unsuccessfully on using them, the next most reliable one should be used if only to continue the solution process. The requirements of the next most reliable heuristics will be less stringent, and their contribution is only admitted as a bridging step to the next step, which is a return to the use of the first set of heuristics. In this way, the ensuing portion of the solution is as reliable as possible. Figure 2 is a generalized block diagram of the program's structure. Details of the individual blocks are discussed in the next sections.

A similar argument holds for the insertion of entry points E_1, E_2 in figure 2. It may well happen that a complete solution will not be reached after the

application of all the heuristics. In that case it may be advantageous to delete the contributions of each of the heuristics and start the problem afresh this time starting at E_2. The reduced requirements of the second heuristic, used first, might assist in finding the correct solution the next time around.

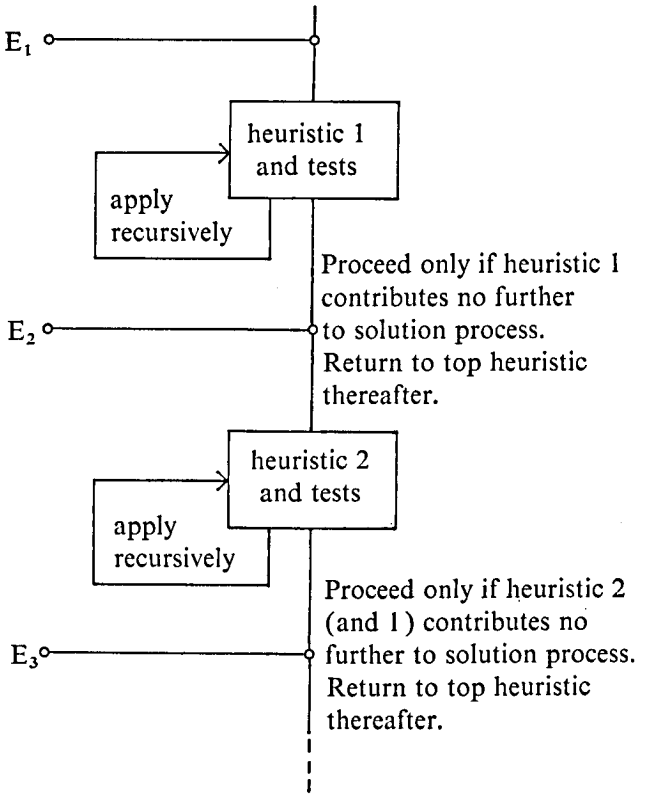


Figure 2

Finally, it is pointed out that the tests to be described are global rather than local. This approach differs from that of Guzman (1968), who, in addressing himself to three-dimensional scene analysis, used local edge properties. Given a set of lines describing the scene, the extracted local properties were used to establish which blocks, for example, partially occluded others. These kinds of local properties cannot be derived directly when one deals with tangram puzzles since only the puzzle's periphery is given. Thus global edge properties are to be used.

3. SOME GEOMETRIC CONSIDERATIONS

Before describing the details of the program, a few words concerning the types of tangram puzzles to be considered here are necessary. It was realized

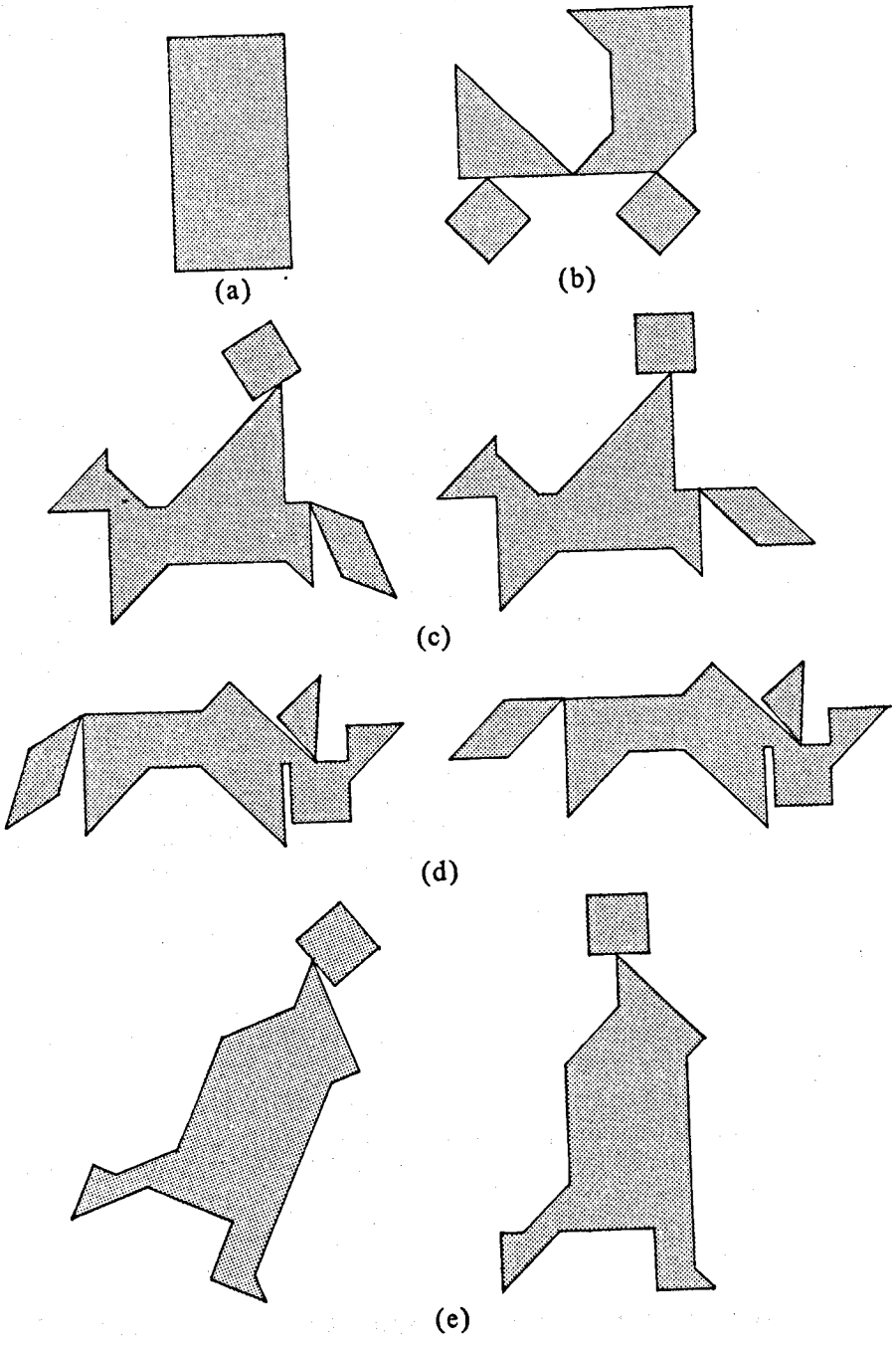


Figure 3

quite early on that the number of corners or vertices of a tangram shape contains considerable information about its possible structure and thus the location of the individual puzzle pieces within it. Consider the first two tangrams shown in figure 3. The outline of the four-cornered convex shape, figure 3(a), provides very little information concerning its structure. The tangram in figure 3(b) can, however, be solved more easily since the numerous corners indicate how some of the pieces might be placed. As will be shown later much information is also contained in the concave portions of the outline since it suggests the existence of contact between two puzzle pieces. Rather than also attempt the solution of the 'tougher' puzzles, containing fewer corners, it was decided to opt for the solution to puzzles containing eleven corners or more, in the hope that a solution for the 'easier' puzzles would at least be obtainable.

A further restriction on the type of tangrams to be considered is the requirement that all the puzzles be filled in. This requirement excludes all puzzles containing loops and holes. The complexity involved in first having to locate the internal puzzle-edges circumscribing the holes, followed by the necessity of having to handle two sets of edges at a time was considered worth avoiding. Luckily the number of such puzzles is very limited.

Of course it is possible to consider puzzles made of fourteen pieces. Such puzzles, employing two sets of seven puzzle pieces, are called double tangrams. While the single-tangram techniques, to be described below, may be equally applicable to double tangrams we will not consider them here.

Some puzzles may contain edges which are not inclined at an angle that is a multiple of 45°. In some cases the entire tangram is given in an inclined form. This presents some problems, due to sampling, when a rectangular array is used. Where the portion of a tangram could be rotated without changing the puzzle's overall structure this was done. Otherwise, the tangram was discarded. Such tangrams would be those in which reorientation either caused contact between non-touching edges or caused the formation of a loop. With the tangram in figure 3(c) reorientation was possible; with the tangram of figure 3(d) it was not possible without materially changing the nature of the puzzle. Inclined tangrams were similarly rotated; see figure 3(e) for an example. Generally, then, a tangram puzzle was not considered well formed unless all of its edges were inclined at an angle whose value is a multiple of 45°.

Figure 1(a) also gives the dimensions used for the individual tangram pieces in terms of inter-picture point distances. In many problems associated with digitized line drawings, the fact that distances between axial picture points and diagonal picture points differ by a factor of $\sqrt{2}$ can be safely ignored. This is not the case here, because true lengths are important. The numbers shown along the edges in figure 1(a) give the length of that edge (in picture point distances) at the current orientation of that edge. The number

in brackets signifies the length of that edge when its inclination is changed from axial to diagonal or vice versa.

One of the problems associated with finding a solution to a puzzle is the fact that the eye may be deceived as to the actual length of an edge comprising the puzzle shape, and short of using a ruler, the length of the edge must be guessed. The guessed length could have been used as input to the program, but this might present a host of side problems that would only stifle the heuristics. Accordingly, lengths were read off the puzzle shapes directly.

4. TANGRAM ANALYSIS

4.1 Subpuzzle determination and extension line construction

The puzzle is input to the machine in terms of its vertices' numbers (corners) and edge lengths. The practice was to start with the leftmost, topmost vertex on the puzzle and proceed in a clockwise direction. The puzzle shape was traced out on an (x, y) array.

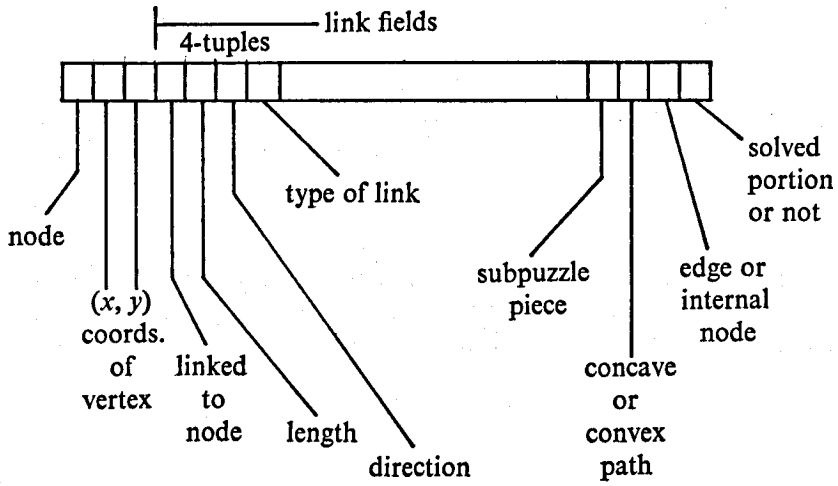


Figure 4

Associated with each vertex is a one-dimensional array the contents of which will be explained as the analysis procedure is developed. The first few items in the array are the vertex's number, and its (x, y) coordinates in the array (see figure 4). Each vertex is connected to two other vertices through two edges of known length and orientation. Each vertex connected to the vertex in question is represented by a 4-tuple within the vertex array. The numbers of the connected vertices, the lengths and direction of the connecting edges are entered into the 4-tuples. The fourth position within each tuple indicates some arc information and is explained later.

The program first attempts to separate the puzzle into two or more subpuzzles. This implies that the separation procedure must ensure that the

subpuzzles do not share a common piece at the position of separation. Where portions of the puzzle meet at a point, the subdivision is clear. The way such severance points are located is by tracing along the edges of the tangram on the (x, y) plane. A return to an already traced point, other than to the starting point, implies a point contact. The tangram portion defined by the edges traced during the first and second encounter constitutes a subpuzzle. The remaining portion of the puzzle is another subpuzzle and the procedure is continued until a return to the starting point is made. The shape shown in figure 3(c) comprises three subpuzzles. Subpuzzles are numbered and the number associated with each vertex is entered in the vertex array. Subpuzzles can also be separated along an internal edge. This edge is formed by extending into the subpuzzle area an edge whose end is very close to a neighboring vertex. Figure 5 shows some examples of subpuzzles formed in this way.

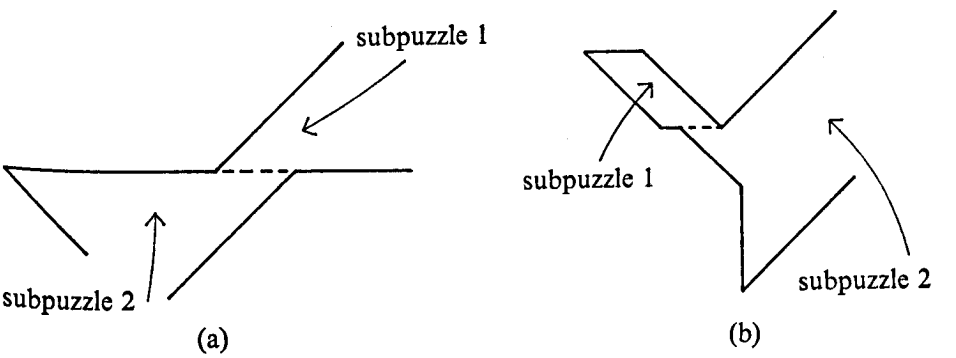


Figure 5

Note that in the case of figure 5(a), the right hand vertex has no meaning relative to the subpuzzle numbered 1. It is erased from the vertex array. Another item on the vertex array is therefore the subpuzzle piece number to which each vertex belongs.

Puzzle vertices fall into two categories: those connecting edges which form a convex edge path and those connecting edges which form a concave path. Determination of the type of each vertex is made by measuring the angular change in the edge at each vertex. The distinction between these two types is important since the contribution of each type to the solution is different and they are thus treated differently. The corner formed by a convex-type vertex strongly indicates a possible puzzle piece fit, whereas a concave type vertex is identified with the location of a line (inside the puzzle area) along which two or more puzzle pieces meet. Consider figure 6(a): it will be observed that the protruding corners contain considerable piece-fit information. No such information is available at the recessed vertices. However, if the edges at the latter points are extended back into the puzzle area a set of guide lines revealing the possible location of further pieces as well as piece-adjacency

information is obtained. The lines so obtained will be referred to as extension lines (see figure 6(b)). Subsequently, when attempts at fitting puzzle pieces are made, only convex paths need be considered, seeing that the concave paths assist in the construction stage only.

Figure 6(b) shows all the possible extension lines. As these are created new

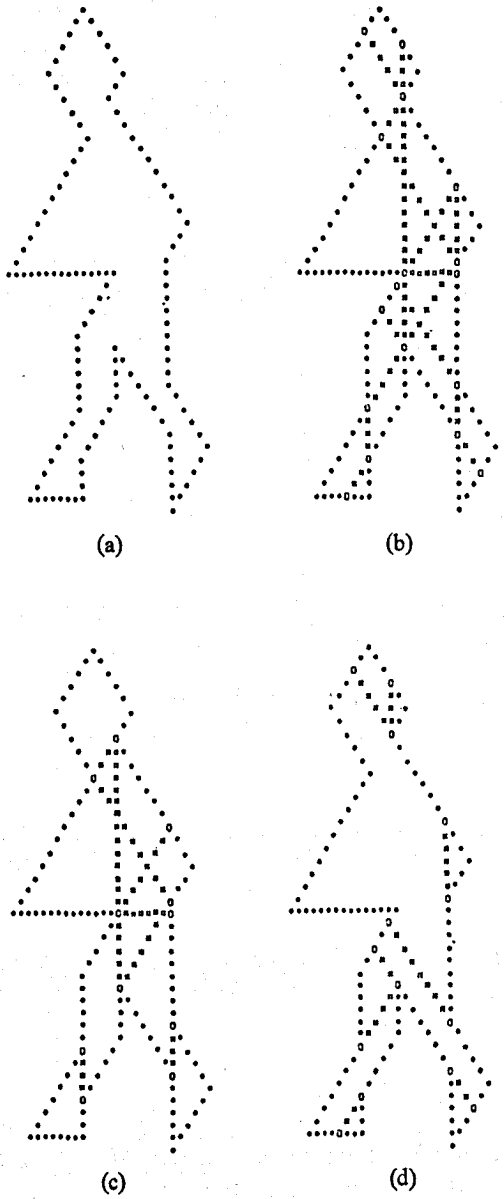


Figure 6

vertices (or nodes) are formed both on the edges and within the puzzle area. (The words 'nodes' and 'vertices' will be used interchangeably henceforth.) Not all of the extension lines prove to be useful; some of them can in fact be misleading guide lines and have to be removed. Extension lines whose length is too short to support a puzzle piece are removed. Using the length of puzzle pieces as in figure 1, it was decided that extension lines in the axial direction whose length was less than seven were to be removed. Extension lines in the diagonal direction whose length was less than five were also to be removed. Extension lines terminating at an edge vertex were retained at all times.

Extension lines were also tested for parallelism in that an extension line of admissible length was removed if it was too close to a parallel edge of the puzzle. The distance criteria were the same as above for axial and diagonal extension lines. The reason for the removal of these lines is that no puzzle piece could possibly lie entirely within the region bounded by these two parallel lines. However, extension lines running parallel to one another were retained even if they were too close, since either one of them could form the boundary of a puzzle piece. The extension lines retained are shown in figure 6(c); those eliminated are shown in figure 6(d).

During the extension-line generation procedure, two new types of nodes (or vertices) are generated. One type is formed inside the puzzle area at the intersection of two or more extension lines, and the other, at the edges, at the intersections of an extension line with an edge. These new nodes are added to the vertex array and an entry as to the type is also made (see figure 4). As the new nodes are generated the information related to individual vertices or nodes in the array must be updated in order to indicate the new interconnections. For example, the formation of a node on an edge between two vertices must be reflected in the array showing that they are no longer connected directly. Furthermore, new extension line interconnections must be entered or updated. The updating is made by altering or adding to the contents of the 4-tuples with each vertex array. The fourth entry in the tuplet indicates whether the connection is via an edge or an extension line. Eventually some internal nodes will become edge nodes as solved puzzle pieces are removed from the puzzle. The edge along which a piece was severed now becomes an edge.

The preliminary tests over – subpuzzle separation, the generation of extension lines and the updating of the vertex list – the solution procedure can begin. We start by describing the various rules that are used; their order of application is discussed later.

4.2 Puzzle pieces: direct-match rule

The program first attempts to locate puzzle pieces *fully described by edges*, rather than by extension lines or by combinations of edges and extension lines. The seven puzzle pieces are stored in the machine as are their reflections

and rotations (in steps of 45°) if different from the corresponding puzzle piece initially stored. The (x, y) plane on which the tangram was originally traced is examined in conjunction with the vertex arrays. The distance and orientation of neighboring vertices are examined at each vertex, in order to determine whether the vertex pattern at the vertex in question matches that of a stored puzzle piece. Once a match is found the corresponding portion is severed from the puzzle and an indication to the effect that the matched puzzle piece is no longer available is made.

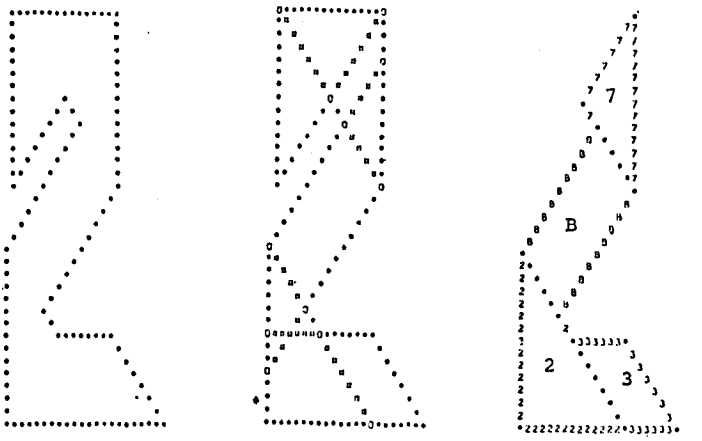
The extraction of a puzzle piece by direct matching occurs in the neighborhood where two subpuzzles meet; the matched piece usually forms one of the subpuzzles (see figures 3(c,d,e)).

4.3 Puzzle pieces: 2½-3½ edge-match rule

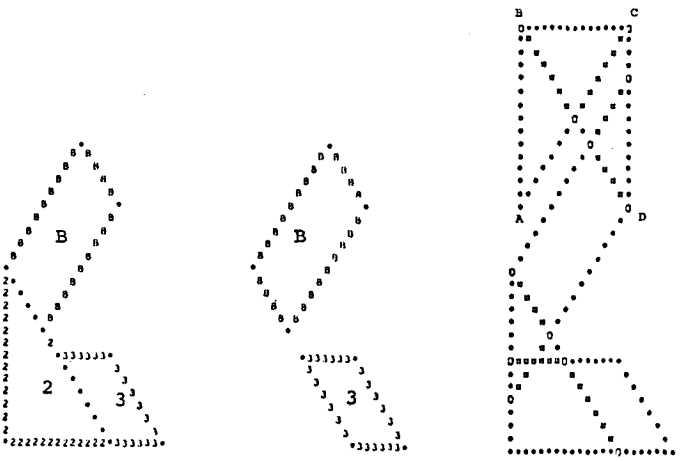
The method of determining the position of puzzle pieces by the rule just described, while being very reliable, cannot be used so frequently in practice for it is not often that a puzzle piece fully bounded by edges is found. Thus some relaxation in the rigor of fit has to be made. Another matching heuristic allows a puzzle piece to be located if most of its periphery is described by edges and only a small portion is described by the constructed extension lines. Stated precisely, the 2½–3½ edge-match rule requires that in the case of triangular shapes *two complete sides must be defined by edges and the remaining side can be defined by a combination of collinear edges and extension lines. Moreover, the combination must include at least one portion of edge. For four sided puzzle pieces the rule requires that the additonal side also be fully described by an edge.*

This heuristic was found to be extremely powerful in locating puzzle pieces. It should be noted that the rule only requires that the third or fourth side of the puzzle piece be defined by a combination of edges and extension lines, that is, their order of appearance as well as their number is unimportant. The tangram of figure 7(a) shows how a partial solution is obtained for a puzzle by the application of this rule. The three largest triangular pieces are extracted in this way; the smaller remaining rhomboid is matched subsequently with a puzzle piece (the significance of the portion labelled B will be explained later). No priority as to the fit containing the greater amount of peripheral edge in the remaining side is given. An argument for having such a priority is illustrated using figure 7(b). Observe that there are two ways of placing the large triangular piece in the upper portion of the puzzle. It could occupy position ABC or BCD, but not both. A priority based on the amount of edge in the remaining side would have the large triangle placed at top left rather than in the top right portion. In this case, however, a complete solution is obtained eventually either way, but in general, a wrong choice may result in failure and a new fit will be sought.

Despite the seemingly loose definition of puzzle pieces by this rule, wrong extraction of pieces occurs rarely. In fact, some puzzles may be completely



(a)



(a)—continued (b)

Figure 7

solved by applying the direct-match rule and a repetitive application of the rule just described. Such a puzzle is shown together with its full solution in figure 8. Note that in figure 8(e) the numbers refer to the order in which puzzle pieces are identified. This is true for all subsequent figures.

Two updating procedures now occur within the vertex array and the (x, y) plane. First, the line shared by the removed piece and the remainder of the puzzle is treated as an edge henceforth, since it now lies on the periphery. Secondly, any vertex along this line considered hitherto as internal now becomes an edge vertex. This updating takes place each time a portion of the puzzle is removed.

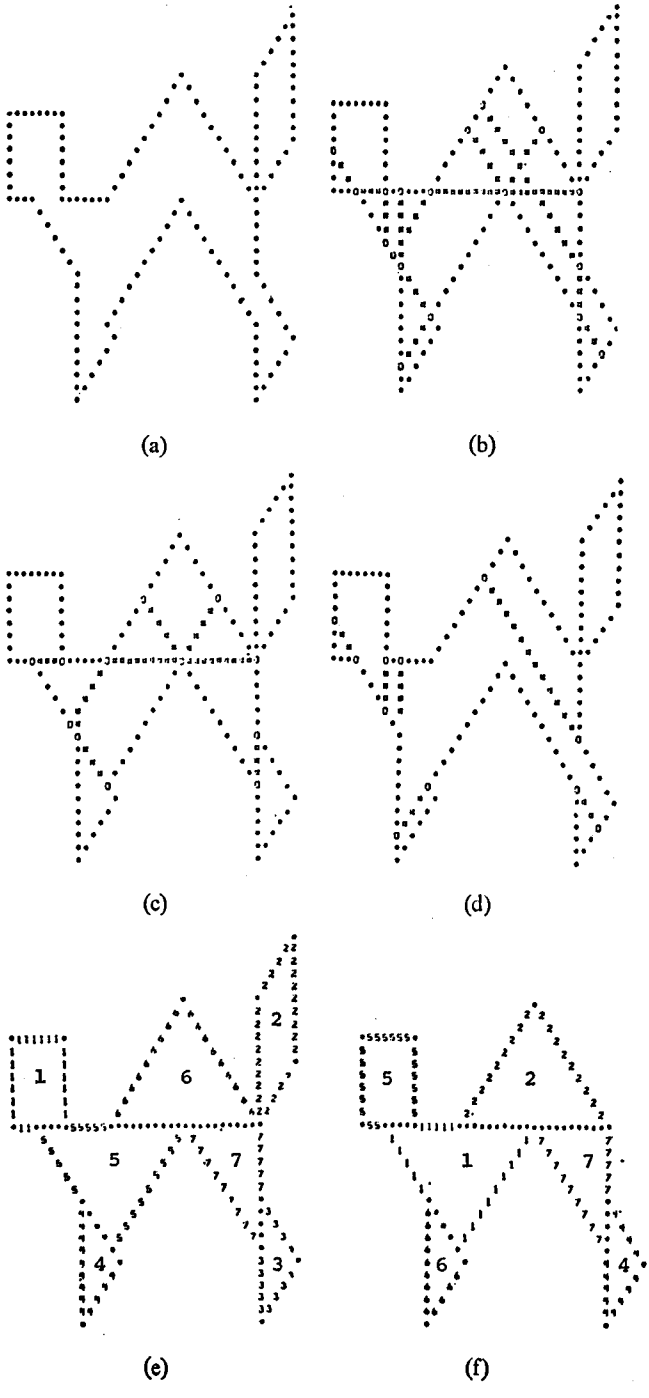


Figure 8

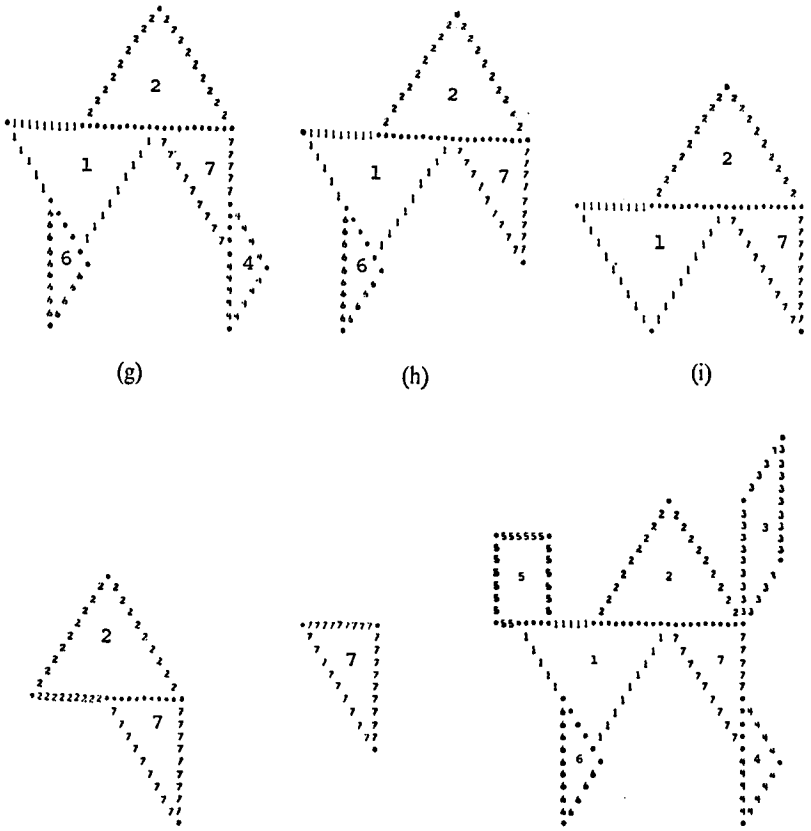


Figure 8 contd.

4.4 Composites: direct-match rule

It is clear that the rigor of the matching requirements in terms of edges and extension lines could be relaxed further. As a matter of fact a matching rule requiring there to be only three out of the four edges of a four sided piece, etc, be they edges or extension lines, could be used. Their position in the ranking of useful heuristics would be low; nevertheless, in the absence of any further progress towards a solution such rules may have to be resorted to. Before these type of rules are discussed a more powerful heuristic is presented. This involves the extraction of a specific group of puzzle pieces all at once rather than the extraction of single puzzle pieces one at a time. Such groups of puzzle pieces are called composites.

Consider the two puzzles shown in figure 9. The extension lines are represented by dots. In particular consider what happens after the application of the rule described in the previous section. In the case of figure 9(a), ABCDEFGA denotes the unsolved part of the puzzle. In figure 9(b) the

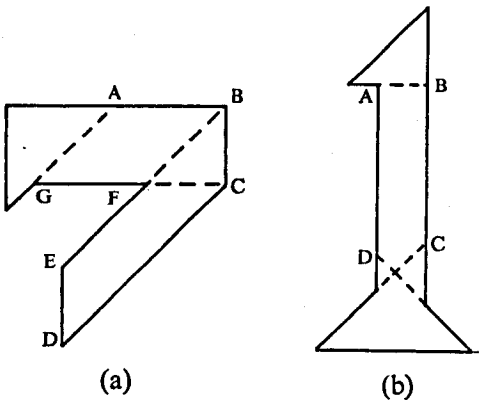


Figure 9

unsolved portion of the puzzle is ABCDA. In both cases there is no further indication as to how the puzzle pieces are distributed. Clearly the as-yet-undiscovered pieces combine in some way to fit the unsolved portion exactly. It appears generally, and it is a contention of this approach, that the unsolved tangram can be partitioned into composite shapes which assume a relatively simple outline. In principle, one form a composite shape could take is the shape outlined by ABCDEFGA. If one pursues this idea a little further one could conceive a variety of types of composite forms of varying degrees of complexity. The formation of these could be a lengthy task and would hinder the generality of the approach. Instead, simpler composites can be used such as ABCFGA and CDEFC, both of which combine to form the remainder of the puzzle. The portion labelled B in figure 7(a) is an example of a composite. In the case of figure 9(b) the puzzle contains the simple composite ABCDA. In obtaining the composites, the problem is not only the generation of simple composite shapes but also how few composites will be sufficient.

A first restriction on the formation of composites is that they must be either three or four sided. The reason for this is that composite shapes will be treated as if they were puzzle pieces and the same edge rules will be applied to them. Also, this restriction limits the complexity of the composites. It will be observed that a given composite can be formed by combining different pieces in more than one way. Consider the four-sided composite shown in figure 10(a). It can be formed in three different ways using a different set of puzzle pieces in each case (see figure 10(b) through (d)). (There are actually six different ways of constructing the composite since there are two small triangular shapes. However, the two pieces are interchangeable.) Note that in figure 10(e) the puzzle pieces of figure 10(d) have just been organized in a different way and therefore do not constitute a distinct composite structure.

Some of the composites can assume the shape of the actual puzzle pieces. The large triangular piece, for example, can be formed using other puzzle

pieces. Puzzle pieces are also considered as composites, however, in the solution process they are treated as individual puzzle pieces as often as possible. An identified puzzle piece will only be treated as a composite once the individual piece is no longer available.

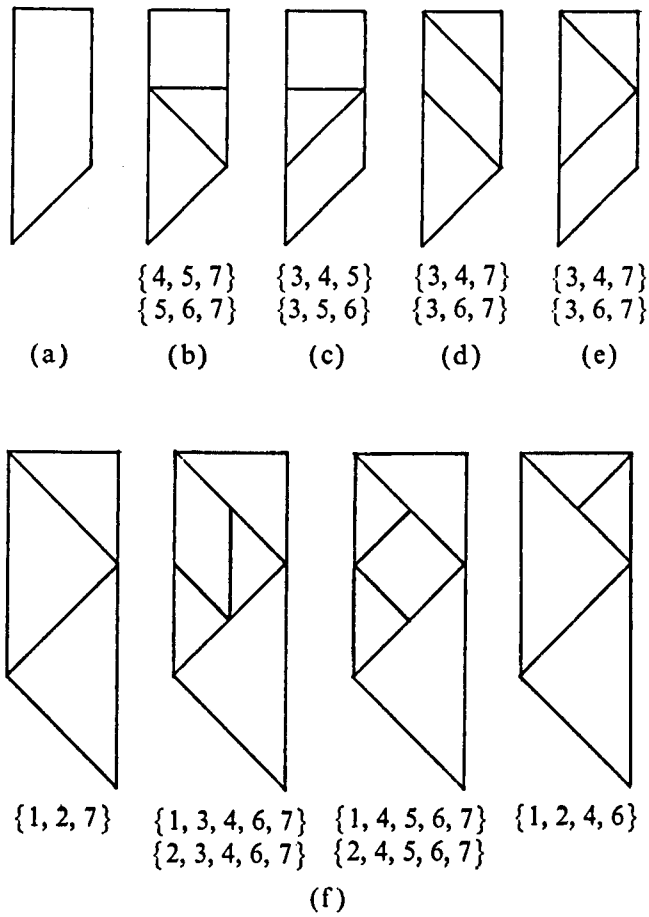


Figure 10

The composite shown in figure 10, as is the case with many other composites, also exists in a similar but enlarged form. Figure 10(f) shows how the larger composite is formed. Once a composite is formed and inserted in the list of composites, its corresponding smaller or larger composite is also added to the list. Composites are stored in the computer in exactly the same way as are the seven puzzle pieces.

Associated with each composite is the list of sets of puzzle pieces which form the composite. As soon as a composite is identified an abeyance list of

these sets is formed. Any set including a puzzle piece no longer available is excluded from the abeyance list. An abeyance list containing no sets whatsoever is an indication that the selected composite cannot be considered, and the solution process continues without this composite having been extracted. As the solution proceeds and more puzzle pieces are placed the abeyance lists corresponding to the identified composites are curtailed as these pieces become unavailable. A complete solution using the abeyance list is obtained at the end after all the remaining unplaced puzzle pieces are distributed so that each composite abeyance list contains one set which is satisfied. The character associated with each composite indicates the composite's identity in the printout. When any of the twenty-one composites are involved in a solution, they are identified by the appropriate character and the appropriate set of puzzle pieces from the abeyance list. Note that a composite in the shape of a puzzle piece will appear with its appropriate composite character.

The fact that any given puzzle can contain only three composites at most is very useful since this can be used as a check to see if an extracted composite is admissible. Furthermore, the fact that there is a maximum of only three composite pieces per puzzle speeds up the processing of the abeyance lists.

The composite piece of figure 9(b) is fully described by edges once the two smaller triangles are identified by the $2\frac{1}{2}$–$3\frac{1}{2}$ rule. The first rule associated with composites, the direct-match rule, *permits the extraction of those composites which are fully defined by edges.* The two composites of figure 9(a) will not be extracted by the rule just stated since each composite is partially defined by an extension line. Composites such as these will be recognized by a later rule.

4.5 Composites: $2\frac{1}{2}$-$3\frac{1}{2}$ edge-match rule

Just as with the individual puzzle pieces, the $2\frac{1}{2}$–$3\frac{1}{2}$ *rule is applied to composites.* It may be argued that there is no need to have separate direct-matching rules and $2\frac{1}{2}$–$3\frac{1}{2}$ edge rules for both the puzzle pieces and the composites, since the program tests them in an identical manner. Eventually, when the heuristics are organized, this may well turn out to be the case. For the moment, however, rules remain separate for each group of shapes. This only means that a particular rule will operate either on the set of puzzle pieces or on the twenty-one composite pieces but the same portion of the program is used in each case.

4.6 Composites: 2-3 edge-match rule

This rule requires still less stringent matching criteria for the extraction of composites. All a potential composite piece must satisfy is that *two of its sides be defined by edges and the remaining side be defined by an extension line. In the case of a four sided composite, three sides must be defined by edges* (see figure 9(a)).

There is one additional requirement, however, which also applies to the rule described in the previous section. Ordinarily, when the composite is

fully described by edges, there is no doubt as to the identity of the composite to be extracted. However, when this rule is relaxed, more than one composite may be extracted at any given time. Consider figure 11: two composites are embedded in the left vertical limb of the puzzle, composite E and composite F.

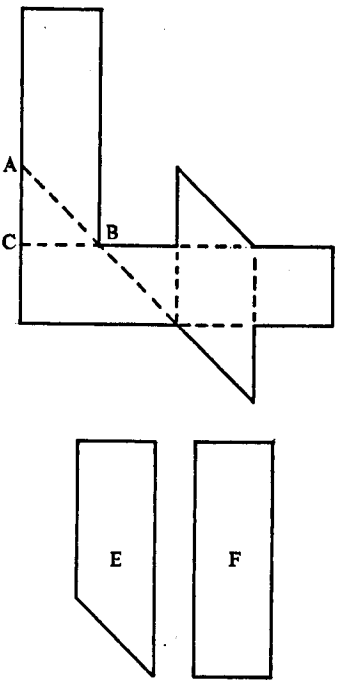


Figure 11

Composite E would have extension line AB as its fourth side, whereas composite F would have its fourth side formed by extension line BC. Both this and the above rule require that the *smallest composite piece be extracted.* In this case the rule would extract composite E.

4.7 Puzzle pieces 2-3 edge-match rule

We now return to further rules for extraction of individual puzzle pieces. The rule under this heading is *the same as that just developed for composite pieces.* This rule never fails to extract the four-sided puzzle pieces correctly, but it can introduce errors when it tries to isolate the smaller triangular pieces. The rule is designed to isolate the small triangular pieces only if no other piece can be isolated using this rule. Figure 12 shows a complete tangram solution obtained using the $2\frac{1}{2}$–$3\frac{1}{2}$ edge rule and the 2–3 edge rule. (The example introduces the need for a good organization of the rules.) The larger triangular piece in figure 12(i) is identified by the 2–3 edge rule.

Figure 13 shows the solution to a tangram puzzle using the $2\frac{1}{2}$–$3\frac{1}{2}$ edge-match rule and the 2–3 edge-match rule. It illustrates the need for the restriction concerning small triangles in the latter rule. Observe that in figure 13(c) the small triangles, remaining after the $2\frac{1}{2}$–$3\frac{1}{2}$ has been applied, could

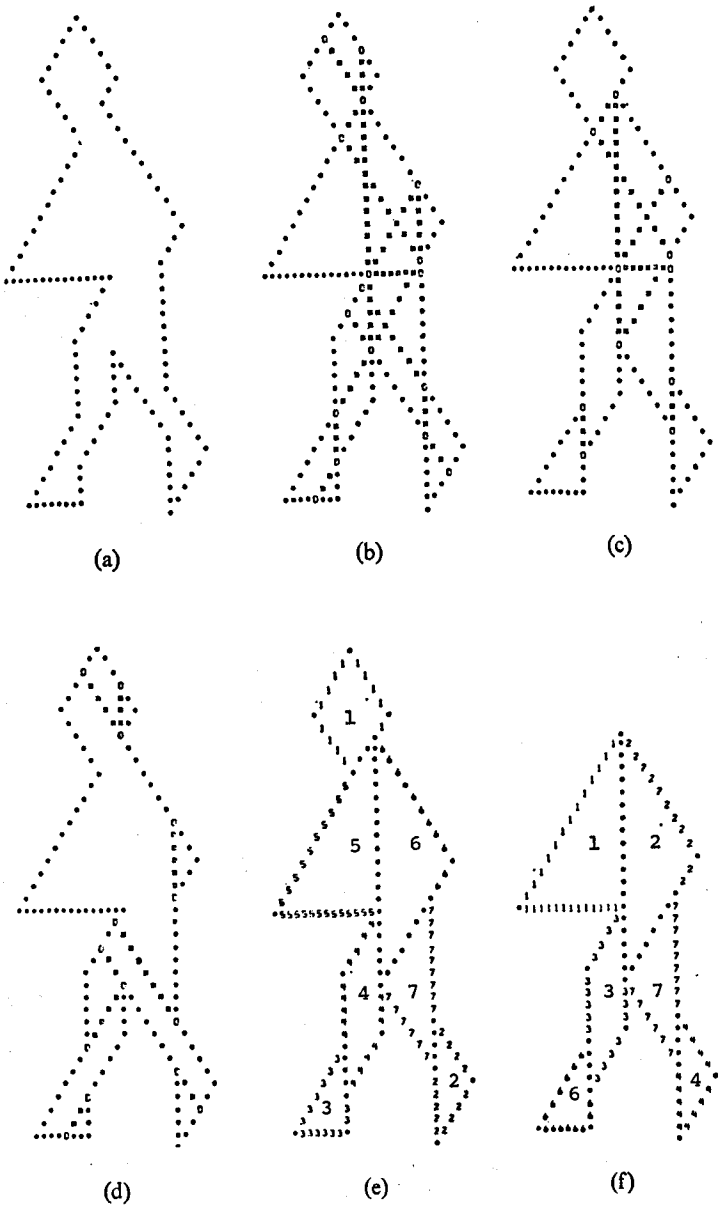


Figure 12

be fitted into areas ABC or DEF. The former choice would lead to an error. Now that the small triangles are matched last this will not occur.

4.8 Puzzle pieces: $1\frac{2}{2}$-$2\frac{2}{2}$ edge-match rule

This rule is one of the most lax. All that will be required for a three-sided puzzle piece to be recognized is that only *one of its sides consist of an edge; the remaining sides can be defined by a combination of edges and extension lines.* The application of the rule to four-sided puzzle pieces is clear. The first puzzle piece in figure 14 is identified using this rule (see figure 14(c)).

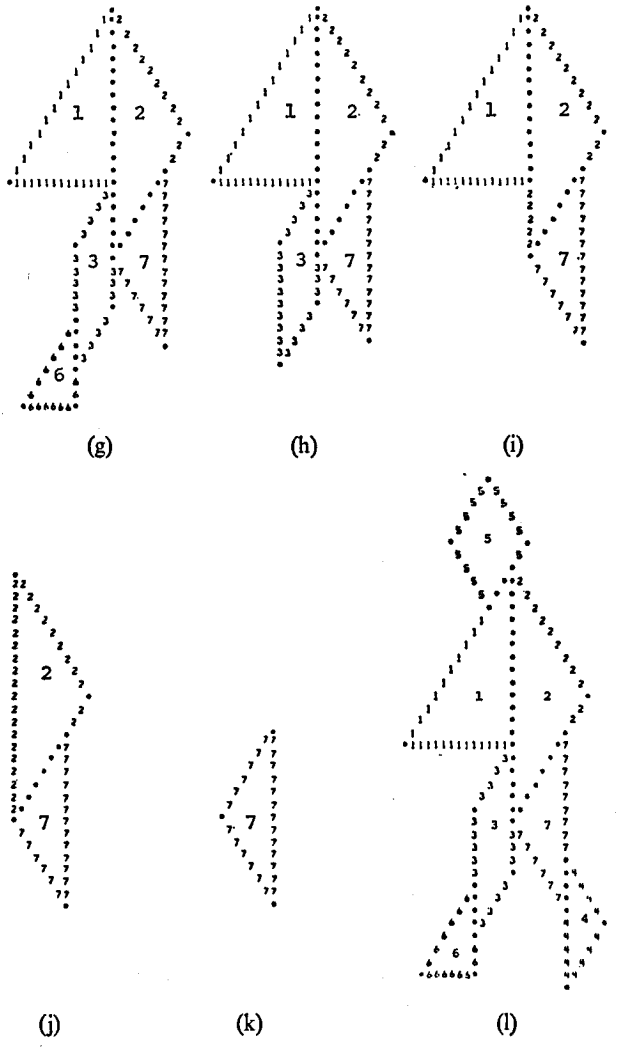


Figure 12 contd.

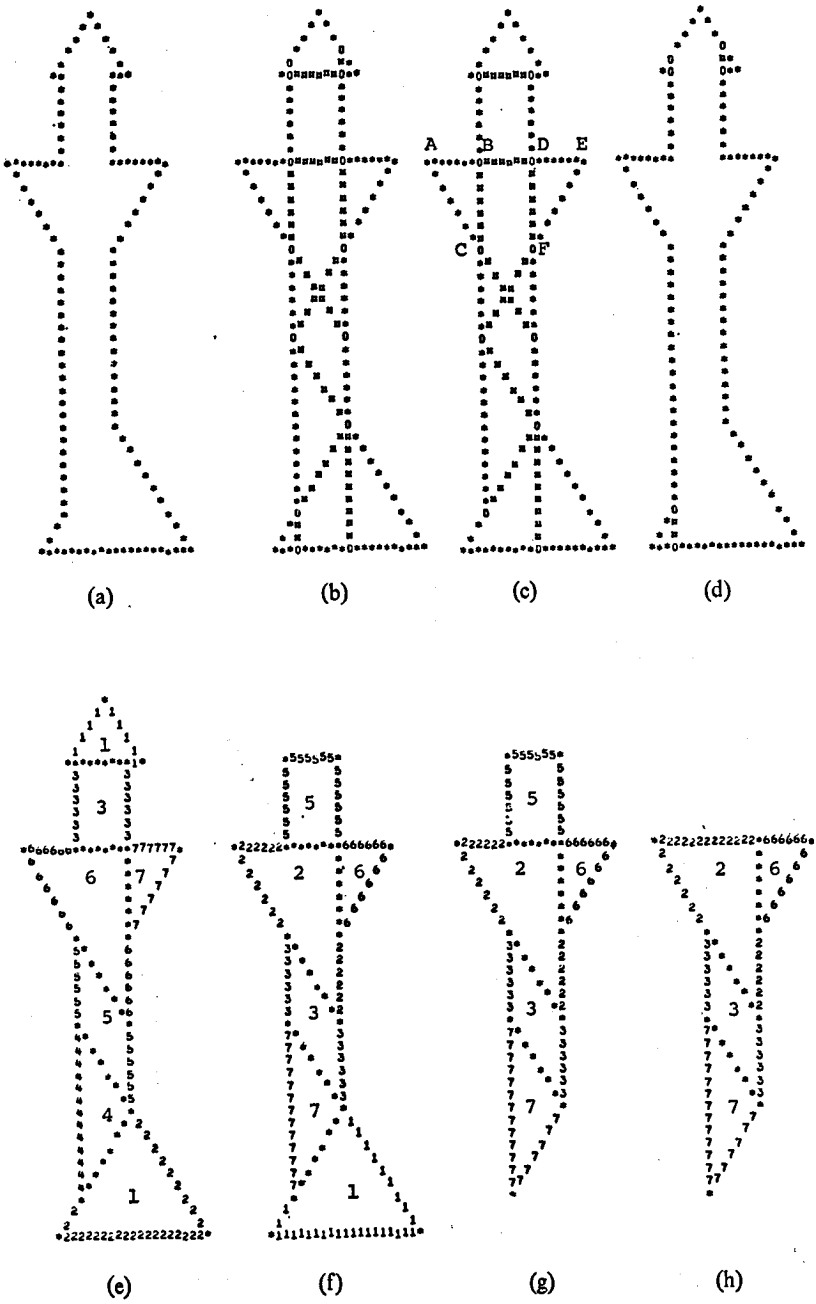
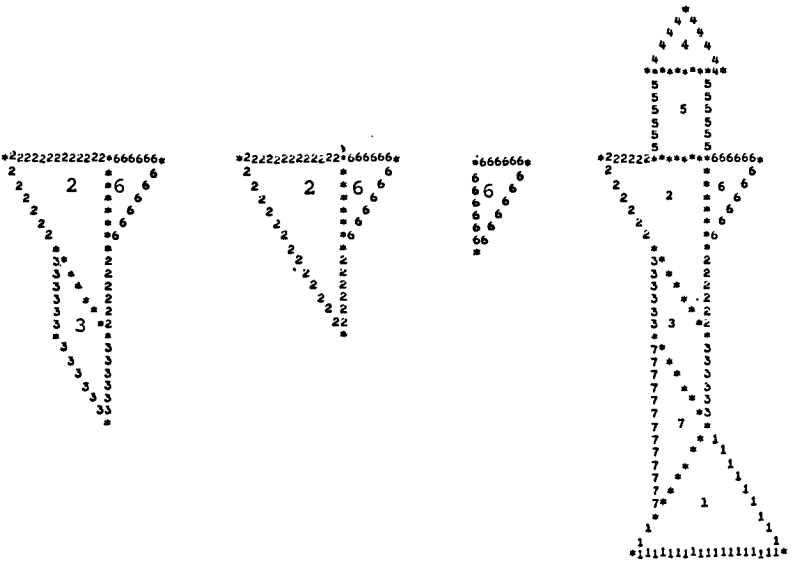
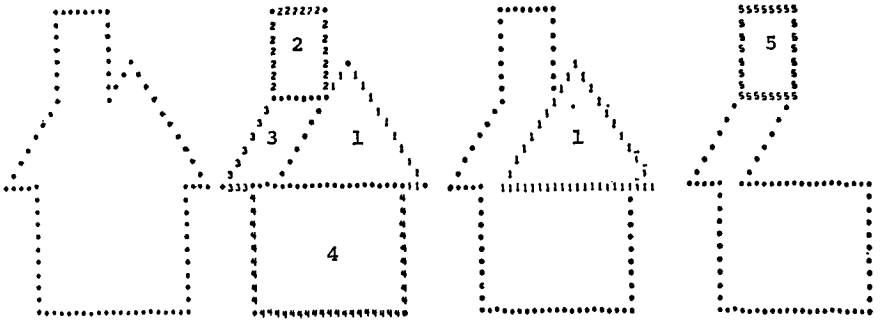


Figure 13

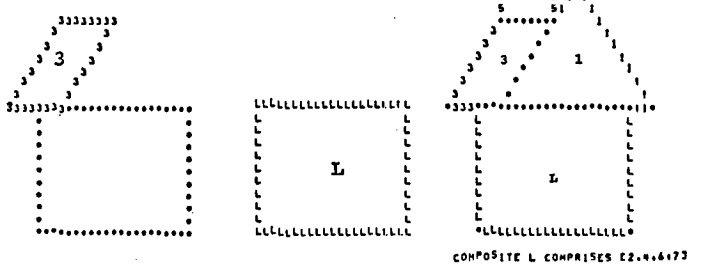


(i) (j) (k) (l)

Figure 13 contd.



(a) (b) (c) (d)



(e) (f) (g)

Figure 14

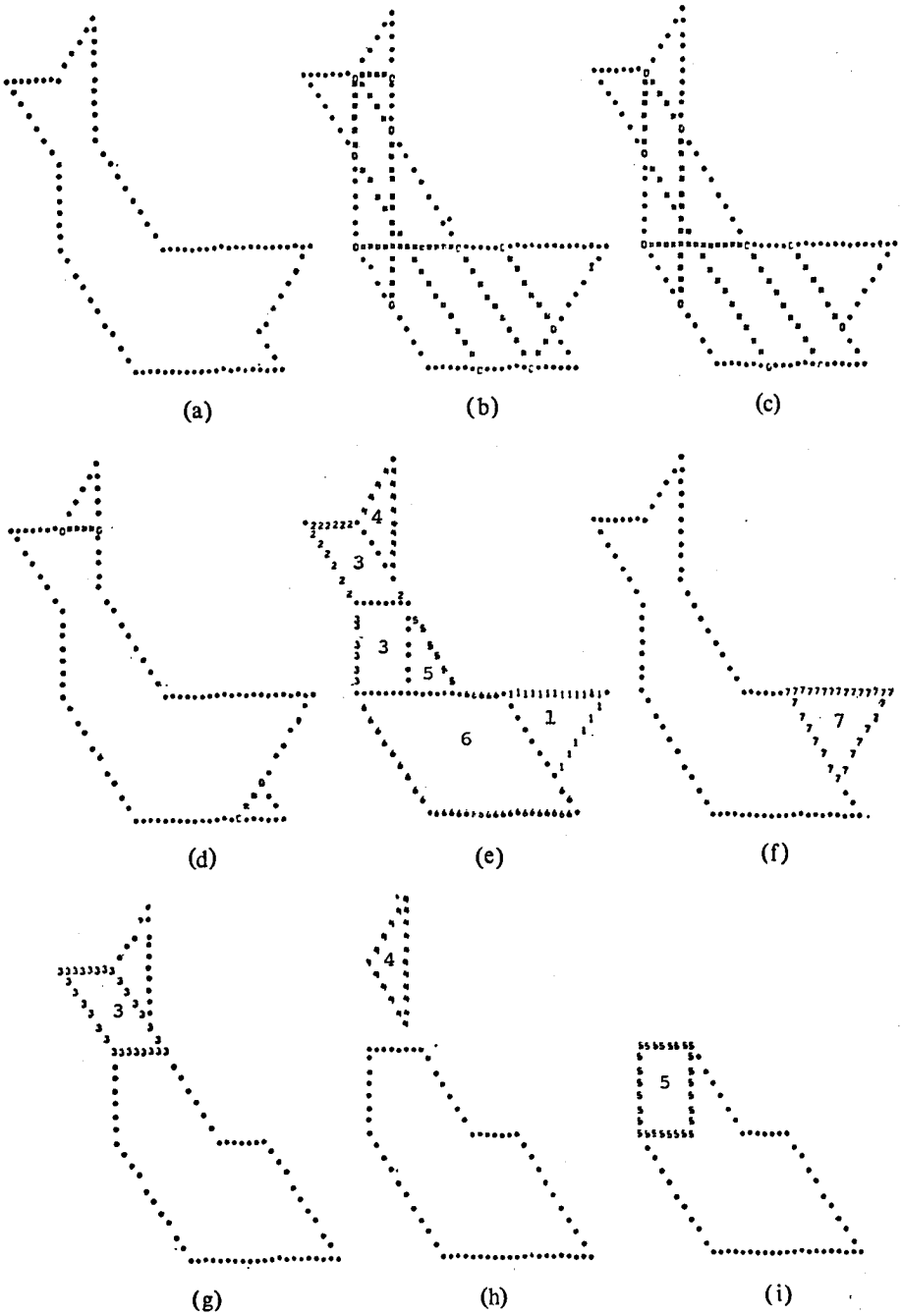


Figure 15

4.9 Puzzle pieces: 3 edge only match rule

This and the next rule apply to the extraction of the two *four-sided* puzzle pieces only. The rule requires these puzzle pieces to be *represented by three sides, each being defined by an edge.* The latter part of the solution to the tangram of figure 14 employs this rule (see figure 14(e)). The format of figure 14 has changed somewhat now to conform to the actual way in which pieces are extracted. The edge numbers in figure 14(b) still refer to the order in which corresponding pieces are removed.

4.10 Puzzle pieces: 2 edge and 1 extension line match rule

This rule, too, requires the *four-sided* puzzle pieces to be defined by *three of their sides*; here, however, *one* side can consist entirely of an extension line. Figure 15 shows a solution to a puzzle employing this rule.

4.11 Puzzle pieces: 2 edge only match rule

This rule applies to the extraction of the *two large* triangles only. It requires only *two sides to be defined by edges. The third side need not exist.* This rule is a 'last resort' rule since it is evoked, usually, after no other rule provided a solution. Figure 16 shows an example where this rule is used to initiate a solution process. Observe that no extension lines are deleted here. Figure 16(c) indicates that the bottom portion of the left limb is solved second. It is indeed extracted then, but its identity is only determined at the end. This should not be confused with figure 16(e) which indicates that the large triangle (also of piece number 2) is identified second.

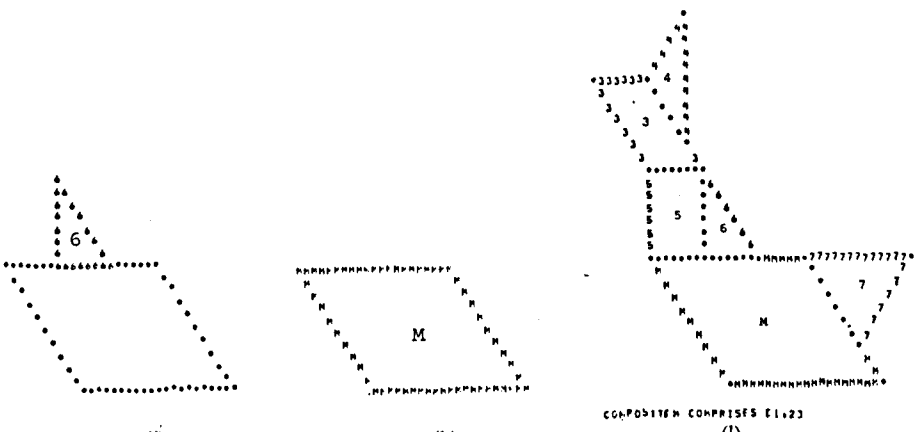


Figure 15 contd.

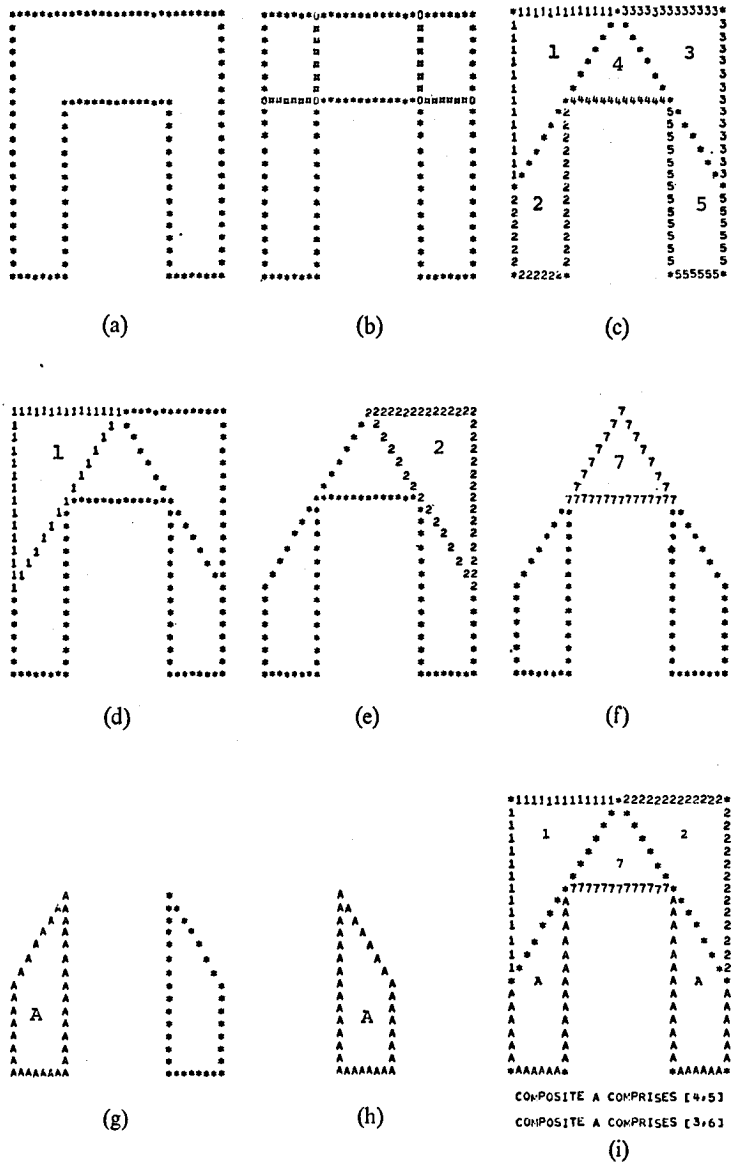


Figure 16

5. ORGANIZATION OF MATCH RULES

The tangram solutions presented in the above sections were derived by applying the puzzle piece extraction rules according to the scheme shown in figure 17. This figure should be examined together with figure 2. Some preliminary ranking is necessary in order to determine the efficacy of the

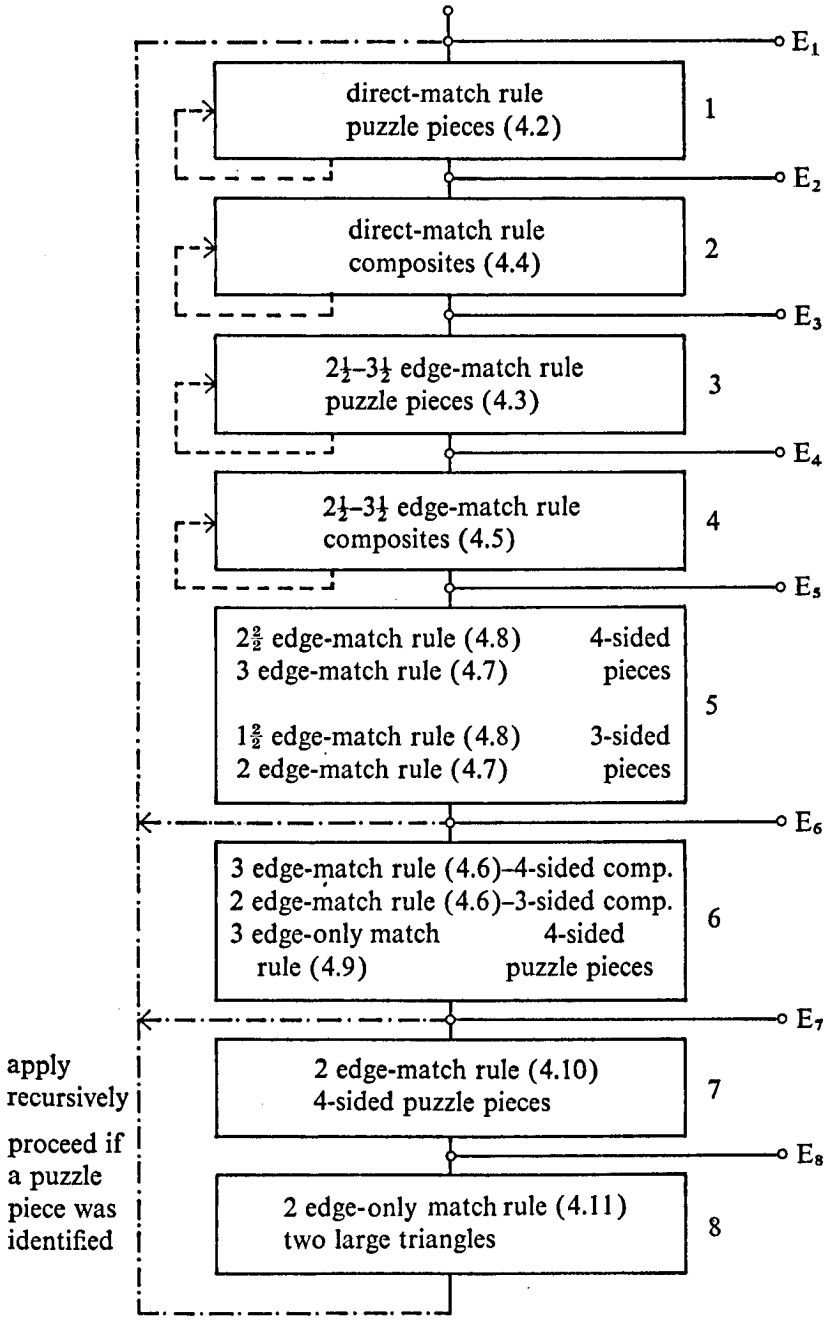

E1
direct-match rule
puzzle pieces (4.2)
1
E2
direct-match rule
composites (4.4)
2
E3
2½–3½ edge-match rule
puzzle pieces (4.3)
3
E4
2½–3½ edge-match rule
composites (4.5)
4
E5
2⅔ edge-match rule (4.8)
3 edge-match rule (4.7)
4-sided
pieces
1⅔ edge-match rule (4.8)
2 edge-match rule (4.7)
3-sided
pieces
5
E6
3 edge-match rule (4.6)–4-sided comp.
2 edge-match rule (4.6)–3-sided comp.
3 edge-only match
rule (4.9)
4-sided
puzzle pieces
6
E7
2 edge-match rule (4.10)
4-sided puzzle pieces
7
E8
2 edge-only match rule (4.11)
two large triangles
8
apply
recursively
proceed if
a puzzle
piece was
identified

Figure 17

proposed extraction rules. By the scheme of figure 17, the first four rules are each applied recursively, that is, the first rule is applied until no further pieces (if any) are extractable, and is then followed by the second rule also applied recursively, and so on up to the fifth rule. The fifth rule is applied once and should a puzzle piece have been recognized an immediate return to the first rule is made. Failing this, the sixth extraction rule is applied. Entry points are declared at the beginning of the program. The reason underlying this preliminary organization is that the first four extraction rules are considered 'cast-iron' rules and should therefore be applied as often as possible.

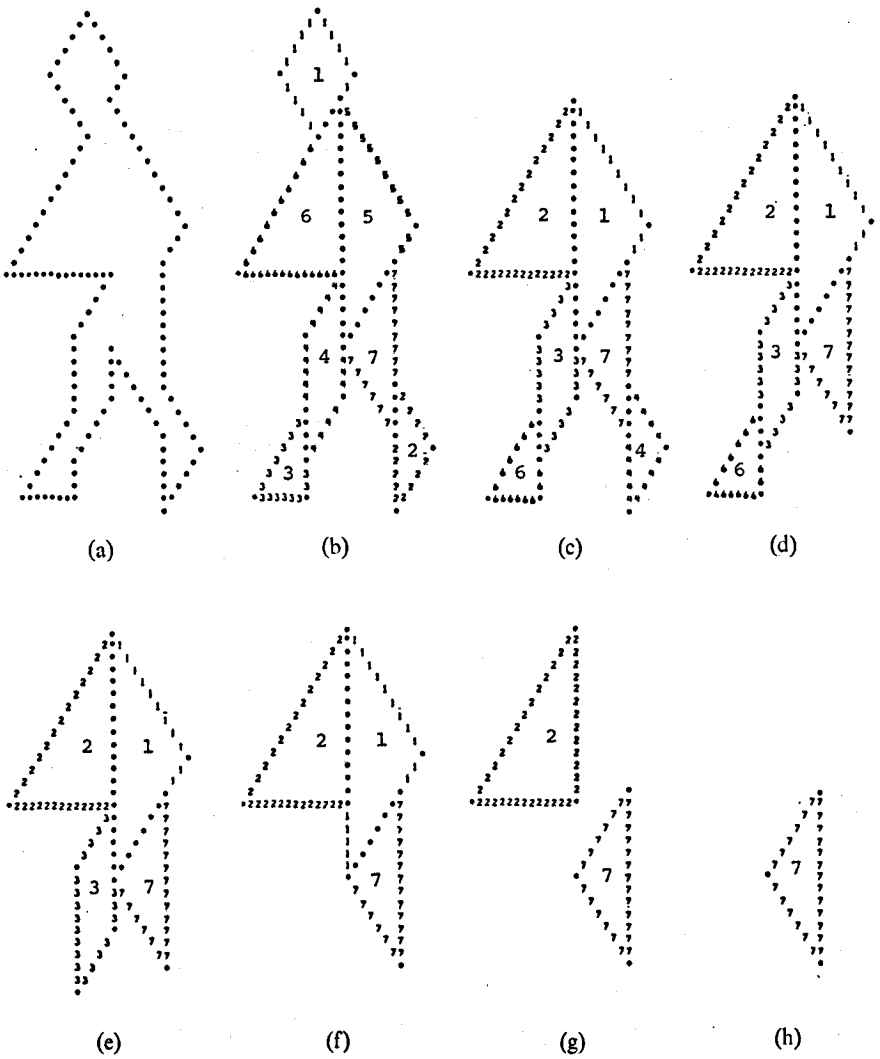


Figure 18

The rules of lower rank, however, are increasingly less reliable and should therefore be applied only when the solution process can proceed no further. Once the solution process is continued more reliable extraction rules should be resorted to again. The question is what is the most efficient way to rank the rules.

Most of the reorganization of the rules in figure 17 will take place within the lower portion of the diagram, since the first few rules seem reliable. It is interesting to see what happens to a tangram solution when the order of rules is changed. Figure 18 shows the solution of the tangram of figure 12.

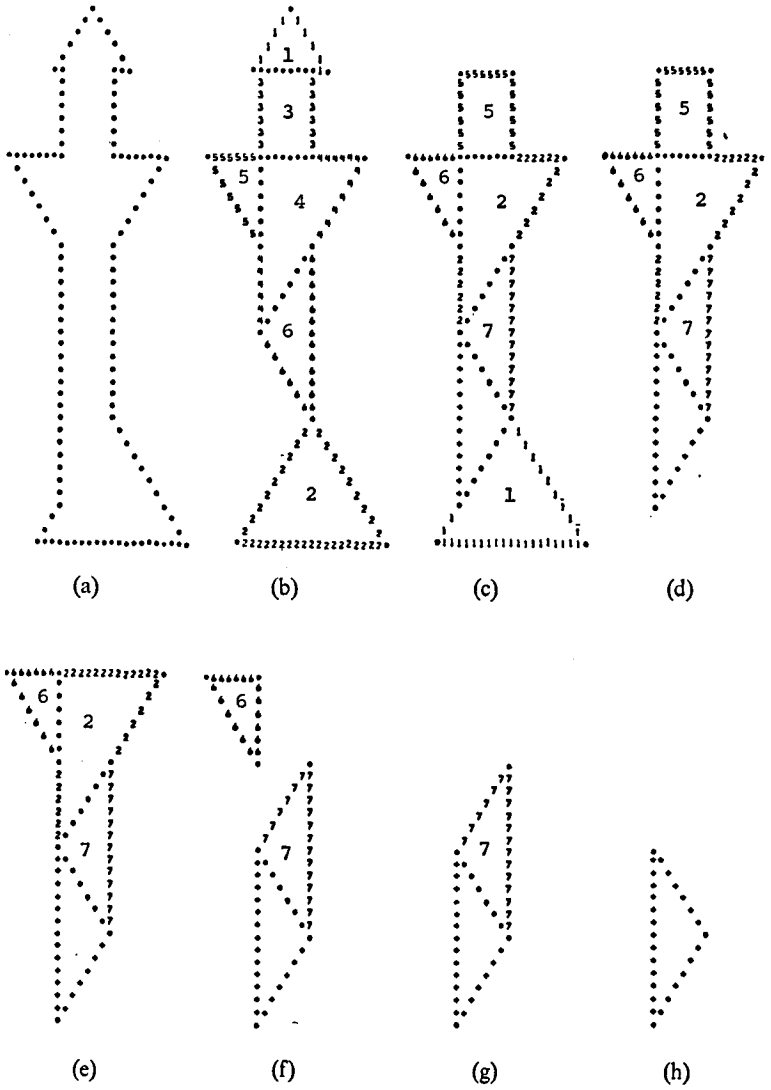


Figure 19

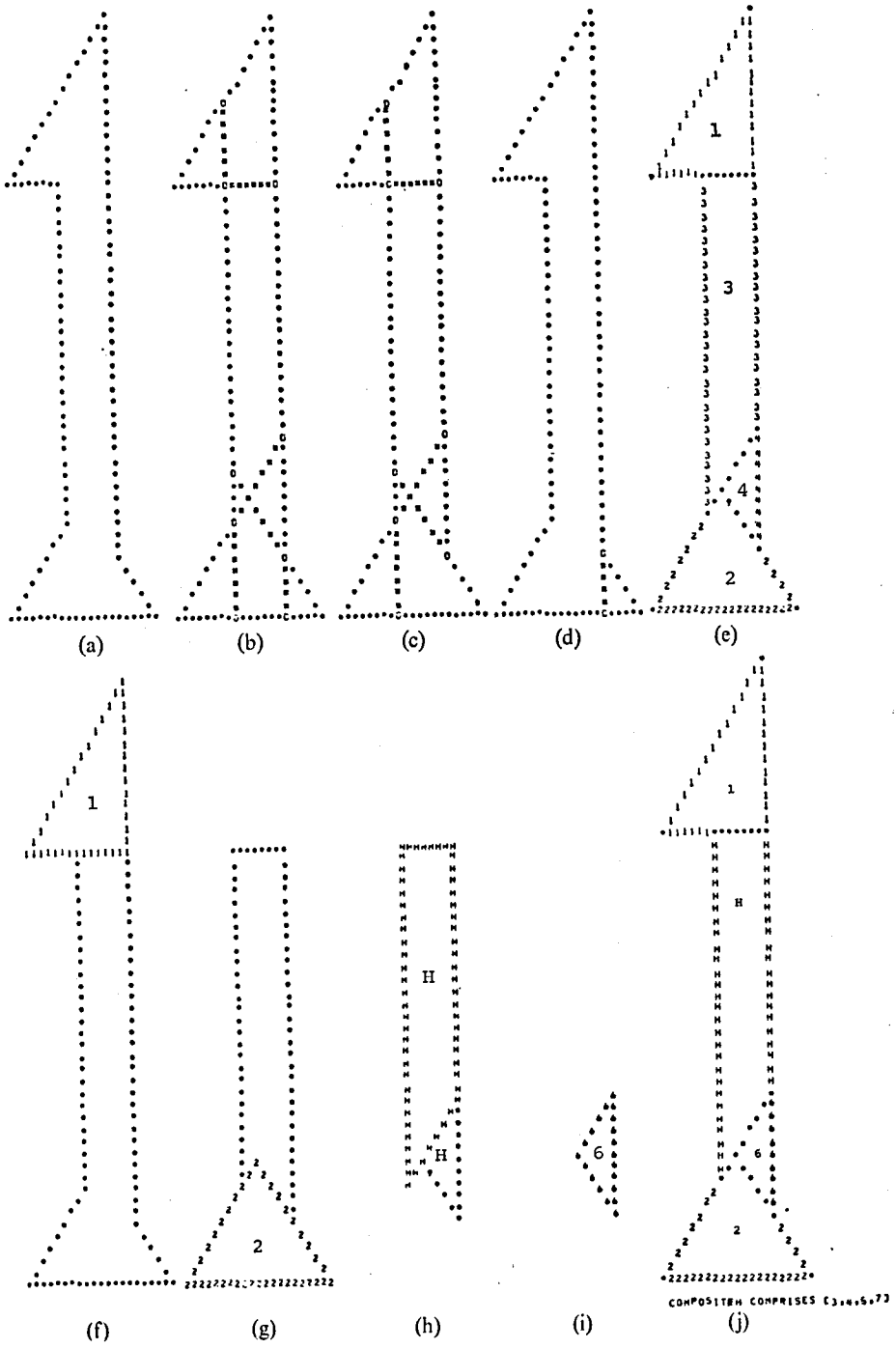


Figure 20

It will be observed on comparing the two solutions that the order in which puzzle pieces are extracted is different. This is because the 2-edge rule preceded the $1\frac{1}{2}$-edge rule in deriving the latter solution (see box 5 in figure 17). A correct solution is obtained either way in this case, but applying the new order of rules to the tangram of figure 13 produces an erroneous result as shown in figure 19. Here the program tries to elicit two medium-sized triangles. Observe that the old format is used here. The rules of figure 17 are used to solve the tangram of figure 9(b), shown in figure 20.

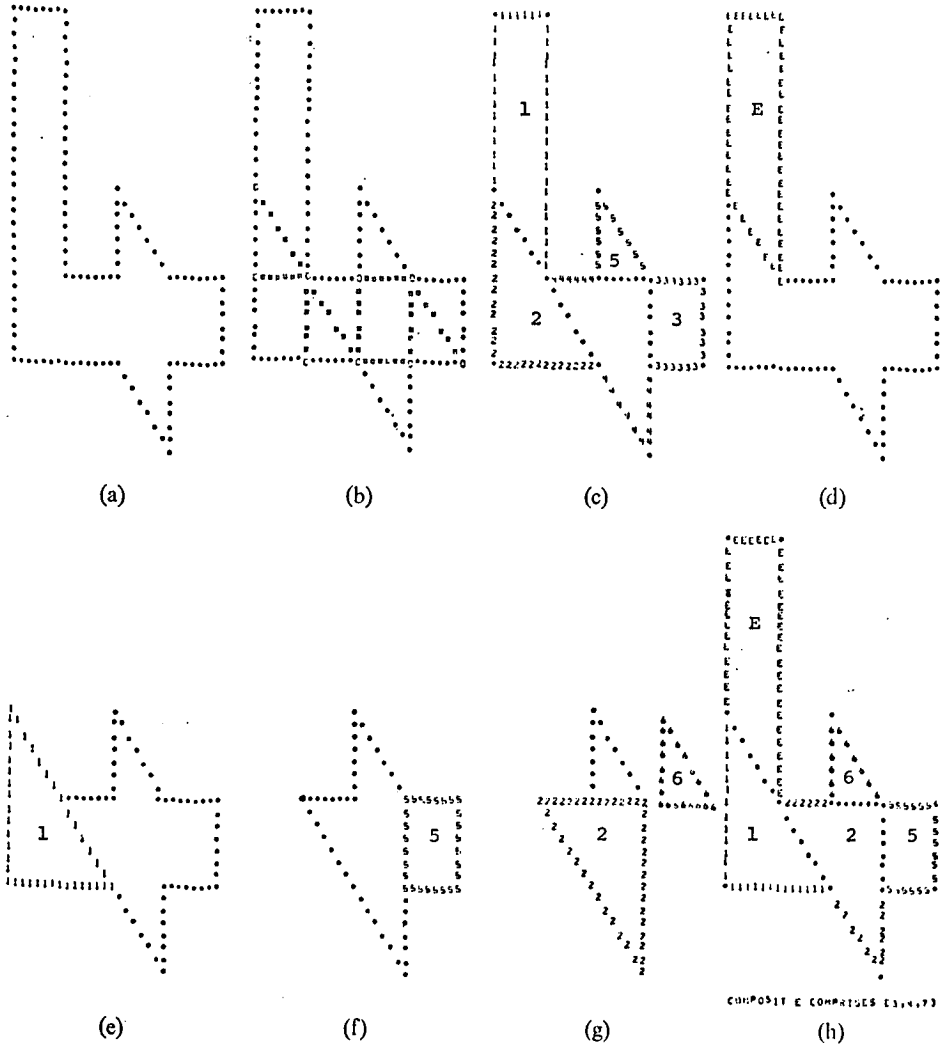


Figure 21

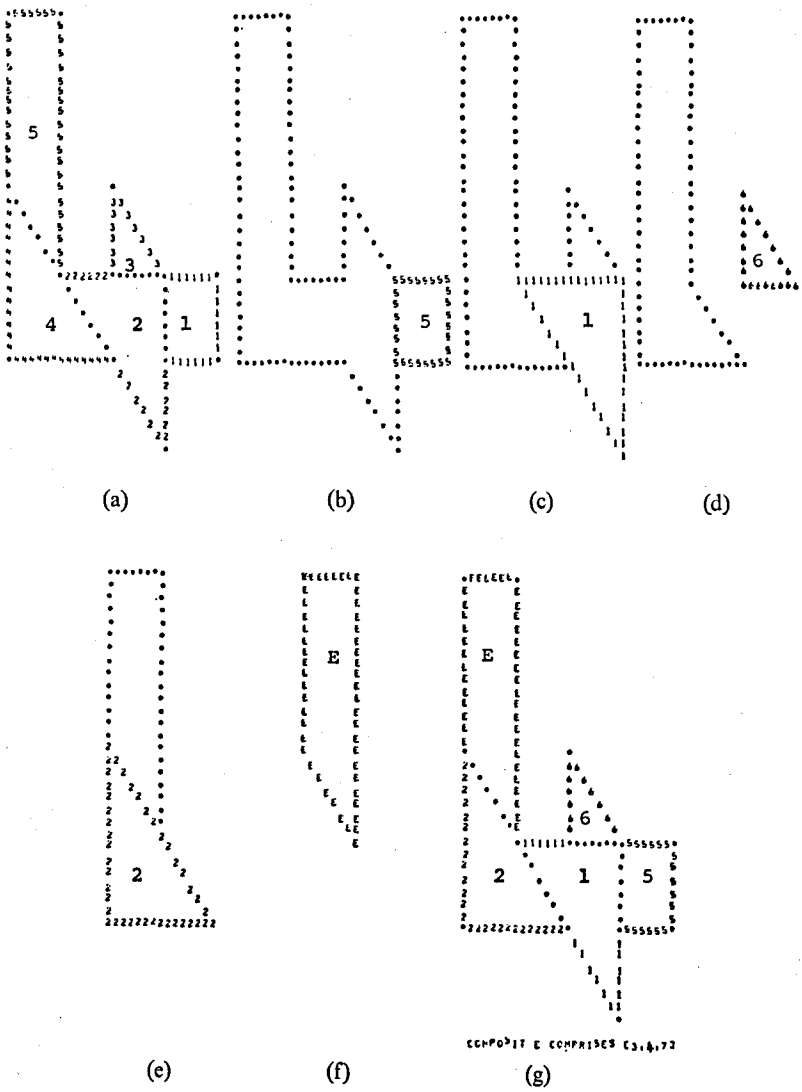


Figure 22

Figure 21 and figure 22 show the difference in the way a solution is obtained when the priorities of the rules for the extraction of puzzle pieces and the rules for the extraction of composites are interchanged. In general, if the rules for the extraction of puzzle pieces are given a higher priority, then the solution will occur earlier. When a composite piece is extracted earlier, the number of attempts at a solution will increase. Figure 23 gives the flow chart of the way the rules are finally organized. The chart consists of two parts:

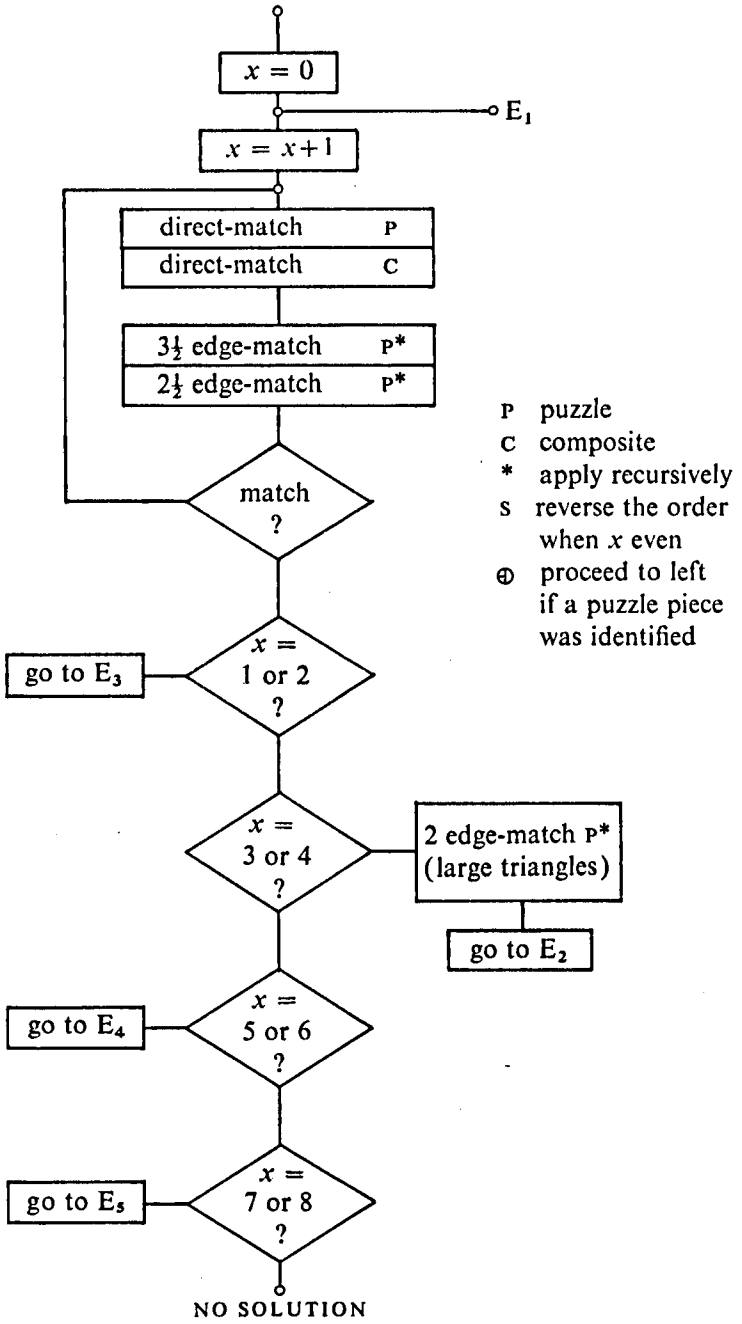

x = 0
E1
x = x+1
direct-match P
direct-match C
3½ edge-match P*
2½ edge-match P*
match ?
x = 1 or 2 ?
go to E3
x = 3 or 4 ?
2 edge-match P* (large triangles)
go to E2
x = 5 or 6 ?
go to E4
x = 7 or 8 ?
go to E5
NO SOLUTION
P puzzle
C composite
* apply recursively
S reverse the order when x even
⊕ proceed to left if a puzzle piece was identified

Figure 23(a)

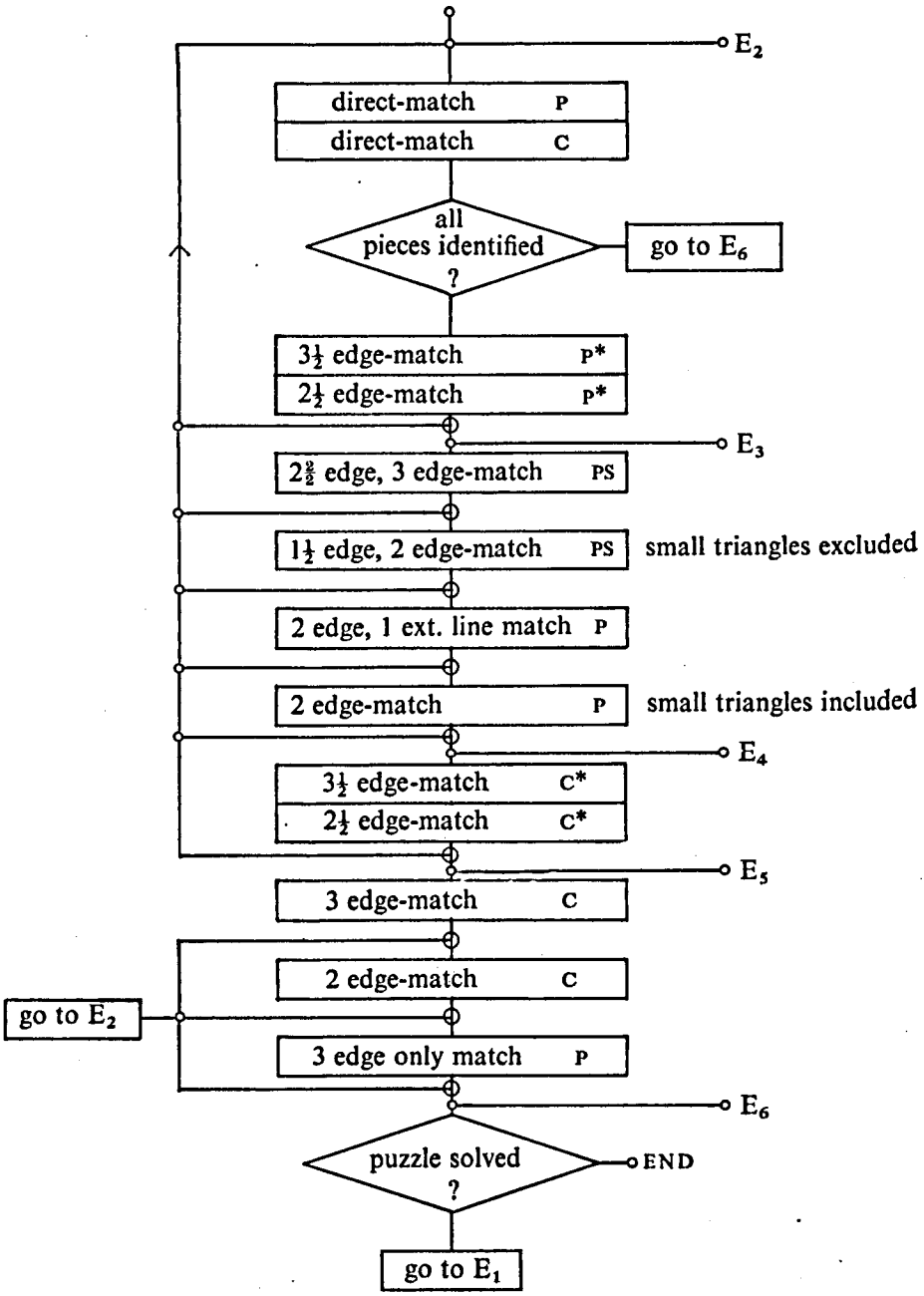

E2
direct-match P
direct-match C
all pieces identified ?
go to E6
3½ edge-match P*
2½ edge-match P*
E3
2⅝ edge, 3 edge-match PS
1½ edge, 2 edge-match PS
small triangles excluded
2 edge, 1 ext. line match P
2 edge-match P
small triangles included
E4
3½ edge-match C*
2½ edge-match C*
E5
3 edge-match C
2 edge-match C
go to E2
3 edge only match P
E6
puzzle solved ?
END
go to E1

Figure 23(b)

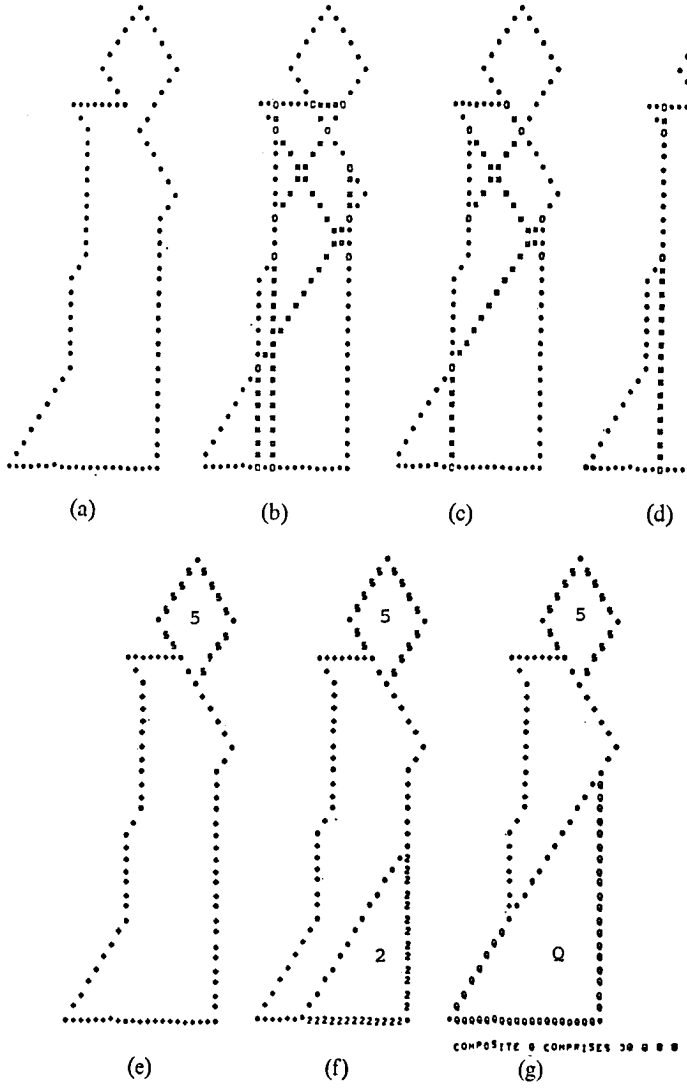


Figure 24

the main program, controlling the order in which the extraction rules are applied, and the remaining part consisting of the rules themselves.

Considerations of space prevent the showing of further results. However, figure 24(a) shows a tangram puzzle for which a solution was not found. The partial solutions obtained at various stages of the solution process are also shown.

CONCLUSION

It has been shown how a tangram puzzle can be solved using some very simple extraction rules. The rules are based upon the location of edges which define individual puzzle pieces and composites of puzzle pieces. The edges are either those defined by the actual puzzle or edges constructed subsequently. The less edges required by a given extraction rule the less reliable this rule is. This leads to the method by which the rules were ranked.

In principle, the approach used here is applicable to any form of puzzle, in which, like the tangram puzzle, the shape of individual puzzle pieces is simple, and their number is restricted. The material presented here could well be used with an interactive scheme in which the user can specify the particular rule he wishes to apply to a puzzle during a solution process.

Acknowledgements

The support of the National Science Foundation, under Grant GJ–754, is gratefully acknowledged.

REFERENCES

Freeman, H. & Garder, L. (1964) Apictorial jigsaw puzzles: the computer solution of a problem in pattern recognition. *IEEE Trans. EC–13*, 118–27.

Guzman, A. (1968) Decomposition of a visual scene into three-dimensional bodies. *Proc. AFIPS 1968 Fall Joint Comp. Conf.*, **33**, 291–304. Washington DC: Thompson Book Co.

Read, R.C. (1965) *Tangrams*. New York: Dover Publications Inc.

13

A Man-Machine Approach for Creative Solutions to Urban Problems

P. D. Krolak and J. H. Nelson
Vanderbilt University
Nashville, Tennessee

Abstract

The problems of the urban areas of the world today are large, complex, and inter-related. Often they involve ill-defined goals. Hence, solutions using the classical techniques of mathematical programming and operations research are frequently unsatisfactory. While many of these problems are being solved solely by human subjective judgment, it is becoming increasingly clear that the scope of the problems is beginning to surpass the skills of the decision makers. This paper outlines a new man-machine approach to creative problem solving which makes efficient use of human and machine resources. An information hierarchy is developed which has a potential impact on many areas of heuristic problem solving. Application of these ideas to several real world problems, garbage collection, school busing, and school busing to achieve a racial balance, are discussed. Other potential applications and future directions for research are also included.

INTRODUCTION

During the last decade a number of researchers have directed their efforts toward the area of problem solving. From the work of authors such as Newell and Simon (1972), theories of human problem solving are beginning to be formulated. Michie (1968), Gold (1971) and other authors (Sackman 1970, Kilgour 1971) have studied the man-machine system to see how efficient it is in solving problems. In another area of research, there are studies (Feigenbaum and Feldman 1963) in which machines (1) perform tasks that require intelligence, that is, mental tasks that are difficult for humans; and (2) attempt to simulate human behavior in problem solving. This paper discusses another aspect of problem solving – that of optimizing or improving the ability of the man-machine problem-solving system in order to find new,

creative solutions to large complex problems, including those with ill-defined goals.

Since a completely satisfactory theory of human problem solving has not yet materialized, it may seem premature to talk of optimizing either the performance of it or of the man-machine system. However, recent work (Krolak *et al.* 1972) in attacking complex distribution problems indicates that this study is both promising and productive and holds important implications for researchers in the areas of cognitive psychology, artificial intelligence, and heuristic problem solving, as well as operations research and management science.

Careful study of the man-machine systems currently in use shows that a great deal of design time went into the man-machine interface and into constructing machine heuristics for solving the problem while very little time was given to asking the question of whether this was the most efficient use of the man-machine resources. Up until this time research has concentrated on developing more efficient and easier-to-use graphics hardware and graphics languages, and on powerful computer problem-solving heuristics and/or algorithms. Similarly, applied mathematicians have concentrated on developing mathematical models that come as close as possible to modeling the real world.

Several important facts have been overlooked, however, and more than one such system has failed to satisfy the goals for which it was developed. Often the model does not match or completely characterize the complexity of the real world problem. Frequently the goals of the design problem are dimly perceived, ill-structured and conflicting. Because real world problems are dynamic in nature a manager frequently receives a report of an optimal solution to a problem that is no longer of interest to him because conditions have changed, the goals do not completely match his own, or the real world problem does not resemble the one modeled. Indeed, the literature of combinatoric programming places more emphasis on finding an optimal answer than it does on the potential benefit to the human once he has paid the price to get this answer. Consequently, many industrial and government decision makers have questioned the benefit derived from such studies. This situation can be reversed only if we begin to look at the man-machine system in a different perspective.

The human problem solver has many talents and/or problem-solving skills that currently are difficult or even impossible to simulate on a machine. The pattern recognition involved in reading human handwriting is one example; playing an excellent game of chess is another. Yet, we tend to ignore these human talents in the design of man-machine systems. One reason that is frequently given is that these talents exist in varying degrees in individuals and that the results of such a problem-solving system would be correlated to an individual's skills. These skills can also be a function of the human's external world and hence, even if we examined individuals to find

those who exhibit such skills, they can be dulled due to boredom, worry, age, or other stresses.

Any man-machine system design should recognize the varying skills of the human and it should strengthen and amplify them rather than reject them. How can this amplification of the human problem solver's skills be accomplished? It can be accomplished by means of a system in which (1) the machine organizes the data into an information hierarchy and stimulates the human's creative problem-solving skills by providing alternative heuristics; and (2) the human processes the resulting information hierarchy, the alternatives generated by the machine, and those external variables which he perceives to be relevant, and arrives at a solution. Thus a properly designed man-machine problem-solving system actually forces the human to provide the solution heuristic but, at the same time, it helps him create it.

The critical component of this system is the creation of the information hierarchy. Here we define information hierarchy to be those data structures which contain the key variables ordered according to their bearing on the problem.

The construction of an information hierarchy also provides many rewarding side benefits. For example, once the key variables have been identified, computer heuristics are frequently suggested that are efficient and accurate. These solutions in turn stimulate the human to combine those solutions which his skills in pattern recognition tell him are superior to any other or to the single solutions.

The man-machine process for amplification of human problem-solving skills

The development of efficient man-machine systems that are reliable, accurate, and have a rapid job-completion time is the desired goal. In addition, systems which stress man's creative skills and amplify them, by their very nature, will produce jobs which are more attractive and rewarding. Rather than stressing mechanization and technical ability they will emphasize the use of man's creative instincts and natural abilities. The author has observed that since the problem solver in his distribution system has a more active role he is stimulated rather than bored, and challenged rather than stymied. Thus, a further reason for looking to the man-machine approach is the possibility of creating a more meaningful and desirable role for future workers. The man-machine problem-solving process discussed here attempts to clarify the tasks, the methodology, and the goals of each component of the system with the overall goal of producing man-machine systems that amplify man's ability to be a reliable, creative, and responsive problem solver and decision maker.

The authors' approach consists of the following; (1) man defines the problem; (2) the computer organizes the data into an information hierarchy in such a fashion that isolates the problem's important features and gives

several solutions to show how the data might be organized into a whole by various heuristic methods; (3) man formulates a solution based on the information hierarchy; (4) man makes a comparison between the computer's solutions, his solution and the computer's organization of the data and creates a composite solution; (5) with the composite, man uses the computer to review and investigate various local and regional problems isolated by step 4; and (6), using his judgment and whatever theoretical information that might be at hand, man continues steps 4 and 5 until he has either exhausted all of the potential benefits to be derived or until further effort will be only marginally beneficial.

Using the above procedure, the decision maker is responsible for directing the search for the final solution, besides his usual role in the man-machine process. In so doing he becomes aware of how the various relationships interact and constrain his design freedom. This education allows him to adapt to future changes more intelligently. In addition, he gains some insight into how to analyse his problem. The system is also efficient in its use of the computer resource. Instead of using the computer to investigate a large number of solutions, as many heuristics currently do and of which only a small fraction are of interest, the computer spends most of its time working productively on those solutions that are likely to provide the final answer and/or are of interest to the human decision maker.

The information hierarchy

In observing students trying to solve various puzzles and problems taken from the literature of management science, one is struck by the extreme range in their problem-solving skills. Most students seem very weak at generating a heuristic while a very few seem to perceive the essentials of the problem and immediately produce satisfactory solutions. This suggests that having all of the data about a problem is often the same as having too little data. A mechanism for creating at least a partial ordering of the data seems to be necessary. This ordering should recognize the importance of relationships and should convey to the human problem solver the underlying structure of the problems, that is, order the data by its global, regional, or local importance. The creation of data hierarchy to establish relationships is a more general concept of data organization than the work done in cluster analysis and numerical taxonomy. In many ways this exemplifies the way in which a human solves many of his daily problems, for example, looking up a word in a dictionary or finding a nearby plumber in the yellow pages of a telephone directory.

Creating this information hierarchy involves finding mathematical models to locate key relationships rather than to optimize a model of the original problem. By re-defining the problem, we drop much of the detail and thus can create an information hierarchy much more cheaply in terms of computational time. This detail will be re-introduced at the interactive stage. In fact,

normally we expect the human to bring into the interactive solution detail and concepts from the real problem that cannot be described in the usual computer model.

While the authors have examined only various scheduling, distribution, and communications network problems they have found that these problems seem to have natural information hierarchies. Work in statistical cluster analysis by Johnson (1965), numerical taxonomy by Sokol and Sneath (1963), and multidimensional scaling by Neidell (1969), provide insights and mechanisms for deriving information hierarchies for an extremely wide range of real world problems.

Data display and the man-machine interface

As we shift the role of the machine over to organization and stimulation, it also becomes necessary to review the interface of our man-machine system. The machine must be capable of providing displays that transmit the information hierarchy and/or alternative solution approaches. The machine must also be able to play a question and answer role and to provide rapid access both to large-scale data files and to a large variety of algorithms and heuristics. The current studies have used only a teletype terminal and light board, which until now have proved adequate. However, the authors have recently examined the plasma display panel (Alpert and Bitzer 1970), which allows a user to superimpose computer graphics and a wide range of audio-visual displays, and it seems to be a more desirable mechanism for solving complex industrial problems at only a modest rise in hardware costs.

A major piece of research, currently in the preliminary stage, is concerned with determining what types of displays transmit to the human the essence of a problem that has no current analog of a map or a chart. Up to this time the problems investigated have all been displayed in map or chart form, but many important classes of problems do not have this convenient property. A forthcoming paper will report on these results.

Computer heuristics based on the information hierarchy concept

In many cases the computer heuristics for solving industrial problems have been extremely crude. In general, these heuristics start out with a collection of rules of thumb, and the computer tries a large number of random combinations of these rules until it arrives at a solution which cannot be improved further (Christofides and Eilon 1969, Hayes 1967, Oberuc 1968). These heuristics encourage accuracy over cost. This frequently overlooks the fact that the potential benefit to be derived is equal to the improved profit in the model solution minus the cost of solving the problem. In some cases the cost of using an exact computer algorithm or heuristic exceeds the benefit, owing to emphasis on accuracy and the use of the computer as a brute force number-cruncher. The authors believe that the use of the information hierarchy provides a global view of the problem and permits the generation of heuristics

that have relatively small errors and require very little computer time (Felts 1970, Krolak, Felts and Marble 1971). Also the errors are normally due to small local errors and regional errors that a human can easily locate and correct during the interactive phase. Thus, the computer heuristics based on information hierarchy ideas are of interest because of their own performance as well as for their ability to stimulate the human within the man-machine process.

URBAN PROBLEM SOLVING

The creative man-machine problem solving techniques of this paper could be applied quite profitably to various areas of study in artificial intelligence. For instance, instead of developing game-playing machines one could develop man-machine systems that would allow amateur players to become aware of the structure of the current game and to perceive the board as a professional player might. This would represent a major departure in analysis from the heuristic programming approach. This research would also provide insight into how to improve the game playing machines for the more complex games (for example, go and chess). While this type of research would seem to be highly challenging and productive, the authors feel that the more pressing needs of society demand precedence.

The urban areas of the world have been growing rapidly during the last several decades. This rapid growth has caused the problems that have always plagued cities – health-care delivery, pollution, crime, slums – to become even more acute. The social, economic and political factors involved in trying to deal with these problems and the increasing demands of the people for a better life compound the problem further. The old methods of reform through democratic processes and judicial relief have proven inadequate in their present condition and in need of modification in order to deal with the increasingly complex interactions of a highly industrialized society with its environment.

A brief look at a few problems that are currently under attack in the urban systems analysis may provide the reader with some insight as to how his own talents might be applied in this area of pressing importance.

In the United States a recent court decision has required that all political districts represent a relatively uniform number of people. Due to this so-called one man/one vote decision many difficulties have arisen in state and local governments. Political boundaries had always been established by legislative branches of the government and it was not uncommon for the parties in power to create districts that were designed to keep them in power rather than ones that were nearly equal in population, compact in size and shape, and whose boundaries corresponded to the traditional natural and social boundaries of the state. Thus courts have continuously had to redesign districts to conform to the later qualifications. Recently, both mathematical programming models and heuristic computer programming techniques have

tried to capture the essence of the courts' fairness criteria. The application of these tools to real problems has had mixed success. While various parties of re-districting litigation have argued the merits of the computer drawn maps the courts have nearly always modified these plans to correspond to their subjective judgments (Steinberg 1972 and REDIST-National Municipal League 1969). The legislatures, likewise, have not widely accepted these computer-drawn maps, tending to play political games but modifying their designs to attempt to satisfy the desires of the court.

The design of location sites for public facilities that deliver services (fire houses, ambulances, post offices, schools, etc.) has recently begun to receive attention (McCloy 1972, Symms 1969, Eastman and Kortanck 1970, Rapp 1972, ReVelle and Swain 1970, Cooper 1967). Simulation models, mathematical programming and heuristics have begun to be used in the location of facilities but many problems that are ignored by these models hamper their usefulness. Man-machine designs currently underway at least allow subjective judgments as part of the input (Eastman 1969, 1971; Rapp 1972). However, the idea of supplying the designer with an information hierarchy that forces him to become aware of a more complete range of ideas, views, and relationships has not been established.

Recently, the authors have applied their ideas to a series of various urban distribution problems:

(1) The traveling salesman problem (given a collection of cities (nodes), route a salesman in such a manner as to form a circuit of minimum cost or distance).

(2) The truck dispatching problem (given a fleet of trucks stationed at one or more warehouses and a collection of customers at various locations with known demands, route the trucks in a fashion to minimize the total distance traveled subject to the demand that no truck exceeds its load capacity).

(3) School busing (same as the truck dispatching problem with the additional requirement that the lengths of the individual bus routes are constrained so that no child has too long a ride).

(4) School busing for racial balance is still a more complex problem. Court-ordered racial balances, school capacity constraints, economic and social constraints must be achieved as well as a host of ill-defined goals.

Traveling salesman

The first attempt was at solving large-scale traveling salesman problems. The results, using a man-machine process, are presented in a series of papers (Felts 1970, Krolak, Felts and Marble 1971). The solutions are accurate for problems up to 200 cities and computer times grow only slightly faster than linearly with the number of cities. The results are also reproducible. Using untrained students, solutions with errors of less than one per cent over the best known solution have been obtained.

Truck dispatching

More recently the author undertook to investigate the truck dispatching problem which, for problems of any practical size, has defied solution by algorithm and even most heuristics leave a great deal to be desired (Christofides and Eilon 1969, Clarke and Wright 1963, Dantzig and Ramser 1959, Hayes 1967, Tillman 1969). Using the man-machine approach the authors found better solutions to all problem sets currently in the literature, and at a cost and speed that indicate that the scheme is efficient enough to be incorporated in the daily routine of dispatchers (Krolak, Felts and Nelson 1972). In addition, the authors have used it to solve a wide variety of related problems, also with good results. The technique was tested on an actual garbage pick-up route of a medium sized city and a fifty per cent reduction in travel resulted. For other approaches to the garbage pick-up problem, see Tanaka (1972) and Bodner (1970).

The information hierarchy of the traveling salesman and dispatching problems is fairly obvious. Distance is the key variable and the recursive application of the assignment algorithm is used to cluster the cities into probable inner city links, regional groupings, and global characteristics. The data are displayed on overlay mats and, together with the map of the problem and a light board, allow the human to organize the information into an accurate approximation of the optimal solution.

School busing

A generalization of the truck dispatching problem that has attracted a good deal of attention in recent years is that of how to operate a school bus fleet efficiently. Besides the usual load constraints the bus route must satisfy limitations on the total length of the ride a child must take to get to school. In addition the bus must be able to perform all required stops and turns safely. This limits the size and kinds of buses that can undertake a given route. The man-machine approach makes greater use of the human's ability to store vast amounts of data and experience about the problem. The fact that an 84-passenger school bus cannot turn around at a certain point and return to the main road, but a 72-passenger bus can, is an example of this. Hence, solutions are feasible in the real world as well as in the model world. This reduces the human problems associated with the acceptance and implementation of any changes in the routing patterns based on computer models. Since those who must do the work also participate in the creative solution of the schedule they have a stake in making the system a success. This overcomes problems such as those experienced by Kingston (1970) where drivers and managers rejected computer generated solutions because they did not understand the output, felt threatened by its existence, and could find so many flaws in the solution.

The man-machine results have been compared to a number of well-known school busing codes (Newton 1970, Gillette and Miller 1971, System 360

V.S.P. 1967) on real world and test problems. The solutions generated by the man-machine system are shorter in total route length, shorter in length of the longest route, and make better utilization of the bus fleet (fewer buses, more even loads and route lengths, etc.). Generating solutions to real world problems requires only slightly more computer and human time than similar sized test problems. Hence, it appears that the man-machine system is superior to current computer heuristics for this type of delivery problem.

Computer heuristics based on the information hierarchy suggested themselves more or less naturally, and they represent only a small incremental cost in computer time over that already expended for the construction of the information hierarchy. These heuristics are both fast and accurate. A recent study of several traveling salesman heuristics reported on in the literature (Krolak, Felts and Nelson 1971, Oberuc 1968, Gillette and Miller 1971) showed that heuristics based on the information hierarchy produce a better solution at a lower computer cost when solving large problems. A possible exception to this observation may be the new heuristic of Lin and Kernighan (1972) which, for problems of 100 cities or less, seems to produce slightly more accurate results for approximately the same to slightly higher computer costs. Since the authors found it easy to generate different heuristics that are accurate and efficient, the human is given sufficient stimulation to examine various alternatives. In this way the human problem solver avoids repetitious, non-productive activity.

School busing for improving equal opportunity in schools

The school busing for racial balance problem is the most difficult test of the man-machine process to date. School desegregation in the United States has been underway for many years. Within the last twenty years the courts and the legislatures have outlawed all forms of *de jure* segregation in schools; however, due to established housing patterns and social and political customs, *de facto* forms still exist in the urban school systems. Recent court decisions have attacked this form of segregation and have relied on a number of methods to bring about a remedy. In almost all cases these remedies involve some combination of teacher and pupil reassignment, redrawing of political and school boundaries, and relocating future school construction sites (Brown 1972, Nelson and Krolak 1972, Clarke and Surkis 1968). Almost all plans depend heavily on large-scale busing to achieve their ends. The resulting plans have been highly unpopular, expensive, and initially very disrupting to implement. While the educational and social results will take years to measure accurately, it is safe to say that the problems caused by planning, implementing, and managing this massive reorganization have taxed school officials to their limit.

It is possible that the reaction of society will be so negative as to bring about an amendment to the constitution or to force through legislation to hamper or reduce the effectiveness of the court orders. While this possibility

exists, the problem still remains and will have to be dealt with until that future time. Currently, the popular dislike for the measure is causing pressures to cut taxes and to refuse to pass school bond issues, thus reducing school revenues. The threatened reduction in revenue coupled with the increased cost of busing and the current state of inflation have placed the schools in a difficult economic position.

The man-machine system currently being designed to attack this problem deals with more of the complexity of the total problem than any previous system. In previous attempts to attack the problem the model addressed itself only to the assignment of children to schools in such a manner as to achieve court-ordered ratios at minimum transportation costs (Kraft 1972, Pugh 1972). The quality of the schools used, the effect on education, the special needs of the bused pupils regarding teachers, equipment, and other educational resources were disregarded. Quite predictably such plans frequently require many modifications before they produce a solution that is also feasible in the real world.

An integer programming model (Nelson and Krolak 1972) can be formulated that can state the desired goal (minimize the total number of children exposed to substandard educational opportunities) subject to the court-ordered racial balances of teacher and pupils, the economics of the operation of schools, bus fleets, and faculties, the assignment of children to schools, the addition or improvement of classrooms, etc. However, an integer program for any reasonable sized school district would result in a problem too large and costly to solve. But, by undertaking the construction of an information hierarchy, a much more modest linear programming problem is obtained. This results in a pupil assignment map overlay together with an overlay of the information hierarchy based on distances and the pupil location which allows for a number of humans to use their experience and subjective judgments to arrive at solutions that are both feasible and acceptable.

Our man-machine design for planning a school system with equal educational opportunity becomes:

(1) *Man defines the problem.* Recent reports and studies done for the school board assess the merit of the physical plants and the cost of various possible improvement, the merit of various teachers, etc. In addition, a pupil-locator data file has been created. This data, when given to the computer, creates a data base for the interactive system. The school board then defines the goal that they are interested in, for example, minimizing the number of children going to the lowest rated schools. The constraints on this action, such as budget considerations, pupil travel time, court orders, etc., are also defined. The goals and constraints may be modified or changed completely as the study progresses at the discretion of the planner.

(2) *The computer organizes the data.* A computer model that approximates the true problem is solved and is used to generate an information hierarchy of relationships. These relationships are then displayed on computer-

generated overlay maps. These maps illustrate what are the most probable matchings of population cells to schools, based on economic factors such as the costs of transportation, renovation and expansion, geographic and ethnic factors, and other considerations. The maps are crude solutions to the real problem and describe the general character of the final solution.

(3) *Man organizes the data into a solution.* Based on the information hierarchy the planner imposes a solution. Owing to the size and complexity of the problem, the planner generates a gross solution, that is draws up the school zones, etc., but does not go into complete detail such as routing the school buses.

(4) *The computer generates a total solution based on the heuristic method.* Because of the speed of its computation, the computer solution allows for the complete details to be worked out. The computer heuristic is based on the information hierarchy data base. Man compares the computer solution with his own solution and perhaps any others generated by parents' groups, etc., and generates a composite or compromise solution.

(5) *The current solution is reviewed to determine how it compares with the information hierarchy.* The opinions of educators and other citizens may be drawn upon to criticize the current solution. Regions of the solutions that are not felt to be particularly well handled are interactively examined and improved, based on modification of various heuristics (Newton 1970, Nelson and Krolak 1972, Gillett and Miller 1971). Solutions that show promise or other details of interest are stored.

(6) *The work is assessed and an estimate of the benefit of further study is made.* The solutions that result are reviewed for their economic and educational value and the decision as to whether to continue work or to accept one of the solutions is made. If it is felt that some benefit can be derived Step 5 is repeated.

The resulting solution contains the following details: (1) recommended budgets and allocation of teachers to schools: (2) renovation plans and location of temporary classrooms; (3) assignment of pupils to schools; (4) bus routes for getting pupils to schools; and (5) any additional data such as percentages by race at each school, maximum distances that pupils travel on each school bus, etc., can also be generated in standard report form for examination by any authorized individual or group. The benefit of such a powerful tool in answering difficult questions, in addition to developing a real time management system, should not be overlooked.

The following two example problems were used to test various features of the system, and although they are not real world problems, they serve to illustrate the technique:

(1) A 91-stop, 3-school problem based on randomly generated coordinates. This problem is based on Problem 28 of the authors' truck dispatching work. Loads were randomly generated, with black bias to the northeast and white bias to the southwest.

2. A hypothetical problem using 150 stops and 5 schools, based on a sector of a large metropolitan school system. Loads were generated, loosely based on the population distribution of the area in question.

In both problems, each stop has a 2-vector associated with it which reflects the number of black and white students to be picked up at that stop. Each bus has a scalar associated with it which identifies its total capacity for pupils, and each school has a 2-vector reflecting its maximum capacity for each type of student, if filled to capacity.

Figure 1 indicates the results of the application of a clustering algorithm to the bus stops only (the schools were excluded from this procedure). Also indicated are the assignment of clusters of stops to the schools. This assignment was generated using a simplified version of the linear program referred to earlier. This model assigns clusters of stops to schools so as to satisfy racial constraints, while minimizing the sum of the radial distances from the centers of gravity of the clusters to the schools.

Figure 2 illustrates the human's generation of preliminary routes based on the requirement that only three vehicles are available at each school. Each cluster shown would be served by one vehicle, and it is assumed that each route is optimized according to the traveling salesman criterion. It should be noted that in order to satisfy the three-vehicle constraint, each bus stop must contribute its load to one and only one school, this requirement thus destroys the racial balance obtained previously (figure 1). Since the balances shown in figure 2 do not satisfy the requirements, the human then attempted to more nearly meet the requirements by simply altering the assignment of routes to schools, while not weighting transportation cost heavily, since later passes at the problem will reduce transportation cost. The results of this crude procedure are shown in figure 3. At this point, the racial balance constraints have not yet been satisfied, and the anticipated transportation cost is still rather high; however, the data of figures 1, 2 and 3 do illustrate a portion of the hierarchical data base which the human will use in proceeding to the interactive phases of the solution process.

Figure 4 shows the results of the application of the interactive process to example problem 1. In this case, the man part of the man-machine process was a student with no prior experience in routing, optimization, etc. His final results after approximately twelve hours of his time and six minutes of computer time are shown in figure 4.

It should be noted that the solution of Problem 28 shown in figure 4 has many overlapping routes; this is a consequence of the restriction of only one bus visiting a stop. In the second problem, designated SW, the bus stops have, for the most part, pupils of only one type, and it has been found that this type of problem is more conducive to LP solution, since the LP solution more nearly indicates the solution of figure 7.

Figure 5 shows the same solution as figure 4, but after the traveling salesman algorithm was applied, with the first link (school to first bus-stop)

allowed to be zero, thus minimizing, for each route, the maximum riding time for students loaded first.

Figures 6, 7 and 8 illustrate the successive stages of solution to example problem 2. The interactive portion of the solution process was performed by the student who solved example 1; however, he had gained experience by this time, and was able to achieve the final solution using much less time.

Additional benefits to be gained by this type of planning

The current problem facing the school boards is the assignment, transportation, and education of the children of metropolitan areas. The best solution to this problem can be found through analysis by an interactive man-machine approach. The following points will illustrate the superiority of man-machine interaction over computer models.

(1) The goals of equal educational opportunity, while being partially measured by such quantities as average dollar spent/pupil/race or percentage of a given race in adequate or above adequate facilities, etc., do not give a complete picture. The subjective views of professional educators, school boards, and the courts are important aspects of the problem but cannot be brought to bear on the question if only a computer model is used. However, within an interactive model these judgments can be utilized.

(2) The interactive total system analysis provides quick and accurate answers to the 'what if' types of questions that arise in the planning stage. Later in the implementation of these plans the system can be used to find rapid solutions to the problems that arise in trying to smooth out random difficulties following the period right after school openings. This cannot be duplicated by piecemeal computer models.

(3) Since an interactive total systems model can address itself to the complete problem and not just to one piece of it, the problem of having too many children assigned to one school, or poor utilization of buses, etc., will not occur. Thus, oversights that frequently occur owing to one department failing to communicate its problem to the others are kept to a minimum.

(4) The cost of systems development, maintenance, and start-up time would appear to be far smaller than computer models based on previous experience with similar problems.

(5) The role of educating the planners to all aspects of the problem is also a key benefit. The ability to draw up and to defend a plan of action is greatly enhanced by this interaction since the planner will have been forced to review many of the questions raised as part of the solution development.

Thus, an interactive program can provide the school board with a powerful planning tool. The judgments of the board provide the final solution based on the use of this tool and, while this may seem to be less objective to some, it provides the human element necessary for so important a decision.

CONCLUSION

The man-machine system to amplify the creative problem-solving skills appears to be successful in allowing the authors to attack a wide range of distribution problems. The potential benefits to be derived by using this approach for resolving many urban problems demands further research. Questions of how to organize and transmit data best to allow a human to attack complex problems with accuracy and reliability should lead to an improved theory of human problem solving. Further work in this area should lead to more well-defined concepts for heuristic construction and to more powerful tools for industrial applications and artificial intelligence research.

Acknowledgements

The authors are indebted to the many students who volunteered their time and efforts towards testing and working the many examples, in particular Wyatt H. Pickler and Walter Steele. The authors would also like to express their thanks to Dr Wayne Felts for the many discussions and insights he contributed to this paper. The Vanderbilt University Computer Center provided assistance in computer time for this study. Finally, this research was in part supported by NSF Grant #GK-4975 L/C.

Table 1. Loads and coordinates for problem 28.

Node	Load (1)	Load (2)	X	Y	Node	Load (1)	Load (2)	X	Y
1	12	5	95·6	26·1	51	14	3	105·2	30·3
2	4	12	2·5	47·6	52	11	12	76·4	12·0
3	4	1	109·2	9·0	53	3	27	5·5	19·1
4	4	11	5·4	44·9	54	6	7	68·8	35·0
5	1	4	49·7	20·2	55	5	25	21·0	22·9
6	3	16	14·8	28·1	56	5	2	85·4	50·1
7	9	4	94·3	29·9	57	2	3	53·3	38·5
8	1	2	60·1	21·1	58	9	3	92·5	41·2
9	6	11	41·6	45·3	59	17	3	101·8	44·7
10	2	2	58·9	44·3	60	4	4	59·5	51·5
11	3	10	1·5	45·6	61	10	9	83·0	5·2
12	9	4	93·6	35·9	62	8	4	85·2	43·8
13	6	17	28·4	37·0	63	13	7	89·1	37·8
14	11	12	77·6	10·3	64	12	4	92·4	46·6
15	11	4	85·4	53·1	65	9	15	57·2	27·8
16	10	7	88·9	8·9	66	5	13	38·4	30·2
17	8	6	76·7	40·1	67	4	9	17·0	53·7
18	1	26	10·3	0·2	68	3	8	51·6	1·6
19	3	29	20·4	4·0	69	10	3	104·6	24·4
20	11	5	84·8	53·3	70	2	9	10·9	33·5
21	8	3	96·3	29·3	71	8	7	83·5	4·2
22	3	14	35·2	6·5	72	3	4	68·8	4·0
23	2	1	91·5	50·8	73	9	6	88·8	10·4
24	1	23	6·0	1·0	74	2	1	95·7	28·1
25	7	8	68·8	32·2	75	2	5	26·3	45·9
26	4	2	97·0	1·1	76	7	3	94·1	34·1
27	6	14	20·3	50·9	77	4	24	2·1	29·3
28	3	0	97·6	55·0	78	3	27	2·6	19·0
29	0	19	1·3	4·2	79	5	3	94·4	2·1
30	3	9	39·0	24·0	80	9	13	56·5	40·2
31	5	12	46·4	33·6	81	5	4	83·8	0·0
32	6	2	95·4	38·6	82	3	8	15·7	47·1
33	7	21	44·8	7·0	83	3	2	93·3	21·5
34	3	19	10·5	24·0	84	13	10	78·0	40·5
35	6	4	83·3	34·2	85	3	15	4·0	32·6
36	7	1	100·3	48·3	86	2	16	25·4	3·0
37	4	25	7·8	28·3	87	6	11	27·2	54·9
38	4	11	49·0	2·6	88	9	6	84·2	33·3
39	2	12	29·3	10·2	89	7	6	81·9	1·1
40	6	5	78·3	33·4	90	4	12	45·9	18·1
41	7	14	25·3	50·5	91	5	26	24·1	19·7
42	6	16	47·9	19·6					
43	1	6	24·1	6·0					
44	10	11	75·7	7·8					
45	8	8	71·7	35·4					
46	9	2	108·4	7·1					
47	8	3	99·4	14·4					
48	4	10	48·2	15·4					
49	9	6	80·2	47·4					
50	12	10	67·2	50·0					

Table 2. Loads and coordinates for problem SW.

Nodes which have loads (0, 10):
20, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 115

Nodes which have loads (10, 0):
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 60, 61, 62, 63, 64, 106, 107, 108, 109, 110, 111, 112, 113, 114, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150

Nodes which have loads (5, 5):
59

N	*X*	*Y*	*N*	*X*	*Y*	*N*	*X*	*Y*	*N*	*X*	*Y*	*N*	*X*	*Y*
1	20	290	31	580	330	61	690	680	91	926	828	121	830	590
2	40	210	32	560	390	62	798	775	92	898	818	122	880	570
3	100	300	33	550	420	63	590	690	93	906	835	123	685	570
4	90	350	34	580	430	64	630	710	94	895	826	124	735	560
5	200	210	35	600	470	65	815	770	95	884	835	125	750	560
6	250	210	36	630	500	66	805	770	96	875	825	126	760	530
7	250	190	37	650	490	67	670	680	97	883	820	127	720	530
8	280	180	38	640	450	68	795	785	98	889	810	128	695	520
9	200	170	39	640	415	69	812	786	99	876	805	129	730	510
10	340	450	40	620	390	70	815	780	100	870	813	130	700	465
11	350	475	41	550	500	71	815	805	101	947	845	131	695	430
12	360	500	42	510	500	72	826	815	102	950	860	132	775	460
13	380	500	43	460	500	73	826	800	103	963	847	133	790	480
14	390	530	44	440	560	74	854	805	104	960	855	134	810	520
15	360	530	45	545	570	75	840	800	105	770	700	135	845	525
16	370	570	46	495	690	76	859	795	106	805	690	136	880	495
17	400	570	47	545	690	77	843	790	107	790	690	137	850	480
18	430	580	48	490	750	78	830	790	108	755	680	138	895	470
19	430	600	49	505	770	79	850	740	109	750	680	139	890	430
20	430	640	50	530	790	80	865	765	110	775	660	140	850	410
21	480	630	51	550	760	81	883	755	111	735	650	141	800	415
22	410	430	52	590	810	82	943	778	112	755	640	142	630	310
23	450	450	53	675	120	83	963	795	113	725	640	143	620	265
24	500	370	54	605	680	84	981	805	114	730	600	144	670	250
25	495	410	55	620	760	85	994	790	115	830	680	145	690	300
26	460	370	56	660	770	86	979	781	116	860	630	146	720	320
27	460	330	57	675	750	87	940	820	117	830	640	147	790	270
28	430	400	58	720	750	88	924	818	118	800	620	148	760	255
29	400	370	59	730	700	89	925	845	119	805	600	149	750	200
30	380	410	60	700	710	90	919	835	120	855	600	150	640	160
S1	810	490	S2	670	700	S3	880	780	S4	220	260	S5	500	520

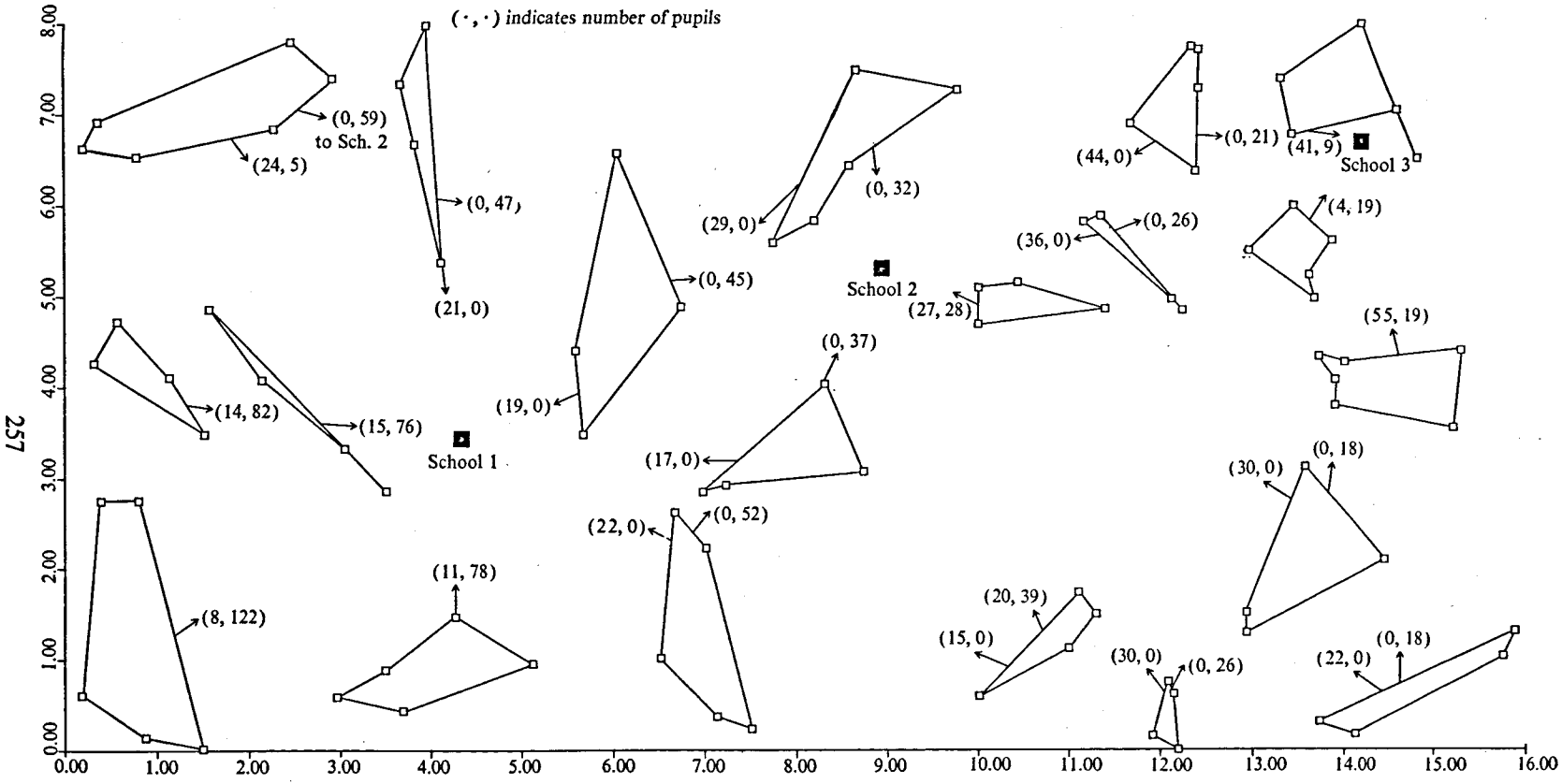


Figure 1. Linear program solution for problem 28. We assume that each school has a capacity of 558 by assuming that three buses, each having a capacity of 186, arrive at each school. The ratio limits used in the linear programming problems were 35–49 per cent.

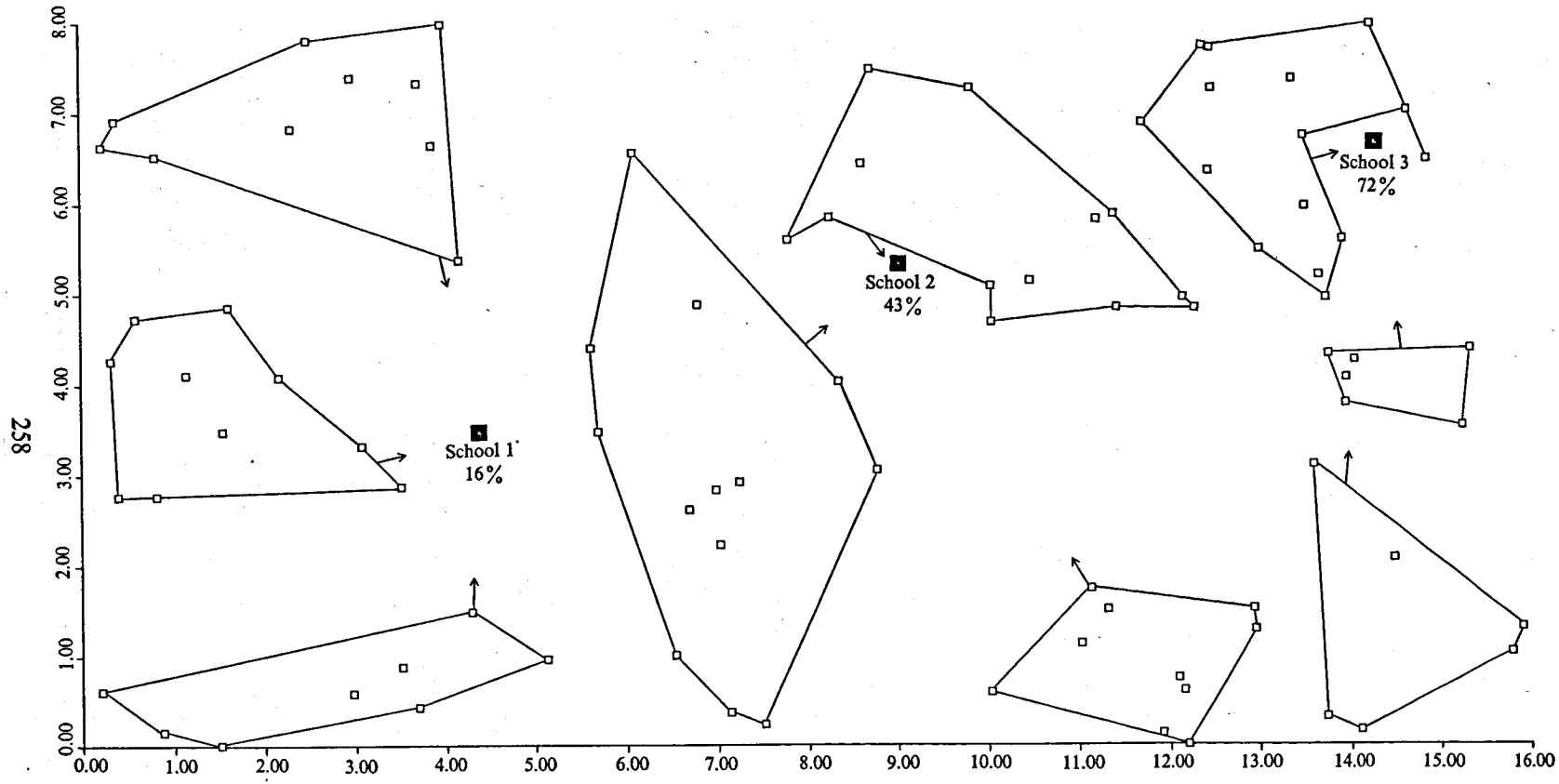


Figure 2. Initial estimate for problem 28. The clusters of stops enclosed in the circles are presumed to indicate which stops will be on a route, although the routes are not explicitly drawn in. Percentages shown by each school are the per cent of black pupils out of the total pupils at that school.

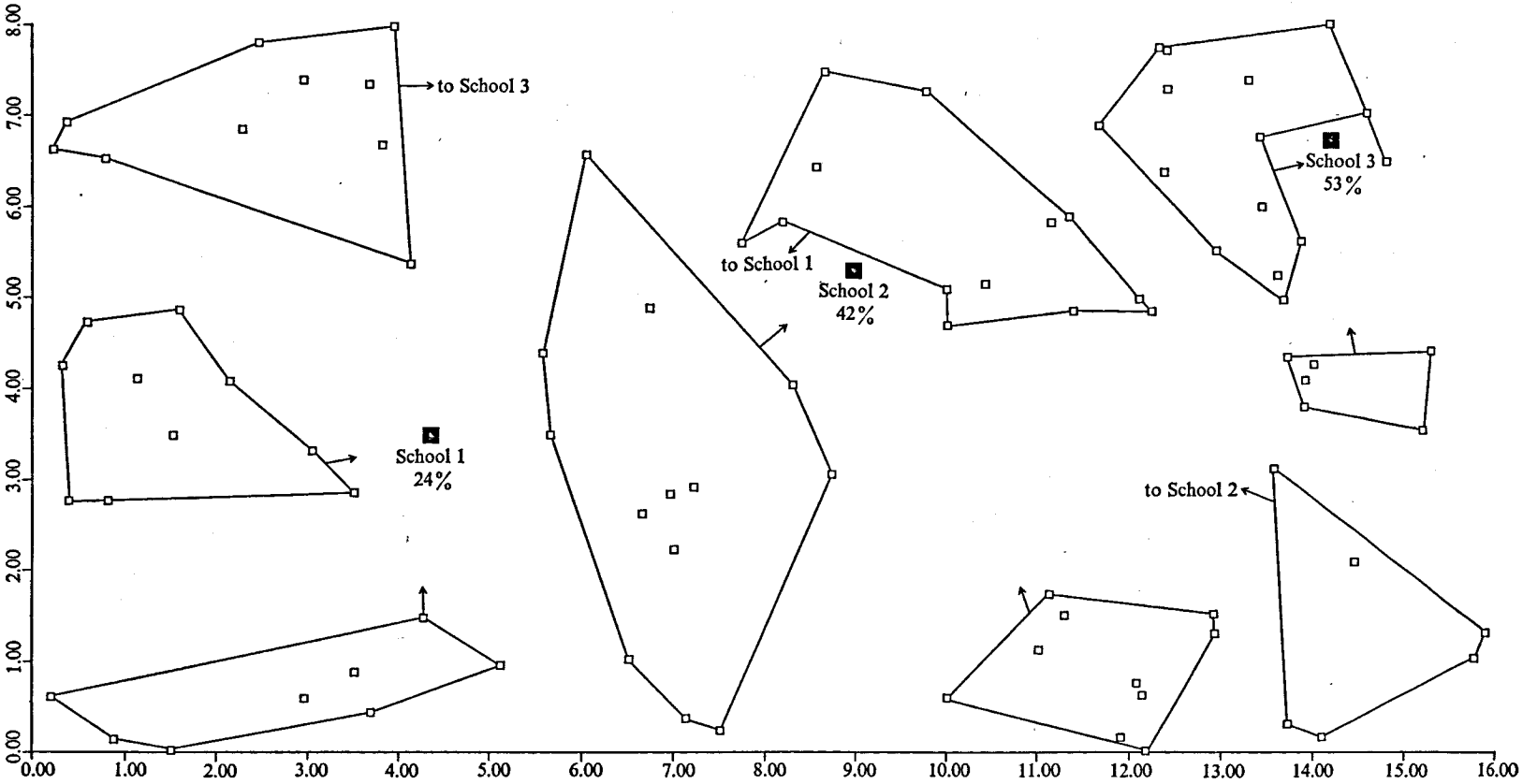


Figure 3. Modified initial estimate for problem 28. Similar to figure 2.

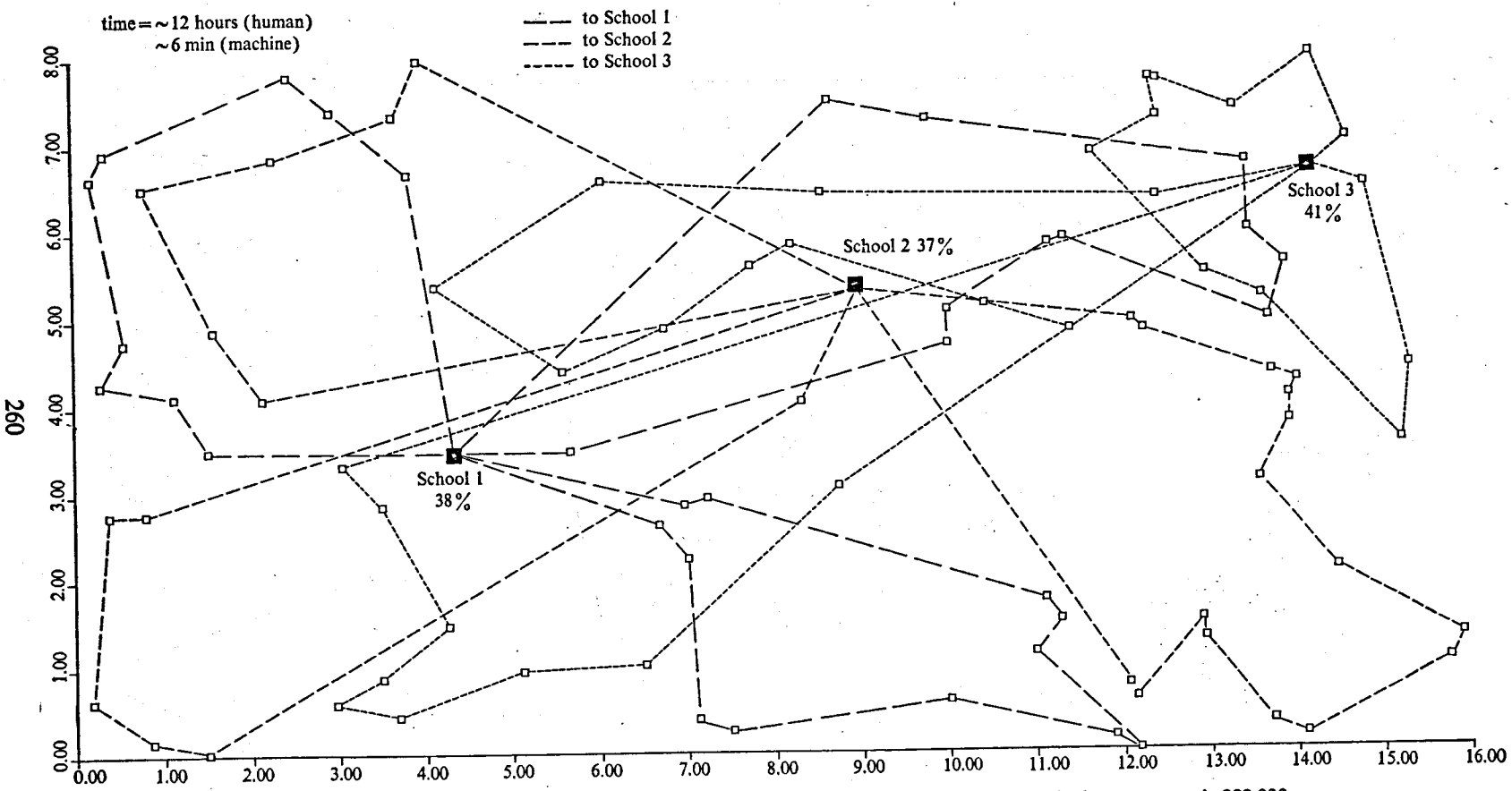


Figure 4. Best solution for problem 28. This is the final result of the man-machine phase. The cost is 1506.34. The longest route is 222.033.

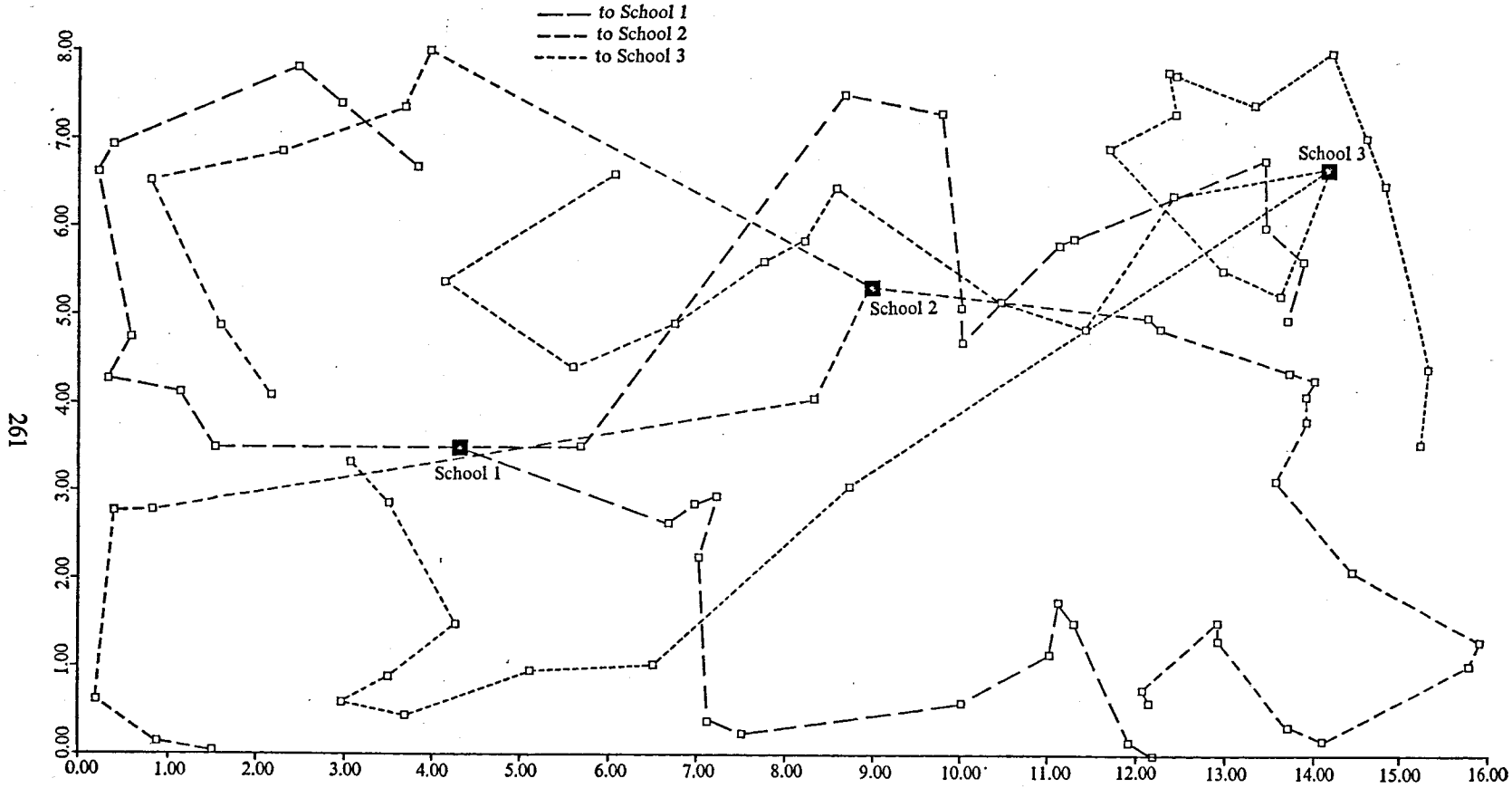


Figure 5. Zero-distance first link solution for problem 28. The routes of figure 4 were used, but the first link in each route (corresponding to the trip from the school to the first bus stop) was set to zero. The routes were then optimized according to the TS criterion. The cost is 1118.08, and the longest route is 162.5, corresponding to the longest pupil riding time. Figures 6, 7, and 8 are similar to figures 1, 2 and 4 respectively.

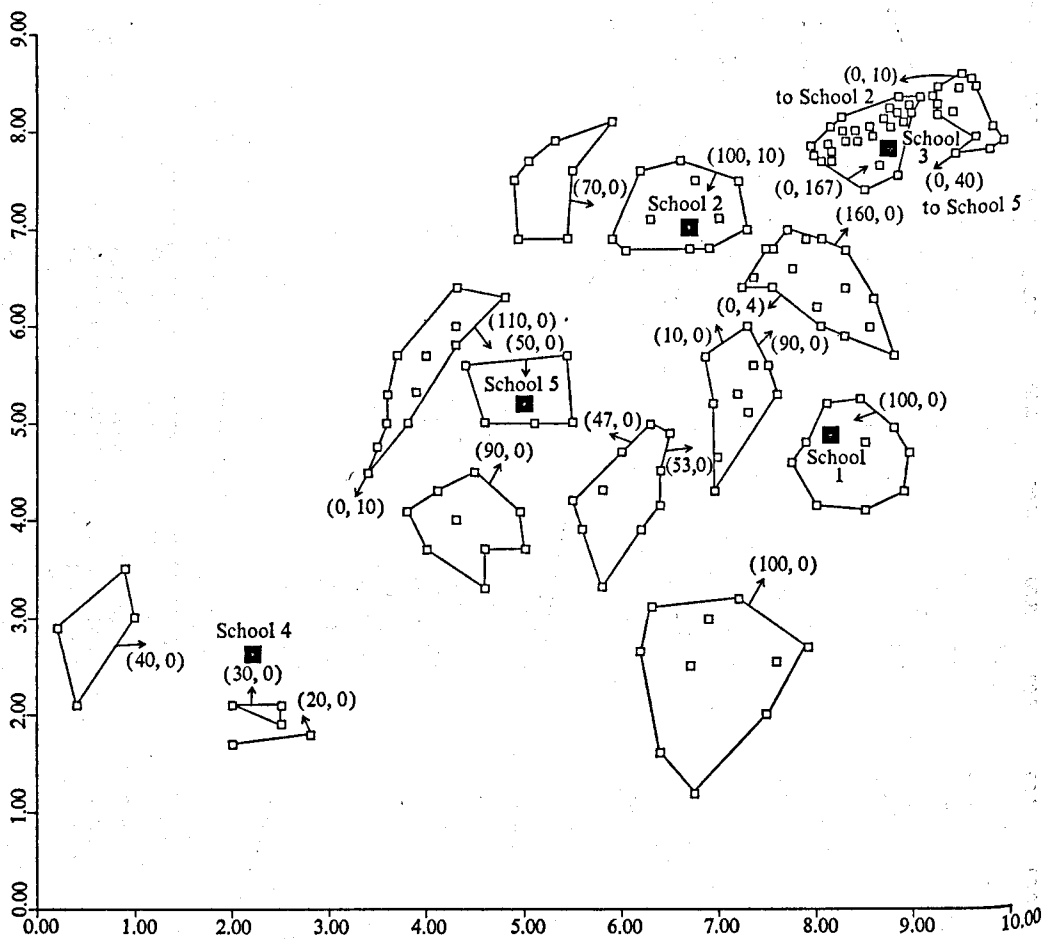


Figure 6. Linear problem solution for problem sw.

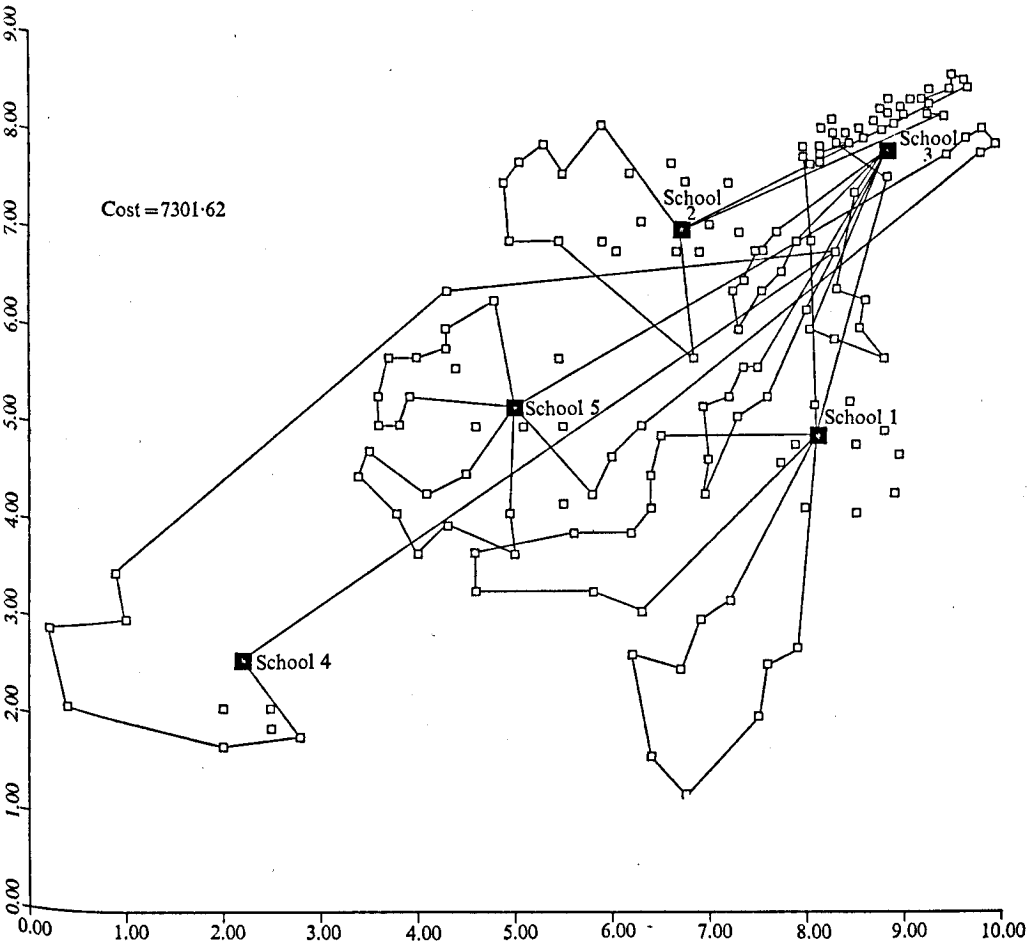


Figure 7. Initial routings for problem sw.

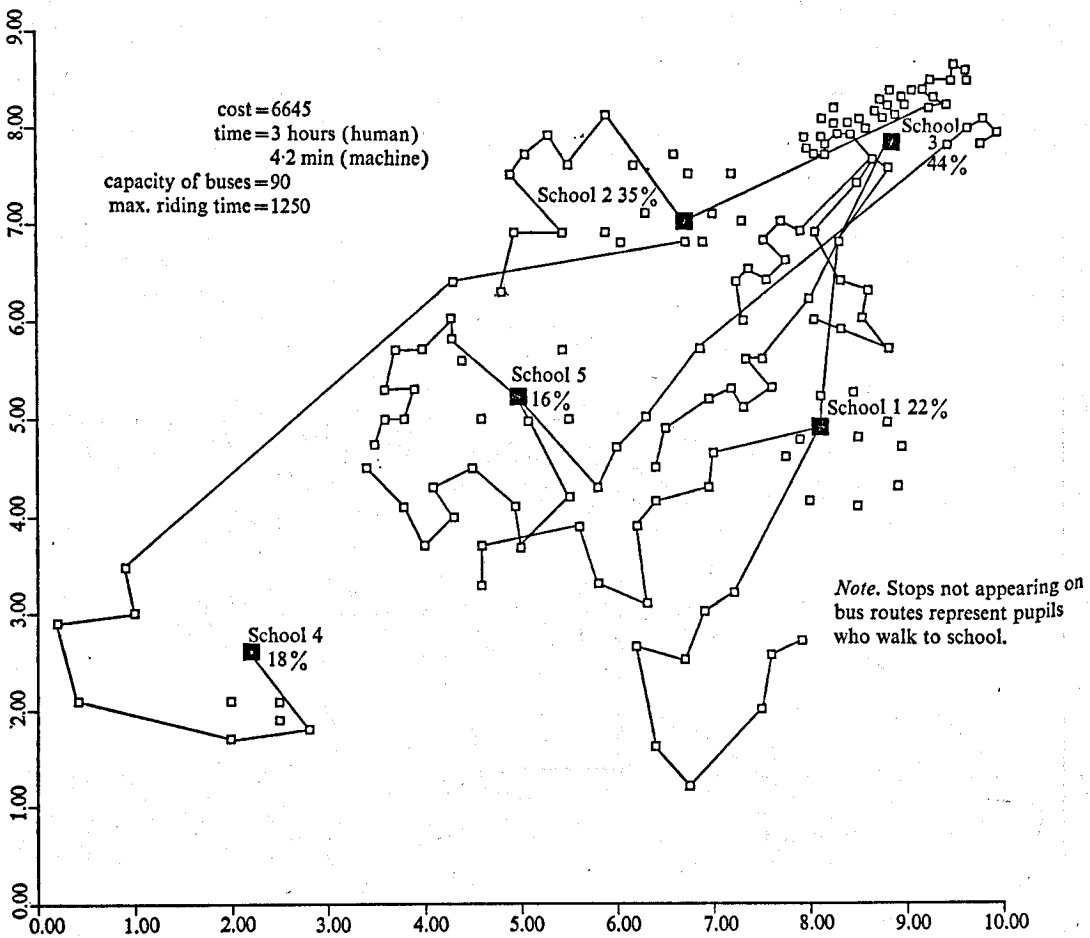


Figure 8. Best solution obtainable to date for problem sw.

REFERENCES

Alpert, D. & Bitzer, D.L. (1970) Advances in computer-based education. *Science*, **167**, 1582–90.

Bennington, B.J. (1969) Man-machine interaction in the design of rotating machines. *Proc. Design Automation Workshop*. Miami Beach, Florida.

Bodner, R.M., Cassell, E.A. & Andrew, P.J. (1970) Optimal routing of refuse collection vehicles. *Journal of the Sanitary Engineering-ASCE*, **96**, 893–904.

Brown, D.M. (1972) *Integration – A Systems Approach*. Working Paper, Dept. of Civil Engineering, Vanderbilt University, Nashville, Tennessee.

Christofides, N. & Eilon, S. (1969) An algorithm for the vehicle-dispatching problem. *Operational Research Quarterly*, **20**, 309–18.

Clarke, G. & Wright, J.W. (1963) Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research*, **11**, 568.

Clarke, S. & Surkis, J. (1968) An operations research approach to racial desegration of school systems. *Socio-Economic Planning Sciences*, **1**, 259–72.

Collins, J.S. (1964) The processing of lists and the recognition of patterns with applications to some electrical engineering systems. Ph.D. Dissertation, University of London.
Cooper, L. (1967) Solutions of generalized locational equilibrium models. *Journal of Regional Science*, **7**.
Dantzig, G.B. & Ramser, J.H. (1959) The truck dispatching problem. *Management Science*, **6**, 80–91.
Eastman, C.M. & Kortanck, K.O. (1970) Modeling school facility requirements in new communities. *Management Science*, **16**, B784–99.
Eastman, C.M. (1969) Cognitive processes and ill-defined problems: a case study from design. *Proc. Int. Joint Conf. on Art. Int.*, Washington DC.
Eastman, C.M. (1971) GSP: a system for computer assisted space planning. *Proc. Eighth Annual Design Automation Workshop*, Atlantic City, New Jersey.
Feigenbaum, E. & Feldman, J. (1963) *Computers and Thought*. New York: McGraw-Hill.
Felts, W.J. (1970) Solutions techniques for stationary and time varying traveling salesman problems. Ph.D. Dissertation, Vanderbilt University.
Gillett, B. & Miller, L. (1971) A heuristic algorithm for the vehicle dispatch problem. Presented at *39th National ORSA Meeting*, Dallas, Texas.
Gold, M. (1967) Methodology for evaluating time-shared computer usage. Ph.D. Dissertation, Alfred P. Sloan School of Management, MIT.
Hayes, R.L. (1967) The delivery problem. Ph.D. Dissertation, Carnegie Institute of Technology.
Helms, B.P. & Clark, R.M. (1971) Selecting Solid Waste Disposal Facilities. *Journal of the Sanitary Engineering-ASCE*, **97**, 443–51.
Johnson, S.C. (1965) Hierarchial clustering schemes. *Psychometricka*, **32**, 241–54.
Kilgour, A.C. (1971) Computer graphics applied to computer-aided design. *Comput. Bull.*, 18–23.
Kingston, P.L. (1970) School bus routing via computer – a case study. Presented at *37th National ORSA Meeting*, Washington, DC.
Kraft, J. (1972) Integration without mass busing. *Chicago Daily News*.
Krolak, P.D., Felts, W.J. & Marble, G. (1971) A man-machine approach toward solving the traveling salesman problem. *Communications of the ACM*, **14**, 5, 327–34
Krolak, P.D., Felts, W.J. & Nelson, J. (1971) A man-machine approach toward solving various routing, scheduling and network problems. *Proc. Art. Int.* AGARD. Rome.
Krolak, P.D., Felts, W.J. & Nelson, J. (1972) A man-machine approach for solving the generalized truck dispatching problem. *Transportation Science*, **6**, 2, 149–70.
Lin, S. & Kernighan, B.W. (1972) A heuristic algorithm for the traveling salesman problem. Presented at *41st National ORSA Meeting*, New Orleans, La.
McCloy, J.P. (1972) Allocation of parks and recreation facilities by computer. *Industrial School Management Report No. 704*. Georgia Institute of Technology, Atlanta, Georgia.
Michie, D., Fleming, J.G. & Oldfield, J.V. (1968) A comparison of heuristic, interactive, and unaided methods of solving a shortest-route problem. *Machine Intelligence 3*, pp. 245–55 (ed. Michie, D.). Edinburgh: Edinburgh University Press.
Neidell, L. (1969) The use of non-metric multidimensional scaling in market analysis. *Journal of Marketing*, **32**, 37–43.
Nelson, J.H. & Krolak, P.D (1972) A man-machine approach for providing equal educational opportunity in urban school systems. Presented at *41st National ORSA Meeting*, New Orleans, La.
Newell, A. & Simon, H. (1972) *Human Problem Solving*. Englewood, N.J.: Prentice-Hall.
Newton, R.M. (1970) Bus routing in a multi-school system. Ph.D. Dissertation, S.U.N.Y. at Buffalo, New York.

Oberuc, R.E. (1968) A practical algorithm for finding solutions to the traveling salesman problem. Presented at *34th National ORSA Meeting.*

Pugh, G.E. (1972) School desegregation with minimum busing. *Lambda Corporation Report.* Arlington, Va.

Rapp, M.H., Goldman, M. & Smallwood, R. (1971) The interactive graphic transit simulator, a tool for planning node-oriented transit systems. *Research Report 7,* Dept. of Civil Engineering, University of Washington, Seattle, Washington.

Rapp, M.H. (1972) Man-machine interactive transit systems planning. *Socio-Economic Planning Science,* **6,** 95–123.

REDIST (1969) *Program Description and Users Manual.* National Municipal League, New York.

ReVelle, C. & Swain, R. (1970) Central facilities location. *Geographical Analysis,* **2,** 30–42.

Sackman, H. (1970) *Man-Computer Problem Solving.* Princeton, N. J.: Auerbach Publishing Co. Princeton, N.J.

Schneider, J.B. (1971) Solving urban location problems: human intuition versus the computer. *Journal of the American Institute of Planners,* **32,** no. 2, 95–9.

Sokol, R. R. & Sneath, P. (1963) *Principles of Numerical Taxonomy.* San Francisco: Freeman and Co.

Steinberg, D.I. (1972) An application of linear programming to legislature redistricting problems. Presented at *41st National ORSA Meeting,* New Orleans, La.

Symons, J.G. (1969) Locating emergency medical service vehicle dispatch centers in a metropolitan area. M.Sc. Thesis, University of Washington, Seattle.

System 360 vehicle scheduling program application description. *IBM Technical Publication No. H20–0464–0,* White Plains, New York, 1967.

Tanaka, M. (1972) A fixed charge problem for the selection of refuse collection schedules. *Proc. of the Midwest AIDS Conf.,* pp. G15–20. Cincinnati, Ohio.

Tillman, F.A. (1969) The multiple terminal delivery problem with probabilistic demands. *Transportation Science,* **3,** 192.

Willard, P.E. (1970) Routing of ships collecting seismic data. Unpublished paper from *7th Mathematical Programming Symposium,* The Hague.

14

Heuristic Theory Formation: Data Interpretation and Rule Formation

B. G. Buchanan, E. A. Feigenbaum and N. S. Sridharan
Computer Science Department
Stanford University

I. INTRODUCTION

Describing scientific theory formation as an information-processing problem suggests breaking the problem into subproblems and searching solution spaces for plausible items in the theory. A computer program called meta-DENDRAL embodies this approach to the theory formation problem within a specific area of science.

Scientific theories are judged partly on how well they explain the observed data, how general their rules are, and how well they are able to predict new events. The meta-DENDRAL program attempts to use these criteria, and more, as guides to formulating acceptable theories. The problem for the program is to discover conditional rules of the form $S \rightarrow A$, where the S's are descriptions of situations and the A's are descriptions of actions. The rule is interpreted simply as 'When the situation S occurs, action A occurs'.

The theory formation program first generates plausible A's for theory sentences, then for each A it generates plausible S's. At the end it must integrate the candidate rules with each other and with existing theory. In this paper we are concerned only with the first two tasks: data interpretation (generating plausible A's) and rule formation (generating plausible S's for each A).

This paper describes the space of actions (A's), the space of situations (S's) and the criteria of plausibility for both. This requires mentioning some details of the chemical task since the generators and the plausibility criteria gain their effectiveness from knowledge of the task.

The theory formation task

As in the past, we prefer to develop our ideas in the context of a specific task area. Thus the computer program under development works with data from organic mass spectrometry, a subset of analytic chemistry, to infer rules for

the theory of mass spectrometry. The data, from which the program will form rules, are a collection of structural descriptions of chemical molecules – each paired with the analytic data produced by the molecule in an instrument known as a mass spectrometer. The analytic data are usually presented in a fragment-mass table (FMT), or mass spectrum – a table of masses of molecular fragments and their relative abundances.

The program is given some knowledge of the task area. In particular, it is given the ability to manipulate topological descriptions of molecules; it is given primitive terms from which descriptions of causal explanations (of the form $S \rightarrow A$) can be written. But it is not given the predictive rules describing the operation of the mass spectrometer: these are essentially the rules it must discover.

As the discussion develops, we hope to make explicit exactly what knowledge is given to the program. We strongly believe that the performance of artificial intelligence programs increases as the knowledge available to them increases. So we have deliberately given it knowledge of the task area. On the other hand, giving it too much knowledge removes the difficulty (and the interest) from the theory formation problem.

Special purpose program

In order to try out some of our ideas on theory formation quickly, we decided to implement a rather special-purpose program before writing the general theory formation program discussed previously (Buchanan, Feigenbaum and Lederberg 1971). This strategy has three main advantages.

(1) Communication between an expert and a novice seems to be more fruitful the more specific the problem under discussion. At least for purposes of discovering the expert's heuristics and asking him to judge the acceptability of results, this is so.

(2) Putting more knowledge of the task area into the program avoids many of the initial clumsy errors of general-purpose programs. This allows us to work on some of the main problems of theory formation before having to construct an elaborate program substructure.

(3) Working on a specific problem also allows us to produce intermediate results of real value to the practising scientist.

Specifically, we have limited our attention to the class of molecules known as estrogenic steroids, well known for their use in oral contraceptives. The molecules from which rules are constructed are all assumed to be of the same class, with a common skeleton shown in figure 1. Instances of this class have additional atoms (substituents) bonded to the skeleton, replacing hydrogen atoms. One instance is shown in figure 2. Although we believe the program is not limited to just this class, we have no way of knowing this until we attempt rule formation for other classes. See pp. 275–80.

2. CONCEPTUAL FRAMEWORK

The program searches a space of theory sentences (or rules) to construct a collection of sentences explaining the experimental data. The nature of this space is described briefly in this section. Appendixes 1 and 2 define the concepts mentioned here more precisely.

Space of theories

We count as a theory a set of predictive rules in conditional form: $S \rightarrow A$. Although a mere collection of rules often does not warrant the laudatory term 'theory', for purposes of defining the task this simple definition will serve. Exploring the space of theories, then, is exploring the space of rule sets where each rule is of the $S \rightarrow A$ form. The S's and A's, in turn, are members of their own separate spaces, so the space of rules is the cross product of the space of S's and the space of A's.

As mentioned above, plausible and interesting actions (A's) are generated first. This is described in detail in the section on data interpretation. Then for each A, plausible and interesting situations (S's) are generated to form alternative rules of the $S \rightarrow A$ form. The program keeps those rules which explain the sample data most simply and completely, and which still allow extrapolation to unobserved data.

The predictive nature of the theory partially dictates the form, as do the other restrictions we initially place on the kinds of theories we want the program to formulate. Appendix 1 characterizes the space of theories more completely.

We are focusing our attention on theories that will help us predict FMT's (mass spectra) for molecules given their chemical structure. In Appendix 1 we begin by defining what is a predictive theory of mass spectrometry (Appendix 1, Part 1). We indicate that the theory is constructed in several stages. First, the data points in the FMT for each molecule are explained as fully as possible. The general rules formulated must be consistent with all of these individual explanations. We restrict the form of these individual explanations by restricting the terms from which explanatory actions can be described. This leads to the idea of an action-based theory (Appendix 1, Part 2). Also for simplicity, the program is restricted to predicting occurrence and non-occurrence of fragment-mass points, without estimating intensities (y-coordinates) for those data points. This kind of theory we call a 0, 1 theory (Appendix 1, Part 3). The samples we study represent not the entire space of molecules, but rather a restricted class of molecules and thus we introduce the concept of partial theories (Appendix 1, Part 4). The range of validity and applicability of a partial theory is only the space formed by a restricted class of molecules. For each molecule in its range a partial theory may leave some of the peaks in the data unexplained.

Space of actions

The search space is generated from the two primitive 'legal moves': bond cleavage and group migration. Plausible actions, or 'moves', are generated by imposing heuristic constraints on the legal move generator. It should be noted that the plausible actions are generated only once for the basic skeleton common to all molecules. For each molecule, the program records the evidence (or lack of it) for each of the actions. A few precise definitions will help at this point:

(a) Action = Cleavage followed by Fragment Selection followed by Migration *or* a set of cleavages followed by fragment selection followed by migration.

(b) Cleavage = The process of removing all the bonds in a cleavage set.

(c) Cleavage Set = A set of bonds in the molecule or fragment (edges of a graph) which are necessary and jointly sufficient for separating the molecule into two distinct parts. Bonds to hydrogen atoms are excluded.

(d) Fragment Selection = the identification of one of the parts of a graph resulting from cleavage.

(e) Migration = The process of moving a specified group of atoms from one part of the molecule or fragment to another. (A null group is allowed.)

The generation of actions can be simply described now by the following steps.

1. Generate all plausible cleavages (that is, all plausible cleavage sets). (See table 1.)
2. Select each of the two fragments.
3. Specify all plausible actions containing one cleavage followed by migration.
4. Specify all plausible actions containing two cleavages followed by migration.
5. Specify all plausible actions with n cleavages followed by migration. (See table 3.)

The constraints on the generator which determine plausibility of cleavages and of actions are of three kinds, listed below. Table 3 shows the plausible ways of putting actions together from the possible cleavage sets (followed by fragment selection and migration) shown in table 1.

1. *Topological Constraint*

For bond cleavage, consider only sets of bonds which separate the molecular graph into two pieces. These are called cleavage sets.

2. *Chemical Constraints*

(a) Do not break a set of bonds together if
  (i) any bonds are in an aromatic ring,
  (ii) two of the bonds are attached to the same carbon atom,
  (iii) any bonds (except ring junctures) are other than single bonds.

(b) Select fragments with k or more carbon atoms only. (Currently $k=6$.)

(c) Do not allow an additional cleavage in an n-ary action if:
  (i) the $n+1$ cleavage sets are not disjoint (some bonds are duplicated);

(ii) there is not sufficient evidence for all $n+1$ cleavages (where 'sufficient evidence' may be defined in various ways). For table 1 this heuristic was not used.

(d) Transfer from -2 to $+2$ hydrogens (only) after a set of cleavages (but not in the middle of a set); that is, limit migration to hydrogen atoms, and allow only -2, -1, 0, 1, or 2 atoms to migrate. (Negative integers specify numbers of atoms lost from the fragment, positive integers specify numbers of atoms gained. Zero indicates no migration, or migrating a null group, if one prefers.)

3. *Methodological Constraints*

(a) If two sets of actions explain the same data point, ignore the more complex set (Occam's Razor).

(b) Confine cleavages to bonds within the skeleton.

Space of situations

A single situation is a description of a class of molecules, so the space of all possible *S*'s is the space of all class descriptions. Some of the terms used in defining the space are explained below.

A *class description* is a Boolean combination of features.

A *feature* is a substituent label followed by a position number. This definition of 'feature' holds only for the special purpose program which assumes a common skeleton with numbered positions for placing additional groups of atoms. In general, features can be described in terms of subgraphs.

A *substituent label* is an arbitrary label describing an atom or group of atoms (substituent) attached to the structural skeleton.

A *position number* is one of the numbers assigned to the nodes of the structural skeleton.

In principle, the generator of situations is quite simple. It must be severely constrained, however, in order to limit the number of *S*'s actually considered. Two main kinds of constraints are considered: constraints on features and constraints on combinations of features.

1. *Constraint on Features*

Consider only features which appear in the sample; that is, consider features which mention only substituent labels and position numbers appearing together in some molecule of the sample.

2. *Constraints on Boolean Combinations of Features*

(a) Allow union and intersection operators only.

(b) Allow only one occurrence of each position number in a combination of features.

(c) Allow only combinations which cover the molecules in the sample; that is, every positive instance in the sample is an instance of the class defined by the combination, or every negative instance is.

(d) Allow only combinations which are consistent with the data; that is,

the class defining positive instances does not also cover known negative instances, and vice versa.
(e) Generate only parsimonious combinations of features; that is, limit the number of terms and the number of Boolean operators.

3. DETAILS OF THE DATA INTERPRETATION PROGRAM

Data interpretation is an essential aspect of theory formation for at least two reasons:
(1) At this early stage, a large volume of experimental data can be compressed into readable results tables. In mass spectrometry, data reduction takes place in many stages. The so-called 'raw data' are initially represented as a continuous graph of detector voltage plotted against time. A sequence of numerical algorithms reduce the data to:
(a) a digital (bar) graph of integrated detector voltages plotted against time of recording the peak centers.
(b) a digital (bar) graph of integrated detector voltages plotted against atomic mass units of the peak centers.
(c) a digital plot of integrated detector voltages *vs.* elemental compositions of fragments (each elemental composition derived from exact mass). Step (c) can be applied to so-called 'high resolution' data, that is, data with precise resolution of the masses accurate to 3 or 4 decimal places.
(d) (c) with detector voltages normalized.
(e) (d) with isotopic contributions deleted.

All of these reductions are made for the routine analysis of mass spectra using the existing corpus of knowledge. Therefore, it is reasonable to start with data already reduced in these ways when attempting to extend the corpus of knowledge. This is especially so because extensions to the corpus will be made using only the primitive concepts which are currently used.
(2) Here, also, the data are re-represented from the form of experimental results suitable to the experiment to the form of results suitable to the theory. These are not separate reasons since the way of organizing the compressed table of results depends upon the conceptualization of the problem for form and content.

The form of the rules to be written is dictated by the form of existing rules. The computer representation for the existing rules has been described elsewhere (Buchanan and Lederberg 1971, Buchanan, Feigenbaum and Lederberg 1971). In short, a rule describing the behavior of a class of molecules in a mass spectrometer is a conditional of the form $S \rightarrow A$ where S is a description of features of molecules of the class (a 'cause') and A is a set of names of processes ('effects') which occur in the instrument. The result of applying a rule is another molecule or molecular fragment which, itself, can be processed by other conditional rules. Because this simple rule form has been easy to use and easy to modify, the program which creates new rules will create them in this form.

Transformation

The data are presented to the program as effects of unnamed actions (peaks in the FMT) for each molecule. Thus, it is necessary to infer the nature of the underlying actions if we want to put the actions into the consequent places of conditional rules. For example, the FMT of the estrogenic steroid esterone, shown in table 2, shows peaks at approximately 70 mass points (above mass 65), each of which can be identified with an elemental composition. Because of small mass differences between some groups of atoms, it is occasionally impossible to determine a unique elemental composition for a mass point within the resolving power of the instrument. In that case, the peak is identified with both (or all) possible compositions. In order to form rules explaining why those peaks appear in the data for estrone, it is necessary to transform each peak in the data to a set of possible actions which are potential explanations of the peak.

Our bias toward heuristic search has strongly influenced our choice of strategies. Generating the plausible actions and pruning with respect to the data is a prime example of heuristically searching the whole space of explanations. The space and the constraints used in generation of actions were described in section 2.

The final result of transformation is a list of peak-action set pairs for each molecule. Peaks or actions which are not a member of some pair are eliminated from further consideration in the theory formation process. Since the list of possible actions has been pruned before attempting to explain the peaks in the data, it is possible that some peaks will be unexplained. These peaks are reported by the program to provide a method of checking the validity of the pruning process. The actions which occur together in a process set as the explanation for a particular peak are redundant explanations for that peak. Table 3 shows the results for the one molecule we have been using for an example. A similar table is given for each molecule in the initial collection of molecules with which the theory formation program works.

Reorganization

The rule formation program must be able to view each action separately to determine the situation in which it is likely to occur. It must have information about the molecules for which the action does and does not occur. Therefore it is necessary to reorganize the information collected so far in this simple way.

Table 4 shows each action for which any evidence was found in the data of any of the molecules. The example molecule we have been following, estrone, is molecule number 1 in this table. As can be seen there estrone (and other molecules) shows many, but not all, of the actions in the table.

Of particular interest in the reorganization of results is the handling of redundant actions. Each action which could have been responsible for a peak in the data is given credit for that peak, and the ambiguity of the peak

(in the form of a list of the other actions which could have explained the peak) is noted. The resulting set of data is a list of actions in which each action is supported by evidence in one or more molecules. Each molecule gives the peak intensity for this action and a list of any other actions identified as being redundant explanations in this molecule. Our desire for compact presentation of results provides the rationale for a final step of reorganization, which is the compression of redundant actions into a single group of actions supported by the same evidence.

4. CHARACTERIZATION OF THE RULE FORMATION PROBLEM

The space of situations described in section 2 gives a rough idea of the nature of the rule formation problem. In Appendix 2 the rule formation problem is posed in more detail. The class of estrogens is defined thus formulating the space of molecules over which theories are to be constructed. The form of input data is presented (Appendix 2, Part A1). We present a series of problem reductions and several steps in the solution of the problem. The first step in the solution is the generation of explanatory hypotheses for the data. This is done by generating possible actions which explain the data and pruning with respect to *a priori* criteria of plausibility and *a posteriori* evidence in the data. Details of the program have been described above. An important reorganization of the data concerns leaving the explanatory mode, listing the behavior of each *molecule* individually, and approaching a generalizing mode by describing each *action* individually. In the course of doing this, the program groups together sets of ambiguous actions that are not distinguishable by the data (Appendix 2, Part A3).

In seeking to introduce generality into the explanations we ignore actions that are specific to single molecules or to a very small number of molecules in the sample. In order that we include only actions that can reasonably be applied not only to the molecules in the data, but also to any of the estrogens, we restrict our attention to skeletal actions. Section B, which develops the theory of skeletal fragmentation, begins with the definition of skeletal actions (Appendix 2, Part B1) and introduces a simple representation for estrogenic molecules using structural coding of substituents around sites. The vocabulary for specifying arbitrary subclasses of estrogenic molecules represented in the data is defined next. The primitive classes (Appendix 2, Part B3) and the non-primitive classes (Appendix 2, Part B4) are defined in a natural way. This vocabulary allows the introduction of requisite generality into the description of actions and specifying the classes of molecules that show evidence for each action. Even at this stage we have captured some explanatory power and generality into the rules that are constructed. The generality introduces predictive power to a small degree. We postpone discussion of enhancing the predictive range of rules to a later section (D) and deal with problem reduction in section C.

Because of the assumption that actions are independent of one another we do not have to treat any set of actions simultaneously (Appendix 2, Part C1). The problem then decomposes into several independent subproblems with accompanying reductions in problem size (Appendix 2, Part C2). Construction of rules then proceeds in the context of single actions. Considerations are delineated such as, which among all the non-primitive classes are suitable for associating with the occurrence of an action (Appendix 2, Part C3). The concepts of cover and honesty to data are important in this context. The space of all rules is the product space formed by taking actions from the space of actions and associating every non-primitive class of molecules with each action. The space of rules is very large. We need good pruning conditions for filtering the rules and evaluation criteria for selecting from those that remain (Appendix 2, Part C5).

The rules then need to be systematically recoded to be made applicable to a wide range of molecules in the estrogen class, rather than to narrow classes defined using substituents occurring in the sample. This enhancement of predictive power is explained in section D. The technique of term replacement is the topic in D1, and interim computer implementation is described in D2. The program Planner (Appendix 2, Part D3) is given the mandate of providing the rule formation program the information it needs to develop rules amenable to term replacement.

The rules collected for different actions now have attributes of explanation, generality and predictive power. These rules have to be codified and combined with the existing corpus of knowledge. But discussion of this aspect of theory formation is beyond the scope of this paper.

FIGURES AND TABLES

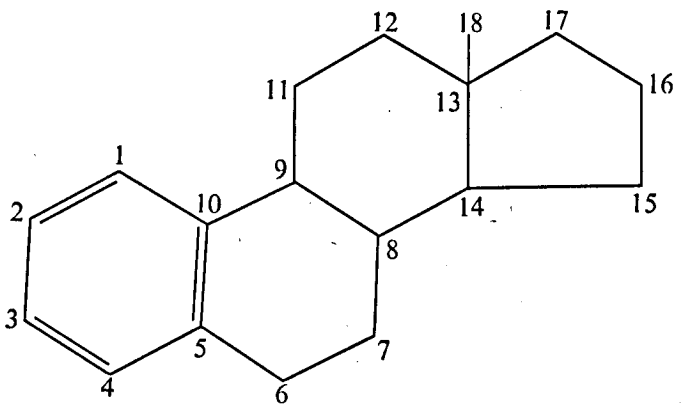


Figure 1. Estrogen skeleton $C_{18}H_{24}$ (hydrogen atoms not shown). All nodes are carbon atoms with the proper number of hydrogen attached. (Proper number = valence of atom minus number of bonds attached, e.g., 4–2 for node 15, or CH2.) The conventional numbering of the nodes in the estrogen skeleton is shown.

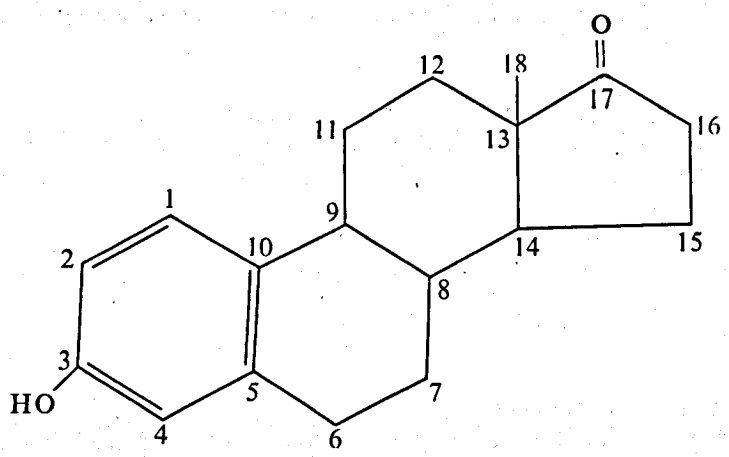


Figure 2. Structural features of estrone ((OH3) (O=17)). The two substituents are OH on node 3 and O= on node 17.

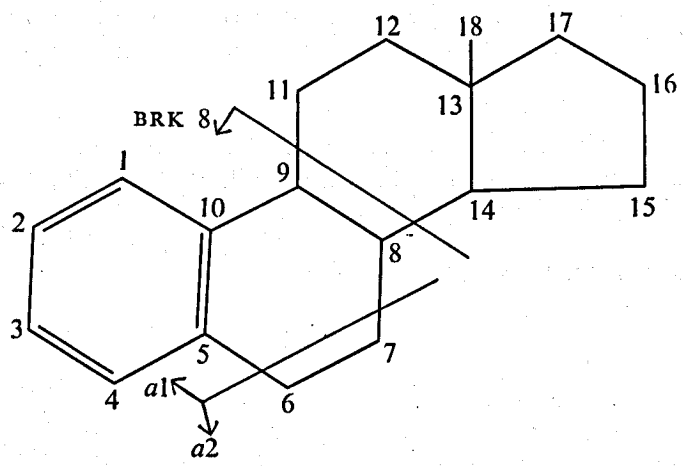


Figure 3. Example of actions in the estrogen skeleton. The arrow indicates fragment selection, the long diagonal line indicates cleavage. No migration occurs in *a*1 or *a*2.

Diagram 1. Conceptualization of meta-DENDRAL.*

[DATA (Analytic Data plus structural features for each of *n* molecules)]

↓

Data Interpretation: Generate actions to explain data points

↓

[Table of Plausible Actions]

↓

Rule Formation: Search for plausible explanations for an action *Ai*

↓

Alternative explanations for each action

$S1 \rightarrow A1 \quad S1 \rightarrow A2 \quad S1 \rightarrow Ak$

$S2 \rightarrow A1 \quad S2 \rightarrow A2 \ldots S2 \rightarrow Ak$

$Sn \rightarrow A1 \quad Sm \rightarrow A2 \quad Sj \rightarrow Ak$

* Unification of rules, the third step of the procedure, has been omitted from this discussion, and consequently from this diagram. The output of the second step, Rule Formation, is a set of rules which is a primitive sort of theory in its own right. But we recognize the convention of withholding the term 'theory' from a 'mere' collection of rules.

Table 1. Some possible cleavage sets for the estrone skeleton generated under constraints. Constraints used for generating the cleavage sets are:
(1) Topological constraint
(2) Chemical constraints (a–i), (a–iii), (b), (d)
(3) Methodological constraints (a), (b).

Cleavage label	Cleavage set
BREAK 1	((13 18))
BREAK 2	((5 6) (6 7))
BREAK 3	((5 6) (7 8))
BREAK 4	((5 6) (9 10))
BREAK 5	((6 7) (7 8))
BREAK 6	((6 7) (9 10))
BREAK 7	((7 8) (9 10))
BREAK 8	((8 14) (9 11))
.	.
.	.
.	.
BREAK 33	((8 9) (9 10) (9 11))
BREAK 34	((8 9) (9 10) (11 12))
BREAK 35	((8 9) (9 10) (12 13))
BREAK 36	((8 14) (13 14) (13 17))
.	.
.	.
.	.
BREAK 65	((8 9) (9 10) (13 14) (14 15))
BREAK 66	((8 9) (9 10) (13 14) (15 16))
BREAK 67	((8 9) (9 10) (13 14) (16 17))

Table 2. Analytic data (FMT) for estrone.

exact mass of fragment	Intensity	Composition of fragment C	H	N	O	P	S
(6.50408085E01	1.10000	5	5	0	0	0	0)
(6.70558381E01	2.00000	5	7	0	0	0	0)
(7.70389282E01	3.00000	6	5	0	0	0	0)
(7.90540497E01	2.70000	6	7	0	0	0	0)
(8.10704498E01	1.60000	6	9	0	0	0	0)
(9.10545516E01	5.20000	7	7	0	0	0	0)
(9.20427483E01	1.20000	3	8	0	3	0	0)
(9.30702147E01	1.60000	7	9	0	0	0	0)
(9.50863121E01	1.20000	7	11	0	0	0	0)
(9.70652752E01	3.00000	6	9	0	1	0	0)
(1.03054406E02	1.20000	8	7	0	0	0	0)
(1.05069888E02	2.30000	8	9	0	0	0	0)
(1.07048971E02	3.40000	7	7	0	1	0	0)
(1.07085457E02	1.30000	8	11	0	0	0	0)
(1.08056890E02	1.10000	7	8	0	1	0	0)
(1.15054584E02	4.30000	9	7	0	0	0	0)
(1.16062028E02	1.80000	9	8	0	0	0	0)
(1.17069416E02	1.30000	9	9	0	0	0	0)
(1.19086545E02	1.10000	9	11	0	0	0	0)
.	.			.			
.	.			.			
.	.			.			
(2.13127281E02	17.2000	15	17	0	1	0	0)
(2.14133907E02	8.30000	15	18	0	1	0	0)
(2.15137354E02	1.50000	15	19	0	1	0	0)
(2.15137354E02	1.50000	11	19	0	4	0	0)
(2.26136093E02	5.20000	16	18	0	1	0	0)
(2.27142082E02	1.40000	16	19	0	1	0	0)
(2.37128383E02	2.10000	17	17	0	1	0	0)
(2.42128541E02	3.70000	16	18	0	2	0	0)
(2.55133098E02	1.20000	17	19	0	2	0	0)
(2.68147613E02	1.30000	18	20	0	2	0	0)
(2.69155298E02	2.50000	18	21	0	2	0	0)
(2.70164935E02	100.000	18	22	0	2	0	0)
(2.71168209E02	18.5000	18	23	0	2	0	0)
(2.72172412E02	1.90000	18	24	0	2	0	0)

Table 3. Some plausible estrone actions inferred from data.

Data points: Composition of fragment	Intensity*	Full action labels†
C18H2402	.5	BRKO:H2
C18H2302	.2	BRKO:H1
C18H2202	26.1	BRKO
C18H2102	.6	BRKO:H-1
C18H2002	.3	BRKO:H-2
C17H1902	.3	BRK1L
C16H1901	.1	BRK15L:H-1
C16H1802	1.0	BRK3L BRK12L BRK18L
C16H1801	1.4	BRK15L:H-2
C15H1901	.2	BRK14L:H1
C15H1801	1.5	BRK14L
.	.	.
.	.	.
.	.	.
C12H1301	.3	BRK10L:H-1 BRK42L BRK3L/47L
C12H1201	5.7	BRK10L:H-2 BRK42L:H-1 BRK3L/47L:H-1
C12H1101	1.0	BRK42L:H-2 BRK3L/47L:H-2
C11H1301	.1	BRK1L/4H:H-2 BRK9L:H1 BRK3L/43L:H2 BRK3L/46L:H2
C11H1201	2.9	BRK9L BRK3L/43L:H1 BRK3L/46L:H1

* Most of the information in this column is currently ignored since the program looks only for the presence or absence of evidence. We expect to make better use of this information, but for the present we do not.

† The action names in this column indicate the cleavages, fragment selection, and migration involved in the action. More than one action name associated with a fragment composition indicates that those actions produce indistinguishable fragments; that is, the actions are redundant explanations of the same data point. The notation for full action labels is:

full action label=BRK ⟨*action label*⟩

action label =⟨*cleavage label*⟩⟨*fragment selection label*⟩
or ⟨*cleavage label*⟩⟨*fragment selection label*⟩⟨*migration indicator*⟩
or ⟨*action label*⟩|⟨*action label*⟩

fragment selection label=L or H
(L indicates the fragment containing the lower-numbered node in the first bond of the cleavage set, H indicates the higher-numbered node.)

migration indicator=: *H* ⟨*num*⟩
(If the migration indicator is absent, no hydrogens are transferred in the action. If *num* is a negative number, that number of hydrogens migrate out of the fragment. If *num* is positive, that number of hydrogen atoms migrate into the fragment.)

Table 4. Partial list of processes for which evidence was found in the data for any of four estrogens.

Full action label*	Molecules which show action† fraction	ID	int.	Partial redundancies‡
ERKO:H2	2/4	4	.6	
		1	.5	
BRKO:H1	2/4	4	.5	
		1	.2	
BRKO	4/4	1	26.1	
		2	26.0	
		3	20.5	
		4	17.4	
BRKO:H-1	3/4	3	.7	
		1	.6	
		4	.3	
BRKO:H-2	4/4	2	.7	
		4	.7	
		1	.3	
		3	.1	
BRK1L:H1	2/4	3	.9	
		4	.2	
BRK1L	3/4	4	.6	
		1	.3	
		3	.3	
BRK3L	3/4	4	3.5	BRK1L/18L:H1
		1	1.0	BRK12L
		3	.6	BRK12L
BRK3L:H-1 BRK12L:H-1 BRK18L:H-1	1/4	3	.2	

* See explanation from table 3. The actions are clustered by groups of fully redundant actions, that is, actions which explain the same data point for all of the molecules showing the action.

† molecule 1=estrone.

‡ Partially redundant actions explain the same data point for one molecule, namely, the one whose number appears in the third column.

REFERENCES

Buchanan, B.G. & Lederberg, J. (1971) The heuristic DENDRAL program for explaining empirical data. *Proc. IFIP Congress 71*, Ljubljana, Yugoslavia. (Also *A.I. Memo 141*, Stanford Artificial Intelligence Project. California: Stanford University.)

Buchanan, B.G., Feigenbaum, E.A. & Lederberg, J. (1971) A heuristic programming study of theory formation in science. *Proc. Second Int. Joint Conf. on Art. Int.*, Imperial College, London. (Also *A.I. Memo 145*, Stanford Artificial Intelligence Project. California: Stanford University.)

APPENDIX 1. DESCRIPTION OF THE THEORY SPACE

1. A *Predictive Theory of Mass Spectrometry* is a total mapping from
   molecules to fragment mass tables

where a molecule involves a description of its chemical structure; and the FMT in a simple form is a list of pairs of the form

(fragment mass . intensity of peak) – low resolution or
(fragment composition . intensity of peak) – high resolution.

It should be noted that exact mass values can be equated with fragment compositions, so that low and high resolution data differ to the accuracy (resolution) of mass measurements.

A *Predictive Theory of Estrogen Mass Spectrometry* is a total mapping from
   estrogenic molecules to fragment mass tables

where an estrogenic molecule is a tuple of substituent radicals on an (implicit) estrogen skeleton which is shown in figure 1.

2. *Assumption.* The FMT's (mass spectra) can be explained entirely in terms of allowable *actions* of a *given molecule.*

This serves to fix instrument characteristics, instrument parameter settings and operating environment and emphasizes the repeatability of a mass spectrometric experiment.

In making this assumption one restricts oneself to a theory that may not explain the entire FMT for each given molecule, but only those portions of the data that can be explained on the basis of a well-defined set of mass spectrometric actions of the molecule. This clearly allows only approximate predictions of FMTs, leaving unexplained effects of solvents, impurities of the sample compound, and other familiar mass spectrometric effects.

Actions (defined more precisely earlier) are allowable sets of cleavages followed by fragment selection and hydrogen migration.

An *Action-based Predictive Theory* is a mapping from
   molecules to set of actions on molecules

along with a well-defined effective procedure for predicting the FMT given a molecule and its set of actions. When the above effective procedure cannot predict the FMT but only the mass values or the composition of the fragments, without intensities, then the theory is a 0, 1 or a binary theory.

3. An *Action-based* 0, 1 *Predictive Theory* is a mapping and a procedure:
   molecule to set of allowable actions on molecule

and (molecule, actions) to table of fragment masses or of fragment compositions.

The set of actions allowable on a molecule can be specified, for example, by a predicate on the set of all defined actions on the molecule.

We will use M to denote the universe of all molecules; E to denote the universe of all estrogenic molecules; and $m \in M$ to denote a molecule in the universe; and Am to denote the set of actions definable on m.

4. A *Partial Theory* uses a predicate that is not total, that is, it is not defined over part or the whole of the domain Am. Another way to define the same is to use a three-valued total function on Am:

function on Am: for $a \in Am$, $a \rightarrow$ {YES, NO, DON'T KNOW}

This allows the mapping on M (or E) to be partial, when the function of Am maps each action to 'DON'T KNOW'.

It is not certain whether the structural information of a molecule is sufficient to allow prediction of intensities in a fragment mass table. We wish to focus our attention for the present on 0, 1 theories. First steps toward refining the 0, 1 theories to predict intensities can be made by

(1) refining the YES, NO entries in the range of the functions to be HIGH, MODERATE, LOW and ZERO on an arbitrary scale of intensities.

(2) associating a confidence with each prediction, thus refining the DON'T KNOW entry in the range.

APPENDIX 2. DESCRIPTION OF THE SPACE OF RULES

A. Formulation of the Rule Formation task for estrogenic molecules (estrogen-meta-DENDRAL)

A1. The space of all estrogens E is defined by the space generated by allowing arbitrary substituent radicals on the kE positions around the estrogen skeleton.

$E = \{e | e \text{ is } (skeleton; sub1, sub2, \ldots, sub\ kE)\}$

where each *subi* is any radical.

Allowable positions of substituents on estrogens:

1, 2, 3, 4, 6, 7, 8, 9, 11, 12, 14, 15, 16, 17, 18 $\quad kE = 15$

An example of one molecule of the estrogen class is estrone, shown in figure 2. It should be evident that $E \subset M$.

Let D denote the set of estrogens for which experimental data is presented, also called the *sample* ($D \subset E$). The experimental data consist of a list of molecule FMT pairs.

$DATA = \{(d, spectrum)\}$ for each d in D.

To derive illustrative examples we have used a reduced sample of 7 molecules derived from the original sample of 66. We shall refer to the latter as the reduced sample and the former as the full sample.

A2. In order to facilitate the development of an action-based theory the FMT is initially transformed to molecule-action sets, with the program described in detail in part C of this paper.

$DATA1 = \{(d, actions)\}$ for each d in D.

The transformed set of data, called $DATA1$ here, is rewritten to a set ($DATA2$) in which the actions are given primary emphasis. Both these transformations are described and illustrated in part C.

$DATA2 = \{(d, (action_i, fragment_i, composition, intensity_i))\}$.

representing explanations of the type: In the FMT of molecule d the fragment of $composition_i$ occurs due to $action_i$.

A3. *Ambiguity of actions and redundancy of explanations*

For two molecules *mi* and *mj*, the bonds (the action sites and consequently the actions) of *mi* can be put in partial 1—1 mapping into the bonds of *mj*, and vice versa. This is obvious for actions involving only sites on the skeleton. Such a mapping can be extended to actions involving the substituents either alone or in combination with the skeletal sites. For simplicity, we are currently generating actions for the skeleton only. Soon we expect to generate and compare actions involving substituents on the skeleton as well.

Actions related by this mapping are *corresponding* actions. An example of two corresponding actions is shown in figure 3, labeled *a*1 and *a*2.

Two actions *ai* and *aj* on a single molecule *m* are *partially ambiguous* for *m* if and only if

fragment composition for *ai* = fragment composition for *aj*.

Corollary. For partially ambiguous actions *ai* and *aj* the intensity associated with *ai* equals the intensity associated with *aj*.

Remark. For a theory that uses and predicts only low-resolution mass spectra, a weaker definition of partial ambiguity would be necessary, which involves the fragment masses instead of the fragment compositions.

Two actions *ai* and *aj* are *totally ambiguous* for *D* when

for every *d* in *D* such that corresponding actions *ai'* and *aj'* are defined, *ai'* and *aj'* are partially ambiguous for *d*.

Multiple explanations for a peak derived from ambiguous actions will be referred to as *redundant* explanations.

Totally ambiguous actions are indistinguishable within the confines of the given sample and data. They may possibly be partially disambiguated by obtaining further experimental data. Partially ambiguous actions, on the other hand, point toward a basic limitation of mass spectrometric recording. Since a fragment mass table records only fragment masses or fragment compositions and no identity of the fragment in reference to the skeleton is preserved, effects of partially ambiguous actions are not differentiable in the data. They can be differentiated only through further assumptions about actions or by resort to additional data.

B. A theory of skeletal fragmentation in estrogens, defined on substituent effects

B1. A theory of skeletal fragmentation is an action-based theory defined only over the skeletal actions *As*.

As = set of actions definable on the skeleton
$= \bigcap_{e \in E} \{$actions defined on $e\}$

B2. *Structural coding for estrogens*

In the sample $D (D \subset E \subset M)$ which is a finite set of estrogenic molecules, each member is described by a finite set of substituent radical placements at specified skeletal positions. To simplify the presentation:

(i) hydrogens attached to skeletal positions are understood when no substituent is specified for some positions.

(ii) double bonds (or unsaturations) are simultaneously specified on pairs of adjacent positions, thus some viable pairs of positions are allowed as position specifications.

(iii) each substituent is described only by a nominal name for present purposes, although structural information about the substituent will be made available when need arises.

Let $Fi(D)$ be the finite set of substituents (i.e. substituent labels) occurring in position i, in the molecules in D, for $i = 1$ to kE

E' = set of molecules definable using $Fi(D)$ for all i
$= \{skeleton; f1 \in F1(D), f2 \in F2(D), \ldots fkE \in FkE(D)\}$

The size of this space of molecules is

$$|E'| = \prod_i |Fi(D)| \quad i \text{ from 1 to } kE.$$

By asserting that, each e' in E' represents at most one molecule we assert that we are neglecting the three-dimensional properties that discriminate molecules. E' is the intended maximum predictive domain of the theory for estrogens under coding with respect to $Fi(D)$.

Example. For a small sample of 7 molecules derived from the original 66 molecules,

(OH 3) (O=17) (DOUBLEBOND 6 7) (DOUBLEBOND 8 9) (CH3 18)
(OH 3) (O=17) (DOUBLEBOND 6 7) (DOUBLEBOND 8 9)
(OH 3) (OH 11) (OH 17)
(OH 3) (O=11) (O=17)
(CH30 3) (O=17)
(OH 3) (O= 17)
(OH 3) (OH 17)

$|D| = 7$ and $KE = 6$;

$F1(D)$ = OH or CH30 on 3
$F2(D)$ = DOUBLEBOND or H on 6 7
$F3(D)$ = DOUBLEBOND or H on 8 9
$F4$(D) = OH, O= or H on 11

$F5(D)$=OH or O= on 17

$F6(D)$=CH3 or H on 18 where we use H as the symbol for default values.

$|E'|=2\times2\times2\times3\times2\times2=96$. Similarly, $|E'|$ for full sample=(approx.) 1.86 million.

B3. *Primitive classes of molecules*

Define Sjk as the set of elements of E' restricted by substituent $fjk \in Fj(D)$ in position j of the skeleton, defined for j from 1 to kE and for k from 1 to $|Fj(D)|$. The primitive classes are classes defined by substituents at each position of the skeleton. There are $\sum_j |Fj(D)|$ primitive classes. Examples of primitive classes (there are 13 for the small sample):

$S11$=all those members of E' having (OH 3)

$S12$=all those members of E' having (CH30 3)

$S42$=all those members of E' having (O= 11)

B4. (*Non-primitive*) *class of molecules*

We can describe different classes of molecules by appropriately expressing them with a combination of set intersection and union operations on the primitive classes. The primitive classes thus generate a Boolean Algebra of sets, yielding a natural and convenient way of defining classes of molecules in E'.

The set in the Boolean Algebra generated by Sjk, define non-primitive classes in terms of membership function Sjk. Each molecule e' in E' can now be described in terms of the set intersection

$$S_{1x1}\cap S_{2x2}\cap \ldots \cap S_{kExkE} = \bigcap_i S_{ixi}, \text{ from 1 to } kE.$$

It should be evident that $|\cap S_{ixi}|=1$ and that $E'=\bigcup_i \bigcup_{x_i} S_{ixi}$. It is also easy to verify that the Boolean Algebra is non-atomic and that the number of sets defined as such is

$$2^{|E'|}$$

thus allowing expression of any arbitrary combinations of molecules to be expressed as a class.

Example. The first molecule in the reduced sample is given by

$S11\cap S21\cap S31\cap S43\cap S51\cap S61$

There are 52 primitive classes of molecules in the full sample. It is true that, for every j

$$S_{jx1}\cap S_{jx2}=\varnothing$$

C. A partitioning of the Rule Formation problem

C1. *Assumption.* The actions associated with the molecules are independent of each other. That is, the mapping of each action a in Am for each molecule m into the set {YES, NO, DON'T KNOW} can be specified independently of the mapping for any other action a' in Am' (for the same or any other molecule).

Corollary. The actions on the skeleton are independent of one another.
Remark. This independence assumption does not preclude the possibility of carrying over information learned from the efforts to solve one subproblem to another. But because rules are formulated for each action separately, it is necessary to unify rules after completion of rule formation. This independence affords a partitioning of the rule formation problem into $|As|$ subproblems each of which can be solved independently.

Let, for $a \in As$,

$Da = \{d | d \in D$ and there is evidence for action a in the spectrum of molecule $d\}$

$Da— = \{d | d \in D$ and there is no evidence for action a in the spectrum of molecule $d\}$.

The skeletal action formally labelled 8 is shown in figure 3.

For this action and the reduced sample,

$D8— = \{$(CH3 18) (O= 17) (OH 3) (DOUBLEBOND 6 7) (DOUBLEBOND 8 9), (O= 17) (OH 3) (DOUBLEBOND 6 7) (DOUBLEBOND 8 9)$\}$

$D8 = \{$(OH 3) (OH 11) (OH 17), (OH 3) (O= 11) (O= 17), (CH30 3) (O= 17), (OH 3) (O= 17), (OH 3) (OH 17)$\}$

$|E'(D8—)| = 2 \times 1 \times 1 \times 1 \times 1 = 2$

$|E'(D8)| = 2 \times 3 \times 2 = 12.$

The reduction in problem size afforded by this partitioning comes about in at least two ways: first, the rule formation for each action can be tackled separately without any information carry-over from one subproblem to another; second, the two Boolean algebras are much smaller than the Boolean algebra of the unpartitioned set. This consideration is examined below in more detail.

We can define $Fj(Da)$ and $Fj(Da—)$ on the partial data domains, and thereby define $Sjk(Da)$ and $Sjk(Da—)$ and generate the Boolean algebras $BA(Da)$ and $BA(Da—)$ to describe classes of molecules that have evidence for the action a and those that do not have such evidence. Since in general certain substituents are contributory to action a and certain other substituents inhibit action a, $Fj(Da)$ and $Fj(Da—)$ are smaller than $Fj(D)$. The size of $BA(Da)$ and $BA(Da—)$ are reduced to a mere fraction of the size of $BA(D)$.

C3. We need to choose one class b from each of the Boolean algebras to define the class of molecules believed to undergo the action and the class of molecules believed not to undergo the said action. There are some prior restrictions we can impose on the choice of b from $BA(Da)$ and $BA(Da—)$. The following discussion using D is to be naturally extended to Da and $Da—$.

Define $B(D) = \{b \in BA(D) | b \cap D$ is non-empty$\}$

= the set of b that can be used to explain at least one molecule in the data domain.

However, since union of sets is allowable in any b, one can postulate a stronger condition on the choice of b. An element $b \in B(D)$ is said to cover or to be a *cover* when $b \cap D = D$ (or restating the same, when $b \subset D$). Define $R(D) = \{b \in B(D) \mid b \text{ is a cover for } D\}$. A *rule* for an action a then consists of two covers such that

$b(a—) \cap Da = \varnothing$ and $b(a) \cap Da— = \varnothing$.

The above two conditions can be interpreted to mean *honesty* to data, in that we require the rules to explain the data completely and without any errors. This condition may be compromised when the reliability of the data is in question or when the evidence in the data is not unambiguous.

Example.

$B(8—) = S11 \cap S21 \cap S31 \cap S43 \cap S51 \cap (S61 S \cup 62)$

$B(8) = (S11 \cup S12) \cap S22 \cap S32 \cap (S41 \cup S42 \cup S43) \cap (S51 \cup S52) \cap S62$

They are respectively equivalent to the statements

$B(8—) = =$(OH 3) (DOUBLEBOND 6 7) (DOUBLEBOND 8 9) (O= 17) (either CH3 or H on 18)

and

$B(8) = =$(OH or CH30 on 3) (OH, O= or H on 11) (O= or OH 17) and (DEFAULT VALUES at 6 7, 8 9 and 18).

C4. Predictions using the rules would employ the logic:

if $m \in b(a—)$ and m is not $\in b(a)$ then predict occurrence of action a;

if $m \in b(a)$ and m is not $\in b(a—)$ then predict non-occurrence of a;

else do not predict (i.e. predict DON'T KNOW).

Example. For (CH30 3) (OH 17) one would predict occurrence of action 8 and for (CH30 3) (DOUBLEBOND 6 7) (DOUBLEBOND 8 9) (OH 17) one would predict DON'T KNOW.

C5. The space of possible rules is very large for each action and we seek appropriate heuristics and guiding principles that can either reduce the number of alternatives to be considered or impose a simple rule for choosing alternatives.

One simple principle that is acceptable practice concerns the 'independence' of substituent effects, unless there is evidence available for 'interaction' of substituents. When a substituent is judged to have an enhancing effect on an action a, among the molecules observed, it may be readily supposed that it will have the same effect on all molecules in the domain. Our approximation to this principle involves the suggestion:

From $Sjk \cap Da$ is not $\varnothing$ and $Sjk \cap Da— = \varnothing$ conclude $Sjk \subset b(a)$;

From $Sjk \cap Da—$ is not $\varnothing$ and $Sjk \cap Da = \varnothing$ conclude $Sjk \subset b(a—)$;

Example. In the above example, it is reasonable to expect that any molecule having (DOUBLEBOND 6 7) will fail to show evidence for action 8 and any molecule having (OH 11) will show evidence for the same action 8.

Parsimony is often a desirable aim in constructing rules and there is a way to state preference for parsimonious rules in our formulation of the problem. A class of molecules stated as a union of disjoint subclasses is our equivalent

of a non-parsimonious class. Rules will be chosen to minimize the number of disjoint subclasses mentioned explicitly in the statement of the rule. Note that the representation of classes of molecules in terms of the primitive classes was really motivated toward this end, for each of the primitive classes is non-disjoint. When one begins to take intersections of the primitive classes and thereby refines them, then one moves away from chances of obtaining a parsimonious class.

There are hosts of other criteria that one can readily formulate that are intuitively valid to exercise, but we would like to experiment with as many as are reasonable, and explore ways of meaningfully combining criteria. The criteria in our current repertoire besides measures of simplicity/complexity of rule statements and degree of generality, include some measures based on the effort spent in formulating and validating the rules. There are criteria that involve prior knowledge of the theories of mass spectrometry and attempted carry-over of information from relevant areas of chemistry about possible effects of substituents on the behaviour of molecules. We expect to keep explicit account of the chemical knowledge that enters into the programs not only for understanding theory formation but also for understanding how changes in the knowledge base alter the result.

D. Extension of the predictive domain of the rules (planner and term-replacement)

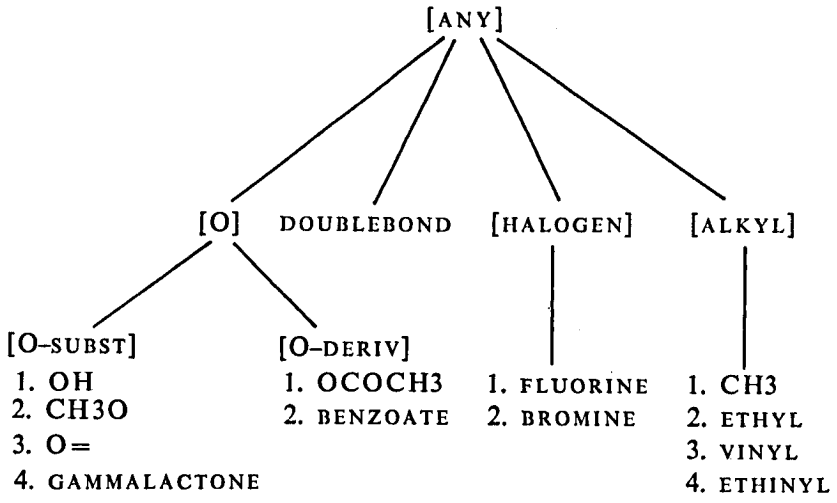
D1. The sets of rules derived for each of the actions are defined each in the narrow context of the molecules showing evidence for the action and others not showing such evidence. The rules may however be extended, as a post-process conceivably, to encompass some molecules not within the strict bounds of $E'(Da)$ and $E'(Da—)$ but within $E'(D)$. The rules also need to be extended into the infinite domain E (all estrogens) as far as possible. One possible technique of removing the bind imposed by the use of only those substituents actually seen in the data, involves the introduction of new substituent labels to serve as generalizations of substituents and effectively replacing selected terms in the rules with the generalized substituents. Such term replacement techniques raise the following questions:

(i) when to replace terms;
(ii) what terms to replace;
(iii) and what terms to use for replacement?

The notion of term-replacement is based on the central assumption that the behaviour of a molecule depends solely on its topological structural features. Therefore, when two substituent radicals exhibit similar effects, one tends to relate the similarity in effects to their common structural parts. That is, extract the 'common' subgraph of the structures and extend the domain of the molecules covered to any substituent built on the common subgraph. Thus an answer to the last question raised above is:

Replace the selected part of a rule that reads '*Sjk*∪*Sjl*' by 'any substituent containing the subgraph common to *Sjk* and *Sjl*'.

D2. In order to spare the initial program the trouble of graph manipulation for purposes of common subgraph determination, we managed to define a hierarchy of the substituents found in our sample of 66 estrogens, by suitably constructing additional substituent labels for the generalization of substituents. The hierarchy is drawn below as a tree with the defined additional terms indicated by brackets.



Example. (*S*41∪*S*42) which corresponds to (OH or O= on 11) can be term-replaced by (O 3) or by (O-SUBST 3) and (*S*61∪*S*62) corresponding to (CH3 or H on 18) can be term replaced by (ALKYL 18).

Term Definitions

O-DERIV Derivatives on any carboxylic acid.

O-SUBST Any substituent that is connected through an oxygen atom, and which is not an O-DERIV.

HALOGEN Any of Fluorine, Bromine, Iodine or Chlorine.

ALKYL Any substituent radical formed out of carbons and hydrogens where each atom is connected to another by only single bonds.

O Any substituent radical that has a leading oxygen atom (thus equivalent to either of O-DERIV and O-SUBST).

ANY Allows any substituent radical.

D3. The program to select rules out of $R(D)$ should be informed of this postprocessing, so that it may form rule-classes amenable to term-replacement. The information that this program needs in order to function effectively is to be generated by a *planning* phase, which as yet is very vaguely defined. The function of planner is to be generally described as analyzing the effects of each of the substituents with respect to each of the actions and classifying

them as having an enhancing, reducing or neutral effect on the intensity of the peak corresponding to the action. It will also seek to discover those interactions between substituents that the rule formation program should consider. The suitability of two substituents to be blended by term replacement will also be assessed by planner by putting in the corresponding structural similarities of substituents and the similarity of their observed effects on intensities of peaks.

Acknowledgements

This research was supported by the Advanced Research Projects Agency (SD-183) and the National Institutes of Health (RR-612-02). Mr William C. White and Dr Dennis H. Smith provided much assistance in formulating and implementing the programs.

PERCEPTUAL AND LINGUISTIC MODELS

15

Mathematical and Computational Models of Transformational Grammar

Joyce Friedman
Department of Computer and Communication Sciences
University of Michigan

INTRODUCTION

In this paper we compare three models of transformational grammar: the mathematical model of Ginsburg and Partee (1969) as applied by Salomaa (1971), the mathematical model of Peters and Ritchie (1971 and forthcoming), and the computer model of Friedman *et al.* (1971). All of these are, of course, based on the work of Chomsky as presented in *Aspects of the Theory of Syntax* (1965).

We were led to this comparison by the observation that the computer model is weaker in three important ways: search depth is not unbounded, structures matching variables cannot be compared, and structures matching variables cannot be moved. All of these are important to the explanatory adequacy of transformational grammar. Both mathematical models allow the first, they each allow some form of the second, one of them allows the third. We were interested in the mathematical consequences of our restrictions.

The comparison will be carried out by reformulating in the computer system the most interesting proofs to date of the ability of transformational grammars to generate any recursively enumerable set. These are Salomaa's proof that the Ginsburg-Partee model can generate any recursively enumerable (r.e.) set from a regular base, and the Peters-Ritchie proof that any r.e. set can be obtained from a minimal linear base. Although modifications are required, it is, as we shall show, possible to obtain these results within the weaker computer model.

Thus, every recursively enumerable language is generated by a transformational grammar with limited search depth, without equality comparisons of variables, and without moving structures corresponding to variables. The comparison reinforces the observation that transformational grammars can be excessively powerful in terms of generative capacity while at the same time lacking features necessary for explanatory adequacy.

An understanding of the role of variables in the structural description of a transformation is essential to the arguments of this paper. For the reader unfamiliar with the notion, we offer here an elementary explanation. Basically the idea is that in giving a structural description to match a tree, part of the description can be omitted in favor of a *variable* which will match any structure. In the simplest case consider the tree

$$S\langle NP\langle \text{rabbits}\rangle\ VP\langle V\langle \text{eat}\rangle\ NP\langle \text{lettuce}\rangle\rangle\rangle$$

where the brackets indicate a tree structure in which S dominates NP and VP, and VP dominates V and NP. Then either of the structural descriptions $NP\ VP$ or $NP\ V\ NP$ matches the tree. But these would not match

$$S\langle AUX\langle \text{do}\rangle\ NP\ \langle \text{rabbits}\rangle\ VP\langle V\rangle \text{eat}\rangle\ NP\ \langle \text{lettuce}\rangle\rangle\rangle$$

because an AUX precedes the leftmost NP. To match either of these trees, and any other tree ending in $V\ NP$, we introduce a variable as the left part of the structural description. The variable will match any initial structure. In the computer notation the percent sign is used for the variable, so we write $\%\ V\ NP$.

MATHEMATICAL AND COMPUTATIONAL MODELS

The two mathematical models limit themselves to the base component and the transformational component of the syntactic part of a grammar. That is, there is no consideration of either semantics or phonology, and furthermore, no mention of the lexical component in syntax. The computer model likewise ignores semantics, although it has now been extended to phonology. Within its syntactic part it has an important lexical component.

For purposes of comparison we restrict ourselves to the components that all three models have in common, that is, the base component and the transformational component. The base component consists of phrase structure rules, with a distinguished initial symbol S. This generates a base tree with S as root. The transformational component maps this tree onto a surface tree by application of a sequence of transformations.

The computer model differs from the mathematical models because of its intended use. It is not designed primarily as a mathematical object, but is the basis for a computer program used by linguists in writing and testing grammars and by students learning about transformational grammar. Some users are interested in the theory of syntax, and write grammars to illustrate points of theory. For example, the program is now being used to investigate case grammars. Other users are interested in describing little-known languages. For them, the grammar represents a hypothesis about the language which is tested by the program. For them transformational theory is simply the best currently available formalism for grammar.

In any case, the user writes a transformational component which is an input to the program. He may also provide a base component and study random sentences, or he may provide a partial or complete base tree. The output of the program is the output of the grammar, that is, a full derivation

from base tree to surface tree. The user can modify or accept his grammar based on this information.

Given this intended use of the program, there are certain natural consequences for the model. The computer form of a grammar should seem natural to a linguist: a computerized transformational grammar shouldn't look too different from those in the literature. The model must allow a choice of alternative expressions, and thus will contain redundant features, and also some features which purists might consider too powerful.

A computer model can be too strong without harm, provided only that the user can specialize it by rejecting options which he feels are unnecessary. A simple example is the choice of repetition modes for transformations. A transformation applies if, on testing, a structural analysis of the tree is found which matches the structural description of the transformation. Four different 'repetition modes' can be used to determine whether one or all such structural analyses will be found, and if and when the corresponding structural changes will be made. Using mnemonics constructed of 'A' for 'analyze' and 'C' for 'change', we can represent these as AC (find the first analysis and apply the change), AACC (find all analyses, then do all changes), ACAC (find the first analysis, do the change, repeat), and AAC (find all analyses, do one randomly selected change). Some linguists might argue that not all of these are necessary; indeed many might feel that one mode suffices for all transformations. For a computer model, however, it is advisable to allow all reasonable possibilities, so that a user may make his own choice. The user is free to experiment, without being committed to the use of excessive power.

Similarly, there may be technical weaknesses in a computer model which are desirable for practical reasons. Although the base grammar specifies that the sentence symbol *S* may be introduced recursively, the computer program will not introduce embedded sentences in the base unless the user has specifically called for one in the input to a particular run. This device is necessary if the output of the generation process is to remain within practical limits. In fact, for transformational grammar, the relation between embedding and embedded sentences must be well specified if a sentence is to result.

In a mathematical mode, on the other hand, it becomes very important to be neither too weak nor too strong, because the investigation of power is a prime purpose in constructing the model. Unbounded processes must be expressed as such, wherever the linguistic theory allows them. That is, results would immediately be suspect if what is really an unbounded process in language were simulated by a bounded process in the mathematical model. Thus, both the Ginsburg-Partee and Peters-Ritchie models attempt to be faithful to linguistic theory in a way that the computer model does not.

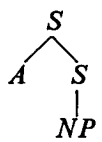
Restrictions of the use of variables

We have mentioned above several important ways in which the computer model is apparently too weak for explanatory adequacy. These are

1. bounded depth of analysis
2. lack of equality comparisons on variables
3. inability to move structures corresponding to variables.

On the other hand, both mathematical models allow unbounded depth of analysis; both allow equality comparisons of variables, although the Ginsburg-Partee model compares terminals only; Peters-Ritchie, but not Ginsburg-Partee, allows movement of structures corresponding to variables.

The bounded depth constraint in the computer model requires that in analyzing a tree to match a structural description the search never goes below an S unless that S is explicitly mentioned. (If S is the only recursive symbol in the base, then an equivalent statement is that no essential use of variables is allowed. Any occurrence of a variable could then be replaced by a finite choice of definite symbols. Thus, all these restrictions are restrictions on variables.) Consider a structural description $S/\langle A\ S\ \langle NP\rangle\rangle$ and the subtree



If the search begins at the top S, the daughters A and S are found and then the NP below this S can be matched. But no match will be found if the structural description is just $S/\langle A\ NP\rangle$, which does not allow the search to go below the intermediate S. This restriction was made because it is in practice useful. It is convenient not to have to consider more than the current S unless that consideration is part of the argument. For example, in the Peters-Ritchie proof there are several structural descriptions (e.g. T3) with the condition *1 is not an S*; in the computer version the condition can be omitted because of the bounded depth constraint (see Appendix B). Further, the user can study the effect of unbounded search by writing the alternation of the case of depth 1, depth 2, and so on up to any finite limit. This of course does not give an exact representation of the transformation, but is adequate for all practical purposes.

A consequence of the decision to block search whenever an S is encountered in the tree but not in the structural description is that it is not possible for a transformation to pull a constituent out of an arbitrarily deeply embedded subtree, raise it over any number of sentence boundaries, and bring it up to a higher sentence. Postal (1971) has argued convincingly that transformations must have this power. It must be possible to write a single transformation that will find a noun phrase and bring it up from an arbitrarily deeply embedded subsentence. Postal's example of the type of sentence whose derivation requires unbounded depth is

Who did Jack find out that Mary believed that Bill said that you thought you saw?

Here the deep structure corresponds roughly to

Q(Jack found out that (Mary believed that (Bill said that (you thought (you saw who)))))

where the parentheses indicate sentence embedding. The final 'who' must be moved up to the top sentence from a sentence arbitrarily far down. There is in general no upper bound to this depth.

There are results that can be proved about transformational grammars if search depth is bounded, that are not easily proved, and are possibly not even true, otherwise. Hamburger (1971) was able to extend to transformational grammars some of Gold's results (1967) on identification of languages in the limit. One of Hamburger's crucial assumptions was that the search depth was bounded.

The remaining two points in which the mathematical models appear more powerful, the equality comparison for variables and moving of structure matched by variables, are primarily motivated by the analysis of conjunction (which indeed poses many problems for transformational grammar).

THEOREMS OF SALOMAA AND PETERS-RITCHIE

In spite of these linguistically weak aspects of the computer model, it retains the full power demonstrated for the mathematical models. We examine two major mathematical results on transformational grammar, and then show that both proofs can be reproduced in the computer system.

Theorem (Salomaa). For an alphabet A, there is a set R of transformational rules such that any recursively enumerable λ-free language L_0 over A is generated by a (restricted) Ginsburg-Partee transformational grammar with regular base and with R as the set of transformational rules. Salomaa's proof is based on the theorem that every recursively enumerable language L_0 can be expressed as

$$L_0 = h(h_1(D \cap K_1) \cap h_1(D \cap K_2))$$

where h and h_1 are homomorphisms, D is a Dyck language, and K_1 and K_2 are regular languages. D, h, and h_1 are determined by A only, and are independent of L_0. The base component generates the language $K_1 a_0 K_2$, where a_0 is a marker, and the transformations carry out the homomorphisms and check whether or not substrings belong to D and whether they belong to the required intersections.

The Salomaa proof seems intuitively to differ from a normal linguistic derivation. In particular, the lack of a transformational cycle seem unnatural. This is closely related to the arbitrary recursion in the base. Not only is S not a recursive symbol, but most other nonterminals of the base can be recursive. Thus, the Salomaa proof seems to use a grammar which is different in obvious ways from the grammar required for natural languages.

Theorem (Peters and Ritchie). L is a recursively enumerable language on the

alphabet $A=\{a_1, a_2, \ldots, a_n\}$ if and only if L is generated by a transformational grammar with the base rules $S \rightarrow S\#$ and $S \rightarrow a_1 a_2 \ldots a_n b\#$.

The Peters-Ritchie proof begins with the fact that every r.e. language is enumerated by some Turing machine Z. They construct a transformational grammar which simulates the Turing machine Z. The terminal string of the tree as it goes through the derivation contains a substring that represents the instantaneous description of the Turing machine. The transformational grammar is set up so that at each cycle exactly one Turing machine instruction is applied. As the derivation proceeds up the tree, the Turing machine is simulated step-by-step, and its instantaneous description is carried along. Each time a cycle is completed, a boundary symbol is erased, and one of the S's given by the first rule of the base component is pruned. The initial (base) tree has enough sub-trees so that there will be one for each instruction to be used. Finally, a very clever scheme is used to erase all the boundary symbols just in case the Turing machine has been adequately simulated. The language of the grammar, that is, those surface strings which do not contain the boundary symbol, corresponds to the set of sentences in the language enumerated by the Turing machine.

These results show that transformational grammars as usually formulated are too powerful. Peters and Ritchie (1969) observe that their result makes impossible an empirically testable statement of the universal base hypothesis for natural languages, unless one enlarges the range of data to be accounted for by a grammar.

There are, of course, other results that show that transformational grammars under certain restrictions generate restricted subclasses of languages. For example, Petrick (1965) showed that, for a particular definition of transformational grammar, only recursive languages are obtained if the depth of embedding is bounded by a function of the length of the sentence. Refined results along these lines are given in Peters and Ritchie (1971).

COMPARISON OF THE THREE MODELS

In comparing the three models we emphasize features used in these two proofs. For each proof, we constructed a computer version that was run on several examples. Appendix A lists our reproduction of Salomaa's proof; Appendix B is the Peters-Ritchie proof. In the discussion we show how these versions of the proofs differ from the originals, and in particular we show that neither of them makes unavoidable use of the more powerful concepts of variable which are lacking in the computer model. Specific transformations of the proofs will be referred to frequently; they will be found in the appendixes.

Base component

No attempt was made to simulate on the computer the base components of the two proofs. The base rules are listed in the Appendixes for completeness

only. All computer runs were made starting with completed base trees. The computer model treats base rules as ordered context-free rules with recursion on the sentence symbol only.

In the absence of any specification of the base component in the Ginsburg-Partee model, Salomaa obtains the base trees by a regular grammar in which the rules are unordered and there are many recursive symbols; however, the sentence symbol is not reintroduced by any rule.

The Peters-Ritchie model allows an unordered set of context-sensitive rules as the base. In the proof under consideration the base is a two-rule minimal linear grammar; recursion is on the sentence symbol only.

Structural description

In reproducing the Salomaa proof in the computer system, a few changes to the transformations were necessary. There is no difficulty with S-bounded search depth, since the base structure is a simple sentence with no embedded S. However, the transformation T8, which checks whether $h_1(k_1)=h_1(k_2)$, does so by an equality test on variables. Equality of variables here is equality of corresponding terminal strings only. T8 cannot be transcribed directly because comparison of variables is not allowed in the computer model. However, by using a device that Salomaa uses elsewhere in the proof, T8 is replaced by a sequence of transformations T8A, T8B, and T8C (see Appendix A), which create an extra copy of the relevant subtree and then compare its halves node by node, deleting if the comparison is satisfactory.

Another difference, though not relevant to our main discussion, is that for Salomaa a structural description is a boolean combination of proper analyses with no explicit mention of substructure. His proof uses boolean combination in three transformations, T5, T7, and T11. In all three the form is a single proper analysis which carries the main burden of the transformation, conjoined with a negation which specifies that no letter of a particular alphabet occurs. This is done to ensure that the previous transformation has applied as often as it can and is now no longer applicable. Thus, if the transformations had been taken as ordered, the negation would be unnecessary. In the computer model a structural description is a sequence of structures and negation is available only with reference to the subanalysis of a mentioned node. This was adequate to the purposes of T7 and T11 (see Appendix A). T5 was rewritten more simply since the boolean combination was in fact unnecessary.

The Peters-Ritchie model has a much richer notion of structural description, specified in terms of a boolean combination of conditions on a factorization of the labelled bracketing representing the tree. The model allows equality comparisons of variables; these compare structures rather than terminal strings. However, this device is not used in their proof.

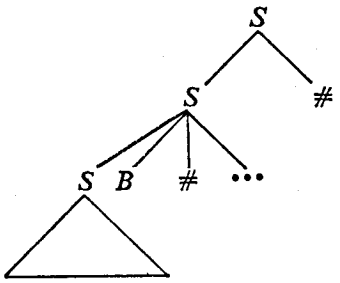
The Peters-Ritchie model does allow unbounded search depth, and it was on this point that some changes were necessary in transcribing the trans-

formations. It was necessary to introduce additional structure in the statements of the transformations; however, some supplementary conditions were thereby eliminated.

The basic scheme for the transformations of the Peters-Ritchie proof can most easily be seen by examining transformation T5. For the case of an alphabet with the three symbols $A1$, $A2$, $A3$, the structural description of T5 can be written in the notation of the computer model as

S⟨S⟨# # # # A1 A2 A3 B # # # # (# #) B⟩B# %⟩#.

Because of the bounded search depth discussed above, the computer version of this transformation mentions two sentence symbols S rather than just the one mentioned in the original proof. The tree matched by this structural description can be represented schematically as:



The sentence tree whose contents are fully given by the structural description is the lowest one in the figure. The dots indicate the position in the tree of the instantaneous description of the Turing machine. The top S in the figure is the one at the top of the current step of the cycle.

All of the structural descriptions of the Peters-Ritchie proof can be rewritten in this form or, as in T3, as choices of a finite set of these. Thus unbounded search depth, although allowed by the notation, is not needed for the proof.

Structural change

The computer model disallows any operations on variables, but is otherwise able to reproduce all changes allowed by the other two models. The Peters-Ritchie model allows deletion, substitution, and adjunction, all of sequences of factors; the Ginsburg-Partee model is similar except that sequences corresponding to variables cannot be moved. Since the Peters-Ritchie proof does not use the ability to move variables, there is no difficulty in transcribing its structural changes to the computer notation.

Sequencing of transformations

The three models differ in the way they specify the order of application of transformations. The computer model provides a language in which a

control program can be written. The Peters-Ritchie model assumes the standard bottom-to-top transformational cycle. In Appendix B, the computer control program is used to provide the transformational cycle for the Peters-Ritchie proof.

The control device of the Ginsburg-Partee model is also quite general, but Salomaa's proof uses only a restricted model in which the control device is ignored. Salomaa's transformations are unordered and are formulated so that at any point in the derivation at most one of them can apply. The computer system requires some order for the rules; in Appendix A the control program simply applies them in the order written. By marking some rules as ACAC the full derivation is carried out in one pass through the rules. An alternative would be to specify all rules as AC and have the control program invoke them repeatedly until none applies.

Parameters

The repetition mode parameter of the computer model was discussed above as an example of deliberate excess power. Although the mathematical model of Ginsburg and Partee provides directly for AC transformations, and indirectly, through the control device, for ACAC transformations, Salomaa's proof uses only AC. In simulating his proof it is convenient, though not necessary, to use both AC and ACAC. The Peters-Ritchie model treats all transformations as AAC. In their proof, transformations T3 and T4 must be regarded as AAC; the others could be indifferently taken as any of the four modes.

The computer model allows transformations to be specified as optional or obligatory. Since the optionality parameter is relegated by Ginsburg and Partee to the control device, which Salomaa does not use, Salomaa's transformations are all obligatory. In the Peters-Ritchie system all transformations are obligatory, although the effect of optionality can be obtained.

Special conditions

The computer model was designed to be neutral with respect to certain special conditions on transformational grammars so that a user might simulate them if desired but would not be required to include them. The condition on recoverability of deletions is required by the Peters-Ritchie model; it is preserved by our version of their proof. Their automatic 'pruning convention' is simulated by the transformation TPRUNE in the computer version. The filtering condition is a condition on the output of a grammar: for Peters and Ritchie the presence of the boundary marker # signals a failed derivation; Salomaa uses the markers $a_0, \ldots, a_6$ for this purpose.

Acknowledgement

This research was supported in part by the National Science Foundation under Grant GS 31309 to The University of Michigan.

REFERENCES

Chomsky, N. (1965) *Aspects of the Theory of Syntax*. Cambridge, Mass.: MIT Press.

Friedman, J. *et al.* (1971) *A Computer Model of Transformational Grammar*. New York: American Elsevier.

Ginsburg, S. & Partee, B. (1969) A mathematical model of transformational grammars. *Information and Control*, **15**, 297–334.

Gold, E.M. (1967) Language identification in the limit. *Information and Control*, **10**, 447–74.

Hamburger, H.J. (1971) *On the Learning of Three Classes of Transformational Grammars*. Ph.D. dissertation, University of Michigan.

Peters, P.S. & Ritchie, R.W. (1969) A note on the universal base hypothesis. *Journal of Linguistics*, **5**, 150–2.

Peters, P.S. & Ritchie, R.W. (1971) On restricting the base component of a transformational grammar. *Information and Control*, **18**, 483–501.

Peters, P.S. & Ritchie, R.W. (forthcoming) On the generative power of transformational grammars.

Petrick, S.R. (1965) *A Recognition Procedure for Transformational Grammars*. Ph.D. dissertation, Massachusetts Institute of Technology.

Postal, P.M. (1971) *Cross-Over Phenomena*. New York: Holt, Rinehart and Winston.

APPENDIX A

```
"COMPUTER EXPERIMENTS IN TRANSFORMATIONAL GRAMMAR:"
"SALOMAA- THE GENERATIVE CAPACITY OF TRANSFORMATIONAL GRAMMARS"
"         OF GINSBURG AND PARTEE"
"INFORMATION AND CONTROL, 18, 227-232 (1971)"


          "THE BASE GENERATES A WORD  K1 A0 K2 "
PHRASESTRUCTURE
     S = Q1.
          "LET G1 BE A REGULAR GRAMMAR GENERATING K1"
          "WITH INITIAL SYMBOL Q1"
          "WITH NONTERMINALS  U1, U2, SAY"
     "**** INSERT HERE ALL RULES  A = X B, B NONTERMINAL, OF G1"
     "**** INSERT HERE A RULE  A = X S2, FOR EACH RULE A = X, X IN I2"
          "LET G2 BE A REGULAR GRAMMAR GENERATING K2"
          "WITH INITIAL SYMBOL Q2"
          "WITH NONTERMINALS  V1, V2, SAY"
     S2 = A0 Q2.
     "**** INSERT HERE ALL RULES OF G2"
     $ENDPSG

     "LET I BE D1, D2"
     "LET I1 BE C1,C2,C3,C4"
     "LET THE SYMBOLS OF I2 BE B1,...,B6, WITH INVERSES BI1,...,BI6

TRANSFORMATIONS

          "T1 DUPLICATES THE BASE WORD"
          "AND INTRODUCES THE MARKER A1 BETWEEN THE TWO COPIES"
 TRANS T1 AC.
   SD I Q1.
   SC 1 ADRIS 1, A1 ARISE 1.

          "THE NEXT RULES OPERATE ON THE LEFTMOST COPY"
          "T2 CHECKS WHETHER OR NOT  K1  BELONGS TO  D "
          "AND IF IT DOES ERASES K1. "
 TRANS T2 ACAC.
   SD % (1 B1 2 BI1, 1 B2 2 BI2, 1 B3 2 BI3, 1 B4 2 BI4, 1 B5 2 BI5,
         1 B6 2 BI6) % S2 A1 Q1.
   SC ERASE 1, ERASE 2.

          "T3 CHANGES THE MARKER TO A2"
 TRANS T3 AC.
   SD S2 2A1 Q1.
   SC A2 SUBSE 2.

          "T4 CHECKS WHETHER OR NOT  K2  BELONGS TO D,"
          "AND, IF IT DOES, ERASES  K2"
 TRANS T4 ACAC.
   SD A0 % (1 B1 2 BI1, 1 B2 2 BI2, 1 B3 2 BI3, 1 B4 2 BI4, 1 B5 2 BI5,
            1 B6 2 BI6) % A2 Q1.
   SC ERASE 1, ERASE 2.

          "T5 CHANGES THE MARKER TO A3, "
          "PROVIDED BOTH  K1  AND  K2  BELONG TO  D "
          "T5 ALSO ERASES THE LEFT MOST BRANCH OF THE TREE"
 TRANS T5 AC.
 "BOOLEAN COMBINATION EXCLUDING LETTERS OF I2 IS UNNECESSARY"
   SD 1A0 2A2 Q1.
   SC ERASE 1, A3 SUBSE 2.

          "THE REMAINING RULES OPERATE ON THE RIGHT-MOST BRANCH"
          "T6 APPLIES THE HOMOMORPHISM  H1  TO BOTH  K1  AND  K2 "
```

```
"TRANS T6"
      "THE ACTUAL SC WILL DEPEND ON H1. THE STRING H1(X) IS ADJOINED"
      "TO THE LEFT OF 3, THEN 3 IS ERASED."
      "TO REPRESENT THE HOMOMORPHISM H1 WE BREAK T6 INTO 2R+8 "
      "TRANSFORMATIONS, AND CARRY OUT THE MAPPING SYMBOL BY SYMBOL"
TRANS T6B1 ACAC.
  SD A3 % 3B1 %.
  SC C1 ALESE 3, C1 ALESE 3, ERASE 3.  "H1(B1)=C1 C1"
TRANS T6B2 ACAC. SD A3 % 3B2 %. SC ERASE 3. "H1(B2)=EMPTY"
TRANS T6B3 ACAC. SD A3 % 3B3 %. SC C2 ALESE 3, ERASE 3. "H1(B3)=C2"
TRANS T6B4 ACAC. SD A3 % 3B4 %. SC C3 ALESE 3, C4 ALESE 3, ERASE 3.
TRANS T6B5 ACAC. SD A3 % 3B5 %. SC C1 ALESE 3, ERASE 3.
TRANS T6B6 ACAC. SD A3 % 3B6 %. SC C2 ALESE 3, C2 ALESE 3, ERASE 3.
TRANS T6BI1 ACAC. SD A3 % 3BI1 %. SC C3 ALESE 3, ERASE 3.
TRANS T6BI2 ACAC. SD A3 % 3BI2 %. SC ERASE 3.
TRANS T6BI3 ACAC. SD A3 % 3BI3 %. SC C4 ALESE 3, C1 ALESE 3, ERASE 3.
TRANS T6BI4 ACAC. SD A3 % 3BI4 %. SC C2 ALESE 3, ERASE 3.
TRANS T6BI5 ACAC. SD A3 % 3BI5 %. SC C3 ALESE 3, C3 ALESE 3, ERASE 3.
TRANS T6BI6 ACAC. SD A3 % 3BI6 %. SC ERASE 3.

      "T7 CHANGES THE MARKER TO  A4, "
      "AFTER HAVING CHECKED THAT ALL APPLICATIONS OF H1 ARE MADE"
TRANS T7 AC (A3).
  SD 1 A3 Q1¬/<% (B1,B2,B3,B4,B5,B6,BI1,BI2,BI3,BI4,BI5,BI6) %>.
  SC A4 SUBSE 1.

      "T8 CHANGES THE MARKER TO A5, PROVIDED H1(K1)=H1(K2)"
"TRANS T8 CANNOT BE TRANSCRIBED DIRECTLY"
"BECAUSE WE DO NOT HAVE THE EQUALITY TEST FOR VARIABLES X"
"INSTEAD WE COPY THE TREES OF INTEREST AND COMPARE BY DELETION"
TRANS T8A AC (A4).
  SD A4  1 Q1 .
  SC 1 ADRIS 1, A5 ARISE 1.
TRANS T8B ACAC (A4).
  SD A4 Q1 A5 % 5(C1,C2,C3,C4) A0 % 6(C1,C2,C3,C4), WHERE 5 EQ 6.
     "COULD AVOID CONDITION 5 EQ 6 BY DOING 4 CASES"
  SC ERASE 5, ERASE 6.
TRANS T8C AC (A4).
  SD 1 A4 2 Q1  3 A5 4 A0.
  SC A5 SUBSE 1, ERASE 3, ERASE 4.

      "T9 ERASES THE BRANCH DOMINATED BY THE NODE S2,"
      "AND CHANGES THE MARKER TO A6"
TRANS T9 AC (A5).
  SD 1 A5 % 3 S2.
  SC A6 SUBSE 1, ERASE 3.

      "T10 APPLIES THE HOMOMORPHISM H"
TRANS T10 ACAC.
      "AN ALTERNATIVE WAY TO DO A HOMOMORPHISM"
  SD A6 % (1 C1,2 C2, 3 C3, 4 C4) %.
  SC D1 ALESE 1, D2 ALESE 1, ERASE 1,
     D1 ALESE 2, ERASE 2,
     ERASE 3,
     D2 ALESE 4, D2 ALESE 4, ERASE 4.

      "T11 ERASES THE MARKER, THUS LEAVING A WORD OF  L0 "
TRANS T11 AC.
  SD 1 A6 Q1¬/<% (C1,C2,C3,C4) %>.
  SC ERASE 1.

    CP I;TREE;II.
  $ENDTRA

  $MAIN FTRIN TRAN.

   "INPUT BASE TREES FOLLOW"
S<Q1<B1 U1<B3 U2<BI3 U1<BI1 S2<A0 Q2<B1 V1<B3 V2<BI3 V2<BI1>>>>>>>>>>.
S<Q1<B1 U1<BI1 U2<B3 S2<A0 Q2<B1 V1<BI1 V2<BI3>>>>>>>>.
  $
```

APPENDIX B

```
"COMPUTER EXPERIMENTS IN TRANSFORMATIONAL GRAMMAR:"
"PETERS AND RITCHIE- ON RESTRICTING THE BASE COMPONENT"
"INFORMATION AND CONTROL, 18, 483-501 (1971) "

PHRASESTRUCTURE
   S = ( S #, A1 A2 A3 B # ) .
     $ENDPSG

          "NOTE: BOUNDED DEPTH OF SEARCH, NO ESSENTIAL VARIABLES "
          "NOTE: NO NUMBERED VARIABLES "
          "NOTE: NO RESTRICTIONS ARE USED "

TRANSFORMATIONS
 TRANS T12."COMBINES T1 AND T2"
          "PRODUCES R BOUNDARIES FOR INNERMOST SENTENCE,"
          "AND POSITIONS THEM CORRECTLY AND ADDS B AS RIGHTMOST SYMBOL"
          "LET N=NUMBER OF TM SYMBOLS"
          "LET R=NUMBER OF TM STATES=4"
   SD 1A1 A2 A3 2B 3# .
   SC 3 ADLES 1, 3 ADLES 1, 3 ADLES 1, 3 ADLES 1, "(R TIMES)"
      2 ADRIS 3.
 TRANS T3 AAC .
          "T3 AND T4 PRODUCE THE OTHER 3 #'S IN THE INNERMOST SENTENCE"
          "AND THE STRING B**U # Y B**V "
          "T3 PRODUCES B**V, THEN T4 PRODUCES #Y, THEN T3 PRODUCES B**U"
  SD (9S< 1# # # # A1 A2 A3 3B (4#,5#) (# #) B > % 8#,
     S<9S<1# # # # A1 A2 A3 3B (4#,5#) (# #) B > % >   8#).
          "CONDITION THAT 1 IS NOT AN S IS UNNEEDED IN THIS FORMAT"
   SC 4 ADRIS 4, 3 ADRIS 9, ERASE 8.
 TRANS T4 AAC .
   SD   S<
      10S<#### (3A1 A2 A3,A1 3A2 A3,A1 A2 3A3) B (5#,#) 6# B > %> 9# .
   SC 5ADLES 6, 3 ADRIS 10, 5 ADRIS 10, ERASE 9.
 TRANS T5  II .
          "INCREASES T (INITIALLY 4) BY 1 WHEN LEFTMOST USABLE SQUARE"
          "OF TAPE IS REACHED FOR THE FIRST TIME"
   SD S< S< # # # # A1 A2 A3 B # # # 2# (# #) B> B # % > # .
   SC 2 ADRIS 2.
 TRANS T6 .
          "INCREASES T BY 2 WHEN RIGHTMOST USABLE SQUARE IS REACHED"
          "FOR THE FIRST TIME"
          "NOTE: A LEMMA IS NEEDED TO THE EFFECT THAT EVERY R.E. "
          "LANGUAGE IS ENUMERATED BY SOME TM THAT SCANS ITS WHOLE"
          "INPUT- THIS IS OF COURSE EASY TO PROVE"
   SD S< S< # # # # A1 A2 A3 B # # 2# # (#) B> %.# (A1,A2,A3,B) B > # .
   SC 2 ADRIS 2, 2 ADRIS 2.
 TRANS T7B.
          "TURING MACHINE ZB: REPLACES ALL A3 BY A2 AND GOES HOME "
          "(S1 A1 R S1) (S1 A2 R S1) (S1 A3 A2 S1) (S1 B L S2) "
    "     (S2 A1 L S2) (S2 A2 L S2) (S2 B R S3) "
   SD S<
     S<#### A1  3A2 A3 B ####(##)(#)B> %  (A1,A2,A3,B) 10# (18(A1,A2),
                                                          14A3       )
     %.B > 17 # .
   SC 18 ALESE 10, 9 ARISE 10, 3 SUBST 14, 2 ADLES 10, 19 ADLES 10,
      4 ADLES 10, ERASE 11, ERASE 12, ERASE 13, ERASE 17 .
 TRANS T7B2.
   SD S<
    S<# 2### A1 A2 A3 B ####(##)(#)B> % 9(A1,A2,A3,B) 10# (11# (A1,A2),
                                                          B        )
     % B > 17 # .
   SC 18 ALESE 10, 9 ARISE 10, 3 SUBST 14, 2 ADLES 10, 19 ADLES 10,
      4 ADLES 10, ERASE 11, ERASE 12, ERASE 13, ERASE 17 .
```

```
TRANS T7B3.
  SD S<
    S<# 2# 19## A1 A2 A3 B ####(##)(#)B>%(A1,A2,A3,B) 10# 11# 188
    % B > 17 # .
  SC 18 ALESE 10, 9 ARISE 10, 3 SUBST 14, 2 ADLES 10, 19 ADLES 10,
     4 ADLES 10, ERASE 11, ERASE 12, ERASE 13, ERASE 17 .
TRANS T8 .
         "T8,T9,T10 ARE CLEAN-UP TRANSFORMATIONS"
         "T8 CHECKS THAT THE SQUARES AT EACH END OF THE TAPE ARE NOT"
         "BEING SCANNED, AND THAT T=7 SO NO EXCESS TAPE WAS PROVIDED"
         "IF OK, T8 ERASES RIGHT-END B AND HALTING STATE, REPLACES"
         "DEEPEST SENTENCE BY S<#> AND POSITIONS A # TO SIGNAL T9"
  SD S<1S< # # # # A1 A2 A3 B # # # # # # #  # B> % (A1,A2,A3,A4,B)
     6# (10# (11# (12#))) (A1,A2,A3,B) % 9B > 13# .
         "   SECOND CHANGE OPERATION HERE COULD BE DONE BY ERASURES"
  SC 6 ADRIS 1, (S|+LOW +DONE|<#>) SUBSE 1,
     ERASE 6, ERASE 10, ERASE 11, ERASE 12,
     ERASE 9, ERASE 13.
TRANS T9 .
         "T9 PASSES A # ACROSS THE TAPE FROM L TO R, ONE SQUARE AT A"
         "TIME, ERASING EACH B ENCOUNTERED"
  SD S< S<#> % 3# (4B,5(A1,A2,A3)) 6* % > 7#  .
  SC ERASE 4, 3 ALESE 6, ERASE 7.
TRANS T10.
         "T10 ERASES FINAL # AND RIGHTMOST SYMBOL IF IT IS B "
  SD S<1S<#> % 3# (4B,A1,A2,A3) > 6# .
  SC ERASE 1, ERASE 3, ERASE 4, ERASE 6.
TRANS TPRUNE .
         "REDUCTION CONVENTION FOR LABELED BRACKETINGS"
  SD 1S<2S> , WHERE 1 DOMBY S .
  SC 2 SUBSE 1.
"TRANSFORMATIONAL CYCLE"
TRANS SMARK  V.
  SD 1S, WHERE 1 NDOM S.
  SC |+LOW| MERGEF 1.
TRANS LOWESTS  V.
  SD (1S|+LOW|, 1S<S|+LOW +DONE| %>, 1S<S|+DONE|<S|+LOW +DONE| %> %>),
     WHERE 1 NINC1 |+DONE|.
  SC |+DONE| MERGEF 1.
CP
  SMARK;
  IN LOWESTS(1) DO< I; II ; TREE > .
    $ENDTRA
```

16

Web Automata and Web Grammars

A. Rosenfeld and D. L. Milgram
Computer Science Center
University of Maryland

Abstract
Equivalences are established between certain classes of web grammars and certain classes of automata that have graph-structured tapes. As a corollary, normal forms for the web grammars are obtained. In particular, it is shown that a normal form exists in which all 'embeddings' are trivial.

1. INTRODUCTION

In 1969 Pfaltz and Rosenfeld (1969) introduced the notion of a *web grammar*, whose language is a set of labelled graphs ('webs'), and whose productions replace subwebs by other subwebs. This notion provides a general formalism for modeling a wide variety of data structures – in particular, relational structures such as those that arise in scene analysis problems. Web grammars for various classes of graphs and maps have been formulated (Montanari 1970, Rosenfeld and Strong 1971), and 'Chomsky hierarchies' for such grammars are under study by several investigators (Pavlidis 1972, Abe, Mizumoto, Toyoda, and Tanaka, in press). These grammars appear to be well suited for scene analysis applications (Shaw 1972); for one such application see Pfaltz (1972).

Webs and web grammars can be defined in a variety of ways; in fact, most of the papers cited above have used modifications of the original definition. For example, the graphs used can be directed or undirected, ordinary graphs or multigraphs; they can be maps rather than graphs (that is, they can be graphs in which the arcs at each node are cyclically ordered); they can be labelled on their nodes, their arcs, or both. Some of these variations may be particularly appropriate for scene analysis purposes – for example, the adjacencies between regions in a scene can be more fully represented by a map than by a graph; and arc labels can be used to represent different relations that may hold between the entities that correspond to the nodes. Nevertheless, the present paper will treat only the original case (ordinary graphs,

node labels only); it is planned to treat other cases, particularly those involving arc labels, in a subsequent paper.

The chief purpose of this paper is to study the relationship between web grammars and web automata – that is, automata that have webs as 'tapes'. One class of such automata has been studied by Shank (1971), but he deals with maps rather than graphs, and uses arc labels rather than node labels. (The map approach has also been pursued by Mylopoulos (1971).) It has therefore been necessary to develop analogous concepts for graphs having node labels only and no arc orderings.

Let V be a finite nonempty set, the elements of which will be called *labels*. A *web on* V is a triple $\omega=(N_\omega, E_\omega, f_\omega)$, where

N_ω is a finite, nonempty set; its elements are called the *nodes* of ω
E_ω is a set of unordered pairs of distinct elements of N_ω; these pairs are called the *arcs* (or *edges*) of ω
f_ω is a function from N_ω into V

The pair $(N_\omega, E_\omega)=G_\omega$ is a graph, called the *underlying graph* of ω. The distinctness of the terms of the pairs in E_ω implies that G_ω is loop-free.

If (m, n) is in E_ω, the nodes m and n are called *neighbors*. We call m and n *connected* if there exists a *path* $m=m_1, \ldots, m_k=n$, $k\geqslant 1$, such that $m_i \in N_\omega$, $1\leqslant i\leqslant k$, and $(m_i, m_{i+1}) \in E_\omega$, $1\leqslant i<k$. If any two nodes of ω are connected, ω itself is called connected.

Let α, ω be webs on V. We say that α is a *subweb* of ω if

N_α is a subset of N_ω
E_α is the restriction of E_ω to N_α
f_α is the restriction of f_ω to N_α

The *complement* of α in ω is the subweb $\omega-\alpha$ defined by restricting E and f to the subset $N_\omega-N_\alpha$ of N_ω.

2. WEB GRAMMARS

A *web grammar* is a 4-tuple $G=(V, V_T, S, P)$, where

V is a finite, nonempty set, called the *vocabulary*
V_T is a nonempty subset of V, called the *terminal vocabulary*
$S \in V-V_T$ is called the *initial symbol*
P is a finite, nonempty set of triples $\pi=(\alpha, \beta, \phi)$, called *productions*, where
  α and β are connected webs on V
  ϕ is a function from $N_\beta \times N_\alpha$ into 2^V (the set of subsets of V)

The significance of the function ϕ will be discussed below.

We say that ω' is *directly derivable* from ω in G if ω, ω' are webs on V, and for some (α, β, ϕ) in P,

(1) α is a subweb of ω

(2) For each m $\in N_\alpha$, all labels of neighbors of m in $\omega-\alpha$ are in $\phi(n, m)$ for some $n \in N_\beta$

(3) For all m_1, m_2 in α and all $n \in N_\beta$, any neighbor of m_1 in $\omega-\alpha$ that has a label in $\phi(n, m_1)\cap\phi(n, m_2)$ is also a neighbor of m_2

(4) ω' is obtained from ω by replacing α by β, and joining each $n \in N_\beta$ to all the neighbors (in $\omega' - \beta = \omega - \alpha$) of each $m \in N_\alpha$ whose labels are in $\phi(n, m)$. Formally:

$$N_{\omega'} = (N_\omega - N_\alpha) \cup N_\beta = N_{\omega-\alpha} \cup N_\beta$$
$$E_{\omega'} = E_{\omega-\alpha} \cup E_\beta \cup \{(n, p) \mid n \in N_\beta, p \in N_{\omega-\alpha};\ (p, m) \in E_\omega \text{ and } f_\omega(p) \in \phi(n, m) \text{ for some } m \in N_\alpha\}$$
$$f_{\omega'} = f_\beta \text{ on } N_\beta;\ f_{\omega'} = f_{\omega-\alpha} \text{ on } N_{\omega-\alpha}$$

Condition (2) in this definition is a form of Montanari's (1970) negative contextual condition: the production cannot be applied if, for any $m \in N_\alpha$, certain labels – namely, those not in the set $\phi(n, m)$ for some $n \in N_\beta$ – occur on neighbors of m in $\omega - \alpha$. Note that if $\bigcup_{n \in N_\beta} \phi(n, m) = V$, the condition is vacuous. Montanari has pointed out that such negative conditions cannot be 'simulated' by enlarging α to include its neighbors, since infinitely many such enlargements of α are possible. On the other hand, a positive condition – the production cannot be applied unless certain labels *do* occur on neighbors of m – *can* be simulated by enlarging α, since such neighbors can only be joined to each other, and to the nodes of α, in finitely many ways. The role of condition (3) will be discussed in Section 2.2; it guarantees that derivations are 'reversible'.

The function ϕ in a production specifies the *embedding* (Pfaltz and Rosenfeld 1969, Montanari 1970) associated with the production: that is, it specifies how to join the nodes of β to the neighbors of each node of α. This too is done in terms of the neighbors' labels – each $n \in N_\beta$ is joined to those neighbors (in $\omega - \alpha$) of each $m \in N_\alpha$ whose labels lie in a certain set $\phi(n, m)$. Condition (2) thus insures that, for each $m \in N_\alpha$, *some* $n \in N_\beta$ will in fact be joined to every neighbor of m in $\omega - \alpha$, since each such neighbor has a label in at least one of the sets $\phi(n, m)$.

Our definition of the embedding process is more general than Montanari's (1970); he specifies the embedding in terms of a function g from N_α into N_β, such that, for each $m \in N_\alpha$, every neighbor of m in $\omega - \alpha$ becomes a neighbor of $g(m)$ in $\omega' - \beta = \omega - \alpha$, and this does not allow a neighbor of m to become a neighbor of more than one $n \in N_\beta$. Our embedding ϕ, as well as our conditions (2–3) for the applicability of a production, are defined in terms of neighbor labels; but as we shall see later on, these definitions are equivalent to more general definitions using any properties of the neighbors that can be computed by a suitable web automaton.

2.2 Web languages

We say that ω' is *derivable* from ω in the web grammar G if there exist $\omega = \omega_1, \ldots, \omega_k = \omega'$, $k \geqslant 1$, such that ω_{i+1} is directly derivable from ω_i, $1 \leqslant i < k$. By the *language* $L(G)$ of G is meant the set of webs on V_T ('terminal webs') that are derivable in G from the 'initial web' consisting of a single node labelled S. Such terminal webs are called *sentences* of G; more generally, any web derivable in G from the initial web is called a *sentential form* of G.

Proposition 1. Any sentential form of a web grammar is a connected web.
Proof. Since the initial web is trivially connected, it suffices to show that if ω is connected and ω' is directly derivable from ω, then ω' is connected. Let r, s be any nodes of ω'. If they are both in N_β, we are done, since β is connected. If r is in N_β and s in $N_{\omega'-\beta}=N_{\omega-\alpha}$, let $s=s_1, \ldots, s_k$ be a path in ω from s to some $s_k \in N_\alpha$, with $s_1, \ldots, s_{k-1}$ not in N_α. Since s_{k-1} is a neighbor of $s \in N_\alpha$ in ω, by definition of ϕ there exists some $t \in N_\beta$ such that s_{k-1} is a neighbor of t in ω'. Thus s is connected to t, and since β is connected, t is connected to r.

Finally, if r and s are both in $N_{\omega'-\beta}$, let $r=r_1, \ldots, r_k=s$ be a path joining them in ω. If no r_i is in N_α, the path is also in ω' and we are done. Otherwise, let r_u and r_v be the first and last r_i's in N_α, so that $1<u\leqslant v<k$. Thus r_{u-1} and r_{v+1} are in $N_{\omega-\alpha}$ and are neighbors of nodes (namely, r_u and r_v) in N_α. It follows as in the previous case that r_{u-1} and r_{v+1} are also neighbors of nodes (call them u and v) in N_β. Since u and v are connected, we thus have r and s connected.//

It will be noted that by our conventions, webs always have nonempty sets of nodes, so that the 'null web' ε is never in any web language. If desired, it can be explicitly adjoined to a language. Alternatively, its presence in a language can be made possible by allowing productions in which $\beta=\varepsilon$. To insure that such productions can never disconnect derived webs, one can simply require that ϕ is empty (that is, $\phi(n, m)=\varnothing$ for all m, n) in any such production, so that by condition (2), the production can be applied only if α has no neighbors.

In string grammars, $L(G)$ can equivalently be defined as the set of terminal webs which can be 'parsed' (that is, from which the initial web can be derived) by applying productions of G in reverse – β is replaced by α rather than α by β. In the web case, a production $\pi=(\alpha, \beta, \phi)$ is applicable only if conditions (2–3) hold, and its application makes use of the embedding function ϕ; thus, to apply π in reverse, we must specify how the reverse embedding function is defined, and we must also be able to guarantee that conditions (2–3) hold for the reversal of π. Let ϕ_R be the function from $N_\alpha \times N_\beta$ into 2^V defined by $\phi_R(m, n)=\phi(n, m)$ for all $m \in N_\alpha$, $n \in N_\beta$, and let $\pi_R=(\beta, \alpha, \phi_R)$; then we have

Proposition 2. Let ω' be directly derived from ω by applying π to a particular instance of α as a subweb of ω. Then π_R is applicable to the resulting instance of β in ω', and the result of its application is ω.
Proof. Each $n \in N_\beta$ is joined only to nodes of $\omega'-\beta=\omega-\alpha$ whose labels are in the sets $\phi(n, m)=\phi_R(m, n)$, so that condition (2) holds for π_R. Moreover, for all n_1, n_2 in N_β and all $m \in N_\alpha$, neighbors of m in $\omega-\alpha$ that have labels in both $\phi(n_1, m)$ and $\phi(n_2, m)$ are indeed joined to both n_1 and n_2 in ω', so that condition (3) holds for π_R.

Let p be a neighbor of $m \in N_\alpha$ in $\omega-\alpha$. By condition (2) for π, p is still a neighbor of some $n \in N_\beta$, and its label is in $\phi(n, m)=\phi_R(m, n)$, so that when π_R

is applied, p becomes a neighbor of m again. Conversely, suppose π_R joins p to m; then p must be a neighbor, in $\omega'-\beta$, of some $n \in N_\beta$, and its label must be in $\phi_R(m, n)=\phi(n, m)$. On the other hand, n must have acquired this neighbor when π was applied, so that p is a neighbor, in $\omega-\alpha$, of some $m' \in N_\alpha$, and its label is in $\phi(n, m')$. By condition (3) for π, it follows that p is also a neighbor of m. Thus applying π_R restores ω exactly.//

We say that a terminal web can be *parsed* in G if the initial web can be derived from it by applying productions of $P_R=\{\pi_R \mid \pi \in P\}$. By Proposition 2 and induction, if ω is a sentence of G, it can be parsed in G by simply reversing its derivation. The converse follows similarly, since clearly $(\pi_R)_R=\pi$ for all π. We thus have

Proposition 3. The set of terminal webs that can be parsed in G is exactly $L(G)$.//

2.3 The languages 'V_T^*' and '$\omega\omega$'

In this section we exhibit grammars for two specific web languages: 'V_T^*', the set of all connected webs on V_T; and '$\omega\omega$', the set of webs consisting of two identical subwebs (in V_T^*), joined together in a manner to be described below. These grammars can serve to illustrate the concepts introduced thus far; they will also play important roles in the proof of the main theorems of this paper.

In describing grammars, we shall use pictures to represent webs – for example, S for the initial web; $A \quad B$ for the web ω having $N_\omega=\{m, n\}$, $E_\omega=\{(m, \dot{n})\}$, $f_\omega=\{(m, A), (n, \dot{B})\}^{\cdot}$. When nodes have distinctive labels, we will sometimes refer to them by their labels rather than introducing symbols to represent the nodes themselves; for example, in the example just given, we might speak of the nodes A and B, and not bother to introduce the symbols m and n. This shorthand will also be used in defining ϕ's, for example '$\phi(B, A)=C$' would mean 'Join B to the neighbors of A (in $\omega-\alpha$) that are labelled C'. We shall say that ϕ is *normal* if there is a one-to-one correspondence g between N_α and N_β such that $\phi(g(m), m)=V$ and $\phi(n, m)=\varnothing$ for all $m \in N_\alpha$ and all $n \neq g(m)$ in N_β. Note that if ϕ is normal, conditions (2–3) always hold.

Proposition 4. Let G_T be the web grammar having the following set of productions:

no.	α	β	ϕ	
1	S	$A \quad B$	$\phi(A, S)=\{A, S\}$; $\phi(B, S)=\{B, S\}$	
2	$\dot{S}$	$\dot{A}\dot{\longrightarrow}\dot{}$	normal	
3	$\dot{A}$	$\dot{B}$	normal	
4	$\dot{B}$	$\dot{S}$	normal	
5 ff.	$\dot{S}$	$\dot{t}$	normal	for each $t \in V_T$

Then $\dot{L}(G_T)^{\cdot}=V_T^*$.

Proof. Clearly any sentence of G_T must be a web on V_T, and by Proposition 1 it must be connected; we shall now show that any such web can in fact be

derived in G_T. It suffices to show that any connected web ω on $\{S\}$ can be derived in G_T, since productions 5 ff. can then be used to relabel this web with arbitrary terminal labels.

We proceed by induction on the number of nodes of ω; if this is 1, the desired web is the initial web itself. Suppose that it has been proved that any web on $\{S\}$ having k nodes can be derived, and let ω have $k+1$ nodes. Let r, s be neighbors in ω; using the reversals of productions 3–4, we can relabel ω so that the following labels occur on the following sets of nodes:

label	*nodes*
A	r, and all nodes ($\neq s$) that are neighbors of r but not of s
B	s, and all nodes ($\neq r$) that are neighbors of s but not of r

We can then apply the reversal of production 1 to convert the relabelled ω into a web having k nodes, and can then use the reversals of productions (2–3) to turn all A's and B's back into S's. By induction hypothesis, the resulting web is a sentential form of G_T; and since derivations are reversible, ω too is thus a sentential form of G_T.//

Note that G_T is 'context-free' in the sense that all its α's are one-node webs and all its productions are always applicable (provided we redefine $\phi(A, S)=\{A, S\}\cup V_T$, $\phi(B, S)=\{B, S\}\cup V_T$ in production 1).

We now describe a grammar G_C for the '$\omega\omega$' language; it has the following productions:

no.	α	β	ϕ
1	S	$X \quad S$	$\phi(X, S)=\phi(S, S)=\emptyset$

This production can be applied only to the initial S, which has no neighbors.

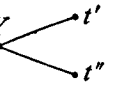
2	S	$A \quad B$	$\phi(A, S)=\{A, S, X\}$, $\phi(B, S)=\{B, S, X\}$
3	S	A	$\phi(A, S)=\{A, B, S, X\}$
4	A	B	$\phi(B, A)=\{A, B, S, X\}$
5	B	A	$\phi(S, B)=\{A, B, S, X\}$

These productions, like nos. 1–4 of G_T, can generate arbitrary connected webs on $\{S\}$, together with a node labelled X which is joined to every other node.

6	X	Y	$\phi(Y, X)=\{S\}$

At any stage when all nodes other than X are labelled S, the X can turn into a Y; this makes further application of productions 2–5 impossible. We now have an arbitrary connected web ω on $\{S\}$, together with a node labelled Y that is joined to all the nodes of ω.

The productions next to be defined will split ω into two 'copies' of itself, identically labelled with arbitrary terminal labels selected from a label set T; the node Y will still be joined to every node of both copies, but it will have a special terminal label $z \notin T$. The set of such terminal webs is our desired language '$\omega\omega$' (or perhaps better: '$\omega z\omega$'). In order to define the remaining productions, we need two sets of nonterminal labels $T'=\{t' \mid t \in T\}$ and $T''=\{t'' \mid t \in T\}$.

7  $\phi(Y, Y)=T'\cup T''\cup\{S\}$
$\phi(t', S)=T'\cup\{S\}$
$\phi(t'', S)=T''\cup\{S\}$

for all $t\in T$; unspecified values of ϕ are understood to be $\varnothing$. Repeated use of these productions splits ω into two 'copies' which have corresponding labels in T' and T'', respectively; Y remains joined to every node of both copies.

8 $\overset{\cdot}{Y}$ $\overset{\cdot}{z}$ $\phi(z, Y)=T'\cup T''$

When all S's have been rewritten by productions no. 7, the Y can turn into a z.

9 $\overset{\cdot}{t'}$ $\overset{\cdot}{t}$ $\phi(t, t')=T\cup T'\cup\{z\}$

10 $\overset{\cdot}{t''}$ $\overset{\cdot}{t}$ $\phi(t, t'')=T\cup T''\cup\{z\}$

for all $\overset{\cdot}{t}\in T$; when the Y has become a z, these productions allow the labels in T' and T'' to turn into the corresponding labels in T. This is the only way that a terminal web can be derived in G_C.

The foregoing remarks constitute an outline of a proof that G_C does indeed generate '$\omega z\omega$'. (Note that except for no. 7, all productions have one-node α's. No. 7, too, could be modified to have a one-node α by omitting the Y; this would give it a disconnected β, but connectedness of sentential forms would still be guaranteed by the fact that the Y remains joined to every other node.)

3. WEB AUTOMATA

3.1 Web acceptors

By a *web acceptor* we shall mean a 9-tuple $M=(V, V_T, Q, q_I, q_F, \gamma, \delta, \mu, \sigma)$, where

V and V_T are as in the definition of a web grammar

Q is a finite set of *states*

$q_I, q_F\in Q$ are called the *initial* and *final states*

σ is a function from $V\times Q$ into $2^{V\times V\times Q\times 2^V\times 2^V}$

γ, δ and μ are functions from $V\times V\times Q$ into $2^{V\times Q}$

Informally, M operates on a given web ω as follows: initially, M is placed on some node of ω in state q_I. At any stage, let M be in state q and be located on node n, having label A.

(1) If n has a neighbor labelled C, and $\gamma(A, C, q)$ contains the pair (B', q'), M can relabel n with B', change to state q', and move to such a neighbor.

(2) If n has no neighbor labelled C, and $\delta(A, C, q)$ contains the pair (B'', q''), M can relabel n with B'' and change to state q''.

(3) If n has a neighbor m labelled C, and $\mu(A, C, q)$ contains the pair (B, q^*), M can merge some such m with n, label the resulting new node with B, and change to state q^*; its position is now on the new node.

(4) Finally, if $\sigma(A, q)$ contains the 5-tuple $(B_1, B_2, \bar{q}, V_1, V_2)$, M can split n into two joined nodes labelled B_1 and B_2, join each B_i to the neighbors of n whose labels are in V_i $(i=1, 2)$, and change to state $\bar{q}$; we regard its new position as on the node labelled B_1. In order to insure that σ never

disconnects ω, we shall assume that it applies only when n's neighbors all have labels in $V_1 \cup V_2$; this can be checked using δ repeatedly.

We say that M *accepts* ω if it ever enters state q_F. The set of webs on V_T that M accepts (from some starting position) will be denoted by $T(M)$.

The operation of M can be defined more precisely using the notion of an ID ('instantaneous description'), which is a triple (ω, n, q), where ω is a web, $n \in N_\omega$, and $q \in Q$. For example, in M's first type of move:

(1) If $f_\omega(n)=A$, and $N_{n,C}=\{x|(n,x)\in E_\omega,\ f_\omega(x)=C\}$, and $(B',q')\in \gamma(A,C,q)$, then (ω,n,q) can give rise to (ω',n',q'), where $n'\in N_{n,C}$; $G_{\omega'}=G_\omega$; and $f_{\omega'}=(f_\omega-\{(n,A)\})\cup\{(n,B')\}$,

and similarly for (2–4). It is customary to define automata using only a single transition function, but we have used four functions here $(\gamma, \delta, \mu, \sigma)$ to simplify the notation.

It should be noted that even if images under the functions γ, δ, μ, σ are always singletons, and even if for any given ID, only one of the functions has a nonempty image, M still cannot be regarded as completely deterministic. This is because the set $N_{n,C}$ need not be a singleton, and we have no way of specifying to which of n's C-labelled neighbors M moves when γ is applied, or which of them it merges with n when μ is applied.

3.2 Web-bounded acceptors

M will be called a *web-bounded acceptor* (WBA) if the functions μ and σ always have empty images. Thus a WBA on a web ω can change state, move around on ω, and relabel ω, but it cannot change the underlying graph of ω by merging or splitting nodes.

An important property of WBA's, which will be used repeatedly below, is that there exists a WBA that can visit all the nodes of any given web ω from any starting point n; it can then erase all the marks that it made on the nodes of ω in the course of this traversal, and return to n. This fact will be referred to as the Traversal Theorem; a proof of this theorem is given elsewhere (Milgram 1972).

Proposition 5. Let $a_1, \ldots, a_k$ (not necessarily distinct) be in V_T. Then there exists a WBA which, when placed at node n of any web on V_T, will enter a state q_s if the neighbors of n have exactly the labels $a_1, \ldots, a_k$, and a state q_f if not.

Proof. Our WBA operates as follows:

(1) Label n with a special symbol $A \notin V_T$ (using the function δ and the fact that n has no neighbor labelled A).

(2) Using γ, go to any neighboring node labelled a_1

(3) Using γ, relabel the a_1' as a_1' (where $a_1' \notin V_T$) and return to n (it is the *unique* neighbor labelled A).

(4) Repeat (2–3) for $a_2, \ldots, a_k$. If this procedure can be carried out successfully, use δ repeatedly (once for each label in V_T) to check that n now has no unprimed neighbors, and if so, enter state q_s.

(5) If the above fails at any stage, enter state q_f.

Note that our WBA can, if desired, also erase all special labels that it creates in the process of testing the neighborhood of n; in fact, it can use analogs of (2–4) to turn primed symbols back into unprimed symbols, and when none are left, it can restore n's original label.//

Proposition 6. Let α be a web on V_T having the node labels $A_1, \ldots, A_k$, all distinct. Then there exists a WBA which, when placed anywhere on a web ω (on V_T) in which $A_1, \ldots, A_k$ occur exactly once each as node labels, will enter a state q_s if the subweb whose nodes have these labels is α, and a state q_f if they do not.

Proof. In α, let the neighbors of A_i be $A_{i,1}, \ldots, A_{i,h_i}$, $1 \leqslant i \leqslant k$. By the Traversal Theorem, our WBA can find each A_i; by the method used to prove Proposition 5, it can test whether or not the labels $A_{i,1}, \ldots, A_{i,h_i}$ and no other A_j's, occur on the neighbors of A_i. If these tests succeed, our WBA enters q_s; if they fail at any point, it enters q_f. Here again, in either case, all marks made in the course of the tests can be erased.//

Proposition 7. Let $a_1, \ldots, a_k$ (not necessarily distinct) be in V_T. Then there exists a WBA which, when placed anywhere on a web ω (on V_T), will in turn find each instance (if any) of $a_1, \ldots, a_k$ as node labels of ω, relabel that instance with the distinct labels $A_1, \ldots, A_k$ ($\notin V_T$), then remove these labels and proceed to the next instance. When all instances have been found, the WBA will enter a state q_s; if there are no instances, it will enter a state q_f.

Proof. If $k=1$, an M_1 that does this can easily be described: it traverses ω, and as it finds each a_1, relabels it A_1, then relabels it a_1', then proceeds to the next unprimed a_1 and repeats the process. When the traversal is complete, if any a_1's were found, M_1 enters q_s; if none were found, it enters q_f. In either case, it can evidently also erase all primes when finished.

Suppose that we have an M_{k-1} that does the job for the labels $a_2, \ldots, a_k$; then we can define our desired M_k as follows: traverse ω; as each a_1 is found, relabel it A_1; now imitate M_{k-1} and find, in succession, all instances of $a_2, \ldots, a_k$ (whether primed or unprimed); when this is done, erase all marks made by M_{k-1} (they must be distinct from the prime), relabel A_1 as a_1', proceed to the next unprimed instance of a_1 and repeat the process. When finished, M_k enters state q_s unless it found no a_1's (or its M_{k-1}-imitator found no $a_2, \ldots, a_k$'s for any a_1), in which case it enters q_f.//

Proposition 8. For any web α on V_T, there exists a WBA which, when placed anywhere on a web ω (on V_T), will find any (one) instance of α as a subweb of ω, relabel it with unique, distinct node labels, and enter a state q_s; or if there is no such instance, enter a state q_f.

Proof. By Proposition 7, our WBA can in turn find each instance of α's node labels in ω; by Proposition 6, it can test each found instance to see if its nodes – which temporarily have unique, distinct labels – form the subweb α. Moreover, our WBA can stop at any such instance of α and enter q_s; or, if it completes the search without finding any instances, it can enter q_f. As usual,

it can erase any marks that it makes in the process, except for the new labels that were given to the selected instance of α.//

Proposition 8 shows that a WBA can find places in any web ω where a given web grammar production might be applicable. We shall next use this result to show that any web grammar can be imitated by a web acceptor.

3.3 Web acceptors can imitate web grammars

Theorem 9. For any web grammar G, there exists a web acceptor M such that $T(M)=L(G)$.

Proof. Given a web ω on V_T, we shall exhibit an M that applies reversals of productions of G to ω. Each time such an application is finished, M can check whether ω now consists of a single node labelled S, and if so, it can go into its accepting state. Clearly such an M accepts the original ω if and only if it can be parsed in G, which by Proposition 3 is our desired result.

Let $\pi=(\alpha, \beta, \phi)$ be the reversal of a production of G. By Proposition 8, M can pick any instance of α as a subweb of ω – or, if no instance exists, M can recognize the fact and try another production. Since M has relabeled the nodes of the selected instance distinctively, it can now check that conditions (2–3) for the applicability of π hold. Specifically, for each node m of α, our M can verify that its neighbors outside α all have labels in $\bigcup_{n\in N_\beta}\phi(n, m)$; and for each m_1, m_2 in N_α, it can verify that if any neighbor p of m_1 outside α has its label in $\bigcup_{n\in N_\beta}[\phi(n, m_1)\cap\phi(n, m_2)]$, then p is also a neighbor of m_2. Moreover, M can distinctively mark each neighbor of α (in $\omega-\alpha$) in a manner that specifies to which nodes of β that neighbor should be joined; this requires only as many distinct marks as there are subsets of N_β. Note that so far, M can be web-bounded.

Next, M can shrink α to a single node n by successively merging pairs of its nodes, using the function μ; note that n is now joined to all the neighbors of α in $\omega-\alpha$. Finally, M can expand n, using the function σ, so as to create the desired β. Specifically, let the nodes of β be $n_1, \ldots, n_k$. We first split n into n_1 and n'_2; label n_1 appropriately and join it to the appropriate neighbors of α; and join n'_2 only to those neighbors of α that still have to be joined to $n_2, \ldots, n_k$. Next, we split n'_2 into n_2 and n'_3; join each to the appropriate neighbors of α; join n_2 to n_1 if they are joined in β; and join n'_3 to n_1 if any of $n_3, \ldots, n_k$ are joined to n_1 in β. This process is repeated for $n'_3, n'_4, \ldots$; at the last step, n'_k is itself n_k, since it is already joined to just the desired neighbors of α and nodes of β. When the construction is complete, M does a final traversal; changes the distinctive labels of the nodes of β (which were needed for the construction process) into β's ordinary node labels; restores the original labels of the neighbors of α; and goes into a state that initiates the next production application. Note that M 'knows' at each stage whether or not ω has been parsed, since this can only happen when β is S and there are no neighbors.//

3.4 Web grammars can imitate web acceptors

Theorem 10. For any web acceptor M, there exists a web grammar G such that $L(G)=T(M)$.

Proof. G begins by generating a web of the form '$\omega z \omega$', as described in Section 2.3; here we want one of the ω's to have terminal labels, the other nonterminal labels (just omit (9) or (10) from the grammar of Section 2.3), and we want z to be a nonterminal as well. Next, G uses the following productions:

no.	α	β	ϕ
1	z t'	W (t', q_I)	normal

for all $t' \in T'$, the nonterminal labels. This production, which can only be used once, changes the z to a W and puts an 'automaton imitator' down at an arbitrary node of the primed copy of ω. We assume, in what follows, that M has vocabulary V_M (where $W \notin V_M$), terminal vocabulary T, and state set Q_M.

2 (A, q) C B' (C, q') normal

for all A, B', C in V_M and all q, q' in Q_M such that $(B', q') \in \gamma(A, C, q)$

3 (A, q) (B'', q'') $\phi((B'', q''), (A, q)) = V_M - \{C\}$

for all A, B'', C in V_M and all q, q'' in Q_M such that $(B'', q'') \in \delta(A, C, q)$

4 (A, q) C (B, q^*) $\phi((B, q^*), (A, q)) = \phi((B, q^*), C) = V$

for all A, B, C in V_M and all q, q^* in Q_M such that $(B, q^*) \in \mu(A, C, q)$. This is not a legal production unless the neighbors of (A, q) and C have label sets such that only common neighbors have common labels. However, we can imitate it by a sequence of legal productions as follows:

(*a*) (A, q) C A_1 C_2 normal

This attaches special labels to the nodes that are to be merged

(b) A_1 D A_1 D_3 normal; for all $D \in V_M$

This attaches the subscript 3 to the label of each neighbor of A_1

(c) A_1 A_4 $\phi(A_4, A_1) = V_2 \cup V_3$

When every neighbor of A_1 has subscript either 2 or 3, we can relabel it as A_4

(d) A_4 C_2 A_4 C_5 normal

When A_1 has become A_4, C_2 can become C_5

(e) C_5 D C_5 D_6 normal; for all $D \in V_M$

(*f*) C_5 D_3 C_5 D_7 normal; for all $D \in V_M$

These attach subscripts 6 to neighbors of C_5 that are not neighbors of A_4, and 7 to neighbors of both C_5 and A_4

(g) C_5 C_8 $\phi(C_8, C_5) = V_6 \cup V_7$

When every neighbor of C_5 has subscript either 6 or 7, we can relabel it as C_8

(h) A_4 C_8 B_9 $\phi(B_9, A_4) = V_3 \cup V_7$; $\phi(B_9, C_8) = V_6 \cup V_7$

This is now legal since neighbors of A_4 only have subscripts 3, those of C_8 only have subscripts 6, and common neighbors have subscripts 7

(i) B_9 — $D_{3,6 \text{ or } 7}$ $\quad$ B_9 — D $\quad$ normal; for all $D \in V_M$

All neighbors of B_9 now get their subscripts removed

(j) B_9 $\quad$ (B, q^*) $\quad$ $\phi((B, q^*), B_9) = V_M$

When no neighbors of B_9 have subscripts, we can give it its final label.

5 (A, q) $\quad$ $(B_1, \bar{q})$ — B_2 $\quad$ $\phi((B_1, \bar{q}), (A, q)) = V_1 \cup \{W\}$;
$\phi(B_2, (A, q)) = V_2 \cup \{W\}$

for all A, B_1, B_2 in V_M, all $q, \bar{q}$ in Q_M, and all V_1, $V_2 \subseteq V_M$ such that $(B_1, B_2, \bar{q}, V_1, V_2) \in \sigma(A, q)$. These groups of productions can evidently imitate M. Specifically, suppose that these productions can produce the sentential form θ from the initial web consisting of the primed copy of ω with a single node m labelled (t', q_I), so that θ has a single node n with label of the form (A, q). Then clearly M, from the ID (ω, m, q_I), can give rise to the ID (θ, n, q); and conversely, if M can do this, then these productions can take the initial web into θ.

6 W — (A, q_F) $\quad$ Z — A $\quad$ normal

for all $A \in V_M$. If the M-imitator ever enters state q_F, the W can change to a Z and the pair can be replaced by its first term; further imitation is now impossible.

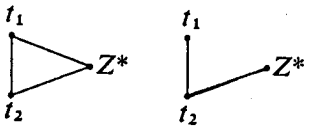
7 Z — A $\quad$ Z $\quad$ $\phi(Z, Z) = \phi(Z, A) = V$

for all $A \in V_M$. The Z can now destroy what remains of the primed copy of ω. These productions, too, can be broken down into legal ones using the method employed at step (4).

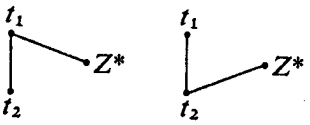
Finally, the Z destroys itself as soon as it has no neighbors with labels in V_M, using the productions described immediately below; thus only the terminal copy of ω is left. Clearly, G can only generate a terminal web ω if the M-imitator accepts the primed copy of ω; and conversely, G can generate any such ω.

To destroy the Z, we use the following productions:

8 Z $\quad$ Z^* $\quad$ $\phi(Z^*, Z) \cap V_M = \varnothing$

9 

normal

10 

normal

11 t — Z^* $\quad$ t $\quad$ $\phi(t, t) = V$; $\phi(t, Z^*) = \varnothing$

for all t, t_1, t_2 in T; note that the last production can apply only if Z^* has no neighbors but t. To see that these productions can in fact always destroy the Z, it suffices to show that if the Z^* is still joined by arcs to $k > 1$ nodes of ω, we can always erase one of these arcs; this eventually gets us to a situation

where (11) applies. If two of these nodes of ω are neighbors, (9) does this. If not, they are in any case connected, and we can reduce the distance between them (that is, the length of the shortest path between them), using (10), until they become neighbors and (9) applies.//

4. NORMAL FORMS FOR WEB GRAMMARS

The equivalence between web grammars and web acceptors (Theorems 9–10) makes it possible to establish various normal results for web grammars: any web language has a grammar whose productions are all of some simple type(s).

4.1 A two-node normal form

Our first normal form result involves the sizes of the subwebs used in productions. We have seen that any G has the same language as some M, and that this M in turn has the same language as a certain other G; moreover, in our construction of the latter G, all productions used had α's and β's with at most two nodes each, except for (7) at the end of Section 2.3, and (9–10) at the end of Section 3.4. We shall now show that these productions can in turn be imitated by two-node productions; we will then have proved

Theorem 11. For any web grammar G, there exists a web grammar G' such that $L(G')=L(G)$, and $|N_\alpha|$, $|N_\beta|$ each $\leqslant 2$ for all productions (α, β, ϕ) of G'.//

To see that (7) can be imitated, we proceed as follows, for all $t \in T$:

7a $\quad \underline{Y \quad S} \qquad \underline{Y^* \quad (S, t)} \qquad$ normal

This picks an S to be split, and also specifies the symbols (t' and t'', corresponding to t) into which it will be split. The Y is starred to insure that only one application of (7a–g) can be made at a time.

7b $\quad \underline{(S, t) \quad x} \qquad \underline{(S, t) \quad x^*} \qquad$ normal

for all $x \neq Y^*$. This stars all the neighbors of the (S, t).

7c $\quad \underline{Y^* \quad (S, t)} \quad \underline{(Y, t)} \qquad \phi((Y, t), Y^*)=V;\ \phi((Y, t), (S, t))=\{S^*\} \cup T'^* \cup T''^*$

When all neighbors have been starred, the (S, t) merges with the Y^*.

7d $\quad \underline{(Y, t)} \qquad \underline{(Y', t) \quad t'} \qquad \phi((Y', t), (Y, t))=V;\ \phi(t', (Y, t))=\{S^*\} \cup T'^*$

7e $\quad \underline{(Y', t)} \qquad \underline{Y'' \quad t''} \qquad \phi(Y'', (Y', t))=V;\ \phi(t'', (Y', t))=\{S^*\} \cup T''^*$

The (Y, t) then splits off t' and t'', which are joined to the appropriate neighbors of the original S.

7f $\quad \underline{Y'' \quad x^*} \qquad \underline{Y'' \quad x} \qquad$ normal

The stars can now be removed.

7g $\quad Y'' \qquad Y \qquad \phi(Y, Y'')=V-(\{S^*\} \cup T'^* \cup T''^*)$

When all stars have been removed, the Y'' can turn into a Y and the process can begin again.

Similarly, to imitate (9), we use

9a $t_1 \quad Z^*$ $\quad t_1^{(1)} \quad Z^{(1)}$ normal

9b $t_2 \quad Z^{(1)}$ $\quad t_2^{(2)} \quad Z^{(2)}$ normal

9c $t_1^{(1)} \quad t_2^{(2)}$ $\quad t_1^{(3)} \quad t_2^{(4)}$ normal

These productions pick and mark the t_1 and t_2 to which (9) will be applied; they can only be used once. If the chosen t_1 and t_2 are not neighbors, the grammar blocks after (9b); this can be avoided, if desired, by allowing $t_2^{(2)} \quad Z^{(2)}$, in the absence of $t_1^{(1)}$ as a neighbor of $t_2^{(2)}$, to be rewritten as $t_2 \quad Z^{(0)}$, and $t_1^{(1)} \quad Z^{(0)}$ to be rewritten as $t_1 \quad Z$.

We now mark all neighbors of $t_1^{(3)}$ with superscript 5's, and all those of $t_2^{(4)}$ with 6's, except that all common neighbors are marked with 7's, and the $Z^{(2)}$ is not marked; we then merge the t_1 and t_2 into a single node (t_1, t_2). All this can be done using rules similar to (4b–h) in the proof of Theorem 10.

9d $(t_1, t_2) \quad t_1^{(8)} \quad t_2^{(9)}$ $\quad \phi(t_1^{(8)}, (t_1, t_2)) = V^{(5)} \cup V^{(7)}$;
$\phi(t_2^{(9)}, (t_1, t_2)) = V^{(6)} \cup V^{(7)} \cup \{Z^{(2)}\}$

The (t_1, t_2) then resplits into a t_1 and a t_2 each of which is joined to its former neighbors, except that the t_1 is no longer joined to the Z.

9e $t_1^{(8)} \quad t^{(5) \text{ or } (7)}$ $\quad t_1^{(8)} \quad t$ normal

9f $t_2^{(9)} \quad t^{(6) \text{ or } (7)}$ $\quad t_2^{(9)} \quad t$ normal

9g $t_1^{(8)} \quad t_2^{(9)}$ $\quad t_1 \quad t_2$ $\quad \phi(t_1, t_1^{(8)}) = V - (V^{(5)} \cup V^{(7)})$;
$\phi(t_2, t_2^{(9)}) = V - (V^{(6)} \cup V^{(7)})$

9h $Z^{(2)}$ $\quad Z^*$ $\quad \phi(Z^*, Z^{(2)}) = V - (V^{(2)} \cup V^{(4)} \cup V^{(9)})$

The neighbor superscripts can now be erased; when all are gone, the t_1 and t_2 can lose their superscripts; when this happens, the $Z^{(2)}$ can turn back into a Z^*.

Finally, to imitate (10), we proceed in much the same way as for (9). Here (9b–c) are replaced by

$t_1^{(1)} \quad t_2$ $\quad t_1^{(3)} \quad t_2^{(2)}$ $\quad \phi(t_1^{(3)}, t_1^{(1)}) = V$;
$\phi(t_2^{(2)}, t_2) = V - \{Z^{(1)}\}$

The remaining steps are identical; note that (9d) now has the effect that the $Z^{(2)}$ is joined to the t_2 rather than to the t_1.

4.2 An 'embedding-free' normal form

We shall now show that if the α's and β's in productions are allowed to have up to three nodes each, we can require all embeddings to be 'trivial', in the sense that for all productions (α, β, ϕ), one of the following holds:

(a) $N_\alpha = \{m\}$, $N_\beta = \{n\}$

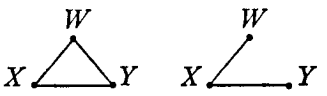
(b) $N_\alpha = \{m\}$, $N_\beta = \{n_1, n_2\}$; $\phi(n_1, m) = V$, $\phi(n_2, m) = \varnothing$ (so that n_2 is joined to nothing but n_1)

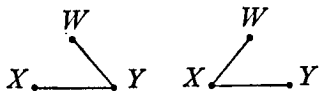
(c) $N_\alpha = \{m_1, m_2\}$, $N_\beta = \{n\}$; $\phi(n, m_1) = V$, $\phi(n_1, m_2) = \varnothing$ (so that the production cannot be applied unless m_2 is joined to nothing but m_1)

(d) $N_\alpha=\{m_1, m_2\}$, $N_\beta=\{n_1, n_2\}$, or $N_\alpha=\{m_1, m_2, m_3\}$, $N_\beta=\{n_1, n_2, n_3\}$; and in either case, $\phi(n_i, m_j)=V$ for $i=j$, $=\varnothing$ for $i\neq j$ (i.e., the embedding is normal)

As in Section 4.1, it suffices to show that the productions of G_C (at the end of Section 2.3) and the productions of Theorem 10 can all be imitated by productions of the above types. Now in G_C, all productions are of type (a) except for (1), (2), and (7); but (1–2) are special cases of (5) in Theorem 10, and we have already seen how (7) can be imitated using only productions of types (a) and (d) together with (4–5) of Theorem 10. Similarly, in Theorem 10, (7) is a special case of (4), (11) is of type (c), and (9–10) can be imitated using only (a, d, 4, 5).* It thus remains only to show how (4–5) can be imitated using (a–d).

Regarding (4), when we wish to replace X Y by Z, we may assume that the neighbors of X and Y have distinctive labels, since (4a–g) in the proof of Theorem 10 are all of types (a) and (d). Let V_x and V_y be the sets of labels of neighbors of X and Y, respectively, so that any neighbor having a label in $V_x\cap V_y$ is a common neighbor. We can now imitate (4) by the following productions:

4a  normal

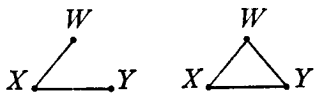
4b  normal; for all W in V_y-V_x

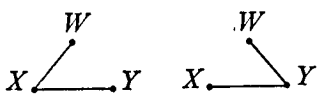
4c X Y $\quad$ Z $\quad$ $\phi(Z, X)=V$; $\phi(Z, Y)=\varnothing$

Here (4a–b) can be used to turn common neighbors or neighbors of Y alone into neighbors of X alone; when this has been done, (4c) applies to merge X and Y.

Similarly, suppose that we wish to replace Z by X Y, where $\phi(X, Z)=V_1$, $\phi(Y, Z)=V_2$, with every neighbor of Z in either V_1 or V_2. We can do this using

5a Z $\quad$ X Y $\quad$ $\phi(X, Z)=V$, $\phi(Y, Z)=\varnothing$

5b  normal; for all W in $V_1\cap V_2$

5c  normal; for all W in V_2-V_1

To insure that (5b–c) go to completion, we can assume that the labels in $V_1\cap V_2$ were given special marks before applying (5a), and that X and Y are given specially marked intermediate labels; we can then remove the marks

* (9g) can be done in three steps, two of type (4), the third of type (d).

from the W's in $V_1 \cap V_2$ as they become adjacent to Y under (5b). When X has no marked neighbors in $V_1 \cap V_2$, and no neighbors in $V_2 - V_1$, we can erase its mark, and then erase Y's mark.

The discussion in the preceding paragraphs gives us

Theorem 12. For any web grammar G, there exists a web grammar G' such that $L(G') = L(G)$ and all productions of G' satisfy (a), (b), (c) or (d).

4.3 More general types of embeddings

In our original definition of a web grammar, we allowed embeddings to be defined only in terms of neighbor labels: nodes of β are joined only to neighbors of nodes of α, and only to those whose labels lie in specified sets. This definition could have been generalized in various ways; for example, we could have allowed nodes of β to be joined to neighbors of neighbors of nodes of α, provided these have specified labels. As another example, we could join nodes of β to nodes that are neighbors of (say) two or more nodes of α, irrespective of their labels.

We shall not attempt to formulate a 'most general possible' notion of embedding here. However, the following seems to be reasonably general: given that the nodes of α have been assigned unique, distinctive labels, let N be any set of nodes such that there exists a WBA which

(a) from initial position on ω at a node r of N, enters a state q_s at r with ω unchanged;

(b) from any other initial position r', enters a state q_f at r' with ω unchanged.

In 'recognizing' nodes of N, the WBA can mark ω, but it must be able to erase all marks after making its decision. Then one could use such sets N to define embeddings: we join each $n \in N_\beta$ to the nodes of $\omega - \alpha$ that belong to some set N_n. Readily, our present definition is a special case of this one, since a WBA can determine whether or not r is a neighbor of some $m \in N_\alpha$ and has label in some set $\phi(n, m)$. The examples given in the first paragraph are also special cases.

This approach will not be pursued further, but we should point out that it is not really more general than our present definition. In fact, a grammar based on our definition can imitate the proposed WBA's and assign special labels to the nodes in the sets N. Furthermore, even if these nodes are not neighbors of α, we can turn them into neighbors of α using productions that shift edges (see the end of Section 3.4). Thus embeddings based on sets of nodes that are accepted by WBA's can be imitated by embeddings based on neighbor labels.

5. ISOTONIC WEB GRAMMARS

Now that it has been established that web grammars are equivalent to web acceptors, one can ask whether there is a natural class of web grammars that is equivalent to the class of web-bounded acceptors. For string and array grammars (Rosenfeld 1971, Milgram and Rosenfeld 1971), it turns

out that the 'monotonic' (or equivalently: context-sensitive) grammars are such a class. We do not know whether the analogous assertion is true for web grammars. We can, however, show that an interesting class of web grammars, the *isotonic* grammars, is in a sense equivalent to the class of WBA's.

By an *isotonic web grammar* (IWG) is meant a web grammar in which, for all productions (α, β, ϕ),

(a) $G_\alpha = G_\beta$

(b) $\phi(n, m) = \varnothing$ whenever $n \neq m$

Evidently an IWG can only relabel subwebs of its initial web, but can never change a web's underlying graph. In order to obtain nontrivial languages, we must therefore define the language of an IWG somewhat differently from that of a general web grammar. Specifically, in analogy with (Rosenfeld 1971, Milgram and Rosenfeld 1971), we define $L_I(G)$, where G is an IWG, as the set of terminal webs that can be derived from any initial web ω_0 having one node labelled S, and all other nodes labelled with a special nonterminal label $\#$.

Theorem 13. For any WBA M, there exists an IWG G such that $L_I(G) = T(M)$.

Proof. G first rewrites the initial S as (q_I, x, x), where x is an arbitrary label in V_T. It then imitates a WBA that traverses the initial web and changes each $\#$ to a label of the form $(\ , y, y)$, where $y \in V_T$; this can be done using productions of the forms

(1) $(q, a, a) \quad \# \qquad (0, a^*, a^*) \quad (q^*, b, b)$ where $b \in V_T$ is arbitrary

(2) $(q, a, a) \quad (0, c, c) \qquad (0, b', b') \quad (q', c, c)$

(3) $(q, a, a) \qquad\qquad (q'', b'', b'')$

G then erases all marks made in the course of the traversal; it has now created an arbitrary web on the vocabulary $\{(0, y, y) \mid y \in V_T\}$, except for one node with label (q, x, x), for some $x \in V_T$, the (arbitrary) position of the initial S. Next, G uses productions of the forms

(4) $(q, a, x) \quad (0, c, y) \qquad (0, b', x) \quad (q', c, y)$

(5) $(q, a, x) \qquad\qquad (q'', b'', y)$

to imitate M on the web consisting of the *second terms* of the triples. If this imitation ever enters its final state, G imitates a WBA that traverses the web, replaces all triples by their *third terms*, and erases all traversal marks; the final result is thus a terminal web. Clearly G can generate such a terminal web if and only if that web is in $T(M)$.//

Theorem 14. For any IWG G, there exists a WBA M such that $T(M) = L_I(G)$.

Proof. By the proof of Theorem 9, M can apply the reversals of the productions of G to a given web. At any stage, it can traverse the web and check whether its node labels are all $\#$'s except for a single S, and if so it can accept the web. Clearly if such an M starts with the terminal web ω, it accepts ω if and only if ω can be parsed in G.//

One could also define other restricted types of web grammars, for example, a 'context-free' type in which the α in any production has only a single node (and the production is always applicable; see Section 2.3); or a 'linear' type

in which, in addition, at most one node of any β has a nonterminal label. For some properties of such grammars, particularly when restricted types of embeddings are used, see Pavlidis (1972) and Abe *et al.* (in press). As in the array grammar case (Rosenfeld and Milgram 1971), however, it appears to be difficult to define simple classes of automata that are equivalent to such types of grammars.

The web grammars considered in this paper involve only undirected webs; but it is straightforward to generalize our results to directed webs by distinguishing the two types of arcs ('in 'and 'out') at each node. For example, a directed web production has two embeddings associated with it, a $\phi_{\text{in}}(n, m)$ that specifies which 'in' arcs of m are joined to n, and a $\phi_{\text{out}}(n, m)$ for the 'out' arcs. A directed web acceptor must be allowed to move in either direction along an arc, just as in the special case where the directed web is a string, we allow it to move either right or left.

Acknowledgements

The support of the Office of Computing Activities, National Science Foundation, under Grant GJ–754, is gratefully acknowledged, as is the help of Mrs Eleanor B. Waters in preparing the manuscript of this paper.

REFERENCES

Abe, N., Mizumoto, M., Toyoda, J. & Tanaka, K. (in press). Web grammars and several graphs. *J. comput. and sys. Sci.*

Milgram, D.L. (1972) Web automata. University of Maryland Computer Science Center Technical Report 182.

Milgram, D.L. & Rosenfeld, A. (1971) Array automata and array grammars. *Proc. IFIP 71 International Congress*, Booklet TA–2, 166–73. Amsterdam: North-Holland.

Montanari, U.G. (1970) Separable graphs, planar graphs, and web grammars. *Information and Control* **16**, 243–67.

Pavlidis, T. (1972) Linear and context-free graph grammars. *J. Ass. comput. Mach.*, **19**, 11–22.

Mylopoulos, J. (1971) On the relation of graph automata and graph grammars. University of Toronto Dept. of Computer Science Technical Report 34.

Pfaltz, J.L. (1972) Web grammars and picture description. *Computer Graphics and Image Processing* **1**, (in press).

Pfaltz, J.L. & Rosenfeld, A. (1969) Web grammars. *Proc. 1st Intl. Joint Conf. on Artificial Intelligence*, Washington, D.C., May 1969, 609–19.

Rosenfeld, A. (1971) Isotonic grammars, parallel grammars, and picture grammars. *Machine Intelligence 6*, pp. 281–94 (eds Meltzer, B. & Michie, D.). Edinburgh: Edinburgh University Press.

Rosenfeld, A. & Milgram, D.L. (1971) Array automata and array grammars, 2. University of Maryland Computer Science Center Technical Report 171.

Rosenfeld, A. & Strong, J.P. (1971) A grammar for maps. *Software Engineering*, vol. 2, pp. 227–39 (ed Tou, J.T.) New York: Academic Press.

Shank, H.S. (1971) Graph property recognition machines. *Mathematical Systems Theory* **5**, 45–9.

Shaw, A.C. (1972) Picture graphs, grammars, and parsing. *Frontiers of Pattern Recognition* (ed. Watanabe, S.). New York: Academic Press.

17

Utterances as Programs

D. J. M. Davies and S. D. Isard
Theoretical Psychology Unit
University of Edinburgh

The digital computer is a good mimic. It can be, and has been, used to simulate a variety of phenomena, among them the use of natural language. Two major virtues of these model-building exercises are the standards of rigour they enforce – if it's not precise, you can't program it – and the fact that once a complex idea has been embodied in a program, its consequences, or some of them, at least, become immediately apparent when the program is run. In principle, of course, one does not actually need a machine to discover what a program will do, but in practice it is a great help.

Computers, or rather computing systems, also have another role to play in the study of language. This has to do with the fact that they are in a sense language users themselves. In so far as English utterances are uttered for their effects, they can be compared with the statements of a programming language. Computing Science provides concepts for describing the effect of running a program on a computer. As Longuet-Higgins (1970) points out, these can be brought to bear on the discussion of English, to describe the effect of utterances on the hearer's state, and also the effect of this state on the way in which utterances are interpreted.

These matters are not dealt with in the ordinary formal semantics of predicate languages, although there have been some developments from predicate language semantics in this general direction. In particular, there is the concept of a 'point of reference' in modal logic (see, e.g. Scott 1970), which can in fact be viewed as something akin to the state of a system.

No one, to our knowledge, has yet proposed in the context of modal logic that points of reference should alter as a result of having sentences interpreted with respect to them. The idea may even sound a bit bizarre at first, but less so when one considers that the point of reference is often supposed to encode, among other things, usually too numerous to mention, states of knowledge and belief and facts about what has already been said. These things certainly change as a conversation proceeds. Also, it is in these changes that the

explanation lies of why people say one thing rather than another. Something very like a system with changing points of reference emerges in Scott and Strachey's (1971) discussion of a semantics for computing languages.

COMPUTERS AND HEARERS

In this paper we are going to concern ourselves primarily with the interaction between utterances and the state of the hearer. This is not because we regard the speaker and the surrounding world as unimportant, but simply because the analogy that we are pursuing here, that between computer and hearer, is of no particular help with regard to them. The reason why it is useful in the first place is that, having noted an outward similarity, we can use our knowledge of the internal workings of the machine to raise hypotheses about the internal workings of the human. If we extend the analogy in the obvious way and match speaker with programmer, the outward similarity remains, but now both are human and their internal workings are about equally mysterious.

Though it ought to be obvious, we shall take the precaution of stressing that we are concerned with 'internal workings' on a programming level, and not with anatomy or hardware. The logic of programs is largely independent of the particular machines which carry them out. Identical programs in, say, FORTRAN can be run on machines of widely differing design. There may come a day when we have sufficiently well-specified language programs and sufficiently good neurophysiological wiring diagrams to raise the question of how such programs might be implemented on such machines. That day is not at hand.

DISOBEDIENCE

Before turning to some English sentences, and what we take to be the computational analogues of their effects, we must recognise a distinction not normally present in computing languages between what we shall call 'expected' and 'actual' effects.

A common manifestation of this distinction is that people often do not do what they are told, not through any misunderstanding, but simply because they do not want to. The reason evidently lies in the fact that a hearer has his own needs and goals independent of the speaker's. Computing systems do not have their own aims in life, but we may still be able to gain some insight by asking where in the computational process we might introduce refusals to obey instructions if we were trying to simulate human reactions.

Suppose, for instance, that we were to give the number 7 the status of an obscenity, and we wanted our machine to, prudishly, refuse to print it, while behaving normally in other respects. Its refusal to tell us the square root of 49 or the sum of 66 and 11 could not then be a result of ignoring the instructions altogether. It must actually do the calculations involved in order to discover that the forbidden digit appears in the answer. A slightly more

complex situation would arise if we wished to instil a 'need' to keep the sum of the variables X and Y between 0 and 10. Here the machine must do something that will amount to trying out its instructions using copies of X and Y, and then running them with the real X and Y if it turns out to be safe to do so.

In both cases, the point of refusal comes after the instructions have been 'understood', and, in fact, tested out. In systems which compile programs – translate them into an internal machine language – before running them, we can insert some testing, and a possible refusal, between compiling and running to achieve the effect we want. *Compiling* then appears to be analogous to understanding what has been said, and *running* to accepting it and carrying out any required actions. The situation is actually a bit more complicated than this, because we would maintain that in the use of English, some 'compiled' instructions are run before the 'compiling' process as a whole is finished. Before enlarging on this point, we would like to note that our model is given a certain psychological plausibility by the intuition that although it is generally a simple matter to refuse to obey a command, it is rather difficult to consciously refuse to understand it; that is, we have located the 'point of refusal' in the machine at what seems to be the place at which a hearer gains conscious control over his activities. There are, of course, times when it is necessary to make a conscious effort to understand (as, for instance, when reading the works of Kant). But such situations are relatively rare in the everyday use of a language that one knows well. Perhaps it is loss of conscious control over one's compiler that really constitutes 'knowing' a language.

Our main reason for postulating some 'running' in the midst of the 'compiling' process is that in a sentence such as

Will the King of France have kippers for breakfast tomorrow? (1)

one is aware of the anomaly well before the end. However, this anomaly can only come to light in the attempt to identify the referent of 'the King of France', which, in our scheme, involves running the relevant bit of program We think that, subject to the qualification below, the rejection of a sentence on the grounds of 'false presuppositions' is best represented by the failure of some bit of program which the hearer attempts to run during the compiling of the main program specified by the speaker. Our qualification involves the situation, already mentioned, when conscious effort is needed to understand a sentence. In this situation, the difficult bit of program is not run on the spot, but is inserted, as program, into the translation, to be run after the 'point of refusal'. If given the order

Tell me the fifth digit in the decimal expansion of π. (2)

one could work out the number 5, but refuse to say it, or one could refuse to work it out in the first place.

However, if the difficult bit of program actually proves to be impossible (e.g. 'Tell me the last digit in the decimal expansion of π') the hearer still feels that the sentence has a false presupposition, even though the program was attempted under conscious control.

This option to postpone the running of a function can be represented computationally with the aid of 'closure functions'. These are functions which are obtained by 'fixing' some of the variables in other functions. In the notation of POP-2, our local programming language (Burstall, Collins, and Popplestone 1971), *plus*(%1%) can be used to represent the (one argument) closure function obtained by fixing the second argument of the (two argument) addition function at 1; that is, it is the successor function.

Now suppose that we wish to translate the sentence

Tell me how many marbles are in the jar. (3)

Let *h* stand for the translation of 'how many marbles are in the jar'. That is, *h* is a piece of program which produces the appropriate number when it is applied. If there are not many marbles in the jar, then *h* will be easy to run, and (3) will translate to

print(%*apply*(*h*)%) (4)

This is a function, of no arguments, obtained by applying *h*, and then fixing the argument of *print* at the resulting value. It might, for instance, be the function *print*(%3%). Noting that the function *print* is equivalent to **lambda** *x*; *print*(*x*) **end**, we can rewrite (4) as

(lambda *x*; *print*(*x*) **end)**(% *apply*(*h*) %) (5)

On the other hand, if there are a lot of marbles and *h* is hard to evaluate, we can compile, instead of (5):

(lambda *x*; *print*(*apply*(*x*)) **end)**(% *h* %) (6)

Here, **lambda** *x*; *print*(*apply*(*x*)) **end** is a function of one argument *x* which must itself be a function. It first evaluates *x* and then prints the result. Therefore, (6) is a function which first evaluates *h* and then prints the result; that is, the evaluation of *h* has been postponed until the running of the final function.

The use of the word 'please' in English provides another distinction between the main function being compiled, and the preliminary functions which are run during the compiling process. 'Please' covers acts over which the hearer has control, and hence attaches itself to whole sentences, and to clauses which translate into programs to be run later on, rather than to, say, noun phrases, whose translations are to be run as soon as possible. In

Please tell me the answer. (7)

the speaker is being polite about the telling, but the question of politeness does not arise with respect to his instruction to *determine* the answer.

STATEMENTS, QUESTIONS AND COMMANDS

The sentences of English, and many other languages, fall into three major categories: statements, questions, and commands. In order to discuss the effects of these on the hearer, we shall introduce three kinds of effect that a function may have in a computation. We shall conduct our discussion initially in terms of POP-2, but a similar analysis applies to other languages, for example LISP (McCarthy, Abrahams, Edwards, Hart and Levin 1962)

and PLANNER (Hewitt 1971). The only feature of POP-2 that requires mention at this point is that it has a *stack*, from which functions get their inputs, and on which they leave their results.

For readers unfamiliar with this concept, the stack can be thought of as a sort of letter drop, a hollow tree, where pieces of program leave items to be collected by other pieces. For instance, a function *square*, when 'called', will go to the stack, take the item it finds there and multiply it by itself, and then put the answer back on the stack. In POP-2, a program may have the following three kinds of effects:

(1) internal side effects: the change is in the contents of some locations, other than the stack, within the machine.

(2) external side effects on peripheral devices: the internal state of the machine does not change, but the world around it does.

(3) results: there is a change in the contents of the stack.

For example, the function

function *setup*; $1 \rightarrow x$; $2 \rightarrow y$; **end**

will have only internal side effects. The values of x and y will now be 1 and 2 respectively. The function *print* has only external side effects. The function *square* mentioned above will only return a result.

STATEMENTS

We wish to distinguish in our model hearer between fairly permanent aspects of mental structure which might be classed as 'knowledge' or 'world view', and the more fleeting phenomenon of 'attention'. To know a fact is not the same thing as to be thinking about it. The only computational correlate for *attention* that we shall deal with at any length is the state of the stack. Although this serves our purposes for the moment, it will obviously be inadequate in the long run. We will not go further into the matter here, but we think that the 'local state' mechanism of PLANNER might help us to model the aspects of attention involved in 'imagining' and 'recalling the past'. In abstract terms, we suspect that the distinction made by Scott and Strachey between 'environment' and 'state' will prove useful.

Although every utterance must engage the hearer's attention to some extent in order to serve its purpose, *statements* are also intended to have other internal effects on the hearer's beliefs. The precise nature of the change that a statement can produce must, of course, depend on the sort of 'belief structure' that the hearer has in the first place. In particular, it will depend on the hearer's 'model' of the speaker.

PROCEDURES

One sort of structure that is sometimes used in 'question-answering' systems consists of a set of predicate calculus formulas. When the system is 'told' an English sentence, it adds a predicate calculus translation of the sentence to its set of 'beliefs'. Hewitt (1971) argues, however, that one must be able to

discuss not just 'facts', but also techniques for *using* them. This idea also informs the work of Winograd (1971), whose system uses *procedures* (pieces of program) in place of formulas with quantifiers. His only 'facts' are atomic formulas.

In predicate calculus terms, he can be viewed as trading axioms for rules of inference. A rather obvious example of a sentence that corresponds better to a rule of inference than to a formula is the English rendering of the rule of *modus ponens.*

A more interesting case, from our point of view, is

If you need any paper-clips, you have to apply in triplicate to the sub-dean. (8)

This provides the hearer (or reader) with information to be used at the time when his paper-clips run out. Notice that it does not appear to be intended to help the hearer decide whether to apply to the sub-dean; only in a joke would anyone conclude that he did not need any paper-clips by noting that he did not have to apply to the sub-dean. Predicate calculus formulas, with their usual semantics, cannot carry this sort of 'instructions for use'.

Whether or not a rule of inference can fully capture the meaning of (8) is not really our concern here. We should like to point out, however, that its use is matched rather nicely by an 'antecedent theorem' in Hewitt's PLANNER system. This is a bit of program which is called into play whenever a specified condition is satisfied within the system. For the case of (8), the condition is a need for paper-clips; the 'theorem' will generate a reminder to apply to the sub-dean.

These 'theorems' also allow us to cope with situations where statements appear to have external affects. For instance, a speaker might use

I want a screw-driver. (9)

to actually obtain a screw-driver, rather than just to inform the hearer of his wants. If we postulate a 'get him what he wants' procedure within the hearer, this can be set off by the knowledge, however obtained, that the speaker wants a screw-driver. The actual production of one is then seen as just a secondary by-product of accepting (9).

If (9) is spoken by a small child to his mother, the effect of the statement itself is that the mother believes him, if she does in fact decide to. But, as a result of believing him, she is left in a state in which she automatically generates a program to provide a screw-driver. She must then decide whether or not to run this program on his birthday.

However, if (9) is spoken by a customer in a hardware store, the store-keeper will probably take it as an instruction in the first place, virtually identical to

Get me a screw-driver. (10)

We suggest that this is because, in the circumstances, he accepts the truth of (9) without thinking about it, thus triggering those procedures which produce programs for satisfying customers' wants; it is only these programs

which are considered consciously. If the store-keeper were to reply 'No!', this would almost certainly mean 'You cannot have one.', not 'I do not believe you.'

Notice that the word 'please' tends to encourage this effect, letting the statement slip by unnoticed, and concentrating attention on its by-products, thus gaining the net effect of a command, as in

I want a screw-driver, please. (11)

QUESTIONS

Questions specify functions that produce results. The intended effect of a question is that its function should be run, and the results delivered to the questioner. We have purposely separated the function, which leaves its results on the stack, from the act of supplying the answer to the questioner. Our reason is that we see this as one of those cases in which a function is run during the translation process, if possible. It is often hard to prevent yourself from thinking of the answer to a question, even if you decide not to say it.

It might be objected that this analysis depends too heavily on the stack, a device peculiar to POP-2 and a few other systems, and perhaps psychologically suspect. However, in many computing systems which do not have stacks as such, there is still some sort of *communication area*, and it makes sense to distinguish between internal effects here and those in the computer's more private regions. The role that the communication area is meant to play in our model is that of 'focus of attention' or 'what has been brought to mind', which, although difficult to define precisely, is still an intuitive concept.

Definite noun phrases also seem to correspond to functions which produce results. They bring their referents to the hearer's attention, or, in POP-2 terms, put a pointer to the referent on the stack. Thus the syntactic similarity between, say

Which book is on the shelf? (12)

and

the book which is on the shelf (13)

is matched by a similarity in effects. Both (12) and (13) translate to a program, meant to be run at translation time, which leaves a mention of the appropriate book on the stack. The difference between them is that (12) puts this pointer directly into a run-time instruction to 'say this', whereas (13) leaves it to be inserted into whatever program corresponds to the larger sentence in which it appears. This could, of course, turn out to be the same as the one (12) gives, as in

Tell me which book is on the shelf. (14)

But it might instead be

Bring me the book which is on the shelf. (15)

or

My aunt recommended that I buy the book which is on the shelf. (16)

Note that all of (12) to (16) will be nonsensical to a hearer who does not share the presupposition that there is a book on the shelf, and therefore cannot construct a 'pointer' to it.

This parallel treatment of questions and relative clauses is implemented in the tic-tac-toe program discussed later on, for constructions such as

What will you do? (17)

and

what you will do. (18)

There is a class of sentences such as

Would you shut the window? (19)

and

Can you please remove your Pekinese from my lap? (20)

whose syntactic form is interrogative, but which do not appear to function as questions. They call for action, not answers, and, in fact, seem to take for granted a 'yes' answer to the question that they would pose if taken literally. An interrogative can be ambiguous as to whether it is intended as a question or a command, for example:

Will you stop making that noise? (21)

The hearer will decide what the speaker's intention is on the basis of the context and extralinguistic factors. Once again, the word 'please' as in

Will you please stop making that noise? (22)

forces us to interpret the sentence as a command.

It might appear that the interrogative (21) can become a command through much the same mechanism as

I want a screw-driver. (9)

becomes a command. That is, through a procedure which is activated by the acceptance of the sentence in its literal meaning. However, there is an objection to this view. The question

Are you going to stop making that noise? (23)

means the same as (21) in its sense as a question, but we are much less likely to regard (23) as a command than we are (21). In fact, in

Are you going to stop making that noise, please? (24)

the 'please' seems to be urging a reply to the question, not making it into a command. A similar contrast can be seen between 'Can you ...?' and 'Are you able to ...?' interrogatives; the former, but not the latter, can function as commands.

What seems to be happening is that the constructions 'can you' and 'will you' are specially treated during translation, and give rise to the alternative translations automatically. The appropriate analysis can then be selected with reference to the context. In these cases, the word 'please' renders it very probable that a command was intended.

Other interrogatives, on the other hand, only give rise to translations as

questions, and the word 'please' usually just encourages the hearer to answer the question instead of refusing to do so. The word 'please' does, of course, have other special uses, for example in

Please may I stay up late tonight, Mum? (25)

but we shall not discuss them further here.

COMMANDS

Any of the effects which we have discussed above can be produced by a command. The reason for this is essentially that we have verbs to name the operations involved, and these verbs can be used in the imperative. Thus to get the effect of a statement, we can say 'Note that . . .', 'Please believe that . . .', 'Be informed that . . .', or (archaically) 'Know ye that . . .'. Questions can be paraphrased 'Tell me whether . . .', 'Say why . . .', 'Tell me what . . .', etc., according to type.

Commands can also have external effects that statements and questions cannot have directly:

Follow that cab! (26)

Wash your ears! (27)

There are also some internal effects that are the special province of commands. One is the setting up of 'antecedent theorems' with external effects. That is, the immediate effect of the command is internal, but there will be an external effect later on if the right conditions are satisfied.

In case of fire, break this glass! (28)

Then there are the internal effects associated with those curious commands, beloved of mathematics authors, of the form:

Consider the function whose value is 1 at all rational numbers and 0 elsewhere. (29)

This shares with

Imagine a striped elephant! (30)

the property of being rather difficult to disobey, if it is understood. The work has all been done at the time of translation by the programs represented by the noun phrases; the command to 'consider' or 'imagine' what has already been brought to one's attention is not really needed. The effect of (29) in drawing the hearer's attention to the function in question is also achieved by

the function whose value is 1 at all rational numbers and 0 elsewhere (31)

The verb 'consider' appears to be a 'no operation' verb, required only to complete the sentence.

There is also a further variety of mental acts – recalling, forgetting, contemplating, mulling over – that seem to involve changes of attention, and which are called upon by commands rather than statements. It is not clear to us what the computational analogues of these mental acts should be.

BREAD-CRUMBS

We wish now to consider how the sort of translation that we have proposed might be carried out, and, in particular, the role of 'parsing' in this process.

There is a well-known method of recursively defining an interpretation for a formalised language, due originally to Tarski (see Tarski 1931), which has since become the standard method for creating a formal semantics. This method assigns an interpretation to a string, viewed as a member of a specific syntactic category. That is, one has defining clauses of the form: 'If the string is a conjunction, A&B, then its interpretation is . . .'. Most formal languages to which the method is applied satisfy a 'unique readability condition'. A given string can belong to at most one syntactic category, and can be formed by applying that category's formation rules to shorter strings in exactly one way; that is, the string formed by conjoining two strings A and B cannot also be formed by conjoining two other strings C and D.

English, of course, does not satisfy the unique readability condition. Many sentences are syntactically ambiguous. It therefore seems reasonable that, in trying to apply this method to English, one should treat not English itself, but *parsings* of English sentences. This is indeed the approach adopted by Keenan (1972) and Montague (1970b) (though not Montague 1970a). Those who have attempted to take syntax seriously in constructing model users of English, for example Winograd (1971) and Woods (1968, 1970), have also, presumably from similar motives, built systems which first parse and then translate the parsed utterance into their forms of internal representation.

This is not quite a fair account, because some translations may be produced, and even run, to help make decisions about alternative parsings. Winograd, in particular, has demonstrated the practical importance of running the translations of parts of sentences as soon as possible when ambiguities have to be resolved. However, in those systems there are two distinct stages, and the translator uses the output of the parser.

The purpose of a model may be to express an idea, rather than to simulate a real process as exactly as possible; in the two pieces of work just mentioned, no particular claim is made for the psychological reality of these separate processes. In Winograd's case, an important part of his achievement is his explanation of the connections between his syntactic and semantic concepts. Merging them computationally might have complicated the exposition.

We should like to propose what seems to us a psychologically more plausible sort of model, in which obtaining the right computational functions plays the central role, and the syntactic structure of an utterance is revealed by the way in which the translation process deals with it. We have recently come to realize that the DEACON system (Thompson 1966) related its syntax and semantics in essentially this spirit, although the syntax and the semantics were both rather different from what we have in mind.

To help clarify our position, we wish to draw an analogy between understanding an utterance, and finding a destination. We have in mind a situation in which you do not actually know the way, but you have a number of heuristics to guide you. Suppose that one of the heuristics involves leaving a trail of bread-crumbs. This might be of use to show when you have gone in a circle, or if you decide to backtrack. The trail of bread-crumbs is to be analogous to the parsing of the utterance. After you have arrived at your destination, the trail does in fact contain all the information needed to get there from where you started. However, that is no reason to go back to where you started and follow it.

Using interpretation to help with parsing is, according to this view, somewhat like exploring the area to help decide in what direction to leave a trail. The real point of the exploration is to decide which way to *go*, and we consult alternative interpretations (by running programs during translation) to decide which interpretation to take, not which parsing.

NUMBER NAMES

As a slightly more substantial illustration of what we are proposing, we have programmed an algorithm for interpreting the English names of natural numbers. We have chosen the following grammar to represent the syntax of number names:

numbername → *unit* | *term* (*term*)* (**and** *unit*)
term → *digitword* **hundred** | *numbername powerword*
unit → *digitword* | *teenword* | *tyword* (*digitword*)
digitword → **one** | **two** | . . . | **nine**
teenword → **ten** | **eleven** | **twelve** | . . . | **nineteen**
tyword → **twenty** | **thirty** | . . . | **ninety**
powerword → **thousand** | **million** | **billion** | **trillion** | . . .

Interpretation proceeds in the obvious way. Words have their usual values. The value of a unit is the value of its single constituent, or, in the case of a *tyword* followed by a *digitword*, the sum of their values. The value of a term is the value of the *powerword* (or 'hundred') times the value of its coefficient; the value of a *numbername* is the sum of the values of its terms and units.

We have in mind here a British dialect in which expressions such as 'three thousand million', and even 'two million million' are allowable. Even so, there are restrictions not expressed by this grammar. We prefer to state these in semantic terms.

(1) In the second expansion of *numbername*, the value of each term's *powerword* must be less than that of the preceding term's *powerword*.

(2) In the second expansion of *term*, if the *numbername* contains a *term*, then the value of the *powerword* must be greater than or equal to the value of the *powerword* of the leftmost *term* in the *numbername*.

The first restriction rules out expressions such as 'one hundred and two

seven thousand' (as opposed to 'seven thousand one hundred and two'); the second rules out 'six million thousand'.

Notice that under this arrangement there are expressions which are syntactically, but not semantically, ambiguous. For instance, 'one million two hundred thousand' might be parsed as either

(one million)((two hundred)thousand) (32)

or as

((one million)(two hundred))thousand (33)

but the second restriction eliminates the latter possibility.

Now, it would be a fairly simple matter to parse number names according to the grammar, and then evaluate the result. And the process, as a whole, could be made more efficient by using evaluation to eliminate ambiguities in the parsing. But it seems more sensible and straight-forward to evaluate as one goes along, storing partial evaluations (that is, numbers) instead of partial trees. Indeed, this seems to be what happens when we interpret number names spoken aloud. (Try saying 'one . . thousand . . nine . . hundred . . and . . seventy . . two', slowly.)

Our algorithm keeps a stack of numbers, the values of terms, together with the powerwords of those terms. When confronted with a new digitword, teenword, or tyword, it puts the value on the stack. When confronted with a powerword, it removes numbers from the stack until it comes to one whose associated powerword has value greater than that of the current powerword. It then adds up all the numbers it has removed, multiplies the sum by the value of this new powerword, and puts the result back on the stack together with the new powerword. At the end, the numbers on the stack are summed.

Here is how the algorithm is applied to 'one million two hundred thousand':

stage of computation	1	2	3	4	5	final
word	one	million	two	hundred	thousand	.
stack	1	1 000 000 'million'	2 1 000 000 'million'	200 'hundred' 1 000 000 'million'	200 000 'thousand' 1 000 000 'million'	1 200 000

The syntactic structure of the number name is revealed, implicitly, by the (semantic) decisions as to which numbers should be added together, and which multiplied. The question of syntactic ambiguity does not arise explicitly. We could, of course, store partial trees along with the numbers, instead of just powerwords, and thus produce a parsing along with our evaluation, but that is not necessary in performing the evaluation.

Another way of viewing the matter is to look upon each syntactic category as a semantic function, namely that which tells us how to interpret the members of that category. In the present example, *numbername*, for example, would represent the function *sum*. Then the expression

numbername(*term*(*unit*(*digitword*('*one*')), '*hundred*'),
'*and*',
unit(*digitword*('*three*'))) (34)

which looks like a parsing of 'one hundred and three', actually evaluates to 103, in the same way that '100+3' does. What our algorithm does is to evaluate the subparts of this expression as soon as they are formed, rather than waiting until the entire expression has been constructed. This technique is, in fact, known to compiler-writers (see Bolliet 1968).

TIC-TAC-TOE

Finally, we should like to touch on a program that we have written which 'understands' a somewhat more robust subset of English than the names of numbers, though we must make it clear that our subset does not approach that of Winograd in size or complexity. This program will be described in detail elsewhere, and we shall give little more here than a sketch of its semantics and some sample output. The reason we mention it at all is that it translates from English to POP-2 in the manner advocated above. English phrases are cashed as POP-2 functions as soon as possible, and some of the functions are run during translation. On the other hand, we have given many of the POP-2 functions English names, so that the translation often bears a striking resemblance to a parsing.

Our main purpose in writing the program was to explore the semantics of English modal verbs (might, can, will, etc.) and hypotheticals (if you had . . .). With this in mind, we thought it best to begin with a universe of discourse in which the notion of *possibility* could be clearly defined. We therefore chose as our setting a game of tic-tac-toe, where one can discuss sensibly what might happen in the future, and also what might have happened in the past, but did not. Our program plays tic-tac-toe against a human opponent, and can answer questions about the course of the game. The board is laid out as a magic square:

2 9 4
7 5 3
6 1 8

each player's objective being to take three numbers which add up to fifteen.

Here is an extract from a game. We begin as the human, playing X, is about to make his second move. The machine's remarks are italicised. Unfortunately, our teletype keyboard lacks a question-mark.

O 9 4
7 **X** 3
6 1 **O**

I take 1.

O 9 4
7 **X** 3
6 **X** **O**

I take 9.
It's not your turn.
You move.

O	**O**	4
7	**X**	3
6	**X**	**O**

If I had taken 4 when I took 1, how could you have won.
By taking 6.
Can I win.
No. If you take 4 *I can take* 6. *If you take* 6 *I can take* 4. *If you take* 3 *I can take* 7. *If you take* 7 *I can take* 3.
What would you do if I took 7.
Take 4.

In essence, the system works by actually constructing past or possible situations and answering questions about what is actually the case with respect to them. It must, of course, be able to distinguish these hypothetical states from the current state of the game. The system has a memory containing the list of moves made, and it uses this to reconstruct past situations. For instance, in the first question above, the 'when'-clause causes the earlier situation to be constructed, and then the 'if'-clause is interpreted within the context of the 'when'-clause. It is crucial here that the translation of the 'when'-clause be run before that of the 'if'-clause. In fact, the 'when' program is run before the translation of the 'if'-clause is complete. This is not of vital importance in the present context, but one can imagine cases where it would be.

Consider:

> If you had held onto your hat when the wind started blowing you wouldn't have lost it. (35)

Here the situation 'when the wind started blowing' defines the referent of 'your hat' in the 'if'-clause. It is your old hat, the one you lost, not the new one you have bought to replace it.

FURTHER GOALS

In this paper we have discussed hearers, rather than speakers, of English. We have not presented a completed system, but rather some outstanding problems in the construction of such a system and an approach towards solving them.

We have argued that a PLANNER-like language provides facilities which are needed to capture the meaning of many English utterances. We also believe that the goal-directed character of PLANNER-like languages will be valuable in modelling speakers, because it will be important to represent the speaker's *reasons* for saying what he says. Bearing these remarks in mind, we are developing a PLANNER-like language, POPLER 1.5, which is adapted to these uses. For example, with respect to the difference between a question

and a command, we have found it necessary to distinguish between a PLANNER *goal* of the form '*goal*: find out whether . . .' and one of the form '*goal*: make it true that . . .'. A description of POPLER 1.5 will be published elsewhere; it is being developed from the earlier system POPLER (Davies 1971).

Acknowledgements

We are greatly indebted to Professor H.C. Longuet-Higgins for his aid and encouragement. If the guise in which his many valuable ideas and suggestions appear here does not do them full justice, the fault is, of course, our own.

We were supported in the course of this work by a grant from the Science Research Council.

REFERENCES

Bolliet, L. (1968) Compiler writing techniques. *Programming Languages* pp. 140 ff. (ed. Genuys, F.). London: Academic Press.

Burstall, R.M., Collins, J.S. & Popplestone, R.J. (1971) *Programming in POP-2.* Edinburgh: Edinburgh University Press.

Davies, D.J.M. (1971) POPLER: A POP-2 Planner. *Memorandum MIP–R–89*, School of Artificial Intelligence, University of Edinburgh.

Hewitt, C. (1971) Description and theoretical analysis (using schemata) of PLANNER. Ph.D. Dissertation, MIT, Cambridge, Mass.

Keenan, E. (1972) Semantically based grammar. *Linguistic Inquiry*, **3**, No. 4.

Longuet-Higgins, H.C. (1970) Computing science as an aid to psycholinguistic studies. *Bull. Institute of Mathematics and its Applications*, **6**, 8–10.

McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P. & Levin, M. I. (1962) *LISP Programmers Manual.* Cambridge, Mass.: MIT Press.

Montague, R. (1970a) English as a formal language. *Linguaggi nella società e nella tecnica*, pp. 189–224 (ed. Visentini, B.) Milan.

Montague, R. (1970b) Universal grammar. *Theoria*, **36**, 373–98.

Scott, D. (1970) Advice on modal logic. *Philosophical Problems in Logic: some recent developments*, pp. 143–73 (ed. Lambert, Karel). Dordrecht, Holland: D. Reidel Publishing Co.

Scott, D. & Strachey, C. (1971) Toward a mathematical semantics for computer languages. *Proc. Symp. on Computers and Automata*, Microwave Research Institute Symposia Series **21.** Polytechnic Institute of Brooklyn.

Tarski, A. (1931) The concept of truth in formalised languages. *Logic, Semantics, Metamathematics.* Oxford: Clarendon Press 1956.

Thompson, F.B. (1966) English for the Computer. *Proc. AFIPS 1966 Fall Joint Comp. Conf.* **29**, 349–56. Washington DC: Spartan Books.

Winograd, T. (1971) Procedures as a representation for data in a computer program for understanding natural language. *M.I.T. Report AI TR–17.*

Woods, W.A. (1968) Procedural semantics for a question-answering machine. *Proc. AFIPS Fall Joint Comp. Conf.* **33**, 457–71. Washington DC: Thompson Book Co.

Woods, W.A. (1970) Transition network grammars for natural language. *Comm. Ass. comput. Mach.* **13**, 591–606.

18

The Syntactic Inference Problem Applied to Biological Systems

G. T. Herman and A. D. Walker
Department of Computer Science
State University of New York at Buffalo

Abstract

The standard scientific procedure of inductive inference has in recent years been vigorously investigated in relation to inferring a grammar of a language from limited information regarding the language. This is referred to as syntactic inference, and has been studied in relation to both natural and artificial languages. An unconnected recent area of investigation is the use of formal language theory in developmental biology. Formal descriptions of the stages of development of an organism are given as strings of symbols. The problem is to devise 'grammatical' rules, which transform the strings of symbols in a way consistent with observations for a particular species. Such rules help us to explain the total development of an organism in terms of local interactions. In this sense, there is an overlap between model building in developmental biology and the syntactic inference problem. This paper describes how various ideas and results connected with the syntactic inference problem carry over to developmental systems, and reports on the so far rather limited amount of original work that has been done in this direction.

1. THE SYNTACTIC INFERENCE PROBLEM

An excellent survey of the syntactic (or grammatical) inference problem has been published by Biermann and Feldman (1971). Although the present paper can be read without prior knowledge of the syntactic inference problem, much of what we shall say here has been influenced by the Biermann and Feldman survey. Let us start with their description of the grammatical inference problem:

> 'A finite set of symbol strings from some language L and possibly a finite set of strings from the complement of L are known, and a grammar for the language is to be discovered.'

As we shall see, for relevance to biological systems, we need to study a

slightly different problem, one that is more general in some ways, but more restricted in other ways.

The syntactic inference problem enters the realm of developmental biology in the following way. A developmental biologist interested in a particular plant is confronted with a large number of experimental results. These vary from purely descriptive reports of the development of that plant in nature, to statistical analysis of the effects of a particular chemical in a series of carefully controlled experiments. His task is to 'explain' on the basis of such experimental results the way in which the particular plant develops. At first it appears that one should try to give such explanations in physico-chemical terms. However, the gap between overall development of a plant and molecular interactions appears to be too wide; satisfactory explanations of developmental phenomena in terms of basic physical and chemical processes have not been forthcoming. In fact, there is a school of thought which doubts that such explanation will ever be forthcoming. For example, Longuet-Higgins (1969) says:

> 'We are beginning to realize that the interest of an organism lies, not in what it is made of, but in how it works.
>
> The most fruitful way of thinking about biological problems is in terms of design, construction and function, which are the concrete problems of the engineer and the abstract logical problems of the automata theorist and computer scientist.
>
> If – as I believe – physics and chemistry are conceptually inadequate as a theoretical framework for biology, it is because they lack the concept of function, and hence of organization . . .. This conceptual deficiency of physics and chemistry is not, however, shared by the engineering sciences; perhaps, therefore, we should give the engineers, and in particular the computer scientists, more of a say in the formulation of Theoretical Biology.'

An alternative approach to direct explanation in physico-chemical terms is to insert in the explanation an intermediate functional step. This is done by introducing some formal framework within which the experimental observations can be described, explained and even predicted. Once such a framework has been accepted, the above described task of the developmental biologist can be broken up into two parts:

(1) Explain the observed experimental phenomena by some model in the framework.

(2) Find the physico-chemical mechanism by which the proposed explanation may be implemented in nature.

Although these tasks are not independent (the testing of a particular physico-chemical explanation will give further experimental results, which will then have to be explained in the formal model), for the purpose of this paper we shall restrict our attention to the first task.

One particular framework has been proposed by Lindenmayer (1968a,b,

1971a,b). In this framework, we assume that an organism at any particular instance of its development can be represented by a finite string of symbols from a finite alphabet. The problem then becomes to find 'grammatical' rules, which will transform the strings in such a way that the sequence of strings produced by repeatedly applying the rules will reflect the biological development.

Although, at first sight, such model appears to be too simple to deal with the complexity of real biological development, it has been demonstrated by Lindenmayer (1968a,b, 1971a,b) and others (Baker and Herman 1972a,b; Herman 1970, 1971a,b, 1972; Raven 1968; Surapipith and Lindenmayer 1969) that, using this model, one can investigate a variety of organisms, as well as a number of significant biological problems. Among the organisms that have been investigated we find red algae, blue-green algae, fungi, mosses, snails and leaves. Among the theoretical problems that have been attacked we find problems associated with the notions of polarity, symmetry, regulation, synchronization, apical and banded patterns, branching patterns, mosaic development, positional information and gradients.

From now on we shall assume that all experimental observations have been translated into finite sequences of strings of symbols, and that what we are required to do is to find the 'grammatical' rules which will 'explain' how such sequences came about. Although this statement must be made more precise before we can hope to achieve anything significant, two points of difference between our problem and the grammatical inference problem as described by Biermann and Feldman (1971) are already clear.

(1) Our input data consist not of symbol strings, but rather sequences of symbol strings. In grammatical terminology, this means that we have some information on how derivations proceed using the grammar.

(2) We can only have positive information available. It does not seem possible to devise an experiment which shows that a particular symbol string cannot be in the 'language' corresponding to a particular species. We could exclude certain strings by definition (for example, 'but that is not what I call an *Anabaena circinalis*'); however, we shall ignore this possibility.

In addition, it is clear that the 'hypothesis space', that is, the set of grammars from which one is to be chosen, must be very different from the type of hypothesis space usually considered in the syntactic inference problem. The grammars must be such that they reflect biological development. This implies that in the derivations of these grammars there must be a high degree of parallelism (development can take place at many, possibly at all, points of an organism at the same time). Also, one must do away with the difference between 'sentence' and 'sentential form' in formal languages; any string that has been used in a proposed derivation must itself be describing an instance in the development of the organism.

2. DEVELOPMENTAL SYSTEMS AND LANGUAGES

Grammars that have been devised to 'explain' biological development are usually described by the generic name 'developmental systems.' The languages generated by them are referred to as 'developmental languages.' There are many different types of developmental systems, some containing quite sophisticated kinds of control mechanisms to reflect the biological situation (see, for example, Rozenberg 1972). For the purpose of demonstrating our ideas on applying the syntactic inference procedures to biological problems, we shall restrict our attention to some very simple types of developmental systems. In these systems, in each step every symbol is replaced simultaneously by a non-empty string of symbols. The string by which a symbol is replaced is determined by the symbol itself and, possibly, by one or both of its neighbors. More precisely, we can define these systems as follows:

For any finite non-empty set G and for any positive integer n, let G^n denote the set of all strings of length n with symbols from G. Let

$$G^+ = \bigcup_{n=1}^{\infty} G^n.$$

For $x \in \{0, 1, 2\}$, we define a DPxL-scheme to be a triple, $\langle G, g, \delta \rangle$, where G is a non-empty finite set, $g \in G$ and δ is a function, $\delta: G^{x+1} \to G^+$.

For any DPxL-scheme $\underline{L} = \langle G, g, \delta \rangle$, and for any element p of G^+, we define $\lambda_{\underline{L}}(p)$ (or $\lambda(p)$ if $\underline{L}$ is understood) as follows. Assume $p = a_1 a_2 \ldots a_k$.

$$\lambda(p) = \begin{cases} \delta(a_1)\delta(a_2)\ldots\delta(a_k), & \text{if } n=0, \\ \delta(ga_1)\delta(a_1a_2)\delta(a_2a_3)\ldots\delta(a_{k-1}a_k), & \text{if } n=1, \\ \delta(ga_1a_2)\delta(a_1a_2a_3)\delta(a_2a_3a_4)\ldots\delta(a_{k-1}a_kg), & \text{if } n=2 \text{ and } k \neq 1. \end{cases}$$

If $n=2$ and $k=1$, $\lambda(p) = \delta(ga_1g)$.

If we consider an element p of G^+ to be a representation of a string of cells, and $\lambda(p)$ to be its immediate successor in the developmental history, we see that the string of cells into which a cell may divide depends in a DP0L-scheme only on the cell itself, in a DP1L-scheme on the cell and its left neighbor, in a DP2L-scheme on the cell and both its neighbors. The abbreviation DPxL-scheme stands for deterministic propagating x-sided Lindenmayer scheme. Such systems have formed the subject matter of some very active research in recent years (van Dalen 1971; Doucet 1971; Feliciangeli and Herman 1972; Herman 1969, 1970, 1971a,b,c; Herman, Lee, van Leeuwen and Rozenberg 1972; Lindemayer, 1968a,b, 1971a,b; Lindenmayer and Rozenberg 1972; Rozenberg and Doucet 1971).

Example. Let $\underline{K} = \langle \{0, 1\}, 1, \delta \rangle$ be a DP2L-scheme defined by

$$\delta(101) = 00,$$
$$\delta(100) = 0,$$
$$\delta(p) = 1, \text{ for all } p \in G^3 - \{101, 100\}.$$

Then

$$\lambda(0) = \delta(101) = 00,$$
$$\lambda(00) = \delta(100)\delta(001) = 01,$$
$$\lambda(01) = \delta(101)\delta(011) = 001,$$
$$\lambda(1) = \delta(111) = 1.$$

For any DPxL-scheme $\underline{L}=\langle G, g, \delta\rangle$, any $p \in G^+$ and any non-negative integer t, we define $\lambda_{\underline{L}}^t(p)$ (or $\lambda^t(p)$, if $\underline{L}$ is understood) by

$$\lambda^0(p)=p$$
$$\lambda^{t+1}(p)=\lambda(\lambda^t(p)).$$

Suppose we have made n observations of the development of a particular organism, and we have translated our observations into strings $p_1, \ldots, p_n$, where the order of the strings indicates the order in which the observations have been taken. If a particular DPxL-scheme $\underline{L}$ is suggested as reflecting the developmental rules for the organism, it must at least have the property that, for $1 \leqslant j < n$,

$$p_{j+1}=\lambda^{t_j}(p_j),$$

where $t_1, \ldots, t_{n-1}$ are positive integers. Note that the DP2L-scheme $\underline{K}$ proposed in the example above will do this for the sequence 0, 00, 01, with $t_1=t_2=1$. In contrast, we have the following result.

There does not exist a DP1L-scheme $\underline{L}$, and positive integers t_1 and t_2 such that

$$00=\lambda_{\underline{L}}^{t_1}(0),$$
$$01=\lambda_{\underline{L}}^{t_2}(00).$$

This result was obtained in the proof of Theorem 5 in Herman (1970).

This very simple example highlights another basic difference in emphasis between the standard syntactic inference problem and the inference of developmental systems from biological data. In the standard grammatical inference problem one assumes that there exists a grammar in the hypothesis space which is consistent with the sample. In developmental biology, one of the very basic questions is: what type of hypothesis space must we assume so that we are assured of finding a developmental system consistent with the observations? In other words, the hypothesis space itself is dependent on the input data. For example, as we have indicated above, for the sequence 0, 00, 01 it would be pointless to search for a DP1L-scheme. When applied to biological development, the syntactic inference problem should usually have two parts:

(1) find an appropriate set of developmental systems;
(2) select a particular system from the set.

To some extent, the same situation exists when the syntactic inference problem is applied to natural (or even artificial) languages. After all, there is great doubt concerning what type of grammar is appropriate for describing natural languages. Yet, in discussing the syntactic inference problem for natural and artificial languages, people have concentrated on the second of the tasks described above. In contrast to this, it is the first task which has been given the greater prominence in biological discussions.

We shall now discuss nine instances of the syntactical inference problem in biological situations. In each of the nine instances we assume that the input $\mathscr{S}$ is a finite set of finite sequences of strings. We shall invariably use the notation m for the number of sequences, n_i for the length of the ith sequence

and $p_{i,j}$ to denote the jth string in the ith sequence of $\mathscr{S}$. Thus,

$$\mathscr{S}=\{\langle p_{1,1}, p_{1,2}, \ldots, p_{1,n_1}\rangle, \langle p_{2,1}, p_{2,2}, \ldots, p_{2,n_2}\rangle, \ldots, \langle p_{m,1}, p_{m,2}, \ldots, p_{m,n_m}\rangle\}.$$

Also, we shall invariably use $l_{i,j}$ to denote the length of $p_{i,j}$ and $a_{i,j,k}$ to denote the kth symbol in the string $p_{i,j}=a_{i,j,1}a_{i,j,2}\ldots a_{i,j,l_{i,j}}$.

Let $\underline{L}=\langle G, g, \delta\rangle$ be a DPxL-scheme, and $\mathscr{S}$ be a finite set of finite sequences of strings.

We say that $\mathscr{S}$ has *property* A with respect to $\underline{L}$ if, and only if, for $1\leqslant i\leqslant m$, $1\leqslant j<n_i$,

$$p_{i,j+1}=\lambda(p_{i,j}).$$

We say that $\mathscr{S}$ has *property* B with respect to $\underline{L}$ if, and only if, there exists a positive integer k such that, for $1\leqslant i\leqslant m$, $1\leqslant j<n_i$,

$$p_{i,j+1}=\lambda^k(p_{i,j}).$$

We say that $\mathscr{S}$ has *property* C with respect to $\underline{L}$ if, and only if, for $1\leqslant i\leqslant m$ and $1\leqslant j<n_i$, there exists a positive integer, $k_{i,j}$, such that

$$p_{i,j+1}=\lambda^{k_{i,j}}(p_{i,j}).$$

It is implicitly assumed in all these definitions that, for $1\leqslant i\leqslant m$, $1\leqslant j\leqslant n_i$, $p_{i,j}\in G^+$.

Note that property A implies property B, which in turn implies property C.

Let $X\in\{A, B, C\}$ and $x\in\{0, 1, 2\}$.

Problem Xx. Give a procedure which for any finite set $\mathscr{S}$ of sequences of strings

(1) decides whether or not there exists a DPxL-scheme $\underline{L}$ such that $\mathscr{S}$ has property X with respect to $\underline{L}$,

(2) produces such an $\underline{L}$, if there is one.

Feliciangeli and Herman (1972) provide answers to A0, A1, A2, B0, B2 and C2, while B1, C0 and C1 are still open. The same paper also discusses a number of similar problems.

3. A SAMPLE SOLUTION

To indicate the type of methods used in the solution of such problems, we shall give a solution of B2.

It is easy to see that for any $\mathscr{S}$ and any DP2L-scheme $\underline{L}$, if $\mathscr{S}$ has property B with respect to L, then, for $1\leqslant i\leqslant m$, $1\leqslant j<n_i$, $1\leqslant i'\leqslant m$, $1\leqslant j'<n_{i'}$, we have that

(1) $l_{i,j}\leqslant l_{i,j+1}$,

(2) if $p_{i,j}=p_{i',j'}$, then $p_{i,j+1}=p_{i',j'+1}$.

Given any $\mathscr{S}$, our procedure first checks whether or not (1) and (2) are satisfied. (This is easy.) If they are not, there is no DP2L-scheme of the required kind. If they are satisfied then the following procedure constructs a suitable DP2L-scheme $\underline{L}=\langle G, g, \delta\rangle$.

Let l be the length of the longest string in $\mathscr{S}$, that is, the maximum $l_{i,j}$. Let G_0 consist of all the $a_{i,j,k}$ in $\mathscr{S}$ and an additional symbol g. The alphabet G of $\underline{L}$ consists of all symbols in G_0, and symbols representing strings of

symbols from G_0 of odd length between 3 and $2l-1$. The symbol which represents the string p will be denoted by $[p]$.

δ is defined in such a way that, starting from any $p_{i,j}$, after $l-1$ steps each cell will be in a state which indicates to that cell what the whole of $p_{i,j}$ is, and how many cells from the end that particular cell is. Then each cell can change into the state of the cell in the corresponding position in $p_{i,j+1}$, with the last cell dividing into many cells if $l_{i,j+1} > l_{i,j}$. This is done in the following way.

For all symbols a, b, c and d in G_0, for all strings p of symbols from G_0 of odd length between 1 and $2l-5$, for $1 \leqslant i \leqslant m$, $1 \leqslant j < n_i$, $1 \leqslant k \leqslant l_{i,j}$, and for all $x, z \in G$,

$$\delta(a\ \ b\ \ c) = [abc],$$
$$\delta([abp]\ \ [bpc]\ \ [pcd]) = [abpcd],$$
$$\delta(g\ \ [bpc]\ \ [pcd]) = [gbpcd],$$
$$\delta([abp]\ \ [bpc]\ \ g) = [abpcg],$$
$$\delta(g\ \ [bpc]\ \ g) = [gbpcg],$$
$$\delta(x\ \ [g^{l-k} p_{i,j}\, g^{l+k-1-l_{i,}}]\ \ z) = \begin{cases} a_{i,j+1,k}, & \text{if } k < l_{i,j}, \\ a_{i,j+1,l_{i,j}}, a_{i,j+1,l_{i,j}+1}, \ldots, a_{i,j+1,l_{i,j+1}}, & \text{if } k = l_{i,j}. \end{cases}$$

For all other symbols x, y and z in G,

$$\delta(xyz) = y.$$

Since $\mathscr{S}$ satisfies condition (2), there is no conflict in the definition of δ and so $\underline{L}$ is a well-defined DP2L-scheme. It is not too difficult to prove in detail that, for $1 \leqslant i \leqslant m$, $1 \leqslant j < n_i$, $p_{i,j+1} = \lambda^l(p_{i,j})$, but we shall just demonstrate this by an example.

Let $\mathscr{S} = \{\langle 0, 00, 01, 001\rangle, \langle 1, 1, 1\rangle\}$.

In this case $G_0 = \{0, 1, g\}$, $l = 3$ and

$$G = \{0, 1, g, [000], [001], \ldots, [ggg], [00000], [00001], \ldots, [ggggg]\}.$$

Using the δ rules as defined above, we get the following processes produced by $\underline{L}$ starting with 0 and 1 respectively.

0
[g0g]
[gg0gg]
0 0
[g00] [00g]
[gg00g] [g00gg]
0 1
[g01] [01g]
[gg01g] [g01gg]
0 0 1

1
[g1g]
[gg1gg]
1
[g1g]
[gg1gg]
1

Note, however, that we have already seen a much simpler looking DP2L-scheme such that

$$\lambda^3(0) = \lambda^2(00) = \lambda(01) = 001,$$
$$\lambda^2(1) = \lambda(1) = 1.$$

4. THE RELEVANCE OF GRAMMATICAL INFERENCE NOTIONS TO BIOLOGICAL PROBLEMS

The very fact that our procedure in the last section produced a DP2L-scheme which does the job, but which is both more complicated and uses a larger k, than another DP2L-scheme which is also appropriate, indicates that our problem statements leave something to be desired. We have made no request in our formation of problem B2 that the DP2L-scheme produced should in any sense be the best. Such and similar considerations are often discussed regarding grammatical inference. We now survey various notions from grammatical inference and discuss their relevance in a biological context.

I. Identification in the limit

This idea originates from Gold (1967). For each of our nine problems in the last section there exists an identification in the limit problem. We shall demonstrate this with an example which corresponds to A1.

Problem IA1. Give a procedure which will do the following:

(i) With any finite set $\mathscr{S}$ of finite sequences of strings, which has property A with respect to some DP1L-scheme, the procedure associates a DP1L-scheme $\underline{L}(\mathscr{S})$.

(ii) Let $\underline{L}$ be any DP1L-scheme, and let $S_1, S_2, \ldots$ be any infinite sequence of sequences of strings with the following properties. Let $\mathscr{S}_t = \{S_1, S_2, \ldots, S_t\}$.

(a) For all t, $\mathscr{S}_t$ has the property A with respect to $\underline{L}$.

(b) If any finite set $\mathscr{S}$ of finite sequences of strings have the property A with respect to $\underline{L}$, then $\mathscr{S} \subseteq \mathscr{S}_t$ for some t.

For all such $\underline{L}$ and $S_1, S_2, \ldots$, our procedure must have the property that there exists a positive integer τ such that

(c) for any finite set $\mathscr{S}$ of finite sequences of strings, $\mathscr{S}$ has the property A with respect to $\underline{L}$ if, and only if, $\mathscr{S}$ has the property A with respect to $\underline{L}(\mathscr{S}_\tau)$,

(d) for all $t \geqslant \tau$, $\underline{L}(\mathscr{S}_t) = \underline{L}(\mathscr{S}_\tau)$.

The underlying biological situation corresponding to problem IA1 can be explained as follows: Suppose that we know that all our observations $\{S_t\}$ for a particular species have the property A with respect to a fixed DP1L-scheme $\underline{L}$, and that every $\{S_t\}$ which has the property A with respect to $\underline{L}$ will eventually be observed. The procedure required in problem IA1 is such that it is guaranteed that, provided we keep on carrying out observations long enough, the procedure will provide us with a DP1L-scheme, which is indistinguishable from $\underline{L}$ using the type of information we have available from our observations and which will remain indistinguishable from $\underline{L}$ forever, irrespective of any additional observations we may make.

We do not know at which point we found a correct DP1L-scheme. At each time t we make a guess $\underline{L}(\mathscr{S}_t)$, but we may change this guess if it contradicts observations obtained later on. In this sense, identification in the limit captures the essence of a scientific procedure: make a hypothesis consistent with the facts as you know them, but do not hesitate to change it if new

facts arise which contradict it. There is a basic belief in most scientists that this procedure eventually converges to something like the 'truth', although we can never be certain that we are already there. However, in the case of IA1, we can give a procedure which can be proved to do the job required.

Solution to IA1.

Let $\mathcal{S}$ be a finite set of finite sequences of strings which has property A with respect to some DPIL-scheme. We use the standard notation for $\mathcal{S}$ as explained previously.

Let $H=\{a_{i,j,k} \mid 1\leqslant i\leqslant m,\ 1\leqslant j\leqslant n_i,\ 1\leqslant k\leqslant l_{i,j}\}$.

We define $\underline{L}(\mathcal{S})=\langle H, h, \phi\rangle$ as follows:

For all $a, b\in H$,

$$\phi(ab)=\begin{cases} p, \text{ if } \langle a, q\rangle\in\mathcal{S} \text{ and } \langle ab, qp\rangle\in\mathcal{S}, \\ \quad \text{for some } p \text{ and } q \text{ in } H^+, \\ b, \text{ otherwise.}\end{cases}$$

Let us order elements of H by ordering all the $a_{i,j,k}$ according to the size of i, j and k (for example, $a_{2,3,5}$ precedes $a_{2,4,1}$) and then eliminating duplications. We define h to be the first element of H such that, for all a in H, $\langle a, \phi(ha)\rangle\in\mathcal{S}$, if there is such an h, and $h=a_{1,1,1}$, otherwise.

Since $\mathcal{S}$ has the property A with respect to some DPIL–scheme, it follows that, for all $p, q, r\in H^+$, $\langle p, q\rangle\in\mathcal{S}$ and $\langle p, r\rangle\in\mathcal{S}$ implies that $q=r$. Hence, $\underline{L}(\mathcal{S})$ is a well defined DPIL-scheme, and our procedure satisfies (i) in the statement of IA1.

Now suppose that $\underline{L}=\langle G, g, \delta\rangle$ is a DPIL-scheme and $S_1, S_2, \ldots$ is an infinite sequence of sequences of strings satisfying (a) and (b) in the statement of IA1. Because (a) is satisfied, the procedure described above will provide us with a DPIL-scheme $\underline{L}(\mathcal{S}_t)$, for all t. Because (b) is satisfied, there exists a τ such that for all $a, b\in G$, $\langle a, \delta(ga)\rangle\in\mathcal{S}_\tau$ and $\langle ab, \delta(ga)\delta(ab)\rangle\in\mathcal{S}_\tau$. It is clear that for all $t\geqslant\tau$, $\underline{L}(\mathcal{S}_t)=\langle G, g', \delta\rangle$, where g' has the property that for all $a\in G$, $\delta(ga)=\delta(g'a)$. Hence, our procedure satisfies (ii) in the statement of IA1.

One can easily state identification in the limit problems IXx for $X\in\{A, B, C\}$ and $x\in\{0, 1, 2\}$. It should also be clear that a solution similar to that of IA1 exists for IA0. The solution of IA2 is only slightly more complicated. There are also a number of biologically reasonable variants of the problem. For example, we assumed in IA1 that, if $L=\langle G, g, \delta\rangle$, g must be manifested in the strings that we observed. Since the interpretation of g is that it is an environmental state, a reasonable alternative to IA1 is the problem in which only sequences without g in them are observed.

We found a solution to IA1. Yet, identification in the limit problems based on only positive information have been found by Gold (1967) to be unsolvable for all but the simplest cases. One difference, of course, lies in the availability of structural information about how the strings have been derived. There are other instances in the literature, where structural information makes grammatical inference in the limit possible from purely positive information

(for example, Crespi-Reghizzi 1971). Another important difference is that we have negative information in an indirect way. For example, we know that if $q \neq p_{1,2}$, then $\{\langle p_{1,1}, q\rangle\}$ does not have the property A with respect to any $\underline{L}$, for which $\mathscr{S}$ has the property A.

There are a number of other approaches for grammatical inference in the limit (see, for example, Biermann and Feldman 1971), and many of these have interesting equivalents for developmental systems. However, so far there has been no substantial work done in this direction.

II. Inference from finite presentation: probability and complexity measures

Problems Xx were stated in such a way that any grammar consistent with the finite observations was considered acceptable. As we have seen, this can lead to algorithms which produce grammars which in some intuitive sense are not the right choice. An identification in the limit procedure can sometimes help by producing a stable grammar which is good for an infinite sample. Even there, at any particular stage, we only have a finite set of observations, and the question arises, which is the 'best' grammar amongst those consistent with the observations.

Biermann and Feldman (1971) suggest that there are three possible ways to proceed to answer this question:

'(1) set up a probabilistic model which provides a technique for computing the most probable grammar,

(2) design a complexity measure for the grammars and their derivations and find the least complex explanation for the data, or

(3) develop a constructive algorithm which converges on a correct answer at the fastest practicable rate and let its intermediate behavior be what it will be.'

From the biological point of view, the first of these approaches appears to be theoretically acceptable, but there is no indication whatsoever that an acceptable probabilistic model is likely to be produced in the forseeable future. Our knowledge of biological processes is too limited to allow us the setting up of a probabilistic model of this type with any confidence. (For that matter, the same criticism seems to be valid when grammatical inference is used in relation to natural languages.)

The second approach appears to be much more promising. Although there can be many reasonable complexity measures defined for each of the nine problems Xx, we shall give one example, which is a version of problem C0 with a rather straightforward complexity measure.

For any DP0L-scheme $\underline{L}=\langle G, g, \delta\rangle$, let the *intrinsic complexity* of $\underline{L}$ be defined as

$$c(\underline{L})=\sum_{a\in G} lg(\delta(a)),$$

where $lg(p)$ denotes the length of the string p.

For any DP0L-scheme $\underline{L}=\langle G, g, \delta\rangle$, and any finite set $\mathscr{S}$ of finite sequences

of strings which have property C with respect to $\underline{L}$, let the *derivational complexity* of $\mathscr{S}$ with respect to $\underline{L}$ be defined by

$$d(\mathscr{S}, \underline{L}) = \sum_{i=1}^{m} \sum_{j=1}^{n_i - 1} k_{i,j},$$

where $k_{i,j}$ is the smallest positive integer such that

$$p_{i,j+1} = \lambda^{k_{i,j}}(p_{i,j}).$$

Let the *complexity* of $\mathscr{S}$ with respect to $\underline{L}$ be defined by

$$\gamma(\mathscr{S}, \underline{L}) = c(\underline{L}) + d(\mathscr{S}, \underline{L}).$$

Problem IIC0. For any finite set $\mathscr{S}$ of finite sequences of strings which has property C with respect to some DP0L-scheme, find a DP0L-scheme $\underline{L}$ such that

(i) $\mathscr{S}$ has property C with respect to $\underline{L}$,

(ii) for any $\underline{L}'$, such that $\mathscr{S}$ has property C with respect to $\underline{L}'$,

$$\gamma(\mathscr{S}, \underline{L}) \leqslant \gamma(\mathscr{S}, \underline{L}').$$

Solution to Problem IIC0. (This solution is based on a method of Feldman, as reported on by Biermann and Feldman 1971.)

Let H be the set of all symbols which occur in $\mathscr{S}$. Let Σ be an effectively enumerable infinite set of symbols, such that H is an initial segment of Σ. Clearly, there exists an effective enumeration of the set $\mathscr{L}$ of all DP0L-schemes $\underline{L} = \langle G, g, \delta \rangle$ such that G is an initial segment of Σ containing H. Furthermore, any DP0L-scheme $\underline{L}$ which has property C with respect to $\mathscr{S}$ will have an isomorphic image in $\mathscr{L}$. We can, therefore, restrict our attention to $\mathscr{L}$ in our search for an optimal DP0L-scheme for $\mathscr{S}$.

Let $\underline{L}_1, \underline{L}_2, \ldots$ be the effective enumeration of $\mathscr{L}$. For each $t \geqslant 1$, we can test in a finite number of steps whether or not $\mathscr{S}$ has property C with respect to $\underline{L}_t$. This is because, for $1 \leqslant i \leqslant m$, $1 \leqslant j < n_i$, we can find out in a finite number of steps whether or not there exists a $k > 0$, such that $\lambda^k_{\underline{L}_t}(p_{i,j}) = p_{i,j+1}$. Simply work out $\lambda^k_{\underline{L}_t}(p_{i,j})$, for $k = 1, 2, 3, \ldots$, until either $p_{i,j+1}$ is found, or $\lambda^k_{\underline{L}_t}(p_{i,j})$ repeats a previous value, or $\lambda^k_{\underline{L}_t}(p_{i,j})$ is longer than $p_{i,j+1}$. One of these three cases must arise, giving us the required decision.

Since $\mathscr{S}$ has property C with respect to some DP0L-scheme, we shall by this method find such a DP0L-scheme $\underline{L}'$. Any candidate $\underline{L}$ for the optimal DP0L-scheme must have the property that

$$c(\underline{L}) \leqslant \gamma(\mathscr{S}, \underline{L}').$$

There are only finitely many $\underline{L}$ in $\mathscr{L}$ with this property, and we can effectively list every one of them. (This is because in order for $c(\underline{L})$ to be bounded by $\gamma(\mathscr{S}, \underline{L}')$, the number of symbols in $\underline{L}$ must be no greater than $\gamma(\mathscr{S}, \underline{L}')$, and the length of any $\delta(a)$ must also be bounded by $\gamma(\mathscr{S}, \underline{L}')$.) For every element $\underline{L}$ of this finite set we can test effectively whether or not $\mathscr{S}$ has property C with respect to $\underline{L}$, and for those with respect to which $\mathscr{S}$ does have property C, we can work out effectively the value of $d(\mathscr{S}, \underline{L})$. Thus, we can find a DP0L-scheme of the type required in problem IIC0.

The method described above is quite general. In particular, the statement and solutions of the analogous Problems IIXx ($X \in \{A, B, C\}$, $x \in \{0, 1, 2\}$) should be clear to the reader.

It appears that there are natural complexity measures for developmental systems, and that optimal syntactic inference using these measures is feasible. In spite of this, practically no work yet has been done in this direction.

The third approach mentioned by Biermann and Feldman (1971) appears to be unreasonable from the point of view of biological model building. Since the collection of a large representative and accurate data set is a slow experimental process, the rate of convergence of an algorithm to a correct answer is not of great importance. It is much more important to have good estimates at the intermediate stages.

III. Methods for inferring developmental systems

None of the algorithms associated with syntactic inference for biological systems have so far been implemented on a computer. What has been done is that individual developmental systems have been produced (and implemented) in answer to some particular situation.

This has usually been done in two stages.

(1) The person intending to build the developmental model looked at the finite set of observations available and generalized it to an infinite set, of which he considered the finite set a typical representative sample.

(2) He built a developmental model consistent with his infinite set.

This heuristic process is an alternative to either identification in the limit or to complexity measures to select the 'best' system consistent with the finite data. However, its success depends very much on the skill and insight of the person doing the generalization.

There has been some work done in the direction of formalizing parts of this heuristic process. For example, Lindenmayer and Rozenberg (1972) discuss the possibility of defining developmental languages in terms of 'locally catenative formulas', which show how in the developmental process a string may be described as a concatenation of previous strings. They state that every such locally catenative formula can be realized by a DP0L-scheme that is, one can produce a DP0L-scheme which generates sequences which satisfy the given locally catenative formula. These notions can be generalized. Algorithms which produce developmental systems for processes described by rather general recurrence schemes are at present under active investigation.

5. GROWTH FUNCTIONS

In the present section we consider a problem which is peculiar to developmental systems. The basic ideas, though not the results, of the present section, are from the yet unpublished works of A. Paz, A. Salomaa and A. Szilard.

One of the easiest things to observe about a filamentous organism is the number of cells it has. Suppose, having observed the development of a particular organism, we generalize our observation by giving a computable function f, such that $f(t)$ is the length of the organism after t steps. The problem then arises to produce a developmental system whose growth function is f.

We shall demonstrate this topic with two examples, but first we give some precise definitions.

Let f be a function mapping the non-negative integers into positive integers. For $x \in \{0, 1, 2\}$, f is said to be a *growth-function* of the DPxL-scheme $\underline{L}=\langle G, g, \delta \rangle$ if, and only if, there exists a $p \in G^+$, such that, for all t,

$$f(t)=lg(\lambda^t(p)).$$

A DP2L-scheme $\underline{L}=\langle G, g, \delta \rangle$ is said to be *symmetric* if, and only if, for all $a, b, c \in G$,

$$\delta(abc)=\delta(cba)=[\delta(abc)]^R,$$

where p^R is the string p written in reversed order. The biological significance of symmetric DP2L-schemes has been discussed by Herman (1971a,b; 1972).

Example

Let f be defined by

$$f(0)=1,$$
$$f(1)=2$$
$$f(n+2)=f(n+1)+f(n), \quad \text{for } n \geqslant 0.$$

Then, there does not exist a symmetric DP2L-scheme $\underline{L}$ such that f is a growth function of $\underline{L}$.

To show this, suppose there exists a symmetric DP2L-scheme $\underline{L}=\langle G, g, \delta \rangle$ such that f is a growth function of $\underline{L}$. Then there must exist an element $a \in G$, such that $lg(\lambda^t(a))=f(t)$, for all t. In particular $lg(\lambda(a))=2$. Since $\underline{L}$ is symmetric, $\lambda(a)=\delta(gag)=[\delta(gag)]^R$. So there must be a symbol b such that $\lambda(a)=bb$. But then

$$\begin{aligned}\lambda^2(a)&=\lambda(bb)\\&=\delta(gbb)\delta(bbg)\\&=\delta(gbb)\delta(gbb)\end{aligned}$$

is of even length, while $f(2)=3$.

The number $f(n)$ is the nth Fibonacci number. It is interesting to note that by starting one step later, we can have the Fibonacci numbers as a growth function.

Let

$$f(0)=2,$$
$$f(1)=3,$$
$$f(n+2)=f(n+1)+f(n), \quad \text{for } n \geqslant 0.$$

Then f is the growth function of the symmetric DP2L-scheme $\underline{L}=\langle \{g, a, b, d, i, l, r, s\}, g, \delta \rangle$, where

$$\delta(aai)=\delta(iaa)=\delta(dss)=\delta(ssd)=\delta(grl)=\delta(lrg)=i,$$
$$\delta(bbd)=\delta(dbb)=\delta(iss)=\delta(ssi)=d,$$
$$\delta(aad)=\delta(daa)=\delta(aag)=\delta(gaa)=\delta(aas)=\delta(saa)=\delta(bds)=\delta(sdb)=\delta(glr)=\delta(rlg)=aa,$$
$$\delta(ais)=\delta(sia)=\delta(bbi)=\delta(ibb)=\delta(bbs)=\delta(sbb)=bb,$$
$$\delta(adb)=\delta(bda)=\delta(ass)=\delta(ssa)=\delta(bss)=\delta(ssb)=\delta(gss)=\delta(ssg)=\delta(aib)=\delta(bia)=\delta(aig)=\delta(gia)=ss,$$
$$\delta(xyz)=v, \quad \text{for all other } xyz \in G^3.$$

Let $p=lr$, then,

$$lg(p)=lg(lr)=2,$$
$$lg(\lambda(p))=lg(aai)=3,$$
$$lg(\lambda^2(p))=lg(aaiss)=5,$$
$$lg(\lambda^3(p))=lg(aaibbdss)=8,$$
$$lg(\lambda^4(p))=lg(aaissbbdaaiss)=13,$$
$$lg(\lambda^5(p))=lg(aaibbdssbbdssaaibbdss)=21,$$
$$lg(\lambda^6(p))=lg(aaissbbdaaissbbdaaissaaissbbdaaiss)=34, \text{ etc.}$$

(This DP2L-scheme is due to the authors and S. Rowland.)

Our second example is concerned with the difference between the growth functions of DPxL-schemes as x varies.

Consider the following result, which was first pointed out to the authors by A. Paz.

If f has the properties that for every m there exists an n such that

$$f(n)=f(n+1)=\ldots=f(n+m),$$

and $\lim_{n\to\infty} f(n)=\infty$, then f is not the growth function of a DP0L-scheme.

The proof of this fact is easy, once we see that $\lim_{n\to\infty} f(n)=\infty$ implies that for some t, $\lambda^t(p)$ contains a symbol a and a positive integer s such that $\lambda^s(a)=qap$, where not both q and p are empty.

From this we show that there exist functions f which are growth functions of DP1L-schemes, but not of DP0L-schemes. For example, let $L=\langle\{g, a, b, c, r\}, g, \delta\rangle$, where

$$\delta(xc)=a, \quad \text{for } x\in\{g, a, b, c\},$$
$$\delta(ga)=c$$
$$\delta(ca)=b$$
$$\delta(cb)=c$$
$$\delta(cr)=ar$$
$$\delta(xy)=y, \quad \text{otherwise.}$$

Letting $p=ar$

$$\lambda(p)=cr,$$
$$\lambda^2(p)=aar,$$
$$\lambda^3(p)=car,$$
$$\lambda^4(p)=abr,$$
$$\lambda^5(p)=cbr,$$
$$\lambda^6(p)=acr,$$
$$\lambda^7(p)=caar.$$

It can be shown, that for all k, there exists an n, such that

$$f(n)=f(n+1)=\ldots=f(n+2^k),$$

and yet $\lim_{n\to\infty} f(n)=\infty$.

Hence, it follows that this f is not the growth function of any DP0L-scheme. The essence of the proof is that unbounded growth functions of DP0L-schemes must grow with at least a certain speed, and we have found an

unbounded growth function of a DP1L-scheme which grows slower than this. It is interesting to note that the same type of argument cannot be repeated to show the existence of a growth function of a DP2L-scheme which is not also a growth function of a DP1L-scheme. This is because the longest period for which an unbounded growth function of a DP2L-scheme $\underline{L}=\langle G, g, \delta \rangle$ can retain the value k is clearly m^k, where m is the number of symbols in G. Along the lines indicated above, it is not too difficult to produce a DP1L-scheme such that it has an unbounded growth function which grows slower than this. The example given above works for the case $m=2$, but other examples can be given for larger m. It is in fact an open problem whether or not there exists a DP2L-scheme which has a growth function, which is not also the growth function of a DP1L-scheme.

6. CONCLUSIONS

The syntactic inference problem has meaningful analogues for developmental languages and systems. The solution of such problems is of great interest from the point of view of biological modelling. In spite of this, very little work has been done in the area, and most of the interesting problems are still open.

Acknowledgement

The research for this paper has been supported by NSF grant GJ–998.

REFERENCES

Baker, R. & Herman, G.T. (1972a) Simulation of organisms using a developmental model, part I, basic description. *Int. J. Bio-Medical Computing*, **3**, 201.

Baker, R. & Herman, G.T. (1972b) Simulation of organisms using a developmental model, part II, the heterocyst formation problem in blue-green algae, *Int. J. Bio-Medical Computing*, to appear.

Biermann, A. & Feldman, J. (1971) A survey of grammatical inference. *Frontiers of Pattern Recognition* (ed. Watanabe, M.S.). New York: Academic Press.

Crespi-Reghizzi, S. (1971) An effective model for grammar inference. *Proc. IFIP Congress 1971*, TA–3, 193–7.

van Dalen, D. (1971) A note on some systems of Lindenmayer. *Math. Syst. Theory*, **5**, 128–40.

Doucet, P. (1971) On the membership question in some Lindenmayer systems. *Indagationes Mathematicae*, **34**, 45–52.

Feliciangeli, H. & Herman, G.T. (1972) Algorithms for producing grammars from sample derivations: a common problem of formal language theory and developmental biology. *J. Comp. Syst. Sciences*, to appear.

Gold, E.M. (1967) Language identification in the limit. *Inf. Control*, **10**, 447–74.

Herman, G.T. (1969) The computing ability of a developmental model for filamentous organisms. *J. Theoret. Biol.*, **25**, 421–35.

Herman, G.T. (1970) The role of environment in developmental models. *J. Theoret. Biol.*, **29**, 329–41.

Herman, G.T. (1971a) Models for cellular interactions in development without polarity of individual cells, part I, general description and the problem of universal computing ability. *Int. J. Systems Sciences*, **2**, 271–89.

Herman, G.T. (1971b) Polar organisms with apolar individual cells. *Proc. IV Int. Cong. Logic Methodology & Philosophy of Science.*

Herman, G.T. (1971c) Closure properties of some families of languages associated with biological systems. *Proc. 5 Ann. Princeton Conf. Inf. Sciences Syst.*, 465.

Herman, G.T. (1972) Models for cellular interactions in development without polarity of individual cells, part II, problems of synchronization and regulation. *Int. J. Systems Sciences*, **3**, 149–75.

Herman, G.T., Lee, K.P., van Leeuwen, J. & Rozenberg, G. (1972) Unary developmental systems and languages. *Proc. 6 Ann. Princeton Conf. Inf. Sciences Syst.*, 114.

Lindenmayer, A. (1968a) Mathematical models for cellular interactions in development, part I, filaments with one sided inputs. *J. Theoret. Biol.*, **18**, 280–99.

Lindenmayer, A. (1968b) Mathematical models for cellular interactions in development, part II, simple and branching filaments with two sided inputs. *J. Theoret. Biol.*, **18**, 300–15.

Lindenmayer, A. (1971a) Developmental systems without cellular interactions, their languages and grammars. *J. Theoret. Biol.*, **30**, 455–84.

Lindenmayer, A. (1971b) Cellular automata, formal languages and developmental systems. *Proc. IV. Int. Cong. Logic Methodology and Philosophy of Science.*

Lindenmayer, A. & Rozenberg, G. (1972) Developmental systems and languages. *Proc. 4 ACM Symp. Theory Comp.*, 214–21.

Longuet-Higgins, C. (1969) What biology is about. *Towards a Theoretical Biology, 2. Sketches*, pp. 227–32. (ed. Waddington, C.). Edinburgh: Edinburgh University Press.

Raven, C.P. (1968) A model of pre-programmed differentiation of the larval head region in *limnaea stagnalis. Acta Biotheoretica*, **43**, 316–29.

Rozenberg, G. (1972) TOL systems and languages. *Inf. Control*, to appear.

Rozenberg, G. & Doucet, P. (1971) On 0-L languages. *Inf. Control*, **19**, 302–18.

Surapipith, V. & Lindenmayer, A. (1969) Thioguanine-dependent light sensitivity of perithecial initiation in *Sordaria fimicola. J. gen. Microbiol.*, **57**, 227–37.

19

Parallel and Serial Methods of Pattern Matching

D. J. Willshaw and O. P. Buneman
Theoretical Psychology Unit
University of Edinburgh

1. INTRODUCTION

This paper is concerned with two aspects of the 'exact match' problem which is that of searching amongst a set of stored patterns to find those specified by a given partial description. We describe how to design simple content-addressable memories, functioning in parallel, which can do this and which, in some sense, can generalise about the stored data. Secondly, we consider how certain graphical representations of data may be suitable for use in efficient serial search strategies. We indicate how such structures can be used in diagnosis when the availability or cost of tests to be applied cannot be determined in advance.

The type of parallel system to be considered is to store descriptions of a set of patterns, and is then to be used to supplement an incomplete description of a newly presented pattern by matching it against those in store. If this partial description matches one or more of the stored patterns then we would like the memory to provide us with the partial description that these patterns share. If the new pattern does not match any in store then we expect that the information supplied will be according to the relationships between the pattern presented and those in store. The information that we require our memory to provide when given an incomplete description as an address is therefore more than just the response 'yes' or 'no'. In this respect our type of system differs from content-addressable parallel memories used in computer technology, and for the same reason its capabilities exceed that of a switching network which is designed to respond positively when the states of its input channels attain one of a number of combinations of binary values (Richards 1971, Renwick and Cole 1971).

This paper generalises the work of Willshaw (1972) to ensembles which conform to few or no logical constraints. The graphical generalisation of the *multitree* which is used by Willshaw to represent the ensemble will be seen to

have properties which suggest its use as a method of data retrieval in conventional storage systems. This method is applicable to the problem of reducing the amount of serial search needed to identify a pattern in store from a given partial description.

2. THE INDUCTIVE NET

We now review the properties of a parallel device called the Inductive Net (Willshaw 1972), which is closely related to the Associative Net (Willshaw, Buneman and Longuet-Higgins 1969). This is designed to store a selection of patterns chosen from a special type of ensemble. When presented with an incomplete description of an ensemble member it is able to use the information in store to supplement the description of that member. The structure imposed on the ensemble enables the Net to have the additional property of generalisation as it can augment descriptions of ensemble members with which it was not explicitly provided.

Each pattern with which the Net deals is in the form of a binary vector of fixed length, each component of the vector representing the value the pattern takes on a particular binary feature. The restriction placed on the ensemble is that no two features are logically independent – that is, for each pair of features not all the four possible combinations of feature values '++', '+−', '−+', and '−−' are possessed by members of the ensemble, which is said to obey the *four point condition* (Buneman 1971).

The Inductive Net is made up of a set of horizontal lines (input lines) crossing at right angles a set of vertical lines (output lines), binary switches being placed at the intersections so formed. Each possible feature value has one horizontal line and one vertical line identified with it. A pattern is stored in the Net by exciting the horizontal lines and the vertical lines identified with its feature values, and turning on each switch which receives excitation along both the lines on which it is placed. In the retrieval mode, the pattern to be used as the address or *cue* (which is usually incomplete) is input by exciting the appropriate horizontal lines. Each binary switch whose horizontal line is excited and which has been turned on sends a pulse of unit strength down its vertical line which then fires if the number of pulses sent down it equals the number of excited input lines. An inhibitory mechanism placed across each pair of output lines identified with the same feature prevents any signal emerging if both lines are simultaneously active. The set of feature values associated with the output produced in this manner is regarded as the response of the Inductive Net to the given cue.

As an example of how the Inductive Net functions figure 1 shows the Net which has stored the following patterns:

A:	+1	−2	+3	−4
B:	+1	+2	−3	−4
C:	+1	+2	−3	+4
D:	−1	−2	−3	−4.

Each pattern is represented by a binary vector of length 4. To assign a value '$+x$' to a pattern means that this pattern takes value '+' on feature x. Similarly for '$-x$'. From this list of feature values we note, for example, that a pattern with feature value $+2$ also has values $+1$ and -3. This simple correlation has been noted in the Net, for along the horizontal line labelled $+2$, switches $+1$ and -3 have been turned on and switches -1 and $+3$ have not. This line is therefore storing the 'rule' that a pattern with feature value $+2$ also has feature values $+1$ and -3. Similarly, we observe that a pattern with feature value $+3$ also has feature values $+1$, -2 and -4, so this 'rule' is stored in horizontal line $+3$. More complex rules such as 'the presence of $+1$ *and* $+2$ implies the presence of -3' have not been stored in the Net because each horizontal line is identified with only one feature value.

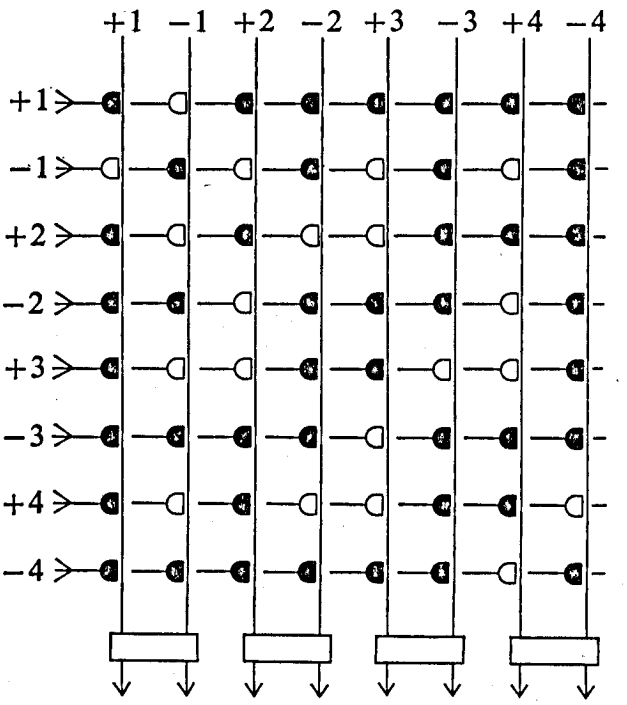


Figure 1. The Inductive Net which has stored patterns A, B, C and D. Switches which are on are coloured black.

In order to discuss the behaviour of the Inductive Net we need the fact that all the information that a selection of patterns from a four point ensemble can tell us about the ensemble itself can be represented by a family of trees which we can draw as a single structure called a *multitree*, in which each node represents a possible member of the ensemble and each link corresponds to the alteration of one or more features.

The multitree constructed from patterns A, B, C and D is shown in figure 2. Each of the four stored patterns is represented by a node and there is a node, which we label X, not identified with any pattern. The links are directed, and the arrows placed on them point towards that region of the multitree having value '+' on the associated feature, the remainder of the multitree taking the value '−'. The reader may like to verify that the multitree of figure 2 is the correct one by checking that its links assign the correct set of feature values to nodes A, B, C and D. (In this simple case each link is identified with only one feature, so that the multitree represents just one tree. In general the multitree represents more than one tree.)

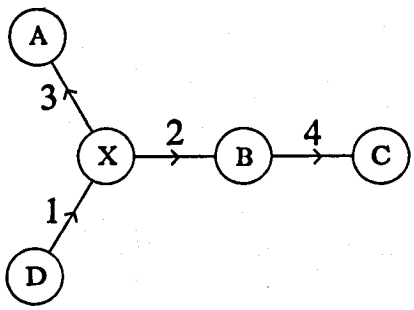


Figure 2. The multitree constructed from patterns A, B, C and D.

Let us imagine that we are given the descriptions of a number of patterns which we know were selected from a four point ensemble, and we are now asked to supplement incomplete descriptions of ensemble members. The appropriate multitree is a very useful tool for this task. First of all, it contains descriptions of all the members of the ensemble whose presence can be inferred from the given sample. That is the reason why the extra node X appears in our example – the four point ensemble from which patterns A, B, C and D were selected must also contain pattern X. Secondly, the multitree has the property that any incomplete set of feature values common to one or more ensemble members specifies that connected region of the multitree containing the nodes associated with them, and so the complete set of feature values common to these ensemble members can be read off from the multitree. For example, if we were asked to supplement the partial description +4, reference to figure 2 shows that feature value +4 specifies node C, and its complete description +1+2−3+4 can be read off from the multitree. Similarly the partial description −2−3 specifies nodes D and X which, as the multitree tells us, share feature values −2−3−4.

It is a straightforward exercise to prove that the Inductive Net can supplement partial descriptions just as well as we would expect from an examination of the multitree constructed from the set of patterns given to the Net to store (Willshaw 1972). Returning to our example once more,

this means that if we input the description +4 to the Net of figure 1 then the output is +1+2−3+4, and the response to the cue −2−3 is −2−3−4.

These are two illustrations of the general theorem which reads:

> If the set of feature values used as input to the Inductive Net specifies a connected region of the multitree then the output is the complete set of feature values common to that region.

The most important property of the Inductive Net is that it is able to supplement descriptions which it was not explicitly given to store. This asset becomes undesirable if we do not wish the Net to generalise in this way. If, for example, we had defined our ensemble to comprise the patterns A, B, C and D and no more, the Net of figure 1 would still supplement partial descriptions on the assumption that X was present as well.

The Inductive Net is therefore not able to augment descriptions of arbitrarily chosen sets of patterns. It is convenient to divide this problem into two parts and consider each separately. We shall show in the next section that we can always produce a parallel system able to supplement descriptions of any set of patterns placed in store. However, if the system is to be able to generalise then the ensemble from which the stored set of patterns is taken must have one of a finite number of types of structure. Some of these types of ensemble do not in fact obey the four point condition. However, if the only hypothesis about the ensemble is that it is not four point (which is in effect a non-hypothesis) then no generalisations can be made. The possible generalisations are prescribed by graphical representations of data of the type discussed later, so we shall delay discussion about the making of generalisations until these representations have been introduced.

3. COMPLEX NETS WHICH DO NOT MAKE GENERALISATIONS

Let us suppose that we would like to design an Inductive Net to supplement the descriptions of A, B, C and D of figure 2, but not X, as it does at the moment. If, for example, we supply as input to the Net the partial description −2−3, this should be enough to distinguish D (−1−2−3−4) from A, B and C. However, reference to the multitree tells us that feature values −2−3 specify nodes D and X, and so the output from the Net is the set of feature values −2−3−4, no information about the value of feature 1 having been recorded on input lines −2 or −3.

The solution is to construct a parallel structure, which we call a *Complex Net*, to record more complex combinations of input and output feature values than the Inductive Net does. We do this by giving the Inductive Net additional horizontal lines, each of which has a mask placed on the front of it which causes the line to fire when a particular combination of feature values occur together in the input pattern. We say that a Complex Net is of size S if each mask looks at no more than S feature values to decide whether or not it should fire. In our example, if we add in a horizontal line which fires only

when feature values -2 and -3 occur in the input pattern, this ensures that the output for the partial description $-2-3$ is now $-1-2-3-4$, as we require. In fact, for this Complex Net to supplement all possible descriptions of A, B, C and D we need the extra masks $+1-3$, $+1-2$, $-2-3$, so this Net is of size 2.

As a second example of the construction of Complex Nets we add to our patterns A, B, C and D two more called E and F. The descriptions of these six patterns are:

A:	$+1$	-2	$+3$	-4
B:	$+1$	$+2$	-3	-4
C:	$+1$	$+2$	-3	$+4$
D:	-1	-2	-3	-4
E:	$+1$	$+2$	$+3$	$+4$
F:	-1	-2	$+3$	-4

This set of patterns violates the four point condition; for example, the four pairs of feature values $+3+4$, $+3-4$, $-3+4$ and $-3-4$ all occur in the above list of feature values.

In order to find out what masks our Complex Net should have so that it may supplement descriptions of just these six patterns, and no more, we use an algorithm which is outlined in section 5 along with the other mathematical propositions. In fact, for our example we need a Complex Net of size 3 with 13 masks of size 2* or higher. These masks are listed below in table 1.

Table 1. Feature values prescribing the masks of size 2 or higher for the Complex Net storing patterns A to F.

$+1-2$	$+1+2+3$
$+1-3$	$+1+2-4$
$-1+2$	$+1+3-4$
$-1-2$	$+2+3-4$
$-1+3$	
$-1-3$	
$+2-3$	
$-2+3$	
$-2-3$	

This ends our discussion of how to design a Complex Net able to answer questions about any set of patterns placed in store. We shall show how Nets of this type can make generalisations after we have discussed certain graphical representations of data.

* The size of a mask is the number of feature values that it must look at in order to decide whether it should fire.

4. GRAPHICAL REPRESENTATIONS AND DIAGNOSTIC KEYS

Burkhard and Keller (1972) have shown that, in the 'closest match' problem, the construction of a graph on the patterns in store can cut down the amount of searching. This problem is that of finding those patterns in a conventional (computer) store which have least Hamming distance from a given 'test' pattern. Their method is to construct that graph on the patterns in store in which pairs of patterns are linked which are separated by a Hamming distance less than some number α. In the search for the patterns closest to the test pattern, if we have already found a pattern distance d from the test pattern we can subsequently discard from the set of closest patterns any pattern we find whose distance is greater than $d+\alpha$ from the test pattern and, without further calculation, we can reject all its neighbours in this graph. The degree to which the search is speeded up depends critically on the choice of α.

In this paper, we have been concerned with the 'exact match' problem, that of identifying a pattern or some of its properties from a partial description of that pattern. Is there a method of decreasing the search time which similarly employs some graph on the patterns? For the time being we shall assume these patterns are to be stored conventionally in either a list or an array and that graphs on these patterns can be constructed either by use of a two-dimensional binary array, or by the use of more complicated pointer structures. Call the n patterns to be stored $O_1, O_2, \ldots, O_n$ each consisting of k binary-valued components or features. A graph on these patterns will be called an F-graph if that graph, when restricted to those patterns which possess any given combination of feature values, is a connected graph. Put otherwise, a graph is an F-graph if for each conjunctive predicate $P = P_1 \cap P_2 \cap P_3 \ldots \cap P_n$ on a subset of the features we can find a connected subgraph whose points are just those patterns which satisfy P. Given an F-graph, we need search only along its links to achieve an exact match and the search can be conducted in such a way that, if we have found a pattern which satisfies some components of the predicate P, then we need move only along links on which these components remain unchanged. Thus a hill climbing procedure on an F-graph which optimises the match with P is bound to achieve an exact match if one exists.

This process is of no use unless the F-graph is such that each pattern is linked, on average, to only a fraction of the others. The complete graph is, trivially, an F-graph but is useless as a device for restricting the search. There is however, for any set of patterns, a unique minimal F-graph on those patterns, that is an F-graph whose lines are lines in any F-graph on those patterns. This graph can be constructed as follows: link every pair of patterns which are Hamming distance one apart by a line, and associate with this line a length of one. Call this graph G_1. Augment G_1 with lines of length two by joining points of Hamming distance two apart, provided these points are not already graphical distance two apart in G_1. Call this augmented graph G_2 and measure distances in G_2 by the shortest length of path (if one exists)

joining two points. Now augment G_2 in the same way with lines of length three, and continue until the largest Hamming distance has been dealt with.

Although the minimal F-graph on the given patterns is well defined, it may be possible, by adding extra patterns, to construct an F-graph with fewer lines. One example of this is the tree-like data considered earlier, where by adding patterns we can achieve an F-graph which is a tree. In this case each new pattern has the property that for every *pair* of features there is some given pattern that has the same values on those features. As we shall see, properties of other F-graphs bear a close relationship to the properties of the networks of the previous section.

It is interesting to compare the use of these graphs with that of another essentially graphical construction, a diagnostic key. This is a number of instructions telling the user how to apply a set of tests to a specimen in order to place it in one of a number of previously determined classes. (Niemela, Hopkins and Quadling 1968; Pankhurst 1970). Such keys depend for their use on the assumption that the result of any test (feature value) can be established if required, and it is also in general impossible to use any partial description that might be available at the outset. While diagnostic keys are usually prepared for their efficiency of operation in terms of the number and cost of the tests necessary to identify an object, their drawback is their extreme inflexibility. We feel that computer diagnosis might benefit from the use of more descriptive data structures; the problem of deciding which tests to perform, when there is a choice, could be computed as the diagnosis proceeds, for it may only be then that the cost or availability of tests can be established.

5. COMBINATORIAL RESULTS AND GENERALISATIONS

We here summarise some useful definitions. Each *pattern* is a binary vector of length k. Associated with the ith component of a set of patterns is the ith *feature* f_i which carries each pattern onto its boolean value. An *elementary predicate* is a predicate of the form f_i or $\neg f_i$ for some feature and an *elementary disjunction* (or *conjunction*) is a disjunction (or conjunction) of elementary predicates.

An elementary disjunction (conjunction)D_1 *contains* an elementary disjunction (conjunction)D_2 if each elementary predicate in D_2 occurs also in D_1. A conjunction D is called a mask if $D \Rightarrow P_i$ for some elementary predicate P_i and there is no smaller conjunction D' contained in D for which $D' \Rightarrow P_i$. The order of a disjunction or conjunction is the number of elementary predicates it contains.

The following propositions hold:

(1) The masks are determined by the minimal elementary disjunctions which are true on every pattern. Equivalently, they are determined by the minimal false disjunctions.

(2) The set of patterns determine and are determined by their masks.

(3) Given a set Ω of patterns, a pattern O' satisfies all the minimal disjunctions for Ω of order less than or equal to some j if for each subset S of the features of order j there is a pattern O in Ω such that

$$f_i(O)=f_i(O') \text{ for all } f_i \text{ in } S.$$

The first of these propositions means that the masks can be calculated from the minimal true disjunctions. The problem of finding these is the same as finding the set of minimal covers contained in a given cover of a set. (A cover of a set S is a set of subsets whose union contains S.) There is a serial algorithm for finding these masks, details of which will be published elsewhere. It is an attractive possibility that the masks might also be calculated by an adaptive mechanism operating on an Inductive Net, but we know of no such method.

The second result ensures, first of all, that if a Complex Net contains all the masks for a set of patterns then those patterns alone are the only complete outputs that this Net can give. However, if an upper limit is placed on the size of the masks that this Net can have, there may be other patterns which have been effectively stored (Proposition 3). Returning to our example of the storage of patterns A to F, if the Net can only have masks of size 1 this means that no minimal disjunctions of order greater than 2 can be used to determine which patterns are in store. Therefore some minimal disjunctions are disallowed and so extra patterns will have been stored in the Net. As proposition 3 indicates, these are found by taking those of the set of 16 possible patterns which were not originally stored. For each of these, if we find that all of the $\binom{4}{2}$ pairs of feature values it possesses occur in the patterns originally stored, we regard this as being a stored pattern too. Having applied this procedure to each of these 10 patterns we are then able to draw the F-graph on the new set of stored patterns. This is shown in figure 3 and we observe that it contains many extra nodes. There are other F-graphs that we can construct from the patterns A to F. If our Complex Net were allowed to have masks of size 2 or less then we check *triplets* instead of *doublets* of feature values in order to find the extra patterns in store. By this means we produce the F-graph shown in figure 4 which contains one extra node. Finally, by checking *quadruplets* we obtain the F-graph which is constructed on the patterns themselves (figure 5). This is as it should be because in this case no minimal disjunctions are disallowed. We say that an F-graph is of order S if it was produced by allowing only minimal disjunctions of order S or less to determine the patterns represented in it.

The point about constructing graphs of this form is that a Complex Net of size $S-1$ – that is, each mask is allowed to look at no more than $S-1$ components of the input – will supplement incomplete descriptions of the patterns represented in the F-graph of order S which may mention patterns other than those given to the Net to store. It is in this sense that a Complex Net is able to perform generalisations. If a Net has had built into it an explicit assumption about the structure of the data it stores, this assumption

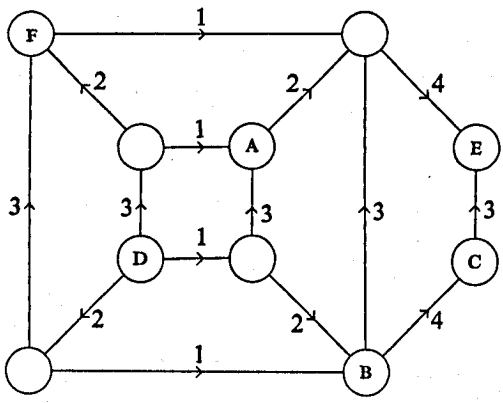


Figure 3. The *F*-graph of order 2 for patterns A to F.

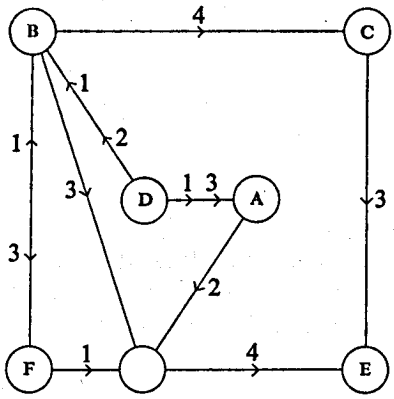


Figure 4. The *F*-graph of order 3 for patterns A to F.

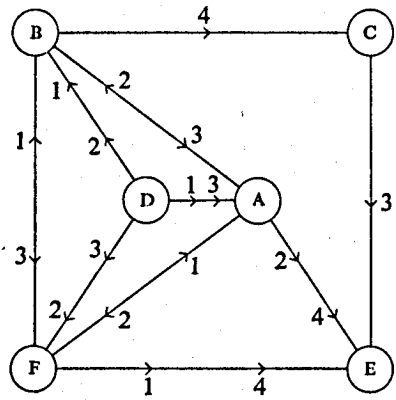


Figure 5. The *F*-graph of order 4 for patterns A to F.

taking the form of an upper limit to the size of the mask it can possess, then the Net will be able to generalise on this basis. On the other hand, if no assumption has been made (that is, there is no limit to the size of mask the Complex Net may have) then it will always supplement descriptions according to the *F*-graph constructed from the patterns given to the Net to store, and so will not be able to generalise.

The family of *F*-graphs that we can construct in this way may also have some application to the diagnosis problem. If we examine the three graphs we have just constructed, the *F*-graph of figure 4 contains fewest links. Assuming that we can mark invented nodes so that they can be discarded when required, it would seem that the *F*-graph with the smallest number of links would be the one to use in diagnosis, where links may be the expensive items. However, we know of no method for producing from a set of patterns the *F*-graph which is minimal in this respect.

We have programs for constructing *F*-graphs and masks for reasonably large data sets, and we hope to put these into practice in some diagnostic application.

Acknowledgements

The authors would like to thank Professor H.C. Longuet-Higgins and Mr C.J. Stephenson for many helpful discussions.

This work was financed by grants from the Science Research Council and the Royal Society.

REFERENCES

Buneman, O.P. (1971) The recovery of trees from measures of dissimilarity. *Mathematics in the Archaeological and Historical Sciences* pp. 387–95 (eds Hodson, F.R., Kendall, D.G. & Tăutu, P.). Edinburgh: Edinburgh University Press.

Burkhard, W.A. & Keller, R.M. (1972) Heuristic approaches to a file searching Problem. *Comp. Science Lab., Princeton Univ. Technical Report 103.*

Niemela, S.I., Hopkins, J.W. & Quadling, C. (1968) Selecting an economical binary test battery for a set of microbial cultures. *Canadian J. Microbiology*, **14**, 271–9.

Pankhurst, R.J. (1970) A computer program for generating diagnostic keys. *Comput. J.*, **13**, 145–51.

Renwick, W. & Cole, A.J. (1971) *Digital Storage Systems.* London: Chapman & Hall.

Richards, R.K. (1971) *Digital Design.* New York: Wiley.

Willshaw, D.J. (1972) A simple model capable of inductive generalisation. *Proc. Roy. Soc. B.*, **182**, 233–47.

Willshaw, D.J., Buneman, O.P. & Longuet-Higgins, H.C. (1969) Non-holographic associative memory. *Nature*, **222**, 960–2.

20

Approximate Error Bounds in Pattern Recognition

T. Ito

Central Research Laboratory
Mitsubishi Electric Corporation, Amagasaki

INTRODUCTION

In the statistical theory of pattern recognition an optimum decision function can be obtained so as to minimize Bayes' decision error. However, it is often difficult to evaluate the Bayes' error Pe because of the discontinuity of the choice function min, which appears in Bayes' error function. A number of approximate error bounds have therefore been proposed: for example, Bhattacharyya bound B, Chernoff bound C, equivocation E. In this paper we introduce a new approximate error bound Q_n and show the following relations among these:

$$Pe \leqslant Q_n \leqslant E \leqslant B \qquad \text{for } n=0, 1, 2, \ldots$$

Moreover, we discuss some other properties of these error bounds and a problem of approximate error bounds for multi-category pattern recognition.

BAYES' DECISION ERROR AND ITS APPROXIMATION

Let $\mathcal{X}=\{x\}$ be an n-dimensional pattern space. Consider the problem of classifying a pattern x into two categories c_1 and c_2. Then, according to statistical decision theory, we have the following Bayes' decision rule:

$$\left.\begin{aligned} p(x, c_1) \geqslant p(x, c_2) &\Rightarrow x \in \mathcal{S}(c_1) \\ p(x, c_1) < p(x, c_2) &\Rightarrow x \in \mathcal{S}(c_2) \end{aligned}\right\}, \tag{1}$$

where $\mathcal{S}(c_1)$ and $\mathcal{S}(c_2)$ denote the sets of patterns of the categories c_1 and c_2, respectively. The decision error for a pattern x is given by

$$r(x)=\min\left[p(x, c_1), p(x, c_2)\right]. \tag{2}$$

Therefore the Bayesian decision error Pe is given by

$$Pe=\int_{\mathcal{X}} \min\left[p(x, c_1), p(x, c_2)\right] dx. \tag{3}$$

In the statistical theory of pattern recognition p can be obtained so as to minimize this decision error. However, it is often difficult to evaluate it

because of the discontinuity of the choice function min. The following approximate error bounds for Pe have been proposed for this reason:

(1) Bhattacharyya bound: B

$$B=\int_{\mathscr{X}}\{p(x,c_1)p(x,c_2)\}^{\frac{1}{2}}\,dx. \tag{4}$$

(2) Chernoff bound: $C(\alpha,\beta)$

$$C(\alpha,\beta)=\int_{\mathscr{X}}[p(x,c_1)]^{\alpha}[p(x,c_2)]^{\beta}\,dx, \tag{5}$$

where $\alpha+\beta=1$, $\alpha\geqslant 0$, $\beta\geqslant 0$.

(3) Equivocation: E

$$E=-\tfrac{1}{2}\int_{\mathscr{X}}p(x)\{p(c_1/x)\log_2 p(c_1/x)+p(c_2/x)\log_2 p(c_2/x)\}\,dx. \tag{6}$$

We propose a new approximate error bound called the Q-function. First, we define the sign function $\mathscr{S}[a-b]$ as follows:

$$\mathscr{S}[a-b]=\begin{cases}1 & (a>b)\\ 0 & (a=b)\\ -1 & (a<b).\end{cases} \tag{7}$$

Then we have

$$\min[a,b]=\tfrac{1}{2}(a+b)-\tfrac{1}{2}(a-b)\mathscr{S}[a-b]. \tag{8}$$

Using this relation, Bayes' error function can be written as

$$\begin{aligned}Pe&=\int_{\mathscr{X}}\min\left[p(x,c_1),\,p(x,c_2)\right]dx\\&=\int_{\mathscr{X}}p(x)\min\left[p(c_1/x),\,p(c_2/x)\right]dx\\&=\tfrac{1}{2}-\tfrac{1}{2}\int_{\mathscr{X}}p(x)(p(c_1/x)-p(c_2/x))\mathscr{S}[p(c_1/x)-p(c_2/x)]\,dx.\end{aligned} \tag{9}$$

The approximation to Pe can be obtained by approximating $\mathscr{S}[a-b]$. One of the simplest functions to use would be

$$y=x^{1/2n+1}. \qquad (n=0,1,2,\ldots) \tag{10}$$

putting

$$y=x^{1/2n+1}=(\mathscr{S}[x])^{2n+1}|x|^{1/2n+1}. \tag{11}$$

If we use this approximation, we have the following approximation Q_n to Pe:

$$Q_n=\tfrac{1}{2}-\tfrac{1}{2}\int_{\mathscr{X}}p(x)\{p(c_1/x)-p(c_2/x)\}[p(c_1/x)-p(c_2/x)]^{1/2n+1}\,dx. \tag{12}$$

We call this Q_n a Q-function.

SOME RELATIONS BETWEEN APPROXIMATE ERROR BOUNDS

We can prove the following relations:

$$Pe\leqslant Q_n\leqslant E\leqslant B. \tag{13}$$

The outline of the proofs is as follows:

(1) *Proof of* $Pe \leqslant Q_n$

We compare eq. (9) and eq. (12). For $|x| \leqslant 1$, we have

$$|\mathscr{S}[x]| \geqslant |x|^{1/2n+1}. \tag{14}$$

Hence

$$(p(x, c_1)-p(x, c_2))\mathscr{S}[p(c_1/x)-p(c_2/x)] \geqslant (p(x, c_1)-p(x, c_2))(p(c_1/x)-p(c_2/x))^{1/2n+1} \geqslant 0. \tag{15}$$

If we subtract the integrals of these functions from $\frac{1}{2}$, we obtain

$$\tfrac{1}{2}-\tfrac{1}{2}\int_{\mathscr{X}} \text{L.H.S.}\, dx \leqslant \tfrac{1}{2}-\tfrac{1}{2}\int_{\mathscr{X}} \text{R.H.S.}\, dx.$$

where L.H.S. and R.H.S. are the left-hand side and the right-hand side of eq. (15), respectively. Thus we obtain

$$Pe \leqslant Q_n. \tag{16}$$

Before proceeding to the proofs of $Q_n \leqslant E$ and $E \leqslant B$, we transform the random variable x into $p(x)$. Then we have

$$Pe = \int_{\mathscr{D}} \min[\pi, 1-\pi]\, dp \tag{17}$$

$$B = \int_{\mathscr{D}} \{\pi(1-\pi)\}^{\frac{1}{2}}\, dp \tag{18}$$

$$E = -\tfrac{1}{2}\int_{\mathscr{D}} \{\pi \log_2 \pi + (1-\pi)\log_2(1-\pi)\}\, dp \tag{19}$$

$$Q_n = \tfrac{1}{2}-\tfrac{1}{2}\int_{\mathscr{D}} (2\pi-1)(2\pi-1)^{1/2n+1}\, dp, \tag{20}$$

where $\mathscr{D}$ is the probability space in terms of $p(x)$, and $\pi = p(c_1/x)$, $1-\pi = p(c_2/x)$.

(2) *Proof of* $Q_n \leqslant E$

Since the integrands of Q_n and E are positive, we may prove

$$\tfrac{1}{2}-\tfrac{1}{2}(2\pi-1)^{(2n+2)/(2n+1)} \leqslant -\tfrac{1}{2}\{\pi \log_2 \pi + (1-\pi)\log_2(1-\pi)\}. \tag{21}$$

First we show $E \geqslant Q_0$, and we prove $Q_n \geqslant Q_{n+1}$ $(n=0, 1, 2, \ldots)$.

Let

$$\psi(\pi) = -\tfrac{1}{2}\{\pi \log_2 \pi + (1-\pi)\log_2(1-\pi)\} - \{\tfrac{1}{2}-\tfrac{1}{2}(2\pi-1)^2\} \tag{22}$$

$$\frac{d\psi(\pi)}{d\pi} = -\tfrac{1}{2}\log_2 \frac{\pi}{1-\pi} + 2(2\pi-1) \tag{23}$$

$$\frac{d^2\psi(\pi)}{d\pi^2} = -\tfrac{1}{2}\frac{1}{\log_e 2}\frac{1}{\pi(1-\pi)} + 4 \tag{24}$$

$$\frac{d^2\psi(\pi)}{d\pi^2} = 0 \Rightarrow \left.\begin{aligned} \pi_1 &= \tfrac{1}{2}\left(1-\left\{1-\frac{1}{2\log_e 2}\right\}^{\frac{1}{2}}\right) \\ \pi_2 &= \tfrac{1}{2}\left(1+\left\{1-\frac{1}{2\log_e 2}\right\}^{\frac{1}{2}}\right) \end{aligned}\right\}. \tag{25}$$

Then $\psi(\pi)$ is

(a) convex in $0<\pi<\pi_1$
(b) concave in $\pi_1<\pi<\pi_2$
(c) convex in $\pi_2<\pi<1$ } (26)

Moreover, $\psi(0)=\psi(\frac{1}{2})=\psi(1)=0$ and $\psi(\frac{1}{4})>0$ and $\psi(\frac{3}{4})>0$. Therefore we have

$$E \geqslant Q_0. \tag{27}$$

Then $Q_n \geqslant Q_{n+1}$ can be shown as follows:

$$Q_n - Q_{n+1} = \left\{\tfrac{1}{2} - \tfrac{1}{2}\int_{\mathscr{D}} (2\pi-1)^{\frac{2n+2}{2n+1}}\, dp\right\} - \left\{\tfrac{1}{2} - \tfrac{1}{2}\int_{\mathscr{D}} (2\pi-1)^{\frac{2n+4}{2n+3}}\, dp\right\}$$

$$= \tfrac{1}{2}\int_{\mathscr{D}} |2\pi-1| \left\{ |2\pi-1|^{\frac{1}{2n+3}} - |2\pi-1|^{\frac{1}{2n+1}} \right\} dp \geqslant 0. \tag{28}$$

Therefore we have $E \geqslant Q_n$ from eq. (27) and eq. (28).

(3) *Proof of* $E \leqslant B$

This proof is carried out similarly. Since the integrands of E and B are positive, we may prove

$$\{\pi(1-\pi)\}^{\frac{1}{2}} \geqslant -\tfrac{1}{2}\{\pi \log_2 \pi + (1-\pi)\log_2(1-\pi)\}. \tag{29}$$

Let

$$\phi(\pi) = \{\pi(1-\pi)\}^{\frac{1}{2}} - \{-\tfrac{1}{2}(\pi \log_2 \pi + (1-\pi)\log_2(1-\pi))\}. \tag{30}$$

$\phi(\pi) \geqslant 0$ can be proved in a manner similar to that of the proof of $E \geqslant Q$.

$$\frac{d\phi(\pi)}{d\pi} = \tfrac{1}{2}\frac{1-2\pi}{\sqrt{\pi(1-\pi)}} + \tfrac{1}{2}\log_2 \frac{\pi}{1-\pi} \tag{31}$$

$$\frac{d^2\phi(\pi)}{d\pi^2} = \tfrac{1}{2}\frac{1}{(\log_e 2)\pi(1-\pi)\{\pi(1-\pi)\}^{\frac{1}{2}}}[-\tfrac{1}{2}\log_e 2 + \{\pi(1-\pi)\}^{\frac{1}{2}}]. \tag{32}$$

The solutions of $[d^2\phi(\pi)]/(d\pi^2)=0$ can be given as

$$\pi_1 = \tfrac{1}{2}[1 - \{1-(\log_e 2)^2\}^{\frac{1}{2}}]$$
$$\pi_2 = \tfrac{1}{2}[1 + \{1-(\log_e 2)^2\}^{\frac{1}{2}}]. \tag{33}$$

Therefore, $\phi(\pi)$ is

(a) convex in $0<\pi<\pi_1$
(b) concave in $\pi_1<\pi<\pi_2$
(c) convex in $\pi_2<\pi<1$ } . (34)

Moreover, $\phi(0)=\phi(\frac{1}{2})=\phi(1)=0$, $\phi(\frac{1}{4})>0$ and $\phi(\frac{3}{4})>0$. Therefore, we have $\phi(\pi) \geqslant 0$, that is,

$$B \geqslant E. \tag{35}$$

Thus, from (1), (2) and (3), we can conclude

$$Pe \leqslant Q_n \leqslant E \leqslant B. \tag{36}$$
$$(n=0, 1, 2, \ldots)$$

COMMENTS ON THE APPROXIMATE ERROR BOUNDS

(1) The Q-function Q_0 for $n=0$ can be transformed as follows:

$$Q_0=\tfrac{1}{2}-\tfrac{1}{2}\int_{\mathscr{X}} p(x)[p(c_1/x)-p(c_2/x)]^2\,dx.$$
$$=\tfrac{1}{2}\int_{\mathscr{X}}\left\{p(x)-\frac{[p(x,c_1)-p(x,c_2)]^2}{p(x)}\right\}dx$$
$$=\int_{\mathscr{X}}\frac{2p(x,c_1)p(x,c_2)}{p(x,c_1)+p(x,c_2)}\,dx. \tag{37}$$

This means that Q_0 is a harmonic mean of $p(x, c_1)$ and $p(x, c_2)$. Thus, the Equivocation E is located between the geometric mean B (Bhattacharyya bound) and the harmonic mean Q_0 (Q-function).

(2) T. Kailath (1967) conjectured 'The Chernoff bound is in general closer to the error probability than the Bhattacharyya bound'. But this conjecture can be shown to be false:

Suppose

$$\int_{\mathscr{D}}\pi^{\alpha}(1-\pi)^{1-\alpha}\,dp=\int_{\mathscr{D}}\pi^{1-\alpha}(1-\pi)^{\alpha}\,dp.$$

Then we have

$$C=\int_{\mathscr{D}}\pi^{\alpha}(1-\pi)^{1-\alpha}\,dp$$
$$=\tfrac{1}{2}\int_{\mathscr{D}}\{\pi^{\alpha}(1-\pi)^{1-\alpha}+\pi^{1-\alpha}(1-\pi)^{\alpha}\}\,dp$$
$$\geqslant\int_{\mathscr{D}}\{\pi(1-\pi)\}^{\frac{1}{2}}\,dp=B. \tag{38}$$

THE Q-FUNCTION AND ERROR PROBABILITY OF THE NN-RULE

The NN-rule is the nearest neighbour decision rule proposed by T. M. Cover and P. Hart (1967). Suppose that we have some knowledge about the pattern space such as $\mathscr{X}\times\Xi=\{(x_1,\theta_1),(x_2,\theta_2),\ldots,(x_n,\theta_n)\}$. The NN-rule states that for a given pattern x, if $\min d(x_i, x)=d(x_j, x)$, and x_j belongs to the category θ_j, then x belongs to the category θ_j, where d is a certain metric.

Let R be the error probability of the NN-rule and Pe the Bayes' decision error. Then Cover and Hart proved that, for a 2-category problem,

$$Pe\leqslant R\leqslant 2Pe(1-Pe). \tag{39}$$

In this section we show $R=Q_0$, assuming Cover-Hart's hypothesis.

Let $x_j\in\{x_1,\ldots,x_n\}$ be the nearest neighbour of x and θ_j be the corresponding category. Then the conditional NN risk $r(x, x_j)$ is

$$r(x,x_j)=p(\theta_1/x)p(\theta_2/x_j)+p(\theta_2/x)p(\theta_1/x_j). \tag{40}$$

We assume Prob. $[x_j\to x]=1$ as in Cover and Hart (1967). Then

$$p(\theta_i/x_j)\to p(\theta_i/x)\qquad\text{for } i=1,2. \tag{41}$$

Hence we have

$$r(x,x_j)\to r(x)=2p(\theta_1/x)p(\theta_2/x) \tag{42}$$

$$R=E[r(x)]$$
$$=\int_{\mathscr{X}} p(x)r(x)\,dx$$
$$=\int_{\mathscr{X}} p(x)\cdot 2p(\theta_1/x)p(\theta_2/x)\,dx \tag{43}$$

$$p(\theta i/x)=\frac{p(x,\theta i)}{p(x)}=\frac{p(x,\theta i)}{p(x,\theta_1)+p(x,\theta_2)} \qquad \text{for } i=1,2. \tag{43}$$

Thus we obtain

$$R=\int_{\mathscr{X}}\frac{2p(x,\theta_1)p(x,\theta_2)}{p(x,\theta_1)+p(x,\theta_2)}\,dx \tag{44}$$

$$R=Q_0. \tag{45}$$

APPROXIMATE ERROR BOUNDS FOR MULTI-CATEGORY PATTERN RECOGNITION

So far we have concerned ourselves with the 2-category pattern recognition problem. In this section we discuss the problem of approximate error bounds in multi-category pattern recognition.

According to statistical decision theory, a decision rule for the multi-category pattern recognition can be given as

$$\max[p(x,c_1),\ldots,p(x,c_m)]=p(x,c_j)\Rightarrow x\in\mathscr{S}(c_j)(j=1,2,\ldots,m). \tag{46}$$

The error rate by this decision rule is

$$Pe_j=p(x,c_1)+\ldots+p(x,c_{j-1})+p(x,c_{j+1})+\ldots+p(x,c_m). \tag{47}$$

Hence Bayes' decision error is

$$P_e^{(m)}=\int_{\mathscr{X}}\min[Pe_1,Pe_2,\ldots,Pe_m]\,dx \tag{48}$$

(1) Chernoff bound for multi-category problem

The Chernoff bound can be given as an extension of the 2-category case in the following way. First we show

$$\min[A_1,A_2,\ldots,A_m]\leqslant A_1^{\alpha_1}A_2^{\alpha_2}\ldots A_m^{\alpha_m}, \tag{49}$$

where $\alpha_1+\alpha_2+\ldots+\alpha_m=1$, $\alpha_i\geqslant 0$, $A_i\geqslant 0$ $(i=1,2,\ldots,m)$. This can be shown by mathematical induction; that is, for $m=1$ and $m=2$, it is obvious, and we assume that it holds for $m=k$. Then

$$\begin{aligned}\min[A_1,\ldots,A_k,A_{k+1}]&=\min[\min[A_1,\ldots,A_k],A_{k+1}]\\&\leqslant\min[(A_1^{\beta_1}\ldots A_k^{\beta_k}),A_{k+1}]\\&\leqslant(A_1^{\beta_1}A_2^{\beta_2}\ldots A_k^{\beta_k})^{\gamma}A_{k+1}^{1-\gamma}\\&=A_1^{\beta_1\gamma}A_2^{\beta_2\gamma}\ldots A_k^{\beta_k\gamma}A_{k+1}^{1-\gamma}\\&=A_1^{\alpha_1}A_2^{\alpha_2}\ldots A_k^{\alpha_k}A_{k+1}^{\alpha_{k+1}}.\end{aligned} \tag{50}$$

$$\begin{aligned}\alpha_1+\alpha_2+\ldots+\alpha_{k+1}&=\beta_1\gamma+\beta_2\gamma+\ldots+\beta_k\gamma+(1-\gamma)\\&=1\end{aligned} \tag{51}$$

Thus we have proved eq. (49). Using this result, the Chernoff bound for the multi-category problem can be given as

$$C(\alpha_1, \alpha_2, \ldots, \alpha_m)=\int_{\mathcal{X}} Pe_1^{\alpha_1} Pe_2^{\alpha_2} \ldots Pe_m^{\alpha_m}\, dx \tag{52}$$

where $\alpha_1+\alpha_2+\ldots+\alpha_m=1$, $\alpha_i \geqslant 0$ $(i=1, 2, \ldots, m)$.

(2) Bhattacharyya bound for multi-category problem

$$B^{(m)}=\int_{\mathcal{X}} \{Pe_1 Pe_2 \ldots Pe_m\}^{1/m}\, dx. \tag{53}$$

This can be obtained as a special case of the Chernoff bound (15) with $\alpha_1=\alpha_2=\ldots=\alpha_n=1/m$.

(3) Equivocation for multi-category problem

The equivocation for the multi-category problem may be taken as

$$E^{(m)}=-\frac{1}{m}\int_{\mathcal{X}} p(x)\{p(c_1/x)\log_2 p(c_1/x)+\ldots+ p(c_m/x)\log_2 p(c_m/x)\}\, dx. \tag{54}$$

In this way we can obtain several approximate error bounds for the multi-category problem, but their relations are not yet known.

(4) Error bounds based on partial dependence between categories

We consider the problem of finding approximate error bounds for the multi-category problem, using error bounds for the 2-category problem.

The Bayes' error $Pe^{(m)}$ can be transformed into

$$Pe^{(m)}=\sum_{i<j}\left\{\int_{\mathcal{X}^{(j)}} p(x, c_i)dx+\int_{\mathcal{X}^{(i)}} p(x, c_j)dx\right\} \tag{55}$$

where

$$\mathcal{X}^{(i)}=\{x \mid p(x, c_i)=\max_k [p(x, c_k)]\}. \tag{56}$$

Let Δ_{ij} and $\hat{\Delta}_{ij}$ be given as

$$\begin{aligned}\Delta_{ij}&=\{x|p(x, c_i)\geqslant p(x, c_j)\}\\ \hat{\Delta}_{ij}&=\{x|p(x, c_i)<p(x, c_j)\}.\end{aligned} \tag{57}$$

Then we have

$$\Delta_{ij}\supseteq\mathcal{X}^{(i)},\ \hat{\Delta}_{ij}\supseteq\mathcal{X}^{(j)}. \tag{58}$$

Hence we have

$$\begin{aligned}Pe^{(m)}&=\sum_{i<j}\left\{\int_{\mathcal{X}^{(j)}} p(x, c_i)\, dx+\int_{\mathcal{X}^{(i)}} p(x, c_j)\, dx\right\}\\ &\leqslant\sum_{i<j}\left\{\int_{\hat{\Delta}ij} p(x, c_i)\, dx+\int_{\Delta ij} p(x, c_j)\, dx\right\}\\ &=\sum_{i<j}\int_{\mathcal{X}} \min [p(x, c_i), p(x, c_j)]\, dx.\end{aligned} \tag{59}$$

Using the approximate error bounds for the 2-category problem, we have the following bounds:

$$Pe^{(m)} \leqslant \sum_{i<j} \int_{\mathscr{X}} \{p(x, c_i)p(x, c_j)\}^{\frac{1}{2}}\, dx \tag{60}$$

$$Pe^{(m)} \leqslant \sum_{i<j} \int_{\mathscr{X}} [p(x, c_i)]^{\alpha_{ij}} [p(x, c_j)]^{1-\alpha_{ij}}\, dx \tag{61}$$

$$(1 \geqslant \alpha_{ij} \geqslant 0)$$

$$Pe^{(m)} \leqslant \sum_{i<j} \int_{\mathscr{X}} \frac{2p(x, c_i)p(x, c_j)}{p(x, c_i)+p(x, c_j)}\, dx \tag{62}$$

$$Pe \leqslant -\tfrac{1}{2} \sum_{i<j} \int_{\mathscr{X}} p(x)\{p(c_i/x) \log_2 p(c_i/x) + p(c_j/x) \log_2 p(c_j/x)\}\, dx. \tag{63}$$

CONCLUSION

In this paper we have introduced a new error bound the Q-function, and we have proved that

(1) $Pe \leqslant Q_n \leqslant E \leqslant B \quad (n=0, 1, 2, \ldots)$,

and (2) Q_0 coincides with the decision error of the NN-rule of Cover and Hart.

Q_0 has been shown to be the harmonic mean, whereas the Bhattacharyya bound B is the geometric mean. A counter-example to T. Kailath's conjecture '$C \leqslant B$' is also given. Moreover, approximate error bounds for the multi-category problem are given, but their detailed properties are left open.

Acknowledgements

The author is indebted to Mr K. Akita and Mr M. Fukushima for stimulating discussions.

REFERENCES

Cover, T.M. & Hart, P.E. (1967) Nearest neighbor pattern classification. *IEEE Trans.* IT-13, no. 1.

Hellman, M. & Raviv, J. (1970) Probability of error, equivocation and the Chernoff bound. *IEEE Trans.* IT-16, no. 4.

Ito, T. (1971) Probability of error in pattern recognition. Technical Report of Automata Theory Group of I.E.C.E. of Japan (in Japanese).

Ito, T. (1971) On approximate error bounds in pattern recognition. Technical Report of Information Theory Group of I.E.C.E. of Japan (in Japanese).

Ito, T. & Fukushima, M. (1971) On approximate error bounds for Gaussian distribution. Technical Report of Information Theory Group of I.E.C.E. of Japan (in Japanese).

Kailath, T. (1967) The divergence and Bhattacharyya distance measures in signal selection. *IEEE Trans. COM-15*, no. 1.

Kobayashi, H. & Thomas, J. (1967) Distance measures and related criteria. *Proc. 5th Annual Allerton Conference on Circuits and Systems.*

21

A Look at Biological and Machine Perception

R. L. Gregory
Brain and Perception Laboratory
University of Bristol

The study of perception is divided among many established sciences: physiology, experimental psychology and machine intelligence; with several others making contributions. But each of the contributing sciences tends to have its own concepts, and ways of considering problems. Each – to use T.S. Kuhn's term (1962) – has its own 'paradigm', within which its science is respectable. This can make co-operation difficult, as misunderstandings (and even distrust) can be generated by paradigm differences. This paper is a plea to consider perceptual phenomena from many points of view, and to consider whether a general paradigm for perception might be found. We may say at once that the status of perceptual phenomena is likely to be odd, as science is in general concerned with the object world; but perceptions are not objects, though they are in some sense related to objects. It is this relation between perceptions and objects which is the classical philosophical problem, and it cannot be ignored when we consider perception as a territory for scientific investigation. This territory is essentially odd: its phenomena tend to be illusions – departures from the world – rather than facts of the world. It requires a conceptual somersault to accept illusions as the major facts of a science! But this we must do; and once this decision is taken, we can hardly expect physics (or physiology) to provide the paradigm for perception.

Machine intelligence and cognitive psychology (though certainly not all perceptual theories) may agree in regarding perceptions as inferences – inferences based on strictly inadequate data. Our reason for seeing perceptions as inferences is that perception is predictive. Perceptual prediction is of two kinds. First: to properties of objects which cannot at that time be sensed directly, or 'monitored'. This applies to hidden parts of objects, to the three-dimensional form of an object given as a plane optical projection, and to non-optical properties such as hardness and mass, which in biological perception are vitally important. The second kind of prediction exhibited by biological perception is prediction to the immediate future. This allows neural

conduction and processing time to be cut to zero, so that in skilled performances there is typically zero 'reaction-time'. (This implies that the classical stimulus-response notion, though applicable to reflexes, is not appropriate for perception.) Prediction to the future can also allow behaviour to continue appropriately through gaps in the available sensory data. Prediction to the future is vitally important because dangers, rewards, problems and solutions, all lie in the future.

Although it is convenient for experimental purposes to think of perception in stimulus-response terms, the immense contribution of stored data, required for prediction, makes us see perception as largely cognitive. Current sensory data cannot be sufficient for perception or control of behaviour: it must select relevant facts and generalisations from the past, rather than control behaviour directly from present stimuli.

The importance of prediction – which requires stored knowledge – makes us see perception in cognitive as well as in physiological terms. Although there must be physiological mechanisms to carry out the cognitive logical processes, of generalising and selecting stored data, the concepts we need for understanding what the physiology is carrying out are not part of physiology. We do not derive a cognitive paradigm from physiology, though every move is made by physiological components. (We find this situation in games, such as chess. The moves are physical moves, of pieces on a board; but we do not understand the game from the moves, without knowing the rules and where success and failure lie.) We may suggest that it is just this essential cognitive component of biological perception – which unfortunately is very difficult to investigate – which makes machine intelligence potentially important for understanding biological perception. But, again unfortunately, programs adequate for comparable machine perceptual performance do not as yet exist. Judging from biological perception, perceiving machines will not be of intellectual interest, or effective, until they are capable of using sensed data for making these two kinds of predictions, based on stored knowledge of objects. To increase the sensory capacity of machines, to try to avoid this cognitive component, is futile as a solution because there will always be hidden aspects and properties of objects, and the future cannot in principle be monitored. So to produce machines with accurate ranging devices, or other features for reducing the ambiguity of images or other sensory data, is merely to postpone the problem. However ingenious the engineering may be they are conceptually dull and may hide the essential problem by their (limited) success.

When we start to compare the visual perception of animals with present machine perception – by regarding both as giving inferences about external objects from ambiguous sensed data – we strike a paradox. The perceptual performance of quite primitive organisms is greater than the most sophisticated machines designed for scene-analysis or self-guided response to objects. On the other hand even the most advanced brains are weaker at performing

logical operations than are simple devices, of cogs or switches. Why is it that machines, such as computers, can perform logical tasks so well where brains fail; and yet cannot begin to compete with biological object-recognition? This may seem strange enough; but if Helmholtz (1963) was right (and there is every reason to believe him right) to regard perceptions as conclusions of 'unconscious inference', then we must face a truly odd situation – for if perception involves sophisticated inference why are brains weaker at performing logically than are machines? We might suppose that ability to infer was developed in brains, at the start of biological perception; but that this ability was locked away, and is not available for other kinds of problem solving. Or we might suppose that the processes of perceptual inference are different from what we call logic.

It is clear that the physical characteristics of components available for seeing machines are very different from those present in brains. The first are relatively free from drift and noise; but they are relatively large, and cannot carry so many inputs or outputs. This makes parallel processing convenient for biological computing, and serial computing more convenient for man-made computers. Can we suppose that these physical differences lead to the supposed different kinds of inference? If so, biological perception seems to demonstrate powers of *parallel* processing, while computers demonstrate very different powers of *serial* processing. In addition, we might argue that the biologically unique power of human logical problem solving is due (in whole or part) to language, and special symbolic aids, including: mathematical and logical notations, 'digit' fingers for counting, and the abacus – all helping us to infer serially.

Can this suggestion be supported? We might start by noting that in general the more an aid differs from what it aids, the greater the improvement it can confer. (To take an example: knives, forks and spoons are so useful as aids to eating, essentially because our hands are not like knives, forks or spoons. So, if we know that Martians eat with such utensils, we could make a shrewd guess that their 'hands' are not like our knives, or forks, or spoons.) If we accept this as a principle, that: *the greater the difference between an aid and the aided, the greater the possible improvement*, then we have some support for the notion that biological non-perceptual problem solving is by *parallel* rather than by serial processing – because it is *serial* aids which are so effective. Although this cannot be claimed as a strong argument, it seems worth some consideration. Finally, if brains are good at perceptual inferences through adopting *parallel* processing, perhaps it will be necessary to adopt parallel processing for machine perception.

However this may be, we have now learned, from painful experience, that machine perception is extremely difficult to achieve. This implies that we do not understand biological perception adequately to design corresponding machines; but perhaps we can learn some useful things from living systems, including ourselves, clearly capable of perception. Continuing our emphasis

on the cognitive component of biological perception, we will assume that the details of the physiology are less important in this context than the kinds of inference (from stored and sensory data) and the cognitive strategies by which objects and the future are inferred. But of course perceptual inference is not infallible. There are errors, and these may occur systematically in certain situations. Much as the logician may use logical paradoxes for revealing the nature of logic, so cognitive psychology can use perceptual phenomena for revealing perceptual assumptions and inference procedures. This is however to assume that at least some perceptual phenomena are due to *misplaced strategies of inference* rather than to *physiological malfunction.* Such visual phenomena as after-images, we may safely attribute to the physiology of the system, because the pattern of intense stimulation of the retina transfers to (is superimposed on) any other pattern. Other visual phenomena are specific to the pattern or to its probable significance in terms of the object world. These phenomena we may attribute to inference strategies rather than to the mechanism carrying out the strategy.

It seems useful, if we regard perceptions as the results of inference, to call perceptions *hypotheses.* This draws attention to their similarity, on this view, to hypotheses in science. In both cases slender data serve to make decisions appropriate through inference to situations which cannot be monitored directly, and which may lie in the future. A detailed comparison of perceptions as hypotheses in this sense could be rewarding. Meanwhile, we are in a position to describe *perceptual phenomena* as 'inappropriate hypotheses'. By asking why inappropriateness is generated, we may learn something of the inference procedures (and sometimes the physiology) of perception. As an example, we shall consider some new visual phenomena in these terms. These phenomena are not distortions, but are perceptually created visual features. If they are generated by misplaced inference, we may call them: 'cognitive creations' (Gregory 1972). This loaded term will be used; but they will be considered in terms of alternative paradigms.

A PARADIGM FOR COGNITIVE CREATIONS?

It is possible to devise simple line figures which evoke illusory contours and create large areas of enhanced or diminished brightness. Unlike the well-known brightness contrast effects, these illusory contours can occur in regions far removed from regions of physical intensity difference; and they can be orientated at any angle to physical present contours. Figure 1a is the figure described by Kanizsa (1955). An illusory triangle is observed whose apices are determined by the blank sectors of the three disks. The 'created' object appears to have sharp contours, which bound a large region of enhanced brightness.

We may discover what features are necessary for producing these creations by removing parts of this figure. Figures 1b,c,d, show such a sequence. Three dots spaced as the apices of an equilateral triangle (figure 1b) give

no created contours, although they are readily seen as indicating a triangle. The broken triangle (figure 1c) does not evoke the figure (except perhaps slightly after the effect has been observed in figure 1a); but combining the equilaterally spaced dots with the broken triangle (figure 1d) does evoke the illusory object, though less markedly than with the sectored disks of figure 1a.

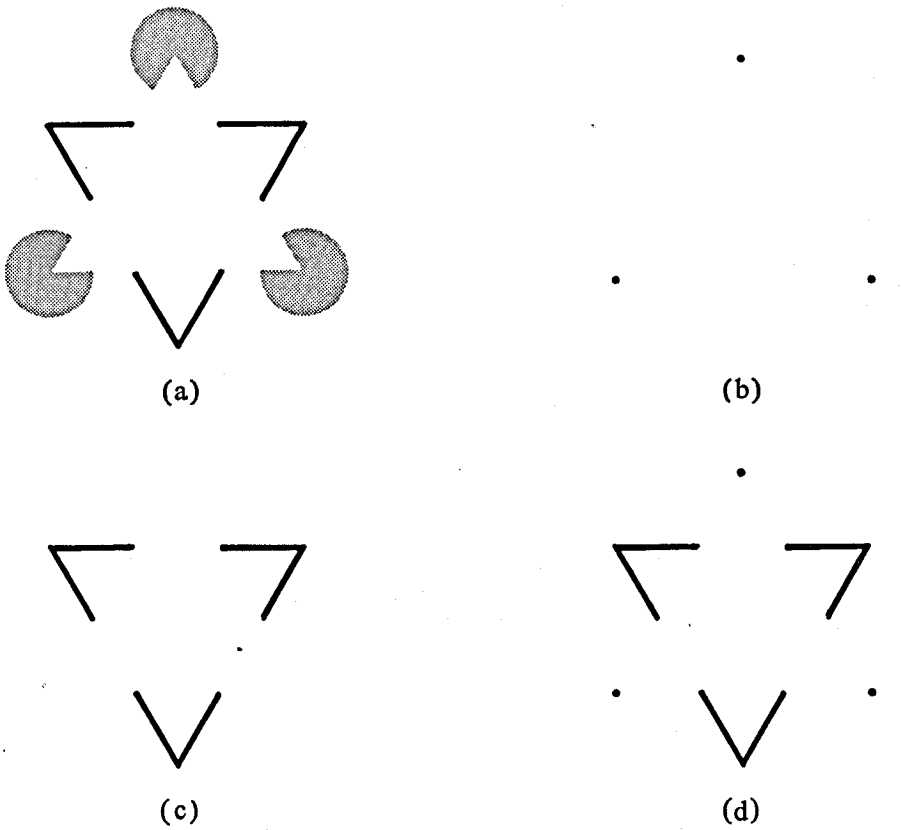


Figure 1

We can discount eye movements as important for these effects, for if the retinal image is stabilised (as by viewing the figures as after-images, produced with an intense electronic flash), then the effects are seen in the after-images, fixed to and moving precisely with the retina, with movement of the eyes. (When the initial positive after-image changes to a negative after-image, the whiter-than-white created area changes to a corresponding blacker-than-black area, just as when the figures are changed from negative to positive by optical means and viewed normally.)

These effects have particular theoretical interest, for they might be explained in terms of at least three very different perceptual paradigms. They might,

with at least initial plausibility, be described in terms of: (1) gestalt, (2) physiological, or (3) cognitive-hypotheses paradigms.

(1) The gestalt paradigm is satisfied by supposing that the 'created' shapes are 'good figures', having high 'closure' and so on (as accepted by Kanizsa (1955) for the first figure).

(2) The physiological paradigm would be satisfied with the supposition that feature detector cells of the striate cortex are activated by the edges of the disk sectors (or less effectively by the dots) to give the appearance of continuous lines, though only their ends are given by stimulation.

(3) The cognitive-hypothesis paradigm, in which perceptions are regarded as going beyond available data to give 'object hypotheses' (Gregory 1970), would be satisfied by supposing that the created features are 'postulated', as supposed masking objects, to 'account' for the blank sectors of what are likely to be complete disks and the breaks in the lines of what is likely to be a complete triangle. The sectors and gaps are supposed, on this cognitive paradigm, to elicit the hypothesis of a masking object, lying in front of the given figure, which is likely to be complete but if complete must be partially masked. Like all other perceptions, this is a cognitive creation; but in this instance it is not appropriate to the facts and so is an illusion.

These paradigm views of the phenomena each give a different logical account and a different logical status to the phenomena. They also each have different experimental predictions, and so can be regarded as scientific rather than metaphysical statements. All three rival paradigms allow that there is a physiological basis; so each can ask: 'where is the fiction generated?' Simple experiments provide clear cut answers which at least rule out several possibilities.

By adopting the technique devised by Witasek in 1899, of sharing parts of the figures between the two eyes with a stereoscope, it is easy to show that the effect is not retinal in origin, but must be after the level of binocular fusion. This follows because the effect holds when the sectored disks are viewed with one eye and the interrupted line triangle with the other eye, when they are stereoscopically fused. By changing the angles of the disk sectors, so that they no longer meet by straight line interpolation, we find that the effect still occurs. The created form is now changed, to give interpolation with a *curved fictional contour*. This may be seen in figure 2. This new effect seems to increase the plausibility of the cognitive fiction notion – for it seems unlikely that 'curved-edge' detectors would be selected by the mis-aligned sectors; and it seems that these concave-curved (and other) figures which are created are not especially 'good' figures in the gestalt sense.

The black line on white background figures give a homogeneous whiter-than-white fictional region. The corresponding negative white line on black background gives a blacker-than-black region. The illusory intensity difference can be measured with a reference light spot, as in a matching photometer. The measured brightness difference is about 10 per cent. Both the

black and the white illusory triangles are reported by our subjects as appearing somewhat in front of the rest of the figure. We have measured this objectively, by matching a stereoscopically viewed marker light spot to the apparent distance of the physical and created parts of the figures. This is compatible only with the cognitive paradigm.

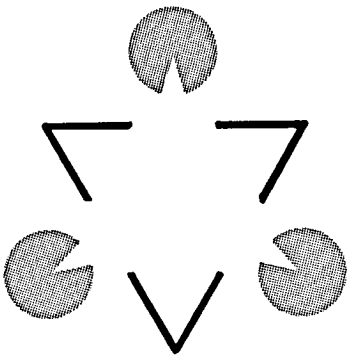


Figure 2

Not only contours, but large homogeneous areas of different brightness are created – but could such areas of different brightness be created by the line detectors of the physiological paradigm? Consider figure 1c. We have lines with gaps; but there is no observed difference in brightness between the inside and the outside of this triangle – and no contours between the gaps. So why should there be contours, and a brightness difference, with the created triangle? The lack of contours in the gaps of figure 1c and the absence of enhanced brightness show that aligned features are not sufficient for producing these effects. What seems to be needed is a high probability of an over-lying object which if it existed would give gaps by masking. This, however, would require processes of a logical sophistication beyond those believed to occur at the striate region; and concepts beyond those of classical physiology – the cognitive concepts of our third paradigm.

We find that at least some of the classical distortion illusions can be generated by these created contours and created regions of different brightness. Figure 3 shows a kind of Poggendorff figure, in which the usual parallel lines are physically absent but are generated by four sectored disks, placed well away from the interrupted oblique figure. Figures such as figure 4 also evoke apparently cognitive contours and they also produce distortion illusions. This seems significant, for how could these distortions be generated by physiological processes, signalling borders, if these borders are not signalled directly by sensory patterns but are, rather, called up from store as fictional hypotheses? It would seem that these effects are not due to physiological interaction effects, such as lateral inhibition; but they are compatible with the notion (Gregory 1965, 1970) that distortions can occur as a result

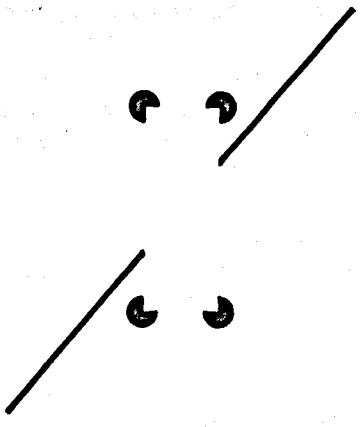


Figure 3

of scaling being set inappropriately to the surrounding objects, or line figures, by following usually predictive assumptions which do not apply to the given objects or line figures. (This happens especially when perspective features occur on flat picture planes.) On this view, it is not the physiology which generates the errors: it is the strategy which leads to error, though of course the strategy is carried out by physiological events. If we do not understand the strategy we will not understand the phenomena, even though we may

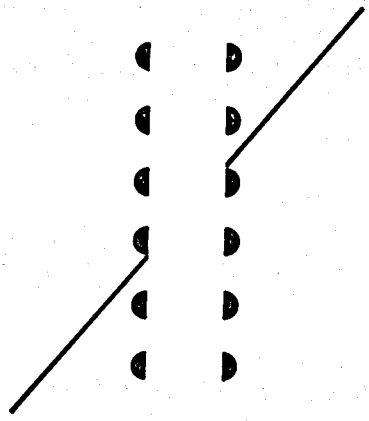


Figure 4

understand the physiology in every detail. If this view is approximately correct, we have an example where simply taking over paradigms of physics or physiology may be unhelpful, or seriously misleading. This remains true though we are not violating any physical or physiological principle with our cognitive paradigm.

The phenomena of perception deserve consideration though they may

appear trivial. There is nothing new in this: some of physics' most dramatic successes came from questioning, and using as tools trivial-looking phenomena which may have been children's toy things. Studying how pith balls, suspended on silk threads, are affected by rubbed amber; and how lode stones, floating freely, point to the only fixed star in the sky, lead to new ways of seeing – to new paradigms of the physical object world. Perhaps only phenomena (and not philosophy) have the power to suggest new paradigms, – to break down old barriers which prevent us seeing how we see.

REFERENCES

Gregory, R.L. (1965) Inappropriate constancy explanation of spatial distortions, *Nature*, **207**, 4999, 891–3.

Gregory, R.L. (1970) *The Intelligent Eye*. London: Weidenfeld & Nicolson.

Gregory, R.L. (1972) Cognitive contours. *Nature*, **238**, 5358, 51–2.

Helmholtz, H. von (1963) *Handbook of Physiological Optics* (1867) (ed. Southall, J.P.C.S.). New York: Dover Publications, Inc.

Kanizsa, G. (1955) Margini quasi – percettivi in campi con stimulazioni omogonea. *Rivista di Psicologia*, **49**, 7, 30.

Kuhn, T.S. (1962) *The structure of scientific revolutions*. Chicago: University of Chicago Press.

PROBLEM-SOLVING AUTOMATA

22

Some Effects in the Collective Behaviour of Automata

V. I. Varshavsky
Leningrad branch of the Central Economico-mathematical Institute, USSR

What degree of formalization should be used in biology in the study of living systems? If we take quantum mechanics, for example, we can distinguish two stages in its establishment. In the first stage, Niels Bohr created the philosophy of quantum mechanics. There were no formulae as yet, or if there were, they were not of the present kind or not quite of that kind. The second stage was a sudden flowering, the conversion to a strict branch of physics with a great many strict formulae. But this is, after all, the second stage, and in biology even the first stage has not yet been reached.
I. Gel'fand.

In the study of complex behaviour, a particularly effective approach is the atomization of behaviour, that is, the isolation of an elementary behaviouristic act, the investigation of devices (automata) which satisfactorily solve the elementary behavioural problem, and finally the synthesis of complex behaviour as the result of combined activity of such devices (Tsetlin 1963, 1969; Varshavsky and Tsetlin 1966, 1967; Varshavsky 1968, 1969, 1972). In the study of models of collective behaviour of automata, a number of effects may be revealed which are of general methodological value for the investigation of collective behaviour. It is even possible that problems which seem to be far removed from behaviour, such as combinatorial problems, may be solved by organizing the collective activity of many 'solvers'. In fact some examples of this kind are already well known. The parallel directional enumeration procedure using local estimates is a process very near to a collective behaviour process. For example, the solution of the travelling salesman problem may be regarded as the result of many salesmen travelling along the graph exchanging information and using local reference information stored at the nodes of the graph. In previous reports (Varshavsky 1968, 1969) we have already referred to a number of problems connected with the organization of collective behaviour. Here we shall consider two models and the effects arising from them, touching only briefly on the conclusions

drawn from these effects. We remark only that the representations obtained here can be effectively used on the one hand for the organization of certain control processes and on the other hand for estimating the possibilities of a number of formal systems.

Games of many automata are of special interest to us. In studying the behaviour of associations of a large number of players (automata) it is natural to distinguish classes of games in which the payoff functions depend on a small number of parameters.

Such games are exemplified by games with limited interaction. In these games the payoff function of each player is determined by the choice of strategies of a small number of players, his 'partners' in the game. A typical example is the game on a circle. In this game, the payoff function of the player $T^j (j=0, 1, \ldots, N-1)$ depends on his strategy and on those of his right- and left-hand neighbours: $V^j(f_\alpha^{(j-1) \bmod N}, f_\beta^{(j)}, f_\gamma^{(j+1) \bmod N})$, or of only one neighbour (e.g. the left-hand one): $V^j(f_\alpha^{(j-1) \bmod N}, f_\beta^{(j)})$. The definition of the game on the circle naturally extends to games on the plane, on a torus, etc.

It should be noted that a game with limited interaction can be defined by an arbitrary difference scheme. The interaction can be conveniently described by an interaction graph. Each player T^j is represented by a vertex j of the graph. If the payoff function $V^j(f)$ depends on the strategy of the player T^i, then an arrow is drawn from vertex i to vertex j.

In the above examples of games with limited interaction, a characteristic feature is that the description of the rules of the game is independent of the number of players taking part. This property of games with limited interaction is characteristic of the class of homogeneous games, those in which the participants have equal rights.

The homogeneous game is defined by only one payoff function, which considerably simplifies its description.

Consider the game Γ^* of N persons $T^0, T^1, \ldots, T^{N-1}$. Let each player have k_j pure strategies $f_1^j, f_2^j, \ldots, f_{k_j}^j (j=0, \ldots, N-1)$. A set $f=(f_{i_0}^0, \ldots, f_{i_{N-1}}^{N-1})$ of pure strategies $f_{i_j}^j$, used by player T^j, is called a play of the game Γ^*. There are N functions $V^j(f)$ given on the set of plays.

We say that we have a mapping g of the game Γ^* onto itself if we are given:
(1) a one-one mapping g of the set of players onto itself: $gT^j=T^{gj}$;
(2) a one-one mapping of the set of strategies of player T^j onto the set of strategies of player T^{gj}: $gf_{i_j}^j=f_{(gi)_{gj}}^{gj}$.

The mapping g determines a mapping of the set of plays $\{f\}$ onto itself. The play $f=(f_{i_0}^0, \ldots, f_{i_{N-1}}^{N-1})$ is mapped by g to the play $gf=(f_{l_0}^0, \ldots, f_{l_{N-1}}^{N-1})$ where $l=gi$ and $gj=s$, $s=0, \ldots, N-1$.

The mapping g is called an automorphism of the game Γ^* if it preserves the payoff functions, that is, if

$$V^j(f)=V^{gj}(gf) \tag{1}$$

for any play f of Γ^*.

The set of automorphisms of Γ^* forms a group G_{Γ^*}. The game is called homogeneous if the group of automorphisms G_{Γ^*} is transitive on the set of players, that is, if for any pair of players T^i and T^j there exists an automorphism g with $gT^i=T^j$. It is obvious that in a homogeneous game the sets of strategies of all the players are pairwise isomorphic. The set of plays $\{gf\}$ $(g \in G_{\Gamma^*})$ is called the invariant set of plays generated by f. Because the game is homogeneous (all players have equal rights), the average gains of all the players on the invariant set of plays are the same, and equal to the arithmetic mean of the gains of all the players in any one play of the invariant set, that is

$$V(f)=\frac{1}{N_{gf}}\sum_{g\in G_{\Gamma^*}} V^j(gf)=\frac{1}{N}\sum_{j=1}^{N} V^j(f), \tag{2}$$

where N_{gf} is the number of plays in the invariant set $\{gf\}$.

We shall call $V(f)$ the value of the play f (the value of the invariant set $\{gf\}$).

In a homogeneous game the payoff function of any player is uniquely determined by the payoff function of one of the players and the automorphism group G_{Γ^*} of the game.

An important class of homogeneous N-person games is that of the symmetric games Σ^*. A homogeneous N-person game is said to be symmetric if its automorphism group G_{Σ^*} is the symmetric group of permutations of the suffixes $0, 1, \ldots, N-1$. If we can number the players' strategies in the symmetric N-person game in such a way that the lth strategy of player T^i is transformed into the lth strategy of player T^{gi} $(i=0, 1, \ldots, N-1;\ g \in G_{\Sigma^*})$, then the payoff function of the jth player will be

$$V^j(f)=V(f_{i_j}, v_1, v_2, \ldots, v_k)=V_{i_j}(v_1, \ldots, v_k), \tag{3}$$

where $v_s=N_s/N$ is the fraction of players who have chosen strategy number s $(s=1, \ldots, k)$ in play f (N_s is the number of players choosing strategy s in play f). Thus the symmetric game is given by the k payoff functions $V_i(v_1, \ldots, v_k)$, independently of the number of players taking part in the game.

We shall now return to the general definition of the N-person game Γ^*. Suppose the players know the payoff functions $V^j(f)$ and that no collusion or formation of coalitions is possible between the players. Suppose that by analysis of the game, using any method of computation, the players have chosen a set of pure strategies $f_{eq}=(f^0_{i_0}, \ldots, f^{N-1}_{i_{N-1}})$. The players will have no reason to alter their strategies if each of them is convinced that his gain cannot be increased provided the rest of the players preserve their strategies. Obviously if collusion were possible among the players, then a combined simultaneous change of strategies by two or more players could increase the gain of each of them, but we have noted above already that collusion is impossible. If the set f_{eq} is chosen so that the above considerations are valid for all players, then f_{eq} is in equilibrium. If it is not advantageous to player T^j to change his strategy alone, then this means that

$$V^j(f^0_{i_0}, \ldots, f^j_{i_j}, \ldots, f^{N-1}_{i_{N-1}}) \geqslant V^j(f^0_{i_0}, \ldots, f^j_{\alpha}, \ldots, f^{N-1}_{i_{N-1}}) \qquad (4)$$

for all $\alpha \neq i_j$.

If (4) holds for any j, then the play f_{eq} is called the pure strategy equilibrium point (Nash point). In the general case of an N-person game, the pure strategy Nash point (Nash play) may not exist.

We shall say that player T^j uses the mixed strategy $\mu^j = (\mu^j_1, \ldots, \mu^j_k)$ if he uses his pure strategy f^j_i with probability $\mu^j_i (i = 1, \ldots, k, \sum_{i=1}^{k} \mu^j_i = 1)$. Nash's basic theorem states that each finite N-person game has at least one equilibrium situation in mixed strategies $(\mu^0_{eq}, \ldots, \mu^{N-1}_{eq})$, that is

$$\overline{V}^j(\mu^0_{eq}, \ldots, \mu^j_{eq}, \ldots, \mu^{N-1}_{eq}) \geqslant \overline{V}^j(\mu^0_{eq}, \ldots, \mu^{j'}, \ldots, \mu^{N-1}_{eq}) \qquad (5)$$

for all $\mu^j \neq \mu^j_{eq}$ and $j = 0, \ldots, N-1$, where

$$\overline{V}^j(\mu^0, \ldots, \mu^{N-1}) = \sum_{i_0, \ldots, i_{N-1}} \mu^0_{i_0} \mu^1_{i_1} \ldots \mu^{N-1}_{i_{N-1}} V^j(f^0_{i_0}, f^1_{i_1}, \ldots, f^{N-1}_{i_{N-1}}) \qquad (6)$$

is the mathematical expectation of the gain of player T^j when the set of mixed strategies $(\mu^0, \ldots, \mu^{N-1})$ is used.

In homogeneous N-person games it follows from their definition that an automorphism $g \in G_{\Gamma^*}$ must preserve the Nash equilibrium in mixed strategies.

In symmetric games there exist symmetric equilibrium strategies, that is there exist

$$\mu^0_{eq} = \mu^1_{eq} = \ldots = \mu^{N-1}_{eq} = \mu_{eq}.$$

A homogeneous game possessing a Nash play in pure strategies will be called a Nash game, and the invariant set of plays generated by the Nash plays will be called the Nash set.

If a Nash play is a play of maximal value it will be called a Moore play. Games having Moore plays will be called Moore games and the corresponding invariant sets of plays called Moore sets.

If in a symmetric N-person game Σ^* the gain of each player T^j in a play f does not depend on the strategy chosen by T^j but only on the distribution of strategies among the players, that is

$$V_j(v_1, \ldots, v_k) = V(v_1, \ldots, v_k) \qquad (j = 1, \ldots, k), \qquad (7)$$

then such a symmetric game will be called a Goore game. Obviously any Goore game is a Moore game.

If the players can enter into coalitions, then two players T^i and T^j may simultaneously change their strategies. For two players in coalition, a combined change of strategies is only profitable if the increase in the gain of one player is greater than the decrease in the gain of the other player. In this case the player whose gain has increased must compensate the other player for the decrease in his gain. Thus in a coalition the players must also organize a system for settling their mutual account. In homogeneous games the players can achieve maximum gain by all simultaneously forming a coalition. If the players then agree to play in turn all the plays of greatest value, the average gain of each player will be equal to the value of the Moore play. A similar effect can be obtained by arranging a 'common fund' procedure. This means

that after each play of the game all gains are pooled and divided equally between the players. Thus the gains of the players in a play of such a game are equal to

$$V(f)=\frac{1}{N}\sum_{j=0}^{N-1} V^j(f). \tag{8}$$

Obviously the game Γ_0^* with a common fund, constructed from the game Γ^*, is a Moore game.

Introducing a common fund procedure into a symmetric N-person game Σ^* converts it into a Goore game G^*.

The above definitions of classes of games can be extended to games (Γ) of automata with independent outcomes. In this case the payoff functions $a^j(f)$ are the mathematical expectations of the gain of automaton A^j in the play f. As for any games with independent outcomes, an equivalent automata game Σ or G can be constructed for any symmetric game Σ^* or Goore game G^*.

For ergodic automata games there exist final probability distributions for plays. If $\sigma(f)$ is the final probability of play f in the ergodic automata game Γ, then $\sigma(gf)=\sigma(f)$ for homogeneous games and the mathematical expectations of the gains of all the automata in a play f are the same in a homogeneous automata game.

In what follows we shall use the following 'slowness hypothesis'.

In the N-automata game Γ with automata A_n^j belonging to an asymptotically optimal sequence such that all the payoff functions $a^j(f)$ belong to an asymptotically optimal domain, a change in the plays of the game as $n\to\infty$ occurs so rarely that a stationary probability distribution of automata states is established. Thus each automaton A_n^j can be replaced by a stochastic automaton $\tilde{A}_n^j$ which changes its action with probability $\gamma_n^j(a^j(f))$ and preserves its action with probability $1-\gamma_n^j(a^j(f))$, where $\gamma_n^j(a^j(f))$ is the stationary probability of change of action by automaton A_n^j under an invariant mathematical expectation of gain equal to $a^j(f)$.

The possibility of replacing the automata A_n^j by the automata $\tilde{A}_n^j$ means that the game $\Gamma(A_n^1,\ldots,A_n^N)$ and the game $\tilde{\Gamma}(\tilde{A}_n^1,\ldots,\tilde{A}_n^N)$ have the same final probabilities of a play f.

An approximate analysis of the game $\tilde{\Gamma}(\tilde{A}_n^1,\ldots,\tilde{A}_n^N)$ shows that the automata in this game almost exclusively play the plays f^0 which satisfy

$$\min_m a_m(f^0)=\max_f \min_m a_m(f). \tag{9}$$

If a Nash play exists in the game, then it is a stationary play for the automata game. However, the outcome of the Nash play is, in general, ensured by an infinitely large memory and in an infinitely long time. On the other hand, the Nash play is not, as a rule, a play of the highest value, that is, the Nash play does not coincide with the Moore play. We shall consider an example of a homogeneous game which clearly displays effects connected

with the dependence of the behaviour of the automata on their memory capacity.

In the distribution game, N automata $A^1, \ldots, A^N$ take part, each of them having k actions (strategies). The game is given by k functions $-v_m \leqslant \alpha_m(v_m) \leqslant v_m$ $(0 \leqslant v_m \leqslant 1, m=1, \ldots, k)$, where v_m is the fraction of the automata which have chosen strategy number m in the play f. The functions $\alpha_m(v_m)$ are called the power functions of the strategies.

The mathematical expectation of the gain of automaton A^j which has chosen strategy number m in the play f is equal to

$$M(A^j, f_m, f)=a_m(v_m)=\frac{\alpha_m(v_m)}{v_m}. \tag{10}$$

The situation modelled by the distribution game is close to the problem of distribution of resources. However, if in the classical statement of the problem of resource distribution it is natural to speak of the collective behaviour of the resource users, the distribution game may be interpreted as the problem of the 'collective behaviour of resources'. For example, the distribution game closely models the situation of man-power distribution where there is a free choice of place of work and the wages in each undertaking depend on the number of men working there. A number of other appropriate interpretations of this game can also be formulated.

For simplicity we shall consider the case when the function $\alpha_m(v_m)$ are concave, that is

$$\frac{d^2\alpha_m(v_m)}{dv_m^2}<0. \tag{11}$$

This assumption is usual in problems such as the distribution of resources, as it ensures the uniqueness of the equilibrium point. Essentially condition (11) means that as the number of automata choosing strategy f_m increases, the value of the power function of this strategy also increases while the mathematical expectation of the gain of each automaton decreases. In fact it follows from the concavity condition that

$$\frac{d\alpha_m(v_m)}{dv_m}<\frac{\alpha_m(v_m)}{v_m} \tag{12}$$

and therefore

$$\frac{d}{dv_m}\left(\frac{\alpha_m(v_m)}{v_m}\right)=\frac{\dfrac{d\alpha_m(v_m)}{dv_m}-\dfrac{\alpha_m(v_m)}{v_m}}{vm}<0,$$

that is, $a_m(v_m)$ is a decreasing function of v_m so that the game has a unique equilibrium situation (Nash point).

We shall say that strategy f_j dominates strategy f_i if $a_j(v_j) \geqslant a_i(v_i)$ for all values of v_j and v_i. In this case strategy f_j is said to be dominant and f_i recessive. It is easy to see that in the Nash equilibrium situation the automata

do not use recessive strategies. Later we shall consider cases where no recessive strategies exist in the game.

If condition (11) is satisfied and there are no recessive strategies and hence no relation of dominance, then the distribution of strategies among the automata in the Nash play is determined by the solution of the system of equations

$$\begin{aligned} &\frac{\alpha_m(v_m^{eq})}{v_m^{eq}} = a^{eq} \qquad (m=1,\ldots,k), \\ &\sum_{m=1}^{k} v_m^{eq} - 1 = 0. \end{aligned} \tag{14}$$

Note that in the distribution game the value of the play f is

$$M(f) = \sum_{m=1}^{k} v_m a_m(v_m) = \sum_{m=1}^{k} \alpha_m(v_m),$$

and the value of the Nash play is

$$M^{eq} = \sum_{m=1}^{k} \alpha_m(v_m^{eq}) = \sum_{m=1}^{k} v_m^{eq} a^{eq} = a^{eq}. \tag{15}$$

If a^{eq} belongs to an asymptotically optimal domain for the automata taking part in the game, then for sufficiently large n the automata play almost exclusively the plays in the Nash invariant set. In studying the behaviour of automata we shall be interested in the dependence of the equilibrium situation (stationary distribution of plays among the automata) upon the size of the memory of the automata.

Let $n=1$; then the probability of a change in the action of the automaton is the same as the probability of losing. The mathematical expectation of a change in the fraction of automata performing action f_m is equal to

$$\overline{\Delta v_m} = \frac{1}{k} \sum_{i=1}^{k} v_i \frac{1-a_i(v_i)}{2} - v_m \frac{1-a_m(v_m)}{2}. \tag{16}$$

By the law of large numbers, as $N \to \infty$, the actual change in the fraction of automata performing action f_m tends to the mathematical expectation. Hence, as $N \to \infty$, we speak of a situation of dynamic equilibrium in which $\overline{\Delta v_m} = 0$ $(n=1, m=1,\ldots,k)$. If $N \to \infty$ and $n=1$, we shall call the invariant set of dynamic equilibrium plays the Antos invariant set, and the corresponding plays Antos plays. An Antos play is stable if $\overline{\Delta v_m} = \operatorname{sgn}(v_m^{(A)} - v_m)$ for all m and $\min_{v_m} (\overline{\Delta v_m})^2 = (\Delta v_m^{(A)})^2$.

From (16) we get a system of equations which determine the distribution of strategies among the automata in the Antos play:

$$\begin{aligned} &v_m^{(A)} - \alpha_m(v_m^{(A)}) = \alpha^{(A)} \qquad (m=1,\ldots,k), \\ &\sum_{m=1}^{k} v_m^{(A)} - 1 = 0. \end{aligned} \tag{17}$$

Summing the first k equations of (17), we find that the value of the Antos play is

$$M^A = \sum_{m=1}^{k} \alpha_m(v_m^{(A)}) = 1 - k\alpha^{(A)}. \tag{18}$$

Consider the case $k=2$. Then at the Nash point

$$\alpha_1(v_1^{eq})=v_1^{eq}a^{eq} \quad \text{and} \quad \alpha_2(1-v_1^{eq})=(1-v_1^{eq})a^{eq},$$

and at the Antos point

$$\alpha_1(v_1^{(A)})=v_1^{(A)}-\alpha^{(A)} \quad \text{and} \quad \alpha_2(1-v_1^{(A)})=1-v_1^{(A)}-\alpha^{(A)}.$$

If $v_1^{eq}<v_1^{(A)}$, then because of the concavity of $\alpha_1(v_1)$

and $\alpha_2(1-v_1)$, $v_1^{(A)}-\alpha^{(A)}<v_1^{(A)}a^{eq}$

and $(1-v_1^{(A)})-\alpha^{(A)}>(1-v_1^{(A)})a^{eq}$,

or $v_1^{(A)}<\alpha^{(A)}/(1-a^{eq})$

and $1-v_1^{(A)}>\alpha^{(A)}/(1-a^{eq})$,

and hence

$$v_1^{eq}<v_1^{(A)}<\tfrac{1}{2}.$$

Similarly, if $v_1^{eq}>v_1^{(A)}$, then $v_1^{eq}>v_1^{(A)}>\frac{1}{2}$. Thus for $k=2$ the Antos point lies on the v_1-axis between the Nash point and the point $v_1=\frac{1}{2}$.

It can be shown similarly that for $k>2$:

$$v_m^{eq}>v_m^{(A)}>1/k \quad \text{or} \quad v_m^{eq}<v_m^{(A)}<1/k,$$

that is, for each strategy the fraction of automata choosing it in the Antos play lies between the fraction of automata choosing it in the Nash play and the number $1/k$.

In a game with a common fund, the total gain of all the automata is divided equally among those taking part in the game and depends only on the distribution of strategies among the automata. Here the mathematical expectation of the gain by all the automata in a play f is equal to $\sum_{m=1}^{k}\alpha_m(v_m)$ and does not depend on the automaton's choice of strategy. Since $\alpha_m(v_m)$ is convex, the function $\sum_{m=1}^{k}\alpha_m(v_m)$ is also concave with respect to each variable v_m. Then by the fundamental theorem on the equilibrium situation in convex games, the game has a unique equilibrium situation (Moore point). The distribution of strategies among the automata in the Moore play is determined by the system of equations

$$\begin{aligned} &\frac{d\alpha_m(v_m)}{dv_m}=\alpha^{(M)} \qquad (m=1, 2, \ldots, k), \\ &\sum_{m=1}^{k} v_m-1=0. \end{aligned} \tag{20}$$

Note that (20) is the condition for a saddle-point of the Lagrange function $Q=\sum_{m=1}^{k}\alpha_m(v_m)-\alpha^{(M)}(\sum_{m=1}^{k}v_m-1)$, which provides $\max\sum_{m=1}^{k}\alpha_m(v_m)$ under the restriction $\sum_{m=1}^{k}v_m-1=0$.

The behaviour of the mean gain of the automata in terms of n is given by the position of the Moore point. For simplicity we shall consider the case $k=2$. The way in which the average gain varies with n for various positions

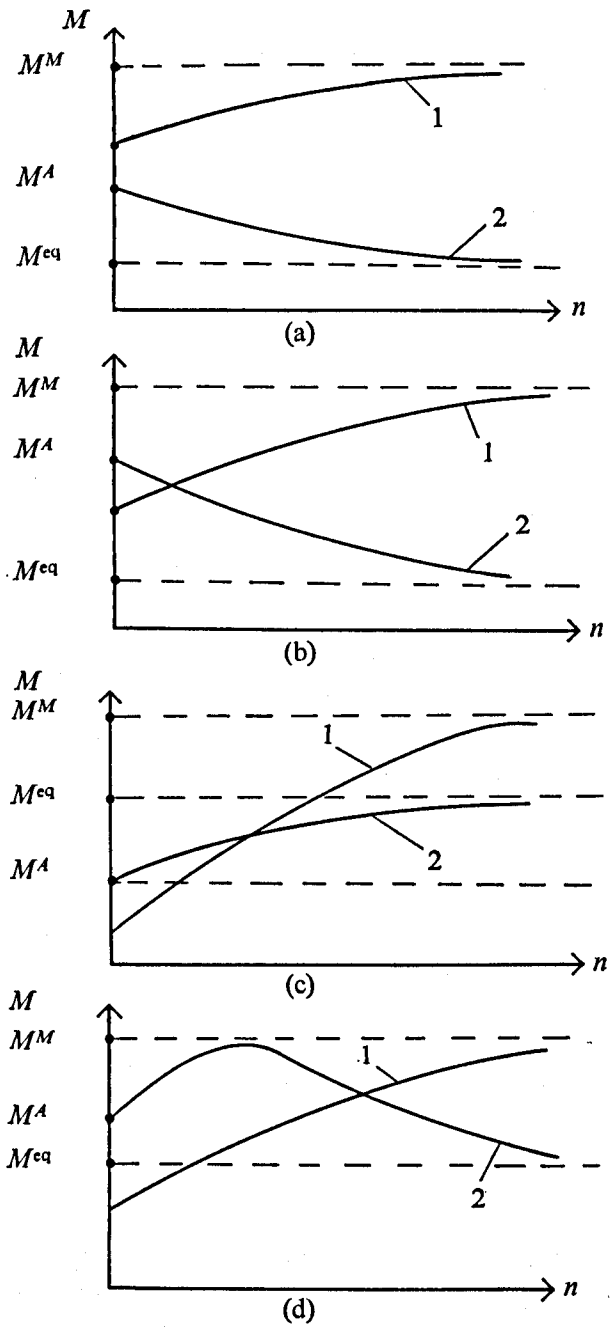


Figure 1

of the equilibrium points is shown in figure 1 (for $k=2$, $y_1=y=1-y_2$). Curve 1 is the game with a common fund, curve 2 the game without a common fund.

$y^N>y^A>1/k>y^M$ figure 1(a),
$y^N>y^A>y^M>1/k$ figure 1(a) or 1(b),
$y^M>y^N>y^A>1/k$ figure 1(c),
$y^N>y^M>y^A>1/k$ figure 1(d).

This suggests the following conclusions:

(a) the increase in complexity in the local means of control (increase of averaging time) with decentralized behaviour may lead to a decrease in average gain (figures 1(a) and 1(b));

(b) the use by the subsystems of a global criterion of the set as a local criterion may be advantageous only if there are sufficiently complex local means of decision-making ($n>n_1$, figures 1(b), 1(c), 1(d));

(c) the local utility functions may be constructed in such a way that the set of subsystems will reach a global extremum in a finite averaging time, that is with sufficiently simple local means of decision-making ($n=n_0$, figure 1(d)); the latter is particularly important if the functions $F_j(y_j)$ vary in time.

The following example of collective behaviour of automata is related to problems in the behaviour of interacting automata such as the 'French flag' problem, the firing squad synchronization problem, and so on. A characteristic feature of these problems is that a combination of locally simple automata can solve problems which are in principle beyond the capability of any one of them separately. For example, in the solution of the firing squad synchronization problem an automaton with eight internal states delays the propagating signal for a time $2^{m+1}-1$, which is in principle impossible for an isolated automaton for $m>2$. In an example considered below we shall try to show that there are very great possibilities in collections of comparatively simple automata. For this purpose we turn to cell models of the growth of figures.

Suppose we have a linear cell space; this means that we have an infinite chain of automata which are in a passive state. Each automaton interacts with its neighbours, that is, the numbers of the internal states of its right and left neighbours act as inputs to the automata. If an automaton and its two neighbours are in the passive state, then they remain like this, and a chain all of whose automata are passive will remain passive indefinitely. If an external signal puts one of the automata into an active state then it begins to act on its neighbours. Thus a sequence of automata in active states may arise in the chain. Such a connected sequence will be called a configuration. It is easy to see that one activated automaton may 'grow' a configuration of infinite length. Here we meet the problem of stopping the growth process in the following form: what is the maximum finite length of a configuration which can be grown from one activated automaton in a linear cell space of identical automata having n internal states?

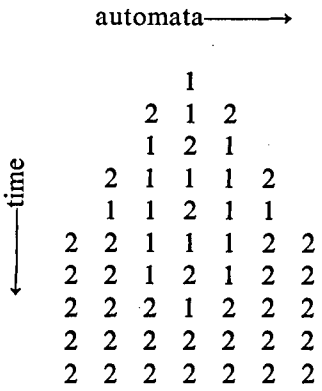


Figure 2

We begin with an example. Let the automaton have three internal states. A diagram of successive phases of growth is given as figure 2. By completely enumerating all possible tables of transition rules* it has been shown that for $n=3$, the maximal length $L(3)=7$. For $n=4$ transition rules have been found giving $L(n)=45$ but this length has not been shown to be maximal. We now turn to considering a universal procedure which will ensure that the process stops for large n.

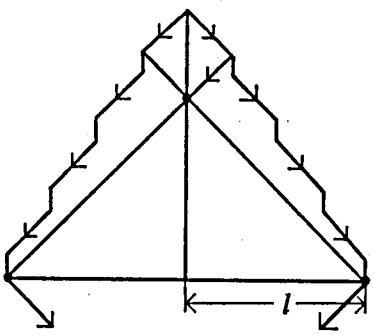


Figure 3

Evidently if infinite growth is to be avoided there must be at least one state which does not activate the passive states of its neighbours. Consider figure 3, which represents a possible picture of the propagation of signals in the chain. The growth front moves along the chain with speed $(k-1)/k$ automata per unit time (where k is the number of states used in forming the front). After the first cycle of motion of the front, signals are sent from the fronts inside the configuration with speed 1 (1 automaton per unit time) which, after overtaking the opposite fronts, cut off the growth process. It is easy to

* We assume here that an automaton which has been activated cannot revert to the passive state, that is, no 'break' in the configuration is allowed during the growth process.

estimate the length of the resulting configuration. In time t from the moment when the stopping signal is generated, the front travels a distance $l-k$ and the stopping signal a distance $l+k$. Hence $t=l+(k-1)=\{[l-(k-1)]k\}/(k-1)$. Taking symmetry and the presence of a central automaton into account, we finally have $L=2(k-1)(2k-1)+1$.

If after the first phase we again generate two new stopping signals while the front continues to travel, a similar reasoning gives

$$L=2(k-1)(2k-1)^2+1.$$

Since each time it is necessary to introduce two new states for a new cycle, we get

$$L(n)=a(k)(2k-1)^{n/2}+1,$$

where

$$a(k)=\frac{2(k-1)}{\frac{1}{2}(2k-1)(k+1)}.$$

This estimate can be substantially improved if the states used as overtaking signals are also used for the formation of the front. We can thus easily obtain a growth of order

$$L(n)\sim a(\tfrac{1}{2}n)^{\frac{1}{2}n}.$$

This estimate, too, can be improved by using a number of tricks. However, we shall consider a modification in the principle of this stopping process. The idea of the second method is shown in figure 4.

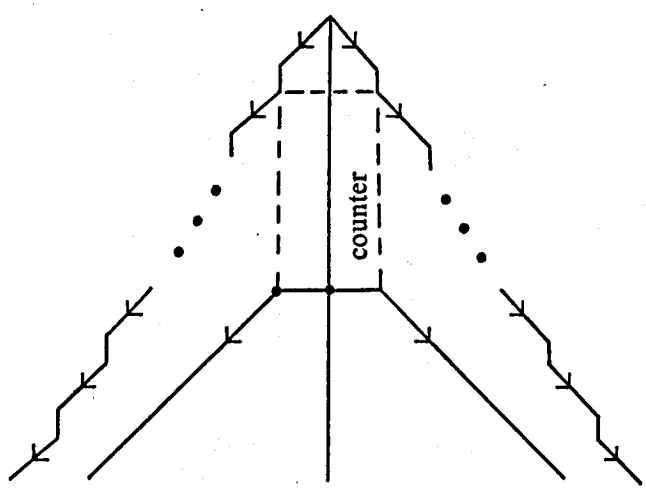


Figure 4

After the first cycle of propagation of the front inside the configuration, a counting device is set up and the overtaking signal starts after time t_c equal to the cycle of operation of the counter. During time t_c the front will travel a distance

$$l_1=\frac{t_c(k-1)}{k}.$$

Forming the balance equation, we have

$$\frac{(l-l_1)k}{k-1}=l,$$

and hence $l=(k-1)t_c'$.

We now consider how to organize the work of the counter. For this four states are required. An example is shown in figure 5. The precise time of operation of the counter is given by a cumbersome formula, so we shall give an approximate estimate. As shown in figure 5, the first stage of the counter counts modulo 4 while the other stages are not worse than modulo 2. Therefore the time of operation of the counter until the overtaking signal appears in the central automaton is

$$t_c'>4.2^{r-1},$$

where r is the number of stages in the counter. The overtaking signal goes past the counter in time

$$t_c''>4.2^{r-1}.$$

Hence the total delay in the counter is $t_c>2^{r+2}$, so that

$$l>(k-1)2^{k+1}$$

In the front, all the states except the passive state are used, so that $k=n-1$ and $L_1>(n-2)2^{n+1}$.

In addition, after the first overtaking signal meets the front a second counter can be set up and a second overtaking signal formed. To do this, any state may be used as the overtaking signal with the exception of the passive state, the first state, the four states used in the operation of the counter, and the signal used as the first overtaking signal. Note that in general the above process can only be carried out for $n\geqslant 7$, and that the number of possible counters is $n-6$.

For the second cycle of formation of the overtaking signal, the length of the counter is $r_2>(n-2)2^n$, and hence

$$t_{c_2}\gg 2^{(n-2)2^n}$$

and $$L_2\gg(n-2)2^{(n-2)2^n}.$$

It follows that

$$L_n\gg(n-2)a_1^{a_2^{\cdot^{\cdot^{\cdot^{a_{n-7}^{2^n}}}}}},$$

where $a_1=a_2=\ldots=a_{n-7}=2^{n-2}$.

This estimate seems rather unexpected. However, it enables us to draw a number of conclusions. Note that there exist only n^{n^3} different transition functions for an automaton having n states and two neighbours. The constructions of automata determined by such transition functions include equivalent constructions and constructions generating configurations of unbounded size. Hence the number of distinct finite lengths which can be grown in a linear cell space of automata with n states cannot exceed n^{n^3}.

automata ⟶

time ↓

1	1	1	5	1	1	1
2	1	1	5	1	1	2
3	1	1	5	1	1	3
4	1	1	5	1	1	4
1	2	1	5	1	2	1
2	2	1	5	1	2	2
3	2	1	5	1	2	3
4	2	1	5	1	2	4
1	3	1	5	1	3	1
2	3	1	5	1	3	2
3	3	1	5	1	3	3
4	3	1	5	1	3	4
1	4	1	5	1	4	1
2	1	2	5	2	1	2
3	1	2	5	2	1	3
4	1	2	5	2	1	4
1	2	2	5	2	2	1
3	2	2	5	2	2	3
4	2	2	5	2	2	4
1	3	2	5	2	3	1
2	3	2	5	2	3	2
3	3	2	5	2	3	3
4	3	2	5	2	3	4
1	4	2	5	2	4	1
2	1	3	5	3	1	2
3	1	3	5	3	1	3
4	1	3	5	3	1	4
1	2	3	5	3	2	1
2	2	3	5	3	2	2
3	2	3	5	3	2	3
4	2	3	5	3	2	4
1	3	3	5	3	3	1
2	3	3	5	3	3	2
3	3	3	5	3	3	3
4	3	3	5	3	3	4
1	4	3	5	3	4	1
2	1	4	5	4	1	2
3	2	5	6	5	2	3
4	2	5	6	5	2	4
1	3	5	6	5	3	1
2	3	5	6	5	3	2
3	3	5	6	5	3	3
4	3	5	6	5	3	4
1	4	5	6	5	4	1
2	5	1	6	1	5	2
3	5	1	6	1	5	3
4	5	1	6	1	5	4
5	1	1	6	1	1	5

Figure 5

This obviously suggests the conclusion that there are 'convenient' and 'inconvenient' sizes. The 'convenient' sizes need comparatively small amounts of local structure information for their realization, while there are also sizes for which the information must be essentially large.

We also point out that realizations of great length can be easily obtained by using the so-called 'wave' algorithms, that is, algorithms based on the interaction of the fronts propagating the signals. It is characteristic that similar algorithms are used for the solution of a wide range of problems in cell structures, such as the French flag problem, the synchronization of chains of automata and the voting problem. It is possible that such 'wave' processes are natural in the organization of behaviour in cell structures.

Acknowledgements

I would like to take this opportunity of expressing my sincere thanks to V.B. Marakhovsky and V.A.Peschansky, who collaborated in the results obtained in problem of stopping growth, for their permission to use these results here. I am also grateful to the participants in the annual extended seminar of the CEMI (LB) cybernetic laboratory for useful discussion and benevolent criticism, and wish particularly to thank Professor Donald Michie for his constant interest in my work and for his courteous invitation to present a report at this symposium.

REFERENCES

Tsetlin, M.L. (1963) Finite automata and models of the simplest forms of behaviour (Russian), *Uspekhi Mat. Nauk*, **18**, No. 4, 3–28. Translation in *Russian Math. Surveys*, **18**, 4 (1963), 1–25.

Tsetlin, M.L. (1969) *Issledovaniya po teorii avtomatov i modelirovaniyu biologicheskikh sistem* (Investigations in automata theory and models of biological systems), Moscow: Nauka.

Tsetlin, M.L. & Varshavsky, V.I. (1967) Collective automata and models of behaviour. *Samonastraivayushchiesya sistemy. Raspoznavanie obrazov. Releinye ustroistva i konechnye avtomaty* (Self-regulating systems. Pattern recognition. Relay apparatus and finite automata), Moscow: Nauka.

Tsetlin, M.L., & Varshavsky, V.I. (1966) Automata and models of collective behaviour. *Proceedings of the Third Congress of IFAC*, paper 35E, London.

Varshavsky, V. I. (1968) Collective behaviour and control problems. *Machine*

Corrigenda arising from late receipt of author's proofs

Page 394, last equation: for *vm* read v_m

Page 395, line 9 from foot should read:

An Antos play is stable if sgn $\overline{\Delta v_m} = \text{sgn}\,(v_m^{(A)} - v_m)$

Page 402, figure 5: between rows 17 and 18 insert

2 2 2 5 2 2 2

automata ⟶

time ⟵

1	1	1	5	1	1	1
2	1	1	5	1	1	2
3	1	1	5	1	1	3
4	1	1	5	1	1	4
1	2	1	5	1	2	1
2	2	1	5	1	2	2
3	2	1	5	1	2	3
4	2	1	5	1	2	4
1	3	1	5	1	3	1
2	3	1	5	1	3	2
3	3	1	5	1	3	3
4	3	1	5	1	3	4
1	4	1	5	1	4	1
2	1	2	5	2	1	2
3	1	2	5	2	1	3
4	1	2	5	2	1	4
1	2	2	5	2	2	1
3	2	2	5	2	2	3
4	2	2	5	2	2	4
1	3	2	5	2	3	1
2	3	2	5	2	3	2
3	3	2	5	2	3	3
4	3	2	5	2	3	4
1	4	2	5	2	4	1
2	1	3	5	3	1	2
3	1	3	5	3	1	3
4	1	3	5	3	1	4
1	2	3	5	3	2	1
2	2	3	5	3	2	2
3	2	3	5	3	2	3
4	2	3	5	3	2	4
1	3	3	5	3	3	1
2	3	3	5	3	3	2
3	3	3	5	3	3	3
4	3	3	5	3	3	4
1	4	3	5	3	4	1

This obviously suggests the conclusion that there are 'convenient' and 'inconvenient' sizes. The 'convenient' sizes need comparatively small amounts of local structure information for their realization, while there are also sizes for which the information must be essentially large.

We also point out that realizations of great length can be easily obtained by using the so-called 'wave' algorithms, that is, algorithms based on the interaction of the fronts propagating the signals. It is characteristic that similar algorithms are used for the solution of a wide range of problems in cell structures, such as the French flag problem, the synchronization of chains of automata and the voting problem. It is possible that such 'wave' processes are natural in the organization of behaviour in cell structures.

Acknowledgements

I would like to take this opportunity of expressing my sincere thanks to V.B. Marakhovsky and V.A.Peschansky, who collaborated in the results obtained in problem of stopping growth, for their permission to use these results here. I am also grateful to the participants in the annual extended seminar of the CEMI (LB) cybernetic laboratory for useful discussion and benevolent criticism, and wish particularly to thank Professor Donald Michie for his constant interest in my work and for his courteous invitation to present a report at this symposium.

REFERENCES

Tsetlin, M.L. (1963) Finite automata and models of the simplest forms of behaviour (Russian), *Uspekhi Mat. Nauk*, **18**, No. 4, 3–28. Translation in *Russian Math. Surveys*, **18**, 4 (1963), 1–25.

Tsetlin, M.L. (1969) *Issledovaniya po teorii avtomatov i modelirovaniyu biologicheskikh sistem* (Investigations in automata theory and models of biological systems), Moscow: Nauka.

Tsetlin, M.L. & Varshavsky, V.I. (1967) Collective automata and models of behaviour. *Samonastraivayushchiesya sistemy. Raspoznavanie obrazov. Releinye ustroistva i konechnye avtomaty* (Self-regulating systems. Pattern recognition. Relay apparatus and finite automata), Moscow: Nauka.

Tsetlin, M.L., & Varshavsky, V.I. (1966) Automata and models of collective behaviour. *Proceedings of the Third Congress of IFAC*, paper 35E, London.

Varshavsky, V. I. (1968) Collective behaviour and control problems. *Machine Intelligence 3*, pp. 217–42 (ed. Michie, D.). Edinburgh: Edinburgh University Press.

Varshavsky, V.I. (1969) The organization of interaction in collectives of automata. *Machine Intelligence 4*, pp. 285–311 (eds Meltzer, B. & Michie, D.). Edinburgh: Edinburgh University Press.

Varshavsky, V.I. (1972) Automata games and control problems. *Proceedings of the Fifth Congress of IFAC*, Paris.

23

Some New Directions in Robot Problem Solving

R. E. Fikes, P. E. Hart and N. J. Nilsson

Artificial Intelligence Center
Stanford Research Institute

Abstract

For the past several years research on robot problem-solving methods has centered on what may one day be called 'simple' plans: linear sequences of actions to be performed by single robots to achieve single goals in static environments. Recent speculation and preliminary work at several research centers has suggested a variety of ways in which these traditional constraints could be relaxed. In this paper we describe some of these possible extensions, illustrating the discussion where possible with examples taken from the current Stanford Research Institute robot system.

1. INTRODUCTION

Background

A major theme in current artificial intelligence research is the design and construction of programs that perform robot problem solving. The usual formulation begins with the assumption of a physical device like a mechanical arm or a vehicle that can use any of a preprogrammed set of actions to manipulate objects in its environment. The generic task of the robot problem solver is to compose a sequence of these actions (or *operators*) that transforms the initial state of the environment (or problem domain) into a final state in which some specified goal condition is true. Such a sequence is called a *plan* for achieving the specified goal.

Most previous work in robot problem solving – for example, the work of Green (1969), Fikes and Nilsson (1971), and Winograd (1971) – has been limited at the outset by the assumption of a certain set of simplifying ground rules. Typically, the problem environment is a dull sort of place in which a single robot is the only agent of change – even time stands still until the robot moves. The robot itself is easily confused; it cannot be given a second problem until it finishes solving the first, even though the two problems may

be related in some intimate way. Finally, most robot systems cannot yet generate plans containing explicit conditional statements or loops.

In this paper we wish to consider some possibilities for relaxing these ground rules. Our suggestions are of course tentative, and perhaps not even entirely original. We hope, nevertheless, that they will illuminate some of the issues by serving as thought-provoking examples of what might be done next in this interesting research area. We specifically exclude from our discussion comments about present and future trends in robot perception (vision, range-finding, and the like), except to make the obvious remark that advances in perceptual abilities will ease the problem-solving burden and vice versa.

As an aside to the reader, we admit to having difficulty in discussing our ideas in an abstract setting; we find we understand the ideas ourselves only when we see examples. Accordingly, we elected to couch our suggestions in the language and framework of a particular robot problem solver, STRIPS (Fikes and Nilsson 1971, Fikes, Hart and Nilsson, in press), that has been under development at Stanford Research Institute. Our discussion, therefore, takes on the tinge of being a critique of this particular system, but we hope that some of the ideas have a more general interpretation. To provide the necessary background for the reader unfamiliar with the STRIPS system, we will try to give in the next few paragraphs as brief and painless a summary as possible.

A summary of STRIPS

A problem environment is defined to STRIPS (STanford Research Institute Problem Solver) by giving two different kinds of information: a model consisting of a set of statements describing the initial environment, and a set of operators for manipulating this environment. An operator is characterized by a precondition statement describing the conditions under which it may be applied, and lists of statements describing its effects. Specifically, the effect of an operator is to remove from the model all statements matching forms on the operator's 'delete list', and to add to the model all statements on the operator's 'add list'. All statements are given to STRIPS in the predicate-calculus language.

Once STRIPS has been given an initial model and a set of operators, it may be given a problem in the form of a goal statement. The task for STRIPS is to find a sequence of operators transforming the initial model, or state, into a final state in which the goal is provable. STRIPS begins operation by attempting to prove the goal from the initial model. If the proof cannot be completed, STRIPS extracts a 'difference' between the initial model and the goal indicating a set of statements that would help to complete the proof. It then looks for a 'relevant operator' that will add to the model some of the statements in the difference. (For example, a difference may include a desired robot location; this difference would be reduced by a GOTO operator.) Once a relevant operator has been selected, its preconditions constitute a subgoal

to be achieved, and the same problem-solving process can be applied to it. When a precondition subgoal is provable in the state under consideration, then the subgoal has been achieved and the operator's effects description is used to produce a new state. This process of forming new subgoals and new states continues until a state is produced in which the original goal is provable; the sequence of operators producing that state is the desired solution.

After STRIPS produces a solution to the specific problem at hand, the solution is *generalized* and stored in a special format called a *triangle table.* The generalization process replaces specific constants in the solution by parameters so that the plan found by STRIPS will be applicable to a whole family of tasks including the special task of the moment. The generalized plan can then be used as a component macro-action in future plans. Such a macro-action is called a MACROP and is also used to monitor the execution of a plan. Roughly, our execution monitor, PLANEX, has the useful ability to find an appropriate new instance of a general MACROP if it finds that the instance being executed fails to work for some reason.

Details of our system for learning and executing plans will be published (Fikes, Hart and Nilsson, in press); we shall give a brief explanation of the triangle table format here, since part of the discussion to be presented in Section 5 depends on it. (The reader may want to defer reading the rest of this section until later.) An example of a triangle table is shown in figure 1 for the case of a plan with three component steps. The cells immediately below each operator contain the statements added by that operator; we have used the notation A_i to represent these add lists. We retain in the cells below the top cell in a column just those statements that are not deleted by later operators. For example, $A_{1/2}$ in cell (3, 2) represents the statements of A_1 that remain in the model after the application of operator OP_2; similarly, $A_{1/2,3}$ represents the statements in $A_{1/2}$ that survive in the model after the application of OP_3.

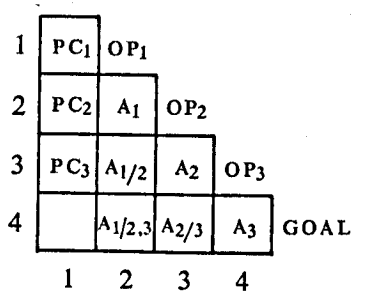


Figure 1. An example of a triangle table.

The left-most column of the triangle table contains certain statements from that model existing before any of the operators were applied. STRIPS, we recall, must always prove the preconditions of an operator from any given model before the operator can be applied to that model. In general, the model statements used to establish the proof arise from one of two sources:

either they were in the initial model and survived to the given model, or they were added by some previous operator. Statements surviving from the initial model are listed in the left-most column, while statements added by previous operators are among those listed in the other columns. By way of example, consider OP_3. In general, some of the statements used to support the proof of its preconditions were added by OP_2, and are therefore part of A_2; some of the statements were added by OP_1 and survived the application of OP_2, and are therefore included in $A_{1/2}$; finally, some of the statements existed in the initial model and survived the application of OP_1 and OP_2, and these statements comprise PC_3.

The triangle table format is useful because it shows explicitly the structure of the plan. Notice that all the statements needed to establish the preconditions of OP_i are contained in Row i. We will call such statements *marked* statements. By construction, all the statements in the left-most cell of a row are marked, while only some of the statements in the remainder of the row may be marked.

Obviously, an operator cannot be executed until all the marked statements in its row are true in the current state of the world. This alone is not a sufficient condition for execution of the rest of the operators in the plan, however. To point up the difficulty, suppose all the marked clauses in Row 2 are true, but suppose further that not all the marked statements in cell (3, 2) are true. We know that OP_2 can itself be executed, since its preconditions can be proven from currently true statements. But consider now the application of OP_3. Since the marked statements in cell (3, 2) are not true, it cannot be executed. Evidently, OP_1 also should have been executed.

The preceding example motivates an algorithm used by PLANEX. The algorithm rests on the notion of a *kernel* of a triangle table, the ith kernel being by definition the unique rectangular subtable that includes the bottom left-most cell and row i. In its simplest form, the PLANEX algorithm calls for executing OP_i whenever kernel i is the highest numbered kernel all of whose marked statements are true. The reader may want to verify for himself that this algorithm avoids the difficulty raised in the previous paragraph.

With this sketchy introduction to the STRIPS system, we can now proceed to more speculative matters of perhaps greater interest.

2. MULTIPLE GOALS

Multiple goals and urgencies

Useful applications of robots may require that the robot system work toward the achievement of several goals simultaneously. For example, a Martian robot may be performing some life detecting experiment as its main task and all the while be on the lookout for rock samples of a certain character. We should also include the possibility of 'negative goals': our Martian robot might be given a list of conditions that it must avoid, such as excessive battery drain and being too close to cliffs.

There are several ways in which to formulate task environments of this sort. In our work with STRIPS, a goal is given to the system in terms of a single predicate calculus wff (well-formed formula) to be made true. We might define multiple goals by a set of goal wffs, each possessing an 'urgency value' measuring its relative importance. Wffs having negative urgency values would describe states that should be avoided. Now we would also have to define precisely the overall objective of the system. In the case of a single goal wff, the objective is quite simple: achieve the goal (possibly while minimizing some combination of planning and execution cost). For an environment with multiple goals, defining the overall objective is not quite so straightforward, but quite probably would include maximizing some sort of benefit/cost ratio. In calculating the final 'benefit' of some particular plan one would have to decide how to score the urgencies of any goal wffs satisfied by the intermediate and final states traversed by the plan. In any case the essence of calculating and executing plans in this sort of task environment would entail some type of complex accounting scheme that could evaluate and compare the relative benefits and liabilities of various goals and the costs of achieving them.

Planning with constraints

There is a special case of the multiple goal problem that does not require complex accounting, yet exposes many key problems in a simplified setting. This special case involves two goals, one positive and one negative. The objective of the system is to achieve the single positive goal (perhaps while minimizing search and execution costs) while avoiding absolutely any state satisfying the negative goal. Thus, we are asked to solve a problem subject to certain constraints – some states are *illegal.*

Many famous puzzles such as the 'Tower of Hanoi Puzzle' and the 'Missionaries and Cannibals Problem' can be stated as constraint problems of this sort. Actually, the legal moves (operators) of these puzzles are usually restricted so that illegal states can never even be generated. But such restrictions on the preconditions of the operators are often difficult and awkward to state. Furthermore, if new constraints are added from time to time (or old ones dropped), the definitions of the operators must be correspondingly changed. We would rather have simple operator preconditions that *allow* operators to be applied even though they might produce illegal states. With this point of view we must add a mechanism that can recognize illegal states and that can analyse *why* they are illegal so that search can be guided around them.

Let us consider a simple example task using the environment of figure 2. Three rooms are connected by doorways as shown and the robot is initially in room R3. The robot can move around and push the boxes from one place to another. Let us suppose that the positive goal is to get one of the boxes into room R1. The negative goal is a box and the wedge, W1, in the same

room; thus any state with this property is illegal. For this example we can consider two operators: PUSHRM(BX, RX, DX, RY) pushes object BX from RX into adjacent room RY through connecting doorway DX. A precondition of PUSHRM is that the robot and object BX be in room RX. GOTORM(RX, DX, RY) takes the robot from room RX into adjacent room RY through connecting doorway DX. We do not complicate the precondition of PUSHRM with such a requirement as 'If BX is a box, RY cannot already contain a wedge,' or the like.

We must now consider what modifications to make to STRIPS so that it does not generate a plan that traverses an illegal state. One obvious necessity is to test each new state produced to see if it is illegal, that is, to see if the negative goal wff can be proved in that state. If the state is illegal, the search for a plan must be discontinued along that branch of the search tree and taken up somewhere else.

In problem-solving systems such as STRIPS that use the GPS means-ends strategy, it will not do merely to discontinue search at points where illegal states are reached. We must also extract information about why the state is illegal so that other operators can be inserted earlier in the plan to eliminate undesirable features of the state.

In our example of figure 2, STRIPS might first decide to apply the operator PUSHRM(BOX1, R3, D3, R1). This application results in an illegal state. If we merely discontinue search at this point, STRIPS might next decide to apply PUSHRM(BOX2, R2, D1, R1) whose precondition requires first the application of GOTORM(R3, D2, R2). But PUSHRM again results in an illegal state. Ultimately there will be no place in the search tree left to search and STRIPS will fail to find a solution.

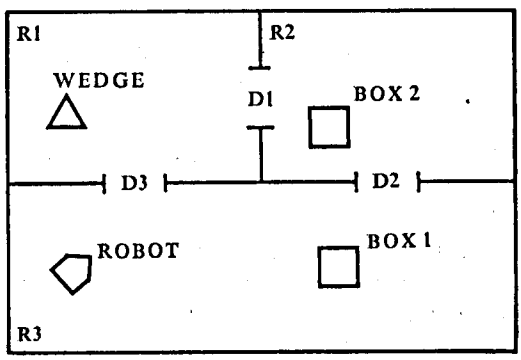


Figure 2. Robot environment for a constraint problem.

Thus, we need an analysis that is able to pursue a chain of reasoning such as the following:

(1) We have just applied PUSHRM(BOX1, R3, D3, R1), resulting in an illegal state.

(2) The reason the state is illegal is because it contains the two statements

INROOM(R1, W1)

INROOM(R1, BOX1).

They were used in the proof that the state was illegal.

(3) The operator PUSHRM is responsible for the occurrence in this state of the statement

INROOM(R1, BOX1).

Presumably the means-ends analysis technique had a reason for applying this operator and probably it was to add this statement.

(4) The other statement INROOM(R1, W1) was in the previous state since it was not added by PUSHRM.

(5) Therefore, let us add – just to this particular application of the operator PUSHRM – the additional precondition ~INROOM(R1, W1).

Thus, we would reconsider the subgoal of proving PUSHRM's preconditions in the node just above the one containing the illegal state. If there are several candidate statements whose negations could be added to the preconditions, we could create alternative nodes for each. Also, we could reconsider the whole chain of ancestor nodes containing PUSHRM's preconditions in their goal lists so that remedial action could be taken earlier in the plan.

In our example, we merely create a subgoal of getting W1 out of room R1 and, say, into room R2. This subgoal is easily achieved by pushing W1 into room R2 creating another illegal state. But this PUSHRM will also have its preconditions automatically elaborated to require that BOX2 be out of room R2. Ultimately, the plan GOTORM(R3, D2, R2), PUSHRM(BOX2, R2, D2, R3), GOTORM(R3, D3, R1), PUSHRM(W1, R1, D1, R2), GOTORM(R2, D2, R3), PUSHRM(BOX1, R3, D3, R1) would be constructed.

It should not be too difficult to modify STRIPS so that it could deal with negative goals in this manner. Further modifications might also be desirable. For example, at the time a relevant operator is selected, we might want to perform checks on it to see if it can be applied at all in light of the constraints. First we would ask whether its precondition formula alone implied the negative goal wff. There would be no point in working towards a state satisfying a subgoal precondition wff if that very wff implied a negative goal. Then we would ask whether the operator's add list alone implied a negative goal. These tests would indicate whether a relevant operator ought to be prohibited before STRIPS wastes time in planning to apply it.

Certain additional complications surround the use of previously learned plans or MACROPS by STRIPS during the planning process. STRIPS would have to check the legality of each intermediate world state produced by the components of a MACROP. If an illegal state is traversed, STRIPS would have the choice of either working around this illegal state to achieve the desired results of the entire MACROP or of planning to eliminate undesirable features before applying any of the MACROP. Decisions about the proper strategy will depend on our experience with systems of this sort.

Execution strategies

So far we have dealt only with questions concerning *planning* under constraints. After a plan is generated, our robot system must execute it, and during the execution we must avoid falling into illegal states.

Our first concern is with the process of plan generalization. STRIPS may have produced a perfectly fine specific plan that traverses no illegal states, but it could happen that its generalization has instances that do traverse illegal states. This difficulty may not be troublesome as concerns use of the MACROP in later planning, since STRIPS presumably will insure that none of these bad instances are used. During execution of the MACROP, however, we will have to make sure that our instantiation process does not select one of the bad instances.

Even if the same instance of the plan is executed as was planned, there is the added difficulty caused by new information discovered about the world during execution. While this new information might not affect at all the possibility of carrying out the original plan, it might inform us that certain of the states to be traversed are illegal. Suppose, for example, that STRIPS did not know of the existence of wedge W1 in the task of figure 2 and planned merely to push BOX1 into room R1. During execution of this plan (but, say, before BOX1 enters room R1), let us suppose that the robot discovers the wedge. What is to be done? Obviously STRIPS must generate a new plan; to do this, the executive system must have a mechanism for recognizing illegal states and for recalling STRIPS.

In certain task environments there might be negative goals that represent constant constraints for all of the tasks ever to be performed. Instead of having to plan to avoid these illegal states every time a new task is confronted, it may be more efficient to build up a list of operator instances whose application is prohibited whenever certain conditions obtain. (For example, don't walk on the train tracks when a train is coming.) STRIPS would then check each operator application to check its legality, and the executive would do likewise. In the executive system, we could regard this activity as a sort of *inhibition* of illegal operators. If a planned operator is inhibited, presumably STRIPS would be called to generate a new plan.

3. DYNAMIC ENVIRONMENTS

Introductory remarks

The STRIPS problem-solving system, as well as the associated execution routines, were designed to work in a stable, relatively static environment. Nothing changed in the environment unless the robot system itself initiated the change. There were no independent, ongoing events; for such environments we did not need the concept of time. Furthermore, the actions that the robot could perform (pushing boxes and moving about) had effects that could quite easily be represented by the simple add and delete lists of the STRIPS operators.

To progress from the simple environment in which STRIPS plans are conceived and executed to a more realistic, dynamic environment requires some new concepts and mechanisms. Some of these have already received attention at SRI and elsewhere. We hope here to discuss some of these requirements and to point out what we think might be profitable approaches.

First, we shall deal with those problems caused by the fact that the ultimate effect of a robot action might not be easily represented by a STRIPS add-list formula. It may require, instead, a computation performed by a process that simulates some aspect of the environment. Next we shall consider the special problems caused by independent processes going on in the environment (perhaps, for example, another independent robot). Last, we shall give a brief discussion of how we might introduce the concept of time. All of our comments are highly speculative and hopefully will raise questions even though they will answer few.

Representation of complex actions

In a world that is relatively 'uncoupled,' the effects of a robot action can be described simply by the STRIPS add and delete lists. The effect of the robot going to place A is to add the wff AT(ROBOT, A) and to delete any wffs saying that the robot is anywhere else. In such a simple world it is unnecessary to inquire in each case whether an action has special side effects or perhaps touches off a chain of events that affect other relations. Such side effects that do occur can simply be taken care of in the add and delete lists. The STRIPS representation of the effects of an action is an example of what we shall call an *assertional* representation. But in more highly coupled worlds, the ultimate effects of actions might depend in a complex way on the initial conditions. These effects might in fact best be modeled by a computation to be performed on the world model representing the initial conditions. We shall say that these effects are modeled by a *procedural* representation.

Several AI researchers have already stressed the need for procedural representations. Winograd, for example, in his BLOCKS program (1971), made use of features in the PLANNER (Hewitt 1971) language to represent actions as procedures. The *consequent theorems* of PLANNER are much like STRIPS operators in that they name a list of relations that are established by an action and also specify the preconditions for the action. Those effects of the action that are explicitly named are represented assertionally as in STRIPS, but side effects and preconditions are represented procedurally. There is a definite order in which preconditions are checked, and there are provisions for directions about what should be done when failures are met. *Antecedent theorems* and *erase theorems* allow more indirect effects of the action to be computed in a procedural fashion.

We note that, even in PLANNER, the named effects of an action are represented assertionally. Assertional representations have a distinct advantage for use in planning since they permit straightforward mechanisms (such

as PLANNER's pattern-directed invocation) for selecting those actions most relevant for achieving a goal. It would seem difficult to employ the strategies of means-ends analysis or chaining backwards if the effects of actions were completely hidden in procedural representations.

The assertional form of the STRIPS operators permitted us to store plans in a convenient form, *triangle tables*. These tables reveal the structure of a plan in a fashion that allows parts of the plan to be extracted later in solving related problems. It is not obvious whether one could find a similarly convenient format if operators were represented procedurally rather than assertionally. One possibility is to use operators represented assertionally as rough models for more precisely defined operators represented procedurally. High level planning could be accomplished as it now is by STRIPS, and then these plans could be checked using the more accurate procedural representations.

Independent processes

In this section we shall consider some complications that arise if there are other independent agents in the world that can perform actions on their own. For example, there might be other robots (with goals different, if not inimical, to our robot). In such a case we would need to compute plans that consider explicitly the possible 'moves' of the other agents. The other agents might not necessarily be governed by goal-oriented behaviour but might merely be other entities in nature with power to cause change, for example lightning, falling rocks, the changing seasons, and so on. In general, our robot system will not be able to predict perfectly the effects of these agents in any given situation. Thus, it might need to use game trees and minimaxing techniques. Here, though, we want to consider just the special case in which our robot system does have perfect information about what other agents will do in any given situation. Even this special case poses some difficult questions, and we think progress can be made by answering some of them first.

Before dealing with these questions, we want first to remark that what is independent and what is causal (that is, caused by our robot) we regard mainly as a matter of definition. It might, for example, be convenient to regard all but the *immediately* proximate effects of a robot action as effects caused by an independent (although perhaps predictable) agent. For example, we could take the view that the only immediately proximate effect of removing the bottom block from a tower of blocks is that the bottom block is no longer in the tower. An independent agent then is given the opportunity (which it never avoids) to act, making the other blocks fall down. Although such a division between causal and independent effects in this case sounds extreme, it may in fact have some computational advantages. At the other extreme of the spectrum of actions come those that indisputably are most conveniently thought of as being performed by an independent agent, say a human or another robot. If the second robot were constructed by the first one, the

actions of the offspring could conceivably be regarded as complex effects caused by the parent, but this interpretation seems obviously unwieldy.

Without other agents of change in the world, nothing changes unless the robot initiates the change. The robot has a set of actions it can execute and, for any initial situation, it can predict the effects of each of them. The 'physics' of such a world are simply described by the (assertional or procedural) representation of the robot's actions. With other agents present the set of physical laws is more complex. Other agents cause changes, and we assume that we can predict them. We need a representational system (again either procedural or assertional) to describe the conditions under which these changes takes place and what the changes are.

An especially interesting situation arises when our robot system can take action to avert a change that would ordinarily take place if the robot were passive. An example is given by the rule: 'The robot will be crushed unless it gets out of the path of a falling rock.' Thus, in our dynamic world populated by other agents (some of which may merely be fictitious agents of nature) we have two main types of rules for describing the 'physics' of the world. Both descriptions are relative to certain specified preconditions that we presume are met:

(1) A relation R will *change* (that is, its truth value will change) if we perform an action a; otherwise it will remain the same (all other things being equal).

(2) A relation R will *stay the same* if we perform some action a; otherwise it will change (all other things being equal).

These two types of rules can be used as justification either to perform an action or to inhibit one. (Of course, as we learned in Section 2, it is only in the case of having multiple goals that it makes sense to speak of 'inhibiting' an action. An inhibited action is one that we have decided not to execute since, simply put, it does more harm than good.) Whether we perform or inhibit an action for each rule depends on whether our goal is to change or maintain the relation R. If we further divide the class of goals into two types, good or positive goals and bad or negative ones, we get eight different kinds of resulting tactics. These are listed, with an example of each, in the chart of table 1.

The split from four to eight tactics depends on whether we choose to distinguish between positive and negative goals. Such a distinction may prove unimportant, and we make it here primarily because it allows a nice correspondence to certain behavioral paradigms explored by psychologists. (There is considerable neurophysiological evidence indicating two separate motivational systems in animals: a positive or pleasure system with foci in the lateral hypothalamus and a negative or pain system with foci in the medial hypothalamus.)

The actions used by STRIPS (as well as most other robot problem solvers) correspond to the tactic 'excite action to bring reward.' The main problems

Table 1. Eight tactics for a dynamic environment.

Physics	Goal Type		Tactic	Example
R will change if we perform a; otherwise it will stay the same.	We want R to change . . .	toward a positive goal.	Excite action to bring reward.	Insert dime in slot to get candy.
		away from a negative goal.	Excite action to escape punishment.	Leave a room having a noxious odour.
	We want R to be maintained . . .	at a positive goal.	Inhibit action to maintain reward.	Do not spit out a fine wine.
		against changing toward a negative goal.	Inhibit action to avoid punishment.	Do not step in front of a moving automobile.
R will stay the same if we perform a; otherwise it will change.	We want R to change . . .	toward a positive goal.	Inhibit action to bring reward.	Do not leave restaurant after having just ordered dinner.
		away from a negative goal.	Inhibit action to escape punishment.	Do not run when out of breath.
	We want R to be maintained . . .	at a positive goal.	Excite action to maintain reward.	Follow the Pied Piper.
		against changing toward a negative goal.	Excite action to avoid punishment.	Save for a rainy day.

posed by a dynamic world for a planning system are seen to be those stemming from the need to be able to construct plans using the other seven tactics as well. Two of the present authors, in collaboration with Ann Robinson, have given some preliminary thought to matters relating to the *execution* of action chains containing all eight types of tactics (Hart, Nilsson and Robinson 1971). We expect that the matter of designing a combined planning and execution system to operate in a dynamic world will present some very interesting problems indeed.

Time

When dealing with a dynamic environment, it becomes increasingly difficult to ignore explicit consideration of the concept of time. We would like a robot system to be able to respond appropriately to commands like:

Go to Room 20 *after* 3:00 p.m., and *then* return *before* 5:00 p.m.

Go to Room 20 three times a day.

Go to Room 20 and *wait until* we give you further instructions.

Leave Room 20 *at* 4:00 p.m.

We would also like the robot system to be able to answer questions dealing with time. For example: 'How many times did you visit Room 20 before coming here?' (See the recent article by Bruce (1972) for an example of a formalism for a question answering system dealing with time.)

A straightforward way to approach some of these issues is to add a time-interval predicate to an assertional model. Thus TIME(A, B) means that it is after A and before B. Whenever the system looks at an external clock, then presumably A and B become identical with the clock time. The model would then need some mechanism for continually revising A and B. A growing spread between A and B would represent the known inaccuracy of the model's internal clock.

For the moment we might for simplicity assume that time stands still while the robot system engages in planning. That is, we assume that the time required for planning is insignificant compared to the time duration of other events such as robot actions. (Incidentally, this assumption is manifestly not true as regards the present SRI robot system.) We can begin to deal with time by including in the operator descriptions the effect the operator has on the time predicate. (We must do the same for descriptions of the effects of independent agents.) Thus if the operator GOTHRUDOOR takes between three and five units of time, its effect on the predicate TIME(A, B) is to replace it by the predicate TIME(A+3, B+5).

We might also want to have a special operator called WAIT(n) that does nothing except let n units of time pass. With these concepts it will probably be a straightforward matter to plan tasks such as 'Be in Room 20 after 3:00 p.m. but not before 5:00 p.m.' The system would calculate the maximum length of time needed to execute the needed actions and would insert the appropriate WAIT operator somewhere in the chain if needed. Since much planning will

probably be done without reference to time at all; it will be most convenient to ignore explicit consideration of time unless the task demands it.

We suspect that allowing time to progress during planning will present many problems. Then the planner must operate with world models that will not 'stand still'. For certain time-dependent tasks; such as 'Meet me in Room 2 at 3:00 p.m.', the planner will have to be able to coordinate successfully the amounts of time spent in planning and task execution.

4. MULTIPLE OUTCOME OPERATORS

One of the interesting properties of a robot system is the inherent incompleteness and inaccuracy of its models of the physical environment and, in most cases, of its own action capabilities. This property implies a degree of indeterminism in the effects of the system's action programs and leads one to consider including in the planning mechanism consideration of more than one possible outcome for an operator application. We might like to model certain types of failure outcomes, such as a box sliding off the robot's pushbar as it is being pushed. We might also like to model operators whose primary purpose is to obtain information about the physical environment, such as whether a door is open or whether there is a box in the room. Munson (1971) and Yates (private communication) have also discussed multiple outcome operators.

We can extend the STRIPS operator description language in a natural way to provide a multiple outcome modeling capability as follows: each description can have n possible outcomes each defined by a delete list, an add list, and (optionally) a probability of the outcome. For example, an operator that checked to see if a door was open or closed might have the following description:

```
CHECKDOOR(DX)
  PRECONDITIONS
    TYPE(DX, DOOR)
    NEXTTO(ROBOT, DX)
  outcome 1                    outcome 2
  DELETE LIST                  DELETE LIST
    NIL                          NIL
  ADD LIST                     ADD LIST
    DOORSTATUS(DX, OPEN)         DOORSTATUS(DX, CLOSED)
  PROBABILITY                  PROBABILITY
    0.5                          0.5
```

We may also want to provide advice about when information gathering operators ought to be applied. For example, it would be inappropriate to apply CHECKDOOR if the robot already knew the value of DOORSTATUS. This advice might be supplied by including in the preconditions a requirement that the information to be gathered is not already known before the operator is applied. This requirement would insure that the operator description

indicates that information is being added to the model rather than being changed in the model and would therefore allow the planner to distinguish CHECKDOOR from an operator that opened closed doors and closed open doors.

Such an 'unknown' requirement would be a new type of precondition for STRIPS, since what is required is not a proof of a statement but a failure both to prove a statement and to prove the statement's negation. This requirement is analogous to the semantic meaning of the THNOT predicate in PLANNER (Hewitt 1971); that is, THNOT $p(x)$ is true if the PLANNER interpreter cannot prove $p(x)$ with some prespecified amount of effort. A capability of handling such preconditions could be added to STRIPS in a natural manner by defining new syntax for the preconditions portion of an operator description and by adding new facilities to the goal testing mechanism in STRIPS to attempt the required proofs.

Let us consider the planner's search space when multiple outcome operators are used. We assume a search tree where each node represents a state and each branch from a node represents the outcome of an operator applied to the node. This is a standard AND/OR tree in which the outcomes from different operators are OR branches and the outcomes from a single operator are AND branches. Any plan is a subtree of this tree consisting of the initial node of the tree and for each node of the plan exactly those offspring nodes produced by the outcomes of a single operator; hence, from each non-terminal node of a plan there will emanate one set of AND branches. Each terminal node must correspond to a world model satisfying the goal wff if the plan is to be successful.

When such a plan is executed, each branch in the tree corresponds to an explicit conditional test. The test determines what was the actual outcome of the action routine and the corresponding branch of the remainder of the plan is executed next. Thus the problem of building into plans explicit conditional statements does not appear difficult; providing explicit loops looks considerably harder.

If probabilities of the different outcomes of operators are defined, then the probability of occurrence of any node in a plan tree (during execution of the plan) might be considered to be the product of the probabilities of the branches connecting the node with the initial node of the tree.

One of the interesting issues facing the planning program is when to stop. One certainly would have the program stop when it achieved absolute success, in that a plan existed each of whose terminal nodes represented a state that satisfied the task; for such a plan each of its operators could produce any of the described outcomes and the plan would still lead to success. Similarly, the planner would stop when it reached an absolute failure; that is, when the tree had been fully expanded and no terminal node of any plan satisfied the task; in such a situation we know that none of the plans in the tree will lead to success no matter what the outcomes of the operators. A node has been fully

expanded when all possible offspring nodes have been included in the tree. Thus, terminal nodes in a fully expanded tree have no possible offspring. To make the concept of a fully expanded node more practical, one may assume that only 'relevant operators' are considered for producing offspring.

But what about other situations? For example, consider the case where the planning tree is fully expanded (as in the absolute failure case) and absolute success has not been achieved, but there are nodes in the tree representing states that satisfy the task. In this situation we have plans in the tree with a nonzero probability of success; that is, for those plans we could specify an outcome for each operator in the plan that would cause the plan to satisfy the task. Since the search tree is fully expanded the planning program must stop, but what does it return as a result? A reasonable answer might be to return the plan with the highest probability of success, where the probability of success of a plan is defined to be the sum of the probabilities of occurrence of the plan's nodes that satisfy the task.

If we are willing to accept plans with a probability of success less than one, then perhaps we should consider stopping the planner before achieving absolute success or full expansion of the tree. For example, we might stop the planner whenever it has found a plan whose probability of success is greater than some threshold. Such a stopping criterion would have the advantage of preventing the system from expending planning effort in a situation where a plan has already been found that is almost certain to succeed.

The system's plan executor can deal with less than absolutely successful planning by being prepared to recall the planner when a terminal node of a plan is achieved and the task is still not satisfied. It may be advantageous to recall the planner before such a terminal node is reached; namely, when a node is achieved in the plan no offspring of which satisfies the task. Even though more plan steps might remain to be taken in this situation, they will not satisfy the task and the planner may determine that some other sequence of steps is more desirable to successfully complete the execution. In fact, one could argue that the planner should be given the opportunity to continue expansion of the search tree after each execution step, since the probabilities of occurrence of nodes and success of plans changes at each step; such a strategy seems an extreme one and probably cannot be justified practically.

Just as the planning program could stop with less than absolute success, there are clear cases when it should stop with less than absolute failure. Two such cases come to mind. The first is where the search tree has been expanded sufficiently to allow determination that no plan in the tree will be able to qualify as a successful plan. This would typically happen when every plan in the tree has a probability greater than some threshold of achieving a terminal node that does not satisfy the task. The second case is where the probability of each unexpanded node in the search tree is less than some small threshold. In this situation the search tree has been expanded to such an

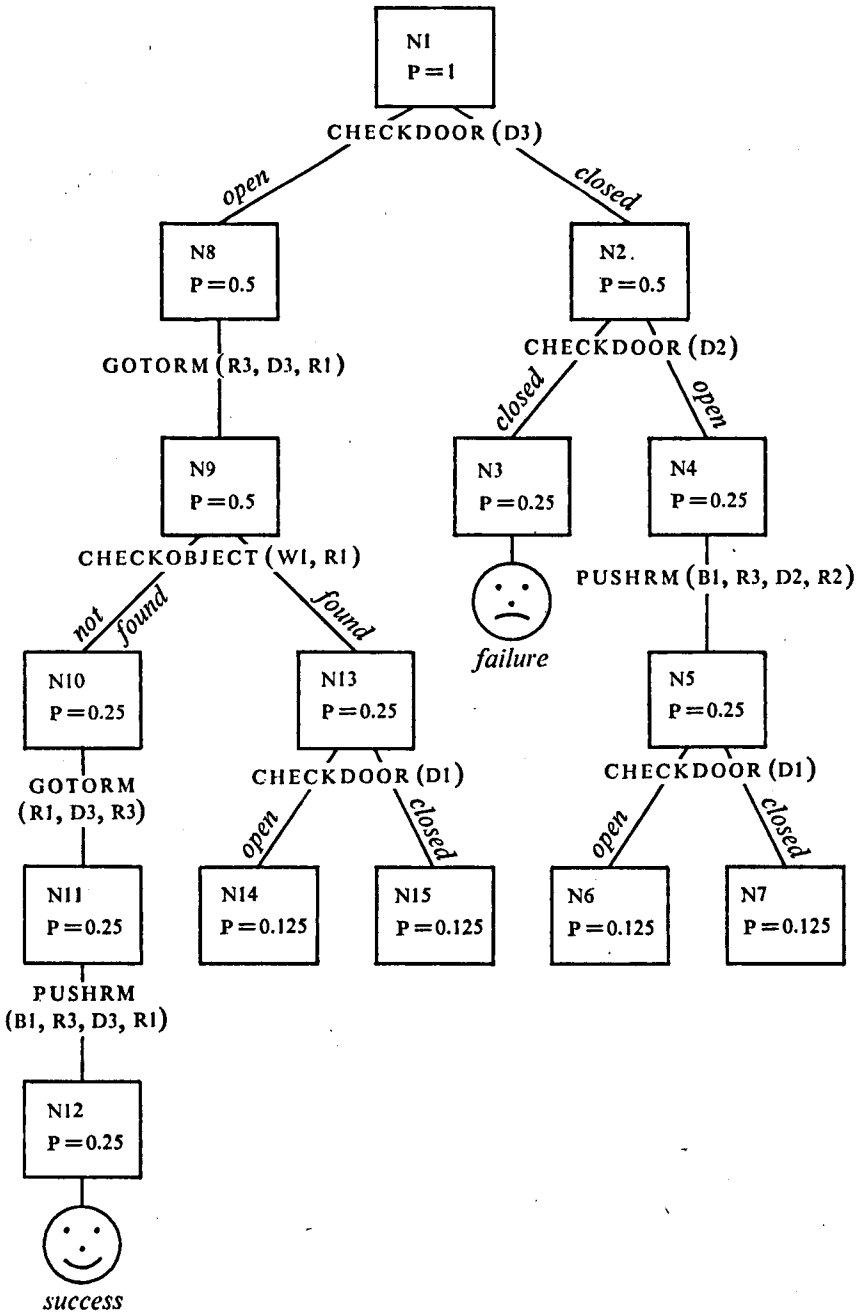


Figure 3. An example of a conditional plan.

extent that none of the states to be considered are very likely and therefore any further planning effort will only minimally increase the probability of success of any plan in the tree.

During planning, a node selection function will be needed to direct expansion of the search tree. One reasonable selection strategy is to first select the most promising plan in the tree and then select a node in that plan for expansion. Plans could be evaluated on such criteria as the probabilities of the plan's success and failure nodes, distance estimates to the goal from the plan's unexpanded nodes, and so on. Nodes within a plan could be evaluated on such criteria as estimated distance to goal and probability of occurrence.

An interesting example of the use of multiple-outcome operators can be seen by reconsidering the wedge and boxes problem discussed in Section 2 above. In the problem we wish the robot to move box B1 into room R1 under the constraint that there should never be a box in the same room with a wedge W1 (see figure 2). Assume now that the system does not know the status (open or closed) of the doors D1, D2, and D3, nor does it know if wedge W1 is in room R1. Hence, the desired plan will contain checks on the status of doors and a check to see if wedge W1 is in room R1. If we further assume that the planner halts when all unexpanded nodes have probability less than 0·2, then the plan shown in figure 3 might be the output from the planner.

This plan begins by checking the status of door D3. If D3 is closed, then D2 is checked. If D2 is closed then there are no relevant applicable operators and therefore this is a failure node. In the case where D2 is open, then box B1 is pushed into room R2 and door D1 is checked. The probability of occurrence of each of the nodes resulting from the check of D1 is less than 0·2; hence they are not expanded further. Back up at the beginning of the plan, if door D3 is found to be open, then the robot goes to room R1 and checks for wedge W1. If W1 is not found then the plan is to go back to room R3 and push box B1 into room R1; this branch of the plan completes the task and therefore forms a success node. In the case where W1 is found to be in room R1, a check is made on door D1. The probability of occurrence of each of the nodes resulting from this check is less than 0·2, and they are therefore not expanded. For this plan the probability of success is 0·25, the probability of failure is 0·25, and the remaining possible outcomes are unexpanded.

5. TEAMS OF ROBOTS

Thus far we have considered robot systems having only a single effector with which to manipulate the world. We now consider some of the problems posed by multiple robot systems.

Let us agree first that a multiple or team robot system is a system composed of several effectors controlled, at least in part, by a common set of programs. If this were not the case, then we would have several single robots whose environments happened to be complicated by the presence of other robots.

To make the discussion specific, imagine a system with two effectors: a mobile vehicle that can fetch boxes, and an arm that can lift boxes and put them on a shelf. For simplicity, we will say that these abilities are characterized by only two operators, FETCH and SHELVE. The preconditions for SHELVing a box are that the box be in the workspace of the arm. There are no preconditions for FETCHing a box (there is an inexhaustible supply of boxes and the vehicle can always bring one to the arm).

We now give the robot system the task of putting three identical boxes on the shelf. It is interesting first to note that typical current problem-solving programs would not need to recognize explicitly that the two available operators are actually implemented by separate devices. Oblivious to this fact, a successful problem solver would produce any of a number of satisfactory plans, including, for example, the sequence FETCH, FETCH, FETCH, SHELVE, SHELVE, SHELVE.

Our interest now centers on the real-world execution of this plan. In particular, we would consider a robot system a bit stupid if it waited until all three FETCHes were successfully executed before it proceeded to SHELVE. What is needed here is a means of analyzing the interdependencies of the components of the plan and of concurrently executing independent actions of the two effectors. Now, the general issue of partitioning a process into partially independent subprocesses is a complicated one that arises in studies of parallel computation (Holt and Commoner 1970). Here, though, we can make use of the triangle-table format to obtain the dependencies directly. Figure 4a is a sketch of the triangle table for the complete plan. Recall that a mark in a cell means that the operator heading the column produced an effect needed to help establish the preconditions of the operator on the same row. The bottom row of the table corresponds to the 'precondition' GOAL, while the first column of the table corresponds to preconditions provided by the initial state of the world. (Since FETCH has no preconditions, and since all of the preconditions for SHELVE are established by FETCHes, the first column in figure 4a has no marks.)

The complete triangle table can be partitioned into a pair of subtables as shown in figures 4b and 4c. To create the FETCH subtable, we copy all marks in F columns and non-F rows of the complete table into the bottom row of the subtable. To create the SHELVE subtable, we copy all marks in S rows and non-S columns of the complete table into the first column of the subtable. Notice the motivation here: the purpose of each FETCH is to satisfy a condition external to the FETCHing robot, and so are external (bottom row) goals. Similarly, the preconditions for the SHELVing robot are established by an agent external to it, and so are external (first column) preconditions. (We defer for a moment discussing a procedure for partitioning a complete triangle table when the effectors are more intimately interleaved.)

Once the two subtables have been constructed, the PLANEX algorithm described earlier can be used – in a slightly modified form – to direct the

actions of the two effectors. It is plain from figure 4b that the vehicle can fetch boxes independently of any arm actions, since the arm contributes nothing toward establishing preconditions for FETCHing. The situation with respect to the arm is a little more complicated. Using the standard PLANEX algorithm, the arm could not execute a SHELVE action until all the FETCH operations had been completed, which of course is the very thing we are trying to avoid.

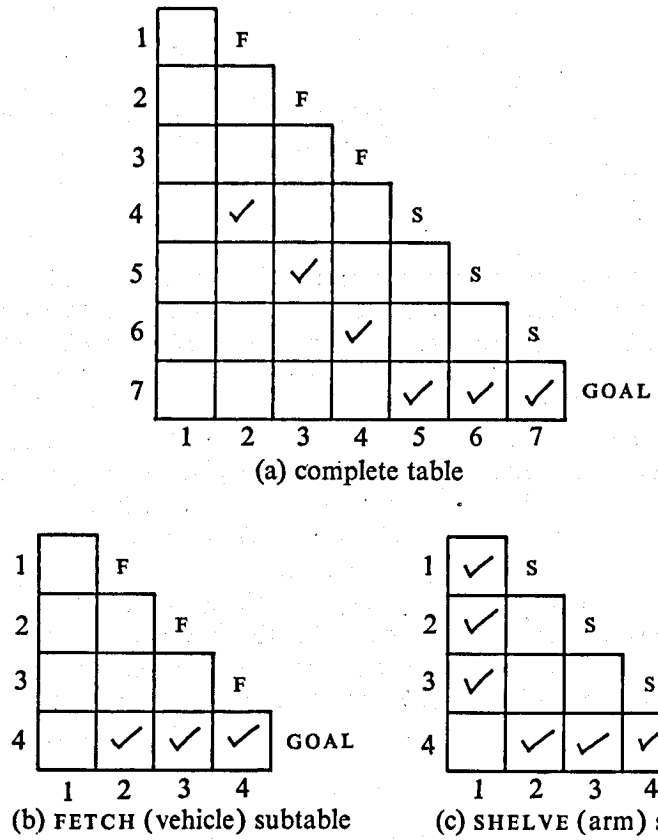


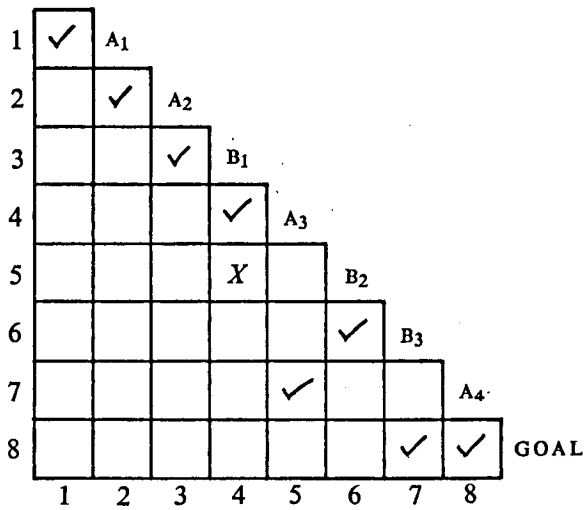
Figure 4. Triangle tables for FETCH–SHELVE example.

We must therefore distinguish between those preconditions of SHELVing that are to be established by the other effector, and those preconditions that must simply be true in the initial world (there are none of the latter in our example). Upon making this distinction, we recognize that, in order to execute the first SHELVE operation, we need only have achieved the effect marked in the top cell of the first column of figure 4c; the other effects marked in succeeding cells of that column will be added by an active agent external to the arm. In other words, whereas a single robot must have all of the basic preconditions of its plan satisfied before it begins executing the plan, the

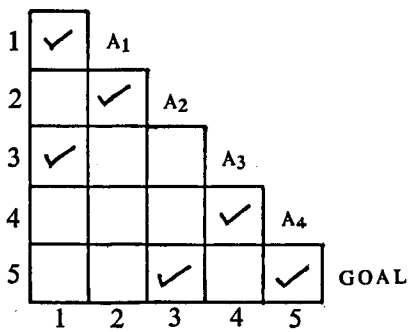
team robot can have some of the initial requirements of its plan satisfied by other robots while its own plan is being executed.

The foregoing discussion can be made more precise and, we believe, can be formulated into a straightforward algorithm for executing linear plans for multiple robots. Instead of pursuing this algorithm, let us return to the question of partitioning a complete two-robot triangle table.

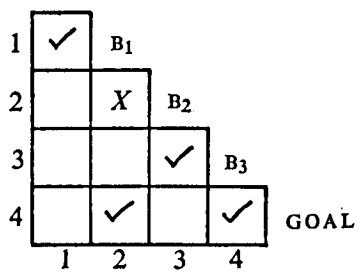
We will present the algorithm by means of an example. Figure 5a shows a complete triangle table for an imaginary system composed of two robots, A and B. For present purposes we need not consider in detail the variety of different operations that A and B can perform; we need only note in the complete table which robot performs each operation. The X in cell (5, 4) is a mark like all the others, but merely for purposes of exposition we have distinguished between the two marks in column 4.



(a) complete table



(b) A subtable



(c) B subtable

Figure 5. Triangle tables for algorithm illustration.

The algorithm proceeds by scanning the columns corresponding to either of the robots; let us begin with B, just to illustrate this insensitivity to order. Every mark in a B column and a non-B row is entered in the corresponding column of the bottom row of the B subtable. Cell (4, 4) of the complete table contains such a mark, and this mark is therefore entered in cell (4, 2) of the B subtable. A mark in a B column and B row of the complete table is entered into the corresponding cell of the subtable; cell (5, 4) contains such a mark. Next, we consider the B rows of the complete table. Every mark in a B row and non-B column is entered in the first column of the corresponding row of the subtable. Cell (3, 3) of the complete table contains such a mark, and this mark is entered into cell (1, 1) of the B subtable. When this process of scanning rows and columns is completed for the B robot, it is repeated for the A robot, producing finally the two subtables shown.

It appears that this basic algorithm is easily extended to any number of robots. If we had n robots, we would partition them into two subclasses: robot 1 and the remaining $n-1$. After processing this 'two-robot' system, we would recursively process the subtable corresponding to the $n-1$ system, and so forth, until n subtables were produced. However, we have not analysed whether systems of more than two robots can encounter such difficulties as 'locked states,' in which each robot waits for the results of all the others.

A final speculation concerns extending the notion of problem-partitioning from the execution phase into the planning phase. Specifically, we would like to give our multi-robot system an executive planner that has a rough characterization about the abilities of each of its effectors, and subordinate planners that do detailed planning for each effector once its goals have been established by the executive. To use the preceding example, we would like the executive planner to recognize that the initial problem can be partitioned into the two subproblems 'fetch three boxes' and 'shelve three boxes.' The ability to partition a problem in this fashion requires, of course, a deep understanding of its structure.

6. OTHER ISSUES OF INTEREST

In the foregoing sections we discussed a number of topics in robot problem-solving research that, we think, may be profitably explored. Of course, we do not mean to imply that there are no other worthy avenues of research. On the contrary, one could compile a long list of such possibilities. In the remainder of this section we mention a few of them, together with some personal observations on the issues they raise.

Task specification languages

A number of early problem-solving programs accepted problem statements in the form of a static assertion to be made true. (STRIPS, obviously, falls in this class.) We are increasingly convinced of the inadequacy of this formulation, chiefly because there is a large class of tasks that are most naturally

posed in procedural terms (Fikes 1970). For example, the task 'Turn off the lights in every room on the corridor' can be posed procedurally in a straightforward manner, but can be posed assertionally only by enumerating the rooms or by introducing awkward notations.

As a general observation, we note that there is a trend for problem solvers to avoid explicit use of state variables because of the attending frame problem. But suppressing state variables makes it difficult to pose problems requiring the identification of distinguished states. Thus, for example, STRIPS cannot naturally be given the problem 'Bring the box from Room 3' because the description of the box depends on a property of the current state.

The importance of incorporating procedural information in problem solvers is now generally recognized. Our point here is that a procedural formalism is badly needed even to *state* an interestingly broad class of problems.

Multiple levels of planning

Two topics of interest entailing multiple levels of planning have arisen from our work with the SRI robot system. The first results from our learning experiments in which we store plans in memory to be used during the creation of future plans. Roughly speaking, we create a description of a plan when we store it away so that STRIPS can consider it to be a new operator, or MACROP, and can include it as a single step in a new plan. A major problem arises when one attempts to generate automatically the operator description for the plan to be stored. Specifically, the components of that description are too detailed. Typically the preconditions of a STRIPS operator will have 4 or 5 statements in it; the preconditions for a typical MACROP might have 15 or 20 such statements. If that MACROP now becomes a single step in some new MACROP, then the preconditions will be even larger in the new operator description. This explosion in precondition complexity is clearly a major barrier to the bootstrapping capabilities of this system.

The source of this difficulty is that the level of detail in a MACROP description does not match the level of planning at which the MACROP is to be used. Somehow, there must be an abstraction process to suppress detail during the construction of a MACROP description. One way in which this suppression of detail might be accomplished is by including only some subset of the preconditions and effects that would normally appear in the MACROP description. The challenge, obviously, is to determine some reasonable criteria for determining those subsets. It may be necessary to consider more sophisticated abstraction schemes involving alteration of predicate meanings or creation of new predicates. For example, the meaning of a predicate such as 'location' in human planning depends on the level at which the planning is being done. In the initial planning stages for a trip to the *MI-7* conference it is sufficient to consider the location of the conference to be Edinburgh. When planning hotel accommodations, the location of the conference as a street address is of

interest. On arrival for the first conference session, the location of the conference as a room number becomes important. Similarly, a robot planner needs to employ meanings for predicates that match its planning level.

The second interesting issue concerning multiple levels of planning is communication among the levels; in particular, transmission of information concerning failures. For example, in the current SRI robot system, STRIPS considers it sufficient for applying GOTO(BOX1) to get the robot into the same room as BOX1. When GOTO(BOX1) is executed, a lower level path-finding routine attempts to plan a path for the robot to BOX1. If no such path exists, the routine exits to PLANEX, which in most cases will try the same GOTO action again. Even if a replanning activity is initiated, STRIPS might still generate a plan involving GOTO(BOX1) under similar circumstances, and the system will continue to flounder. The problem here is the lack of communication between the path finding planner and STRIPS. Somehow, the failure of the path planner should be known to a planning level that can consider having the robot move obstacles or re-enter the room from another door.

In general, what is needed is a system capable of planning at several levels of abstraction and having the appropriate operator descriptions and world models for each planning level. Given a task, a plan is constructed at the highest level of abstraction with the least amount of detail. The planner then descends a level and attempts to form a plan at the new level. The higher-level plan is used to guide the new planning effort so that in effect the attempt is to reform the higher-level plan at the current level of detail. This process continues until the desired level of detail is obtained or a failure occurs. In the case of a failure, the result of the more detailed planning needs to include some information as to why the failure occurred, so that an intelligent alternative plan can be formulated at a higher level.

'Ad hocism' in robot planning

An important issue in the design of planners for robot systems centers on the amount of '*ad hoc*ism' one will allow to be preprogrammed into the system. STRIPS, to cite one example, is a general purpose planner requiring only simple operator descriptions and a set of axioms describing the initial state of the world. Our design of STRIPS can be viewed as an extreme on the side of generality, emphasizing automatic mechanisms for extracting planning decision criteria from any problem environment presented.

A currently popular strategy for designing a planning program is to use one of the new languages, PLANNER (Hewitt 1971) or QA4 (Derksen, Rulifson and Waldinger 1972), and write programs as consequent theorems and antecedent theorems instead of operator descriptions. This approach has the advantage of allowing the person defining the problem environment to guide the plan formation process by ordering goals and giving advice as to which operators to consider. In general, this means that the problem

definer is free to build into the programs information about the problem domain that will aid in the planning process.

These alternative approaches prompt an inquiry into the goals of problem-solving research. If the purpose is to design a planner for a fixed physical environment, a given robot with a fixed set of action routines, and a fixed set of predicates and functions to appear in the models, then one builds into the system as much specific information as possible about how to accomplish tasks in the environment. For example, one might specify the optimal paths between any two rooms, provide recognition programs for each object in the environment, and so forth. Such a system might be useful for some particular application, but interest in its design as a research task is minimal.

What kind of robot planning system is of interest for problem-solving research? We suggest that a key issue in the consideration of that question is the system's capability of being extended. One type of extendability of interest is exploration. That is, we want a robot system to be capable of going into new areas of its environment, adding information obtained in the exploration to its world model, and finally, performing tasks in the new area. An exploration capability limits the level of '*ad hoc*ness' we can program into the system. For example, the planner cannot depend on pre-programmed paths between rooms and preprogrammed object recognition programs if exploration is going to bring new rooms and new objects into consideration.

A second type of extendability of interest to us is the learning of new action capabilities. One would like the system to produce automatically new action routines (like, for example, the STRIPS MACROPS) from its experience; but even merely allowing human introduction of additional action programs places restrictions on the level of '*ad hoc*ness' in the system. For example, if there is only one action routine in the system for moving a box from one room to some other room, and if we know that no new actions will ever be added to the system, then we could preprogram the planner to call that action whenever a box was needed in a room. But if we allow the possibility of a new action being added to the system that, say, is specially designed to move a box to an *adjacent* room, then we want our system design to be such that the new program can be easily added and that the planner will then make intelligent use of it.

Another aspect of the extendability issue is that a system should be a good experimental tool for research. That is, we might want to be continually extending the system to include new areas of interest such as those discussed in this paper. Hence, we would like the basic formalism and structure of our system to be flexible enough to allow exploration of these areas without requiring large amounts of reprogramming effort.

In conclusion, then, we can say that it seems certainly worthwhile to provide facilities in planning programs for easy specification of *ad hoc* information, and STRIPS is still deficient in that regard. The danger seems to

be that AI researchers may be seduced into designing systems that are so dependent on advice about specific situations that they have no extendability and are little more than programs containing solutions to a very restricted set of problems. Our discussion in this section has attempted to point out that one way to temper the nature and extent of a problem-solving program's dependence on *ad hoc* information is to consider various ways in which one might wish the program to be extendable.

Acknowledgement

This research was sponsored by the Office of Naval Research under Contract N00014–71–C–0294 and the Defence Advanced Research Projects Agency under Contract DAHC04–72–C–0008.

REFERENCES

Bruce, B.C. (1972) A model for temporal references and its application in a question-answering program. *Art. Int.*, **3**, 1–25.

Derksen, J., Rulifson, J.F. & Waldinger, R.J. (1972) QA4 language applied to robot planning. *AIC Technical Note 65*, SRI Project 8721. California: Stanford Research Institute.

Fikes, R.E. (1970) REF-ARF: a system for solving problems stated as procedures. *Art. Int.*, **1**, 27–120.

Fikes, R.E. & Nilsson, N.J. (1971) STRIPS: a new approach to the application of theorem proving to problem solving. *Art. Int.*, **2**, 189–208.

Fikes, R.E., Hart, P.E. & Nilsson, N.J. (in press) Learning and executing robot plans. *Art. Int.*

Green, C.C. (1969) Application of theorem proving to problem solving. *Proc. Int. Joint Conf. on Art. Int.*, pp. 219–40. Washington DC.

Hart, P.E., Nilsson, N.J. & Robinson, A.E. (1971) A causality representation for enriched robot task domains. Tech. Report for Office of Naval Research, Contract N00014–71–C–0294, SRI Project 1187. California: Stanford Research Institute.

Hewitt, C. (1971) Procedural embedding of knowledge in PLANNER. *Proc. Second Int. Joint Conf. Art. Int.*, pp. 167–82. London: British Computer Society.

Holt, A. & Commoner, F. (1970) Events and conditions. *Record of Project MAC Conf. Concurrent Systems and Parallel Computation*, pp. 1–52. Massachusetts: Association for Computing Machinery.

Munson, J.H. (1971) Robot planning, execution, and monitoring in an uncertain environment. *AIC Technical Note 59*, SRI Project 8259. California: Stanford Research Institute.

Winograd, T. (1971) Procedures as a representation for data in a computer program for understanding natural language. *A.I. Technical Report 17*, MIT. Artificial Intelligence Laboratory, Cambridge, Mass. MIT. Also available as *Understanding Natural Language*. Edinburgh: Edinburgh University Press 1972.

Yates, R., private communication.

24

The MIT Robot

P. H. Winston

Artificial Intelligence Laboratory
Massachusetts Institute of Technology

INTRODUCTION

Research in machine vision is an important activity in artificial intelligence laboratories for two major reasons. First, understanding vision is a worthy subject for its own sake. The point of view of artificial intelligence allows a fresh new look at old questions and exposes a great deal about vision in general, independent of whether man or machine is the seeing agent. Second, the same problems found in understanding vision are of central interest in the development of a broad theory of intelligence. Making a machine see brings one to grips with problems like that of knowledge interaction on many levels and of large system organization. In vision these key issues are exhibited with enough substance to be nontrivial and enough simplicity to be tractable.

These objectives have led vision research at MIT to focus on two particular goals: learning from examples and copying from spare parts. Both goals are framed in terms of a world of bricks, wedges, and other simple shapes like those found in children's toy boxes.

Good purposeful description is often fundamental to research in artificial intelligence, and learning how to do description constitutes a major part of our effort in vision research. This essay begins with a discussion of that part of scene analysis known as body finding. The intention is to show how our understanding has evolved away from blind fumbling toward substantive theory.

The next section polarizes on the organizational metaphors and the rules of good programming practice appropriate for thinking about large knowledge-oriented systems. Finding groups of objects and using the groups to get at the properties of their members illustrates concretely how some of the ideas about systems work out in detail.

The topic of learning follows. Discussing learning is especially appropriate here not only because it is an important piece of artificial intelligence theory but also because it illustrates a particular use for the elaborate analysis machinery dealt with in the previous sections.

Finally a scenario exhibits the flavor of the system in a situation where a simple structure is copied from spare parts.

EVOLUTION OF A SEMANTIC THEORY

Guzman and the body problem

The body-finding story begins with an *ad hoc* but crisp syntactic theory and ends in a simple, appealing theory with serious semantic roots. In this the history of the body-finding problem seems paradigmatic of vision system progress in general.

Adolfo Guzman started the work in this area (Guzman 1968). I review his program here in order to anchor the discussion and show how better programs emerge through the interaction of observation, experiment, and theory.

The task is simply to partition the observed regions of a scene into distinct bodies. In figure 1, for example, a reasonable program would report something like (ABC) and (DE) as a plausible partitioning of the five regions into, in this case, two bodies. Keep in mind that the program is after only one good, believable answer. Many simple scenes have several equally justifiable interpretations.

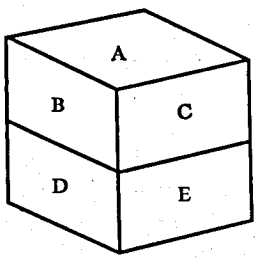


Figure 1. The task of the body-finding program is to understand how the regions of the scene form bodies.

Guzman's program operates on scenes in two distinct passes, both of which are quite straightforward. The first pass gathers local evidence, and the second weighs that evidence and offers an opinion about how the regions should be grouped together into bodies.

The local-evidence pass uses the vertices to generate little pieces of evidence indicating which of the surrounding regions belong to the same body. These quanta of evidence are called links. Figure 2 lists each vertex type recognized and shows how each contributes to the set of links. The arrow links always argue that the shaft-bordering regions belong together; the fork more ambitiously provides three such links, one for each pair of surrounding regions, and so on. The resulting links for the scene in figure 1 are displayed superimposed on the original drawing in figure 3a. Internally the links are represented in list structure equivalent to the abstract diagram in figure 3b.

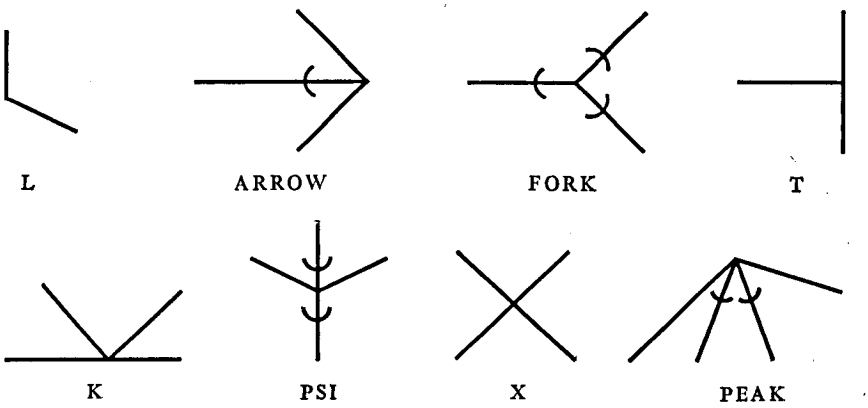


Figure 2. The Guzman links for various vertex types.

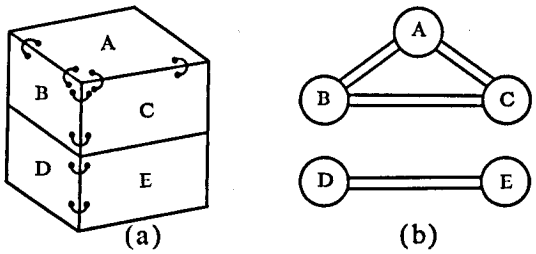


Figure 3. The links formed by the vertices of a simple scene.

There the circles each represent the correspondingly lettered region from figure 3a. The arcs joining the circles represent links.

The job of pass two is to combine the link evidence into a parsing hypothesis. How Guzman's pass two approached its final form may be understood by imagining a little series of theories about how to use the evidence to best

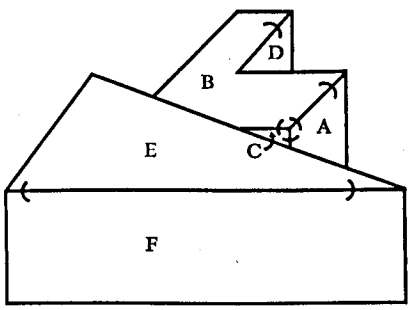


Figure 4. Various linking algorithms cause this to be seen as two, three, or four bodies.

advantage. Figure 3a is so simple that almost any method will do. Consequently figure 4 and figure 5 are used to further illustrate the experimental observations behind the evolving sequence of theories.

The first theory to think about is very simple. It argues that any two regions belong to the same body if there is a link between them. The theory works fine on many scenes, certainly on those in figure 3a and figure 4. It is easy, however, to think of examples that fool this theory, because it is far too inclined toward enthusiastic region binding. Whenever a coincidence produces an accidental link, as for example the links placed by the spurious psi vertex in figure 5, an error occurs in the direction of too much conglomeration.

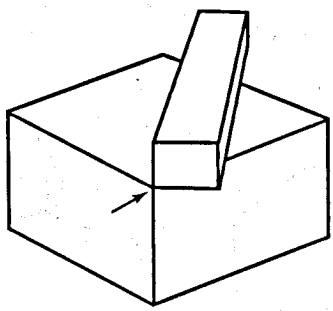


Figure 5. A coincidence causes placement of an incorrect link.

The problem is corrected in theory two. Theory two differs from theory one because it requires two links for binding rather than just one. By insisting on more evidence, local evidence anomalies are diluted in their potential to damage the end result. Such a method works fine for figure 5, but as a general solution the two link scheme also falters, now on the side of stinginess. In figure 4, partitioning by this second theory yields (AB) (C) (D) (EF).

This stinginess can also be fixed. The first step is to refine theory two into theory three by iterating the amalgamation procedure. The idea is to think of previously joined together region groups as subject themselves to conglomeration in the same way as regions are joined. After one pass over the links of figure 4, we have A and B joined together. But the combination is linked to C by two links, causing C to be sucked in on a second run through the linking loop. Theory three then produces (ABC) (D) (EF) as its opinion.

Theory four supplements three by adding a simple special-case heuristic. If a region has only a single link to another region, they are combined. This brings figure 4 around to (ABCD) (EF) as the result, without re-introducing the generosity problem that came up in figure 5 when using theory one. That scene is now also correctly separated into bodies.

Only one more refinement is necessary to complete this sequence of imagined theories and bring us close to Guzman's final program. The required addition is motivated by the scenes like that of figure 6. There we

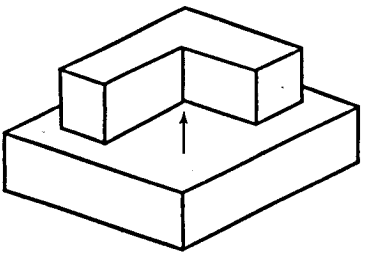


Figure 6. The fork vertex causes the two bodies to be linked together unless the offending links are inhibited by the adjacent arrows.

have again too much linking as a result of the indicated fork vertex. Although not really wrong, the one-object answer seems less likely to humans than a report of two objects. Guzman overcame this sort of problem toward the end of his thesis work not by augmenting still further the evidence weighing but rather by refining the way evidence is originally generated. The basic change is that all placement of links is subject to inhibition by contrary evidence from adjacent vertices. In particular, no link is placed across a line if its other end is the barb of an arrow, a leg of an L, or a part of the crossbar of a T. This is enough to correctly handle the problem of figure 6. Adding this link-inhibition idea gives us Guzman's program in its final form. In the first pass the program gathers evidence through the vertex-inspired links that are not inhibited by adjacent vertices. In the second pass, these links cause binding together whenever two regions or sets of previously bound regions are connected by two or more links. It is a somewhat complex but reasonably talented program which usually returns the most likely partition of a scene into bodies.

But does this program of Guzman's constitute a theory? If we use an informal definition which associates the idea of useful theory with the idea of description, then certainly Guzman's work is a theory of the region-parsing aspect of vision, either as described here or manifested in Guzman's actual machine program. I must hasten to say, however, that it stands incomplete on some of the dimensions along which the worth of a theory can be measured. Guzman's program was insightful and decisive to future developments, but as he left it, the theory had little of the deep semantic roots that a good theory should have.

Let us ask some questions to understand better why the program works instead of just how it works. When does it do well? Why? When does it stumble? How can it be improved?

Experiment with the program confirms that it works best on scenes composed of objects lacking holes (Winston 1971) and having trihedral vertices. (A vertex is trihedral when exactly three faces of the object meet in three-dimensional space at that vertex.)

Why should this be the case? The answer is simply that trihedral vertices most often project into a line drawing as L's, which we ignore, and arrows and forks, which create links. The program succeeds whenever the weak reverse implication that arrows and forks come from trihedral vertices happens to be correct. Using the psi vertex amounts to a corollary which is necessary because we humans often stack things up and bury an arrow-fork pair in the resulting alignment. From this point of view, the Guzman program becomes a one-heuristic theory in which a link is created whenever a picture vertex may have come from a trihedral space vertex.

But when does the heuristic fail? Again experiments provide something of an answer. The trihedral vertex heuristic most often fails when alignment creates perjurous arrows. Without some sort of link inhibition mechanism, it is easy to construct examples littered with bad arrows. To combat poor evidence, two possibilities must be explored. One is to demand more evidence, and the other is to find better evidence. The complexity and much of the arbitrary quality of Guzman's work results from electing to use more evidence. But using more evidence was not enough. Guzman was still forced to improve the evidence via the link-inhibition heuristic.

The startling fact discovered by Eugene Freuder is that link inhibition is enough! With some slight extensions to the Guzman inhibition heuristics (Rattner 1970), complicated evidence weighing is unnecessary. A program that binds with one link does about as well as those that are more involved. By going into the semantic justification for the generation of links, we have a better understanding of the body-linking problem and we have a better, more adequate program to replace the original one. This was a serious step in the right direction.

Shadows

Continuing to trace the development of MIT's scene-understanding programs, the next topic is a sortie into the question of handling shadows. The first work at MIT on the subject was done by Orban (1970). His purpose was to eliminate or erase shadows from a drawing. The approach was Guzman-like in flavor, for Orban worked empirically with vertices, trying to learn their language and discover heuristic clues that would help establish shadow hypotheses. He found that fairly complex scenes could be handled through the following simple facts: (1) a shadow boundary often displays two or more L-type vertices in a row; (2) shadow boundaries tend to form psi-type vertices when they intersect a straight line; and (3) shadows may often be found by way of the L's and followed through psi's.

Orban's program is objectionable in the same way as Guzman's is; namely, it is largely empirical and lacking in firm semantic roots. The ideas work in some complex scenes only to fail in others. Particularly troublesome is the common situation where short shadow boundaries involve no L-type vertices.

After Orban's program, the shadow problem remained at pasture for some

time. The issue was avoided by placing the light source near the eye, thus eliminating the problem by eliminating the shadows. Aside from being disgusting aesthetically, this is a poor solution because shadows should be a positive help rather than a hindrance to be erased out and forgotten.

Interest in shadows was re-awakened in conjunction with a desire to use more knowledge of the three-dimensional world in scene analysis. Among the obvious facts are the following:

(1) The world of blocks and wedges has a preponderance of vertical lines. Given that a scene has a single distant light source, these vertical lines all cast shadows at the same angle on the retina. Hence when one line is identified as a shadow, it renders all other lines at the same angle suspect.

(2) Vertical lines cast vertical shadows on vertical faces.

(3) Horizontal lines cast shadows on horizontal faces that are parallel to the shadow-casting edges.

(4) If a shadow line emerges from a vertex, that vertex almost certainly touches the shadow-bearing surface.

With these facts, it is easy to think about a program that would crawl through the scene of figure 7, associating shadow boundaries with their parent edges as shown. One could even implement something, through point four, that would allow the system to know that the cube in figure 7 is lying on the table rather than floating above it. Such a set of programs would be on the same level as Freuder's refinement of Guzman's program with respect to semantic flavor. We were in fact on the verge of implementing such a program when Waltz radicalized our understanding of both the shadow work and the old body-finding problem.

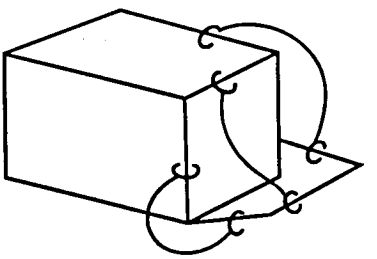


Figure 7. Simple heuristics allow shadow lines to be associated with the edges causing them.

Waltz and semantic interpretation

This section deals with the enormously successful work of Waltz (1972a, 1972b). Readers familiar with either the work of Huffman (1971) or that of Clowes (1971) will instantly recognize that their work is the considerable foundation on which Waltz's theory rests.

A line in a drawing appears because of one or another of several possibilities in the physical structure: the line may be a shadow, it may be a crack between two aligned objects, it may be the seam between two surfaces we see, or it may be the boundary between an object and whatever is at the back of it.

It is easy enough to label all the lines in a drawing according to their particular cause in the physical world. The drawing in figure 8, for example, shows the Huffman labels for a cube lying flat on the table. The plus labels represent seams where the observer sees both surfaces and stands on the convex side of the surfaces with the inside of the object lying on the concave. The minus labels indicate the observer is on the concave side. And the arrowed lines indicate a boundary where the observer sees only one of the surfaces that form the physcial edge.

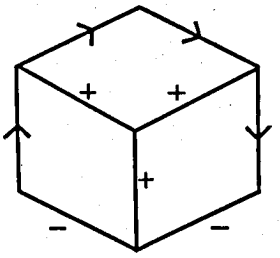


Figure 8. Huffman labels for a cube. Plus implies a convex edge, minus implies concave, and an arrow implies only one of the edge-forming surfaces is visible.

A curious and amazing thing about such labeled line drawings is that only a few of the combinatorially possible arrangements of labels around a vertex are physically possible. We will never see a L-type vertex with both wings labeled plus, no matter how many legal line drawings we examine. (It is presumed that the objects are built of trihedral vertices and that the viewpoint is such that certain types of coincidental alignment in the picture domain are lacking.) Indeed it is easy to prove that an enumeration of all possibilities allowed by three-dimensional constraints includes only six possible L-vertex labelings and three each of the fork and arrow types. These are shown in figure 9.

Given the constraints the world places on the arrangements of line labels around a vertex, one can go the other way. Instead of using knowledge of the real physical structure to assign semantic labels, one can use the known constraints on how a drawing can possibly be labeled to get at an understanding of what the physical structure must be like.

The vertices of a line drawing are like the pieces of a jigsaw puzzle in that both are limited as to how they can fit together. Selections for adjacent vertex labelings simply cannot require different labels for the line between them.

Given this fact a simple search scheme can work through a drawing, assigning labels to vertices as it goes, taking care that no vertex labeling is assigned that is incompatible with a previous selection at an adjacent vertex. If the search fails without finding a compatible set of labels, then the drawing cannot represent a real structure. If it does find a set of labels, then the successful set or sets of labels yield much information about the structure.

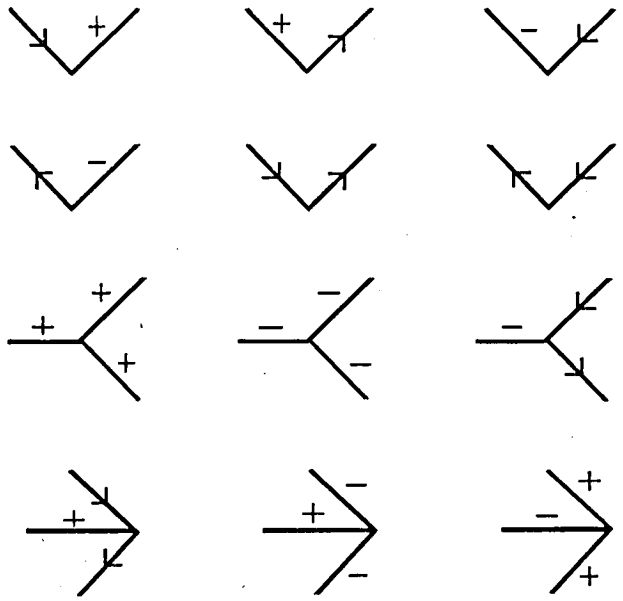


Figure 9. Physically possible configurations of lines around vertices.

Waltz generalized the basic ideas in two fundamental ways. First, he expanded the set of line labels such that each includes much more information about the physical situation. Second, he devised a filtering procedure that converges on the possible interpretations with lightning speed relative to a more obvious depth-first search strategy.

Waltz's labels carry information both about the cause of the line and about the illumination on the two adjacent regions. Figure 10 gives Waltz's eleven allowed line interpretations. The set includes shadows and cracks. The regions beside the line are considered to be either illuminated, shadowed by facing away from the light, or shadowed by another object. These possibilities suggest that the set of legal labels would include $11 \times 3 \times 3 = 99$ entries, but a few simple facts immediately eliminates about half of these. A concave edge may not, for example, have one constituent surface illuminated and the other shadowed.

With this set of labels, body finding is easy! The line labels with arrows as part of their symbol (two, three, four, five, nine, ten and eleven) indicate

places where one body obscures another body or the table. Once Waltz's program finds a compatible set of line labels for a drawing, each body is surrounded by line labels from the arrow class.

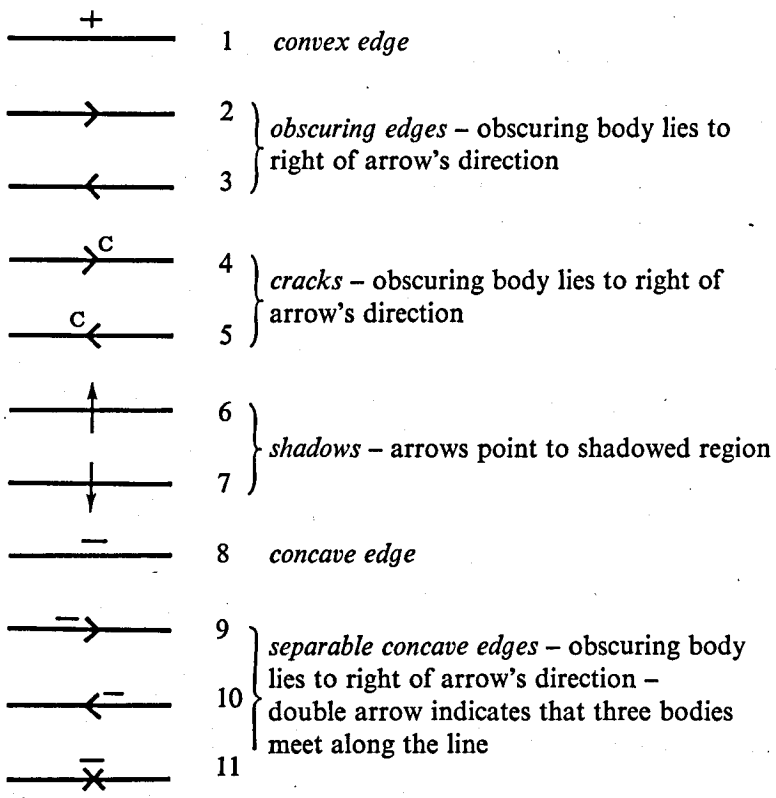


Figure 10. Line interpretations recognized by Waltz's program.

To create his program, Waltz first worked out what vertex configurations are possible with this set of line labels. Figure 11 gives the result. Happily the possible vertex labelings constitute only a tiny fraction of the ways labels can be arrayed around a vertex. The number of possible vertices is large but not unmanageably so.

Increasing the number of legal vertex labelings does not increase the number of interpretations of typical line drawings. This is because a proper increase in descriptive detail strongly constrains the way things may go together. Again the analogy with jigsaw puzzles gives an idea of what is happening: the shapes of pieces constrain how they may fit together, but the colors give still more constraint by adding another dimension of comparison.

Interestingly, the number of ways to label a fork is much larger than the number for an arrow. A single arrow consequently offers more constraint and

less ambiguity than does a fork. This explains why experiments with Guzman's program showed arrows to be more reliable than forks as sources of good links.

	approximate number of combinatorially possible labelings	approximate number of physically possible labelings
	2,500	80
	125,000	70
	125,000	500
	125,000	500
	6×10^6	10
	6×10^6	300
	6×10^6	100
	6×10^6	100
	6×10^6	100
	3×10^8	30

Figure 11. Only a few of the combinatorially possible labelings are physically possible.

Figure 12 shows a fairly complex scene. But with little effort, Waltz's program can sort out the shadow lines and find the correct number of bodies.

What I have discussed of this theory so far is but an *hors d'oeuvre*. Waltz's forthcoming doctoral dissertation has much to say about handling coincidental alignment, finding the approximate orientation of surfaces, and dealing with higher order object relations like support (Waltz 1972b). But without making use of these exciting results, I can comment on how his work fits together with previous ideas on body finding and on shadows.

First of all Waltz's program has a syntactic flavor. The program has a table of possible vertices and on some level can be thought to parse the

scene. But it is essential to understand that this is a program with substantive semantic roots. The table is not an amalgam of the purely *ad hoc* and empirical. It is derived directly from arguments about how real structures can project on to a two dimensional drawing. The resulting label set, together with the program that uses it, can be thought of adequately as a compiled form of those arguments whereby facts about three-dimensional space become constraints on lines and vertices.

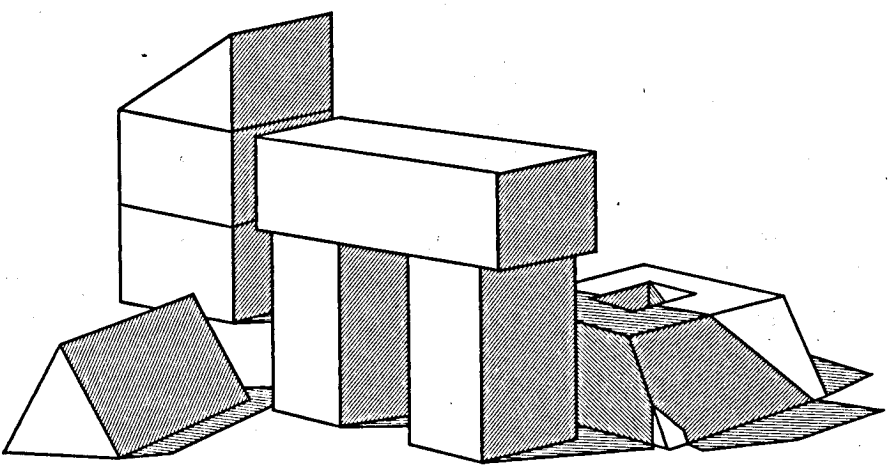


Figure 12. Waltz's program easily handles complicated scenes.

In retrospect, I see Waltz's work as the culmination of a long effort beginning with Guzman and moving through the work of Orban, Ratner, Winston and Huffman and Clowes. Each step built on the ideas and experiments with the previous one, either as a refinement, a reaction, or an explanation. The net result is a tradition moving towards more and better ability to describe and towards more and better theoretical justification behind working programs.

SYSTEM ISSUES

Heterarchy

Waltz's work is part of understanding how line drawings convey information about scenes. This section discusses some of our newer ideas about how to get such understanding into a working system.

At MIT the first success in copying a simple block structure from spare parts involved using a pass-oriented structure like that illustrated in figure 13. The solid lines represent data flow and the dashed lines, control. The executive in this approach is a very simple sequence of subroutine calls, mostly partitioned into one module. The calling up of the action modules is fixed in advance and the order is indifferent to the peculiarities of the scene.

Each action module is charged with augmenting the data it receives according to its labeled speciality.

This kind of organization does not work well. We put it together only to have quickly a vehicle for testing the modules then available. It is often better to have one system working before expending too much effort in arguing about which system is best.

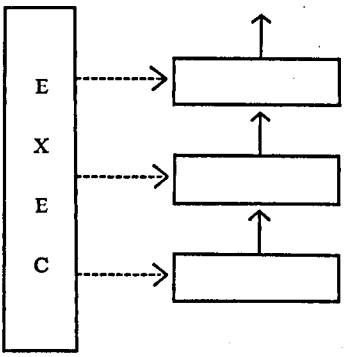


Figure 13. The simple pass-oriented system metaphor.

From this base we have moved toward another style of organization which has come to be called heterarchical (Minsky and Papert 1972). The concept lacks precise definition, but the following are some of the characteristics that we aim for.

1. A complex system should be goal-oriented. Procedures at all levels should be short and associated with some definite goal. Goals should normally be satisfied by invoking a small number of subgoals for other procedures or by directly calling a few primitives. A corollary is that the system should be top down. For the most part nothing should be done unless necessary to accomplish something at a higher level.
2. The executive control should be distributed throughout the system. In a heterarchical system, the modules interact not like a master and slaves but more like a community of experts.
3. Programmers should make as few assumptions as possible about the state the system will be in when a procedure is called. The procedure itself should contain the necessary machinery to set up whatever conditions are required before it can do its job. This is obviously of prime importance when many authors contribute to the system, for they should be able to add knowledge via the new code without completely understanding the rest of the system. In practice this usually works out as a list of goals lying like a preamble near the beginning of a routine. Typically these goals are satisfied by simple reference to the data base, but if not, notes are left as to where help may be found, in the PLANNER (Hewitt 1972) or CONNIVER style (McDermott and Sussman 1972).

4. The system should contain some knowledge of itself. It is not enough to think of executives and primitives. There should be modules that act as critics and complain when something looks suspicious. Others must know how and when the primitives are likely to fail. Communication among these modules should be more colorful than mere flow of data and command. It should include what in human discourse would be called advice, suggestions, remarks, complaints, criticism, questions, answers, lies and conjectures.
5. A system should have facilities for tentative conclusions. The system will detect mistakes as it goes. A conjectured configuration may be found to be unstable or the hand may be led to grasp air. When this happens, we need to know what facts in the data base are most problematical, we need to know how to try to fix things, and we need to know how far ranging the consequences of a change are likely to go.

Graphically such a system looks more like a network of procedures rather than an orderly, immutable sequence. Each procedure is connected to others via potential control-transfer links. In practice which of these links are used depends on the context in which the various procedures are used, the context being the joint product of the system and the problem undergoing analysis.

Note particularly that this arrangement forces us to refine our concept of higher- versus lower-level routines. Now programs normally thought to be low level may very well employ other programs considered high level. The terms no longer indicate the order in which a routine occurs in analysis. Instead a vision system procedure is high or low level according to the sort of data it works with. Line finders that work with intensity points are low level but may certainly on occasion call a stability tester that works with relatively high-level object models.

Finin's environment driven analysis

Our earliest MIT vision system interacted only narrowly and in a predetermined way with its environment. The pass-oriented structure prevents better interaction. But we are now moving toward a different sort of vision system in which the environment controls the analysis. (This idea was prominent in Ernst's very early work (Ernst 1961).)

Readers who find this idea strange should see an exposition of the notion by Simon (1969). He argues that much of what passes as intelligent behavior is in point of fact a happy co-operation between unexpectedly simple algorithms and complex environments. He cites the case of an ant wandering along a beach rift with ant-sized obstacles. The ant's curvacious path might seem to be an insanely complex ritual to someone looking only at a history of it traced on paper. But in fact the humble ant is merely trying to circumvent the beach's obstacles and go home.

Watching the locus of control of our current system as it struggles with a complicated scene is like watching Simon's ant. The up and down, the around and backing off, the use of this method then another, all seem to be

mysterious at first. But, like the ant's, the system's complex behavior is the product of simple algorithms coupled together and driven by the demands of the scene. The remainder of this section discusses some elegant procedures implemented by Finin that illustrate two ways in which the environment influences the MIT vision system (Finin 1972).

The vision system contains a specialist whose task is to determine what we call the skeleton of a brick. A skeleton consists of a set of three lines, one lying along each of the three axes (Finin 1972). Each of the lines in a skeleton must be complete and unobscured so that the dimensions of the brick in question may be determined. Figure 14 shows some of the skeletons found in various situations by this module.

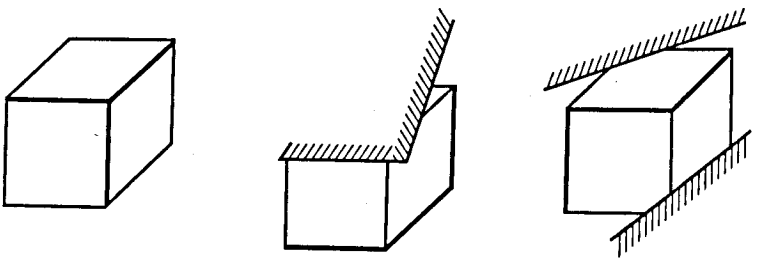


Figure 14. Some skeletons found for bricks.

The only problem with the program lies in the fact that complete skeletons are moderately rare in practice because of heavy obscuring. Even in the simple arch in figure 15a, one object, the left side support, cannot be fully analyzed,

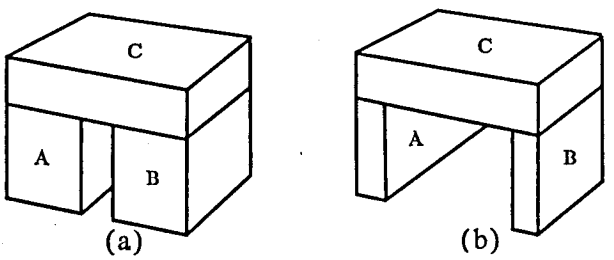


Figure 15. In one case, A's depth is extrapolated from B's. In the other no hypothesis can be confirmed.

lacking as it does a completely exposed line in the depth dimension. But humans have no trouble circumventing this difficulty. Indeed, it generally does not even occur to us that there is a problem because we so naturally assume that the right and left supports have the same dimensions. At this point let us look at the system's internal discourse when working on this scene to better understand how a group-hypothesize-criticize cycle typically works out:

Let me see, what are A's dimensions. First I must identify a skeleton. Oops! We can only get a partial skeleton, two complete lines are there, but only a partial line along the third brick axis. This means I know two dimensions but I have only a lower bound on the third. Let me see if A is part of some group. Oh yes, A and B both support C so they form a group of a sort. Let me therefore hypothesize that A and B are the same and run through my check list to see if there is any reason to doubt that.

Are A and B the same sort of objects?

Yes, Both are bricks.

Are they both oriented the same way?

Yes, that checks out too.

Well, do the observable dimensions match?

Indeed.

Is there any reason to believe the unobservable dimension of A is different from its analogue on B?

No.

OK. Everything seems all right. I will tentatively accept the hypothesis and proceed.

Through this internal dialogue, the machine succeeds in finding all the necessary dimensions for the obscured support in figure 15a. Figure 15b shows how the conflict search can fail at the very last step.

Grouping amounts, of course, to using a great deal of context in scene analysis. We have discussed how the system uses groups to hypothesize properties for the group's members and we should add that the formation of a group is in itself a matter of hypothesis followed by a search for evidence conflicting with the hypothesis. The system now forms group hypotheses from the following configurations, roughly in order of grouping strength:

1. Stacks or rows of objects connected by chains of support or IN-FRONT-OF relations.
2. Objects that serve the same function such as the sides of an arch or the legs of a table.
3. Objects that are close together.
4. Objects that are of the same type.

To test the validity of these hypotheses, the machine makes tests of good membership on the individual elements. It basically performs conformity tests, throwing out anything too unusual. There is a preliminary theory of how this can be done sensibly (Winston 1971). The basic feature of this theory is that it involves not only a measure of how distant a particular element is from the norm, but also of how much deviation from the norm is typical and thus acceptable.

Note that this hypothesis-rooted theory is much different from Gestaltist notions of good groups emerging magically from the set of all possible groups. Critics of artificial intelligence correctly point out the computational implausibility of considering all possible groups, but somehow fail to see the

alternative of using clues to hypothesize a limited number of good candidate groups.

Naturally all of these group-hypothesize-criticize efforts are less likely to work out than are programs which operate through direct observation. It is therefore good to leave data-base notes relating facts both to their degree of certainty and to the programs that found them. Thus an assertion that says a particular brick has such and such a size may well have other assertions describing it as only probable, conjectured from the dimensions of a related brick, and owing the discovered relationship to a particular grouping program. Using such knowledge is as yet only planned, but in preparation we try to refrain from using more than one method in a single program. This makes it easy to describe how a particular assertion was made by simply noting the name of the program that made it.

Visual observation of movement provides another way the environment can influence and control what a vision system thinks about. One of the first successful projects was executed at Stanford (Wickman 1967). The purpose was to align two bricks, one atop the other. The method essentially required the complete construction of a line drawing with subsequent determination of relative position. The Japanese have used a similar approach in placing a block inside a box.

The MIT entry into this area is a little different. We do not require complete recomputation of a scene, as did the Stanford system. The problem is to check the position of an object, which has just been placed, in order to be sure that it lies within some tolerance of the assigned place for it. (In our arm, errors in placement may occasionally be on the order of $\frac{1}{2}''$.)

Rather than recompute a line drawing of the scene to find the object's coordinates, we use our model of where the object should be to direct the eye to selected key regions. In brief, what happens is as follows:

1. The three-dimensional coordinates for selected vertices are determined for the object whose position is to be checked.
2. Then the supposed locations of those vertices on the eye's retina are easily computed.
3. A vertex search using circular scans around each of these supposed vertex positions hill-climbs to a set of actual coordinates for the vertices on the retina (Winston and Lerman 1972). From these retinal coordinates, revised three-dimensional coordinates can be determined, given the altitude of the object.
4. Comparing the object's real and supposed coordinates gives a correction which is then effected by a gentle, wrist-dominated arm action.

The vertex locating program tries to avoid vertices that form alignments with those of other objects already in place. This considerably simplifies the work of the vertex finder. With a bit more work, the program could be made to avoid vertices obscured by the hand, thus allowing performance of the feedback operation more dynamically, without withdrawing the hand.

LEARNING TO IDENTIFY TOY BLOCK STRUCTURES

Learning

This section describes a working computer program which embodies a new theory of learning (Winston 1970). I believe it is unlike previous theories because its basic idea is to understand how concepts can be learned from a few judiciously selected examples. The sequence in figure 16, for example,

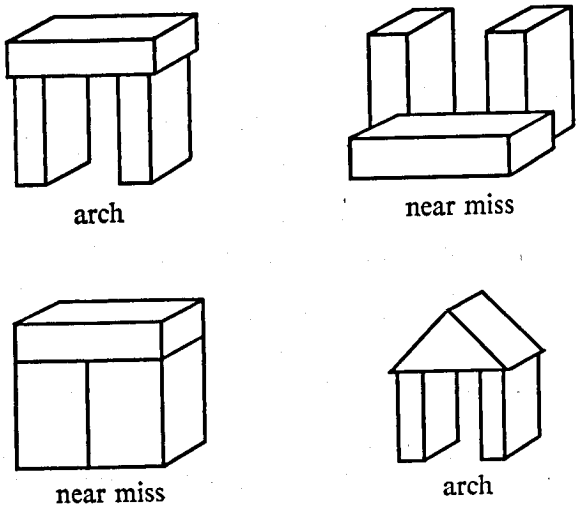


Figure 16. An arch training sequence.

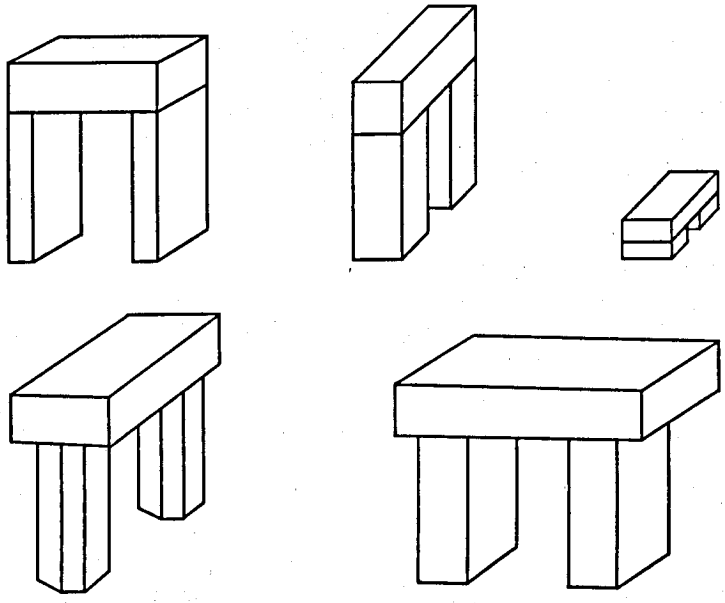


Figure 17. Structures recognized as arches.

generates in the machine an idea of the arch sufficient to handle correctly all the configurations in figure 17 in spite of severe rotations, size changes, proportion changes and changes in viewing angle.

Although no previous theory in the artificial intelligence, psychology, or other literatures can completely account for anything like this competence, the basic ideas are quite simple:

1. If you want to teach a concept, you must first be sure your student, man or machine, can build descriptions adequate to represent that concept.
2. If you want to teach a concept, you should use samples which are a kind of non-example.

The first point on description should be clear. At some level we must have an adequate set of primitive concepts and relations out of which we can assemble interesting concepts at the next higher level, which in turn become the primitives for concepts at a still higher level. The operation of the learning program depends completely on the power of the analysis programs described in the previous sections.

But what is meant by the second claim that one must show the machine not just examples of concepts but something else? First of all, something else means something which is close to being an example but fails to be admissible by way of one or a few crucial deficiencies. I call these samples near-misses. My view is that they are more important to learning than examples and they provide just the right information to teach the machine directly, via a few samples, rather than laboriously and uncertainly through many samples in some kind of reinforcement mode.

The purpose of this learning process is to create in the machine whatever is needed to identify instances of learned concepts. This leads directly to the notion of a model. To be precise, I use the term as follows:

> A model is a proper description augmented by information about which elements of the description are essential and by information about what, if anything, must not be present in examples of the concept.

The description must be a proper description because the descriptive language – the possible relations – must naturally be appropriate to the definitions expected. For this reason one cannot build a model on top of a data base that describes the scene in terms of only vertex coordinates, for such a description is on too low a level. Nor can one build a model on top of a higher-level description that contains only color information, for example, because that information is usually irrelevant to the concept in question.

The key part of the definition of model is the idea that some elements of the description must be underlined as particularly important. Figure 18 shows a training sequence that conveys the idea of the pedestal. The first step is to show the machine a sample of the concept to be learned. From a line drawing, the scene analysis routines produce a hierarchical symbolic description which carries the same sort of information about a scene that a human uses and understands. Blocks are described as bricks or wedges, as

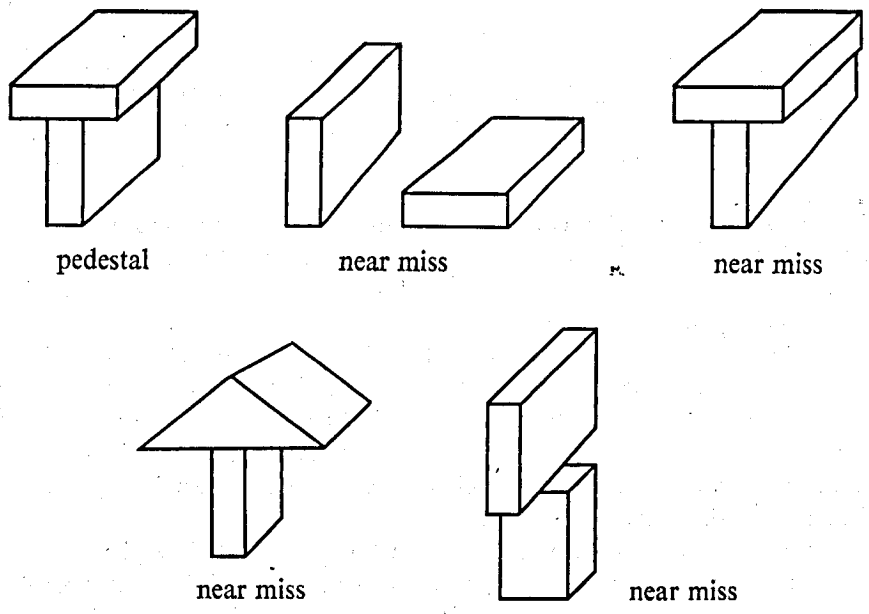


Figure 18. A pedestal training sequence.

standing or lying, and as related to others by relations like IN-FRONT-OF or supports.

This description resides in the data base in the form of list structures, but I present it here as a network of nodes and pointers, the nodes representing

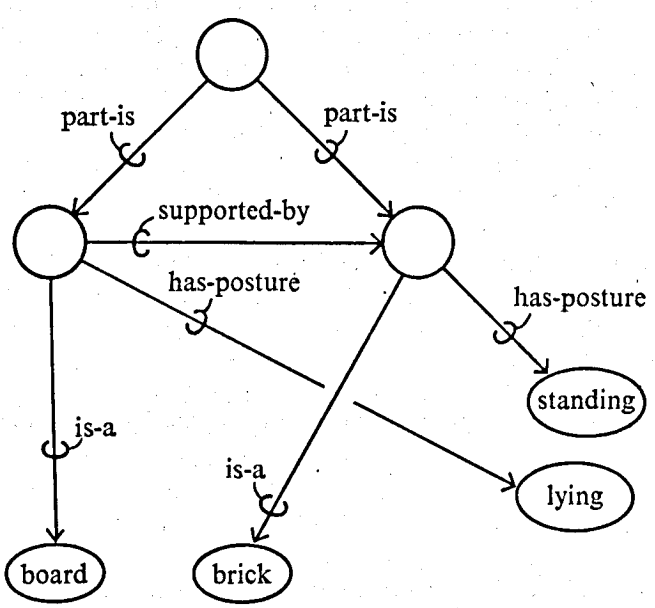


Figure 19. A pedestal description.

objects and the pointers representing relations between them. (See figure 19 where a pedestal network is shown.) In this case, there are relatively few things in the net: one node representing the scene as a whole, and two more for the objects. These are related to each other by the SUPPORTED-BY pointer and to the general knowledge of the net via pointers like IS-A, denoting set membership, and HAS-POSTURE, which leads in one case to standing and in the other to lying.

Now in the pedestal, the support relation is essential – there is no pedestal without it. Similarly the posture and identity of the board and brick must be correct. Therefore, the objective in a teaching sequence is to somehow convey to the machine the essential, emphatic quality of those features. (Later on we will see further examples where some relations become less essential and others are forbidden.)

Returning to figure 18, note that the second sample is a near-miss in which nothing has changed except that the board no longer rests on the standing brick. This is reflected in the description by the absence of a SUPPORTED-BY pointer. It is a simple matter for a description comparison program to detect this missing relation as the only difference between this description and the original one which was an admissible instance. The machine can only conclude, as we would, that the loss of this relation explains why the near-miss fails to qualify as a pedestal. This being the case, the proper action is clear. The machine makes a note that the SUPPORTED-BY relation is essential by replacing the original pointer with MUST-BE-SUPPORTED-BY. Again note that this point is conveyed directly by a single drawing, not by a statistical inference from a boring hoard of trials. Note further that this information is quite high level. It will be discerned in scenes as long as the descriptive routines have the power to analyze that scene. Thus we need not be as concerned about the simple changes that foil older, lower-level learning ideas. Rotations, size dilations and the like are easily handled, given the descriptive power we have in operating programs.

Continuing now with our example, the teacher proceeds to basically strengthen the other relations according to whatever prejudices he has. In this sequence the teacher has chosen to reinforce the pointers which determine that the support is standing and the pointers which similarly determine that the supported object is a lying board. Figure 20 shows the model resulting.

Now that the basic idea is clear, the slightly more complex arch sequence will bring out some further points. The first sample, shown back in figure 16, is an example, as always. From it we generate an initial description as before. The next step is similar to the one taken with the pedestal in that the teacher presents a near-miss with the supported object now removed and resting on the table. But this time not one, but two differences are noticed in the corresponding description networks, as now there are two missing SUPPORTED-BY pointers.

This opens up the big question of what is to be done when more than one

relationship can explain why the near-miss misses. What is needed, of course, is a theory of how to sort out observed differences so that the most important and most likely to be responsible difference can be hypothesized and reacted to.

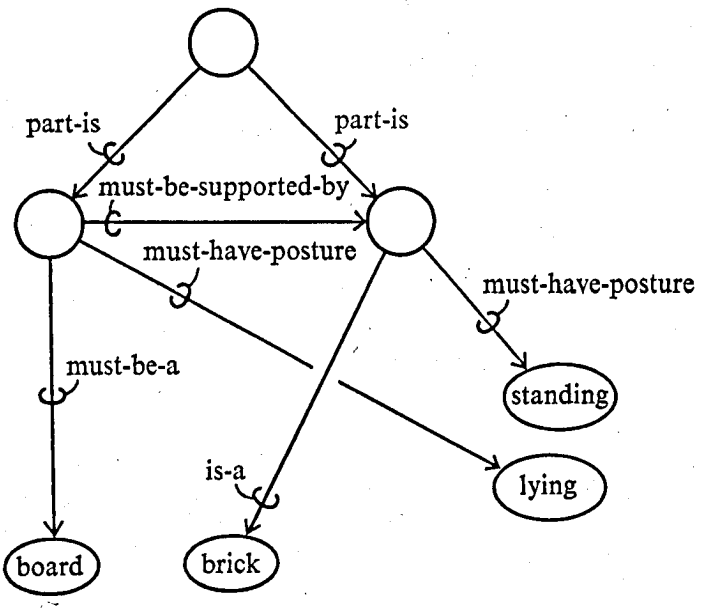


Figure 20. A pedestal model.

The theory itself is somewhat detailed, but it is the exploration of this detail through writing and experimenting with programs that gives the overall theory a crisp substance. Repeated cycles of refinement and testing of a theory, as embodied in a program, is an important part of an emerging artificial intelligence methodology.

Now the results of this approach on the difference ranking module itself include the following points:

First of all, if two differences are observed which are of the same nature and description, then they are assumed to contribute jointly to the failure of the near-miss and both are acted on. This handles the arch case where two support relations were observed to be absent in the near-miss. Since the differences are both of the missing pointer type and since both involve the same SUPPORTED-BY relation, it is deemed heuristically sound to handle them both together as a unit.

Secondly, differences are ranked in order of their distance from the origin of the net. Thus a difference observed in the relationship of two objects is considered more important than a change in the shape of an object's face, which in turn is interpreted as more important than an obscured vertex.

Thirdly, differences at the same level are ranked according to type. In the

current implementation, differences of the missing pointer type are ranked ahead of those where a pointer is added in the near-miss. This is reasonable since dropping a pointer to make a near-miss may well force the introduction of a new pointer. Indeed we have ignored the introduction of a support pointer between the lying brick and the table because the difference resulting from this new pointer is inferior to the difference resulting from the missing pointer. Finally, if two differences are found of the same type on the same level, then some secondary heuristics are used to try to sort them out. Support relations, for example, make more important differences than one expects from TOUCH or LEFT-RIGHT pointers.

Now these factors constitute only a theory of hypothesis formation. The theory does make mistakes, especially if the teacher is poor. I will return to this problem after completing the tour through the arch example. Recall that the machine learned the importance of the support relations. In the next step it learns, somewhat indirectly, about the hole. This is conveyed through the near-miss with the two side supports touching. Now the theory of most important differences reports that two new touch pointers are present in the near-miss, symmetrically indicating that the side supports have moved together. Here, surely the reasonable conclusion is that the new pointers have fouled the concept. The model is therefore refined to have MUST-NOT-TOUCH pointers between the nodes of the side supports. This dissuades identification programs, later described, from ever reporting an arch if such a forbidden relation is in fact present.

Importantly, it is now clear how information of a negative sort is introduced into models. They can contain not only information about what is essential but also information about what sorts of characteristics prevent a sample from being associated with the modeled concept.

So far I have shown examples of emphatic relations, both of the MUST-BE and MUST-NOT-BE type as introduced by near-miss samples. The following is an example of the inductive generalization introduced by the sample with the lying brick replaced by a wedge. Whether to call this a kind of arch or report it as a near-miss depends, of course, on the taste of the machine's instructor. Let us explore the consequence of introducing it as an example, rather than a near-miss.

In terms of the description network comparison, the machine finds an IS-A pointer moved over from brick to wedge. There are, given this observation, a variety of things to do. The simplest is to take the most conservative stance and form a new class, that of the brick or wedge, a kind of superset.

To see what other options are available, look in figure 21 at the descriptions of brick and wedge and the portion of the general knowledge net that relates them together. There various sets are linked together by the A-KIND-OF relationship. From this diagram we see that our first choice was a conservative point on a spectrum whose other end suggests that we move the IS-A pointer over to object, object being the most distant intersection of A-KIND-OF

relations. We choose a conservative position and fix the IS-A pointer to the closest observed intersection, in this case right-prism.

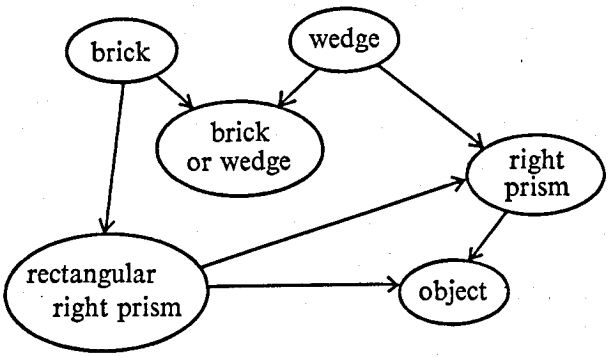


Figure 21. Relations between brick, wedge, and object. All pointers are A-KIND-OF pointers.

Again a hypothesis has to be made, and the hypothesis may well be wrong. In this case it is a question of difference interpretation rather than the question of sorting out the correct difference from many, but the effect is the same. There simply must be mechanisms for detecting errors and correcting them.

Errors are detected when an example refutes a previously made assumption. If the first scene of figure 22 is reported as an example of concept *X* while the second is given as a near-miss, the natural interpretation is that an *X* must be standing. But an alternative interpretation, considered secondary by the ranking program, is that an *X* must not be lying. If a shrewd teacher wishes to force the secondary interpretation, he need only give the tilted brick as an example, for it has no standing pointer and thus is a contradiction to the primary hypothesis. Under these conditions, the system is prepared to back up to try an alternative. As the alternative may also lead to trouble, the process of backup may iterate as a pure depth-first search. One could do

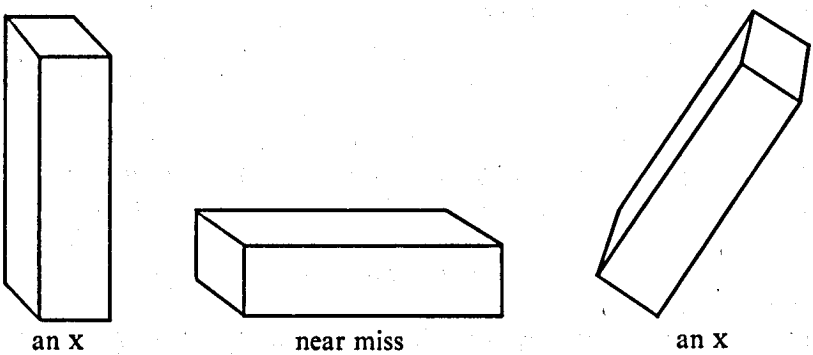


Figure 22. A training sequence that leads to backup.

better by devising a little theory that would back up more intelligently to the decision most likely to have caused the error.

I mentioned just now the role of a shrewd teacher. I regard the dependence on a teacher as a feature of this theory. Too often in the past history of machine learning theory the use of a teacher was considered cheating and mechanisms were instead expected to self organize their way to understanding by way of evolutionary trial and error, or reinforcement, or whatever. This ignores the very real fact that humans as well as machines learn very little without good teaching. The first attempt should be to understand the kind of learning that is at once the most common and the most useful.

It is clear that the system assimilates new models from the teacher and it is in fact dependent on good teaching, but it depends fundamentally on its own good judgement and previously learned idea to understand and disentangle what the teacher has in mind. It must itself deduce what are the salient ideas in the training sequence and it must itself decide on an augmentation of the model which captures those ideas. By carefully limiting the teacher to the presentation of a sequence of samples, low level rote-learning questions are avoided while allowing study of the issues which underly all sorts of meaningful learning, including interesting forms of direct telling.

Identification

Having developed the theory of learning models, I shall say a little about using them in identification. Since this subject both is tangential to the main thrust and is documented elsewhere (Winston 1970), I shall merely give the highlights here.

To begin with, identification is done in a variety of modes, our system already exhibiting the following three:

1. We may present a scene and ask the system to identify it.
2. We may present a scene with several concepts represented and ask the system to identify all of them.
3. We may ask if a given scene contains an instance of something.

Of course, the first mode of identifying a whole scene is the easiest. We simply insist that (1) all the model's MUST-BE-type pointers are present in the scene's description, and (2) all the model's MUST-NOT-BE-type pointers must not be present. For further refinement, we look at all other differences between the model and scene of other than the emphatic variety and judge the firmness of model identification according to their number and type.

When a scene contains many identifiable rows, stacks, or other groups, we must modify the identification program to allow for the possibility that essential relations may be missing because of obscuring objects. The properties of rows and stacks tend to propagate from the most observable member unless there is contrary evidence.

The last task, that of searching a scene for a particular concept is a wide-open question. The method now is to simply feed our network-matching

program both the model and the larger network and hope for the best. If some objects are matched against corresponding parts of the model, their pointers to other extraneous objects are forgotten, and the identification routine is applied. Much remains to be done along the lines of guiding the match contextually to the right part of the scene.

COPYING TOY BLOCK STRUCTURES

I here give a brief description of the system's higher-level functions along with a scenario giving their interaction in a very simple situation. The main purpose is to illustrate the top-down, goal-oriented and environment-dependent flavor of the system. Code samples are available elsewhere (Winston 1971).

Figure 23 shows the possible call paths between some of the programs. Note in particular the network quality that distinguishes the system from the earlier pass-oriented metaphor.

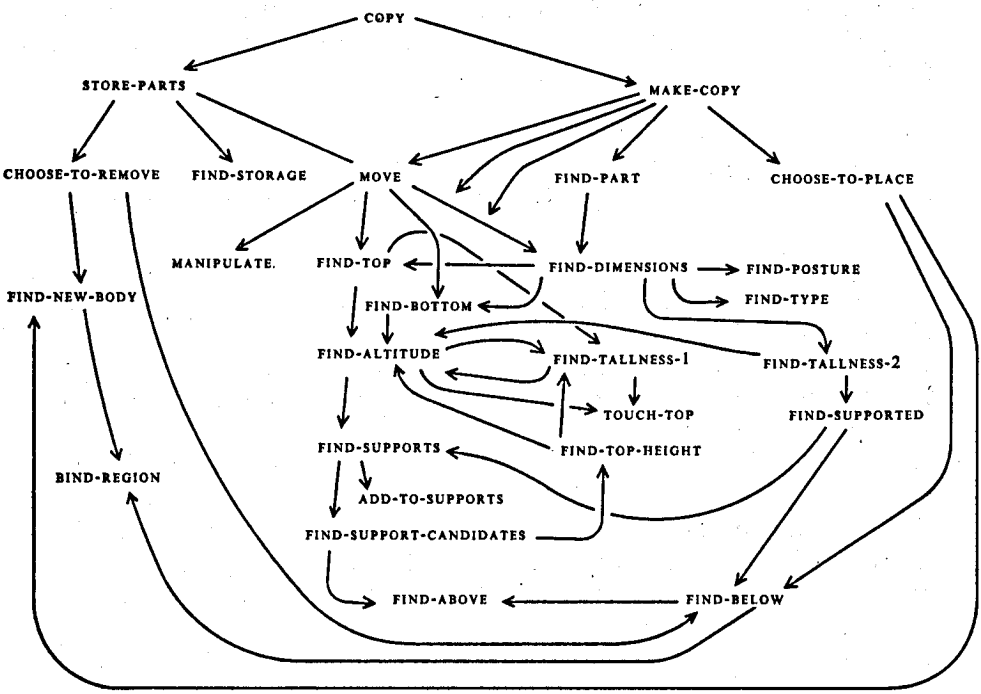


Figure 23. The vision system.

Clarity requires that only a portion of the system be described. In particular, the diagram and the discussion omits the following:

1. A large number of antecedent and erasing programs which keep the blocks world model up to date.

2. A large network of programs which find skeletons and locate lines with particular characteristics.
3. A large network of programs that uses the group-hypothesize-criticize idea to find otherwise inaccessible properties for hidden objects.
4. A network of programs that jiggles an object if the arm errs too much when placing it.

The functions

COPY. As figure 23 shows, COPY simply activates programs that handle the two phases of a copying problem; namely, it calls for the spare parts to be found and put away into the spare parts warehouse area, and it initiates the replication of the new scene.

STORE-PARTS. To disassemble a scene and store it, STORE-PARTS loops through a series of operations. It calls appropriate routines for selecting an object, finding a place for it, and for enacting the movement to storage.

CHOOSE-TO-REMOVE. The first body examined by CHOOSE-TO-REMOVE comes directly from a successful effort to amalgamate some regions into a body using FIND-NEW-BODY. After some body is created, CHOOSE-TO-REMOVE uses FIND-BELOW to make sure it is not underneath something. Frequently, some of the regions surrounding a newly found body are not yet connected to bodies, so FIND-BELOW has a request link to BIND-REGION. (The bodies so found of course, are placed in the data base and are later selected by CHOOSE-TO-REMOVE without appeal to FIND-NEW-BODY.)

FIND-NEW-BODY. This program locates some unattached region and sets BIND-REGION to work on it. BIND-REGION then calls a collection of programs by Eugene Freuder which do a local parse and make assertions of the form:

(R17 IS-A-FACE-OF B2)
(B2 IS-A BODY)

These programs appeal to a complicated network of subroutines that drive line-finding and vertex-finding primitives around the scene looking for complete regions (Winston 1972).

FIND-BELOW. As mentioned, some regions may need parsing before it makes sense to ask if a given object is below something. After assuring that an adjacent region is attached to a body, FIND-BELOW calls the FIND-ABOVE programs to do the work of determining if the body originally in question lies below the object owning that adjacent region.

FIND-ABOVE-1 and FIND-ABOVE-2 and FIND-ABOVE-3. The heuristics implemented in the author's thesis (Winston 1970) and many of those only proposed there are now working in the FIND-ABOVE programs. They naturally have a collection of subordinate programs and a link to BIND-REGION for use in the event an unbodied region is encountered. The assertions made are of the form:

(B3 IS-ABOVE B7)

MOVE. To move an object to its spare parts position, the locations, and dimensions are gathered up. Then MANIPULATE interfaces to the machine language programs driving the arm. After MOVE succeeds, STORE-PARTS makes an assertion of the form:

(B12 IS-A SPAREPART)

FIND-TOP. The first task in making the location calculations is to identify line-drawing coordinates of a block's top. Then FIND-TALLNESS and FIND-ALTITUDE supply other information needed to properly supply the routine that transforms line-drawing coordinates to *xzy* coordinates. Resulting assertions are:

(B1 HAS-DIMENSIONS (2·2 3·1 1·7))
(B1 IS-AT (47·0 −17·0 5·2 0·3))

Where the number lists are of the form:

(⟨smaller *x*-*y* plane dimension⟩
⟨larger⟩
⟨tallness⟩)
(⟨*x* coordinate⟩ ⟨*y*⟩ ⟨*z*⟩ ⟨angle⟩)

The *xyz* coordinates are those of the center of the bottom of the brick and the angle is that of the long *x*-*y* plane axis of the brick with respect to the *x* axis. Two auxiliary programs make assertions of the form:

(B12 HAS-POSTURE STANDING / LYING)

(B7 IS-A CUBE / BRICK / STICK / BOARD)

wherever appropriate.

FIND-DIMENSIONS. This program uses FIND-TOP to get the information necessary to convert drawing coordinates to three-dimensional coordinates. If the top is totally obscured, then it appeals instead to FIND-BOTTOM and FIND-TALLNESS-2.

SKELETON. This identifies connected sets of 3 lines which define the dimensions of a brick (Finin 1971, 1972). It and the programs under it are frequently called to find instances of various types of lines.

FIND-TALLNESS-1. Determining the tallness of a brick requires observation of a complete vertical line belonging to it. FIND-TALLNESS-1 uses some of SKELETON's repertoire of subroutines to find a good vertical. To convert from two-dimensional to three-dimensional coordinates, the altitude of the brick must also be known.

FIND-TALLNESS-2. Another program for tallness looks upward rather than downward. It assumes the altitude of a block can be found but no complete vertical line is present which would give the tallness. It tries to find the altitude of a block above the one in question by touching it with the hand. Subtracting gives the desired tallness.

FIND-ALTITUDE. This determines the height of an object's base primarily by finding its supporting object or objects. If necessary, it will use the arm to try to touch the object's top and then subtract its tallness.
FIND-SUPPORTS. This subroutine uses FIND-SUPPORT-CANDIDATES to collect together those objects that may possibly be supports. FIND-SUPPORT-CANDIDATES decides that a candidate is in fact a support if its top is known to be as high as that of any other support candidate. If the height of a candidate's top is unknown but a lower bound on that height equals the height of known supports, then ADD-TO-SUPPORTS judges it also to be a valid support. At the moment the system has no understanding of gravity.
FIND-STORAGE. Once an object is chosen for removal, FIND-STORAGE checks the warehouse area for an appropriate place to put it.
MAKE-COPY. To make the copy, MAKE-COPY, CHOOSE-TO-PLACE, and FIND-PART replace STORE-PARTS, CHOOSE-TO-REMOVE and FIND-STORAGE. Assertions of the form:

(B12 IS-A SPAREPART)
(B2 IS-A-PART-OF COPY)
(B2 IS-ABOVE B1)

are kept up to date throughout by appropriate routines.
CHOOSE-TO-PLACE. Objects are placed after it is ensured that their supports are already placed.
FIND-PART. The part to be used from the warehouse is selected so as to minimize the difference in dimensions of the matched objects.

A scenario

In what follows the scene in figure 24a provides the spare parts which first must be put away in the warehouse. The scene to be copied is that of figure 24b.

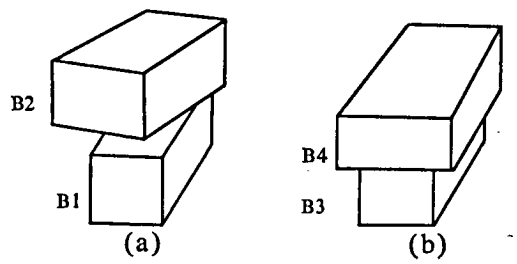


Figure 24. A source of spare parts and a scene to be copied.

COPY begins the activities.
STORE-PARTS begins supervision of disassembly.

CHOOSE-TO-REMOVE
FIND-NEW-BODY
BIND-REGION

CHOOSE-TO-REMOVE parses a few regions together into a body, B1. A great deal of work goes into finding these regions by intelligent driving of low-level line and vertex finding primitives.

FIND-BELOW
  BIND-REGION
  FIND-ABOVE

A check is made to ensure that the body is not below anything. Note that B2 is parsed during this phase as required for the FIND-ABOVE routines. Unfortunately B1 is below B2 and therefore CHOOSE-TO-REMOVE must select an alternative for removal.

FIND-BELOW
  FIND-ABOVE

B2 was found while checking out B1. CHOOSE-TO-REMOVE now notices it in the data base and confirms that it is not below anything.

FIND-STORAGE finds an empty spot in the warehouse.

MOVE initiates the work of finding the location and dimensions of B2.

FIND-TOP
  FIND-ALTITUDE
    FIND-SUPPORTS
      FIND-SUPPORT-CANDIDATES
        FIND-TOP-HEIGHT
          FIND-ALTITUDE
            FIND-SUPPORTS
              FIND-SUPPORT-CANDIDATES
                FIND-TOP-HEIGHT
          FIND-TALLNESS-1
        FIND-TALLNESS-1

FIND-TOP proceeds to nail down location parameters for B2. As indicated by the depth of call, this requires something of a detour as one must first know B2's altitude, which in turn requires some facts about B1. Note that no calls are made to FIND-ABOVE routines during this sequence as those programs previously were used on both B1 and B2 in determining their suitability for removal.

FIND-DIMENSIONS. A call to this program succeeds immediately as the necessary facts for finding dimensions were already found in the course of finding location. Routines establish that B2 is a lying brick.

MANIPULATE executes the necessary motion.

CHOOSE-TO-REMOVE
  FIND-BELOW
FIND-STORAGE

B2 is established as appropriate for transfer to the warehouse. A place is found for it there.

MOVE
  FIND-TOP

FIND-DIMENSIONS
MANIPULATE

The move goes off straightforwardly, as essential facts are in the data base as side effects of previous calculations.

CHOOSE-TO-REMOVE
FIND-NEW-BODY

No more objects are located in the scene. At this point the scene to be copied, figure 24, is placed in front of the eye and analysis proceeds on it.

MAKE-COPY
CHOOSE-TO-PLACE
FIND-NEW-BODY
BIND-REGION

B3 is found.

FIND-BELOW
BIND-REGION
FIND-ABOVE

B3 is established as ready to be copied with a spare part.

FIND-PART
FIND-DIMENSIONS
FIND-TOP

Before a part can be found, B3's dimensions must be found. The first program, FIND-TOP, fails.

FIND-BOTTOM
FIND-ALTITUDE
FIND-SUPPORTS
FIND-SUPPORT-CANDIDATES
FIND-TOP-HEIGHT

FIND-DIMENSIONS tries an alternative for calculating dimensions. It starts by finding the altitude of the bottom.

FIND-TALLNESS-2
FIND-SUPPORTED
FIND-BELOW
FIND-ABOVE
FIND-SUPPORTS
FIND-SUPPORT-CANDIDATES

FIND-TALLNESS-2 discovers B4 is above B3.

FIND-ALTITUDE
TOUCH-TOP
FIND-TALLNESS-1

FIND-ALTITUDE finds B4's altitude by using the hand to touch its top subtracting its tallness. B3's height is found by subtracting B3's altitude from that of B4.

MOVE
MANIPULATE

Moving in a spare part for B3 is now easy. B3's location was found while dealing with its dimensions.

CHOOSE-TO-PLACE
  FIND-BELOW
FIND-PART
  FIND-DIMENSIONS
    FIND-TOP
MOVE
  MANIPULATE

Placing a part for B4 is easy as the essential facts are now already in the data base.

CHOOSE-TO-REMOVE
  FIND-NEW-BODY

No other parts are found in the scene to be copied. Success.

CONCLUDING REMARKS

This essay began with the claim that the study of vision contributes both to artificial intelligence and to a theory of vision. Working with a view toward these purposes has occupied many years of study at MIT and elsewhere on the toy world of simple polyhedra. The progress in semantic rooted scene analysis, learning, and copying have now brought us to a plateau where we expect to spend some time deciding what the next important problems are and where to look for solutions.

The complete system, which occupies on the order of 100,000 thirty-six bit words, is authored by direct contributions in code from over a dozen people. This essay has not summarized, but rather has only hinted at the difficulty and complexity of the problems this group has faced. Many important issues have not been touched on here at all. Line finding, for example, is a task on which everything rests and has by itself occupied more effort than all the other work described here (Roberts 1963, Herskovits and Binford 1970, Griffith 1970, Horn 1971, Shirai 1972).

REFERENCES

Clowes, M. (1971) On seeing things. *Art. Int.*, **2**, 79–116.

Ernst, H. (1961) MH–1, a computer-operated mechanical hand. D.Sc. Dissertation, Department of Electrical Engineering. MIT, Cambridge, Mass.

Finin, T. (1971) Finding the skeleton of a brick. *Vision Flash 19*, Artificial Intelligence Laboratory. Cambridge, Mass: MIT.

Finin, T. (1972) A vision potpourri. *Vision Flash 26*, Artificial Intelligence Laboratory. Cambridge, Mass.: MIT.

Griffith, A. (1970) Computer recognition of prismatic solids. *MAC Technical Report 73*, Project MAC. Cambridge, Mass.: MIT.

Guzman, A. (1968) Computer recognition of three-dimensional objects in a visual scene. *MAC Technical Report 59*, Project MAC. Cambridge, Mass.: MIT.

Herskovits, A. & Binford, T. (1970) On boundary detection. *A.I. Memo 183*, Artificial Intelligence Laboratory. Cambridge, Mass.: MIT.

Hewitt, C. (1972) Description and theoretical analysis (using schemata) of PLANNER: a language for proving theorems and manipulating models in a robot. *A.I. Technical Report 258*, Artificial Intelligence Laboratory. Cambridge, Mass.: MIT.

Horn, B.K.P. (1971) The Binford-Horn line finder. *Vision Flash 16*, Artificial Intelligence Laboratory. Cambridge, Mass.: MIT.

Huffman, D. (1971) Impossible objects as nonsense sentences. *Machine Intelligence 6*, pp. 295–323 (eds Meltzer, B. & Michie, D.). Edinburgh: Edinburgh University Press.

McDermott, D. & Sussman, G. (1972) The CONNIVER Reference Manual. *A.I. Memo 259*, Artificial Intelligence Laboratory. Cambridge, Mass.: MIT.

Minsky, M. & Papert, S. (1972) Progress report. *A.I. Memo 252*, Artificial Intelligence Laboratory. Cambridge, Mass.: MIT.

Orban, R. (1970) Removing shadows in a scene. *A.I. Memo 192*, Artificial Intelligence Laboratory. Cambridge, Mass.: MIT.

Shirai, Y. (1972) A heterarchical program for recognition of polyhedra. *A.I. Memo 263*, Artificial Intelligence Laboratory. Cambridge, Mass.: MIT.

Simon, H. (1969) *The Sciences of the Artificial*. Cambridge, Mass.: MIT Press.

Sussman, G., Winograd, T. & Charniak, E. (1971) MICRO-PLANNER Reference Manual. *A.I. Memo 203A*, Artificial Intelligence Laboratory. Cambridge, Mass.: MIT.

Rattner, M. (1970) Extending Guzman's SEE program. *A.I. Memo 204*, Artificial Intelligence Laboratory. Cambridge, Mass.: MIT.

Roberts, L. (1963) Machine perception of three-dimensional solids. *Technical Report 315*, Lincoln Laboratory. Cambridge, Mass.: MIT.

Waltz, D. (1972) Shedding light on shadows. *Vision Flash 29*, Artificial Intelligence Laboratory. Cambridge, Mass.: MIT.

Waltz, D. (in preparation) Doctoral Dissertation and *A.I. Technical Report* in preparation, Artificial Intelligence Laboratory. Cambridge, Mass.: MIT.

Wichman, W. (1967) Use of optical feedback in the computer control of an arm. *A.I. Memo 56*, Stanford Artificial Intelligence Project. California: Stanford University.

Winston, P.H. (1968) Holes. *A.I. Memo 163*, Artificial Intelligence Laboratory. Cambridge, Mass.: MIT.

Winston, P.H. (1970) Learning structural descriptions from examples. *A.I. Technical Report 231*, Artificial Intelligence Laboratory. Cambridge, Mass.: MIT.

Winston, P. H. (1971) Wandering about the top of the robot. *Vision Flash 15*, Artificial Intelligence Laboratory. Cambridge, Mass: MIT.

Winston, P.H. (1972) Wizard. *Vision Flash 24*, Artificial Intelligence Laboratory. Cambridge, Mass.: MIT.

Winston, P. H. & Lerman, (1972) Circular scan. *Vision Flash 23*, Artificial Intelligence Laboratory. Cambridge, Mass.: MIT.

25

The Mark 1.5 Edinburgh Robot Facility

H. G. Barrow and G. F. Crawford
Department of Machine Intelligence
University of Edinburgh

INTRODUCTION

In May 1971 the Mark 1.5 Edinburgh robot system went on-line as a complete hand-eye system. Two years earlier the Mark 1 device had been connected to the ICL 4130 computer of the Department of Machine Intelligence and Perception. The Mark 1 (Barrow and Salter 1970) had been little more than a semi-mobile T.V. camera, with coarse picture sampling (64×64 points, 16 levels), a limited range of movement over a three-foot diameter circular platform, and a pair of touch-sensitive bumpers. Within eighteen months we had developed suitable basic software, and a 'teachable' program capable of recognizing irregular objects via the T.V. camera (Barrow and Popplestone 1971). However, the restrictions upon movement, the limited range of actions which could modify the 'world', and the shortcomings of the video system, made more advanced work difficult. Plans were therefore laid for the construction of the Mark 2 device.

The Mark 2 robot system will possess moderately sophisticated eyes and a hand which can manipulate objects with a reasonable degree of precision. At the time of writing (May 1972) we have two fixed T.V. cameras and a hand with 6 degrees of freedom; the Mark 2 is intended to possess steerable, controllable T.V. cameras and a hand with 7 degrees of freedom and tactile feedback. The present equipment thus represents a useable system, not yet up to full Mark 2 specification, but considerably more useful than the Mark 1.

DESIGN CONCEPTS

It is important that the complete system should be as self-reliant as possible. If it depends much upon human assistance to pre-process information or to put things right when they go astray, it is all too easy in one's research to avoid the central issues of a problem, and produce a 'solution' which does not survive when confronted by real situations. We therefore wish that the hardware should not impose unreal restrictions upon our research: it should permit

the system to extricate itself from as many predicaments as is reasonably possible.

We had a choice of a number of system configurations: a static hand-eye system, as the Stanford System (McCarthy, Earnest, Reddy, and Vicens 1968); a freely-moving robot, as the SRI device (Nilsson 1969); or even a distributed system with many sensors and effectors only some of which have human counterparts, as HAL 9000 (Clarke 1968).

In the Mark 1 device we implemented a suggestion from Derek Healy that when a robot is complicated and linked to a fixed computer, and its world is simple, it is better to keep the robot still and move the world. From its own point of view, the robot cannot tell whether it or the world moves, and one can simulate free movement in a restricted area. In principle the Mark 1 robot had three degrees of freedom over its plane world (two translations and one rotation). In practice it was limited by the inertia of the platform and the manner in which it was supported.

For the Mark 2, we chose a world platform about 2 metres square with two translational degrees of freedom, driven by servo-systems. Over the platform is a bridge from which are suspended a hand and an eye. Thus the system is like a man at a workbench, who can lean over it and move sideways, but not walk round the other side: the device is free-ranging but always constrained to face in the same direction.

If an object is dropped or knocked over, the hand must be able to restore it to its correct position and orientation. In general, this demands three translational degrees of freedom, three rotational ones, and grasp. If the object has suitable stable states when resting on the platform (for example, a brick), two-stage manipulation can be used to re-orient it with only two degrees of rotational freedom. Because the platform is moveable, two degrees of freedom are detached from the hand; only vertical motion, grasp and two rotations remain.

It is important that the hand possesses suitable tactile sensory equipment. It is valuable to know the force applied to the grasped object, if only to know whether it is grasped at all. Inevitably, errors of interpretation and guidance will result in attempts to push the hand or its contents through objects; such attempts must be detected to be corrected. Fitting objects together ultimately depends upon tactile feedback: the locating surfaces may be hidden from view and previous inspection and dead-reckoning are not sufficiently accurate.

The main sense for the system was chosen to be vision. From a practical point of view, television offers good resolution and there is a well-developed field of technology. We thus selected the T.V. camera as the principal sensory organ for our robot system.

The system possesses a large number of elements to which we wish to give commands or from which we wish to obtain data. There are also many occasions upon which it is desirable to give sequences of commands to two

or more elements in conjunction, for example moving the platform in a straight line, in a particular direction, or closing the hand until resistance is felt. It was therefore decided that all sensors and effectors should be connected directly to a small satellite computer linked to the main time-sharing machine, and a Honeywell 316 was chosen for this purpose. The satellite is responsible for tactical control of the robot system, and it can perform certain low-level sensory analysis relieving the main machine of part of its burden. It is also a dedicated machine using machine-code programs which can run faster than their high-level language counterparts on the time-shared computer. Testing and installation of additional equipment can be performed using only the satellite.

The robot system is available as a standard peripheral of our Multi-POP time-sharing system, and is available to the user at any console. Several packages exist in the Program Library for using various parts of the robot hardware. There is an executive program which gives the user control over the satellite from a time-sharing console, as well as programs for driving the platform and hands, and for reading pictures from the T.V. camera.

THE PERIPHERALS

Figure 1 shows the configuration of the business end of the robot. There is a platform, or 'world', which moves bodily in two directions, and a bridge over it. From the bridge is suspended the 'hand' over the centre of the platform's area of action. At one end of the bridge is a cage which supports the oblique T.V. camera 'eye'. The overhead camera (not shown in this photograph) is normally suspended from the bridge.

Platform

The platform is about 2 m square, and made of light and rigid sandwich board. It is fixed to a carriage which has two degrees of freedom, each driven by a wire drive from a D.C. position servo-system. One motor drives it along the axis of the bridge, the other at right angles to this. The action resembles that of a giant X-Y plotter: the two motions are independent and may be easily programmed to reproduce any trajectory.

Because the position sensing is performed at one end of a drive wire and the load is at the other, there is inevitably some bounce when the table is moved. The overshoot is only a fraction of a centimetre and the settling time is very short. It causes us no problems and could be eliminated by improving frictional damping.

Positional accuracy is determined mainly by the accuracy of the sensing potentiometer ($\simeq 0.5\%$). When the command signal comes from a 12-bit D to A converter, positional increments are about 0.4 mm. When we measured repeatability, we found it to be better than 25 μm.

The speed of movement of the platform is approximately 250 mm per second in each direction, taking about 5 seconds to travel between extremes.

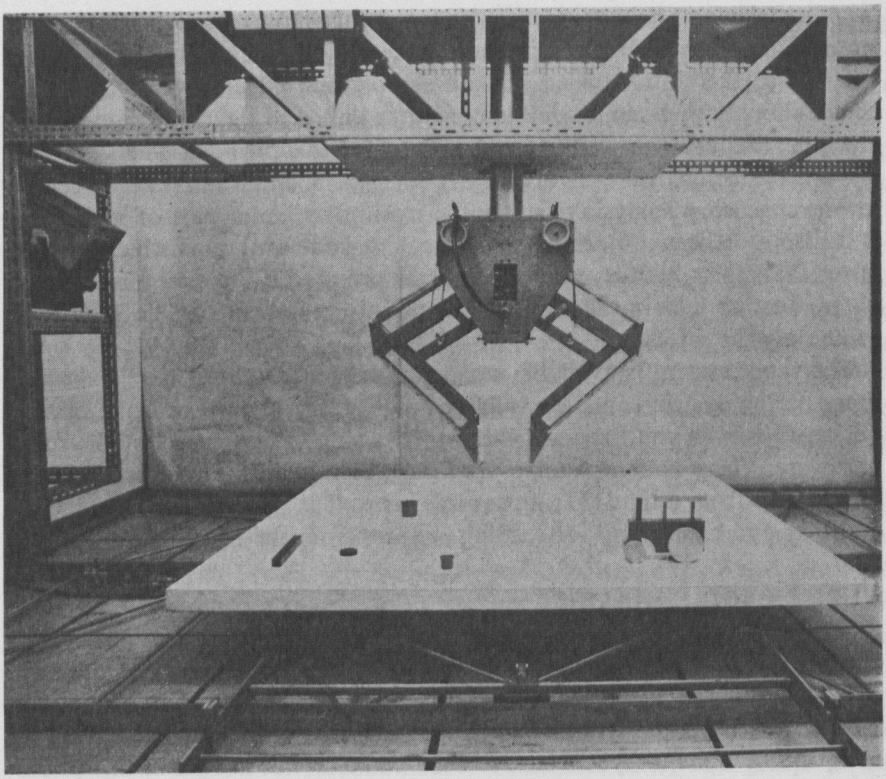


Figure 1

Its acceleration is of the order of 0.1 g so there are few problems of stability of objects piled on it.

Hand

The 'hand' resembles a pair of hands and arms. There are two vertical, parallel plates, or 'palms' which can be driven towards each other to grip an object, and raised together to lift it. Each palm is suspended below the motor housing by a linkage which permits vertical and horizontal movements to remain linear and independent, and which maintains the palm in a fixed orientation throughout its motion. There is a D.C. position servo-system for the horizontal movement of each palm, but one servo drives both palms vertically. The palms can be driven horizontally in opposite directions to grip or release an object, or in the same direction to shift it sideways.

The whole hand assembly is suspended from a column which can rotate about a vertical axis through nearly two turns, driven by a wire drive from a D.C. servo-system. Objects can thus be turned round, as well as lifted.

The forces which can be exerted are quite large, up to 20 kg, both gripping and lifting. We have therefore fitted controllable force limiting circuits to

reduce this to about 5 kg. The palms can grasp objects up to 370 mm across, and be raised to a height of 270 mm. Corresponding positional increments are therefore 0.4 mm and 0.3 mm for 10-bit D to A converters. Rotational range is ± 6 radians with increments of 0.006 rad. (about 0.7 degrees). Each movement takes about 5 seconds to traverse the full extent. The palm surfaces are about 85 mm by 140 mm and are covered with a layer of foam plastic. Objects only 2 mm in diameter have been picked up from the platform under computer control.

T.V. cameras

The oblique camera of the system is mounted in a frame hung below one end of the bridge. It is fixed in orientation and looks along the line of the bridge at the platform about 250 mm in front of the point directly under the hands. It is a commercial closed circuit T.V. camera with a 1″ $f2.8$ lens and the normal C.C.T.V. characteristics of 625 lines, 50 frames/sec. The camera can see an area of platform roughly 1 m deep by 0.5 m broad, that is, only a part of the whole.

A second camera is mounted near the middle of the bridge and looks, via a mirror, vertically downward. It sees an area of platform about 300 mm × 400 mm, and can be used for detailed examination of objects from its special viewpoint. It can also be used in conjunction with a projector which casts a stripe of light via a second mirror obliquely on to the platform. From the appearance of the stripe, the 3-dimensional properties of objects can be measured (Popplestone 1971).

A computer-controlled reed relay switch enables either camera to be selected and connected to the video sampler unit.

Video Sampler

The device which digitizes T.V. picture information is very simple. The computer sets the co-ordinates of a point in the picture; about 750 × 290 sample points can usefully be used. When the T.V. scan reaches that point, the video voltage waveform is sampled and digitized. The resulting number is read by the computer: it ranges from 0 (black) to 255 on a logarithmic scale.

The Z (vertical) position is found by counting the T.V. lines, and the X (horizontal) position by counting ticks of a clock. A normal T.V. picture is transmitted by sending all the odd lines in one field, and then all the even lines in the next. The two fields together comprise one frame of 625 lines and 25 frames are transmitted each second. We have not yet bothered to discriminate between odd and even fields, so the vertical resolution is limited to only 312 lines (of which some are outside the actual picture area). We normally consider only every other point across the picture, yielding a matrix of approximately 400 × 300 points. Because the aspect ratio of the T.V. picture is 4:3, they are distributed in a square array. It should be noted that the

resolution of a good commercial camera is limited by its optics and electronics to about 400 lines.

When the T.V. scan reaches the selected point in the picture, the average amplitude of the video waveform over a 90 nano-second window is digitized on a logarithmic scale. Absolute light level is not important, but relationships between light levels are. Moreover, differences of brightness are affected by many variables, whereas ratios of brightness depend primarily upon the properties of the surfaces concerned, and less on their environment. Experience with the Mark 1, which had linear encoding, showed it had a paucity of discriminable levels in dark areas, and a surfeit in light areas. Logarithmic encoding was thus an attractive proposition.

The digitizing unit must be primed by the computer. When the scan reaches the selected point, the video signal amplitude is digitized and held in a register until the computer reads it. The action of reading also reprimes the sampler.

Consideration of the mode of use of the sampler, and the limitations of the computer to which it is connected, led us to conclude that the best compromise was to take only one sample per T.V. line. Accordingly, digitization is permitted to take up to 40 μs. The satellite computer can prime the sampler sufficiently rapidly to keep up with the T.V. scan, thus taking a complete column of samples each frame. To read 64 columns takes 1.28 seconds, too slow for some purposes but adequate for many robot activities, which are computer-bound anyway.

Some T.V. cameras have A.C. coupled outputs; it is therefore necessary to restore the waveform's D.C. component before passing it on to the sampler. This is done by a module which clamps the waveform to a reference level when it is known to represent black. Originally, this point was taken to be the 'back porch', just after the line synchronizing pulse, which is 'electronic black'. However, the camera used had automatic level adjustments which modified the 'real' black level. A strip of black felt was therefore stuck over the image plane to provide a strip of real black at one side of the picture, and the circuit adjusted to clamp on this real black level.

The pre-processing module also extracts line and frame synchronizing pulses from the video waveform and emits them as TTL logic levels for driving the X and Z co-ordinate units. From these in turn it receives appropriate pulses to mix with the original video waveform so that a pair of white cross-wires is generated to indicate on a T.V. monitor the picture point currently being sampled.

Other peripherals

In addition to the special hand and eye equipment described above, we have also connected a number of general purpose peripherals to the robot system. These include a Tektronix type 611 storage tube display, a 16-channel, 10-bit A to D converter, and reed relay inputs and outputs.

SATELLITE COMPUTER

Building the robot system round a small satellite computer was an attractive proposition because the satellite can fulfil tactical control functions. It can monitor sensors while performing actions, can co-ordinate several simultaneous movements, or it can perform low level picture processing. Such activities are time-consuming, but demand only simple programs: they are better performed by a small dedicated machine than a large time-sharing computer.

The machine we chose is a Honeywell 316, a 16-bit small computer, with 8K of memory. The decision was predominantly an economic one and, at the time it was made, the market was comparatively restricted. There is now a much wider choice among small computers, many of which have more exciting instruction sets, and are comparatively cheap.

All sensors and effectors of the robot are interfaced to the satellite machine, which is linked to the time-sharing machine. It was intended to give the satellite as much autonomy as possible, and to avoid making the relationship between the two machines heavily one-sided. The link is therefore two independent data paths which can operate simultaneously in opposite directions.

Interfacing

It was fairly clear that most peripheral devices would require some form of data buffering. Computer data transfers are usually synchronous; the information is only presented for a few fleeting micro-seconds. But devices need to transfer asynchronously: D to A converters need to maintain their outputs until they are told to change, and picture samplers must hold their data until the computer is ready to accept it. We also felt that in an experimental situation, it is desirable to keep one's devices as simple and general-purpose as possible, so that they may readily be modified or moved around.

We have based all our peripheral construction (including the link between the computers) upon two units, the input and output buffers. An output buffer contains a 16-bit register, and is connected as a normal peripheral to the I/O bus of the computer, which can transfer single words into its register. It presents a rather simple appearance at a socket into which the external device is connected. An input buffer unit is likewise a 16-bit register connected as a peripheral of the computer, which can read its contents. It too has a socket which presents simple interface conventions to the outside world.

Buffer interface conventions

In designing this interface we drew upon the ideas behind the British Standard Interface (British Standard 4421, 1969). Consider two units, *A* and *B* (figure 2), and a transfer of information between them from *A* to *B*. For example, *A* may be an output buffer and *B* a D to A converter, or *B* may be an input buffer and *A* a video sampler. The transfer takes place as follows:

Unit *A* places the data on the parallel data bus and then sets READY to be TRUE.
Unit *B* reads the data and then sets $\overline{\text{ACCEPTED}}$ to be FALSE
Unit *A* sets READY to be FALSE and removes the data

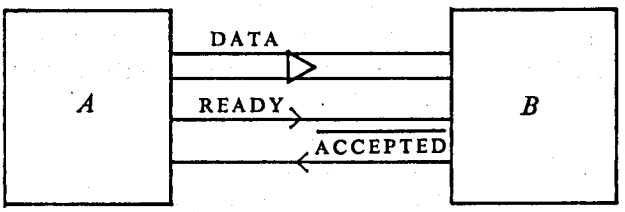


Figure 2

In addition to the READY and $\overline{\text{ACCEPTED}}$ lines, there is one additional RESET line for each unit. The RESET signal sent by *A* to *B* sets *B* into a state in which it is ready to receive data from *A*. The RESET signal from *B* to *A* sets *A* into the state in which it prepares the next word of data. *A* and/or *B* may be reset by external means (the computer or a push-button), and when this occurs a RESET signal is sent to the other unit.

Unlike the B.S. Interface, the data path is 16-bits wide and its signal levels are TTL logic levels: transfers can occur at full TTL speed.

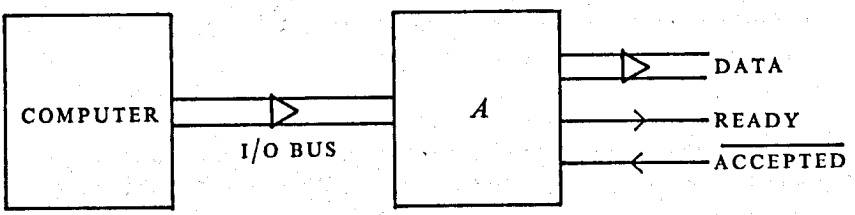


Figure 3

Consider now a single output buffer (figure 3). The unit contains a register to hold data, and a bistable which represents its state, EMPTY or FULL. When it is EMPTY, the unit ignores its mate and responds only to the computer. The computer can transfer a word to *A*'s register, which also sets its state to FULL. *A* then directs its attention to its mate and ignores the computer, which cannot now perform another transfer. READY becomes TRUE and unit *B* can read data from *A*. When $\overline{\text{ACCEPTED}}$ goes FALSE, *A*'s state becomes EMPTY and it can respond to the computer once more.

Consider an input buffer (figure 4). Unit *B* also possesses a register and a state bistable. When *B* is EMPTY it can participate in a transfer with *A*, and ignore the computer. On successful transfer, *B* sets $\overline{\text{ACCEPTED}}$ FALSE and becomes FULL. It now ignores *A* and can be read by the computer. When the computer reads the contents of *B*'s register, *B* becomes EMPTY once more.

At any stage an input or output buffer can be interrogated by the computer to determine whether it is FULL or EMPTY. A FULL input buffer or an

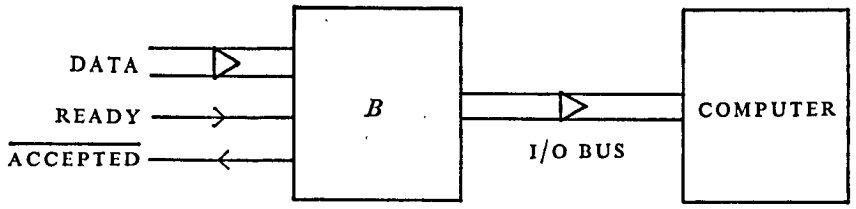


Figure 4

EMPTY output buffer can potentially interrupt the computer. Such interrupts can be permitted or inhibited for each buffer independently. Each buffer can also be reset by the computer to the EMPTY state.

Linking computers

Clearly, in accordance with the above description, any input buffer can be connected to any output buffer. If both are connected to the same computer, little is gained, but if they are connected to different computers, it becomes possible to transfer data from one machine to the other. Two such data paths in opposite directions give fully independent channels of communication.

We linked the H316 and ICL 4130 computers by designing input and output buffers for each, and then connecting them together. In each path are two registers: a specially built path need only contain one register, but it is much simpler to construct a link from standard buffer units.

Devices

Peripheral devices for the robot system have all been designed to conform to the standard interface conventions. They are connected to output and/or input buffers, and not directly to the I/O bus of the computer. This has several practical advantages; we have removed a complex piece of logic from the device, the design of which is simplified; buffer units can be mass-produced on printed circuits to improve reliability; devices and buffers can be readily swapped around to aid fault analysis; devices are machine-independent, only two units (the buffers) need to be re-designed if a different computer is to be used.

Modularity has also been maintained as far as devices are concerned. The same D to A converter module can be used to drive any servo-system, or the visual display, because we have endeavoured to keep any peculiarities confined as closely as possible to the equipment concerned. This philosophy has undoubtedly aided us on many occasions.

TOWARDS MARK 2

As we remarked earlier, in order to be reasonably self-reliant, the robot needs the ability to turn objects over, as well as to turn them round. We are already working upon the addition of such a facility. We intend to equip each palm with a motor-driven plate which will lie parallel to the present palm and will

rotate about an axis normal to its surface. The two plates must turn in synchrony, with as little relative movement as possible to avoid twisting the object.

When it attains full Mark 2 specification, the hand will be able to hold most objects in any orientation at any point in its workspace, though it may be necessary to perform a two-stage manipulation to do so. Clearly, however, it will still not possess enough degrees of freedom to apply an arbitrary rotation to the object in its grasp, for example, turning a key in a normal keyhole, or a doorhandle. When we wish to tackle such tasks, we shall be thinking about a Mark 3 device.

The Mark 1.5 hand does not yet possess any tactile sense. At present we use vision to determine whether an object has been successfully picked up, but in cases of delicate manipulation, visual observation is not sufficient and touch is essential.

We intend to add force transducers to each wrist so that we can deduce such information as strength of grip, the weight of objects held, and whether the hand or its contents have collided with something. While being a good deal less sophisticated than that of the human hand, this tactile sensory system will enable us to study a variety of manipulative tasks, like putting rods into holes.

The robot's visual system is to be upgraded also. Despite the reasonably high resolution of the T.V. sampling system, it is ultimately limited by the resolution of the vidicon, and the focal length of the lens. We need a wide-angle lens in order to see a usefully large area of platform, but such a lens does not permit adequately detailed observation of a single object. We need to employ either a lens turret, or a zoom lens, under computer control. When a narrow angle lens is used, its small field of view demands pan and tilt movements of the camera (we cannot always rely on platform movements, for example, when the object is held) as well as focusing.

We need to have tighter control over the internal operation of the camera: automatic control circuits cause shifts of level as well as changes in gain. Tenenbaum (1970) has shown the advantages to be obtained from active control of the functions of the T.V. camera. He also worked on the analysis of colour. We hope to fit a colour wheel.

SOFTWARE

A number of program packages are available in the program library for using the equipment from a time-shared console attached to the main computer. The language used is POP-2 (Burstall, Collins and Popplestone 1971). Core occupancies refer to 24-bit words of ICL 4130 core.

LOADER

The most fundamental program for the H316 satellite is a 16-word loader which resides permanently in a write-protected area of store. It reads 8-bit bytes from the link and assembles them as 16-bit words, representing address

or contents alternately. It stores the contents in the corresponding address repetitively until it reads zero as the address, in which case it jumps to the location specified by the following word.

LIB HONEY

This is a POP-2 program which assembles H316 symbolic machine code programs. It permits the user to define macros which have the full power of POP-2. Thus, it is possible to extend or re-define the assembly language at will. Facilities are also provided (when used in conjunction with [LIB LINK EXEC]) for accessing and modifying H316 core locations by symbolic names. (4K of POP-2)

LIB LINK EXEC

This library package provides communication between the satellite and main machines. From a Multi-POP console, the user can control the satellite, send data to it, read core locations, start up programs, check on their progress and debug them. The package consists of a set of POP-2 routines, and a set of H316 machine code routines to handle transfers between the two machines, and carry out the user's requests.

As far as possible, the capabilities of the two machines are equal. In particular, the two may operate autonomously, both being able to initiate requests and each responding to the other. The H316 executive is an interrupt handling program. A message from the ICL 4130 arriving via the link causes an interrupt entry to the executive, which saves the volatile registers on a pushdown stack and then interprets and obeys the message. Such interrupts can be nested. The user can employ the interrupt handling routines for his own purposes.

Multi-POP does not provide the user with interrupt facilities, so the POP-2 link executive routines are not interrupt driven. The user can check whether the H316 is attempting to send a message and if so read it and interpret it. The H316 has the ability to request the call of any POP-2 routine of a specified set (which can be as large as desired). Thus an error routine can be called, instead of the printing routine expected by the user, or flags can be set or cleared to permit linked parallel processing.

Some of the POP-2 routines provided for controlling the satellite are as follows:

HSET();	Initializes the H316 to an idle loop with interrupts enabled and registers cleared.
HREADY();	Interrupts current program to await further commands. (All commands could be given while program is still running.)
HGOON();	Return to interrupted program.
HLOC(N);	Reads contents of H316 location N, or updates contents, for example, PRINT(HLOC(1024)); or 3→HLOC(1000);

HREG(N);	Reads from or writes to a volatile register (like HLOC) N=0 modifier register 1 main accumulator 2 extension accumulator 5 program counter, etc.
HDEV(N);	Reads from or writes to H316 peripheral with device address N.
HAPPLY(ARG1, . . ., ARGN, SUBR, N);	Causes H316 to jump to specified subroutine with the given arguments. (N specifies how many arguments there are.)
HBRK(LOC);	Sets a break point at the specified location. When a breakpoint is encountered, the location and accumulator contents are printed on the POP-2 console, the executive is entered and awaits instruction until HGOON(); is given.
DTOH(FILENAME);	Loads a machine code file into the satellite.
HTOD(N, LOC, FILENAME);	Dumps N locations of satellite core, starting at LOC on to disc under the given file name.

The link executive programs are written to be extensible. It is easy to provide H316 routines called from the ICL 4130, as extra executive facilities. The executive functions give a solid base upon which to build further. (2K of POP-2, 0.5K of H316 core)

LIB ROBOT MOVE

This is a package for driving the motors of the robot. (It is written as an extension of [LIB LINK EXEC]).

The problem it is designed to overcome is that of ensuring smooth movements from point to point when more than one motor is involved, for example, moving the platform in a straight line. If the new destinations were output to several servo channels simultaneously, all the motors would run at full speed, and get to their new positions at different times, giving a peculiar trajectory. An H316 program drives all the motors involved in small steps at the rate of the slowest so that the trajectory generated is a straight line (except for the rotations). At the end of the movement, an appropriate pause is made to allow any mechanical oscillations to die away.

A general-purpose function is provided for setting some outputs instantaneously, and tracking others smoothly. It is usually more convenient to split up movements into lifting, turning, etc., and a number of functions are provided, as shown below.

A co-ordinate system is embedded in the platform with its origin at the centre, X and Y in its plane and Z vertically upwards. Distances are measured in centimetres, angles are measured in radians. The position of the platform (or rather of the robot over the platform) is specified by the location directly under the centre of the hand.

MOVETO(X, Y); positions the platform point (X, Y, 0) under the hand.
MOVEBY(DX, DY); makes an incremental movement (DX, DY, 0).
RAISETO(Z); raises (or lowers) the hand to Z cms. above the platform.
RAISEBY(DZ); incremental raise.
GRASPTO(W); sets width between palms to W cm.
GRASPBY(DW); incremental grasp.
TURNTO(THETA); rotates hand about vertical axis.
TURNBY(DTHETA); incremental turn.
TILTTO(PHI); rotates palms.
TILTBY(DPHI); incremental palm rotation.
} to be added shortly.

Current positions are kept available in global variables XNOW, YNOW, THETANOW, etc. (0.75K of POP-2, 0.12K of H316 core)

LIB PICTURE PACK TWO

The total number of sampleable points (over 200,000) inside the T.V. picture makes it impractical to read and store them all. It is necessary to be able to read a subset of them. As mentioned earlier, the sampling is performed for picture points one by one; the computer must do any necessary scanning. The satellite is therefore responsible for scanning the appropriate subset of picture points (which it can do more efficiently and faster than the main machine) and storing the samples (packed two to each 16-bit word) in its core. The data are then transmitted, *en bloc*, to the main machine, where they are loaded into an array of 8-bit elements.

The library package consists of two parts, the picture taking routines in H316 machine code, written as an extension to the executive, and their POP-2 counterparts. The following POP-2 functions are available:

PICINT(X, Z); causes a single picture point, (X, Z) to be sampled, and the brightness is returned as the result.
SETPIC(X, Z); does not cause a sample to be taken, but merely sets the indicating cross wires on the T.V. monitor to point (X, Z). (Useful for diagnostic purposes, etc.)
LOADPIC takes as its arguments a specification of a rectangular window in the picture, in terms of upper and lower values of X and Z co-ordinates, together with increments in each. For example, X runs from XLO to XHI in steps of XINC. Normally the increments are chosen to give a square array of sampled picture points (that is, XINC:ZINC=2:1). When LOADPIC exits, the global variable RETINA contains the required array of brightness values. There are several other global variables which are also set by LOADPIC, and contain information such as the position of the T.V. camera when the picture was taken.

DISPLAY is a general purpose routine for printing pictures. It is given a function to be displayed (for example, an array), the ranges of x and z over which its values are to be printed and a specification of which characters are to be printed for the various values, overprinting if desired (2.8K of POP-2, 0·2K of H316 core).

LIB PERSPECTIVE

There are four co-ordinate systems associated with the robot (figure 5):
Absolute – a 3-D Cartesian system fixed to the platform.
Relative – a 3-D Cartesian system fixed to the robot.
Picture – a 2-D Cartesian system, specifying a picture point.
Retina – the 2-D system defined by the indices of the array RETINA.

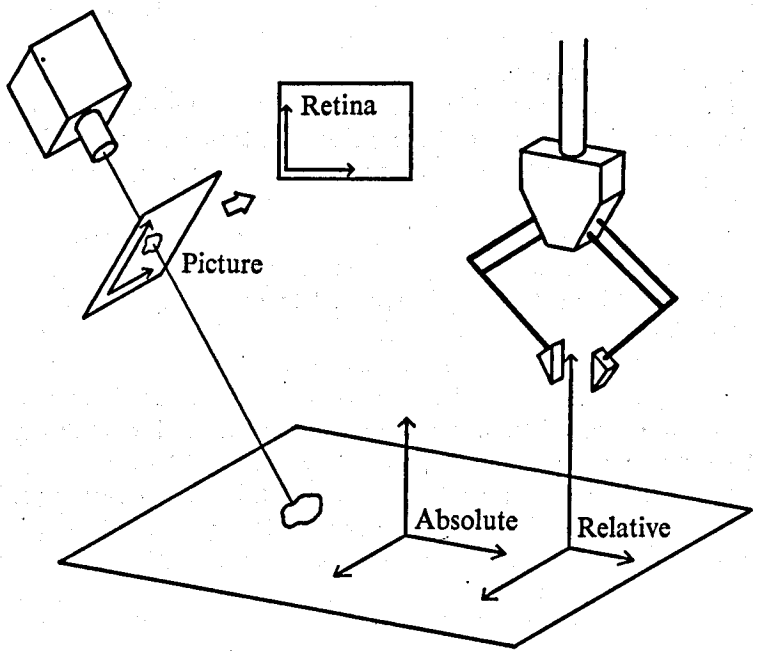


Figure 5

Clearly, we wish to be able to relate a point of the array to points in the absolute system, to determine where objects are from an analysis of the picture. This package provides a set of routines for conversions between systems. Some of the available functions are:

ABSTOREL	RELTOABS
RELTOPIC	PICTOREL
PICTORET	RETTOPIC

One point needs explaining. Absolute and Relative positions are represented by triples, but picture and retina points are duples. RELTOPIC and PICTOREL relate triples to duples and vice versa. These transformations

are performed by the well-known method of matrix operations upon homogeneous co-ordinates, the matrices being defined for transformations between the picture plane and the platform plane. RELTOPIC performs a two-stage process of projecting the relative position on to the platform along the line of sight, and then performing the matrix application. It is a many-one mapping. PICTOREL is the only one-many mapping. It produces as its result a point in the platform plane which corresponds to the given picture point. The platform point, together with the camera position, specifies a ray on which the point of interest must lie. (0.5K of POP-2)

LIB VISUAL

Provides a debugging facility. An array in core can be bugged so that whenever it is accessed, the T.V. monitor cross wires are moved to a corresponding point in the picture. It enables the user to see which areas of the picture the computer is considering. (0.12K of POP-2)

LIB REGION FINDER

The region finding process of Barrow and Popplestone (1971) has been found to be of considerable use, and has therefore been incorporated into the program library. As well as a procedure for this process, a simple and fast routine is provided for finding well-defined regions, for example, dark objects against a light background. (4K of POP-2)

Software development

Naturally, as hardware is extended and improved, the software which uses it is also extended and improved. Periodically, however, it may become necessary to re-think and re-write. For example, the addition of a second T.V. camera, not originally anticipated, demands some way of switching between data associated with particular cameras; the perspective transformation depends on the camera in use. At the time of writing, an improved set of library programs has been written, but not yet issued for general use. While possessing similar facilities to their predecessors, the scope of the programs is somewhat wider.

One of our aims is to provide an ever-extending library of facilities, with abilities of higher and higher order becoming routinely available. Clearly, as we proceed, such facilities become more interdependent and their development demands systematization of the knowledge we embodied in them, as well a re-appraisal of what has gone before in the light of experimental experience.

Acknowledgements

A number of people have contributed to the development of the equipment and programs described in this paper. A major contribution was made by Mr S.H.Salter of the Bionics Research Laboratories, University of Edinburgh. Steve Salter performed most of the design and construction of the mechanics and associated servo-systems, and also of the T.V. sampling system. Ken Turner, of the Department of Machine Intelligence, designed and built the prototype buffer units, and helped in the development of some of

the H316 programs. We wish to acknowledge general financial support given by the Science Research Council, and also by the GPO Telecommunications Headquarters in the form of a contract in connection with the mechanization of parcel handling.

REFERENCES

Barrow, H.G. (1971) LIB LINK EXEC. *Multi-POP Program Library.* Edinburgh: School of Artificial Intelligence.

Barrow, H.G. (1971) LIB PERSPECTIVE. *Multi-POP Program Library.* Edinburgh: School of Artificial Intelligence.

Barrow, H.G. (1971) LIB PICTURE PACK TWO. *Multi-POP Program Library.* Edinburgh: School of Artificial Intelligence.

Barrow, H.G. (1971) LIB VISUAL. *Multi-POP Program Library.* Edinburgh: School of Artificial Intelligence.

Barrow, H.G. & Popplestone, R.J. (1971) Relational descriptions in picture processing. *Machine Intelligence 6*, pp. 377–96 (eds Meltzer, B. & Michie, D.). Edinburgh: Edinburgh University Press.

Barrow, H.G. & Popplestone, R.J. (1972) LIB REGION FINDER. *Multi-POP Program Library.* Edinburgh: School of Artificial Intelligence.

Barrow, H.G. & Salter, S.H. (1970) Design of low-cost equipment for cognitive robot research. *Machine Intelligence 5*, pp. 555–66 (eds Meltzer, B. & Michie, D.). Edinburgh: Edinburgh University Press.

British Standard 4421 (1969) *Specification for digital input/output interface for data collection systems.* London: British Standards Institute.

Burstall, R.M., Collins, J.S. and Popplestone, R.J. (1971) *Programming in POP–2.* Edinburgh: Edinburgh University Press.

Clarke, A.C. (1968) *2001: A Space Odyssey.* London: Arrow.

Dunn, R.D. (1971) LIB LINK-BASED KEY-IN LOADER. *Multi-POP Program Library.* Edinburgh: School of Artificial Intelligence.

Dunn, R.D. (1971) LIB ROBOT MOVE. *Multi-POP Program Library.* Edinburgh: School of Artificial Intelligence.

McCarthy, J., Earnest, L.D., Reddy, D.R. & Vicens, P.J. (1968) A computer with hands, eyes, and ears. *Proc. 1968 Fall Joint Comput. Conf.*, **33**, pp. 329–38. Washington DC: Thompson Book Company.

Nilsson, N.J. (1969) A Mobile Automaton: an application of artificial intelligence techniques. *Proc. of Int. Joint Conf. on Art. Int.*, pp. 509–20. Washington DC.

Popplestone, R.J. (1971) *How Freddy could pack parcels.* (Privately circulated.)

Salter, S.H. (1972) *Arms and the robot.* (Privately circulated.)

Tenenbaum, J.M. (1970) Accommodation in computer vision. *Stanford Artificial Intelligence Project Memo, AIM–134.*

INDEX

Abe 307, 323
ACE 3, 4, 6, 10, 13
admissibility 176, 178, 180, 183-4
Ajenstat 63
Akita 376
ALGOL-60 23, 25, 51-2, 103
Allen 191
Alpert 245
analytic engine 14-15, 20
Anderson 86
Andrews 132
Arbib 42
Atlas 7

Babbage 6, 7, 9, 10, 13-16
backtracking 115, 135, 207-8, 335
Baker 343
Barrow 465-80
Bayes' error 369-70, 373-5
behaviour 389-90
Bennett 78
Bernays 92
Bhattacharyya bound 369-70, 373-4, 376
Biermann 341, 343, 350-2
Binford 462
Bitzer 245
Bodner 248
Bohr 389
Bolliet 337
Boyer 101-16, 155, 157, 162, 194
branch and bound algorithms 181
Broadbent 12
Brown 249
Bruce 417
Buchanan 267-90
Buneman 357-67
Burkhard 363
Burks 4
Burstall 23-50, 52, 89, 132, 155, 328, 474

Carbonell 196
Carson 122, 131
category theory 24, 29, 41-2, 49-50
Champernowne 5
Chandler 11-13
Chang 117, 178, 184-5
Chernoff bound 369-70, 373-5
chess 242, 246, 378
Chomsky 293, 307
Christofides 245, 248
Church 5
Clarke, A.C. 466
Clarke, G. 248
Clarke, S. 249
cleavage sets 270-1, 277, 279, 281
Clowes 437, 442
cluster analysis 244-5, 252
Cohn 58, 118
Cole 357
Coles 29
Collins 155, 328, 474
Colmerauer 194
Commoner 423
compiler 51, 56-7, 60-3, 327, 337
complexity 73, 167, 173, 180-3, 187, 189-93, 350-2
CONNIVER 443
Cook 78
Coombs 7, 12, 13
Cooper, D.C. 91-9
Cooper, L. 247
Cooper, W.S. 91
correctness 23, 25, 30, 34, 37, 51-2, 56-7, 91
Cover 373, 376
Crawford 465-80
Crespi-Reghizzi 350

Dantzig 248
Darlington 48, 129-37
Davies, D.J.M. 325-39
Davies, D.W. 5
DEACON 334
decision theory 374
DENDRAL 267, 276, 282
Derksen 428

Deutsch 205-40
developmental systems 341, 344-5, 350, 352
Diffie 52
Djikstra 52
Doucet 355
du Feu 172
Dyck language 297

Earnest 466
Eastman 247
Easton 78
Eccles 14
Eckert 6, 7, 15-17
edge matching 216, 219, 222-5, 229, 231, 237-8
EDSAC 17
EDVAC 4-6, 11, 15, 17
Eilenberg 41-2
Eilon 245, 248
ENIAC 4, 5, 10, 11, 13-15, 17, 19
Ernst 135, 141, 144

Feliciangeli 344, 346
Felts 246-7, 249, 254
Feigenbaum 241, 267-90
Feldman 241, 341, 343, 351-2
Fibonacci number 353
Fikes 135, 144, 196, 405-30
Finin 444-5, 458
flow diagram 23, 25, 32
Flowers 7, 10, 12
Floyd 23-6
FORTRAN 326
frame problem 195-7, 199, 200, 203
Frankel 10
Freeman 205
Freuder 436-7, 457
Friedman 293-306
Fukushima 376

game 390-1
game playing 246, 390
game tree *see* tree
Gardner 205
Gillett 248-9, 251
Ginsburg 293, 295-7, 299-301
Giv'eon 42
go 246
Gold, E.M. 297, 348-9
Gold, M. 241
Goldstine 4, 5, 9, 16
Good, D.I. 25-7
Good, I. J. 7, 8, 11, 17
Goodwin 14
Gordon 48, 194
Gould 78, 83, 132
GPS 135, 141-2, 144, 410
grammar 293-5, 297-9, 301, 335, 341, 344-5, 350
  string 310, 322
  transformational 293-5, 297-8, 301, 303, 305
  web 307-11, 313, 316-17, 319, 322-4
graph 144, 173, 270, 272, 307-8, 314, 323, 363-7, 389
  and-or 167, 169-71, 173, 176-7, 180, 183-5, 189
  theorem-proving 167-73, 176-7, 183-4, 189
Green 129, 135, 144, 146, 405
Gregory 377-85
Griffith 462
ground image 118-19
GT4 141-2, 157
Guard 78
Guzman 209, 432-7, 441-2

HAL 9000 466
Hall 181
Halsbury 4, 6, 17
Hamburger 297
Hart 82, 88, 135, 168, 181, 184, 373, 376, 405-30
Hamming distance 363-4
Hartree 14
Hayes, K.C. jr. 205-40
Hayes, P.J. 48, 116, 131, 153, 194, 196-8
Hayes, R.L. 245, 248
Healy 466
Helmholtz 379
Herman 341-56
Herskovits 462
heuristic 73, 168, 172-3, 180, 182, 184-5, 188-92, 198-9, 205, 208-9, 212, 216, 219, 222, 241-9, 251, 254, 268, 270-1, 287, 335, 352, 434, 436-7, 453, 457
heuristic search *see* search
Hewitt 25, 135, 196, 328-30, 413, 419, 428, 443
Hoare 24-5, 37
Hoffman 116, 132
Holt 423
Honeywell H316 467, 473-6, 480
Hopkins 364
Horn 462
higher order logic *see* logic
Hilbert 92
Huet 132
Huffman 437-8, 442

IBM 360/50 130, 248
ICL 4130 116, 155, 157, 465, 473-6
inductive inference 341
inductive net 358-61, 365
Isard 325-39
Ito 369-76

Jensen 78, 132
Johnson 245
Jordan 14

Kahn 7
Kailath 373, 376
Kanizsa 380, 382
Kant 327
Kaplan 52
Keenan 334
Keller 363
Kernighan 249
Kerse 48
Kilburn 7
Kilgour 241
King 25-6
Kingston 248
Knuth 11
Kortanck 247
Kowalski 82, 86, 88, 115-16, 120-1, 124, 127, 131, 136, 141, 153, 157, 167-94
Kraft 250
Kreisel 92
Krivine 92
Krolak 241-66
Kuehner 115, 117-28, 136, 157, 191-2
Kuhn 377

lambda calculus 51, 74
lambda conversion 132
Landin 52
Lawvere 41
LCF 51-3, 57, 59-60, 63
Lederberg 268, 272
Lee 344
Lehman 19
Lerman 447
Lindenmayer 342-4, 352
Linn 249
LISP 23-4, 48, 52, 54, 103, 105, 328
list processing 23, 25, 27-8, 30, 37, 55, 176
Lloyd 58
logic 379-80
  first-order 129, 131-2, 141
  higher-order 129, 132, 135, 180
  modal 325
  ω-order 83
  see predicate calculus
London 24-5, 52
Longuet-Higgins 325, 339, 342, 358, 367
Lovelace 6, 10, 15-16
Loveland 121
Lucchesi 132
Luckham 191

MacLane 49
MADM 7
Manna 132, 135
Marakhovsky 403
Marble 246-7
Marinov 180, 190-1
mass spectrometry 267-9, 272, 281, 283, 288
Mauchly 6, 7, 15-17
McCarthy 6, 24-5, 51-2, 62, 66-7, 70, 91, 195-6, 198-9, 328, 466
McCloy 247
McDermott 443
Meltzer 116, 192, 193
Menzler 4, 13
Michel 14, 20
Michie 6, 7, 11, 17, 141-65, 177, 180, 241, 403
Milgram 307-24
Miller 248-9, 251
Milner 51-70
Minsky 443
Mizumoto 307
modal logic *see* logic
Montanari 307, 309
Montague 334
Moore 101-16, 155, 157, 162
Morris, L. 52, 63
Morris, J.B. 25, 86
MOSAIC 13
Moses 89
multidimensional scaling 245
multi-POP 475
multitree 357, 359, 360-1
Munson 418
Mylopoulos 308

Neidell 245
Nelson 241-66
Nevins 78
Newell 135, 141, 241
Newey 63
Newman 4, 7, 9, 14, 16-17
Newton 248, 251
Niemela 364
Nilsson 88, 135, 144, 168-9, 181, 184-5, 188, 196, 405-30, 466
Norfolk 19
numerical taxonomy 244-5

Oberuc 245, 249
Occam's razor 271
Orban 436, 442

Pankhurst 364
Painter 24, 52, 62, 66-7, 70
Papert 443
paramodulation 81, 86-8, 132
Partee 293, 295-7, 299-301
Paterson 92
pattern matching 357
pattern recognition 242, 369, 374

Pavlidis 307, 323
Paz 352, 354
PCF-2 195, 199
perception 377-80, 382, 384, 406
  see vision
Peschansky 403
Peters 293, 295-301, 305
Petrick 298
Pfaltz 307, 309
Phelps 16
Phillips 4, 13-15, 19-20
Pickler 254
picture 211, 436, 469-71, 478-9
Pietrzykowski 78, 132, 135
PLANNER 135-6, 197-9, 201-2, 328-30, 338-9, 413-14, 419, 428, 443
Plotkin 48, 73-90, 180, 194
Poggendorf figure 383
Pohl 125, 127, 167-8, 177, 181, 186
POP-2 115-16, 155, 157, 162, 328-9, 331, 337, 474-7, 479
POPLER 1.5 338-9
Popplestone 129-30, 155, 163, 328, 465, 469, 474, 479
Postal 296
Poupon 25, 29
predicate calculus 52, 91, 130, 143, 195, 198-200, 329-30, 406, 409
  see logic
Presburger 91-2, 98
problem solving 129, 135, 141, 144, 193-8, 200, 241-2, 254, 379, 405, 412, 426-7, 429-30
program schema 92
Pugh 250
puzzle 205-8, 211-17, 219-23, 225, 229-30, 232, 235-8, 240, 244, 409

QA3 116
QA4 428
Quadling 364

Ramser 248
Randell 3-20
Raphael 168, 181, 184, 194-6
Rapp 247
Rattner 436, 442
Raven 343
Reddy 466
REDIST 247
Rees 7
Renwick 17
ReVelle 247
Roberts 462
Robinson, A.E. 417
Robinson, G.A. 122, 131-2
Robinson, J.A. 74, 78, 116, 118, 124, 131, 135, 141, 143, 146-8, 150-1, 162, 193
robot 135, 199, 405-6, 408-10, 412-15, 417-18, 422-9, 465-7, 470-1, 473-4, 476, 478
Rosenfeld 307-24
Rescher 196, 198, 203
resolution 73-4, 80, 82, 86-8, 101, 106-8, 111, 115-17, 120-2, 125, 129, 131-2, 134-6, 143-4, 146-8, 150, 155, 157-60, 162, 169, 180, 189-92
Richards 357
Ritchie 293, 295-301, 305
Ross 141-65, 177
Rowland 354
Rozenberg 344, 352
Rulifson 428

Sackman 241
Salomaa 293, 297-9, 301, 352
Salter 465, 479
Sandewall 48, 195-204
scene analysis 209, 307, 378, 431, 437, 462
Schonfield 78
Scott 51, 325-6, 329
search 135, 141, 173, 181, 201, 267, 269, 293, 296, 305, 357
  backward 200-1
  bi-directional 125, 127-8, 167, 169, 172, 176, 178-80, 183-4, 186, 190
  breadth first 148, 185, 198
  depth first 148, 161, 167-8, 179, 181, 454
  diagonal 127, 181-3, 186, 188-90
  graph 176, 185
  heuristic 136, 144, 180, 273
  serial 358
  unbounded 296, 299-300
search space 88, 121, 127-8, 158, 161, 167-70, 172, 175-77, 179, 183-4, 188-9, 192, 270
search strategy 126, 135, 143, 158, 161, 167-9, 172-3, 176-83, 185-6, 188-93, 198
search-tree *see* tree
Settle 78
Shank 308
Shannon, C. 8
Shannon, G.J. 141-65
Shaw 307
Shirai 462
Sibert 141, 143, 162, 180
Siklossy 180, 190-1
Simon 196, 198, 241, 444
Slagle 178, 184-5
Smith 191
Sneath 245
SNOBOL-4 130, 133
Sokol 245
Southwell 20
Sridharan 267-90
Steele 254

Steinberg 247
Stephenson 367
Strachey 326, 329
STRIPS 135, 196-7, 200, 202, 406-15, 418-19, 426-9
Strong 307
Surapipith 343
Surkis 249
Sussman 196, 443
Swain 247
Symms 247
synchronization problem 398, 403
Szilard 352

Tanaka, K. 307
Tanaka, M. 248
tangram puzzle 205-7, 209, 211, 239-40
Tarski 334
Tenenbaum 474
theorem-proving graph *see* graph
Thompson 334
tic-tac-toe 332, 337
Tillman 248
Toyoda 307
Trask 15
travelling salesman problem 247, 389
traversal theorem 314-15
tree 24, 37, 41-8, 76, 114-15, 121-2, 143, 148-54, 157-8, 202, 289, 294-6, 298-300, 304, 336, 359-60, 364, 420
  and-or 168, 170-1, 176-7, 183, 185, 419
  binary 28, 37
  derivation 122, 143
  game 414
  plan 419-20
  search 88, 120, 144-6, 185, 410, 419-20, 422
tree automata 24, 41
triangle table 407-8, 414, 423-5
Tsetlin, 389
Turing, A. 3-10, 14, 16-17, 19-20
Turing, Mrs. S. 5
Turing machine 298, 300
Turner 479

unification algorithm 74-5, 77-8, 82, 89, 101, 105-6, 131-2

van Dalen 344
van Leeuwen 344
Varshavsky 389-403
Veenker 116, 132
Vicens 466
vision 406, 431, 435, 444-5, 447, 456, 462, 466 *see* perception
von Neumann 4-10, 15-17

Waldinger 132, 135, 428
Walker 341-56
Waltz 437, 439-42
Waters 324
web grammar *see* grammar
West 9
Wegbreit 25, 29
Weyhrauch 51-70
Wickman 447
Wiener 20
Wilkes 16-17
Williams 7, 9, 17
Willshaw 357, 360
Wilson 191, 194
Winograd 196, 330, 334, 337, 405, 413
Winston 431-63
Wirth 37
Witasek 382
Womersley 5, 13-15, 19-20
Woods 334
Wos 122, 131-2, 136, 190-1
Wright, J.B. 41-2
Wright, J.W. 248
Wynn Williams 11, 14

Yates 116, 418