9

STRATEGY-BUILDING WITH THE GRAPH TRAVERSER

D. MICHIE
EXPERIMENTAL PROGRAMMING UNIT
UNIVERSITY OF EDINBURGH

I shall discuss automatic methods of search for solutions in problems susceptible of a particular formal representation, namely that on which the Graph Traverser program (Doran & Michie 1966, and see Doran p. 105) has been based. In this representation which is essentially that of Newell, Shaw & Simon (1960) a problem consists of a set of *states*, one or more of which is labelled 'goal', together with a rule-book. The rule-book lays down for each state what *moves* may be made from it to reach other, neighbouring, states. Solution consists in a demonstration that a goal state can be reached from some given initial state via a sequence of intermediate states, sometimes with the additional requirement that an actual sequence, or *path*, be demonstrated as part of the solution; in this last case, as in the sliding block problem discussed later, the goal (or goals) is typically specified in full as part of the statement of the problem. In other cases, where the goal is only specified in terms of some defining property (such as cost in a transportation problem, or simplicity in a problem of algebraic manipulation) we are interested solely in discovering states possessing this property, and not at all interested in the particular paths leading to them. Indeed the *only* thing 'given' in the problem statement apart from the description of the set of states may be the defining property of the goal state(s): in that case, if the method of attack is to remain within the formal bounds considered here, either the problem-solving program or its author must propose an initial state from which to launch the search, and must also make up the rule-book. Burstall's program for designing electricity distribution networks (p. 65) provides an excellent illustration of this last eventuality. In Burstall's (see pp. 65-85) problem a solution is defined as a network, the calculated

cost of which does not exceed some stated quantity, and which satisfies certain security constraints. After an initial trial network has been proposed, the program then follows rules invented to allow moves of the following two elementary kinds within the limits set by the security constraints:

(1) adding a line joining points i and j for any $i \neq j$;
(2) deleting a line joining points i and j for any $i \neq j$;

together with two more obtained as compounds of these; namely

(3) two operations of (1) above;
(4) two or more operations of (2) above.

For any substantial number of points in the network, the number of possible moves which can be generated in these four categories becomes very large, necessitating drastic methods of selection. The way in which this selection is effected is discussed by Burstall (*loc. cit.*). The point I wish to make here is that the basic moves by which neighbouring states are inter-transformable *may* be unalterably fixed in the structure of the problem as given, or definitions of allowable moves *may* be imported into the problem in order to give it a structure susceptible to general search methods. A fuller classification of problem types is given by Doran & Michie (1966) in a paper to which the reader is referred for the background information which the present discussion takes for granted.

TWO APPROACHES

Work on automatic problem-solving has tended in the past to polarise around two basic approaches. One approach, based on *state-evaluation*, generates all the states of the problem which can be reached in a small number of moves from the current state, and then seeks by some process of evaluation to decide which state shall form the next point of departure. Most, but not all, of the work on game-playing has been of this kind (for review see Michie 1966). The other approach applies selection *not* to the individual states of the problem but rather to the set of operators used for effecting transformations of state. In the classical studies of Newell, Shaw & Simon (1960) selection is applied by going down a priority sequence of operators, applying to each in turn a number of *tests*, first of applicability to the current state and then of whether the operator conduces towards one or another of various desirable intermediate states, or *subgoals*. I shall refer to this approach as placing the emphasis on *operator-selection*.

THREE STAGES

Present and planned work with the Graph Traverser program can be divided into three stages.

In *stage* 1, which has monopolised our attention so far, we concentrate on the exploitation and improvement of evaluation functions, with operator-selection deliberately side-tracked. For this purpose the program's 'develop'

procedure (which operates upon the given state of the problem to generate descendant states) is set simply to generate immediate descendants by applying the elementary moves. In other words, in this mode the develop procedure contains *only* the rule-book.

In *stage* 2 the develop procedure is extended so that it (*a*) generates operators with a more ambitious reach than the elementary moves by stringing together combinations of the latter, and (*b*) applies to a given state not all available operators but only the most promising selection from the full set. In this stage, selection is integrated with state-evaluation into a unified system, suitable for use as a tool for improving an already-existing problem-solving strategy.

In *stage* 3 the program is endowed with the ability, in any given problem environment, to construct capabilities (*a*) and (*b*) for itself out of the fruits of its own experience of that environment. At this stage the program is no longer a tool in the hand of the designer but acquires something of the designer's own role.

Stage 1, which has already been implemented and extensively tested, is discussed elsewhere in this volume (Doran, p. 105). Stage 2 has been the subject of some preliminary experimentation which I shall here report, small in amount, but sufficient for an initial demonstration of feasibility. Stage 3 belongs to the future. I shall conclude my account by detailing a series of evolutionary steps leading from the most primitive form of stage 1 to a fully-fledged form of stage 3, each step needing for its accomplishment no more than the systematic application of existing methods.

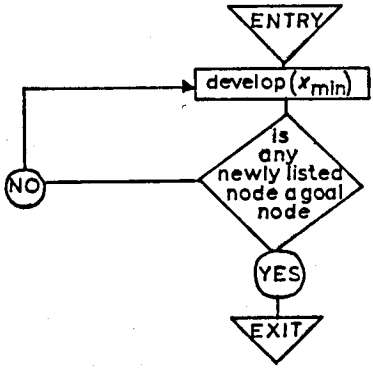
THE SYNTHESIS OF OPERATORS AND THE NEED FOR PRUNING

Elementary moves may be strung together to form compound moves. Suppose that we have three moves at our disposal called, say 'jump', 'hop' and 'skip', and that the sequence 'hop, then skip, then jump' is found often to be useful. Then it may be advantageous to add a new operator 'hop-skip-jump' to the armoury. 'Operator' is here used as a generic term to denote both moves and compound moves. I. J. Good once referred conversationally to the use of highly compounded chains of moves as 'the method of seven-league boots'. In the spirit of this phrase one should perhaps regard Burstall's (see pp. 65-85) program as proceeding in part in one-league boots (elementary moves) and in part in two-league boots.

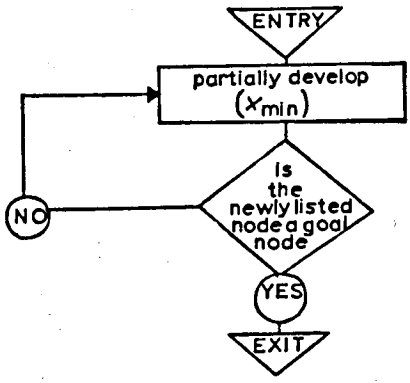
The main difficulty about the method of seven-league boots is that the added range brings with it a proliferation of choices. If for a given problem there are available an average of m moves from each problem-state, then the number of operators which can be formed by compounding sequences of n moves will be of the order of m^n. Drastic methods of pruning are therefore desirable, and it is this pruning process to which we refer when we talk about 'operator selection'.

SELECTION BY PARTIAL DEVELOPMENT

Consider the rather striking consequence which follows from an extremely natural modification of the current version of the Graph Traverser. In its present form the cycle of iteration is as follows:



where x_{min} denotes the minimum-valued undeveloped node. Note that since 'develop' involves applying in turn *all* the operators in the set, no selection is involved. Now consider what would happen if for 'develop' we substituted 'partially develop' denoting by this a procedure which produces only *one* descendant for evaluation (compare the 'short-listing' procedure proposed by Maynard Smith & Michie 1961). First envisage this being done by a random selection from among those members of its operator set which had not previously been applied to the node in question. We now have the following picture of the main loop:



i.e., list *one* descendant not previously listed and evaluate it.

The effect on the search strategy brought about by this innocent-looking modification is radical. It can most easily be illustrated by assuming a hypothetical case, typical of those with which real life abounds, in which the operator set is very large, so that to develop completely one node of the problem-graph would entail the drawing of literally thousands of lines radiating from it. Now suppose that at some stage of the search x_{min} has

the value 200, and that the develop procedure is called by the program. Old style, the application of operators to locate next-neighbours marches grimly on, as in Fig. 1, with no means of arresting the process short of locating *all* the descendants of x_{min} even though it is clear from the very first *partial* development that x_1 is a more promising node than x_{min} for further development. The new-style search shifts the label 'x_{min}' as soon as x_1 has been evaluated and the relation $x_1 < x_{min}$ discovered, as in Fig. 2.

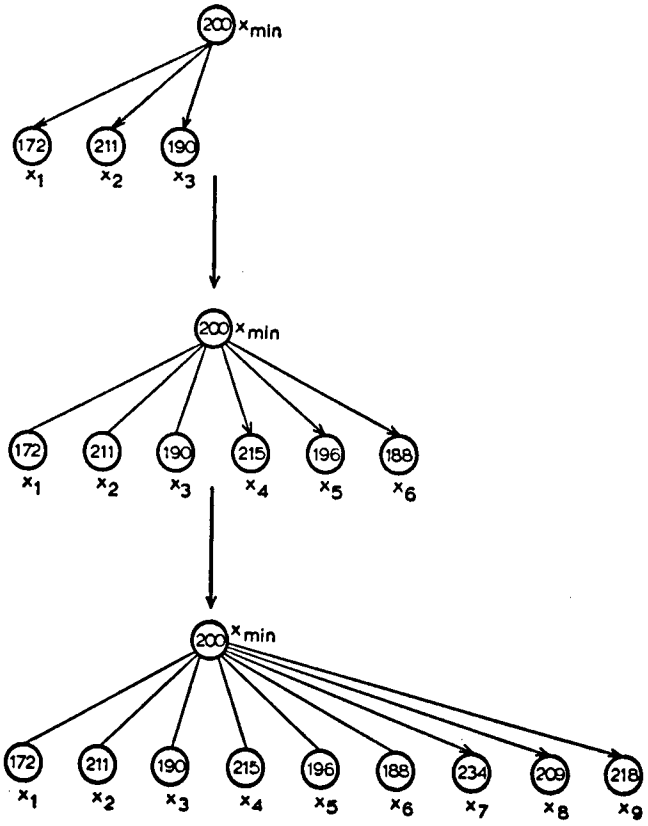


FIG. 1.

A considerable pruning of the search tree can be achieved by this fundamentally simple extension of the Graph Traverser's philosophy of opportunism, as has been well shown by Popplestone (pp. 31-46) in his application of essentially this form of the program to a theorem-proving problem.

CONDITIONAL CHOICE STRATEGIES

In the extreme case of operator selection only *one* operator is chosen for application at each given state of the problem, and this choice is made according to abstracted strategic features of the given state. Thus in algebraic

simplification one might follow a rule that *whenever* an expression of the form $\frac{a}{b}+\frac{c}{d}$ appeared, the compound move 'cross-multiplication' should be applied. Indeed a conditional-choice strategy is almost invariably what the human designer or mathematician or programmer produces when he is asked to devise a strategy for a specific problem. The search then proceeds, not as a spreading tree in the manner of Fig. 1 of Doran's paper (p. 109) (trees grow *downwards* in this book), but more like the non-branching stem of a bamboo as shown in Fig. 3. Each node in the diagram can be expanded, to show what goes on at each stage, into the form shown in Fig. 4.

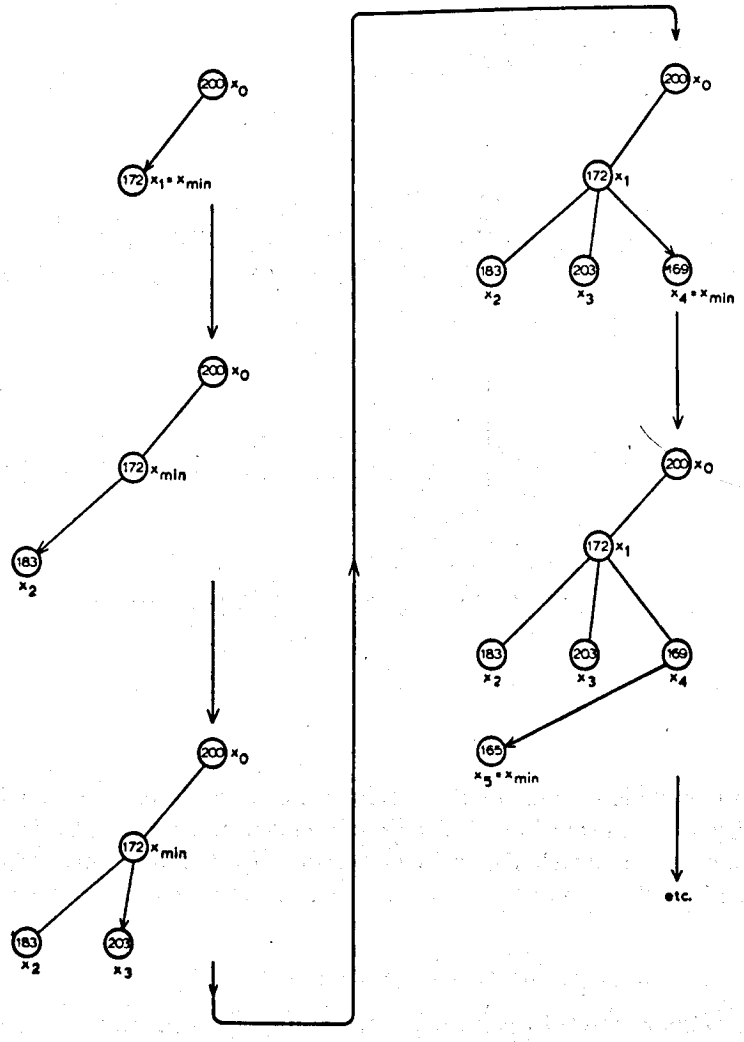


FIG. 2.

The nature of such algorithms can be illustrated rather vividly by considering the task of devising a conditional-choice strategy for a simple puzzle, the Eight-puzzle, of which we have made much use while developing the Graph Traverser program.

The Eight-puzzle is a poor relation of the well-known Fifteen-puzzle (Johnson & Storey 1879, Tait 1880). It requires the rearrangement of eight numbered blocks by a succession of sliding movements within a 3×3 square frame. We take, by convention, the following configuration to be the goal,

$$\begin{matrix} 1 & 2 & 3 \\ 8 & 0 & 4 \\ 7 & 6 & 5 \end{matrix}$$

where the zero stands for the empty space. P. D. A. Schofield (pp. 125-33 of this volume) has shown that the $8!-1$ possible centre-empty starting configurations are distributed over a range of difficulty from minimal paths of 4 moves up to minimal paths of 30 moves. Intelligent human subjects with no previous

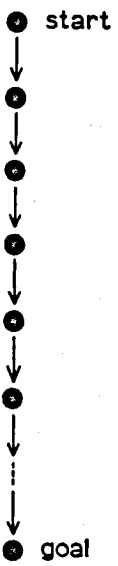


FIG. 3. A conditional-choice strategy depicted as a non-branching tree.

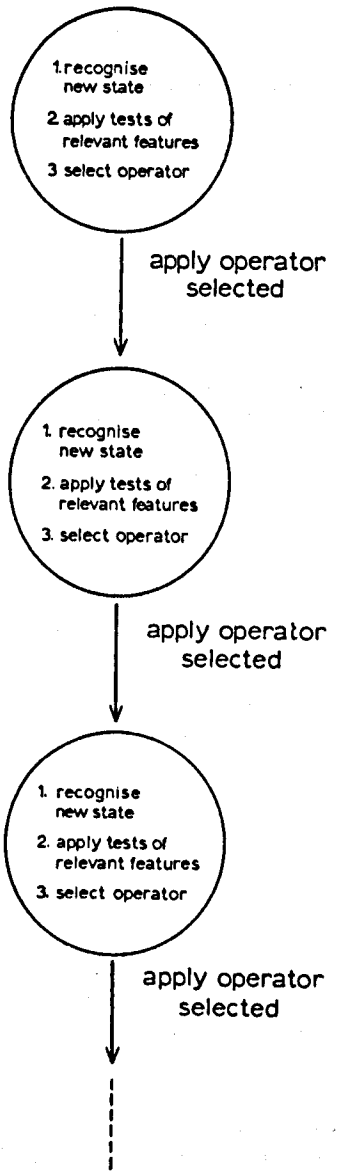


FIG. 4. Expansion of Fig. 3 to show what happens at each node.

experience of the puzzle achieve path efficiencies in the range 30-40 per cent for the more difficult configurations, where path efficiency

$$= \frac{\text{no. of moves in minimal path}}{\text{no. of moves in actual path}}$$

With intensive practice the average rises to about 50 per cent. We have found one individual, Mr Roger Chambers, who can consistently score 90 per cent.

A conditional-choice strategy giving about 75 per cent path efficiency can be fairly easily designed by setting up as *first* objective that the longest existing numerical sequence of blocks (like the string 456 in the following example)

6 8 2
5 0 7
4 1 3

should be extended until a sequence of six or more has been achieved, and as *second* objective that the blocks should be got to their homes by a sustained process of circulating them around the outer 'ring road'.

The flow chart shown in Fig. 5 depicts the main loop of this rather simple, but moderately effective, 'simple chaining' strategy. I am now going to indicate how such a strategy can be neatly embedded in the 'develop' procedure of the Graph Traverser, in such a way that the 'evaluate' procedure can be used to hammer the weak points into shape and so improve the strategy. My object will be to show, with the aid of this toy problem, the immediate potential applicability of the Graph Traverser as a design tool.

DESIGNING A STRATEGY FROM THE TOP DOWN

Let us view the simple chaining strategy described above as a means of developing a node of the problem graph, in the sense of locating its immediate descendant(s) for evaluation. We write 'descendant(s)' with the terminal 's' bracketed because the whole point of a conditional-choice strategy in its complete form is that only *one* operator is selected at each stage; any evaluation of the resulting single descendant node would thus be a phantom calculation without effect on the search. This is evident from the flow chart shown in Fig. 6, in which the main Graph Traverser program itself plays a dummy role, merely standing as a gateway between the exit and entry of successive applications of a conditional-choice strategy. I shall now say why I want to place conditional-choice strategies within the Graph Traverser framework.

The weakness of a conditional-choice strategy, as with man-made algorithms generally, is that while *some* of the mappings from state-features to operator-choices may have been made by the algorithm's author with great confidence, there are likely to be others where he has specified an operator more because *something* must be specified than because he knows that the choice is the best possible in the presence of the given features.