7

AN APPROACH TO AUTOMATIC PROBLEM-SOLVING

JAMES DORAN
EXPERIMENTAL PROGRAMMING UNIT
UNIVERSITY OF EDINBURGH

Summary

A digital computer program, the Graph Traverser (Doran & Michie 1966), can seek a solution to any problem which may be interpreted as that of finding a path from one specified node of a graph to another. Emphasis is placed upon the *evaluation* of intermediate states of the problem (nodes of the graph) according to the extent to which they resemble the 'goal' state.

Sample results from first applications of the program, and possible future developments, are discussed. The program is related to other problem-solving programs.

INTRODUCTION: PROBLEMS AND PROBLEM-SOLVING PROGRAMS

How to travel from London to Birmingham may, in some circumstances, be a 'problem'. So may be a crossword puzzle, the planning of a school timetable, the solving of an algebraic equation, and the curing of a sick person. I shall not attempt to *define* a 'problem', except to suggest that the solver is typically faced with an essentially passive 'situation' or 'system' which must be 'manipulated' until it satisfies certain requirements.

This paper is primarily concerned with digital computer programs to solve problems. It describes one such program in detail. This is the Graph Traverser (Doran & Michie 1966), which is being developed at the University of Edinburgh Experimental Programming Unit.

Problem-solving programs

A problem-solving program must solve, or help solve, problems. It must

be applicable to a range of problems, narrow or broad, but need not necessarily be useful for all the problems within that range.

Even when a problem is precisely stated, and when a method of solution known to be infallible is available, it does not follow that this provides a useful method of attack in practice. A solution must be obtained within certain limits of time, and the machine will have a finite 'memory'. Methods have to be found which are useful within these restrictions. Typically, they will be 'probably useful' rather than 'infallible'. Such methods are commonly called 'heuristic'.

Before discussing in detail an example of a simple heuristic problem-solving program, the Graph Traverser, I shall give an indication of the most important work carried out in this field, stressing the more fundamental ideas. For a more extensive, and admirably clear, review of such work the reader is referred to Newell & Ernst (1965).

The GPS (*G*eneral *P*roblem *S*olver) program of Newell, Shaw and Simon (1960) is well known. It is designed to permit its application to a range of problems such as proving trigonometric identities, proving theorems in the propositional calculus and compiling computer programs. The GPS attempts to solve a problem by constructing a hierarchy of 'goals', where a goal involves one of the following:

(1) transforming one object into another;
(2) reducing the difference between one object and another;
(3) applying a transformation (operator) to an object.

Given a goal, the program seeks either to attain it or to reduce it to a set of subsidiary goals (subgoals), each being of the type which the GPS can handle. Then each subgoal is similarly treated. Thus *subgoals* and *recursion* are major features of the program.

A descendant of the GPS is 'Gaku', a sophisticated problem-solving system with a number of new features (Hormann 1965). Gaku attempts to *plan*, that is, to use past experience to 'outline' a solution to a problem, and also to *induce* from the solutions of particular problems a general strategy which will suffice to solve any problem of a similar type. The system, which is still under development, has been little tested as yet. Ultimately it is intended that it should be capable of learning not only from direct experience, but also from a 'tutor' or other 'secondary' source of information.

Another program, SAINT (*S*ymbolic *A*utomatic *INT*egrator (Slagle 1963)), which is influenced by the GPS, has a more restricted field of application, namely symbolic integration problems. For example, if SAINT is asked to integrate

$$\int \frac{x^4}{(1-x^2)^{5/2}}\,dx$$

it will, like a human solver, try various lines of attack, and various substitutions, before ultimately finding a sequence of transformations leading to an integral of standard type.

The organisation of SAINT has a number of similarities to that of the Graph Traverser program. It is a characteristic of SAINT, however, but not of the Graph Traverser, that it pays little attention to the problem of *which* line of attack to develop next, reserving its effort for *how* that attack is to be developed.

Another important and famous program is the checkers (draughts)-playing program of Samuel (1959). This is a 'game-playing' rather than 'problem-solving' program, and, unlike the programs so far described, is designed primarily to perform well, with no attempt to simulate human behaviour. Its performance has been, in fact, remarkably good. When deciding what move to play, it follows out a large number of alternatives, including, of course, the opponent's moves, and then comes to a decision based upon estimates of the values of the board positions found at the limit of its 'lookahead'. In order to attach values to each of these board positions, the program uses an 'evaluation function' based upon various features of the position. Furthermore, different versions of the program are capable of improving their play in two ways:

(1) by keeping for reference a catalogue of the board positions so far encountered, together with certain information about each; and
(2) by modifying the parameters of the evaluation function in the light of the discrepancies between the estimated value of the current board position, and the values of the positions discovered during 'lookahead'.

An important class of problem-solving program is characterised by the use of a subset of natural English for communication between the user and the program. The emphasis is on ease of communication, rather than problem-solving power, or even generality of application.

Raphael's (1964) SIR (*S*emantic *I*nformation *R*etriever) can 'understand' and manipulate simple statements such as 'X is part of Y' and 'Z is to the right of W'. Each of the variables may be replaced by any noun so that the program can 'understand':

'The boy is to the left of the table'

and can combine this with:

'The table is to the left of the chair'

to discover:

'The boy is to the left of the chair'.

Notice that all the program knows about the boy, the table and the chair is the given set of relationships, together with any others which it can *deduce*.

Bobrow (1964) describes 'a question-answering system for high school algebra word problems'. If the problem is posed:

'The price of a radio is $69.70. If this price is 15% less than the marked price, find the marked price.'

The program will respond:

'The marked price is $82.00.'

Here, as with SIR, the program can 'recognise' certain relationships (this time primarily algebraic) between entities, and work with them to solve the problem. Moderately large variations are permissible in the form of words used to pose the problem.

There are a number of different aspects to a problem-solving program. I shall distinguish four:

(1) the external *application problem* to which the program has been applied;
(2) the *internal problem* which the program generates from the application problem and which it must solve in order to produce a solution to the application problem (this internal problem will be an 'idealised' version of the application problem, and will typically vary little from one application to another. It is important to realise that more than one internal problem may be capable of derivation from a given application problem);
(3) the *strategy* which the program uses to solve the internal problem;
(4) the translation process from the application problem to the internal problem. (This involves an *application package* of information, large or small, which must be provided for each different application.)

The Graph Traverser program will now be described in these terms.

THE GRAPH TRAVERSER PROGRAM

Work on the Graph Traverser to date has concentrated primarily on one particular aspect of automatic problem-solving, the *evaluation function*, which has been examined in some detail in the context of simple puzzles. In a problem-solving context, an evaluation function may be defined as a means of measuring the 'promise' of a potential problem state, as part of the process of deciding which possible course of action to investigate next. The aim has been to develop methods by which the program may generate, or at least improve, its own evaluation function.

The program has, however, been written with sufficient generality to permit a much wider range of application and experimentation.

Description of the Graph Traverser program

The description of the program and its organisation will follow the 'internal problem', 'strategy', 'application package' framework given in the introduction.

In order to discuss the internal problem of the program it is convenient to use a little of the language of graph theory (see, for example, Berge 1962). A *graph* in the mathematical sense is a set of *nodes* any two of which may or may not be connected by an *arc* which may be directed or undirected. A graph is said to be *symmetric* if all its arcs are undirected.

A *path* from one node to another is a sequence of arcs, each either undirected or pointing 'forwards', which extends from the first to the second node. The *distance* from one node to another is the smallest number of

arcs which form a path joining them. A graph is *connected* if for any two nodes x, y there is a sequence of arcs of any type between them. A *tree* is a connected graph without any kind of 'loop', as in the examples of Fig. 2. The *descendants* of a node are all nodes which can be reached from it along a path.

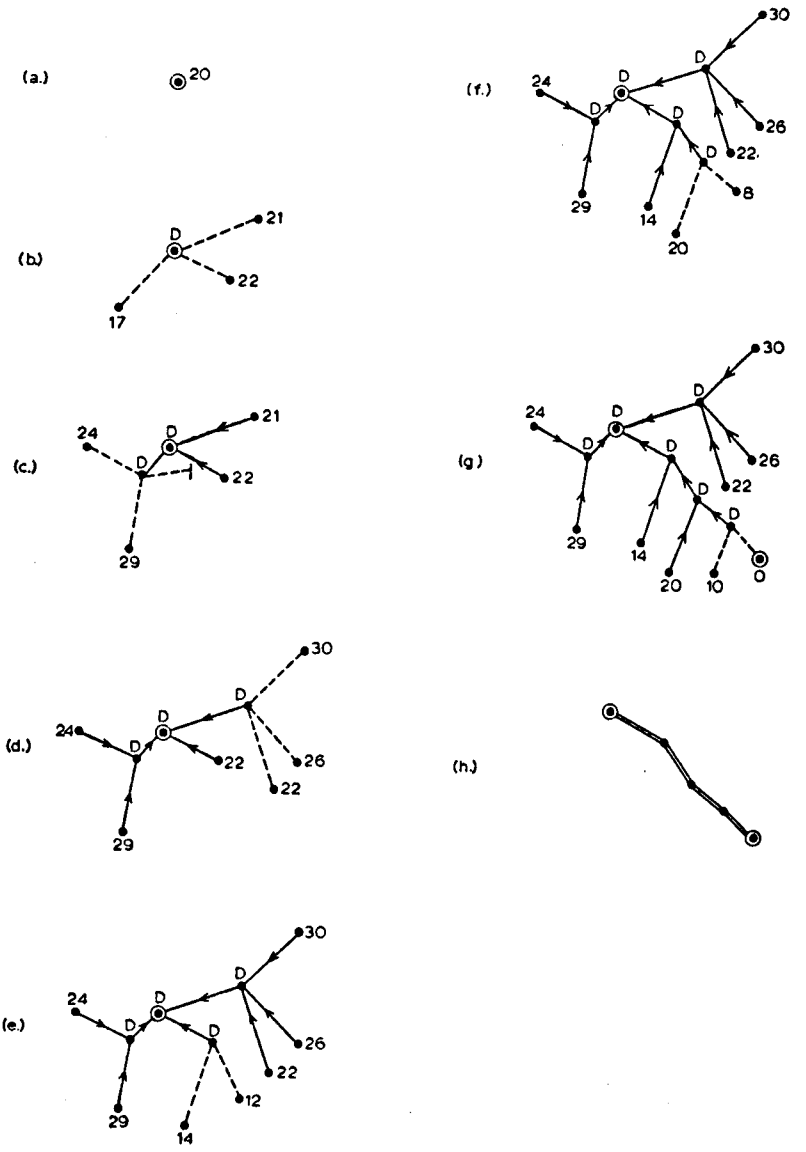

FIG. 1. The Graph Traverser in action. Successive diagrams show a search tree being enlarged by successive developments until the goal is located and a solution path determined. The start and goal nodes are ringed, the symbol 'D' indicates a developed node, and the figures are the values assigned to nodes by the program's (imperfect) evaluation function. (Reproduced from Doran & Michie 1966.)

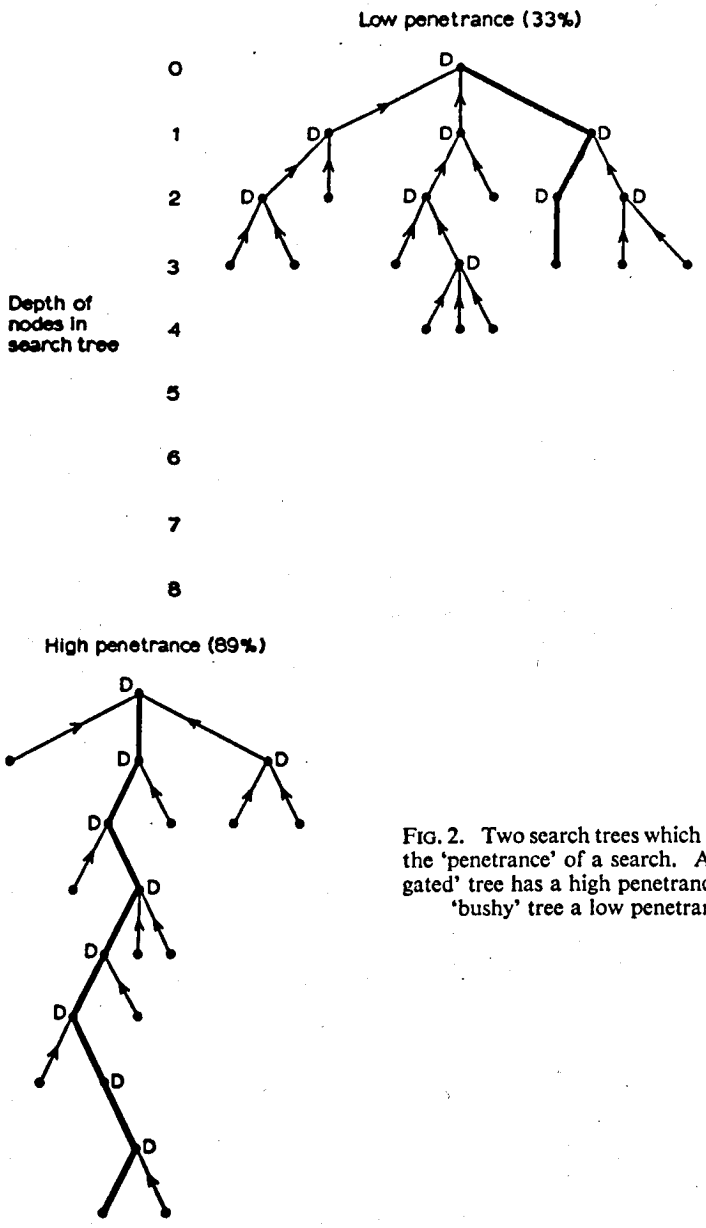

FIG. 2. Two search trees which illustrate the 'penetrance' of a search. An 'elongated' tree has a high penetrance, and a 'bushy' tree a low penetrance.

The internal problem attacked by the Graph Traverser may now be described. It is given a 'problem graph', with two of its nodes labelled *start* and *goal*, to find a path from the start to the goal. Minsky (1961) remarks, 'Almost any problem can be converted into a problem of finding a chain between two terminal expressions in some formal system.'

The graph is defined to the program as a set of distinct integer matrices,

each of which corresponds to a node, together with a procedure 'develop' which given a node lists all the immediate (i.e., one-arc) descendants of that node. The program seeks always to achieve maximum economy, that is, the minimum amount of searching, and is unconcerned as to the length of the path it finds from the start to the goal.

The program's strategy is founded upon the evaluation function. This is embodied in a procedure evaluate which when applied to a node attaches to it a 'value', hopefully correlated with the distance of the node from the goal.

The search proceeds iteratively. At the commencement of an iteration the program has stored the nodes it has so far discovered by repeated application of 'develop', together with the following information about each:

(1) its value, as obtained by applying 'evaluate'; and
(2) a pointer to the node from which it was developed.

The former is required for directing the search and the latter for constructing a path when the search terminates.

The iteration proceeds by finding the 'undeveloped' node with the smallest value (i.e., greatest 'promise') and using 'develop' to find all nodes adjacent to it. Every node *not previously located* is evaluated and stored. The program is thus constructing a 'search tree'. The iteration is now complete.

Initially the program holds one node, the start, and development continues until the goal is located. A procedure is then entered which constructs and prints out a path from the start to the goal. Should the number of nodes stored ('size' of the tree) reach a pre-set limit before the goal is located, the program will select the most promising undeveloped node, and print out a 'partial path' to that. A fresh search is then initiated using the selected node as the new start. The term 'partial search' will be used for the growth of one of a sequence of search trees. It follows from this that search could continue indefinitely. In practice, a 'resignation' criterion is included.

Fig. 1 shows the program at work on a problem graph. To understand what follows, it is essential that the reader understands this diagram. A fuller explanation has been published elsewhere (Doran & Michie 1966).

The crucial importance of the evaluation function in the search should be clear. If the function is constant, so that it conveys no information, then the program's strategy reduces to a systematic search of the nodes of the graph working from the start outwards.

The application package

The internal problem and strategy of the Graph Traverser have now been described, and there remains only to specify the application package.

To apply the program to any particular problem it is necessary:

(1) to write input and output routines which convert the external problem-state into its internal integer matrix representation, and vice-versa;

(2) to write a procedure 'develop' which, given a node, must list all immediate descendants of it so as to reflect the relationships between the external problem-states; and
(3) to write a procedure 'evaluate' which must embody an evaluation function.

In addition various other quantities, such as the dimensions of the node-matrices, the maximum tree size permissible, and variable parameters of the evaluation function, will be set from the data tape.

The Graph Traverser: operational characteristics

The program is written in Elliott Algol. The language provides a high degree of flexibility, not, however, extending to list processing, a restriction which is more felt as the work advances.

The actual programming is, on the whole, naïve. No attempt has been made to produce a particularly efficient program, as it seemed more sensible to concentrate on what the program did, rather than how it did it. One exception to this rule is that, although a node has been stated to be an integer matrix, the program saves space by usually handling nodes in compressed form. Thus there are, in all, three levels of representation of a problem-state:

(1) in whatever form it takes on the data tape;
(2) as an integer matrix; and
(3) in the compressed form as one or more integers.

When results obtained with the program are described, an indication will be given of the machine time required for typical operations. On the NCR Elliott 503 digital computer (using 8000-word core store only) search trees of between 500 and 1000 nodes are the largest which can be handled.

APPLICATION OF THE GRAPH TRAVERSER

An indication only of the program's performance in practice will be given here. For further details see Doran & Michie (1966).

Measures of performance

The three most interesting measures of the program's performance over a particular search of a graph are (1) the length of the path produced (P) (i.e., the number of arcs comprising the final path), (2) the total number of nodes developed (D), and (3) the total number of nodes added to the search tree (T). Since every node on the constructed path except the last must have been developed, it follows that $P \leqslant D$, and obviously $D \leqslant T$. Denote the minimal path length for a given start and goal by P^*. This will also be the minimum number of developments which must be made to find a path from the start to the goal. Then we define the *path efficiency* as P^*/P. Similarly, we define the *development efficiency* as P^*/D.

The calculation of both these efficiencies requires a knowledge of P^*. Typically, however, P^* will be unknown. The ratio P/D, which is the fraction of the total number of nodes developed which are incorporated into the actual path found, *can* always be calculated (even, by using the partial path, when the search has ended without the goal being located), and is of considerable importance. It will be called the 'penetrance' of a search, or partial search, and may be thought of as representing the degree to which the search tree is 'elongated' rather than 'bushy' (see Fig. 2).

The Eight-puzzle and the Fifteen-puzzle

The initial applications of the Graph Traverser have been to certain simply defined sliding block puzzles, which are particularly well adapted to the investigation of evaluation functions.

The Fifteen-puzzle is of this type and is often found as a toy. It consists of fifteen numbered square pieces in a 4×4 frame. One particular configuration of the pieces might be:

4	7	10	12
13	9	3	1
15	14	0	6
2	5	8	11

where the zero represents a space. The problem posed is to convert one given configuration into another by appropriate sliding movements of the pieces. A 'move' is defined as the movement of one piece into an adjacent square.

The Eight-puzzle is a reduced version of the Fifteen-puzzle, and has eight pieces in a 3×3 frame.

For the purposes of this work, arbitrary goal configurations have been defined for these two puzzles. These are:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	0

1	2	3
8	0	4
7	6	5

It should be clear that if nodes are identified with puzzle-configurations and arcs with moves, then these puzzles fit perfectly the schema of the Graph Traverser.

Note that for any particular goal configuration, only half the possible ways of setting up either puzzle lead to a soluble problem. All the configurations attempted by the program are chosen to be soluble. The combinatorial aspects of the Eight-puzzle are examined elsewhere in this volume by P. D. A. Schofield (p. 125).

In a study of human solving of the Eight-puzzle (see Hayes, Michie, Pole & Schofield 1965), it was found that groups of untrained subjects had averages of path efficiency of from 30 to 40 per cent. Practised subjects attain 80 per cent or even 90 per cent.

Tests of the Graph Traverser on the Eight-puzzle were run using an evaluation function involving a single variable parameter w. This was of the form $P+wS$ where P summed the distances of the pieces from their 'correct' location, and where S measured the extent to which the pieces were in their 'correct' sequence.

With the most favourable setting of w, the path efficiency was about 75 per cent, and the development efficiency was about 55 per cent. Furthermore, the penetrance was closely correlated with the development efficiency. This is important, for, as has been mentioned, the penetrance can be calculated where the efficiencies cannot, and is therefore potentially useful as a general measure of the efficiency of a search.

Table 1 refers to a test battery of ten randomly chosen Fifteen-puzzle configurations. Using the best evaluation function so far constructed, the Graph Traverser solves six out of these ten configurations within a maximum

TABLE 1

Results obtained by applying the Graph Traverser to ten randomly selected soluble Fifteen-puzzle configurations, using the best available evaluation function. The symbol '—' implies that the search was terminated, the size of the tree having reached 500 nodes. The penetrances of the unsuccessful searches are shown for purposes of comparison.

	Fifteen-puzzle configurations									
	F(1)	F(2)	F(3)	F(4)	F(5)	F(6)	F(7)	F(8)	F(9)	F(10)
Path length	99	115	84	—	82	—	—	78	—	83
Nodes developed	165	182	99	—	126	—	—	114	—	111
Tree size	394	440	225	—	304	—	—	284	—	266
Penetrance, %	60·0	63·2	84·8	58·1	65·1	48·0	39·3	68·4	36·9	74·8

tree size of 500 nodes. This evaluation function involves several features of a Fifteen-puzzle configuration analogous to those Eight-puzzle features mentioned above.

The approximately constant ratio between the number of nodes developed and the tree size is a consequence of the homogeneity of the problem graph.

To solve a typical Eight-puzzle configuration on an Elliott 503 takes the program about fifteen seconds, and a Fifteen-puzzle configuration requires about four minutes.

Algebraic manipulation

Given a set $\{a, b, c, d, e, f, g, h, i, j\}$ upon which is defined a binary operation '*', we may write expressions of the form

$$(c * (((a * b) * d) * (e * f))).$$

If the operation '*' is commutative and associative, then we may apply these properties to transform one expression into another. For example, by

applying first commutativity and then associativity we may transform the expression

$$((a * b) * c)$$

first into $((b * a) * c)$

and then on into $(b * (a * c))$

The performance of the Graph Traverser when given a simple task of this type is shown in Table 2. Notice again the correlation between the development efficiency and the penetrance as the parameter varies. A very simple evaluation function was used which attached varying weights to the symbols of the expression, and gave as its value the sum of the moments of the weights about the left-hand end. In fact, the expressions can be transformed one into the other in just eight steps.

The most important difference between this simple algebraic manipulation problem and the sliding block puzzles is that the number of immediate descendants of a given node in the former case is much higher. This has the effect of reducing (severely, if the algebraic expression is large) the number of developments possible during the growth of a tree of a given size.

TABLE 2

Results obtained by applying the Graph Traverser to a simple algebraic manipulation problem. An evaluation function with one variable parameter was employed. The symbol '—' indicates failure to solve the problem within a search tree of 500 nodes. (Reprinted from Doran & Michie 1966.)

Starting expression: $((h * (a * f)) * ((c * d) * c))$
'Goal' expression: $(((((a * c) * c) * d) * f) * h)$

	Parameter setting for evaluation function										
	5	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**20**
Path count	—	—	13	9	9	10	10	10	—	—	—
Nodes developed	—	—	46	15	16	17	19	19	—	—	—
Tree size	—	—	268	112	117	124	132	132	—	—	—
Penetrance, %	24·4	11·5	28·3	60·0	56·3	58·8	52·6	52·6	18·2	13·6	10·2

FUTURE DEVELOPMENT OF THE GRAPH TRAVERSER

The remainder of this paper will be devoted to a discussion of some of the ideas and facilities which may become part of the Graph Traverser during further work. The material will range from work currently under way to speculation.

One important topic will *not* be considered except briefly at this point. The difficulty noted above, that in some circumstances a node will have inconveniently many immediate descendants, leads us to an important observation about the Graph Traverser. As so far described, it can recognise and manipulate the possible *states* of a problem, but not the transformations, or *operators*, which change those states. Thus it manipulates Eight-puzzle *configurations*, but not, in other than a trivial sense, the *moves* which connect

those configurations. Yet, commonly, an operator will be capable of definition quite independently of any particular problem state. Even in the Eight-puzzle we may define the operator 'north-move', say, which, where applicable, moves into the empty cell the piece immediately to its 'north'. Given any Eight-puzzle configuration it is clear whether 'north-move' is applicable, and, if so, what its action is.

Once manipulation of operators is possible, then the number of immediate descendants of a node may be reduced, effectively, by applying during the development of the node only a 'promising' subset of the operators actually applicable.

This topic is discussed in the article by Dr Donald Michie in this volume.

Range of application

The applications of the Graph Traverser to date have been within a restricted class of problem, for reasons already indicated. However, more attention is now being paid to the generality of the program, and, for example, the algebraic manipulation experiments represent merely a first attempt in this sphere (see also R. J. Popplestone's application of the program to a symbol-manipulation problem in this volume (p. 31)).

One of the weaker aspects of the Graph Traverser in its role as a general problem-solving program is the magnitude of its application package. To apply the program to a problem unlike any previously attacked may require a considerable programming effort. This situation cannot be improved in any simple way. The program must be equipped with a general problem specification such that the user may indicate his own particular problem with the minimum of effort. This is not easy. The work of Raphael (1964) and Bobrow (1964) already mentioned is relevant at this point.

Considerable extensions to the range of applicability of the program come from the realisation that its internal problem can be significantly altered with very little modification to the program itself.

Firstly, we may easily remove the requirement that the goal node be fully specified, and replace this by a 'goal recognition' rule. This corresponds, for example, to a sliding block puzzle where the 'goal' requires a subset of the blocks to be in particular locations irrespective of the positions of the others.

Secondly, we may specify that the path is *not* of primary importance, but that the main concern is to find a node which satisfies the 'goal recognition' rule. This situation seems appropriate to a range of applications including transportation problems, timetabling and allocation problems generally. The structure of the problem graph is now likely to be a matter of definition, rather than inherent in the problem itself.

Strategies and problem graphs

If the Graph Traverser's internal problem changes, then clearly its strategy

should change correspondingly. Why go to trouble to construct a solution path, when no path is required? The reader may have noticed that in each of the examples of the program at work, the problem graphs have been *symmetric*. When solving the Eight-puzzle, the program should perhaps have grown a search tree from the *goal* simultaneously with that grown from the start. This is an example of a more general point. The program assumes for the purpose of its search that the problem graph is quite general. However, the problem graph may well be symmetric, or a tree, or structured in some other way.

This raises two possibilities:

(1) that the program should be *told* when the problem graph has some property which will make the search easier; and
(2) more ambitiously, that the program should grow a small search tree, and then before continuing, inspect this tree for useful properties which it would then provisionally assume to hold for the entire problem graph.

Possibility (1) will be implemented in some form in the foreseeable future. (2) is much more complex, and will not be pursued further here, except to remark that in some circumstances it may be advantageous for the program to solve a problem under an assumption which is in fact not entirely sound, and then to 'correct' the solution.

'Pruning' schemes

There is a limit to the size which the search tree of the Graph Traverser can attain. Either this will be fixed by the storage capacity of the machine upon which the program is running, or else it will be determined by the time taken to work with a large tree. The reaction of the program when all the available storage space has been filled by the tree has already been described: a partial path is fixed upon, all the tree except a single node is erased, and a new partial search begins. This is unnecessarily catastrophic. There is no need to throw away so much hard-won information and it seems foolish to do so. The latest version of the Graph Traverser commits itself only to a single additional step along the path (or some specified number of steps), and only that part of the tree thereby rendered valueless is discarded. The distinction is illustrated in Fig. 3, where the significance of the diagrams is as follows:

(*a*) The search tree at its growth limit (12 nodes).
(*b*) The old version of the program ('static' tree) commits itself to a path to the undeveloped node with least value. This partial path is output. The new version of the program ('dynamic' tree) takes just one 'step', and this is output.
(*c*) Deletion occurs. The dynamic tree version deletes only that part of the tree dependent on the portion of path selected.

(*d*) Two developments later. The situation illustrates how the old version may make a mistake.

Of course, the dynamic tree version of the program has to go through the 'pruning' process fairly frequently, and this is a disadvantage.

Table 3 presents the results of a simple experiment to compare the performance of the two versions of the program. Twenty-four Eight-puzzle configurations were used for this purpose. Twelve of these (test battery A) each required at least 30 moves for solution, this being the longest possible minimal solution path. The remainder (test battery C) were randomly chosen. The maximum size of the tree was fixed at 40 nodes, and the extent of the 'dynamic' pruning was determined by deciding that at each pruning operation a path segment three nodes long was to be output.

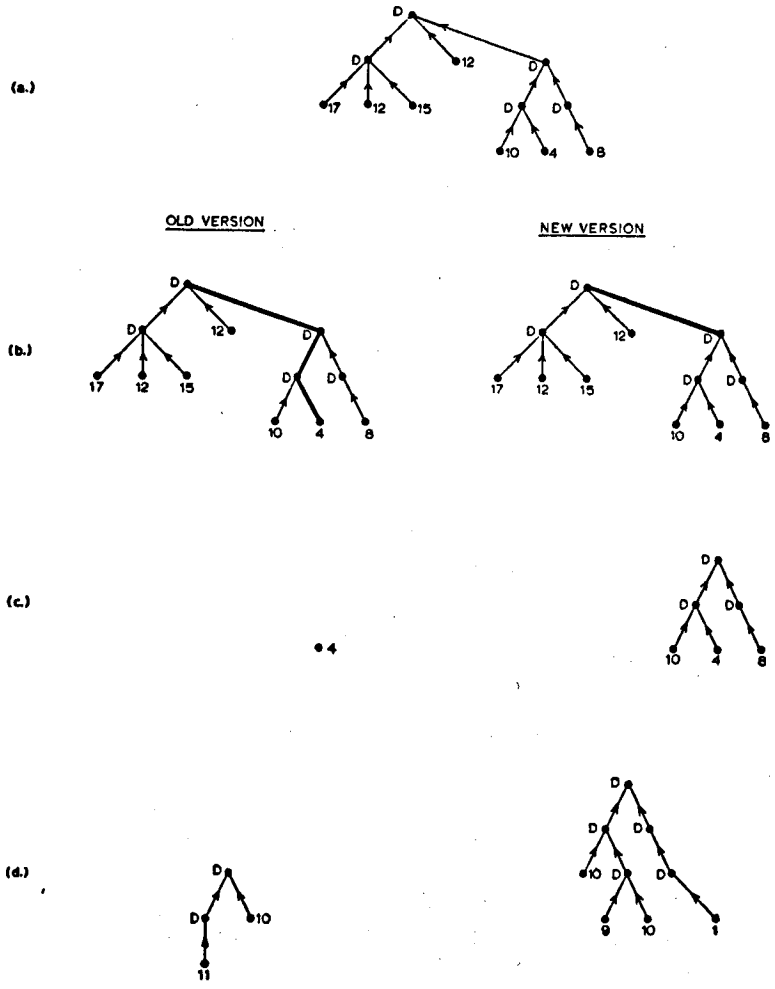

FIG. 3. Illustration of the operation of a 'dynamic' search tree as used by the latest version of the Graph Traverser. Earlier behaviour is shown for comparison.

TABLE 3

Performance of the old ('static' tree) version of the Graph Traverser compared with the new ('dynamic' tree) version over 24 Eight-puzzle configurations. The maximum tree-size permitted was 40 nodes, and a search was terminated (indicated by '—') when more than 200 developments had taken place without the goal having been located. Figures in brackets are minimal pathlengths. The evaluation function was not varied.

Test battery 'A'	'Static' tree		'Dynamic' tree	
	Nodes developed	Path length	Nodes developed	Path length
A1(30)	—	—	—	—
A2(30)	65	42	63	40
A3(30)	48	42	52	42
A4(30)	64	42	70	42
A5(30)	—	—	72	44
A6(30)	48	42	52	42
A7(30)	—	—	126	62
A8(30)	84	58	89	54
A9(30)	36	32	39	32
A10(30)	—	—	150	56
A11(30)	31	30	33	30
A12(30)	42	32	45	32
Test battery 'C'				
C1(20)	34	32	34	28
C2(24)	43	34	45	34
C3(22)	79	38	77	38
C4(22)	—	—	—	—
C5(24)	54	38	54	34
C6(24)	31	28	32	28
C7(22)	28	24	29	24
C8(14)	26	16	26	16
C9(22)	42	28	39	28
C10(24)	—	—	60	36
C11(8)	8	8	8	8
C12(18)	27	20	27	20

It will be seen that where a configuration was solved in both modes, performance was virtually identical. However, of the six configurations not solved by the 'static' version of the program four were solved by the 'dynamic' version, and none was solved by the former but not by the latter. Thus there is some evidence that the 'dynamic' tree is of real value just at the point where the old approach falters. As yet, of course, the evidence refers only to the Eight-puzzle.

It is easy to imagine, though less easy to implement, more complicated pruning schemes. For example, it would be possible to delete high-valued (i.e., unpromising) branches without committing the program to any additional path segment.

Program improvement of evaluation functions

Can the Graph Traverser improve its own evaluation function? If it can be made to do this, then the problems posed by the program's limited memory will have been attacked in a new way since the evaluation function itself will have become a repository of acquired knowledge.

Suppose that the program has been given a sequence of similar but not identical problems to solve, each problem involving several partial searches. It is required to find the 'best' evaluation function while solving the first few problems, and then to use this to solve the remainder. By 'best' evaluation function I mean here merely 'that setting of the parameters of the current function which optimises its performance over this class of problem'. Suppose that at the end of each partial search an adjustment of the function parameters is to be made. How can this be done so as to converge on the best function?

One attempt to answer this question runs as follows. If the program can calculate from a partial search a 'figure of merit' indicating how 'good' was the evaluation function which controlled that search, then it can seek to optimise the function by a standard hill-climbing operation. That is, it can carry out a sequence of partial searches with varying settings of the function parameters, and use its 'figure of merit' to converge on the optimal settings. The situation will typically be 'noisy', that is, the figure obtained will vary over different partial searches even for a fixed evaluation function.

The penetrance has already been mentioned as potentially useful as a general measure of the efficiency of a search—intuitively, it measures how 'confident' the search has been, independently of the tree size and the structure of the problem graph. Can the penetrance of a search be used as a 'figure of merit' in the sense introduced above?

We have seen that, when a search has been successfully completed, the penetrance has been strongly correlated with the development efficiency. To some extent this is to be expected—if the development efficiency is high, then the penetrance *must* be. However, the converse need not be true—it is possible for a search to have a high penetrance, but to be circuitous. For a small problem graph this hazard is slight, for the evaluation function can usually be made to take a minimum at the goal node giving little opportunity for deviation from the true path to go unrecognised. If the problem graph is large, however, so that several partial searches are needed, it seems quite possible that a partial search should have a high penetrance, but in fact be quite misdirected and therefore controlled by a poor evaluation function.

There is some experimental evidence on this point. The test battery of Fifteen-puzzle configurations already mentioned was attacked by the Graph Traverser with a number of different parameter settings for the evaluation function. Each attempt at a configuration involved at most two partial searches, each with a maximum tree size of 250 nodes. When mean penetrance over the first partial search was plotted against the proportion of configura-

tions solved within the second, a strong positive correlation was found, suggesting that here at least the penetrance was a guide to the effectiveness of the evaluation function.

Distance estimators

In the last section I discussed one possible answer to the question: 'How can the Graph Traverser adjust the parameters of its evaluation function at the end of each partial search so as to converge on the optimum settings?'

Another approach to this problem starts from the idea that the evaluation function, in the rather restricted sense in which it is being used here, is a particular case of something a little more general. As we have used it, an evaluation function effectively estimates the distance from a node to the goal node. Let us now define a *distance estimator* as a function which, when applied to any two nodes, specifically estimates the distance over the graph from the first to the second.

The purpose of this definition is to transfer attention from the '*goal seeking*' ideas associated with an evaluation function to '*graph describing*' ideas associated with the concept of a distance estimator. Notice that we may discuss good or bad distance estimators for a particular graph quite independently of any problem associated with that graph. Nevertheless, given a good distance estimator, *a fortiori* we have a good evaluation function for use in any particular search over the graph.

Now, if a distance estimator is required for a particular graph, but the information available is, in fact, merely a small fragment of the graph itself together with evidence that the graph is relatively 'homogeneous', then it is natural to find that estimator which best 'fits' the graph fragment available for inspection, and then to adopt this estimator for the graph as a whole.

But this is exactly the situation in which the Graph Traverser finds itself at the end of a partial search. The program has available its search tree, which is effectively a 'fragment' of the problem graph, and it is seeking a good distance estimator, as this automatically provides a good evaluation function. It is plausible therefore that initially the program should be provided with a rudimentary distance estimator which, at the end of each partial search, should be modified until it best 'fits' the search tree.

One specific method of performing such a 'fitting' process would begin by labelling each node of the tree with its 'depth' (as defined in Fig. 2). The depth of a node is the distance over the tree from the 'root' node to it. The program would then adjust the distance estimator until the correlation between these tree distances and the corresponding estimated distances given by the estimator was maximised. The tree distances that would be used in this calculation are not necessarily exactly the true distances over the problem graph. The method must assume, however, that they are nearly so.

Having obtained the distance estimator which best 'fits' the search tree, the program would then, as indicated above, use it as an evaluation function to guide the next partial search.

It is worth stressing the important practical difference between this approach to evaluation function optimisation, and that based on the penetrance. Using the penetrance, optimisation can take place only over a potentially rather long sequence of partial searches. It is hoped that the distance estimator approach, however, will permit considerable progress to be made *after just one partial search*, and that in general the optimisation process will be much speeded.

CONCLUSION

My aim has been to discuss problem-solving programs in a rather general way, and to describe in some detail the Graph Traverser program, its progress and its possible future. The Graph Traverser is not yet much more than an elementary exercise in machine intelligence. Even so, it will be instructive to end by seeking in the program three of the features of general problem-solving programs mentioned in the introduction.

Planning (see Minsky 1961) involves using additional information about a problem to obtain an 'outline' solution which will provide intermediate 'subgoals' for the search proper. This process does not appear explicitly in the Graph Traverser. Yet, in a rather vague sense, an evaluation function may be regarded as a generalisation of the notion of subgoals. Furthermore, the concept of operator-synthesis, the concatenation of several operators to form a macro-operator (Michie in this volume), is close to the type of planning which occurs in Gaku (Hormann 1965).

Induction, or the formation of a general rule from particular instances, will occur whenever the Graph Traverser seeks to draw general conclusions from the properties of some particular part of the problem graph. Relevant here are both the proposed improvement of an evaluation function by the study of a search tree, and also the suggestion that the program might induce structural properties of the problem graph from an inspection of a small portion of it.

Program-user interaction has as yet no place in the Graph Traverser. Yet it seems very plausible that problem-solving programs and human problem solvers are destined to work in co-operation rather than in isolation. The procedures 'develop' and 'evaluate' of the Graph Traverser could each involve interaction with an 'on-line' user. The value of such collaboration in actual problem-solving is difficult to judge, but the experiment would be interesting.

Indeed, the most useful role of the Graph Traverser is perhaps that of a 'test bed', not only for evaluation functions but for a wide range of ideas important to automatic problem-solving.

ACKNOWLEDGEMENTS

The research described in this paper is supported by the Science Research Council. I am greatly indebted to Dr Donald Michie, with whom I have carried out most of the experiments described here. He is the originator of

many of the better ideas which the paper contains. I am also grateful to my colleague Dr. R. M. Burstall for helpful criticisms and comments.

REFERENCES

Berge, C. (1962). *The theory of graphs and its applications.* Alison Doig, Trs. London: Methuen.

Bobrow, D. G. (1964). A question-answering system for high school algebra word problems. *AFIPS Conference proceedings*—1964 *Fall Joint Computer Conference*, Pt. 1, pp. 591-614. Baltimore: Spartan Books.

Doran, J. E., & Michie, D. (1966). Experiments with the Graph Traverser program. *Proc. R. Soc.* (*A*), **294**, 235-259.

Feigenbaum, E. A., & Feldman, J., Eds. (1963). *Computers and Thought.* New York: McGraw Hill.

Hayes, J. E., Michie, D., Pole, K. E., & Schofield, P. D. A. (1965). A quantitative study of problem-solving using sliding block puzzles: the Eight-puzzle and a modified version of the Alexander Passalong Test. *Experimental Programming Reports.* No. 7. Edinburgh: Experimental Programming Unit, University of Edinburgh.

Hormann, A. (1965). Gaku: an artificial student. *Behav. Sci.*, **10**, 88-107.

Minsky, M. (1961). Steps towards artificial intelligence. *Proc. Inst. Radio Engrs.*, **49**, 8-30.

Newell, A., & Ernst, G. (1965). The search for generality. *Information Processing* 1965: *Proceedings of IFIP Congress* 1965. Vol. 1, pp. 17-24. Kalenich, W. A., ed. London: Macmillan.

Newell, A., Shaw, J. C., & Simon, H. A. (1960). A variety of intelligent learning in a general problem solver. *Self-organising systems.* Yovits, M. C., & Cameron, S., eds. pp. 153-189. London: Pergamon Press.

Raphael, B. (1964). A computer program which 'understands'. *AFIPS Conference proceedings*—1964 *Fall Joint Computer Conference.* Pt. 1, pp. 577-589. Baltimore: Spartan Books.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *I.B.M. Jl. Res. Devl.*, **3**, 211-229. Reprinted in Feigenbaum and Feldman (1963).

Slagle, J. R. (1963). A heuristic program that solves symbolic integration problems in freshman calculus. *Computers and Thought.* Feigenbaum, E. A., & Feldman, J., eds. pp. 191-203. New York: McGraw Hill.