

Michie

THE
HISTORY
OF
THE
CITY
OF
EDINBURGH
FROM
THE
FIFTH
TO
THE
SEVENTEENTH
CENTURY

Edinburgh

MACHINE
INTELLIGENCE 3

A*

MACHINE INTELLIGENCE 3

edited by

DONALD MICHIE

Department of Machine Intelligence and Perception
University of Edinburgh

with a preface by

THE EARL OF HALSBURY

EDINBURGH at the University Press

© 1968 Edinburgh University Press
22 George Square, Edinburgh 8

Printed in Great Britain at
The Kynoch Press, Birmingham

85224 004 X

Library of Congress
Catalog Card Number
67-13648

PREFACE

The impact of computers on modern practice and thought can be analysed into four blocks of about five years each, beginning in 1948. During the course of the first five years the world became sprinkled with a sufficiently large handful of computers to arouse popular interest in them. It was in 1953 accordingly that I wrote an introduction to Lord Bowden's popular best-seller *Faster than Thought*, which represented a response to this interest. The problems of those days were mostly concerned with hardware design, particularly reliability. These problems were solved by the coming of the transistor with micro-circuits on the horizon during the course of the second block of five years. Software problems began to become urgent during the course of the third block of five years. The demand of users for higher order languages threw a tremendous burden on compiler writers as FORTRAN, COBOL, and ALGOL were evolved, and it gradually became clear that problems of compiler writing, logic, syntax, and semantics were all involved with one another together with other non-numerical uses of computers and the simulation of other types of hardware (e.g. neural networks capable of learning) in an interdisciplinary complex which has become known as computing science.

Five years ago, at the beginning of the fourth block of five years, the Department of Scientific & Industrial Research recognized this subject and the need to provide for it as part of its grant-aiding machinery by setting up a Computing Science Committee, of which I became the first chairman. When DSIR was reconstructed the Committee continued in being under the successor-body, the Science Research Council, with whose funds much of the research published here has been carried out.

The University of Edinburgh was at that time becoming aware of an energetic and computer-minded young Reader in the Medical Faculty – Donald Michie – who undertook a survey of all computing science going on in the United Kingdom on behalf of the Committee of which he later became a member. He is now Professor Michie of the Department of Machine Intelligence and Perception at Edinburgh University, and his annual 'Machine Intelligence Workshop' is increasingly becoming an international rather than a domestic event. This, the third volume of *Machine Intelligence*, published

PREFACE

with commendable speed after the conclusion of the 'workshop', records many of the papers given on the occasion of its third meeting.

The workshop, as its name implies, is not a conference devoted to speculation but a venue where hard results are hammered out and reported on, nearly all of them by young men.

From the beginning this has been a young man's subject. The creative imagination and energy of youth have been released on an unprecedented scale into this essentially interdisciplinary field. Looking at the range covered it is probable that only the young could assimilate the interconnections of such a ramifying field of subject matter.

With this commendation I wish the publication as wide a circulation as it deserves, for progress in computing science is at breakneck speed and the problem of keeping up with it is a very severe one.

THE EARL OF HALSBURY

February 1968

CONTENTS

INTRODUCTION	ix
MATHEMATICAL FOUNDATIONS	
1 The morphology of prex—an essay in meta-algorithmics. J.LASKI	3
2 Program schemata. M.S.PATERSON	19
3 Language definition and compiler validation. J.J.FLORENTIN	33
4 Placing trees in lexicographic order. H.I.SCOINS	43
THEOREM PROVING	
5 A new look at mathematics and its mechanization. B.MELTZER	63
6 Some notes on resolution strategies. B.MELTZER	71
7 The generalized resolution principle. J.A.ROBINSON	77
8 Some tree-parsing strategies for theorem proving. D.LUCKHAM	95
9 Automatic theorem proving with equality substitutions and mathematical induction. J.L.DARLINGTON	113
MACHINE LEARNING AND HEURISTIC PROGRAMMING	
10 On representations of problems of reasoning about actions. S.AMAREL	131
11 Descriptions. E.W.ELCOCK	173
12 Kalah on Atlas. A.G.BELL	181
13 Experiments with a pleasure-seeking automaton: J.E.DORAN	195
14 Collective behaviour and control problems. V.I.VARSHAVSKY	217
MAN-MACHINE INTERACTION	
15 A comparison of heuristic, interactive, and unaided methods of solving a shortest-route problem. D.MICHIE, J.G.FLEMING and J.V.OLDFIELD	245
16 Interactive programming at Carnegie Tech. A.H.BOND	257
17 Maintenance of large computer systems—the engineer's assistant. M.H.J.BAYLIS	269

CONTENTS

COGNITIVE PROCESSES: METHODS AND MODELS

- 18 The syntactic analysis of English by machine. J.P. THORNE,
P. BRATLEY and H. DEWAR 281
- 19 The adaptive memorization of sequences. H.C. LONGUET-
HIGGINS and A. ORTONY 311

PATTERN RECOGNITION

- 20 An application of Graph Theory in pattern recognition.
C.J. HILDITCH 325

PROBLEM-ORIENTED LANGUAGES

- 21 Some semantics for data structures. D. PARK 351
- 22 Writing search algorithms in functional form. R.M. BURSTALL 373
- 23 Assertions: programs written without specifying unnecessary
order. J.M. FOSTER 387
- 24 The design philosophy of POP-2. R.J. POPPLESTONE 393

- INDEX 403

INTRODUCTION

In a recent conversation, R.A. Brooker referred to an early categorization of the two schools which divide the world of computer science as the 'primitives' and the 'space cadets'. At no point is the need to find a meeting place and a market place between these schools more pressing than in the attempt to specify and construct intelligent machinery.

The space cadet operates with abstractions not because he enjoys being above anybody's head but because he believes that only by profound theoretical *coups* can our goals become even remotely attainable. So might early mechanical engineers have reasoned in planning the creation of an automotive engine: first develop the abstract notion of an automotive principle; then a mathematical theory to enshrine it; then invent the steam engine; only then develop the crafts of the designer, the wheelwright and the gear-cutter. By contrast, the passion of the primitives is to make something and to make it work, not shrinking from any methods however earthy. Their defence: that one Law of Thermodynamics, or even three, does not make a steam engine.

There is of course everything to be said against a doctrine which denies the arts of *ad hoc* implementation a central role in the development of a technology. Certainly, Machine Intelligence is no more than a technology, although perhaps the most rapidly growing of all on the contemporary scene. Yet the present decade is one of special excitement precisely because we are beginning to see the first great conquests (such as J.A. Robinson's contribution to the present volume) in the mathematics of mechanizable reasoning, without which all the most ingenious primitivism in the world will not avail. One can glimpse the same process at work in Amarel's analysis of problem formulation, and in a number of contributions in the area of programming theory and language design. Here the philosophic mantles of prophets absent from this collection, notably, J. McCarthy, C. Strachey, and P. Landin, are conspicuously in evidence. If one had to sum up the language trend one might say that it is directed towards a future in which *if a passage of text is respectable mathematics in a generally accepted notation, then it will compile*.

This volume comprises the papers delivered at the third Annual Machine Intelligence Workshop held in Edinburgh, 11-16 September 1967. Gratitude is

INTRODUCTION

owed to the University of Edinburgh for hospitality, and to its Principal, Professor M.M. Swann, FRS, who delivered a memorable luncheon address to the participants. We also owe particular thanks this year to the Science Research Council, who defrayed the Workshop's costs, and to the Edinburgh University Press, who have taken over responsibility for publishing the series.

DONALD MICHIE

February 1968

MATHEMATICAL FOUNDATIONS

The Morphology of prex— An Essay in Meta-algorithmics

J. Laski

Centre for Computing and Automation
Imperial College of Science and Technology, London

Ad-hoc definitions

A set is a prex, as is a list, as is a ring,
as is a queue . . .

Note: prex is the plural of prex.

INTRODUCTION

If we may refer to elements of some sort that may be distinguished ($a \neq b$) one from another, it may also be useful to refer to conceptual objects built up in various ways from some of these elements. The *prex* over these elements are some of the simplest such objects.

As instantaneous objects, prex must be decomposable; thus we need selectors. There is a family of several distinct kinds of prex but they all have in common a single *Selector* (*the* or *a*) that picks out one of their elements. This is what distinguishes them from the more general concept tree whose selectors (*car* and *cdr*) may yield either an element or a tree.

Again, as instantaneous objects, prex must be recognizable; thus we require *Predicates* of which there are two ('equality of two prex' and 'nil prex'). How these (with the *Constructors*) relate to the selectors distinguishes the various siblings of the prex family such as 'list', 'set', 'queue', 'bag', etc.

But prex are also historic objects. Constructors applied to a prex will change the instantaneous value. Thus we have one *Creator* ('make nil prex'), two *Updaters* ('try to add an element' and 'try to take away an element') and the obvious *Destroyer* ('forget').

That our objects change their values, or even cease to exist, distinguishes what we are concerned with from a traditional mathematical system (e.g. that described by set-theory). Such mathematical systems concern themselves with constant objects so that the time-sequence of evaluation of an expression can have no effect on its value, unlike the case of objects on which a

computation takes place. Thus I call this work 'meta-algorithmic' rather than 'metamathematical'.

The purpose of distinguishing the varieties of prex—the name is a contraction of 'predicate extension'—is to understand their use in specifying an intensive name for the range of repetition of a quantifier-built expression or command, as introduced in CSL or other so-called simulation languages. A detailed treatment of quantification is being prepared for a further paper. The purpose here is to provide a complete catalogue of the distinct kinds of prex, naming them and enabling them to be distinguished one from another. This will be done in two ways, syntactically and semantically.

Formally, we give first a group of axioms that relate syntactically expressions obtained by applying constructors, predicates and selectors to any prex. Then we give a group of assertions, the truth or falsity of which distinguishes the varieties of prex. The sense in which we have a complete morphology of prex is that we can produce a prex that results from any consistent assignment of values (*true*, *false*) to these assertions.

Semantically, we can represent each kind of prex by lists of references to representations of their elements; then we can distinguish the distinct kinds by appropriate distinguishing interpretations of their constructors, predicates and selectors in terms of those of the lists. Since, for discrete modelling language we shall want to use prex of objects of the same (extensional) type, we present here an aesthetic syntax that might well be part of such a programming language rather than the not very appealing syntax it is convenient to use in the earlier part of the paper.

A METALANGUAGE

To talk about prex and their elements we require both an object language and a meta-language. The object-language is made up of signs which may form well-formed expressions; according to the syntax these may designate elements (in the domain α), prex (α -lists, α -sets, etc) or provable or disprovable sentences. These latter will be taken as true or false respectively when they appear, as they always do, in statements of the meta-language.

The metalanguage is, of course, used to express meaningful statements about the object language and I must start by assuming that the reader understands by it what I intend him to understand by it. To make this possible, I shall start with part of our common cultural heritage, the English Language. Fortunately I do not require its full richness, but only a sparse subset. To this I add some special signs from the vocabularies of predicate logic and ISWIM. These signs are to be construed with the intuitive content of what they formalize in these systems; to help the unfamiliar reader, some are listed below with a comment on their intuitive meaning. This list is not exhaustive, nor is their syntax (especially binding-power) fully defined. Thus the meta-language remains unformalized and after this section we need no longer consider a meta-meta-language.

isa	type recognizer
ϕ, ψ	a ' ϕ ' together with a ' ψ '
$\{\phi, \psi\}$	a ' ϕ ' or a ' ψ '
$\langle \phi, \psi \rangle$	a ' ϕ ' followed by a ' ψ '
'juxtaposition'	functional application
$[\phi \rightarrow \psi, \chi]$	CPL or LISP conditional expression
$\phi \Rightarrow \psi$	a function with domain ϕ and range ψ
ϕ for ψ	' ψ ' is the definiens, ' ϕ ' the definiendum
where	subsidiary definition
and	simultaneous definition
rec	recursive definition

For example:

D1 *pred* for $\{true, false\}$

allows us to refer to what some other writers call boolean.

SYNTAX

The prex primitives

We are now in a position to introduce the prex primitives:

the creator μ ;	the updaters π, δ ;	the destroyer ν ;
the selector $\otimes$;	the predicates η, ϕ .	

			<i>Informal explanation</i>
C1	μ	isa $\Rightarrow \alpha$ -prex	creates a new nil prex
U1	π	isa $\langle \alpha$ -prex, $\alpha \rangle \Rightarrow \alpha$ -prex	tries to add an element
U2	δ	isa $\langle \alpha$ -prex, $\alpha \rangle \Rightarrow \alpha$ -prex	tries to remove an element
F1	ν	isa α -prex $\Rightarrow$	forgets the prex
P1	η	isa $\langle \alpha$ -prex, α -prex $\rangle \Rightarrow pred$	equal R-value
P2	ϕ	isa α -prex $\Rightarrow pred$	recognizes the nil prex
S1	$\otimes$	isa $\beta \Rightarrow \alpha$ where $(\forall A) (A \text{ isa } \beta: \Leftrightarrow: A \text{ isa } \alpha\text{-prex} \ \& \ \sim \phi A)$	selects an element from a non-nil prex

It is convenient to introduce infix operators for π, δ, η and some simple non-primitive predicates. Notice that, as defined, the evaluation of the predicate ϵ updates A . Further, between each primitive updating of A in this evaluation other processes may use or update A . For ϵ to be well defined we must be able to exclude the updaters of other processes. It is often, indeed, convenient to make a pseudo-primitive by:

- updating A at the end of the evaluation to its original value.
- excluding also selectors and predicates of other processes

MATHEMATICAL FOUNDATIONS

- D2 $A \uparrow a$ for πAa
 and $A \downarrow a$ for δAa
 and $A=B$ for ηAB
 and rec $a \in A$ for $[\phi A \rightarrow \text{false}, a \equiv \otimes A \rightarrow \text{true}, a \in \delta A \otimes A]$
 where a isa α and A isa α -prex and B isa α -prex
- D3 $A \neq B$ for $> A=B$
 and $a \notin A$ for $> a \in A$
 where a isa α and A isa α -prex and B isa α -prex

In what follows we adopt certain meta-linguistic simplifications and conventions:

1. we drop the explicit mention of the type α and speak simply of prex and elements
2. $a, b, c, \dots; A, B, C, \dots; i, j, \dots$ specify conformal elements, prex and integers respectively.

The comments beside the axioms give their intended content and should, formally, be ignored.

The prex axioms

- | | |
|--|---|
| x1 $(\forall A, B) (\phi A. \& \phi B: \rightarrow: A=B)$ | <i>comment</i>
unique nil prex |
| x2 $\phi \mu$ | the creator makes
a nil prex |
| x3 $(\forall A) (\sim \phi A. \rightarrow. A \downarrow \otimes A \neq A)$ | the selected element
is removable |
| x4 $(\forall A, a) (\sim \phi A: \rightarrow: a \in A, \leftrightarrow. a \in A \downarrow b \uparrow b \text{ where } b \equiv \otimes A)$ | when replaced
yield a coextensive
prex. |
| x5 $(\forall A, a) (a \notin A. \rightarrow. A \uparrow a \neq A)$ | can always adjoin
a new element |
| x6 $(\forall A, a) (a \in A \uparrow a)$ | |
| x7 $(\forall A, B, a) (A=B: \rightarrow: A \uparrow a=B \uparrow a. \&. A \downarrow a=B \downarrow a) \uparrow$ and $\downarrow$ preserve
equality | |

Notice the use of *where* to provide temporary storage in x4.

The following theorems follow from these axioms and definitions:

- T1 $(\forall A, a, b) (b \in A \uparrow a: \leftrightarrow b \equiv a \vee b \in A)$
 T2 $(\forall A, a) (\phi A: \rightarrow: A \uparrow a \neq A)$
 T1 follows from x3 and D2
 T2 follows from x6 and D2

The remaining axioms concern equality and the finitude of prex and it is convenient to introduce further subsidiary definitions. The pseudo-value *undef*, enables functions that are in intended application partial to be replaced by total functions.

- D4 $\text{rec } \mathcal{S}(A, i) \text{ for } [i=0 \rightarrow A,$
 $\phi(\mathcal{S}(A, i-1)) \vee \mathcal{S}(A, i-1) = \text{undef} \rightarrow \text{undef},$
 $\mathcal{S}(A, i-1) \rightarrow \otimes \mathcal{S}(A, i-1)]$
- D5 $\mathcal{E}(A, i) \quad \text{for } [\mathcal{S}(A, i-1) = \text{undef} \rightarrow \text{undef}, \otimes \mathcal{S}(A, i)]$
- D6 $\mathcal{N}(A) \quad \text{for least } i (\phi(\mathcal{S}(A, i)))$
- x8 $(\forall A) (\exists i) (\phi(\mathcal{S}(A, i)))$ finitude
- x9 $(\forall A, B) (A=B: \rightarrow: < \mathcal{E}(A, 0), \dots, \mathcal{E}(A, \mathcal{N}(A)) > =_{\text{perm}} < \mathcal{E}(B, 0), \dots, \mathcal{E}(B, \mathcal{N}(B)) >)$
- where *perm* is a permutation of $\mathcal{N}(A)$ elements.

The following two theorems follow immediately:

- T3 $(\forall A, B) (A=B \leftrightarrow B=A)$
- T4 $(\forall A, B) (A=B: \rightarrow: \mathcal{N}(A) = \mathcal{N}(B))$

The axioms x1-x9 are believed to be independent, consistent and categorize the prex family. However, no formal proofs were attempted. To distinguish prex siblings we now list various assertions whose truths or falsity determine which sibling we have. These are not independent.

As a further simplifying convention we omit in what follows the initial universal quantifiers.

The distinguishing assertions

- A1 $(a \in A) \rightarrow A \uparrow a = A$ element multiplicity
- A2 $a \in A$ evaluates as a pseudo-primitive peepability
- A3 $A=B \rightarrow \otimes A = \otimes B$ ordinality
- A4 $A=B: \rightarrow: ((\forall i) (0 \leq i \leq \mathcal{N}(A): \rightarrow: (\exists j) (0 \leq j \leq \mathcal{N}(A)$
 $\& (\forall k) (0 \leq k < \mathcal{N}(A) \rightarrow \mathcal{E}(A, i+k \bmod \mathcal{N}(A))$
 $= \mathcal{E}(B, j+k \bmod \mathcal{N}(B))))))$
- A5 $A \uparrow a \neq A \rightarrow \otimes A \uparrow a = a$ LIFO
- A6 $A \uparrow a \neq A: \rightarrow: a \equiv \mathcal{E}(B \uparrow a, \mathcal{N}(A)+1)$ where $B=A$ FIFO
- A7 $\otimes A = \text{the } a \in A \text{ with max } f(a) \text{ where } f \text{ is a } \alpha \Rightarrow \text{number sorted list}$

A2 is a generalization of the nation of 'no-peeping' that distinguishes a push-down, in which only one element is visible without updating the pushdown, from a stock, in which any element is visible, but not, of course, removable. We do not distinguish the two kinds of no-peeping prex: that in which $\otimes A$ is visible without updating A and that in which merely looking for $\otimes A$ updates A . For a stack or an unordered prex A can be restored to its original value since, obviously,

$$(A \downarrow b \uparrow b \text{ where } b \equiv \otimes A) = A$$

This does not hold for a queue. For a sorted list this holds only if removing and replacing $\otimes A$ has no side-effects on the value of $\lambda' Af(A)$. Thus no-peeping prex have the important property that they may not be directly copied. The lup has only the update, not the load defined. For these constructed

MATHEMATICAL FOUNDATIONS

objects, however, $\otimes A$ is not an isolatable primitive and therefore I do not consider them further here.

Streams are analogs of queues, and there are corresponding constructed objects that are analogs of other prex and used similarly to communicate data from one information system to another. They have the very interesting property that $\uparrow$ and $\downarrow$ are updaters of distinct processes. Again, such constructed objects do not, with respect to a single process, have a full complement of primitives and I do not consider them further here.

The varieties of prex

If $A_1, \dots, A_7$ were independent one from another there would be 128 prex siblings. This is reduced to, at most, 24 by the following two theorems and one convention. That there are precisely 24 follows since A_1 and A_2 are each independent of the other assertions.

T5 $A_5 \& A_6. \rightarrow. \sim A_7: \&: A_6 \& A_7. \rightarrow. \sim A_5$
 $: \&: A_7 \& A_5. \rightarrow. \sim A_6$

T6 $A_5 \vee A_6 \vee A_7. \rightarrow. A_3 \& A_4$

C6 $A_3 \& A_4 \rightarrow. A_5 \vee A_6 \vee A_7$

T5 states that stack, queue and sorted list are incompatible notions. This follows from the incompatible definitions of $\otimes A$ for each sibling. We now prove T6.

Suppose A_5 and $A = B$

Then we can prove A_3 by induction on i for the sequence
of prex $\mathcal{S}(A, i), \mathcal{S}(B(A, i) \mid i = \mathcal{N}(A), \dots, l, o$

A_4 holds since $\mathcal{E}(A, i) = \mathcal{E}(A, j). \leftrightarrow. i = j$

Similar proofs exist for A_6 and A_7 .

It remains to discuss C6, the converse of T6, which states that a ring with a distinguished head is a stack, or a queue, or a sorted list. Consider the following argument.

Given A, B such that $A = B$. A_3 and A_4 imply that there is some well-defined selection rule for $\otimes$. This must be a criterion based on the value of some function either of the elements themselves or the sequence in time in which they were added to the prex or both. A_7 is the general case for the elements themselves, A_5 and A_6 the simple functions of history 'first' and 'last'. C6 is a rejection of more complex functions of history or mixed functions.

Thus, in Table 1, which gives the siblings of the prex family, and in the later Semantic Interpretations, prex with pathological criteria such as

if $\mathcal{N}(A)$ is prime, first, otherwise max f ,

which are contrary to C6, are excluded as beneath disdain.

Table 1, then, lists by name the legitimate siblings of the prex family, with ticks and crosses signifying whether or not A_1 – A_7 hold. The names used

	A1	A2	A3	A4	A5	A6	A7	
bag-np	x	x	x	x	x	x	x	A1
bag	x	✓	x	x	x	x	x	2
set-np	✓	x	x	x	x	x	x	3
set	✓	✓	x	x	x	x	x	4
list-np	x	x	✓	x	x	x	x	B1
list	x	✓	✓		x	x	x	2
rf-list-np	✓	x	✓	x	x	x	x	3
rf-list	✓	✓	✓	x	x	x	x	4
ring-np	x	x	x	✓	x	x	x	C1
ring	x	✓	x	✓	x	x	x	2
rf-ring-np	✓	x	x	✓	x	x	x	3
rf-ring	✓	✓	x	✓	x	x	x	4
pushdown	x	x	✓	✓	✓	x	x	D1
stack	x	✓	✓	✓	✓	x	x	2
rf-pushdown	✓	x	✓	✓	✓	x	x	3
rf-stack	✓	✓	✓	✓	✓	x	x	4
queue-np	x	x	✓	✓	x	✓	x	E1
queue	✓	x	✓	✓	x	✓	x	2
rf-queue-np	x	✓	✓	✓	x	✓	x	3
rf-queue	✓	✓	✓	✓	x	✓	x	4
sorted list-np	x	x	✓	✓	x	x	✓	F1
sorted list	x	✓	✓	✓	x	x	✓	2
rf-sorted list-np	✓	x	✓	✓	x	x	✓	3
rf-sorted list	✓	✓	✓	✓	x	x	✓	4

Table 1

accord, as near as I can judge, with their usual connotation in a programming context.

However, not all possibilities are in general programming use, and, if needed, are usually provided by interpreting them by more usual prex with explicitly coded selectors, a technique that is of course available for the pathological prex swept under the wrong side of the blanket above. I have introduced one explicit new name and two qualifiers.

The prefix 'rf' connotes repetition-free. In general usage unordered prex are repetition-free, while ordered prex allow the same element to appear more than once. 'bag' is introduced for the unordered prex with repetition. 'Two blue balls, seven black balls, ...'.

The suffix 'np' connotes no-peeping, that is to say that the $\mathcal{E}$ -predicate cannot be decided without explicitly transforming:

$$A = \mathcal{P}(A, 0) \text{ to } \mathcal{P}(A, 1) \text{ to } \dots \text{ to } \mathcal{P}(A, \mathcal{N}(A))$$

Curiously enough, I have not seen this in a programming context except for the stack (pushdown) in which it is easiest to restore the R -value of A by putting the successive $\mathcal{E}(A, i)$ into a working pushdown.

SEMANTICS

The list model

We can give meaning to each kind of prex by mapping instantaneous prex onto lists of references to the maps of their elements and, historically, by interpreting the primitive or defined operations and predicates of the distinct prex siblings appropriately in terms of operators and predicates of these lists. We require, therefore, operators and predicates of lists; these can, of course, only be defined ostensively. In what follows $L, M, \dots$ designate lists, $l, m, \dots$ their elements; $i, j, \dots$ integers. A slightly relaxed meta-language will be used for clarity.

The list definition

Some selectors, etc. are:

```

s10 head (L)    isa l or null
s11 tail (L)    isa l or null
s12 count (L)   isa i
s13 next (l,L) here l ∈ remove (tail (L), L) isa l
s14 element (i,L) where l ≤ i ≤ count (L) isa l
D10 place (l,L) for value of § let j=1, I=nil-integer set;
                        § if element (j,L) ≡ do I ↑ j;
                        j:=j+1 § repeat while j ≤ count (L);
                        result is I §

x10 place (head (L), L)=1
x11 place (tail (L), L)=count (L)
x12 place (next (l,L), L)=1+place (l,L)

t10 i . ∈ place (l,L) ↔ element (i,L)=1
t11 i ∈ place (element (i,L), L)
    
```

Some updaters are:

```

u10 insertafter (l,i,L) where i < count (L) isa L
u11 insertbefore (l,i,L) where i < count (L) isa L
u12 remove (i,L)       where i < count (L) isa L
    
```

The following axioms apply:

```

x13 count (insertafter (l,i,L))=count (insertbefore (l,i,L))
      =count (L)+1
x14 count (remove (i,L))=count (L)-1
x15 i+1 ∈ place (l, insertafter (l,i,L))
    
```

$$\begin{aligned}
x16 \quad & j \in \text{place}(m, \text{insertafter}(l, i, L)) : \leftrightarrow : j \leq i \ \& \ j \in \text{place}(m, L) \\
& \qquad \qquad \qquad .v.j = i + 1 \ \& \ l \equiv m \\
& \qquad \qquad \qquad .v.j > i + 1 \ \& \ j - 1 \in \text{place}(m, L) \\
x17 \quad & j \in \text{place}(m, \text{insertbefore}(l, i, L)) : \leftrightarrow : j < i \ \& \ j \in \text{place}(m, L) \\
& \qquad \qquad \qquad .v.j = i \ \& \ l \equiv m \\
& \qquad \qquad \qquad .v.j > i \ \& \ j - 1 \in \text{place}(m, L) \\
x18 \quad & \text{element}(i, L) \in \text{remove}(i, L) : \leftrightarrow : \mathcal{N}(\text{place}(\text{element}(i, L), L)) > 1 \\
x19 \quad & j \in \text{place}(m, \text{remove}(i, L)) : \leftrightarrow : j < i \ \& \ j \in \text{place}(m, L) \\
& \qquad \qquad \qquad .v.j \geq i \ \& \ j - 1 \in \text{place}(m, L)
\end{aligned}$$

This completes the ostensive definition of lists DSWIM?¹

Lists as given above have more structure than is required to interpret certain prex-sibling. We therefore introduce the function

select (*L*) isa $i \leq count(L)$

whose value for a given L is some integer in $\langle 1, \text{count}(L) \rangle$ to escape the temptation to allow the structure of the interpretation to persuade us to admit illegal constructors, predicates and selectors to the prex itself. Clearly in any practical interpretation any simple function will do depending on coding convenience.

The Aesthetic Syntax

- I $a \in A$
 - II *the* A
 - III a from $A < C_1 >_0 < \text{else } C_2 >_0$
 - IV a into $A < C_1 >_0 < \text{else } C_2 >_0$
 - V *next* (i, a, A)
- $\otimes$
 $\Downarrow$
 $\Uparrow$
- I is only available for which the test can be evaluated without side effects on A , i.e. those for which 'no-peepability' does not apply.
 - II is always defined
 - III the optional clauses C_1 , $\text{else } C_2$ need neither appear. If they do, however, C_1 is performed if $a \Downarrow A \neq A$, C_2 otherwise.
 - IV The optional clauses C_1 $\text{else } C_2$ may only appear if $(\exists a) (A \Uparrow a = A)$ i.e. if the prex is not a repetition-free sibling. They need not appear for the other prex. If they do, however, C_1 is performed if $A \Uparrow a \neq A$, C_2 otherwise.
 - V is defined only for ordered siblings. Approximately, it transforms $\mathcal{E}(a, j)$ to $\mathcal{E}(a, j, +1)$. More precisely, since the generalized prex allows repetition

$$\mathcal{E}(i,A)=a \ \& \ i < \mathcal{N}(A): \rightarrow: next(i,a,A)=\langle i+1, \mathcal{E}(i+1,A) \rangle$$

We may also define the function of two arguments

$$next(a, A) = second(next(least\ i(i \in place(a, A)), a, A))$$

¹ Do you See *What I Mean*? If not, I will point harder.

MATHEMATICAL FOUNDATIONS

Since for repetition-free prex, $\mathcal{N}(\text{place}(a, A)) \leq 1$, it is sufficient to interpret this latter function.

We can now define $I-V$ for A1-F4. a, A map to 1, L , respectively.

If C_1, C_2 are not referred to in the interpretations of III and IV, only forms of the command in which they do not appear are allowed. Codings are for clarity of exposition rather than efficiency. *ND* signifies not definable. If the code name for a prex-sibling is given, the interpretation is to be as previously given for that sibling.

A1 *bag-np*

I *ND*

II *element (select (L), L)*

III *test (a=the A) thendo § remove (select (L), L); C₁ § ordo C₂;*

IV *insertbefore (l, 1, L)*

V *ND*

[NB (α) is the interpretation of α just defined]

A2 *bag*

I *value of § for i=1 to count (L) do if 1=element (i,L) thendo result is true; result is false. §*

II *A1*

III *test (a ∈ A) thendo § for i=1 to count (L)*

do if l=element (i,L) thendo § remove (i,L);
break § C₁ §

ordo C₂;

IV *A1*

V *ND*

A3 *set-np*

I *ND*

II *A1*

III *A1*

IV *if value of § for i=1 to count (L) do if l=element (i,L)*
thendo result is false;
result is true §

thendo § insertbefore (1, 1,L); C₁ § ordo C₂;

V *ND*

A4 *set*

I *A2*

II *A1*

III *A2*

IV *A3*

V *ND*

B1 list-np

- I ND
- II A1
- III A1
- IV A1 [NB There are also the concepts of 'insertbefore' 'insertafter' for lists]
- V $[i \in \text{place}(l, L) \ \& \ i \neq \text{count}(L) \rightarrow \langle i+1, \text{element}(i+1, L) \rangle, \text{undef}]$

B2 list

- I A2
- II A1
- III A2
- IV A1
- V B1

B3 rf-list-np

- I ND
- II A1
- III A1
- IV A3
- V $[i \neq \text{count}(L) \ \& \ i \neq \text{undef} \rightarrow \text{element}(i+1, L), \text{undef}]$
 $\text{where } i = \text{the } i \ni 1 = \text{element}(i, L)$

B4 rf-list

- I A2
- II A1
- III A2
- IV A3
- V B3

C1 ring-np

- I ND
- II A1
- III A1 [NB there is also a concept of 'insertbetween' for rings]
- IV A1
- V $[i \in \text{place}(1, L) \rightarrow i \neq \text{count}(L) \rightarrow \langle i+1, \text{element}(i+1, L) \rangle, \langle 1, \text{head}(L) \rangle, \text{undef}]$

C2 ring

- I A2
- II A1
- III A2
- IV A1
- V C1

MATHEMATICAL FOUNDATIONS

C3 *rf-ring-np*

- I ND
- II A1
- III A1
- IV A3
- V $[i \neq \text{undef} \rightarrow i \neq \text{count}(L) \rightarrow \text{element}(i+1, L), \text{head}(L), \text{undef}]$
where $i = \text{the } i \triangleright 1 = \text{element}(i, L)$

C4 *rf-ring*

- I A2
- II A1
- III A2
- IV A3
- V C3

D1 *pushdown*

- I ND
- II $\text{element}(1, L)$
- III $\text{test } l = \text{element}(1, L) \text{ thendo } \S \text{ remove}(1, L); C_1 \S \text{ ordo } C_2;$
- IV A1
- V B1

D2 *stack*

- I A2
- II D1
- III D1
- IV A1
- V B1

D3 *rf-pushdown*

- I ND
- II D1
- III D1
- IV A3
- V B3

D4 *rf-stack*

- I A2
- II D1
- III D1
- IV A3
- V B3

E1 *queue-np*

- I ND
- II $\text{element}(\text{count}(L), L)$

III test ($1 = \text{element}(\text{count}(L), L)$) thendo § remove ($\text{count}(L), L$); C_1 §
 ordo C_2 ;

IV A1

V B1

E2 queue

I A2

II E1

III E1

IV A1

V B1

E3 rf-queue-np

I ND

II E1

III E1

IV A3

V B3

E4 rf-queue

I A2

II E1

III E1

IV A3

V B3

F1 sorted list-np [NB $f(l) = f(a)$ and may vary while $a \in A$]

I ND

II element (value of § let $x = -\text{infinity}$ and j be integer;
 for $i = 1$ to $\text{count}(L)$ do if $x < f(\text{element}(i, L))$
 do § $x \leftarrow (\text{element}(i, L))$;
 $j \leftarrow i$ §

result is j §, L)

III test ($a = \text{the } A$) thendo § for $i = 1$ to $\text{count}(L)$ do
 if $l = \text{element}(i, L)$ thendo § remove (i, L)
 break § C_1 §

ordo C_2 ;

IV A1

V [$l \neq \text{element}(i, L) \rightarrow \text{undefined}$,
 value of § let $x = -\text{infinity}$ and $ii = i$;
 for $j = i + 1$ to $\text{count}(L)$
 do if $f(\text{element}(j, L)) = f(l)$
 do result is $\langle j, \text{element}(j, L) \rangle$;

MATHEMATICAL FOUNDATIONS

```

for j=1 to count (L)
  do if  $x < f(\text{element}(j,L)) < f(l)$ 
    do  $x, ii \leftarrow f(\text{element}(j,L)), i$ ;
  result is [ $ii = i \rightarrow \text{underlined}, < j, \text{element}(j,L) > ] \S$ 

```

[NB a simpler representation of sorted lists is possible if $f(a)$ is fixed while $a \in A$]

F2 sorted-list

```

I   A2
II  F1
III F1
IV  A1
V   F1

```

F3 rf-sorted-list-np

```

I   ND
II  F1
III F1
IV  A3
V   value of  $\S$  let  $i, ii$  both = the  $j \ni l \equiv \text{element}(j,L)$  and  $x = -\text{infinity}$ ;
    for j=i+1 to count (L)
      do if  $f(\text{element}(j,L)) = f(l)$ 
        do result is element (j,L);
    for j=1 to count (L)
      do if  $x < f(\text{element}(j,L)) < f(l)$ 
        do  $x, ii \leftarrow f(\text{element}(j,L)), j$ ;
    result is [ $ii = i \rightarrow \text{undefined}, \text{element}(j,L) ] \S$ 

```

F4 rf-sorted-list

```

I   A2
II  F1
III F1
IV  A3
V   F3

```

PHILOSOPHIC CODA

The interest of this paper lies not only in the nature of the data-objects it explicates but in the methodology of the explication itself. This methodology is tied up with an ontology of what we describe as being data when we write programs to manipulate data and when we cause these programs to be executed in some information system.

When we express our programs, our operands are conceptual objects of differing *types*: by a type we mean that 'operators' may properly—i.e. making up well-formed expressions and well-formed commands—be applied

to these objects and what will be the result of doing so. If the operator is a selector it will yield an object or its value; if an updater it will change values of objects to which it is applied. Some operators do both.

In a particular information system data-objects are represented by bit-patterns arranged in data-structures; the effect of executing a process which involves certain data-objects, is to refer to and manipulate, in a way that interprets the program of that process, the data-structures that represent them.

The purpose of making the distinction between data-object and data-structure is to emphasize that there are many data-structures that can be used to represent a given type of data-object and the same data-structure may represent many types of data object.

When we are concerned with programming languages we are concerned with data-objects; when we are concerned with running programs, we are concerned with data-structure. Implementation is concerned with the choice of representation functions that relate the two.

Type further determines what are the primitive selectors, predicates and updaters of a data-object. If several processes may concurrently refer to data-object no primitive updater may operate simultaneously (in parallel) with any other updater, selector or predicate. Notice that we are speaking of the single data-object which may be known under different aliases to different processes. (Any process, however, may request that some sequence of operations on this data-object may have no extraneous updaters in parallel with it.)

A third notion concerns two kinds of data type, the intensive and the extensive. The first, which has been the concern of most computing is like 'integers' or 'reals' where new temporary L-value may be created during an evaluation. Extensive data (e.g. 'the entry points in a stored library') does not allow the creation of anonymous temporary L-values in this way. Constructed data-objects and the objects from which they are constructed, which is what we have been concerned with, are usually of an extensive type.

A final remark on the nature of semantic interpretation: I do not understand what is meant by a model of a formal system that is not itself a formal system. Thus ISWIM must always stop and say DSWIM. However much I say what I mean, what I use to do so is necessarily a *donné* and must end with Do you see what I mean? To which the due answer is 'I think so'.

Acknowledgments

The S.R.C. (ATLAS Computer Laboratory) supported the research that is reported here. Rod Burstall first drew my attention to the use of ISWIM and axioms to explicate sets and provided the definition of $\mathcal{E}$, and I am grateful to him and others of my friends for many extensive and intensive discussions of these ideas.

BIBLIOGRAPHY

- Buxton *et al.* (1966), *C.P.L. Reference Manual* (ed. Strachey, C). Privately distributed.
Culik, K. (1968), On simulation methods and some characteristics of simulation languages. *Proc. of IFIP Conference on Simulation Languages* (Oslo, 1967).

MATHEMATICAL FOUNDATIONS

Duncan, F. (1966), The ultimate meta-language. *Formal Language Description Languages for Computer Programming*, pp. 295-9 (ed. Steel, T.B.). Amsterdam: North Holland Press.

Landin, P.J. (1966), The next 700 programming languages. *Communs Ass. comp. Mach.*, 9, 166.

Laski, J.G. (1968), A data structure for simulation and other discrete programming. A driver for simulation. *Proc. of IFIP Conference on Simulation Languages* (Oslo, 1967).

Morris, C.W. (1938), Foundations of the theory of signs. *International Encyclopaedia of Unified Signs, I*, pp. 1-59. Chicago: University of Chicago Press.

Russell, B. and Whitehead, A.N. (1911), *Principia Mathematica*. Cambridge.

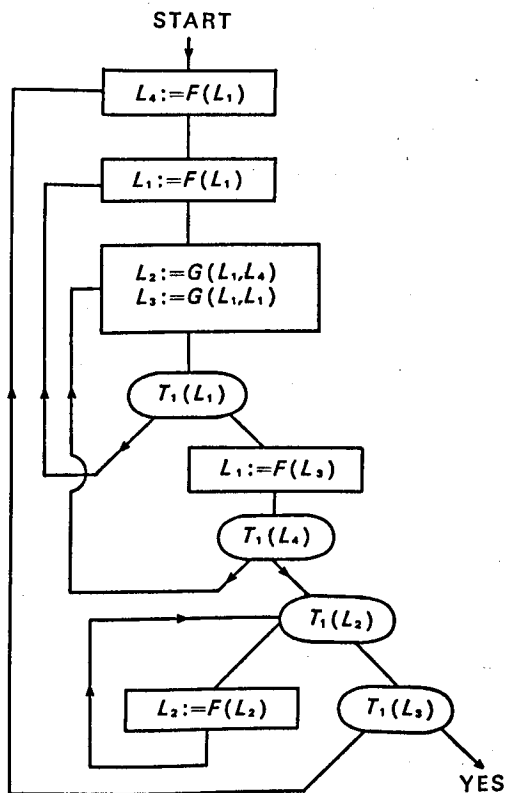


Figure 1. Schema P

2

Program Schemata

M. S. Paterson

Department of Pure Mathematics
University of Cambridge¹

Summary

Program schemata provide a simple and natural model in which many features of actual computer programs are represented and in which we can sensibly ask such questions as:

Do two different programs do the same thing?

How can we simplify a given program?

We enquire whether or not there exist effective algorithms to answer these questions.

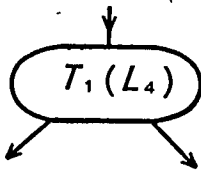
1. INTRODUCTION

1.1. Definitions

Figure 1 (*opposite*) is a *program schema* given in the form of a flow diagram.

$$L_2 := G(L_1, L_4)$$

is an *assignment instruction* which gives to the *location* L_2 a new value, which depends on the current values of L_1, L_4 , and on the as yet unspecified function G .



is a *test instruction* which applies the similarly unspecified predicate (two-valued function) T_1 to the value of L_4 and causes the left or right exit path to be taken according to whether the result is 0 or 1 respectively.

To provide an *interpretation* for program schemata, we choose some finite

¹Now at Atlas Computer Laboratory, Chilton, Berkshire.

or infinite set of values, then make an initial assignment of values from the set to each location and an assignment of appropriate functions and predicates on the set to the function and test symbols of the schemata. Given such an interpretation, the corresponding *execution sequence* through any schema, starting from the initial instruction, is defined in the obvious way. An execution sequence may terminate at an exit path labelled either 'YES' or 'NO', or may never terminate. The *value of the schema* under that interpretation is 'YES', 'NO', or 'UM' respectively, or sometimes we may say that the schema *succeeds*, *fails*, or *diverges* respectively. We define two schemata to be *equivalent* if their respective values are the same under all interpretations.

1.2. Equivalence problem

How can we tell whether or not two given schemata are equivalent? We can sometimes decide that they are inequivalent by finding a particular interpretation under which they evidently have different values. For example, the schema P of figure 1 clearly has the value 'YES' under an interpretation for which T_1 takes the value 1 for all arguments. P is therefore inequivalent to the schema D of figure 2, which always diverges.

One way to prove the equivalence of two schemata is by passing from one to the other by a chain of simple transformations, each of which obviously preserves equivalence. Some examples of such transformations are shown in figure 3. It should be clear in each case that they do preserve equivalence.

1.3. Example

We will now try to apply some such transformations to the schema P of figure 1. We observe first that if the test on L_4 ever takes the value 0, L_4 is never again assigned to and the schema can only diverge. We can therefore replace the left exit of this test by a trivial loop. It can now be more easily seen that, to avoid divergence, the first value assigned to L_1 in each outer loop, which is also the value assigned to L_4 , must have the test-value 1, and so the left exit of the test on L_1 can also be replaced by a trivial loop. The resulting schema P' is shown in figure 4.

The test on L_4 is redundant. L_2 and L_3 are initially assigned the same value, and so, after 'unwinding' the L_2 -loop a little, the L_3 -test can also be dispensed with. The result is P'' in figure 5. Further transformations yield P''' (figure 6). With this hint the reader is invited to convince himself that the schema P is actually equivalent to the schema Q .

1.4. Simplification

One of our reasons for studying program schemata is that the simplifications which can be performed on schemata are those that we could do on programs in a programming language about which we were given a minimum amount of information. Schema transformations could thus form a part of a very general 'machine-independent' optimization algorithm for computer programs, and we should naturally like this part to be as efficient as possible.

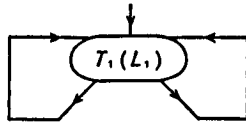


Figure 2, Schema D

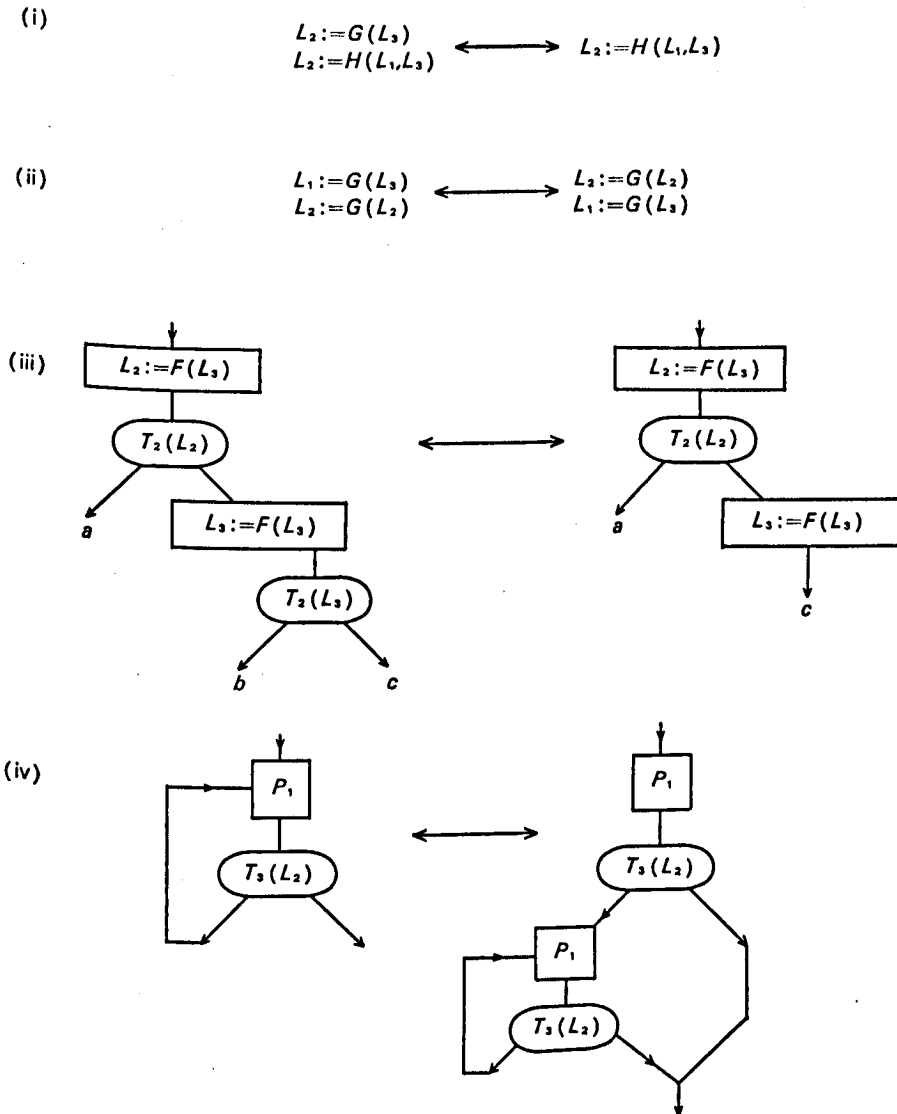


Figure 3

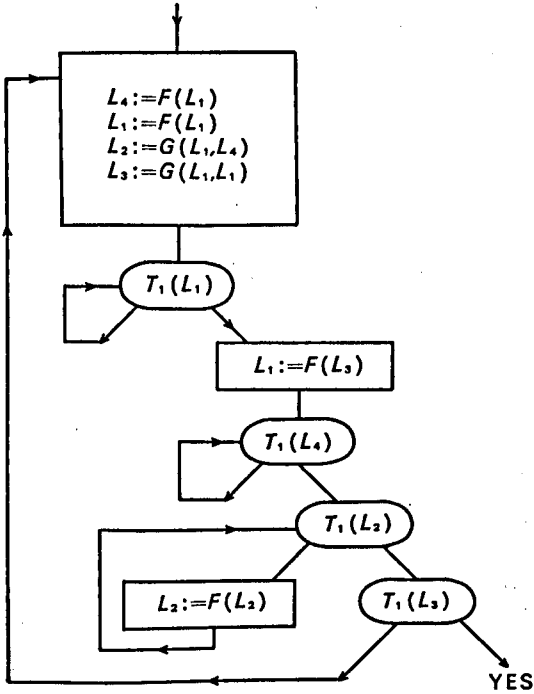


Figure 4. Schema P'

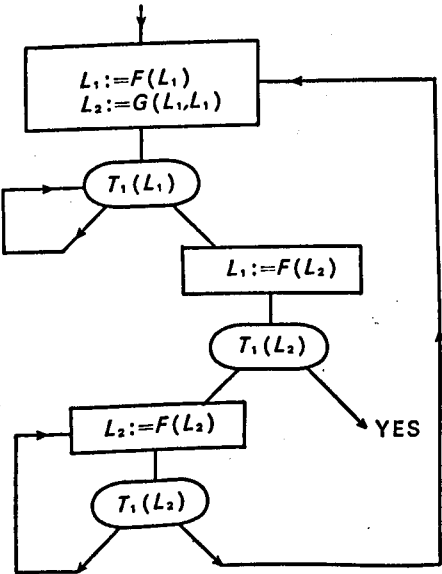


Figure 5. Schema P''

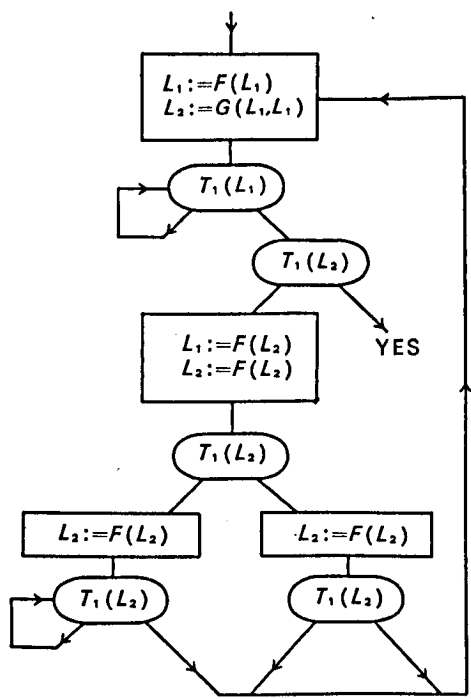


Figure 6. Schema P'''

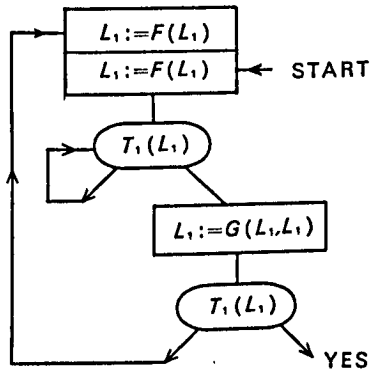
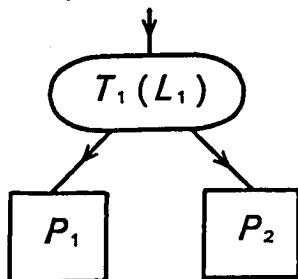


Figure 7. Schema Q

For many definitions of the 'simplicity' of a schema, the problems of 'optimizing' a given schema, that is finding the simplest equivalent schema, and of deciding whether or not two schemata are equivalent are interchangeable. For example, the optimization of the schema below usually presupposes a decision on the equivalence of P_1 and P_2 as schemata. In the other direction, if we suppose that program schemata could be enumerated in order of (decreasing) simplicity, then a theoretically adequate algorithm for optimization would consist in testing each schema in turn and selecting the first one equivalent to the given schema. It therefore seems appropriate to avoid the choice of a more or less arbitrary criterion of 'simplicity' and to consider only the problem of deciding the equivalence of pairs of schemata.



2. DECISION PROCEDURES

2.1. Loop-free schemata

For schemata whose flow-diagrams have no loops, the decision procedure for equivalence is easy. We illustrate a fairly practical solution by an example (figure 8). The condition (on the interpretation) that this schema has the value 'YES' can be expressed as a formula of the propositional calculus:

$$T_1[L_2] \ \& \ T_2[G(L_1, F(L_3))] \\
\vee. \neg T_1[L_2] \ \& \ T_2[H(L_2)] \ \& \ T_2[G(H(L_2), L_2)]$$

It is easy to see that two such schemata are equivalent if and only if their corresponding formulae are equivalent in the propositional calculus, and this is readily decidable.

2.2. Schemata which always halt

This decision procedure can now be extended to the class of schemata which halt under all interpretations. An example is shown in figure 9 whose longest execution sequence is of 31 instructions. Given any schema, we can 'unwind' the loops to produce an equivalent infinite tree. We then proceed systematically outwards from the initial instruction in that tree and examine each test instruction to see whether the same value has been tested previously by the same predicate on the way to that point. If so, then the subtree dependent on one of the exit branches from this test can be 'pruned' away and the test instruction removed, leaving an equivalent tree.

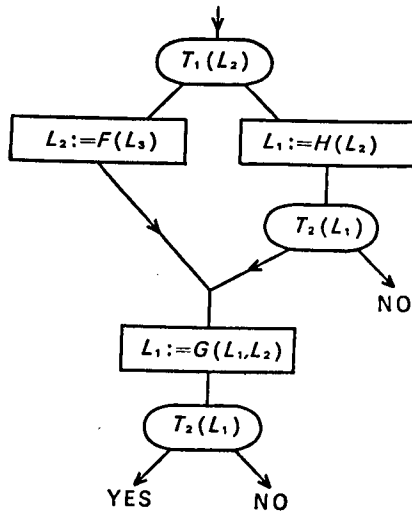


Figure 8

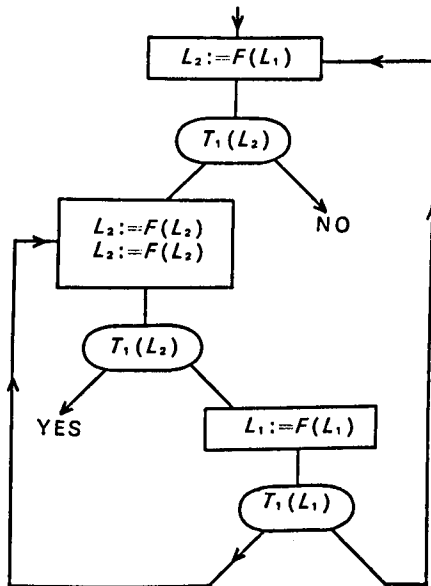


Figure 9

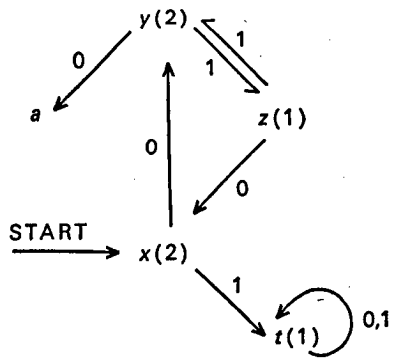


Figure 10. Schema *N*

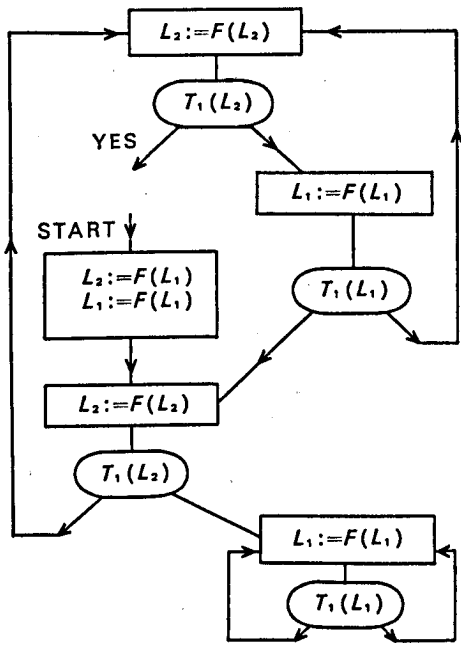


Figure 11. Schema *P(N)*

If this procedure does not eventually stop, there will be an infinite number of nodes in the pruned tree, and so an infinite path in it is guaranteed by König's infinity lemma. By the construction this path must be consistent with respect to all the tests on it, which is a necessary and sufficient condition for it to be the execution sequence corresponding to some interpretation. Thus for schemata which halt under all interpretations, the construction must produce an equivalent, finite, loop-free schema, after which the decision procedure of 2.1 can be used.

3. UNSOLVABILITY OF GENERAL EQUIVALENCE PROBLEM

3.1. 'Rule books'

If we could find a finite collection of effective, equivalence-preserving transformations so that any pair of equivalent schemata could be transformed one into the other by a chain of these transformations, i.e. a *rule-book*, then we would have a *partial procedure* to decide equivalence. This is an algorithm which, when presented with equivalent schemata, would eventually prove this, but might fail to reach any conclusion when given an inequivalent pair. The main result of this section will be to show that there is no effective procedure for determining whether or not two general schemata are equivalent, and indeed that there is not even a partial procedure. Hence any attempt to complete the transformations of figure 3 to an adequate rule-book must fail.

3.2. Two-headed automata¹

In proving these results it is convenient to introduce some *two-headed automata*. These are finite automata equipped with two one-way reading heads, which independently travel down the same input tape. Alternatively we may consider them as two-tape automata, as described by Rabin and Scott (1959), to which we always present identical pairs of input tapes. For each non-terminal state is specified which head is to read the next symbol and also, for each symbol, which is to be the next state. There is in addition one *terminal state*, a , and if this is reached the automaton is said to *accept* the tape. The automata start with both heads over the first symbol of the tape and in a specified *initial state*.

We give an example of a simple two-headed automaton (figure 10). (The head to be used is shown in parentheses after each state.) The reader should verify that N accepts just those tapes which first deviate from the sequence:

010110111011110

with an extra 0. When N is operating on such a tape, Head 1 reads along a sequence of 1's while Head 2 verifies that the following sequence contains

¹ These are also described by A.L. Rosenberg. The unsolvability proof given here is essentially that given by Rosenberg (1966).

precisely one more 1. After this, Head 1 is positioned to read the latter sequence and Head 2 is ready for the next piece of tape. An extra 1 causes N to enter a closed loop, but an extra 0 allows N to accept the tape. The *modus operandi* of this automaton is a useful introduction to the more complicated automata to be described.

For any two-headed automaton A , with input alphabet $[0, 1]$, there is a natural associated program schema $P(A)$. It should be a sufficient explanation of how this is obtained to display $P(N)$ corresponding to the automaton N above (figure 11). The first two instructions merely ensure that L_1 and L_2 start with the same values. $P(A)$ succeeds under an interpretation I if and only if the binary sequence of the values under I corresponding to:

$$T_1[F^2(L_1)], T_1[F^3(L_1)], T_1[F^4(L_1)], \dots$$

is accepted by A . For example, $P(N)$ succeeds under any interpretation where:

$$\begin{aligned} F &\text{ is the successor function,} \\ T_1(x) &= 0 \text{ if } x \text{ is prime} \\ &= 1 \text{ otherwise,} \\ &\text{and the initial value of } L_1 = 1, \end{aligned}$$

just as N accepts the tape:

$$010101110 \dots$$

3.3. Turing machines

We shall prove our main result by reducing the equivalence problem for schemata to the 'halting problem' for Turing machines. For a description of Turing machines, an explanation of our notation, and a proof of the theorem we use, the reader is referred to Davis (1958).

The states of our Turing machines are denoted by $q_1, q_2, \dots$, and the tape symbols by $S_0, S_1, \dots$. An *instantaneous description* gives the current expression on the tape and displays to the left of the scanned symbol the current internal state. For any Turing machine Z , if α is the instantaneous description:

$$PS_w q_i S_u Q$$

where P, Q are strings of tape symbols, then β succeeds α under Z , ($\alpha \rightarrow \beta$), if

$$\begin{aligned} \beta &\text{ is } S_o P S_w q_j S_u Q S_o \text{ and } q_i S_u S_u q_j \text{ is a rule of } Z, \\ \text{or } \beta &\text{ is } S_o P S_w S_u q_j Q S_o \text{ and } q_i S_u R q_j \text{ is a rule of } Z, \\ \text{or } \beta &\text{ is } S_o P q_j S_w S_u Q \text{ and } q_i S_u L q_j \text{ is a rule of } Z. \end{aligned}$$

α is *terminal* if there is no succeeding β . A *computation* of Z is a finite sequence of instantaneous descriptions $\alpha_0, \alpha_1, \dots, \alpha_p$ such that $\alpha_{i-1} \rightarrow \alpha_i$ for $i = 1, \dots, p$, and α_p is terminal. It is a well-known theorem that there is no effective (partial) procedure to determine, of a given Turing machine Z and instantaneous description α , when there is no computation of Z from α , i.e. when Z fails to halt after starting in the situation described by α .

3.4. Proof of unsolvability

For any Z and α_0 , we can effectively construct a two-headed automaton $A(Z, \alpha_0)$ over the alphabet $[S_0, S_1, \dots; q_1, q_2, \dots; *]$ which will accept just those tapes, if any, which start:

$$\alpha_0 * \alpha_1 *, \dots, * \alpha_p *$$

where $\alpha_0, \dots, \alpha_p$ is a (completed) computation of Z . Briefly, the action of $A(Z, \alpha_0)$ is as follows. It verifies with Head 2 that the tape starts with α_0 , then while Head 1 reads this α_0 it checks that the next part of the tape is α_1 . Now Head 1 is ready at the start of this α_1 and Head 2 is prepared to read on. This checking operation continues until either Head 2 finds a terminal description and the tape can be accepted, or else the check fails and the automaton is made to enter some trivial loop. The checking operation is easy. Apart from extra S_0 symbols at the beginning and end, the succeeding description differs only from its predecessor in two or three symbols around the state symbol and the automaton has only to verify that this change is in accordance with the rules of Z .

We observe that $A(Z, \alpha_0)$ accepts some tapes if and only if Z halts from α_0 . By suitably coding the alphabet of $A(Z, \alpha_0)$ into a binary alphabet, we may readily construct a two-headed automaton $A'(Z, \alpha_0)$ with alphabet $[0, 1]$ which also accepts some tapes if and only if Z halts from α_0 . For any automaton A , $P(A)$ diverges under all interpretations if and only if A never accepts a tape. Denoting $P(A'(Z, \alpha_0))$ by $P(Z, \alpha_0)$, we remark that:

$P(Z, \alpha_0)$ is equivalent to D if and only if
 Z fails to halt from α_0 ,

where D is the schema of figure 2, which always diverges. There is no effective (partial) procedure which tells us, for arbitrary Z, α_0 , when Z fails to halt from α_0 , and so the equivalence problem for schemata is effectively unsolvable and indeed not partially solvable.

4. FURTHER RESULTS

4.1. Other relations between schemata

Because we are interested in program schemata as a model of computation, it might be more appropriate to restrict ourselves to interpretations in which all the functions and predicates are recursive, or to interpretations which can be defined over a finite set of values. We then obtain the relations of *recursive equivalence* and *finite equivalence*. Thus two schemata are recursively equivalent if they have the same values for all recursive interpretations. The interested reader may like to verify that these relations are all different. A rather weaker relation, misnamed *weak equivalence*, is obtained by considering for each pair of schemata only those interpretations under which both of them halt. The unsolvability of each of these relations can be proved. Indeed we can

prove, at one stroke, the unsolvability of any relation lying between equivalence and weak equivalence.

4.2. Solvable classes of schemata

Having established the unsolvability of the equivalence problem for a surprisingly simple class of schemata, we have a better vantage point from which to survey the scene and find interesting solvable classes. The single test, single monadic function and two locations of our unsolvable schemata allow little scope for further restriction. However, we can solve the equivalence problem for schemata with only a single location, which are then similar in structure to ordinary finite automata and the 'logical schemes' of Ianov (1958 and 1960).

By using the unsolvability of the halting problem of a fixed, 'universal', Turing machine (see Davis, 1958) presented with arbitrary initial tapes, we construct an unsolvable class of schemata which all have the same loop structure. Thus it is not necessary for unsolvability for the schemata to have arbitrarily complicated flow structures, but it is an open problem to find just how simple a structure will suffice. Apart from loop-free schemata, we have a decision procedure for schemata in which no two loops are 'nested' or intersect, but subject so far to the restriction that all the functions be monadic.

One of the more striking features of the 'unsolvable' schemata we constructed is that they duplicate their calculations and tests in a way we would be loath to recognize in our computer programs. This suggested closer study of *free* schemata, which never test the same value twice, and *liberal* schemata, which never calculate the same value twice. For any liberal schema we can construct an equivalent free schema, but not vice versa. The schema in figure 12

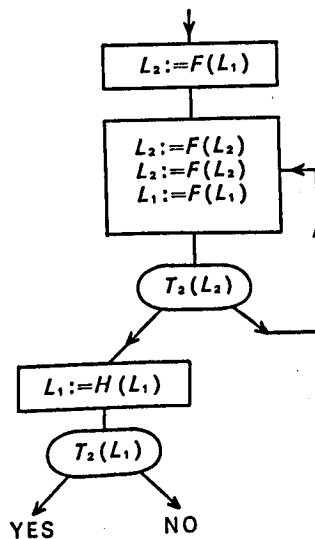


Figure 12

is free but 'essentially illiberal'. A further distinction is that whereas we can effectively decide whether or not a schema is liberal, the decision problem for freedom is unsolvable.

If we impose the fairly restrictive condition on schemata, that in any execution sequence any value that is computed is used as an argument by the next assignment instruction, the resulting schemata are clearly liberal and we have obtained a decision procedure for their equivalence. The solvability of the equivalence of liberal schemata remains an important open problem. We see that, if the first two instructions of the 'simulation' schemata are deleted, they are then liberal and simulate two-tape finite automata, where the two heads read two different tapes.

The equivalence problem for two-tape automata, that is, whether two given automata accept the same set of pairs of tapes, is a well-known open problem in automata theory. Its unsolvability would imply unsolvability for the equivalence of liberal schemata, but we believe it to be solvable, in which case the solution should lead us to new solvability results for wider classes of schemata.

Acknowledgments

This report is the first published account of work done by the author at Cambridge in 1965-7. Further results and proofs are to be found in Paterson (1967) and in Luckham, Park and Paterson (in preparation) The author wishes to acknowledge the help and encouragement given by his supervisor, Dr D.M. Park, during the course of this work.

REFERENCES

- Davis, M. (1958), *Computability and unsolvability*. New York: McGraw-Hill.
- Ianov, Iu (1958), The logical schemes of algorithms. *Problems of cybernetics I*, pp. 75-127 (Russian edition).
- Ianov, Iu (1960), The logical schemes of algorithms. *Problems of cybernetics I*, pp. 82-140 (English translation). Oxford: Pergamon Press.
- Paterson, M. (1967), *Equivalence problems in a model of computation*. Dissertation submitted for Ph.D. Cambridge.
- Rabin, M. & Scott, D. (1959), Finite automata and their decision problems. *IBM JI Res. Dev.*, 3, 114-25.
- Rosenberg, A. (1966), On multi-head finite automata. *IBM JI Res. Dev.*, 10, 388-94.

Language Definition and Compiler Validation

J. J. Florentin

Centre for Computing and Automation
Imperial College of Science and Technology
London

Abstract

A method is proposed for checking that a compiler translates in accordance with the user language definition. It involves formalizing the compiler action into a series of automata models, each with a state transition table. A finite number of input-output experiments are then to be done to recover the state transition tables, and these are compared to the specifications. In a language with operating system facilities, details of the operating system must be included in the automata states. The problems arising here are briefly discussed.

1. INTRODUCTION

This is an account of some initial thoughts on compiler validation—that is, verifying that a particular compiler translates source programs in accord with the language definition. It is essential to solve this problem for the following kinds of reasons:

(i) Language standardization committees are faced with the question—what is the use of a language standard if there is no effective way of guaranteeing that a particular compiler meets the standard?

(ii) There exist many schemes where the same language must be run on different machines. For instance, a 'computer grid' scheme has been put forward. At present such schemes are unworkable because small incompatibilities in compilers for the same language on different machines would not allow the same program to be run on any machine in the grid.

(iii) Any theory of software must be based on linking together measurable quantities. Progress, both theoretical and practical, depends on rigorous specification and measurement. Compiler validation is one step towards this.

1.1. The technical problem

Validation may be expressed technically in several ways. The most comprehensive view is that of McCarthy and Painter (1966). They define two formal systems: one describes source programs and the other the machine. In describing the source language evaluation each possible, correct, source program is taken as an initial axiom. Fixed rules then transform the program to its value relative to the store contents. The machine formal system starts with empty registers, accumulator, and program counter. This initial state is transformed by rules which depend on the current machine language program. The compiler now translates between these formal systems by taking a source language axiom across to a machine program which acts as a transformation rule. For a particular language they show that the two systems produce the same terminal value from programs which are compiler equivalent.

In this paper a much simpler problem is posed. Although simpler, it is adequate for many practical purposes, such as those mentioned in the introduction. A language is defined in the usual way by its action on an interpreter automaton. Then we ask if the compiler produces analogous effects in its machine when translating source program. A useful answer appears to be possible when the language is highly structured and when the compiler is written in a very systematic way.

This approach makes it necessary to systematize language semantics, and some suggestions are put forward here.

1.2. Scope of language

The facilities included in high-level languages have increased greatly in the past few years. Early languages included facilities for doing arithmetic only; recently, languages such as PL/I have included the handling of large data files, control over interrupts to the operating system, and running of a hierarchy of concurrent tasks. It is clear that eventually compiler validation must include means of verifying that these run-time facilities are correctly implemented. A brief discussion of this is given.

2. BASIS OF SUGGESTED METHOD

The real work in validation is the reduction of the compiler to a form where its essential action can be reconstructed by a finite number of input-output experiments. One way of approaching this is to specify the various translation and interpretation processes within the compiler as formal automata. The essential action of these devices is then contained in a state transition table. The input-output experiments then have to recover these state transition tables. Here the term output means any way of extracting a trace of the program action, that is a record of store changes, jumps, and subroutine calls.

To form an automaton model of program action the *state* must first be identified. This will contain a neat layout of all stored data, the current instruction number or subroutine being called and other similar information.

Then the state transition table must be formed. For this the language definition must be used. The number of input-output experiments needed and the size of the table depend on the language definition, and hence on any structure within it. Strong structures make validation simpler, besides making the definition easier to understand.

2.1. A simple illustration

The simplest example which will show the above points is a syntax recognizer or acceptor. This only indicates whether an input word is actually in the language. It is slightly less complex than a compiler component, and here is just given in outline.

The acceptor is for a simple precedence syntax (see Wirth and Weber, 1966). The language is generated by a context-free grammar, and has the extra structure that for every pair of symbols which can appear in a sentential form a precedence $-1, 0, 0, 1$ is given. In a sentential form each string of symbols which forms the right-hand side of a production, and which may be reduced to a single symbol, can be found by looking at successive precedences for each pair of symbols in the string. A succession of precedences like $-1, 0, 0, 1$ shows the presence of a reducible string. The quoted reference gives a more complete explanation.

The main storage of the automaton is made up of three strings:

in holding the input

temp which is temporary storage

out which builds up the right hand side of the production

The action of the acceptor is first described by a program depending heavily on subroutines. There are two main routines—*forward* and *back*. In addition the routine *prec* yields the precedence, and *prod* finds the required left-hand side of a grammar production. Only an outline program is given, and, for instance, no error detection in the input string is considered.

```

forward(in, temp, out) = if null(in) then STOP
                        else
                          if prec(hd(in), hd(temp)) ≥ 0 then
                            forward(tail(in), hd(in) || temp, out)
                          else
                            back(in, temp, out)
back(in, temp, out)   = if null(out) then back(in, tail(temp), hd(temp)
                                                                || out)
                        else
                          if prec(hd(temp), hd(out)) = 0 then
                            back(in, tail(temp), hd(temp) || out)
                          else
                            forward(in, prod(out) || temp, nil)

```


MATHEMATICAL FOUNDATIONS

The state of the corresponding automaton is given by the quadruple

$$(hd(in), hd(temp), hd(out), Q)$$

where Q is a state variable having two values—forward or back—depending on which subroutine is in force.

Given the precedence table and the grammar, the state transition table of the automaton can be developed. This could be done by a computer. By using a trace routine which records the changes in the given state components the transitions caused by all possible inputs could be checked. A note on reducing the number of states to be checked is made below. Hopefully, the trace routine could be made error-free.

A second version of the acceptor will now be given with sequential imperative statements. Simultaneous assignments will be used. Variables i, t, o are special registers for subscripting the stores $in, temp$ and out , which now appear as vectors.

1. $o, t, i \leftarrow 1, l, n$
2. if $null(in)$ then goto 12
3. if $prec(in(i), temp(t)) \geq 0$ then goto 8
4. $out(1), t \leftarrow temp(t), t-1$
5. if $prec(in(i), out(o)) = 0$ then goto 10
6. $t, temp(t+1), o \leftarrow t+1, prod(out), 1$
7. goto 2
8. $t, temp(t+1), i \leftarrow t+1, in(i), i-1$
9. goto 2
10. $o, out(o+1), t \leftarrow o+1, temp(t), t-1$
11. goto 5
12. STOP

The automaton state corresponding to this program is the 9-tuple

$$(o, t, i, pred, prod, out(o), temp(t), in(i), progcount)$$

$prod$ and $pred$ are special registers whose contents are undefined when the corresponding subroutines have not been called; $progcount$ is a special register containing the current instruction number.

Deriving the state transition table is more complicated here. Also, the question arises—how many values of i, t , and o should be tried in experiments? At this point the array feature of the acceptor writing language is being tested. This kind of difficulty comes up constantly. To some extent the array facility must be tested; however, it will be noted that it can be tested separately from the action of the acceptor program. An analogous question arises in the previous program, for the fact that the recursive subroutine mechanism works without fault also needs to be checked. Again, the action of the subroutine

facility can be checked independently of the program. Actually, some imprecision will probably be accepted here, and checking will only be done to a moderate depth of facilities.

2.2. Cutting down the number of experiments

Usually only a sub-set of possible states in the automaton can be attained when the compiler is driven by a correctly written program. To cut down the number of input-output experiments it is possible to check only the attainable states. As an example, in the simple precedence acceptor above, it is quite usual to find that the precedence matrix is only sparsely filled in, and only a limited number of triples ($hd(in)$, $hd(temp)$, $hd(out)$) can appear for a syntactically correct input string.

Of course, the chosen method of language definition and program writing affects the amount of experimenting to be done. Little is now known of these important factors.

3. SEMANTIC STRUCTURES

The need for validation leads to a study of semantic structures. Just as syntactic structures can be associated with particular parsing automata, so a semantic structure can be specified by the construction, and action, of an interpreter automaton. An outline example will be given to illustrate this.

3.1. A stack interpreter

This semantic interpreter is based on Dijkstra's ALGOL compiler (see Dijkstra, 1963). It will accept, roughly, the class of postfix languages. The way in which the form of the input language is restricted is discussed later. The interpreter is diagrammed in figure 1, and contains

- (i) a controller to initiate reading of successive input program symbols followed by appropriate action;
- (ii) an evaluator for the primitive functions of the language. Primitive function evaluation has no side effects, and leaves the arguments unaltered;
- (iii) a stack store for unravelling functional composition;
- (iv) a table to effect the naming facilities in the user language, and further to allow address manipulation and instruction labelling;
- (v) a data store with access to it controlled by the table.

Symbols from the user input program string are read into the stack and then interpreted by reference to the tables. The stack may be read, and written to, at the top, but may only be read in the interior. It handles names rather than actual data. It is very well known that functions with arguments written in postfix notation, like $xy+$, can be evaluated within the linear discipline of the stack. The effect of restricted store access can be seen more strongly in functional composition. For instance, the composition shown in figure 2a cannot be handled in a stack, whereas that of figure 2b, which yields the same result,

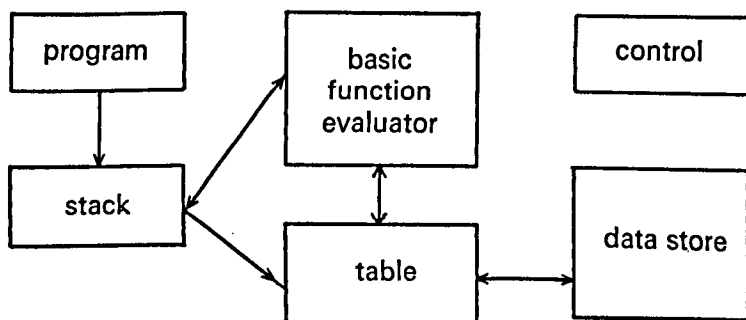


Figure 1. Semantic interpreter

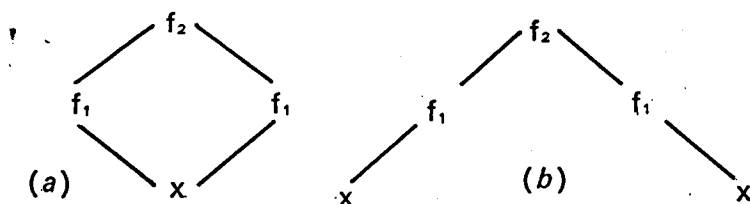


Figure 2 (a) and (b)

can be. In figure 2 the symbol f stands for a primitive function, and x for an argument. Thus, the ways in which the user can express functions are controlled by the stack, and these restrictions are passed to the user by the language syntax. In this way syntax is critically linked to semantics.

The table links symbols in the user, or rather postfix, program to their interpretation as functions or data names. The search algorithm for the table must be specified to allow multiple naming, and restricted scopes of names as in ALGOL. It could be useful to let the user manipulate table entries directly, and thus allow instructions like $a \equiv b$ to mean that a and b both name the same storage location. Other address manipulation can be regarded in the same way.

A detailed specification of the operation of the data store must include a statement of the varieties of data representation possible, whether program can be held, and if the links for related data, as in a tree, are kept within the store or the table.

The possible elaborations of the user language by fully using the facilities of an interpreter of this kind is under investigation. Notice that the precise nature of the primitive functions is not so important as the methods of composing them.

4. COMPILER COMPONENTS

A compiler will have several components, like:

1. A pre-processor to convert user programs to internal computer form, and also make user defined names into special symbols. The validation automaton to be associated with this will not come directly from the language definition because it involves the special characteristics of the card punches or other devices. It is likely that it could be treated as a finite automaton, which would be a simplification.
2. A translator from the internal user program to a fully parsed tree form. The automaton here can be derived directly from the syntactic part of the language definition. It might typically be a pushdown device to handle the syntax specified by a context-free grammar, plus an extra store to handle requirements such as the difference of labels in the same block. This is the easiest part to treat at present.
3. A translator from the parsed tree form to machine code. This should be based on the semantic interpreter given in the language definition. Besides modelling the analysis action of the interpreter it is additionally necessary to check that the machine code corresponds to the changes in interpreter machine state. This will be made easier if the generator blocks of machine code are carefully constructed to correspond to interpreter machine actions.
4. Various optimization routines intended to produce more efficient machine code. These act at different stages of compiling. Some act directly on the parsed tree form, while some act directly on the machine code produced by the last translator. These will be a serious problem unless they can be put in at a distinct stage of compiling. For instance, if the parsed tree can be put through an optimizer before translation, and then the machine code can be put through a second optimizer, then the two optimizers act exactly like extra stages of translation. Unfortunately, little is known about systematic optimization.

A study of compilers is being undertaken in order to carry through the systematization of each stage as indicated above, but substantive conclusions are not available yet.

A systematic examination of the components of a compiler will bring a deeper understanding of its action. Also, it will lead to a study of the relation between semantics and syntax, which will lead to a better appreciation of the possibilities in language design. One aspect of this would be a determination of semantic similarities with only syntactic differences. The importance of reversible, or information lossless, syntactic transformations has been pointed out to the author by Mr R.A. Francis and is under investigation.

5. OPERATING SYSTEM FACILITIES

Modern languages have the capability of initiating concurrent tasks which can communicate with each other. This makes the framework of compiler

validation change. The viewpoint shifts from that of the *program* to that of the *process* within a *computing system*.

A process is an interpreter which may be obeying any designated sequence of program and referring to any data available in the computing system. Processes can bring into being new processes, and processes are removed when no longer needed. At any time the computing system consists of a number of processes in an orderly hierarchy, and stores of data and program which can be accessed by the processes. Processes can refer to the same data or program simultaneously. Rules must be set up to control the initiation and termination of processes, and to set conditions on access to the stores by each process.

Internally the process could be very like the previous semantic interpreter. But it must have a number of extra components to reflect the working of the operating system. Examples of extra state components required by a process are:

- (i) the identity of the program it is obeying;
- (ii) the parts of the data stores it has access to, and under what conditions;
- (iii) the status of any lower level processes it has initiated;
- (iv) the status of any process with which it has to communicate.

Notice that it is not necessary to include the status of the parent process, since it cannot be affected by commands in the current process.

Whenever operating system instructions are written into a program their effect must be validated by the transitions induced in these extra state components. An obvious approach to validation would be to separate, as far as possible, commands which affect the operating system, and those which affect ordinary problem actions. This is now being considered.

6. CONCLUSIONS

The method proposed for compiler validation is first to divide the compiler action into a number of independent stages. Then the action of each stage is formalized as an automaton having an input-output behaviour controlled by a state transition table. After the compiler has been written, a finite number of input-output experiments are done to recover the state transition tables. These measured state transition tables are then compared to the specified ones.

This proposal leads back to a study of language definition and the design of languages with strong syntactic and semantic structures. When operating system facilities are included in the user language the number of components of the formal automata states has to be increased. The rules governing the operating system now are needed to find the state transitions.

Currently the Language Definition Group at Imperial College are carrying out a study of syntactic and semantic structures, and are beginning to consider the notion of state components due to the action of the operating system.

Acknowledgment

The work described in this paper was carried out as part of a project supported by IBM United Kingdom Laboratories Ltd.

REFERENCES

- Dijkstra, E.W. (1963), An ALGOL 60 translator for the *Annual Review of automatic programming*, 3, pp. 27-42 (ed. Goodman R.). Oxford: Pergamon Press.
- McCarthy, J. & Painter, J. (1966), Correctness of a compiler for arithmetic expressions. *Computer Science Technical Report No. CS38*, Stanford University.
- Wirth, N. & Weber H. (1966), EULER-A generalisation of ALGOL, and its formal definition. *Comms Ass. comput. Mach.*, 9, 13-25.

Placing Trees in Lexicographic Order

H. I. Scoins

Computing Laboratory
University of Newcastle upon Tyne

BACKGROUND AND MOTIVATION

The ideas described in this paper are part of a project, being conducted by the author and C.R. Snow, which is aimed at forming a general, but efficient, algorithm to prove or disprove that two given linear graphs of n points are isomorphic.

To the pure mathematician the problem is trivial. He says: Consider all the $n!$ possible permutations of the labels of one of the graphs. If for one, or more, of the permutations the two graphs match then they are isomorphic, otherwise not.

The worker with a computer background says: $n!$ is a very large number. Even our largest and most powerful computers have little hope of making the number of straightforward comparisons appropriate for $n=20$ (say). Therefore, we must attempt to impose some classification or structure on the set of all possible linear graphs, so that most of the comparisons need not be made explicitly and only a small subset of particular comparisons made.

To illustrate that there is a problem consider the three linear graphs of 6 points and 11 lines in figure 1. It is not immediately obvious that two are

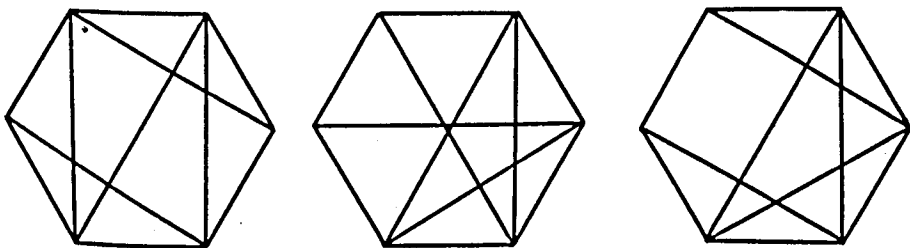


Figure 1

isomorphic and one is on its own, yet each has a partial specification of 4, 4, 4, 4, 3, 3, i.e. there are four points with four lines each and two with only three. If you know the answer you may be able to find reasons, but if the isomorphism is unknown the problem is much more difficult.

There have been other attempts to tackle this problem which some think is in general impossible. Perhaps all that can be hoped for is something like an algorithm which will always work for $n=100$ (say) in less than one hour on a computer costing one million pounds. Even the simplest algorithms will prove most pairs of graphs dissimilar quite quickly. The difficulties arise when they are isomorphic, but have to be proved to be so.

The problem has been considered in part by workers interested in minimizing the band-width of a matrix so as to enable a computer to solve a corresponding set of simultaneous equations in a compact form. (Alway and Martin, 1965; Obruca, 1966).

Following a rather different approach is the work of Unger (1964) and of Read and Parris (1966). The latter has ideas similar but not exactly the same as those expounded here.

The long view we take of this problem is that one can handle a tree which spans a linear graph much more readily than the whole graph. If it is possible to determine a unique spanning tree which is independent of the labelling of the graph under consideration, the problem of the comparison can be reduced to first finding those spanning trees, comparing them and if these skeletons are isomorphic, then moving on to compare the flesh of the two co-trees (i.e. the two sets of lines not included in the spanning tree).

In this contribution the problem of the isomorphism of two trees is considered, as well as the possibility of forming an index which can be used to describe the tree uniquely when its labels are ignored.

THE BINARY ROOTED ORDERED TREE

The simplest and probably the most common type of tree used in computer work is the binary, rooted, ordered tree which carries information of some sort at its extremities or terminals. This is the type of tree involved in 'list-processing', if one excludes cyclic list structures.

In such a tree every node is either a terminal node, or else it has just two nodes 'above' it. Each node has one node 'below' except the 'root' which has no 'below'. Figure 2 (a) gives an example of such a tree, having seven terminal nodes, five internal nodes and the root R . The fact that the tree has been drawn out on paper imparts an ordering on the upward branches so that we can meaningfully refer to the left and right branches at any non-terminal node.

Occasionally it is useful to separate the root node into two separate nodes to distinguish between the whole tree which stands on R and the point at which this tree forks into its two main branches. This is illustrated in 2 (b) and in this case the tree is said to be 'planted' on R .

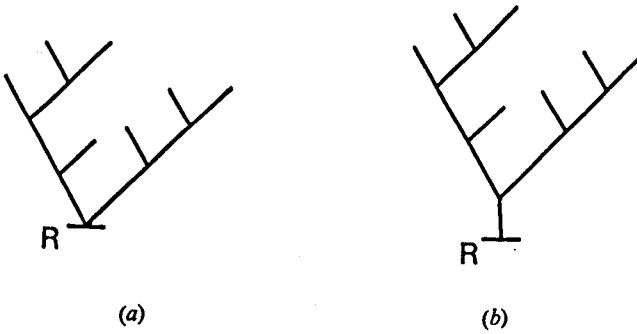


Figure 2

Suppose that, in our example, we have seven elements, A to G, which we attach to the terminals of the tree, and suppose we label the non-terminal nodes 1 to 6 (figure 3). Then the numbered nodes represent the various sub-sets and sub-sets of sub-sets of the set A to G. The whole set has been split into

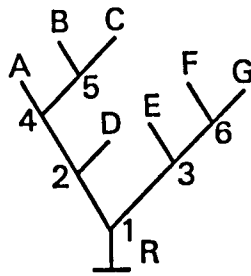


Figure 3

two parts, each part being split into two and then split again recursively as long as there is more than one element in any sub-set so formed. Thus the tree represents a 'complete' partition of A to G with the retention of order, so that splitting into (A, B, C, D) and (E, F, G) is different from splitting into (E, F, G) followed by (A, B, C, D). If this ordering is ignored we have a different and much smaller set of possibilities.

The number of different trees of the ordered type with seven terminals is 132, rather too many to display; but with four terminals, for example, there are five cases, illustrated in figure 4. It is the placing of such a set of trees in a lexicographic order that we wish to examine.

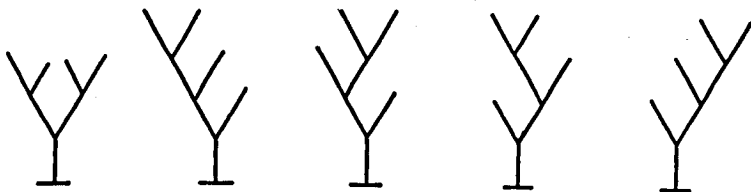


Figure 4

THE GENERATING FUNCTION APPROACH

The counting or generating function for these trees is:

$$\mathcal{T}(x) \equiv \sum_{n=1}^{\infty} t_n x^n = x + x^2 + 2x^3 + 5x^4 + 14x^5 + 42x^6 + 132x^7 + \dots,$$

where t_n is the number of binary ordered rooted trees with exactly n terminals. $\mathcal{T}(x)$ satisfies the equation $\mathcal{T}(x) = x + \{\mathcal{T}(x)\}^2$ (Scoins, 1967) which implies that asymptotically $t_{n+1} \simeq 4t_n$ and that $t_n \simeq 2^{2n-7}$ is a very rough approximation for $n \geq 7$.

This functional equation satisfied by the generating function, $\mathcal{T}(x)$, can be used to form an index number for each tree of the form (n, r) which indicates that a particular tree has n terminals and is the r -th within that partial specification, when the t_n cases are placed in sequence according to certain rules. Moreover, this index number is an efficient representation of the tree, and not just a reference number of some stored list of some other representation.

The quadratic equation implies (but is usually derived from)

$$t_n = \sum_{s=1}^{n-1} t_s t_{n-s}, \quad n > 1,$$

which is only another form of the statement that a tree with n terminals necessarily consists of a composition of a tree with s terminals as left part, and a tree of $n-s$ terminals as right part, s being permitted all values from 1 to $n-1$. This permits us to divide up the set, T_n , of the t_n possibilities into $n-1$ sub-sets, each of which is a product set of all the t_s possibilities in T_s taken with each and every possibility in T_{n-s} . If the product sets are considered in the order of increasing s , and if within each product set we group under fixed left part while taking the right part possibilities in their proper order, we have a valid numbering system.

For example, if we take the example in figure 2, its reference number is $(7, r)$. The 132 cases in T_7 are taken as six product sets, i.e.

$$T_7 = T_1 \times T_6 + T_2 \times T_5 + T_3 \times T_4 + T_4 \times T_3 + T_5 \times T_2 + T_6 \times T_1$$

which contain $1 \times 42, 1 \times 14, 2 \times 5, 5 \times 2, 14 \times 1, 42 \times 1$ cases respectively.

Our example is in the product set $T_4 \times T_3$ since the left part has four terminals, and therefore $66 < r \leq 76 (=42+14+10+10)$.

The right part is in $T_3 = T_1 \times T_2 + T_2 \times T_1$ and specifically it is the one element in the product set $T_1 \times T_2$. Thus the right part has reference (3, 1).

The left part is in $T_4 = T_1 \times T_3 + T_2 \times T_2 + T_3 \times T_1$ and is in $T_3 \times T_1$. The particular case in T_3 is the first of two and so the left part has reference (4, 4).

Thus $r = 66 + 3 \times 2 + 1$, where the 3×2 is accounted for by those cases in $T_4 \times T_3$ which have earlier left part, and the 1 by the fact that the right part has reference (3, 1).

In general, (n, r) is a concatenation or composition of (s, p) and $(n-s, q)$ and these are related by

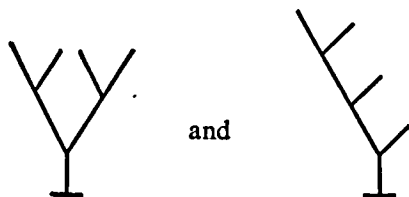
$$r = \sum_{k=1}^{s-1} t_k t_{n-k} + (p-1) \times t_{n-s} + q.$$

The five trees displayed in figure 4 are in the sequence (4,3), (4,5), (4,4), (4,2), (4,1).

With this reference system one can calculate r for a given tree or alternatively deduce the structure of the tree given only (n, r) and a table of t_i ($1 \leq i \leq n$). Thus we have been able to give each tree a two-component reference number; in other words, we have enumerated or numbered-off the set and can also use this reference together with a simple algorithm to deduce the structure of the tree. In subsequent discussion we shall see that other representations produce other lexicographic orders and that the index number, i.e. the number found if we number-off the set 1, 2, 3, ..., usually cannot be used without a dictionary containing all or most of the trees in that particular representation sorted into the appropriate proper order.

THE NON-ORDERED BINARY ROOTED TREE

If we discard the order property of our binary trees, we have, in the case of four terminal nodes, only the two cases



the second of these two being isomorphic with four of the trees illustrated in figure 4.

The counting series for this species of tree is:

$$\mathcal{R}(x) \equiv \sum_{n=1}^{\infty} r_n x^n = x + x^2 + x^3 + 2x^4 + 3x^5 + 6x^6 + 11x^7 + \\ + 23x^8 + 46x^9 + 98x^{10} + \dots$$

MATHEMATICAL FOUNDATIONS

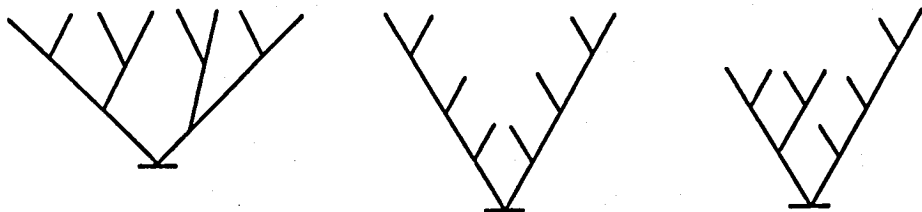
This generating function does not satisfy such a simple functional equation as $\mathcal{T}(x)$ discussed earlier, and the coefficients are best considered as formed by:

$$r_n = \sum_{s=1}^{[n/2]} r_s r_{n-s}.$$

Here the upper limit is the integer not greater than $n/2$ and there is a special modification to be applied to any term $r_{n/2} \times r_{n/2}$ for even n . The recurrence relation essentially says, if there are r_s possible cases in the left part and r_{n-s} cases in the right, we cover all possibilities once only by the restriction $s \leq n-s$. However, in the case $2s=n$ (even) we have to count only once the different possibilities and the corresponding number of cases is

$$r_{n/2} \times (r_{n/2} + 1) / 2.$$

Thus, in r_8 the contribution from $r_4 \times (r_4 + 1) / 2$ consists of



Although a functional equation which the generating function satisfies can be found, it does not seem to be a satisfactory basis for putting the trees in sequence, and this applies generally to the other types of tree considered here.

It is interesting to note that the counting series described here is an especially simple sub-problem of that considered by Riordan and the counting series is at first sight the same as that given by Riordan (1958) for unlabelled trees, which are discussed in much more detail by Harary and Prins (1959). In fact the counting series agree up to and including the eighth coefficient and only then become distinct—an interesting example of the traps that lie in wait for those who study correspondences between sets. The trees discussed here are also related to the distinct possibilities of instantaneous binary codes to represent sets of n distinct elements, when the probability of the occurrence of these distinct elements is non-uniform (see, for example, Abramson, 1963).

A MINIMAL REPRESENTATION OF A TREE— THE TERMINATED BINARY SEQUENCE

From what has gone before, it is clear that a binary tree is essentially something which, unless it is fundamental and indivisible, splits up into two parts in a unique manner. If the two parts are always interchangeable without loss of significance we have a non-ordered tree, but when we must make a distinction

between the two parts, we are considering an ordered tree. These ideas go far beyond the simple pictorial concept used up to this point and the following is a wide but useful attempt to define a general class of binary ordered rooted trees.

Let R represent a set of independent and distinct elements which are essentially indivisible (atoms), and let S be a set of different ways of associating together any two of these elements or constructs from them. A class of trees T can be defined on R and S such that:

- (i) All $r \in R$ also $\in T$.
- (ii) If $t_1, t_2 \in T$ then the construct, whatever the construction rule may be, is formed by associating t_1, t_2 as an ordered pair, with some $s \in S$. This construct also $\in T$.
- (iii) If $t_3 \in T$ then either $t_3 \in R$ or $\exists s \in S, t_1$ and $t_2 \in T$ such that t_3 is identically the construct of t_1, t_2 and s . Moreover this decomposition *must be unique*.

If the background sets R and S are the identifiers of independent real variables $a, b, c, \dots$ and the set of binary operators $+, -, \times, /$, together with brackets to define the composed form, we see that bracketed algebraic expressions are trees in the above sense. However, this class of trees is of wider import. The manufacture of a ship or a car consists of the putting together of parts, although after assembly the sequence of decomposition is not obvious unless one has a record of the construction sequence.

Clearly by introducing the unique decomposition we may be forced to distinguish between two forms which are entirely equivalent. Who cares in which order the wheels were put on the car?

An interesting example for the reader to study is the various expressions which can be formed by the division operation $/$. With brackets we have the construction rule, in Backus normal form,

$$\langle \text{expression} \rangle ::= \langle \text{identifier} \rangle \mid (\langle \text{expression} \rangle / \langle \text{expression} \rangle)$$

which can always be uniquely decomposed. It is a separate matter that $((a/b)/(c/d))$ and $((a/c)/(b/d))$ are arithmetically equivalent, but that is another big problem and not to be discussed here.

At the very simplest level, let the set R contain 0 as its only element and let the set S contain just the symbol 1. The composition rule is expressed as

$$\langle T \rangle ::= 0 \mid 1 \langle T \rangle \langle T \rangle.$$

This is a prefixed form and the representation of the 'tree' is a sequence of 0's and 1's. The zeros correspond to the terminal nodes of the binary tree and the ones to the forks at the non-terminal nodes. Following this correspondence the five trees in figure 4 correspond respectively to

1100100, 1110000, 1101000, 1011000, 1010100.

In general, a tree of n terminal nodes is represented by an arrangement of n

MATHEMATICAL FOUNDATIONS

zeros and $n-1$ ones such that if one starts at the left-hand end of the sequence and keeps a running total of the number of zeros and the number of ones encountered, the binary sequence is terminated, and the tree is completed, as soon as the number of zeros exceeds the number of ones by unity; hence the 'terminated binary sequence'.

Taking any binary sequence, starting at an arbitrary position and moving from left to right through the sequence, we must inevitably find some sequences which terminate in the above sense. If the sequence is finite it may contain at its right-hand end an incomplete tree, but in general it represents a forest.

To emphasize the correspondence between the pictorial tree and the binary representation, Table 1 indicates the correspondence between the labelled tree in figure 3 and its binary representation.

binary digit position	1	2	3	4	5	6	7	8	9	10	11	12	13
binary digit	1	1	1	0	1	0	0	0	1	0	1	0	0
corresponding node in figure 3	1	2	4	A	5	B	C	D	3	E	6	F	G

Table 1

The terminated binary sequence (t.b.s.) starting at position 1 represents the whole tree, while the subsequences commencing at positions 2 and 9 represent the two main branches, and themselves are properly terminated sequences.

The number of distinct ordered trees with seven terminals is 132, requiring a minimum of 8 bits for representation purposes, so that the 13 bits required is not absolutely minimal in the sense of information theory. However the t.b.s. in general requires $2n-1$ bits, while the number of trees increases by roughly a factor of 4 as n increases by one. Hence one has good value from the bits used in the t.b.s. especially as n increases.

The terminated binary sequence is thought by the writer to be a minimal representation for an unlabelled binary ordered rooted tree, but it should not be thought to be necessarily convenient for manipulation purposes. One can construct from it by fairly simple algorithms such quantities as the number of terminals in a given sub-tree and thence get back to the type of identification discussed earlier. One can set up cross references so that one can immediately tell, for example, that node 2 has right neighbour 3 and node 3 left neighbour 2, or construct other forward and backward pointers. However, rather more computer storage would then be required and might not be appropriate if a large set of trees had to be kept in computer storage.

Although we shall not consider it here, there is also a post-fixed form of binary representation based on the rule:

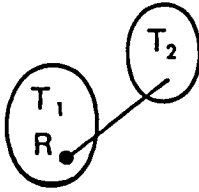
$$\langle S \rangle ::= 0 | \langle S \rangle \langle S \rangle 1,$$

corresponding to the reverse-Polish command structure of machine code programs.

THE ORDERED ROOTED MULTIFURCATING TREE OF n NODES

The general ordered rooted tree may have any number of nodes (≥ 0) above a given node and any such will possess a sense of order. If we make use of our general definition of a tree and use a convenient combination rule we can immediately establish that the multifurcating trees with n nodes can be mapped one-to-one into the binary trees with n terminals.

Let T_1 and T_2 be two ordered rooted trees of p and q nodes respectively. Define the composition of T_1 and T_2 by taking T_2 and planting it on a stalk on the root of T_1 and to the right of T_1 . Pictorially the figure below is the composition with root R , and total nodes $p+q$. (see Scoins, 1967).



Thus the terminated binary sequence can also be considered as a representation of an ordered rooted tree of n nodes. The correspondence chosen here (other composition rules are possible) has some especially interesting features, which are brought out by figure 5, which corresponds to the t.b.s. given in Table 1 with the same labels.

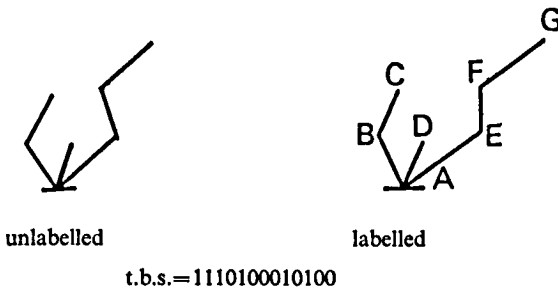


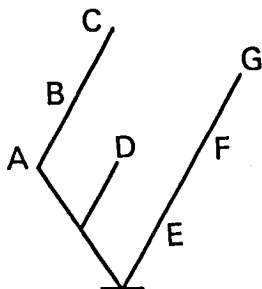
Figure 5

The sub-trees numbered 1 to 6 in figure 3 and in Table 1, correspond in figure 5 to

- 1 whole tree on A,
- 2 tree with extreme right branch EFG deleted,
- 3 sub-tree standing on E,
- 4 sub-tree ABC, on A,
- 5 sub-tree BC on B,
- 6 sub-tree FG on F.

MATHEMATICAL FOUNDATIONS

What has been done can be described in terms of a kind of 'condensation' process carried out on figure 3, where the figure below is an intermediate stage; but if one tries to use this technique to establish correspondences with pencil and paper by drawing, one seems to make rather more mistakes than one would like.



It is however worth noting that

- (i) a group of k 1's in the t.b.s. corresponds to a k -way fork,
- (ii) the zero which terminates that run of k 1's corresponds to the node at which the k -fork occurs,
- (iii) the node sequence corresponding to the left to right order of the zeros in the t.b.s. is given by the algorithm:

start at the root, go up left, labelling as you go; when you can't go up any more, come back down and go up again as soon as possible, taking as left a path as possible without arriving by an upward movement at a node that is already labelled. Label in natural order only when ascending.

SEQUENCING BY RECURSION AND BY BINARY VALUE

If one wishes to construct all the ordered rooted trees for given n by computer program one can employ an approach which generates the whole set of T_n in some systematic order via some representation and this order of generation can be considered a type of lexicographic ordering.

The two ALGOL procedures described in this section both form the terminated binary sequence for each tree in an integer array $A[1:2 \times n - 1]$ and carry out the operation $Q(n, A)$, which might be just printing the tree in some form, as each tree is formed.

The first, *RECTS1*, forms the t.b.s. by entry to a higher level recursive procedure *recTS* which does the main work and forms the t.b.s. in the sequence given by the number pair representation (n, r) discussed earlier.

procedure *RECTS1*(n, A, Q); **value** n ; **integer** n ;
 integer array A ; **procedure** Q ; **comment** $Q(n, A)$ acts on each
 completed tree, $A[1:2 \times n - 1]$;

```

begin integer array  $K[1:2 \times n - 1]$ ; integer  $m$ ;
procedure  $recTS(k, i)$ ; value  $i, k$ ; integer  $i, k$ ;
if  $k = 1$  then begin  $A[i] := 0$ ;
    if  $i = m$  then  $Q(n, A)$  else  $recTS(K[i + 1], i + 1)$ ;
end
else begin integer  $p$ ;
    for  $p := 1$  step 1 until  $k - 1$  do
        begin  $A[i] := 1$ ;
             $K[i + 2 \times p] := k - p$ ;
             $recTS(p, i + 1)$ ;
        end
    end  $recTS$ ;
 $m := 2 \times n - 1$ ;  $recTS(n, 1)$ ;
end  $RECTS1$ ;
    
```

The function of the integer array K is to record the number of nodes to be allocated to the right part of the tree while the left part is being formed.

This algorithm forms the five trees of four nodes in the sequence 1010100, 1011000, 1100100, 1101000, 1110000. These are in ascending order when considered as binary integers, but if the reader forms for himself the 14 cases for $n = 5$, he will find that one of the fourteen is out of place relative to numerically increasing value of the t.b.s.

The second algorithm uses the rules which the t.b.s. must satisfy to be a legitimate form and produces the t.b.s. for given n in numerically descending order. (A minor adjustment will produce the reverse ascending order). The algorithm is given to illustrate the point that each representation has its own natural lexicographic or reversed lexicographic sequence of trees.

```

procedure  $TBS(A, n, Q)$ ; value  $n$ ; integer  $n$ ;
    integer array  $A$ ; procedure  $Q$ ;
begin integer  $r$ ;
procedure  $tbs(ones, i, ex)$ ;
    value  $ones, i, ex$ ; integer  $ones, i, ex$ ;
    if  $ones = 0$  then begin  $r := r + 1$ ;  $Q(n, A)$ ;
        end
    else begin  $A[i] := 1$ ;  $tbs(ones - 1, i + 1, ex + 1)$ ;
         $A[i] := 0$ ;
        if  $ex > 0$  then  $tbs(ones, i + 1, ex - 1)$ ;
        end  $tbs$ ;
    end  $TBS$ ;
    
```

In this algorithm there is a separation at each level of recursion of those cases where $A[i] = 1$ from those where $A[i] = 0$. The parameter r is not essential

and simply numbers the cases 1, 2, 3, . . . , while the parameter ex represents the current excess of ones over noughts.

In the case of $n=4$ we have the sequence

1110000, 1101000, 1100100, 1011000, 1010100.

The reader is recommended to study the case of $n=5$ for himself.

THE NON-ORDERED, ROOTED, MULTIFURCATING TREE

The rooted multifurcating tree with order can have that order discarded, so that we may consider as entirely equivalent, for example, the tree in figure 5(a) and the trees in figure 6.

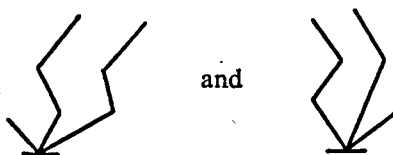


Figure 6

This is not the same as the removal of the ordering in the bi-furcating case, for the counting series is now

$$\mathcal{S}(x) \equiv \sum_{n=1}^{\infty} s_n x^n = x + x^2 + 2x^3 + 4x^4 + 9x^5 + 20x^6 + 48x^7 + 115x^8 + \dots$$

cf. Harary and Prins (1959).

One method of forming a collection of this set of trees within a computer is to generate all the examples covered by $\mathcal{S}(x)$, as, for example, in the previous section, and form the binary representations in some sequence. It is then relatively easy to formulate operations which are equivalent to swapping branches about at each internal node and to discover, as the set T_n is covered, whether an equivalent tree has already been found. This method of generating all cases and throwing away the second and subsequent isomorphs is clearly extremely wasteful, as Snow (1966) discovered in some earlier work, for the magnitude of s_n only increases roughly as 2^n . The better approach is to form a logical description of the set S_n so that only one example is found, while still using representations which contain the 'order' which we really wish to ignore.

It has not proved practicable to use the t.b.s. form as the basis for this latter approach, but it has been carried out using a 'height' representation. Before discussing this it is necessary to discuss the general problem of how we might say that a tree T_1 is less than, or occurs earlier in some sequence than, another tree T_2 .

Up to this point we have only considered at the same time collections of trees with the same number of nodes, n . Although it is not obvious, this carries with it an assumption that there is an overall ordering by n , and only when this constraint was discarded could further progress be made.

Let us consider two trees T_1 and T_2 which are ordered rooted trees and decompose according to the rules implied earlier into T_3, T_4 and T_5, T_6 respectively.

We wish to make a statement of the form $T_1 < T_2$ or $T_1 = T_2$ or $T_1 > T_2$. For equality let us take only identity in the unlabelled ordered form, and consider isomorphism between rooted forms as something different.

If we consider the right part as something added later, we would naturally require that if $T_3 = T_5$ and $T_4 < T_6$ then $T_1 < T_2$. We would also expect for such a system that if $T_3 < T_5$ then $T_1 < T_2$ whatever the relationship between T_4 and T_6 .

We may further break down a typical multifurcating tree, T , and consider it as an ordered composition of k sub-trees ($k > 1$) $T_1 T_2 T_3 \dots T_k$ as indicated in figure 7.

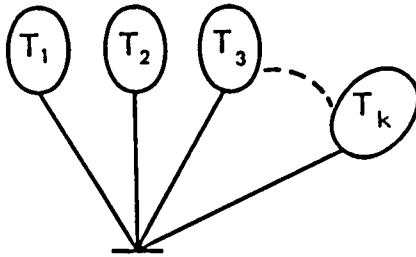


Figure 7

Whatever our rule for sequencing in detail, the set counted by $\mathcal{S}(x)$ requires only one version of the possible permutations of the order $T_1 T_2 \dots T_k$, as well as just one isomorph of each component. We shall say that an ordered rooted tree is in 'canonical form' if

$$T_1 \geq T_2 \geq T_3 \dots \geq T_k$$

and such that each component is itself in canonical form.

The 'heavy' part of the canonical form is taken to be on the left rather than on the right so as to preserve correspondence with the previous definition of the decomposition method for the multifurcating tree.

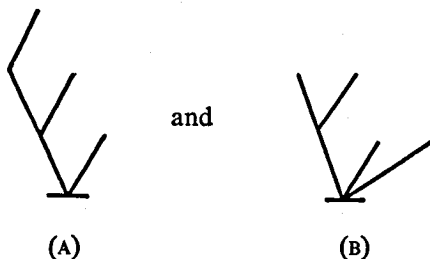
Consider now two trees T and T' with k and k' planted sub-trees respectively referred to as $T_1 T_2 \dots T_k$ and $T'_1 T'_2 \dots T'_{k'}$. Then T will be taken to be $\leq T'$ as $T_1 \leq T'_1$ even if T and T' are not in canonical form.

MATHEMATICAL FOUNDATIONS

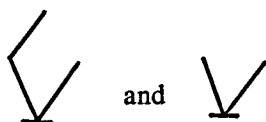
In the case of $T_1 = T'_1$, let $T_s = T'_s$, $s = 1, 2, 3, \dots, r < \min(k, k')$. Then $T \leq T'$ as $T_{r+1} \leq T'_{r+1}$. If $r = \min(k, k')$ then $T \leq T'$ as $k \leq k'$ and if $k = k' = r$, then and only then is $T = T'$.

Thus far we seem to have an incomplete specification for the 'earlier than' relationship. However, when one combines the above with the intrinsically sensible statement that the isolated point is first in sequence and that the next is the rooted tree of two nodes we have a complete specification.

This 'earlier than' relationship applies to the whole class of ordered trees but is most useful in discussing the sub-class of non-ordered rooted trees. For example, consider the two trees shown below.

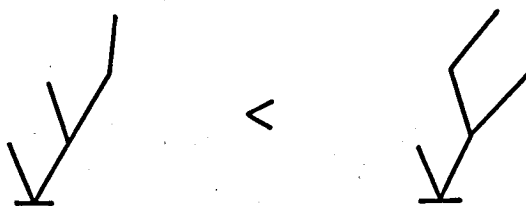


Their relative position is determined first by the relationship between their left-most planted parts



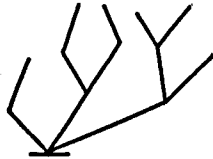
The relation between these is that of their left-most planted parts which are $\perp$ and $_$ respectively. Thus tree $A >$ tree B , since $\perp > _$.

However, if they had not been in canonical form the relationship might be different, for similar analysis shows, for example, that

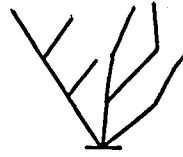


We observe that in pictorial terms it is not difficult to change a rooted tree into canonical form, and that when it is in canonical form the heights of the planted sub-trees standing on any node form a non-increasing sequence from

left to right. Figure 8 is a more sophisticated example. Figure 9 puts the rooted trees of 5 nodes in 'increasing' sequence, after they are put into canonical form. The important feature of this general pattern of sequencing is that there are an infinite number of trees which lie between any two trees of different heights. For example all trees of height three or less are 'earlier' than the last indicated in figure 9, whatever the number of nodes that they may contain.



non-canonical



canonical

Figure 8

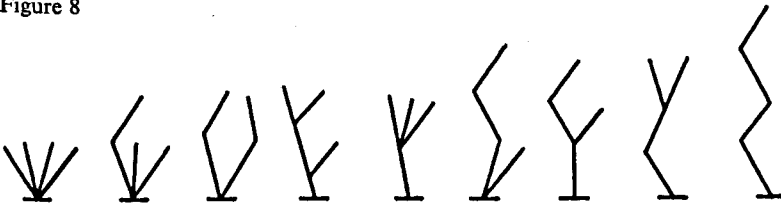


Figure 9

THE HEIGHT REPRESENTATION

In this section we shall confine attention to trees in canonical form, although the representation is capable of dealing with the wider class.

Let us label the nodes of the ordered tree under consideration from bottom to top and left to right in the same manner as that implied by the zeros in the terminated binary sequence. Let the height of the i -th node be $H[i]$. Then $H[i+1] \leq H[i]+1$, all i . Moreover $H[1]=0$ and $H[i]>0$, $i>1$.

This integer array $H[1:n]$ we shall term the height representation of a tree and if the tree is in canonical form we can make further restrictions on the permitted values.

The tree in figure 8 can be represented by

$$0, 1, 2, 3, 3, 2, 1, 2, 3, 2, 3, 1, 2 \quad (n=13)$$

We can tell from this that it is in canonical form for the sequence

$$1, 2, 3, 3, 2 \geq 1, 2, 3, 2, 3 \geq 1, 2$$

in the natural manner for comparing two sequences of integers lexicographically. Similarly the sub-sequences commencing with 2 and belonging to the same main branches are also in proper order

$$2, 3, 3 \geq 2; \quad 2, 3 \geq 2, 3$$

Thus the comparisons are reduced to comparing sequences of integers either to confirm the canonical property or to compare two canonical trees.

The procedure $NORTS(n, H, Q)$ generates all the rooted trees of n nodes in ascending order and carries out the operation $Q(n, H)$ on each as it is formed. The inner recursive procedure $P2(i, w)$ keeps track of those parts of the tree which are the same between the completion of consecutive trees. A non-recursive approach requires a stack to carry the equivalent of i and w at the different levels of entry.

```

procedure  $NORTS(n, H, Q)$ ; value  $n$ ; integer  $n$ ;
    integer array  $H$ ; procedure  $Q$ ;
begin integer  $y, z$ ;
    procedure  $P2(i, w)$ ; value  $i, w$ ; integer  $i, w$ ;
    begin integer  $u, h, s, t$ ;
     $u := H[w]$ ; comment  $u$  is a partial upper bound to  $H[i+1]$ ;
    for  $h := 1$  step 1 until  $u-1$  do
        begin  $H[i+1] := h$ ;  $t := i$ ;
        for  $s := t$  while  $H[s] \neq h$  do  $t := s-1$ ;
        if  $i+1 = n$  then  $Q(n, H)$  else  $P2(i+1, t+1)$ 
        end;
     $H[i+1] := u$ ; if  $i+1 = n$  then  $Q(n, H)$  else  $P2(i+1, w+1)$ 
    end  $P2$ ;
    for  $z := 2$  step 1 until  $n$  do
        begin for  $y := 1$  step 1 until  $z$  do  $H[y] := y-1$ ;
        if  $z \neq n$  then  $P2(z, z)$  else  $Q(n, H)$ ;
        end;
    end  $NORTS$ ;
    
```

The parameter z is the number of nodes in the left-most chain and the y for-statement fills in the corresponding portion of the array H . When $z = n$ we have an especially simple case.

$P2(i, w)$ builds up the H -vector at position $i+1$, taking into account the partial sequence which starts at position w ($\leq i$). A distinction has to be made when $H[i+1]$ attains its upper limit: w must be stepped on by one, for we then have a maximal sub-sequence.

This procedure and a non-recursive equivalent have been carefully studied on KDF.9 using the WALGOL and KALGOL systems. The interesting result is that the extra cost in time of the recursive approach is found to be only slight in the KALGOL system, and the average time per case generated approaches a constant value as n increases, if the operation Q is time independent of n .

To illustrate the representation for the nine trees shown in figure 9, the corresponding vectors are

0 1 1 1 1	0 1 2 2 1	0 1 2 3 2
0 1 2 1 1	0 1 2 2 2	0 1 2 3 3
0 1 2 1 2	0 1 2 3 1	0 1 2 3 4

This representation is not as compact as the t.b.s., but the trees can be generated in sequence by the nest of for-statements implicit in *NORTS* or any variant of it and then readily converted to t.b.s. form if required for reference purposes. However the examples given so far do not disclose the difficulties which arise for larger values of n and which account for the apparent complexity of *NORTS*.

Suppose $n=14$ and we have the partial specification

$$H[1] \dots H[14]=0, 1, 2, 3, 3, 2, 1, 2, 3, 2, 1, 2, 3, ?.$$

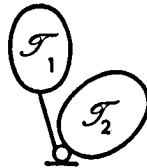
The last node has an upper bound on its height of 2 since $h[10]=2$. The parameter w in *P2* provides the necessary stack of backward pointers.

FREE TREES

A free tree is one that is non-ordered and without a root. A diameter of such a tree has the greatest path length between any two nodes in the tree. It can readily be shown that if the diameter is even then the mid-point of every diameter is a unique point in the tree called the centre. If such a tree is taken with centre as root and the sub-trees planted on the root put in descending order from left to right, and the same recursively, we have a unique, ordered rooted tree corresponding to the free tree. Since the root is the centre, the ordered rooted tree has the constraint imposed that the first two planted sub-trees are to be of equal height.

It is not difficult to generate in proper sequence all free trees which have a centre and possess n nodes in all using the height representation.

However, if a free tree has an odd diameter there is a pair of mid-points jointed by the bi-centre, and some difficulty arises. It is perfectly practicable to generate in proper sequence all free trees with odd diameter by using the height representation to generate trees of the form



where T_2 is of the same height as T_1 before planting, but such that $T_2 \geq T_1$. This gives the bi-centred trees in their proper order. However, it has proved impossible so far to generate the interleaved set of centred and bi-centred trees except by carrying out a comparison between the next centred tree and the next bi-centred tree, which is essentially a sorting operation which we have previously avoided.

REFERENCES

- Abramson, N. (1963), *Information Theory and Coding*, p. 89. New York: McGraw-Hill.
- Alway, G. & Martin, D. (1965), An algorithm for reducing the bandwidth of a matrix of a symmetrical configuration. *Comp. J.* **8**, 264.
- Harary, F. & Prins, G. (1959), The number of homeomorphically irreducible trees and other species. *Acta math. Stokl.*, **101**, 141-62.
- Obruca, A.K. (1966), *The Manipulation of Trees and Linear Graphs within a Computer and some applications*. PH.D. Thesis, University of Newcastle upon Tyne.
- Read, R.C. & Parris, R. (1966), Graph isomorphism and the coding of graphs. *UWI/CC3 Research Report*, University of West Indies.
- Riordan, J. (1958), *Introduction to Combinatorial Analysis*, pp. 125 and 134. New York: John Wiley.
- Scoins, H.I. (1967) *Machine Intelligence I*, pp. 3-15 (ed Michie, D. and Collins, N.L.). Edinburgh: Oliver and Boyd.
- Snow, C.R. (1966) *An Investigation into the Equivalence of Free Trees*. M.Sc. Dissertation. University of Newcastle upon Tyne.
- Unger, S.H. (1964) GIT-A heuristic program for testing pairs of directed line graphs for isomorphism. *Communs Ass. comput. Mach.*, **7**, 26.

THEOREM PROVING

A New Look at Mathematics and its Mechanization

B. Meltzer

Metamathematics Unit
University of Edinburgh

INTRODUCTION

Most published work on theorem proving by computer has belonged to one of two schools: either heuristic methods like those of Newell, Shaw and Simon are used, or axiomatic-complete methods such as those of Davis, Prawitz and J.A. Robinson. Both schools have, however, been toying with another approach, which makes more or less use of models or interpretations of the mathematical theory concerned. For example, in the heuristic school Gelernter has used geometrical figures as aids in proving theorems of geometry, and in the axiomatic school Loveland has used a method he terms 'model elimination'.

In this paper I wish to consider the model approach to theorem proving. To do this it will be necessary to take a look not only at the mechanization of mathematics but at mathematics itself.

THE SIGNIFICANCE OF UNDECIDABLE PROPOSITIONS IN MATHEMATICAL THEORIES

The classical results on undecidability and unsolvability of Gödel, Turing, and Church have been frequently discussed in connection with the development of mechanized mathematics. In particular Church's Theorem has proved extremely valuable for accurately delimiting the possibilities of various schemes. These uses are in effect of a negative character: they tell us that certain things we would like to do cannot be done, but they do not point to better ways of doing what we want to do.

However, I think the significance of Gödel's discovery of undecidable propositions is rather different and has not been properly appreciated; a proper appreciation would provide a useful pointer for future developments.

Since Gödel announced the existence of true undecidable propositions in formal number theory in 1930, the characteristic attitude of most of those

interested has been that an undecidable proposition is a pathological phenomenon. This reaction is probably due to the peculiar syntactical structure of the particular proposition constructed by Gödel, and to the fact that prejudices die hard—in spite of his result people could not quite stomach the demonstrated fact that formal number theory based on the Peano axioms did not completely describe elementary arithmetic, i.e. was not categorical. I wish to suggest that this attitude is inappropriate, because undecidable propositions are not only extremely numerous in mathematics but also in a certain sense among the most interesting and significant ones.

Let us review Gödel's result.¹ One can write out the conjunction of axioms of formal number theory—these are essentially Peano's axioms, including that of mathematical induction, and they employ only one predicate, that of equality, one function, the successor function, and one constant 0. Gödel constructed a proposition G , which he showed by informal reasoning was true but which nevertheless could not, either itself or its negation, be derived in the formal system if the latter were consistent. This proposition G , when interpreted, is a proposition of arithmetic, of the form 'a certain definite number does not belong to a certain precisely defined class of numbers'. But because of the way he constructed this number and this class it could also be interpreted, metamathematically, as meaning ' G is not formally derivable from the axioms'.

One's first reaction on meeting this result is that there must surely be something wrong with the system of inference used. But it has been indubitably demonstrated that the rules of inference used are such that from true premisses one is always led to true conclusions—so in that sense all is well with the logical system. What has gone wrong then? The answer is rather surprising: if we tried formally to deduce this true proposition of arithmetic G from the axioms we would in effect be trying to prove far too much; for it can be shown that if G had been true for all possible interpretations or models of the axiom system, it would then indeed have been derivable formally. Therefore there must be one or more models of the axiom system for which G is false. And these models, clearly, cannot be arithmetic or, at least, arithmetic as we ordinarily understand it.

So the reason that G could not be proved formally, even although it is true in arithmetic, was that we were trying to prove it true for a large number (in fact an infinite number) of other systems than simple arithmetic—and it just is not true for all these. The modes of inference we were allowing ourselves were too restrictive—they would only work for such propositions as were true for *all* the 'pseudo-arithmetics'; they would not work for a proposition true in our standard arithmetic only, for example. I shall come back to this point.

¹ Gödel's original proof was for a formal system embodying a higher-order logic. The result, however, holds also for a formalization of number theory in the lower predicate calculus, which is the system we have in mind in the present discussion. It is well known that all mathematical theories can be formulated in this calculus.

It is clear then that the axioms do not categorically specify simple arithmetic. One might therefore ask whether they cannot be supplemented by others so as to 'pin down' arithmetic. It can be shown that this is not possible.

Is the existence of undecidable propositions of this kind in formal theories of merely 'academic' interest? By no means. Undecidable propositions constitute in a sense some of the most important parts of mathematics. To show this, let me take another discipline, for example, elementary group theory, as formulated in the lower predicate calculus. The axioms are as follows:

$$\begin{array}{ll} (x)(y) & P(x,y,f(x,y)) \\ (x) & P(e,x,x) \\ (x) & P(g(x),x,e) \\ (x)(y)(z)(u)(v)(w) & \bar{P}(x,y,u) \vee \bar{P}(y,z,v) \vee \bar{P}(x,v,w) \vee P(u,z,w) \\ (x)(y)(z)(u)(v)(w) & \bar{P}(x,y,u) \vee \bar{P}(y,z,v) \vee \bar{P}(u,z,w) \vee P(x,v,w) \end{array}$$

Now consider all those propositions which are true only for Abelian groups, i.e. the groups for which the product of two elements is independent of their order. In this theory *all* these propositions are undecidable—which is remarkable, when one considers the vast number of books and monographs devoted to their study! One might object that this consideration does not really amount to much, for if one adds the axiom of commutativity the undecidability of many of these propositions vanishes. It is true that in this new theory, these Abelian theorems are no longer undecidable, but there will be other undecidable propositions which are true only of certain special subclasses of Abelian groups. In parenthesis, let me say that Tarski has shown that only theories which admit of a finite model, i.e. an interpretation with a finite number of elements, can possibly be categorical. Undecidable propositions are the rule over vast territories of mathematics.

To sum up the situation, it has been customary in the consideration of formal theories to concentrate attention almost exclusively on those propositions which are true in all models and therefore provable, and those which are false in all models and therefore disprovable. But the undecidable propositions, those that are true in some models, but false in others, are among the most interesting in mathematics.

THE MODEL APPROACH

What I wish to do is mainly to present a fresh point of view rather than new results and facts, though I shall mention one or two of these also. This may be useful, because so often advances in knowledge come from new points of view rather than from new facts.

In considering mathematics we must not be dominated completely by the axiomatic approach, but we should consider the *model* of our theory, the *actual objects* that we are talking about. The study of interpretations or models of theories was put on a precise rigorous basis by Tarski many years ago. Let me briefly review the idea. Given a set of sentences of a formal theory, like the axioms of group theory we considered earlier, we postulate

some universe of objects—the domain of our interpretation. For example, we might have in mind some particular group, say the group of symmetry rotations of a square about its centre. In this case our domain consists of four elements; the zero, 90 degrees, 180 degrees and 270 degrees anti-clockwise rotations. We label the elements by the constants e , a , b and c respectively. To complete the interpretation we need to specify the values of the functions and the truth values of the predicates appearing in our sentences. In the present example, for instance, the value of $g(a)$ will be c , g being the group inverse function. The truth-value of $P(e,a,b)$ will be False, since the interpretation of the predicate P is that the third argument is the product of the first two. If the values of the functions for every set of arguments and the truth-values of the predicates for every set of arguments taken from the elements of the domain are specified, the interpretation is determined. And if with this specification of domain, functions, and predicates, the sentences concerned turn out to have the value True then the interpretation is said to be a model of the sentences.

One of the most important facts about models, which I have implicitly used in the earlier discussion, follows from Gödel's Completeness Theorem. This is to the effect that for theories formulated in the lower predicate calculus, if every model of the axioms is a model of a given sentence S , then S is formally derivable from the axioms.

HERBRAND'S THEOREM AS A LINK BETWEEN THE AXIOMATIC AND MODEL APPROACHES

Herbrand's 'Recherches sur la théorie de la démonstration', although published as long ago as 1930, is only today coming to be recognized for the seminal work it was. Both logicians and workers in the field of mechanized mathematics are still discovering in that paper results, hints and anticipations of great interest. The main theorem of the paper is, as is well known, the basis of most theorem-proving programs and thinking on the subject.

Herbrand, who died in his early twenties, would most probably have approved of the uses made of his results in theorem proving, but would possibly have frowned on the kind of further application of them that I wish to discuss here. For his attitude was based on a philosophical outlook about the nature of mathematics, which can be described as 'finitist'. He was very careful to couch all his results in such terms, that all operations and concepts concerned could essentially be carried out or constructed in a finite number of steps. He would have had no sympathy with the Tarski model approach, which freely considers *all possible* interpretations and models, over domains of objects of any finite or infinite number, whether constructible or not. It was indeed this deliberate restriction of his conceptual apparatus which prevented Herbrand in his paper independently discovering Gödel's Completeness Theorem.

Nevertheless, the Herbrand approach is in my view to a large extent a

model approach, and by its careful dovetailing of models with axiomatics it appears to be most fruitful in methods and ideas.

Let me briefly state Herbrand's main result in the form in which it is used for theorem proving on computers.

Given a set of axioms of a theory expressed in the lower predicate calculus, by certain well known transformations one converts them into a logical conjunction of so-called *clauses*. Each clause is a disjunction of atomic predicates, either negated or not, over arguments which are constants or universally quantified variables. Existential quantifiers will have been got rid of by the use of what are known as Skolem functions. In our group theory example the identity element e , the product function f , and the inverse function g can be looked upon as Skolem functions, e being a function of no variables. The alleged theorem is negated and the resulting formula put in the same standard form as the axioms. The aim of a theorem-proving program then is to show that the conjunction $S = A \ \& \ \bar{T}$, where A represents the axioms and T the alleged theorem, leads to a contradiction.

One now forms the so-called 'Herbrand universe' of constants, the set of constants which can be constructed from all the constants appearing in S and all the function symbols appearing in S . Herbrand's Theorem states that S is unsatisfiable if and only if there is a finite set of instantiations of S over the Herbrand universe (by 'instantiation' we mean the result of consistently substituting constants of this universe for the variables in S) which is truth-functionally contradictory, i.e. false in the truth-table sense for all possible assignments of truth values to these instantiated or *ground* predicates.

What this amounts to is that a very particular domain of objects, namely the elements of the Herbrand universe of constants, has been chosen for the interpretations of the theory. Different choices of truth-values for the ground predicates correspond to different interpretations all over this same domain. Thus, Herbrand's Theorem amounts to saying that to show that T is a theorem one only needs to show that no models over this domain satisfy S (and since this domain is by its nature never of cardinality greater than that of the natural numbers, the relationship between this result and the well-known Skolem-Löwenheim Theorem is apparent—I am referring to the theorem that if any set of sentences has a model at all it has a denumerable model).

So, whereas Tarski talks of all possible models, Herbrand in effect talks only of models over his special domain. And the surprising thing is that this is quite sufficient for ensuring completeness of proof procedures.

THE USE OF MODELS IN THEOREM PROVING BY COMPUTER

The general importance of the model approach has now been made clear. To indicate why it may be important for theorem proving in particular, consider the following example.

Suppose in trying to prove a theorem in elementary group theory one

THEOREM PROVING

wished to make use of the fact that there was an element of the group which had an inverse different from itself. In the ordinary axiomatic approach, this would be impossible, for the existence of such a possibility is an undecidable proposition, since it would not be true for the group consisting of a single element or two elements. Thus the mere fact that the axioms apply to such trivial groups as that consisting of the identity element alone (hardly deserving the name of a group) hampers the process of finding a proof. Again it is true that one can add an axiom which effectively excludes this case; but this does not really deal with the fundamental difficulty. There may well be all kinds of other restrictions imposed on proofs for the structures one is really concerned with, which one would not know in advance in attempting to prove a new and complex theorem. The only way of avoiding this difficulty is somehow to take account of the structure or model directly.

Just how this is to be done most effectively merits research. In his geometry-theorem-proving programmes Gelernter (1963) used diagrams in a haphazard and approximate manner for the purpose of testing the truth of formal sentences which one desired to use as legitimate parts of a formal proof. Loveland made a more essential, and perhaps more interesting, use of models in the following way: he demonstrated unsatisfiability by actually trying to build up models, assigning truth-values to successive predicates by a chaining schema, and showing that these attempts failed. His method incorporates what is in my view one of the most 'intelligent' features to be incorporated in any theorem-proving programme. It has a built-in ability to detect the most 'sensitive' branches of the search tree and to concentrate on these.

MODEL-BIASED RULES OF INFERENCE?

Let us consider again the example previously given of elementary group theory without the axiom of commutativity. As I pointed out, if one wished to find theorems or proofs of results true for Abelian groups, one might do this merely by adding the axiom of commutativity, but this purely formalistic approach tends to obscure what one is really doing. It is quite conceivable that one could operate successfully without actually adding any new axioms but simply by using the facts of assignment of predicates and functions in the particular interpretation or set of interpretations in which one is interested—in this case the interpretations corresponding to Abelian groups. The possibility I wish to raise here is that one might be able to realize this use of a particular interpretation by formally modifying the notion of 'immediate consequence' in the formal system. Examples of immediate consequence are the classical rule of *modus ponens*, by which from A and $A \supset B$ we conclude B , and J.A. Robinson's resolution operating on two clauses. But we have seen that inference systems based on such rules are in a sense far too restrictive. For example, in formal number theory, their tailoring was such that one attempts to prove propositions not only for standard arithmetic but for a vast number of other 'pseudo-arithmetics'. Is it then possible that, with a given

model or set of models in mind, one could have modified rules of immediate consequence, which would not suffer—or at least suffer less—from this restricting handicap?

An example of this kind of thing has already appeared in the very interesting treatment of equality, which has been developed at Rice University by J.A. Robinson and E.E. Sibert. Sibert's system uses not only an extended version of Robinson's resolution but two further inference rules, and what they effectively do is to allow one to dispense with the troublesome axioms of equality. Very recently Robinson has improved on this by finding that only a single inference rule, which is a generalization of resolution, is required. He discusses this in a paper in this volume.

A METHODOLOGICAL PRINCIPLE FOR MATHEMATICAL THEORIES

We have seen that it would be desirable in theorem-proving programs to make direct use of the models in which we are interested. Consequently, it is important to have methods for restricting the range of models we must consider. I wish to give here an account of a principle which is of use in this respect.

As before, let $S = A \ \& \ \bar{T}$ be expressed as a set of clauses using Skolem functions, where A is the conjunction of axioms of a theory formulated in the lower predicate calculus, and T a proposition it is desired to prove.

We define an 'Herbrand model' as an assignment of truth values to all the ground atoms, i.e. atomic predicates of S whose variable places are occupied by constants of the Herbrand universe. We define a 'test-model' as an Herbrand model which satisfies the conjunction of axioms A .

Our result then is as follows:

S is unsatisfiable if and only if it is false in all test models.

The proof is immediate. From Herbrand's Theorem it follows that S is unsatisfiable if and only if it is false in all Herbrand models. But it is certainly false in all such which are not test-models, since A is false in them; hence the result.

So, to show that T is a theorem, we must show that no test-model satisfies $A \ \& \ \bar{T}$, i.e. $A \supset T$; in other words, we must show that every test-model satisfies $A \supset T$, i.e. satisfies T .

Thus we need to prove our theorem only for the category of test-models, and in identifying this sub-category of all possible models we may—since the test-models satisfy the axioms—use any knowledge we have of the theory.

As an example of the application of this principle let us consider again elementary group theory, as earlier axiomatized, with the predicate and axioms of equality added. Since the latter introduce no new function symbols, the Herbrand models are unaltered.

Consider the proof of any proposition T which has the prenex form

$$\bar{T} \text{ is } \begin{array}{l} (x)(Ey_1) \dots (Ey_r) B(x, y \dots y_r) \\ (Ex)(y_1) \dots (y_r) [\sim B(x \dots y_r)] \end{array}$$

THEOREM PROVING

The Skolemized form of this is:

$$(y_1) \dots (y_r) [\sim B(a, y_1, \dots, y_r)]$$

where a is the constant or function of no variables which has been introduced to get rid of the existential quantifier ($\exists x$).

The Herbrand universe is therefore constructed from e , a constant a , the inverse function g and the closure function f . So the category of test-models consists of those groups which can be formed from these, i.e. it consists in effect of the cyclic groups. So T will need to be proved only for cyclic groups. If this is done, however, it will be true for all groups, even continuous groups with a cardinality greater than that of the infinite cyclic group!

I cannot think of any interesting theorem of group theory of this logical form; but consider the following, which is a starred ('difficult') example in Birkhoff and MacLane's 'Survey of Modern Algebra':

'Prove that if $x^2=e$ for all elements of a group, then the group is commutative'.

The clauses that have to be added for the negation of this proposition are:

$$P(x, x, e)$$

$$\sim [f(a, b) = f(b, a)],$$

so that the test-models are all groups which can be generated by *two* elements a and b . The proof proceeds by first showing that every element is of the form $a^m b^n$, where m, n are integers, positive, zero or negative, and then by demonstrating commutativity for two elements of this form. In this case, the proof is no simpler for a human mathematician than the standard proof with no model, but it is an open question whether it is simpler for a machine.

The test-model principle, which should be usable in mechanized mathematics once one has developed suitable ways of taking account of the features of special models, is in a sense a sharpening of the Skolem-Löwenheim Theorem. It was remarkable enough that to prove a theorem in an axiomatic theory it was necessary to prove it only for all denumerable models. But it is still more surprising that one does not even need to do it for all these but only for a certain quite definitely specified subset of them. The choice of particular subset is dependent on the logical form of the theorem concerned, but once this is known, we have the unexpected phenomenon of deducing immediately very general results from the proofs of very particular ones.

Acknowledgment

I wish to thank P.J.Hayes for very valuable discussions.

REFERENCES

- Gelernter, H. (1963), Realization of a geometry-theorem-proving machine. *Computers and thought*, p. 134 (ed Feigenbaum, E.A. & Feldman, J.). New York: McGraw-Hill.
Loveland, D.W. (to be published), Mechanical theorem-proving by model elimination. *J. Ass. comput. Mach.*

Some Notes on Resolution Strategies

B. Meltzer

Metamathematics Unit
University of Edinburgh

A RELATIONSHIP BETWEEN SET-OF-SUPPORT AND P_1 -DEDUCTION STRATEGIES

The set-of-support strategy, proposed by Wos *et al.* (1965) and proved complete by J.A. Robinson, involves carrying out only such resolutions in which at least one of the parent clauses has an ancestry traceable back to a clause in the original negated-theorem set. P_1 -deduction (Robinson, 1965a), which is also complete, involves carrying out only such resolutions in which one of the parent clauses is positive, i.e. contains no negated atoms.

P_1 -deduction can be used with the aid of renaming (Meltzer, 1966) in many cases to provide a sharpening of the set-of-support strategy. This depends on the following considerations.

The set of clauses constituting the proper axioms of a first-order theory often has the following property: either it contains no positive clauses, i.e. clauses in which every predicate letter is un-negated, or it can be converted into this form by a consistent renaming of some of its predicates by means of their negatives. It is actually the case (cf. Meltzer, 1966) that any set of clauses with this property must be satisfiable, though the converse is not necessarily true. Nevertheless, for most of the theories to which automatic theorem proving has been applied, it is the case that the set of axiom clauses has this property. For example, this is so for the theory of partial order, group theory and the theory of fields.

Now let S_0 be the set of axiom clauses of a theory with this property and T the set of clauses equivalent to the negation of the alleged theorem. Then either S_0 contains no positive clauses or can be made to do so by some renaming. On the other hand, if the alleged theorem is a theorem, the union of S_0 and T remains unsatisfiable after any renaming and therefore (cf. Meltzer, 1966, Theorem 4) must contain at least one positive clause. Let T^+ be the resulting subset of the positive clauses of T .

Now the basic result connected with P_1 -deduction (cf. Robinson, 1965a)

THEOREM PROVING

is that a set of clauses is unsatisfiable if and only if the empty clause can be generated by successive resolutions, in each one of which one of the parent clauses is positive. In fact, such a deduction is what is meant by a P_1 -deduction. Therefore, if P_1 -deduction is applied to the above set, an empty clause should be generated. Furthermore, clearly every clause in this deduction will have its ancestry traceable back to some clause in T^1 , since S_0 , after renaming, contains no positive clauses. Hence, although P_1 -deduction has been applied, the effect is similar to what would have resulted from a set-of-support strategy, with, however, the following two differences, both of which represent a sharpening of the strategy. First, T^1 —which is now the effective set-of-support—is in general a proper subset of T . Secondly, even with this truncated set-of-support, only those resolutions having one positive parent clause are carried out, whereas the ordinary set-of-support strategy would allow resolutions between non-positive clauses, provided only they have an ancestry back in T^1 . In spite of these restrictions, the method is still complete, as follows from the fact that P_1 -deduction is complete (Robinson, 1965).

As an example consider the theorem of elementary group theory treated by G.A. Robinson *et al.* (1964):

'Prove that if a non-empty subset of a group contains xy^{-1} whenever it contains x and y then the subset is closed under multiplication.

Use the lemma [previously proved on their machine] that the inverse of the inverse of x is equal to x .'

In their version the axioms consist of 18 clauses and the negation of the theorem of 6. The straightforward application of set-of-support strategy would make all these 6 clauses the set-of-support. But if all three predicates are renamed by their negatives, only one clause in the set-of-support becomes positive. Hence the effective set-of-support involved in applying P_1 -deduction is reduced to one clause.

FLEXIBLE STRATEGIES

All strategies developed until now, such as unit preference (Wos *et al.*, 1964), set-of-support (Wos *et al.*, 1965), hyper-resolution (Robinson, 1965a), P_1 -deduction (Meltzer, 1966), are fixed ones in the sense that the basic criterion for choice of pairs of clauses to resolve is never changed throughout the program, and also no clauses either of the input set or those generated are discarded. (One qualification, however: those with experience of resolution programs will know what a vast number of redundant clauses can be generated, i.e. clauses which are either alphabetic variants or particular cases of existing clauses. Most programs incorporate subroutines for getting rid of most of these).

One would, however, without sacrificing completeness, like to have more flexible strategies, and I wish to report some results in proof theory which promise developments in this direction.

The procedure these results make possible is of the following kind. Suppose

one has an unsatisfiable set S of clauses. One divides the set into two disjoint subsets: an 'active' set T_0 and a 'passive' set S_0 . One then forms the set of resolvents, for each of which at least one of the parents is in T_0 . Then the set consisting of S_0 and these resolvents must also be unsatisfiable. Therefore the active set T_0 may now be discarded and the new joint set only be used in the rest of the programme.

So, having chosen on some appropriate criterion an active subset of the clauses in hand, one 'impregnates' the remaining clauses with this active set and then throws the active set away. In continuing the process, one may use any other criterion—if one wants to—for choosing the next active set. For example, in one case one may use unit-preference, in another a choice appropriate to P_1 -deduction, and, in fact, any other criterion suggested even by arbitrary heuristic considerations. Our results ensure that even if one chooses an active set on purely heuristic grounds and then throws it away, completeness is not endangered.

Another incidental advantage may occur in the treatment of equality. For the main thing wrong with treating equality as merely another predicate is that one usually thereby generates very many trivial, irrelevant equations. But one could to some extent ensure that this does not happen, or at least reduce its incidence, by seeing that the axioms of equality are not usually included in one's choice of active set. There may even be cases where it would be worth including some of them once in an active set and thereby, when this set is thrown away, effectively deprive them of most of their deleterious consequences.

These possibilities arise from the following:

Theorem. Let S be an unsatisfiable set of clauses consisting of two disjoint subsets T_0 and S_0 . Then there exists an integer n such that $S_0 \cup R$ is unsatisfiable, where R is the set of all resolvents having at least one parent in T_0 , down to the n th level of resolution.

The proof depends on two lemmas:

Lemma 1. Let S be an unsatisfiable set of *ground* clauses, $C \in S$ and $S_0 = S - \{C\}$. Let R be the set of first level resolvents of C with members of S_0 . Then $S_0 \cup R$ is unsatisfiable.

Proof

Let $A_1, A_2, \dots, A_m$ be the set of atoms appearing in S .

Let $C = \{L_1, L_2, \dots, L_k\}$, where $k \leq m$ and L_i is either A_i or $\bar{A}_i$.

Suppose $S_0 \cup R$ were satisfiable.

Let M be a model satisfying it. M must therefore satisfy S_0 and hence cannot satisfy C .

Therefore

$$M = \{L_1, L_2, \dots, L_k, J_{k+1}, \dots, J_m\},$$

where J_i is either A_i or $\bar{A}_i$.

Now consider the model

$$M^1 = \{L_1, L_2, \dots, L_k, J_{k+1}, \dots, J_m\}.$$

THEOREM PROVING

Every clause D in S_0 is satisfied by M^1 , for the following reasons. If D contains either at least one J_i or at least two $\bar{L}_i$, this is obvious. If D does not, it must contain one $\bar{L}_i$ since it is satisfied by M . But this would result, by resolution with C , in a clause in R which would not contain this $\bar{L}_i$ nor any other $\bar{L}_i$ or J_i and so would not be satisfiable by M ; hence D cannot contain such a single $\bar{L}_i$.

So S_0 is satisfied by M^1 . But the latter satisfies C also, thus contradicting the assumption that the full set S is unsatisfiable. Hence $S_0 \cup R$ is unsatisfiable.

Lemma 2. Let S be an unsatisfiable set of *ground* clauses consisting of two disjoint sets T_0 , containing n clauses, and S_0 . Let R be the set of resolvents having at least one parent in T_0 , down to the n th level of resolution. Then $S_0 \cup R$ is unsatisfiable.

Proof

Let $T_0 = \{C_1, C_2, \dots, C_n\}$.

By Lemma 1, if R_1 is the set of first-level resolvents of C_1 with $S - \{C_1\}$, then $\{C_2, \dots, C_n\} \cup S_0 \cup R_1$ is unsatisfiable. Similarly, one may single out C_2 and obtain that $\{C_3, \dots, C_n\} \cup S_0 \cup R_1 \cup R_2$ is unsatisfiable, where R_2 is the set of first-level resolvents of C_2 with $\{C_3, \dots, C_n\} \cup S_0 \cup R_1$. Repeating this process with the successive C_i a total of n times, one obtains the following unsatisfiable set:

$$S_0 \cup R_1 \cup R_2 \cup \dots \cup R_n.$$

But $R_1 \cup R_2 \cup \dots \cup R_n \subset R$.

Hence $S_0 \cup R$ is unsatisfiable.

Proof of Theorem. Since S is unsatisfiable, by Herbrand's Theorem, there is a finite set of instances of clauses of S which is unsatisfiable. Let T'_0 be the subset of these which consists of instances of the set T_0 , and S'_0 the subset which consists of instances of S_0 . Let n be the number of clauses in T'_0 .

Let R' be the set of resolvents of these ground clauses, down to the n th level of resolution, which have at least one parent in T'_0 . By Lemma 2, $S'_0 \cup R'$ is unsatisfiable.

Every clause in S'_0 is an instance of a clause in S_0 . That also every clause C' in R' is an instance of a clause in R , can be shown by mathematical induction as follows.

Suppose first the resolution level of C' is 1. It is therefore a resolvent of a clause A' in T'_0 and B' in $T'_0 \cup S'_0$. A' is an instance of a clause A in T_0 , and B' is an instance of a clause B in $T_0 \cup S_0$. Therefore, by Robinson's fundamental result (1965), there is a resolvent C of A and B such that C' is an instance of C . But C is in R . Therefore C' is an instance of a clause in R .

Now assume that this result is true for clauses of levels $1, 2, \dots, i$. Let C' be any clause in R' of level $i+1$. It is therefore a resolvent of a clause A' in T'_0 and a clause B' which is of level i in R' . B' is thus an instance of a clause in R and A' is an instance of a clause in T_0 . Therefore by Robinson's result there is a resolvent C of A and B such that C' is an instance of C . But C is in R .

Therefore C' is an instance of a clause in R .

Hence a set of instances of $S_0 \cup R$ is unsatisfiable, and therefore $S_0 \cup R$ is unsatisfiable.

The converse of the Theorem is obvious, since resolution is a valid form of inference.

Acknowledgments

The theorems about 'active' and 'passive' subsets of a set of input clauses developed from a suggestion of H. Millar, and the author wishes to thank him, Mrs P. Poggi, Dr J. L. Darlington, and Mlle A. Preller for valuable discussions.

REFERENCES

- Meltzer, B. (1966), Theorem-proving for computers: some results on resolution and renaming. *Comput. J.* 8, 341-3.
- Robinson, G. A., Wos, L. T. & Carson, D. F. (1964), Some theorem-proving strategies and their implementation. *AMD Tech. Memo No. 72*. Argonne National Laboratory.
- Robinson, J. A. (1965), A machine-oriented logic based on the resolution principle. *J. Ass. comput. Mach.*, 12, 23-41.
- Robinson, J. A. (1965a), Automatic deduction with hyper-resolution. *Int. J. comput. Math.*, 1, 227-34.
- Wos, L. T., Carson, D. F. & Robinson, G. A. (1964), The unit preference strategy in theorem-proving. *AFIPS*, 26, 615-21, Fall, J. C. C. Washington, D.C.: Spartan Books.
- 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, 536-41.

The Generalized Resolution Principle

J.A. Robinson
College of Liberal Arts
Syracuse University, N.Y.

1. INTRODUCTION

The generalized resolution principle is a single inference principle which provides, by itself, a complete formulation of the quantifier-free first-order predicate calculus with equality. It is a natural generalization of the various versions and extensions of the resolution principle, each of which it includes as special cases; but in addition it supplies all of the inferential machinery which is needed in order to be able to treat the intended interpretation of the equality symbol as 'built in', and obviates the need to include special axioms of equality in the formulation of every theorem-proving problem which makes use of that notion.

The completeness theory of the generalized resolution principle exploits the very intuitive and natural idea of attempting to construct counterexamples to the theorems for which proofs are wanted, and makes this the central concept. It is shown how a proof of a theorem is generated automatically by the breakdown of a sustained attempt to construct a counterexample for it. The kind of proof one gets depends entirely on the way in which the attempt to construct a counterexample is organized, and the theory places virtually no restrictions on how this shall be done. Consequently there is a very wide freedom in the form which proofs may take: the individual inferences in a proof may be very 'small' or very 'large' (in a scale of measurement which, roughly speaking, weighs the amount of computing necessary to check that the inference is correct). It is even correct to infer the truth of a true proposition *in just one step*, but, presumably, to offer such a proof to someone who wishes to be convinced of the proposition's truth would not be helpful epistemologically. His conviction would come, not from contemplating the proof itself, but rather from examining the computation which shows the correctness of its single inference step.

THEOREM PROVING

2. QUANTIFIER-FREE FIRST-ORDER PREDICATE CALCULUS WITH EQUALITY

2.1. Syntax

The expressions of the calculus are either simple or composite, and if they are composite they have a unique *applicative structure* consisting of two parts, an *operator* and an *operand*. The intention is that, when it is *interpreted* as explained in 2.2 below, every expression shall *denote* something, and that the entity denoted by a composite expression AB (where A is the operator, and B the operand, of AB) shall always be the result of *applying* the entity denoted by A to the entity denoted by B . The expressions are all built up from primitive symbols in a systematic way explained below.

2.1.1. Vocabularies. A *vocabulary* is a set V of symbols, partitioned into disjoint subsets as follows: $I(V)$ is the set of *individual symbols* in V ; and, for each natural number $n \geq 0$, the set $F_n(V)$ is the set of *function symbols of degree n* in V , and $R_n(V)$ is the set of *relation symbols of degree n* in V . It is possible that some, and even all but finitely many, of the sets $I(V)$, $F_0(V)$, $R_0(V)$, $F_1(V)$, $R_1(V)$, $\dots$, should be empty; but we assume that at least one of them is not. We shall usually employ lower case letters for individual symbols and upper case letters for function and relation symbols.

2.1.2. Expressions over a vocabulary. Let V be a vocabulary. Then the *expressions over V* are the terms, sentences, sequences of terms, sets of terms and sets of sentences defined below, together with the members of V themselves. In these definitions, the references to individual, function and relation symbols are to be taken as restricted to V .

2.1.2.1. Terms. A term is either an individual symbol or else has the form FT where F is a function symbol of degree n and T is a sequence of n (not necessarily distinct) terms.

2.1.2.1.1. Sequences of terms. A sequence of n terms is simply the empty string when $n=0$. When $n>0$, a sequence of n terms is a parenthesized list $(T_1, \dots, T_n)$ each component in which is a term. It is not necessary that all of the components be distinct.

2.1.2.2. Atoms. An atom either has the form RT where R is a relation symbol of degree n and T is a sequence of n terms, or else is an *equation* $=S$ where $=$ is the equation symbol and S is a nonempty set of terms.

2.1.2.2.1. Sets of terms. A set of terms is a list of terms enclosed in a pair of braces: $\{T_1, \dots, T_n\}$. When no terms at all appear between the braces the set is said to be *empty*. A term is said to be *in* a set if and only if it is one of the components of the list, and two sets are regarded as being the same if every term which is in one of them is also in the other. In other words, the order and multiplicity of terms in a set is irrelevant.

2.1.2.3. Literals. A literal is either an atom or has the form $\neg A$ where $\neg$ is the negation symbol and A is an atom. The literals A , $\neg A$ are *complementary*, and each is *the complement* of the other.

2.1.2.4. Sentences. A sentence is either an atom, or a conjunction $\& C$ where

$\square$ is the conjunction symbol and C is a set of literals; or a disjunction $\sqcup C$ where $\sqcup$ is the disjunction symbol and C is a set of literals; or a negation $\neg S$ where S is a sentence.

2.1.2.4.1. Sets of sentences. A set of sentences is a list of sentences enclosed in a pair of braces, exactly as in 2.1.2.2.1 above.

2.1.3. We will usually write an equation $=\{T_1, T_2\}$, having only two components, in the more conventional fashion as: $T_1=T_2$. Also a conjunction $\square\{S_1, \dots, S_n\}$ will usually be written: $(S_1 \wedge \dots \wedge S_n)$ or even simply as $S_1 \wedge \dots \wedge S_n$. Likewise a disjunction will usually be written with the familiar $\vee$ interposed between its components. However the *empty* conjunction and the *empty* disjunction will always be written respectively as: $\square$, $\sqcup$, omitting the pair of braces enclosing nothing.

2.2. Semantics

2.2.1. Terms and sentences become meaningful only when an interpretation is provided for the symbols in the vocabulary over which they are written. Thereupon, as explained in detail below, each term and each sentence acquires a *denotation*, that is to say *something which it denotes*, under that interpretation. Sentences always denote one or other of the two truth values *true*, *false*. Terms always denote some specific object in the so-called *universe* of the interpretation.

2.2.2. Interpretations. Formally, an interpretation is a mapping g of a vocabulary V (called *the vocabulary of the interpretation*) onto a collection of entities all of which are constructed out of a certain set D (called *the universe of the interpretation*). Specifically, g maps each individual symbol in V onto a member of D , each function symbol of degree n in V onto a function from D^n to D , and each relation symbol of degree n in V onto a function from D^n to $\{true, false\}$. The entity $g(E)$ onto which each symbol E in V is mapped by g is said to be *denoted by E under g* or to be *the denotation of E under g* .

2.2.3. Denotations of logical symbols, sets and sequences. The logical symbols $\neg$, $\square$, $\sqcup$, and $=$ always denote the same entities, under every interpretation. Indeed, the negation symbol denotes that function from truth values to truth values which when applied to *true* gives *false* and conversely; the conjunction symbol and the disjunction symbol each denote a function from *sets* of truth values to truth values, with $\square$ denoting the function which gives *false* when applied to a set containing *false*, and giving *true* when applied to other sets; and $\sqcup$ denoting the function which gives *true* when applied to a set containing *true*, and giving *false* when applied to other sets. The equation symbol always denotes that function from nonempty sets to truth values which gives *true* when applied to sets containing exactly one object, and which gives *false* when applied to other nonempty sets. A set or sequence of expressions is always taken to denote the set or sequence of things which are denoted by the constituent expressions.

2.2.4. In general, an expression with operator A and operand B denotes,

THEOREM PROVING

under an interpretation g of the vocabulary in which the expression is written, the entity which the function $g(A)$ gives when it is applied to the entity $g(B)$ which is denoted by B under g .

2.2.5. By virtue of the above explanations, we can regard any interpretation g as automatically extended to the set of all expressions over the vocabulary of g . In particular each sentence over the vocabulary of g denotes either *true* or *false* under g , and we say that the sentence is *true under g* , or that g *satisfies* it, in the first case, and that the sentence is *false under g* , or that g *falsifies* it, in the second case.

From our explanations above it is easy to verify that the empty conjunction is true under every interpretation and that the empty disjunction is false under every interpretation.

An interpretation can neither satisfy nor falsify a sentence which is not among the sentences over its vocabulary, for there must occur in such a sentence at least one nonlogical symbol which is without any denotation. Whenever, in the remainder of this paper, we speak of an interpretation and a sentence in the context of inquiring whether the former satisfies or falsifies the latter, we should be understood as taking it for granted that the vocabulary of the interpretation is large enough to contain each nonlogical symbol which occurs in the sentence.

2.2.7. It sometimes happens that an interpretation g not only satisfies a sentence S , but that also it *would* satisfy S , no matter what other members of the universe of g were to be denoted by the individual symbols in the vocabulary of g . We can express this situation more precisely with the help of the concept of *structurally equivalent* interpretations. Two interpretations are said to be *structurally equivalent* if their universes and vocabularies are the same and if each assigns the same denotation to the function symbols and relation symbols in their common vocabulary. Thus, the only way in which structurally equivalent interpretations can differ at all is in the denotations they assign to individual symbols.

Then the situation might arise that not only does g satisfy S , but *every structural equivalent of g satisfies S* .

We shall say that g *strongly satisfies S* if, and only if, every structural equivalent of g satisfies S . Obviously, if g strongly satisfies S then g satisfies S , because g is certainly a structural equivalent of itself. But the converse is not in general true.

Intuitively, g strongly satisfies S only when g satisfies the result of *universally quantifying S* with respect to all of the individual symbols that it contains. What S says about the objects denoted under g by those individual symbols is true of all of the objects in the universe of g . In the present quantifier-free system, it is the notion of strong satisfaction which fills the gap left by doing away with the device of quantifiers.

In a similar way we say that g *strongly falsifies S* if, and only if, every structural equivalent of g falsifies S . A little reflection shows that g strongly

falsifies S only when, intuitively, g satisfies the result of universally quantifying $\neg S$ with respect to its individual symbols.

In the quantifier-free predicate calculus, individual symbols are *not* variables. They always denote particular, fixed objects under an interpretation, as indeed do all nonlogical symbols.

2.2.8. If X is a *set* of sentences and g is an interpretation, we shall say that g (strongly) satisfies X if, and only if, g (strongly) satisfies each sentence in X .

2.3. Propositions

The point of the whole enterprise of logic is to be able to formulate, investigate and settle, any *proposition* which asserts that a sentence Y *follows from* a set X of sentences. (To facilitate our discussion we shall say that Y is the *conclusion*, and the members of X the *premisses*, of the proposition P). Now there are two senses in which *follows from* can be taken, in our present system. The first sense, which we shall call the *ground* sense, is explained by saying that:

2.3.1. Y *follows from* X if, and only if, among the interpretations which satisfy X , there is none which falsifies Y .

The second sense, which we shall call the *general* sense, is explained by saying that:

2.3.2. Y *follows from* X if, and only if, among the interpretations which *strongly* satisfy X , there is none which falsifies Y .

In order to help keep the distinction between these two senses of *follows from* clear, we adopt the following notation for propositions: we write $X \Rightarrow Y$ to represent the proposition that Y follows from X in the *ground* sense, and we write $X \rightarrow Y$ to represent the proposition that Y follows from X in the *general* sense. We say that $X \Rightarrow Y$ is a *ground proposition* and that $X \rightarrow Y$ is a *general proposition*.

2.3.3. From the definitions 2.3.1 and 2.3.2 it can be readily checked that if $X \Rightarrow Y$ then $X \rightarrow Y$. This is so because the interpretations which *strongly* satisfy X are all contained in the set of interpretations which merely satisfy X . The converse is untrue, however. For example, $\{P(x)\} \Rightarrow P(y)$, but it is not the case that $\{P(x)\} \rightarrow P(y)$.

2.3.4. It is not hard to show that if $X \rightarrow Y$ then any interpretation which *strongly* satisfies X will also *strongly* satisfy Y (although, in the definition 2.3.2, it is only formally necessary that it merely *satisfy* Y). For if g strongly satisfies X , then so does h , where h is any structural equivalent of g , and hence h also satisfies Y ; therefore any structural equivalent of g satisfies Y , and therefore g strongly satisfies Y .

2.3.5. A *counterexample* to a ground proposition P is an interpretation which satisfies the premisses of P but falsifies its conclusion. Similarly, a counterexample to a general proposition is an interpretation which *strongly* satisfies its premisses but falsifies its conclusion. Intuitively, a proposition says of

itself that it has no counterexample, and it is true if this is in fact the case, and false otherwise.

A proposition which is true is also called a *theorem*.

2.4. Ground propositions are decidable

As we have seen, a proposition is an intuitively meaningful assertion, which says something quite concrete and specific about all of the ways in which interpretations can affect the premisses and the conclusion of the proposition. As such it is either true or false according as what it asserts to be the case *is* the case or not. Strictly speaking, a proposition is an assertion of the *semantical metalanguage* of our system, and is not to be counted among the sentences of the object language as characterized in 2.1. However, even though a proposition is always either true or false it is not always by any means obvious which. Fortunately, in the case of ground propositions, there is an *algorithmic* way of correctly deciding the truth or falsehood of them, which we now go on to explain.

2.4.1. The method of denotation tables. From the definition 2.3.1 it looks as if, in order to decide whether or not a ground proposition P is true, we would have to examine all interpretations whatsoever of its premisses and conclusion. This would of course be quite out of the question, as there are at least as many interpretations of a vocabulary as there are sets to act as their universes.

However, it is not in fact necessary to do this. For the only way in which an interpretation can affect the premisses and conclusion of P is by providing a truth value as denotation for each of the atoms which appear in any of them. Since there can be only finitely many (say, n) such atoms there can be only finitely many (at most 2^n) different ways in which the set of them can be mapped by an interpretation onto truth values. We can list all of these ways in a *denotation table*, in just the same way as is done in constructing a *truth table* in the propositional calculus. Indeed, if we *first* construct a truth table for the set of atoms involved, and *then* remove from it any mapping which 'violates the semantics of the equation symbol' in the way explained precisely below, then we in fact get the denotation table for the set of atoms. Once we have the denotation table, we can easily check mechanically to see whether any of the mappings in it corresponds to a counterexample for P , and thereby settle whether P is true or false. Herein consists the decision procedure for ground propositions.

2.4.2. Constructing a denotation table for a set of atoms.

2.4.2.1. Conflation. Let T be a set of terms, and K be a partition of T . Then we say that two expressions Q and R are *conflated* by K if, for some (relation or function) symbol S , Q is $S(A_1, \dots, A_n)$, R is $S(B_1, \dots, B_n)$, and for each $j, j=1, \dots, n$, A_j and B_j lie in the same block of K . Two blocks of K are said to *overlap* if there are two terms, one in each of the blocks, which are conflated by K . By the *closure* of K we mean the partition which is obtained from

K by repeatedly merging (i.e., forming the union of) pairs of blocks which overlap, until there are none left which do.

2.4.2.2. Admissible mappings. A mapping g of a set A of atoms onto truth values is *admissible* if, and only if, for the partition K defined below, g maps atoms, which are conflated by K , onto the same truth value; and g maps an equation $=S$ onto *true* only if S is included in some block of K .

In the above definition, K is the partition of the set T of all terms which appear in any of the atoms in A , determined as follows: first we let M be the partition of T in which two terms lie in the same block if, and only if, g satisfies some equation $=S$ in A whose S contains both of the terms: then we let K be the closure of M .

2.4.2.3. The denotation table for a set A of atoms is then the set of all mappings g of A onto truth values, such that g is *admissible*.

2.5

Because of the *method of denotation tables* one can always, at least in principle, directly determine whether a ground proposition P is true or not. Of course the amount of computing involved in carrying out the method on P will (no matter how efficiently the work of constructing the denotation table and checking each map in it might be organized) increase as P becomes 'larger'. There will indeed be a 'size' of P which is as large as can feasibly be managed, using this method, for every computing agency, man or machine, with a fixed amount of computing power. Furthermore there is a point beyond which a human is not well served epistemologically by merely being *informed* that a proposition is true, or that it is false, even if his informant is entirely reliable and is somehow known to be so. One wants to be told not only *that* a proposition is true or false, but also, so to speak, *why*. It would surely be most unsatisfying intellectually to be told by an omniscient demon that, for example, Fermat's Last Theorem is indeed true, if he did not also provide us with a *proof* of it which we could understand. We go on, therefore, to discuss *inference* and *proof* in the present system and in general.

3. INFERRING CONSEQUENCES AND PROVING THEOREMS

3.1

To *prove* a proposition is to *show that it is true*. Presumably, for someone who can see directly that a proposition is true, a proof of that proposition is unnecessary. If he cannot see directly that the proposition is true, however, then he must be given a way of doing so which requires that he see directly the truth only of propositions which he *can* directly settle, without mediation. The following fundamental *consequence principle* provides the framework within which this might be done: *if Y follows from Z , and if each sentence in Z follows from X , then Y follows from X .*

It is straightforward to check that the consequence principle holds if *follows from* is construed in either the *ground* sense or the *general* sense of 2.3.

THEOREM PROVING

But it is important to realize that the consequence principle holds when *follows from* is simply taken in the unformalized sense of ordinary mathematical usage.

The proposition that Y follows from X can be seen to be true, therefore, by one who can see that Y follows from Z and that each member of Z follows from X . If there are k sentences in Z then the original problem (to see that Y follows from X) can be replaced by $k+1$ problems of exactly the same sort as the original one. Obviously, if the reduction is to have any point, each of the $k+1$ problems must be, in some sense, *easier to solve* than the original one. However, it is not at all necessary to treat the reduction itself as a problem, for the consequence principle requires no justification either in general or in any particular application.

These general remarks supply the motivation and background for the formal concept of *proof* which is introduced below.

3.1.1. A *tree of sentences* is a tree to each node of which is attached a sentence, which is said to be the sentence *at* that node. It is possible that the same sentence should be attached to more than one node in a tree of sentences. A *ground proof* is a tree of sentences such that, if Q is the sentence at any interior node N of the tree, and $P_1, \dots, P_k$ are the sentences at the nodes which are immediately outward from N in the tree, then the *proposition at* N , $\{P_1, \dots, P_k\} \Rightarrow Q$, is *true*. A *general proof* is defined in exactly the same way except that $\Rightarrow$ replaces $\Rightarrow$ in the definition.

Every ground proof is also a general proof, by 2.3.3, but not necessarily conversely.

If all of the sentences at the tips of the proof P are in X , and if the sentence Y is at the root of P , then P is a *proof of* the proposition that Y follows from X , in the ground or the general sense according as P is a ground proof or a general proof.

3.2

In order to see, then, that a tree of sentences is in fact a proof, one must be able to see that the proposition at each of its interior nodes is in fact a *theorem*. We now describe a method which automatically produces proofs for ground theorems, in which the interior theorems are necessarily 'obvious' to the agent (man or machine) with the computing power that must have been available in order that the proof could have been produced at all. An agent having only very little computing power can produce only proofs which have interior theorems of an extremely simple kind. An agent with greater computing power can produce proofs with fewer, but 'larger' interior theorems. An agent with sufficiently great computing power will be able to prove the theorem itself in a single 'obvious' step.

3.3. Semantic trees

Let K be the set of atoms in the premisses or the conclusion of a ground proposition P . Let T be a tree of sentences, each one of which is a conjunction

whose literals are all atoms, or complements of atoms, in K . Then T is a *semantic tree for P* if the following four conditions are satisfied at each node N of T , where C is the conjunction of all the literals in all of the conjunctions at the nodes on the branch of T down to and including N , and $C_1, \dots, C_k$ are the conjunctions at the nodes of T immediately outward from N :

3.3.1. there is no atom L in K such that both $C \Rightarrow L$ and $C \Rightarrow \neg L$;

3.3.2. $C \Rightarrow (C_1 \vee \dots \vee C_k)$;

3.3.3. there is no literal M in C_j such that $C \Rightarrow M$ (for $1 \leq j \leq k$);

3.3.4. if N is a tip of T , then either $C \Rightarrow L$ or $C \Rightarrow \neg L$ for each L in K .

3.4. Discussion of this definition

The intuitive idea behind the definition of 3.3 is that as we move down a branch of a semantic tree for P we encounter, at each node, a further quantum of information in an increasingly more complete description of an interpretation of the vocabulary in which the premisses and conclusion of P are written. The conjunction C of all the literals in all of the sentences on a branch of T is a complete description of an interpretation in the sense that it portrays one possible way in which an interpretation g can make each sentence S over that vocabulary denote a truth value: if S is true under g then $C \Rightarrow S$, and if S is false under g then $C \Rightarrow \neg S$. Conditions 3.3.1 and 3.3.4 are imposed in order to ensure just this. Condition 3.3.3 is theoretically dispensable. It merely ensures that each component of each new quantum is in fact new information, not deducible from the part of the description which is already given. Condition 3.3.2 is imposed in order to guarantee that every possible interpretation is described by some branch of T . For no matter what interpretation g of the premisses and conclusion of P we consider, the conjunction C at the root of T is satisfied by g ; and in general, if g satisfies C then g satisfies $(C \wedge C_j)$, for some j , by condition 3.3.2. Therefore there is some branch of T which g satisfies. But to say that g satisfies C , and to say that C completely describes g , is to say the same thing, when N is a tip of T .

3.5. Failure points; counterexample trees

A *counterexample tree for P* , where P is a ground proposition, is a semantic tree for P in which certain nodes are classified as *failure points* as follows (retaining the notation of 3.3 and 3.4): the node N in a semantic tree T for P is a *failure point of T* if $C \Rightarrow \neg Z$ for some premiss Z of P or if $C \Rightarrow Y$ where Y is the conclusion of P .

Obviously any branch of T which contains a failure point cannot describe a counterexample for P . Therefore, if P is true, every branch of every counterexample tree for P must contain a failure point.

3.6. Inference points of counterexample trees

A node N in a counterexample tree T for P is called an *inference point of T* if the following two conditions are satisfied:

3.6.1. N is not a failure point of T ;

3.6.2. each of the nodes immediately outward from N is a failure point of T .

3.7. Standard form for propositions

We obtain considerable simplification in the subsequent discussion if we are able to assume that the propositions P we deal with are all in a certain standard form, namely, the form in which P satisfies the two conditions:

3.7.1. the conclusion of P is $\square$;

3.7.2. each premiss of P is a disjunction.

There is no loss of generality involved in this assumption since every proposition is *equivalent* to a proposition in this standard form in the strict sense that every counterexample of the one is also a counterexample of the other. To see this, it is enough to note that $\{X_1, \dots, X_n\} \Rightarrow Y$ is equivalent to $\{X_1, \dots, X_n, \neg Y\} \Rightarrow \square$; that replacing $\neg \square\{S_1, \dots, S_n\}$ by $\square\{\neg S_1, \dots, \neg S_n\}$ or replacing $\neg \square\{S_1, \dots, S_n\}$ by $\square\{\neg S_1, \dots, \neg S_n\}$ anywhere in a proposition produces an equivalent proposition; that deletion of $\neg\neg$ anywhere in a proposition produces an equivalent proposition; and finally that replacing, on the left hand side of a proposition, the conjunction $\square\{S_1, \dots, S_n\}$ by the n disjunctions $\square S_1, \dots, \square S_n$ produces an equivalent proposition.

3.8. Making inferences at inference points

Let us now examine more closely the situation at an inference point N in a counterexample tree T for a proposition P in standard form. Let $C, C_1, \dots, C_k$ be defined at N as in 3.3, and let $P_1, \dots, P_k$ be premisses of P such that:

3.8.1. $\{C, C_j\} \Rightarrow \neg P_j$ for each $j, 1 \leq j \leq k$.

Since P_j is a disjunction, and since N is not a failure point of T , we can write P_j as $(A_j \vee B_j)$, where A_j and B_j are disjunctions, and where

3.8.2. $C \Rightarrow \neg A_j$ for each $j, 1 \leq j \leq k$

but

3.8.3. $C \Rightarrow \neg B_j$ for no $j, 1 \leq j \leq k$.

It is possible that A_j is empty, but not that B_j is. Because of 3.8.1 and 3.8.2. we have

3.8.4. $\{C, C_j\} \Rightarrow \neg B_j$ for each $j, 1 \leq j \leq k$

and by definition of a counterexample tree we have:

3.8.5. $C \Rightarrow (C_1 \vee \dots \vee C_k)$.

From 3.8.4 and 3.8.5 it immediately follows that:

3.8.6. $\{B_1, \dots, B_k\} \Rightarrow \neg C$

and that therefore there is at least one choice of a disjunction B (namely the disjunction of all the complements of literals of C) such that:

3.8.7. $\{B_1, \dots, B_k\} \Rightarrow B$ and $B \Rightarrow \neg C$.

Now let Q , for any B which satisfies, 3.8.7, be the disjunction:

3.8.8. $(A_1 \vee \dots \vee A_k \vee B)$;

then it readily follows that

3.8.9. $\{P_1, \dots, P_k\} \Rightarrow Q$

and also that

3.8.10. $C \Rightarrow \neg Q$.

For from the second part of 3.8.7 we know that $C \Rightarrow \neg B$, and this, with 3.8.2, immediately gives 3.8.10; while the first part of 3.8.7 immediately gives 3.8.9. Suppose that the proposition P is: $X \Rightarrow \square$.

If we add the sentence Q to the set X , to obtain the set X' , and define the tree T' to be the result of classifying as extra failure points the nodes of T at which Q is false, then T' , as it stands, is a counterexample tree T' for the proposition $X' \Rightarrow \square$. Now, since $X \subseteq X'$, every failure point of T is a failure point of T' . However, *the node N is certainly a failure point of T'* , because of 3.8.10, *but not of T* , by 3.6.1. If we define the size of a counterexample tree to be the number of nodes in it which are not failure points of it, then we can express the above situation by saying that *the size of T' is strictly less than the size of T* . Now if $X \Rightarrow \square$ is true, every branch of T contains a failure point, and therefore T contains an inference point, unless the root of T is itself a failure point. Hence the same will be true of T' . Therefore the above construction can be iterated to produce a sequence of counterexample trees $T, T', \dots, T^{(n)}$ for a sequence $X \Rightarrow \square, X' \Rightarrow \square, \dots, X^{(n)} \Rightarrow \square$ of theorems each of which is equivalent to $X \Rightarrow \square$ and therefore true; with the sizes of the successive trees forming a strictly decreasing sequence of numbers ≥ 0 . Hence, for some n , the size of $T^{(n)}$ will be zero. This means that the root of $T^{(n)}$ is a failure point for $X^{(n)} \Rightarrow \square$, which can happen only if $\square$ was inferred at some inference point in $T^{(n-1)}$, and added to $X^{(n-1)}$ to form $X^{(n)}$. If we attach to each of the failure points of T , a sentence in X which is falsified at that point; and then thereafter attach, to each of the new failure points in T' , the sentence Q which is added to X to get X' , and so on, through the sequence to $T^{(n)}$; then $T^{(n)}$ will actually be a proof of $X \Rightarrow \square$. Each of the inferences in this proof will have been made automatically, from the materials available in the immediate neighbourhood of the corresponding inference point. Notice that a special case of a counterexample tree for any theorem $\{X_1, \dots, X_k\} \Rightarrow \square$ is the tree having just $k+1$ nodes, namely a root N and k tips $N_1, \dots, N_k$ immediately outward from N , with the conjunction $\square$ attached to N , and the conjunction C_i attached to $N_i, i=1, \dots, k$, where C_i is just the conjunction of all the complements of literals in X_i . Then each of $N_1, \dots, N_k$ is a failure point, and the construction of the present paragraph shows that $\square$ would be inferred directly from $\{X_1, \dots, X_k\}$. Of course, to know that this simple tree is a counterexample tree for $\{X_1, \dots, X_k\} \Rightarrow \square$ is already to know that $\{X_1, \dots, X_k\} \Rightarrow \square$, because this is the content of 3.3.2 in this case. The upshot of this paragraph is therefore this; that from any counterexample tree T for a theorem $X \Rightarrow \square$, we automatically get a proof of $X \Rightarrow \square$, in which each inference is an application of the following principle:

3.8.11. from $(A_1 \vee B_1), \dots, (A_k \vee B_k)$ one may infer the 'resolvent' $(A_1 \vee \dots \vee A_k \vee B)$, whenever $\{B_1, \dots, B_k\} \Rightarrow B$, where B is a disjunction.

It is this principle which we call the *generalized ground resolution principle*.

THEOREM PROVING

3.9. Discussion of the generalized ground resolution principle

The principle 3.8.11 specializes, in various ways, to all of the various versions of the ground resolution principle (Robinson, 1965), when B is $\square$. When B is not $\square$, 3.8.11 condones inferences of a rather more general character, including all those which involve the notion of equality in an essential way. It is to be noted that in applying 3.8.11 one has to *discover* a B satisfying the side condition $\{B_1, \dots, B_k\} \Rightarrow B$. In the construction of proofs described in 3.8, this discovery is done automatically, and emerges from the information available at an inference point of a counterexample tree. There is, however, a more subtle way of selecting B in that construction than the one there mentioned, which we now explain.

3.9.1. Selecting a B . It is possible to make a better choice of B than simply (as was indicated in 3.8) to set it equal to the disjunction of all the complements of the components of C . The conditions which B has to satisfy, in order that the argument of 3.8 go through, are (in the notation of that argument):

3.9.1.1. $\{B_1, \dots, B_k\} \Rightarrow B$ and $B \Rightarrow \neg C$.

Now it is possible to consider the set M of all disjunctions whatever, in which there occur only the equality symbol, relation symbols and terms that occur in $B_1, \dots, B_k$, and C . *There are only finitely many of these.* A denotation table for the set of atoms in M can be constructed, and with its help, one can compute *all* of the disjunctions B in M satisfying 3.9.1.1, and then choose the *simplest* of these with which to construct the *resolvent* Q 3.8.8. In order that this be done mechanically one must of course specify a computable measure of the simplicity of B , such as: the number of symbols in B , or the number of components in B .

4. GENERAL PROOFS

Now that we have the facility, given a ground proposition $X \Rightarrow \square$ which is true, to construct automatically a *proof* of $X \Rightarrow \square$ by using the method explained in 3.8 of converting a counterexample tree for $X \Rightarrow \square$, we go on to consider next the question of constructing, given a general proposition $X \Rightarrow \square$ which is true, a *proof* of $X \Rightarrow \square$.

4.1. Variants, instances, and substitutions

4.1.1. A *substitution* is an operation θ which can be performed on an expression E to obtain another expression $E\theta$; the operation consists of replacing each occurrence in E of each of a list $x_1, \dots, x_n$ of distinct *individual symbols* by an occurrence of the corresponding term in a list $t_1, \dots, t_n$ of (not necessarily distinct) *terms*. It is always assumed that t_i is different from x_i . We write $\theta = (t_1/x_1, \dots, t_n/x_n)$. The *empty substitution*, conventionally denoted by ϵ , is the (null) operation on E of replacing nothing in E . Thus, $E\epsilon = E$ for all E . The *composition* $\theta\lambda$ of two substitutions is the substitution μ such that $E\mu = (E\theta)\lambda$ for all E . The components of $\theta\lambda$ are easily obtained from those of θ and λ : indeed, if $\theta = (t_1/x_1, \dots, t_n/x_n)$ and $\lambda = (u_1/y_1, \dots, u_m/y_m)$, then

$\theta\lambda = (t_1\lambda/x_1, \dots, t_n\lambda/x_n, u_1/y_1, \dots, u_m/y_m)^*$ where $*$ indicates the operation of deleting any component $t_j\lambda/x_j$ in which $t_j\lambda = x_j$, and any component u_i/y_i such that y_i is among $x_1, \dots, x_n$. Composition of substitutions is associative, and ϵ is both a left and a right identity:

4.1.1.1. $(\theta\lambda)\mu = \theta(\lambda\mu)$ for all θ, λ, μ ;

4.1.1.2. $\epsilon\theta = \theta\epsilon = \theta$ for all θ .

4.1.2. *Instances.* An expression Y is an *instance* of an expression X if $Y = X\theta$ for some substitution θ .

4.1.3. A substitution $\theta = (t_1/x_1, \dots, t_n/x_n)$ is *invertible*, and has the *inverse* $\theta^{-1} = (x_1/t_1, \dots, x_n/t_n)$, if $(x_1/t_1, \dots, x_n/t_n)$ is a substitution, that is, if $t_1, \dots, t_n$ are distinct individual symbols.

4.1.4. *Variants.* An expression Y is a *variant* of an expression X if Y is an instance $X\theta$ of X for some invertible θ such that $X = Y\theta^{-1}$.

Obviously, if Y is a variant of X then X is a variant of Y ; if X is a variant of Y and Y is a variant of Z then X is a variant of Z ; and X is a variant of X .

4.2. Lemma

For any expressions $X_1, \dots, X_n$ and substitutions $\theta_1, \dots, \theta_n$, we can find variants $X'_1, \dots, X'_n$ of $X_1, \dots, X_n$, and a substitution θ , such that:

$$(X_1\theta_1, \dots, X_n\theta_n) = (X'_1\theta, \dots, X'_n\theta)$$

(i.e., $X_i\theta_i = X'_i\theta$ for all $i = 1, \dots, n$).

The proof is very easy.

4.3

If X is a *set* of expressions and θ a substitution, then by $X\theta$ we mean the *set* of all expressions $E\theta$, where E is in X .

4.4

Let X be a set of expressions and θ a substitution. Let P be the partition of X determined by the rule that E and F are in the same block of P if and only if $E\theta = F\theta$. We say that P is *induced in X by θ* .

4.5

Let X be a set of expressions and P a partition of X . We say that P is a *unifiable* partition of X if and only if there is some substitution θ which induces P in X .

4.6

Two substitutions are said to be *equivalent over* a set X of expressions if they induce the same partition in X .

4.6.1. *Comment.* There is obviously an equivalence class of substitutions over X for each unifiable partition of X . If X is finite, there are then clearly only finitely many such partitions and hence only finitely many such equivalence classes of substitutions.

THEOREM PROVING

4.7

Let X be a finite set of expressions. A set $\{\theta_1, \dots, \theta_n\}$ of substitutions is said to be a *basis* of X if for each unifiable partition P of X there is exactly one θ_i in the set which induces P in X .

4.8

Prime bases of sets of expressions. Let X be a finite set of expressions. A basis $\{\sigma_1, \dots, \sigma_n\}$ of X is said to be a *prime basis* of X if, for any basis $\{\theta_1, \dots, \theta_n\}$ of X , we have

4.8.1. $\{\theta_1, \dots, \theta_n\} = \{\sigma_1 \lambda_1, \dots, \sigma_n \lambda_n\}$ for some set $\{\lambda_1, \dots, \lambda_n\}$ of substitutions.

Comment. Every finite set X of expressions has a prime basis. Moreover, given X , we can *compute* a prime basis of X ; and given a prime basis of X we can *compute*, for any other basis of X , the substitutions λ_i of 4.8.1. These computations are made by means of the *prime basis algorithm*, explained in the next paragraph.

4.9. The prime basis algorithm

Given the finite set X of expressions as input, we can, for each partition P of X , calculate the substitution $\sigma(P)$ by applying to P the *unification procedure* described in 4.9.1 below.

Then let $\{\sigma_1, \dots, \sigma_n\}$ be the set of all $\sigma(P)$ such that $\sigma(P)$ induces P in X . This set is a prime basis for X .

4.9.1. *Unification procedure.* Given a partition P of a set X of expressions, we compute a substitution $\sigma(P)$ as follows:

Step 1. Put $\beta_0 = \epsilon$, $k=0$, and go to step 2.

Step 2. If $B\beta_k$ is a singleton for each block B of P , put $\sigma(P) = \beta_k$ and *halt*. Otherwise:

Step 3. let B be any block of P such that $B\beta_k$ is not a singleton, let E, F be two distinct expressions in $B\beta_k$, and let W, Y be the two distinct *expressions* obtained by analyzing E, F as:

$$E = AWR, F = AYS$$

for some (possibly empty) string A of symbols and some strings (possibly empty) R and S of symbols.

Step 4. If one of Y, W is an individual symbol x and the other is a term t in which x does not occur, put $\mu_k = (t/x)$, $\beta_{k+1} = \beta_k \mu_k$, add 1 to k , and return to step 2. Otherwise, put $\sigma(P) = \beta_k$, and *halt*.

4.9.2. *Comment.* The freedom of choice (of B, E, F , and x) in steps 3 and 4 of the unification procedure will of course be removed in some fixed way in any mechanization of the procedure. For our present purposes we assume the method of choice fixed but we do not insist on any one way of fixing it. On this assumption, the sequences $\beta_0, \dots, \beta_k$ and $\mu_0, \dots, \mu_{k-1}$ of substitutions, generated as the procedure returns repeatedly to step 2, are fixed functions of P which we call the *unification sequences for P* . It is straightforward to show

that if θ is any substitution which induces P in X , then $\sigma(P)$ induces P in X and that moreover $\theta = \sigma(P)\lambda$, where $\lambda = \lambda_k$, the final member of the sequence $\lambda_0, \dots, \lambda_k$ of substitutions determined as follows: put $\lambda_0 = \theta$, and then for $j \geq 0$, solve the equation $\mu_j \lambda_{j+1} = \lambda_j$ for λ_{j+1} . For then the equation $\theta = \beta_j \lambda_j$ is easily seen by induction to be satisfied for $j = 0, \dots, k$. For $j = 0$, the equation holds because $\beta_0 = \epsilon$ and $\lambda_0 = \theta$. And if the equation holds for $j < k$ it must hold for $j + 1$, because $\beta_{j+1} \lambda_{j+1} = (\beta_j \mu_j) \lambda_{j+1} = \beta_j (\mu_j \lambda_{j+1}) = \beta_j \lambda_j$.

4.10

Now let $X_1, \dots, X_n$ be sentences and $\theta_1, \dots, \theta_n$ be substitutions, and consider the instances $X_1\theta_1, \dots, X_n\theta_n$ of $X_1, \dots, X_n$ by these substitutions. By Lemma 4.2 we can find variants $X'_1, \dots, X'_n$ of $X_1, \dots, X_n$, and a substitution θ , such that $X_1\theta_1, \dots, X_n\theta_n = X'_1\theta, \dots, X'_n\theta$. If Y is any sentence such that $\{X_1\theta_1, \dots, X_n\theta_n\} \Rightarrow Y$, then $\{X'_1\theta, \dots, X'_n\theta\} \Rightarrow Y$. Let T be the set of all of the terms which occur in any of the sentences $X'_1, \dots, X'_n, Y$. Suppose that the sentence Y has the form $Y_1 t_1 \theta \dots Y_n t_n \theta Z$, where each of the terms $t_1, \dots, t_n$ is in T and none of the strings $Y_1, \dots, Y_n, Z$ contains a term $t\theta$, with t in T . Then put $Y' = Y_1 t_1 \dots Y_n t_n Z$, so that if we apply the substitution θ to Y' we obtain the sentence Y back again. Thus, $\{X'_1\theta, \dots, X'_n\theta\} \Rightarrow Y'\theta$. However, the denotation table for $\{X'_1\theta, \dots, X'_n\theta\} \Rightarrow Y'\theta$, and that for $\{X'_1\theta', \dots, X'_n\theta'\} \Rightarrow Y'\theta'$, where θ' is any substitution equivalent to θ over T , are completely isomorphic; and in particular we have that $\{X'_1\sigma, \dots, X'_n\sigma\} \Rightarrow Y'\sigma$, where $\sigma = \sigma(P)$, P being the partition induced in T by θ . Recalling that we can find λ such that $\theta = \sigma\lambda$, and remarking that $X_i \mapsto X'_i\sigma$ for each i , we conclude that the following is the case (putting $Y'\sigma = X$):

4.10.1. If $\{X_1\theta_1, \dots, X_n\theta_n\} \Rightarrow Y$ then we can find a sentence X such that $\{X_1, \dots, X_n\} \Rightarrow X$ and a substitution λ such that $Y = X\lambda$.

4.11

The 'lifting lemma' 4.10.1 can be used to obtain, from any ground proof of a theorem $\{X_1\theta_1, \dots, X_n\theta_n\} \Rightarrow Y$, a general proof of a theorem $\{X_1, \dots, X_n\} \Rightarrow X$ with the property that Y is an instance of X . One simply takes the given ground proof and applies 4.10.1 repeatedly, from the tips inward.

4.12

But a more general conclusion can be drawn from the discussion in 4.10. Let $X_1, \dots, X_n$ be (not necessarily distinct) sentences, and let $X'_1, \dots, X'_n$ be variants of $X_1, \dots, X_n$ no two of which have an individual symbol in common. Let T be the set of all terms which occur in any of $X'_1, \dots, X'_n$, and let S be a prime basis of T . If σ is any substitution in S and Y' is any sentence, we can determine whether $\{X'_1\sigma, \dots, X'_n\sigma\} \Rightarrow Y'\sigma$, and, if so, obtain by means of 4.10.1 a general theorem of the form $\{X_1, \dots, X_n\} \Rightarrow X$, where $X = Y'\sigma$. The general theorems which are obtainable in this way are all of the general theorems which can be obtained by applying 4.10.1 to ground theorems of the form $\{X_1\theta_1, \dots, X_n\theta_n\} \Rightarrow Y$. We can make special use of this in order to arrive at the generalized resolution principle, in the next paragraph.

4.13. The generalized resolution principle.

A special case of the discussion in 4.12 arises when $Y'\sigma$ comes from $X_1'\sigma, \dots, X_n'\sigma$ by the generalized ground resolution principle 3.8.11. In this case we have a general theorem $\{X_1, \dots, X_n\} \Rightarrow X$ in which X_i is $(A_i \vee B_i)$, $i=1, \dots, n$ and X is the sentence $(A_1' \vee \dots \vee A_n' \vee B')\sigma$, where $(A_1' \vee B_1'), \dots, (A_n' \vee B_n')$ are the variants $X_1', \dots, X_n'$ and B' is any disjunction which satisfies the condition that $\{B_1'\sigma, \dots, B_n'\sigma\} \Rightarrow B'\sigma$.

4.13.1. If, therefore, we apply 4.11 to a proof of $\{X_1\theta_1, \dots, X_n\theta_n\} \Rightarrow \square$, each inference in which is an application of the generalized ground resolution principle, we get a proof of $\{X_1, \dots, X_n\} \Rightarrow \square$, each inference in which is an application of the *generalized resolution principle* stated as follows:

4.13.2. Generalized resolution principle

From $(A_1 \vee B_1), \dots, (A_n \vee B_n)$ one may infer the 'resolvent' $(A_1' \vee \dots \vee A_n' \vee B')\sigma$, provided that $\{B_1'\sigma, \dots, B_n'\sigma\} \Rightarrow B'\sigma$, where $(A_1' \vee B_1'), \dots, (A_n' \vee B_n')$ are variants of the sentences $(A_1 \vee B_1), \dots, (A_n \vee B_n)$, σ is a member of a prime basis of the set T of all terms which appear in $(A_1' \vee B_1'), \dots, (A_n' \vee B_n')$, and B' , where B' is a disjunction.

Comment. It is not necessary that $(A_1 \vee B_1), \dots, (A_n \vee B_n)$ all be different from each other. The intention behind our formulating the principle in this way is to emphasize that the several premisses of an application of the generalized *ground* resolution principle may well include distinct instances of one and the same sentence. Such inferences, when 'lifted' to the general level, correspond to applications of 4.13.2 in which the same sentence appears more than once in the listing of the premisses. The *ground* principle 3.8.11 is simply the special case of 4.13.2 for $\sigma = \epsilon$ and $(A_i' \vee B_i') = (A_i \vee B_i)$, $i=1, \dots, n$.

4.14. The completeness theorem for the generalized resolution principle

The fundamental theorem of logic, in our present notation, states that:

4.14.1. For any finite set X of sentences: if $X \Rightarrow \square$, then, for some $k \geq 1$ $\{X_1\theta_1, \dots, X_k\theta_k\} \Rightarrow \square$, where $X_1, \dots, X_k$ are in X and $\theta_1, \dots, \theta_k$ are substitutions.

From this proposition, the construction of 3.8, and 4.13.1, we obtain the *completeness theorem* for the generalized resolution principle:

4.14.2. For any finite set X of sentences: if $X \Rightarrow \square$ then there is a proof of $X \Rightarrow \square$ in which each inference is an application of the generalized resolution principle.

4.14.3. Comment. Indeed, if $X \Rightarrow \square$ then the *immediate* inference of $\square$, directly from X , is an application of the generalized resolution principle.

4.15

If, in the construction of 3.8, we impose further restrictions on the form of the counterexample trees which may be used, we obtain corresponding restrictions on the forms of the inferences which will be made when the counterexample trees are converted by the construction into proofs. To each such set

of restrictions on the form of counterexample trees will correspond a system of logic with a single inference principle that is a correspondingly restricted form of the generalized resolution principle, and the entire argument will provide the completeness theorem for that particular system of logic.

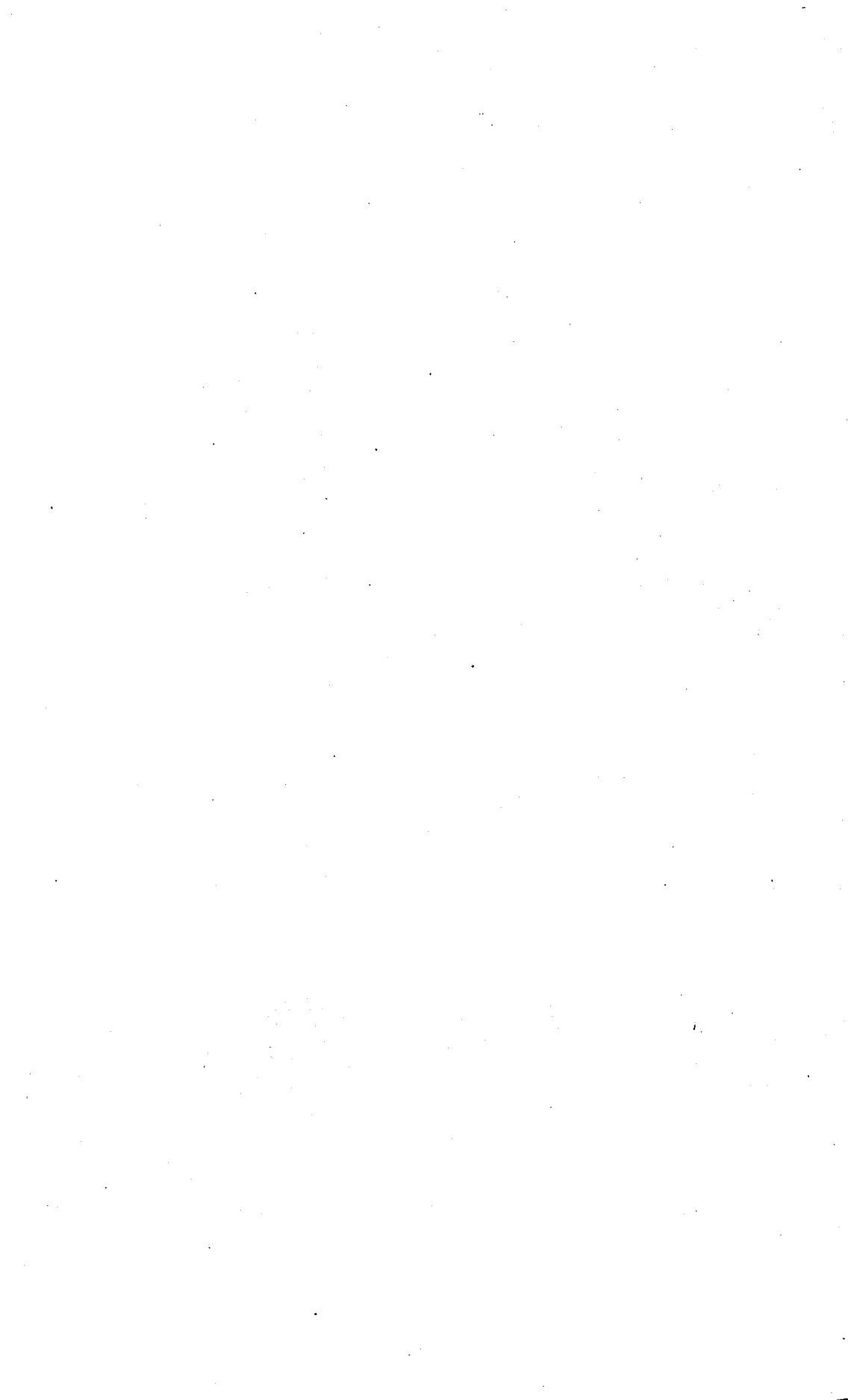
4.15.1. Pairwise resolution. The original *resolution principle* of Robinson (1965) corresponds to the restriction that (in the notation of 3.8) we always have $k=2$ and that C_1, C_2 are always $L, \neg L$ for some atom L in K . Actually this restriction gives rather more, in that the resulting system (as have all systems obtained in this way) 'has equality built in'.

4.15.2. Resolution with set of support. *Example 2:* If we restrict the counterexample tree for $X \Rightarrow \square$ in such a way that, for some satisfiable subset Y of X , the negation of the conjunction C at an inference node N never follows from Y (as may always be done) then the inferred sentence Q at N in the resulting proof will never follow from Y alone (for $C \Rightarrow \neg Q$; and if $Y \Rightarrow Q$ we would have $Y \Rightarrow \neg C$) and will thus be a proof with *set of support* $X - Y$. In this way we obtain the systems (Wos, Carson and Robinson, 1965) in which the *set of support principle* is always observed, and we therefore have the completeness theorem for any such system.

4.15.3. Clash resolution. If all but one of $C_1, \dots, C_k$ always contain a single literal, and the remaining one contains the complements of all of these literals, then we obtain the *clash resolution* system (Robinson, 1967), of which the system of 4.15.1 is a special case.

REFERENCES

- Robinson, J.A. (1965), A machine-oriented logic based on the resolution principle. *J. Ass. comput. Mach.*, **12**, 23-41.
- Robinson, J.A. (1967), A review of automatic theorem-proving. *Annual symposia in applied mathematics XIX*. Providence, Rhode Island: American Mathematical Society.
- 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**, 536-41.



Some Tree-paring Strategies for Theorem proving

David Luckham
Department of Mathematics
University of Manchester

ABSTRACT

Proofs of theorems from elementary algebra and number theory have been obtained using a resolution program with a memory capacity limited to 200 clauses. The object has been to study the effectiveness of model partition strategies in cutting down the number of deductions considered. For this reason proofs were obtained without the unit preference strategy where possible. Our results indicate that these strategies are certainly useful but often have the unfortunate consequence of excluding the shortest proof from consideration. The question of which is the best such strategy for a given problem is not straightforward.

What we shall describe here is essentially a pilot study towards the goal of developing a flexible theorem-proving system. The first step in building a program capable of dealing with elementary propositions from different areas of mathematics is to choose a good basic procedure for first-order proof theory. There now exists a wide selection of procedures, most of them differing greatly in both formulation and implementation, so that accurate comparison of any two from the selection is usually difficult. The choice, therefore, has to be made on the basis of partial answers to practical questions of the following sort. (i) For what domains of reasoning is the mechanical proof procedure likely to be practicable? (ii) How complicated a task is it to program the procedure? (iii) What are the possibilities for improving the procedure, in general, or for special classes of propositions? That is to say, one's natural inclination is to take as first choice the simplest procedure for which there is any really encouraging evidence in reply to question (i), and which presents good chances for improvement.

From this point of view, the Resolution Principle (Robinson, 1965) seems a good choice, since its application to elementary group theory has been

THEOREM PROVING

quite successful (Wos, Carson and Robinson, 1965), and the possibility of more powerful efficiency strategies is indicated in Robinson (1965a). There is an initial question of whether to program this procedure in its simple form or in the hyper-resolution form (Robinson, 1965a). However, it is easily seen that a program for the simple form can be extended (in more than one way) to the hyper-resolution form if need be. At the outset perhaps one should try to write a simple and flexible program with an eye to being able to make additions and modifications easily and, of course, correctly.

On the basis of such considerations we chose to program a simple form of the Resolution Principle (let us call it 'pairwise-resolution' since the program only computes resolvents of pairs of clauses and does not try to do anything more general). This, we recall (see Robinson, 1965 and Luckham, 1967), is a refutation procedure: to prove a theorem T of an elementary theory from axioms (or starting hypotheses) $A_1, \dots, A_n$ the program starts with the set of clauses $\{A_1, \dots, A_n, \neg T\}$, and computes all resolvents of pairs of clauses in the starting set (call these the level 1 deductions), adds these to the set of clauses, computes all further resolvents (level 2 deductions) and repeats this process indefinitely. The empty clause, NIL, will eventually turn up precisely in the case that T is a theorem of the theory, and if it occurs among the level k deductions, we shall say that the program has found a level k proof of T . Figure 1 represents a tree of all possible deductions from a set of four initial clauses (a node at level 1 represents the finite set of resolvents of two level 0 clauses).

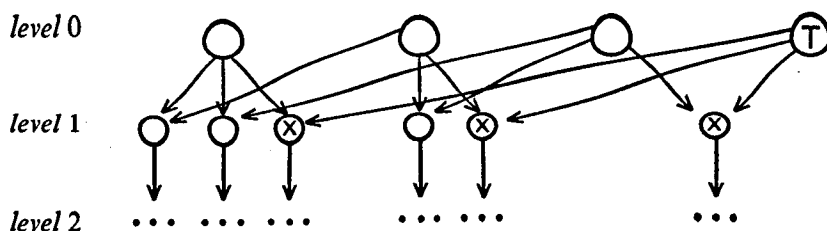


Figure 1

If D is a clause occurring in the tree of all deductions, we may define the proof-tree (or the proof), $Tr(D)$, of a particular occurrence of D as the least set of clauses satisfying, (i) D is in $Tr(D)$, and (ii) if C is in $Tr(D)$ and C is a resolvent of A and B then A and B are in $Tr(D)$. A proof is simply the set $Tr(NIL)$ for some occurrence of NIL, and we shall refer to the level-0 clauses in $Tr(NIL)$ as a *proof set*.

Now, the primary problem with a procedure of this type is one of memory space. The tree of all possible deductions grows so rapidly that it would not be realistic to attempt to generate proofs using such a procedure without incorporating additional strategies for reducing the number of deductions which

must be considered. So, first of all, we turn our attention to the problem of finding effective ways of 'paring down' the tree of deductions to manageable proportions—or more accurately for the theorems we have in mind, that part of the tree containing the deductions up to about level 6 or 7.

The basic flow diagram of a resolution program is given in figure 2; the CHOICE subroutine chooses the next pair of clauses to resolve from the tree of all clauses so far generated and retained, RESOLVE generates the resolvents (or deductions); and EDIT attempts to throw away as many of these as it can without excluding the chance of finding a proof. BOOK-KEEP adds the remaining deductions to the tree, and if NIL occurs among them, the proof which has been found is recovered by tracing back the 'ancestors' of NIL in the tree.

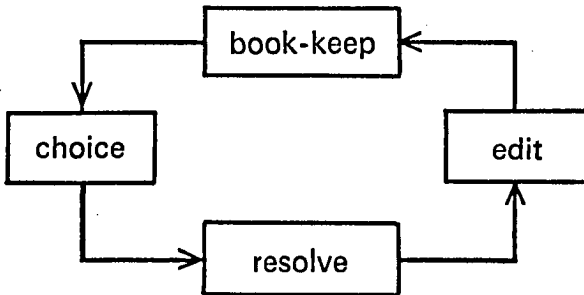


Figure 2

Thus efficiency strategies for theorem proving fall naturally into two classes: choice strategies for deciding the order in which the deduction tree is to be generated, and more strongly, for deciding which part of the tree may be ignored altogether, and edit strategies for eliminating trivial or superfluous deductions.

First a brief word about the latter class of strategy which is very elementary and was held constant throughout the experiments reported below. Deductions were discarded at the edit stage of the program on the basis of two types of consideration. First, gross estimates about the probable simplest form of a proof: a clause was eliminated if (i) it contained more than l literals or (ii) it contained a term with a depth of nesting of function symbols greater than d . In the examples (appendix B with the exception of 4) $l=5$ and $d=3$. Such strategies are of course logically incomplete in that if the gross estimate is wrong and the parameters l or d set too low then all possible proofs will be excluded. Secondly, simple logical criteria for unnecessary deductions: a clause was eliminated if it was (iii) a tautology, (iv) obtainable from another clause by a (1-1) change of variables, (v) subsumed by another clause. The alphabetic variants test (iv) was made only on clauses of length less than 3, and test (v) was made on a clause only with respect to unit clauses, in order to reduce the computation time. In practice the gross strategies were

THEOREM PROVING

important in reducing the deduction trees in the algebra examples (1-5) to a manageable size, but less so in the number theory theorems. It is certainly true that more sophisticated edit strategies can be formulated which would result in a further large decrease in the number of unnecessary deductions made by the program, and we shall say a little more about this later on.

We turn now to our main consideration, the choice strategies. A good example of the 'weak' form of choice strategy which affects only the order in which deductions are made, is the Unit Preference of Wos, Carson and Robinson (1964). This, very simply, is as follows. The program generates the deduction tree in a level-wise order with the exception that those deductions which follow from a 'new' deduction (of level n , say) and the set of unit clauses are generated first as far as level $n+k$. (Here k has a previously fixed value; if k was not predetermined, the strategy would be incomplete because it would be possible for the program to 'run away' down a useless branch of the tree.) Since a number of the end deductions of a proof usually involve a unit clause this allows the program to 'look ahead' and find a level $n+k$ proof without generating all of the levels $n, n+1, \dots, n+k$. Obviously such a strategy can be crucial (as it was in our example 4(b)) but we have done without it as much as possible because our primary interest has been to study the effectiveness of other ways of paring the levels of the deduction tree. (Notice that proofs of example 1 could be obtained using only unit preference!)

The first example of a 'strong' choice strategy is the Set-of-Support (Wos, Carson and Robinson, 1965) which depends on the following theorem.

Theorem 1. Let S be an inconsistent set of clauses and T a subset of S such that $S-T$ is consistent. Then a proof will occur in that part of the tree of all possible deductions from S which contains at level 1 only those deductions that follow from at least one of the clauses in T .

For example in figure 1, if T is the level 0 clause indicated, then only the marked level 1 clauses and those clauses of higher level which do not depend on any of the unmarked level 1 clauses, need be generated. T is said to support this part of the tree. The effect of applying this theorem is to delay the exponential growth of the number of deductions generated by the program until level 2, provided a suitable set T can be found. In the simple examples so far considered where one expects to find short proofs (say level 5 or 6) this strategy has been shown to be very useful and the choice of T to be sometimes critical (see Wos, Carson and Robinson, 1965, for details). The theorem itself is really a statement about the way in which proof-trees may be reordered so as to preserve the property of being a proof: clearly if $S-T$ is consistent, deductions from clauses of T must occur at some level in any proof-tree; the theorem simply states that there exist proofs all of whose level 1 deductions follow from at least one clause of T . (In fact, any set $T \subseteq S$ will provide a logically complete strategy when used as the Support set if it contains a clause occurring in a proof.) Hence there is the possibility that this strategy may

sometimes exclude the shortest proofs from consideration and thus defeat itself by forcing the program to search deeper levels of the deduction tree; however, the use of unit preference will tend to counter this effect.

It is natural to ask if more powerful choice strategies can be found which restrict the exponential growth of the deduction tree at every level. Towards this goal, J.A. Robinson has suggested the method of P_1 -deduction (Robinson, 1965a); a clause is called *positive* if it contains no negated atoms:

Theorem 2. If S is an inconsistent set of clauses then there is a proof in that part of the tree of all deductions from S consisting of only those deductions that follow from a positive clause.

This appears to be a very useful fact to know, for now one can choose to compute at each level in the deduction tree only those resolvents of pairs of clauses one of which is positive (Robinson calls these P_1 -deductions). So one might expect the strategy to yield a significant paring down of the deduction tree, and initial computations with elementary group theory examples support this (a sceptic would point out that the axioms here have a very special form), but one might also expect the problem of having to search deeper levels of the tree. However, leaving practical considerations aside for the moment, it is clear that this theorem is capable of generalization so as to yield a family of choice strategies, and we consider next a very simple-minded extension of it.¹

We adopt the notation and terminology of Luckham (1967) and Robinson (1965). The following symbols denote the concepts indicated: S , a set of clauses; H , the Herbrand domain of terms composed from the variables and function symbols in S ; K , a subset of H ; $H(S)$, the set of all instances of S obtained by substituting terms from H (i.e. the Herbrand expansion of S);² $R(A, B)$, a resolvent of clauses A and B . As before, $R(S)$ denotes the set of all clauses in S and all resolvents of pairs of clauses in S ; $R^{n+1}(S) = R(R^n(S))$. We shall sometimes treat a clause C (i.e. a disjunction of literals) simply as a set of literals; e.g. $M \cap C = \phi$ denotes that none of the literals in C occurs in M .

In the following we deal with a special class of models, relational systems on H .

Definitions

- (i) M is a model =df. M is a non-empty set of literals occurring in $H(S)$, no two of which are complementary.
- (ii) M is a model for clause C (notation $M \models C$) =df.
 $M \cap C_i \neq \phi$ for all $C_i \in H(C)$.
- (iii) $\langle M_1, M_2 \rangle$ is a model partition of $H(S)$ =df.
 - (a) M_1 and M_2 are models,
 - (b) $M_1 \cup M_2$ is the set of all literals in $H(S)$,

¹ An extension has been suggested by B. Meltzer (1966).

² Notice that this is *not* the set of ground instances of S defined in Robinson (1965).

THEOREM PROVING

(c) $M_1 \cap M_2 = \phi$,

(d) for all n , if $C \in R^n(S)$ and $C \neq \text{NIL}$, then

either $M_1 \models C$ and $H(C) \cap M_2 = \phi$

or $M_2 \models C$.

(iv) $R_1^n(S) = \text{df. } \{C \mid C \in R^n(S) \ \& \ M_1 \models C \ \& \ H(C) \cap M_2 = \phi\}$

$R_2^n(S) = \text{df. } \{C \mid C \in R^n(S) \ \& \ M_2 \models C\}$.

Remarks (a). If $\text{NIL} \notin R^n(S)$ then $R_1^n(S) \cup R_2^n(S) = R^n(S)$.

(b) The conjunction of clauses in S is satisfiable if and only if there is a model M (in our sense) such that $M \models S$ (this is usually called Herbrand's theorem).

(c) If S is not satisfiable then $R_1^n(S) \neq \phi$ and $R_2^n(S) \neq \phi$; in particular (for the case $n=0$) $S_1 \neq \phi$ and $S_2 \neq \phi$.

(d) Condition (d) places a strong restriction on the class of pairs of models that form partitions. The simplest example of a partition (when S is unsatisfiable) is to take as M_1 the set of all positive literals in $H(S)$, and as M_2 the set of all negated literals in $H(S)$. More generally, if the set of literals in S is partitioned into two sets U_1 and U_2 such that $H(U_1)$ and $H(U_2)$ are disjoint and neither contains a complementary pair, then $\langle H(U_1), H(U_2) \rangle$ is a model partition; in this case the models are closed under substitutions from H and it is easily seen that (d) is satisfied. Thus, if S consists of the six clauses

1. $P(x) \vee L(1g(x))$

2. $P(x) \vee L(g(x)x)$

3. $\neg P(x) \vee \neg D(xa)$

4. $\neg L(1x) \vee P(f(x))$

5. $\neg L(1x) \vee D(f(x)x)$

6. $\neg P(a)$

one example of a partition is

$M_1 = H(\{\neg P(x) \ \neg D(xa) \ L(1g(x))\})$

$M_2 = H(\{P(x) \ D(f(x)x) \ L(g(x)x) \ \neg L(1x)\})$;

here, $S_1 = \{3, 6\}$ and $S_2 = \{1, 2, 4, 5\}$.

Definition. Corresponding to each partition $\langle M_1, M_2 \rangle$ there is a restricted tree of resolvents defined by:

$\tilde{R}(S) = \text{df. } S \cup \{C \mid C = R(A, B) \ \& \ A \in S_1 \ \& \ B \in S_2\}$

$\tilde{R}^{n+1}(S) = \text{df. } \tilde{R}^n(S) \cup \{C \mid C = R(A, B) \ \& \ A \in \tilde{R}_1^n(S) \ \& \ B \in \tilde{R}_2^n(S)\}$.

Remarks. Clearly $\tilde{R}^n(S) \subseteq R^n(S)$. But if a partition is to be useful its restricted tree must contain somewhat fewer deductions at each level (or at the lower levels) than the general tree; e.g. at level 1, $\tilde{R}(S) \neq R(S)$ if and only if $\neg(R(S_2) \subseteq \tilde{R}(S))$.

Now, whenever $\langle M_1, M_2 \rangle$ is a model partition of $H(S)$, the choice strategy of generating the restricted tree, $\tilde{R}(S), \tilde{R}^2(S), \dots$, of deductions from S is complete:

Theorem 3. There exists an n such that $\text{NIL} \in R^n(S)$ if and only if there exists an m such that $\text{NIL} \in \tilde{R}^m(S)$.

The proof has been relegated to appendix A.

As corollaries of Theorem 3 we have the completeness of the strategies suggested in Robinson (1965a) (i.e. Theorem 2) and Meltzer (1966), since these are particular model partition strategies. Also, the proof in appendix A contains some extra information: given any R -proof in the tree of all deductions, there is a corresponding $\tilde{R}$ -proof in the restricted tree (no matter which model partition is chosen) having the same proof set (not counting repetitions) and containing only terms occurring in the given proof. This does not say that if we take the shortest R -proof then the corresponding $\tilde{R}$ -proof will be the shortest $\tilde{R}$ -proof! But it does at least tell us that some of the gross editing strategies will not interact unfavourably with the model partition strategies if they have not already excluded all proofs.

Returning to a practical point of view, we now ask how best to make use of Theorem 3. We need to know if proofs can be generated efficiently (in some sense) by using model partition choice strategies, and, if so, how to find a strategy that works well on a given problem. These questions depend on the relation between n and m (for clearly $m \geq n$, and in the case of most partitions we must expect $m > n$) and on the growth of the levels $\tilde{R}^i(S)$. Unfortunately, useful information about these factors is hard to come by, and the best plan is simply to run some programs and see what happens.

Appendix B contains a sample of results obtained by running a resolution program with a small selection of model partition choice strategies. The edit strategies were the same in all cases except example 4(b). The program conformed to the basic flow diagram of figure 2, was written in LISP (McCarthy 1962) and run on an IBM 7090. The memory capacity was only 200 clauses, a severe limitation. All the model partitions used were pairs of models closed under substitutions obtained by the simple procedure outlined in remark (d) above. Thus in the group theory examples 1-4 two models were used, $M^+ = H(\{P(x y z)\})$, $M^- = H(\{\neg P(x y z)\})$,¹ which result in two partitions, $\langle M^+, M^- \rangle$ and $\langle M^-, M^+ \rangle$. The first strategy (called $+P$ deduction) generates Robinson's P_1 deductions, and the second (called $-P$ deduction) generates those deductions that follow from a clause containing only negated literals. The strategies in the number theory examples were equally simple: e.g. in example 7, $\{+L, -P, +D\}$ refers to the strategy obtained by using the partition $M_1 = \{L(xy), \neg P(x), D(xy)\}$ and $M_2 = \{\neg L(xy), P(x) \neg D(xy)\}$; this generates those deductions depending on a clause containing only positive occurrences of relation symbols L, D and negated occurrences of P .

The use of only these choice strategies, without unit preference except in 4(b), enabled proofs to be obtained within the limitations of the program, so that one may conclude that these strategies are of practical value (certainly the program could not have obtained proofs using only the editing strategies). It is interesting to compare the statistics given here with those in Wos, Carson and Robinson (1965) where a program with a much larger memory capacity

¹ Henceforth all the models discussed are closed under substitutions, so we omit the H .

and roughly the same editing strategies, but using as choice strategies unit preference and set of support, was tried on similar problems. We note that all of the examples given here have short proofs (about level 5) so that the effect of unit preference, when it was used, on the speed and efficiency in finding a proof, was significant. The question of which partition is best for a given problem is not so easy to answer. It is tempting to choose the one such that $\bar{R}^1(S)$ is smallest, but this does not guarantee that the proof generated will be the shortest or the most efficient in terms of the efficiency ratio of appendix B (see examples 6-8). However, the best practical suggestion at the moment is to use this choice strategy together with unit preference (e.g. example 4).

The translation of resolution proofs given here into a more natural form is easy. It will be seen that the steps in the number theory proofs correspond to the steps in the proofs generated by Wang's natural deduction system (Wang, 1965). We conjecture that the two systems will yield roughly the same proofs of elementary theorems with about the same efficiency. If this is true, then a resolution procedure (which is much simpler) seems to be a good choice for the basic logical part of a 'stock-in-trade' theorem-proving system.

Lastly, a few words about three of the possible improvements. An immediate improvement can be made to the editing strategies. Many of the deductions retained by the program are trivial in the sense that all further deductions depending on them are either trivial or already known (and this fact can be determined by looking ahead a few levels in the deduction tree). For example, the clause, $\neg P(g(x_1) x_8 e) \vee P(e x_8 x_1)$ (ex. 3, proof (b), line 7), is non-trivial in example 3 but is clearly trivial in the converse problem (i.e. to prove the existence of right inverses from the left inverse and identity axioms) although it would not be eliminated at the moment. This situation can be improved by a combination of 'lookahead' and 'reduction to normal form' procedures. Secondly, the addition of the equality relation to the resolution procedure in a sensible way (see J.A. Robinson, this volume) would enable the sort of theorems dealt with in Norton (1966) to be obtained by a very limited program. Finally, the use of efficient decision procedures for classes of sets of clauses of length 2 (see Krom, 1967) could provide a good improvement on the unit preference strategy.

APPENDIX A

Let K be a subset of H . In the proof below we shall use the notions of resolution and model restricted to a set K of terms. Let U be a subset of the clauses of S .

Definitions

- (i) $K(U|K) = df.$ the set of instances of clauses of U containing only terms in K .
- (ii) $R(U|K) = df.$ the subset of $R(U)$ consisting of those clauses containing only terms in K .

(iii) M is a model restricted to K for clause C (notation $M \models_K C$) = *df.* $M \cap C_i \neq \phi$ for all substitution instances C_i of C containing only terms in K .

Proof of Theorem 3. Let $\tilde{R}$ be associated with the model partition $\langle M_1, M_2 \rangle$.

If $\text{NIL} \in \tilde{R}^m(S)$ then $\text{NIL} \in R^m(S)$, so in one direction the theorem is trivial.

Assume $\text{NIL} \in R^n(S)$, and let U be the proof set of a proof-tree and K be the finite set of terms used in obtaining that proof-tree. Note that $\text{NIL} \in R^n(U|K)$.

We restrict consideration to $\tilde{R}(U|K)$; below we shall write $\tilde{R}_j^i$ to denote $\tilde{R}_j^i(U|K)$.

CLAIM: if $\text{NIL} \notin \tilde{R}^i(U|K)$ then $\tilde{R}^{i+1}(U|K) \neq \tilde{R}^i(U|K)$.

In order to prove this claim, let us first consider the set $K(U|K)$. $R^n(U|K)$ is obtained from $K(U|K)$ by complementary literal elimination (c.l.e.) on pairs of clauses without making any further substitutions. If there is a model M containing a literal from each clause in $K(U|K)$, it is not possible to obtain the empty clause by c.l.e. Therefore, for every model M there is a clause C in $K(U|K)$ such that $M \cap C = \phi$. Furthermore, because $U \subseteq \tilde{R}^i$, for every model M there is a clause C in $\tilde{R}^i$ and a substitution σ such that the terms of $C\sigma$ belong to K and $M \cap C\sigma = \phi$.

Let M' be a model restricted to K for $\tilde{R}_1^i$ having a maximal intersection with M_2 ; M' satisfies

(i) $M' \models_K \tilde{R}_1^i$

(ii) $(\forall M) (M' \cap M_2 \subsetneq M \cap M_2 \rightarrow \neg(M \models_K \tilde{R}_1^i))$.

Consider any $C\sigma$ (as above) such that $M' \cap C\sigma = \phi$. This means that M' is not a model restricted to K for C , so that $C \notin \tilde{R}_1^i$. Thus $C \notin \tilde{R}_2^i$ (because $\tilde{R}_1^i \cup \tilde{R}_2^i = \tilde{R}^i$), and therefore $M_2 \cap C\sigma \neq \phi$.

Choose such a C and σ so that $M_2 \cap C\sigma$ is minimal. Let $L \in M_2 \cap C\sigma$. Then L is a literal (containing only terms in K) such that $L \in M_2 - M'$. This implies that $\neg L \in M'$ because $M' \cap M_2$ is maximal: indeed there must be a clause D in $\tilde{R}_1^i$ which 'forces' $\neg L$ to be in M' in the sense that there is a substitution θ such that the terms of $D\theta$ belong to K and $D\theta \cap M' = \{\neg L\}$; if this were not so, $\neg L$ could be omitted from M' without spoiling condition (i), and L introduced instead, thus contradicting condition (ii).

Hence there is a resolvent E of the above C and D which has an instance expressible in the form (neglecting separation of variables), $E\tau = (C\sigma - \{L\}) \cup (D\theta - \{\neg L\})$. It is clear that $E \in \tilde{R}^{i+1}(U|K)$. So it remains only to argue that E is not already in $\tilde{R}^i$:

(a) if $E = \phi$ then $E \notin \tilde{R}^i$ by hypothesis

(b) $E\tau \cap M' = \phi$, therefore $E \notin \tilde{R}_1^i$

(c) Suppose $E \in \tilde{R}_2^i$; $D\theta \cap M_2 = \phi$ (because $D \in \tilde{R}_1^i$) so that

$M_2 \cap E\tau \subsetneq M_2 \cap C\sigma$, which contradicts the minimality of $M_2 \cap C\sigma$.

This proves the claim.

Now, starting with U , compute successively $\tilde{R}^1(U|K)$, $\tilde{R}^2(U|K)$, This process must 'close' after a finite number of steps since restriction to the

THEOREM PROVING

finite set of terms K means that only a finite number of clauses can be generated. Thus there must be an m such that $NIL \in \tilde{R}^m(U|K)$, which implies that $NIL \in \tilde{R}^m(S)$. Q.E.D.

Corollary. Given any proof in the tree of all deductions, $R^n(S)$, $n=1,2,\dots$ and any model partition, there is a proof in the restricted tree, $\tilde{R}^n(S)$, $n=1,2,\dots$, having the same proof set and containing only terms occurring in the given proof.

APPENDIX B

Some examples of proofs

Here are some examples of resolution proofs of theorems from elementary algebra (previously considered by Wos *et al.*, 1965) and elementary number theory (based on an analysis of proofs of such theorems given by Wang (1965). Each line in a proof is a disjunction of literals (the disjunction connective is omitted), and each step in a proof is made from the preceding lines indicated by pairwise resolution. The *efficiency ratio* (e.r.) of a proof is the ratio of the number of deductions in the proof to the total number of deductions retained in memory when the proof was found. This is a somewhat hasty measure of performance, but will serve to give the reader a rough idea of what the program did. It will be noticed in some of the proofs that some deductions can be eliminated; this is because the machine remembers only the first way it finds of making a deduction.

Example	Strategy	e.r.	e.r.(u.p.)	Level
1	$+P$	1/12	1/2	4
1	$-P$	1/5	1	4
2	$-P$	1/7	3/7	6
3	$-P$	1/17	1/7	7
3	$+P$	1/13	1/5	5
4	$-P$	—	1/4	10
4	$+P$	1/10	1/7	6
5	$-P, +S, -R$	1/12	—	4
6	$+P, +M, -D$	1/9	1/4	4
6	$+P, -M, -D$	1/15	1/4	6
7	$+L, -P, +D$	1/3	—	7
7	$+L, +P, +D$	1/5	—	5
8	$+P, +L, +D$	1/5	—	5
8	$-P, +L, +D$	1/2	—	7

Example 1

'In a closed associative system with left and right solutions to equations there is an identity element.'

Two proofs were found without using unit preference; proof (a) using $+P$

deduction (positive clauses against non-positive clauses), and proof (b) using $-P$ deduction (negative clauses against non-negative clauses). In this case the $-P$ strategy gave a more efficient result.

1. $P(g(x_1 x_2) x_1 x_2)$ (left solutions)
 2. $P(x_1 h(x_1 x_2) x_2)$ (right solutions)
 3. $P(x_1 x_2 f(x_1 x_2))$ (closure)
 4. $\neg P(x_1 x_2 x_3) \neg P(x_2 x_4 x_5) \neg P(x_1 x_5 x_6)$
 $P(x_3 x_4 x_6)$ (associativity)
 5. $\neg P(x_1 x_2 x_3) \neg P(x_2 x_4 x_5) \neg P(x_3 x_4 x_6)$
 $P(x_1 x_5 x_6)$
 6. $\neg P(j(x_1) x_1 j(x_1))$ (no right identity)
- Proof (a)*
7. $P(x_6 h(x_1 x_2) x_9) \neg P(x_4 x_2 x_9) \neg P(x_4 x_1 x_6)$ from 4, 2
 8. $\neg P(g(x_1 x_2) x_5 x_{12}) P(x_2 h(x_1 x_5) x_{12})$ from 7, 1
 9. $P(x_5 h(x_4 x_4) x_5)$ from 8, 1
 10. NIL from 9, 6
- Proof (b)*
7. $\neg P(x_3 x_7 j(x_1)) \neg P(x_4 x_1 x_7) \neg P(x_3 x_4 j(x_1))$ from 4, 6
 8. $\neg P(x_1 x_4 x_{10}) \neg P(g(x_1 j(x_4)) x_{10} j(x_4))$ from 7, 1
 9. $\neg P(x_4 x_7 x_4)$ from 8, 1
 10. NIL from 9, 2

Example 2

'In a group the right identity element is also a left identity.'

Using $-P$ deduction, a level 6 proof was found without unit preference (with 47 uneliminated deductions in memory). A similar attempt using $+P$ deduction did not yield a proof after 150 deductions were generated and kept.

1. $P(x_1 g(x_1) e)$ (right inverses)
 2. $P(x_1 e x_1)$ (right identity)
 3. $\neg P(x_1 x_2 x_3) \neg P(x_2 x_4 x_5) \neg P(x_1 x_5 x_6)$
 $P(x_3 x_4 x_6)$ (associativity)
 4. $\neg P(x_1 x_2 x_3) \neg P(x_2 x_4 x_5) \neg P(x_3 x_4 x_6)$
 $P(x_1 x_5 x_6)$
 5. $\neg P(e a a)$ (e is not a left identity)
- Proof*
6. $\neg P(x_4 x_5 a) \neg P(x_3 x_5 a) \neg P(e x_3 x_4)$ from 5, 4
 7. $\neg P(e x_7 a)$ from 6, 2
 8. $\neg P(x_1 x_5 a) \neg P(x_2 x_4 x_5) \neg P(x_1 x_2 e)$ from 7, 3
 9. $\neg P(a x_4 e) \neg P(x_4 x_6 e)$ from 8, 2
 10. $\neg P(a x_1 e)$ from 9, 1
 11. NIL from 10, 1

THEOREM PROVING

Example 3

'In a group with right inverses and right identity every element has a left inverse.'

Two proofs were obtained without unit preference using (a) $-P$ deduction (giving a level 7 proof) (b) $+P$ deduction (a level 5 proof). The rate of growth of the deduction tree in each case was as follows:

Level	1	2	3	4	5	6
Number of retained clauses (proof (a))	7	14	22	35	64	100
Number of retained clauses (proof (b))	17	35	38	97	—	—
1. $P(x_1 g(x_1)e)$						(right inverses)
2. $P(x_1 e x_1)$						(right identity)
3. $\neg P(x_1 x_2 x_3) \neg P(x_2 x_4 x_5) P\neg(x_1 x_5 x_6)$ $P(x_3 x_4 x_6)$						
4. $\neg P(x_1 x_2 x_3) \neg P(x_2 x_4 x_5) \neg P(x_3 x_4 x_6)$ $P(x_1 x_5 x_6)$						
5. $\neg P(x_1 a e)$						(a has no left inverse)
<i>Proof (a)</i>						
6. $\neg P(x_5 x_6 e) \neg P(x_4 x_8 a) \neg P(x_1 x_4 x_5)$						from 5, 4
7. $\neg P(e x_8 a) \neg P(x_1 x_8 e)$						from 6, 2
8. $\neg P(e g(x_1) a)$						from 7, 1
9. $\neg P(x_1 x_5 a) \neg P(x_2 g(x_8) x_5) \neg P(x_1 x_2 e)$						from 8, 3
10. $\neg P(a x_4 e) \neg P(x_4 g(x_{10}) e)$						from 9, 2
11. $\neg P(a x_{12} e)$						from 10, 1
12. NIL						from 11, 1
<i>Proof (b)</i>						
6. $P(x_5 x_6 x_1) \neg P(x_4 x_8 e) \neg P(x_1 x_4 x_5)$						from 2, 3
7. $\neg P(g(x_1) x_8 e) P(e x_8 x_1)$						from 1, 6
8. $P(x_1 x_7 x_8) \neg P(x_1 x_8 x_8) \neg P(e x_8 x_7)$						from 2, 4
9. $P(e g(g(x_3)) x_3)$						from 1, 7
10. $\neg P(e g(x_1) x_9) P(x_1 x_9 e)$						from 1, 8
11. $P(g(x_9) x_9 e)$						from 9, 10
12. NIL						from 11, 5

Example 4

'If an associative system has an identity element and the square of every element is the identity, then the system is commutative.'

Two proofs were obtained; proof (a) using $+P$ deduction without unit preference was found at level 6; proof (b) using $-P$ deduction was found at level 10 only by using a special unit preference 'lookahead' on clauses of length ≥ 4 .

1. $P(e x_1 x_1)$ (left identity)
2. $P(x_1 e x_1)$ (right identity)
3. $\neg P(x_1 x_2 x_3) \neg P(x_2 x_4 x_5) \neg P(x_1 x_5 x_6)$
 $P(x_3 x_4 x_6)$ (associativity)

4. $\neg P(x_1 x_2 x_3) \neg P(x_2 x_4 x_5) \neg P(x_3 x_4 x_6)$
 $P(x_1 x_5 x_6)$ (every element has order 2)
 5. $P(x_1 x_1 e)$ (elements a, b
 6. $P(a b c)$ do not commute)
 7. $\neg P(b a c)$

Proof (a)

8. $P(x_5 x_1 x_8) \neg P(x_3 e x_8) \neg P(x_3 x_1 x_5)$ from 5, 3
 9. $P(x_1 x_7 x_8) \neg P(e x_8 x_8) \neg P(x_1 x_6 x_7)$ from 5, 4
 10. $P(c x_5 x_7) \neg P(a x_6 x_7) \neg P(b x_5 x_8)$ from 6, 3
 11. $\neg P(x_3 x_1 x_9) P(x_3 x_9 x_1)$ from 1, 9
 12. $\neg P(x_7 x_3 x_1) P(x_1 x_3 x_7)$ from 2, 8
 13. $\neg P(a e x_9) P(c b x_9)$ from 5, 10
 14. $P(c b a)$ from 2, 13
 15. $P(c a b)$ from 14, 11
 16. $P(b a c)$ from 15, 12
 17. NIL from 16, 7

Proof (b)

8. $\neg P(x_4 x_5 c) \neg P(x_3 x_5 a) \neg P(b x_3 x_4)$ from 7, 4
 9. $\neg P(b x_7 a) \neg P(e x_7 c)$ from 5, 8
 10. $\neg P(b c a)$ from 1, 9
 11. $\neg P(x_1 x_5 a) \neg P(x_2 c x_5) \neg P(x_1 x_2 b)$ from 10, 3
 12. $\neg P(x_5 c x_{12}) \neg P(x_1 x_{12} a) \neg P(x_3 x_4 b)$
 $\neg P(x_2 x_4 x_5) \neg P(x_1 x_2 x_3)$ from 11, 4
 13. $\neg P(x_1 e a) \neg P(x_3 x_4 b) \neg P(x_2 x_4 c)$
 $\neg P(x_1 x_2 x_3)$ from 12, 5
 14. $\neg P(x_3 x_4 b) \neg P(x_2 x_4 c) \neg P(a x_2 x_3)$ from 13, 2
 15. $\neg P(x_2 b c) \neg P(a x_2 e)$ from 14, 1
 16. $\neg P(a b c)$ from 15, 5
 17. NIL from 16, 6

Example 5

'If S is a non-empty subset of a group such that if x, y belong to S then $x.y^{-1}$ belongs to S , then the identity e belongs to S .'

A level 4 proof was obtained (46 clauses generated and retained) without unit preference, by using the model partition: $M_1 = \{\neg P(x y z) \neg R(x y), S(x)\}$, $M_2 = \{P(x y z), R(x y) \neg S(x)\}$.

1. $P(e x_1 x_1)$ (identity axioms)
 2. $P(x_1 e x_1)$
 3. $P(x_1 g(x_1) e)$ (inverses)
 4. $P(g(x_1) x_1 e)$
 5. $P(x_1 x_2 f(x_1 x_2))$ (closure)
 6. $R(x_1 x_1)$ (equality axioms)
 7. $\neg R(x_1 x_2) \neg R(x_2 x_3) R(x_1 x_3)$
 8. $\neg R(x_1 x_2) R(x_2 x_1)$

THEOREM PROVING

9. $\neg P(x_1 x_2 x_3) \neg P(x_1 x_2 x_4) R(x_3 x_4)$
10. $\neg P(x_1 x_2 x_3) \neg P(x_2 x_4 x_5) \neg P(x_1 x_5 x_6)$
 $P(x_3 x_4 x_6)$
11. $\neg P(x_1 x_2 x_3) \neg P(x_2 x_4 x_5) \neg P(x_3 x_4 x_6)$
 $P(x_1 x_5 x_6)$
12. $\neg R(x_1 x_2) \neg P(x_3 x_4 x_1) P(x_3 x_4 x_2)$
13. $\neg R(x_1 x_2) \neg P(x_3 x_1 x_4) P(x_3 x_2 x_4)$
14. $\neg R(x_1 x_2) \neg P(x_1 x_3 x_4) P(x_2 x_3 x_4)$
15. $\neg R(x_1 x_2) R(f(x_3 x_1) f(x_3 x_2))$
16. $\neg R(x_1 x_2) R(f(x_1 x_3) f(x_2 x_3))$
17. $\neg R(x_1 x_2) R(g(x_1) g(x_2))$
18. $\neg S(x_1) \neg S(x_2) \neg P(x_1 g(x_2) x_3) S(x_3)$ (premisses of the theorem)
19. $\neg S(x_1) \neg R(x_1 x_2) S(x_2)$
20. $S(a)$
21. $\neg S(e)$ (negation of theorem)

Proof

22. $S(x_3) \neg P(x_1 g(a) x_3) \neg S(x_1)$ from 20, 18
23. $\neg P(a g(a) x_4) S(x_4)$ from 20, 22
24. $S(e)$ from 23, 3
25. NIL from 24, 21

Example 6

'If a is a prime number and $a = \frac{b^2}{c^2}$ then a divides b .'

This is the first step in the proof that the square root of any prime is irrational, and is shown here to be easily deducible from Euclid's theorem: if a prime divides the product of two integers then it must divide at least one of the integers. The choice of 'starting hypotheses' is motivated by the analysis in Wang (1965) of the proof of the irrationality of the square root of a prime. Two proofs were obtained without unit preference: proof (a) (level 4) using the model partition

$$M_1 = \{P(x), M(x y z), \neg D(xy)\}, M_2 = \{\neg P(x), \neg M(x y z), D(x y)\},$$

and proof (b) (level 6) using the model partition

$$M_1 = \{P(x), \neg M(x y z), \neg D(x y)\}, M_2 = \{\neg P(x), M(x y z), D(x y)\}.$$

The growth of the deduction trees was as follows:

Level	1	2	3	4	5
Number of clauses retained (proof (a))	16	31	35	—	—
Number of clauses retained (proof (b))	11	18	19	29	78

(Below, interpret $P(x)$ as ' x is a prime', $M(x y z)$ as ' $x \times y = z$ ', $D(x y)$ as ' x divides y ', and $S(x)$ as x^2 .)

1. $P(a)$
2. $M(a S(c) S(b))$
3. $M(x_1 x_1 S(x_1))$
4. $\neg M(x_1 x_2 x_3) M(x_2 x_1 x_3)$
5. $\neg M(x_1 x_2 x_3) D(x_1 x_3)$
6. $\neg P(x_1) \neg M(x_2 x_3 x_4) \neg D(x_1 x_4) D(x_1 x_2)$
 $D(x_1 x_3)$
7. $\neg D(a b)$

Proof (a)

8. $D(x_3 x_1) \neg D(x_3 S(x_1)) \neg P(x_3)$ from 3, 6
9. $\neg P(a) \neg D(a S(b))$ from 7, 8
10. $\neg D(a S(b))$ from 1, 9
11. $D(a S(b))$ from 2, 5
12. NIL from 10, 11

Proof (b)

8. $D(a x_2) \neg D(a x_4) \neg M(x_2 b x_4) \neg P(a)$ from 7, 6
9. $\neg P(a) \neg M(b b x_5) \neg D(a x_5)$ from 7, 8
10. $\neg D(a x_6) \neg M(b b x_6)$ from 1, 9
11. $\neg M(b b x_3) \neg M(a x_2 x_3)$ from 5, 10
12. $\neg M(a x_4 S(b))$ from 3, 11
13. NIL from 2, 12

Example 7

'Any number greater than 1 has a prime divisor.'

We give two proofs obtained without unit preference: a level 7 proof (a) was found using model partition $M_1 = \{L(x), \neg P(x), D(x y)\}$, $M_2 = \{\neg L(x), P(x), \neg D(x y)\}$; a level 5 proof (b) was found with $M_1 = \{L(x), P(x), D(x y)\}$, $M_2 = \{\neg L(x), \neg P(x), \neg D(x y)\}$. The starting hypotheses (lines 1-9 below) include the least number principle; interpret $D(x y)$ as ' x divides y ', $L(x y)$ as ' $x < y$ ', and $P(x)$ as ' x is a prime'. The growth of the deduction trees was as follows:

Level	1	2	3	4	5	6	7
Number of clauses retained (proof (a))	14	17	24	29	35	41	—
Number of clauses retained (proof (b))	24	34	69	93	—	—	—
1. $D(x_1 x_1)$							
2. $\neg D(x_1 x_2) \neg D(x_2 x_3) D(x_1 x_3)$							(properties of 'divides')
3. $P(x_1) D(g(x_1) x_1)$							
4. $P(x_1) L(1 g(x_1))$							(every non-prime x
5. $P(x_1) L(g(x_1) x_1)$							has a divisor $g(x)$
6. $L(1, a)$							s.t. $1 < g(x) < x$)
7. $\neg P(x_1) \neg D(x_1 a)$							
8. $\neg L(1 x_1) \neg L(x_1 a) P(f(x_1))$							(a is the least counter-
9. $\neg L(1 x_1) \neg L(x_1 a) D(f(x_1) x_1)$							example to the theorem)

THEOREM PROVING

Proof (a)

10. $\neg P(a)$	from 1, 7
11. $L(g(a) a)$	from 10, 5
12. $D(f(g(a)) g(a)) \neg L(1, g(a))$	from 9, 11
13. $L(1, g(a))$	from 4, 10
14. $D(g(a) a)$	from 3, 10
15. $D(f(g(a)) g(a))$	from 13, 12
16. $\neg D(x_1 g(a)) D(x_1 a)$	from 2, 14
17. $\neg L(1 g(a)) P(f(g(a)))$	from 8, 11
18. $D(f(g(a)) a)$	from 16, 15
19. $\neg P(f(g(a)))$	from 18, 7
20. $P(f(g(a)))$	from 13, 17
21. NIL	from 20, 19

Proof (b)

10. $P(a) D(f(g(a)) g(a)) \neg L(1 g(a))$	from 9, 5
11. $P(a) P(f(g(a))) \neg L(1 g(a))$	from 8, 5
12. $D(f(g(a)) g(a)) P(a)$	from 4, 10
13. $D(x_3 x_1) \neg D(x_3 g(x_1)) P(x_1)$	from 3, 2
14. $P(f(g(a))) P(a)$	from 4, 11
15. $D(f(g(a)) a) P(a)$	from 13, 12
16. $P(a) \neg D(f(g(a)) a)$	from 7, 14
17. $P(a)$	from 15, 16
18. $\neg P(a)$	from 7, 1
19. NIL	from 17, 18.

Example 8

'There exist infinitely many primes.'

Several proofs of this theorem have been obtained with different strategies (without u.p.) and sets of 'starting axioms'. In most cases the previous Theorem (example 7) and some properties of the factorial function were assumed at the start. Proof (a) below was obtained starting with the eight 'starred' axioms using $M_1 = \{+P, +L, +D\}$. Proof (b) was obtained using $M_1 = \{-P, +L, +D\}$ and starting with either the eight starred axioms or all twelve axioms listed.

Level		1	2	3	4	5	6	7	Total Generated
(a)	8 axioms	16	24	38	55	—	—	—	112 clauses
(b)	8 axioms	10	14	19	23	30	37	—	78 clauses
(b)	12 axioms	15	20	30	39	49	60	—	114 clauses

Interpret $L(x x)$ as ' $x < x$ ', $D(xy)$ as ' x divides y ', $P(x)$ as ' x is a prime', $f(x)$ as ' $x! + 1$ '.

- 1*. $\neg L(x x)$
- 2*. $\neg L(x y) \neg L(y x)$
3. $D(x x)$

4. $\neg D(x y) \neg D(y z) D(x z)$
 5. $\neg D(x y) \neg L(y x)$
 - 6*. $L(y x) \neg D(x f(y))$
 - 7*. $L(x f(x))$
 8. $P(a)$
 - 9*. $\neg P(x) \neg L(a x) L(f(a) x)$
 - 10*. $P(x) D(h(x) x)$
 - 11*. $P(x) P(h(x))$
 - 12*. $P(x) L(h(x) x)$
- Proof (a)*
13. $P(x_3) L(f(a) h(x_3)) \neg L(a h(x_3))$
 14. $P(f(x_1)) L(x_1 h(f(x_1)))$
 15. $L(f(a) h(f(a))) P(f(a))$
 16. $P(x_2) \neg L(x_2 h(x_2))$
 17. $P(f(a))$
 18. $L(f(a) f(a)) \neg P(f(a))$
 19. $L(f(a) f(a))$
 20. NIL
- Proof (b)*
13. $L(f(a) f(a)) \neg P(f(a))$
 14. $L(f(a) f(a)) D(h(f(a)) f(a))$
 15. $L(f(a) f(a)) L(h(f(a)) f(a))$
 16. $L(f(a) f(a)) L(a h(f(a)))$
 17. $L(h(f(a)) f(a))$
 18. $L(f(a) f(a)) \neg P(h(f(a)))$
 $\quad L(f(a) h(f(a)))$
 19. $\neg L(f(a) h(f(a)))$
 20. $\neg P(f(a))$
 21. $\neg P(h(f(a))) L(f(a) f(a))$
 22. $P(h(f(a)))$
 23. $L(f(a) f(a))$
 24. NIL

(negation of theorem)

(previous theorem)

from 9, 11
 from 6, 10
 from 13, 14
 from 2, 12
 from 15, 16
 from 7, 9
 from 17, 18
 from 19, 1

from 7, 9
 from 10, 13
 from 12, 13
 from 6, 14
 from 1, 15
 from 9, 16
 from 2, 17
 from 1, 13
 from 18, 19
 from 11, 20
 from 21, 22
 from 23, 1

REFERENCES

- Krom, M.R. (1967), The decision problem for a class of first-order formulas in which all disjunctions are binary. *Zeitschr. f. math. Logik u. Grundlagen d. Math.*, **13**, 15-20.
- Luckham, D.C. (1967), The resolution principle in theorem-proving. *Machine Intelligence I*, pp. 47-61 (eds Collins, N. L. and Michie, D.) Edinburgh: Oliver & Boyd.
- McCarthy, J. et al. (1962), *Lisp 1.5 Programmers Manual*. Cambridge, Mass: MIT Press.
- Meltzer, B. (1966), Theorem-proving for computers: some results on resolution and renaming. *Comput. J.* **8**, 341-3.
- Norton, L.M. (1966), *ADEPT - a heuristic program for proving theorems of group theory*. Ph.D. Thesis, MIT project MAC-TR-33.

THEOREM PROVING

- Robinson, J.A. (1965), A machine-oriented logic based on the resolution principle. *J. Ass. comput. Mach.*, 12, 23-41.
- (1965a), Automatic deduction with hyper-resolution. *Int. J. comput. Math.*, 1, 227-34.
- Wang, H. (1965), Formalization and automatic theorem-proving. *Proceedings of IFIP Congress 1965*, 1, 51-8. Washington D.C.: Spartan Books.
- Wos, L.T., Carson, D.F. & Robinson, G. A. (1964), The unit preference strategy in theorem-proving. *AFIPS*, 26, 615-21, Fall, J.C.C. Washington, D.C.: Spartan Books.
- 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, 536-41.

Automatic Theorem Proving with Equality Substitutions and Mathematical Induction

J. L. Darlington

Rheinisch-Westfälisches Institut für
Instrumentelle Mathematik
Bonn

Many theorem-proving programs now in existence employ a matching algorithm that determines whether two or more literals or atomic predicate expressions, belonging to formulae in conjunctive normal form, have common substitution instances (*see* Guard *et al.*, 1967, pp. 16–18). According to this algorithm, two literals beginning with the same n -place predicate P will match if the remainders of the two formulae, which may consist of variables, constants and functional expressions of these, are either identical or can be made so under some set of substitutions for the variables. The output of this algorithm is the answer 'match' or 'no match' and, in the event of a successful match, a 'sublist', or set of substitutions that will make the two literals equal. If L_1 matches L_2 , and if L_1 and $\bar{L}_2$ (the negation of L_2) belong respectively to two formulae F_1 and F_2 in conjunctive normal form, then a 'resolvent' (*see* Robinson, 1965) of F_1 and F_2 may be generated by 'cutting' L_1 and L_2 from F_1 and F_2 , and combining the remainders into a single formula

$$F_3 = F_1' \vee F_2'$$

where F_1' results from making the substitutions on 'sublist' throughout $F_1 - L_1$, and F_2' is generated analogously from $F_2 - L_2$. Let us take, for example, the Euclidean theorem 'if x is a prime that divides $y.z$ then x divides y or z ', write it in conjunctive normal form as

$$(i) \quad \bar{P}(x) \vee \bar{D}(x, y.z) \vee D(x, y) \vee D(x, z)$$

and resolve it against the atomic formula

$$(ii) \quad \bar{D}(a, b) \quad (a \text{ is not a divisor of } b).$$

Two obvious consequences or resolvents of (i) and (ii) are

$$(iii) \quad \bar{P}(a) \vee \bar{D}(a, b.z) \vee D(a, z)$$

$$(iv) \quad \bar{P}(a) \vee \bar{D}(a, y.b) \vee D(a, y)$$

THEOREM PROVING

obtained by matching (ii) against the third and fourth literals respectively of (i). A third resolvent

$$(v) \quad \bar{P}(a) \vee \bar{D}(a, b.b)$$

may be generated by matching (ii) against the third literal of either (iii) or (iv), or directly against the third and fourth literals of (i). The 'sublists' generated by these three matchings are $(x=a, y=b)$, $(x=a, z=b)$ and $(x=a, y=b, z=b)$.

The matching process just described is defined for first-order functional logic, where the variables range over individuals. We next describe an extension of matching to second-order logic, where the variables may take predicates as values. The need for such an extension is obvious if we wish to formulate axioms such as B1 and B7 in the following system B of Hilbert-Bernays (1934) for order theory, cited by Hasenjaeger (1950).

- B1 $(x=y) \rightarrow (f(x) \rightarrow f(y))$
- B2 $-(x < x)$
- B3 $(x < y) \ \& \ (y < z) \rightarrow (x < z)$
- B4 $x < Sx$
- B5 $(x < y) \rightarrow (Sx=y \vee Sx < y)$
- B6 $0=0$
- B7 $f(0) \ \& \ (f(x) \rightarrow f(Sx)) \rightarrow f(y).$

The function 'little f of x ' used in B1 and B7 is a second-order function of one argument, whose introduction into resolution-type theorem proving may be explained as follows. There is a well-understood sense in which a predicate expression such as $.Ixx=e$ (which might be interpreted as ' x -inverse times x equals e ') is a function of the free variables occurring in it. Thus we may write

$$f(x) = .Ixx = e.$$

Logicians since the days of Frege have referred to such entities as 'propositional functions', since they become 'propositions' upon quantification or instantiation of their free variables. Universally quantifying x in the above formula yields the general proposition 'for any x , I of x times x equals e ', whilst instantiation of x by a yields the specific proposition ' I of a times a equals e '. Propositional functions are distinguished from functions such as Ix or $x^2.y$, which may occur within propositional functions and which do not become propositions when their variables are instantiated. We now generalise the notion of a propositional function, so that a predicate expression is a function not only of its free variables but also of any well-formed formula (*wff*) occurring in it. Thus $.Ixx=e$ may be regarded as a function of $.Ixx$, Ix , x (first occurrence), x (second occurrence) and e . As a further generalisation, a predicate expression may be regarded as a function not only of any *wff*

occurring in it but also of any *wff* that *matches* any *wff* occurring in it, so that we may write not only

$$\begin{aligned} f(.Ixx) &= .Ixx = e \\ f(Ix) &= .Ixx = e \\ f(x) &= .Ixx = e \\ f(e) &= .Ixx = e \end{aligned}$$

but also

$$\begin{aligned} f(.Iyy) &= .Ixx = e \\ f(z) &= .Ixx = e \\ f(Huv) &= .Ixx = e \end{aligned}$$

etc. Note that 'little *f*' is a function of one argument, though that one argument may be quite a complicated function of several variables and constants.

Let us now rewrite axiom B1 of Hilbert-Bernays in conjunctive normal form,

$$(vi) \quad x \neq y \vee f(x) \vee f(y)$$

and resolve it against

$$(vii) \quad .ez = z$$

which is a statement of left identity for a group, written with a *z* so that (vi) and (vii) will have no variables in common, a condition required by the matching algorithm. Matching (vii) against the first literal of (vi) in the conventional way yields

$$(viii) \quad f(.ez) \vee f(z)$$

and a second formula

$$(ix) \quad f(z) \vee f(.ez)$$

if equality is taken to be symmetrical, but using the new '*f*-matching' technique, (vii) may be matched also against the second literal *f*(*x*) of (vi), as follows. Scanning from left to right, the *x* of *f*(*x*) matches in turn *.ez*, *e*, *z* and *z*, generating in turn the sublists

$$\begin{aligned} (x = .ez, f(q) = q = z) \\ (x = e, f(q) = .qz = z) \\ (x = z, f(q) = .eq = z) \\ (x = z, f(q) = .ez = q) \end{aligned}$$

which are interpreted as follows. Any equality whose left-hand side is not *f*(*q*) represents a substitution that will make the argument of *f* equal to a well-formed piece of the literal being matched, in this case (vii). (A sublist may of course contain more than one such equality, in which case all the substitutions must be made in order, for the argument of *f* to equal a well-formed

THEOREM PROVING

piece of the literal.) The last equality in each sublist is a pseudo-formula in which the q merely marks the place where a well-formed piece of the literal has been matched by the argument of f . These f -equalities generate specific equalities upon replacement of q by actual wff's, such as $a, x, P(x,y)$, etc. The four sublists are then applied in turn to

$$(vi') \quad x \neq y \vee f(y)$$

which results from cutting $f(x)$ from (vi), to generate the four resolvents

$$(x) \quad \begin{aligned} &.ez \neq y \vee y = z \\ &e \neq y \vee .yz = z \\ &z \neq y \vee .ey = z \\ &z \neq y \vee .ez = y. \end{aligned}$$

The reader may verify that these clauses are intuitively valid consequences of (vii), obtainable by letting y stand in turn for the various well-formed bits of (vii).

The resolvents (x) are 'normal' clauses, in the sense that they do not contain f . In conjunction for example with

$$(xi) \quad .Ixx = e$$

which is a statement of left inverse in a group, they yield the following unit clauses (assuming equality to be symmetrical):

$$(xii) \quad \begin{aligned} &..Ixxz = z, .ee = .Ixx, .e.Ixx = e, \\ &.e.Ixx = e, .ee = .Ixx \end{aligned}$$

of which the latter two are redundant. The set (xii), generated from (vi), (vii) and (xi) via (x), may also be generated from these same three clauses in a second way:

(vi) and (xi) yield

$$\begin{aligned} (xiii) \quad &f(.Ixx) \vee f(e) \\ (xiv) \quad &f(e) \vee f(.Ixx) \end{aligned}$$

assuming equality to be symmetrical. The first literal of (xiii) matches (vii) in two ways, generating the sublists

$$\begin{aligned} (z = .Ixx, f(q) = .eq = z) \\ (z = .Ixx, f(q) = .ez = q) \end{aligned}$$

while the first literal of (xiv) matches (vii) in three ways, generating the sublists

$$\begin{aligned} (f(q) = .qz = z) \\ (z = e, f(q) = .eq = z) \\ (z = e, f(q) = .ez = q). \end{aligned}$$

The first set of sublists is applied to $f(e)$, in the way described earlier, generating

$$.ee = .Ixx, .e.Ixx = e$$

while the second set of sublists is applied to $f(.Ixx)$, generating

$$..Ixxz = z, .e.Ixx = e, .ee = .Ixx$$

which together amount to the set (xii).

Formula (vi) is thus seen to be a *general* axiom for equality substitutions: it enables one to substitute equals for equals no matter where they may occur in a formula, and in fact does the job of an infinite number of particular equality substitution axioms, such as

$$\begin{aligned} x \neq y \vee \bar{F}_1(x) \vee F_1(y) \\ x \neq y \vee \bar{F}_2(x,z) \vee F_2(y,z) \\ x \neq y \vee \bar{F}_2(z,x) \vee F_2(z,y) \\ \dots \dots \dots \\ x \neq y \vee \bar{F}_1(Sx) \vee F_1(Sy) \\ x \neq y \vee \bar{F}_1(SSx) \vee F_1(SSy) \\ \dots \dots \dots \\ x \neq y \vee \bar{F}_2(Sx,z) \vee F_2(Sy,z) \\ x \neq y \vee \bar{F}_2(SSx,z) \vee F_2(SSy,z) \\ \dots \dots \dots \end{aligned}$$

etc.

As a further proof of the power of axiom (vi), we next show how it may be used to generate the reflexive, symmetric and transitive properties of the equality relation. To generate reflexivity, one must start with a premise such as (vii), $.ez = z$. This clause, together with (viii), $f(.ez) \vee f(z)$, which is a direct consequence of (vi) and (vii), yields

$$(xv) \quad z = z.$$

This deduction is intuitively valid, since (viii) says in effect that any occurrence of $.ez$ may be replaced by z ; and making this change in (vii) yields (xv). What the matching algorithm in fact does is to match the first literal, $f(.ez)$, of (viii) against $.ex = x$, which is merely a transliterated form of (vii), generating the sublist

$$(x = z, f(q) = q = x)$$

which, applied to $f(z)$, yields (xv). Symmetry of equality follows easily from

THEOREM PROVING

(vi) and (xv): the second literal of (vi), $f(x)$, matches $z=z$ in two ways, generating the sublists

$$\begin{aligned}(x=z, f(q)=q=z) \\ (x=z, f(q)=z=q)\end{aligned}$$

the first of which, when applied to $x \neq y \vee f(y)$, yields

$$(xvi) \quad z \neq y \vee y = z.$$

And transitivity of equality follows easily from (vi) and (xvi), since an immediate resolvent of these two clauses is

$$y \neq x \vee f(x) \vee f(y)$$

which, together with any pair of unit clauses

$$a=b, b=c$$

will generate

$$a=c$$

via the intermediate clause

$$f(b) \vee f(a).$$

We next show how the f -matching technique may be used to formulate a principle of mathematical induction, such as axiom B7 of Hilbert-Bernays. What is required is a principle that will enable one to deduce a unit clause of the form $f(x)$ from two clauses of the form $f(0)$ and $f(x) \vee f(Sx)$, where the f has of course the same interpretation throughout. The following pair of clauses will do the trick:

$$\begin{aligned}(xvii) \quad & f(0) \vee f(gx) \vee f(x) \\ (xviii) \quad & f(0) \vee f(Sgx) \vee f(x)\end{aligned}$$

where gx is a Skolem function standing for an existentially quantified variable and Sx stands for 'successor of x '. These two axioms will enable one to deduce, for example, $F(x, b)$ from $F(0, b)$ and $(\bar{F}x, b) \vee F(Sx, b)$, via the following argument:

$$\begin{aligned}(1) & f(0) \vee f(gx) \vee f(x) \\ (2) & f(0) \vee f(Sgx) \vee f(x) \\ (3) & F(0, b) \\ (4) & \bar{F}(x, b) \vee F(Sx, b) \\ (5) & F(gx, b) \vee F(x, b) & (1) \text{ \& } (3) \\ (6) & \bar{F}(Sgx, b) \vee F(x, b) & (2) \text{ \& } (3) \\ (7) & F(Sgx, b) \vee F(x, b) & (4) \text{ \& } (5) \\ (8) & F(x, b) & (6) \text{ \& } (7)\end{aligned}$$

The f -matching technique is employed in steps (5) & (6), where in each case $F(0,b)$ matches $f(0)$, generating the sublist

$$f(q) = F(q,b).$$

$F(x,b)$ could also be proved by *reductio ad absurdum* from $F(a,b)$, as follows:

- (1) $f(0) \vee f(gx) \vee f(x)$
- (2) $f(0) \vee f(Sgx) \vee f(x)$
- (3) $F(0,b)$
- (4) $F(x,b) \vee F(Sx,b)$
- (5) $F(a,b)$
- (6) $F(gx,b) \vee F(x,b)$ (1) & (3)
- (7) $F(Sgx,b) \vee F(x,b)$ (2) & (3)
- (8) $F(ga,b)$ (5) & (6)
- (9) $F(Sga,b)$ (5) & (7)
- (10) $F(Sga,b)$ (4) & (8)
- (11) contradiction (9) & (10)

Formulae (xvii) and (xviii) taken together amount to a *general* axiom for mathematical induction, which in fact does the job of an infinite number of particular mathematical induction axioms, such as:

$$\begin{aligned}
 &F_1(0) \vee F_1(gx) \vee F_1(x) \\
 &F_1(0) \vee F_1(Sgx) \vee F_1(x) \\
 &F_1(S0) \vee F_1(Sgx) \vee F_1(Sx) \\
 &F_1(S0) \vee F_1(SSgx) \vee F_1(Sx) \\
 &\dots\dots\dots \\
 &F_2(0,y) \vee F_2(gx,y) \vee F_2(x,y) \\
 &F_2(0,y) \vee F_2(Sgx,y) \vee F_2(x,y) \\
 &F_2(S0,y) \vee F_2(Sgx,y) \vee F_2(Sx,y) \\
 &F_2(S0,y) \vee F_2(SSgx,y) \vee F_2(Sx,y) \\
 &\dots\dots\dots \\
 &F_2(y,0) \vee F_2(y,gx) \vee F_2(y,x) \\
 &F_2(y,0) \vee F_2(y,Sgx) \vee F_2(y,x) \\
 &F_2(y,S0) \vee F_2(y,Sgx) \vee F_2(y,Sx) \\
 &F_2(y,S0) \vee F_2(y,SSgx) \vee F_2(y,Sx) \\
 &\dots\dots\dots
 \end{aligned}$$

etc. In a similar way, the two clauses

- (xix) $f(0) \vee f(gx) \vee f(x)$
- (xx) $f(0) \vee -(x < gx) \vee f(x) \vee f(y)$

formulate a general principle of strong induction that does the job of an infinite number of specific axioms.

THEOREM PROVING

In order to test the usefulness of the new f -matching technique, the author incorporated it into a COMIT theorem-proving program, written at the Rheinisch-Westfälisches Institut für Instrumentelle Mathematik in Bonn and currently running on the Institute's IBM 7090. An earlier version of this program was coded in POP-1 and run on the University of Edinburgh Experimental Programming Unit's Elliott 4120 during the summer of 1966. The results obtained so far have been quite encouraging, in that the new technique has enabled the program to produce shorter and more elegant proofs of various theorems than those previously obtained by conventional resolution-type methods. A good example of a proof requiring both equality substitutions and mathematical induction is the proof by Hasenjaeger (1950) that axiom B5 in the afore-mentioned system B of Hilbert-Bernays is deducible from the others if axiom B6, $0=0$, is replaced by the general axiom of reflexivity of equality,

$$B6' \quad x=x.$$

Using two special definitions:

$$K(y) = df (x < y) \rightarrow (Sx = y \vee Sx < y)$$

$$L(x, y) = df \neg (x < Sy) \vee x = y \vee x < y$$

the following four cases are proved, which together amount to a proof of the dependence of B5:

$$(1) K(0)$$

$$(2) L(0, x)$$

$$(3) K(y) \rightarrow (L(x, y) \rightarrow L(Sx, y))$$

$$(4) K(x) \rightarrow K(Sx)$$

(2) and (3) entail $K(y) \rightarrow L(x, y)$, a result which is useful in proving (4), and

(1) and (4) entail $K(x)$, whence B5 follows directly. The four cases were proved by the program using *reductio ad absurdum*, as follows:

Proof of Hasenjaeger 'Beitrag'

Case 1

$$(1) x \neq y \vee f(x) \vee f(y)$$

B1

$$(2) \neg (x < x)$$

B2

$$(3) \neg (x < y) \vee \neg (y < z) \vee x < z$$

B3

$$(4) x < Sx$$

B4

$$(6) x = x$$

B6'

$$(7) f(0) \vee f(gx) \vee f(x)$$

B7

$$(8) f(0) \vee f(Sgx) \vee f(x)$$

$$(9) a < 0$$

$$(10) Sa \neq 0$$

assumptions for *reductio*

$$(11) \neg (Sa < 0)$$

$$(12) a < gx \vee a < x$$

(7) & (9)

$$(13) \neg (a < Sgx) \vee a < x$$

(8) & (9)

$$(14) a < ga$$

(2) & (12)

- | | |
|-----------------------------|-------------|
| (15) $-(a < Sga)$ | (2) & (13) |
| (16) $-(ga < x) \vee a < x$ | (3) & (14) |
| (17) $a < Sga$ | (4) & (16) |
| (18) contradiction | (15) & (17) |

Case 2

- | | |
|--|--------------------------------|
| (1) $-(8)$ | as in Case 1 |
| (9) $x \leq y \vee x \neq y$ | |
| (10) $x \leq y \vee -(x < y)$ | definition of $x \leq y$ |
| (11) $-(x \leq y) \vee x = y \vee x < y$ | |
| (12) $-(0 \leq a)$ | assumption for <i>reductio</i> |
| (13) $x \leq x$ | (6) & (9) |
| (14) $0 \leq gx \vee 0 \leq x$ | (7) & (13) |
| (15) $-(0 \leq Sgx) \vee 0 \leq x$ | (8) & (13) |
| (16) $0 \leq ga$ | (12) & (14) |
| (17) $-(0 \leq Sga)$ | (12) & (15) |
| (18) $-(0 < Sga)$ | (10) & (17) |
| (19) $-(0 < S0) \vee 0 < Sgga$ | (7) & (18) |
| (20) $-(0 < S0) \vee -(0 < SSgga)$ | (8) & (18) |
| (21) $0 < Sgga$ | (4) & (19) |
| (22) $-(0 < SSgga)$ | (4) & (20) |
| (23) $-(0 < x) \vee -(x < SSgga)$ | (3) & (22) |
| (24) $-(Sgga < SSgga)$ | (21) & (23) |
| (25) contradiction | (4) & (24) |

Case 3

- | | |
|--|---------------------------------|
| (1) $-(8)$ | as in Case 1 |
| (9) $-(x < b) \vee Sx = b \vee Sx < b$ | |
| (10) $-(a < Sb) \vee a = b \vee a < b$ | |
| (11) $Sa < Sb$ | assumptions for <i>reductio</i> |
| (12) $Sa \neq b$ | |
| (13) $-(Sa < b)$ | |
| (14) $a \neq x \vee Sx < Sb$ | (1) & (11) |
| (15) $-(x < Sa) \vee x < Sb$ | (3) & (11) |
| (16) $-(a < b) \vee Sa = b$ | (9) & (13) |
| (17) $a < Sb$ | (4) & (15) |
| (18) $a \neq b$ | (2) & (14) |
| (19) $-(a < b)$ | (12) & (16) |
| (20) $-(a < Sb) \vee a < b$ | (10) & (18) |
| (21) $a < b$ | (17) & (20) |
| (22) contradiction | (19) & (21) |

Case 4

- | | |
|--|--------------|
| (1) $-(8)$ | as in Case 1 |
| (9) $-(x < b) \vee Sx = b \vee Sx < b$ | |
| (10) $-(x < Sb) \vee x = b \vee x < b$ | |

THEOREM PROVING

(11) $a < Sb$	assumptions for <i>reductio</i>
(12) $Sa \neq Sb$	
(13) $\neg (Sa < Sb)$	
(14) $x \neq b \vee Sa \neq Sx$	(1) & (12)
(15) $x \neq b \vee \neg (Sa < Sx)$	(1) & (13)
(16) $\neg (Sa < x) \vee \neg (x < Sb)$	(3) & (13)
(17) $Sa \neq b$	(4) & (15)
(18) $\neg (Sa < b)$	(4) & (16)
(19) $a \neq b$	(6) & (14)
(20) $\neg (a < b) \vee Sa = b$	(9) & (18)
(21) $\neg (a < Sb) \vee a < b$	(10) & (19)
(22) $a < b$	(11) & (21)
(23) $\neg (a < b)$	(17) & (20)
(24) contradiction	(22) & (23)

The proofs of the above cases took 27, 205, 150 and 182 seconds, respectively, and there were 25, 89, 87 and 77 clauses left in memory at the time the proof was obtained. The unit preference, set of support and literal bound strategies of Wos *et al.* (1964, 1964a, 1965) were employed, in addition to a 'non-unit deletion strategy' that eliminates every non-input non-unit clause that appears in the i -th generation, after the $i+1$ -th generation. The machine proofs by and large gave no unexpected results save in Case 2 which the program proved without axiom B1 even though B1 was available. The original paper-and-pencil proof of Case 2 used one application of B1 and one of B7, while the machine proof used instead two applications of B7; but when the proof was attempted without the non-unit deletion strategy the program reproduced the proof using B1. For purposes of comparison, the latter proof is given below.

Case 2 (using B1)

(1) $\neg (18)$	as in first proof
(19) $0 = ga \vee 0 < ga$	(11) & (16)
(20) $x \neq ga \vee \neg (0 < Sx)$	(1) & (18)
(21) $\neg (0 < x) \vee \neg (x < Sga)$	(3) & (18)
(22) $0 \neq ga$	(4) & (20)
(23) $\neg (0 < ga)$	(4) & (21)
(24) $0 < ga$	(19) & (22)
(25) contradiction	(23) & (24)

The two different ways of proving Case 2 arise from the fact that $\neg (0 < Sga)$ in conjunction with

B1 $x \neq y \vee f(x) \vee f(y)$
and
B4 $x < Sx$
yields
 $0 \neq ga$

while the same unit clause, $-(0 < Sga)$, in conjunction with

$$\begin{array}{l} \text{B7} \quad f(0) \vee f(gx) \vee f(x) \\ \quad \quad f(0) \vee f(Sgx) \vee f(x) \end{array}$$

and B4 yields

$$0 < Sgga, -(0 < SSgga).$$

The first proof employs an f -match between $-(0 < Sga)$ and the third literal of B1 generating the sublist

$$(y = ga, f(q) = 0 < Sq)$$

while the second proof employs an f -match between $-(0 < Sga)$ and the third literals of B7, generating the similar sublist

$$(x = ga, f(q) = 0 < Sq)$$

and either inference leads to a proof.

Though the machine was able to prove the four cases, it would be incorrect to say that this constitutes a completely mechanical proof of the dependence of B5, since the division of the proof into the four cases was done by hand. Furthermore, a key role was played by the definitions of $K(y)$ and $L(x, y)$, and also by the combination of $x = y$ and $x < y$ into a single predicate $x \leq y$. The present author has not seen any discussion of the importance of definitions of this sort for automatic theorem proving, much less a description of any program or algorithm that can mechanically construct such useful definitions without generating at the same time a great mass of trivial ones.

The program was next given the job of proving four standard examples taken from group theory. In example G1, the program was given $Gxyx = y$ (existence of left solution) and $xHxy = y$ (existence of right solution) in addition to axiom (vi), the general equality substitution axiom. It then proceeded to generate $xHyy = x$ (existence of right identity), $Gxxy = y$ (existence of left identity) and $Gxx = Hyy$ (left identity equals right identity), all in the same run. This proof differs from previous machine proofs of these same propositions in not employing *reductio ad absurdum* and in proving them in the same run rather than separately. The same holds true of example G2, in which the program was given $ex = x$ (left identity) and $Ixx = e$ (left inverse) in addition to axiom (vi), and proceeded to generate $xe = x$ (right identity), $xIx = e$ (right inverse) and $IIX = x$ (double inverse of x equals x), among other clauses. In example G3, the program was given $ex = x$, $xx = e$ (the square of every element is the identity) and $ab \neq ba$ (assumption that the group is not commutative) in addition to axiom (vi), and generated a contradiction. And in example G4, the program proved by *reductio* the theorem, cited by Wos *et al.* (1964a) and by Meltzer in his contribution to this volume, that a non-empty subset that contains xIy whenever it contains x and y is closed under multiplication. The original statement of this problem

THEOREM PROVING

contained 24 clauses, a large number of which were merely special equality axioms which our program was able to replace by the single axiom (vi). The program also dispensed with $I!x=x$, a lemma used in the original version, as well as two four-literal clauses formulating associativity of group multiplication, and consequently required only seven clauses to formulate the same example. The program is able to dispense with special axioms for left and right associativity, since it handles these instead by a rule which automatically associates everything to the right as far as possible and takes this to be the standard form for products. Thus, if $..!xxy=y$ is generated it will automatically be stored in the form $!x.xy=y$. The program, however, is capable of associating to the left during a resolution, so that $!x.xy=y$ in conjunction with axiom (vi) will generate

$$.!xx \neq z \vee .zy=y$$

and other clauses depending on the left-associated form, as well as

$$.!x \neq z \vee .z.xy=y$$

and other clauses depending on the right-associated form. This feature is particularly useful in step (13) of example G3, where $.a.bb \neq a$ is generated in one step from $.a.ba \neq b$ and $.xy \neq z \vee .xz=y$ by employing first a left and then a right association.

The program employs standard forms to reduce the variety of clauses in two further ways. Alphabetic variance is diminished considerably by a rule which writes the variables in each clause in the order x,y,z,u,v,w , and equalities are written so that the left side contains at least as many terms as the right side.

Examples G1-G4 are given below.

Example G1

(1) $.Gxyx=y$	existence of left solution
(2) $.xHxy=y$	existence of right solution
(3) $x \neq y \vee f(x) \vee f(y)$	eq sub axiom
(4) $f(.Gxyx) \vee f(y)$	(1) & (3)
(5) $f(x) \vee f(.Gyxy)$	(1) & (3)
(6) $f(.xHxy) \vee f(y)$	(2) & (3)
(7) $f(x) \vee f(.yHyx)$	(2) & (3)
(8) $.Gxy.xHyx=z$	(2) & (5), r assoc
(9) $.Gxy.zHxz=y$	(1) & (7)
(10) $.xHyy=x$ (right identity)	(4) & (9), l assoc
(11) $.Gxxy=y$ (left identity)	(6) & (8)
(12) $f(.xHyy) \vee f(x)$	(3) & (10)
(13) $Gxx=Hyy$ (left identity equals right identity)	(11) & (12)

Example G2

- | | |
|---|--------------------|
| (1) $.ex = x$ | left identity |
| (2) $.Ixx = e$ | left inverse |
| (3) $x \neq y \vee f(x) \vee f(y)$ | eq sub axiom |
| (4) $f(.Ixx) \vee f(e)$ | (2) & (3) |
| (5) $f(e) \vee f(.Ixx)$ | (2) & (3) |
| (6) $.Ix.xy = y$ | (1) & (5), r assoc |
| (7) $.IIxe = x$ | (4) & (6) |
| (8) $f(.Ix.xy) \vee f(y)$ | (3) & (6) |
| (9) $.IIxy = .xy$ | (6) & (8) |
| (10) $f(.IIxe) \vee f(x)$ | (3) & (7) |
| (11) $.xe = x$ (right identity) | (9) & (10) |
| (12) $.x.Ixy = y$ | (8) & (9) |
| (13) $.xIx = e$ (right inverse) | (4) & (9) |
| (14) $IIx = x$ (double inverse of x
equals x) | (10) & (11) |

Example G3

- | | |
|------------------------------------|--|
| (1) $.ex = x$ | left identity |
| (2) $.xx = e$ | square of every element
is the identity |
| (3) $x \neq y \vee f(x) \vee f(y)$ | eq sub axiom |
| (4) $.ab \neq .ba$ | assumption for <i>reductio</i> :
group is not commutative |
| (5) $f(.xx) \vee f(e)$ | (2) & (3) |
| (6) $f(e) \vee f(.xx)$ | (2) & (3) |
| (7) $.x.xy = y$ | (1) & (6), r assoc |
| (8) $.xe = x$ | (5) & (7) |
| (9) $.xy \neq z \vee .xz = y$ | (3) & (7) |
| (10) $.a.ba \neq b$ | (4) & (9) |
| (11) $x \neq e \vee .yx = y$ | (3) & (8) |
| (12) $.x.yy = x$ | (2) & (11) |
| (13) $.a.bb \neq a$ | (9) & (10), l & r assoc |
| (14) contradiction | (12) & (13) |

Example G4

- | | |
|---|---|
| (1) $.ex = x$ | left identity |
| (2) $.Ixx = e$ | left inverse |
| (3) $x \neq y \vee f(x) \vee f(y)$ | eq sub axiom |
| (4) $\bar{0}x \vee \bar{0}y \vee 0.xIy$ | special assumption:
if x and y belong to
subset 0 , then x times
y -inverse belongs to 0 |

THEOREM PROVING

(5) $0a$	assumptions for <i>reductio</i> :
(6) $0b$	subset 0 is not closed
(7) $\bar{0}.ab$	under multiplication
(8) $f(.ex) \vee f(x)$	(1) & (3)
(9) $f(.Ixx) \vee f(e)$	(2) & (3)
(10) $f(e) \vee f(.Ixx)$	(2) & (3)
(11) $.Ix.xy=y$	(1) & (10), r assoc
(12) $\bar{0}x \vee 0.aIx$	(4) & (5)
(13) $\bar{0}x \vee 0.bIx$	(4) & (6)
(14) $\bar{0}x \vee 0.xIb$	(4) & (6)
(15) $x \neq b \vee \bar{0}.ax$	(3) & (7)
(16) $0.bIb$	(6) & (13)
(17) $.IIxe=x$	(9) & (11)
(18) $f(.Ix.xy) \vee f(y)$	(3) & (11)
(19) $.IIxy=.xy$	(11) & (18)
(20) $\bar{0}.a.IIbe$	(15) & (17)
(21) $.bIb \neq x \vee 0x$	(3) & (16)
(22) $f(.IIxe) \vee f(x)$	(3) & (17)
(23) $.xe=x$	(19) & (22)
(24) $.xIx=e$	(9) & (19)
(25) $.IIbe \neq x \vee \bar{0}.ax$	(3) & (20)
(26) $\bar{0}.a.IIb$	(23) & (25)
(27) $0e$	(21) & (24)
(28) $0.eIb$	(14) & (27)
(29) $\bar{0}Ib$	(12) & (26)
(30) $0Ib$	(8) & (28)
(31) contradiction	(29) & (30)

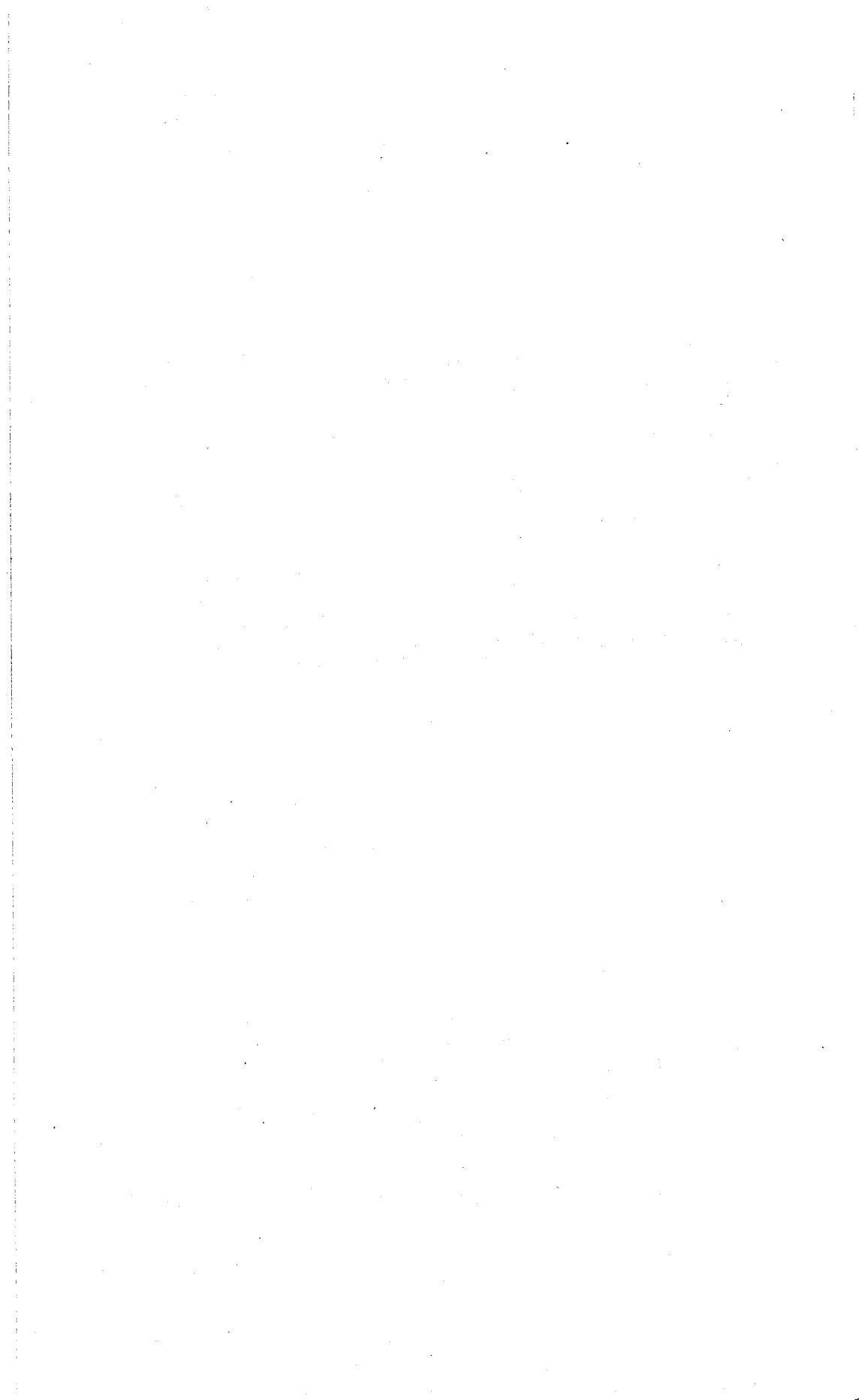
The running times for G1, G2 and G3 were under five minutes apiece, with fewer than 50 clauses retained in each case, mainly because of special deletion heuristics that were employed, such as deleting clauses that contain triple or quadruple inverses, or that contain *.ee*, etc. Even with these special rules, however, and despite the fact that only 121 clauses were retained, the program required over 22 minutes to prove G4. The proofs of G1–G4 are similar in spirit to those obtained by Guard *et al.* (1967), though simpler in the sense that our proofs are produced without going into triple and quadruple inverses and without using a special axiom for associativity.

Other proofs obtained by the program include example q1 of Wang (1965), which requires equality substitutions, and a simpler proof of the number-theory proposition (*see* Darlington 1968) that the square root of a prime is irrational. Our proof of Wang's q1 is superior in terms of formulation to that obtained by Veenker (1967), who used specific equality substitution axioms instead of a general axiom like ours, though Veenker's proof, produced by a machine-language program, is far shorter than ours in terms of running time.

Plans for the future include the production of an on-line version of the program for the IBM 360/50 recently installed at the IIM in Bonn, and the incorporation into it of new techniques, such as those presented by Luckham, Meltzer, and Robinson in their contributions to this volume.

REFERENCES

- Darlington, J.L. (1968), Some theorem-proving strategies based on the resolution principle, *Machine Intelligence* 2, pp. 57-71 (ed. Michie, D.). Edinburgh: Oliver & Boyd.
- Guard, J.R. *et al.* (1967), CRT-aided semi-automated mathematics, AFRCL-67-0167. Princeton, New Jersey: Applied Logic Corporation.
- Hasenjaeger, G. (1950), Ein Beitrag zur Ordnungstheorie, *Arch. math. Logik GrundForsch.* 1, 30-1.
- Hilbert, D. & Bernays, P. (1934), *Grundlagen der Mathematik I*. Berlin: Springer-Verlag.
- Robinson, J.A. (1965), A machine-oriented logic based on the resolution principle, *J. Ass. comput. Mach.*, 12, 23-41.
- Veenker, G. (1967), *Beweisverfahren für den Prädikatenkalkül*. Doctoral dissertation, University of Tübingen.
- Wang, H. (1965), Formalization and automatic theorem-proving, *Proceedings of IFIP Congress 1965*, 1, 51-8. Washington, D.C.: Spartan Books.
- Wos, L.T., Carson, D.F. & Robinson, G.A. (1964), The unit preference strategy in theorem-proving, *AFIPS*, 26, 615-21. Fall J.C.C. Washington, D.C.: Spartan Books.
- (1964a), Some theorem-proving strategies and their implementation, *AMD Tech. Memo No. 72*, Argonne National Laboratory.
- (1965), Efficiency and completeness of the set of support strategy in theorem-proving, *J. Ass. comput. Mach.*, 12, 536-41.



MACHINE LEARNING AND HEURISTIC PROGRAMMING

On Representations of Problems of Reasoning about Actions

Saul Amarel
RCA Laboratories
Princeton, N.J.

1. INTRODUCTION

The purpose of this paper is to clarify some basic issues of choice of representation for problems of reasoning about actions. The general problem of representation is concerned with the relationship between different ways of formulating a problem to a problem solving system and the efficiency with which the system can be expected to find a solution to the problem. An understanding of the relationship between problem formulation and problem solving efficiency is a prerequisite for the design of procedures that can automatically choose the most 'appropriate' representation of a problem (they can find a 'point of view' of the problem that maximally simplifies the process of finding a solution).

Many problems of practical importance are problems of reasoning about actions. In these problems, a course of action has to be found that satisfies a number of specified conditions. A formal definition of this class of problems is given in the next section, in the context of a general conceptual framework for formulating these problems for computers. Everyday examples of reasoning about actions include planning an airplane trip, organizing a dinner party, etc. There are many examples of industrial and military problems in this category, such as scheduling assembly and transportation processes, designing a program for a computer, planning a military operation, etc.

The research presented in this paper was sponsored in part by the Air Force Office of Scientific Research, under Contract Number AF49(638)-1184. Part of this work was done while the author was on a visiting appointment at the Computer Science Department of the Carnegie Institute of Technology, Pittsburgh, Pa. At Carnegie Tech. this research was sponsored by the Advanced Research Projects Agency of the Office of the Secretary of Defense under Contract Number SD-146.

We shall analyze in detail a specific problem of transportation scheduling—the ‘missionaries and cannibals’ problem (which is stated in section 3)—in order to evaluate the effects of alternative formulations of this problem on the expected efficiency of mechanical procedures for solving it, and also in order to examine the processes that come into play when a transition takes place from a given problem formulation into a better one. After the initial verbal formulation of the missionaries and cannibals problem in section 3, the problem undergoes five changes in formulation, each of which increases the ease with which it can be solved. These reformulations are discussed in sections 4 to 11. A summary of the main ideas in the evolution of formulations, and comments on the possibility of mechanizing the transitions between formulations are given in section 12.

2. PROBLEMS OF REASONING ABOUT ACTIONS

A problem of reasoning about actions (Simon, 1966) is given in terms of an initial situation, a terminal situation, a set of feasible actions, and a set of constraints that restrict the applicability of actions; the task of the problem solver is to find the ‘best’ sequence of permissible actions that can transform the initial situation into the terminal situation. In this section, we shall specify a *system of productions*, P , where problems of reasoning about actions can be naturally formulated and solved.

In the system P , a basic description of a situation at one point in time is a listing of the basic features of the situation. The basic features are required for making decisions about actions that can be taken from the situation. We call a situation a *state of nature* (an N -state). The language in which N -states are described is called an *N -state language*. Such a language is defined by specifying the following:

- (i) a non-empty set U_0 called the *basic universe*; this set contains the basic elements of interest in situations (the individuals, the objects, the places);
- (ii) a set of basic predicates defined for elements of U_0 (properties of elements and relations between elements);
- (iii) a set of rules of formation for expressions in the language.

The rules of formation determine whether an N -state language is a linear language, a two-dimensional (graphic) language, or it has some other form. Regardless of the form taken by an expression in an N -state language, such an expression is meant to assert that a given element in U_0 has a certain property or that a given subset of elements in U_0 are related in a specified manner. Thus, an expression in an N -state language has the logical interpretation of a true proposition about a basic feature of the situation. A finite set (possibly empty) of expressions in an N -state language is called a *configuration*. The empty configuration will be written Λ . In the logic interpretation, a (non-empty) configuration is a conjunction of the true assertions made by its component expressions. The set union of two configurations is itself a

configuration. If α and β are configurations, then their union will be written α, β . A *basic description*, s , of an N -state is a configuration from which all true statements about the N -state (that can be expressed in the terms of the N -state language) can be directly obtained or derived. Thus a basic description completely characterizes an N -state. Henceforth we shall refer to an N -state by its basic description.

A derived description of an N -state at one point in time is a listing of compound features of the N -state. Compound features are defined in terms of the basic features, and they are intended to characterize situations in the light of the problem constraints, so that decisions about the legality of proposed actions can be made. We denote by $d(s)$ a derived description that is associated with an N -state s . The language in which derived descriptions are formulated is an extension of the N -state language, and it is called the *extended description language*. Such a language is defined by the following:

- (i) a set U_1 called the *extended universe*, where $U_0 \subset U_1$ (this is not necessarily a proper inclusion); the extension of U_0 contains compound elements of interest (definable in terms of the basic elements in U_0), and possibly new elements (not obtainable from U_0) that are used for building high level descriptions;
- (ii) a set of new predicates defined for elements of U_1 (properties and relations that are required for expressing the constraining conditions of the problem);
- (iii) a set of rules of formation for expressions in the language.

The rules of formation in this language are identical with those of the N -state language. Each expression in the extended description language has the logical interpretation of a proposition about a compound feature in a situation. A derived description $d(s)$ is a set of expressions in the extended description language (it is a configuration in the language). In the logical interpretation, $d(s)$ is a conjunction of the propositions that are specified by its constituent expressions.

The *rules of action* in the system P specify a possible next situation (next in time with respect to a given time scale) as a function of certain features in previous situations. The complexity of a problem about actions is determined by the nature of this dependence. There is a sequential and a local component in such a dependence. The sequential part is concerned with dependencies of the next situation on features of sequences of past situations. We will not be concerned with such dependencies in this paper. The local part is concerned with the amount of local context that is needed to determine a change of a basic feature from one situation to the next.

In the specification of a rule of action, an N -state is given in terms of a *mixed description* s' , which is written as follows:

$$s' = s; d(s), \quad (2.1)$$

where s is the basic description of the N -state, and $d(s)$ is its associated

derived description. Let A be a feasible action and let (A) denote the rule of action that refers to A . A rule of action is given as a transition schema between mixed descriptions of N -states, and it has the following form:

$$(A): s_a; d(s_a) \rightarrow s_b; d(s_b) \quad (2.2)$$

The feasible action A is defined as a transformation from the N -state s_a to the N -state s_b . If A is applied at s_a , then the next N -state will be s_b . The rule (A) specifies the condition under which the application of A at s_a is permissible. This is to be interpreted as follows: 'If $d(s_a)$ and $d(s_b)$ are both satisfied, then the application of A at s_a is permissible.' A derived description $d(s)$ is satisfied if it is true under the logical interpretation. The rule (A) imposes a restriction on the mapping $A: s_a \rightarrow s_b$, i.e. it restricts the domain of the feasible action. Thus, given an N -state s_a for which A is a feasible action, A can be applied at s_a only if the N -state s_b that results from the application of A has certain compound features that are specified in $d(s_b)$.

Let $\{(A)\}$ be the (finite) set of rules of action and let $\{s\}$ be the set of all possible N -states. The set $\{(A)\}$ specifies a relation of *direct attainability* between the elements of $\{s\}$. Given any two states s_x, s_y from $\{s\}$, the N -state s_y is directly attainable from s_x if and only if there exists a permissible action in $\{(A)\}$ that can take s_x to s_y . Let us denote by T the relation of direct attainability.¹ The expression $s_x T s_y$ asserts that the N -state s_x can occur *just earlier than* s_y in a possible evolution of the system. Thus, the relation T represents local time order for the system P .

A *trajectory* from an N -state s_a to an N -state s_b is a finite sequence $s_1, s_2, \dots, s_m$ of N -states such that $s_1 = s_a, s_m = s_b$, and for each $i, 1 < i \leq m$, s_i is directly attainable from s_{i-1} . For any pair of N -states s_a, s_b , we say that s_b is *attainable from* s_a if and only if $s_a = s_b$ or there exists a trajectory from s_a to s_b . We denote the relation of attainability from s_a to s_b by $s_a \Rightarrow s_b$. The notion of a schedule is close to the notion of a trajectory; it is the sequence of actions that are taken in moving over the trajectory.

Now a problem of reasoning about actions can be formulated in the system P as follows: Given

- (i) an N -state language
- (ii) an extended description language
- (iii) a set of rules of action
- (iv) an initial N -state and a terminal N -state,

find the shortest schedule (or the shortest trajectory) from the initial N -state to the terminal N -state (if a schedule exists at all).

The set of all N -states, partly ordered under the relation T , defines a space σ that we call the N -state space. The search for a solution trajectory takes place in this space.

¹ This relation is very close to the relation 'earlier' introduced by Carnap (1958), and denoted T , in his language for space-time topology. In Carnap's case, T represents time order between two world points that are on the same trajectory.

Commonly, the initial formulation of a problem of reasoning about actions is a *verbal formulation*. Given the initial verbal formulation, there are several possible N -state languages and extended description languages that can be used for formulating the problem in the system of productions P . The choice of the universe U_1 and of the features in terms of which situations are described can strongly influence the amount of effort that is needed in order to find a solution in the formulation P . Here is an important decision point where problem solving power is affected by the choice of a problem representation. In addition, strong improvements in problem solving power may result from the discovery and exploitation of regularities in N -state space. The discovery of such regularities is facilitated by appropriate representations of N -state space. We shall illustrate these points by discussing in detail in the following sections a sequence of formulations of an extended version of the Missionary and Cannibals problem.

3. TRANSPORTATION PROBLEMS: INITIAL FORMULATION, F_1 , OF M&C PROBLEMS

Many transportation scheduling problems are problems of reasoning about actions. Such problems can be formulated as follows. Given a set of space points, an initial distribution of objects in these points, and transportation facilities with given capacities; find an optimal sequence of transportations between the space points such that a terminal distribution of objects in these points can be attained without violating a set of given constraints on possible intermediate distribution of objects.

An interesting subclass of these transportation scheduling problems is the class of 'difficult crossing' problems, typified by the 'Missionaries and Cannibals' problem. This problem appears frequently in books on mathematical recreations. It has also received attention in the dynamic programming literature (Bellman and Dreyfus, 1962) and in the literature on computer simulation of cognitive processes. (Simon and Newell, 1961). The following is a verbal formulation of the 'missionaries and cannibals' problem (we call it formulation F_1). Three missionaries and three cannibals seek to cross a river (say from the left bank to the right bank). A boat is available which will hold two people, and which can be navigated by any combination of missionaries and cannibals involving one or two people. If the missionaries on either bank of the river, or 'en route' in the river, are outnumbered at any time by cannibals, the cannibals will indulge in their anthropophagic tendencies and do away with the missionaries. Find the simplest schedule of crossings that will permit all the missionaries and cannibals to cross the river safely.

In a more generalized version of this problem, there are N missionaries and N cannibals (where $N \geq 3$) and the boat has a capacity k (where $k \geq 2$). We call this problem the *M & C problem*. We shall refer to the specific problem that we have formulated above (where $N = 3, k = 2$) as the *elementary M & C problem*.

4. FORMULATION F_2 OF THE M&C PROBLEM IN ELEMENTARY SYSTEMS OF PRODUCTIONS

We shall formulate now the M&C problem in a system of productions of the type described in section 2. We start by specifying a simple but straightforward N -state language.

The universe U_0 of the N -state language contains the following basic elements:

- (i) N individuals $m_1, m_2, \dots, m_N$ that are missionaries and N individuals $c_1, c_2, \dots, c_N$ that are cannibals,
- (ii) an object (a transportation facility)—the boat b_k with a carrying capacity k ,
- (iii) two space points p_L, p_R for the left bank and the right bank of the river respectively.

The basic relations between basic elements in U_0 are as follows:

- (i) at ; this associates an individual or the boat with a space point (example: $at(m_1, p_L)$ asserts that the missionary m_1 is at the left bank),
- (ii) on ; this indicates that an individual is aboard the boat (example: $on(c_1, b_k)$ asserts that the cannibal c_1 is on the boat).

A set of expressions, one for each individual and one for the boat (they specify the positions of all the individuals and of the boat) provides a basic description of a situation, i.e. it characterizes an N -state. Thus, the initial N -state for the M&C problem can be written as follows:

$$s_0 = at(b_k, p_L), at(m_1, p_L), at(m_2, p_L), \dots, at(m_N, p_L), at(c_1, p_L), at(c_2, p_L), \dots, at(c_N, p_L). \quad (4.1)$$

The terminal N -state is attained from (4.1) by substituting p_R for p_L throughout.

The verbal statement of the M&C problem induces the formulation of an extended description language where a non-empty extension of U_0 is introduced together with certain properties and relations for the elements of this extension. The compound elements in the extension of U_0 are defined in terms of notions in the N -state language. These compound elements are the following six subsets of the total set $\{m\}$ of missionaries and the total set $\{c\}$ of cannibals:

- $\{m\}_L = \{x | x \in \{m\}, at(x, p_L)\}$; the subset of missionaries at left,
- $\{m\}_R = \{x | x \in \{m\}, at(x, p_R)\}$; the subset of missionaries at right,
- $\{m\}_b = \{x | x \in \{m\}, on(x, b_k)\}$; the subset of missionaries aboard the boat.

The three remaining compound elements $\{c\}_L, \{c\}_R, \{c\}_b$ are subsets of the total set of cannibals that are defined in a similar manner.

In the M&C problem, the properties of interest for the specification of permissible actions are the sizes of the compound elements that we have just

introduced, i.e. the number of elements in the subsets $\{m\}_L$, $\{m\}_R$, etc. Let M_L , M_R , M_b , C_L , C_R , C_b denote the number of individuals in the sets $\{m\}_L$, $\{m\}_R$, $\dots$, $\{c\}_b$ respectively. These are variables that take values from the finite set of nonnegative integers $J_0^N = \{0, 1, 2, \dots, N\}$. These integers are also elements of the extension of U_0 . They bring with them in the extended description language the arithmetic relations $=$, $>$, $<$, as well as compound relations that are obtainable from them via the logical connectives $\sim$, $\vee$, $\wedge$, and also the arithmetic functions $+$, $-$. A derived description $d(s)$ which is associated with an N -state s is a set of expressions that specify certain arithmetic relations between the variables M_L , M_R , etc. whose values are obtained from s .

The rules of formation that we shall use for description languages are of the type conventionally used in logic; they yield linear expressions. Expressions are concatenated (with separating commas) to form configurations. The basic description given in (4.1) is an example of a configuration in the linear language.

The verbal statement of the M & C problem does not induce a unique choice of a set of feasible actions. We shall consider first a 'reasonable' set of *elementary* actions that are assumed to be feasible and that satisfy the given constraints on boat capacity and on the possible mode of operating the boat. The set of permissible actions is a subset of this set that can be obtained by specifying the appropriate restrictions on the relative number of missionaries and cannibals in the two river banks as well as 'en route'.

$\{(A)'\}_1$: *Elementary feasible actions in Formulation F_2 that are sensitive to boat constraints*. In the following transition schemata, α denotes an arbitrary configuration that completes a basic description of an N -state:

Load boat at left, one individual at a time (LBL)'

For any individual x ,

$(LBL)'$: $\alpha, at(b_k, p_L), at(x, p_L); (M_b + C_b \leq k - 1) \rightarrow \alpha, at(b_k, p_L), on(x, b_k); \Lambda$

Move boat across the river from left to right (MBLR)'

$(MBLR)'$: $\alpha, at(b_k, p_L); (M_b + C_b > 0) \rightarrow \alpha, at(b_k, p_R); \Lambda$

Unload boat at right, one individual at a time (UBR)'

For any individual x ,

$(UBR)'$: $\alpha, at(b_k, p_R), on(x, b_k); \Lambda \rightarrow \alpha, at(b_k, p_R) at(x, p_R); \Lambda$

In addition, we have the three following elementary actions in $\{(A)'\}_1$ 'Load boat at right one individual at a time (LBR)', 'Move boat across the river from right to left ($MBRL$)', and 'Unload boat at left one individual at a time (UBL)'. The definitions of these actions are obtained from the previous definitions by substituting p_L for p_R and p_R for p_L in the corresponding actions. For example, the definition of ($MBRL$)' is as follows:

$(MBRL)'$: $\alpha, at(b_k, p_R); (M_b + C_b > 0) \rightarrow \alpha, at(b_k, p_L); \Lambda$

The six elementary actions that we have just introduced can be used together in certain sequences to form macro-actions for transferring *sets* of individuals from one river bank to the other. A transfer of r individuals from left to right, where $1 \leq r \leq k$; can be effected by a sequence

$$\underbrace{(LBL)', (LBL)', \dots, (LBL)'}_{r \text{ times}} (MBLR)', \underbrace{(UBR)', (UBR)', \dots, (UBR)'}_{r \text{ times}} \quad (4.2)$$

This sequence of actions starts with an empty boat at left and ends with an empty boat at right.

We can view the sequence of elementary actions in (4.2) as a transfer macroaction that is composed of two parts: the first part consists of the initial loading sequence for the boat, or equivalently the unloading sequence for the place that is the origin of the transfer. The second part starts with the river crossing and is followed by an unloading sequence for the boat, or equivalently by the loading sequence for the place that is the destination of the transfer. Since the constraints of the problem are given in terms of the relative sizes of various sets of individuals at points that can be considered as ends of loading (or unloading) sequences, then it is reasonable to attempt the formulation of actions as transitions between such points. We use these considerations in the formulation of a set of feasible compound actions that are only sensitive to boat constraints.

$\{(A)'\}_2$: Compound feasible actions in formulation F_2 that are sensitive to boat constraints,

Load empty boat at left with r individuals, $1 \leq r \leq k$, $'(L'BL)'$.

Here we have a class of transition schemas that can be specified as follows:

For a set of r individuals $x_1, \dots, x_r$, where $1 \leq r \leq k$,

$$(L'BL)': \alpha, at(b_k, p_L), at(x_1, p_L), \dots, at(x_r, p_L); (M_b + C_b = 0) \rightarrow \alpha, at(b_k, p_L), on(x_1, b_k), \dots, on(x_r, b_k); \Lambda$$

In these transitions, r is the number of individuals from the left bank that board the boat for a crossing.

Move boat (loaded with r individuals) across the river from left to right and unload all its passengers at right $(MBLR + U^rBR)'$.

Here also we have a class of transition schemas which is defined as follows:

For a set of r individuals $x_1, \dots, x_r$, $1 \leq r \leq k$,

$$(MBLR + U^rBR)': \alpha[e], at(b_k, p_L), on(x_1, b_k), \dots, on(x_r, b_k); \Lambda \rightarrow \alpha[e], at(b_k, p_R), at(x_1, p_R), \dots, at(x_r, p_R); \Lambda,$$

where $\alpha[e]$ stands for a configuration that is constrained by the condition e , which is as follows: no expression in the form $on(y, b_k)$, for any individual y is included in α . This is a way of saying that, after the crossing, all the r

passengers that have initially boarded the boat in the left bank, have to leave the boat and join the population of the right bank.

In addition to the two compound actions defined above, we have the two following compound actions in $\{(A)\}_2$: 'Load empty boat at right with r individuals, $(L'BR)'$,' and 'Move boat (loaded with r individuals) across the river from right to left and unload all its passengers at left $(MBRL + U'BL)'$ '. The definitions of these compound actions are obtained from the definitions for $(L'BL)'$ and $(MBLR + U'BR)'$ by substituting p_L for p_R and p_R for p_L in the corresponding compound actions.

The compound actions that we have just introduced define the feasible transitions between N -states that are constrained only by the conditions on the transportation facility. Consider now a restriction on these compound actions that provides a set of rules of action where consideration is given to all the constraints of the M&C problem.

$\{(A)\}_2$: First set of rules of action in formulation F_2 .

$(L'BL)$.

For a set of r individuals $x_1, \dots, x_r$, where $1 \leq r \leq k$,

$(L'BL)$: $\alpha, at(b_k, p_L), at(x_1, p_L), \dots, at(x_r, p_L); (M_b + C_b = 0) \rightarrow$
 $\alpha, at(b_k, p_L), on(x_1, b_k), \dots, on(x_r, b_k); ((M_L = 0) \vee (M_L \geq C_L)),$
 $((M_b = 0) \vee (M_b \geq C_b)).$

These compound actions are a subset of the compound actions $(L'BL)'$, where a valid next N -state is such that if any missionaries remain in the left bank then their number is no smaller than the number of cannibals remaining there, and also if any missionaries board the boat, then their number is no smaller than the number of cannibals that have also boarded the boat. Note that if an individual, say a missionary, is aboard the boat and the boat is at p_L , then the individual is not considered as a member of $\{m\}_L$, and therefore he is not counted in M_L .

$(MBLR + U'BR)$.

For any r , where $1 \leq r \leq k$,

$(MBLR + U'BR)$: $\alpha [e], at(b_k, p_L), on(x_1, b_k), \dots, on(x_r, b_k); \Lambda \rightarrow \alpha [e],$
 $at(b_k, p_R), at(x_1, p_R), \dots, at(x_r, p_R),$
 $((M_R = 0) \vee (M_R \geq C_R)).$

Here the restricted configuration $\alpha [e]$ has the same meaning as in $(MBLR + U'BR)'$. The present compound actions are a subset of $(MBLR + U'BR)'$, where a valid next N -state is such that if any missionaries are present in the right bank then their number is no smaller than the number of cannibals there.

In addition to the transitions $(L'BL)$ and $(MBLR + U'BR)$, we also have the two transitions $(L'BR)$ and $(MBRL + U'BL)$, that are obtained from the previous ones by appropriately interchanging the places p_L and p_R throughout the definitions.

With the formulation of the permissible transitions between N -states, it is now possible to specify a procedure for finding a schedule of transfers that would solve the general M&C problem. Each transfer from left to right will be realized by a sequence ($L'BL$), ($MBLR+U'BR$), and each transfer from right to left will be realized by a sequence ($L'BR$), ($MBRL+U'BL$). Essentially, the selection of compound actions for each transfer amounts to finding r -tuples of individuals from a river bank that could be transferred to the opposite bank in such a way that cannibalism can be avoided in the source bank, in the destination bank and in the boat; i.e. the *non-cannibalism conditions*

$$((M_L=0) \vee (M_L \geq C_L)), ((M_b=0) \vee (M_b \geq C_b)), ((M_R=0) \vee (M_R \geq C_R)) \quad (4.3)$$

are all satisfied at the end of each of the two compound actions that make a transfer.

The formulation of compound actions and of problem solving procedures can be simplified *via* the utilization of the following property of our problem: *Theorem.* If at both the beginning and the end of a transfer the non-cannibalism conditions $((M_L=0) \vee (M_L \geq C_L))$ and $((M_R=0) \vee (M_R \geq C_R))$ are satisfied for the two river banks, then the non-cannibalism condition for the boat, i.e. $((M_b=0) \vee (M_b \geq C_b))$, is also satisfied.

Proof. At the beginning and the end of each transfer we have $M_L+M_R=C_L+C_R=N$; also, by supposition, the following two conditions hold simultaneously both at the beginning and at the end of a transfer:

$$\begin{aligned} (1) & ((M_L=0) \vee (M_L=C_L) \vee (M_L > C_L)), \\ (2) & ((N-M_L=0) \vee (N-M_L=N-C_L) \vee (N-M_L > N-C_L)). \end{aligned} \quad (4.4)$$

The conjunction of the above two conditions is equivalent to the following condition:

$$(M_L=0) \vee (M_L=N) \vee (M_L=C_L). \quad (4.5)$$

But now in order to maintain this condition over a transfer, the boat can either carry a pure load of cannibals (to conserve $(M_L=0)$ or $(M_L=N)$) or a load with an equal number of missionaries and cannibals (to conserve $(M_L=C_L)$) or a load with a number of missionaries that exceeds the number of cannibals (for a transition from $(M_L=N)$ to $(M_L=C_L)$ or $(M_L=0)$, or a transition from $(M_L=C_L)$ to $(M_L=0)$). This conclusion is equivalent to asserting the non-cannibalism condition for the boat, i.e. $((M_b=0) \vee (M_b \geq C_b))$.

The previous theorem enables us to eliminate the non-cannibalism condition for the boat when we formulate permissible actions for realizing a transfer from one side of the river to the other. This permits the introduction

of a single compound action per transfer. We can write then a new set of rules of action as follows:

$\{(A)\}_3$: Second set of rules of action in formulation F_2

Transfer safely a set of r individuals from left to right ($T^L R$).

For a set of r individuals $x_1, \dots, x_r$, where $1 \leq r \leq k$,

$(T^L R): \alpha, at(b_k, p_L), at(x_1, p_L), \dots, at(x_r, p_L); (M_b + C_b = 0) \rightarrow$
 $\alpha, at(b_k, p_R), at(x_1, p_R), \dots, at(x_r, p_R); (M_b + C_b = 0),$
 $((M_L = 0) \vee (M_L \geq C_L)), ((M_R = 0) \vee (M_R \geq C_R))$

Transfer safely a set of r individuals from right to left ($T^R L$).

The definition of this transfer action is obtained from ($T^L R$) by interchanging the places p_L and p_R throughout the definition.

It is clear that the formulation of the second set of rules of action has the effect of appreciably reducing the size of the N -state space that has to be searched, relative to the search space for the first set of rules of action. The transfers act as macro-actions, on basis of which the solution can be constructed without having to consider the fine structure of their component actions (loading the boat, unloading, crossing the river), thus without having to construct and consider intermediate N -states that are not needed for the key decisions that lead to the desired schedule.

Note that the reduction of the search space becomes possible because of the use of a formal property of our problem that enables the elimination of a redundant condition. The examination of the set of conditions of a problem, with the objective of identifying eliminable conditions and of reformulating accordingly the N -state space over which search proceeds, is one of the important approaches towards an increase in problem solving power.

5. FORMULATION F_2 OF THE M&C PROBLEM IN AN IMPROVED SYSTEM OF PRODUCTIONS

The notions that we have initially introduced in the description languages of the production systems of the previous sections reflect a general *a priori* approach to problems of reasoning about actions (i.e. consider as basic elements the individuals, the objects and the places that are specified in the problem, and consider as basic relations the elementary associations of individuals to places, etc), and also a problem-specific process of formulating concepts and attributes that are suggested from the verbal statement of the problem and that appear necessary for the expression of permissible transitions in the N -state space (notions such as M_L, C_L , etc. and the associated integers and arithmetic relations).

After several formulations of the problem, it becomes apparent that the description languages can be *restricted* and the formulation of N -states and of transitions between N -states can be considerably simplified. First, it is obvious that there is no need to use distinct individuals in the formulations. It suffices to use the compound elements, i.e. the sets $\{m\}_L, \{m\}_R, \{m\}_b, \{c\}_L, \{c\}_R, \{c\}_b$. Furthermore, since the conditions of the problem are expressed as

arithmetic properties of the sizes of the compound elements, it suffices to consider the entities $M_L, M_R, M_b, C_L, C_R, C_b$, the set of integers J_0^n and the arithmetic relations and operations. The main idea in this language restriction is that only those elements are to remain that are necessary for expressing the rules of action—that define the permissible transitions between N -states.

Because of the conservation of the total number of missionaries and the total number of cannibals throughout the transportation process, we have for each N -state (i.e. for each beginning and end of a transfer action) the following relationships:

$$M_L + M_R = C_L + C_R = N. \quad (5.1)$$

Thus, it is sufficient to consider explicitly either the set M_L, M_b, C_L, C_b or the set M_R, M_b, C_R, C_b ; we choose to consider the former. Finally, we introduce two variables B_L, B_R in the restricted language such that

$$\begin{aligned} at(b_k, p_L) &\equiv (B_L = 1) \equiv (B_R = 0) \\ at(b_k, p_R) &\equiv (B_L = 0) \equiv (B_R = 1). \end{aligned} \quad (5.2)$$

In the restricted N -state language the basic description of an N -state has the form

$$(M_L = i_1), (C_L = i_2), (B_L = i_3),$$

where i_1, i_2 are integers from J_0^N , and i_3 is 1 or 0. Such a description can be abbreviated to take the form of a vector (M_L, C_L, B_L) , whose components are the numerical values of the key variables. The vector description shows explicitly the situation at the left river bank. Thus, the initial N -state of the M & C problem—expressed in the abbreviated vector notation—is $(N, N, 1)$, and the terminal N -state is $(0, 0, 0)$.

We can now express the rules of action as follows:

$\{(A)\}_4$: Set of rules of action in Formulation F_3 .

Transfer safely a mix (M_b, C_b) from left to right (TLR, M_b, C_b) .

Any pair (M_b, C_b) such that $1 \leq M_b + C_b \leq k$, specifies a feasible action; for each such pair, we have a transition:

$$\begin{aligned} (TLR, M_b, C_b): (M_L, C_L, 1); \Lambda \rightarrow (M_L - M_b, C_L - C_b, 0); \\ ((M_L - M_b = 0) \vee (M_L - M_b \geq C_L - C_b)), \\ ((N - (M_L - M_b) = 0) \vee (N - (M_L - M_b) \geq N - (C_L - C_b))). \end{aligned}$$

Here M_b, C_b are the number of missionaries and the number of cannibals respectively that are involved in the transfer.

Transfer safely a mix (M_b, C_b) from right to left (TRL, M_b, C_b) .

Again, any pair (M_b, C_b) such that $1 \leq M_b + C_b \leq k$, specifies a feasible action; for each such pair, we have a transition:

$$\begin{aligned} (TRL, M_b, C_b): (M_L, C_L, 0); \Lambda \rightarrow (M_L + M_b, C_L + C_b, 1); \\ ((M_L + M_b = 0) \vee (M_L + M_b \geq C_L + C_b)), \\ ((N - (M_L + M_b) = 0) \vee (N - (M_L + M_b) \geq N - (C_L + C_b))). \end{aligned}$$

The restriction of the N -state language, and the introduction of new basic descriptions for N -states and of new rules of transitions between N -states has a significant effect on the relative ease with which a solution of the M&C problem can be found. The irrelevant variety of transitions that is possible when individuals are considered, is now reduced to a meaningful variety that depends on the relative sizes of appropriately defined groups of individuals. In reasoning about the M&C problem, a completely different viewpoint can now be used. We do not have to think of individuals that are being run through a sequence of processes of loading the boat, moving the boat, etc. but we can concentrate on a sequence of vector additions and subtractions that obey certain special conditions and that should transform a given initial vector to a given terminal vector. The construction of a solution amounts to finding such a sequence of vector operations. The transition to the present formulation of the M&C problem illustrates an important process of improving a problem solving system by choosing an 'appropriate' N -state language and by using this language in an 'appropriate' way to define N -states and transitions between them.

6. FORMULATION F_4 OF THE M&C PROBLEM IN A REDUCTION SYSTEM

The previous formulations F_2 and F_3 of the M&C problem were in systems of productions. A solution to our problem in these systems amounts to finding the shortest schedule (or the shortest trajectory) from the initial N -state to the terminal N -state, if there exists a trajectory between these states (i.e. if there exists a solution at all). Note that this is a typical problem of derivation.

Let us formulate now the problem in a form that will permit us to specify a reduction procedure¹ for its solution. To specify the search space for the reduction procedure we need the notions of problem states (P -states) and the set of relevant moves—terminal and nonterminal. These notions correspond respectively to formulas, axioms and rules of inference in some natural inference system (Amarel, 1967).

P -states are expressions of the form $S = (s_a \Rightarrow s_b)$. In its logic interpretation, such an expression is a proposition that means ' s_b is attainable from s_a '. Thus, it is equivalent to the logical notion $\text{CAN}(s_a, s_b)$ that has been used by McCarthy (1963) and Black (1964) (in their formalization of problems of 'ordinary reasoning'), and that has been recently discussed by Newell (1966) and Simon (1966).

In the following, we consider the formulation F_3 in the improved system of

¹ We have studied previously reduction procedures in the context of theorem-proving problems (Amarel, 1967) and syntactic analysis problems (Amarel, 1965). In these cases, the initial formulation of the problem was assumed to be in a system of productions. However, in the M&C problem, a formulation in a system of productions is a derived formulation that results from the translation of an initial verbal formulation.

productions as the starting point for the present formulation F_4 . Thus, the initial P -state for the general $M \& C$ problem is

$$S_0 = ((N, N, 1) \Rightarrow (0, 0, 0)). \quad (6.1)$$

A relevant *nonterminal move* corresponds to the application of a permissible action at the left N -state of a P -state. Thus, given a P -state $S_i = (s_a \Rightarrow s_b)$, and a permissible action A that takes s_a to s_c , then the application of the action at s_a corresponds to the application of a move (call it A also) that reduces S_i to the P -state $S_j = (s_c \Rightarrow s_b)$. We can represent such a move application as follows:

$$\begin{array}{c} S_i = (s_a \Rightarrow s_b) \\ \updownarrow A \text{ (a permissible action that takes } s_a \text{ to } s_c) \\ S_j = (s_c \Rightarrow s_b) \end{array}$$

In the logic interpretation, such a move corresponds to the inference ' S_j implies S_i ' (this is the reason for the direction of the arrows). In other words, 'if s_b is attainable from s_c , then s_b is also attainable from s_a (because s_c is known to be attainable from s_a)'.

A terminal move in the present formulation, is a move that recognizes that the left and right sides of a P -state are identical; we call it M_t . Logically, such a move corresponds to the application of an axiom scheme for validation in the natural inference system.

A solution is a sequence of P -states, attained by successive applications of nonterminal moves, starting from the initial state and ending in a state where the terminal move applies. In the logic interpretation, a solution is a proof that the initial P -state is valid, i.e. that the terminal N -state is attainable from the initial N -state. From a solution in the reduction system, it is straightforward to attain a trajectory in the system of productions or the schedule of actions that is associated with such a trajectory.

7. THE SEARCH FOR SOLUTION IN THE REDUCTION SYSTEM

A simple search process by successive reductions can be used to obtain the solution. All relevant nonterminal moves are taken from a P -state. If a new P -state is obtained which is identical to a parent P -state in the search tree, then the development below that P -state stops. This guarantees the attainment of a simplest schedule if one exists and it provides a basis for a decision procedure, i.e. if all possible lines of development from the initial P -state are stopped, then no solution exists.

The search graphs for the cases $(N=3, k=2)$ and $(N=5, k=3)$ are shown in figure 7.1. These are condensations of search trees that are obtained by retaining only one copy of a P -state and its continuations. For simplicity, except for the initial and terminal P -states, all the P -states are represented by their left N -states (they all share the same right side; i.e. the desired terminal

N -state). The branches of the graphs represent move applications. The arrows indicate the direction of transfer actions for move applications. A solution is indicated in figure 7.1 by a path in heavy lines. The schedule associated with a solution path is shown at the left of each graph as a sequence of transfer actions. Thus one (of the four possible) optional schedules for the elementary $M \& C$ problem ($N=3, k=2$) reads as follows:

- (1) Transfer two cannibals from left to right.
- (2) Transfer back one cannibal to the left.
- $\vdots$
- (6) Transfer one missionary and one cannibal from right to left.
- $\vdots$
- (11) Transfer two cannibals from left to right.

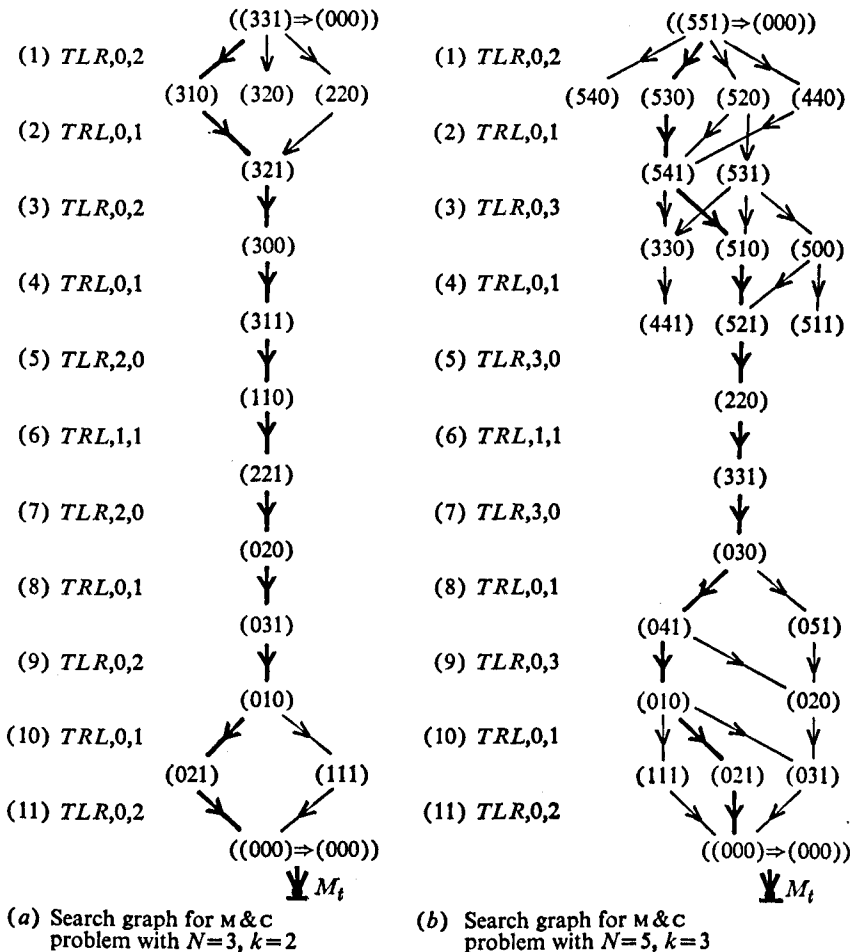


Figure 7.1. Search graphs for $M \& C$ problems in formulation F_4 .

In each case shown in figure 7.1 there is more than one solution. However, it is interesting to note that even if there is a certain amount of variety at the ends of the solution paths, the central part of the path has no variety (in the cases presented here, the center of the path is unique, in some other cases there may be two alternatives at the graph's neck, as we shall see in a subsequent example for $N=4$, $k=3$).

It should be evident from these search graphs that the M&C problem is a relatively simple problem that can be easily handled in an exhaustive search with a procedure of reduction type. There is no need for heuristics and complex rules for selecting moves and organizing the search. It is noteworthy that such a problem, while easily handled by computer procedures, is a relatively difficult problem for people. If one's approach is to try alternative sequences in some systematic manner (the computer approach that was just described) he becomes quickly memory limited. Also, people tend not to consider moves that, even though applicable to a situation, appear to be *a priori* bad moves on basis of some gross criterion of progress. In the elementary M&C problem, the sixth move in the schedule is such a stumbling block—yet it is the only move applicable.

Because of the one-sided development of the solution (from the initial N -state forward in time), and because of the exhaustiveness of the search, the process of searching for a solution would be the same if a reduction procedure (as described here) or a *generation procedure*, based directly on the formulation F_3 , were used. In a generation procedure, all the sequences of N -states that are attainable from the initial N -state are constructed. The system is actually made to run over its permissible trajectories. The reduction approach was introduced at this stage, in order to show the equivalence between the generational approach (where the system is made to run between two given points) and the reductionist-logical approach (where essentially a proof is constructed that a trajectory exists between the two given points). While the reduction-logical approach has no advantage over the generational approach in the present formulation, there are cases where such an approach is especially useful. For example, in the next stage of formulation of the M&C problem it is convenient and quite natural to develop the approach to solution *via* a reduction procedure and its associated logical interpretation.

8. DISCOVERY AND UTILIZATION OF SYMMETRIES IN THE SEARCH SPACE. FORMULATION F_4 OF THE M&C PROBLEM

From an analysis of the search graphs for M&C problems (such as those in figure 7.1), it becomes apparent that the situation in search space is *symmetric with respect to time reversal*. Roughly, if we run a movie of a schedule of transportations forwards or backwards, we can't tell the difference. Consider two N -states (M_L, C_L, B_L) and ($N-M_L, N-C_L, 1-B_L$) in N -state space. When the space is viewed from the vantage point of each N -state in this pair, it appears identical, provided that the direction of transitions is 'perceived' by one N -

state as opposite to the direction 'perceived' by the other N -state. For example, consider the points (311) and (020) in the elementary M & C problem (see figure 7.1(a)). If we consider (311) on a normal time path, then it is reached *via* (TRL,0,1) and it goes to the next state *via* (TLR,2,0); if we consider (020) under time reversal, then it is reached *via* (TRL,0,1) and it goes to the 'next' state *via* (TLR,2,0). We shall consider now this situation more formally.

In our previous formulations of the M & C problem within production systems, the rules of action define a relation of direct attainability T between successive N -states (see section 2). Thus, for any two N -states s_a, s_b , the expression $s_a T s_b$ asserts that the N -state s_a occurs just earlier than s_b on a trajectory in N -state space. Consider now the converse relation $\check{T}$. The expression $s_a \check{T} s_b$ asserts that s_a occurs *just after* s_b on a trajectory.

We shall consider specifically in the following discussion the formulation of the M & C problem in the improved system of productions, i.e., the formulation F_3 . Let σ be the space of N -states, partly ordered under the relation T , and $\check{\sigma}$ its dual space (i.e., $\check{\sigma}$ has the same elements of σ , partly ordered under $\check{T}$). Consider now the following mapping θ between N -states:

$$\theta: (M_L, C_L, B_L) \rightarrow (N-M_L, N-C_L, 1-B_L) \quad (8.1)$$

We can also write θ as a vector subtraction operation as follows:

$$\theta(s) = (N, N, 1) - s. \quad (8.2)$$

Theorem. For any pair of N -states s_a, s_b the following equivalence holds:

$$s_a T s_b \equiv \theta(s_a) \check{T} \theta(s_b),$$

or equivalently

$$s_a T s_b \equiv \theta(s_b) T \theta(s_a);$$

i.e. the spaces $\sigma, \check{\sigma}$ are anti-isomorphic under the mapping θ . Furthermore, the move that effects a permissible transition from s_a to s_b is identical with the move that effects a permissible transition from $\theta(s_b)$ to $\theta(s_a)$.

Proof. Consider any permissible N -state (i.e. the non-cannibalism conditions are satisfied at this state) with the boat at left; suppose that this N -state is described by the vector $s_a = (M_L, C_L, 1)$. Corresponding to s_a we have an N -state described by $\theta(s_a) = (N-M_L, N-C_L, 0)$. Note that, in general, the non-cannibalism conditions (stated in (4.4)) are invariant under θ . Thus, the N -state described by $\theta(s_a)$ is also permissible. We can also write in vector notation,

$$\theta(s_a) = (N, N, 1) - s_a. \quad (8.3)$$

Consider now a transition from left to right at s_a , defined by some pair (M_b, C_b) such that $1 \leq M_b + C_b \leq k$. A transition of this type is always *a priori* possible if $M_L + C_L \neq 0$ in s_a (i.e. if there is *somebody* at left when the boat is there—a condition which we are obviously assuming); however the *a priori* possible transition is not necessarily permissible—in the sense of satisfying the

non-cannibalism conditions at the resulting N -state. The transition defined by (M_b, C_b) yields a new vector s_b that is related to s_a by vector subtraction as follows:

$$s_b = s_a - (M_b, C_b, 1). \quad (8.4)$$

This can be verified by examining the rules of action. Corresponding to s_b we have via the mapping θ ,

$$\begin{aligned}\theta(s_b) &= (N, N, 1) - s_b = (N, N, 1) - s_a + (M_b, C_b, 1) \\ &= \theta(s_a) + (M_b, C_b, 1).\end{aligned}\quad (8.5)$$

Suppose first that s_b is permissible (which means that the move defined by the pair (M_b, C_b) is permissible, and the relation $s_a T s_b$ holds); then $\theta(s_b)$ is also permissible because of the invariance of the non-cannibalism conditions under θ . Now in the N -state described by $\theta(s_b)$ the boat is at left and a left to right transition defined by (M_b, C_b) is possible (in view of (8.5) and noting that the components of $\theta(s_a)$ cannot be negative). This transition yields a vector $\theta(s_b) - (M_b, C_b, 1)$, which is identical with $\theta(s_a)$. Since $\theta(s_a)$ is permissible, then the transition defined by (M_b, C_b) (which takes $\theta(s_b)$ to $\theta(s_a)$) is permissible, and the relation $\theta(s_b) T \theta(s_a)$ holds. It is inherent in this argument that the same move that takes s_a to s_b , also takes $\theta(s_b)$ to $\theta(s_a)$.

Suppose now that s_b is not permissible (which means that the relation $s_a T s_b$ does not hold); then $\theta(s_b)$ is not permissible either, and the relation $\theta(s_b) T \theta(s_a)$ does not hold.

A similar argument can be developed for a right to left transition. This establishes the anti-isomorphism and the relationship between symmetric moves.

The situation can be represented diagrammatically as follows:

$$T \begin{array}{c} s_a \rightarrow \cdots \rightarrow \theta(s_a) \\ \downarrow T \\ s_b \rightarrow \cdots \rightarrow \theta(s_b) \end{array} \quad \begin{array}{c} \theta \\ \theta \end{array} \quad \begin{array}{c} \theta(s_a) \\ \uparrow T \\ \theta(s_b) \end{array} \quad \begin{array}{c} \tilde{T} \\ \downarrow \tilde{T} \end{array} \quad (8.7)$$

Corollary. For any pair of N -states s_a, s_b , the following equivalence holds:

$$(s_a \Rightarrow s_b) \equiv (\theta(s_b) \Rightarrow \theta(s_a)).$$

The proof is an extension of the previous proof.

The recognition of the anti-isomorphism permits us to approach the problem simultaneously, and in a relatively simple manner, both in the space σ and in its dual space. The reasoning behind this dual approach relies on the logical properties of the attainability relation $\Rightarrow$, and on the properties of the anti-isomorphism.

Consider an attainability relation ($s_0 \Rightarrow s_b$), where s_0 is the initial N -state and s_b is an arbitrary N -state such that $s_b \neq s_0$. Let us denote by $\{s_1\}$ the set of all N -states that are directly attainable from s_0 ; thus

$$\{s_1\} = \{s | s_0 Ts \text{ holds}\}. \quad (8.8)$$

We have then

$$(s_0 \Rightarrow s_b) \equiv \bigvee_{s \in \{s_1\}} (s \Rightarrow s_b). \quad (8.9)$$

If $s_b = s_t$, where s_t is the desired terminal N -state, then we have as a special case of (8.9),

$$(s_0 \Rightarrow s_t) \equiv \bigvee_{s \in \{s_1\}} (s \Rightarrow s_t). \quad (8.10)$$

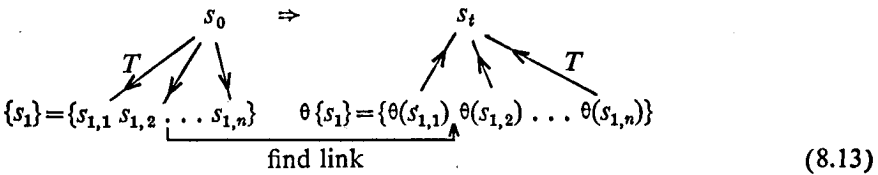
From the previous corollary, and since $\theta(s_t) = s_0$ in the M&C problem, we can write the equivalence (8.10) as follows:

$$(s_0 \Rightarrow s_t) \equiv \bigvee_{s \in \{s_1\}} (s_0 \Rightarrow \theta(s)). \quad (8.11)$$

By using (8.9) in (8.11) we obtain:

$$(s_0 \Rightarrow s_t) \equiv \bigvee_{s_i \in \{s_1\}} \left(\bigvee_{s_j \in \{s_1\}} (s_j \Rightarrow \theta(s_i)) \right). \quad (8.12)$$

The situation can be shown schematically as follows:



The terminal N -state s_t is attainable from s_0 if and only if any of the N -states from which s_t is directly attainable is itself attainable from any N -state that is directly attainable from s_0 .

Now for each growth below $s_{1,i} \in \{s_1\}$, there is a corresponding *image growth* below $\theta(s_{1,i})$. Let us denote the set of all N -states that are directly attainable from elements of $\{s_1\}$ by $\{s_2\}$; thus

$$\{s_2\} = \{s | s_a \in \{s_1\}, s_a Ts \text{ holds}\}. \quad (8.14)$$

Let us call the image of $\{s_2\}$ under θ , $\theta\{s_2\}$. Repeating the previous argument we obtain that s_t is attainable from s_0 if and only if any of the N -states in $\theta\{s_2\}$ is attainable from any of the N -states in $\{s_2\}$. This type of argument can be continued until either a set $\{s_n\}$ at some level n does not have any new progeny, or an N -state in $\theta\{s_n\}$ is directly attainable from an N -state in $\{s_n\}$.

From the preceding discussion, it is clear that we can develop the search for solution simultaneously, both forward from the initial N -state and backward from the terminal N -state, without having to spend search effort in both sides. Only the sets $\{s_1\}$, $\{s_2\}$, $\dots$ $\{s_n\}$, that represent the forward exploration of the search space from the initial N -state, have to be constructed. The exploration from the terminal N -state backwards is directly obtainable as the image of the forward exploration under time reversal (i.e. under the anti-isomorphism). This means that the knowledge of the symmetry property permits us to cut the depth of search by a factor of two—which is a substantial reduction in expected search effort. Note, however, that as is the case in any two-sided approach to search, new problems of coordination and recognition arise because of the need to find links between the forward moving search front and its backward moving image. In our present problem, because of the relative narrowness of the moving fronts, this problem of recognizing a linking possibility is not too difficult.

Let us formulate now a reduction procedure for carrying out the two-sided solution construction activity that we have just described. We introduce here a broader concept of a problem state, the *total P -state*, Σ_i :

$$\Sigma_i = (\{s_i\} \Rightarrow \theta\{s_i\}), i = 0, 1, 2, \dots$$

where i indicates the number of transitions from one of the schedule terminals (initial or terminal N -state) and the current total P -state. In its logic interpretation, an expression Σ_i stands for the proposition 'there exists an N -state in $\{s_i\}$ from which some N -state in $\theta\{s_i\}$ is attainable'.

A *nonterminal move* in the present formulation is a broader notion than a nonterminal move in our previous reduction procedure. Here, a nonterminal move effects a transition between Σ_i and Σ_{i+1} in such a manner that $\Sigma_i \equiv \Sigma_{i+1}$. Such a move represents a combination of parallel transfers, half of which are source-based and they are found by direct search, and the other half are destination-based and they are computed on basis of the symmetry property.

A *terminal move* in the present formulation establishes links between N -states in $\{s_i\}$ and N -states in $\theta\{s_i\}$ that are directly attainable from them.

A *solution* (or correspondingly an attainability proof) has the form of a chain of total P -states that start with $\Sigma_0 = (s_0 \Rightarrow s_t)$ and that ends with a total P -state Σ_n where a terminal move applies. A trajectory (or a schedule) is obtained from this solution by tracing a sequence of N -states that starts with s_0 ; it is followed by a directly attainable N -state in $\{s_1\}$; it continues this way up to $\{s_n\}$, and then it goes to $\theta\{s_n\}$, $\theta\{s_{n-1}\}$, $\dots$ up to $\theta(s_0) = s_t$.

The development of the solution for the elementary M & C problem in the present formulation is shown in figure 8.1.

The total P -state Σ_i is valid because there is a link (via *TRL*, 1,1) between 110 and 221. The darkened path shows a solution trajectory. The schedule associated with the trajectory is given at left. The same transfer actions apply at points of the trajectory that are equidistant from the terminals. Thus, in the

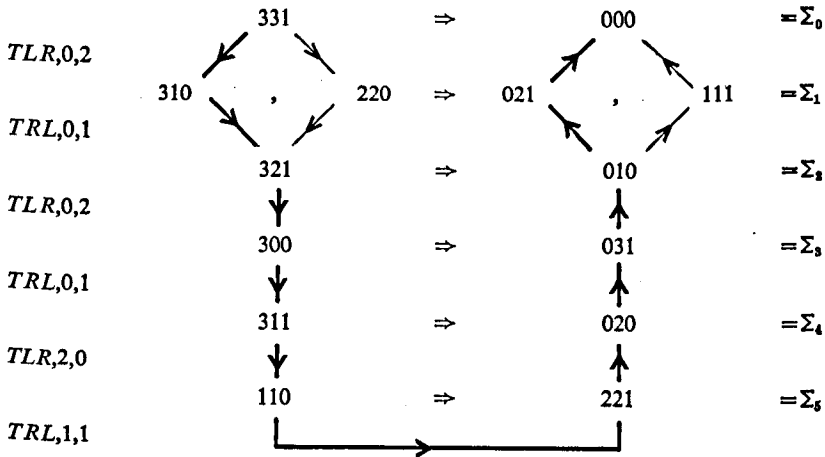


Figure 8.1. Search graph for the elementary $m \& c$ problem in the formulation F_s .

present case, we have a schedule which is symmetrical with respect to its middle point. Note that the solution development given in figure 8.1 is a folded version of the solution development which is given in figure 7.1(a).

It is of interest to develop the solution for the case $N=4$, $k=3$ within the present formulation; this is given next in figure 8.2.

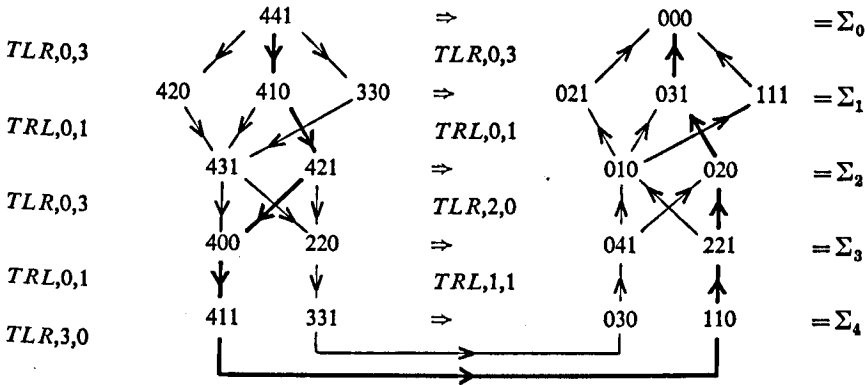


Figure 8.2. Search graph for the $m \& c$ problem ($N=4$, $k=3$) in the formulation F_s .

The total P -state Σ_4 is valid, since a terminal move composed of two links applies at Σ_4 . The darkened path in figure 8.2 shows one solution trajectory. The schedule associated with the trajectory is shown in the sides of the solution graph. Note that in the present case the trajectory is not symmetrical. While the two halves of the search graph are images of each other under θ , the two halves of a trajectory are not. Roughly the situation is as follows: Two main sequences of N -states grow from each of the two sides; these two

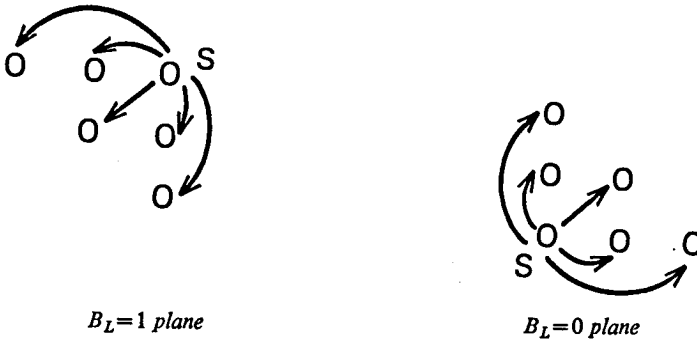
sequences are images of each other under θ ; a solution trajectory starts with one of these sequences from the one side, and then at its middle point, rather than continuing with the image of the initial sequence, it flips over to the image of the second sequence.

In the present formulation, it is possible again to develop a solution *via* a generation procedure that would operate in an equivalent manner to the reduction procedure that we have described here. However, the direct correspondence between the logic of the solution and the elements of the reduction procedure make the latter more convenient to use.

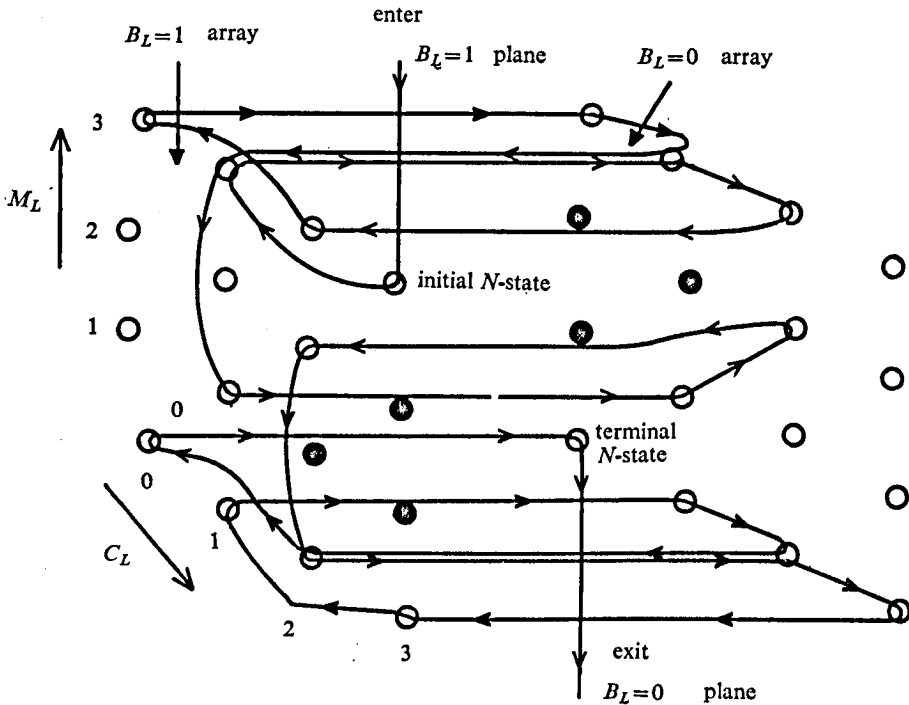
9. DISCOVERY OF SOLUTION PATTERNS IN AN APPROPRIATE REPRESENTATION OF N -STATE SPACE

One of the significant ways of increasing the power of a problem solving system for the M&C problem is to look for some characteristic patterns in its search space that go beyond the properties that we have discussed so far. To this end, it is extremely important to find a representation of the search space that enables a global view of the situation, so that reasoning about a solution can first proceed in broad terms and it can then be followed by the detailed scheduling of actions. We shall present next such a representation of the space of N -states. This representation utilizes the basic description of N -states that was introduced in the formulation F_3 of the M&C problem.

The number of possible N -states for an M&C problem equals the number of possible valuations of the vector (M_L, C_L, B_L) ; this number is $2(N+1)^2$. We represent the space of N -states by a limited fragment of three-dimensional space with coordinates M_L, C_L and B_L . This fragment consists of two parallel square arrays of points, that are disposed as follows: One array is on the plane $B_L=0$ and the other on the plane $B_L=1$; the points on each array have coordinates (M_L, C_L) , where the values of M_L, C_L are $0, 1, 2, \dots, N$. Thus, each point corresponds to a possible N -state. Such a representation for the N -state space of the elementary M&C problem is shown in figure 9.1. The blackened points stand for non-permissible N -states (i.e. the non-cannibalism conditions are violated in them). The feasible transitions from an N -state s in a given B_L plane to other N -states in the same plane are shown in figure 9.2. These feasible transitions reflect mainly boat capacity. A feasible transition is not permissible if it leads to a non-permissible N -state. Thus, starting from an N -state in the $B_L=1$ plane, a transition can be made to any permissible point within a 'distance' of 2 lattice steps in the plane, in a general southwestern direction; after the movement in the plane is carried out (it represents 'load-in the boat' at left) a left-to-right transfer action is completed by jumping from the $B_L=1$ array to the $B_L=0$ array in a direction parallel to the B_L axis. A right-to-left transfer starts from an N -state in the $B_L=0$ plane; a transition is first made to a permissible point within a 'distance' of 2 lattice steps in the plane, in a general northeastern direction; after this transition, the transfer is completed by jumping across to the $B_L=1$ array.

Figure 9.1 Feasible transitions in space of N -states

A solution for the elementary M & C problem is shown in figure 9.1 as a path in N -state space. It is suggestive to regard the solution path as a thread entering the initial N -state, leaving the terminal N -state, and woven in a specific pattern of loops that avoids going through the non-permissible points in N -space. Furthermore, the solution shown in figure 9.1 requires the 'least

Figure 9.2. Space of N -states for elementary M & C problem

amount of thread' to go from the initial N -state to the terminal N -state within the imposed constraints in the weaving pattern. It is easy to see that the solution trajectory shown in figure 9.2 is the same as the solution shown in figure 7.1(a).

We can simplify the representation of N -state space by collapsing it into a single square array of $(N+1)^2$ points (figure 9.3). This requires a more complex specification of the possible transitions. We represent a left-to-right transfer by an arrow with a black arrowhead, and a right-to-left transfer by an arrow with a white arrowhead. In the previous two-array representation, a black arrow corresponds to a movement in the $B_L=1$ plane that is followed by a jump across planes, and a white arrow corresponds to a movement in the $B_L=0$ plane followed by a jump across planes. A point in the collapsed space is given by two coordinates (M_L, C_L) , and it can represent either of the two N -states $(M_L, C_L, 1)$ or $(M_L, C_L, 0)$. The point (M_L, C_L) in association with an entering black arrowhead represents $(M_L, C_L, 0)$; in association with an entering white arrowhead, it represents $(M_L, C_L, 1)$. A sequence of two arrows $\rightarrow \rightarrow$ represents a round trip left-right-left. A sequence of arrows, with alternating arrowhead types, that starts at the initial point (N, N) and ends at the terminal point $(0, 0)$ represents a solution to the M & C problem.

The collapsed N -state space for the elementary M & C problem is shown in figure 9.3. The solution path shown in this figure represents the same solution

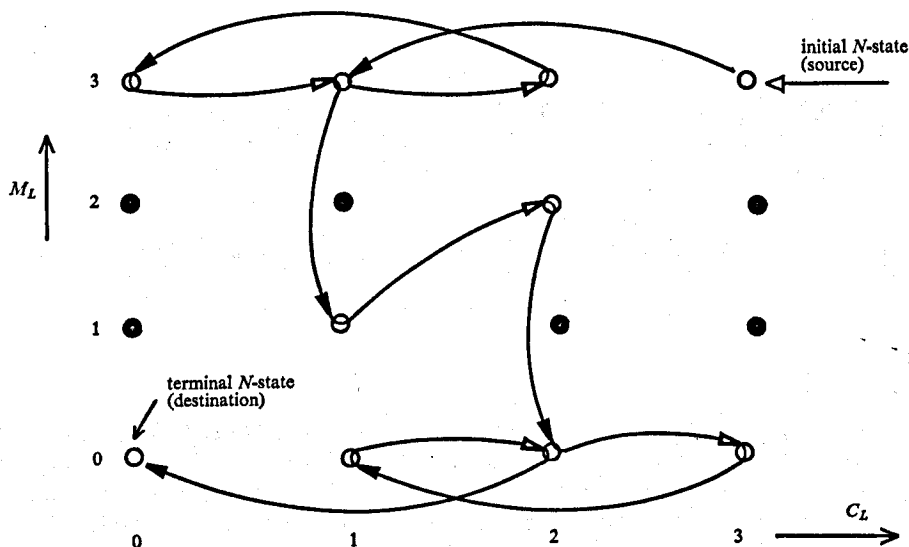


Figure 9.3. Collapsed N -space for elementary M & C problem

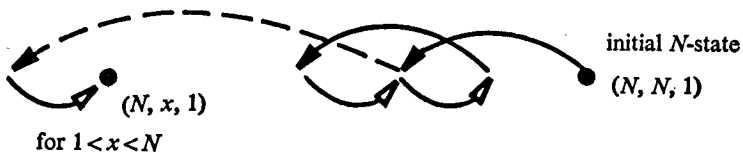
be obtained directly as consequences of the problem constraints; we have used them in the proof of the eliminability of the 'boat condition' in section 4, and it is conceivable that they could be derived mechanically with techniques that are presently available. Note, however, that the problem of obtaining these conditions is not a theorem proving task but a *theorem finding task*.

Let us concentrate now on the **Z** region of interest in the collapsed N -state space of an M&C problem, and let us attempt to find general characteristic features of solution paths. Since the **Z** region is the permissible territory, it is reasonable to expect that features of solution paths are describable in terms of movement types over this **Z**. By examining the diagram in figure 9.4 we shall try first to identify certain properties of solution paths that will permit us to characterize the solution schema that we have used in the elementary M&C problem (see figure 9.3).

In the diagram of the **Z** region, this solution schema can be seen to consist in general of four main parts, (i) to (iv). An arrow $\triangleleft---$ denotes a sequence of transitions the last of which brings the boat to the left river bank, and an arrow $\blacktriangleleft---$ denotes a sequence of transitions that terminates with the boat at right.

The following general properties of solution paths are suggested by examining the situation in figure 9.4:

- (i) On the $M_L = N$ line, any of the points $(N, x, 1)$, where $1 < x < N$, are attainable from the initial point $(N, N, 1)$ by a 'horizontal' sequence of transitions of the following type:



More generally, any point $(N, x, 1)$, where $1 \leq x \leq N$, can be attained from any other point $(N, y, 1)$, where $1 \leq y \leq N$, by some 'horizontal' sequence of transitions that is similar to the one just shown. Roughly, this indicates that 'horizontal' movements over the $M_L = N$ line are *easily achievable* by a known routine of steps.

- (ii) If k is the boat capacity, and if $k \geq 2$, then any of the points $(N, N-x, 1)$, where $0 < x \leq k$, can reach, via a single transition ($TLR, x, 0$), a point $(N-x, N-x, 0)$ on the diagonal of the **Z** region. From this point, a ($TRL, 1, 1$) transition can lead to a point $(N-x+1, N-x+1, 1)$ on the diagonal. While the first transition in this pair determines the size of the 'jump' from the $M_L = N$ line to the diagonal, the second transition is necessary for

'remaining' on the diagonal. Thus, we can regard this pair of transitions as a way of achieving a 'stable jump' from the line $M_L = N$ to the diagonal. It is clear from this discussion that a boat capacity of at least two is necessary for realizing a 'stable jump'. Note that the second transition in the pair corresponds to the critical move of returning one missionary and one cannibal—in general, an equal number of missionaries and cannibals—to the left, in mid schedule. As we have observed before, this is an unlikely move choice if the problem solver has a general notion of progress that guides his move preferences uniformly over all parts of the solution space. Only after knowing the local structure of this space, is it possible to see immediately the inevitability of this move. Now, the remotest point of the diagonal (from the initial point) that can be reached by this pair of transitions is $(N-k+1, N-k+1, 1)$.

- (iii) A point on the diagonal can directly attain a point on the line $M_L = 0$ if its distance from that line does not exceed k . Thus, to move from the $M_L = N$ line to the $M_L = 0$ line in two 'jumps', by using the diagonal as an intermediate support, we need a boat capacity that satisfies the following condition:

$$k \geq \frac{N+1}{2} \quad (9.1)$$

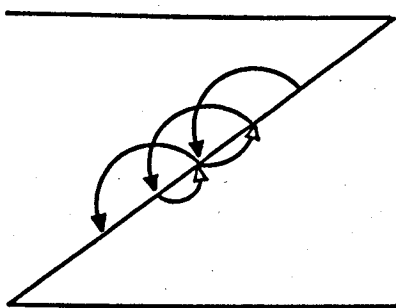
(Thus, for $N=5$ and $k=2$ there is no solution. This specific result could have been obtained in any of our previous formulations by recognizing that a definite dead end is attained in the course of searching for a solution. However, it is obtained much more directly from our present analysis; furthermore, we can easily assign the reason for the unsolvability to the low capacity of the boat.)

- (iv) On the $M_L = 0$ line, any of the points to the right of the terminal point, can reach the terminal point $(0,0,0)$ by a 'horizontal' sequence of transitions of the type shown in (i). More generally, any point $(0,x,0)$, where $0 \leq x < N$, can be attained from any other point $(0,y,0)$, where $0 \leq y < N$, by some 'horizontal' sequence of transitions. Again, this indicates roughly that 'horizontal' movement over the $M_L = 0$ line are easily achieved by a known routine of steps.

From the general properties just discussed we can characterize a general solution pattern, which we call the *zig-zag pattern*, by the following sequence of global actions: (i) starting from the initial point, slide on the $M_L = N$ line, over a 'horizontal' transition sequence, up to the point $(N, N-k, 1)$; (ii) jump on the diagonal, *via* two transitions, to the point $(N-k+1, N-k+1, 1)$; (iii) jump off the diagonal to the $M_L = 0$ line; (iv) slide on the $M_L = 0$ line, *via* a 'horizontal' transition sequence, to the terminal point.

It can be easily verified that the solutions to the three cases that we have

presented previously, i.e. $(N=3, k=2)$, $(N=4, k=3)$ and $(N=5, k=3)$, follow precisely the zig-zag pattern that we have outlined. If $N=6$, then in order to use the present solution scheme, a boat of capacity 4 is needed (see the condition (9.1)). When a boat capacity of 4 (or more) is available, then any M & C problem is solvable. This property is due to the fact that the following pattern of transitions, that allows one 'to slide along the diagonal', is possible when $k \geq 4$:



The 'sliding along the diagonal' for $k=4$ is realized by a 'diagonal' sequence of round trips of the type: $(TLR, 2, 2)$, $(TRL, 1, 1)$, $(TLR, 2, 2)$, $(TRL, 1, 1)$, etc., where each round trip realizes a net transfer of two individuals from left to right.

For cases with $k \geq 4$ it is possible to use a simple and efficient solution pattern, the *diagonal pattern*, that has a single global action, as follows: starting from the initial point slide down the diagonal via a 'diagonal' transition sequence that takes in each round trip $\frac{k}{2}$ missionaries and $\frac{k}{2}$ cannibals to the right (when k is even—otherwise it takes $\frac{k-1}{2}$ of each) and it returns one missionary and one cannibal back, except in the last trip, until the terminal point is reached. It is also possible to construct solution patterns that combine parts of the zig-zag pattern with parts of the diagonal pattern. Such a combined solution scheme is shown in figure 9.5.

For the M & C problem (i.e. find a path from $(N, N, 1)$ to $(0, 0, 0)$), it can be shown that if the boat capacity k is high, and if k is even, then the pure diagonal pattern of solution is always better than any combined pattern (in terms of number of trips required for a schedule); if k is odd, then there are cases where a small advantage is gained by starting the schedule with the first two round trips of the zig-zag pattern; if $k=4$, and $N \geq 6$, then the diagonal solution pattern, the zig-zag pattern or the combined pattern of figure 9.5, when it applies, are all of equivalent quality.

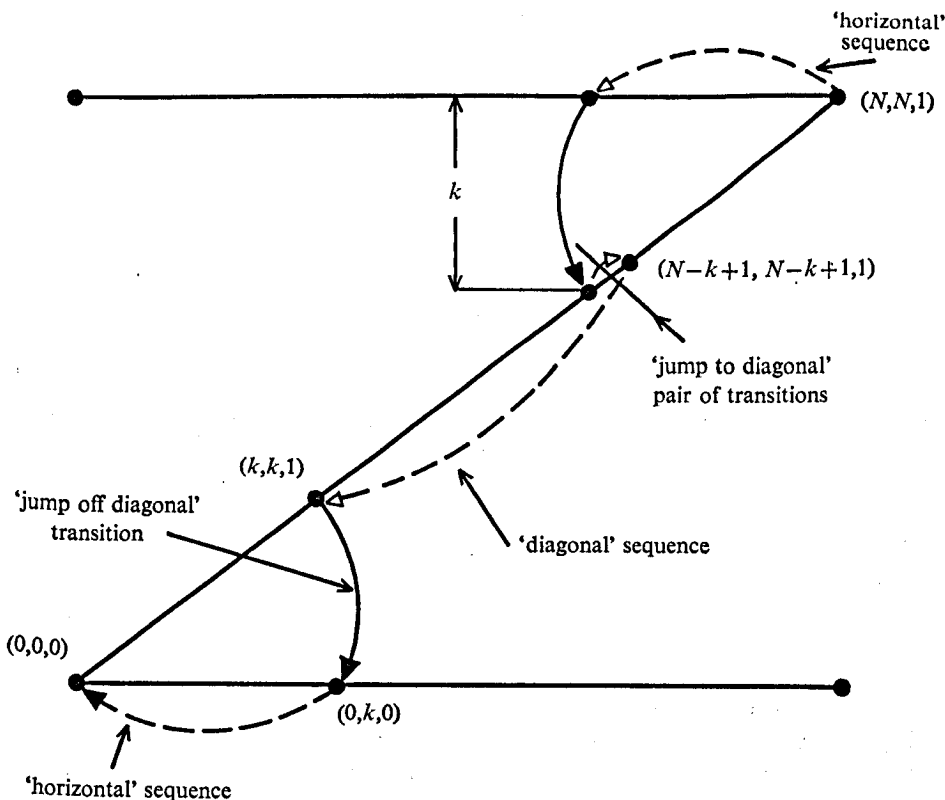


Figure 9.5. Combined scheme of solution shown on the Z region

10. FORMULATION F₆ OF EXTENDED M&C PROBLEM IN A MUCH IMPROVED PRODUCTION SYSTEM THAT CORRESPONDS TO A HIGHER LEVEL SEARCH SPACE

After the exploration of solution patterns in our array representation of N -state space, and after new global transition concepts are developed, it is possible to re-formulate the M&C problem (in fact, an extended version of this problem) in a new and much improved system of productions to which there corresponds an N -state space that has many fewer points than in any of the previous spaces.

From the analysis of possible global movements in the N -state space, we can now formulate the following set of *macro-transitions*:

$\{(A)\}_6$: set of rules of (macro) action in formulation F_6 .

$(H_1): (N, C_L, 1); 0 < C_L < N, k \geq 2 \rightarrow (N, N, 1)$

$(H_1, J_1): (N, C_L, 1); 0 < C_L \leq N, k \geq 2 \rightarrow (N-k+1, N-k+1, 1)$

$(D): (M_L, C_L, 1); 0 < M_L = C_L \leq N, k \geq 4 \rightarrow (0, 0, 0)$ (10.1)

$(J_2): (M_L, C_L, 1); 0 < M_L = C_L \leq k \rightarrow (0, C_L, 0)$

$(D, J_2): (M_L, C_L, 1); M = C_L > k \geq 4 \rightarrow (0, k, 0)$

$(H_2): (0, C_L, 0); 0 \leq C_L < N, k \geq 2 \rightarrow (0, C'_L, 0); 0 \leq C'_L < N, C_L \neq C'_L$

Each of these macro-transitions is realized by a routine of elementary transitions. Thus, (H_1) is realized by a 'horizontal' sequence of transitions that slides a point on the $M_L=N$ line to the corner point $(N,N,1)$, with the least number of steps; (H_1, J_1) is realized by a 'horizontal' sequence of transitions that takes a point on the $M_L=N$ line to the point $(N, N-k, 1)$ on that line, and then it is followed by a pair of transitions that effects a 'stable jump' to the point $(N-k+1, N-k+1, 1)$ on the diagonal, all this with the least number of steps; (D) is realized by a 'diagonal' sequence of transitions that takes a point on the diagonal to the bottom of that diagonal, in the least number of steps; (J_2) is realized by a single transition that effects a 'jump' from a point on the diagonal to the $M_L=0$ line; (D, J_2) is realized by a 'diagonal' sequence of transitions that takes a point along the diagonal to the point $(k, k, 1)$, and then it is followed by a transition that effects a 'jump' to the point $(0, k, 0)$ on the $M_L=0$ line, all this with the least number of steps; (H_2) is realized by a 'horizontal' sequence of transitions that takes a point on the $M_L=0$ line to another point on that line, in the smallest number of steps.

The formulation of the macro-transitions enables us to approach a problem of finding the best schedule for an M&C problem (or extensions of this problem) by first solving the problem in a higher order space, where we obtain a set of possible *macro-schedules*—that are defined in terms of macro-transitions—and then converting the macro-schedules to schedules by compiling in the appropriate way the macro-transition routines. Note that the present formulation is suitable for handling conveniently a class of problems which is larger than the strict class of M&C problems that we have defined in section 3; specifically, an arbitrary distribution of cannibals at left and right can be specified for the initial and terminal N -states. By certain changes in the specification of the macro-transitions, it is possible to consider within our present framework other variations of the M&C problem, e.g. cases where the boat capacity depends on the state of evolution of the schedule, cases where a certain level of 'casualties' is permitted, etc.

Let us consider now the following example:

Example 10.1. The initial situation is as follows: nine missionaries and one cannibal are at the left river bank and eight cannibals are at the right bank; a boat that has a capacity of four is initially available at left. We wish to find the simplest safe schedule that will result in an interchange of populations between the two river banks.

The search graph in the higher order space gives all the macro-schedules for the case of a constant boat capacity of four; this graph is shown in figure 10.1. The macro-transitions are applied on the left side of a P -state (i.e., the macro-schedule is developed forward in time) until a conclusive P -state is reached. The number within square brackets that is associated with a macro-transition indicates its 'weight', i.e., the number of trips in the routine that realizes the macro-transition. Thus, we have macro-schedules of weights 15, 21, and 27. The simplest macro-schedule is given by the sequence

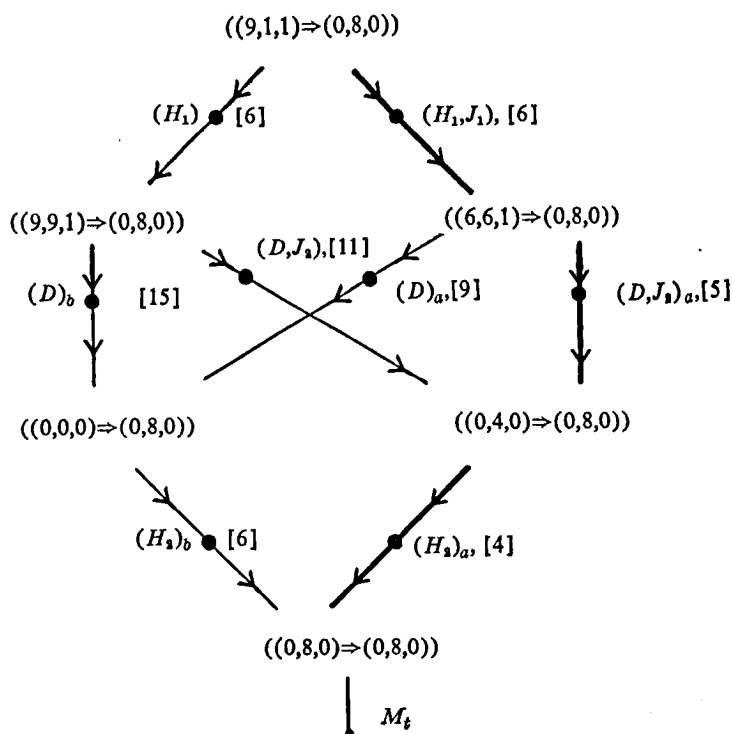
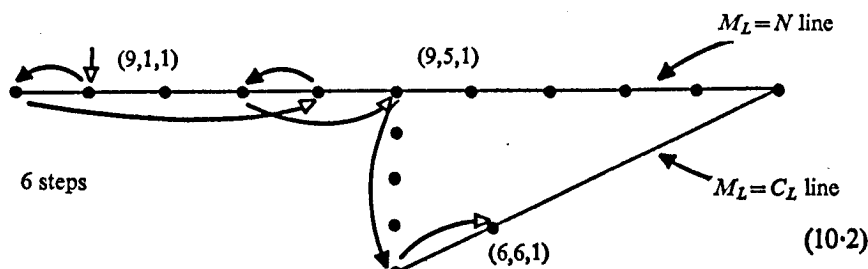


Figure 10.1 Search graph in higher order space for the example 10.1

(H_1, J_1) , $(D, J_2)_a$, $(H_2)_a$ of macro-transitions, which corresponds to the darkened path in figure 10.1.

The situation in the collapsed N -state space is shown in figure 10.2. The patterns of the alternative macro-schedules are shown schematically in the lower part of the figure.

After a macro-transition is specified, its realization in terms of elementary transitions is easily carried out by a compiling routine. For example, the macro-transition (H_1, J_1) in our problem is realized as follows by a routine (H_1, J_1) with initial N -state $(9, 1, 1)$ and a terminal N -state $(6, 6, 1)$:



As a second example, consider next the realization of the macro-transition $(D, J_2)_a$, by a routine (D, J_2) from (6.6.1) to $(0, 4, 0)$; see (10.3).

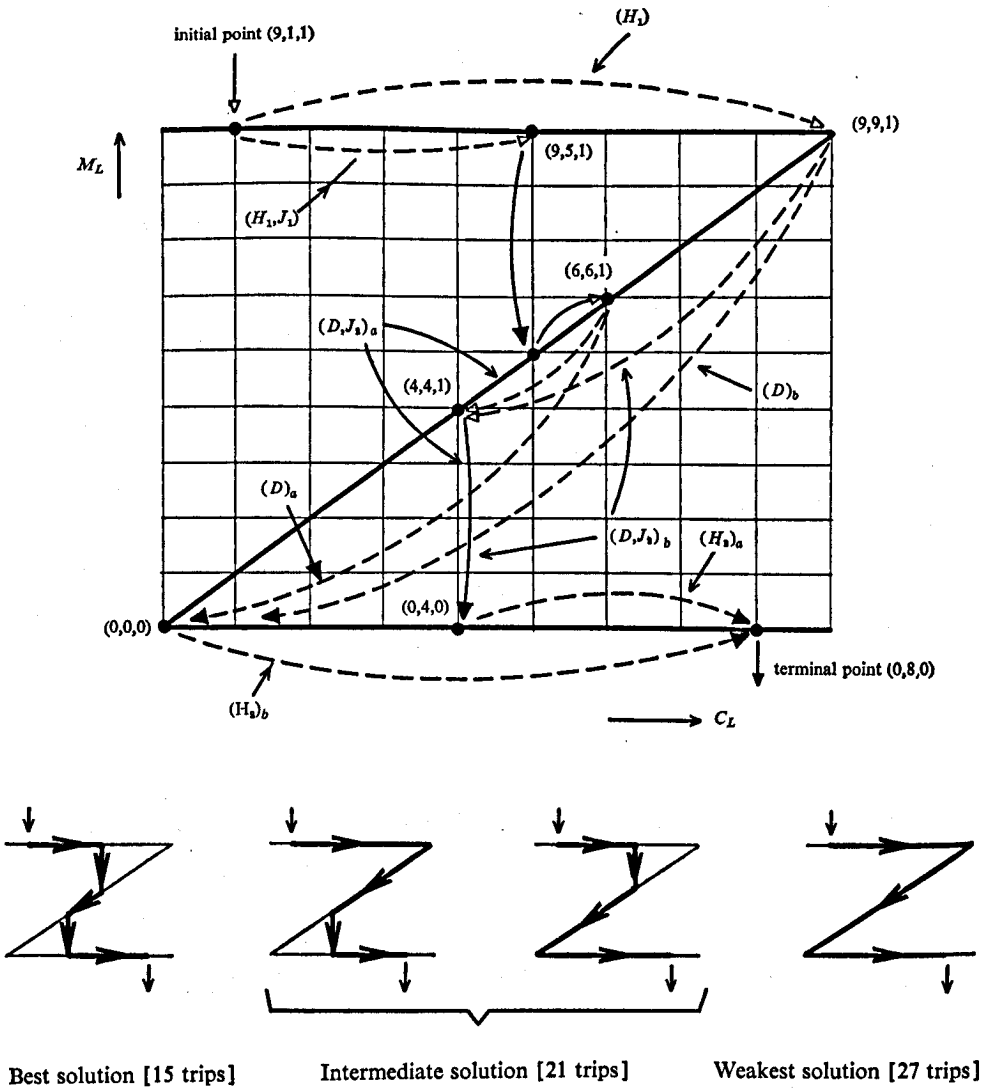
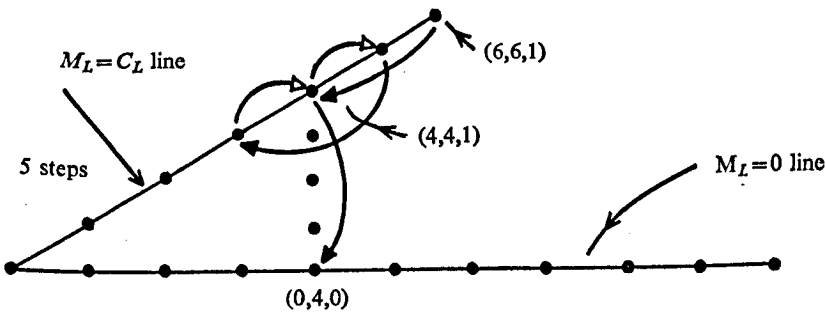


Figure 10.2. Collapsed N -state space for the example (10.1)



(10-3)

If we think of the problem in terms of path finding in the $\mathbf{Z}$ region of the collapsed N -state space, we can immediately see analogies with simple 'monkey problems'. These are problems suggested by McCarthy (1963), where a schedule of actions has to be found for a monkey that has to reach certain specified goals by moving in three-dimensional space, transferring objects from place to place, reaching objects, etc. It is clear that 'monkey problems' are simple prototypes of problems of reasoning about actions in the real world, such as assembling a physical object from parts, navigating a vehicle in a heavy traffic, etc. We can visualize our problem in the following way: a monkey is at the upper level of a two-level structure that has in its side an inclined stairway, and his goal is to reach a bunch of bananas that is at the lower level and at a certain distance from the stairway landing; suppose that the detailed geometry of the situation is as shown in the diagram of figure 10.2, where the scale of distances is in yards; suppose further that the monkey can always see the entire situation (the structure is essentially transparent): he can move over each level by using a 'horizontal' sequence of steps, he can move down the stairway by using a 'diagonal' sequence of steps, and he can safely jump vertical distances that do not exceed four yards; find a safe path that will bring the monkey to the bananas in the smallest number of steps. Clearly, the best solution trajectory for this monkey problem is isomorphic with the best solution that we have obtained for our original problem.

The solution of our illustrative problem (in any of the interpretations) would have been much more painful if the possible transitions were given as specifications of elementary steps. The availability of integrated, goal oriented, routines that specify macro-transitions is responsible for a substantial reduction in problem solving effort. A macro-transition is an expression of knowledge about the possibility of realizing certain sequences of transitions. It is a theorem about possible actions in the universe in which we are solving problems. Thus, the macro-transition (H_1, J_1) (see (10.1)) can be roughly interpreted in the 'monkey and bananas' context as asserting that it is possible for

the monkey to go from *any* place on the upper level (except one corner point) to a place on the stairway which is four yards below the upper level. The proof of this assertion consists in exhibiting a sequence of realizable elementary steps that can be used by the monkey for going from any of the initial places at the upper level to the terminal place. Note that the elementary steps have themselves the status of macro-steps with respect to a lower level of possible actions. For example, in the M & C problem, we are using now a transfer across the river as an elementary step, and this transfer is realized by more elementary actions of loading the boat, moving it, and unloading it; in the 'monkey and bananas' interpretation, an elementary step may be realized in terms of certain sequences of muscle actions.

11. RELATIONSHIPS BETWEEN THE INITIAL SEARCH SPACE AND THE HIGHER LEVEL SEARCH SPACE

The high level space σ^* in which macro-schedules are constructed consists of a subset α of the set of $2(n+1)^2$ N -states, with the elements of α partially ordered under the attainability relation that is defined by the macro-transitions $\{(A)\}_s$ (given in (10.1)). The set α contains the following elements: the initial and terminal N -states that are specified in the problem formulation, and four N -states $(N, N, 1)$, $(N-k+1, N-k+1, 1)$, $(0, 0, 0)$, and $(0, k, 0)$ and the set of N -states $\{s | M_L=0, B_L=0, 0 < C_L < k\}$. The initial or terminal N -states may coincide with some of the other elements; the set α has at most $5+k$ elements.

Let us examine the relationship between the new space σ^* and the space σ of $2(N+1)^2$ N -states. Consider the three sets $\{s\}_{top} = \{s | M_L = N\}$, $\{s\}_{diagonal} = \{s | M_L = C_L\}$ and $\{s\}_{bottom} = \{s | M_L = 0\}$ in σ . They correspond to the top line, the diagonal line, and the bottom line respectively of the permissible Z region in σ . Each of these sets has one or more characteristic points that we call *entrance points* and the set of $\{s\}_{bottom}$ has a characteristic point that we call an *exit point*. The entrance point of $\{s\}_{top}$ is the initial N -state of the M & C problem, and the exit point of $\{s\}_{bottom}$ is the terminal N -state of the M & C problem; these are two elements of α . The entrance points of $\{s\}_{middle}$ are the N -states $(N, N, 1)$ and $(N-k+1, N-k+1, 1)$; these are two elements of α (note that $(N, N, 1)$ can be an entrance point of $\{s\}_{top}$ also). The entrance points of $\{s\}_{bottom}$ are $(0, 0, 0)$ $(0, k, 0)$ and the points of the set $\{s | M_L=0, B_L=0, 0 < C_L < k\}$; all of these are elements of α also. The macro-transitions (H_1) and (H_1, J_1) specify two possible ways of reaching an entrance point in $\{s\}_{middle}$ from an entrance point in $\{s\}_{top}$. The macro-transitions (D) , (J_2) , (D, J_2) specify three possible ways of reaching an entrance point in $\{s\}_{bottom}$ from an entrance point in $\{s\}_{middle}$. Finally, the macro-transition (H_2) specifies a way of reaching an exit point of $\{s\}_{bottom}$ from an entrance point in the same set.

We can think of the three sets $\{s\}$ as *easily traversable areas*, where a path for going from one point to another can be found with relative ease. However,

the critical points of the problem occur at the points of transition, the 'narrows', between easily traversable areas. These are represented by the intermediate entrance points. A substantial increase in problem solving power is obtained when such 'narrows' are identified, and when general ways of going from 'narrow' to 'narrow' are developed. Our macro-transitions provide precisely the capability of going from one 'narrow' into an easily traversable area, and then through that area to another 'narrow' that leads to the next easily traversable area or to the desired terminal exit.

The space σ^* is an *abstraction* of the space σ . Formally, a simplest solution to an M&C problem is attainable in σ^* if and only if it is attainable in σ . Furthermore, the minimal path linking two points in σ^* is identical with the minimal path between the same two points in σ . In σ^* attention is focused on a small number of well chosen critical points of σ . By looking for paths between points in σ^* , we solve the problem in at most three 'leaps', and then we can 'fill in' the details with the help of the definitions for the macro-transitions.

The main difficulty in finding an appropriate abstraction for the problem space lies in the discovery of the critical 'narrows' in that space, or more generally, of the topology of easily traversable areas and their connections in the problem space. After the 'narrows' are found, it is possible to build an abstract problem space that is based on them and that has ways of moving among them. It appears significant for the discovery of features in problem space—that lead to a formulation of an abstracted space—to have an appropriate representation of the space. Such is, we feel, the array representation that we have used for σ .

12. SUMMARY AND CONCLUDING COMMENTS

It is reasonable to expect that most 'real life' problems of reasoning about actions will not be formulated at the outset within a formal system. In many cases, the problem will have an initial verbal formulation. If such a problem is to be solved by a computer system, then the system must be able to accept a verbal formulation of the problem, and to convert this formulation into a form that is acceptable to a suitable problem solving subsystem. We have not considered in this paper the linguistic problem of *translating* from the natural language input into an 'internal' machine language that is acceptable to a problem solving program. This problem is receiving considerable attention at present (see Simmons, 1965). However, the question of choosing an 'appropriate' machine language, into which the verbal statement of the problem is to be translated, has received much less attention to date. In this paper, we are taking a first step towards understanding the nature of this question. Our notion of 'appropriateness' here is meant in the sense of suitability with respect to the *efficiency* of the problem solving process. In order to approach such a question of optimal choice of language, it is important to clarify the relationships between the language in which a problem is formulated for a problem solving system and the efficiency of the system. The systems of

production P (introduced in section 2) provide a conceptual framework where such relationships can be studied. The 'internal' formulation of a problem amounts to specifying a system P , i.e. specifying the N -state language, the extended description language, the rules of action, and the two N -states that correspond to the initial and terminal situations between which the problem solving system is to find a solution trajectory. There exists considerable experience at present with computer realizations of problem solvers that work with formulations of problems in systems of production. GPS is an important prototype of such a problem solving system (see Newell, Shaw and Simon, 1960). To each system P there corresponds an N -state space over which the search for solution takes place. A good measure of the difficulty of the problem task is given by the size of the N -state space that must be searched to find a solution. Therefore, given a certain class of problems, we can evaluate the relative merits of languages for representing these problems in systems of productions by comparing the sizes of their associated N -state spaces that must be searched to obtain solutions.

In the specification of description languages for a system of productions where a given problem is to be formulated, the choice of basic elements (the universe U_0) and of basic predicates (properties and relations of the basic elements) is critical. This choice should provide enough expressive power for formulating the rules of action in a manner that reflects all the conditions of the problem. This is always possible if the elements and the predicates are chosen at a low enough, atomic, level; unfortunately, descriptions built of atomic elements have astronomical N -state spaces. Thus, we are confronted with the problem of finding the coarsest possible elements and predicates that can form descriptions that are fine enough for expressing the rules of action in the required detail. This is a difficult problem for people; at present, it is still more difficult for machines. In the M&C problem, we see that the initial formulation F_2 in a system of productions is much poorer than the formulation F_3 where instead of using individuals as elements, the sizes of certain sets of individuals (a much coarser notion) are considered to be the basic elements of the problem universe.

It appears desirable at present that an automatic translator whose task is to convert a verbal statement of a problem about actions to a machine formulation of the problem should have as its target language a language of descriptions that is atomic enough to accept quickly a great variety of problems about actions. The design of such a language seems possible and is now under study. The task of taking a possibly cumbersome system of productions P_1 from the output of such a translator and producing a better system P_2 —in which the search for solution takes place—should then be delegated to the problem solving system. This is in accordance with our general thesis that it is an important function of the problem solver to find the most appropriate representation of his (its) problem. The separation of the initial translation process and the process of finding the most appropriate internal language for a problem

appears to be methodologically desirable at present—given our state of knowledge about problem representations and conversions between them. It is conceivable, however, that the design of these two processes will be combined in the future. Undoubtedly, a unified approach to these two processes will strengthen both.

The rules of action of a system P play the role of the *laws of motion* that govern action sequences in the space of N -states. They are analogous to the differential equations that specify the possible time traces of a physical dynamic system. They are also analogous to the productions of combinatorial systems. Different types of problem conditions are reflected in different *forms* of rules of action. The non-cannibalism conditions of the M & C problem are easily expressible in the form of required derived descriptions for consequence of actions. As in the cases of differential equations and combinatorial systems, it is to be expected that there are classes of forms of rules of action to which there correspond problem spaces with certain special properties, characteristic patterns, etc. The identification and study of such classes would be an important contribution to the theory of problem solving processes. Even though such knowledge may not have direct implications for the design of problem solving systems that attempt to find a solution by intelligent search in a *given* problem space, it is most likely that it will be of great significance for the design of a system that would attempt to discover regularities in a problem space and that would subsequently use them for formulating new spaces where the process of searching for a solution becomes much easier.

An initial improvement in the formulation of the M & C problem came from the recognition that one of the conditions of the problem (non-cannibalism in the boat) is redundant. This permitted the formulation of new actions, as sequences of elementary actions, and it resulted in the effective elimination of many intermediate N -states. Hence, knowledge of the redundancy property permits a shrinkage of N -state space, i.e. an increase in problem solving efficiency. As shown in section 4, the redundancy of the boat condition can be established by deductive reasoning from the rules of action. Such reasoning can be carried out by machine theorem proving processes that are within the present state of the art. However, the process of *looking for* a redundant condition among the conditions of the problem is not a simple deductive process. It is a process of logical minimization. This also could be mechanized without much difficulty at present. The idea of eliminating redundant, irrelevant, conditions in a problem is an old and useful idea in the art of problem solving. It would pay then to have enough logical capabilities in a problem solving system in order to effectively attempt such eliminations.

In the M & C problem, an automatic conversion from the formulation F_2 to F_3 seems possible within the present state of the art. The conversion is based on the elimination of the redundant boat condition, the specification of compound transfer actions as sequences of the previous elementary actions (this is made possible by the previous elimination) and the formulation of new

basic elements for the N -state language; this latter formulation can be guided by the form of the derived descriptions in the rules of action.

In section 6 we have shown that the formulation of the M & C problem in a production system is strongly equivalent to its formulation in a reduction system (which is a theorem proving system). A rule in the system of productions directly corresponds to a move (or a rule of inference) in the reduction system; the search trees are identical in the two systems. The reduction system has the advantage of showing clearly the logic of the attainability relations, as the search for solution evolves.

For each formulation of a problem in a system of productions it is always possible to specify an equivalent formulation in a reduction system. At worst, the search for solution in the reduction system will be identical with the search in the production system. In some cases, where the rules of action are *context free*, it is possible to specify stronger rules of inference in the reduction system, and to obtain as a consequence searches for solution that are faster than in a production system. A context free rule of action has the property that a given subconfiguration of an N -state can go to a specified subconfiguration of the next N -state regardless of the context of these subconfigurations in their respective N -states. In the M & C problem, the rules of action are strongly *context dependent*.

For example, no decision on the transfer of missionaries can be made independently of a decision on the transfer of cannibals or on the position of the boat. Thus, a reduction system cannot give an essential advantage in the M & C problem. An example where a reduction approach has considerable advantage for the solution of a problem that is formulated in a system of productions is the syntactic analysis of context free languages (see Amarel, 1965).

After the language of descriptions of a problem in a system of productions becomes reasonably efficient – as in the formulation F_3 in the M & C case – then the main improvements in problem solving power come from the discovery and exploitation of *useful properties* in the search space. An important property of this type is the *symmetry under time reversal* that we have found in the M & C problem. This property enables us to cut the depth of search for solution in N -state space by a factor of 2 – a significant reduction, hence a significant increase in problem solving efficiency. The symmetry property can be utilized by thinking in terms of a combined development of the search both from the initial N -state ahead in time, and from the terminal N -state back in time. However, only the development from one side is actually carried out. As soon as a search front reaches a point where there are linking possibilities between it and its image, then the search stops and a solution is found. In the present case, the formulation of the problem in a reduction system enables a clear development of the logic of search.

The symmetry property is strongly suggested by observing search graphs of the M & C problem (such as in figure 7.1) and also by examining the array representation of the N -state space. To establish the symmetry property (in

section 8) we have used reasoning that is based on properties of the expressions for the rules of action. Again, such deductive reasoning is mechanizable at present. The mechanization of the more difficult task of *looking for* symmetries of certain type, given appropriate representations of solutions is also within sight. Given a newly discovered symmetry property, its utilization for problem solving requires reasoning *about* the problem solving process at a meta-level. This can be carried out with relative ease if the process is considered from the viewpoint of a reduction procedure and its logic interpretation.

In order to discover useful properties in the N -state space it is very important to have 'appropriate' representations of that space. In the M&C problem, the array representation (introduced in section 9) of N -state space has proved extremely fruitful. People have found the solution of M&C problems much easier when formulated as path finding in the array. Also, it is relatively easy for people to discover the properties that lead to the definition of macro-transitions. Is the 'appropriateness' of our array representation due solely to certain properties of the perceptual and reasoning processes of humans? Would this representation be as appropriate for (some) machine processes of pattern discovery? These remain open questions at present. In general, the problem of *choosing* a representation of N -state space, and of *discovering useful regularities* of solution trajectories in this representation, require much more study. Further exploration of these problems in the context of the 'dance floor' array representation of our M&C problem may provide interesting insights into them.

The definition of macro-transitions enables the formulation of the M&C problem in an extremely powerful system of productions (formulation F_6). The size of the N -state space is drastically reduced and a solution is obtained with practically no search, regardless of the size of the problem (sizes of populations to be transported and boat capacity). Macro-transitions act as well-chosen lemmas in a mathematical system; they summarize knowledge about the possibility of reaching certain critical intermediate points in the search space from some other points. The new N -state space that is based on macro-transitions is an *abstraction* of the previous N -state space. Only certain *critical points* of the lower level space appear in the abstracted space. We can reason in broad lines about the solution—and construct in the process a macro-schedule—by trying to establish a path, made of macro-transitions, that goes through some of these critical points. Once the macro-schedule is built, it is straightforward to obtain a detailed schedule by compiling the routines of action sequences that define the macro-transitions. The idea of finding a small set of points in the search space that are necessary and sufficient for the construction of the solution, is central in our last approach. In discussing the importance of such an approach, Simon (1966) brings the example of the simplex method in linear programming, where only the subspace made of the boundary points of the space of feasible points is searched for a solution.

The evolution of formulations of the M&C problem from its verbal statement to its last formulation in the abstracted subspace of the N -space is accompanied by a continuous and sizable increase in problem solving efficiency. This evolution demonstrates that the choice of appropriate representations is capable of having spectacular effects on problem solving efficiency. The realization of this evolution of formulations requires solutions to the following four types of problems:

- (i) The choice of 'appropriate' basic elements and attributes for the N -state language.
- (ii) The choice of 'appropriate' representations for rules of action and for the N -state space.
- (iii) The discovery of useful properties of the problem that permit a reduction in size of the N -state space. Specifically, the discovery of a redundant condition in the problem, the discovery of symmetry in the problem space, and the discovery of critical points in the problem space that form a useful higher level subspace.
- (iv) The utilization of new knowledge about problem properties in formulating better problem solving procedures.

Given solutions to (i) and (ii), it is conceivable that the approach to the solution of (iii) and (iv) is mechanizable—assuming good capabilities for deductive processing. There is very little knowledge at present about possible mechanizations of (i) and (ii). However, if experience in problems of type (iii) and (iv) is gained, then at least the notions of 'appropriateness' in (i) and (ii) will become clearer.

Acknowledgment

This work was greatly stimulated by discussions with A. Newell and H. A. Simon.

REFERENCES

- Amarel, S. (1965), Problem solving procedures for efficient syntactic analysis, *ACM 20th National Conference*.
- Amarel, S. (1967), An approach to heuristic problem solving and theorem proving in Propositional Calculus, *Computer Science and Systems*. Toronto: University of Toronto Press.
- Bellman, R. & Dreyfus, S. (1962), *Applied Dynamic Programming*. Princeton: Princeton University Press.
- Black, F. (1964), *A Deductive Question Answering System*. Unpublished Doctoral Dissertation, Harvard University.
- Carnap, R. (1958), *Introduction to Symbolic Logic and its Application*. New York: Dover Publications.
- McCarthy, J. (1963), Situations, actions, and causal laws. *Stanford Artificial Intelligent Project, Memo No. 2*.
- Newell, A. (1966), Some examples of problems with several problem spaces, *Seminar Notes*, CIT, Feb. 22.

- Newell, A., Shaw, T. & Simon, H. A. (1960), Report on a General Problem-Solving program for computer, *Proceedings of the International Conference on Information Processing*, pp. 256-64. Paris: UNESCO.
- Simmons, R. F. (1965), Answering English questions by computer: a survey, *Communications of the ACM*, 8.
- Simon, H. A. (1966), On reasoning about actions, CIT # 87, Carnegie Institute of Technology.
- Simon, H. A. & Newell, A. (1961), Computer simulation of human thinking and problem solving. *Datamation*, June-July. 1961.



Descriptions

E.W. Elcock

Computer Research Group
University of Aberdeen

INTRODUCTION

This paper considers some aspects of the problem of describing a certain class of objects as constructs formed from given primitive objects, and of recognizing realizations of a described object in a given environment of primitive objects.

My interest in this problem stems from work done in collaboration with A.M. Murray on the design of programs which can acquire an increasing capability to play board games as a result of the program's own analyses of games which it has played and lost (*see* Elcock and Murray, 1967). The last part of this paper relates the present work to this earlier work.

The simple formal language used for descriptions in what follows reflects this interest in particular board games. The design of a more comprehensive descriptive language (*cf.* Banerji, 1965) is an interesting problem, but will not be pursued in the present paper. For reasons which it is part of the function of the present paper to make clear, it is intended that such language design, together with the further development of the ideas reported here, will take place within the framework of an assertional programming system of the general kind discussed by J.M. Foster in this volume.

The paper is in three parts. Part one introduces descriptions and considers the recognition process. Part two comments on certain features of sequencing in the recognition process. Part three briefly relates the presented view of description and recognition to the earlier work of A.M. Murray and myself on game-playing programs.

1. DESCRIPTIONS AND THE RECOGNITION PROCESS

As mentioned above, objects are to be regarded as constructs from primitive objects and, for the reasons given above, it will be sufficient in what follows to make the following definitions.

A description of a primitive object is a set of names.

Example: the unordered pair of names (p,q) .

The names p and q might be interpreted as names of points, and the description that of a line with end-points p and q .

A description of an object D is a set of names D_n and a set of primitives D_p .

Example: $D_n = (x,y,z)$; $D_p = ((x,y), (y,z), (z,x))$.

With the given interpretation of the primitive (p,q) the described object has the interpretation 'triangle'. Degenerate triangles could be excluded by adding to the description the explicit constraint $x \neq y$; $y \neq z$; $z \neq x$.

In what follows it is a general constraint that no two names of a D_n are the same, with the result that explicit constraints such as that discussed in the example become unnecessary.

The notation $\{D_n; D_p\}$ will be used for a description of an object. The D_p may be regarded as predicating certain primitive relations over names from the 'bound' set of names D_n .

A particular environment E is a set of names E_n , and a set of primitives E_p .

Example: $E_n = (a,b,c,d)$; $E_p = ((a,b), (b,d), (d,c), (c,a), (b,c))$

An interpretation of this E is given in figure 1.

An environment is formally just a description of an object. It is distinguished simply because of the asymmetry of the kind of question we are going to ask, namely:

Given a particular environment E and described objects $D^1, D^2, \dots$ are there any realizations of $D^1, D^2, \dots$ in E ? Alternatively, can $D^1, D^2, \dots$ be regarded as constituent sub-objects of E ?

An interesting digression is to consider the problem of replacing a description

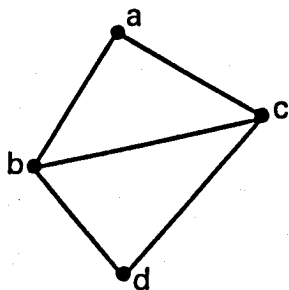


Figure 1

by an equivalent 'simpler' description. Thus, using a reasonably self-explanatory notation: if we have the description

$triangle(x,y,z)$ asserts $(x,y) \& (y,z) \& (z,x)$,

and the description

$environment(a,b,c,d)$ asserts $(a,b) \& (b,d) \& (d,c) \& (b,c)$,

how is the concept 'simpler' to be defined and, given an appropriate definition, how do we recognize

$environment(a,b,c,d)$ asserts $triangle(a,b,c) \& triangle(b,c,d)$

as an equivalent 'simpler' description of the described object environment?

In a trivial sense, since we expect the statement of a description to allow the construction of an algorithm for recognizing all realizations of the described object in any given environment, the equivalence of descriptions is a sub-problem of the equivalence of algorithms. Nevertheless, over a particular universe of descriptions of interest, it may be possible to take advantage of the nature of the descriptions to formulate a complete procedure for determining equivalence.

Returning to the problem of recognition of realizations of a described object D in a given environment E : a search for a realization of D in E is taken to be an attempt to identify, for each primitive of D_p , the bound names referenced by the primitive with the names referenced by a primitive of E_p , in a way which preserves the uniqueness of names over the description set of names D_n , and over the environment set of names E_n . Successful identification of the complete set of primitives of D_p constitutes recognition of a particular realization of D in E .

For the D and E given in the examples above, an actual process of recognition can be described very informally as follows:

Start with the job 'realize D in E '.

We select a primitive from D_p , (x,y) say, and a primitive from E_p , (a,b) say.

We identify the bound names x and y in (x,y) with the names a and b in (a,b) and assert:

(1) $((x \text{ is } a) \& (y \text{ is } b))$ or $((x \text{ is } b) \& (y \text{ is } a))$

in the context of the unrealized residual description $\{(x,y,z):((y,z), (z,x))\}$. This identification results in the two partial realizations:

$D^1 = \{(a,b,z):((b,z), (z,a))\}$,

$D^2 = \{(b,a,z):((a,z), (z,b))\}$,

and the end result of this stage is the set of jobs:

(realize D^1 in E);

(realize D^2 in E);

(realize D in E excluding the identification (x,y) in D_p with (a,b) in E_p).

To carry on we select one of these jobs and repeat the process. Suppose we select (realize D^1 in E) and suppose we try identifying (b,z) in D^1_p with (c,d) in E_p . This would give rise to the assertions

$$((b \text{ is } c) \ \& \ (z \text{ is } d)) \quad \text{or} \quad ((b \text{ is } d) \ \& \ (z \text{ is } c))$$

both alternatives making two names in E_n the same and therefore terminating, unsuccessfully, this particular branch of the recognition process.

On the other hand, the identification of (b,z) in D^1_p with (b,c) in E_p would give

$$((b \text{ is } b) \ \& \ (z \text{ is } c)) \quad \text{or} \quad ((b \text{ is } c) \ \& \ (z \text{ is } b)).$$

The second of these would terminate either for the reason that $(z \text{ is } b)$ makes two names in the associated D_n the same or because $(b \text{ is } c)$ makes two names in E_n the same.

The first alternative however succeeds and generates the partial realization

$$D^3 = \{(a,b,c):(c,a)\}$$

and the job (realize D^3 in E) is added to the set of alternative jobs.

2. SOME COMMENTS

(i) There are clear links between this kind of recognition process and some of the work of I.E. Sutherland and others in Computer Graphics. In particular there are links between a description as formulated here and Sutherland's concept of a constraint (*see* Sutherland, 1963). The essential difference would seem to be that Sutherland applies constraints only to 'appropriate' objects (e.g. the constraint 'square' to an object $((a,b),(b,c),(c,d),(d,a))$). Here one is interested in an arbitrary given object and asks for all sub-objects in the given object which meet the constraint (satisfy the description). Nevertheless, the following two comments are as relevant to Sutherland's concept of applying a constraint as to the present problem.

(ii) It should be clear that in the recognition process as described, the particular order in which jobs are selected from the current set of alternative jobs is totally irrelevant to the end result.

(iii) The particular sequencing can be, and usually is *very* relevant to the number of partial realizations (alternative jobs) generated and hence to the total amount of work done.

Thus, in the context of the E used above let

$$D = \{(w,x,y,z):((w,x),(x,y),(y,z),(z,w)))\}$$

with the interpretation 'quadrilateral'.

If the recognition process begins by setting up the partial realizations obtained by identifying (w,x) with successive primitives from E_p ; and if the next stage is to set up for each of these partial realizations the partial

realizations obtained by identifying (y,z) with successive primitives from E_n , then a great deal of work is going to be done which could have been avoided by a different sequencing.

In a situation like this one would like to be able to exploit the irrelevance of sequencing to the end result, without necessarily incurring penalties of inefficient processing. In stating the problem one would like not to be concerned with sequencing as a primary responsibility at all, but merely give the set of assertions which constitute the description, the set which constitutes the environment, and a generator of the required set of assertions of the kind

$$((a,b) \text{ is } (x,y))$$

(where *is* is used with the meaning 'can be identified with') to initiate the process of recognition.

The selection of which assertion to consider next might be left partially or entirely to the machinery which administers the assertional system by supplying or not supplying hints in the form of either an invariant 'badness' associated with particular assertions, or a dynamic 'badness' computable in the context of the current state of the assertion machine. The 'badness' in general would be a measure of the expected cost in time and space of considering the assertion (*cf.* Burstall, in chapter 22 of this volume).

In his paper in this volume J. M. Foster discusses some aspects of a limited assertional machine. It is our intention to develop these ideas further.

3. RELATION TO PREVIOUS WORK

The present work was done in parallel with that of J. M. Foster, and the program which implements the recognition process of §1 is a special implementation which embodies only some of the ideas of assertional scheme. Nevertheless, even in this restricted form it has turned out to be of considerable interest in relation to earlier work by A. M. Murray and myself on the design of programs to play Go-Moku.

Go-Moku is one of a class of games which can be described as follows: There is a set of points. Two players take turns to play at an unplayed point. There is a set of n -tuples of points such that the first player to play at all points of an n -tuple of the set wins.

Consider the trivial game O's and X's (Tic-Tac-Toe). The empty board can be considered as a realization of the environment

$$E = (a,b,c,d,e,f,g,h,i) : (a,b,c), (d,e,f), (g,h,i), (a,d,g), \\ (b,e,h), (c,f,i), (a,e,i), (c,e,g)$$

a diagram of eight lines each with three constituent points (figure 2).

Interpret 'own play' as meaning 'delete played point', and 'opponent's play' as meaning 'delete all primitives of which the played point is a constituent'.

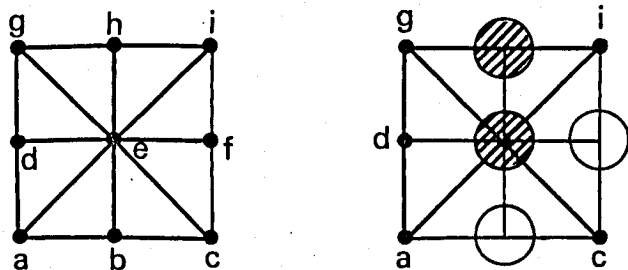


Figure 2

Suppose we have a description

$$D = (x, y, z) : (x, y), (x, z)$$

of a class of critical board situations. Then, after own play e then f then h then b , the environment is now

$$E = \{(a, c, d, f, g) : (g, i), (a, d, g), (c, g), (a, i)\}$$

and a request for realizations of D in E will produce all variants of winning sequences of play starting with own play at the realization of x in a realization of D in E (i.e. g or i).

O's and X's is not a very exciting game in itself. However, the recognition and board updating machinery used above demand only an E and D 's of the general form discussed in §1.

To switch from manipulating particular line point diagrams to playing O's and X's; cubic O's and X's; squares; hex; Go-Moku; indeed, any game of the class stated only involves the trivial matter of setting up the requisite E with the appropriate E_p consisting of the set of point sets which constitute the primitive winning patterns.

In papers published in earlier volumes of *Machine Intelligence*, A.M. Murray and I have discussed methods by which descriptions of classes of critical board situations could be automatically acquired by a game-playing program as a result of its analyses by the program of games played and lost by the program. This automatic acquisition of descriptions can also be made to depend only on the particular structure of D 's and E 's discussed here.

The description and recognition machinery presented in this paper, together with earlier work, therefore provide the ingredients for the design of a single program which can in effect accept a definition of any game in the class of game mentioned at the beginning of this section, and which can acquire the ability to play the game as well as it can be taught to play.

REFERENCES

- Banerji, R.B. (1965), Towards a formal language for describing object classes. *Information System Sciences*. New York: Spartan Books.
- Elcock, E.W. & Murray, A.M. (1967), Experiments with a learning component in a Go-Moku playing program. *Machine Intelligence 1*, pp. 87-103 (eds Collins, N.L. and Michie, D.). Edinburgh: Oliver and Boyd.
- Elcock, E.W. & Murray, A.M. (1968), Automatic description and recognition of board patterns. *Machine Intelligence 2*, pp. 75-88 (eds Dale, E. and Michie, D.). Edinburgh: Oliver and Boyd.
- Sutherland, I.E. (1963), Sketchpad. *MIT technical report No. 296*. Cambridge, Mass.: MIT Press.

Kalah on Atlas

A. G. Bell

Atlas Computer Laboratory
Chilton, Berkshire

I INTRODUCTION

This is a report on work done with the S.R.C. Atlas Computer at Chilton. The original intention was to demonstrate the on-line typewriter to visitors *via* a simple system which reacted to the user, in this case by refusing to be beaten twice in the same way at the game of Kalah.

The mechanism to achieve this is a memory, built up from information obtained in previous games, which is stored on magnetic tape. The program was designed to keep the size of this memory to small proportions by implementing two mechanisms the author believes to be commonly used by humans when solving problems. The two mechanisms are:

1. ignoring irrelevant information in the sense that, although it exists, it is highly probable that its precise structure or properties cannot alter the relevant information or *characteristics* of the problem being considered, and
2. accepting positions close to a solution or win, providing the opponent is further from a win.

Some of the difficulties of testing these ideas in practice are discussed and suggestions are made on how to overcome them, in particular with the game of solo whist.

II KALAH - THE RULES OF THE GAME

Kalah is an extremely ancient game, said to have originated in the Middle East. All that is required to play are 14 holes scooped in sand and the requisite number of pebbles. These pebbles need no distinguishing marks because it is the position of a pebble, or counter, rather than any intrinsic property, which denotes its possible move and ownership.

The more modern board and version of the game is given in figure 1. Each player controls a row of round *pits* on his side and the capsule-shaped bowl at his right called his *kalah*. The object of the game is to get the larger number of counters (playing pieces) in one's own *kalah*.

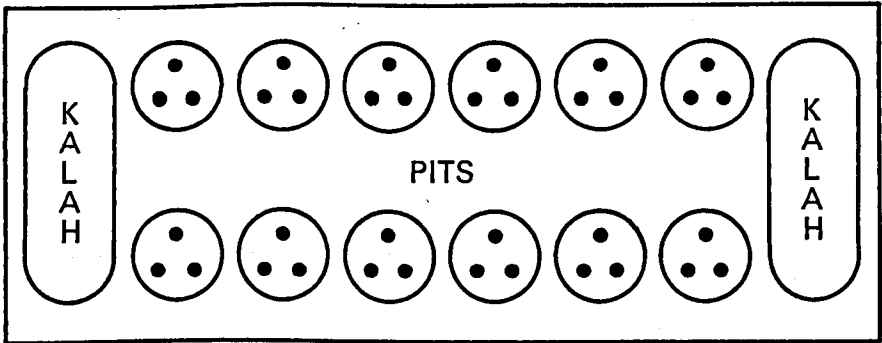


Figure 1

The number of counters used depends upon the time available and age of the players. For a short game and youthful players three counters are placed in each *pit* as shown. Six in a *pit* is generally agreed to make the most interesting game.

Players alternate in moving. Each player empties any one of his *pits* deemed advantageous, and, leaving it empty, distributes one counter into each *pit*, moving in an anti-clockwise direction. If there are enough counters to reach beyond his own *kalah* they are distributed one by one into the *pits* on the opposite side and then belong to the other player. The only place ever skipped is the opponent's *kalah*. Once in a *kalah*, the counters remain there until the end of the game.

The method of play, by distributing stepwise to the right, is subject to two simple rules:

1. If the last counter lands in your own *kalah*, you have another turn. By planning to have the right number of counters in two or more *pits* it is possible to have several turns in succession.
2. If the last counter lands in an empty *pit* on your own side opposite a non-empty opponent's *pit*, you capture all the counters in that *pit*, and place them, together with the one making the capture, in your own *kalah*. A capture ends your turn.

In reaching an empty *pit* on your own side, it makes no difference whether you have moved a single counter one space, or distributed all around the board and back to your own side. Indeed, if a *pit* contains 13 counters it is guaranteed to capture because the last piece goes back into the *pit* just vacated and the opposite *pit* must contain at least one piece.

The game ends when all the *pits* on one side are empty. The first player out usually loses because he receives none of the counters left in *pits* on the other side. They go into the *kalah* on the side of the player who has been able to save them. A good player will force the other to distribute and play out. On

the other hand, it is difficult to hold on to a lot of counters against a skilful player who tries to force their distribution.

III ILLUSTRATIVE GAME

This game was actually played against the program. The program is playing the top row and the author the bottom row (*see figure 2*).

The notation of the moves is as follows. The pits on the machine's side of the board are numbered M1 to M6 and on the player's side P1 to P6 (*see figure 2.1*). The pit chosen to be emptied is italicized and identified in the right hand column, and when a capture occurs the pit emptied is denoted in brackets.

A brief commentary on the game follows:

1. The author's opening move is the one claimed by Russell (1964) to be the best.
2. The program has 10 possible replies but has already rejected one from experience; this is its second attempt.
3. Unfortunately the author does not know how to stay in a winning position because this move, analysed later, is a loser.
4. The program takes full advantage of the position, i.e. capturing by emptying M6 in 4.5, otherwise it will lose.
5. The author threatens to capture M5.
6. The program therefore empties it and in return threatens to capture P5 for a crushing win. It has also calculated that it has a good chance to win or draw from this position irrespective of opponents play.
7. Pathetic.
8. The program continues on its winning course.
9. Resigns.

IV STRUCTURE OF THE PROGRAM

The program has, basically, 4 mechanisms. They are:

1. List legal moves.
2. Mini-maxing with simple evaluation.
3. Memory plus back-tracking to speed up accumulation of information.
4. Compression mechanisms to curtail size of accumulated information in memory.

IV.1 List legal moves

Because of the simplicity of the rules, the routine produced to calculate all the possible legal moves from a given position comprises about 100 Atlas Basic Instructions and runs extremely fast. This is an important property of the program because this routine is used a great deal in the look-ahead and back-track analysis, especially in the opening moves where it is unlikely to recognize any of the positions.

The program does not print its own continuation positions because they

MACHINE LEARNING AND HEURISTIC PROGRAMMING

		M1	M2	M3	M4	M5	M6		
1		3	3	3	3	3	3		
	MK0							0 PK	P2
		3	3	3	3	3	3		
		P6	P5	P4	P3	P2	P1		
2		3	3	3	3	3	4		
	0							1	M1
		3	3	3	3	0	4		
3		0	3	3	3	3	4		
	1							1	P1?
		4	4	3	3	0	4		
4.1		0	3	3	4	4	5		
	1							2	M3
		4	4	3	3	0	0		
		continuation							
4.2		1	4	0	4	4	5		
	2							2	M1
		4	4	3	3	0	0		
		continuation							
4.3		0	4	0	4	4	5		
	3							2	M4
		4	4	3	3	0	0		
		continuation							
4.4		1	5	1	0	4	5		M1
	4							2	
		4	4	3	3	0	0		
		continuation							
4.5		0	5	1	0	4	5		
	5							2	M6
		(4)	4	3	3	0	0		c(P6)
		0	6	2	1	5	0		
5.1	10							2	P3
		0	4	3	3	0	0		
		continuation							
		0	6	2	1	5	0		
5.2	10							3	P1
		0	4	3	0	1	1		
		continuation							
5.3		0	6	2	1	5	0		
	10							4	P4
		0	4	3	0	1	0		
6.1		0	6	2	1	5	0		
	10							4	M5
		0	4	0	1	2	1		
		continuation							
6.2		1	7	3	2	0	0		M1
	11							4	
		0	4	0	1	2	1		
		continuation							
6.3		0	7	3	2	0	0		M2
	12							4	(claims win or draw)
		0	4	0	1	2	1		
7.1		1	0	3	2	0	0		
	13							4	P1
		1	5	1	2	3	1		
		continuation							
7.2		1	0	3	2	0	0		
	13							5	P5
		1	5	1	2	3	0		
		continuation							
7.3		1	0	3	2	0	0		
	13							6	P6
		1	0	2	3	4	1		
		1	0	3	2	0	0		

Figure 2

cannot be evaluated. When the last piece does land in its own *kalah* it buffers that position and starts from scratch with the next pit. Then, having investigated all the initial moves from the six pits, it recalls the continuation positions and either exhausts them or produces more continuation moves which go back into the buffer which is, in effect, a first in-last out stack. The result of this is that positions which are reached through many continuation moves will be the first to be investigated by the mini-max routine, and the cut off facility, in conjunction with the simple evaluation of position, (described in Section IV.2) should be enhanced.

IV.2 Mini-maxing with simple evaluation

Mini-maxing is a standard tool for playing full information games on machines. A good description of this tool is given by Michie (1966). In brief, the *kalah* routine looks $1\frac{1}{2}$ moves (or 3 plys) ahead by generating all the positions that can exist in that number of moves and ear-marks those positions it calculates to be to its own best advantage. However, as the opponent is assumed to evaluate positions in the same way and also has at least one intervening move, some of the positions are discarded as unattainable.

In order to prevent the evaluation of every position produced a simple cut off mechanism is incorporated. Consider figure 3. The routine takes the highest

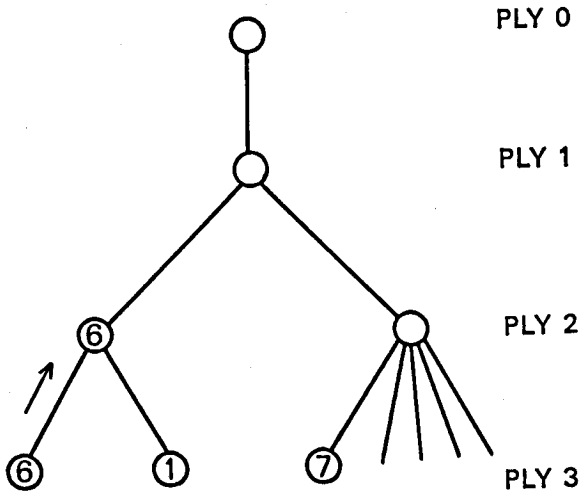


Figure 3. Positions are evaluated at PLY 3

value of the groups (in level 3) which are connected to the nodes in the second row, i.e. it maximizes. The highest of the left hand group of (6, 1) is 6 and this has been entered *via* the arrow. On investigation of the second group the routine finds that the first value it obtains (i.e. 7) is greater than the lowest value on level 2 (i.e. 6), and therefore can ignore all the remaining positions.

This mechanism works at any level, but the test of greater than or less than depends on whether the routine is at a maximizing level or a minimizing level.

The simple function chosen to evaluate positions at the 3rd level is to maximize $(Mk) - (Pk)$, i.e. to get more counters in the machine's *kalah* than the opponent can get in his own or, failing that, to keep the difference as low as possible. Actually this is a rather bad motivation because good players utilize a waiting game and the memory will have to over-ride some of the moves chosen eventually. Another weak feature of the program is that it will always accept the first victory it calculates or recalls. Therefore, the first move which will guarantee to obtain 19 counters in its own *kalah* is the one it will take and, indeed, is quite reasonable. Unfortunately, as its memory builds up, it tends to assess moves much earlier in the game and, instead of a brilliant, immediately crushing move, it will take the longest, dreariest path during which, as will be seen, the opponent has the chance to destroy the machine's illusion of being in an absolute winning position.

IV.3 Memory plus back track analysis

The information stored in the memory is a description of positions encountered which have a definite outcome, i.e. they are winning or losing positions. Before the first game the memory is empty and the moves chosen come entirely from the simple evaluation function with mini-maxing. Eventually the program realizes that it will either win or lose in the next $1\frac{1}{2}$ moves. If it wins it only records its present position and tags it as a winner, i.e. a good position to try for, if given the chance, in a later game. If, however, it loses, it attempts

(a) to find out why, and

(b) to build up its store of winning/losing positions.

Both these aims are attempted by back-track analysis. This is simply a short investigation of the region of the game tree in which the program finds itself. The argument is that, now it has eliminated a previously chosen move because it inevitably lost, it can consider more deeply the alternatives to that move and possibly even find a win. Also, by looking at positions closely related to its losing position, more winning/losing positions should be produced which are worth storing. By recording the course of the game the program can return to the position one move before it inevitably lost and can assess that losing position from its memory. Hence it will choose differently. A count is set to evaluate 100 nodes in that region of the game tree, and, whenever all the nodes have been evaluated at the lower level, the previous position played in the actual game is recalled and the back-track analysis continues. This results in the program, having terminated this activity, never playing the same game twice and also being unpredictable at what point in the game it will diverge from its previous play. This rote learning with associated back-up scores is well described by Samuel (1960). Figure 4 shows clearly that once the node $\otimes$ has been evaluated and the same board position is found in a later game, then its score has, in effect, already been backed up by 3 levels, and if it

kalah by 2^8 . This then points into a table of 1280 entries (a cut off is made if $mk > 20$). It still entails a linear scan to be made but the entry could be improved by using the contents of the first one or two pits next to the *kalah* and hashing them.

Positions are stored, then, in the manner described and when the program is considering other positions it similarly compacts them, calculates the entry point into the table, and compares the position with those in the memory which have the same *kalah* contents, ensuring that all the pits' contents also agree. If a position is recognized, then the program will either accept it if it is a winner or reject it if it is a loser. The memory is split into ten parts each with 1280 entries and the program checks which of the ten replies the player has chosen and, in order to confine the game, reads that part of the total memory down. This, incidentally, is not a good thing to do, as the author attempts to explain later, nor is it wise to assume that 1280 entries is sufficient to hold the required information. These features are described because, once implemented and the program discovered that one of the ten replies was a loser, the aim of the project developed from playing an unbeatable game of *kalah* to studying ideas and techniques which may be applicable to a range of problems.

IV.4 Memory compression mechanisms

In the introduction, two memory compression mechanisms are mentioned. It is worth adding that the simple evaluation of position function mentioned in section IV.2 also restricts the size of the memory in the sense that there are positions from which the machine could choose many moves which still guarantee a win. The evaluation function tends to take the shortest route and hence many plodding paths and positions to a win are ignored. However, the two mechanisms considered by the author to be more akin to the way humans treat games of this type are called *characteristics* and *parameters*.

Characteristics. Consider figure 5.1. The machine is playing the top row and is presented with this position which is a winner (figure 5.2) via the moves M1 (continue) M2 (continue) M1. The machine will therefore store this position as a winner but it does not need to store the contents of every pit because it is immaterial to the winning sequence what the other pits contain (they may contain any legal permutation of the remaining counters in the pits which have not changed their contents). Similarly in figure 5.4 the winning sequence of moves is M6-M1-M3-M1-M2-M1-M4 (capture). The quick way to extract the required information is by non-equivalencing the compressed position before the winning move with the compressed position after the winning move. Non-equivalencing two equal values gives the answer zero; two dissimilar values give a non-zero answer. Thus, non-equivalencing figures 5.1 and 5.2 gives the answer in figure 5.3 of which pits are relevant or *characteristic* of the win. These are M1 and M2 plus the *kalah* (which must have a minimum value of 16).

Unfortunately this is not sufficient in every case, for if this operation is

5.1	16	1	2	0	0	0	0	16
		1	0	0	0	0	0	
5.2	19	0	0	0	0	0	0	16
		1	0	0	0	0	0	
5.3	16	1	2	x	x	x	x	x
		x	x	x	x	x	x	
5.4	7	0	0	2	0	0	6	16
		0	0	5	0	0	0	
5.5	19	0	0	0	0	1	0	16
		0	0	0	0	0	0	
5.6	7	x	x	2	x	0	6	x
		x	x	5	x	x	x	
5.7	7	0	0	2	0	0	6	x
		x	x	5	x	x	x	

Figure 5

carried out on figures 5.4 and 5.5 then the result shown in figure 5.6 is incorrect, i.e. 6 counters in M6, 0 in M5, 2 in M3, a minimum value of 7 in the *kalah* and at least 5 counters in the pit opposite M3. The following additional statement must therefore be included: 'Any pit to the right of an altered pit on my side may also be relevant to the winning position and consequently must be stored'. This results in the position given in figure 5.7.

The extraction of relevant features always guarantees that the contents of at least 5 of the opponent's pits plus his *kalah* will be ignored plus the fact that, at the most, two pits can be assigned minimum values for their contents.

The author emphasizes that this extraction mechanism is heavily dependent on the features of the game of *kalah*. In a more general framework one should consider removing pieces from the board until it can be shown that their presence is essential to the win.

Parameters. The second mechanism, called *parameters*, is a misnomer, but the principle involved is simple. It is akin to the queen being captured in chess for no loss, i.e. one of the players gets so far ahead in the game that the opponent will retire before reaching an actual losing position.

The program records all the previous positions in a game. When it wins or loses, it scans the contents of each *kalah* in all the previous moves, and places

M_k	P_k
19	18
18	18
17	9
16	4
15	2
14	2
13	1
12	1
11	0
10	0

Figure 6. Relative values of *kalahs* for pseudo winning-losing positions (rounded to nearest integer)

a 90 per cent bias on the results. An example (using an actual value achieved by the program) is that if the program can obtain 17 counters in its *kalah* and its opponent has only 9, then 90 per cent of the time (or better) the game has been won from that *kalah's* relation. We define the relation to be nearly as good as a win, and it will therefore be stored in the memory. The actual relative values produced in the machine are given in figure 6. Due to an error in the program, it classes draws to be as good as wins, but to correct this delusion would entail altering the program and destroying the memory already accumulated. This mechanism does not conflict with the *characteristics* mechanism and it reduces the size of the tree to be searched. We again note that the content of the opponent's *kalah* is still not relevant information to be stored in the memory. The position given in figure 7 has good prospects providing the

	1	2	0	0	1	0	
13							4
	6	4	2	0	2	1	

Figure 7

opponent has 4 or less counters in his own *kalah*; otherwise it is assessed by the evaluation function. In the example given, the program would accept the position at either of its maximizing levels as equivalent to a win despite the fact that the opponent has an intervening move (even if the position is reached) before the certain end of the game. In fact, the example is 'cooked' and the opponent has a crushing set of 15 continuation moves to a certain win. This will modify the program's opinion that 16 against 4 is a successful ratio.

It is now obvious that the decision to split the memory into ten parts is a bad thing, for the program has already stored positions which will be useful when it comes to play the other replies. One solution to this problem is to tell the program the correct moves for the opening game, to let the evaluation function play the middle game, and to rely on the *characteristics/parameters* memory to play the end game.

V RESULTS

In reply to the best opening move for 3-in-a-pit *kalah* (figure 2.1), the program initially chose the position shown in figure 8, i.e. M3-M2. The total number

	5	0	0	3	3	4	
2							1
	4	4	3	3	0	4	

Figure 8

of possible positions which can be reached from this configuration is over 2000. With the feeble goal of the evaluation function, many of the paths can be ignored because the program tends to stay on the path to the quickest win.

The memory generated without *characteristics* or *parameters* numbered just over 400 positions in order to prove that this reply will inevitably lose. To build up this memory quickly entailed the author playing the program, beating it, checking how much of the game had been analysed by the back-track analysis, and then playing the program through moves it had still not analysed. Thus the program learnt quickly and this procedure could be used off-line.

When the *characteristics* mechanism was added the memory was only reduced to about 300 entries. This seems disappointing until it is remembered that the positions being recorded will be useful when replies to other opening moves are studied—that is, the program is, to a certain extent, analysing the end game of *kalah*.

Finally, when the *parameters* have been calculated, the memory drops to about 150 entries but the program becomes more likely to claim a win, and then to lose. Surprisingly, people seem to enjoy this aspect of the program because if it thinks it will win, it says so, and, if a good waiting game is then played, the program can lose badly.

Having found that the position in figure 8 is a losing response the program next chose the position in figure 2.2 (the illustrative game) and trouble was encountered because the author did not know the correct response to this move. Consequently the memory began to build up to the danger point of the program believing that 9 times out of 10 it will win from this position which, of course, affects the way it plays that part of the game it has already solved. The unfortunate state of affairs has now been reached where the program, having learnt to play against a good opponent, now begins to deteriorate against a bad opponent. In fact, in demonstrations to visitors, the memory built up is not retained because the program can and does win from appallingly bad positions which may change the *parameters* drastically.

VI OTHER GAMES

The work described in this paper would seem to be applicable to other games, in particular that of solo whist. This card game is suitable because, quite often, many of the cards held by each of the four players are relatively unimportant and it would appear that the compression mechanisms may be effective in this game (which differs markedly from *kalah* in that it is not a full information game). Another advantage of teaching a program to call and play solo whist is that it is easy to write a program to defend calls or contracts by giving full information to the program which defends the contract, i.e. the program knows where the voids are, knows the boss cards and if a finesse is worth while, etc. The trick is to have the program which has to learn the game play against this simple but powerful game teacher. This should save a great deal of time involved in humans having to play the program, or inspect its memory, in order to teach it what configurations of cards are worth calling contracts on and in what order to play the cards. This project has been started.

REFERENCES

- Hopgood, F. R. A. (1966), Hash tables. *A.C.T.P. summer school notes*.
- Michie, D. (1966), Game-playing and game-learning automata. *Advances in programming and non-numerical computation* (ed. Fox, L.) London: Pergamon Press.
- Russell, R. (1964) Kalah—the game and program. *Stanford artificial intelligence project, Memo. No. 22*.
- Samuel, A. L. (1960), Programming computers to play games. *Advances in computers 1*. New York: Academic Press.

Experiments with a Pleasure-seeking Automaton

J. E. Doran

Department of Machine Intelligence and Perception
University of Edinburgh

INTRODUCTION

Attempts to write 'intelligent' computer programs have commonly involved the choice for attack of some particular aspect of intelligent behaviour, together with the choice of some relevant task, or range of tasks, which the program must perform. The emphasis is sometimes on the generality of the program's ability, sometimes on the importance of the particular task which it can perform. Well-known examples of such programs are Newell, Shaw, and Simon's General Problem Solver (1959; *see also* Ernst and Newell, 1967), which is applicable to a wide range of simple problems, Samuel's checker (draughts) playing program (1959, 1967), and the program written by Evans (1964), which solves geometric analogy problems.

However, there is another approach to the goal of machine intelligence which stresses the relationship of an organism to its environment and which sets out from the start to understand what is involved in this relationship.

Long ago Grey Walter (1953) experimented with mechanical 'tortoises' which could range over the floor in a lifelike manner. Toda (1962), in a whimsical and illuminating paper, has discussed the problems facing an automaton in a simple artificial environment. Friedman (1967), a psychologist, has described a computer simulation of instinctive behaviour involving an automaton equipped with sensory and motor systems. Andrae and Gaines (Andrae, 1964), in their STELLA project, have considered the design of an automaton faced with general control tasks. Sandewall (1967) has gone deeply into an automaton/environment relationship with a rather more formal approach. This list is far from complete. In particular, robots of various kinds are under construction at a number of research centres, notably at the Stanford Research Institute (Nilsson and Raphael, 1967).

The reader may find it helpful to meditate on the situation of, say, a rat in a cage, as seen by the rat. I hope the reader will agree that the animal perceives

its rather simple environment in a variety of ways, that it remembers, learns, predicts, and acts towards goals which ultimately derive from basic necessities such as food, sleep, and the avoidance of pain. Can one write a computer program which, however primitively, simulates the rat and its surroundings and which demonstrates the simulated rat displaying such rudimentary intelligence as it is reasonable to require?

In attempting to write such programs I suggest that the ultimate objective is a classification of possible environments for the automaton (e.g. the simulated rat) *from the standpoint of the automaton*, together with an optimal automaton design in some non-trivial and useful sense for each class of environment. The approach described in this paper involves setting up a fairly 'natural', if simple, environment in the hope that the automaton design ultimately achieved by a combination of insight and trial and error experimentation will not only perform well in that environment, but will also display acceptable marks of intelligence.

THE AUTOMATON AND ITS ENVIRONMENT

I shall now try to specify a little more precisely the various concepts at issue, with the objective of constructing a helpful frame of reference. First of all I wish to distinguish three things:

- (a) the running *computer program*, together with associated data, which mimics, however primitively, a portion of the real world. At this level the automaton is only arbitrarily distinguished from its environment;
- (b) the 'objective' view of the automaton in its environment. This corresponds to looking at a rat in a cage from outside the cage, and implies a separate observer and viewpoint. I shall refer to the automaton's *objective environment*, that is, its surroundings as we see them;
- (c) the environment as perceived by the automaton, corresponding to the rat's perception of its surroundings. This paper is primarily concerned with the automaton's efforts to understand and control this *subjective environment*. The reader must bear this last distinction in mind throughout.

The subjective environment involves a sequence of (subjective environment) *states*, each of which is the total possible perception by the automaton at any instant. There is a (subjective environment) *transition rule*, unknown to the automaton, which gives the next state in the sequence, given the history of the system including the automaton's *actions*. This rule will often be conveniently expressed in terms of the objective environment. Each of the actions available to the automaton may be applied at any time. They are distinguishable but otherwise unstructured. The following points are important.

1. 'time' means time as measured by a clock within the running program ((a) above);

2. the use of a sequence of states is a convenient approximation to one continuously varying state;
3. states will typically be very complex including (from the viewpoint (b) above) perceptions of the automaton's own internal functioning;
4. the actions will in general enable the automaton to achieve some measure of control over its future.

To motivate the automaton we associate with a state some idea of its *desirability*. Suppose that certain variables appearing in the state have bounds within which their values should lie. The desirability of a state then refers to the extent to which these constraints are met. The automaton must act so as to maximize the desirability of the states it encounters. If the desirability may be represented by an integer, then the automaton's motivation may be made more precise, if a little artificial, by assigning to it a fixed, known, lifetime and requiring it to maximize the mean desirability of its states over that lifetime. Notice that the desirability function cannot be varied by the automaton.

The automaton performs information processing operations needed to decide which actions to select and when. It is natural to suppose that processing proceeds at a *finite speed* (program time), and that information *storage capacity* is limited. These constraints form part of the design problem. The rate of processing achieved by the automaton relative to the rate of change of its subjective environment and to the lifetime available to it is very important.

The design of the automaton must take into account the anticipated properties of its subjective environment. The reader may find it illuminating to imagine himself (the automaton) before a screen on which is displayed a complex pattern which changes from time to time (sequence of states). He has access to a row of buttons (actions) any of which he can press at any time, and he has a given scale of preference for the patterns which occur. He has a limited supply of pencil and paper. His task is so to press buttons that over a given period of time the preference level is kept as high as possible. Naturally the reader's strategy will vary according to the information he is given relating button-pressing to the patterns displayed.

I shall now take as an example a particularly simple, if artificial, class of subjective environments which will serve to introduce some further concepts.

SUBJECTIVE ENVIRONMENT GRAPHS

Let the reader again imagine himself before the screen introduced in the last section, but now given the following additional information:

1. that no information is contained in any similarity between patterns shown. That is, patterns may only usefully be said to be identical or non-identical;
2. that the pattern displayed changes only when a button is pressed;
3. that the effect of pressing a particular button when a particular pattern is displayed is always the same, and that the effect is always immediate.

The reader may care to consider the strategy he would adopt.

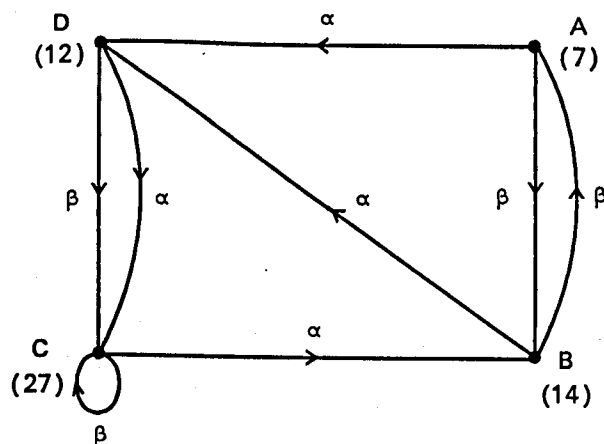


Figure 1. An environment graph—one way of specifying a simple subjective environment transition rule. The nodes correspond to states, the labelled arcs to action transitions, and the figures give the corresponding desirabilities

Figure 1 represents this situation in abstract form. The transitions of the subjective environment are described by a graph with 'labelled' arcs, the nodes of the graph representing states, and the arcs with a particular label representing the consequences of a particular action. Thus if the automaton is 'at' state D of figure 1, and applies the action α , then it immediately finds itself 'at' state c. The figures give the desirability of each state. How can the automaton be designed so that it performs well in any subjective environment of this type, without it ever knowing which particular graph it is faced with?

At any instant the automaton will be in some state, and we may assume that its history is accessible. Thus it might have stored that its history is as shown in figure 2 (a). If so we can now distinguish *three* relevant graphs. These are:

1. the *subjective environment graph* (figure 1),
2. the *stored graph* which is that portion of the subjective environment graph which the automaton has stored in its memory as a result of its experience (figure 2 (b)), and
3. the *option graph* which is that fragment of the stored graph which the automaton 'knows' how to reach (figure 2(c)).

The automaton has, I suggest, a broad choice between *exploration* and *exploitation*. Thus it can either decide to 'move' to some state in its option graph, and once there try out a new action, *or* it can select some suitably desirable state in its option graph, move to it, and do no more. In the first case the *goal* might be B and the *plan* $\beta \alpha \beta$, and in the second the goal might be C and the plan β^* . Which of these two types of plan the automaton should adopt will depend upon (1) the expected mean desirability to be achieved by

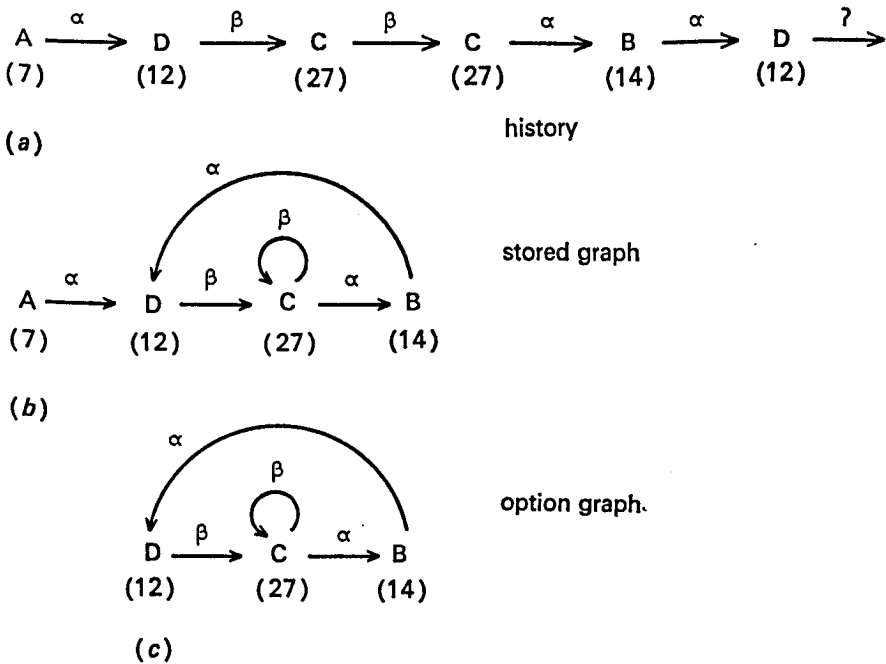


Figure 2. A transition history based on the graph of figure 1, together with the corresponding stored and option graphs (see text)

exploration at the best exploration goal, and (2) the desirability of the best exploitation goal.

The reader should note that I am using the word 'plan' to refer to a situation where a course of action is determined upon in advance, and then carried through without further 'thought'. For this class of subjective environments at least, there is clearly no point in reconsidering a plan part way through its implementation.

An important complication is introduced by the time that the automaton must take to select a plan, which has the consequence that the selection process will involve a succession of steadily more promising plans (of either type) until the time cost of further improvement tilts the balance in favour of actual implementation of the current best plan. The time cost of decision making is more pressing the lower the desirability of the current state.

After an exploratory action the automaton may either be forced to explore again, since it has encountered a quite new state, or it may find itself at a state it recognizes when there will be a non-trivial option graph and therefore an opportunity for further plan formation.

In practice, heuristics will be needed at many points in the automaton design. The difficulty of finding optimal decision strategies taking into account the time and storage constraints is far too great for precise solution. Whenever

there is a choice between a small benefit immediately and a possibly large benefit in the future, then a plausible heuristic will probably be needed.

The difficulties which arise as soon as more interesting subjective environments are considered are best introduced in a more concrete context. I shall therefore now describe the present computer program which involves a subjective environment with complex states, and with a more complicated transition rule. For a more extensive general discussion the reader is referred to Doran (1967, 1967a).

THE POP-2 PROGRAM: OBJECTIVE AND SUBJECTIVE ENVIRONMENTS

The program now to be described is written in the list processing language POP-2 (see the paper by R.J. Popplestone in this volume) which has been developed at the Department of Machine Intelligence and Perception of the University of Edinburgh. The language is implemented on an Elliott 4100 computing system and is oriented towards machine intelligence work.

The reader may find his understanding of this program and its behaviour aided if he keeps in mind the following analogy. A small boy lives at the busy centre of a large city. One day he is taken to a quiet suburb where he has never been before, and left to find his own way home. We suppose that he is too shy to ask someone the way and that he does not think of buying a map. He therefore starts walking, always preferring streets or districts where there is traffic and bustle, the more the better, since he remembers that he lives in a very busy place. Sometimes he realizes that he has walked in a circle, and then he sets off in some new direction from the busiest point he remembers how to reach. Ultimately, we suppose, he arrives home after a very long walk. Suppose that he is now taken back to the remote suburb and left there a second time. Now he should return home in less time and by a much more direct route, for he has the memory of his previous trek to guide him and therefore, for example, can plan ahead to some extent.

This small boy is very much in the situation of the simulated automaton now to be introduced. The automaton's surroundings, which I shall describe first, correspond to the city in the foregoing analogy.

The objective environment or 'enclosure' provided for the automaton consists of a square area (10 units $\times$ 10 units) with boundary and interior walls. Figure 3, which is an example of program output, shows this enclosure. The walls are not uniform but are 'formed' of letters of the alphabet. The automaton is represented by an asterisk and has location, and orientation to the top, left, right, or bottom of the picture. The automaton's co-ordinates are always integral. Note that 'objective' output is preceded by a double set of asterisks, 'subjective' by a single set.

The actions available to the automaton are the following:

- (a) STEP—to move forwards into the next unit square.
- (b) LEFT—to turn through a right angle to its left.

*** SAMPLE IS [C 1 STAND 40 1138]
 *** I KNOW THIS
 *** I HAVE A PLAN
 *** STEP
 *** *** TIME 1378 X4 Y2 FACING TOP

CCCCCCCCC
 E * B
 E DDD B
 E HHD B
 E IJ HHHAG
 E IJ HHHHG
 E IJ G
 E IJ G
 E G
 FFFFFFFFF

*** SAMPLE IS [C 0 STEP 41 1379]
 *** FORSEEN
 *** RIGHT
 *** *** TIME 1398 X4 Y2 FACING RIGHT

CCCCCCCCC
 E * B
 E DDD B
 E HHD B
 E IJ HHHAG
 E IJ HHHHG
 E IJ G
 E IJ G
 E G
 FFFFFFFFF

*** SAMPLE IS [B 5 RIGHT 39 1399]
 *** FORSEEN
 *** EXPLORE
 *** STEP
 *** *** TIME 1425 X5 Y2 FACING RIGHT

CCCCCCCCC
 E * B
 E DDD B
 E HHD B
 E IJ HHHAG
 E IJ HHHHG
 E IJ G
 E IJ G
 E G
 FFFFFFFFF

Figure 3. Sample output from the automaton/environment program showing the objective environment, typical subjective state vectors, and the automaton's 'chatty' remarks

(c) RIGHT—to turn through a right angle to its right.

(d) STAND—to remain still.

If STEP is impossible since the automaton is against a wall, then STEP has the effect of STAND. Selecting the action STAND must not be confused with selecting no action at all.

Note that the automaton knows nothing of the effect of these actions except what it learns through experience.

The automaton's subjective environment state is a 5-vector of the following form:

[⟨wall⟩, ⟨distance⟩, ⟨last action⟩, ⟨desirability⟩, ⟨time⟩]

Examples of state vectors appear in figure 3 after the words 'SAMPLE IS'. The first shown is

[C 1 STAND 40 1138]

and this implies that the automaton is facing the C wall, that there is one empty square between it and the wall, that its last action was STAND, that the desirability of the current state is 40, and that the system time is 1138.

Note that the automaton's view of its surroundings is very restricted. It can see only which 'letter' is in front of it, and how far away it is. Thus the subjective environment is very different from the objective environment. Let us call the first two elements of the state vector the *reduced state* (C1 in the above example). This reduced state is the unit used by the automaton in its processing. Now a given reduced state does *not* uniquely specify the automaton's true location, and this turns out to be both a source of confusion and of assistance to the automaton. Confusion occurs because the subjective environment may not react in the future as it has in the past, assistance because the automaton may react correctly in quite new situations because from its point of view they are *not* new but are identical to previously experienced situations.

Two further points need comment. First, the appearance of the automaton's last action in the state vector means that the automaton remembers past actions in much the same way as it remembers other information. Second, the desirability is included as a direct perception in this program for simplicity. It is calculated as a simple numerical function of the reduced state, namely:

$$50 - (\text{DISTANCE} + 3 * \text{WALL})$$

where WALL is 1 in the case of A, 2 in the case of B, and so on. The desirability is maximized when the automaton is facing and against the letter A, and we may then say that the automaton is in its 'nest'. There are local maxima elsewhere.

The automaton is typically placed at some point in its enclosure and expected to find its way to the nest and remain there. If moved out of the nest it should quickly return. More generally, it should always behave 'sensibly'.

Simple program facilities are provided which permit the automaton to be moved about, guided, and reset to some point in its history by an 'on-line' experimenter.

THE POP-2 PROGRAM: ORGANISATION OF THE AUTOMATON

Figure 4 shows a skeleton flow chart for the automaton. The cycle is as follows. The automaton first reads a state vector (SAMPLE), and stores in its memory

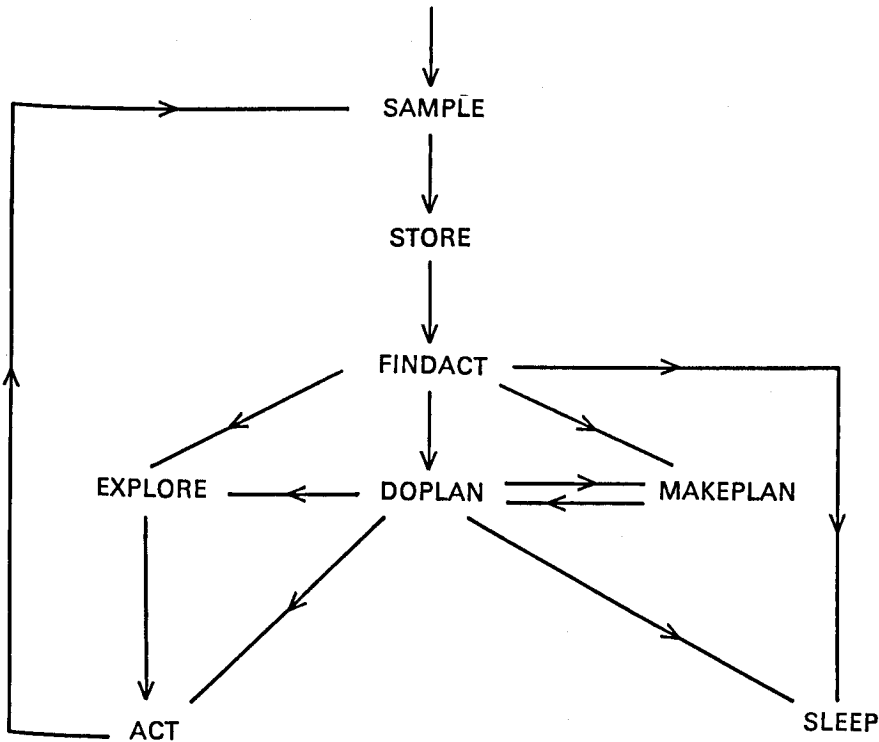


Figure 4. Outline flow-chart of the automaton

the last state vector transition (STORE). It then enters FINDACT and decides which of the following alternatives is appropriate:

EXPLORE—select randomly an action never before tried at this reduced state.

MAKEPLAN—carry out a lookahead and form a plan.

DOPLAN—select the next action of the current plan or go to EXPLORE

MAKEPLAN or SLEEP if this is indicated by the current plan.

SLEEP—stop.

In **ACT** the automaton actually implements an action, and the state vector and objective environment change appropriately before the automaton again goes to **SAMPLE**.

Sufficient has been said about the action of **SAMPLE** in the previous section. **STORE** will now be explained by reference to figure 5, which shows the arrangement of the automaton's 'memory' as a sort tree. The sequence of transitions shown at the top of the figure is represented as the tree structure shown. In general suppose that the automaton has observed a transition from state vector **x** to state vector **y**. It stores this as follows:

- (a) the tree node corresponding to the reduced state of **x** is located by starting at the top and branching appropriately;
- (b) relevant information kept at this node, for example, the desirability of the reduced state and the time of the last encounter is updated (not shown in figure);
- (c) the appropriate terminal node is located (by reference to item 3 of **y**) and a pair containing the time of **x** and the reduced state of **y** (a *consequence*) is added to the list of observed consequences already there.

Branches are created if they do not already exist.

The length of the list of observed consequences kept at each terminal node is limited by the program parameter **FORGETP**. If a new addition would cause the length of the list to exceed the limit set, then the oldest consequence is deleted to make room.

This storage system has the following properties:

1. the detailed history of the automaton can be recovered (with a little difficulty) provided that no information has been erased in the manner just indicated, and
2. the consequences in the past of applying a particular action to a particular reduced state are very easily retrieved.

The function of the *plan vectors* shown in the figure will be explained in the next section.

EXPLORATION AND PLAN FORMATION

In **FINDACT** the automaton must choose between the options **EXPLORE**, **DOPLAN**, **MAKEPLAN**, and **SLEEP**. To do this the automaton first refers to its memory to establish if it has ever before encountered the current reduced state. If not, and if the desirability of the reduced state is at the *target value*, then **SLEEP** is chosen, otherwise **EXPLORE**. The general significance of this target value will be explained below.

If the reduced state is recognized then the automaton inspects the corresponding plan vector (see figure 5) to establish whether it has already decided what to do in this situation. If so then it selects **DOPLAN**. Note that the stored plan instruction implemented in **DOPLAN** may cause entry to **EXPLORE**, **MAKEPLAN**, or **SLEEP**, rather than merely indicating a basic action.

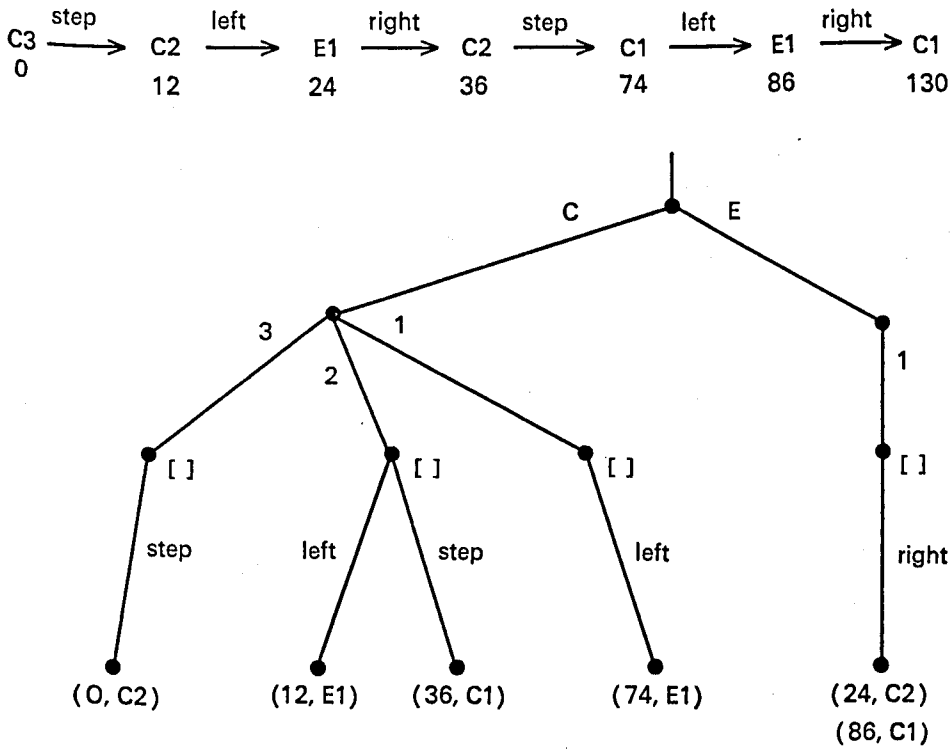


Figure 5. The memory tree. The transition history at the top of the figure is stored as the tree shown below. [] indicates location of plan vector. For details see text

MAKEPLAN is entered if the current reduced state is recognized, but not anticipated as part of a plan. Its main function is to grow a 'lookahead tree' of a type analogous to that used in many game-playing programs, and to select a plan. Figure 6(a) shows this tree with unimportant details omitted.

The tree is grown (downwards) by reference to the automaton's memory, and corresponds to the option graph earlier defined. Each node in the figure represents a reduced state. Each branch labelled with a Greek letter corresponds to the selection of a particular action. The unlabelled branches represent the observed consequences of applying given actions to given reduced states. Thus in the figure the root node corresponds to the automaton's current reduced state. The actions α and β have been tried previously in this state. α has been applied twice with differing consequences. β has been applied

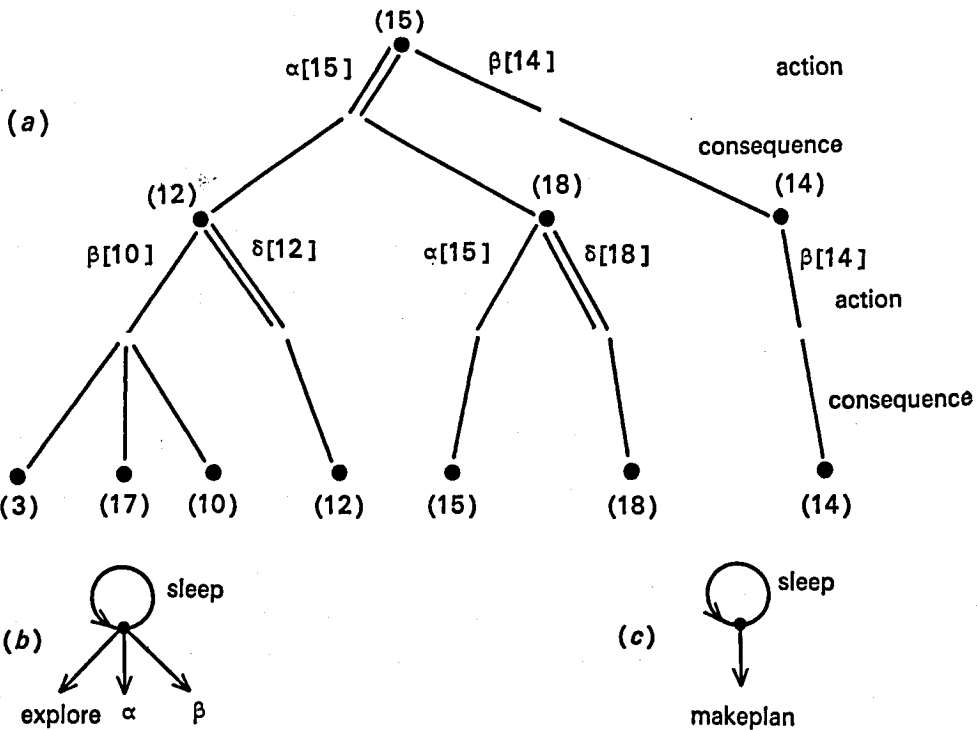


Figure 6. The lookahead mechanism. The tree (a) is grown by reference to the memory tree of figure 5, and values are assigned to the terminal states and 'backed-up' as indicated. Diagrams (b) and (c) show in greater detail non-terminal and terminal nodes respectively. For full details see text

once. The first consequent reduced state of the application of α has itself had β and δ applied to it, and so on. This lookahead tree is taken out to a fixed 'depth' determined by the program parameter `DEPTH`. Having 'called to mind' the relevant information, the automaton now selects an action for its current reduced state, and for all of the consequences that that action may have, and so on out to the depth of the tree. That is, the automaton selects a plan. In more detail, it uses the following process which is a crude form of the 'expectimax' process described by Michie and Chambers (1968a).

- (a) Values are assigned to the reduced states forming the branch tips. These values are loosely to be interpreted as the future mean desirability level to be expected if the automaton actually reaches the corresponding reduced state, and continues planning from there.

- (b) The reduced states one step back from the terminal states now have values assigned to them. Each action tried is considered and a value associated with it which is the mean of the values of its observed consequences. The greatest such action value is then assigned to the reduced state, and the corresponding action is adopted as the plan action for that reduced state.
- (c) This process is carried up the tree until the root state is given a value.

In figure 6(a) a consistent set of values has been indicated (square brackets for action values and round brackets for state values) and the plan actions are shown by double lines. The most important information kept in the plan vector of a reduced state is the plan value together with the corresponding plan action.

The process is rather more complicated than has been described, for at each non-terminal state of the lookahead tree the automaton has additional SLEEP and EXPLORE options. The true situation is indicated in figure 6 (b). At terminal states the automaton has a choice between the MAKEPLAN option—corresponding to the assigned value already mentioned—and SLEEP (figure 6 (c)). Each option is assigned a value, the greatest value is selected and passed up the tree, and the corresponding choice (say from α, β , EXPLORE, SLEEP) is stored away in the plan vector for use if the corresponding reduced state should actually be encountered.

How are these various values estimated? This involves a target value which is a desirability level the automaton is 'told' it can achieve but not exceed (compare the 'level of aspiration' used by the BOXES program (Michie and Chambers, 1968a)).

SLEEP—the corresponding value is simply the desirability of the reduced state at which sleep is proposed.

EXPLORE—the value assigned lies between the corresponding SLEEP value and the target value, according to the expression

$$S + (TV - S) * \text{EXPVALP}$$

where S is the SLEEP value, TV the target value and EXPVALP is a program parameter lying between 0 and 1.

MAKEPLAN (i.e. tip value)—this is calculated in a manner akin to that used for EXPLORE, but is made rather larger following the argument that planning can be expected to give better performance than random exploration. This increment is made a decreasing function of the number of times the corresponding reduced state has been encountered using the parameter TRANSITP.

This outline description of the functioning of the lookahead and planning mechanism has avoided a number of complexities which, although trying in practice, have little general significance. For example, branches of the lookahead tree often coalesce or loop with consequent difficulties.

	TRIALS										totals	
	1	2	3	4	5	6	7	8	9	10		
INCARNATIONS	A	449 43	179 14	322 22	344 18	265 19	203 18	211 17	1625 50	288 16	427 20	4313 237
	B	528 41	129 12	122 12	111 12	143 12	143 12	143 12	143 12	143 12	144 12	1749 149
	C	722 66	194 18	295 21	257 22	451 28	245 16	297 17	394 21	227 14	269 16	3351 239
	D	3436 191	525 18	157 12	486 16	265 13	257 14	378 14	259 13	321 14	452 14	6536 319
	E	858 58	193 14	374 19	417 18	426 19	303* 13	623 23	535* 18	688* 20	2275* 52	6692 254
	F	589 46	183 14	270 16	215 14	249 14	179 12	200 12	200 12	222 13	189 12	2496 165
means		1097 74	234 15	257 17	305 17	300 18	222 14	309 16	526 21	315 15	626 21	

Table 1. Sample results showing the automaton's performance in the environment of figure 3. Each incarnation involved a blank initial memory and ten successive trials from the starting point of figure 7. The upper figure of each entry is time taken to reach the nest and the lower figure is the number of actions used. Asterisks indicate unsuccessful trials. The minimum possible number of actions required was 12.

design. As a result of experimentation with the program the following remarks may be made.

The automaton successfully uses its record of its past explorations to form and implement plans. These plans enable it to find its way to its nest by a much more direct route on the second or subsequent trials than that followed on the first trial. The planned route need not be optimal, however. Figure 7 shows a typical example of this improvement. Note that the automaton does much more than merely retrace its steps, and that the improved route involves an objective location not previously visited by the automaton. Prior to trial 1, the automaton's memory was quite blank. Table 1 presents some sample detailed results. In each of six 'incarnations' the automaton was placed at the starting point of figure 7 with a blank memory, and left to find its way to the nest. It was then replaced nine times, but allowed to cumulate its memory. The upper figure of each entry in the table is the time taken to reach the nest in tens of basic units, and the lower figure is the number of actions used. The following points are worthy of note:

- (a) the incarnations are not all identical since, particularly in the initial stages, the automaton sometimes engages in random exploration;
- (b) performance on trial 2 is uniformly much better than on trial 1;
- (c) the automaton's behaviour does not necessarily become stereotyped (see TRANSITP below). During trial A8, for example, the automaton tries an exploration action and gets thoroughly 'lost' in consequence;
- (d) the asterisks indicate trials when the automaton 'gave up' just before reaching the nest (for fairly sensible reasons).

Figure 8 shows trials, A, B, C of Table 1 plotted as graphs.

The behaviour of the automaton depends heavily upon the values chosen for the parameters mentioned previously.

FORGETP—this fixes the maximum number of consequences held at each branch tip of the memory tree. A value of 6 rather than 3 for this parameter increased the automaton's thinking time significantly. The automaton was also slower to escape from the fairly common type of situation in which it is misled by events recalled from the past which are not in fact relevant, and in consequence loops repeatedly (*see below*).

EXPVALP—this is involved in value setting as indicated above. If it is set too low (0.2 rather than 0.6) then, in the type of environment described, the automaton will tend to 'sleep' at points of locally high desirability.

DEPTH—taking the lookahead to a fixed depth of 7 rather than 3 made the automaton less likely to be trapped in locally desirable areas, but markedly increased the thinking time.

TRANSITP—this helps determine the relative value which the automaton assigns to planning rather than exploration. Suitably set it causes the automaton to vary from time to time an apparently fixed route.

The point has already been made that the automaton has a deliberately limited view of its surroundings. This causes it to generalize a response judged best in one situation to all situations which have the same reduced state. In the type of objective environment used here, the automaton will sometimes benefit from this built-in tendency to generalization, and will sometimes repeatedly go astray. In passing we may note that the automaton is assuming a truly stochastic transition rule, when this is not the case in reality.

Finally, consider again the question raised at the end of the last section. The basic version of the program makes no further use of plans once they have been fully or partially implemented. However, two variants of this program have been tested, the first of which (Variant A) uses an old plan value attached to a reduced state as a source of information when calculating a makeplan value, and the second of which (Variant B) uses an old plan vector as if it had been calculated as part of the current lookahead. Both variants are generalizations of the basic program in that they still reject plan information of over a fixed age.

Simple experiments with these programs suggest that Variant A is superior to the basic program in situations where a particularly deep lookahead is

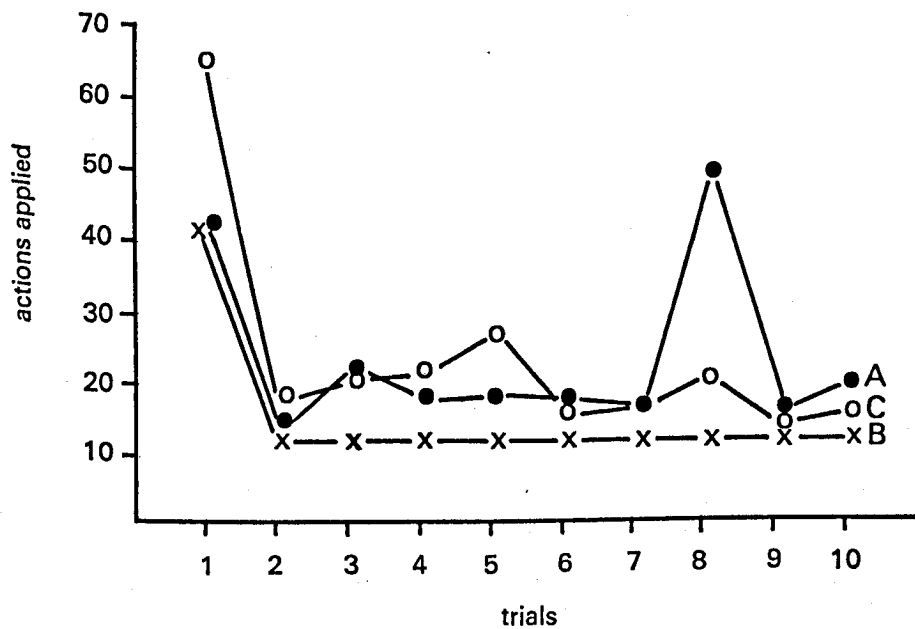
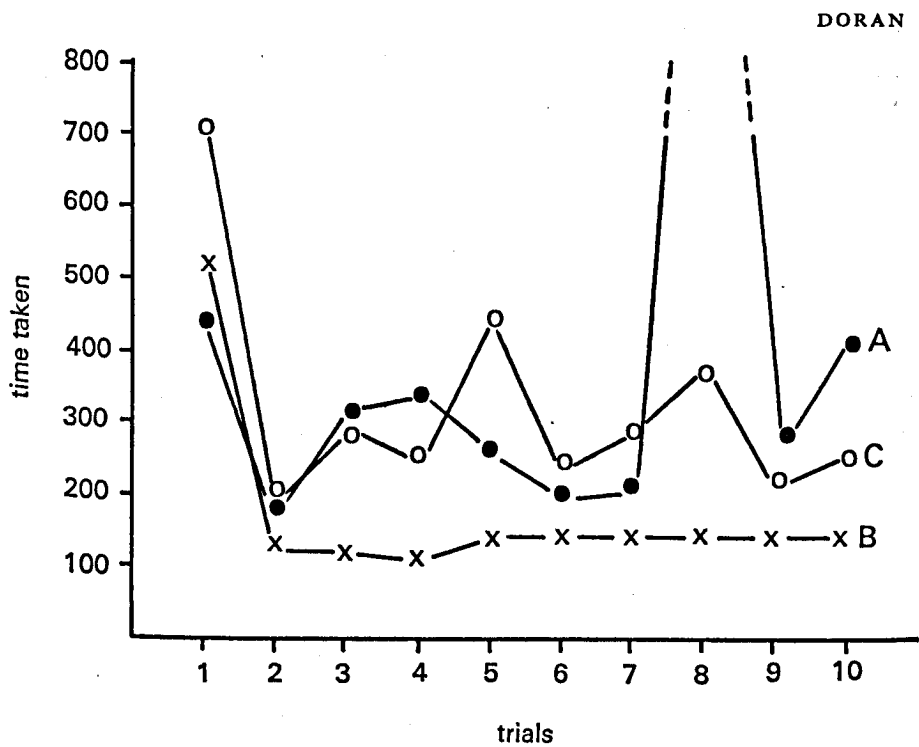


Figure 8. Incarnations A, B, and C of Table 1 plotted as graphs

essential. It is using a rote learning technique analogous to that of Samuel (1959) to increase effective lookahead depth. Variant B has the effect that when a planned route has once been followed, it is thereafter always followed 'without further thought' until the plans defining the route become too old. The automaton then forms fresh plans. Following a route 'without further thought' implies much more rapid movement, but no chance at all of possibly beneficial variation.

MINOR LINES OF DEVELOPMENT

It will be clear to the reader upon reflection, if not before, that the program that I have described is no more than a first attempt at a very large and complex objective. Even without considering any major modification to the automation design, there are the following directions in which improvements to it could certainly be made.

- (a) The *lookahead tree* used in forming plans is organized in a very arbitrary way. There are surely better ways for the automaton to decide whether or not to continue consideration of some part of the option graph than for it to use a fixed depth rule. For example, the work on the Graph Traverser program (Doran and Michie, 1966) immediately indicates the use of a state evaluation function of some kind to direct the growth of the tree. This could merely use the desirability, but more complex mechanisms can easily be imagined.

A complementary approach is to consider the probability that a given branch will ever be reached. In the extreme case when this probability is zero, further growth of the lookahead tree from that branch is entirely wasteful. Some probabilistic generalization of the ' $\alpha-\beta$ heuristic' is perhaps required here (Edwards and Hart, 1963; see also Samuel, 1967).

- (b) Once the lookahead tree has been grown there comes the problem of selecting a plan. The current simple method of selecting actions is open to the criticism that one 'unlucky' consequence of an action in a given situation can damn it for ever. This is essentially the 'Two-Armed Bandit' problem, and the reader is referred elsewhere for a discussion of its significance (e.g. Jacobs, 1967; Michie, 1966).
- (c) The repeated use of plan information has already been discussed above. There are certainly more possibilities than those which have been actually tried.
- (d) The present method of deleting information from the memory is very simple. More complex methods could be tried and could be chosen to use the expected properties of the subjective environment.

These various improvements to the design of the automaton, though of considerable interest, would not in themselves add up to any major step forward. Nor would this be achieved by a study of the automaton's performance

in a variety of environments. I propose now to introduce some ideas which perhaps dig a little deeper.

MAJOR LINES OF DEVELOPMENT

The present automaton can ultimately learn the truth of such statements as the following (figure 3):

If I am facing the G wall and two steps from it, and if I turn right, then I shall be facing the F wall and against it, or facing the F wall and one step from it, or facing the F wall and two steps from it, with (pseudo-) probabilities P_1 , P_2 , P_3 , respectively.

It cannot, however, learn the truth of either of the two following statements:

(A) If I am facing the G wall and I turn right, then I shall be facing the F wall.

(B) The action STAND never has any effect.

If the automaton is to learn such facts as this it must be able to generate them in some representation. (A) might be handled as follows. Consider again the sort tree shown in figure 5, and find the node corresponding to the reduced state c3. Both the plan information at that node, and the recorded consequence of the action STEP, can be regarded as statements about c3. We would like the automaton to record and use broadly similar statements about c, more generally, about all nodes higher in the tree. A method of achieving this is to enable the automaton to seek statements which are true for all nodes on branches out of some node and to associate them with that higher-level node, (compare the proposed 'lumping' of boxes by the BOXES program (Michie and Chambers, 1968)). Plan formation would then involve a more complex 'lookahead' tree which would be grown using the *simplest* statements which would give satisfactory prediction.

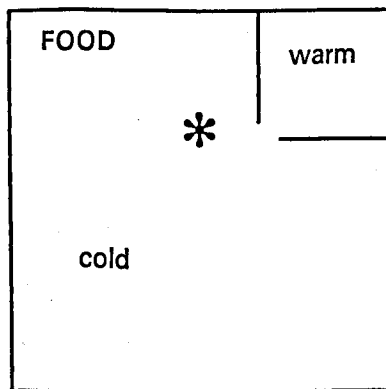
This latter step leads us to the realization that the automaton need not always sample the full state vector—it need only note that part of the vector needed for its current planning.

To cope with statement (B)

The action STAND never has any effect.

a further mechanism seems necessary. Each node of the sort tree can be regarded as a test. Suppose that we allow these tests to be not merely a matter of looking at the next element in the current state vector, but general tests on both the current state vector and on immediately preceding ones (compare the EPAM program of Feigenbaum, 1961,). Statement (B) can now perhaps also be handled for it is saying that the action STAND always leads to a situation which is (effectively) identical with that preceding it, and we have now allowed the automaton to test for this. Of course there is now the problem of how the automaton should decide *which* tests to apply and in which order to apply them; but at least the door seems open.

Objective environment



State vector

[⟨wall⟩, ⟨distance⟩, ⟨last action⟩,
⟨hunger⟩, ⟨temperature⟩, ⟨time⟩]

Figure 9. An environmental situation requiring major improvements in the design of the automaton

Consider finally figure 9, which shows a more complex environment and state vector. The automaton is provided with a 'nest' which is 'warm' but which it must leave to obtain 'food' which is kept outside in the 'cold'. The state vector is augmented by elements indicating the degree of hunger and the temperature, and the explicit perception of desirability is omitted. A new action, EAT, is provided.

The kind of behaviour we require of the automaton is that it should be able to find its way to the nest, and that it should only leave the nest when it is sufficiently hungry to brave the cold, returning to the nest immediately it has eaten. This 'sensible' behaviour is to be learned from an initial naïve state, perhaps with some simple 'tuition'.

The automaton must take into account two desirability factors, namely the hunger and the temperature, and must use the remainder of the state vector to help it control these factors. There is a conflict between these two factors, and more important hunger is essentially discontinuous even for optimal automaton behaviour, for it changes sharply with the single action EAT.

Consider the following sequence of events which might overcome these difficulties:

- (a) Initially the automaton is in the nest, is warm, and is not hungry.
It therefore sleeps.

- (b) The automaton wakes because it becomes hungry. This implies a basic scan of the state vector while asleep.
- (c) The automaton notices that it is hungry (rather than cold) and recalls its subjective state when it last ate. This state it adopts as a *goal*. This is a new *ad hoc* mechanism which can be regarded as a first step towards 'means end analysis' in the General Problem Solver program sense, and ultimately towards the complex type of planning discussed in Miller, Galanter, and Pribam (1960).
- (d) The automaton then seeks a plan directed towards achieving its goal. Both the growth of the lookahead tree and the choice of plan can be guided by an evaluation function using the features of the automaton's goal. Note that the feature of the goal state that it is cold can be used to help guide the search.
- (e) Having formed and implemented one or more plans the automaton will ultimately reach its goal. At this point the 'food' goal is cancelled, to be replaced at once by a directly analogous 'warmth' goal.
- (f) Plans are made and implemented for the new goal. It is ultimately achieved and the automaton is returned to (a) above.

The automaton would *learn* to carry out the above operations in the sense that it would gather for itself the experience needed to form the plans involved. The goal setting and planning capacity would be inbuilt. 'Tuition' might prove necessary in the initial stages and could take the form of forcing the automaton to choose certain actions and thus perceive the consequences of them.

CONCLUDING REMARKS

In this paper I have described a program which simulates a heuristic automaton in a very primitive but natural environment. The environment is natural in the sense that its properties are analogous to those of the world around us, but primitive in the sense that those properties are vastly simplified.

The reasoning behind this program argues that intelligence is very much a response to our own everyday environment and that to understand its nature it is desirable, perhaps essential, to study the properties of that environment. By environment I must emphasize that I mean *subjective* environment. Thus we have been primarily concerned with the automaton's world as the automaton sees it, not as we the outsiders see it. Both this stress on the subjective nature of the automaton's problems, and the fact that I have proposed no formal representation of the information gathered and processed by the automaton are open to dispute. What does seem clear, however, is that the only way to make concrete progress is to write computer programs and to see how they perform.

Acknowledgments

The research described in this paper is sponsored by the Science Research Council under the general supervision of Professor Donald Michie. Many people have made helpful suggestions and criticisms concerning this work, but I am particularly in debt to my colleague Dr R. M. Burstall.

REFERENCES

- Andreac, J. H. (1964), *STELLA: a scheme for a learning machine. Proceedings of second IFAC congress*, pp. 497–502. London: Butterworth.
- Doran, J. E. and Michie, D. (1966), Experiments with the Graph Traverser program. *Proc. R. Soc. A*, 294, 235–59.
- Doran, J. E. (1967), Criteria for intelligence in a programmed automaton. *Memorandum MIP-R-26*. University of Edinburgh: Department of Machine Intelligence and Perception.
- Doran, J. E. (1967a), Designing a pleasure-seeking automaton. *Memorandum MIP-R-28*. University of Edinburgh: Department of Machine Intelligence and Perception.
- Edwards, D. J. & Hart, T. P. (1963), The α - β heuristic. *Artificial intelligence project memorandum No. 30* (revised). RLE and MIT computation centre.
- Ernst, G. & Newell, A. (1967), Some issues of representation in a general problem solver. *AFIPS*, 30, 583–600. Spring J.C.C. Washington: Spartan Books.
- Evans, T. G. (1964), A heuristic program to solve geometric-analogy problems. *AFIPS*, 25, 327–38. Spring J.C.C. Baltimore: Spartan Books.
- Feigenbaum, E. A. (1961), The simulation of verbal learning behaviour. *Proceedings of the Western J.C.C.*, 19, 121–32. Baltimore: Spartan Books.
- Friedman, L. (1967), Instinctive behaviour and its computer synthesis. *Behav. Sci.*, 12, 85–108.
- Jacobs, O. L. R. (1967), *Introduction to Dynamic Programming*, pp. 114–19. London: Chapman and Hall.
- Michie, D. (1966), Game-playing and game-learning automata. *Advances in programming and non-numerical computation*, pp. 183–200 (ed. Fox, L.). Oxford: Pergamon Press.
- Michie, D. & Chambers, R. A. (1968), 'Boxes' as a model of pattern-formation. *Towards a Theoretical Biology I*, pp. 206–15 (ed. Waddington, C. H.). Edinburgh: Edinburgh University Press.
- Michie, D. & Chambers, R. A. (1968a), BOXES: an experiment in adaptive control. *Machine Intelligence*, 2 (eds Dale, E. and Michie, D.). Edinburgh: Oliver and Boyd.
- Miller, G. A., Galanter, E. & Pribram, K. H. (1960), *Plans and the structure of behaviour*. New York: Holt.
- Newell, A., Shaw, J. C. & Simon, H. A. (1959), Report on a general problem-solving program. *Proceedings of the international conference on information processing*, pp. 256–64. Paris: UNESCO.
- Nilsson, N. J. & Raphael, B. (1967), Preliminary design of an intelligent robot. *Computer and information sciences* 2, pp. 235–59 (ed. Tou, J. T.). New York: Academic Press.
- Samuel, A. L. (1959), Some studies in machine learning using the game of checkers. *IBM J. Res. Dev.*, 3, 211–29.
- Samuel, A. L. (1967), Some studies in machine learning using the game of checkers II—recent progress. *Stanford artificial intelligence project memorandum*, No. 52.
- Sandewall, E. (1967), An environment for an artificial intelligence. *Report No. 10*. University of Uppsala: Department of Computer Sciences.
- Toda, M. (1962), The design of fungus-eater: a model of human behaviour in an unsophisticated environment. *Behav. Sci.*, 7, 164–83.
- Walter, W. G. (1953), *The living brain*. New York: Norton.

Collective Behaviour and Control Problems

V. I. Varshavsky

Leningrad Branch of the Central Economic-Mathematical Institute

... societies of insects have solved this problem—have integrated in a single whole all their tiny individual brains by means whose mystery we are beginning to penetrate. Rémy Chauvin Les sociétés animales de l'abeille au gorille Paris 1963

Progress in technology and the complexity of the social structure of human society makes us face the necessity of creating so-called large-scale systems. These systems are of great variety; yet, from the point of view of control, they possess a number of common features. In particular, they are 'large'. What are the main difficulties confronting us in the construction of such systems?

Performance quality of a large-scale system is as a rule estimated by a function called performance quality criterion. An analytical representation of the criterion is in most cases unknown and the only information available is its current value. The performance criterion depends, among other things, on a great number of uncontrollable parameters, hence its behaviour changes greatly with time. Besides, the number of parameters with which a system can be described is usually too large, therefore direct use of available criterion values for optimization is impossible. High complexity and the frequent unfeasibility of an analytical description for large-scale systems makes all the classical design techniques inapplicable. It is believed that these difficulties are due to lack of adequate means of description of the large-scale systems and that development of such means will improve the situation. This is no doubt true. I think, however, that the language appropriate for the purpose will differ from the language of analytical formulae. In this report an attempt has been made to discuss some problems concerning large-scale systems in terms of the collective behaviour of their components.

Difficulties arising in the application of classical techniques and algorithms are in the main due to their universality. On the other hand, specific problems

possess a particular inner organization, owing to the fact that they originate in practical needs. By exploiting this organization one may often obtain a solution. This was first pointed out by I.M. Gel'fand and M.L. Tsetlin in 1962:

In problems of this kind a search is necessary. In the course of the search, progress towards the goal is achieved and information necessary for further progress is obtained. But the search is efficient only if the problem structure, i.e. the intrinsic features helping to reach the goal in reasonable time, is taken into account. These features are not known in advance but more or less plausible guesses can be made. The structure is exploited by means of putting various hypotheses (plausible conjectures about the problem structure) and constructing search tactics based on these hypotheses. In the course of the search the hypotheses are tested: one, which makes the search successful, is true. Resorting to the help of hypotheses is deliberate rejection of a search through all possibilities; without it there is no chance of succeeding in any more-or-less complex situation.

In the problem of finding the minimum of a function the hypothesis is that its variables fall into two groups. The first one (which includes almost all variables) unites the variables whose alteration leads to considerable alteration of function value. Selection of such parameters, which are called inessential, gives rise to no difficulties and can be comparatively rapidly made.

The second group of parameters includes a few variables (or functions of these few variables) whose alteration causes comparatively small variation of the function. These parameters are called essential. (see also Gel'fand, Gurfinkel and Tsetlin, 1962)

For instance, accuracy of shooting greatly depends on how you hold your gun (by the muzzle or by the butt), on the state of your nervous system (you are sober or suitably intoxicated), etc. However, these parameters are inessential, for optimization over them can be rapidly performed. The essential parameters here are accuracy of sighting at the objective and smoothness of firing; their relative effect on the quality of shooting is much less than that of the first group of parameters.

The application of the above considerations allows us to obtain solutions for a number of complex computational problems, and in particular for the problem of phase scattering of elementary particles. Determination of specific features and regularities of the inner organization is of great importance for obtaining computational procedures for the solution of complex problems. Heuristic programming is also based on having relevant information about a problem under consideration. Strictly speaking, such an approach is no novelty from the methodological viewpoint. Usually, any analytical technique is initially conjectured and only after that is a rigorous formal proof supplied. However, classical techniques and algorithms are applicable to broad classes

of problems while the new complex problems we encounter more and more in every-day life demand specially oriented methods. This puts in the forefront the heuristic aspect of problem solving and the study of heuristic algorithms comes to be of great importance. Unfortunately, human society can boast only a few men of genius; but society cannot afford to have the solution of its important problems wholly dependent on man's intellectual abilities. We would hardly find it satisfactory if our salary depended on book-keepers' ingenuity.

Heuristics is used in technology on an even larger scale. By this we mean not the invention of new facilities but the design of complex systems on the basis of ready-made blocks. Consider the problem of computer design, for example. In the framework of automata theory there is a theoretical possibility of the optimal design of the computer looked upon as an automaton. But any attempt to proceed along this line would be futile. On the other hand, the designer does handle the problem, and quite successfully, proceeding from non-formal considerations concerning the type of algorithms to be performed by the computer, its components and their interaction, etc., the whole of the computer being divided into a number of relatively simple blocks solving some particular problem. Many complex modern control systems are designed in the same way. Thus, the process of design of complex systems includes a non-formal stage: design of system structure, distribution of functions of control, dismemberment of the system into more-or-less autonomous parts. It is natural to describe control organized in this manner as heuristic. Heuristics displays itself here both in methods and in algorithms for the solution of problems, and in ways of implementing these algorithms.

Research shows that complex biological systems consist of entities with a high degree of autonomy. Multicellular organisms consist of cells, which under appropriate conditions are able to exist out of the organism, their behaviour being sufficiently complex. In such 'superorganisms' as families of ants or bees, the possibility of individual existence is higher. And finally, human beings, members of the most complex biological system, enjoy autonomy to a very great extent.

However complex the behaviour of the object is, whatever its criterion function and motivation, up to the social motivation in society, it is the result of the individual behaviours of a great number of elementary components. How these individual behaviours of elementary components, each having a goal of its own, form the expedient behaviour of the whole system is one of the most fascinating problems of modern science. It is natural to suppose that there are basically only a few fundamental principles in the organization of such systems. The importance of understanding these principles can hardly be overestimated. The models of collective behaviour of automata, constructed by M.L. Tsetlin are highly satisfactory for the study of such principles. A survey of research on models of collective behaviour of automata can be found in the report by Tsetlin and Varshavsky (1966). A number of important

points and formulations of problems can be found in the report by Gel'fand, Gurfinkel and Tsetlin (1962). In this report, in particular, the principle of least interaction for joint functioning of nervous centres ('game' of nervous centres) is formulated.

In the present paper some examples of control organization which are based on general ideas inspired by the studies of automata collective behaviour will be discussed. The author started work in this direction in collaboration with the late Dr M.L. Tsetlin and most of what is presented had been completed before his premature death.

In the examples that follow we were interested not so much in the design of specific systems as in the possibility of a heuristic approach to organization of interaction between autonomous parts of a control system.

For our first example let us consider the problem of synchronization of commutator and decommutator in a telecommunication system with time-shared channels in the presence of noise (*see* Tsetlin and Varshavsky, 1966 and Varshavsky, Meleshina and Tsetlin, 1965). In the first example the period of commutation is known in advance; in the second it is not known. In the first case the solution is reduced to the organization of the behaviour under which every automaton solves its own problem independently of others. In the second case there are two levels in the system: automata of the first level act independently of each other, while those on the second level are fastened together by 'mutual interests' and produce control signals which organize the behaviour of the first level automata.

The system under consideration works as follows. The commutator questions data units in turn in accordance with a program and conveys obtained information through a transmitter into a communication channel. The receiving-end decommutator detects and identifies the signals pertaining to different data units. Thus the decommutator should perform simultaneously the same sequence of operations as the commutator. The above situation arises when it is necessary to provide a control for an object carrying data units on the basis of the received telemetering information. Synchronization of the commutator and decommutator includes the problem of synchronization of logical networks of the receiver with the corresponding networks of the transmitter for correct decoding of the information received. We assume that to enter into synchronous regime the transmitter sends a tuning signal before the operation starts (or possibly in the course of operation). This signal is a binary code K of length equal to n bits. The time over which a data unit is switched on to the channel is the time corresponding to the transmission of one bit of the code (length of bit). We also assume that both the receiver and the transmitter operate in discrete time and have the same standard of time (sampling period) determining the length of bit. The problem is complicated by the fact that the system can work in several modes characterized by different length of bit (duration of data reading from one data unit) and different number of interrogated data units. Information about the mode is

contained in synchronizing code. Thus we have two problems: (1) synchronization by the moment of change of bits, i.e. by the moment of change of channels; and (2) synchronous reproduction of the signal by the receiver. For the first problem, the period (or a few periods) of carrier frequency can be chosen as a time unit. For the second problem, it is natural to adopt the length of bit as this unit. To allow for the fact that the signals are corrupted by noise in the communication channel we assume that the probability of an error at the receiving end is equal to p , ($p < 1/2$). The errors are supposed to be independent at every moment of time.

It has been shown (see Varshavsky, Meleshina and Tsetlin, 1965) that the above two problems can be solved in one and the same manner. Here we shall discuss only the reproduction of the synchronizing code. Consider a device producing signals that are compared with signals coming from the transmitter. The device is 'fined' when its output signals do not coincide with the received signals, and 'rewarded' otherwise. Hence the problem of the synchronizator design is reduced to the design of an automaton with optimal behaviour in periodic random environment. An automaton is said to operate in the periodic environment $P(p_1(t); p_0(t))$ if its output 1 at time t leads to the fine at time $t+1$ with probability $p_1(t)$, while the output 0 leads to the fine with probability $p_0(t)$. Probabilities $p_1(t)$ and $p_0(t)$ are periodic functions of the time with period T . Let the period be known and consider the group of automata shown in figure 1. The period of the commutator is equal to T and hence, as is readily seen, every automaton A perceives the periodic random environment

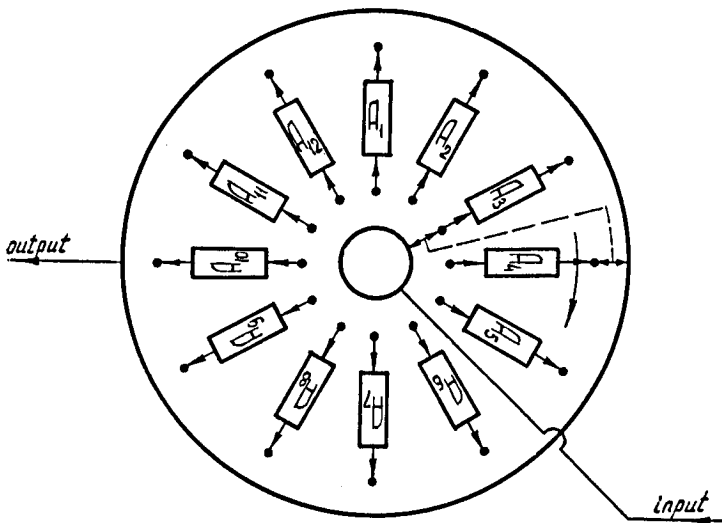


Figure 1

as a stationary one, i.e. the probability for the automaton to be fined does not change with time. If the construction of every automaton is asymptotically optimal for the stationary random environment (see Tsetlin and Varshavsky, 1966) then the total average fine for the above automata differs little from

$$1/T \sum_T \min(p_1(t), p_0(t)),$$

provided the memory of every automaton is of sufficient capacity. In other words the probability of correct reproduction of the synchronizing code tends to unity as the memory capacity of the automata infinitely increases. Now let the period be unknown but not exceed some value T_{max} . The problem of determining the period can be reduced to the Goore game (see Tsetlin and Varshavsky, 1966). Consider the construction shown in figure 2. It differs from the construction shown in figure 1 by the presence of the additional automata B , each of which has two possible actions, 0 or 1. The number of positions of the commutator is equal to T_{max} . The commutator can stop only in positions corresponding to a automaton B whose action is 1. Therefore the period of rotation of the commutator is equal to the number of automata providing action 1. After one or several revolutions of the commutator, automata B are fined with probability equal to the average fine of the automata A during that time. It is not difficult to see that the probability of fine for automata B is minimal when the period of rotation of the commutator is equal to or a multiple of the period T . Analytical study of the behaviour of the above system is quite complex. The results of computer simulation of the system are given in Table 1, where the following notation is adopted: n is the

n	T_{max}	T	p_1	p_0	M	τ	k
10	10	5	1/8	1/8	0.125	170	11100
10	10	7	1/8	1/8	0.125	147	1111000
10	20	6	1/8	1/8	0.125	210	110000
10	24	12	1/8	1/8	0.125	203	111111001000
10	24	11	1/8	1/8	0.125	208	11111001000
15	24	12	1/8	1/8	0.125	175	111111001000
15	10	5	1/8	1/8	0.125	200	11100
15	10	7	1/8	1/8	0.125	144	1111000
15	20	11	1/8	1/8	0.125	228	11111001000
15	24	12	1/8	1/8	0.125	192	111111001000
15	24	6	1/8	1/8	0.125	250	110000
10	20	7	1/8	1/8	0.125	171	1110000
10	20	7	1/4	1/4	0.250	262	1110000
10	30	10	1/3	1/3	0.333	250	1111110000
10	30	10	1/8	1/4	0.175	320	1111110000
10	30	10	1/3	1/8	0.250	375	1111110000
10	30	10	1/4	1/3	0.283	382	1111110000

Table 1

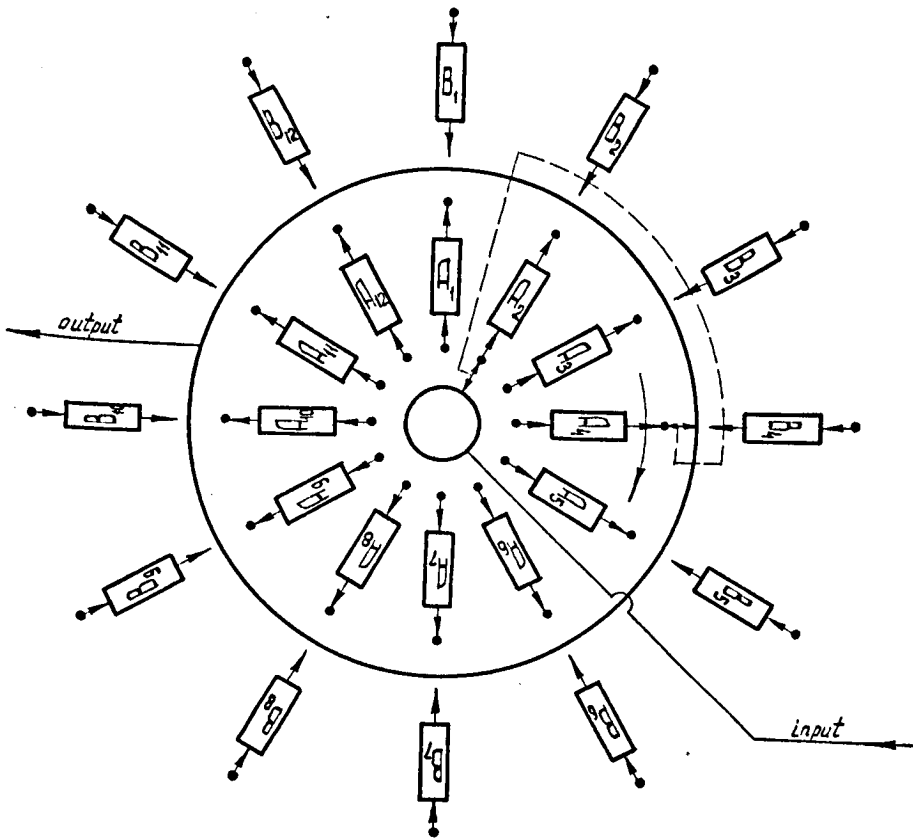


Figure 2

memory capacity of the automata with linear tactics (see Tsetlin and Varshavsky, 1966) (the same for both kinds of automata, A and B); T_{max} is the number of pairs of automata taking part in the operation; T is the period of the environment; M is the average number of fines per one tact, imposed on the system after the end of the transient; p_1 and p_0 are the probabilities of incorrect reception of 1 and 0 correspondingly. The stability of the processes in the system has been estimated by the average number of fines to within the third decimal place, all the experiments including not less than 50,000 tacts. As is seen from Table 1 the system is stable according to the criterion associated with M . This criterion apparently does not 'feel' random short deviations from the stable regime.

Table 1 also shows that duration of the transient is comparatively short. Perhaps this is due to the choice of the initial states of the automata with linear tactics—they were chosen to be on the boundary of act changing. Construction of the pertinent device gives rise to a number of extra requirements concerning the choice of a synchronizing code and a group of 'phasing' places, where signal values do not change. However, we shall not enter into discussion of these matters. It is worth noting only that the system under consideration is highly reliable: random failures of automata do not practically affect performance quality, and the system with variable period is stable with respect to complete failure of individual automata.

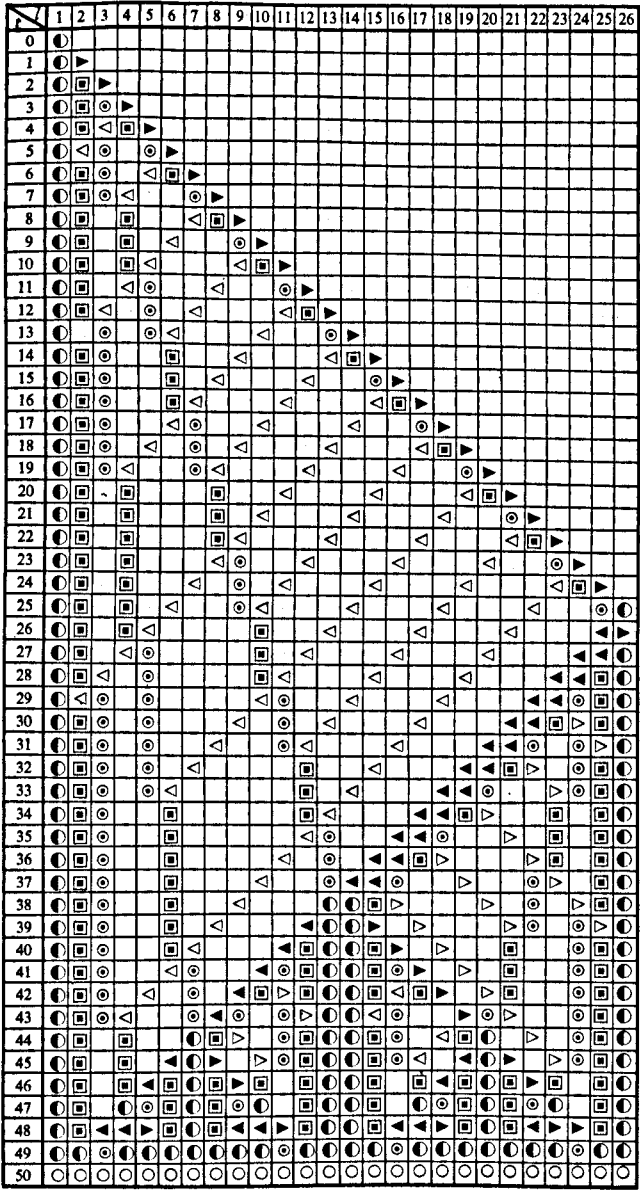
The above simple example shows how, on the basis of means capable of solving a simple problem, collective behaviour can be organized for the solution of more complex problems. Yet, in our intuitive notion of collective behaviour and its potentiality, we are not content with the simple summation of individual efforts of members. The natural question arises whether there is a possibility for a collective to possess a kind of behaviour more complex than that representing the simple sum of behaviours of its members. In automata language this question reads; 'Can an automaton imbedded in a collective solve problems which are essentially more complex than those it can solve on its own?'. A positive answer to the question seems to contradict the fundamental laws of Nature. However, consider a pair of automata each having k inner states. The complexity of problems that can be solved by every automata is constrained by the value of k . The system of automata has k^2 inner states and one can hope that there could be an organization of interaction between two automata such that each of them was able to solve a problem requiring $k^2/2$ inner states. The above is simply speculation. But now let us turn to an example which actually provides a positive answer to the question formulated above.

We shall discuss the problem of synchronizing an automata chain as posed by J. Myhill, which first appeared in print in a paper by E.F. Moore (1964). The problem is that of causing all parts of a self-reproducing machine to be turned on simultaneously. Consider a chain of soldiers, each soldier being able to exchange information with his two neighbours. The chain is finite, so

the soldiers at the ends of the chain have only one neighbour each. One of the end soldiers receives the command and on this the whole squad must 'come to an agreement' and fire their weapons simultaneously. Having received information from his neighbours each soldier responds with a constant time lag, the same for all soldiers. The question is whether there exists a local algorithm of behaviour that would be the same for each soldier, with slight modification only for the two at the ends of the chain. Being independent of the number of soldiers in the chain, the algorithm should provide the synchronization of the firing. A formal statement of the problem can be given as follows.

Consider a chain of k identical automata A in which every automaton is connected only with its two neighbours. Find a structure of the automata A (if any) such that, having received a starting command at time $t=0$, one of the end automata can cause every automaton of the chain to go into one particular terminal state, all at exactly the same time T ; the complexity of the structure is to be independent of the number of automata. According to E.F. Moore (1964) the problem was first solved by J. McCarthy and M. Minsky. In 1962 E. Goto found a solution where the synchronization took place at minimal time $T=2k-2$. But the automaton proposed by Goto had too many inner states. In 1965 V.I. Levenstein published a brilliant solution (see Levenstein, 1965) with 9-state automaton and minimal synchronization time $T=2k-2$. A. Waksman (1966) published a solution for the 16-state automaton.

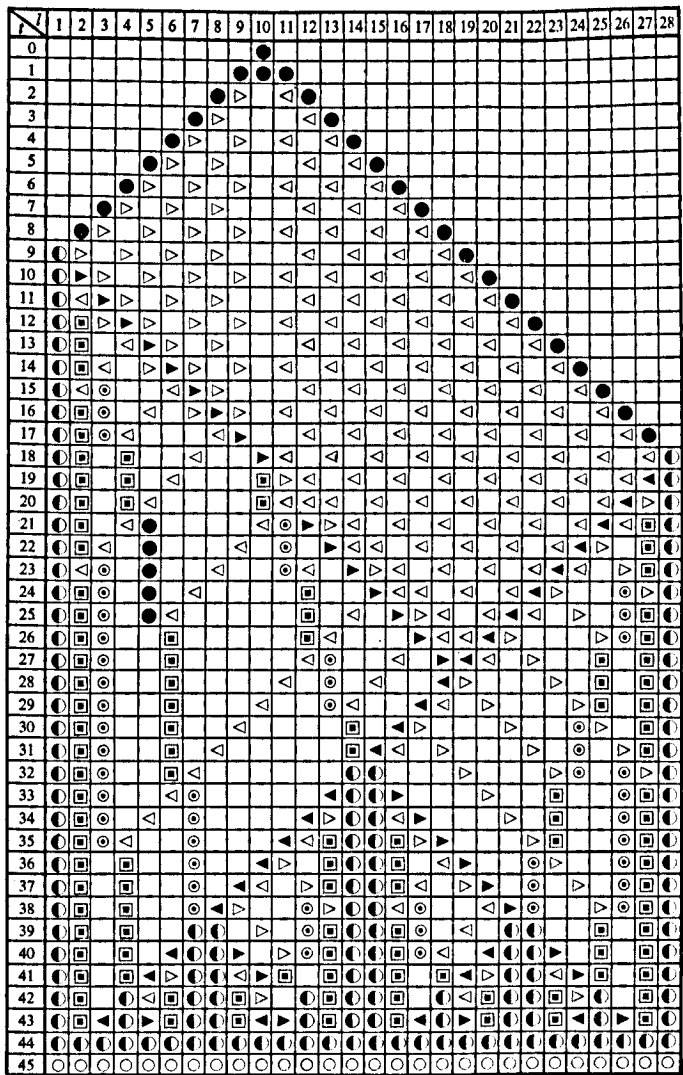
Let us sketch in brief Levenstein's solution. Consider a chain of Moore automata, the inner state of automata is at the same time the signal that propagates down the chain. A system of signals is arranged to cause a segment of chain to divide itself in two, the center automaton undergoing the transition into the preterminal state. An automaton goes into a terminal (synchronized) state if both its neighbours and itself are in the preterminal state (there are some exceptions due to the difference of situations when the number of automata in the chain is odd or even). The first bisection of the chain is made as follows. The initiating signal transfers the end automaton into preterminal state and two signals start to propagate down the chain from this automaton. The first signal has the velocity 1, the second one has the velocity $1/3$ (a signal propagates with velocity $1/p$ if it passes to a neighbouring automaton after having stayed in the preceding one for p units of time). The first signal goes as far as the end of the chain, transfers the second end automaton into preterminal state and reflects, preserving the same velocity. The meeting of the reflected signal with the signal having velocity $1/3$ occurs at the center of the chain and the corresponding automaton (or two automata if the number of automata in the chain is even) goes into the preterminal state. If the reflected signal goes on down the chain and if the third signal with the velocity of propagation $1/7$ has started from the first end automaton, these signals will meet at a distance of $1/4$ from the beginning of the chain. Generally, if every automaton on assuming the preterminal state sends out a



State notation

H	L	K	K	C	C	3	3
●	○	▶	◀	⊗	⊠	▽	△

Figure 3



State notation

H	И	Ж	K	$\vec{K}$	$\overleftarrow{K}$	C ₀	C ₁	3	$\overleftarrow{3}$
○	○	●	▶	◀	◉	◻	▷	◁	

Figure 4

sequence of signals propagating with velocities $1/(2^{p+1}-1)$ and if automata occurring at the meeting point of the signals go into preterminal state, the process of successive divisions into two of emerging segments of the chain will take place. The main difficulty Levenstein so brilliantly surmounted was in organizing the sequence of signals with velocities of propagation $1/(2^{p+1}-1)$ emitted by an automaton with finite number of states. This seemingly impossible task can be done, because p_{max} is bounded and the organization of interacting signals allows the inner states of all the automata to be used for the purpose. The picture of signal propagation for the chain of 26 automata is shown in figure 3. The details concerning the solution of the problem as well as rigorous proofs can be found in the original work of Levenstein.

Myhill's problem gives rise to a whole class of problems. Thus, one can consider the synchronization problem in which the initiating signal can be applied to any of the automata of the chain (see Varshavsky, Marakhovsky and Peshchansky, in press). In this case the chain can be synchronized in time $T=2k-2-a_{min}$, where a_{min} is a distance between the nearest end automaton and the automaton to which the initiating signal is applied. The underlying automaton has 10 inner states and is in fact a slight modification of Levenstein's automaton. The picture of signal propagation for the chain of 28 automata is shown in figure 4. The approach we used for the solution of the above problems can be extended to a system of automata arranged in a rectangular lattice.

Another problem of the same type has been considered by V. Peshchansky, V. Marakhovsky and the author (in press). We shall refer to it as the voting problem. Let there be a chain of k objects each able to be in one of p states (i.e. send out one of p possible output signals). There is a chain of k automata associated with the above chain and the correspondence between the objects and automata is one-to-one. Every automaton can exchange information with its two neighbours and perceive a state of one object. The goal is to construct an automaton whose complexity is independent of a number of objects and such that, having received an initiating signal at time $t=0$, one of the end automata will go at time T into the state corresponding to the state assumed by most of the objects. The tricky part of the problem is that the same kind of automaton with the fixed number of states is required to do the job, regardless of the number of objects in the chain. Roughly speaking, no automaton is permitted to count a number of objects. Hence, we try to reach the goal by comparing the times required for a signal to go the round of automata corresponding to the objects that are in the same states. It is convenient to rearrange the signals coming to the automata from the objects in such a way that the same signals would form compact chains. The times required for the signals to come through such chains are then to be compared. Now consider a system organized in this way. At time zero, let all the automata be in an initial state H . In addition, we shall distinguish a subset of

states E_{1j} ($j=1, \dots, p$) of the set of automata states. Any signal from the subset E_1 can be chosen as the initiating signal; let it be applied to the left-end automaton. Then the transition from the initial state to the states of subset E_1 is defined as follows. If an automaton A and its neighbour to the right are in the state H , and its neighbour to the left is in a state from E_1 , then the automaton A goes into the state E_{1a} , where a is the state of the object O associated with the automaton A . In this way, the read-out of the states of the objects goes into the automata chain. The read-out finished, outputs of the objects have no further effect on the behaviour of the automata. Now turn to the process of ordering the input signals in the automata chain. The relevant algorithm groups automata which are in the same states into continuous chains, the bigger the number of a state the more to the right its chain is placed. It is sufficient for the purpose to make a change of the state numbers of the neighbouring automata whenever the right automaton's state is less than that of the left automaton. The system operation is illustrated in figure 5 and

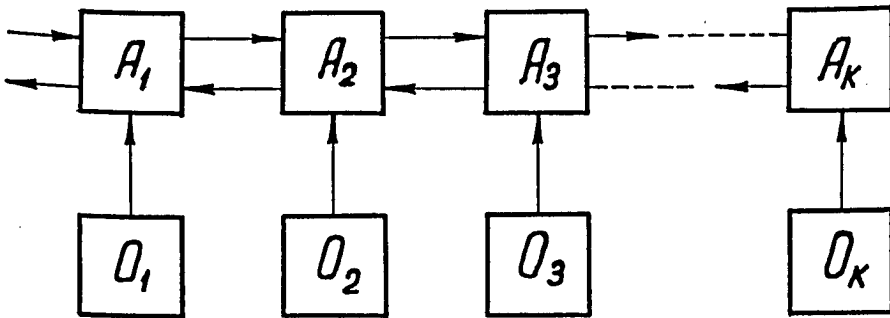


Figure 5

Table 2. The state at the front of the signal E_1 propagates down the chain of automata and just behind it ordering of the states takes place. It is not hard to see that the above ordering is unambiguous, and that there can be no situations when an automaton has to exchange the numbers of inner states with two neighbours at the same time. Having emerged at the front of the signal E_1 the state with the maximum number goes on moving with velocity equal to that of the front itself, while the state with the minimum number, having emerged at the front of the signal E_1 , moves in the opposite direction with velocity 1. Therefore, if the signal on reaching the end of the chain generates a reflected signal, the latter propagates down the ordered chain and can be used for comparing the lengths of the chains. At the end of the chain the signal E_1 generates the reflected signal E_2 , at the front of which the automata

MACHINE LEARNING AND HEURISTIC PROGRAMMING

i	1	2	3	4	5	6	7	8	9	10	11	12
t	2	3	1	2	3	2	3	1	2	3	2	1
Variables												
States of automata												
1	2 ₁											
2	2 ₁	3 ₁										
3	2 ₁	3 ₁	1 ₁									
4	2 ₁	1 ₁	3 ₁	2 ₁								
5	1 ₁	2 ₁	2 ₁	3 ₁	3 ₁							
6	1 ₁	2 ₁	2 ₁	3 ₁	3 ₁	2 ₁						
7	1 ₁	2 ₁	2 ₁	3 ₁	2 ₁	3 ₁	3 ₁					
8	1 ₁	2 ₁	2 ₁	2 ₁	3 ₁	3 ₁	3 ₁	1 ₁				
9	1 ₁	2 ₁	2 ₁	2 ₁	3 ₁	3 ₁	1 ₁	3 ₁	2 ₁			
10	1 ₁	2 ₁	2 ₁	2 ₁	3 ₁	1 ₁	3 ₁	2 ₁	3 ₁	3 ₁		
11	1 ₁	2 ₁	2 ₁	2 ₁	1 ₁	3 ₁	2 ₁	3 ₁	3 ₁	2 ₁	2 ₁	
12	1 ₁	2 ₁	2 ₁	1 ₁	2 ₁	2 ₁	3 ₁	3 ₁	3 ₁	2 ₁	3 ₁	1 ₁
13	1 ₁	2 ₁	1 ₁	2 ₁	2 ₁	2 ₁	3 ₁	3 ₁	2 ₁	3 ₁	1 ₁	3 ₁
14	1 ₁	1 ₁	2 ₁	2 ₁	2 ₁	2 ₁	3 ₁	2 ₁	3 ₁	1 ₁	3 ₁	3 ₁
15	1 ₁	1 ₁	2 ₁	2 ₁	2 ₁	2 ₁	2 ₁	3 ₁	1 ₁	3 ₁	3 ₁	3 ₁
16	1 ₁	1 ₁	2 ₁	2 ₁	2 ₁	2 ₁	2 ₁	1 ₁	3 ₁	3 ₁	3 ₁	3 ₁
17	1 ₁	1 ₁	2 ₁	2 ₁	2 ₁	2 ₁	1 ₁	2 ₁	3 ₁	3 ₁	3 ₁	3 ₁
18	1 ₁	1 ₁	2 ₁	2 ₁	2 ₁	1 ₁	2 ₁	2 ₁	3 ₁	3 ₁	3 ₁	3 ₁
19	1 ₁	1 ₁	2 ₁	2 ₁	1 ₁	2 ₁	2 ₁	2 ₁	3 ₁	3 ₁	3 ₁	3 ₁
20	1 ₁	1 ₁	2 ₁	1 ₁	2 ₁	2 ₁	2 ₁	2 ₁	3 ₁	3 ₁	3 ₁	3 ₁
21	1 ₁	1 ₁	1 ₁	2 ₁	2 ₁	2 ₁	2 ₁	2 ₁	3 ₁	3 ₁	3 ₁	3 ₁
22	1 ₁	1 ₁	1 ₁	2 ₁	2 ₁	2 ₁	2 ₁	2 ₁	3 ₁	3 ₁	3 ₁	3 ₁
23	1 ₁	1 ₁	1 ₁	2 ₁	2 ₁	2 ₁	2 ₁	2 ₁	2 ₁	3 ₁	3 ₁	
24	1 ₁	1 ₁	1 ₁	2 ₁	2 ₁	2 ₁	2 ₁	2 ₁	2 ₁	3 ₁	3 ₁	
25	1 ₁	1 ₁	1 ₁	2 ₁	2 ₁	2 ₁	2 ₁	2 ₁	2 ₁	3 ₁	3 ₁	
26	1 ₁	1 ₁	1 ₁	2 ₁	2 ₁	2 ₁	2 ₁	2 ₁	2 ₁	3 ₁	3 ₁	
27	1 ₁	1 ₁	1 ₁	2 ₁	2 ₁	2 ₁	2 ₁	2 ₁	2 ₁	3 ₁	3 ₁	
28	1 ₁	1 ₁	1 ₁	2 ₁	2 ₁	2 ₁	2 ₁	2 ₁	2 ₁	3 ₁	3 ₁	
29	1 ₁	1 ₁	1 ₁	2 ₁	2 ₁	2 ₁	2 ₁	2 ₁	2 ₁	3 ₁	3 ₁	
30	1 ₁	1 ₁	1 ₁	2 ₁	2 ₁	2 ₁	2 ₁	2 ₁	2 ₁	3 ₁	3 ₁	
31	1 ₁	1 ₁	1 ₁	2 ₁	2 ₁	2 ₁	2 ₁	2 ₁	2 ₁	3 ₁	3 ₁	
32	1 ₁	1 ₁	1 ₁	2 ₁	2 ₁	2 ₁	2 ₁	2 ₁	2 ₁	3 ₁	3 ₁	
33	1 ₁	1 ₁	2 ₁									
34	1 ₁	2 ₁										
35	2 ₁											

Notation												
H	E ₁₁	E ₁₂	E ₁₃	E ₂₁	E ₂₂	E ₂₃	E ₃₁	E ₃₂	E ₃₃	E ₄₁	E ₄₂	E ₄₃
	1 ₁	2 ₁	3 ₁	1 ₂	2 ₂	3 ₂	1 ₃	2 ₃	3 ₃	1 ₄	2 ₄	3 ₄

Table 2

undergo the transition from the state E_{1a} into the state E_{2a} . The underlying idea for comparing the lengths of the chains is illustrated in figure 6. An example of the behaviour of the whole system is given in Table 2. For the sake of simplicity we do not consider the case when there are several chains of states of the same length. More detailed discussion can be found in Varshavsky, Marakhovsky and Peshchansky (b). Note finally that the automata providing the solution have $4p+1$ inner states and the time required for the solution is equal to $2(k+c)$, where c is the maximum number of the objects in the same state.

The most interesting point in the above examples is apparently that a collective of automata is able to handle the problem that is more complex than the simple sum of problems which each single automaton of this complexity is able to handle on its own. This is due to organized interaction between the automata. In the synchronization problem each automaton has only 9 inner states and still it manages to generate the sequence of time delays $2^{p+1}-1$, which is essentially impossible for an isolated automaton with such a number of states. However, it would be stretching the notion to consider the above problems as those of a control nature. Their work is associated with rigid programs and any external disturbances, for instance applying two initiating signals to a pair of different automata in the synchronization problem, lead to a complete failure of the system's work. There is no interaction between the system and environment, which is so characteristic of control, in the problems considered above. A more natural statement of the synchronization problem from the point of view of control is contained in a paper by Gel'fand and Tsetlin (1960) on continual models of control system. I. P. Lukashevich has used these models for simulation of the synchronous work of heart muscle (see Lukashevich, 1963, 1964). On the other hand the problems we have just considered admit full analytical treatment of the work of interacting automata systems, which becomes formidable in the study of automata behaviour in an external environment. Besides, as has been already noted, these models have exemplified automata interaction under which each automaton used the memory of the whole collective for solution of its local problem and was able to handle problems otherwise beyond its power. It is worth noting that this phenomenon is characteristic of human collectives as well. In studies of the behaviour of ants it is known as 'the group effect' (see Khalifman, 1958). In the control problems concerned with behaviour in an external environment it is usually very difficult to organize purely logical interaction between members of a collective. In this case considerably greater effect can be produced by organizing interaction through local utility functions (estimators). An important point to stress is that the local utility functions do not generally coincide with the performance quality criterion. Two examples of such organization follow.

The first is connected with arranging a queue discipline in a queueing system (see Varshavsky, Meleshina and Tsetlin (1968)). It is a tradition in

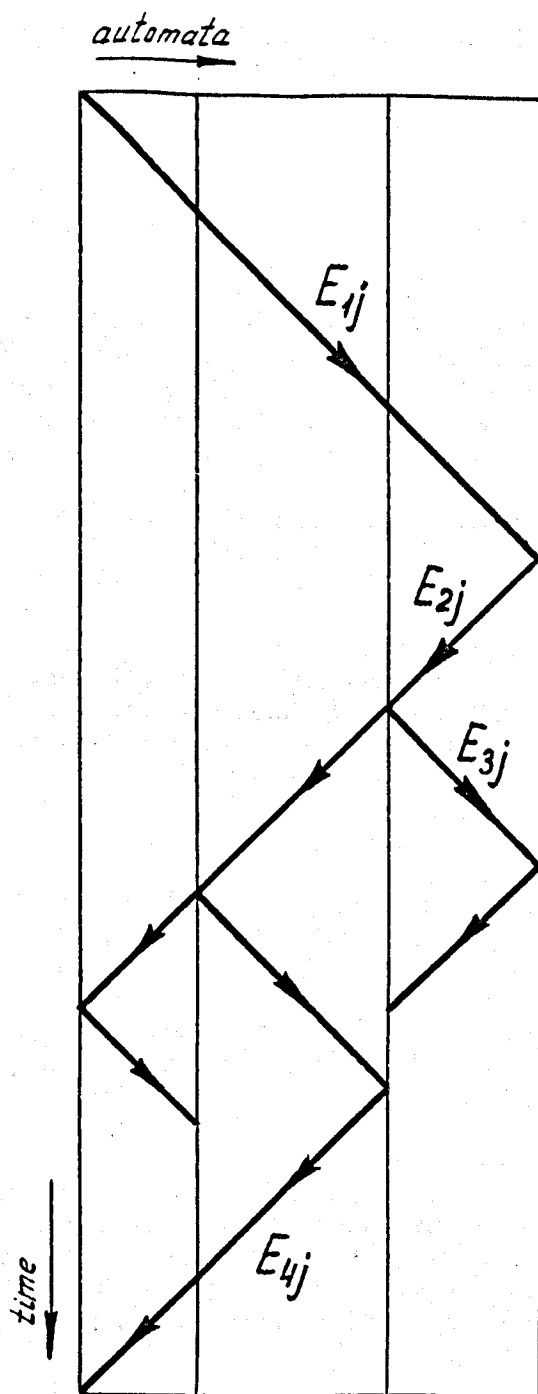


Figure 6

queueing theory to illustrate the ideas with the telephone exchange operation. Let us therefore consider a telephone exchange with k available channels and a subscribers. The rate at which demands for connection from the i th subscriber arrive is λ_i (for simplicity arrivals are supposed to form a Poisson process). The mean length of connection of the i th subscriber is equal to τ_i . Connections are made in accordance with the queue which is the same for all subscribers and channels. Analysis of queueing systems quality, understood, for example, as the expected waiting time or expected queue size, can be improved by introducing a system of priorities (see Saaty, 1961). It is known that in the case of equal rates the highest priority is to be granted to a subscriber with the shortest mean service time. But distribution of priorities requires *a priori* information concerning the probabilistic characteristics of input flows that are often not available or essentially change with time. Our aim is to establish the system of priorities *in line*, i.e. while the system operates. In the construction to be described, each channel 'decides' on the order of priority for the subscriber independently of other channels. We know that preference should be given to 'profitable' subscribers whose conversations are short and, in case there are several such subscribers, to those whose calls are more frequent. With this in mind let us introduce the fee to be paid for a conversation irrespective of its length. It is convenient to choose the fee equal to the mean length of the conversation in the system, T_m .¹ At the moment the subscriber is switched on to the channel an amount of money equal to T_m is placed into a special counter ('money-box'). As the conversation progresses the amount reduces by unity per one unit of time. If the duration of the conversation turns out to be less than T_m , then there is a certain sum left in the money-box of the channel and this gives the commensurate priority to the subscriber next time he makes a connection; otherwise he is given no priority. The automata of the linear tactics type can be employed as the money-boxes. An act of the automata is determined by the number of the subscriber whose 'money' is now in the money-box. State transition graphs are shown in figure 7. Input signal C_1 corresponds there to the arrival of the call at the channel while the automaton is switched on; signal C_2 to when the automaton is busy and switched on; and C_3 to when it is switched out. The automaton may change its acts (i.e. change the number of the subscriber having priority for this channel) only when in the state numbered 0. The money-box can be of limited capacity; for the automaton, the capacity of the money-box corresponds to the number of its inner states (the memory capacity).

Introducing a kind of competition between the subscribers makes it possible for the channel to secure more profitable subscribers. For this purpose there are two automata at every channel—the principal one (A) and the reserve one (B). Let a_j and b_j be the numbers of acts (subscribers) of principal and

¹ The approximate value of T_m is supposed to be known; otherwise it is to be determined while the operation unfolds.

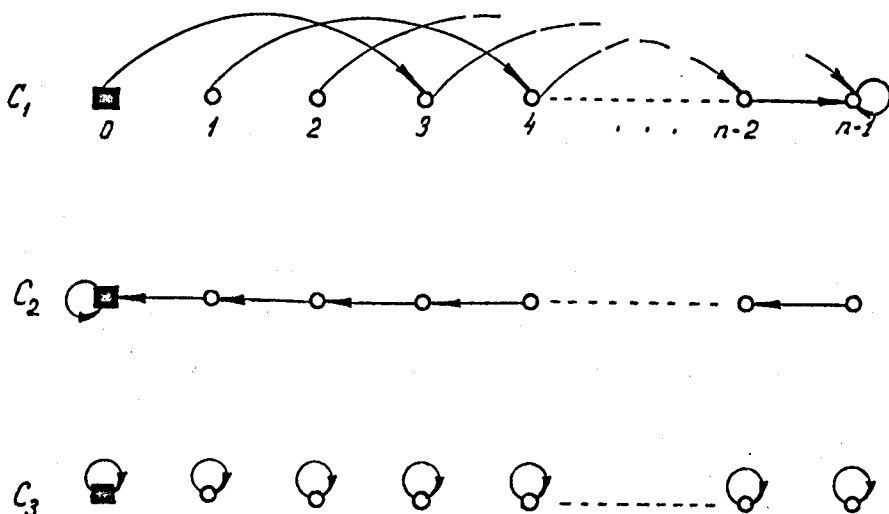


Figure 7

reserve automata of the j th channel respectively. If there are no idle channels, an arriving call takes its place in the common waiting line and awaits service. If some channel becomes free and the number of its principal automaton's state coincides with the number of one of the waiting calls, then this call is served out of turn. Otherwise there are two possibilities: (1) if the number of the reserve automaton state and the subscriber's number coincide, then the interaction between the subscriber and reserve automaton goes in accordance with the graph shown in figure 7; (2) if the number of the reserve automaton state and the subscriber's number do not coincide, then the current state of the reserve automaton becomes zero and the automaton begins to examine another call. When the channel becomes free the numbers of inner states of principal and reserve automata are compared, and the automaton with the bigger number is given the role of the principal automaton. Note that assigning channels to a certain group of subscribers improves such an important characteristic of telephone exchange as the number of commutations. A commutation in the channel is said to take place if two calls with different numbers have successively arrived. Commutations of this sort generate considerable disturbances and reduce audibility.

Computer simulation of the above system has been carried out; the results of the experiments are given in Tables 3-7. The results given in Table 3 corresponds to the system whose probabilistic characteristics do not change in time. The following notation is adopted in the Tables: n is the automaton memory capacity; W_0 is the value of the fee; W_n^{-1} is the mean waiting time

number of channels	number of sub- scribers	flow characteristics		n	N	W_0	W_n^1	W_n^2	S_0	S
		λ_i	τ_i							
1	2	4	$\lambda_i = \frac{1}{10} (1 \leq i \leq 4)$ $\tau_1 = \tau_3 = 2$ $\tau_2 = \tau_4 = 12$	8	4	50	31.0	31.0	0.98	0.85
2	4	4	$\lambda_1 = \lambda_3 = \frac{3}{10}$ $\lambda_2 = \lambda_4 = \frac{1}{10}$ $\tau_1 = 2; \tau_3 = 12$ $\tau_2 = 7; \tau_4 = 7$	8	4	50	31.5	34.4	0.97	0.70
3	4	4	$\lambda_i = \frac{1}{8} (1 \leq i \leq 4)$ $\tau_1 = \tau_3 = 2$ $\tau_2 = \tau_4 = 12$	8	4	50	31.2	31.0	0.98	0.75
4	4	6	$\lambda_1 = \frac{1}{4}; \lambda_i = \frac{1}{20} (2 \leq i \leq 6)$ $\tau_1 = 2$ $\tau_i = 12 (2 \leq i \leq 6)$	8	4	50	31.2	31.0	0.96	0.68
5	4	7	$\lambda_1 = \lambda_3 = \frac{1}{8}$ $\lambda_i = \frac{1}{20} (3 \leq i \leq 7)$ $\tau_1 = \tau_3 = 2$ $\tau_i = 12 (3 \leq i \leq 7)$	8	4	50	34.0	31.0	0.98	0.80
6	4	8	$\lambda_i = \frac{1}{10} (1 \leq i \leq 8)$ $\tau_1 = 2; \tau_3 = 12$ $\tau_i = 7; (3 \leq i \leq 8)$	8	4	50	36.0	38.5	0.98	0.90
7	4	16	$\lambda_i = \frac{1}{10} (1 \leq i \leq 4)$ $\lambda_j = \frac{1}{48} (5 \leq j \leq 16)$ $\tau_1 = \tau_3 = 2; \tau_2 = \tau_4 = 12$ $\tau_i = 7 (5 \leq i \leq 16)$	8	4	50	34.0	35.2	0.98	0.85
8	8	16	$\lambda_i = \frac{1}{10} (1 \leq i \leq 16)$ $\tau_i = 2 (1 \leq i \leq 8)$ $\tau_j = 12 (9 \leq j \leq 16)$	8	4	50	32.0	31.0	1.00	0.90

Table 3

obtained experimentally; W_n^2 is the mean waiting time calculated according to formulae obtained for the case when priority does not depend on the channel's number and is given to the subscriber with the shortest mean call duration; S_0 is the mean number of the commutation for the system without priorities; and S is the mean number of commutations for the system with elaboration of priorities. In Tables 4 and 5, four subscribers have two possible sets of call durations, (2,2,12,12) and (12,12,2,2), that form a simple Markov chain with the transition probability equal to δ . In Table 4 the dependence of system parameters on δ is given, and in Table 5 the dependence on the memory capacity of automata (the number of inner states). Tables 6 and 7 show the same dependences for the case when call durations 2 and 12 form independent Markov chains at every channel. The data given in Table 3 show that in three cases experimental mean times are less than the calculated ones. This is explained by the fact that in these three cases we are dealing with three different call durations. In the system under consideration, a system of priorities is elaborated whereas the calculations were carried out under the assumption

n	N	number of channels	flow characteristics		δ 0.002	δ 0.008	δ 0.032	δ 0.100	δ 0.320	
			state 1	state 2						
			$\tau_1=\tau_2=2$ $\tau_3=\tau_4=12$	$\tau_1=\tau_3=12$ $\tau_2=\tau_4=2$						
4	4	4	$\lambda_1=\lambda_2=\lambda_3=\lambda_4=\frac{1}{8}$		31.2	31.4	37.6	41.5	45.0	W_n^1
					0.82	0.84	0.86	0.92	0.95	S
					50.0	50.0	50.0	50.0	50.0	W_0
					0.98	0.98	0.98	0.98	0.98	S_0

Table 4

δ	N	number of channels	flow characteristics		1	2	3	4	8	10	12	14	n
			state 1	state 2									
			$\tau_1=\tau_2=2$ $\tau_3=\tau_4=12$	$\tau_1=\tau_3=12$ $\tau_2=\tau_4=2$									
$\frac{1}{64}$	4	4	$\lambda_1=\lambda_2=\lambda_3=\lambda_4=\frac{1}{8}$		48.4	43.1	39.5	36.2	38.4	39.5	41.3	42.3	W_n^1
					50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	W_0

Table 5

n	N	number of channels	flow characteristics		δ 0.002	δ 0.004	δ 0.008	δ 0.016	
			state 1	state 2					
			$\tau_i=2$	$\tau_i=12$					
4	4	4	$(i=1, 2, 3, 4)$		180.5	80.4	60.8	33.0	W_n^1
			$\lambda_1=\lambda_2=\lambda_3=\lambda_4=\frac{1}{8}$		0.79	0.78	0.78	0.77	S
					240.0	120.0	90.0	55.0	W_0
					0.99	0.99	0.99	0.99	S_0

Table 6

δ	N	number of channels	flow characteristics		1	2	3	4	8	14	n
			state 1	state 2							
			$\tau_i=2$	$\tau_i=12$							
$\frac{1}{64}$	4	4	$(i=1, 2, 3, 4)$		52.5	45.7	42.1	38.0	38.4	41.3	W_n^1
					55.0	55.0	55.0	55.0	55.0	55.0	W_0

Table 7

that there was only one priority, that for the subscriber with the shortest call duration. The experimental results show that the system described above operates quite satisfactorily.

In all the examples of collective behaviour we have considered, the local problem was handled by a finite automaton. It is evident that devices other than finite automata can be used for organization of collective behaviour. Moreover, to make progress in the study of control organization for a complex system we ought to use any effective means for the solution of local problems.

We now turn to the example where our local tool will be a conventional algorithm of feedback control rather than a finite automaton. This example was studied by the author in collaboration with M. Meleshina and V. Perekrest (in press) in connection with the problem of allocation (*see* Bellman and Dreyfus, 1962). We shall consider two possible statements of the problem of one-dimensional allocation of resources which, as we shall see, can be treated in effect in one and the same way. Suppose we have a certain quantity of an economic resource that can be money, fuel, raw material, etc., and a number of consumers, who can, by utilizing this resource, achieve a certain economic effect referred to as the 'return'. It is supposed that the returns obtained by different consumers can be measured in a common unit. The problem is that of dividing our resource so as to maximize the total return. A number of mathematical devices such as, for example, dynamic programming (Bellman and Dreyfus, 1962) and the Lagrange multipliers method, can be employed for the solution of the problem. However, all the techniques known so far require the utility functions (i.e. the dependences of the returns of the consumer from the quantity of the resource obtained) to be known in advance. In addition the decision-making process is centralized. We shall consider the allocation problem under the assumption that only current values of the returns are known and that the optimal allocation is to be found as the process of interaction between the consumers unfolds. But first let us give the second statement of the allocation problem. Consider a control system consisting of the object to be controlled (the plant) and the controller. The plant and the controller are linked by a communication channel. There are several servo units in the system and the channel is such that the units are switched on to the controller in succession. So we have a multiloop control system, the i th control loop being closed periodically for time τ_i . The bigger the value of τ_i , the better the performance quality of the i th loop. The problem is that of dividing the controller time (period T) between the control loops so as to maximize (minimize) some performance criterion. Thus, period T can be interpreted as a resource, and let us suppose that we have to minimize the min-max of some error functions associated with performance quality of the control loops. Now consider a possible organization of collective behaviour for problems of this kind. Let τ_i be the share of the resource given to the i th consumer and t_i be the share demanded by him ($0 \leq \tau_i \leq 1$, $0 \leq t_i \leq 1$). If the

i th consumer asks for t_i of the resource then he is given

$$\tau_i = \begin{cases} t_i & \text{if } \sum_{j=1}^n t_j \leq 1 \\ \frac{t_i}{\sum_{j=1}^n t_j} & \text{if } \sum_{j=1}^n t_j > 1 \end{cases}$$

Now our problem is reduced to that of choosing a new local utility function for each consumer to be maximized (minimized) in the process of elaboration of his demand for the resource, which provides desired allocation. Let us begin with maximization of the total return. Introducing Lagrange multipliers enables one to write the system of equations for the optimal allocation of the resource:

$$\begin{aligned} \frac{dy_i(\tau_i)}{d\tau_i} - \lambda &= 0 \\ \sum_{j=1}^n \tau_j - 1 &= 0 \end{aligned} \quad (1)$$

In (1) the Lagrange multiplier λ has the significance of the price of a unit of the resource, and the system of equations can be deduced as follows. Let the local utility function be the difference between the return and the cost of the resource obtained. Then the maximum of the utility functions is determined by the system

$$\frac{\partial}{\partial t_i} \left[\phi_i(\tau_i) - \lambda \tau_i \right] = \frac{\partial \tau_i}{\partial t_i} \left[\frac{d\phi_i(\tau_i)}{d\tau_i} - \lambda \right] = 0 \quad (2)$$

and control can be realized through changing λ —the price of the resource. An implementation of such control is suggested by a possible approach to the solution of system (1). Denote

$$\frac{d\phi_i(\tau_i)}{d\tau_i} = \gamma_i(\tau_i) \text{ and } \gamma_i^{-1}(\lambda) \text{—inverse function for } \gamma_i(\tau_i)$$

Then the optimal value of λ can be found by solving the equation with respect to λ :

$$\sum_{j=1}^n \gamma_j^{-1}(\lambda) - 1 = 0. \quad (3)$$

From equations (1) to (3) the organization of control and interaction immediately follows:

$$\begin{aligned} t_i &= \kappa_1 \left[\frac{d\phi_i(\tau_i)}{d\tau_i} - \lambda \right] \quad 1 \leq i \leq n \\ \lambda &= \kappa_2 \left[\sum_{j=1}^n t_j - 1 \right] \end{aligned} \quad (4)$$

or, in a different form, which was actually used for computer simulation:

$$\begin{aligned}\Delta t_i &= \kappa_1 \left[\frac{\Delta \phi_i(\tau_i)}{\Delta \tau_i} - \lambda \right] \\ \Delta \lambda &= \kappa_2 \left[\sum_{j=1}^n t_j - 1 \right]\end{aligned}\quad (5)$$

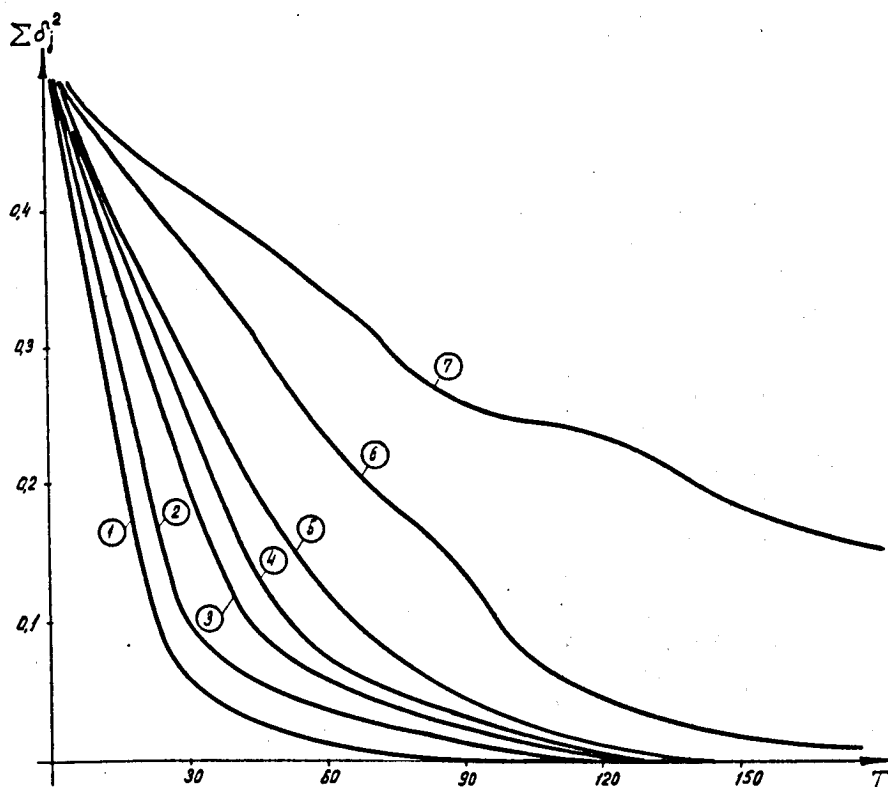
Variation in time of the distance between the current allocation vector and the optimal allocation vector for 10 consumers and various gains k_1 and k_2 are shown in figure 8, the dependences of the return from obtained amount of the resource being given by

$$\begin{aligned}\phi_1 &= 2.125 \ln(1 + \tau_1) - 0.1 \\ \phi_2 &= 2.125 \ln(1 + \tau_2) - 0.2 \\ \phi_3 &= \sqrt{2\tau_3} - 1 \\ \phi_4 &= \sqrt{3\tau_4} - 1 \\ \phi_5 &= \frac{6}{\pi} \sin \frac{\pi}{3} \tau_5 \\ \phi_6 &= 2 \sin \tau_6 \\ \phi_7 &= \tau_7 (2.125 - \tau_7) \\ \phi_8 &= \frac{1}{2} \tau_8 (4.25 - \tau_8) \\ \phi_9 &= 2.25 (1 - e^{-\tau_9}) \\ \phi_{10} &= 2.25 (1 - e^{-2\tau_{10}})\end{aligned}$$

At this juncture questions concerning stability of such a system arise. Analysis of stability is beyond the scope of the present paper. We shall note only two points. The first one is associated with existence of an equilibrium point in the system as such. This question can be considered as the problem of the existence of an equilibrium in a non-cooperative n -person game. The second one is the problem of the stability of control process that is extensively treated in the stability theory of differential (and difference) equations. In the above example we employed the feed-back control algorithm as the local algorithm. However, more complex local algorithms might be required to allow for the presence of time lag between the moment of demand arrival and that of supplying a corresponding share of the resource, or they might be due to a specific form of local return functions.

Now we turn to the problem of time distribution in the control system. Putting aside the cases when there exist solutions on the boundaries one may say that for limited resource (T) min-max distribution is achieved when errors in all the loops are equal. This immediately leads to the following organization of the control:

$$\begin{aligned}\Delta t_i &= \kappa_1 [\phi_i(\tau_i) - \lambda] \\ \Delta \lambda &= \kappa_2 [1 - \sum t_j]\end{aligned}\quad (6)$$



- ① — $K_1 = 1$
 $K_2 = 0.125$
- ② — $K_1 = 1$
 $K_2 = 0.0625$
- ③ — $K_1 = 0.5$
 $K_2 = 0.0625$
- ④ — $K_1 = 1.5$
 $K_2 = 0.0625$
- ⑤ — $K_1 = 0.25$
 $K_2 = 0.0625$
- ⑥ — $K_1 = 0.125$
 $K_2 = 0.0625$
- ⑦ — $K_1 = 0.0625$
 $K_2 = 0.0625$

Figure 8

It is not hard to see that systems (5) and (6) are identical. The latter is characteristic in its use of a difference scheme with the step size equal to the period of commutation. Here, as above, fulfilment of some natural conditions on the local utility functions is necessary for the existence and stability of the solution.

What seems important in such organization of control is a very small amount of information circulating in the system: the consumer reports what quantity of the resource he needs and the central controller gives in reply the value of λ . The organization described above makes it easy to improve the reliability of the system without rearranging the interaction in it. All one has to do is to change the algorithm for finding λ . It is noteworthy that the stability of the system does not depend on a given total demand determining λ . This is clear from a consideration of the linear approximation of the system near the equilibrium point. If a failure occurs, then a consumer will demand the same amount of the resource irrespective of the amount obtained. Adding one more control parameter in (4) enables us to attain the optimal distribution. The system in this case is described by equations

$$\dot{i}_i = \kappa_1 \left[\frac{d\phi_i(\tau_i)}{d\tau_i} - \lambda \right]$$

$$\dot{\lambda} = \kappa_2 [\Sigma t_j - c]$$

$$\dot{c} = \kappa_3 \left(\Sigma \frac{\partial}{\partial c} \phi_j(\tau_j) \right)$$

But only some preliminary experimental results are available for this case.

Both problems just described—that of priority elaboration and that of resource allocation—are characterized by the fact that there is no direct link between local criteria and criterion of the behaviour of the whole system in the organization of collective behaviour. In the first problem the local algorithm 'gives preference to the most profitable client', yet this permits the reduction of the mean waiting time for all clients. In the second problem the local algorithm sees to it that the partial derivative of the local return (or, in the second version, the value of the return) will not deviate from a prescribed level (price), while the local algorithm of price elaboration provides the equality of supply and demand. This distribution of tasks in the system maximizes the total return.

To keep the report within a reasonable size I have contented myself with a few examples. However, it is perhaps clear that analytical and conceptual treatment of a problem enables one to establish local estimations providing the solution. It would seem that the question of establishing local estimations is of paramount importance in heuristic methods.

Acknowledgments

In conclusion I take this opportunity to express my sincere thanks to Professors N.G. Boldyrev and L.I. Rosonoer, to Dr M.M. Bongard, to Dr G.V. Epifanov, and to Mr M.S. Epel'man for (sometimes very severe) criticism. They found some aspects of the report trivial, others arguable. Unfortunately, in trying to eliminate these defects, I have not been entirely successful, but I hope that my main points will be clear.

REFERENCES

- Bellman, R. & Dreyfus, S. (1962), *Applied dynamic programming*. Princeton, N.J.: Princeton University Press.
- a. Gel'fand, I.M., Gurfinkel, V.S. & Tsetlin, M.L. (1962), O taktikah upravleniya slozhnymi sistemami v svyazi s fiziologiyey. *Biologicheskie aspekty kibernetiki*. Moscow: Akad. Nauk SSSR.
- Gel'fand, I.M. & Tsetlin, M.L. (1960), O kontinual'nykh modelyakh upravlyayushchykh sistem. *Doklady Acad. Nauk. SSSR*, 131, 6, 1242.
- Gel'fand, I.M. & Tsetlin, M.L. (1962), O nekotorykh sposobakh upravleniya slozhnymi sistemami. *Uspekhi matematicheskikh nauk*, 17, 1(103).
- Khalifman, I.A. (1958), *Parol' skreshchennykh antenn*. M., Detgiz.
- Levenstein, V.I. (1965), Ob odnom metode resheniya zadachi sinkhronizatsii tsepi avtomatov za minimalnoe vremya. *Problemy peredachi informatsii*, 1, 4.
- Lukashevich, I.P. (1963), Issledovanie na EVM nepreryvnykh modeley upravlyayushchykh sistem. *Biofizika*, VIII, 6.
- Lukashevich, I.P. (1964), Ritmika odnorodnoy tkani i modelirovanie povedeniya serdechnoy myshtsi. *Biofizika*, IX, 6.
- Moore, E.F. (1964), The firing squad synchronization problem. *Sequential machines*. Reading, Mass.: Addison-Wesley Publishing Co. Inc.
- Saaty, T.L. (1961), *Elements of queueing theory with applications*. New York, Toronto, London: McGraw-Hill.
- , Tsetlin, M.L. & Varshavsky, V.I. (1966), Automata and models of collective behaviour. *Proceedings Third Congress IFAC*, London, paper 35.E.
- Varshavsky, V.I., Marakhovsky, V.B. & Peshchansky, V.A. (in press (a)), Nekotorye varianty zadachi o sinkhronizatsii tsepi avtomatov. *Problemy peredachi informatsii*.
- Varshavsky, V.I., Marakhovsky, V.B. & Peshchansky, V.A. (in press (b)), The voting problem in chains of automata. *Engineering Cybernetics*.
- Varshavsky, V.I., Meleshina, M.V. & Perekrest, V.T. (in press), Kollektivnoe povedenie v zadache o raspredelenii resursov. *Avtomatika i telemekhanika*.
- Varshavsky, V.I., Meleshina, M.V. & Tsetlin, M.L. (1965), Povedenie avtomatov v periodicheskikh sluchaynykh sredakh i zadacha sinkhronizatsii pri nalichii pomekh. *Problemy peredachi informatsii*, 1, 1.
- Varshavsky, V.I., Meleshina, M.V. & Tsetlin, M.L. (1968), Organizatsiya distsipliny ozhidaniya v sistemakh massovogo obsluzhivaniya s ispol'zovaniem kollektivnogo povedeniya avtomatov. *Problemy peredachi informatsii*, 4, 1.
- Waksman, A. (1966), An optimum solution to the firing squad synchronization problem, *Inf. Control*, 9, 1.

MAN-MACHINE INTERACTION

A Comparison of Heuristic, Interactive, and Unaided Methods of Solving a Shortest-route Problem

Donald Michie

Department of Machine Intelligence and Perception
University of Edinburgh

J.G.Fleming

University of Manchester

and

J.V.Oldfield

Department of Computer Science
University of Edinburgh

INTRODUCTION

We have compared three methods of solving a problem: (1) a human being equipped with paper and pencil, (2) a computer equipped with a heuristic problem-solving program, (3) a human being equipped with a computer-driven display and light pen, and a program for editing and evaluating trial solutions. It seems obvious, and is certainly often stated, that the co-operative combination of man and machine should be more effective than either man or machine alone. Detailed quantitative assessments of the presumed advantage are rather rare. Ideally one would like to survey a wide variety of different design problems in this way. We have as yet only tackled problems of one category, the 'Travelling Salesman', but the results seem of sufficient interest to report at this stage.

Graph Traverser

The heuristic problem-solving program referred to above is the Graph Traverser (Doran and Michie, 1966; Doran, 1968). The program can be applied to any search problem which involves discrete steps. The user must provide a description of the class of states and transformations which define the problem. It was originally developed using simple sliding block puzzles, and we shall devote some remarks to one of these, the Eight-puzzle. In figure 1, the arrow is labelled with the minimum number of unit steps required to solve

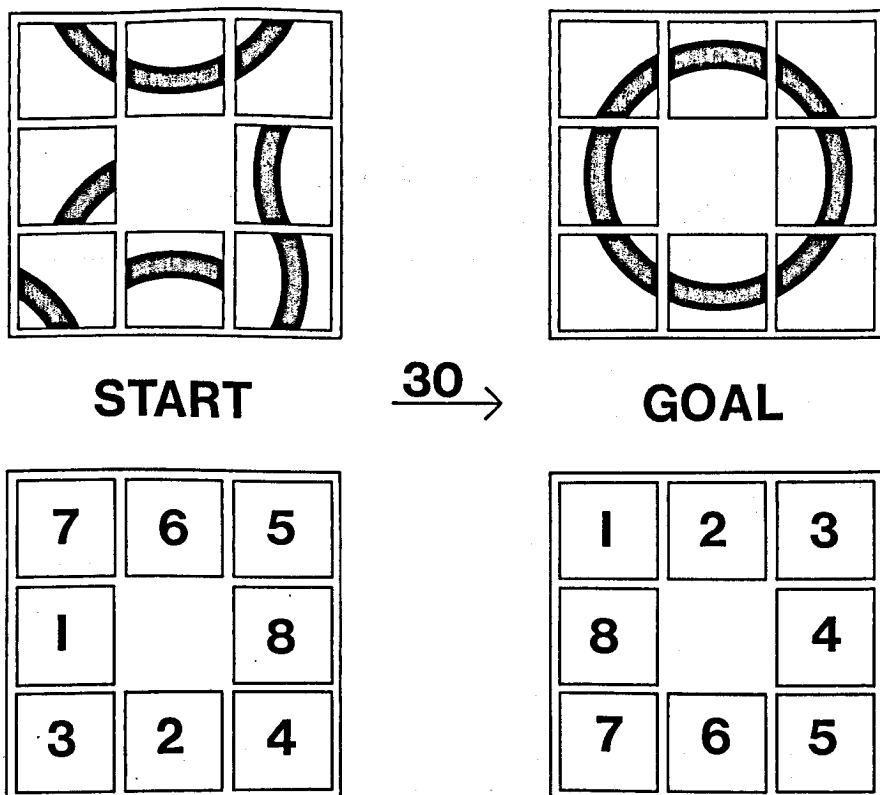


Figure 1. An Eight-puzzle problem with a minimal path of 30 moves, shown both in geometrically and numerically coded forms

the case chosen for illustration. The 20160 different soluble configurations fall into 1439 distinct symmetry classes, of which 12 have minimal paths of 30 (Schofield, 1967). Two variant forms of representation, geometrical and numerical, are shown in the figure, and both forms were used in tests of human problem-solving behaviour. The form of visual representation turned out not to have important effects on solving efficiency, except that a change from one to the other was accompanied by a marked slowing of the rate of making moves.

Solution efficiency, expressed as $100 \times (\text{minimal no. of moves}) / (\text{actual number of moves})$, proved to be independent of the difficulty of the puzzle, measured by the length of the minimal path. Under intensive practice the

average efficiency of thirty-one subjects rose from 30 per cent to about 50 per cent. The best three subjects chosen from these were able to improve with further practice from about 65 per cent to 75 per cent. None did as well as our colleague Mr R. A. Chambers, who consistently averaged 90 per cent (95 per cent in the 'digits' rather than 'circles' representation).

The Graph Traverser, using unit steps as the total repertoire of state-transformations, and a short-term memory capacity of 100 states, also averages about 90 per cent when equipped with a well-designed evaluation function (Ross, 1967). It would seem a natural step to measure the further gain to be got from man-machine co-operation by testing, e.g. Chambers harnessed to the Graph Traverser in its on-line mode (*see* Doran, 1968). This would, however, be uninformative, since the levels attained by human and machine methods *without* the benefit of co-operation are so high as to leave little margin for improvement. We therefore regarded the sliding-block work as a preliminary skirmish and turned our attention to a difficult and industrially important problem—the 'Travelling Salesman'—on which the Graph Traverser program has already been used with some success.

MATERIALS AND METHODS

Minimum traverse version of the Travelling Salesman

In the version of the Travelling Salesman which we will consider, it is required to find a shortest route linking n points scattered over a two-dimensional surface so that a complete tour is formed, i.e., each point is joined by straight lines to just 2 other points. The 'cost' of a solution is thus proportional to the sum of the lengths of n straight lines. In the general form of the Travelling Salesman the costs are given by a cost matrix, which may be arbitrary. We are here concerned with the special case where the cost of joining point i to point j can be computed as $\sqrt{\{(x_i - x_j)^2 + (y_i - y_j)^2\}}$. The relevant area of application in the industrial context is the 'minimum traverse' criterion for choosing the route which a numerically-controlled machine tool is to follow in carrying out some repetitive operation. Here the points represent holes to be punched or drilled at specified places in a sheet of metal. The shorter the tour performed by the tool, the less, in general, the time consumed in the operation.

Shen Lin's method

No algorithmic solution to the Travelling Salesman problem is known. Good solutions can be obtained by human solvers working with paper and pencil. Various heuristic methods of computer solution have been proposed, of which the method of Shen Lin (1965) is particularly simple and effective. An improvement on his results has been obtained by embodying the set of state-transformations used by Shen Lin in the 'develop' procedure of the Graph Traverser (Doran, 1968). The modification thus introduced can be described as follows. Shen Lin's basic operation consists in breaking a complete tour

in three places and rejoining the severed ends in all possible ways. For a given tour there are $\frac{n!}{(n-3)13!}$ ways of selecting the three links to be broken, so that any given 50-point tour has of the order of 100,000 descendants which can be generated by applying the Shen Lin set of transformations. Shen Lin's program starts from a randomly constructed tour and applies transformations to this trial solution in a random order until a descendant of lower cost than its parent is generated. The process is then restarted on the new candidate. When a solution is found which exhausts the entire set of transformations without producing an improvement, search terminates and is restarted from a new randomly generated trial solution. After a given number of such randomly initiated searches, the best of the terminal solutions is taken.

Extension of Shen Lin's method by use of the Graph Traverser

The consequence of using the Graph Traverser is that after failing to improve a given solution the search is *not* immediately terminated and restarted from a

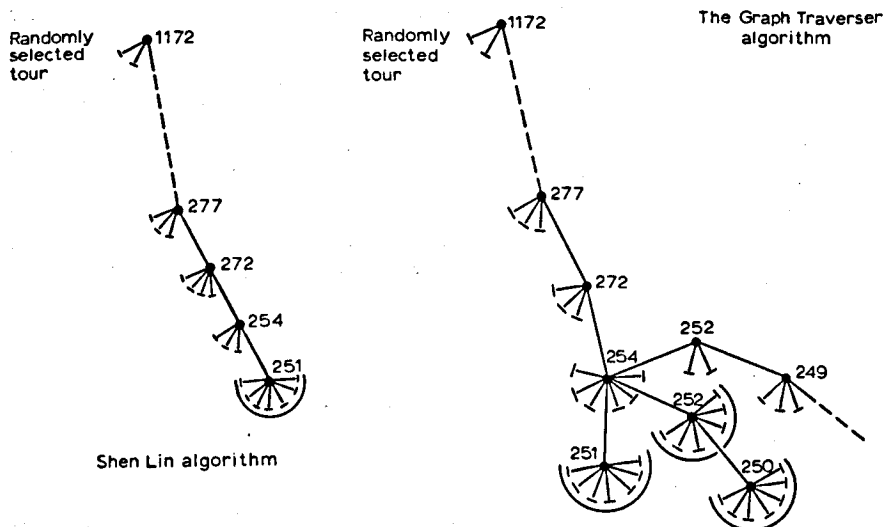


Figure 2. Two algorithms for the Travelling Salesman problem. Trial solutions are successively generated and each is used as the starting point of a new trial solution so long as improvement continues.

In Shen Lin's version, search terminates at the solution from which no improved immediate descendant can be generated. In the general purpose problem-solving program, the Graph Traverser, the distinguishing feature is continuation beyond this point by the launching of new searches from the previous best solution

randomly generated state. Instead it is restarted from the best previously generated solution to which all possible transformations have not yet been applied. The effect of this back-tracking feature depends on the length of the back-tracking step. Taking the total length of the search tree from the start to the Shen Lin terminus as n , restarting the search from level $n-1$ proved ineffective for 50-point problems: the new search almost invariably ended at the same local optimum as the old search. But when the back-track level was set to 100 (n ranged from 150–180) the benefits were substantial. The contrast between the Shen Lin and the Graph Traverser search algorithms is illustrated in figure 2.

Solution by unaided and computer-aided humans

Five 50-point problems were constructed, choosing co-ordinates by random sampling from a uniform distribution. The same five problems were given to four human subjects of whom two were undergraduate students, one was a member of the research staff, and one of the secretarial staff. The problems were presented as assemblages of dots, each problem being set out on a separate sheet of graph paper. The subject was allowed pencil, india rubber and straight edge but no measuring instruments. Unlimited time was allowed, the average time taken being a little more than an hour for the total of five problems. The same subjects were brought back after the lapse of several days and given the opportunity to improve their solutions by the interactive use of a computer-driven display and light pen as shown in plate 1. The program, written in assembly language by J.V. Oldfield for the PDP-7 computer, enables the user to join one point to another in a straight line, to delete lines, and at any stage to call for the 'cost' (i.e. the total line length) of the currently displayed trial solution to be output on the teletype. In mode 1 only these facilities are available. We shall refer to this 'no back-track' version as TS1. In the 'back-track' mode, developed by J.G. Fleming, the user has the facility of restoring the display to the best solution obtained so far, this being stored internally. In addition he has the facility of saving any sub-optimal state which he particularly likes, and returning to this from any subsequent state. Only one such sub-optimal solution may be thus pigeon-holed at any one time. The second, back-track mode, referred to below as TS2, can be seen to be essentially a Graph Traverser search, in which the user takes the place of the program's develop procedure. This manner of exploiting the human solver as an adjunct to a problem-solving program has been termed by Collins (1964) the 'human sub-routine'.

After the lapse of a further week or so, the same subjects were allowed to improve their previous solutions by using a program with the full save-and-restore facilities described above. The time taken by subjects when using the display program was very much greater than for their paper and pencil attempts, in some cases running to several hours for the completion of all five problems.

RESULTS

A summary of the results is set out in Table 1, which shows averages over the 4 subjects. The performance of the Graph Traverser algorithm is set out for comparison and in an additional row are tabulated the results obtained by R. A. Chambers, the especially gifted problem-solver whom we mentioned earlier.

<i>problem</i>		A	B	C	D	E	<i>average</i>
means of 4 subjects	unaided	4469	4641	4723	4300	4276	4482
	with ts1	4363	4523	4525	4177	4062	4330
	with ts2	4220	4486	4308	4078	3970	4212
Graph Traverser		4144	4400	4192	3976	3964	4135
Chambers with ts2		4144	4400	4192	3976	3924	4127

Table 1. Quality of the solutions obtained for five 50-point Travelling Salesman problems in arbitrary units of cost (i.e. total length of tour)

To control the possibility that a practice effect was involved in the improvements observed in the sequence 'unaided' → 'ts1' → 'ts2', a further four subjects were subjected to the same testing procedure, except that the three tests were presented in reverse order (i.e. 'ts2' → 'ts1' → 'unaided'). Observed improvements averaged less than half of one per cent, indicating that practice effects can be disregarded.

DISCUSSION

It is evident that for the Travelling Salesman problem the use of on-line graphics by the human solver can substantially assist him. The average total improvement obtained by the 4 'standard' subjects (excluding Chambers) was 6 per cent, and thus represents a non-trivial gain in economic terms. Approximately half of this gain was obtained with the ts1 version of the graphics program, which lacked the back-track feature, and hence is attributable solely to the ability to summon instant cost calculations in respect of given trial solutions. This facility can also be used for rapid evaluation of rival *partial* solutions (Chambers in particular availed himself very freely of this when using ts2). The other half of the total improvement can reasonably be attributed to the back-track feature of the ts2 graphics program, whereby the user can at any time command the best solution obtained to date to be

Plate 1. (opposite) A subject using the interactive graphics program on a Travelling Salesman problem



displayed. It is interesting that the further freedom to save a fancied solution, not the best, exercised no attraction for the subjects, who in general ignored it.

The back-track feature

Back-tracking proved its worth in a quite general and consistent fashion—both in the Shen Lin v. Graph Traverser comparison and in the comparison between the TS1 and TS2 programs as aids to the human solver. Could the human solver benefit, as does the off-line program, from having available a larger unit of back-tracking? This could be tested by offering the facility in the form 'restore to previous best prior to level $n-m$ ' where m could be set to 1, 2, 3, . . . , etc.

As for the use of the previous best *obtained in an earlier session*, the behaviour of Chambers stood out from the rest. He invariably refused the option of using as starting point his previous best solution—e.g. his best unaided solution as starting point for using the TS1 graphics program. Instead, he began afresh with the virgin pattern of 50 points, stating that he did not wish to be committed or biased to investigate the neighbourhood of his previous solution, in case it might require radical re-organisation to obtain further improvement. Nonetheless the various methods and approaches tended to converge on the same, or very similar, solutions. This can be seen in figure 3, in which the detailed progress of one of the five problems, problem c, is followed. The Graph Traverser solution is identical with that obtained in computer-aided mode by the best of the four subjects, and virtually identical with the other solution shown. It is also identical with that obtained by Chambers in computer-aided mode.

Why use computer graphics?

The question next arises: do these results indicate a use for computer graphics for problems of this kind? A tabulation of some relevant indices is given in Table 2. On the face of it, the man-machine combination wins. But since the Graph Traverser has not been coded for run-time efficiency, judgment should be suspended.

method	indices of utility, per problem		
	time	cost (shillings)	improvement
unaided (mean of 4 subjects)	15 mins	2	6 per cent
interactive graphics	30 mins	100	2 per cent
heuristic program	120 mins	800	

Table 2. A rough calculation of the cost and effectiveness of the three methods. The gains obtained by replacing unaided by interactive methods are purchased more cheaply than the further improvement which can be got by use of an off-line heuristic program

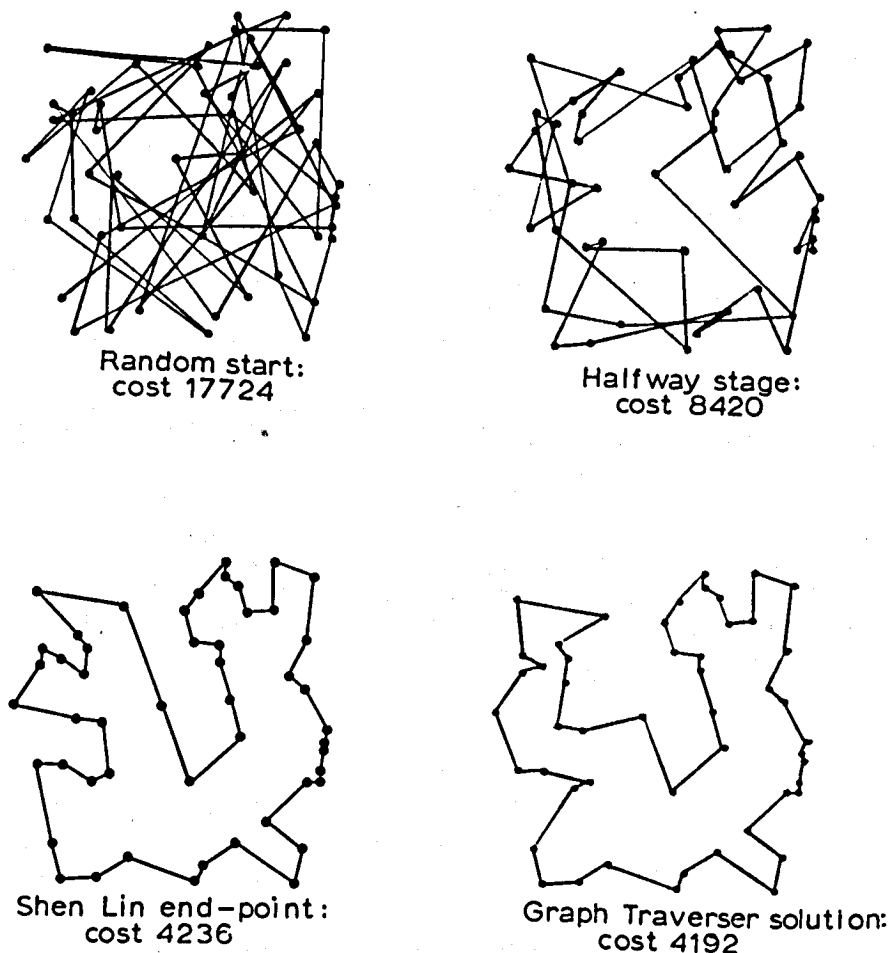
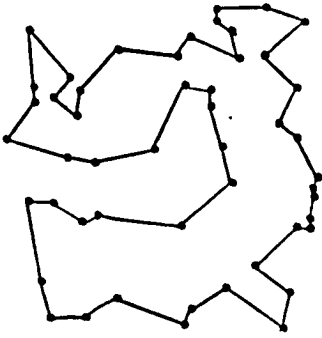
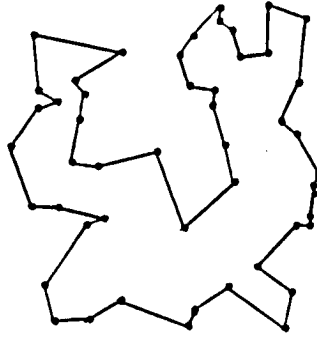


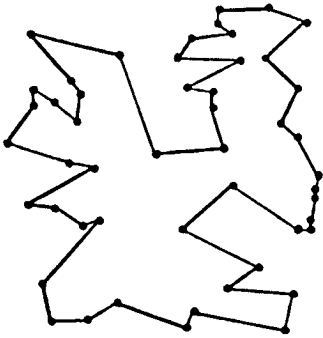
Figure 3. The first four diagrams show successive trial solutions in the application of this program to a Travelling Salesman problem. The first is a randomly constructed tour (the problem is to link n cities by as short a route as possible), the second is the state of affairs halfway through the program's problem-solving activity. The third corresponds to the Shen Lin termination point and the fourth is the best solution obtained. The four diagrams (*facing*) show the efforts of human solvers, on the left-hand side unaided, and on the right-hand side with the assistance of the interactive graphical program illustrated in the photograph



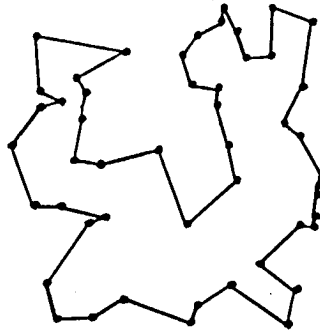
Human 1 unaided:
cost 4304



Human 1 + interactive graphics:
cost 4192



Human 2 unaided:
cost 4664



Human 2 + interactive graphics:
cost 4196

This discussion overlooks the fact that intensive study has been invested in the Travelling Salesman problem, including Shen Lin's proposal of the powerful 3-cut set of operators. As soon as we move to a problem in which invisible capital of this kind has not been invested, matters take on a different look. Consider a network problem, related to the Travelling Salesman: n points are distributed on a flat surface of which one is known as the origin. It is required to find a complete tour of all points, consisting of a number of excursions departing from and returning to the origin such that no excursion connects more than m points. Figure 4 shows a solution of the Coin Collector problem, as we call it, constructed with $n=32$, $m=7$.

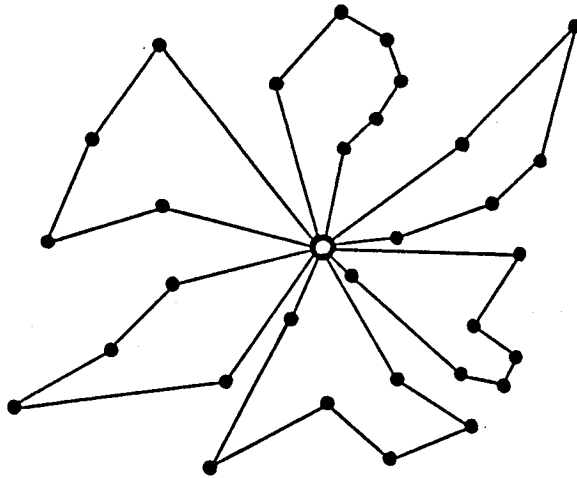


Figure 4. A specimen solution of a 'Coin Collector' problem with $n=32$, $m=7$ (see text)

Clearly it would be possible to study the Coin Collector with a view to finding an operator set for the Graph Traverser. If the problem were of great economic importance and likely to persist for a long time, this could be worth doing. But the quick and easy way would be to equip a gifted human solver with an interactive graphics program such as we have described.

Concluding remarks

When we embarked on this study, we were confident that 50-point problems would extend the powers of man and machine. It turned out otherwise, since both the off-line and the on-line approaches proved capable of getting within a fraction of one per cent of the presumed optimum. Indeed, even without any aids at all, Chambers proved capable of getting within $1\frac{1}{2}$ per cent of best. This circumstance allows us to pose rather clearly the conditions under which interactive graphics have benefits to offer.

The man-machine combination should, we suggest, be regarded as a means of forcing an entry into territory which would be otherwise impenetrable, but which it is intended ultimately to subjugate to the arts of full mechanization. We thus see interactive techniques not as an end in themselves but as a step towards having the problem solved entirely by the machine behind the scenes. We see a progression of stages: algebraic simplification, for example, was initially a job for man alone, because in the state of the art prevailing a few years ago it was too difficult to mechanize. At the present moment this family of problems is transitional, with scope for collaborative work using graphic methods. Soon the more purely tactical aspects of symbol manipulation will be so well mechanized that they will cease entirely to obtrude on the user, just as he is untroubled today by the niceties of floating-point arithmetic which monopolized so much attention in earlier years. The wave-front of interactive graphics will then roll on to problems at a more strategic level. What is strategic today is tactical tomorrow. The present study has caught a particular problem—50-point Travelling Salesman—in transition. Further effort devoted to the problem should go to improve the off-line program. Further effort devoted to the *on-line* program should be directed towards more complex and highly structured problems.

Acknowledgments

We owe warm thanks to Miss Susan Swann, Miss Dorothy Caldecutt, and Mrs Joy Kennedy for invaluable help with the testing of subjects and also to the subjects themselves for their co-operation. We also thank Dr J. S. Collins for help with certain aspects of the estimation of costs, and Dr R. M. Burstall for suggesting extending the length of the Graph Traverser's back-track step for this application.

This work was supported by grants from the Science Research Council and the Medical Research Council.

REFERENCES

- Collins, J. S. (1964), *The Processing of lists and the recognition of patterns with application to some electrical engineering systems*. Ph.D. Dissertation. University of London.
- Doran, J. E. & Michie, D. (1966), Experiments with the Graph Traverser program. *Proc. R. Soc. (A)*, 294, 235–59.
- Doran, J. E. (1968), New developments of the Graph Traverser. *Machine Intelligence 2*, pp. 119–35 (eds Dale, E. and Michie, D.). Edinburgh: Oliver & Boyd.
- Lin, S. (1965), Computer solutions of the Travelling Salesman problem. *Bell System Tech. J.*, 44, 2245–69.
- Ross, R. (1967). Personal communication.
- Schofield, P. D. A. (1967), Complete solution of the Eight-puzzle. *Machine Intelligence 1*, pp. 125–33 (eds Collins, N. L. and Michie, D.). Edinburgh: Oliver & Boyd.

Interactive Programming at Carnegie Tech

A. H. Bond

Department of Computer Science
Carnegie Institute of Technology
Pittsburgh, Pa.

1. PROLOGUE—THE RELATIONSHIP OF CONVENTIONAL COMPUTING SCIENCE RESEARCH TO ARTIFICIAL INTELLIGENCE

Ernst and Newell (1967) have discussed programming languages as problem solvers. An ALGOL program 'solves' a certain restricted class of problems. The program is constructed using the primitive control structures and primitive data structures of ALGOL. An artificial intelligence program also solves a class of problems in an analogous way. It has a heuristic control structure rather than one which is straightforwardly deterministic, but Newell and Ernst thought this an unimportant distinction. One might consider an artificial intelligence program as a choice of data structure and control structure which could have been a choice of language. They did not consider the design of a language for artificial intelligence, but it seems to me that such programs are so diverse that an extendable language with suitable primitive structures would be appropriate.

Second, an interesting paper by Simon (*see* Simon, 1967) compares the central nervous system to a time-sharing monitor program. He argues that processing in the human brain must be serial, and that given the multiple needs of the organism in an unpredictable environment, one is led to a monitor-like control structure with hierarchies of goals which are 'motivations', and an interrupt mechanism which leads him to a theory of emotion. Certainly one would like a control structure which can serve a multiplicity of motives simultaneously, as opposed to the dogged, weakly contextual control of most problem-solving programs.

2. INTRODUCTION

In this paper I try to describe the current state of interactive programming in so far as I have been concerned with it at Carnegie-Mellon. I describe the

graphics system on the G-21, the main computer at present. The graphic monitor and special procedures in ALGOL and Formula ALGOL provide a software system for writing programs with human-program interaction. Some of the difficulties of system programming and of the design of languages which have well-defined semantics during interaction and which provide the needed facilities are discussed. Examples of programs which have been written to interact graphically are described.

3. THE G-21 GRAPHICS SYSTEM

3.1. Hardware

As shown in figure 1, the G-21 is a 64K computer with two central processors, which runs a card and teletype queue and a teletype servicing monitor. There is a general filing system called AND. There are three very advanced scopes (see Quatse, 1966), also serviced by a special auxiliary 'scope monitor'. Rather than using a special satellite computer, we have an auxiliary 8K module of core which is switchable, i.e. it can be switched, by a G-21 machine order, to be addressable by the G-21 central processor, at which time it replaces a normal module. The entire normal core consists of eight modules. Programs can run using 32K or 64K and they can interact with the switchable module. The module contains code which receives interrupts and performs actions under control of the human scope user; it also contains display material which is a sequence of *scope opcodes* which is acted upon by the *scope scanner*. The scope scanner is a separate processor which displays the display material on the screens and enables display material to be entered by human action at the scopes. Thus, the scope module is used by two separate processors; one which interprets the bit patterns as G-21 machine orders, and the other which interprets them as display commands. I suppose any autonomous I/O unit is similar in this respect. It conjures up the idea of a large computer with several processors, some doing computing—setting up I/O material—and others performing the I/O operations. One can enter characters (a large set is provided) or lines, but no curves (these have to be fabricated from lines), and the scanner will also perform certain editing operations on the display material, like clearing a display region or deleting or inserting an element. The display material is broken into a chain of linked modules which the scanner constantly scans. Each module is headed by a delimit scope opcode saying on which scope(s) the display is to be shown. Each scope can have four separate display modules, called its pages; they are of arbitrary length and can be displayed or superimposed visually at will at the console.

3.2. The scope monitor

The scope monitor (see Bond, 1967) manages the core available space and performs managerial operations on request by the human user or a G-21 program in 'lower core'. It time-shares the three scopes and has a simple, variable increment, interrupt stack for each scope. The various facilities

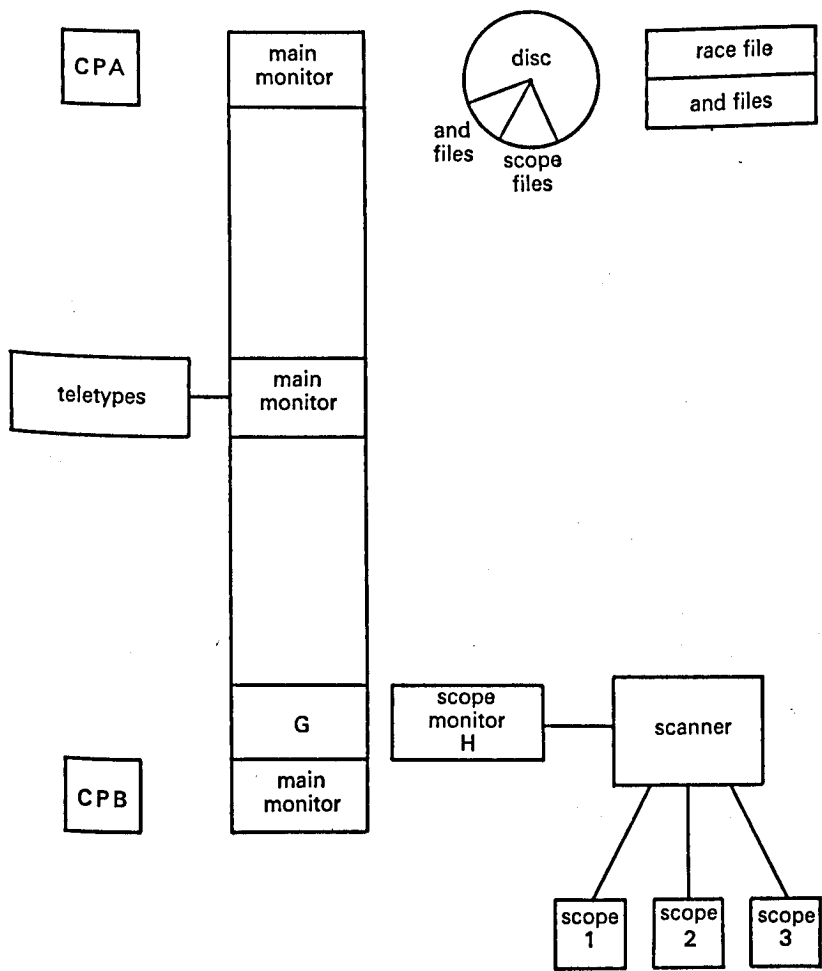


Figure 1. The α -21 and H-module

SOME OF THE OPERATIONS PROVIDED BY THE SCOPE MONITOR

Option state

- Interrupt No. 2. select management state
3. select program state
 4. select debug state
 5. select text handling state
 6. select user manual state
 7. select drawing state
 8. select user program interaction state
 9. layout
 10. select text editing state

Management state

2. Save page N as scope file M .
3. Read in scope file M to page N .
4. Append page N_1 to page N .
5. Display directory of scope files.
6. Get N blocks for page M .
7. Enable page M for human input.
8. Disenable page M .
9. Delete page M .
10. Create space on a scope file.

Program state

2. Convert page N and move to $G-21$ input buffer.
 4. Submit input buffer to $G-21$ queue.
 5. Display input file.
 6. Display input file.
 7. Forward ten lines through file.
 8. Back ten lines.
 13. Allow program submitted from scope N to interact.
- Interrupt 1 always switches on or off the main display.

Figure 2

provided by the scope monitor are demanded by twenty interrupt buttons, the meanings of the buttons being shown in an explanatory display (like a menu). The operations are arranged in groups called *states*, and one changes state by interrupt also in a hierarchical fashion. We think that the commands available form quite a comprehensive set (some are listed in figure 2), and we are trying to make it easy for ordinary users to use the scopes using ordinary ALGOL and Formula ALGOL programs.

3.3. Writing interactive programs

The way one writes an interactive program is simply by using commands very similar to the interrupt commands. Each of these corresponds to a monitor user routine, which we call *B routines*. A list of *B routines* is given in figure 3.

B ROUTINES

- B(-1). Announce user program to scopes.
- B(0) and B(1). Convert characters to and from display format.
- B(2) and B(3). Move characters and vectors to display region.
- B(4) and B(5). Move display region into program array.
- B(6) and B(7). Read and set the cursor.
- B(8), B(10), B(11). Read knobs, read switches and set switches.
- B(12), B(13), B(21), B(22). Set compare character, define compare routine, remove compare character, reset compare routine.
- B(14), B(23). Set memory-full routine and reset.
- B(15), B(16), B(17), B(18), B(19), B(20), B(28). Get display space for a page, enable the page, disable the page, delete the space for the page, disable all pages, clear a page.
- B(24). Set compare character, compare routine and set cursor to receive set.
- B(25). Define button interrupts.
- B(26). Display a page on another scope, if permitted.
- B(29), B(30), B(31), B(32). Move page or array to a scope file and vice versa.
- B(33), B(34). Read in a 3-digit integer and read in a string of characters.
- B(42). Declare AND files as subsystem files.

Figure 3

By a succession of B routine calls in a program one can set up display or read in display from the scope face. In ALGOL, a procedure called B is provided, whose first argument BNUM is the number of the B routine being called. The procedure B performs all the necessary module switching, checking, memory protection, etc. The means available for human-program interaction are as follows:

- (i) each displays text to be read by the other;
- (ii) each displays a general display of lines and text;
- (iii) various hardware attachments:
 - (a) two 'analog knobs' with values in [0,63] can be set by humans and read by program;
 - (b) the cursor can be set and read by both;
 - (c) eight switches can be set and read by both;
- (iv) the use of interrupts
 - (a) compare interrupt on a certain (set of) character(s) on a certain page;
 - (b) memory full interrupt;
 - (c) button interrupts, numbered 1-19.

Figure 4 also lists a library of ALGOL routines written using the procedure B, giving a fairly comprehensive set. Note that input and output of card images in the CMU implementation can use an ALGOL array as buffer, and so the scope i/o has the full formatting and conversion facilities of the ALGOL-20 i/o language.

SCOPE LIBRARY

1. INTEGER PROCEDURE LOC(N);
2. B(BNUM, R52, R53, R54, R55, R56);
3. HEADER(X,Y);
4. VECTOR(X,Y,SG);
5. CHA2STR(C1, C2, C3, SG1, SG2, SG3);
6. NUM(X,Y,N);
7. LINE(X1,Y1,X2,Y2);
8. CURVE(X,Y,T,DT,TA,TB);
9. SCALEX(X); SCALEY(Y);
10. READPAGE(N,RBUFF);
11. PRINTONPAGE(N,WBUFF,X,Y);
12. BUTTIN(ENT,CONSW,INTNO,SCOPENO,COMPCHAR);
13. COMEIN(X,Y,CHAR,PAGENO,ENT);

Figure 4

3.4. Permission to interact and interaction with more than one scope

Before any B procedure can be called, B(-1) must be called. This announces the user program to the scope monitor which allows interaction provided the program was submitted from the scope, and not from some other medium, and that the job card corresponds to the user logged-in on the scope. The program also checks, just before, that the scope monitor is indeed present, by checking a 'clobber word' in the scope monitor core area. If a program wishes to interact with more than one scope, it sets a variable (a register) when calling the B routine. This will lead to an error unless the human at that scope has first instructed the scope monitor, by means of an interrupt, that it will allow the program submitted from scope N to interact. The program can display and read from scopes for which permission has been given. When an interrupt is generated by the human, the scope monitor passes the number of the interrupting scope to the program.

4. USER INTERRUPTS

The program can define certain points to be entered on interrupt. The scope monitor, of course, processes all interrupts, and recognizes those destined to be processed by the user program and passes control at an appropriate point, together with useful information. The compare interrupt is set by putting a special scanner opcode in the module of core corresponding to a given page. Any subsequent attempt to type in a character, of the type designated only, on that page will cause the scanner to generate an interrupt and place the recognized character in a fixed location to be read by the scope monitor. User interrupts of this type can occur in any state; however, the button interrupts are defined by the 'menus' until the monitor is in the 'user program interaction state'. When these interrupts are first defined and the entry points declared to the scope monitor, the user program also passes certain integer arguments which are locations of variables in the user program, i.e. the user program

essentially declares some variables for communication with the scope monitor. On interrupt, the scope monitor places all the information, like interrupt number, scope number, etc., in the agreed locations and passes control to the user. To prevent multiple interrupt problems, it was found unsatisfactory to use the hardware control register of the G-21 which prevents any interrupts occurring, as this was too powerful. Instead, one of the declared variables is a user interrupt control switch and is consulted by the scope monitor before passing control. If the switch is set, the scope monitor 'remembers' the interrupt (it does not queue multiple interrupts, however) and keeps inspecting the switch. Thus, one has a shared ALGOL variable, which if used foolishly could cause havoc. For example, the interruption of the evaluation of an expression involving the scope number might have one value for it in one part and another in another part of the evaluation; however, this mechanism, if used sensibly, can be advantageous.

The interruption of programs written in high-level languages has its problems. In machine or assembly code one can simply pass control anywhere; the only errors are those to do with the interruption of a printing command, in which further printing is attempted. This leads to a machine halt from the printer on our machine. If data is overwritten by the new control, in particular return 'marks', then one assumes the programmer knows what he is doing; he must save any information as it enters the interrupt entry point. In ALGOL, first of all, note that we always use integer variables which are locations of variable, i.e. the address of where the value of the variable is stored. This needs the library procedure LOC. I believe something like this has been proposed in ALGOL X, and is defined in CPL. Normally, the user's interrupt entry point is to his own interrupt service routine; we shall call this the UISR. It makes decisions about what to do next. If it wishes to return to the interrupted computation, then one has to be careful. If it is a simple procedure or simply nested set of procedures without *gotos* then one can simply return through the UISR return mark to the monitor and thence back to the interrupted computation; this is arranged by the scope monitor. If one has wandered further afield using *gotos* then a B routine is provided to allow it to return to the interrupted computation. The interrupted computation can then continue provided no damage has been done. In genuine recursive ALGOL, the return marks of all routines will be all right; in non-recursive implementations, not so. A call on a monitor routine (the normal monitor, not the scope monitor) which is overwritten by a similar call in the UISR will lead to trouble. Printing in the UISR will upset NAME lists in the ALGOL printing routines, and indeed it can be said that if a return to the interrupted computation is envisaged, the UISR must not print or communicate with the monitor. In most programs the UISR does not return, however, and usually initiates some entirely new task.

Something more powerful than ALGOL, like Formula ALGOL (FORML for short), is another matter entirely. The implementation at Carnegie-Mellon

has two features which interest us here. First, it compiles code which is heavily dependent on (non-recursive) run-time routines. Second, all marks and values and printing pointers are stacked in a communal 'historian' stack. The `UISR` has to be restricted to a very small subset of `FORML`; in fact, simple arithmetic expressions and assignments, and local backward `gotos` only, to avoid disturbing the historian.

The typical program is usually happy to wait in a passive loop looking at a switch and then to act accordingly. One only really wants to interrupt if one sees one's program on a long, worthless task and wishes to terminate this action prematurely. Usually one issues a command and waits for completion before issuing the next command. It is interesting to observe that interactive programs must take on more of a monitor structure than normal programs, also that flow-charting of such programs can sometimes be tricky with control lines appearing out of nowhere, and presumably also going to nowhere, viz. the monitor. Whether sophisticated interactive programs will have to worry about queueing of interrupts or priorities of interrupts is not clear. An advantage of interrupts is that they do tend to give a quick response to the human.

Before going on to discuss interactive programs, I shall first describe briefly the other type of interactive user system, viz. the user submonitor and also the modular version of the monitor.

5. THE MODULAR VERSION OF THE SCOPE MONITOR AND USER SUBMONITORS

In the new version, close to completion, the scope monitor has been restructured as a small resident part $\sim 1000_{10}$ words and a set of relocatable modules of code which swap as required from the disc.

Any user can write a 'submonitor' consisting of an indefinite number of linked modules. These are declared, i.e. assigned logical numbers, by the human. They may be on scope files or `AND` files, on the disc, and they are used in the same way as system modules. In this way it is now possible to extend the scope monitor indefinitely in modular form, assembling and debugging individual modules only. The assembly of suitable modular code is achieved by a small set of macros, and any assembly program can be so converted. Unfortunately we do not have base registers, so we use a relocation algorithm to relocate the modules at run-time. No high-level language is available for writing this kind of system, but the macro assembler, `SPITE`, is quite nice. We now have the situation of one large available space, and modules of system code, user code, data, or display material all being allocated space. A text editor submonitor has been written by M. Coleman, which provides extra text manipulation facilities, actuated by graphical commands.

6. CONTINUOUS OPERATIONS AND CONTINUOUS CONSTRAINTS

The scope monitor provides various continuous operations for the human

user on request. It will continuously sample a selected area of core and display an octal dump. Using the analog knobs, one can select any area in the G-21 for display. One can request a rotation mode in which a rotation of an analog knob rotates the vectors on a selected page. One can request a curve drawing mode in which the scope monitor puts in line elements as the cursor is moved.

No continuous operations are provided for a user program at the moment, but we feel that some continuous monitor capability could be provided. The user could specify some condition and the monitor could continually check it and interrupt when necessary. This is similar to the PL/1 operation ON, but is provided by the monitor rather than being part of the user program. Hardware interrupts can be used to monitor access of, or storage into, flagged locations.

7. INTERACTION WITH TYPEWRITERS

Although some provision is available in the main monitor for interaction with a teletype, no one has yet used this facility in a user program. The routines are used by a desk calculator program, similar to JOSS, which is a part of the monitor, and available on line to any teletype user. The assembler SPITE does have provision for error correction by human intervention at assembly time using a teletype; however, this involves reinitiation of the assembly by job-card.

8. INTERACTIVE PROGRAMS

1. *Brooker Morris Compiler-Compiler*. This has been implemented on the G-21 by F.R.A. Hopgood (1967). We simply added graphical input-output. Input cards are typed on one page and cleared as they are processed. The last few lines of line printer output are displayed on another page. The response time is imperceptible.
2. *The GRASP System*. This is a graphic service program written by E.M. Thomas (1966, 1967) in ALGOL for the G-21. It is a model building language whose format is like a series of procedure calls. It can also be used as an outer block to an ALGOL program, in which case the procedure calls can be freely incorporated into ALGOL statements. It differs from Sketchpad in several ways, and it does not have constraint satisfaction built in.
3. *GRANIS*. A graphical inference program written in Formula ALGOL by L.S. Coles (1967). It reads in a picture from the scope face and a statement and decides whether the statement is true or false about the picture.
4. *A Pattern Recognition Heuristic Program*, by T.W. Calvert (1967). It allowed some human intervention, but mainly for the purpose of gaining insight into its behaviour rather than for a man-machine production system.
5. A graph drawing, interactive iterative and curve fitting system in ALGOL is under development by the present author. Its purpose is to experiment with a very intense interaction between a human and a set of functions.

A typical layout of an interactive program is given in figure 5.

MAN-MACHINE INTERACTION

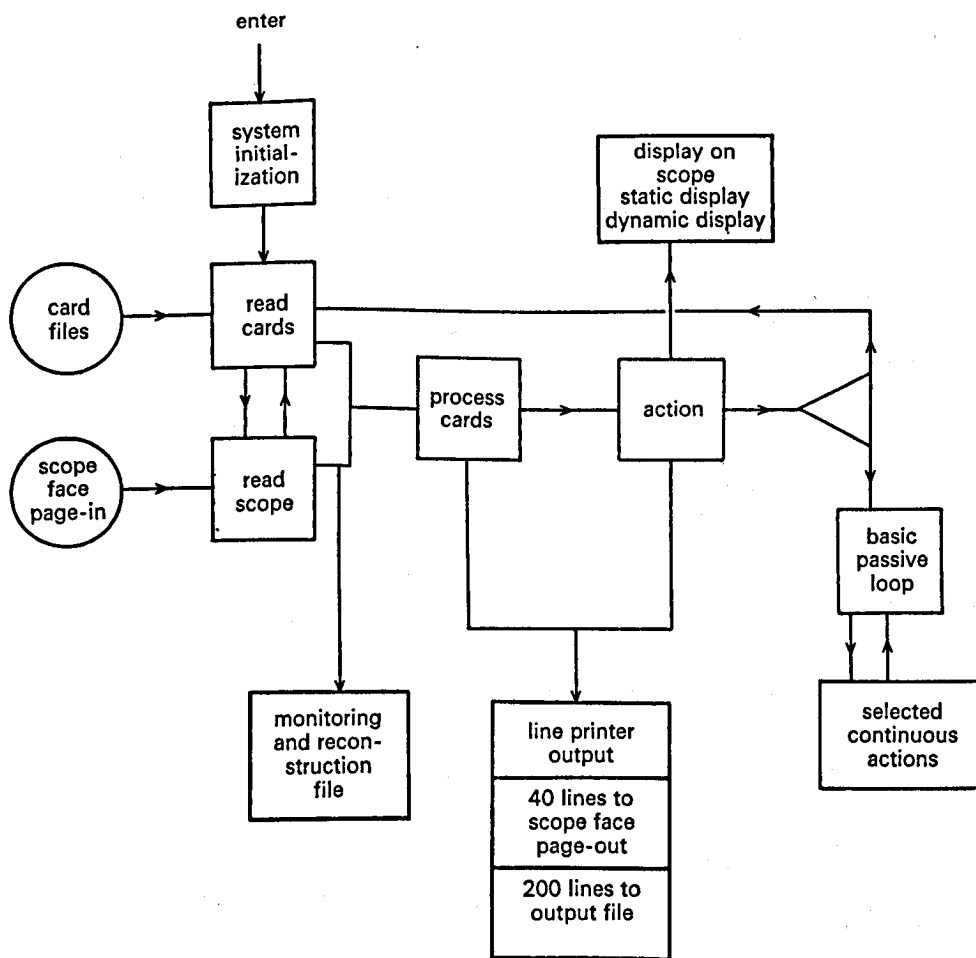


Figure 5. Control structure of interactive system

9. THE FUTURE

In the coming year our two new computers, a Univac 1108 and an IBM 360/67, will take over the main computing load. Some conversational compilers are provided, and we will write some systems ourselves. The LCC project by Perlis, Van Zoren, and Mitchell is a modified conversational ALGOL with a delightful dynamical structure. It is being implemented on the 360. There will be graphics on both machines, but it is not yet clear what hardware will be suitable or what software will be available. The G-21 will gradually become used mainly for graphics. There are several application programs being written in ALGOL for oil mining analysis, structural stress analysis, electrocardiogram recognition, architectural design, etc. Also, there are a few artists beginning to try to use the computer as an artistic medium or artistic aid. Eventually the main monitor will be altered to allow the three scope programs, one from each scope, to monopolize the machine with a swapping system. In the coming year we hope to get some experience with ordinary users writing interactive graphical programs in the usual high level languages.

REFERENCES

- Bond, A.H. (1967), Scope user manual. *CIT document*, CIT.
 Calvert, T.W. (1967), Ph.D. thesis. Department of Electrical Engineering, CIT.
 Coles, L.S. (1967), Ph.D. thesis. Department of Computer Science, CIT.
 Ernst, G.W. & Newell, A. (1967), Generality and GPS. *CIT document*, CIT.
 Hopgood, F.R.A. (1967), The machine independent implementation of the Brooker-Morris compiler compiler.
 Quatse, J.T. (1966), A visual display system suitable for time shared use. *CIT publication*, CIT.
 Simon, H.A. (1967), Motivational and emotional controls of cognition. *Psychol. Rev.* 74, 29.
 Thomas, E.M. (1966), GRASP user manual. *CIT document*, CIT.
 Thomas, E.M. (1967), GRASP—a graphic service program. *Proceedings 22nd A.C.M. National Conference*, p. 395. London: Academic Press.

Maintenance of Large Computer Systems – The Engineer's Assistant

M. H. J. Baylis
Atlas Computer Laboratory
Chilton, Berkshire¹

SUMMARY

This paper describes some of the difficulties in maintaining large computer systems and considers how machine perception techniques can be applied to the problem. A program called the 'engineer's assistant' is described as a step in the right direction.

Elaborations are made on the meaning of 'the right direction', and an experimental implementation of some of these ideas is described.

INTRODUCTION

A large computer system is usually maintained by a team of engineers. Inevitably these tend to be of different abilities, and some of them become more knowledgeable and therefore more proficient than others in specific areas of the machine. Although computer 'down time' has become more expensive by an order of magnitude on present large systems compared to the last generation, the engineers are not mending faults an order of magnitude faster. Moreover some faults which can occur nowadays are considerably more complicated than last generation faults. Therefore the best engineers are usually called upon in times of trouble, and these increase their experience at the expense of the other engineers. When these expert engineers move on to other jobs they take with them an invaluable sum of accumulated knowledge about the machine.

Further, when a unit gives trouble after a lengthy period of successful working, everyone present has forgotten how to proceed in tracing the fault. Symptoms often show which cause engineers to say 'we had this trouble two years ago, what did we do then?'

As a better than typical example, consider the present maintenance system on the I.C.T. Atlas computer at S.R.C., Chilton, remembering that Atlas was

¹ Now with I.C.T. Ltd.

designed around 1959. The engineers have a number of programming aids for preventive maintenance and fault diagnosis.

These include

- (i) one or more stand-alone test programs for each part of the machine;
- (ii) an engineering operating system which will multi-process many of these test programs;
- (iii) a test-programming system which can run tests as object programs during normal computing service;
- (iv) the operating system itself.

During scheduled maintenance time, (i) and (ii) are used for preventive work on those parts which have not been inspected during computer service time with (iii).

The operating system includes a number of tests. Instead of 'idling' in those periods when it is impossible to carry on with object program execution, random tests are made on the arithmetic units, and every five minutes a few tests are made on special equipment such as the instruction counter and the digital clock.

For all incidents of machine failures, the operating system outputs what information it can about the state of the machine and the malfunction. This maintenance system works well for finding normal faults but is of very limited use in finding obscure faults.

In the last few years significant improvements have been made by computer manufacturers in maintenance methods. As the cost of computer components has dropped, it has become feasible to provide much more checking logic in the machine. This extra logic allows for quicker detection of malfunctions and frequently permits the current state of the machine to be preserved for inspection. However, it seems to me that these improvements are not of much help in finding obscure intermittent faults.

FAULTS AND THEIR FINDING

We might group faults into two broad classes: those which provide symptoms regularly (solid faults), and those which provide symptoms very intermittently.

The first class is, of course, the easier to investigate. Such aids as test programs, oscilloscopes, etc., are well understood. The second class provides much more interest. How does one find a fault which shows itself once in an hour or once in a week? The answer, one likes to think, is by logical deduction from the evidence, backed up by intuitive guesses as to what could go wrong, based upon knowledge of previous machine behaviour. At present rather haphazard techniques are followed, such as the following:

- (i) the incident might be ignored, with the hope that it will either go away or develop into a more solid fault;
- (ii) engineers' tests or tests made from user programs are run for lengthy periods while various engineering attitudes are struck (e.g. varying the voltage margins);

- (iii) guidance is sought from possibly helpful system program writers as to what was going on in the machine at the time of trouble.

When a large machine is first commissioned, intermittent faults tend to be hidden in the noise level, compared to solid faults. After a year or so the ratio is more than 80:20 in favour of intermittent faults. One reason for this is that multiple machine failures occur for each intermittent fault before it can be eliminated.

The general solution to this problem of minimizing down time must be to automate the processes of deduction as far as possible. The solution presented here consists of some primitive hardware and a sophisticated computer program to supplement the standard maintenance methods. The name 'engineer's assistant' is given to this system. We consider the program first.

OBJECTS OF THE ENGINEER'S ASSISTANT PROGRAM

The major program objectives are, in order of complexity:

1. to accumulate a memory of the symptoms and remedies for all previous faults;
2. to make this information available as needed:
 - (a) to aid engineers in fault-finding;
 - (b) to produce statistical information about the machine;
3. to provide question and answer guides to aid fault-finding;
4. to investigate and diagnose faults with little or no human assistance.

Taking these objectives in order, it will be realized that the major problem in implementation of 1 and 2 is that of data structures. The program is required to hold information about many interrelated objects; to access, add, delete, modify, and do calculations on objects and their properties, including such properties as interrelationships with other objects.

The work done in implementing a primitive engineer's assistant program on Atlas has led to a belated recognition that the correct data structure is a 'plex'. Plexes, like hash tables, are something that everyone invents for himself, but the formal definition and exploitation is due to Douglas Ross of M.I.T. (see Ross, 1961, and Ross and Rodriguez, 1963).

The name is taken from plexus—a network, or interwoven combination of parts in a structure. A plex is an interconnected set of n -component elements. The components of an element are its properties usually contained in successive addresses within the element. Typical entries would be the type and name of the element, and information which might range from binary bits, instructions, and symbolic data to different sorts of link connections with other elements. A moment's thought shows that most, if not all, data structures can be regarded as examples of simplified plexes. The tree structure of LISP (see LISP 1.5 programmer's manual, 1966) consists of elements with only two components; the 'ring-structure' of Sutherland's Sketchpad project is another example (see Sutherland, 1963).

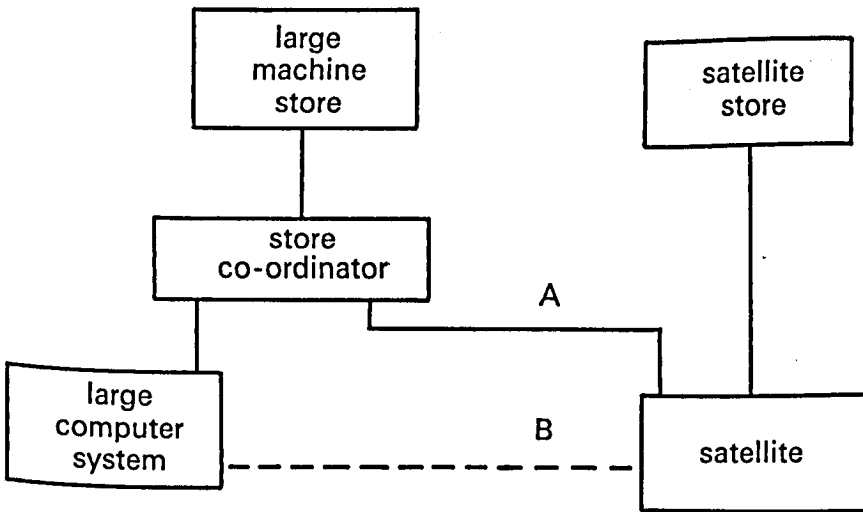
For our particular application there needs to be a considerable number of

types of element, e.g. installation, hardware units, sub-units, packages, faults, etc., where all elements of any one type have a consistent layout of information. The interlinking of elements will not only start off fairly complicatedly but will get more and more involved as the memory matures. For example, an element of type 'package' with name '890 board' might be reached through the route 'installation-(S.R.C. Chilton), area-(core store system), unit-(stack 5), fault-(digit 16 dropout)' etc. Or by the route 'package type- (read amplifiers),' or, again, by 'fault-(any), position-(MO3/10.4/16), date-(1/1/67 to 8/1/67) . . .', and so on.

So far, about thirty element types can be seen as necessary. Having decided on a data structure, the next decision is to choose a language in which the problem can be expressed. If most of the program can be generalized enough to be machine independent, then it will be that much more valuable. At M.I.T., extensions have been made to ALGOL by Ross *et al.* for handling plexes. There is some evidence that ALGOL X will be suitable. Another contender is CPL.

Providing a question and answer guide for fault-finding is relatively easy, assuming that the best engineers can explain how they mend faults. There is one essential requirement: a facility for the user to modify the questions and answers and provide new alternatives. In fact, for such a program to be really machine-independent it must start off asking for questions and enquiring what answers can be given. In order that the program can use its memory for finding any similar faults, the format for such a dialogue must be in the same form as the definitions given to elements in the plex.

We now come to consider the automatic investigation of faults. It should be emphasized again that, although the system will work for solid faults, it is the difficult intermittent faults that we are interested in finding. These may be of two types: either they cause a machine stoppage or they do not. In either case the fault may be in the central machine, that is, in the logic which controls the processing of instructions, and in practice it is these faults which are hardest to track down. We can envisage an engineer's assistant program (EAP) running in a partially broken machine, but because that may not be always possible it seems desirable that there should be a separate satellite computer for the program. The minimum connections needed are as shown in figure 1. With such an arrangement the EAP could follow its own question and answer route independently, that is, it could phrase questions and select some machine instructions which could answer them. This code would be positioned in the large machine store and the large computer instructed to obey it. Note that a body of code can be an element on a plex structure. There is little doubt that such a system would work. It would, however, have only limited success because, with many intermittent faults, the symptoms are transient and disappear before they can be recorded by any program. Also, symptoms manifested by programs are at present far too general; in the absence of more detailed evidence there can be too many possibilities to investigate. There is,



A: the satellite can access some or preferably all of the large machine store

B: a control link at interrupt or high level

Figure 1. Basic connections for maintenance satellite

therefore, a requirement for much more intimate evidence to be accessible by program, and without such evidence progress will be slow.

HARDWARE CONSIDERATIONS

Most computers have an engineer's console on which rows of lamps flicker. These lamps show the states of important control points in the central computer, and if the machine stops then the lamps give some idea of what was going on at the time.

Some observations are worth making:

1. There are not nearly enough lamps on most if not all computers.
It should be easily possible to monitor all decision points in the logic.
2. The setting of lamps should be incidental to the more important function of setting bits in registers which can be accessed by program.
3. It should be possible to 'freeze' the states of these lamps and registers when specified conditions are satisfied, and to allow the computer to continue without altering these states. By having access to such powerful evidence the EAP run from a suitable computer should be able to make rapid and effective diagnoses. The freezing of states can be done in a number of ways.

MAN-MACHINE INTERACTION

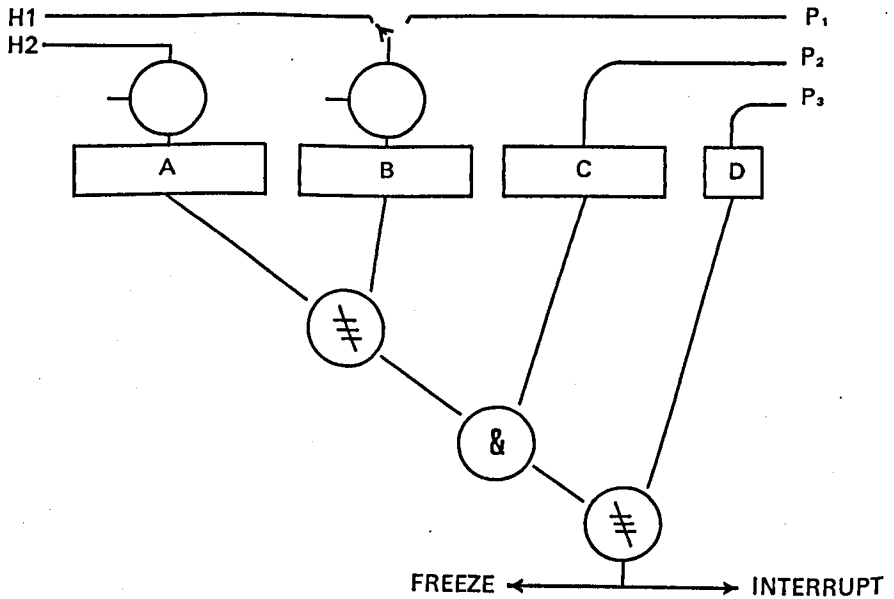


Figure 2. A possible simple check unit

Consider the check unit shown in figure 2. A, B, and C are buffer registers, probably equal in size to the machine word length. D is a one bit register. A and B can be set by hardware events H1, H2. B can alternatively be set by program P₁. C and D are set by program P₂, P₃. With this checker, A and B can be compared over a field C; D will then determine whether to freeze on equivalence or non-equivalence. If it is required to freeze when there is a difference between H1 and H2 in field C, then D will set to a zero, for example.

It is not feasible to position these checkers everywhere they might be wanted. Sets of floating plugs and sockets could be provided and the program or engineer could decide where they should be placed. A better method would be a switching matrix so that the EAP could connect the checkers it had to the parts of the machine in which it was currently interested. As a speculative matter, if checkers could be connected together under program control then very sophisticated conditions for freezing could be set up.

To our basic connections between a large computer system and a maintenance satellite we have now added special hardware to provide detailed information in the form of 'snapshots' for a perceptive program. We now leave these ideas and consider what has been implemented so far at the Science Research Council Atlas Computer Laboratory by S.R.C. and I.C.T. in collaboration.

THE ATLAS ENGINEER'S ASSISTANT

This program was mainly implemented by Andrew du Plessis (S.R.C., now Cambridge University) with the assistance of the chief maintenance engineer Peter Dean (I.C.T. Ltd.) and myself (*see* du Plessis, 1967). Its object was to explore the possibilities of a systematic data retrieval system applied to maintenance engineering. Most of the program consists of fault location procedures for the different parts of the machine. By running the test programs specified and thus answering the questions asked, some progress towards diagnosing faults will be made. When the program has reached the limit of the assistance it can give (it will then be pinpointing a faulty area, a particular group of packages, or an individual package), it can output records of all previous repaired faults which have followed the same diagnostic route. An underlying assumption here is that a component which has failed once is likely to fail again; in practice this is true enough on Atlas to make the program useful. It is possible to retrieve fault histories without going through the question/answer procedure, and to extract information about package types. The program is controlled on-line with a simple command language. After identifying itself, the program asks for a directive. Some of these directives are described below.

GUIDE 'UNIT'

This calls the fault location guide for the specified unit. Exit from the question/answer routine back to the command language is made by typing 'return'. The questions usually list alternative answers, and only the first few characters are necessary for valid replies. A typical example might be:

TYPE DIRECTIVE

guide main store

ARE ALL STACKS BROKEN? 'YES' OR 'NO'

no

IS FAILURE 'TOTAL' OR 'INTERMITTENT'?

total

IS TOTAL FAILURE EVIDENT IN 'LOSS OR PREPULSE', 'ADDRESS TEST', OR ANY 'OTHER' TEST?

address

TYPE AS TEST RESULT 'SINGLE' DIGIT FAULT, 'BOTH HALVES' OR 'ONE HALF' FAILING

both

RUN ADDRESS TEST ON FAILING STACK. LOOK DOWN 970 AND 971 MONITORS TO SEE MOVEMENT. TYPE 970,971, 'STOPPED' OR 'MONITORS ALL MOVING'

monitors

CHECK ALL 972 CURRENT WAVEFORMS, THEN TYPE FOR PULSE 'O/K', 'HALF-SIZE' OR 'NONE'

half

MAN-MACHINE INTERACTION

PULL OUT VOLTAGE 964's ONE BY ONE UNTIL PULSES GO FULL-SIZE. REPLACE FAILING 964

IS A CASE HISTORY OF SIMILAR FAULTS REQUIRED?

yes

TYPE STACK NUMBER

11

THE CASE HISTORY WILL BE OUTPUT ON THE LINEPRINTER. IF STILL U/S CHECK VOLTAGE LEVELS. TYPE DIRECTIVE finish

BOARD (*b* * *Ua* * *p/q*)

The format for this directive is shown above where:

b is the particular package type number

Ua is a unit or sub-unit of the computer

p/q is a particular set of package positions in the machine.

The last part or last two parts may be omitted. Thus it is possible to obtain information about all packages of type *b*, to restrict this to a unit or to positions within a unit, e.g. the directive BOARD (822) would produce output of records of the form:

Installation Chilton

Serial number 131

Incident on 29/03/67

822 board, number 146309 in sub-unit R, failed at 169/4*16.12B6 causing fault type 03/B02.

PAR test 21 showed PAR 58 U/s. Board changed.

FAULT

The information retrieved by FAULT is a listing of faults which have occurred in a particular unit. The type of fault may also be specified. The format is similar to that for BOARD, and specifies the unit of the machine, an optional fault type and, optionally, a point in the fault location guide for that unit.

Other directives include DATE for setting start and end dates between which the program will operate, and PLACE. The latter commands information retrieval to be only from the specified installation. There are of course editing and updating programs for altering the memory. Engineers are forbidden to mend faults without contributing to the memory.

The system has been in use for some time. Its main uses have been in helping to train junior engineers and in presenting statistical information in a useful way. For example, it showed that a certain package position in the central machine had been changed significantly often. Once this had been observed, it was quickly realized that, although changing this package appeared to cure a fault, the trouble was in fact caused by an earlier logic stage.

Its limitations are that the data structure is not very sophisticated, the

diagnosis tree cannot be altered easily, and the program is completely machine dependent. It has been observed that sometimes engineers cannot answer the questions. For example, the question may specify a monitor point and ask about a certain waveform; the engineer may not be certain what the waveform should look like. If the dialogue were conducted from a visual display unit instead of a typewriter, then the expected pulse shape could be displayed.

ATLAS HARDWARE MODIFICATIONS

Frank Fennel of I.C.T. Ltd. is modifying some of the S.R.C. Atlas central machine hardware along the general lines indicated previously. A second console has been built which monitors nearly every significant point in the machine. At present the monitored positions light lamps. Various freeze facilities are being added by means of simple checkers (the basic one consists of two floating buffers and a non-equivalence circuit). Plug and socket connections have been wired on the main frames so the checkers can be manually put into position. An assessment of engineering success will be made over the next few months which will decide how much further effort is put into the project. It has been found that to check some parts of the machine (e.g. function decoding) it is necessary to build checkers which duplicate the relevant machine hardware.

CONCLUSIONS

The system outlined in this paper might be of considerable use in the maintenance of large computer systems. A perceptive program which can obtain detailed information about the machine it has to maintain is not an unrealistic proposition. It is possible to envisage a satellite computer deciding to vary the voltage levels over a small area of a large machine, to monitor the behaviour, and to change packages without engineering intervention.

The situation will not be radically altered in the next generation of multi-layer integrated circuit machines. For as far ahead as can be seen, if an area of a computer has intermittent failures there will be some interest in finding out about them.

There will be a need for a formalized accumulative record of machine faults both for local maintenance help and for deciding whether any failures are general enough to warrant package modifications or design changes.

The best test programs at present for a large computer are the operating system and user's programs. Engineers' test programs should eventually become redundant except for finding the simplest of solid faults—in which environment they are reasonably successful.

At the S.R.C. Atlas Laboratory a satellite will shortly be attached to Atlas. The machines will share a common disc file and also be linked to look like peripherals to each other (*see* Baylis, 1966, 1966a). This will provide a good framework, in view of the work already done, for a project to produce a more sophisticated 'engineer's assistant'.

REFERENCES

- Baylis, M.H.J. (1966), Time-sharing on Atlas. *S.R.C. Atlas Laboratory publication*. Chilton: Atlas Computer Laboratory.
- Baylis, M.H.J. (1966a), Satellite Survey 1966. *S.R.C. Atlas Laboratory publication*. Chilton: Atlas Computer Laboratory.
- LISP 1.5 programmer's manual* (1966), Cambridge, Mass.: MIT Press.
- du Plessis, A.A. (1967), The Atlas engineer's assistant. *S.R.C. Atlas Laboratory publication*. Chilton: Atlas Computer Laboratory.
- Ross, D.T. (1961), A generalised technique for symbol manipulation and numerical calculation. *Communs Ass. comput. Mach.*, 4, 147-50.
- Ross, D.T. & Rodriguez, J.E. (1963), Theoretical foundations for the computer aided design system. *AFIPS*, 23, Spring J.C.C., 305-22.
- Sutherland, I.E. (1963), Sketchpad, a man-machine communications system. *AFIPS* 23, Spring J.C.C., 329-46.

COGNITIVE PROCESSES: METHODS AND MODELS

The Syntactic Analysis of English by Machine

J. P. Thorne

Department of English Language

P. Bratley and

H. Dewar

Department of Computer Science

University of Edinburgh

1. INTRODUCTION

In this paper we describe a program which will assign deep and surface structure analyses to an infinite number of English sentences.¹ The design of this program differs in several respects from that of other automatic parsers presently in existence. All these differences are a consequence of the particular aim we have pursued in writing the program, which represents an attempt to construct a device that will not only assign a syntactic analysis to any English sentence—that is, a record of the syntactic structure that the native speaker perceives in any English sentence—but which also, to some extent, simulates the way in which he perceives this structure. This is not to say that the analyzer differs from others because we have based its design upon the findings of psycholinguistic experiments. For one thing very few experiments on the perception of syntactic structure have been carried out and for the most part the results have been fairly inconclusive. But it is the case that we have, as far as possible, treated the task of constructing an automatic parser as being itself a psycholinguistic experiment. That is to say, any proposal regarding the possible operation of the program has been judged (mainly as the result of introspection) according to whether or not it seemed to be consistent with human behaviour. And this has led to our incorporating certain features which are absent from other automatic parsing systems.

Among the most notable of these features is the program's ability to assign syntactic labels to an infinite number of words while operating with a finite dictionary. As far as we know, all other automatic parsers of English (or

¹ This work was supported by the Office for Scientific and Technical Information Grant No. ID/102/2/06 to Professor Angus McIntosh. H. Whitfield and D.J. Dakin have also been associated with the work at various times.

Russian, etc.) are constructed on the assumption that they will eventually incorporate a dictionary containing every word in the English (Russian, etc.) language and listing the parts of speech to which each word can be assigned. It seemed to us essential that a syntactic analyzer should be able to deal with any sentence in the language without having to have access to such a dictionary, not only because the compilation of such a dictionary is a quite impracticable task, but because it seemed to us extremely unlikely that a speaker of the language needs to internalize and employ such a dictionary in order to recognize the syntactic structure of sentences in his language, if only because people are obviously able to recognize the syntactic structure of sentences containing words that they have never heard before.

Another important characteristic of the program is that it only needs to make one pass through the sentence it is analyzing and that any element in the sentence is analyzed once and once only. Again our reason for ensuring that the program should not call for multiple passes to be made through the sentence under analysis is the result of our conviction that under ordinary circumstances human beings do not need more than one, although (as with getting the program to operate with only a finite dictionary) it hardly surprised us to discover that this also had a significant bearing on the efficiency of the program.

But undoubtedly the most important decision that resulted from our attempt to construct a model for the perception of syntactic structure was our decision that the program should assign both deep and surface structure analyses to sentences. Our use of the terms 'deep structure' and 'surface structure' can be briefly (though inadequately) explained by the following example. Consider the sentence *The girl I liked left*. Any English speaker, having heard this sentence, possesses the information contained in the statement that in this sentence *The girl I liked* is the subject and *left* is the predicate, and that *The* is a definite article, *girl* a noun, etc. This is information about the surface structure of the sentence. But any English speaker also knows that in this sentence *The girl* is not only the subject of *left* but is also the object of *liked*, even though, of course, the correct surface form of the sentence is *The girl I liked left* not *The girl I liked the girl left* or *The girl I liked her left*. Since we derive this information from the sentence without making any reference to the context—as here where it is used merely as an example—then it is clear that we derive this information not from the context (as is sometimes suggested) but from our perception of the structure of the sentence, even though, as we have seen, this part of the structure is not actually realized in the surface form of the sentence. This information forms part of the deep structure of the sentence. Nearly all the automatic parsers now in operation give information only about the surface structure of sentences.¹

¹ For accounts of other programs which assign deep structure analyses see Kay (1967), Petrick (1965), and Zwicky *et al.* (1965). For an explication of the concepts of deep and surface structure see Chomsky (1965).

2. METHODS AND APPROACH

Four constraints were originally imposed on the program:

- (i) the program must not rely on looking up every word of an input sentence in a dictionary;
- (ii) it must process each sentence in a single left-to-right pass;
- (iii) it must be constructed in such a way that at every stage of the analysis process, some 'memory' would be remembering the decisions made up to the current point in the sentence, and some predictive process be looking ahead to what could possibly arrive next;
- (iv) (in fact a consequence of (ii)) it must analyze each part of the sentence once and once only.

To these constraints we subsequently added a fifth:

- (v) it must undertake deep and surface structure analyses simultaneously.

2.1

It does not seem to us reasonable to suppose that a person hearing or reading a sentence in any sense 'looks up each word in an internalized dictionary' in order to assign form-class information to it before proceeding with the analysis process. Several reasons can be advanced against such a hypothesis, perhaps the most cogent being that people can analyze sentences containing words which they have never heard before and which, therefore, they certainly cannot have in any internalized dictionary. For instance, nobody should have any difficulty in deciding the syntactic structure of the sentence *He has gone to shoot a grison*, although most people will not have heard of a *grison* before and will not know what it is. Again, with *He is going to displease his mother-in-law* or *She will be furious*, although particular words may be unknown the syntactic structures of the sentences are clear. Indeed, far from its being the case that from a knowledge of the form classes to which particular words belong one deduces the structure of a sentence, it seems much more likely that from the rest of the structure of a sentence one can derive the classification of words in it. In the three instances: *He ruled with an iron hand*, *Strike while the iron is hot*, and *I will iron your shirt tomorrow*, the knowledge that *iron* can be an adjective or a noun or a verb would clearly be of no help in determining the complete syntactic structure of the sentences. In fact, it is because one recognizes the structure of the sentences that one knows that *iron* is an adjective in the first, a noun in the second, and a verb in the third.

This is not true of all words, however. It seems that words such as prepositions, pronouns, and conjunctions, which have fixed syntactic functions, play an essential part in the recognition of sentence structure. Words with fixed syntactic functions we call *closed-class words*, and all others *open-class words*, for the reason that all the former classes have a finite and, in fact, determinate

number of members while the latter have, in principle, an infinite number of members and are indefinitely extendable. Accordingly the program is designed to require access to a dictionary containing only the closed-class words.

2.2

Very many sentences are syntactically ambiguous. Nearly all sentences contain elements which taken separately are ambiguous. It is our conviction that usually in listening to such sentences all possible interpretations of these elements are considered simultaneously (certain kinds of jokes providing a possible exception). For this reason the analyzer is designed to go through each sentence in a single left-to-right pass. If any part of an input sentence is syntactically ambiguous, then all the possible analyses are developed simultaneously. That is to say, it is not the case that it first tries one analysis and then backtracks to see if any others are possible. The progress of the analysis process is recorded on a continually growing data structure, and when the end of the sentence is encountered, each possible analysis is to be found represented as a path through this structure.

2.3

There is a good deal of evidence to suggest that the efficiency with which human beings recognize the syntactic structure of sentences is to some extent the result of their ability, having heard part of a sentence, to predict the structure of the remainder. If one hears a sentence which breaks off suddenly in the middle, one is not left feeling that the sentence is ungrammatical but rather that the end of the sentence is missing. This seems to suggest that some kind of predictive mechanism is at work and that at some stage an expected outcome did not in fact occur. It seems likely that having heard (say) the subject of a sentence, we are then predicting, in some sense, the occurrence of a verb to go with that subject. If we look again at two of the examples given above, *He ruled with an iron hand* and *I will iron your shirt tomorrow*, then it is quite obvious that the kinds of words that can follow *He ruled with an . . .* and *I will . . .* are different. Roughly speaking, in the first case we are predicting either a noun or an adjective, while in the second case we are predicting a verb. Accordingly the operation of the analyzer is a process of making and checking predictions about syntactic structure. The source of information for these predictions is a representation of a grammar.

2.4

If it is reasonable to assume that in recognizing the syntactic structure of a sentence one considers all possible interpretations of ambiguous items simultaneously, then it seems equally reasonable to assume the converse—that unambiguous items are analyzed only once, no matter how many different possible analyses of the whole sentence or parts of the sentence they may enter into. Thus if the first part of a sentence has been analyzed in two different

ways and it then turns out that both analyses lead to an identical set of predictions, the rest of the sentence should be analyzed once only. It should not be necessary to produce two distinct but identical analyses for the rest of the sentence. Take for example the sentence *When he has fixed dates he will ring us*. There is an obvious ambiguity here (in one interpretation *has* is taken as an auxiliary, in the other as a main verb). But the ambiguity is confined to the first clause, and both analyses lead to identical predictions being made at the start of the second. Similarly in the sentence *He rolled up the bright red carpet*, the phrase *the bright red carpet* is either the object of the phrasal verb *rolled up* or the object of the preposition *up*. But the analysis of *the bright red carpet* as a noun phrase is the same for both interpretations. In these cases the program operates in such a way that analysis paths leading to the same predictions are conflated.

2.5

In Spring 1966 a first simple model incorporating all the features described above was implemented on KDF9. It had been deliberately designed to analyze only the surface structure of input sentences because at that time our idea was that this surface-structure analysis should then be used as the input to a separate deep-structure analyzer. Despite this limitation, the model (Thorne *et al.*, 1966) was extremely useful, as it enabled us successfully to test for the first time the feasibility of using only a closed-class dictionary, and of using a predictive technique conflating identical predictions.

But undoubtedly its main usefulness lay in its demonstrating to us that this approach was essentially wrong. The trouble with a two-stage analyzer, that is, one comprising two components, a surface structure analyzer and a deep structure analyzer, the output of the first being the input to the second, is that in the case of many types of sentences the surface structure analyzer produces a large number of incorrect analyses which the deep structure analyzer has to discard. This is particularly noticeable in the case of sentences containing embedded clauses or conjunctions like *and* and *but* (where the crucial factor is the surface structure analyzer's inability to take account of deletions in the deep structure). Any *ad hoc* attempts to reduce the number of analyses turned out to have the undesirable consequence that in many cases the number was reduced to zero. It became clear to us that the large number of wrong analyses produced by the surface analyzer was a direct result of the fact that it had to work independently, without access to any deep structure information, and that an important consequence of constructing an analyzer which would not only undertake deep as well as surface structure analysis but which would undertake both tasks simultaneously, would be that the number of incorrect surface structure parsings would be greatly reduced. The output of the present model shows that it is indeed the case. Again one notices that the result of designing the analyzer bearing in mind human behaviour (presumably in perceiving the syntactic structure of a sentence we do not first

perceive the surface structure and then work out the deep structure) is a considerable gain in its efficiency.

3. OUTLINE OF THE ANALYZER

The input consists of English sentences in a more or less normal orthographic form. The sentences are read and processed a word at a time and at the end of each sentence the analysis or analyses produced are displayed on the printer. In analyzing the sentences the program makes use of information about the syntactic functions of individual words derived from the closed-class dictionary and information about sentence structure derived from a representation of a grammar. At the start of a sentence the analyzer has a fixed set of initial predictions, and as it progresses through the sentence the existing predictions are tested, those which are satisfied are recorded, and new predictions are formed on the basis of the satisfied predictions and information about sentence structure obtained from the grammar. At the end of the sentence the record of satisfied predictions indicates all the possible analyses of the sentence with respect to the grammar. Because there is in principle no upper limit to the number of predictions which may arise in the course of analysis it is necessary to use some form of dynamic data structure to record the state of the predictions at successive points in the sentence. The analyses must also be recorded for subsequent output. In the surface-structure analyzer (as in other predictive analyzers) the prediction structure and the analysis structure were distinct. In the present model a single structure serves for both purposes. This may be viewed in two ways, either as an analysis record controlling the selection of further predictions from the grammar, or as a prediction structure in which fulfilled predictions are not discarded but are retained to become the record of the analysis. This method of implementation reflects quite literally the predictive principle; that is, that the way in which the later part of a sentence may be analyzed depends upon the analysis of the earlier part.

4. THE GRAMMAR

The grammar incorporated in the analyzer is a form of transformational grammar. A transformational grammar consists of a base component and a transformational component. The base component specifies a set of strings which correspond roughly to the simple or kernel sentences of the language and the transformational component accounts for complex sentences by deriving them from the basic underlying strings. The base component of the grammar associated with the analyzer here described differs in a number of respects from the type usually proposed. The three most significant differences are:

1. The base component is (structurally) a *regular grammar* rather than the customary *context-free phrase-structure grammar*.¹ A regular (or finite-state)

¹ For an account of regular grammars (expressions) in the context of automata theory see Kleene (1956).

grammar enumerates a set of strings without assigning any hierarchical structure to them. The adequacy of such a grammar as a base component depends in part on acceptance of the principle that all recursive constructions in a language necessarily involve transformations, since they cannot be generated by means of the base rules alone. In the case of English this principle has the possibly controversial consequence that all noun phrases have to be regarded as transforms because of the occurrence of phrases like *the old man's hat*, in which the relation of determiner is recursively realized by a possessive. (In fact, since possessives are an example of left-branching recursion, they are not beyond the weak generative capacity of a regular grammar; however, the genuinely hierarchical structure of the construction cannot be represented by such a grammar.) The total exclusion of phrase-structure rules from the grammar may be felt to be in some ways too strong a constraint. On the other hand, it means that we avoid some of the problems arising from the fact that in certain cases conventional phrase-structure grammars assign too much structure to base strings, with the result that statements of transformations become uneconomical.

2. The form of the grammar also allows for the specification of properties like number, case, tense, etc., in the form of lists of feature-values associated with the elements in the rules. This provides an apparatus for sub-categorization and for the application of rules for feature concord. The fact that sub-categorial distinctions are represented in this form and not by the addition of extra categories results in economy both in the grammar and in the analysis procedure.

3. Explicit recognition is given in the grammar to the distinction between the concepts of syntactic form (involving such classificatory terms as *article*, *noun*, *nominal clause*, etc.) and syntactic function (involving such relational terms as *determiner*, *head*, *subject*, etc.). It is of course desirable that a syntactic analyzer for a natural language should not simply label the components of a sentence with category names but should also mark the relations which hold among the components, but it is also the case that markers of syntactic relations (hereafter SRMs) are more appropriate for the statement of certain generalizations such as those affecting feature concord.

Thus the base component is a regular grammar specifying a set of unstratified strings. The elements in the strings have three constituents: an SRM, a category name, and a list of feature values. The base component directly enumerates the prelexical strings for such simple sentences as *I like Sylvia*, *She visited him yesterday*, *He must have moved*, and so forth. In addition the grammar contains substitution rules and transformational rules proper. The substitution rules govern the realization of elements in the grammar—for the most part in a context-sensitive manner—either by lexical items or by transforms. The structural simplicity of the base makes it possible to regard the majority of the transformational rules as meta-rules in the sense that they operate on other rules to produce derived rules rather than operating on

structural descriptions to produce new structural descriptions. A practical consequence of regarding transformational rules in this way is that in many cases the effect of a transformational rule can be represented by a small number of derived rules of the same form as the base rules. Accordingly the grammar table (hereafter GT) actually used by the analysis program has the form of a finite-state network or directed graph—a form appropriate for the representation of a regular grammar—but it contains in addition to the base rules the derived rules for those transformations whose resultants are representable within this framework. The remaining transformational rules have to be applied dynamically in the course of the analysis procedure. It should be noted that the inclusion of SRMs and features in the grammar ensures that relevant syntactic information is preserved in the derived rules (e.g. selectional constraints persevere and basic relations remain marked).

4.1

The output of the analyzer reflects the form of the grammar. Analyses are displayed in the printout in a series of levels, the level structure reflecting the transformational structure of the sentence. The analysis produced for the simple sentence *She visited him yesterday* is

1	SE:STAT	TE:.	1
2	SU:she AV:visited OB:him MO:yesterday		2

(The letters preceding the colons are abbreviations for SRMs. SE stands for *sentence*, SU for *subject*, AV for *active verb*, OB for *object*, MO for *modifier*, and TE for *terminator*. See Appendix III for full details of the conventions used and for examples.) It will be noted that within the top-level phrase, STATEMENT, no structure has been assigned and the syntactic relations are shown as being realized directly by individual words. For the complex sentence *He asked who admired Descartes* the analysis is

1	SE:STAT	TE:.	1
2	SU:he AV:asked OB:INDQ		2
3	SU:who AV:admired OB:Descartes		3

In this case the relation of object of the main clause is shown as being realized by a transform, INDirect question.

The way in which deep-structure information is preserved, despite the considerable differences produced by different transformations in the form of identical deep-structure elements, is illustrated by the analysis for the sentence *Mary hates my teasing her*.

1	SE:STAT	TE:.	1
2	SU:Mary AV:hates OB:GER		2
3	SU:my AV:teasing OB:her		3

In the interests of keeping the printout compact, the feature values associated with the components of the sentence are not reproduced in the output. Information about such facts as the relation of *who* to the pronouns *he* and *she* and the relation of *my* in the GERund to the pronoun *I* is contained in the dictionary, so that, if one wanted to, it would be possible to amend the program in such a way that the output relating to the deep structure would be in the form of kernel sentences, e.g. in the case of the last example *Mary hates it* and *I tease her*.

5. THE CLOSED-CLASS DICTIONARY

It has already been mentioned that the program does not need to have access to a complete dictionary giving the possible form-class assignments of every word. It operates with a list containing only certain classes of words. Some further consideration will now be given to the composition and use of this closed-class dictionary (CCD).

5.1

Three types of items are held in the CCD. First, there is a list of the grammatical formatives such as prepositions, pronouns, conjunctions, articles, and so on; secondly, there is a list of special verbs; and thirdly, there is a list of suffixes. Syntactic information about each word in the CCD is provided by one or more code words, which constitute the dictionary entry for the item. Each code word specifies a category or form class to which the item belongs and a list of values for the features associated with that category.

The relatively small class of items which belong to the first type mentioned above must be listed exhaustively in the CCD. This is necessary because words of this type provide essential information about the structure of any sentence.

The second list is of words which give information about certain kinds of sentence structure. For instance, it may be remarked that *Fred gave the dog biscuits* is ambiguous, whereas *Fred lost the dog biscuits* is not. This reflects the fact that the verbs *give* and *lose* have different properties. In particular, *give* may take two objects while *lose* may take only one object. We could have chosen to treat all open-class words as potential double-object verbs, which would result in an unacceptable analysis being produced for the second sentence. Instead it seems preferable to say that open-class words, when treated as verbs, can take at most one object, and to list verbs like *give* in the CCD so that two correct analyses are produced for the first sentence and only one for the second. Similarly, certain other classes of verbs are listed in the CCD; for instance, those which take a complement instead of an object (*He looked a fool*), and those which take an object and an infinitive (*He made her cry*).

The suffixes contained in the CCD are restricted to inflections. Examples of the endings included are *-s*, *-ed*, *-ing*, *-s*, and *-s'*. Like the grammatical formatives, these elements carry essential information about sentence structure. The dictionary look-up procedure in the program automatically recognizes

these endings and detaches them from the stem of the word. Simple checks are incorporated to ensure that words such as *bed* and *bus* are not treated as if they ended with the inflectional *-ed* or *-s*. Other exceptions, either individual words (e.g. *news*, *lens*) or non-inflectional ending classes (e.g. *-ss*, *-ous*), must be listed in the CCD. Conversely, words for which the information normally carried by the suffixes is specified in a non-standard way (e.g. the past tense forms of strong verbs) must also be listed as exceptions.

A fuller account of the contents of the CCD has been given elsewhere (Bratley and Dakin, 1968). It is worth emphasizing again that, compared to a complete dictionary of English, the CCD is very short. The list of grammatical formatives contains a few hundred items, and while the lists of words of the second type have not been fully enumerated at the present time, the total number of items in the dictionary should not exceed 2000.

5.2

The CCD is stored as a tree structure so that, for instance, the four words *this*, *the*, *them*, and *to* would be represented in the form shown in figure 1, where

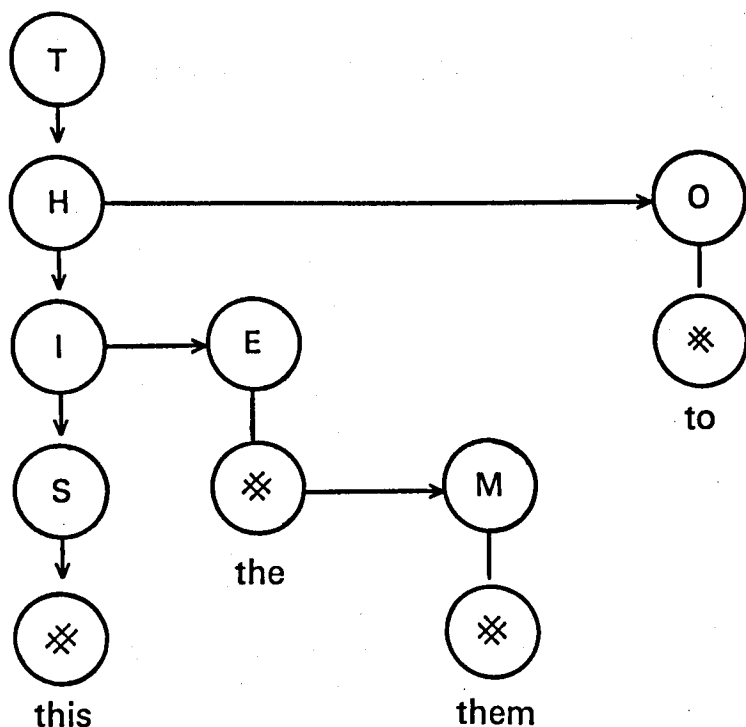


Figure 1

- (i) ✖, representing 'end-of-word', is treated as an extra letter;
- (ii) downward links represent progression through the successive letters of a word;
- (iii) cross-wise links represent an alternative choice of letter at the same position in a word;
- (iv) there are pointers from the 'end-of-word' nodes to the relevant dictionary entries.

The endings are stored in a similar tree structure, and the use of default links from the main structure enables inflected words to be recognized in almost exactly the same way as listed words.

Special entries are contained in the dictionary for open-class words, proper names, and numerals. Since the information carried by an inflected word may not be categorical but may be a function of the stem of the word as well as the suffix, provision is made for including with each item an operation code which is held in the 'end-of-word' node. This code specifies an operation to be performed on the feature values entered for the stem of the word or, if the stem is not listed, on the entry for open-class words. The use of operation codes is not in fact confined to the ending-classes; they may also be used to indicate the relationship of an individual word to another item. The following fragment of the CCD in the form in which it is submitted to the set-up routine illustrates the kind of facilities provided. (In the interests of simplifying the set-up procedure, feature values are written as numbers corresponding to the required binary patterns.)

```

-s = -(23)
-ed = -(24)
-ing = -(25)
-ss = —
the : art  0 1 0 3 0
a,an : art  0 1 0 1 0
can,must : modl 0 1 7 8 0
           : noun 0 1 3 1 0
they : pro  0 1 1 2 1
their=they (1)
know : verb 0 3 6 8 50
knew=know (10)
known=knew (9)
fought = -(4)

```

6. THE ANALYSIS PROCEDURE

The task of the analysis procedure is essentially the progressive construction of a data structure in which the predictions realized by successive words of the sentence to be analyzed are recorded and which is then used to determine what new predictions may be made for the following words. The procedure has available to it the stored GT and the dictionary look-up routine which

assigns codings to each input word in the manner described in the preceding section. The analysis structure produced is basically a diverging tree with nodes which may themselves represent sub-trees. At the start of the operation the structure consists of a single node which represents the root of the main tree.

For each word of the sentence the 'open ends' of the structure (i.e. the latest active nodes so far added on each analysis path) are traversed and the predictions specified by them are tested. In general, the immediate predictions which may be made from a node are determined by means of a reference to an element in the GT, the successors of this element in the finite-state network representing the possible continuations of the analysis path, but in a number of cases, some of which are mentioned below, they are computed from information held in the analysis structure. For a prediction of an element which is realizable directly by a lexical item, the codings of the input word are matched against the predicted values and if the match is successful a new simple node is added to the analysis path. In the case of a prediction realizable by a transform phrase, reference is made to a table (TPT) to determine if this is the first prediction of the phrase encountered for the current word. If it is, a new tree is established with a root node specifying the initial predictions for the phrase and these predictions are in turn investigated. If it turns out that none of the initial predictions is successful, a failure indicator is set in the TPT, otherwise a node representing the new sub-tree is added to the analysis path from which the original prediction was made. Where a transform phrase prediction is not the first encountered, it is necessary only to perform the last-mentioned step—unless the failure indicator is set in the TPT, in which case no action is taken. Thus multiple predictions of the same type of phrase give rise to the establishment of a single sub-tree.

If a node terminates a complete analysis of a transform phrase, the overall feature values for the phrase analysis are computed and recorded so that subsequently the higher-level paths on which the phrase prediction was encountered can be reactivated. The reactivation does not occur until all complete analyses of the phrase ending on the same word have been recognized, so that a single reactivation can be made with feature values which represent all the analyses. For example, one possible analysis of the noun phrase *the research computing needs*, in which *computing needs* is taken as a relative clause, makes it singular, while the other makes it plural; the value for the feature of number that is required for the whole phrase is therefore singular-or-plural. The feature values for a phrase are matched against the predicted values on the higher-level paths in the same way as for lexical items.

Certain types of node (e.g. the root nodes of sub-trees) are created provisionally and do not remain on the structure unless they eventually lead to a successful analysis of the current word, but otherwise the structure grows progressively and analysis paths which peter out are not removed, so that, in fact, at the end of the sentence the analysis structure represents not only all complete analyses but all partial analyses as well. However, the recognition of

identical phrase predictions—which prevents ambiguity at a higher level from being extended to a lower level—and the single representation of multiple analyses of a phrase—which prevents ambiguity at a lower level from being extended to a higher level—serve to keep the size of the analysis structure and the amount of processing required within reasonable bounds (see figure 2).

When the end of a sentence has been reached, all complete analyses recorded on the analysis structure are traced and printed out. If at any point before the end of the sentence no prediction remains, this indicates that the grammar does not provide any analysis for the sentence (either because the grammar is incomplete or because the sentence is in fact ungrammatical), and the comment *no complete analyses* is printed out.

It was indicated in the section describing the grammar associated with the analyzer that while the resultants of most transformational rules are incorporated in the G T, the effect of some cannot be completely specified in this way and must be developed in the course of the analysis process. The two main instances of this relate to constructions involving inversion and constructions involving co-ordination.

Inversion is exemplified in such interrogative and relative constructions as

Is he bringing his wife? (inversion of auxiliary)

the book which he bought (inversion of object of *bought*)

the boy who she said kicked her (inversion of subject of *kicked*)

Which hat will she buy? (inversion of both auxiliary and object)

The rules for these constructions are formulated in such a way that the inverted words or phrases are accepted at the point at which they occur in the sentence (that is, their surface structure position), but with the assignment of a dummy SRM (OO) indicating that in general it is not possible at this point to specify what syntactic relation is exemplified by the element. All subsequent nodes added to an analysis path involving an inverted element include a link back to it. The effect of this is to make the item available for fulfilling a subsequent prediction, so that in these cases a prediction is satisfied by a null input (that is, without using up any words of the sentence). The realization of a prediction in this way is marked in the output by an asterisk following the SRM. Thus for the examples listed above the analysis printout would be:

1	SE:QUES				TE:?	1
2	OO:is	SU:he	AU:*	AV:bringing	OB:CNP	2
3				DE:his	HE:wife	3

DE:the	HE:book	AT:REL			
	OO:which	SU:he	AV:purchased	OB:*	

DE:the	HE:boy	AT:REL			
	OO:who	SU:she	AV:said	OB:INDS	
		SU:*	AV:kicked	OB:her	

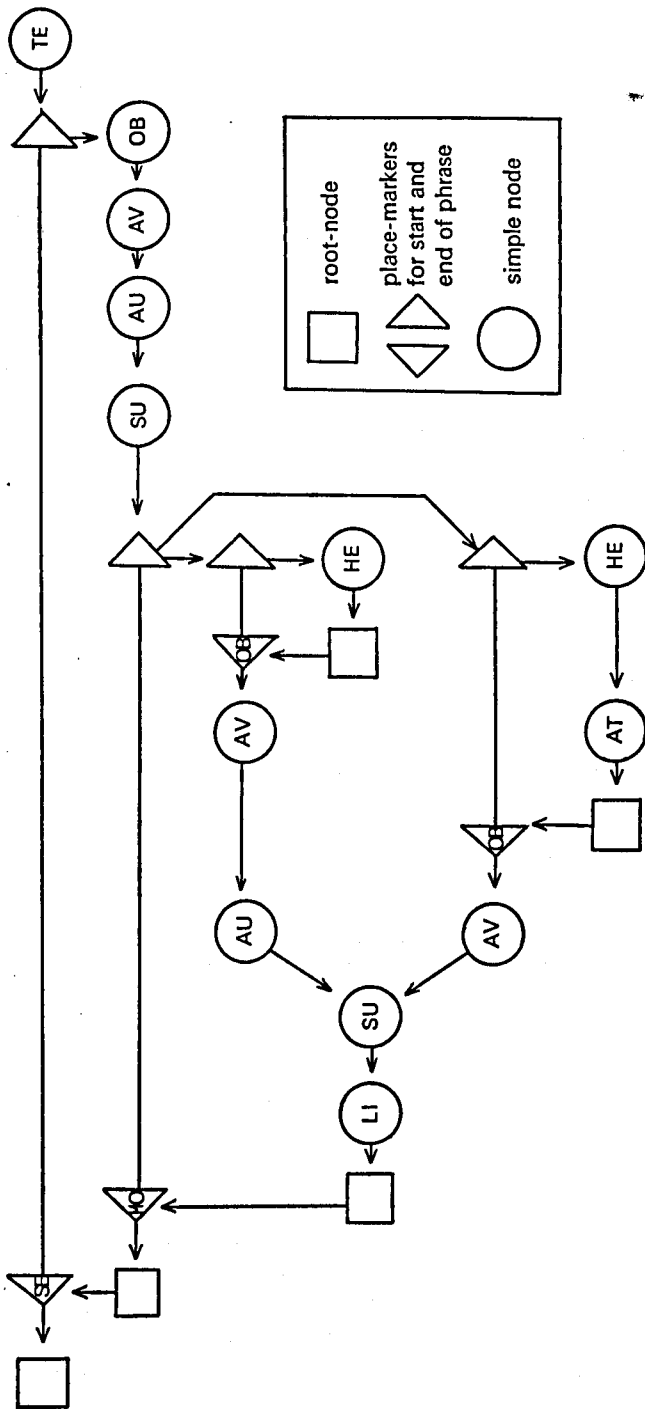


Figure 2. Illustration of analysis structure for sentence 22 (showing only those nodes which figure in complete analyses)

1	SE:QUES				TE:?	1	
2	OO:CNP	OO:will	SU:she	AU:*	AV:buy	OB:*	2
3	DE:which	HE:hat					3

An asterisk following an SRM more generally indicates the deletion of an element, as for example in imperative sentences like *Go*, where it indicates the non-realization of the deep structure subject.

1	SE:IMP		TE:.			1
2	SU:*	AV:go				2

Similarly in sentences like *I met a man I know*, the combination of the two conventions (OO:*) is used to indicate the absence of the relative pronoun. As with the full form of the relative clause, the deep structure position is also indicated (in this case by OB:*)).

1	SE:STAT				TE:.	1	
2	SU:I	AV:met	OB:CNP			2	
3		DE:a	HE:man	AT:REL		3	
4			OO:*	SU:I	AV:know	OB:*	4

The procedure for dealing with co-ordination (constructions involving *and*, *or*, etc.) consists essentially in the reinstatement, following the co-ordinator, of predictions which have been made at an earlier point. Co-ordinators are not in fact predicted, and when they are encountered they are accepted without reference to the GT. The nodes created for these words record previous predictions which may now be reinstated, all subsequent nodes added to the analysis paths being marked with a co-ordination link (cf. the treatment of inversion). This link indicates that at some point the two limbs of the co-ordination must be 'brought together', i.e. analysis of the part of the sentence following the co-ordinator word must reach the same point as that reached immediately before it, so that a prediction common to both is satisfied.

In the output the point from which predictions were reinstated is marked with a left bracket and the point at which a successful common prediction was satisfied is marked by a right bracket. The brackets thus indicate the scope of the co-ordination. The analysis of the sentences *She danced and sang* and *Have John and his sister arrived?* is shown below. (For further examples see Appendix III.)

1	SE:STAT		TE:.			1
2	SU:she	(AV:danced and AV:sang)				2

1	SE:QUES			TE:?	1	
2	OO:have	(SU:John and SU:CNP	)	AU:*	AV:arrived	2
3		DE:his	HE:sister			3

Some of the problems of dealing with this type of construction are rather intractable. In particular, it is difficult to devise suitable conditions regarding the degree of similarity which has to be exhibited by the two limbs of a co-ordination. On the one hand it is possible to impose the very strict constraint that only predictions which have been *satisfied* in the first limb may be re-instated for the second, thus requiring identity of analysis for the two limbs. However, this would preclude the successful analysis of such entirely acceptable sentences as *Is he or is he not coming?* and *He asked for and was given a glass of water*. On the other hand, if the principle is adopted of permitting the reinstatement of any prediction which has been *made* (but not necessarily satisfied) in the first limb, there is a danger of accepting such examples of syntactic syllepsis as **He said his prayers and that he died a happy man*. In fact we have adopted a principle which is nearer the second of those cited above but which incorporates requirements of feature concord that effectively exclude at least the more obviously undesirable cases. The ordering of transformations associated with inversion and co-ordination relative to each other is also problematic. The two (apparently equivalent) analyses produced for sentence 28 reflect this difficulty. The problem here is not simply to eliminate one analysis, but to discover criteria for determining which this should be.

7. CONCLUSION

We have shown that it is possible to construct an effective syntactic analyzer operating under the constraints listed in Section 2. As one would expect, it has certain limitations.

1. A result of not using a full dictionary of English is, of course, that in the case of some sentences incorrect analyses are produced as well as the correct analysis. For example, in the case of the sentence *The cat adores fish*, as well as the analysis in which *adores* is taken as a verb, an analysis is produced in which *adores* is taken as a noun. (Notice that in the case of a sentence like *The girl guides fish* one would require these two analyses.) It should be emphasized that only a small number of incorrect analyses are attributable simply to a lack of form class information. It follows from this that in order to obtain a substantial improvement by supplying the program with a complete dictionary it would be necessary for it to provide more than merely form class information. It would, in fact, need to contain information about other features, such as, in the case of nouns, whether they are abstract or concrete, animate or inanimate, etc. A possible extension of the program which would enable it to derive information about such features automatically is discussed in Appendix I.

2. The program is not a general-purpose analyzer for arbitrary transformational grammars. It was pointed out in Section 4 that in order to give effect to certain transformational rules specific procedures had to be incorporated into the program. Given the present formulation of transformational rules in linguistic theory, it is doubtful whether a generalized algorithm is possible.

3. The attempt to present a perspicuous representation relating deep structure information to the actual surface structure of the sentence has led in some cases to a certain ambivalence—for example, the assignment of SRMS to grammatical formatives as if they were lexical elements rather than realizations of bundles of syntactic features.

4. At the present time the analyzer takes no account of *derivational affixes*. As a result it is unable to recognize nominalizations like *declaration*, *performance*, *management*, etc., and relate them to their underlying sentential forms.

5. There are deficiencies in the treatment of certain kinds of constructions. Some of those affecting the analysis of sentences containing conjunctions like *and* have already been noted in Section 6. Similar problems (in an even more acute form) arise in the case of sentences containing words like *as* and *than*. In fact such sentences are outside the range of the analyzer. The reason for this—the partial failure to handle *and* and the total failure to handle *as* and *than*—is very simple. It is directly related to the fact that we ourselves have an incomplete understanding of the grammar of *and* and only vague ideas about the grammar of *as* and *than*. Only when developments in linguistic theory have resulted in a formalism capable of explicating the structure of these sentences will it be reasonable to expect an automatic analyzer to produce adequate analyses for sentences of this kind. It is necessary to make this obvious point in view of the many claims made in recent years to the effect that the development of an automatic syntactic analyzer will in itself help in solving these kinds of linguistic problems.

REFERENCES

- Bratley, P. & Dakin, J. (1968), A limited dictionary for syntactic analysis. *Machine Intelligence 2*, pp. 173–81 (eds Dale, E. and Michie, D.). Edinburgh: Oliver and Boyd.
- Chomsky, N. (1965), *Aspects of the theory of syntax*. Cambridge, Mass.: MIT Press.
- Kay, M. (1967), Experiments with a powerful parser. *Deuxième conférence internationale sur le traitement automatique des langues*. Grenoble.
- Kleene, S.C. (1965), Representation of events in nerve nets and finite automata. *Automata Studies*. Princeton.
- Petrack, S.R. (1965), *A recognition procedure for transformational grammars*. Ph.D. thesis, MIT.
- Thorne, J.P., Dewar, H., Whitfield, H. & Bratley, P. (1966), A model for the perception of syntactic structure. *Colloque international sur l'informatique*. Toulouse.
- Zwicky, A.M., Friedman, J., Hall, B.C. & Walker, D.E. (1965), The MITRE syntactic analysis procedure for transformational grammars. *AFIPS*, 27. Fall J.C.C. Washington D.C.: Spartan Books.

APPENDIX I: A POSSIBLE EXTENSION OF THE PROGRAM

There is an interesting extension that could be made to the program. For the reasons given, words like *boy* and *laugh* are not entered in the dictionary employed in the analysis procedure, which means that no information for them is obtained from the dictionary. But using information derived from the analyses it produces the program could, so to speak, 'learn' that *boy* is a noun and *laugh* a verb, and could construct for itself another dictionary—an open-class dictionary—in which this information would be stored.

Notice that in many cases during the early stages of running the program it is inevitable that incorrect entries would be made in the open-class dictionary. Given the sentence *The cat adores fish*, the analyzer produces two analyses—the desired analysis and one in which *The cat adores* is taken as a noun phrase on the analogy of phrases like *the boy scouts*. *Adores* would therefore be entered tentatively both as a verb and a noun. But it seems reasonable to suggest that after a while an automatic correction routine could be run on the open-class dictionary which, for example, would discover any word entered as being both a noun and a verb but for which, while there have been unambiguous instances of its being labelled as a verb, no cases have been found in which it has been labelled as a noun without, at the same time, an analysis being produced in which it has been labelled as a verb. In this case the dictionary entry for the word would be modified by the deletion of the label *noun*. If at a later stage the same sentence were submitted for analysis, two analyses would again be produced. But now both analyses could be checked against the open-class dictionary the program has itself constructed. If taking the sentence as ambiguous meant treating one of the words as a part of speech different from that the dictionary records it as belonging to, then this would be a sufficient reason for dropping this analysis. Given an analysis of a sentence in which every word functions as the part of speech as which it usually functions, we are unlikely also to accept another analysis for the sentence in which one of the words now functions in an entirely unexpected way. For example, no one is likely to take the sentence *Power corrupts* as an imperative, on the analogy of a sentence like *Bring water*, because this would involve taking *corrupts* as a noun and there is a perfectly acceptable analysis of the sentence in which *corrupts* functions in the expected way, as a verb.

Following out this procedure it would be possible for the program not only to acquire the information that *boy* is a noun and *laugh* a verb, but also the information that *laugh* is an intransitive verb. But if the program is to acquire all the information the English speaker has about these words, it is also necessary that it should contain the information that *boy*, for example, is a concrete noun and an animate noun. It is possible that this kind of information too might be automatically derived. For this to happen, however, it would first be necessary for information about such properties to be supplied for certain words. Say, for example, we were to include in the original closed-class

dictionary the word *surprise* plus the information that it is a verb that must always take an animate noun as its object. Then, when the sentence *The boy surprised the teacher* has been analyzed, the program will have learnt not only that *teacher* is a noun but also that it is an animate noun. If the next sentence to be analyzed is *The teacher laughed*, it would now have learnt that *laugh* is the kind of verb that can take an animate subject. In this way it seems that the original information concerning the properties of a few words could be spread over the whole lexicon.

Making the analytic procedure and the open-class dictionary arising from it operate together in this way would have the following effect: as the open-class dictionary using the information produced by the analyzer improved so too would the analyses produced. However, there are many problems here. Many verbs can take animate, inanimate and abstract subjects, and the fact that up to a certain point the program has not encountered an instance of a verb taking one type of subject is no guarantee that it will not do so. Moreover, it is by no means clear which verbs, or how many verbs, or even whether verbs, should be chosen as the starting point. Nevertheless, this looks as though it would be an interesting field for experiment.

APPENDIX II: PROGRAMMING DETAILS

The analysis program runs at present on the KDF9 computer at Edinburgh, which has 16K 48-bit words of core store. Core cycle time is about 4 to 6 microseconds, depending on overlapping, and simple fixed-point instructions take about 2 microseconds to execute in the high-speed registers. The machine-code instruction format is variable length, so that on average two or three instructions can be held in one computer word. Of the 16K words of core available, about 2K are taken up by the supervisor and operating system, about 4K are occupied by the instructions of the program, and the rest are available for data space. In addition to the analysis routines the program contains routines to accept and set up the grammar and dictionary in the required internal format. The fact that the whole CCD is small enough to be held in main store naturally results in a considerable economy in the overall processing time for a sentence. It is convenient to hold the compiled program, including the grammar and dictionary, on magnetic tape, but the program itself requires no backing store.

The program is written in Atlas Autocode, a high-level language akin to ALGOL. This provides the essential features of recursive procedure calling and the specification of parameters either by name or value, as well as a number of simple facilities for handling non-numeric data. As the version of the language currently available does not include provision for Boolean and shift operations, a few small procedures for manipulating operands have had to be programmed in machine code. The program currently runs to about 1000 lines of Atlas Autocode, equivalent to some ten thousand instructions after compilation.

Sentences to be analyzed are submitted in free format on paper tape produced by a Flexowriter. At present the input is subject to the restrictions that only proper names and the pronoun *I* may start with a capital letter and that abbreviations terminated by a period may not be used.

APPENDIX III: EXAMPLES OF OUTPUT

The computing time for each sentence and the total number of nodes in the analysis structure are recorded on the line following the sentence. The timing figure (in seconds to three decimal places) includes the time spent in consulting the CCD; it does not include the time spent in outputting the analyses. The arrangement of the printout is hierarchical, with the elements entering into the analysis of each category being printed out under the category name.

The abbreviations used are listed below.

SRMS (*syntactic relation markers*)

[SE	sentence]
TE	terminator
SU	subject
AV	active verb
OB	object
MO	modifier
AU	auxiliary
DE	determiner
HE	head (of noun phrase)
AT	attribute
IN	indirect object
LI	link (preposition or conjunction)
PO	prepositional object
PA	particle
CO	complement
OO	INVERTED ELEMENT

Category names

STAT	statement
QUES	question
IMP	imperative
INDS	indirect statement
INDQ	indirect question
INFC	infinitive clause
NOMC	nominal clause
PARC	participial clause
SUBC	subordinate clause
GER	gerund
REL	relative
PREC	prepositional clause
CNP	complex noun phrase

1 she visited him yesterday.

(time taken: 0-565 seconds nodes used: 8)

1	SE:STAT	TE:.	1
2	SU:she AV:visited OB:him MO:yesterday		2

2 he must have moved.

(time taken: 0-557 seconds nodes used: 12)

1	SE:STAT	TE:.	1
2	SU:he AU:must AV:have AV:moved		2

3 the cat adores fish.

(time taken: 1-044 seconds nodes used: 32)

1	SE:STAT	TE:.	1
2	SU:CNP AV:adores OB:CNP		2
3	DE:the HE:cat HE:fish		3

1	SE:STAT	TE:.	1
2	SU:CNP AV:fish		2
3	DE:the AT:cat HE:adores		3

4 this cat adores fish.

(time taken: 0-978 seconds nodes used: 28)

1	SE:STAT	TE:.	1
2	SU:CNP AV:adores OB:CNP		2
3	DE:this HE:cat HE:fish		3

5 go.

(time taken: 0-427 seconds nodes used: 11)

1 SE:IMP TE:.. 1
2 SU:* AV:go 2

6 Mary hates my teasing her.
(time taken: 1-035 seconds nodes used: 29)

1 SE:STAT TE:.. 1
2 SU:Mary AV:hates OB:GER 2
3 SU:my AV:teasing OB:her 3

7 the boy who kissed the girl laughed uproariously.
(time taken: 1-281 seconds nodes used: 36)

1 SE:STAT TE:.. 1
2 SU:CNP AV:laughed MO:uproariously 2
3 DE:the HE:boy AT:REL 3
4 SU:who AV:kissed OB:CNP 4
5 DE:the HE:girl 5

8 the boy who the girl kissed laughed uproariously.
(time taken: 1-294 seconds nodes used: 38)

1 SE:STAT TE:.. 1
2 SU:CNP AV:laughed MO:uproariously 2
3 DE:the HE:boy AT:REL 3
4 OO:who SU:CNP AV:kissed OB:* 4
5 DE:the HE:girl 5

9 the boy the girl kissed laughed uproariously.
(time taken: 1.176 seconds nodes used: 37)

1 SE:STAT	TE:..	1
2 SU:CNP		2
3 DE:the HE:boy AT:REL		3
4 OO:* SU:CNP	AV:kissed	4
5 DE:the HE:girl	OB:*	5
	AV:laughed	
	MO:uproariously	

10 Fred gave the dog biscuits.
(time taken: 0.928 seconds nodes used: 23)

1 SE:STAT	TE:..	1
2 SU:Fred AV:gave OB:CNP		2
3 DE:the AT:dog HE:biscuits		3

1 SE:STAT	TE:..	1
2 SU:Fred AV:gave IN:CNP OB:CNP		2
3 DE:the HE:dog HE:biscuits		3

11 Fred lost the dog biscuits.
(time taken: 0.863 seconds nodes used: 19)

1 SE:STAT	TE:..	1
2 SU:Fred AV:lost OB:CNP		2
3 DE:the AT:dog HE:biscuits		3

12 pick a ripe banana.
(time taken: 0.879 seconds nodes used: 26)

1 SE:IMP	TE:..	1
2 SU:* AV:pick OB:CNP		2
3 DE:a AT:ripe HE:banana		3

13 when did John say he would come?

(time taken: 1-294 seconds nodes used: 30)

W 1 SE:QUES TE: ? 1
2 OO:when OO:did SU:John AU:* AV:say MO:* OB:INDS 2
3 SU:he AU:would AV:come 3

1 SE:QUES TE: ? 1
2 OO:when OO:did SU:John AU:* AV:say OB:INDS 2
3 SU:he AU:would AV:come MO:* 3

14 he observed the man with the telescope.

(time taken: 1-061 seconds nodes used: 26)

1 SE:STAT TE: . 1
2 SU:he AV:observed OB:CNP 2
3 DE:the HE:man MO:PREC 3
4 LI:with PO:CNP 4
5 DE:the HE:telescope 5

305

1 SE:STAT TE: . 1
2 SU:he AV:observed OB:CNP MO:PREC 2
3 DE:the HE:man LI:with PO:CNP 3
4 DE:the HE:telescope 4

15 power corrupts.

(time taken: 0-724 seconds nodes used: 20)

1 SE:STAT TE: . 1
2 SU:CNP AV:corrupts 2
3 HE:power 3

1 SE:IMP TE.: 1
 2 SU:* AV:power OB:CNP 2
 3 HE:corrupts 3

16 he is waiting.
 (time taken: 0-535 seconds nodes used: 12)

1 SE:STAT TE.: 1
 2 SU:he AV:is CO:waiting 2

1 SE:STAT TE.: 1
 2 SU:he AU:is AV:waiting 2

17 playing cards intrigues me.
 (time taken: 1-045 seconds nodes used: 28)

1 SE:STAT TE.: 1
 2 SU:GER AV:intrigues OB:me 2
 3 SU:* AV:playing OB:CNP 3
 4 HE:cards 4

18 playing cards intrigue me.
 (time taken: 1-026 seconds nodes used: 29)

1 SE:STAT TE.: 1
 2 SU:CNP AV:intrigue OB:me 2
 3 AT:playing HE:cards 3

19 I dislike playing cards.
 (time taken: 0-951 seconds nodes used: 21)

1 SE:STAT TE.: 1
 2 SU:I AV:dislike OB:CNP 2
 3 AT:playing HE:cards 3

1 SE:STAT TE: 1
 2 SU:I AV:dislike OB:GER 2
 3 SU:* AV:playing OB:CNP 3
 4 HE:cards 4

20 the rascal who John claimed committed the crime has escaped.
 (time taken: 1-840 seconds nodes used: 44)

1 SE:STAT
 2 SU:CNP
 3 DE:the HE:rascal AT:REL
 4 OO:who SU:John AV:claimed OB:INDS
 5 SU:* AV:committed OB:CNP
 6 DE:the HE:crime
 TE: 1
 AU:has AV:escaped 2
 3
 4
 5
 6

21 whose book did you say you wanted?
 (time taken: 1-537 seconds nodes used: 36)

1 SE:QUES TE: 1
 2 OO:CNP OO:did SU:you AU:* AV:say OB:INDS 2
 3 DE:whose HE:book SU:you AV:wanted OB:* 3

22 when he has fixed dates he will ring us.
 (time taken: 2-285 seconds nodes used: 60)

1 SE:STAT TE: 1
 2 MO:SUBC
 3 LI:when SU:he AV:has OB:CNP
 4 AT:fixed HE:dates
 TE: 2
 SU:he AU:will AV:ring OB:us 3
 4
 1 SE:STAT TE: 1
 2 MO:SUBC
 3 LI:when SU:he AU:has AV:fixed OB:CNP
 4 HE:dates
 TE: 2
 SU:he AU:will AV:ring OB:us 3
 4

23 the queen's sister's husband took good photographs.
(time taken: 1-521 seconds nodes used: 45)

1	SE:STAT				TE:.	1
2	SU:CNP					2
3	DE:CNP	HE:husband	AV:took	OB:CNP		3
4	DE:CNP	HE:sister's	AT:good	HE:photographs		4
5	DE:the	HE:queen's				5

24 a lawyer who cheats the clients he sees deserves censure.
(time taken: 1-863 seconds nodes used: 47)

1	SE:STAT				TE:.	1
2	SU:CNP					2
3	DE:a	HE:lawyer	AT:REL		AV:deserves	OB:CNP
4		SU:who	AV:cheats	OB:CNP	HE:censure	
5			DE:the	HE:clients		
6				AT:REL		
				OO:*	SU:he	AV:sees
					OB:*	

25 has the portrait they bought disappeared?
(time taken: 1-001 seconds nodes used: 22)

1	SE:QUES				TE:?	1
2	OO:has	SU:CNP				2
3	DE:the	HE:portrait	AT:REL		AU:*	AV:disappeared
4		OO:*	SU:they	AV:bought		3
				OB:*		4

26 he rolled up the bright red carpet.
(time taken: 1-363 seconds nodes used: 42)

1	SE:STAT				TE:.	1
2	SU:he	AV:rolled	MO:PREC			2
3		LI:up	PO:CNP			3
4		DE:the	AT:bright	AT:red	HE:carpet	4

1 SE:STAT TE:.. 1
 2 SU:he AV:rolled PA:up OB:CNP 2
 3 DE:the AT:bright AT:red HE:carpet 3

27 she handed John a pear and Mary an apple.
 (time taken: 1-322 seconds nodes used: 34)

1 SE:STAT TE:.. 1
 2 SU:she AV:handed (IN:John OB:CNP and IN:Mary OB:CNP) 2
 3 DE:a HE:pear DE:an HE:apple 3

28 the plants he watered and tended flourished.
 (time taken: 1-480 seconds nodes used: 45)

1 SE:STAT TE:.. 1
 2 SU:CNP AV:flourished 2
 3 DE:the HE:plants AT:REL 3
 4 OO:* SU:he (AV:watered and AV:tended) OB:* 4

1 SE:STAT TE:.. 1
 2 SU:CNP AV:flourished 2
 3 DE:the HE:plants AT:REL 3
 4 OO:* SU:he (AV:watered OB:* and AV:tended OB:*) 4

29 are the elephant and the kangaroo he adopted obeying him?
 (time taken: 1-661 seconds nodes used: 49)

1 SE:QUES TE:?? 1
 2 OO:are SU:CNP AU:* AV:obeying OB:him 2
 3 (DE:the HE:elephant and DE:the HE:kangaroo) AT:REL 3
 4 OO:* SU:he AV:adopted OB:* 4

1 SE:QUES TE:?? 1
 2 OO:are SU:CNP AU:* AV:obeying OB:him 2
 3 (DE:the HE:elephant and DE:the HE:kangaroo AT:REL) 3
 4 OO:* SU:he AV:adopted OB:* 4

The Adaptive Memorization of Sequences

H. C. Longuet-Higgins and
A. Ortony¹

Department of Machine Intelligence and Perception
University of Edinburgh

1. INTRODUCTION

One of the most remarkable and useful mental faculties of human beings is their ability to memorize, recognize and recall sequences. Familiar examples are telephone numbers, the words of a poem, the letters of an alphabet, and even such complicated things as the holds on a rock climb or the notes of a piano concerto. Our memories can accommodate vast numbers of such sequences, and retrieve them with astonishing speed and accuracy. How would a systems analyst set about designing such a store?

The present paper describes an algorithm for recording a large number of data sequences and retrieving any one of them in a minimal number of elementary steps. The algorithm is specially designed to cope with non-random sequences, in which the probability of occurrence of a particular member is strongly dependent on the members which have already occurred. Most natural information sources generate sequences with this property; the notes of folk tunes and the words of sentences are cases in point. In testing our algorithm we have actually applied it to the individual words of free English and French text, regarding each word as a non-random sequence of letters. No attempt has been made to memorize actual word sequences, but the algorithm is designed to exploit the non-randomness of the word choice in developing and maintaining an efficient search for the successive letters of individual words.

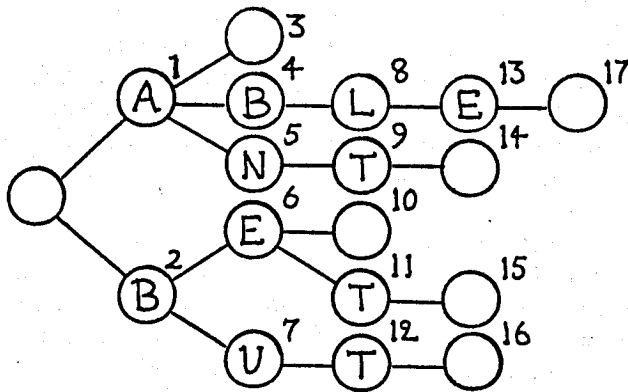
There is one very simple—in principle—way of storing words (from now on we shall speak of words regarded as letter sequences, though most of our remarks will apply equally to other types of sequence), and that is to compile a dictionary. The words of a dictionary are conventionally stored in lexicographical order, that is to say, if the letter s_n occurs after the letter s_n' in the

¹Now with Department of Philosophy, Fourah Bay College, University of Sierra Leone.

alphabet, then the word $(s_1s_2 \dots s_{n-1}s_n \dots)$ will occur after the word $(s_1s_2 \dots s_{n-1}s'_n \dots)$ in the dictionary. Bearing in mind the additional rule that $(s_1s_2 \dots s_{n-1})$ takes precedence over $(s_1s_2 \dots s_{n-1} \dots)$, we may represent the dictionary as a logical tree, in which each node holds a letter of the alphabet, except that each terminal node is left blank, indicating the completion of a word. The tree is of course ordered: the nodes following a given node are arranged in alphabetical order with the understanding that a blank takes precedence over all letters. For example, the dictionary

A
ABLE
ANT
BE
BET
BUT

will be represented by the tree



A dictionary is, however, useless without a search routine, and this routine should serve, as far as possible, both for identifying words that are already in the dictionary and for storing those that are not. Human beings, of course, use various tricks for locating words in dictionaries (and in locating the place where a new word should be entered). But there is one simple and infallible way of searching, which does not even require a knowledge of alphabetical order—and it is highly questionable whether alphabetical order plays any part in human word recognition. This is to search in turn for the node carrying the first letter, the node carrying the second letter, and so on. For example, suppose that the current state of the dictionary is as shown above, and that the word ANTS is encountered. The primary nodes (1 and 2) are examined

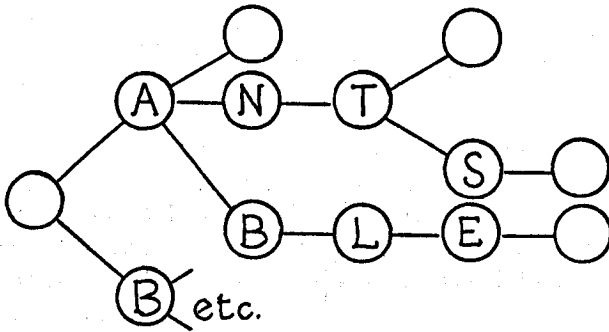
in turn for the letter A, and the examination succeeds at the first attempt. So nodes 3, 4, and 5 are inspected for the letter N; this letter is found at node 5. Nodes 8 and 9 are then examined in turn, and T is found at node 9. The only successor of node 9 is node 14, and examination of this node fails to reveal s, so a new node (18, say) has to be created to carry this letter. In an ordinary dictionary node 18 would have to be positioned below node 14, but this need not concern us at the moment. We are now at the end of the word ANTS, so one more node (19) must be added, and left blank.

Having stored ANTS we can now use much the same routine for retrieving it when it next occurs. More briefly: A is found at 1, N is found not at 3 or 4 but at 5, T is found not at 8 but at 9, s is found not at 14 but at 18, and a blank is found at 19, indicating that the word ANTS is indeed present in the dictionary. So storage and retrieval proceed along very much the same lines, as they should in a well-designed memory; one does not want to have to complete the process of deciding whether a word is present or not before starting on the process of storing it.

So far we have done little more than formalize the process of dictionary search, but we are now in a position to appreciate one point more clearly: there is nothing sacred about alphabetical order. The routine just described would work just as surely if the order of the nodes emerging from a given node were entirely arbitrary, and even if it were altered from time to time. This fact opens the possibility of speeding up the processes of storage and retrieval by adjusting the node orders according to the statistics of the text under study. Indeed, if the statistical characteristics of the text change systematically, and if the process of word retrieval is to remain efficient under such a change, the node orders in the tree must be susceptible to alteration on every occasion when a word is stored or retrieved.

A fanciful way of seeing what sort of changes to make in the node orders is to imagine a guessing game between two players called Memory and Senses. Memory looks after the tree, and Senses reads each word letter by letter. Suppose the word under study is ANT. Memory cannot read it himself, so he looks at his tree and asks Senses 'Is the first letter an A?'. The answer is 'Yes'. Next question: 'Is that the end of the word?' (Memory is at node 3.) Answer: 'No'. 'Then is the next letter a B?' Answer: 'No'. 'Then is it an N?' Answer: 'Yes'. At this point we interrupt the game to consider Memory's reactions. 'I got the first letter right first time,' he thinks, 'but it took me three guesses to get the second. I would have done better, when trying to guess the second letter, to ask whether it was an N before asking whether it was a B. I must remember to reverse the order of those questions next time.'

What this amounts to is interchanging the branches A—B and A—N, so that the tree now looks like this:



This kind of flexibility in the tree will clearly pay off in the long run, because by interchanging neighbouring branches one will tend to promote the commonest word continuations to positions of high precedence. Conversely, if a once common word continuation becomes rare, it will gradually become demoted in the course of time, so that the search routine adapts itself to secular changes in the word frequencies of a long piece of text.

We will now describe how we have used ALGOL for programming the flexible tree, and then report some results that we have obtained with it.

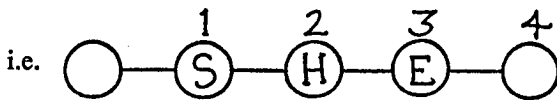
2. AN ALGOL SPECIFICATION OF THE TREE

At first sight ALGOL might seem a rather unsuitable language for specifying the growth and development of the flexible tree, because in ALGOL all arrays must be specified in advance, and we do not know in advance where the tree is going to sprout new branches. This difficulty may, however, be quite easily overcome by associating with each node not only a letter (or a blank) but also a pair of 'forwarding addresses'. The first of these is to be visited if the required letter is found at the node in question; the second is to be visited if it is not found there. If the symbol at a node is a blank, there is obviously no need for the former forwarding address, because if a blank is the required symbol the search will have been completed and the word will have been recognized. As to the second forwarding address, this may or may not be filled in. If it is not, and the required symbol is not found at the node, a new node or set of nodes must be created in order to accommodate the rest of the word, which by this time will be known to be unfamiliar. The creation of a new node calls for a new triplet of locations—to hold the letter or blank and the two forwarding addresses—and these may be supplied from a 'reserve' array with 3 rows and N columns, where N is bounded by the available storage capacity.

Before explaining how to program the interchange of two branches let us illustrate the above remarks by seeing how the program acts when given the

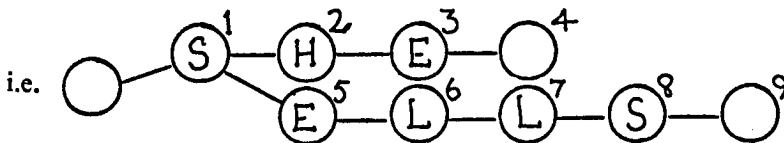
first two words of the tongue-twister 'She sells sea shells on the sea shore'. The reserve array is initially empty. The first letter of SHE (or, rather, its numerical equivalent) is entered in the first column, at the top; since this symbol is not a blank another column will be needed for the next symbol, and its address must be entered in the second place in the first column. The next available column in the reserve array is number 2, so a 2 is entered in the second place in the first column, and the letter H is entered in the first place in the second column. Similarly, a 3 is entered in the second place of the second column, and an E is entered in the first place in the third column. The next symbol is a space (or other mark of punctuation) so column 4, which is needed to hold this information, receives a blank as its first entry, and its second and third places are left unfilled. (In what follows we use a 0 to represent an unfilled place in the array.) The result is:

address	1	2	3	4
first entry	S	H	E	-
second entry	2	3	4	0
third entry	0	0	0	0



Now for the word SELLS. The first letter-s-is looked for at the only existing primary node-1-and is found there. The second letter is looked for at node 2, but is not found there. Since column 2 has no second forwarding address, a new column must be called up from the reserve array, and the next available is column 5. The letter E is therefore entered at the top of column 5, and columns 6 to 9 are called into play to hold the rest of the word. But the number 5 must be entered at the bottom of column 2, to indicate that this is the address to be visited in future if, after s, the required letter is not found in column number 2. Result:

address	1	2	3	4	5	6	7	8	9
first entry	S	H	E	-	E	L	L	S	-
second entry	2	3	4	0	6	7	8	9	0
third entry	0	5	0	0	0	0	0	0	0

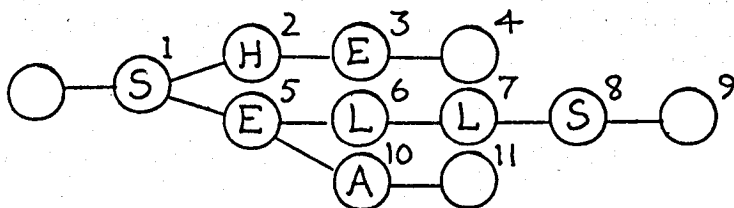


COGNITIVE PROCESSES: METHODS AND MODELS

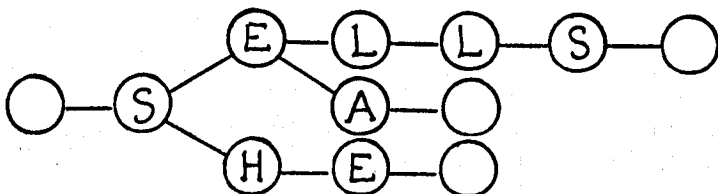
(Note that although the E of SELLS is connected to the initial s directly in the tree, in the array the connection is indirect; this is because the E is an *alternative* to the H, not a possible *successor* to it.)

We now come to the third word, SEA. If we carried on just as before the end result would be like this:

address	1	2	3	4	5	6	7	8	9	10	11
first entry	S	H	E	-	E	L	L	S	-	A	-
second entry	2	3	4	0	6	7	8	9	0	11	0
third entry	0	5	0	0	0	10	0	0	0	0	0



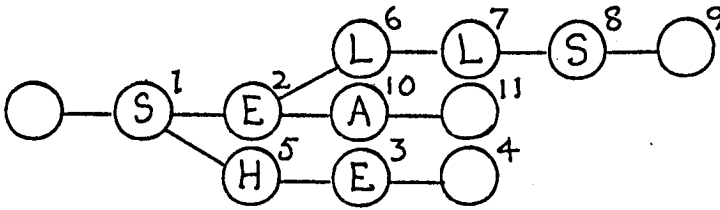
But we want to be able, now that SE has occurred for a second time, to give this word-beginning precedence over SH, which was the beginning of the first word. We want, in fact, to interchange two branches of the tree so that it looks like this:



This result is achieved by the simple expedient of interchanging the first entries and the second entries, in columns 2 and 5, which are the two nodes whose relative positions on the tree we wish to exchange. The array becomes:

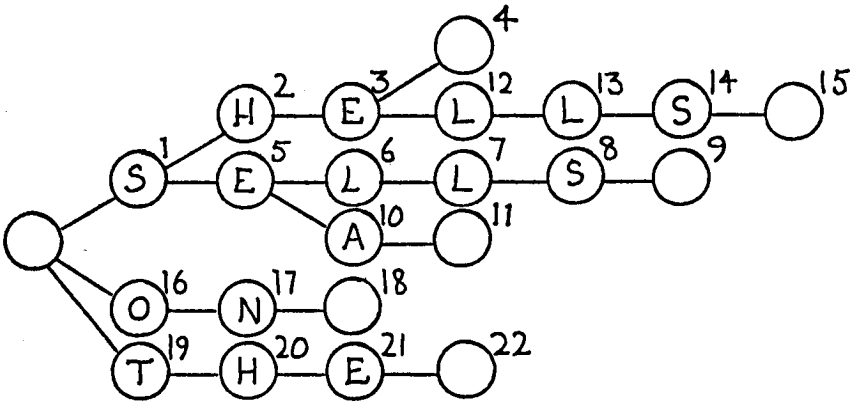
address	1	2	3	4	5	6	7	8	9	10	11
first entry	S	E	E	-	H	L	L	S	-	A	-
second entry	2	6	4	0	3	7	8	9	0	11	0
third entry	0	5	0	0	0	10	0	0	0	0	0

If we draw out the tree which is implied by this array—



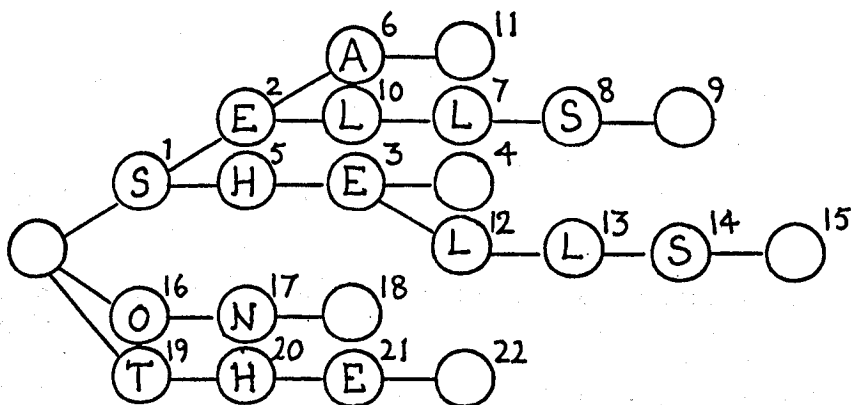
— we discover that not only have the nodes in question been interchanged, but also (as we would like) the branches which are attached to them. The simplicity of this operation makes it possible to save some space; if the first and second entries in each column are amalgamated to form a single number (from which the components can be extracted without difficulty), the interchange of two branches involves merely the swapping of two of these numbers, and instead of a 3 by N array we can make do with a 2 by N array. But this is a detail of implementation rather than of logic.

Let us follow the growth of the tree just a little further, to show what happens when the word SEA occurs again. As the reader can verify, after the words SHE SELLS SEA SHELLS ON THE the tree will look like this:



Now comes the word SEA. The first letter is found on the first guess (at node 1). The second letter is not found at node 2, but at node 5, so the entries at these nodes must be interchanged in the manner already described. The third letter is looked for at node 6 but is not found there; it is found, however, at node 10, so the entries at nodes 6 and 10 must also be interchanged. The next symbol, a

blank, is looked for at node 11, and there it is, so the word has been recognized. After the two interchanges the tree has this appearance:



Plainly the word SEA is the one which can now be recognized with the fewest wrong guesses (actually none); this shows the adaptive nature of the program.

Before reporting our results on natural language storage and retrieval, it will be useful to explain how one may conveniently assess the performance of the program. In retrieving a word which is already stored in the tree one will, in general, make a certain number of wrong guesses; the number of wrong guesses per word retrieved would therefore be a quantity indicating the efficiency of retrieval. But in reading a text one is constantly coming across new words, and we also need a measure of the amount of labour taken to store one of these. We have employed, in compiling our statistics, the number of wrong guesses needed to identify the 'stem' of the word, using this term to signify that part (if any) of the word which is already present on the tree. Thus the stem of the next (new) word SHORE is simply SH; thereafter the letters ORE cannot be located by guessing, they have to be stored anew. The point of this measure is that if the tree were *not* flexible the number of wrong guesses made in storing the stem of a word would be equal to the number of wrong guesses made in retrieving it on a second reading of the same text (for the first word, of course, this number is zero). With a flexible tree the two numbers are not in general equal, but the difference has been found to be slight, and so we shall without further apology define the 'oddity' of a word as the number of wrong guesses made in *either* retrieving it *or* storing its stem if it is a new word.

3. RESULTS WITH NATURAL LANGUAGE

Our first experiments were carried out on a highly non-random text, namely the Basic English Dictionary. The words were read in dictionary order, and the average oddity of the words was computed at the same time. To estimate the effect of interchanging the branches of the tree—the characteristically

adaptive feature of the program—the dictionary was read twice, once with interchange and once without. Here are the results:

	<i>flexible tree</i>	<i>inflexible tree</i>
first 50 words	7.70	8.50
next 50 words	4.46	7.38
100	4.77	9.15
300	5.86	13.07
362	8.79	25.12

The words of Basic English are almost exactly 5 letters long on the (un-weighted) average, so the average number of wrong guesses per symbol (there being 6 symbols per word) is about 1.4 with interchange and about 4.1 without when the last 362 words are being read. One can see why storage without interchange gets progressively more inefficient; one wastes a large number of guesses on the earlier letters of the alphabet when in fact the initial letter of a late word must lie near the end of the alphabet. No significance should be attached to the slight drop in oddity between the first 50 and the next 50 words of the dictionary; this is an accident of the distribution of word lengths.

Next we decided to give the program a field trial on literary English—or rather American—namely some of Damon Runyon's short stories. Runyon makes frequent use of the apostrophe, which we decided to treat as a word terminator. For instance, DIDN'T was treated as two words, DIDN and T; with this convention Runyon's words are only 4 letters long on the average. Here is a table giving the average oddity of successive groups of words in *Guys and Dolls*:

	<i>flexible</i>	<i>inflexible</i>
first word	0.00	0.00
next 3 words	2.00	2.00
5	5.00	5.00
10	6.20	6.90
30	6.57	7.17
50	6.92	7.28
100	8.62	9.35
300	8.81	9.57
500	8.61	9.28
1000	8.68	9.67
3000	8.53	10.72

(In passing, we may mention that after the reading of these 4999 words 913 words were present in the dictionary, and the number of columns of the reserve array taken to store these words and their terminators was 3647—almost exactly 4 columns per word. To fill our reserve array, which had 8000

columns, we needed to read about 16,000 words of Runyon, at the end of which process there were 1998 words occupying 7994 columns.)

Various points about this table are worth noting. First, the effect of interchange is much less than in the case of the Basic English Dictionary; this is because the early part of the story is a fair statistical sample of the whole, so that even without continuous adjustment the tree works quite efficiently, provided (as in this case) the order of precedence at each node is set by the text itself rather than imposed arbitrarily from outside. Secondly, it is remarkable how rapidly the average oddity of the words attains a plateau value and stays there, particularly when interchange is in force; the average oddity of the last 3000 words is almost exactly the same as that of the first 200. This encourages one to think that the program will be able to handle really large dictionaries without serious loss of efficiency.

As just remarked, the interchange routine, though effecting a noticeable improvement, is less important in this experiment than it would be if the tree were 'primed' with a rather different sort of text from the story itself. This contention is borne out by the results of our next experiments, in which we primed the tree with the Basic English Dictionary (B.E.D.), not allowing interchange of branches when the dictionary was being read in. The average oddity of the words in the B.E.D. was 17.08 under this restriction. Having thus primed the tree we then read in the first 3850 words of *Guys and Dolls* (G. & D.); in the first run interchange was called for, and in the second run it was ruled out. Here are the results:

	<i>flexible</i>	<i>inflexible</i>
first 50 words	15.14	17.12
next 100	12.49	16.26
200	11.41	17.86
500	9.79	16.81
1000	9.38	16.91
2000	8.98	17.13

The results are striking, and show that after about 1000 words of G. & D. the flexible tree had almost completely recovered from the malign influence of the alphabetical order of the B.E.D. The inflexible tree, however, was still suffering the ill effects of its early training, with no sign of recovery.

These experiments employing the B.E.D. have a certain artificiality, so we thought it worth while to test the flexible tree on literary text from two different sources, namely Runyon and Proust. The average oddity of 1000 words of each text, presented to the flexible and the inflexible trees, was as follows:

	<i>flexible</i>	<i>inflexible</i>
Runyon (English)	8.45	9.13
Proust (French)	9.40	11.21

(Proust's words are longer, on the average, than Runyon's.) Having determined these overall properties of the two texts, we then proceeded to read

them in successively, first with the interchange facility and secondly without. The results were:

	<i>flexible</i>	<i>inflexible</i>
1000 English words	8.55	9.1
then 1000 French words	11.0	15.5
1000 French words	9.4	11.2
then 1000 English words	10.3	15.0

Again we note an improvement, of about 5 units of oddity, in the ability of the tree to adapt itself to a new language when interchange of the branches is called into play.

We feel that these results show beyond doubt the gain in efficiency and adaptability resulting from the interchange routine. We must add that in our thinking about how to construct this adaptive program we owe much to the work of Samuel, whose checkers program (*see* Samuel, 1959) uses an idea of much the same kind for referring to previously encountered board positions.

Before ending this section we would like to make some remarks about the information-theoretical aspects of the flexible tree. The average oddity of a Runyon word, presented to the flexible tree, is about 8.6. The average word length is 5.0 symbols (4.0 letters), so the average number of answers (e.g. 'Yes, the next letter is an A', or 'No, the next letter is not an A') needed to retrieve a word is 13.6. The answer to a yes-or-no question provides not more than one bit of information on the average, so the average number of bits per symbol of text cannot exceed $13.6/5.0 = 2.7$. A random sequence of letters and spaces would require $\log_2(26+1) = 4.8$ bits per symbol to specify it. The ratio $2.7/4.8$ is thus an upper limit to the redundancy of English prose, regarded as a stochastic sequence of letters and terminators, and this ratio is about 0.6. In the classic book *The Mathematical Theory of Communication*, Shannon (1949) estimates the redundancy of English as 0.5. The closeness of this figure to the upper limit 0.6 estimated from the flexible tree indicates that the tree is working close to the theoretical optimum—a most surprising result if one bears in mind that questions of the form 'Is the next letter a vowel?' are not available in the program.

4. A PRACTICAL APPLICATION—THE CAT

In the course of developing the flexible tree we realized that it could be used as the basis for a potentially useful invention, the Computer-Assisted Typewriter, or CAT. The CAT is based on the idea that when one is typing someone looking over one's shoulder can often predict what letter one will type next, given the first letter or letters of a word. (He can also often predict what word one will type next, but this process is beyond our present programming capabilities.) Prediction is particularly reliable when one is working within a small vocabulary, because the number of possible continuations of a given letter sequence is then more restricted. There are various ways in which the

CAT can operate. One may feed it with a dictionary before starting to type, and then get it to intervene and type the remaining letters of any word which one has begun to type and which has a unique continuation; or one may let it build up a vocabulary as one types, or both. Since, very often, one will want to use a word which the CAT has not yet met, it is useful to be able to cancel the CAT's guesses. This is done by typing a stroke (/) followed by a space, after which the CAT types back the letters one originally typed before it interrupted. One will also make mistakes oneself, and these can be cancelled (back to the beginning of the current word) by typing two strokes in succession. The resulting text will tend to be rather messy in appearance, so the CAT is also programmed to keep a record of what was not cancelled, and to type this out on demand in fair copy. At the present stage of development the CAT takes no account of punctuation—it identifies letters but treats all punctuation marks as spaces—so it is not yet a commercial proposition. But with some refinement it could be of real assistance to someone working within a small vocabulary such as that of a programming language; such words as 'BEGIN' or 'PROCEDURE' or identifiers introduced by the programmer, would be automatically completed by the CAT after the programmer had typed only one or two symbols.

Here is an actual collaboration with the CAT, carried out on the console typewriter; the capitals are typed by the CAT, the lower case letters by the operator, who also signals the end of his text by typing the symbols ++:

tomorrow and TOMORROW AND TOMORROW creeps in TOMORROW/ This
petty PETTY/ Pace from day TOMORROW/ TO DAY till this/ THE last
syllable of recordef// recorded tiLL/ TIME ++
PLEASE TYPE I FOR A FAIR COPY . . . I
TOMORROW AND TOMORROW AND TOMORROW CREEPS IN THIS
PETTY PACE FROM DAY TO DAY TILL THE LAST SYLLABLE OF
RECORDED TIME

The reader will doubtless think of many ways in which this idea could be refined and extended; we are working on some of these at the moment.

Acknowledgment

We would like to thank Professor Donald Michie for much valuable encouragement and comments in the course of this work.

REFERENCES

- Samuel, A. L. (1959), Some studies in machine learning using the game of checkers. *I.B.M. J. Res. Devl.*, 3, 211–29. Reprinted in Feigenbaum and Feldman (1963).
Shannon, C.E. & Weaver, W. (1949), *The mathematical theory of communication*, p. 56. Illinois: Illinois University Press.

PATTERN RECOGNITION

An Application of Graph Theory in Pattern Recognition

Judith Hilditch¹

Medical Research Council
Clinical and Population Cytogenetics
Research Unit, Edinburgh

INTRODUCTION

Euler's discussion of the Königsberg bridge problem can be said to have been the beginning of graph theory as a mathematical discipline. This now famous article dealt with an essentially graphical problem, little imagination being required to consider the various places to be visited as points, these points being connected by lines representing the bridges. Indeed the problem can be drawn out on paper as points connected by lines without loss or addition of relevant information. It was not however until a century later, when it was realized that electrical networks and the structures of molecules and crystals could be represented by a graphical structure of essentially similar kind that there was a resurgence of interest in graph theory. At about the same time graph theory escaped from its pictorial origins with the study of binary relations as graphs, the representation of a graph as a drawing of points interconnected by lines becoming merely a visual aid rather than an essential part of the problem. This has remained so for the ever widening range of subjects to which graph theory has since been found applicable—from game theory and programming through switching circuits and communications theory, consumer behaviour and psychology, to problems from biology.

However, recent suggestions that graph theory might prove useful in pattern recognition (Narasimhan, 1964; Philbrick, 1966), have, on this one small branch of the subject, elevated the picture of points and lines to the position of being not merely a visual aid, but the starting point of the problem itself. This means that the situation is in some ways more restricted than that generally dealt with by graph theory; for example we know that, by their very nature, the graphs with which we deal must be finite, planar and undirected. On the other hand we usually do not know in advance whether a graph is

¹ Present address: Derbyshire House, St Chad's Street, London W.C.1

connected, whether it is a tree, what the maximum degree of its nodes is, etc., and we certainly cannot apply to it the elegant theories constructed around such special systems as for example ordered rooted binary trees. Because of this, and also because we are rapidly taken outside the scope of graph theory by such considerations as the actual positions of points and the length and curvature of lines, it is in answering the more basic questions such as 'Is this graph connected?' that we have found the techniques of graph theory most useful.

FROM PICTURE TO ABSTRACT GRAPH

One definition of a graph as an abstract mathematical concept (based on that given by Ore (1962)) is the following:

There is a set N consisting of elements which shall be considered to be connected in some fashion. These elements are called nodes, and N the node set. A graph $G(N)$ is a family of associations or pairs

$$a = (m, n) \quad m, n \in N$$

which indicates which nodes shall be considered to be connected. Each pair (m, n) we call an arc.

With a picture that consists basically of line-like elements, as in bubble and spark chamber photographs, and in our own field of chromosome analysis (see figure 1) it is simple to see intuitively the structure of the picture as a graph—the line-like elements being the arcs, their ends and points of intersection the nodes. However, having obtained a digitization of such a picture (this is usually in the form of a rectangular raster or matrix of points, the value of each element of the matrix being related to the darkness of the corresponding part of the picture) a formalization of these intuitive ideas is necessary—in particular the idea of a line must be precisely defined. This is somewhat difficult in the usual case where width and length of lines are variable, and noise is an ever-present problem. A method of overcoming this difficulty is described by McCormick (1963) and Narasimhan (1964) for use, in particular, in bubble chamber work. Their method consists of thinning and gap filling the line-like elements of a picture to reduce distortion due to variations in width, and gaps and holes in the lines. After the line-like elements have been standardized in this way they are recognized as being lines by the ratio of their length to their breadth. At about the same time Blum (1964) proposed a transformation for applying to patterns, which he called the 'Medial Axis Transform'. This produces a line drawing from a plane figure of any shape. It can probably be most simply described by the grass fire analogy—if we consider the boundary of some shape drawn out on a uniform dry grass field, and then the grass at all points on the boundary set alight at the same moment, the front of the fire will move away from the boundary at a steady rate until at certain points different parts of the front will meet. The medial axis is the locus of points at which this occurs. This set of points, plus

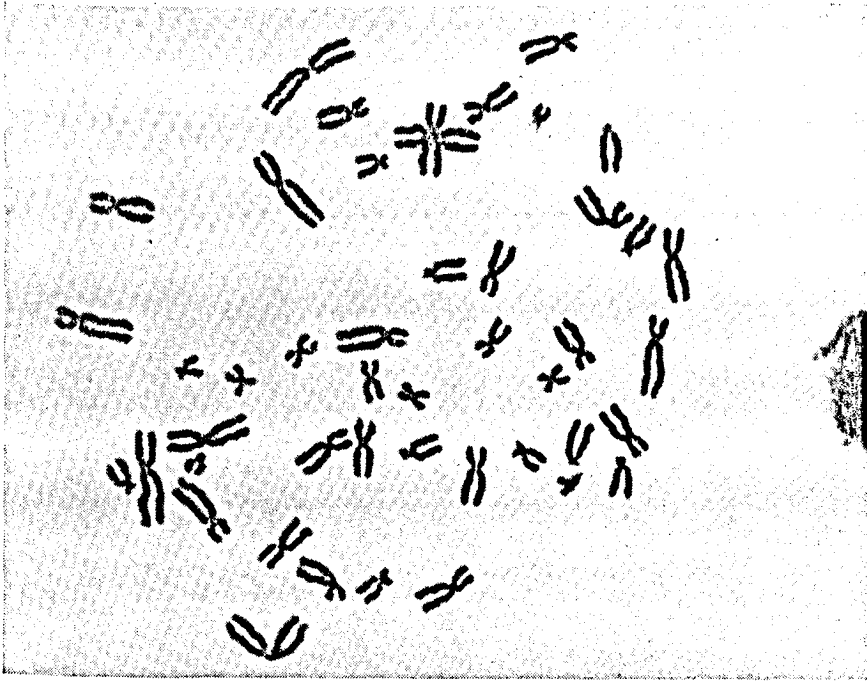


Figure 1. A metaphase spread. This picture is a photograph of a typical cell of the type which we are attempting to analyse ($\times 2900$)

a function giving for each point the time at which the fronts met, completely defines the original shape. This locus of points forms a type of 'skeleton' for the picture. Philbrick (1966) simulated the Medial Axis Transformation on a digital computer—the original pattern being digitized on a rectangular grid, and time units quantized so that the pattern boundary propagates a discrete distance of two units in each time unit, and after each step, points at which the fronts intersected are detected and listed as belonging to the medial axis. The method which we have implemented for producing skeletons of chromosomes is somewhat similar to this in that it depends on propagation of the boundary of the object from the edge inwards in discrete steps (in our case steps of one unit). However, we are dealing with pictures which already have a line-like structure (chromosomes) and this makes it possible for us to place much greater constraints on the form of the final skeleton.

THE SKELETON

I shall not describe in detail the method we use for producing the skeleton, but simply say that having obtained a digitization of the pattern on a rectangular raster, with points in the pattern given the value one and points outside the value zero, the boundary points of the pattern are removed layer by layer, and at each step all those points on the edge of the pattern whose removal

PATTERN RECOGNITION

would not alter connectivity and which do not lie at the tip of a line are deleted. (Connectivity is defined by considering each pattern point to be adjacent to its eight immediate neighbours as shown in figure 2, though it is of course possible to define connectivity in other ways.)

4	3	2
5	x	1
6	7	8

Figure 2. The neighbours of a point. With a square raster the point x may be considered to be adjacent to the eight points 1 to 8, or only to the points 1, 3, 5 and 7, (and of course there are other possibilities)

The resulting skeleton usually follows the main features of the original configuration (as would be hoped) but in addition, in order to make its treatment as a graph as simple as possible, we have tried to ensure that it is as close an approximation to a line drawing as is possible in a discrete space. To be more precise, the skeleton is such that the removal from it of any one point other than the tip of a line would alter its connectivity. An example is shown in figure 3. Each point of the skeleton has one, two or more neighbouring points also in the skeleton. Connected sets of points each having exactly two adjacent points in the skeleton we call arcs, connected sets of points each having more than two adjacent points in the skeleton are called junctions, and individual points with only one neighbour in the skeleton are tips. In general the arcs correspond to the line-like parts of the original pattern, the junctions to the places where these meet, and the tips to their ends.

To treat the skeleton as an abstract graph we consider each tip or junction to be a node, and this gives us the node set N , and, following the intuitive idea that two nodes should be connected if there is a line joining them, we define two nodes n_1 and n_2 to be connected if there exists an arc a such that a point of n_1 is adjacent to a point of a and a point of n_2 also is adjacent to a point of a .

LABELLING THE SKELETON

To produce this graph it is necessary to identify the nodes, and determine which of them are connected to each other. Having obtained the skeleton as a matrix of points having positive values if they lie on the skeleton and value

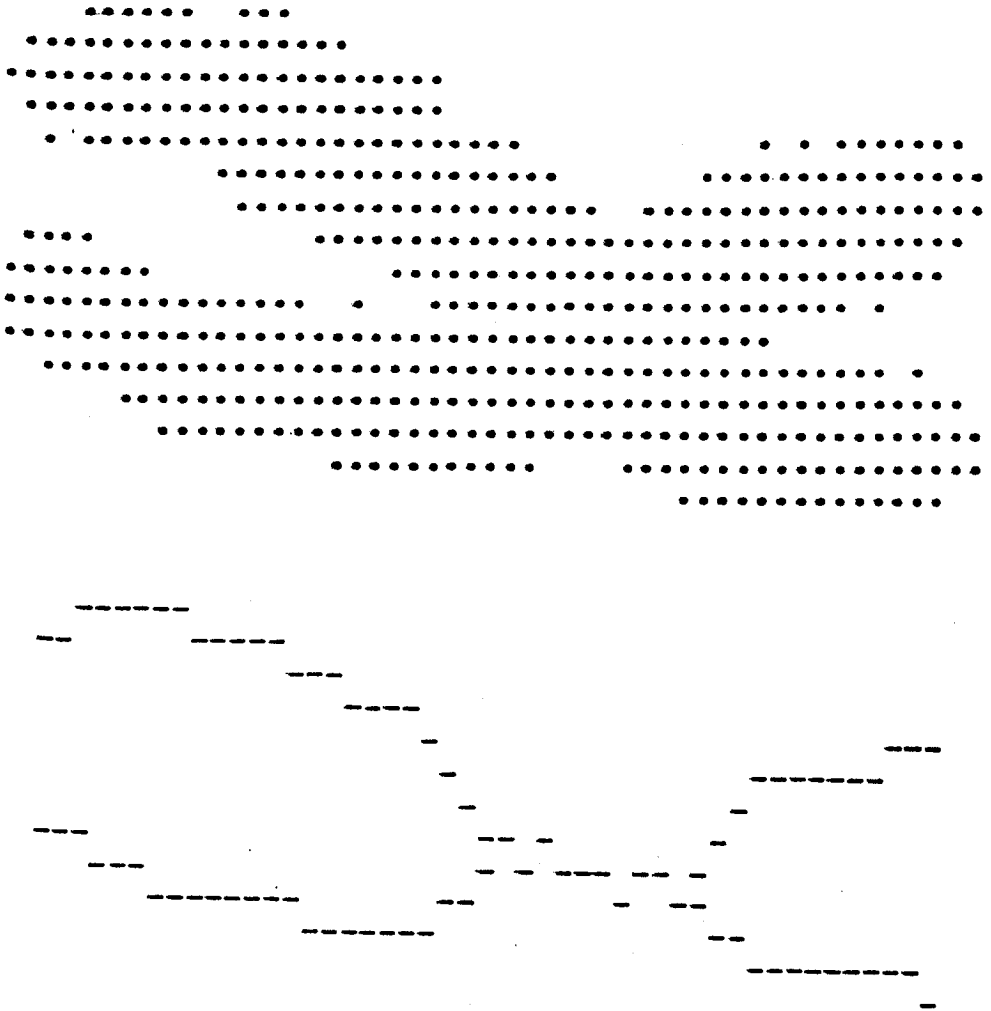


Figure 3. A line printer representation of a chromosome and its skeleton. Images are initially digitized on a scale of seven grey levels; however, before the skeleton can be produced certain noise filtering operations are applied to the image and it is then converted to binary form. It is this binary image from which the skeleton is obtained, and which is shown above

zero otherwise, the arcs and nodes are identified as follows: first the arcs are differentiated from the nodes by the simple procedure of setting negative all points with exactly two non-zero neighbours; the arcs and nodes are then labelled by assigning a unique positive value to the points of each node and a unique negative value to the points of each arc. This is done by applying an algorithm based on one described by Rosenfeld and Pfaltz (1965).

Suppose we are given a finite domain D , and a function f on D , and wish to transform f so that it takes a unique value on each connected subset of D on which it is greater than some lower limit LIM ; at all other points of D it is to be given a value $BKGRND \leq LIM$. This can be done by repeatedly applying the following algorithm at each point of D in turn:

Let $L := \max(f(D) + K)$ where K is some large number, then for each point x

if $f(x) \leq LIM$	$f(x) := BKGRND$
else	$f(x) := \max(L, \{f(n)\})$
and then if $f(x) = L$	$L := L - 1$

where $\{f(n)\}$ is the set of values of the neighbours of the point x —i.e. those points which are considered adjacent to x —and the point x itself.

In this way the value of x is set to the value of that one of its neighbours which has been given the label of highest value, or if none of them has been labelled it is given the label L and L is decremented to provide another new label when required. L must of course be given a sufficiently high initial value to remain greater than $\max\{f(D)\}$ after it has been decremented.

This algorithm can be used for various types of connectivity by suitably defining the neighbours of a point; for example, working with a square raster, we take the neighbours of point x to be the eight points 1 to 8 shown in figure 2. An alternative method would be to consider only the four points 1, 2, 5, and 7 as being neighbouring points of x . The algorithm is applied to all points of the domain repeatedly until a pass is made in which the value of none of the points is altered. It can be shown (see Rutovitz *et al.*, 1968) that at this stage each connected area of points has been given a unique label lying between the initial and final values of L .

This is made more efficient by applying the algorithm to the points in forward raster sequence (starting at the upper left corner and working down to the lower right) at the first pass, so that labels are propagated downwards and to the right, then in exactly the opposite order at the next pass so that they are propagated upwards and to the left. If the order in which the points are dealt with is reversed at each pass in this way the number of passes required to label a picture is dependent upon how sinuous the connected areas in the picture are. For example the area in figure 4 (a) would require only one pass to label it and a second pass to determine that labelling was complete, whereas the area in figure 4 (b) would require six or seven passes. The skeletons which we require to label consist of comparatively straight lines and so require only a few passes, and this method is not therefore too inefficient.

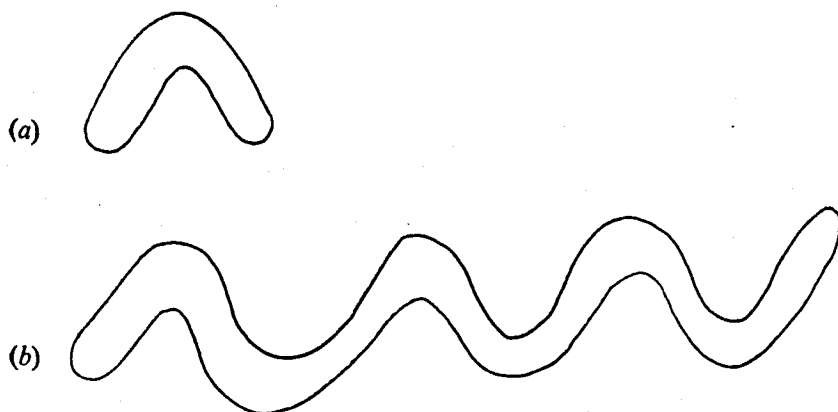


Figure 4. Two hypothetical components of a picture. Component (a) would require only one pass of the labelling algorithm to label all its points with the same value, whereas component (b) would require six or seven

The algorithm described above could equally well be inverted, i.e. points with value less than LIM being labelled, L being given an initial value much lower than LIM , $f(x)$ being set to the minimum of L and the values of its neighbours, and L being incremented instead of decremented. In order to label our skeletons and give a unique positive value to each of the nodes and a unique negative value to each of the arcs we apply both these approaches at once, labelling independently the connected areas of points with positive values (the nodes) and the connected areas of points with negative values (the arcs). The exact algorithm is as follows (using the same notation as before):

Points with value greater than $LIMHIIH$ are labelled with values starting at LH , where $LH \gg LIMHIIH \geq BKGRND$ and points with value less than $LIMLOW$ are labelled with values starting at LL ($LL < LIMLOW \leq BKGRND$)

if $LIMLOW \leq f(x) \leq LIMHIIH$ $f(x) := BKGRND$

if $f(x) > LIMHIIH$ $f(x) := \text{maximum}$
($LH, \{f(n)\}$)

and if $f(x) = LH$ $LH := LH - 1$

if $f(x) < LIMLOW$ $f(x) := \text{minimum}$
($LL, \{f(n)\}$)

and if $f(x) = LL$ $LL := LL + 1$

Figure 5 shows an example of a resulting labelled skeleton.

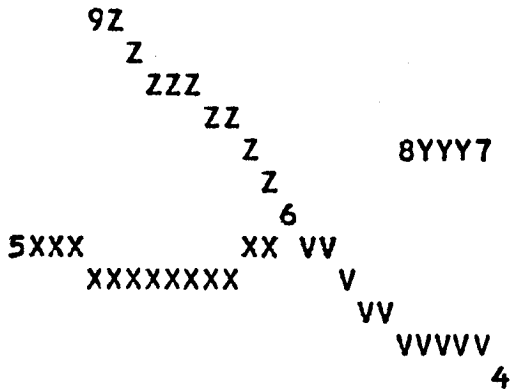
ADJACENCY MATRIX

In the abstract graph defined by the labelled skeleton an arc a connecting nodes n_1 and n_2 can be considered as the point (n_1, n_2) in the product space $N \times N$. As usual the elements in this product can be represented in a square array A with the elements of N serving as coordinates along the two axes. At the position with coordinates (n_1, n_2) one places the figures 1 or 0 depending on whether or not there is a corresponding arc in the graph. Thus one obtains a *node incidence matrix* which completely describes the graph if it has single arcs, i.e. only one arc joining any two nodes. If the graph has two or more arcs joining the same pair of nodes one can replace the symbols 0 or 1 at (n_1, n_2) by the multiplicity $\rho(n_1, n_2)$ —that is the number of arcs joining n_1 and n_2 . A graph may also be defined by an incidence matrix of another kind—this has rows corresponding to nodes, and columns corresponding to arcs, an element at (a, n) being non-zero if and only if the arc a terminates or begins at node n . This is called a *node-arc incidence matrix*.

Finally one can consider an *arc incidence matrix* which has both rows and columns corresponding to arcs, and elements are non-zero if and only if the two arcs have a common endpoint. For our purposes the node incidence matrix has been found the most convenient, with notation chosen such that the elements (n_1, n_1) corresponding to loops (arcs starting and ending at the same node) lie on the diagonal, and this we call an *adjacency matrix*. As our arcs are undirected (if an arc connects n_1 to n_2 it equally connects n_2 to n_1) the matrix is symmetric. In the case when the graph has no more than one arc connecting any two nodes, and thus elements take only the values 0 and 1, a variation of the adjacency matrix has been found useful. In this a non-zero element at (n_1, n_2) is given the value of the label assigned to the arc connecting nodes n_1 and n_2 . We call this matrix a *labelled adjacency matrix*; it is very similar to the matrix of a line drawing described by Hodes (1961). Although with graphs as sparse as those with which we are working the adjacency matrix representation is rather wasteful of storage space, our graphs are in general so small as to make this unimportant. Figure 5 shows an example of a labelled adjacency matrix for the skeleton of a chromosome.

CHROMOSOME ANALYSIS

At this stage I feel I should describe briefly what chromosome analysis involves. This is most clearly shown by an illustration. Figure 6 shows a normal chromosome spread, with its karyotype underneath. Karyotyping consists of grouping the chromosomes as shown, the grouping being done according to the length of the chromosomes, and the positions of their centromeres, i.e. the constrictions where the arms are joined (though the trained cytologist is in fact aided by certain other qualitative factors which we shall not concern ourselves with here). The normal human has 46 chromosomes falling into the groups shown, and abnormalities (inherited, or due to environmental damage) are recognized by the total number of chromosomes being incorrect,



ADJACENCY MATRIX

	9	8	7	6	5	4
9	.	.	.	Z	.	.
8	.	.	Y	.	.	.
7	.	Y	.	.	.	.
6	Z	.	.	.	X	V
5	.	.	.	X	.	.
4	.	.	.	V	.	.

Figure 5. Line printer representation of a labelled skeleton, and its labelled adjacency matrix. Points with positive value are represented by numbers, and points with negative value by the letters of the alphabet. Thus each different letter represents an arc and each different number a node

or certain groups having the wrong number of chromosomes, or chromosomes having abnormal shapes, e.g. ring chromosomes, dicentrics (chromosomes with two centromeres), fragments of chromosomes with no centromere, etc. At present this task is performed either by photographing the cell, cutting out the chromosomes and arranging them into their groups, or, as in our laboratory in Edinburgh, by eye, directly down the microscope, by trained technicians.

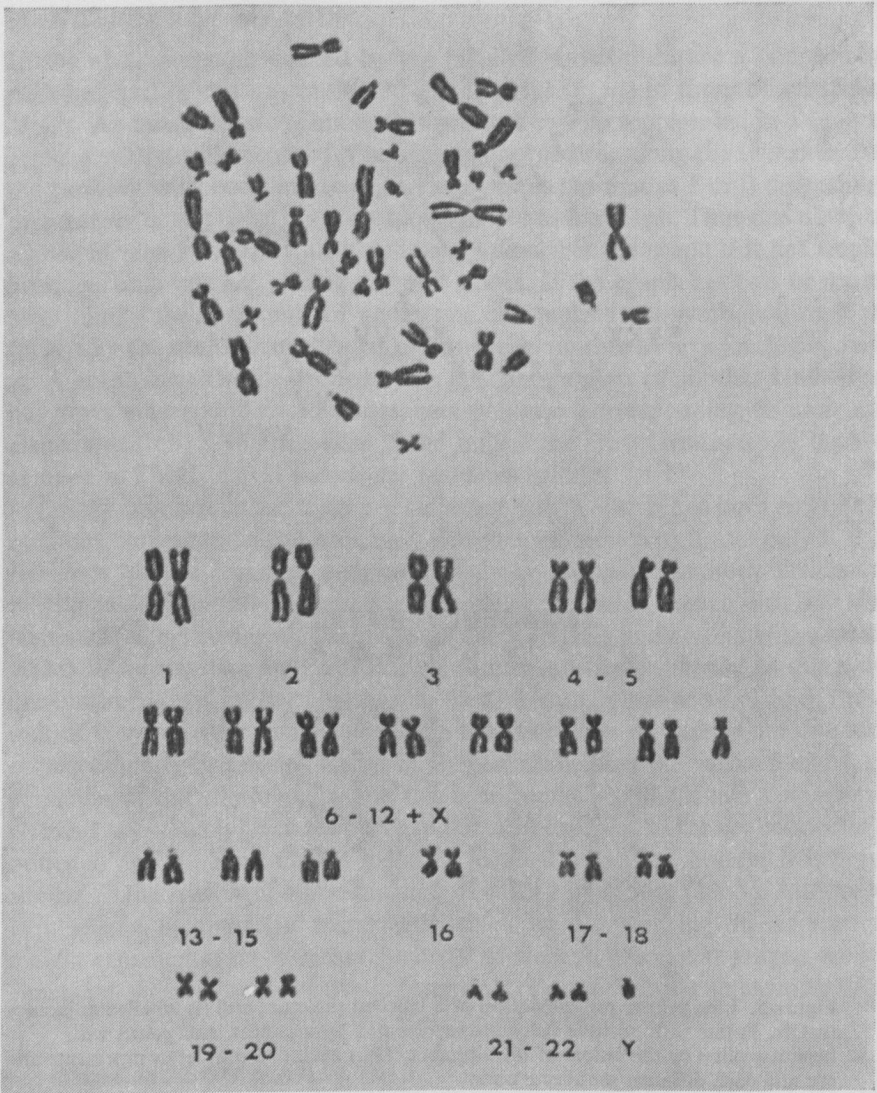


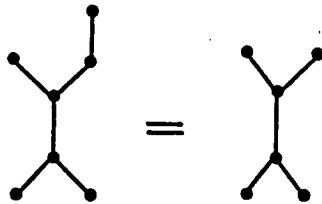
Figure 6. A photomicrograph of a human cell at mitosis, and the karyotype produced from it. The karyotype is obtained by grouping the chromosomes according to length and centromere position ($\times 1600$)

If, as in figure 6, the chromosomes were always well separated from each other the problem of automating their analysis would be comparatively simple; however, as in any biological system complexity is the norm, and the cell shown in figure 1 is more typical, with overlapping chromosomes, bent chromosomes, and chromosomes lying so close together that after digitization they will undoubtedly appear to be touching. Nevertheless, the majority of chromosomes are still of the more easily analysable type. Because of this we are working towards a system which involves a hierarchy of programming techniques, described more fully by Rutovitz (1967) and Hilditch and Rutovitz (1967). Within this system the simpler techniques are applied to the chromosomes first, with acceptance criteria to decide whether or not they are applicable. For those chromosomes which do not prove amenable to these techniques we go on to more complex and time expensive methods, such as that which I am describing, and it should be born in mind that the techniques described here are intended for application only to the less straightforward chromosome configurations.

Nevertheless we shall start by considering the simpler case:

THE CHROMOSOME GRAPHS

A normal, free standing chromosome which does not have its arms twisted over each other in any way should produce a skeleton which is a realization of one of the four graphs shown in figure 7(a); these we call 'chromosome graphs'. Figure 7 also shows digitizations of four chromosomes and their skeletons, each of which corresponds to one of these graphs. Conveniently enough the chromosome graphs happen to be four of the simplest graphs that there are. They are all connected; they are all trees, i.e. they have no loops or cycles. Moreover, if we ignore nodes of degree two, and consider the two arcs terminating at such a node to be one continuous arc, as illustrated below, then the chromosome graphs can be shown to be the only trees with four or less tips (except of course for the trivial case of the isolated node):



Having obtained a graph from a digitized object, our approach is to test whether or not it is a chromosome graph, and if not then to convert it to one or more chromosome graphs in some way.

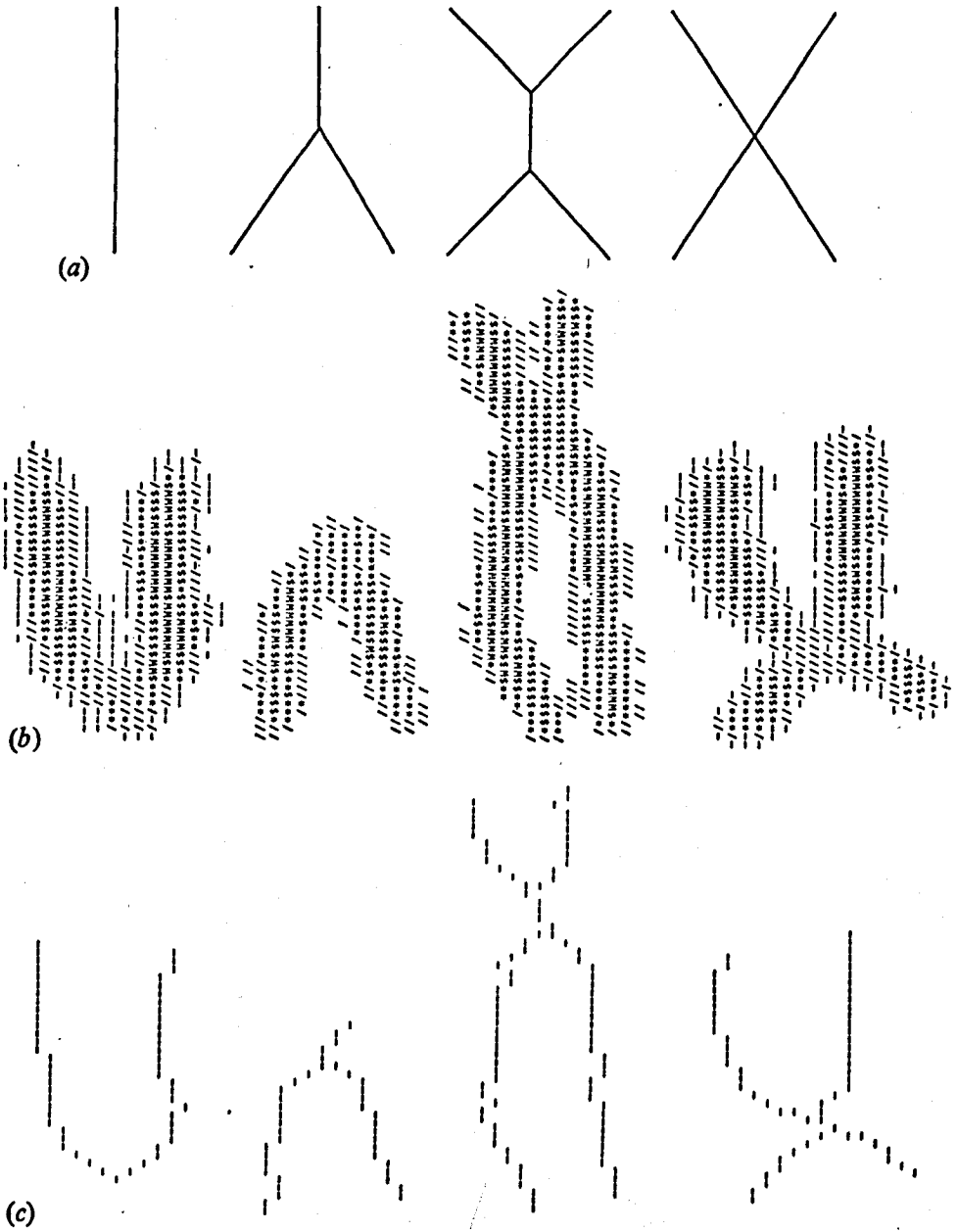


Figure 7. (a) The four chromosome graphs. (b) and (c) Four chromosomes, and their skeletons, each of which is an example of one of the chromosome graphs. The chromosomes are digitized on a seven level grey scale, and in the line printer representations shown here the characters are chosen so that the grey level at each point of the digitization corresponds approximately to the overall darkness of the printed character

OBTAINING A CONNECTED TREE

The first steps in doing this are to test whether the graph is connected, extract its components if it is not, and then determine if these components are trees.

The adjacency matrix can conveniently be used to determine whether the graph is connected, the rows and columns of the matrix being exchanged in such a way that the matrix can be factored into the direct sum of the adjacency matrixes of its disjoint components. This is done as follows: Let the node corresponding to row i (or column i) in the adjacency matrix be referred to as node i . The rows and columns of the matrix are exchanged so that the nodes adjacent to node 1 become nodes 2 to a_1 (say). Then further rows and columns are exchanged to make those nodes adjacent to node 2 and not already included in 1 to a_1 become nodes $a_1 + 1$ to a_2 , those adjacent to node 3 become $a_2 + 1$ to a_3 , and so on. Eventually a value i is reached such that nodes 1 to a_i consist of all those in the component of the graph which contains node 1. Subsequent operations do not therefore add any new nodes to this component. When $n = a_i$ we will find that all the nodes adjacent to node n have already been shifted to the left of column n , and this signals the completion of the component containing node 1, and the sub-matrix

$$A(k, j), 1 \leq k \leq n, 1 \leq j \leq n$$

is the adjacency matrix for this component of the graph. Continuing from node $n + 1$ will then produce the adjacency matrix for the next component, and so on, until the whole graph has been exhausted. If the original graph were connected then the first component obtained by this method would consist of the whole graph.

We now need to determine whether these components are trees, and, as it is possible to show that a connected graph is a tree if and only if the number of nodes in the graph is one more than the number of arcs, this is a very simple attribute to test. If the graph is not a tree then it is desirable to determine the position of its cycles so that these can be more closely examined to discover the cause, e.g. touching or overlapping chromosome arms. Cycles only a few arcs in length can be identified from the adjacency matrix. For example a loop or cycle of length 1, consisting of one arc starting and ending at the same node i is indicated by a non-zero element a_{ii} on the diagonal of the adjacency matrix, and an element a_{ij} with value greater than 1 indicates more than one arc connecting nodes i and j , and hence a cycle of length two passing through these nodes.

More generally if we define a *walk* as being a sequence of arcs

$$W = (n_0, n_1)(n_1, n_2)(n_2, n_3) \dots (n_{m-1}, n_m)$$

such that consecutive arcs always have a common endpoint (noting that the same arc may appear more than once in the sequence), it can be shown that if we consider the adjacency matrix raised to the power m

$$A^m = \{a_{ij}^{(m)}\}$$

PATTERN RECOGNITION

the element $a_{ij}^{(m)}$ gives the number of walks of length m between nodes i and j .

Thus for example if

$$a_{ij}^{(2)} \neq 0$$

then there exists at least one node k giving a walk

$$(i, k)(k, j)$$

so that if we also have

$$a_{ji} \neq 0$$

i.e. an arc joining i to j , then there must be a cycle of length three passing through nodes i , k and j unless $k=i$ or $k=j$ in which case there is a loop at either node i or node j .

This cannot however be generalized to find cycles of any length, since a walk of length m between nodes i and j implies the existence of a walk of length $m+2$ between them, namely that obtained by backtracking along one arc of the sequence, and these two walks of different lengths between nodes i and j certainly do not imply a cycle through i and j . So, although cycles which occur in chromosome work are generally short enough to be identified by considering the elements of the adjacency matrix and its powers, alternative approaches are required for the longer cycles which are bound to occur.

THE TIPS

Of the nodes of a graph the tips (nodes of degree one) are of special interest to us as they usually correspond to the ends of chromosome arms. The junction of the chromatids at the centromere should give rise to one or more nodes of higher degree of course, but these may equally well be due to touching or overlapping arms, or some combination of these. The tips become important once the graph of a chromosome configuration has been found to be a tree, or has been converted to a tree. For example the simple procedure of counting the tips will determine whether or not a tree is one of the chromosome graphs. If it is found to have too many tips, then, assuming that noise filtering procedures have not left any spurious tips, we require to find subgraphs which, having four or less tips each, are chromosome graphs.

It is sometimes possible to do this, for example, by determining that the tips in a certain subset belong to the same chromosome; the smallest subgraph of the tree needed to connect these tips would then be the graph for this chromosome. That such a subgraph will always exist is proved below:

THE CONNECTING GRAPH

Let T be a finite tree defined on a node set N , and let R be any subset of the nodes N .

We define the *connecting graph* of R to be a subgraph C of T satisfying the following conditions:

- (1) the node set of C includes R ;
- (2) C is connected;

- (3) if C^1 is any subgraph of C with a node set which includes R then either C^1 is not connected, or $C^1 = C$.

Such a C exists; for T^1 satisfies conditions (1) and (2), and if it does not satisfy (3), then it has a subgraph $T^1 \neq T$ which satisfies (1) and (2). This subgraph either satisfies (3) or itself has a subgraph satisfying (1) and (2) and so on. Thus as T is finite it must be possible in a finite number of such steps to obtain a subgraph of T satisfying all three conditions. This C is also unique. If not we assume that there are two such, namely C_1 and C_2
 and let $I = C_1 \cap C_2$ (I is not the zero graph because it contains at least all the nodes in R)

If I is connected then by (3)

$$I = C_1 \text{ and } I = C_2, \text{ i.e. } C_1 = C_2$$

If I is not connected then it must consist of two or more disjoint components. Let i be a node in one of these and j a node in another.

Now $i \in N$ and $j \in N$ and $T(N)$ is a tree, therefore there is a unique path P in T connecting i and j (a path is a walk in which no arc appears more than once). But i and j are both in the node set of C_1 , and C_1 is connected, so there is a path in C_1 connecting i and j . As C_1 is a subgraph of T this must be the path P . Similarly P is also a path in C_2 . So all the arcs constituting P are in both C_1 and in C_2 . They are therefore in I , and thus P is a path in I . But P connects i and j , which is contrary to the assumption that i and j are in disjoint components of I . Therefore C is unique.

EXTRACTING THE CONNECTING GRAPH

The program which we have developed for finding the connecting graph for a given subset of the nodes of a tree uses the adjacency matrix as follows: Starting from any one of the given subset of nodes (node i , say) the elements of the corresponding column in the adjacency matrix are tested in order. When a non-zero element is found the corresponding row number (j , say) gives the number of the next node. The j th column is now tested for a non-zero element other than the element (i, j) . This gives the next node, and so on. A temporary record is kept of the path followed (i.e. the value of each node and arc visited) and whenever a node belonging to the specified subset is encountered the current state of the temporary path record is copied to a permanent list—at least those parts of it which have not already been added for some earlier node. The process is continued and whenever a tip is reached the path is retraced, its temporary record being deleted at the same time, until a node with an alternative branch is reached, i.e. a column with a non-zero element succeeding that just retraced, but not the one by which this node was first reached.

This continues until all the nodes in the subset have been found. This must happen eventually as the process, if allowed to continue, will finally cover the whole graph. When all the nodes have been found the permanent record contains a list of all nodes and arcs in the required connecting graph.

This method is in effect a 'depth first' type of tree search. The tree is considered as rooted at the chosen starting node, and ordered in that the branches to nodes which come earlier in the adjacency matrix are considered as preceding those to nodes later in the matrix. This is illustrated in figure 8.

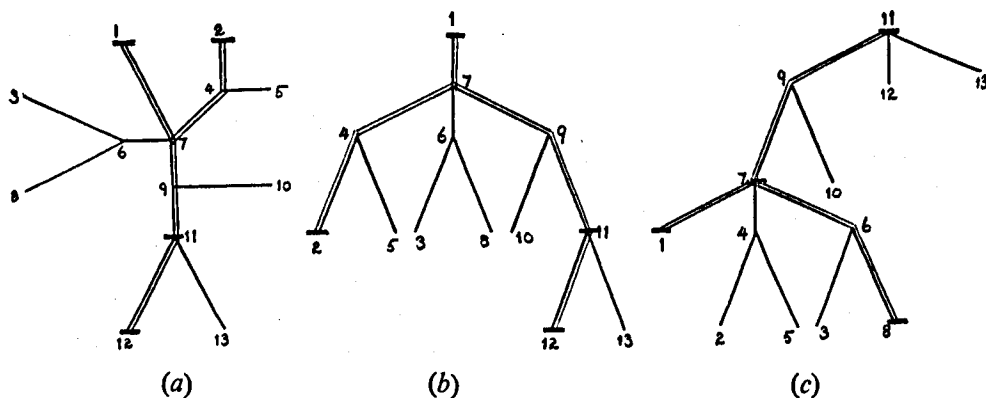


Figure 8. The connecting graph for a subset of the nodes of a tree.

(a) An example of a tree with the nodes numbered according to their positions in the adjacency matrix.

(b) The same tree with node 1 taken as root, and the branches ordered according to the values of their terminal nodes.

(c) The same tree with node 11 taken as root, and the branches ordered.

If the tree is considered to take the form shown in (b) a 'depth first' type search will find the connecting graph for nodes 1, 2, 11 and 12. This connecting graph is shown by the double lines in (a) and (b). The double lines in (c) show the connecting graph for nodes 11, 7, 1 and 8

LIMITATIONS OF GRAPH THEORY

Using these methods it is possible, if a tree is thought to represent more than one chromosome, to select some subset R of the nodes of the tree and then extract the connecting graph for R . However the only tips in such a connecting graph must be elements of the chosen node set R (because otherwise they could be removed without causing the graph to become disconnected). So if R has four or less elements, then whichever nodes these may be, the connecting graph must have four or less tips, and being a tree is therefore a chromosome graph. This means that in order to select those subgraphs most likely to correspond to chromosomes we must go outside the realms of graph theory and take into consideration such factors as the spatial relationships between the nodes and arcs in the skeleton in selecting R .

AN APPLICATION

A brief description of the method used for choosing subgraphs in the case of a particular configuration, and the way in which these subgraphs are used to

reconstruct the constituent chromosomes of the configuration is given below as an example of the way in which the graph theoretic methods described earlier can be used. Various steps in the process—from the original digitization to the reconstructed chromosomes—are illustrated in figure 9.

The configuration chosen is a comparatively straightforward and also comparatively common one. It consists of two of the larger chromosomes overlapping at or near their centromeres. From this is produced a connected graph which is recognised as being a tree with eight tips.

We wish to obtain the two subgraphs of this graph (each having four tips) which are most likely to correspond to the two chromosomes. This is done by selecting two sets of four tips from the eight, and then taking the connecting graph for each of these sets. The tips are chosen using a combinatorial method; all possible selections of two sets each consisting of two pairs of tips are considered.

For each selection the distance between the tips in each pair, d_i ($i=1,4$), is calculated, and also the distance between the two pairs in each set, D_i ($i=1,2$). The correct selection should have the tips in each pair close together, and the two pairs in each set far apart. The selection chosen is therefore that which minimises the expression.

$$\sum_{i=1}^4 d_i - a \sum_{i=1}^2 D_i$$

The skeleton corresponding to the connecting graph for each of these two sets of four tips is then constructed in a domain equal to that of the original digitization. A distance transformation (Rosenfeld and Pfaltz, 1965) is then applied to this domain, so that each point is given a value equal to its 'distance' from the skeleton ('distance' being in a metric which is a rough approximation to Euclidean distance). From these distance transforms two chromosomes are then constructed. This is done by dividing the density value at each point in the original digitization between the new chromosomes in a proportion dependent on the 'distance' of the point from the two skeletons. This proportion is determined by a matrix $\{p_{ij}\}$; the element p_{ij} is the proportion given to a point which is at 'distance' i from its skeleton, and at 'distance' j from the other skeleton. Thus, if a point is far from one skeleton and near to the other, then the whole of the density value at that point is given to the latter, whereas if a point is equidistant from the two, the value is divided equally between them. In the first instance the values of the elements p_{ij} have been determined in an entirely *ad hoc* manner. Nevertheless, as can be seen from figure 9, the results are, we feel, sufficiently encouraging to validate the use of this method and justify its further development.

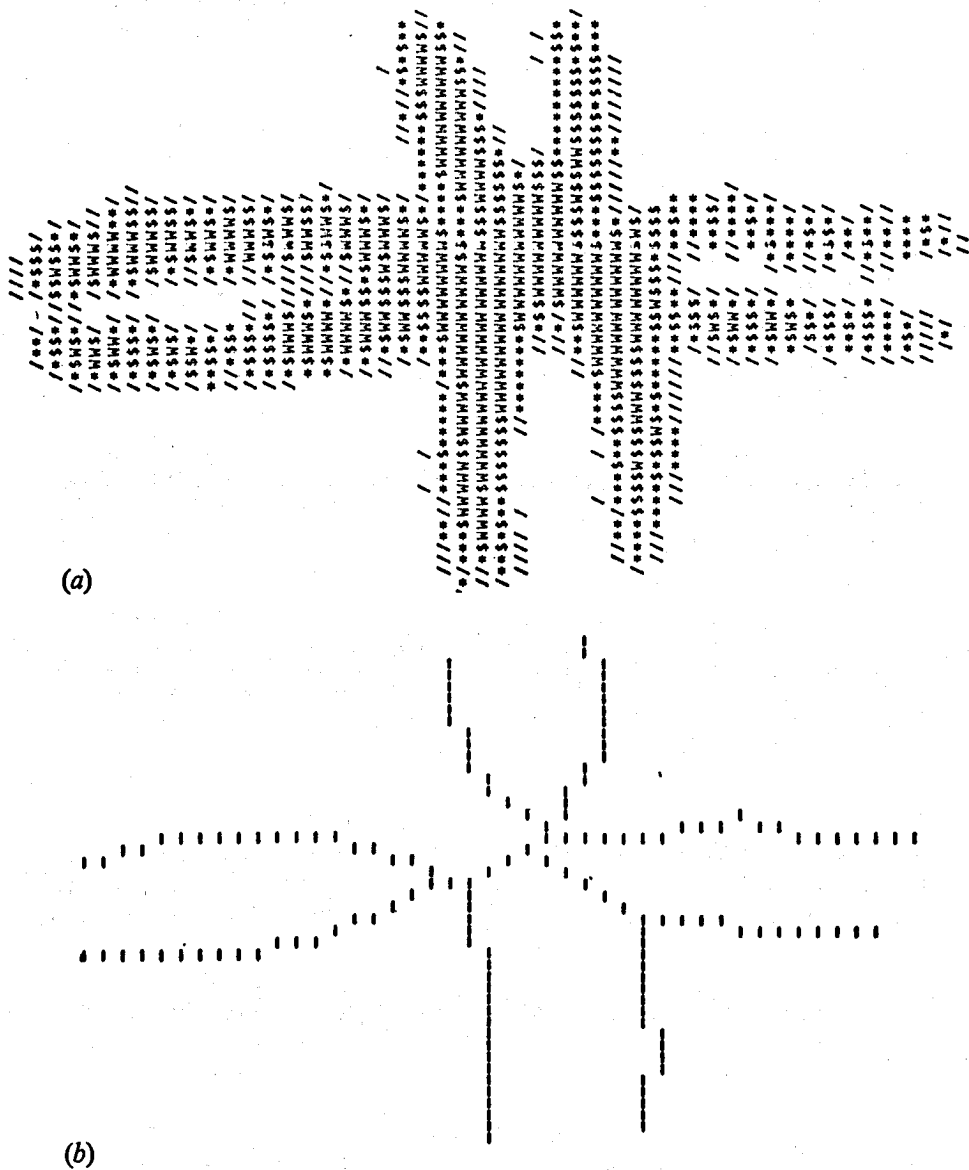


Figure 9. Resolving the two component chromosomes in an overlap.
 (a) Digitization of a configuration consisting of two overlapping chromosomes.
 (b) Skeleton of the configuration

[illegible]

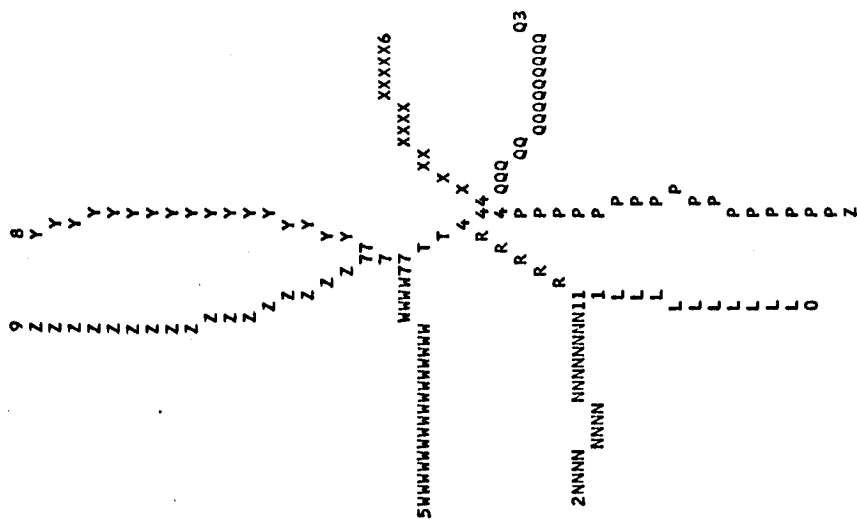
(d)

Figure 9.

(c) Labelled skeleton.

(d) Labelled adjacency matrix of skeleton

(c)



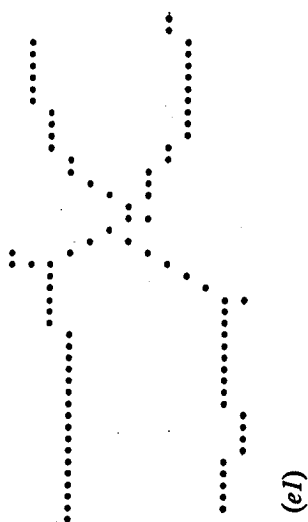
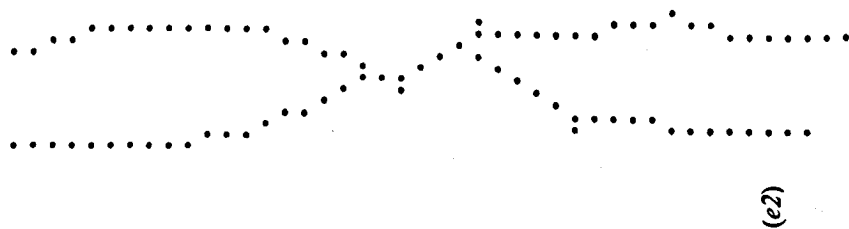


Figure 9 (e1) and (e2). Two skeletons obtained by selecting two sets of tips from the original, finding the connecting graph for each of these, and reconstructing the corresponding parts of the skeleton

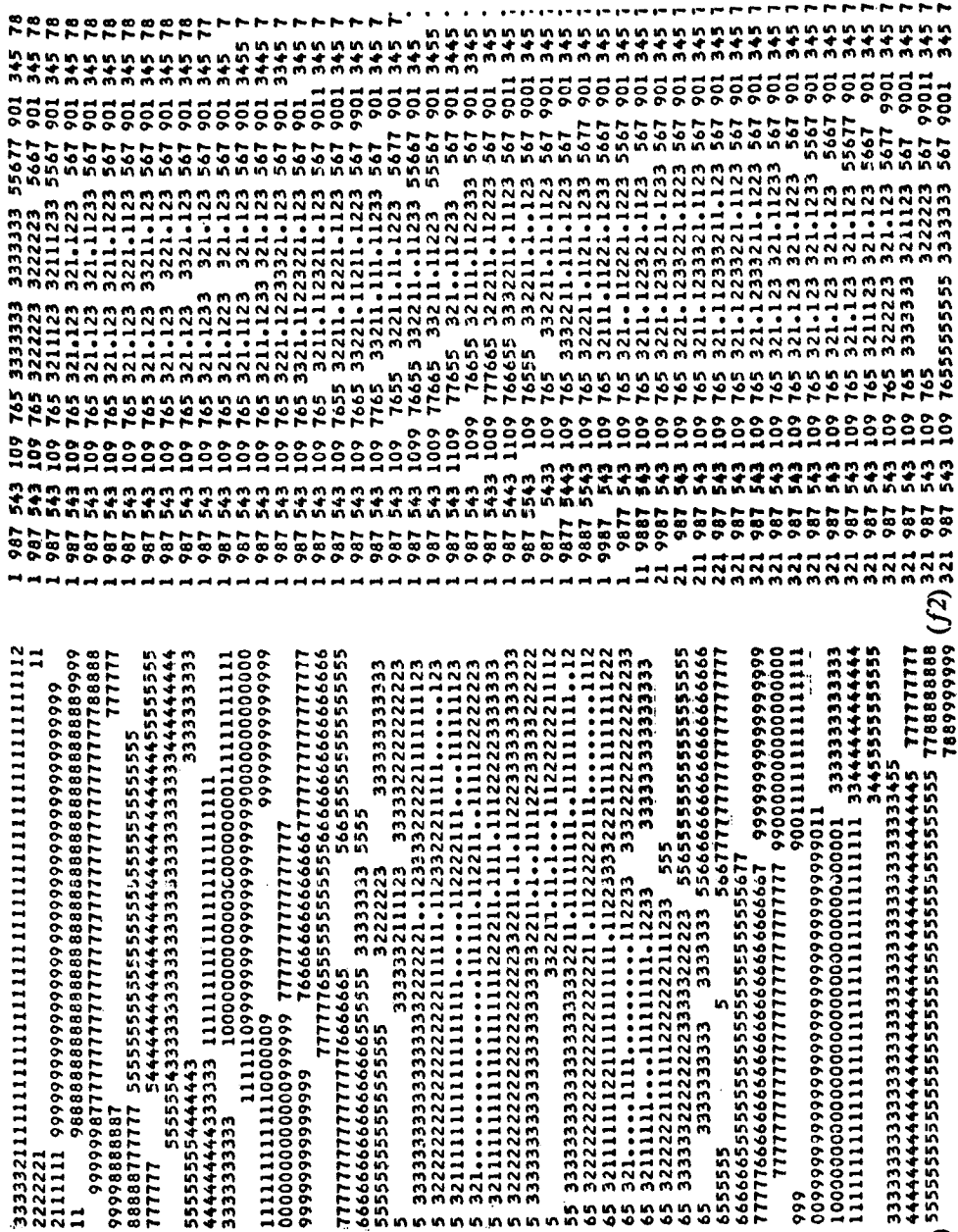


Figure 9 (*f1*) and (*f2*). Result of applying the distance transform to these two skeletons. Points with value zero (i.e. points on the skeleton) are represented by "."; the last decimal digit is given for points with non-zero value, with the exception of multiples of four which are printed as blanks

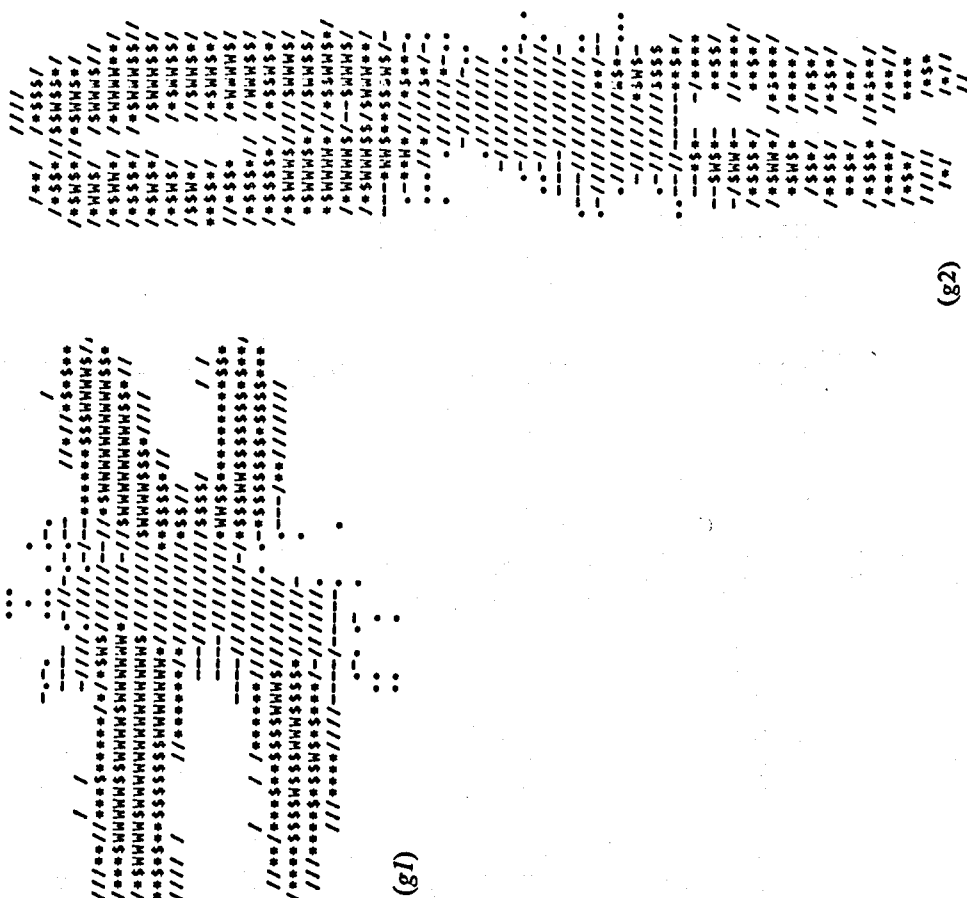


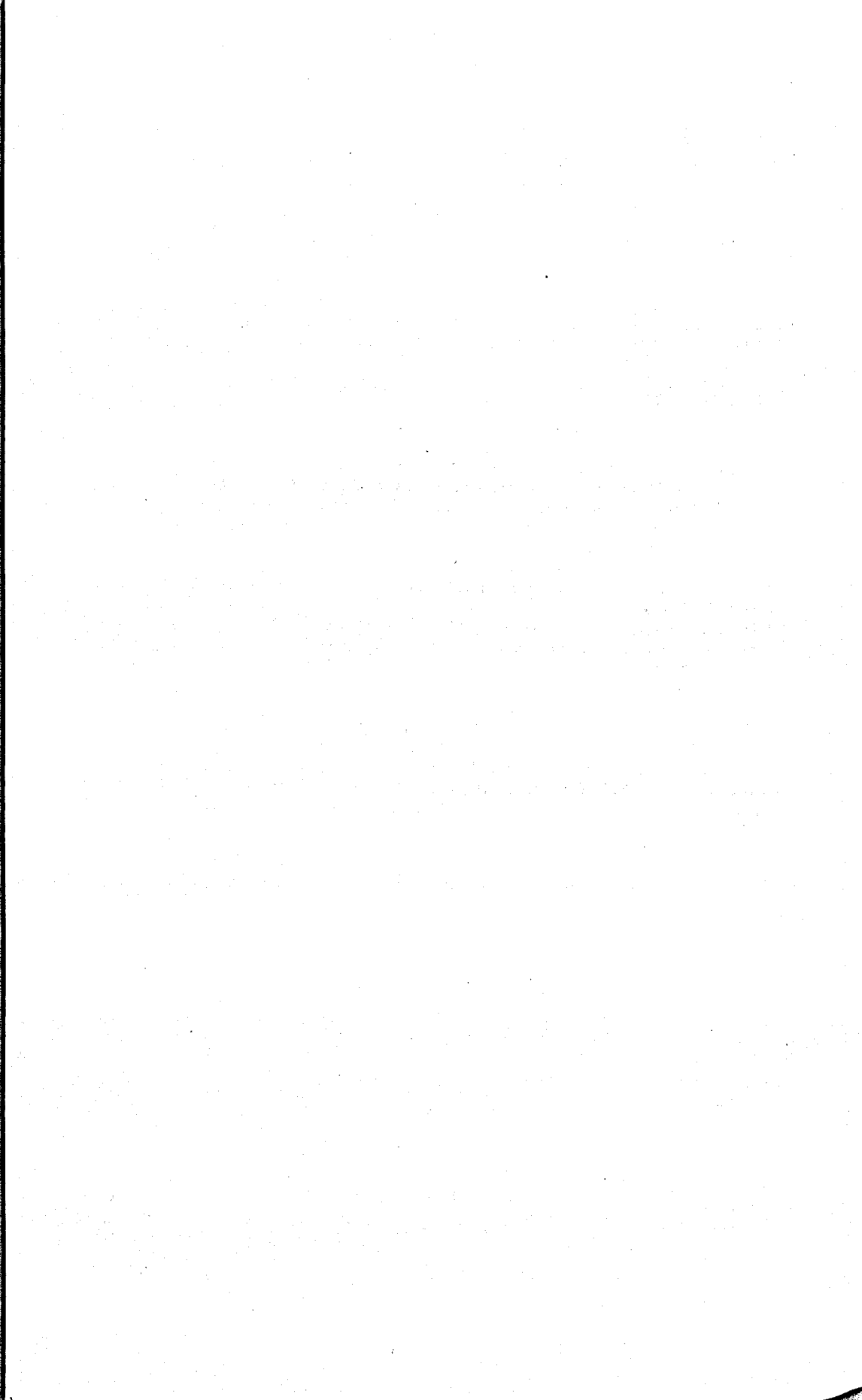
Figure 9 (g1) and (g2). The resulting reconstructed chromosomes

Acknowledgments

The work described in this article was done with the collaboration of several of my colleagues of the Clinical and Population Cytogenetics R.U., especially S. Farrow, K. Paton and B. Stein. Also Dr Denis Rutovitz to whom I am indebted both for many of the ideas which this paper contains, and for his continued help and encouragement without which it would never have been completed. Thanks are due also to Dr Patricia Jacobs for guidance through the by-ways of cytogenetics, and to N. Davidson and other photographic staff of the unit for the various photographs. All experimental work has been done using a Fidac film scanner, built by the National Biomedical Research Foundation, in conjunction with the Imperial College (London) 7090 computer used by courtesy of the college and IBM.

REFERENCES

- Blum, H. (1964), *A transformation for extracting new descriptors of shape*. Bedford, Mass.: Air Force Cambridge Research Laboratories.
- Hilditch, J. & Rutovitz, D. (1967), Algorithms for chromosome recognition. *Proceedings of the Conference on Data Extraction and Processing of Optical Images in the Medical and Biological Sciences*. New York Academy of Sciences (in press).
- Hodes, L. (1961), *Machine Processing of Line Drawings*. MIT: Lincoln Laboratory.
- McCormick, B. H. (1963), The Illinois pattern recognition computer—Illiac III. *I.E.E.E. Transactions on Electronic Computers* EC-12,5.
- Narasimhan, R. (1964), Labeling schemata and syntactic descriptions of pictures. *Inf. Control*, 7, 151–79.
- Ore, O. (1962), *Theory of Graphs*. American Mathematical Society.
- Philbrick, O. (1966), A study of shape recognition using the medial axis transformation. *Physical Sciences. Research Papers*, 288. Bedford, Mass.: Air Force Cambridge Research Laboratories.
- Rosenfeld, A. & Pfaltz, J. L. (1965), *Pattern Recognition IV: Sequential Operations in Digital Picture Processing*. University of Maryland, Computer Science Centre.
- Rutovitz, D. (1967), Machines to classify chromosomes? *Human Radiation Cytogenetics*. Amsterdam: North Holland Press.
- Rutovitz, D., Farrow, S. Hilditch, J. & Stein, B. (1968), *Computer Processing of Digital Images*.



**PROBLEM-ORIENTED
LANGUAGES**

Some Semantics for Data Structures

David Park

Programming Research Group
University of Oxford

1. INTRODUCTION

The aim of this article is to describe some ways to discuss problems of interpretation arising in connection with 'data structures', as they are provided for high-level programming languages.

It is well known to machine-code programmers and to users of list processing languages that methods of representing data by means of systems of 'pointers' and 'code-words' are indispensable for certain sorts of problem. This has been particularly the case for heuristic programming, and the use of such structures is now widespread in other areas also; the design of an appropriate 'data structure' to represent some feature of interest in a particular problem is now accepted practice, as well as sophisticated techniques for manipulating the pointers and code-words involved.

In designing a high-level language, therefore, it is now desirable that it include powerful features for doing this sort of thing, in as general a way as can be managed. However, this gives rise to certain problems in connection with defining the 'semantics' of the language (i.e. what the user is to expect of his program when run). It becomes important to distinguish rigorously between those data structures which are 'pointer-like' and those which are conceived of more conventionally.

As an example of the confusion which can arise, we will consider a particularly simple data-structure, constructed from a pair of real numbers. Suppose we want to add to a language, which will be taken as CPL here, a facility for constructing and decomposing such objects. The syntactic problem is trivial; we add a new type *pair* to the type symbols of the language, and three basic functions to the repertoire of system functions. These are *Pair*, a function taking two *real* parameters, and constructing an object of type *pair* from them, and selector functions *H*, *T*, each taking a parameter of type *pair* and producing a *real* result, the appropriate component of the pair. We suppose

that variables of type pair can be assigned to, and that assignments to components, such as

$H[A] := 3 \cdot 14$

are permitted. These last possibilities give rise to the confusion. Consider, for example, the following fragment of a CPL program:

```

let A = Pair[2, 2]
let B = Pair[2, 2]
let a ≈ H[A]           || Initialization 'by reference'; see 3.1.
Write[A = B]           || Output true or false
H[B] := 3
A := B
    
```

With the following three possible continuations:

(i)	(ii)	(iii)
$H[B] := 4$	$H[A] := 4$	$a := 4$

each followed by

$Write[a, H[A], H[B], (A = B)]$

The two interpretations (A), (B) I have in mind produce the following output:

(i)	(ii)	(iii)
(A) true, 3, 3, 4, false	true, 4, 4, 3, false	true, 4, 4, 3, false
(B) false, 2, 4, 4, true	false, 2, 4, 4, true	false, 4, 3, 3, true

The discrepancies between these results are explicable, of course, by saying that in (B), pointers are held as representations of pairs, which indicate the storage space containing their values; a has a location within the storage space originally associated with A , but after $A := B$ this association no longer holds. On the other hand, interpretation (A) associates A once and for all with a particular storage area, and an assignment to A copies into this area.

This is fine, for such a simple-minded structure, if one understands about pointers and storage areas. But it does raise some fundamental problems for the language designer, of the following sort:

- (a) How can such explanations be rigorized, say in order to prove results about programs involving pairs?
- (b) Does the explanation characterize all that the programmer needs to know about pairs?
- (c) Are there other sorts of pairs with similar characteristics, but nevertheless different in some obscure way?

These are questions about the 'abstract' nature of data structures, and as such it seems fair to call them 'semantic' questions.

It is important to appreciate that the real difficulties here arise from assignment statements in the language, in particular from the interaction between the assignment $A := B$, and assignments to components of A and B . In a language without assignments these problems disappear, and one could confine oneself to a description of the objects in traditional mathematical

terms. Nevertheless, programmers continue to use assignments in their programs, often in very natural and powerful ways, especially in connection with complicated data structures.

These problems will be attacked in two rather different ways. In Section 2 we sketch an axiomatic system which can be used to prove results about 'computational objects', and derive some results concerning the 'pairs' of Section 1. This is intended primarily as an exercise in formulation (the proofs involved are more or less trivial), the aim being to put the formulation in a form which could without difficulty be applied to the semantics of any programming language. What this section is primarily concerned with is the relationship between assignment operations, various notions of identity, and notions of 'location'.

In Section 3 a method is described for representing a particular choice of data structure in terms of other objects whose properties might already be known to the user. These are the objects known in CPL as 'functions', 'routines', 'load-update-pairs'. To the extent that the semantics of the latter sort of object are defined unambiguously, so will the semantics of the data structures concerned, using this method.

CPL (CPL *Working Papers*, 1966) is used throughout as an exemplar of the sort of programming language to which these methods are applicable. However, of current programming languages, only CPL has the features which enable the reduction of Section 3 to go through, as far as I know.

The *L*-value-*R*-value distinction in Section 2, and the *LUP* device used in Section 3, are due to Strachey (1966, 1968); without these crucial ideas, the sort of development I describe would not be possible.

2. SOME ABSTRACT CONSIDERATIONS

It is possible to formulate some of the relevant properties of data structures, assignments and 'locations' in an abstract fashion, which does not seem to depend too much on the particular programming language involved, and which avoids any reference to particular representations of the objects involved. This section will describe one approach to such a formulation; we first present (informally) (2.1) an abstract model for talking about 'computational objects', then state (2.2) some plausible basic propositions about this model, and finally (2.3) deduce some consequences of these. Theorem 2.3.1 will show that, assuming these basic propositions, there is a very close two-way relationship between the interpretation of assignment statements of the language and notions of identity in the abstract model. This theorem appears to embody a useful general principle, and will be used here to exclude from consideration a number of alternative interpretations of the data structures of the last section which might otherwise appear plausible.

2.1. *L*-values. *R*-values

We want to examine just the effects of single *commands* (in fact, assignment

commands) in a programming language, which we take here to be CPL (to avoid having to talk of syntax). Commands may contain occurrences of *expressions*. Certain expressions are *identifiers*. Expressions are classified into *types*; the type of an occurrence of an expression is a 'manifest' property, i.e. determined by inspection of the text of the program alone. We restrict ourselves here to single commands which are not qualified by declarations of new identifiers. An occurrence of an expression within a command can be classified as an *L-occurrence* or as an *R-occurrence*, depending on its context; in CPL this classification is determined by certain assumptions about the operator (i.e. arithmetic operator, function identifier, assignment operator, routine identifier, etc.) which 'governs' the occurrence of the expression, and this classification is 'manifest' also. In an assignment command of the form

$$E_1 := E_2$$

the occurrence of E_1 is an *L-occurrence*, and that of E_2 an *R-occurrence* (we use the letter E with a subscript as a variable standing for an expression, and the letter I with a subscript as a variable standing for an identifier). The significance of this classification will be made clear below. (Of course there are occurrences which are neither *L-* nor *R-occurrences*, e.g. occurrences within quotation marks of some sort.)

It is necessary to talk about the *effect* of a command. The device adopted here is to refer to an instantaneous *state* of a computation. The state $\langle \mathcal{L}, C \rangle$ consists of two mappings $\mathcal{L}$ and C ; $\mathcal{L}$ is a map from a certain class of expressions, the class of *legal expressions* at that point, into a class of *locations*; C is a mapping from the class of locations into a class of *objects* (which need not be disjoint from the class of locations). The composition of C and $\mathcal{L}$ is denoted by $\mathcal{R}$. Thus $C(\mathcal{L}(E_1)) = \mathcal{R}(E_1)$ for any legal expression E_1 . $\mathcal{L}(E_1)$ is referred to as the *L-value* of E_1 ; $\mathcal{R}(E_1)$ is the *R-value* of E_1 .

The method adopted here will be to relate certain assertions about the state at a given moment to assertions about states resulting from commands which might be executed immediately afterwards. Subscripts on $C, \mathcal{L}, \mathcal{R}$ will be used to indicate which state is referred to. Thus $\mathcal{R}_0(E_1)$ is the *R-value* of E_1 at the moment in which we are interested, and $\mathcal{R}_1(E_1)$ is the *R-value* which results after some given command (which should be clear from context) has been executed. The *effect* of the given command determines the new state $\langle \mathcal{L}_1, C_1 \rangle$. Thus if two commands produce different mappings $\mathcal{L}_1$ or C_1 , their effects can be said to be different.¹

¹ It seems undesirable to assume the converse; if the command is an 'output' command, it has an effect not reflected in the state as envisaged here. On the other hand, the effect of an 'input' command is determined by some 'environment' which again is not specified in the state. This, of course, leaves the concept of 'effect' somewhat vague, which may be its right and proper condition if it is to stand for what computer people have in mind when they use the word. The most we need assume here is that the effect of a command is determinate in so far as it affects the state; and there is some general

The intention of the *R*-value-*L*-value distinction should be clear to those familiar with the CPL literature (Strachey, 1966, 1968; CPL *Working Papers*, 1966). Roughly speaking, the *R*-value of an expression is its 'value' in the conventional sense, of the ALGOL 60 Report (Naur, 1963), say. Thus, after

$$\begin{aligned} &\text{let } a, b = 2, 2 \\ &a := a + b \end{aligned}$$

the *R*-value of *a* is 4 (the numeral '4', the number 4, or some representation of either, according to taste.) On the other hand, the *L*-value of *a* is what is required to make sense of the command $a := a + b$ above, and distinguish its effect from that of $b := a + b$. What is relevant here is clearly not the current *R*-value of *a* (which happens to be the same as that of *b*). On the naïve level, what does seem to be relevant is some concept of the 'location' of *a*; it is this naïve sense which is intended to be captured by the concept of '*L*-value'. The term 'value' is used, since many expressions can be construed as descriptions of location, as well as of the objects which live in them; the process of obtaining the location described can be regarded as a sort of evaluation. For example

$$(a > b \rightarrow a, b) := 0$$

makes intuitive sense, as assigning 0 to the larger of *a* and *b*; the expression on the left is evaluated by taking $\mathcal{R}(a)$, $\mathcal{R}(b)$, comparing them and giving result $\mathcal{L}(a)$, $\mathcal{L}(b)$ depending on the comparison. In an *R*-context, of course, the relevant value would be $\mathcal{R}(a)$ or $\mathcal{R}(b)$.

In fact it makes some sense to turn this idea on its head, as we do here, and maintain that the only evaluation process to be accounted for in specifying the language is the *L*-evaluation process; *R*-evaluation is obtained from *L*-evaluation by composition with *C*, i.e. $\mathcal{R}(E_1) = C(\mathcal{L}(E_1))$, for all expressions E_1 . This implies a certain artificiality when it comes to expressions such as $(a + b)$ which do not describe locations as well as objects. It is necessary to adopt the device that $\mathcal{L}(a + b)$ is always a *constant location*, where such a location *l* has the property that $C(l)$ does not change (of course $\mathcal{R}(a + b)$ may change, under assignments to *a* and *b*, but at any instant this value is a constant location). Thus

$$a + b := 2$$

criterion for saying whether the 'effects' of two alternative commands are the same or not. Another way of putting this is that an 'interpretation' of this abstract formulation would need to specify not only a particular machine, and a way of translating our states into machine states, but also an environment for the machine, some notion of what the 'essential' effects of the machine on its environment are, and some rules as to how the environment reacts to these effects. This point is, fortunately, irrelevant to the issue at hand, except in providing some justification for approaching the problem of semantics *via* formulations for deducing incomplete assertions about effects from incomplete assertions about states and environments, rather than attempting complete descriptions of them.

must be considered erroneous at some stage (situations can be devised for which such an error can only be detected dynamically). This device may be not completely without intuitive content; fancifully, it gives semantic sense (in our very limited use of the word) to the constant-variable distinction of informal mathematics; an expression denotes a 'variable' if it has an L -value whose contents can in principle change.

To give L -evaluation the principal role is almost as much as to say that L -values are to the identifiers of a program as numerical values are to variables of a numerical formula. However this analogy does not go through completely, since it is not possible in general to L -evaluate an expression out of context, knowing just the L -values of the identifiers it mentions (which would be the case if the analogy did hold); one may also need R -values for them, which may depend on the context of the occurrence. This is the case, for example, with the expression $(a > b \rightarrow a, b)$, whose L -value depends on $\mathcal{R}(a), \mathcal{R}(b)$.

An alternative way of conceiving of L -values and R -values is to take as primitive notions equivalence relations L -equivalence, R -equivalence which here appear as 'having the same L -value', 'having the same R -value'. L - and R -values can then be regarded as equivalence classes of legal expressions with respect to these relations (the contents mapping C becomes the mapping taking an L -equivalence class into the R -equivalence class containing it). We are interested, however, in the interpretation of L - and R -values as the representations manipulated by a machine, and in whether certain commands 'change' these; it requires some additional apparatus to talk about change in the equivalence class formulation, in order to 'label' equivalence classes as they vary during execution.

Translated in terms of these equivalence relations, the statement that expressions cannot in general be L - or R -evaluated out of context is to say that there is no overall notion of 'substitutivity of equivalence' with respect to the equivalence notions, unless one can guarantee somehow that the equivalence is one which holds throughout any execution of the portion of program under consideration (as one can for L -equivalent identifiers, having due regard for scope rules). One can formulate substitution rules of very limited validity, and these are presented below as basic propositions 2.2.1, 2.2.2.

2.2. Basic Propositions

It is necessary to assume, in formulating 2.2.1–2.2.6 below, which concern the effects of single commands, that only legal expressions 'without side-effects' are involved in the commands considered, i.e. that no effect of a command is attributable to the evaluation process for its component expressions, and, particularly, that the R - and L -values of these do not depend on the order in which they are evaluated, or on the number of times they are evaluated. This is an annoying restriction to impose, but it is not possible to extend these principles very far in the absence of such a condition without

referring in more detail to the evaluation rules of the language, which we wish to avoid doing here.

2.2.1 If $\mathcal{L}_0(E_1) = \mathcal{L}_0(E_2)$, then E_1 may be substituted for E_2 (or *vice versa*) at one or more L -occurrences or R -occurrences in a command K without alteration to the effect of K ; in particular $\mathcal{L}_1, C_1$ are independent of the substitution.

2.2.2 If $\mathcal{R}_0(E_1) = \mathcal{R}_0(E_2)$, then 2.2.1 holds when restricted to R -occurrences of E_1, E_2 .

It should be emphasized that 2.2.1, 2.2.2 are not to be regarded as definitions of the notions of L -equivalence, R -equivalence. Even if one restricts considerations of 'effect' to considerations of the resulting state $\langle \mathcal{L}_1, C_1 \rangle$, the conclusions of 2.2.1, 2.2.2 turn out to be undecidable properties of E_1, E_2 and the state, when one comes to consider the functions and $LUPs$ of Section 3.

The essential property of the assignment command itself appears to be the following:

2.2.3 If $E_1 := E_2$ is a legal assignment¹ then after $E_1 := E_2$ the following identity holds:

$$C_1(\mathcal{L}_0(E_1)) = \mathcal{R}_0(E_2)$$

We are assuming here rather more general properties of locations than are associated with the notion of 'address' in a machine. This is essential because we wish, for example, to investigate the relationship between $\mathcal{L}(A)$, $\mathcal{L}(H[A])$ where A is an identifier of type *pair* as in Section 1. One of the questions one can ask is whether these locations 'share', i.e. whether assignments to one change the object in the other or not. What is characteristic of machine addresses is the relation of *disjointness*. This can be taken as a primitive symmetric relation on locations, with the following properties.

2.2.4 If l_1 is disjoint from l_2 then (i) $l_1 \neq l_2$; (ii) $C(l_2)$ is unchanged by any command $E_1 := E_2$ with $\mathcal{L}_0(E_1) = l_1$.

We will not use the disjointness relation in this section, but it is clearly relevant to the following momentary property of expressions.

Definitions. E_1 *affects* $\mathcal{L}(E_2)$ (or $\mathcal{R}(E_2)$) if there is a legal assignment $E_1 := E_3$ which changes $\mathcal{L}(E_2)$ (or $\mathcal{R}(E_2)$).

E_1 is *tame* if no legal assignment $E_1 := E_2$ changes $\mathcal{L}(E_1)$.

Thus, if E_1 does not affect $\mathcal{L}(E_2)$ and $\mathcal{L}_0(E_1)$ is disjoint from $\mathcal{L}_0(E_2)$, then E_1 does not affect $\mathcal{R}(E_2)$. Note that there are non-tame expressions, for example the left side of

$$(b \rightarrow b, c) := \text{false}$$

with $\mathcal{R}_0(b) = \mathcal{R}(\text{true})$ and $\mathcal{L}(b) \neq \mathcal{L}(c)$.

¹ An assignment $E_1 := E_2$ is *legal* if E_1, E_2 are legal expressions, of the same type, and $\mathcal{L}_0(E_1)$ is not a constant location.

2.2.5 For identifiers I_1 , $\mathcal{L}(I_1)$ is not affected by any assignment.

Therefore every identifier is tame. 2.2.5 amounts to a convention for labeling certain L -equivalence classes. (There is an awkwardness in applying this in the case of parameters called by name in ALGOL 60 procedures; either 2.2.5 stands, in which case $\mathcal{L}(I_1)$, for such an identifier I_1 , must be chosen to be distinct from other L -values, although it might satisfy the conclusion of 2.2.1 from time to time; or occurrences of I_1 must be dealt with as calls for parameterless procedures which can be used on the left sides of assignments, in the way that 'call by name' is treated in the current proposal (Sept. 1967) for ALGOL X; the simplest device is probably to drop 2.2.5 for such identifiers.)

For 2.3 it will be necessary to refer to a further primitive property of locations, of being *basic*. The intention of this notion is that the state is completely determined by specifying just $\mathcal{L}$ restricted to identifiers, and C restricted to basic locations, the language rules determining how $\mathcal{L}$ and C may be extended. There are non-basic locations in CPL (see LUPs in Section 3.1).

2.2.6 Let $\langle \mathcal{L}, C \rangle$ be a state and E_1 a legal expression of type T in this state. Then there is a state $\langle \mathcal{L}', C' \rangle$ such that

- (i) $\mathcal{L}', C'$ extend $\mathcal{L}, C$;
- (ii) there is a legal identifier I_1 of type T in state $\langle \mathcal{L}', C' \rangle$ such that $\mathcal{R}(I_1) = \mathcal{R}(E_1) = \mathcal{R}(E_1)$, and $\mathcal{L}(I_1)$ is basic;
- (iii) for any E_2 legal in $\langle \mathcal{L}, C \rangle$, I_1 does not affect $\mathcal{L}'(E_2)$, $\mathcal{R}'(E_2)$, and E_2 does not affect $\mathcal{R}'(I_1)$.

Note. (1) $\langle \mathcal{L}', C' \rangle$ is effectively the state which results from obeying the CPL declaration

let $I_1 = E_1$

where I_1 is chosen to be distinct from previously legal identifiers. Property (iii) results from the choice of $\mathcal{L}(I_1)$ to be disjoint from all previously allocated L -values.

(2) Properties 2.2.1–2.2.5 should be viewed as constraints on states and on effects, which are implied by the complete body of language rules; the word 'state' should be taken as 'state satisfying the language rules' in 2.2.6, and this is implicit in the other rules also. The state resulting from a command is, of course, assumed also to satisfy the language rules.

(3) For notational reasons, an application of 2.2.6 is referred to as 'choosing a new identifier', and the notation for the current state $\langle \mathcal{L}, C \rangle$ used rather than referring to the revised state $\langle \mathcal{L}', C' \rangle$.

2.3. Some Consequences for pairs

We can establish the following very simple result.

2.3.1 *Theorem.* Let $\mathcal{F}$ be any function from expressions to expressions (which preserves legality). The following conditions on states satisfying the language

rules, on a type T , and on legal expressions E_1, E_2 of type T , are equivalent:

- (i) if $\mathcal{R}(E_1) = \mathcal{R}(E_2)$ then $\mathcal{L}(\mathcal{F}(E_1)) = \mathcal{L}(\mathcal{F}(E_2))$.
- (ii) If $\mathcal{L}(E_1)$ is basic,¹ and if E_1 is tame and does not affect $\mathcal{R}(E_2)$ or $\mathcal{L}(\mathcal{F}(E_2))$, then after $E_1 := E_2$, $\mathcal{L}_1(\mathcal{F}(E_1)) = \mathcal{L}_0(\mathcal{F}(E_2))$.

Proof. (i) $\Rightarrow$ (iii). This is a direct consequence of 2.2.3; E_1 is tame, so $\mathcal{L}_1(E_1) = \mathcal{L}_0(E_1)$. Therefore $\mathcal{R}_1(E_1) = C_1(\mathcal{L}_0(E_1)) = \mathcal{R}_0(E_2)$ by 2.2.3. Since E_1 does not affect $\mathcal{R}(E_2)$, $\mathcal{R}_1(E_1) = \mathcal{R}_1(E_2)$. Therefore $\mathcal{L}_1(\mathcal{F}(E_1)) = \mathcal{L}_1(\mathcal{F}(E_2))$ by (i); the conclusion follows, since E_1 does not affect $\mathcal{L}(\mathcal{F}(E_2))$.

(ii) $\Rightarrow$ (i). Suppose $\mathcal{R}_0(E_1) = \mathcal{R}_0(E_2)$. From 2.2.6, choose I_1 of type T , not affecting $\mathcal{R}(E_1)$, $\mathcal{R}(E_2)$, $\mathcal{L}(\mathcal{F}(E_1))$, $\mathcal{L}(\mathcal{F}(E_2))$. Consider the effect of $I_1 := E_1$. From (ii) and 2.2.5, $\mathcal{L}_1(\mathcal{F}(I_1)) = \mathcal{L}_0(\mathcal{F}(E_1))$.

But from 2.2.2, the effect is the same as that of $I_1 := E_2$. Therefore $\mathcal{L}_1(\mathcal{F}(I_1)) = \mathcal{L}_0(\mathcal{F}(E_2))$. Hence $\mathcal{L}_0(\mathcal{F}(E_1)) = \mathcal{L}_0(\mathcal{F}(E_2))$.

Remarks. (1) The result holds with $\mathcal{R}(\mathcal{F}(E))$ replacing $\mathcal{L}(\mathcal{F}(E))$ throughout, and can easily be extended to cover a variety of other transitive symmetric conditions on E_1, E_2 .

(2) Without further restrictions on $\mathcal{F}$, the conditions on $\mathcal{R}(E_2)$, $\mathcal{L}(\mathcal{F}(E_2))$ in (ii) are necessary. Suppose the language is CPL, but the rules always permit p as a boolean identifier; (ii) fails taking p as E_1 , $\sim p$ as E_2 , $(E \equiv p)$ for $\mathcal{F}(E)$ (p affects $\mathcal{R}(\sim p)$, but not $\mathcal{L}(\sim p \equiv p)$); it also fails with p as E_1 , false as E_2 , $\sim p$ for $\mathcal{F}(E)$, with $\mathcal{R}_0(p) = \mathcal{R}(\text{true})$ (this time p affects $\mathcal{L}(\sim p)$, but not $\mathcal{R}(\text{false})$). (Note that $\mathcal{L}(\text{true}) \neq \mathcal{L}(\text{false})$).

From 2.3.1, the basic propositions, and a number of additional assumptions about pairs, it is possible to narrow down the class of plausible interpretations of these data structures.

We confine ourselves here to properties of legal identifiers I_1, I_2 etc., of type pair, about which we assume the following:

2.3.2 If $\mathcal{L}(I_1)$ is basic, after $I_1 := I_2$, $\mathcal{R}_1(H[I_1]) = \mathcal{R}_0(H[I_2])$ and $\mathcal{R}_1(T[I_1]) = \mathcal{R}_0(T[I_2])$.

2.3.3 The expressions $H[I_1]$, $T[I_1]$ are always tame and never affect each others' L - or R -values.

The bifurcation of interpretations of pairs can be looked at as springing from the two following alternative definitions of identity between pairs.

2.3.4A $\mathcal{R}(I_1) = \mathcal{R}(I_2)$ iff $\mathcal{R}(H[I_1]) = \mathcal{R}(H[I_2])$ and $\mathcal{R}(T[I_1]) = \mathcal{R}(T[I_2])$.

2.3.4B $\mathcal{R}(I_1) = \mathcal{R}(I_2)$ iff $\mathcal{L}(H[I_1]) = \mathcal{L}(H[I_2])$ and $\mathcal{L}(T[I_1]) = \mathcal{L}(T[I_2])$.

We will assume that these two conditions are incompatible, i.e. that there exists a state with I_1, I_2 satisfying the right hand condition of 2.3.4A, but such that $\mathcal{L}(H[I_1]) \neq \mathcal{L}(H[I_2])$, say. We will also assume that for any I_1 there is an I_2 such that $\mathcal{R}(H[I_1]) \neq \mathcal{R}(H[I_2])$.

How do changes to pairs and their components interact? We consider for the moment just the component H . Firstly what is the effect of an assignment

¹ The assumption is not necessary for (i) $\Rightarrow$ (ii).

$I_1 := I_2$ with $\mathcal{L}(I_1)$ basic, on $\mathcal{L}(H[I_1])$ ($\mathcal{R}(H[I_1])$ is covered by 2.3.2)? There appear to be two reasonable possibilities again:

2.3.5A After $I_1 := I_2$, $\mathcal{L}_1(H[I_1]) = \mathcal{L}_0(H[I_1])$.

2.3.5B After $I_1 := I_2$, (i) $\mathcal{L}_1(H[I_1]) = \mathcal{L}_0(H[I_2])$

(ii) $C_1(\mathcal{L}_0(H[I_1])) = \mathcal{R}_0(H[I_1])$

From 2.3.1, if 2.3.5B holds, then $\mathcal{R}(I_1) = \mathcal{R}(I_2)$ implies $\mathcal{L}(H[I_1]) = \mathcal{L}(H[I_2])$, so that 2.3.4B is the only choice for 2.3.4 compatible with 2.3.5B. On the other hand, if 2.3.4B holds, then 2.3.5B(i) holds for I_1 not affecting $\mathcal{R}(I_2)$, $\mathcal{L}(H[I_2])$; from 2.2.6 and the assumption above I_1, I_2 can be chosen with this property, and with $\mathcal{R}_0(H[I_1]) \neq \mathcal{R}_0(H[I_2])$, so that $\mathcal{L}_0(H[I_1]) \neq \mathcal{L}_0(H[I_2])$, and 2.3.5A cannot hold. So that of the four combinations for 2.3.4, 2.3.5, two are incompatible.

From 2.3.2, I_1 affects $\mathcal{R}(H[I_1])$ in both sorts of interpretation. But does $H[I_1]$ affect $\mathcal{R}(I_1)$? Assuming 2.3.4A any change to $\mathcal{R}(H[I_1])$ affects $\mathcal{R}(I_1)$. Assuming 2.3.4B, however, one can show that $H[I_1]$ never affects $\mathcal{R}(I_1)$. The proof of this fact goes as follows: from 2.2.6, choose I_2 such that $\mathcal{R}(I_2)$ is not affected by $H[I_1]$, and $\mathcal{R}_0(I_2) = \mathcal{R}_0(I_1)$. Then $\mathcal{L}_0(H[I_2]) = \mathcal{L}_0(H[I_1])$ and $\mathcal{L}_0(T[I_2]) = \mathcal{L}_0(T[I_1])$ by 2.3.4B. Consider the effect of $H[I_1] := I_3$; since I_2 is not affected by $H[I_1]$, $\mathcal{R}_1(I_2) = \mathcal{R}_0(I_2)$; since $H[I_1]$ is tame and does not affect $\mathcal{L}(T[I_1])$, by 2.3.3, $\mathcal{L}(H[I_1])$, $\mathcal{L}(T[I_1])$ are unchanged. Since $\mathcal{L}_0(H[I_1]) = \mathcal{L}_0(H[I_2])$, $H[I_1] := I_3$ has the same effect as $H[I_2] := I_3$, by 2.2.1. By repeating the above argument, $\mathcal{L}(H[I_2])$, $\mathcal{L}(T[I_2])$ are unchanged. So $\mathcal{R}_1(I_1) = \mathcal{R}_1(I_2)$, by 2.3.4B. But then $\mathcal{R}_1(I_1) = \mathcal{R}_1(I_2) = \mathcal{R}_0(I_2) = \mathcal{R}_0(I_1)$. These results are summarised in (i), (ii) and (iii) of the following:

2.3.6 Theorem

- (i) The pairs (2.3.4A, 2.3.5B), (2.3.4B, 2.3.5A) are each incompatible.
- (ii) Assuming 2.3.4A, any command affecting $\mathcal{R}(H[I_1])$ also affects $\mathcal{R}(I_1)$.
- (iii) Assuming 2.3.4B, $H[I_1]$ never affects $\mathcal{R}(I_1)$.
- (iv) Assuming 2.3.4A, the occurrence of I_1 in $H[I_1] := I_2$ is not an R -occurrence, if $\mathcal{R}_0(I_2) \neq \mathcal{R}_0(H[I_1])$.

To establish (iv), choose I_3 not affected by $H[I_1]$, with $\mathcal{R}_0(I_3) = \mathcal{R}_0(I_1)$. If I_1 in $H[I_1] := I_2$ is an R -occurrence, the effect is the same as that of $H[I_3] := I_2$; but the latter command affects I_3 , and the former does not, by choice of I_3 .

Remarks (1) Statement (iii), above, can be looked at as implying that $\mathcal{R}(A)$, assuming 2.3.4B, is 'pointer-like', in the sense that it is invariant with respect to changes in the objects pointed at. On the other hand, (ii) implies that $\mathcal{L}(H[I_1])$ is a 'sublocation' of $\mathcal{L}(I_1)$, in the sense that any changes to $\mathcal{R}(H[I_1])$ change $\mathcal{R}(I_1)$.

(2) Similar results hold for the T component. If one considers a mixed interpretation, for which, say $\mathcal{R}(I_1) = \mathcal{R}(I_2)$ iff $\mathcal{R}(H[I_1]) = \mathcal{R}(H[I_2])$ and $\mathcal{L}(T[I_1]) = \mathcal{L}(T[I_2])$, then 2.3.5A, 2.3.6 (ii), (iv) hold for H components, and 2.3.5B, 2.3.6 (iii) hold with T substituted for H .

(3) There are (pathological) counterexamples to the assertions

$$2.3.4A \Rightarrow 2.3.5A$$

$$2.3.4B \Rightarrow 2.3.5B \text{ (ii),}$$

using the method of 3.2.

2.4. $Pair[a,b] = Pair[a,b]$?

We have not discussed properties of the function *Pair* in 2.3, since in one important case, the *Pair* function for representation (B) in 3.2, this function must be considered to have a side-effect, and is therefore not covered by the propositions of 2.2. For this version of *Pair*, it happens that the evaluation of $\mathcal{R}(Pair[a,b])$ affects the next value of this expression. This gives rise to the disturbing feature that the value of

$$Pair[a,b] = Pair[a,b] \quad (*)$$

may be false. This is disconcerting, and calls for some explanation (the question has worried a number of people; in particular it was raised by J. McCarthy at the Edinburgh Workshop from which this volume arises).

In fact the value of the expression (*) is ambiguous without further specification of the evaluation process. Representing it in the following form

$$\text{value of } \S \text{ let } A = Pair[a,b]; \text{ result is } (A = A) \S$$

the value is certainly true; and some languages do permit this sort of manipulation of expressions ('collecting common sub-expressions'), which is of course valid in the case of expressions without side effects.

Some such result is to be expected of any interpretation of pairs satisfying 2.3.4B, assuming that this is incompatible with 2.3.4A, and that $\mathcal{R}(a=b) = \mathcal{R}(\text{true})$ only if $\mathcal{R}(a) = \mathcal{R}(b)$. To be more precise, a contradiction is obtained if we assume that there exists a state, and legal identifiers a, b with the following properties

$$(i) \mathcal{R}_0(a) = \mathcal{R}_0(b)$$

$$(ii) \mathcal{L}_0(H[a]) \neq \mathcal{L}_0(H[b])$$

$$(iii) \text{ after } a := b, \mathcal{L}_1(H[a]) = \mathcal{L}_0(H[b])$$

$$(iv) \text{ after } a := a, \mathcal{L}_1(H[a]) = \mathcal{L}_0(H[a]).$$

All that is involved is an application of 2.2.2 to the command $a := b$.

If we assume in addition that (ii) is demonstrably the case, viz. that there is some continuation K which produces different observable effects depending on whether $\mathcal{L}(H[a]) = \mathcal{L}(H[b])$ or not, and that a does not affect $\mathcal{L}(H[b])$, then the violation of 2.2.2 is a verifiable one, i.e. the two continuations

$$a := a; K$$

$$a := b; K$$

have different observable effects. (An example of such a K might be $\S H[a] := H[a] + 1; \text{Write}[H[b]] \S$.)

In (A)-like interpretations, of course, the value of the expression (*) is always true. Also, one can deny that $\mathcal{R}(a) = \mathcal{R}(b)$ should be a necessary condition for $\mathcal{R}(a=b)$ to be true; however, one would then have to accept that one had identified discernibles; with the proviso that the method adopted

here, of varying the commands executed while holding the initial state fixed, is regarded as a valid means of discerning differences between 'computational objects'. In a programming language without assignments or their equivalent, such as 'pure' LISP, of course these facilities for discernment do not exist and an interpretation in which (*) was false would give cause for reinterpretation.

3. REPRESENTING DATA STRUCTURES IN CPL

3.1 Functions. Routines. Load-Update Pairs

We will represent the two choices (A), (B) for interpretations of pairs in terms of CPL functions, routines and *LUPs* (load-update pairs). We first give a brief account of the relevant semantics of these latter objects, using the notation of Section 2. This is done here by indicating the effect of various constructions on the state $\langle \mathcal{L}, C \rangle$.

As indicated in 2.2, the state $\langle \mathcal{L}, C \rangle$ is determined by the language rules for CPL from a *reduced state* $\langle \mathcal{P}, \hat{C} \rangle$ where $\mathcal{P}$ is the restriction of $\mathcal{L}$ to current legal identifiers, and $\hat{C}$ is the restriction of C to a certain subset of the basic locations, (the *current* basic locations); it is not assumed that there is any containment relation between $\text{range}(\mathcal{P})$ and $\text{domain}(\hat{C})$; there may be non-basic *L*-values for identifiers, e.g. constant locations or *LUPs*; and there may be current basic locations which are not *L*-values of current legal identifiers.

Declarations. $\mathcal{P}$ is affected by declarations but not by commands. Every declaration has a scope, a segment of program within which $\mathcal{P}$ takes the modified form given it by the declaration. The previous $\mathcal{P}$ is restored when execution or evaluation 'leaves' the scope. The scopes of the declarations below are all that portion of the smallest 'block' containing the declaration which follows the declaration (we will not here give an account of recursive definitions, or definitions involving *in* or *where*). Blocks are indicated by matching pairs of the brackets § § (which correspond to the symbols *begin* and *end* of ALGOL).

Simple Declarations

(i) let $I_1 \simeq E_1$,

has the effect of modifying $\mathcal{P}$ so that $\mathcal{P}_1(I_1) = \mathcal{P}_0(E_1)$ (we refer to this process as *initialization by reference* of I_1 to $\mathcal{P}(E_1)$).

(ii) let $I_1 = E_1$,

has the effect of extending the domain of $\hat{C}$ by a further basic location l which is disjoint from all other basic locations. $\mathcal{P}$ is modified so that $\mathcal{P}_1(I_1) = l$, and $\hat{C}_1(l) = \mathcal{R}_0(E_1)$ (this is *initialization by value* of I_1 to $\mathcal{R}(E_1)$).

(iii) let constant $I_1 = E_1$,

has the effect of modifying $\mathcal{P}$ so that $\mathcal{P}(I_1) = l$ where l is a constant location with $C(l) = \mathcal{R}_0(E_1)$ (this is *constant initialization* of I_1 to $\mathcal{R}_0(E_1)$).

Remarks. (1) The possibility of side-effects in the evaluations of E_1 above cannot be excluded (the function $\text{Pair}[a,b]$ for interpretation (B) below has a side-effect); these would be reflected in $\hat{C}_1$.

(2) The interpretation of initialization by value presents a problem in connection with side-effects. Regardless of whether $\mathcal{R}(E_1)$ has a side-effect or not, $\hat{C}$ is changed by obeying the definition, since its domain is extended. In the terms of Section 2.2, it is not clear whether the trivial expression

value of § let $x=1$; result is x §

should be taken to have a side-effect or not (this is the syntactic form for converting a block into an expression whose L -value is a constant location, in this case $\mathcal{L}(1)$). The problem is that of the *extent* of basic location $\mathcal{P}(x)$ as opposed to the *scope* of the identifier x . Is $\hat{C}$ restricted on leaving the block, at the same time as $\mathcal{P}$ is restored to its previous condition? Here the natural answer appears to be that it is; the expression would then not have a side-effect. However, below we will come across examples (expressions producing '=' functions) where the other choice seems to be natural (although unfamiliar to implement; storage for local variables would in general be 'off-stack' and subject to more sophisticated garbage-collection problems). We will therefore consider basic locations here to be 'immortal', in that $\hat{C}$ is not altered on leaving blocks. (This is contrary to the current interpretation of CPL; to obtain declarations legal in the current sense, it is necessary to replace our let $I_1=E_1$ below by let $I_1=\text{Store}[E_1]$, Store being a basic function to obtain 'off-stack' storage). This interpretation has the disadvantage that in the terms of Section 2, any expression whose evaluation calls for local variables must be considered to have a side-effect, strictly speaking.

Functions and Routines. A formal parameter list has the form of a (possibly empty) sequence of identifiers separated by commas, each identifier possibly preceded by indicators of type and 'mode' (value or reference). The type and mode of a formal parameter are obtained by searching leftwards for type and mode indicators from its occurrence in this list. If there is no mode indicator to the left, the mode is value. We omit type and mode indicators in the forms below for explaining function and routine declarations.

Functions and routines are intended as objects representing evaluation and execution processes. Routines resemble very closely ALGOL 60 procedures, with the exception that 'call by reference' replaces the concept of 'call by name' (and is different). '=' functions resemble ALGOL 60 type procedures, only the body is an expression, and not a block. Free variables of these processes are called by reference, and in this respect they resemble ALGOL 60 procedures. '=' functions, however, call their free variables by value, and these values are 'frozen' at the time of definition of the function. This distinction turns out to be a crucial one.

(i) let $I_1[I_2, I_3, \dots, I_n]=E_x$

is a constant initialization of I_1 to r , where r is an object with the following property: let $\langle \mathcal{P}_0, \hat{C}_0 \rangle$ be the reduced state on obeying the declaration, then in reduced state $\langle \mathcal{P}_i, \hat{C}_i \rangle$, if $\mathcal{R}_i(E_1)=r$, then $\mathcal{L}_i(E_1[E_2, E_3, \dots, E_n]) = \mathcal{L}_x(E_x)$, where $\langle \mathcal{L}_x, C_x \rangle$ is the state determined by the reduced state $\langle \mathcal{P}_x, \hat{C}_x \rangle$, with $\hat{C}_x = \hat{C}_i$ (possibly modified by side-effects of $E_1, E_2, \dots, E_n$), and $\mathcal{L}_x$ is obtained by constant initialization of each 'free identifier' I_x of E_x (identifiers in E_x which are not formal parameters, and not subject to declaration within E_x) to $\mathcal{R}_0(I_x)$, and initialization by reference or value of each formal parameter $I_j, 2 \leq j \leq n$ to $\mathcal{L}_i(E_j)$ or $\mathcal{R}_i(E_j)$, depending on the mode of I_j .

(ii) let $I_1[I_2, I_3, \dots, I_n] \equiv E_x$

has the same effect as in (i), with the exception that each free identifier I_x of E_x is initialized by reference to $\mathcal{P}_0(I_x)$.

(iii) let $I_1[I_2, I_3, \dots, I_n]$ be K

where K is a command, causes constant initialization of I_1 to r , where the effect of any command, in reduced state $\langle \mathcal{L}_i, C_i \rangle$,

$E_1[E_2, E_3, \dots, E_n]$

with $\mathcal{R}_i(E_1)=r$ is obtained by executing K with $\langle \mathcal{L}_x, C_x \rangle$ as in (ii).

(iv) the insertion of variable after let modifies the effect of (i)–(iii) so as to initialize I_1 by value.

Load-Update Pairs. There is a 'basic function' LUP , taking as parameters a parameterless function and a routine with one parameter, such that, if $\mathcal{L}_0(LUP[E_1, E_2])=l$, then l has the following properties in state $\langle \mathcal{L}_i, C_i \rangle$

(i) $C_i(l) = \mathcal{R}_i(I_1[])$, assuming $\mathcal{R}_i(I_1) = \mathcal{R}_0(E_1)$

(ii) the effect of $E_3 := E_4$ with $\mathcal{L}_i(E_3)=l$

is the same as that of $I_2[E_4]$, assuming $\mathcal{R}_i(I_2) = \mathcal{R}_0(E_2)$ (the assumptions about I_1, I_2 may be eliminated, using suitable initializations of these identifiers.)

Remarks. (1) In strict CPL, LUP is not a basic function; there is a special syntactic device for obtaining load-update pairs. In terms of this, LUP can be defined for a particular type, say *real*. The definition is

let function LUP [real function f , routine R]

be § load $f[]$

update R [rhs] §

(2) The function LUP provides a way to construct non-basic locations. We can define a function *Sign*, for example, by

let *Sign*[reference real x] = reference of

§ let $f[] \equiv (x \geq 0 \rightarrow '+', '-')$

let R [character c] be

§ $x := (c = '+' \rightarrow Mod[x], -Mod[x])$ §

result is $LUP[f, R]$ §

after which the sequence

let $a, b = 1, -2$

$Sign[a] := Sign[b]$

has the effect of setting a to be -1 . Note that $\mathcal{L}(Sign[a])$ is a 'sublocation' of $\mathcal{L}(a)$, in the sense of 2.3. The reference of construction is described below.

(3) The characteristic property 2.2.3 of locations is not guaranteed for load-update pairs, and this device can therefore be misused. The intention is, however, that 2.2.3 should hold, although this general property of f, R (that after $R(E_1)$, $\mathcal{R}_1(f[]) = \mathcal{R}_0(E_1)$) is recursively undecidable, in general. Note that this property holds for values of the function $Sign$, and for the *LUPs* given below.

Result Clauses. The symbols **value of**, **reference of** form expressions from blocks containing occurrences of commands of the form

result is E_1

The L -value of such an expression is obtained by executing the block until a command of the above form is reached in state $\langle \mathcal{L}_i, C_i \rangle$, say, and is either a constant location l with $C(l) = \mathcal{R}_i(E_1)$, or the location $\mathcal{L}_i(E_1)$, depending on whether **value of** or **reference of** is used.

Other features. A detailed description of CPL is to be found in *CPL Working Papers* (1966). In particular, that document should be referred to for rules concerning 'types' of expressions.

Identity. The boolean expression $E_1 = E_2$ has the interpretation that

$$\mathcal{R}(E_1 = E_2) = \mathcal{R}(\text{true}) \text{ iff } \mathcal{R}(E_1) = \mathcal{R}(E_2).$$

It is necessary to specify rules for the latter identity. These are the natural ones on objects of numerical or logical types, but require definition for functions, routines, and also for locations.

(i) *Functions and Routines.* The weakest relation satisfying 2.2.2 can be shown to be recursively undecidable for functions and routines; any decidable relation satisfying 2.2.2 must in practice depend on the text associated with the object, in some way. The rule adopted in CPL is as follows, for functions and routines produced by the forms of declaration discussed above:

Let r_1, r_2 be the two objects involved, then $r_1 = r_2$ iff

- (a) r_1, r_2 resulted from the same declaration in the text of the CPL program being run (this is to say that every such function or routine R -value is associated with some point in the text of a program).
- (b) Suppose r_1, r_2 resulted from obeying the declaration in states $\langle \mathcal{L}_1, C_1 \rangle, \langle \mathcal{L}_2, C_2 \rangle$; then, for each free identifier I_x of the body of the declaration either $\mathcal{R}_1(I_x) = \mathcal{R}_2(I_x)$, in the case of '=' function declarations, or $\mathcal{L}_1(I_x) = \mathcal{L}_2(I_x)$, in the case of '≡' function

declarations and routine declarations. (Note that recursive functions and routines are not discussed here; in these cases it is necessary to test the isomorphism of the two graphs obtained from r_1, r_2 by following chains of free variable connections.)

(ii) *Locations*. Identity between basic locations is a primitive notion (i.e. very strong).

For constant locations, $l_1 = l_2$ iff $C(l_1) = C(l_2)$.

For *LUPs*, suppose l_1, l_2 are formed from functions r_1, r_2 and routines s_1, s_2 . Then $l_1 = l_2$ iff $r_1 = r_2$ and $s_1 = s_2$.

These definitions of *R*- and *L*-identity satisfy 2.2.1, 2.2.2.

3.2 Pairs

The representation of pairs in terms of functions, routines and *LUPs* is now quite straightforward. We will define versions of the functions *Pair*, *H* and *T* to replace the basic functions assumed in Section 1. This will provide a representation of pairs as real functions of one parameter, the parameter being a real function of two real parameters. These representations will have the properties of the interpretations (A), (B) respectively, of Section 1, under assignment and identity.

We need two auxiliary functions:

```
let First[reference real x, y] = x
let Second[reference real x, y] = y
```

The definitions for *Pair*, *H*, *T* are then the following:

(A) let *Pair*[real x, y] = value of § let *f*[real function *g*] = *g*[x, y]
result is *f* §

```
let H[reference real function f] = reference of
    § let h[ ] ≡ f[First]
    let R[real x] be
    § f := Pair[x, f[Second]] §
    result is LUP[h, R] §
```

```
let T[reference real function f] = reference of
    § let h[ ] ≡ f[Second]
    let R[real y] be
    § f := Pair[f[First], y] §
    result is LUP[h, R] §
```

(B) let *Pair*[real x, y] = value of
§ let *x1* = *x*
let *y1* = *y*
let *f*[real function *g*] ≡ *g*[*x1*, *y1*]
result is *f* §

```
let H[real function f] = f[First]
let T[real function f] = f[Second]
```

The function produced by *Pair* in (A) is a '=' function, so that 2.3.4A holds for objects produced by a particular occurrence of this definition of *Pair*, according to the rules for function identity given above. 2.3.5A holds, since h, R in H take f by reference only, so that $\mathcal{L}(H[f])$ is unaltered by assignments to f . Also 2.3.6 (iv) is satisfied, since the parameter f in H is taken by reference.

On the other hand, *Pair* in (B) produces a ' $\equiv$ ' function, and 2.3.4B holds. $\mathcal{L}(H[f])$ depends only on $\mathcal{R}(f)$, therefore 2.3.5B (i) holds, and $\mathcal{L}(H[f])$ is basic, and disjoint from $\mathcal{L}(f)$, so 2.3.5B (ii) and 2.3.6 (iii) hold.

Finally, the programs of Section 1 produce the results indicated for interpretations (A) and (B), using these representations (the reader should try hand-simulating these, if he is unfamiliar with the concepts introduced in 3.1).

Other representations suggest themselves, for example the representation (C) obtained by taking *Pair* as in (B), and H, T as in (A). This has some of the properties one requires, but does not satisfy 2.3.4A or 2.3.4B. In fact, the condition for identity lies strictly between these two conditions. The opposite case, with *Pair* as in (A), H, T as in (B) does not permit assignments to components.

A 'mixed' representation is obtained as follows

```
(D) let Pair1 [reference real x,y]=value of § let f[real function g]≡
                                     g[x, y] §
let Pair [real x, y]=value of § let constant x1=x
                               let y1=y
                               result is Pair1 [x1, y1] §
let H [reference real function f]=reference of
    § let h[ ]≡f[First]
    let R[real x] be
    § let constant x1=x
    f:=Pair1 [x1, f[Second]] §
    result is LUP[h,R] §
```

let $T[\text{real function } f]=f[\text{Second}]$

This representation has the property that

$$\mathcal{R}(f)=\mathcal{R}(g) \text{ iff } \mathcal{R}(H[f])=\mathcal{R}(H[g]) \text{ and } \mathcal{L}(T[f])=\mathcal{L}(T[g])$$

and the mixture of (A) and (B) properties pointed out in 2.3.

3.3 Other Data Structures

List Structure. In this section we will indicate how (A)- and (B)-like representations of more complex data structures can be obtained. These representations may be considered not quite so satisfactory as those of 3.2, since it will be necessary to invoke items of type *general* in CPL. Such items are intended to carry a type indication with them dynamically. There is a basic function *Type* which takes one parameter of type *general* and has as its value

the current type (other than **general**, but possibly **general function**) of that parameter. We make the following assumptions:

- (i) The symbol **general** replaces the symbol **real** throughout the definitions of 3.2.
- (ii) The transfer function *General*[*x*], which converts its parameter to type **general**, is inserted around all actual parameter expressions of functions and routines in 3.2 and below.
- (iii) The definitions below are subject to the declaration

prefer general

which sets the type to be **general** for formal parameters whose types are not otherwise specified.

Note. The representations given below are idealistic; type **general** has not been implemented on CPL compilers. More satisfactory facilities for declaring data structure types will be provided in the final version of CPL. The problem of specifying such types is an interesting one, but is rather a different problem, and will not be discussed here.

List processing functions can be represented in the following way:

```
let Cons=Pair; let Car=H; let Cdr=T;
let Atom [x]=(Type [x]=string)
let NIL [g]=Error [ ]           || causes Error if called.
let Null[f]=(f=NIL)
```

With either (A) or (B) representations of *Pair*, *H*, *T*, these are essentially the functions of 'pure' LISP (McCarthy, 1962), although without property lists for atoms. Thus

Cons['A',*Cons*[*Cons*['B',*Cons*['C',NIL]],NIL]]

is a representation of the *S*-expression

(*A*, (*B*, *C*)).

However the interpretation of $a=b$ depends on whether (A) or (B) is chosen. With representation (A) $a=b$ has the same interpretation as the LISP function *equal*[*a*; *b*]. With representation (B) $a=b$ is the *eq*[*a*; *b*] of LISP.

Functions from 'impure' LISP can be added along the following lines:

```
let Rplaca[reference a,b]=reference of § Car[a]:=b;
                                result is a §
```

and so on (the symbol *reference* is unnecessary for the (B) representation).

The function *nconc* of LISP can be represented very neatly. We need a recursive function definition (whose intention should be clear, though not specified in 3.1).

```
let recursive End[reference a]=(Null[a]→a, End[Cdr[a]])
let Nconc[reference a,b]=reference of § End[a]:=b
                                result is a §
```

As might be expected, with representation (B) these correspond to the functions of LISP with the same names. They are also meaningful with representation (A). In this case

$$\text{Car}[a] := b$$

has the same effect as

$$a := \text{Cons}[b, \text{Cdr}[a]],$$

and

$$\text{End}[a] := b$$

the same effect as

$$a := \text{Append}[a, b]$$

with *Append* as defined in LISP. Notice that with this representation, $\mathcal{L}(\text{End}[a])$ is a *LUP* which, in effect, specifies a route back through the list structure to a basic location $\mathcal{L}(f)$, where f is the free identifier of the routine R . In this *LUP* is a further *LUP* specifying the previous *Cdr*, and this *LUP* contains another 'incarnation' of R for which $\mathcal{L}(f)$ specifies the previous *Cdr*; and so on, back to an $\mathcal{L}(f)$ which is a basic location. If a is not null, then the effect is just the same for

$$\text{End}[\text{Cdr}[a]] := b.$$

Representation (A) therefore provides a version of list processing which, in effect, is incapable of exploiting 'sharing' properties of list structure (in fact the machine storage for list structure is shared, assuming that implementations of functions do not copy 'free variable lists', which is the intended method of implementing function assignments; 'pure' LISP functions, with parameters by value, would run comparably well, or badly, on either representation).

Vectors. To represent vectors it is necessary to avoid introducing 'variadic' functions (functions taking an indeterminate number of parameters), since these are not definable in CPL, although the basic system functions *NewVector*, *FormVector* are 'variadic'. We do this here by defining a constructor function *Vector*, which takes as parameters a function f to generate values for elements, and the subscript upper and lower bounds, m and n . *Vector* and *Sub* are recursive functions, again.

let recursive *Vector*[f , index m , n]

$= (m > n \rightarrow \text{NIL}, \text{Pair}[\text{Pair}[m, f[m]], \text{Vector}[f, m+1, n]])$

let recursive *Sub*[reference A , value index i]

$= (A = \text{NIL} \rightarrow \text{Error} [],$

$H[H[A]] = \text{General}[i] \rightarrow T[H[A]],$

$\text{Sub}[T[A], i])$

|| subscript out of range.

$Sub[A, i]$ is then in effect the element Ai of the vector A . This representation, as an 'association list', is of course ludicrously inefficient, but does have the formal properties expected of vectors. With representation (A) of $Pair, H, T$, this corresponds to the interpretation of vectors in ALGOL, PL/I and other languages. CPL has (B)-type vectors. Note that, as with *nconc*, assignment to a vector element causes copying of the path through the list back to a basic location, with representation (A) (this is analogous to the technique of 'aliasing' sometimes used in implementing (A)-type arrays so that their storage can be shared, although in that technique one copies back only as far as there are other references to locations on the route).

Matrices. These can be represented as vectors of vectors; $f[i, j]$ is taken as a function to generate values for A_{ij} . The matrix bounds are $(m, n), (p, q)$.

```
let Matrix[f, index m,n,p,q]
    =value of § let g[index i]=value of § let h[index j]=f[i, j]
                                     result is Vector[h,p,q] §
                                     result is Vector[g,m,n] §
let Sub2[reference A, value index i,j]=Sub[Sub[A,i],j]
```

Some ingenuity is required to extend this method to defining a general constructor *Array*, with a parameter specifying the dimensionality required; and to defining a corresponding general subscripting function. This task is left as a diversion for the reader.

4. COMMENTS AND CONCLUSIONS

Section 2.3 indicated, though not conclusively, an essential dichotomy between interpretations of a very simple data structure. This dichotomy was based on the relationship that held between identity of structure and identity of components. Since it appeared sensible to talk about two principal sorts of identity between computational objects, there were two corresponding identity rules between structures (and various 'mixed' interpretations, which treated different components in different ways). The choice was seen to determine other properties which might be relevant to a complete characterization of the properties of the data structure, particularly under assignment commands, which at this stage cause all the trouble. Whether a 'complete characterization' is obtainable just in the terms of 2.2, and what its nature would be, is not clear. The counterexamples obtained suggest that the notion of 'location' requires further assumptions to be made, in order to obtain a cleaner theory, and these assumptions remain to be formulated.

In Section 3 it was shown how one could define interpretations along the lines of those singled out in 2.3, in terms of various processes in CPL. However sophisticated these processes may be to implement, they also appear to me to be logically more primitive than the structures which they are used to represent. If one looks behind the scenes at the implementation of such objects, one sees that what goes on in 'evaluating' a process (obeying a

function or routine definition) is in effect a data structure construction operation; space is reserved for a 'free variable list'. The structures constructed in this way can be regarded as finite directed graphs (trees if there are no recursive functions) with processes at nodes communicating with (calling or returning a result to) adjacent nodes. A call to a process can be regarded as a selection operation, with function and parameter reversed (recall that one pair of definitions for H , T was $H[f] = f[\text{First}]$, $T[f] = f[\text{Second}]$). This suggests to me that, if one is looking for a 'most general' form of data structure, the acceptance of such 'processes' as fully manipulable computational objects may be the crucial step.

REFERENCES

- CPL *Working Papers* (1966), University of London Institute of Computer Science.
 McCarthy, J. et. al. (1962), *LISP 1.5 Programmer's Manual*. Cambridge, Mass: MIT Computation Center.
 Naur, P., ed. (1963), Revised Report on the Algorithmic Language ALGOL 60. *Communs Ass. comput. Mach.*, 6, 1-23.
 Strachey, C. (1966), Towards a Formal Semantics. *Formal Language Description Languages*, pp. 198-216 (ed. Steel, T.B.). Amsterdam: North Holland Press.
 Strachey, C. (1968), Fundamental Concepts in Programming Languages. *Proceedings of 1967 NATO Summer School. Copenhagen*.

Writing Search Algorithms in Functional Form

R. M. Burstall

Department of Machine Intelligence and Perception
University of Edinburgh

1. INTRODUCTION

A great many machine intelligence programs perform some sort of search, and a number of investigators have discussed techniques of searching trees.

One well known method, which we will call 'depth-first' search is particularly attractive because it is very easy to program by recursion. The resulting programs are short and rather transparent. It has the disadvantage however that it is by no means the most efficient sequence of searching a tree in many cases. Indeed if the tree is infinite it may cause the search to go right out of control. The attractiveness of 'depth-first' recursive searching was pointed out a long time ago by Newell, Shaw, and Simon, but they also remarked on its pitfalls. More recently Slagle in his 'Dedecom' program (*see* Slagle, 1965) used a 'depth-first' recursive search for convenience, but found that it led to severe limitations on the problem-solving ability of his program. A number of examples of problems amenable to tree search methods are given by Golomb and Baumert (1965).

Most machine intelligence tasks are difficult enough to make ease and transparency of programming an important consideration. Hence it would be interesting to discover a programming technique which would preserve the simple program structure associated with recursive depth-first search while allowing a more efficient and flexible sequence of searching. This paper puts forward such a technique. The search method in question which we will call 'controlled' search is that used by the Doran and Michie 'Graph Traverser' program (*see* Doran, 1967, 1968; Doran and Michie, 1966; Michie, 1967).

We first consider different methods of searching, quite apart from the question of programming. We then describe the recursive programming technique for depth-first searching and discuss the possibility of introducing a new form of expression into programming languages to enable us to treat

more general searches without restructuring the program. It turns out that this new form of expression is not really a special extension of programming languages but can be defined in terms of Landin's generalized jump operator. This is much more satisfactory.

2. SEARCH TECHNIQUES

We will consider the problem of searching a tree to find a node with a given property. If it is necessary to search the whole tree, e.g. to find the node which is optimal in some sense, the sequence in which the search is carried out is unimportant. If however we are content to discover just one node with the given property the sequence in which the nodes are searched may be of great interest. We will consider three possibilities and illustrate them using the following simple problem.

The tree is defined in terms of two functions of integers

$$\begin{aligned}f(n) &= n^2 - 2n + 3 \\g(n) &= 2n^2 - 5n + 4\end{aligned}$$

The problem is to find a value of n with some property p by repeatedly applying either f or g to a starting value n_0 . For example

$$\begin{aligned}n_0 &= 0 \\p(n) &= 30 < n \leq 40\end{aligned}$$

Part of the corresponding search tree is shown in figure 1.

2.1. Depth-first search

To search a tree starting from a node n by depth-first search:

1. Check whether n itself has the required property. If so the search terminates successfully;
2. If n has no successors the search terminates unsuccessfully;
3. Take each successor of n in some specified sequence and perform a depth-first search starting from that node. Continue until one of the successors leads to a successful search.

The sequence of search for the problem stated above is shown in figure 2, making the assumption that the tree is restricted to a finite one by ignoring all nodes greater than 500, and taking $f(n)$ before $g(n)$.

If this restriction is removed the search fails completely as only the topmost branch of the tree will ever be explored.

The effectiveness of a depth-first search can be improved by choosing a suitable sequence for taking the successors of a given node, e.g. smallest first. But this will not remedy the inability of a depth-first search of an infinite tree to recover from a single wrong decision.

$f(n) = n^2 - 2n + 3$
 $g(n) = 2n^2 - 5n + 4$
 $p(n) = 30 < n \leq 40$
 $n = 38$ satisfies $p(n)$

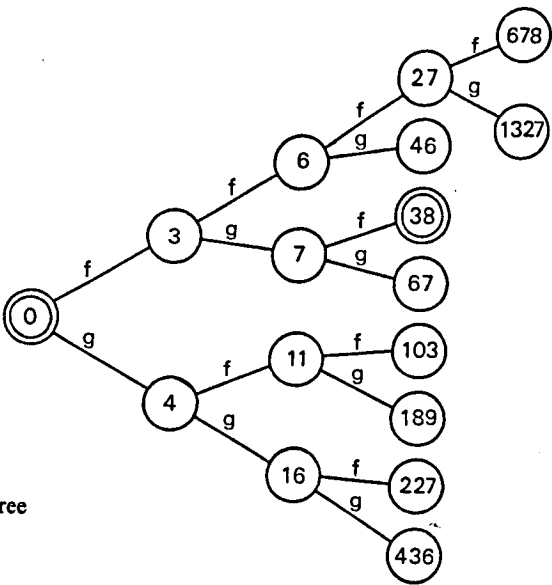


Figure 1. The search tree

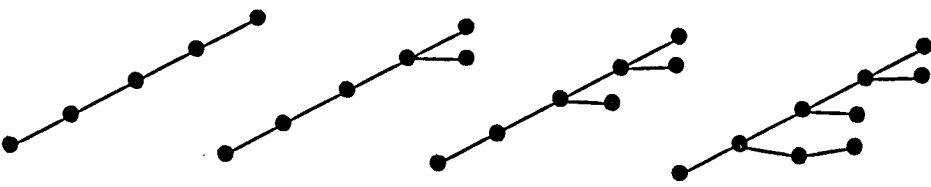


Figure 2. Stages in depth-first search

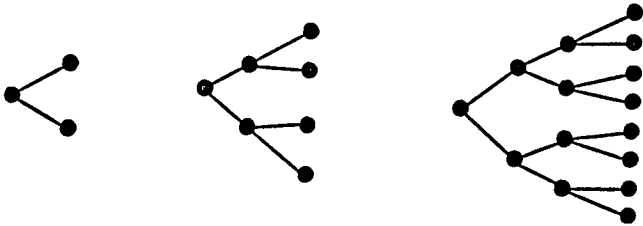


Figure 3. Stages in parallel search

2.2. Parallel search

This is in a sense the opposite of depth-first search. To search the trees starting from a set of nodes N by parallel search:

1. If N contains a node with the required property the search terminates successfully.
2. If N is empty or no node in N has any successors the search terminates unsuccessfully.
3. Let N' be the set of all nodes which are successors of a node in N (taken together for all nodes in N). Search the trees starting from N' by parallel search.

The sequence of search for the example problem is shown in figure 3. It avoids the difficulties associated with infinite branches but is very rigid and does not take advantage of any information that one branch is more promising than another, i.e. it leaves no scope for heuristics.

2.3. Search controlled by a reluctance function

In this method of search we may think of a 'frontier' of those nodes whose successors have not been examined. This frontier is extended by choosing any node in it and examining its successors. The node chosen may be that which has the minimum value of some function called a 'reluctance function', normally chosen on heuristic grounds. (Doran and Michie call this an 'evaluation function' but we use the word 'evaluate' here in another sense which would lead to confusion.)

This method of search has been extensively discussed by Doran and Michie who used it in their program called the 'Graph Traverser'. They have shown that given a suitable reluctance function it is an effective method of tackling a number of problems.

The sequence is illustrated in figure 4. The main point to notice is that the next node whose successors are examined is not necessarily one of those produced at the previous move, i.e. the frontier will be pushed forward for a while in one region, but if the reluctance function indicates that further advances here are not as profitable as had been anticipated some other part of the frontier may be extended.

It is easy to see that by choosing a suitable evaluation function parallel search and depth-first search can be produced as special cases of search controlled by a reluctance function. Specifically:

- (i) *Parallel search*. The reluctance function is the distance (number of arcs) between the starting node and the given node. The rule that the frontier node with least value of the reluctance function is taken first causes the frontier to advance one layer at a time (see figure 5).
- (ii) *Depth-first search*. Taking a tree with binary branches for simplicity the reluctance function assigns to each node a binary fraction whose digits are (left to right) 0 or 1 according to the choices made in obtaining that node from the original node (see figure 6).

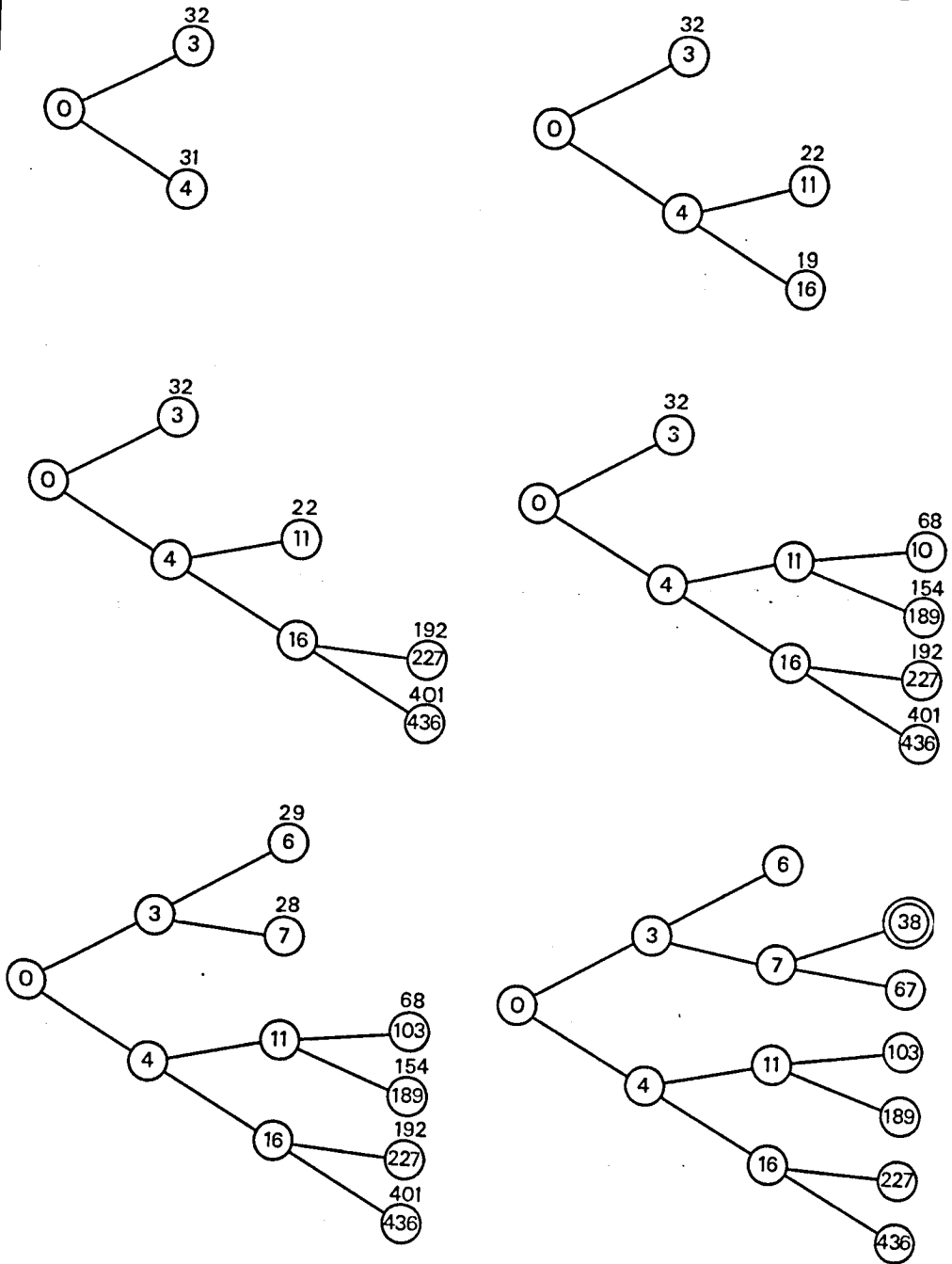


Figure 4. Controlled search. Reluctance function: $r(n) = |n - 35|$. Values of reluctance are written above the frontier nodes

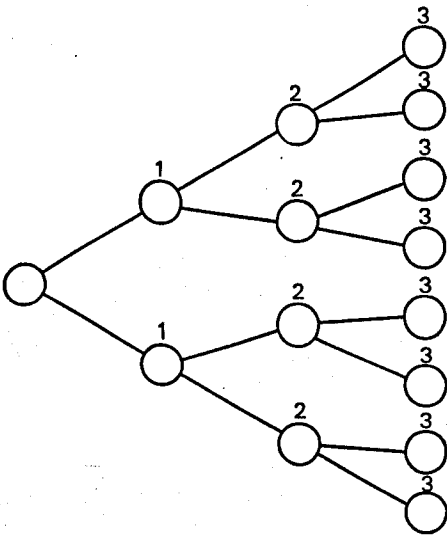


Figure 5. Values of reluctance function to produce parallel search

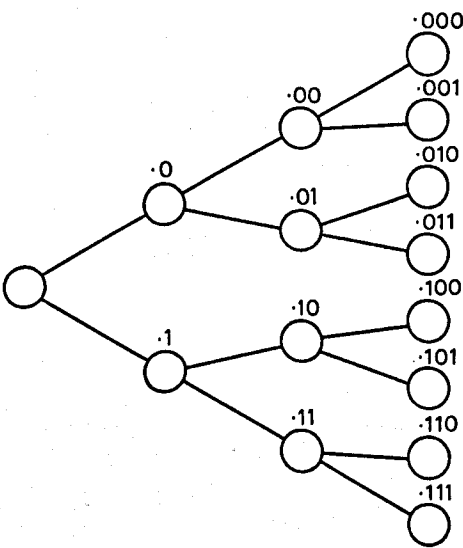


Figure 6. Values of reluctance function to produce depth-first search

3. PROGRAMMING

We will write programs in functional notation using the ISWIM language (Landin, 1966). The only point which needs explanation is that 'let $X = D$ ' denotes a definition qualifying the expression which follows, otherwise the notation should be fairly obvious even without an acquaintance with ISWIM. We also use the conventional notation for sets and operations on them.

We continue to use the previous example—this involves no real lack of generality, since the particular functions f and g are irrelevant, and the cases of more than two successors of a node or of more complicated trees follow naturally from our discussion.

We introduce one extra predicate *terminal* which is to be true if n has no successors, i.e. f and g are not to be applied to it.

Suppose first that we wish to obtain the set of all nodes which satisfy the predicate p . This set is computed by *pnodes* where

```
recursive pnodes (n) = if terminal (n) then nil
                      else if p(n)      then {n}
                      else pnodes(f(n)) ∪ pnodes(g(n))
```

We have assumed for convenience that if a node n satisfies p we are not interested in the successors of n .

If we wish to obtain just one node satisfying p it is found using depth-first search by *pnode*, where

```
recursive pnode(n) = if terminal(n) then nil
                    else if p(n)      then {n}
                    else let pfn = pnode(f(n))
                        if null(pfn) then pnode(g(n)) else pfn
```

It is this simple form of recursive program which makes depth-first search attractive in spite of its pitfalls. Its power may be seen by a slightly more complicated example where we require as a result the cost of getting to a node with property p , the cost of going from n to $f(n)$ being $cf(n)$ and that of going from n to $g(n)$ being $cg(n)$.

```
recursive cost(n) = if terminal(x) then ∞
                   else if p(n)      then 0
                   else let costfn = cost(f(n))
                       if costfn = ∞ then cg(n) + pcost(g(n))
                       else cf(n) + costfn
```

Here some work (adding up costs) is done on exit from the recursion.

It seems at first sight that in order to program the more sophisticated controlled search method a completely different program structure is necessary and the elegant recursive approach must be abandoned. Doran and Michie in their 'Graph Traverser' use an approach which in its essentials may be expressed thus:

Suppose that $\phi(n) = \{f(n), g(n)\}$, i.e. the set of successors of n , and that $r(n)$ is the reluctance function for a node, saying how undesirable it is.

```

pnode(n) =
  let  $N = \{n\}$ 
  loop: let  $nmin = \text{least}(r, N)$ ;
        if  $\exists n \in \phi(nmin) [p(n)]$  then goto out;
         $N := N - \{nmin\} \cup \phi(nmin)$ ; goto loop;
  out: result any( $p, \phi(nmin)$ )
    
```

Here $\text{least}(r, \text{nodes})$ finds the member of N with least value of r and any($p, \phi(nmin)$) finds any member of $\phi(nmin)$ satisfying p .

This could of course have been written recursively, although it is essentially an iterative algorithm. Thus:

```

recursive pnode(N) =
  let  $nmin = \text{least}(r, N)$ ;
  if  $\exists n \in \phi(nmin) [p(n)]$  then any( $p, \phi(nmin)$ )
  else pnode( $N - \{nmin\} \cup \phi(nmin)$ )
    
```

The depth of recursion now corresponds to the total number of nodes examined and not as previously to the distance along the branch from the start to the current node. The problem above of costing the path from the start to the solution node is no longer so easy. It would require that a back pointer be passed on explicitly with each node so that when the solution is found a retrace can be done and the cost can be worked out. This is an important limitation of this way of programming controlled search: dealing with the answer when found needs special programming.

4. ALTERNATIVE EXPRESSIONS

The point of this paper is to show how the controlled search can be programmed in a manner very similar to the recursive method for depth-first search. To do this we may introduce a new kind of expression called an 'alternative expression' which could be thought of as an extension of any programming language of the ALGOL family. This kind of expression involves a special evaluation rule which we will explain informally. In the next section we will show that the rules for alternative expressions can be made quite precise in terms of 'J' operator, a form of generalized jump (Landin, 1966). Indeed if our programming language is equipped with this operator or an equivalent generalized jump facility, we do not need to introduce alternative expressions as a new feature at all since a function can be defined which has the same effect.

Consider the expression

$$h(x+1, y-2)$$

To evaluate it we must evaluate BOTH $x+1$ AND $y-2$.

But consider the expression

$$h(x) < 3 \text{ and } k(x) > 0$$

To evaluate this it is sufficient to evaluate EITHER $h(x) < 3$ OR to evaluate $k(x) > 0$ provided that the one evaluated has truth value false (if not, the other expression must also be evaluated).

Now suppose that $fpos(n)$ is the smallest number greater than n which has a property p and $fneg(n)$ is the largest number less than or equal to n which has p . Then if we just want an expression whose value is *some* number which has p we might write

$fpos(0)$ alternatively $fneg(0)$ (alternatively is a new basic symbol)

Here it would be sufficient to evaluate EITHER $fpos(0)$ OR $fneg(0)$.

To evaluate this expression we might get the computer to work on both in turn taking as result whichever it managed to evaluate first.

Evaluating $fpos(0)$ might well involve evaluating another such alternative expression and we would think of the possible calculation as a tree of which it would be sufficient to evaluate completely just one branch. If we do not wish the computer to use its discretion as to which branch to evaluate we must provide some extra information about each branch, i.e. reverting to our controlled search idea we must provide the value of the evaluation function for that branch. We will call this an alternative expression.

$$\begin{aligned} \langle \text{alternative expression} \rangle &::= \langle \text{expression} \rangle \text{ reluc } \langle \text{real expression} \rangle \\ &\quad \text{alternatively} \\ &\quad \langle \text{expression} \rangle \text{ reluc } \langle \text{real expression} \rangle \end{aligned}$$

$fpos(0) \text{ reluc } 5 \text{ alternatively } fneg(0) \text{ reluc } 10$

We now program a search controlled by an evaluation function in a manner corresponding exactly to a depth-first search (we use r for the reluctance function).

```
recursive pnode(n)=if p(n) then {n}
                  else (pnode(f(n)) reluc r(f(n))
                        alternatively pnode(g(n)) reluc r(g(n)))
```

The idea here is that in the recursive evaluation of $pnode$ a lot of alternative expressions will be activated each with two component expressions. The evaluation mechanism is to consider all the component expressions which have been produced by activation of alternative expressions but have not yet been evaluated themselves, and evaluate the one which has least value of the reluctance. This means that the evaluation mechanism has somehow to keep a list of component expressions to be evaluated, with suitable information about their environments, and a link to the alternative expression to which

they belong. We assume at the moment that there is only one such list for a particular program.

One or two remarks are worth making. There is no reason why a program should not contain more than one alternative expression. In this case the tree of possible evaluations will not be homogeneous—it will have nodes, some of which correspond to activations of one alternative expression, some to activations of other alternative expressions.

Likewise there is no reason why an alternative expression should have just two components. We may allow more or even less than two. We may easily want the number of components to be determined dynamically.

We have not included in our last definition of *pnode* the case where *n* is a terminal node. One way of dealing with this would be to insert there an alternative expression with no components i.e. the other branches of the tree already in existence are to be followed but this node does not give rise to any new branches.

It is a bad principle, however, to extend a programming language to give some new features unless we are quite sure that the extension is of sufficient generality. Can we make use of some more generally desirable feature of programming languages and avoid the need for 'alternative expressions'?

5. USE OF THE *J* OPERATOR

In another paper presented at this symposium* Landin puts forward a jump operator '*J*' which enables his functional programming language ISWIM to handle departures from the normal sequence of evaluation, such as error exits. The case of a search which is to be terminated as soon as it is successful is closely analogous to an error exit, and we now show how *J* may be used to program 'alternative expressions'. This means that we withdraw the above proposal for alternative expressions (which was merely an explanatory device) and offer instead an equivalent library function called *oneof*.

We keep the component expressions to be evaluated together with associated values of their reluctance function on a list called jobs. They are ranked in ascending order of the values of their reluctance function.

How are we to represent the component expressions? If we simply write down the expressions they will get evaluated straight away, which is not our intention. What we need is a function which when applied to a list of jobs will produce the required value. It is this function which is stored on the jobs list. Before it is stored however, we must apply *J* to it. This means that if it ever gets applied it will return its result as the result of the '*oneof*' expression of which it is a component.

Thus the task of the function *oneof* is to add to a list of jobs supplied to it as a parameter the two component functions supplied to it with their associated values, but only after applying *J* to them. It must then take the first component function off the jobs list and apply it to the rest of the jobs list.

* See note at end of this paper.

The function *oneof* is as follows:

```

oneof(f,rf,g,rg,jobs)=
  let jobs=addjob(J(f),rf,jobs);
  let jobs=addjob(J(g),rg,jobs);
  let ((jh,rh),jobs)=next(jobs); N.B. next produces the head and tail
    jh(jobs)                      of a list.
  where recursive addjob(jf,rf,jobs)=if null (jobs) then ((jf,rf)) else
    let ((jh,rh),jobs)=next(jobs);
    if rf≤rh then (jf,rf)::(jh,rh)::jobs      N.B. :: is an infix
    else (jh,rh)::addjob(jf,rf,jobs) operator for 'cons'.

```

Given *oneof* we can now write *pnode* again very easily

```

recursive pnode(n)=pnoden
  where pnoden(jobs)=if p(n) then {n}
    else oneof(pnode(f(n)),r(f(n)),
              pnode(g(n)),r(g(n)),jobs)

```

To find a node satisfying *p* starting from a node *n0* we evaluate

pnode(*n0*)(*nil*)

The action is not difficult to grasp once we realize that the functions denoted by *jf*, *rg*, *jh* above always jump back as soon as they are executed to the *oneof* expression which gave them birth, returning their value as the result of that expression.

We have written *oneof* to take two components but there is no difficulty in writing it to take a list of any number of components (we supply a list of function-reluctance pairs and use *maplist*).

The difficulty about terminal nodes can then be dealt with by invoking *oneof* and giving it an empty list of component jobs. This allows existing branches of the tree to be processed further without creating any new ones.

Thus the definition of *oneof* as a library function enables us to carry out controlled searches in an elegant and transparent recursive manner without extending the language with any new form of expression or special evaluation mode. Although our example has been a simple one, much more complex searches may be programmed using the same function *oneof*.

NOTE ON LANDIN'S J OPERATOR

P.J.Landin gave a talk at the Machine Intelligence Workshop about his *J* operator, a generalized jump facility. Unfortunately the written version was not available in time for publication in this volume. Since my paper makes use of this operator, Mr Landin has kindly agreed to my including a short explanation of it here. He hopes to publish a fuller account shortly and the reference may be obtained from him at Queen Mary College, University of

London. Thus, the work described below is due to Landin, but the explanation is mine and he is not responsible for any defects in it.

In a previous paper Landin (1966) introduced the notion of a 'program point'; this is analogous to a function (or ALGOL procedure) but has a non-standard mode of exit. Instead of returning control to the point from which it is called, it returns control to the point from which the function in which it is defined is called. An example in ALGOL format may help.

```
integer procedure f(x); integer x;
    integer program point f1(y); real y; f1:=entier(y×100);
    real procedure g(z); integer z;
        .....
        if trouble >0 then f1(1/z) end; z:=z+1;
        .....
    end of g;
    ...
    v:=g(3/x);
    ...
end;
...
u:=f(0); u:=u+1;
```

Here $f1$ is an integer program point (i.e. an integer procedure with non-standard exit) defined directly within the integer procedure f . Its job is to return an answer which would do as the answer to its parent f in case computing the answer to f ran into some trouble, e.g. during the internal procedure g . Thus, if g calls $f1$ during execution of f , the subsequent statement ' $z:=z+1$ ' is *not* executed, but instead an immediate exit from its parent procedure f takes place, the result of f being $f1(1/z)$, i.e. $entier(1/z \times 100)$. This value is assigned to u and the next statement executed is ' $u:=u+1$ '. Thus the remainder of g and f have been short circuited, with a forced exit returning the result of $f1$ as the result of its parent f .

```
In Iswim we have
let f(x)=
    let program point f1(y)=entier(y×100)
    and g(z)=
        ...
        if trouble >0 then f1(1/z); z:=z+1;
        ...
    v:=g(1/x);
    ...
u:=f(0); u:=u+1;
```

A possible improvement to this device, suggested by Landin in his talk, is to replace $f1$ by an ordinary integer procedure, say $f0$, and have a special

operation J which when applied to $f0$ converts it into a program point. The parent of $f1$ would then be determined by the procedure in whose body the J occurred (i.e. the innermost such procedure). Thus we could write for example

```
let f0(y) = entier(y × 100);
...
let f(x) =
  let f1 = J(f0)
  and f(z) =
    ...
    if trouble > 0 then f1(1/z);
etc.
```

The point is that J incorporates a 'fire escape' leading to the exit point of its parent procedure, and attaches this fire escape to the procedure to which it is applied, for use instead of that procedure's normal exit mechanism.

REFERENCES

- Doran, J.E. (1967), An approach to automatic problem-solving. *Machine Intelligence 1*, pp. 105-23 (eds Collins, N.L. & Michie, D.). Edinburgh: Oliver and Boyd.
- Doran, J.E. (1968), New developments of the Graph Traverser. *Machine Intelligence 2*, pp. 119-35 (eds Dale, E. & Michie, D.). Edinburgh: Oliver and Boyd.
- Doran, J.E. & Michie, D. (1966), Experiments with the Graph Traverser program. *Proc. R. Soc. (A)*, 294, 235-59.
- Golomb, S.W. & Baumert, L.D. (1965), Backtrack programming. *J. Ass. comput. Mach.*, 12, 516-24.
- Landin, P.J. (1966), The next 700 programming languages. *Communs Ass. comput. Mach.*, 9, 157-66.
- Michie, D. (1967), Strategy building with the Graph Traverser. *Machine Intelligence 1*, pp. 105-23 (eds Collins N.L., & Michie, D.) Edinburgh: Oliver and Boyd.
- Slagle, J.R. (1965), Experiments with a deductive question-answering program. *Communs Ass. comput. Mach.*, 8, 792-8.

Assertions: Programs written without specifying Unnecessary Order

J. M. Foster
Computer Research Group
University of Aberdeen

1. INTRODUCTION

This paper describes a language for writing algorithms in which less information about the order in which the individual operations are carried out is given than is normal for programming languages. It is at present partly implemented on an Elliott 4120. Various other systems have been described and suggested (*see* Anderson, 1965; Laski and Buxton, 1962; Markowitz, Hausner, and Karr, 1962; and Opler, 1965).

There are various motives for avoiding the giving of specific information about the order of the component statements of a program. First, it enables transformations to be carried out more easily on the program. This is important both for compilation, where the processes of optimization and of translation into machine code are both transformations into equivalent algorithms, and for artificial intelligence, if we try to improve the computer's representation of its problem. Secondly, there exists a class of problems where the order of the operations cannot be easily specified in advance. This includes the recognition of complex objects, as in syntax analysis, and certain problems of simulation of events occurring in time.

2. ASSERTIONS

The basic elements in the system are not instructions to do something, as are the statements of ALGOL, but assertions about the data, such as $a=b$ or $a+b=c$. The evaluation of a program of assertions is not the obeying in specified sequence of a set of instructions, but an attempt to find data which satisfies the assertions.

For the purposes of exposition, let us consider the ordinary type of syntactic description of a computing language.

$$\langle \text{identifier list} \rangle ::= \langle \text{identifier} \rangle \mid \langle \text{identifier} \rangle, \langle \text{identifier list} \rangle$$

This means that an example of an identifier list may be constructed either from an identifier, or from an identifier followed by a comma followed by an identifier list. Looked at from the point of view of a syntax recognizer, this says that a program for recognizing an identifier list may be made from one which recognizes an identifier or an identifier and then a comma and then an identifier list. This could be written out in some such way as the following:

identifier list = *identifier* or (*identifier andthen comma andthen identifier list*)

Notice that the compound operator *andthen* means both that all the items so connected must be true (*and*) and also that they correspond in a particular sequence to the input (*then*).

The normal relation between *and* and *or*, and the restriction to a particular stream of characters can be removed by introducing parameters.

identifier list(a,b) = *identifier(a,b)* or (*identifier(a,c)* and *comma(c,d)* and *identifier list(d,b)*)

where *identifier list(a,b)* means that from the place *a* (a tape index number) to the place *b* there is an example of an identifier list, and similarly for the other assertions. The identifiers *c* and *d* belong only to these assertions so they might be localized

identifier list(a,b) new *c,d* = *identifier(a,b)* or
(*identifier(a,c)* and *comma(c,d)* and *identifier list(d,b)*)

The intention of the new part of the definition of the assertion is that the assertions in the body of the definition refer to the same new variables *c* and *d*. There need be no implication that the assertions are tested in any particular order.

Before describing how the assertions can be tested in a sensible order, and how ordering can be imposed externally, two examples will be discussed.

Example 1. To write an assertion that between two specified places on an input stream there is an identifier followed by an equals sign followed by an identifier, and that the internal variables which correspond to the textual forms of the identifiers are equal. This can be imagined as part of a simple interpreter.

F(s,u) new *x,y,p,q,z,t* = *tape(s,x,y)* and *tape(y,'=',z)*
and *tape(z,t,u)* and *lookup(x,p)* and *lookup(t,q)* and *p=q*

tape is a primitive assertion that between the places *s* and *y* (in the first example of it) there occurs a textual identifier which is the value of *x*. The primitive, *lookup(x,p)*, is the assertion that the textual identifier which is the value of *x* corresponds with the variable *p* in the environment.

The operator '=' is intended to mean 'is substitutable for'. Thus the definition can be rewritten

F(s,u) new *x,y,p,z,t* = *tape(s,x,y)* and *tape(y,'=',z)*
and *tape(z,t,u)* and *lookup(x,p)* and *lookup(t,p)*

Example 2. To write the assertion that between two specified places on an input stream there is the textual form of an expression and that, when the identifiers occurring in it are made to correspond with internal variables and the expression is evaluated, the resulting value is that of a particular variable. An expression is confined to identifiers and the + sign.

$$E(a,b,c) \text{ new } d,e,f,h,i,j = (\text{tape}(a,d,c) \text{ and } \text{lookup}(d,b)) \\ \text{or } (\text{tape}(a,e,f) \text{ and } \text{tape}(f, '+', i) \text{ and } E(i,j,c) \\ \text{and } \text{lookup}(e,h) \text{ and } \text{plus}(h,j,b))$$

In this definition a and c are the places on the tape, and b is the variable of which the value is the value of the expression. The primitive $\text{plus}(h,j,b)$ asserts that $h+j=b$.

Note that it is possible to carry out the following transformation on the above assertion. This transformation has a similarity to the distributive law, and is one that might add efficiency (by avoiding unnecessary evaluation).

$$E(a,b,c) \text{ new } d,f,h,i,j = \text{tape}(a,d,f) \text{ and } ((\text{lookup}(d,b) \text{ and } f=c)) \\ \text{or } (\text{lookup}(d,h) \text{ and } \text{tape}(f, '+', i) \text{ and } E(i,j,c) \\ \text{and } \text{plus}(h,j,b))$$

3. ORDERING

There are three methods for deducing ordering information for attempting to test assertions, in the system.

First, a number may be associated with an assertion to indicate the cost of attempting it. Clearly it is worth while to test assertions which immediately throw away information like '=' and '+' and there may be other costs known to the programmer from the nature of his problem.

Second, each assertion has a number of pass sets associated with it. There is no point in attempting to satisfy $\text{tape}(a,b,c)$ if none of the variables has a value, but if a has a value or if b and c have values or if a and b and c all have values, then it may be worth while attempting the assertion. In this case the pass sets would be

$$a, b, c \\ a \\ b, c$$

The effect of testing the assertion would be different in each of these cases. If all the variables had values, testing the assertion would be a check on its validity. If it were false, there would be no need to continue testing the other assertions connected to it by the operator **and**. If a alone had a value, then values could be supplied for b and c and the other assertions tested to see if they were consistent with these values. Likewise, if b and c had values, a value could be generated for a .

PROBLEM-ORIENTED LANGUAGES

Suppose that the pass sets for $lookup(x,y)$ are

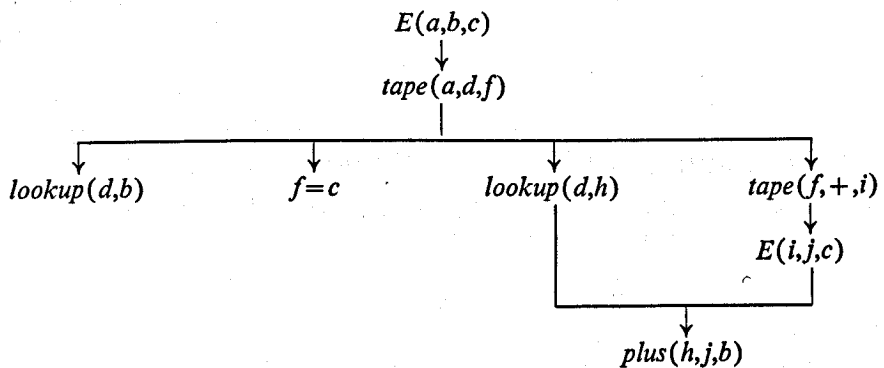
x
 x, y

and those for $plus(p,q,r)$ are

p, q
 p, r
 q, r
 p, q, r

and consider the version of $E(a,b,c)$ given at the end of *Example 2*.

Suppose that the value of a alone is given. Then the tree structure below expresses the necessary ordering. In other cases, such as $E(a,b,c)$ where all the arguments have values, the ordering cannot be expressed in so simple a tree structure.



The third method of ordering is intended to prevent too uneconomical a use of the store. If two disjoint sets of assertions use one identifier each, then these two identifiers can share the same store provided that, as soon as one of the sets of assertions has been started, the other is locked out until the first is completed. However, there need not be any reason to choose one set to test first rather than the other. This corresponds to the locking out of sections of program from one another, which is normally done in interrupt control programs; however, the locking belongs to an identifier rather than to a segment of program.

It may be that the use of this principle of ordering will lead to a situation in which nothing can be done, because every assertion is locked out. For example

$f(xa,ya) \quad g(xa,yb) \quad h(xb,ya) \quad i(xb,yb)$

where xa and xb are sharing a store and so are ya and yb , will lead to an impossible situation. This can be resolved by copying one of the variables, and by making all references to this variable refer to the new copy.

This is precisely what is done in recursion, where the use of a new variable of the same name as an old one leads to the use of a copy of the new item in schemes where a stack is used, and to a copy of the old item in schemes where the program refers to the same store and its contents are pushed down.

REFERENCES

- Anderson, J. P. (1965), Program structures for parallel processing. *Comms Ass. comput. Mach.* **8**, 786-8.
- Laski, J. G. & Buxton, J. N. (1962), Control and simulation language. *Comput. J.*, **5**, 194-9.
- Markowitz, H.M., Hausner B., & Karr, H.W. (1962), *Simscrip: a simulation programming language*. New Jersey: Prentice Hall.
- Opler, A. (1965), A procedure oriented statement to facilitate parallel processing. *Communs Ass. conf. Mach.*, **8**, 306-7.

The Design Philosophy of POP-2

R.J. Popplestone

Department of Machine Intelligence and Perception
University of Edinburgh

INTRODUCTION

POP-2 (Burstall & Popplestone, 1968) represents a fairly far-reaching revision, extension and systematization of the author's POP-1 (Popplestone, 1968). The thoughts expressed here consequently represent a point of view elaborated jointly with my co-designer of POP-2, Dr R. M. Burstall.

AIMS

POP-2 is a language to be implemented on real machines, using modest resources of manpower. An implementation of the language must be possible which permits large problems to be tackled. This implementation must not be too inefficient in its use of machine time, or too profligate in its use of store. The language must also take into account such properties of real machines as overwritable store—that is to say it must not be a purely constructive language: it must allow assignment.

POP-2 handles a large range of structures such as list cells (cf. LISP), functions (cf. CPL) and records (called beads in AED). Efficiency is important here—LISP is perfectly general, but its representation of such structures in list-cells is inefficient in its use of store.

Bearing in mind the need for a wide range of structures, the concepts should be as few and simple as possible. An example of this simplification is that such apparently diverse things as ALGOL procedures, ALGOL arrays, binary operators, and AED 'components' can all be represented in a natural fashion as POP-2 functions. Functions in POP-2 differ in their semantic properties, and in the syntactic properties of their names, but they are all the same in essence: not only are they all handled in the machine as data structures, but they constitute a single class of data structure.

This brings us to the subject of items. Anything which can be the value of a variable is an item. *All items have certain fundamental rights.*

1. All items can be the actual parameters of functions

2. All items can be returned as results of functions
3. All items can be the subject of assignment statements
4. All items can be tested for equality.

Equality does not, however, follow the usual mathematical axioms, e.g. $\text{CONS}(2,3) \neq \text{CONS}(2,3)$, where `CONS` is the LISP list-cell constructor. The impracticability of providing a computed version of mathematical equality is clear if one considers the case of functions. Nevertheless, care is taken in the definition of the language to make sure that the meaning of the equality is clearly defined. It corresponds to the LISP 'EQ', or the concept of equal address in mechanistic terms.

It should be noted where existing languages fail to provide an adequate 'item's charter of rights': ALGOL arrays cannot be returned as results of procedures. AED components cannot act as parameters of procedures. Certain consequences follow from the charter of rights given above. Thus in POP-2 we can say `SQRT→PIG`; and later discover that `PIG(144)` has the value 12. It should be noted also that certain words and symbols in POP-2 are *not* the names of items. Thus, while `+`, `LOG`, `CONS`, are the names of items, `)`, `THEN`, and `GOTO`, are not. Statement labels are not items: i.e. 'computed gotos' are not permissible. The same effect can be obtained by exploiting the freedom to assign to functions. The role of non-item words is syntactic.

A consequence of clause 2 of the items' charter of rights is that POP-2 requires a more general form of storage control than is provided by a stack or push-down store. In fact a generalized form of the LISP scheme is provided. An ALGOL array can be placed on a stack precisely because it cannot survive the activation of the procedure or block in which it occurs. This has a profound effect on the class of problems which it is possible to tackle in a natural way using POP-2. An important corollary is that the space occupied by the code which represents functions is itself under the general storage control scheme.

The syntax of the language should be neat, unobtrusive and simple, rather than elaborate. Additional semantic power should not be locked away from the user by syntactic devices. Thus a POP-2 array is created by applying a function `NEWARRAY` to a description list, and assigning the result to a variable, none of which operations involves any new syntax.

POP-2 is designed for on-line use. The language features which reflect this are (1) the ability to have expressions executed immediately, and (2) the limited block structure of the language. A POP-2 program tends to be a sequence of independent function definitions, which can be redefined independently.

ITEMS

Items are simple or compound. This distinction is made for practical reasons: a simple item is an economical quantity of information to move around a

machine, a compound item is not. Thus the limit to what is to be regarded as a *simple* item has been set with an eye to the amount of information that can be put in one word of a machine.

Items are classified into *types*. The words *item class* and *type* are synonymous. It is possible to determine the type of any item at any time. Except in the case of functions, it is not possible to restrict the range of variables to any particular type of item. This means that type checking must be done dynamically, that is to say, all functions must check that they have been supplied with appropriate arguments. This leads to reduced efficiency. The other reason for attaching types to variables is to permit errors to be discovered at compile-time. The conceptual type structure of the programmer is very much finer than the type structure of languages like ALGOL. Thus when we say 'let s be the speed of the vehicle v ' or 'let w be the weight of the house h ', s , v , w , and h , are, at one level, of different type, and a language that fully mirrored the thought processes of the human would allow this distinction to be made. Thus it is felt that if POP-2 were to have type restrictions on variables at all then it must have a type structure extendable by the user.

However, type structure is not simple. For instance, it is meaningful to say 'let w be the weight of the vehicle v ' but it is not meaningful to say 'let s be speed of the house h '. Thus any extended type structure would be too complex to put in what is conceived of as essentially a simple language. Not only would it be too complex, it could be too cramping. One could not define a function MAPLIST to apply any function to all members of any list if one had to specify the types of its arguments. Indeed, the ability to write 'metaphorical programs' where a set of functions for, say, performing a search procedure can be used with widely differing types of object being searched for, is very important in machine intelligence. Thus on the one hand, Burstall's (1968) program for fact retrieval uses one set of functions for both syntax analysis and deduction, while on the other hand the Graph Traverser, an ALGOL program (Doran and Michie, 1966) is handicapped by operating solely on rectangular arrays.

Numbers, integer and real, are simple. For reasons of efficiency the truth values TRUE and FALSE are represented as the numbers 1 and 0 respectively, and as such are simple. There are also operations which allow numbers to be treated as bit-patterns.

Type conversion between integer and real items is performed automatically at run-time.

Compound items can be thought of mechanistically as pointers to areas of store. Certain standard compound items are provided. These include

List cells

List cells and operations of the LISP type are provided. There are, however, a class of entities, which are variously called streams or files, the processing of which is sequential, and which it is desirable to handle in a manner analogous to the manner in which lists are handled. It is not practicable to convert these

things into lists because of the prohibitive amount of store they would occupy. Landin (1966) proposed a representation in terms of functions without side effects. In POP-2 a more efficient solution is provided: *dynamic lists*. These are lists of the LISP type, except that the last cell in the list contains a function. There are operations HD and TL on dynamic lists, which, like CAR and CDR of LISP, select respectively the first member and the remainder of the list. When the last cell of a dynamic list is reached, the resident function is applied to no arguments, and the result is taken to be the next member of the list.

The list is terminated when the function produces TERMIN as its result. In this case TRUE is written into the first position in the last cell. The function NULL which tests for the emptiness of a list, returns the result TRUE when it is given a list cell with TRUE in its first position and a function in its second.

As an example, suppose *f* is a function to read a character off an input tape. The statement

```
FNTOLIST(F)→X;
```

using the standard function FNTOLIST, will set *x* to be a dynamic list built from *F*. The function COUNT defined by

```
FUNCTION COUNT X;
  VARS N; 0→N;
  LOOP: IF NULL(X) THEN N EXIT
        IF HD(X)=HD(TL(X)) THEN N+1→N CLOSE
        TL(X)→X; GOTO LOOP
  END
```

which counts the number of pairs of identical adjacent characters on the input tape, is only fractionally more convenient than the same function written using *F* directly, but the gain increases with the complexity of the operations being performed.

Words

System routines are available for converting sequences of symbols into data cells called words. These contain the first eight characters of the word. Characters are converted into words according to the conventions of the POP-2 compiler. Thus CAT), +, and ++ are words. Quotes are used to distinguish words from identifiers e.g. "CAT" denotes the word CAT while CAT denotes the current value of an identifier CAT.

Words are *standardized*. That is to say, on construction of a word, a check is made to see whether a word having the same sequence of letters has been encountered before, and if so the previous incarnation is used. Constant words are quoted in program, and "DOG"="DOG" always produces TRUE, because of this standardization. There is a function MEANING on words which associates an item with a word (similar to the property list of a LISP atom). "CHIEN"→MEANING ("DOG").

This function allows one to construct an updateable dictionary, the entries of which are undefined until assignment is first made to them. This example of assignment is analogous to $x \rightarrow HD(Y)$.

Users may themselves define classes of compound items.

Records

Records are data cells with a number of components fixed over a *record class*. Thus the class of list cells is a record class each member of which has two components, namely, HD and TL. The amount of information in each component of a record is specified in a description list, which is handed to the function RECORDFNS which is used to create a record class. The description list for list-cells would be [0 0], 0 in a description list being used to indicate a component which can be a full compound item. The result of RECORDFNS is a number of functions for manipulating the records of the class being defined. Thus the class of list-cells would be defined by

```
RECORDFNS("LIST", 1000, [0 0]) → BACK → FRONT → DEST → CONS;
```

where FRONT and BACK are CAR and CDR in LISP, and DEST is a function that 'explodes' a list-cell into its components.

Strips

Strips are cells with a number of components which is variable over a class of strips. The components of a strip are all of the same size, and are accessed by a subscripting function associated with each strip class. Thus, if s is a strip, w is an integer and SUBSCR is a subscripting function for the class of strips to which s belongs, then SUBSCR(N, s) produces item N of s .

Arrays are functions which have strips attached to them as work-space. The way in which this attachment is performed will be described in the section on functions.

There seems to be a need for classes of objects with properties intermediate between strips and records. These would have the variable size of strips, and the different component types of records. Thus, a record of a person might have a selector-subscriptor NAMECHAR, such that NAMECHAR(I, P) took the i 'th character of the name of the person P .

FUNCTIONS

Functions are compound items, and as such are the values of variables. LAMBDA $x; x*x$ END → SQUARE; (for lambda notation see Landin, 1966) assigns that function which takes x onto its square to the variable SQUARE. A 'sugared' version of this is

```
FUNCTION SQUARE  $x; x*x$  END
```

Binary operations like $+$ $-$ $<>$ ($<>$ is the infix version of concatenate as applied to lists) are identifiers which are the names of functions. These identifiers are recognized by the compiler as naming infix operations. The

PROBLEM-ORIENTED LANGUAGES

modifier **NONOP** causes them to be treated as normal functions, e.g. **NONOP * → PROD**; causes the value of **PROD** to be set to the multiplication function. Note that only the semantic, not the syntactic, content of ***** passes to **PROD** in this assignment, so that to apply the new function we must say **PROD(x,y)**. If we had wished **PROD** to be an infix, we would have proceeded as follows:

```

VAR OPERATION 3 PROD;
NONOP * → PROD;

```

now **x PROD y = x * y** gives the value **TRUE**. The numeral in this declaration sets the precedence.

Some functions have a meaning when used to the right of an assignment arrow. Thus **4 → A(3,2)** is meaningful if **A** is an array, or **34 → HD(L)**; if **L** is a list. Every function has a component called its **UPDATER**. Suppose we are keeping records of people in lists, giving details of their ages and sexes. Thus: **[SOCRATES MALE 2436]** or **[SATAN MALE 100000]**. One can define the functions **AGE** and **SEX** on these records. Thus:

```

FUNCTION SEX X;
  HD(TL(X))
END

```

Everyone who has an age has birthdays. According to *Malleus Maleficarum* (Institoris, 1948), Satan, for reasons which need not detain us here, needs to change his sex frequently. The functions **AGE** and **SEX** as defined above will not perform this updating operation. However the statement

```

LAMBDA X Y; X → HD(TL(Y)) END → UPDATER(SEX);

```

will change the **UPDATER** component of **SEX** in such a way that if we say

```

[SATAN MALE 100000] → P;

```

then later

```

"FEMALE" → SEX(P);

```

will update the record **P** as required.

There are two methods of creating new functions dynamically. One is **POPVAL**, which is a function which compiles program, in the form of a list. The other is *partial application*.

Suppose that **F** is a function of n arguments: we may 'partially apply' it to an argument list of only m elements by writing **F(% $x_1, x_2, \dots, x_m$ %)**. The result is a new function of $n-m$ arguments, obtained by 'freezing' the values of the last m arguments to the values possessed by $x_1, x_2, \dots, x_m$ at the time of the partial application. Thus suppose **DIST** is a function for finding the distance between pairs of places, and **EDINBURGH** is a place, then **DIST(% EDINBURGH %)** is a function for finding the distance of a place from Edinburgh, and we would be at liberty to assign it, e.g. **DIST(% EDINBURGH %) → MILEAGE**. **MILEAGE(LONDON)** now gives 380.

The following illustrates the use of partial application in defining a functional. Suppose ZERO is a function for finding one zero of a function. Consider the function INVERSE defined by

```
FUNCTION INVERSE F;
  AUXINVERSE(% F %)
END
```

where AUXINVERSE is the function defined by

```
FUNCTION AUXINVERSE Y F;
  ZERO(LAMBDA X; Y-F(X) END)
END
```

Then, leaving aside the niceties of numerical analysis, INVERSE is a functional which takes a function of one argument over the reals onto its inverse. Thus INVERSE(SIN) is ARCSIN. This is deduced as follows: according to the definitions INVERSE(SIN) is AUXINVERSE(% SIN %), that is, AUXINVERSE with F 'frozen' to SIN, that is a function with the same result as

```
LAMBDA Y; ZERO(LAMBDA X; Y-SIN(X) END) END
```

Given Y, the result of this function will be that x for which $Y - \sin(x) = 0$, that is that x for which $\sin(x) = Y$, that is ARCSIN(Y).

The above definition of INVERSE will seem unnecessarily complicated on first reading. Could F not be a non-local of AUXINVERSE and the partial application be dispensed with? If this were so, when we came to evaluate ARCSIN the value of F might have been changed, for instance by another application of INVERSE to, say, LOG. However, with the partial application in place, if we say INVERSE(LOG) → EXP, then ARCSIN and EXP are two functions which are obtained from AUXINVERSE by freezing F to the values SIN and LOG respectively.

Landin (1966) uses an equivalent scheme. The frozen formal parameters of POP-2 correspond to the environment part of his closures. While his method, as embodied in CPL, is syntactically more elegant, partial application technique of POP-2 is perhaps more flexible, because the user can decide which variables are to be frozen.

FUTURE DEVELOPMENTS

Now that an implementation of POP-2 is available on a machine (an Elliott 4120) it is time to consider the future. There are three lines of development. The first is to make extensions to the language to make use of backing store and to make more efficient use of machine time.

The second is to build a library of *functions* to be used as building bricks of intelligent programs. The increased power that partial application gives POP-2 is very important here. In languages like ALGOL, one can write *learning programs*: it is difficult to write *learning procedures*. This is because the only *memory* that a procedure can have to itself is in the form of 'own' variables.

Since these have only one incarnation, the learning mechanism enshrined in a procedure can only be used in one sub-problem of the whole problem. In POP-2 on the other hand, a learning mechanism encoded in a function can be attached to data concerned with many sub-problems by partial application. Arrays may be regarded as *rote-learning functions*, a point of view implicit in the 'memo function' concept proposed by Michie (1968).

One can write in POP-2 a function which one might call, by analogy with NEWARRAY, NEWINTERP. NEWINTERP() produces a function called an interpolator. Suppose we are studying devices called tripistors. These pass a current that depends on the applied voltage. If we say NEWINTERP() → TRIPISTOR; we have a function which we can teach to represent tripistor behaviour. Suppose we have observed that a tripistor passes a current of 10 amps when 1 volt is applied, and 0 amps when 0 volts are applied, then this information is conveyed by the statements:

10 → TRIPISTOR(1); 0 → TRIPISTOR(0);

If we now ask for TRIPISTOR(0.5) the machine produces the estimate of 5, by linear interpolation. As further information is fed in so better interpolations will be possible. Meanwhile, in another part of the program, the same interpolation mechanisms have been learning the behaviour of livertrons. When these lessons have been adequately learnt, they can be applied in predicting the behaviour of a circuit containing both tripistors and livertrons.

Finally, as POP-1 was used to develop POP-2, so POP-2 will be used to develop its successor. What sort of language will this be? One can regard existing algorithmic languages including POP-2 as being *imperative*. They are used to describe the solution to a problem. The next generation of languages will be *indicative*. They will be used to describe a problem.

Both kinds of language have been used by logicians: the lambda calculus is their imperative language, the predicate or functional calculus the indicative languages. Algorithms for interpreting these languages are known – e.g. the resolution principle for the functional calculus. Let us consider the following statements in BNF – which can be regarded as an indicative language:

<NP> ::= THE <NOUN>
<NOUN> ::= CAT | DOG

We can regard this as a 'sugared' version of the following statements in functional calculus.

$\forall x (ISNP(x) \iff (\exists y (ISNOUN(y) \text{ and } x = [THE] \langle \rangle y)))$
 $ISNOUN([CAT]) \text{ and } ISNOUN([DOG])$

In order to use resolution, these have to be rewritten in conjunctive normal form. Leaving out the $\implies$ of the $\iff$ we get:

- (1) not(ISNOUN(y)) or not ([THE] $\langle \rangle$ y = x) or ISNP(x)
- (2) ISNOUN([CAT])
- (3) ISNOUN([DOG])

These statements can be used analytically or generatively. To use them analytically, suppose it is conjectured that $ISNP([THE] <> [CAT])$, i.e. that $[THE] <> [CAT]$ is a nounphrase. Then one adds the denial of this statement to 1-3 and asks for a contradiction.

(4) $not(ISNP([THE] <> [CAT]))$

Resolving (4) with (1) gives

(5) $not(ISNO\dot{U}N(y))$ or $not([THE] <> y = [THE] <> [CAT])$

Resolving (5) with (2) gives

(6) $not([THE] <> [CAT] = [THE] <> [CAT])$

a contradiction, by resolution with $x=x$ —an equality axiom.

To use (1)–(3) generatively one need only deny the existence of a nounphrase.

(4a) $not(ISNP(x))$

Resolving (3) with (1) gives

(5a) $not([THE] <> [DOG] = x)$ or $ISNP(x)$

Resolving (5) with $x=x$ gives

(6a) $ISNP([THE] <> [DOG])$

a contradiction with (4a), and an instance of a nounphrase.

While the deduction process in such an indicative language is algorithmic, the choice of deductions to be made is not. Foster, in this volume, proposes a method whereby a user of such a system could control the direction of deductions. With sufficiently rigid control one would have an imperative language.

Work at Edinburgh would be concerned with a heuristically controlled system. The study of imperative systems viewed as highly constrained indicative systems seems important to the understanding of milder heuristic constraints.

Acknowledgments

The research described in this paper is sponsored by the Science Research Council. I am particularly in debt to my colleague Dr R. M. Burstall for his many helpful suggestions and criticisms.

REFERENCES

- Burstall, R. M. (1968), A combinatory approach to relational question answering and syntax analysis. *Proceedings of Nato Summer School Copenhagen*. Amsterdam: North Holland Press.
- Burstall, R. M. & Popplestone, R. J. (1968). The POP-2 reference manual. *Machine Intelligence 2*, pp. 205–46 (eds. Dale, E. & Michie, D.). Edinburgh: Oliver & Boyd.

PROBLEM-ORIENTED LANGUAGES

- Doran, J.E. & Michie, D. (1966), Experiments with the Graph Traverser program. *Proc. R. Soc. (A)*, 294, 235-59.
- Institoris, H. (1948), *Malleus Maleficarum* (translated M. Summers). London: Pushkin Press.
- Landin, P.J. (1966), A lambda-calculus approach. *Advances in programming and non-numerical computations*, pp. 97-141. Oxford: Pergamon Press.
- Michie, D. (1968), *Memo functions: a language feature with rote-learning properties*. Submitted to *Proc. IFIP 68*. Also available as *Research memorandum MIP-R-29*, Edinburgh: Department of Machine Intelligence and Perception.
- Popplestone, R.J. (1968). Pop-1: an on-line language. *Machine Intelligence 2*, pp. 185-94 (eds. Dale, E. & Michie, D.). Edinburgh: Oliver & Boyd.

INDEX

- Abramson 48, 60
AED 393-4
ALGOL 37, 38, 52, 257, 265, 267, 272, 314,
358, 363, 370, 380, 384, 393-4
Formula ALGOL 263, 265
allocation problem 237-41
Alway 44, 60
Amarel 143, 170
Anderson 387, 391
Andreae 195, 216
Atlas computer 269-78
- Banerji 173, 179
basic English 318-20
Baumert 373, 385
Baylis 277, 278
Bellman 135, 170, 237, 242
Bernays 114, 115, 118, 120, 127
Black 143, 170
Blum 326, 347
Bond 258, 267
Bratley 290, 297
Burstall 177, 179, 212, 216, 395, 401, 402
Buxton 17, 387, 391
- Calvert 265, 267
Carnap 134, 170
Carson 75, 93, 98, 101, 112, 127
Chambers 206, 207, 213, 216, 247, 250 251,
254
Chauvin 217
Chomsky 282, 297
chromosome analysis 332-47
Church 63
coin collector problem 254
Coleman 264
Coles 265, 267
Collins 249, 255
compiler validation 33-41
CPL 5, 263, 272, 351-5, 362, 365, 370, 393
CSL 4
Culik 17
- Dakin 290, 297
- Darlington 126, 127
Davis 28, 30, 31, 63
Dean 275
Dewar 297
Dijkstra 37, 41
Doran 200, 212, 216, 245, 247, 255, 373,
376, 379, 385, 395, 402
Dreyfus 135, 170, 237, 242
du Plessis 275, 278
Duncan 18
- Edwards 212, 216
eight-puzzle 245-7
Elcock 173, 179
Ernst 195, 216, 257, 267
Euclid 108, 113
Evans 195, 216
- Farrow 347
Feigenbaum 70, 213, 216
Feldman 70
Fennel 277
Fleming 249
Foster 173, 177, 401, 402
Frege 114
Friedman, J. 297
Friedman, L. 195, 216
- Gaines 195
Galanter 215, 216
Gelernter 63, 68, 70
Gel'fand 218, 220, 231, 242
General Problem Solver 195
Gödel 63, 64
completeness theorem 66
Golomb 373, 385
Go-Moku 177-8
Goto 225
GRANIS 265
graph 326
linear 43
graph theory 325-47
Graph Traverser 212, 245, 247-55, 373, 376,
379, 395

INDEX

- GRASP 265
- group theory 65, 68-70, 99, 104-8, 123-6
- Guard 113, 126, 127
- Gurfinkel 218, 220, 242

- Hall 297
- Harary 48, 54, 60
- Hart 212, 216
- Hasenjaeger 114, 120, 127
- Hausner 387, 391
- height representation 57
- Herbrand 99
- Herbrand's theorem 66-7, 69, 74, 100
- Hilbert 114, 115, 118, 120, 127
- Hilditch 347
- Hodes 332, 347
- Hopgood 187, 193, 265, 267
- hyper-resolution strategy 72, 96

- Ianov 30, 31
- Institoris 398, 402
- ISWIM 4, 379, 382, 384

- Jacobs 212, 216

- Kalah 181-92
- KALGOL 58
- Karr 387, 391
- Kay 282, 297
- Khalifman 231, 242
- Kleene 286, 297
- König's infinity lemma 27
- Krom 102, 111

- Landin 18, 374, 379, 380, 382, 383-5, 397, 399, 402
- Laski 18, 387, 391
- Levenstein 225, 228, 242
- LISP 5, 101, 271, 362, 368, 369, 393-4, 395
- Loveland 63, 68, 70
- Luckham 96, 99, 111, 127
- Lukashevich 231, 242

- McCarthy 34, 41, 101, 111, 143, 163, 170, 215, 225, 361, 368, 371
- McCormick 326, 347
- Marakhovsky 228, 231, 242
- Markowitz 387, 391
- Martin 44, 60
- mathematical induction 118-19
- Meleshina 220, 221, 231, 237, 242
- Meltzer, 71, 72, 75, 99, 101, 111, 123, 127
- Michie 185, 193, 206, 207, 212, 213, 216, 245, 255, 373, 376, 379, 385, 395, 400, 402
- Miller 215, 216
- Minsky 225
- missionaries and cannibals problem 135-64, 166-70
- Mitchell 267
- monkey problems 163-4
- Moore 224, 225, 242
- Morris 18
- Murray 173, 177, 178, 179
- Myhill 224, 228

- Narasimhan 325, 326, 347
- Naur 371
- Newell 63, 135, 143, 166, 170, 171, 195, 216, 257, 267, 373
- Norton 102, 111
- noughts and crosses 177-8
- number theory 108-11, 113, 126

- Obruca 44, 60
- Oldfield 249
- Opler 387, 391
- Ore 326, 347

- P_1 -deduction 71-2, 99, 101
- P_2 -deduction 72
- Painter 34, 41
- Parris 44, 60
- Paterson 31
- Peano axioms 64
- Perekrest 237
- Perlis 267
- Peshchansky 228, 231, 242
- Petrick 282, 297
- Pfaltz 330, 341, 347
- Philbrick 325, 327, 347
- PL/1 34, 370
- plex 271
- POP-1 120, 393
- POP-2 200, 393-402
- Popplestone 200, 402
- Prawitz 63
- predicate calculus 78
- prex 3-17
- Pribam 215, 216
- prime basis algorithm 90
- Prins 48, 54, 60
- Proust 320

- Quatze 258, 267
- queueing theory 233-7

- Rabin 27, 31
- Read 44, 60
- resolution principle 77-93, 95, 96
 - generalized 87
 - generalized ground 92
- resolution strategies 71-5
- Riordan 48, 60
- Robinson, G. A. 72, 75, 93, 95, 98, 101, 127
- Robinson, J. A. 63, 68, 69, 71, 72, 74, 75, 93, 95, 96, 99, 101, 111, 113, 127
- Rodriguez 271

- Rosenberg 27, 31
- Rosenfeld 330, 341, 347
- Ross, D. T. 271, 278
- Ross, R. 247, 255
- Runyon 319-20
- Russell, B. 18
- Russell, R. 183, 187, 193
- Rutovitz 330, 347

- Saaty 242
- Samuel 186, 193, 195, 212, 216, 321, 322
- Sandewall 195, 216
- Schofield 246, 255
- Scoins 46, 60
- Scott 27, 31
- set-of-support strategy 71, 98
- Shannon 321, 322
- Shaw 63, 166, 170, 195, 216, 373
- Shen Lin 247, 248, 249, 255
- Sibert 69
- Simmons 165, 171
- Simon 63, 132, 135, 143, 166, 169, 170, 171, 195, 216, 257, 267, 373
- Skolem functions 67, 69, 118
- Skolem-Löwenheim theorem 67, 70
- Slagle 373, 385
- Snow 43, 54, 60
- solo whist 192
- Steel 18
- Stein 347
- Strachey 17, 353, 355, 371
- Sutherland 176, 179, 271, 278

- Tarski 65, 66, 67
- Thomas 265, 267
- Thorne 285, 297

- Toda 195, 216
- travelling salesman problem 247-55
- tree searching 373-84
- trees 43-59
 - binary rooted ordered 44
 - non-ordered binary rooted 47
 - ordered rooted multi-furcating 51
 - non-ordered rooted multi-furcating 54
 - free 59
- Tsetlin 218, 219, 220, 221, 222, 231, 242
- Turing 63
- Turing machine 28, 30
- two-headed automata 27-9

- Unger 44, 60
- unit preference strategy 72, 98

- Van Zoren 267
- Varshavsky 219, 220, 221, 222, 224, 228, 231, 242
- Veenker 126, 127
- voting problem 228

- Waksman 225, 242
- WALGOL 58
- Walker 297
- Walter 195, 216
- Wang 102, 104, 108, 112, 126, 127
- Weaver 322
- Weber 35, 41
- Whitehead 18
- Whitfield 297
- Wirth 35, 41
- Wos 71, 72, 75, 93, 98, 101, 104, 112, 122, 123, 127
- Zwicky 282, 297

