Experiments with the Graph Traverser program

By J. E. Doran and D. Michie

Experimental Programming Unit, University of Edinburgh

(*Communicated by D. G. Kendall, F.R.S.—Received 2 February* 1966)

An automatic method is described for the solution of a certain family of problems. To belong to this family a problem must be expressible in the language of graph theory as that of finding a path between two specified nodes of a specified graph.

The method depends upon the evaluation of intermediate states of the problem according to the extent to which they have features in common with the goal state. We define evaluation functions each of which assigns to any state of the problem a value which is in some way related to its 'distance' from the goal state. Equivalently we assign to nodes of the corresponding graph values which are related to the distance over the graph from the goal node. Distance is reckoned as the smallest number of arcs needed to connect two nodes.

An Algol program, the Graph Traverser, has been written to operate in this context. It is designed in a completely general way, and has two 'empty' procedures one of which must be written to specify the structure of the graph, that is the constraints of the problem, and the other to define an evaluation function.

Results obtained by supplying the program with definitions of various sliding block puzzles and also a simple problem of algebraic manipulation are reported for a range of evaluation functions.

Introduction

A heuristic method is one that seeks to obtain 'good' solutions for a small fraction of the cost which would be involved in obtaining optimal solutions. To take for illustration one example among many, Burstall (1966) has described a heuristic program for the design of electricity distribution networks. Optimal solutions to this problem can be obtained by integer linear programming. However, the calculations required to apply this method increase exponentially with the number of stations in the network, and if this number exceeds nine or ten, the method becomes impracticable. Burstall's program can process networks with 16 stations, and generates solutions which, although not always optimal, compare well with those obtained by experienced human designers.

An admirably clear review of research in heuristic methods has recently been presented by Newell & Ernst (1965), who survey work relevant to the characterization of general problem-solving procedures. They limit their treatment, as does this paper, to those problems which are susceptible of a particular formal representation, namely, a set of discrete states to which may be applied a set of permitted transformations ('moves', 'operators'). Thus stated, the task is to find a sequence of transformations which will convert some initial state into a final state, or goal. The history of work in this general area has shown a certain tendency to polarize around two distinct approaches. At one pole, attention is concentrated upon the *evaluation of states*, while at the other pole the emphasis is upon the *selection of operators*. In the first case we ask: 'To which state shall we next apply operators?' In the second case we ask: 'Which operator is to be applied next to this state?'

The first preoccupation has been characteristic of much of the work on automatic game-playing; indeed Turing's original proposal of the idea was in the context of his experiments with chess-playing automata (Turing 1953; Michie 1966). A standard method (see Samuel 1960) has been to work out all possible combinations to a fixed number of moves ahead, evaluate, according to some strategic features, all the board positions generated in this way, and use these evaluations to trace a path back to the current position. This path is used to define the next move to be selected, as shown in figure 1.

The work based primarily on *operator selection* is typified by the studies made by Newell, Shaw & Simon (1960) with their 'General Problem Solver' program. Here evaluations are made only to the crude degree necessary to define an ordered set of

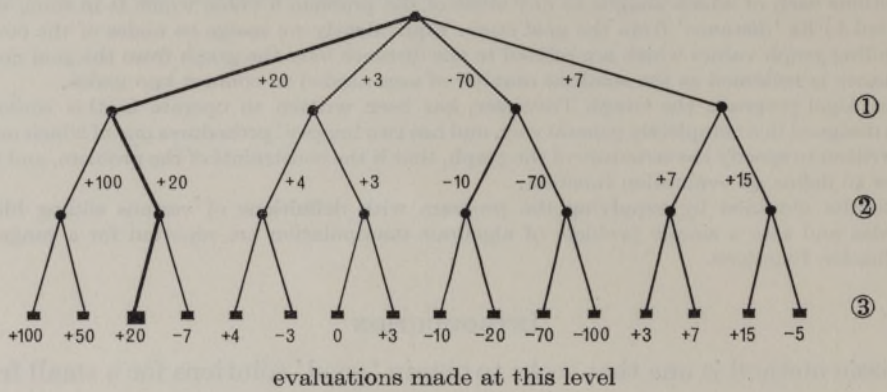

FIGURE 1. Simplified diagram showing how evaluations are backed-up through the 'tree' of possible moves to arrive at the best next move. The process starts at level (3) by assigning scores (high if favourable, low if not) to the board positions inspected. At level (2) the score assigned to a position is the maximum of the scores of the descendant positions (machine moves seek to *maximize* the score). At level (1) the score assigned is the minimum of the scores at the lower level (opponent's moves seek to *minimize* the score). At the top of the diagram the machine selects the move leading to the highest score, thus doing the best it can allowing for the action of the opponent.

intermediate goals (subgoals). Operator selection is then applied by asking about each operator in turn 'does it directly promote subgoal 1? If not, does it promote subgoal 2?, ..., etc.'

The present work constitutes the first stage of an attempt to bind these two basic procedures into a unified framework. This framework takes the form of a computer program which we call the Graph Traverser. The results reported here are concerned exclusively with state-evaluation, and the means whereby a search based upon this principle alone may be organized efficiently. But indication is also given of lines along which the program might be enabled to improve its own evaluations.

Although we have used sliding block puzzles to investigate heuristic principles, these puzzles are not the point of interest of the work: on the contrary our approach, and the Graph Traverser program which implements it, claims a wide generality. As an illustration of this generality we include in our report some preliminary data gained by presenting the program with an exercise in elementary algebraic

manipulation. We should also mention in this context some recent work by our colleague Popplestone (1966) who has found the program useful in his study of heuristic methods in elementary group theory.

While the logical design and experimental development of the Graph Traverser has been conducted jointly the programming itself has been the work of one of us (J. E. D.).

Problems and graphs

A problem, of the type with which we are concerned, is a one-person game. In contrast to two-person games, the state of the game remains undisturbed between the player's successive moves. A convenient formal representation is that in which

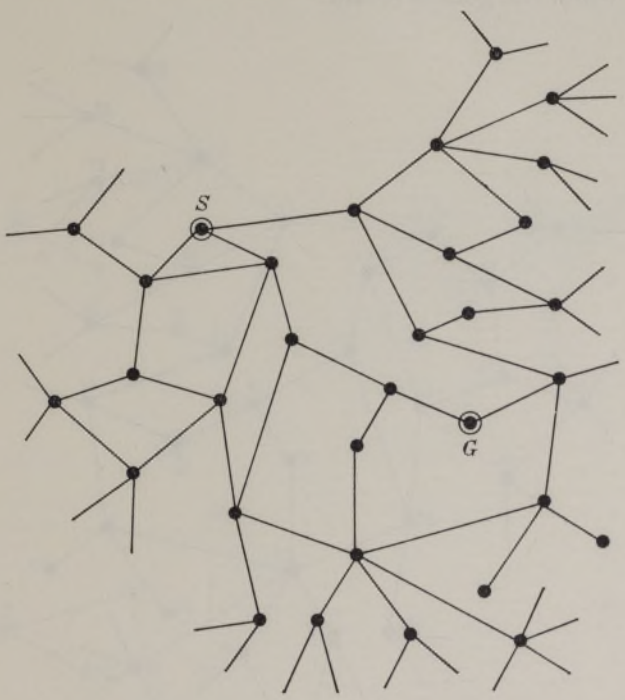

FIGURE 2. Part of a symmetric graph with start (S) and goal (G) nodes marked.

a game is identified with a graph in which the nodes represent states, and the arcs represent permitted transitions (legal moves). A graph in the mathematical sense may be thought of as a set of nodes some of which are connected to some others by arcs which may be directed (oriented graph) or undirected (symmetric graph). Figure 2 shows part of a symmetric graph (i.e. arcs represented by lines rather than arrows).

The task is to find a path across the graph from the start to the goal as economically as possible, i.e. with as little labour as possible expended in the search, avoiding, as far as possible false trials, blind alleys and meanderings far from the final path. If the path is short, we say that the solution is 'elegant'. If the search was short, we say that the solution is 'economical'.

The Graph Traverser program

The program, written in Algol, has three main characteristics:

(1) It can be applied to any problem which can be translated into the abstract 'graph traversal' terms specified in detail below.

(2) In its present version it seeks always to achieve maximum economy, that is minimal search, and is satisfied to find any path consistent with this.

(3) In order to carry out its search it must be *given* an evaluation function which enables nodes of the graph to be evaluated according to their estimated distance from the goal. If the evaluation function is constant and therefore contains no information, then the strategy of the program reduces to systematic enumeration terminating only when the goal is found.

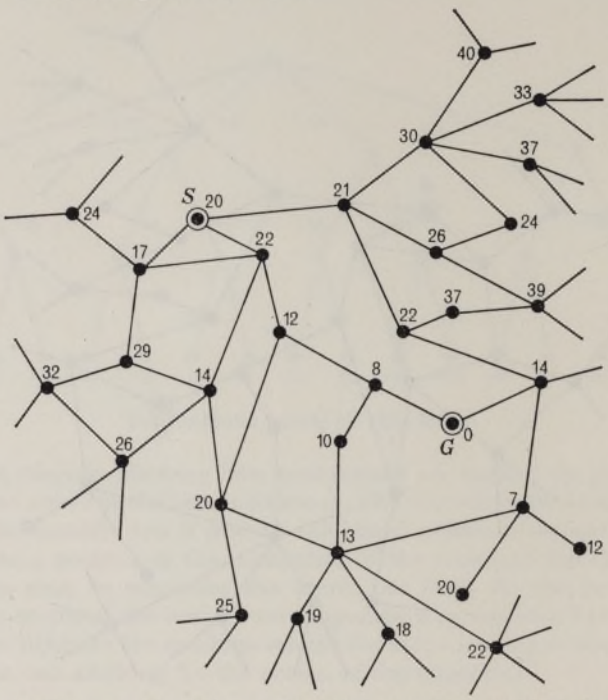

FIGURE 3. The graph of figure 2 with values attached to the nodes.

The particular graph to be investigated is specified to the program by means of a procedure 'develop' which, given a node, produces a list of all nodes adjacent to the given node. For the purposes of this program, which is subject to the restrictions of the language Algol 60, a node, which represents a problem state, is an integer matrix. Two nodes must be specified to the program as the start and the goal. In addition, a procedure 'evaluate' must be made available which, given a node, applies some evaluation function and delivers the corresponding value.

The search proceeds iteratively. At the start of an iteration the program has stored the nodes it has so far discovered, 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.

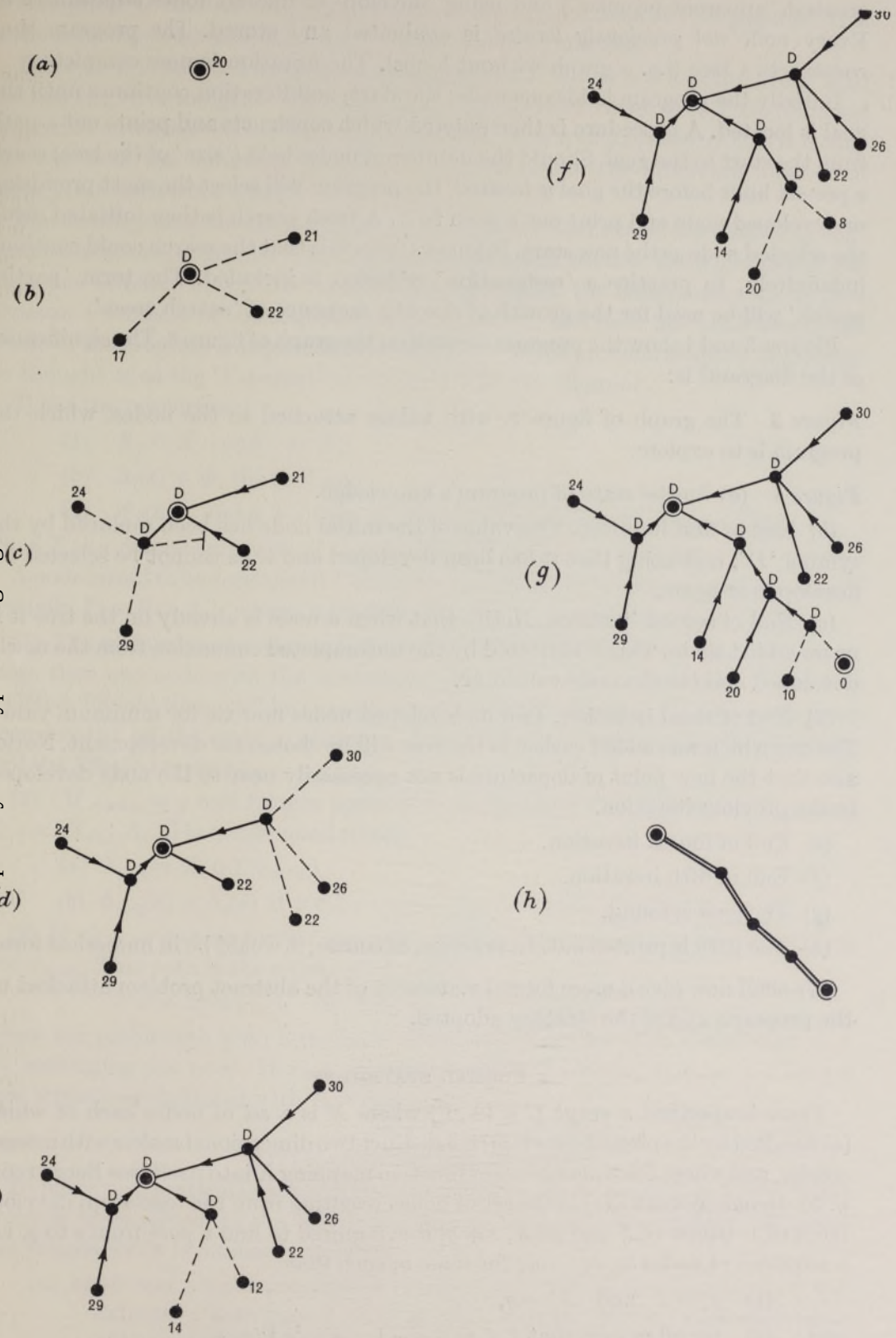

FIGURE 4. The Graph Traverser searching the graph of figure 2.
For commentary see text.

greatest 'apparent promise') and using 'develop' to find all nodes adjacent to it. Every node *not previously located* is evaluated and stored. The program thus constructs a tree (i.e. a graph without loops). The iteration is now complete.

Initially the program holds one node, the start, and iteration 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 held ('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 path to it. A fresh search is then initiated using the selected node as the new start. It follows from this that the search could continue indefinitely. In practice a 'resignation' criterion is included. The term 'partial search' will be used for the growth of one of a sequence of 'search trees'.

Figures 3 and 4 show the program at work on the graph of figure 2. The significance of the diagrams is:

Figure 3 The graph of figure 2, with values attached to the nodes, which the program is to explore.

Figure 4 (*a*) Initial state of program's knowledge.

(*b*) End of first iteration. The value of the initial node has been replaced by the symbol '*D*', indicating that it has been developed and thus cannot be selected for development again.

(*c*) End of second iteration. Notice that when a node is already on the tree it is never added again. This is indicated by the uncompleted connexion from the newly developed node to the node valued 22.

(*d*) End of third iteration. Two undeveloped nodes now tie for minimum value. The one which was added earlier to the tree will be chosen for development. Notice also that the new point of departure is not necessarily near to the node developed in the previous iteration.

(*e*) End of fourth iteration.

(*f*) End of fifth iteration.

(*g*) The goal is found.

(*h*) The path is printed out. In practice, of course, it would be in numerical form.

We shall now give a more formal statement of the abstract problem attacked by the program, and of the strategy adopted.

Formal statement

There is specified a *graph* $G \equiv \{X, \Gamma\}$ where X is a *set* of nodes each of which (as handled by the present program) is a distinct two dimensional *matrix* with integer entries, and where Γ is a *many-valued* function mapping X into itself (see Berge 1962, p. 5). In our context $\Gamma(x)$ is the set of nodes resulting from the (one-step) development of x. Given $s \in X$ and $g \in X$, $s \neq g$, it is required to find a *path* from s to g, i.e. a sequence of nodes $x_1, x_2, \ldots, x_n$ for some n, such that

(1) $x_1 = s$ and $x_n = g$,

and (2) for all m such that $1 \leqslant m \leqslant n-1$, $x_{m+1} \in \Gamma(x_m)$.

It is required to find such a sequence with as few applications of Γ as possible. This is the *economy* condition. Note that G need not be symmetric.

The search is assisted by an *evaluation function* E which is a function from X to the non-negative integers. The values taken by $E(x)$ as x varies over X are intended to be rank-order correlated with the 'distances' from x to g, where by this is meant one less than the minimum possible number of nodes (the minimum number of 'arcs') in a path from x to g.

The strategy implemented by the program uses Γ and s together with E to construct a sequence of graphs $T_i \equiv \{X_i, \Delta_i\}$ each of which is a *tree*. Each member of the sequence of trees, except the first, is constructed by the enlargement of its predecessor (members of the sequence do not coexist), and each has built into it the whole of the program's acquired information about the problem graph G. $\Delta_i(x)$ may be thought of as the 'Γ-parent' of the node x in the ith graph.

T_i has the properties:

(1) $X_i \subset X$ and $s \in X_i$;

(2) $\Delta_i(s) \equiv \Phi$, the null set;

(3) if $x \in X_i$ and $x \neq s$ then $\Delta_i(x) \equiv \{y\}$ where y is such that $x \in \Gamma(y)$.

Thus T_i is a partial subgraph of G *except that the arrows are reversed.*

A node is said to be *developed* if Γ has been applied to it and *undeveloped* otherwise. Initially $T_0 \equiv \{\{s\}, \Delta_0\}$, where $\Delta_0(s) \equiv \Phi$. T_{i+1} is constructed from T_i as follows:

(1) The undeveloped node $x \in X_i$ for which $E(x)$ is least is found. Should there be more than one node with the minimum value, that which was earliest located is selected. Should there still be a tie—and this can occur if the nodes concerned have been located by the same application of Γ—then an arbitrary selection is made. Call the selected node $x_{\min.}$.

(2) If $x_{\min.} \neq g$ *and* further space is available then Γ is applied to $x_{\min.}$ and $T_{i+1} \equiv \{X_{i+1}, \Delta_{i+1}\}$ is constructed where

(*a*) $X_{i+1} \equiv X_i \cup \Gamma(x_{\min.})$

and (*b*) $\Delta_{i+1}(x) \equiv \Delta_i(x)$ if $x \in X_i$, and $\Delta_{i+1}(x) \equiv \{x_{\min.}\}$ if $x \in X_{i+1} - X_i$.

(3) If $x_{\min.} = g$ *or* no further space is available then a path is constructed from s to $x_{\min.}$. This path is the sequence of nodes

$$s = \Delta_i^m(x_{\min.}), \quad \Delta_i^{m-1}(x_{\min.}), \quad \ldots, \quad \Delta_i(x_{\min.}), \quad x_{\min.}$$

where the pathlength is m; Δ_i(node) is interpreted here as a node rather than as a set containing one node. If $x_{\min.} = g$ then search terminates. If $x_{\min.} \neq g$ then a new sequence is initiated with $T_0 = \{\{x_{\min.}\}, \Delta_0\}$.

Application of the program

To make any particular application, a translation must be made from the 'real' problem to be solved, to the 'ideal' problem embedded in the program. To make this translation it is necessary to write:

(*a*) input and output procedures which control the relationship between the external ('data tape') and internal ('integer matrix') representations of the problem states;

(*b*) a procedure, 'develop', specifying the graph function Γ; and

(*c*) a procedure 'evaluate', specifying the evaluation function E.

It is also necessary to adjust certain data tape parameters which set bounds to the graph set X. Also specified via the data tape are:

(*a*) the starting and goal nodes;

(*b*) the total number of locations available for storage of the tree, subject to a limitation imposed by the machine size;

(*c*) the severity of the resignation criterion.

The role of the evaluation function

For a given application, 'evaluate' may be changed at will, and certain adjustments to this function will usually be possible via the data tape. The function given to the program can be 'infallible' or 'useless' or 'worse than useless'. In the first case its rank correlation with the distances over the graph will be unity, in the second case zero, and in the third case negative. The fallibility of the evaluation function determines the search economy, measuring this in terms of the number of applications of the procedure 'develop' needed to find a path from the start to the goal.

Search economy is only fully defined in these terms if viewed from the point of view of the program. Overall economy also involves minimizing the 'cost' of each application of 'develop', as well as minimizing the number of times this cost is incurred. In practice this relates to the simplicity in some sense of the evaluation function. In the present form of the program this brand of economy is entirely the responsibility of the user.

We now consider members of a restricted class of puzzles known as sliding block puzzles, starting with the eight-puzzle. This puzzle will provide our first illustration of the action of the Graph Traverser program.

The eight-puzzle

The eight-puzzle is one of a large class of sliding block puzzles, in which the solver is typically required to manipulate square or rectangular objects on a bounded plane so as to rearrange them into some specified configuration. Gardner (1964, 1965*a*, *b*, *c*) has devoted some stimulating discussions to these puzzles.

An early and famous example to which we shall return later in this paper is the fifteen-puzzle, consisting of fifteen numbered square pieces set in a 4×4 array, one cell of the array being empty. The eight-puzzle is a simpler member of the same family, there being only eight numbered pieces set in a 3×3 array. We shall arbitrarily define the goal configuration as follows:

1 2 3
8 0 4
7 6 5

denoting the empty square by a zero. Before proceeding further, two points should be noted.

(1) Half the possible ways of setting up the puzzle are soluble and half are not. Solubility implies that a sequence of moves *can* be found which takes the starting configuration into the goal configuration. Equivalently, solubility implies that a path connecting the starting and goal nodes of the puzzle graph does exist. In the present instance it demands that the starting configuration should be an even permutation of the goal configuration (Johnson & Story 1879; Tait 1880). We shall only concern ourselves with the subset of soluble configurations, which can collectively be represented by a *connected* graph.

(2) The puzzle looks easy, but it is not. Three groups of subjects, about a dozen in each group, were given a battery of mental tests, including the five eight-puzzle subtests shown in figure 5 (Hayes, Michie, Pole & Schofield 1965). The group averages for efficiency of solution ranged from 30 to 40 %, where the *path efficiency* of a solution is defined as:

$$\frac{\text{minimum possible path length}}{\text{actual path length}}.$$

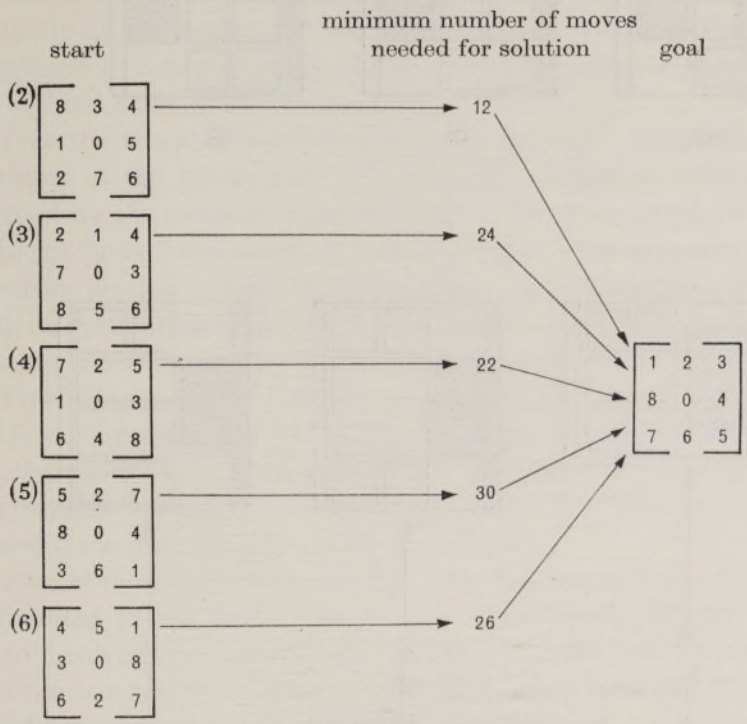

FIGURE 5. Five eight-puzzle starting configurations used for testing human performance.

Efficiency was not affected in any way by the length of the minimal path, except for very short minimal paths ('easy puzzles'). These results were obtained with subjects to whom the puzzle was entirely unfamiliar. In unpublished further observations of the improvement of efficiency with practice, the best subjects attained path efficiencies exceeding 70 %. When we ourselves, and a number of colleagues familiar with the puzzle, were given randomly chosen configurations to solve, efficiencies ranged from 70 to 90 %.

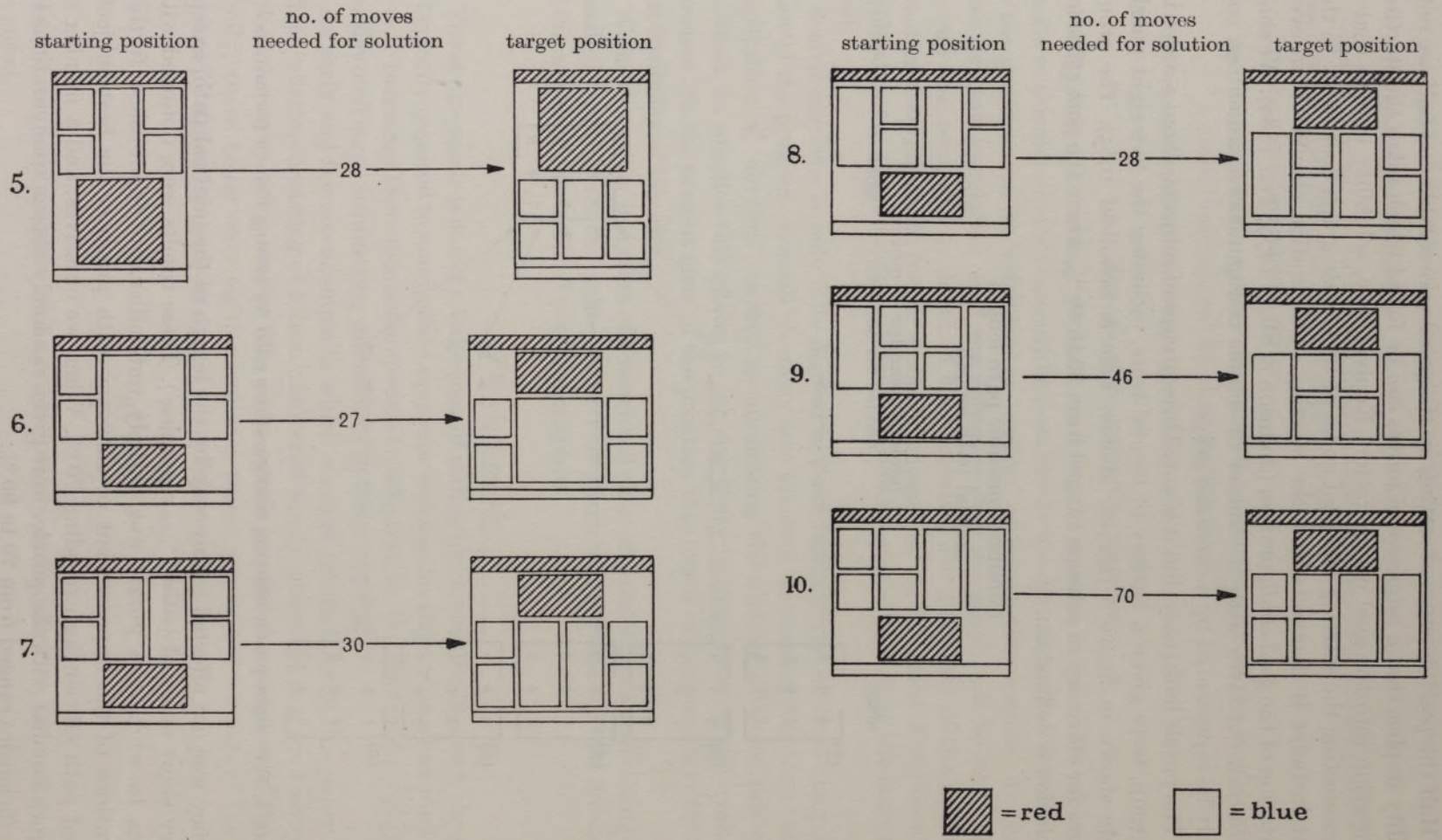

FIGURE 6. Passalong sliding block puzzle configurations 5 to 10. The Graph Traverse's performance on configurations 7 to 10 is indicated in table 4.

The same subjects were tested with the Passalong, a sliding block puzzle used by psychologists for mental testing (Alexander 1946) (see figure 6). The results obtained contrasted with those from the eight-puzzle in a way which indicated that the nature of the intellectual task may itself be very different. We mention this finding to indicate that the range of problems comprised by sliding block puzzles may offer a more diversified field of investigation than one might at first suppose.

Solution of the eight-puzzle by computer program

A computer program can be written to analyse the eight-puzzle exhaustively by a 'brute force' technique, involving enumeration of the 20 160 centre-empty 'normal' positions, starting from the goal and working outwards. Such a program has been written and successfully run by P. D. A. Schofield (see Hayes *et al.* 1965). Elegance is maximized by this method, in the sense that the shortest path is always obtained; economy, on the other hand, is at a minimum since the space searched is effectively the whole graph. The fact that the eight-puzzle can be, and has been, completely analysed in this way makes it a particularly suitable starting-point for an examination of heuristic methods, where the aim is to effect the greatest possible economy at the sacrifice of as little elegance as possible. Human problem-solving behaviour is conspicuous precisely in the capacity to develop effective approaches without attempting enumeration.

Consider how the Graph Traverser deals with the eight-puzzle when equipped with a definition of the puzzle and with a simple evaluation function. Without being concerned for the moment about how such a function might be constructed, let us examine in figure 7 the record of a specimen run. The values produced by the function used are plotted in figure 8 for successive nodes along the path found.

It is worth recapitulating in this specific context two features of the program:

(1) The next node to be developed is always the lowest-valued undeveloped node, regardless of its distance from the previous node to be developed. Search is thus *not* constrained into connected steps, but pushes forward whichever sector of the front is currently evaluated as the most promising. 'Disconnected developments' therefore occur when the 'main line' of search fails. The path is filled in retrospectively by a backward trace from the goal, once this is found.

(2) When two or more undeveloped nodes tie for the lowest value, the node which was earliest added to the tree is chosen for development. Should this rule be insufficient to break the tie, an arbitrary selection is made. We have subsequently realized that random selection would be preferable, since variation of the arbitrary rule employed turned out to have non-trivial consequences.

A crude evaluation function

In the above example of the Graph Traverser as applied to the eight-puzzle nothing was said as to how the evaluation function used was obtained, except that it was *given* to the program.

Two features of an eight-puzzle configuration suggest themselves as particularly relevant for evaluation purposes—the 'position' of the pieces and their 'sequence' and these were used to construct the function which controlled the search in

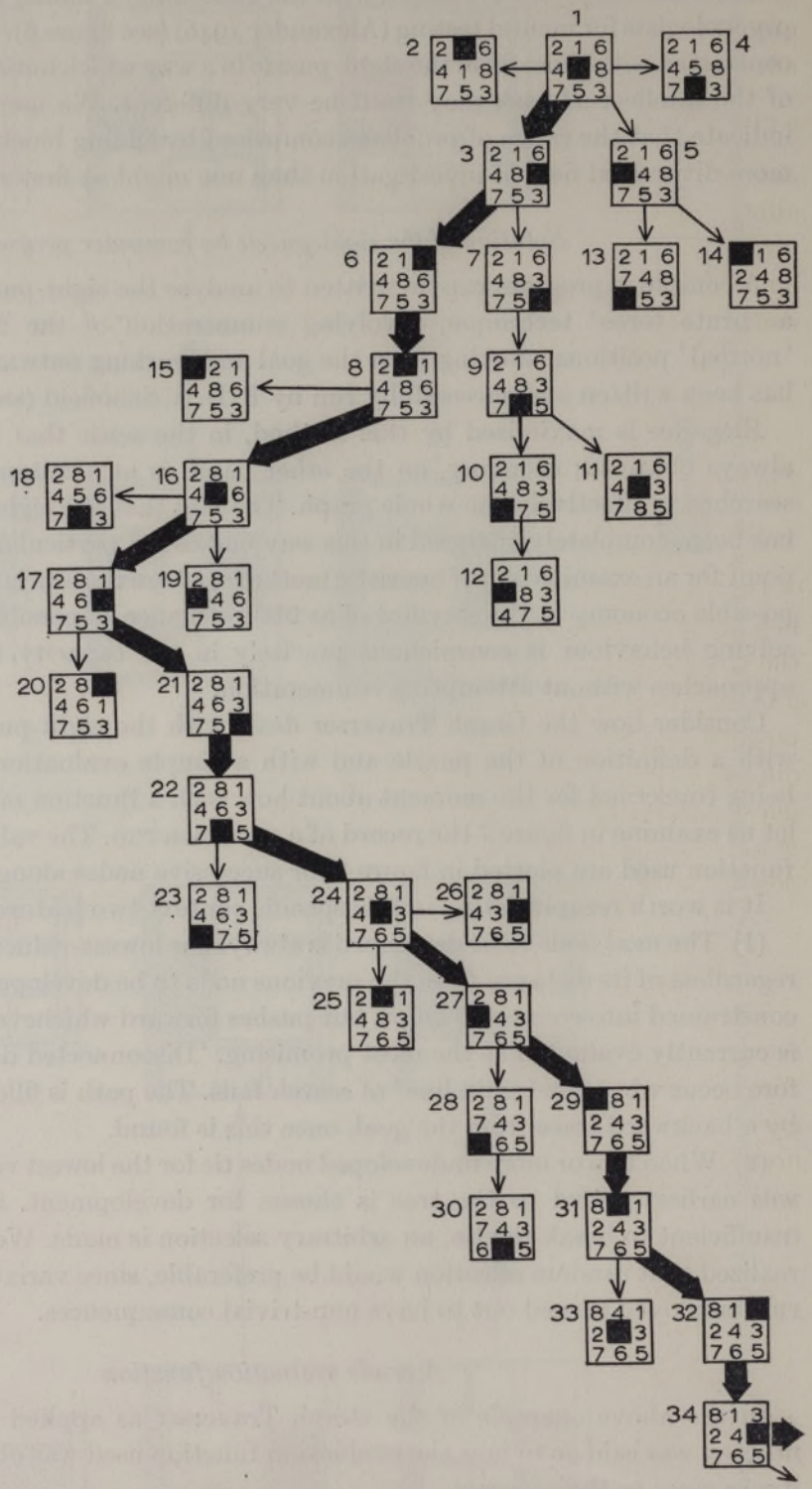

FIGURE 7. A Graph Traverser search of the eight-puzzle graph. The values of the configurations are ringed. The unringed figures give the order in which the configurations were added to the search tree.

figure 7. A 'position score' p_i can be assigned to the ith piece, according to the number of moves it is distant from 'home', disregarding for the purpose the barrier offered by intervening pieces (i.e. 'city block' distance). A 'position count', P, of a given configuration of the puzzle is then obtained as $\sum_{i=1}^{8} p_i$. A 'sequence score', s, can be assigned to each piece by checking round the non-central squares in turn, allotting 2 for every piece *not* followed by its proper successor and 0 for every other piece, except that a piece in the centre scores one. An empty non-central square is ignored for applying the succession criterion so that in, for example, the following configuration:

2 0 3
5 8 6
1 7 4

2 is held to be followed by 3. A 'sequence count', S, for the configuration is formed as the sum of the sequence scores, $\sum_{i=1}^{8} s_i$, which in the case illustrated is 13.

Having isolated these two simple features as hopefully relevant to goal-seeking behaviour, we may take some weighted combination of them, in the form $P + wS$, as the value of the configuration. Notice that a small change in w will not always cause a change in the strategic choices imposed by the evaluation function, as both the position and sequence count can take only a finite number of values. However, the number of settings of w which are in general distinct is sufficiently large that one may safely think in terms of continuous variation of performance from $w = 0$ to $w = 24$. For $w > 24$ a unit change in sequence count outweighs even the largest possible difference of position count.

Measures of performance

There are two interesting measures of the program's performance over a particular search of a graph: (1) the length of the path produced (P) (i.e. the number of arcs comprising the final path), and (2) the total number of nodes developed (D). Since every path node but the last must have been developed, but not every developed node is necessarily included in the path, it follows that $P \leqslant D$. Denote the minimal path length for a given start and goal by P^*. Then P^*/P = path efficiency (as defined earlier). There is a corresponding idea applicable to D, the number of nodes

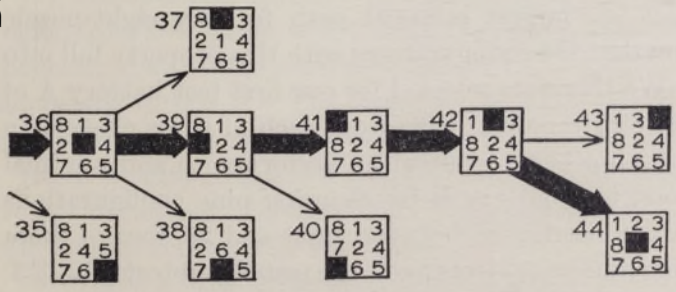

FIGURE 7. (*cont.*)

developed, yielding a measure of the amount by which this figure exceeds the minimum necessary. It is easy to see that the minimum necessary is equal to P^*, so that a measure of efficiency in respect of development is P^*/D, which we shall call the 'development efficiency'.

Finally we note that P^*/D can be written $P^*/P \times P/D$. This is a useful decomposition in drawing our attention to the further quantity P/D which is the fraction of the total number of nodes developed which are incorporated into the actual path found. We shall later see that this quantity, which we shall refer to as the 'penetrance', is of great importance where, in contrast to the eight-puzzle, P^* is unknown.

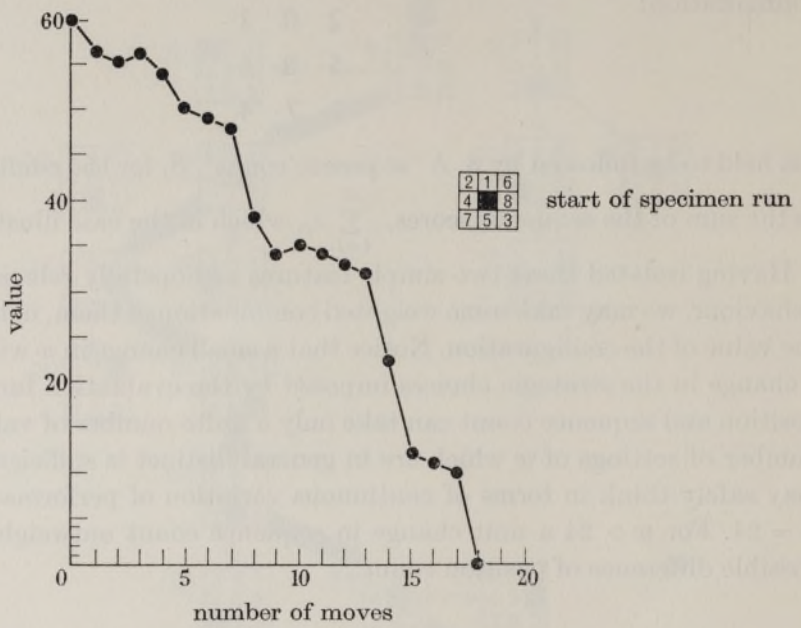

FIGURE 8. The values along the final path of the search of the eight-puzzle graph shown in figure 7.

It can be thought of as representing the degree to which the search tree is 'elongated' rather than 'bushy'. Its reciprocal, D/P, could reasonably be termed the 'blind alley ratio'. To recapitulate then

$$\text{path efficiency} \times \text{penetrance} = \text{development efficiency}.$$

RESULTS

Schofield's results show that the largest minimal path for any eight-puzzle starting configuration is 30, and that the configurations with this property fall into 12 distinct symmetry classes. We therefore selected for our first test battery A of starting configurations an arbitrary representative from each of these classes. In order to investigate the relationship between program performance and minimal path length, we formed a second test battery B by choosing nine configurations whose minimal path lengths were distributed between eight and 28, four of these being configurations upon which human subjects had been tested (subtests E2, E3, E4 and E6 in Hayes *et al.* (1965)). Each of these 21 initial configurations was

attacked by the Graph Traverser by means of the evaluation function described above with w set successively at 0, $\frac{1}{9}$, $\frac{2}{3}$, $\frac{3}{2}$, 9. Notice that when $w = 0$, sequence is ignored, and evaluation is based upon position only.

Table 1 presents the performance indices, for a selected value of w, for different minimal path lengths, while table 2 presents mean values of the various performance indices for different values of the weighting coefficient w.

TABLE 1. RESULTS OBTAINED BY APPLYING THE GRAPH TRAVERSER TO EIGHT-PUZZLE CONFIGURATIONS WITH VARYING MINIMAL PATH LENGTHS.

A simple evaluation function with one variable parameter ($w = 9$) was used.

	minimal path lengths									
	8	12	14	18	20	22	24	26	28	30
test battery ...					B					A
path efficiency (%)	100	100	54	100	77	100	48	57	93	72
development efficiency (%)	100	92	23	78	74	85	16	45	88	47
penetrance (%)	100	92	43	78	96	85	34	79	94	63

TABLE 2. RESULTS OBTAINED BY APPLYING THE GRAPH TRAVERSER TO TWO TEST BATTERIES OF EIGHT-PUZZLE CONFIGURATIONS

A simple evaluation function with one variable parameter (w) was used. B* denotes test battery B after exclusion of two configurations (MP = 8 and MP = 12, text).

	$w = 0$	$w = \frac{1}{9}$	$w = \frac{2}{3}$	$w = \frac{3}{2}$	$w = 9$
	Test battery A				
median path length	60	47	46	40	43
median number of nodes developed	$277\frac{1}{2}$	209	$164\frac{1}{2}$	61	$66\frac{1}{2}$
mean path efficiency	52·7	56·9	67·6	76·3	72·0
mean development efficiency	10·9	18·2	25·4	47·1	46·7
mean penetrance	20·8	29·7	36·2	60·0	63·3
	Test battery B*				
mean path efficiency	41·2	35·3	61·9	75·3	75·5
mean development efficiency	5·3	6·2	21·9	56·7	58·4
mean penetrance	12·0	14·7	30·8	70·3	72·7

The following points may be noted:

(1) For both test batteries the best performance on the various criteria lies in the region from $w = \frac{3}{2}$ to $w = 9$. Results, not shown here, obtained by further increasing the value of w (i.e. the relative weight allotted to 'sequence') showed no change from the results with $w = 9$. The picture is of comparatively poor performance at $w = 0$, but improving, as w increases, until a plateau is reached.

(2) If we now compare the figures for the various efficiencies we see that they are closely correlated. There is thus no evidence that by changing w elegance can be sacrified for economy or vice versa.

(3) There is little evidence from this sample of any continuing trend relating the various measures of efficiency to the length of the minimal path (i.e. the 'distance'

of the starting configuration from the goal). The two shortest minimal path lengths (table 1) are perhaps exceptions and these two 'easy' puzzles have been excluded in compiling tables 2 and 3.

It is a consequence of point (2) that the optimal values of w could have been located by inspecting the penetrance alone. Now the difference between a 'toy' problem, as is the eight-puzzle, and a 'real' problem, is that in the latter we typically have no idea of the length of the minimal path, and therefore can calculate neither path efficiences nor development efficiencies. The penetrance, however, can *always* be calculated, and is therefore potentially of the greatest use as a general measure of the efficiency of an evaluation function in solving a 'real' problem. Even more important, the penetrance may be calculated, and therefore progress estimated, *during* the solution of a problem, thus opening the door to methods whereby the program might improve its own evaluation function during the course of a long search.

With these ideas in mind, we next tried the program on the fifteen-puzzle.

Experiments with the fifteen-puzzle

For the fifteen-puzzle a limit of 500 was set on the tree size, and a single starting configuration was randomly chosen for preliminary tests. This configuration was the following:

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

Tests were conducted under a 'stop rule', according to which the search was abandoned as soon as a partial search, as previously defined, failed to decrease the mean value of the nodes encountered in it by more than 5 %, as compared with the previous partial search. As a start, evaluation was based on piece-positions only.

Preliminary results revealed two undesirable features. The first of these consisted in the *stranding* of a piece at a considerable distance from its home, cut off by an intervening barrier of more-or-less correctly positioned pieces. An example is the following, encountered after 90 moves:

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

Here all pieces are two moves or fewer from their respective homes, i.e. $p \leqslant 2$, with the single exception of piece number 3, for which $p = 5$. Expressions of the form $\sum_{i=1}^{15} h_i^a p_i^b$, where $0 < a < 1 < b$, were found to be effective. h_i was defined as the distance separating the empty square from the ith piece, expressing distance as before in unit steps.

The cure of 'stranding' threw a second feature into prominence, namely the presence of intra-row and/or intra-column reversals. An illustration is provided by the following configuration, encountered after 190 moves:

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

Although superficially this appears close to solution, since every piece is either home or next door to home, such configurations are in fact rather far from the goal. Solution requires a quite radical disruption of the degree of order which has been built up, and a good evaluation function should reflect this fact.

The matter was dealt with in an *ad hoc* fashion, by addition of a term, R, counting the number of such reversals present. The function finally adopted thus took the form $\sum_{i=1}^{15} h_i^a p_i^b + cR$, with a, b and c representing adjustable parameters determining the relative weightings given to the three features expressed by the h, p and R terms.

A test battery of ten starting configurations was now set up, by adding to the configuration shown earlier a further nine, drawn from a table of random permutations. By running the program on these ten with different settings of the parameters a, b and c, a systematic exploration was now made of the response of the system to variation in these weightings. Three levels were taken for each parameter, thus:

$$a = 0, \tfrac{1}{2}, 1; \quad b = 1, 2, 3; \quad c = 100, 300, 500;$$

so that there were in all 27 'treatment combinations'. The best performance was with $a = \frac{1}{2}$, $b = 2$, $c = 100$. At these settings of the parameters, six out of the ten puzzles were solved within the limitation of a single search tree. To grow a complete tree took about 4 min on an Elliott 503 computer.

Again, optimization could have been successfully performed using penetrance alone, since the mean value found at these settings, 60 %, was the highest encountered over all the 27 combinations. (Compare the eight-puzzle results of tables 1 and 2.) Some combinations resulted in uniform failure to solve—for example all those with $a = b = 1$.

It seemed of interest to try the most successful version of the evaluation function ($a = \frac{1}{2}, b = 2, c = 100$) on the eight-puzzle, to compare performance with that obtained from the function specially designed for the smaller problem. The results compared surprisingly well, as evidenced by the summary given in table 3.

Predictive power of penetrance

It is a natural extension of earlier definitions to calculate, as a measure of progress, the penetrance of a 'partial search'. The latter has been defined in terms of the limit set to the size of the search tree. A partial search consists in the growing of the tree up to the present limit, remembering that each time this limit is reached in the course of a long search, the corresponding 'partial path' is printed out and the tree is

erased. Resetting of weighting coefficients could also occur at the point, with the use as 'figure of merit' of

$$\text{penetrance} = \frac{\text{number of nodes in the partial path}}{\text{number of nodes developed in the partial search}}.$$

In this way program-improvement of the evaluation function itself can be envisaged. A relevant test of the possible usefulness of penetrance in this respect is to see to what extent it can predict the performance of a given evaluation function over other areas of the graph than those from which it was calculated. In particular we would like to know whether the penetrance of one partial search can be used as a guide to what is likely to happen in subsequent ones.

TABLE 3. RESULTS OBTAINED BY APPLYING THE BEST FIFTEEN-PUZZLE EVALUATION FUNCTION TO THE EIGHT-PUZZLE, COMPARED WITH RESULTS OBTAINED USING THE STANDARD EIGHT-PUZZLE EVALUATION FUNCTION.

Function 1 (eight-puzzle): $\Sigma p_i + 9\Sigma s_i$.
Function 2 (fifteen-puzzle): $\Sigma h_i^{\frac{1}{2}} p_i^2 + 100R$.

Test battery A

configuration ...	1	2	3	4	5	6	7	8	9	10	11	12
minimum pathlength	30	30	30	30	30	30	30	30	30	30	30	30
development efficiency (F 1)	81	34	91	48	28	30	23	26	47	56	54	43
development efficiency (F 2)	30	53	67	48	48	65	31	39	86	35	100	75

	mean development efficiency	mean path efficiency	mean penetrance
function 1	47	72	63
function 2	56	76	73

Test battery B

configuration ...	1	2	3	4	5	6	7	8	9
minimum pathlength	8	12	14	18	20	22	24	26	28
development efficiency (F 1)	100*	92*	23	78	74	85	16	45	87
development efficiency (F 2)	73*	100*	58	51	28	32	59	49	43

	mean development efficiency	mean path efficiency	mean penetrance
function 1	58	75	73
function 2	46	71	65

* Not included in means, see text.

To investigate this point we re-ran the program on the same test battery over a restricted range of 11 different parameter settings, with the further difference that the search tree was limited to a size of 250, and two successive partial searches were permitted. The degree of success was only slightly lower than before. Excluding those cases in which the goal was found in the first partial search, we have plotted, in figure 9, a measure of subsequent performance against the penetrance *of the first partial search only*. The evident correlation supports the proposed use of penetrance as a promise measure.

A trial of the program on the Passalong sliding block puzzle

As a first trial of the adaptability of the program, a 'develop' procedure was written for the Passalong (see figure 6). A simple evaluation function was devised. First, one 'black mark' was allotted for each of the 12 cells of the 3×4 array not covered by a piece of the correct type. Secondly, additional terms were added to

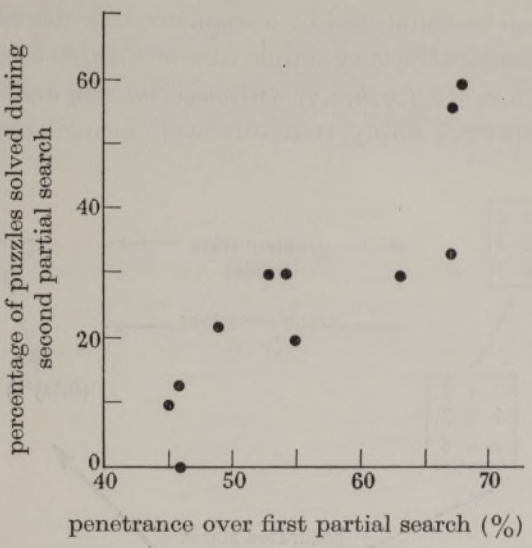

FIGURE 9. Scatter diagram showing a success measure over one partial search plotted against the pentrance of the previous partial search.

TABLE 4. RESULTS OBTAINED BY APPLYING THE GRAPH TRAVERSER TO THE PASSALONG SLIDING BLOCK PUZZLE, BY MEANS OF A SIMPLE EVALUATION FUNCTION

For the configurations, refer to figure 6.

configuration	minimum pathlength possible	actual pathlength	no. of developments	path efficiency (%)	development efficiency (%)	penetrance (%)
7	30	30	32	100	94	94
8	28	30	156	93	18	19
9	46	58	258	79	18	23
10	70	72	359	97	19	20

express various features of a configuration, such as degree of right-left symmetry, mutual proximity of the vertical rectangles, freedom of movement of the horizontal rectangle, etc.

The results of this limited trial, set out in table 4, show good performance on all criteria for configuration 7. Thereafter we find path efficiency high but development efficiency low—compare the abrupt increase in difficulty for human solvers at this point (Hayes *et al.* 1965)—indicating that the evaluation function is no longer fully adequate for these configurations. With a *totally* inadequate evaluation function i.e. a constant function, the program is guaranteed to find the minimum path, at the cost of an abysmally low development efficiency. In such a case it is in effect

employing the well known, and laborious, algorithm of locating first all nodes at distance 1 from the origin, then all at distance 2, etc. (see Berge 1962, pp. 67–68). At this extreme point the heuristic element has disappeared.

Application to algebraic manipulation

The next application—algebraic manipulation—was chosen to be further afield.

The problem as stated here is to demonstrate equivalence between two expressions, i.e. to show that they can be connected by a sequence of legitimate operations. We chose for the first experiments the very simple case of a single binary operation, '$*$', on a set of elements $\{a, b, c, d, e, f, g, h, i, j\}$ with *associativity* and *commutativity*. We can see that these properties imply that any well formed expression, however

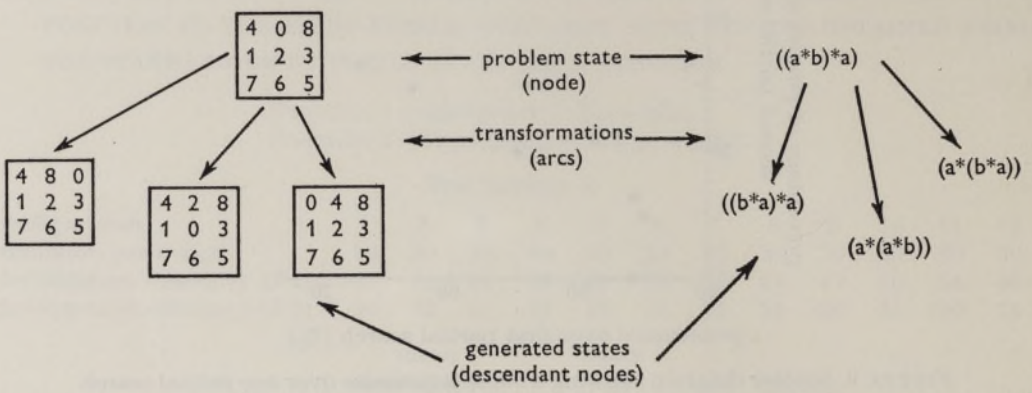

FIGURE 10. Diagrammatic representation of the relationship between the general graph traversal problem, the eight-puzzle, and a simple algebraic manipulation problem.

bracketed, is equivalent to any other well formed expression containing, in any order, the same set of elements, but this is not known to the program, which seeks to demonstrate equivalence from first principles. The analogy here with the eight-puzzle or fifteen-puzzle is that although *we* can see that the solubility of a given configuration is determined by whether it is an even permutation of the goal configuration, the program seeks to demonstrate solubility by constructing a path.

The way in which a correspondence was set up between the algebraic problem and the graph traversal schema is shown in figure 10, with the corresponding identifications for the eight-puzzle shown for comparison. Just as the 'development' of an eight-puzzle configuration generates all the configurations of the puzzle which can be reached by a single move from the state in question, so an algebraic expression is 'developed' by generating all expressions which can be derived from it by a single application of either the commutativity or the associativity rule. We adopted standard goal expressions of the type

$$(((a*b)*c)*d),$$

or

$$((((a*a)*a)*h)*k),$$

or

$$((d*e)*e),$$

where alphabetical order is required and where the 'open brackets' are concentrated to the left of the expression.

As an evaluation function, the sum $\sum_s w_s p_s$ was calculated, where the summation is over the symbols of the expression *excluding*) and $*$, and where w_s is a weight assigned to s—for example the weights $10, 9, 8, 7, \ldots, 1, 0$ might be assigned to $(, a, b, c, \ldots, i, j$ respectively—and where p_s is the ordinal number of the symbol in the string, again excluding) and $*$. In practice, the weights assigned to the symbols $(, a, b, c, \ldots, i, j$ were

$$Q, 9, 8, 7, \ldots, 0$$

where Q is an adjustable parameter.

The results obtained for one simple task are summarized in tables 5 and 6. They are mainly of interest as illustrating once again the increase in search economy attendant on the use of even the simplest evaluation function (cf. tables 1 and 2), as well as indicating that the field of symbol manipulation is open to the Graph Traverser approach.

TABLE 5. RESULTS OBTAINED BY APPLYING THE GRAPH TRAVERSER TO A SIMPLE ALGEBRAIC MANIPULATION PROBLEM

A simple evaluation function with one variable parameter was employed. A dash indicates failure to solve the problem within a search tree of 500 nodes.

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
pathcount	—	—	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

TABLE 6. SOLUTION PATHS TO AN ALGEBRAIC MANIPULATION PROBLEM

The task is to convert the expression $((h*(a*f))*((c*d)*c))$ into $(((((a*c)*c)*d)*f)*h)$ using only the associativity and commutativity of the operation $*$. The solution on the left was found by the Graph Traverser, and that on the right by hand.

	shortest path found by program (parameter setting = 9)	shortest path known
0	$((h*(a*f))*((c*d)*c))$	$((h*(a*f))*((c*d)*c))$
1	$(((c*d)*c)*(h*(a*f)))$	$(((c*d)*c)*(h*(a*f)))$
2	$(((c*d)*c)*((a*f)*h))$	$(((c*d)*c)*((a*f)*h))$
3	$((((c*d)*c)*(a*f))*h)$	$((c*(c*d))*((a*f)*h))$
4	$(((((c*d)*c)*a)*f)*h)$	$(((c*(c*d))*(a*f))*h)$
5	$(((a*((c*d)*c))*f)*h)$	$((((c*(c*d))*a)*f)*h)$
6	$(((a*(c*(c*d)))*f)*h)$	$(((a*(c*(c*d)))*f)*h)$
7	$(((a*((c*c)*d))*f)*h)$	$((((a*c)*(c*d))*f)*h)$
8	$((((a*(c*c))*d)*f)*h)$	$(((((a*c)*c)*d)*f)*h)$
9	$(((((a*c)*c)*d)*f)*h)$	

DISCUSSION

The fact that over a range of problems the program was able to find solutions which could be called 'good' by human standards is not in itself of great significance since in these experiments a 'short term memory' of some hundreds of problem

states was permitted—an order of magnitude greater than the span available to the human solver. What is significant is

(1) That this level of performance can be reached by a search method utilizing state evaluation only, of a fairly simple sort, before any principles of operator-selection have been built into the system.

(2) That the program proved fairly easy to adapt to problems as mutually dissimilar as sliding block puzzles and algebraic manipulation.

(3) That the program showed itself an effective instrument not only for the implementation of evaluation functions in 'production' runs, but also for their initial development by trial and error.

(4) That the 'penetrance' gave good indications of potential usefulness for endowing the program with the facility of improving its given evaluation function. For this purpose, there must be some measure available to the program by which it can judge how well it is doing when in the midst of an attempt to solve some problem. The penetrance, which is a function of the structure of the search tree, promises to serve this purpose. More generally, if the program is to 'learn', it can only do so by using the information it has stored about the problem, i.e. by using the information embedded in its search tree.

Some analogies are offered here by the work of Samuel (1960) mentioned earlier in particular by his techniques of adjusting his scoring polynomial (evaluation function) so as to tend to give equal values to configurations lying on the same minimax chain within the stored search tree. The analogy would be strengthened if an attempt were made by the program to adjust the evaluation function so as to reflect the *metric* properties (as compared with *structural* properties, such as measured by the penetrance) of the stored search tree—i.e. assign suitably different values to nodes lying far apart on a branch and vice versa.

This idea is developed further in an article by Doran (1966) where there is also a description of a new version of the program which uses a *dynamic* tree. By this is meant that the program, on reaching the growth limit, behaves less catastrophically. It no longer selects the most promising terminal node, and then commits itself to a path to that node before deleting the entire tree to create new working space. Instead, the program commits itself only to a *single* 'move', and only that part of the tree thereby rendered valueless is erased.

In the detailed description of the graph traversal schema, a specific problem type was described, and the applications we have discussed have all fallen within this type. In this context, we wish to make two points. First, our applications have all had in common a particular limiting condition, namely that their problem graphs have been symmetric. This means that the search strategy has been inefficient to the degree that a search tree could usefully have been grown from the goal node simultaneously with growth from the start. To see this, consider the analogy with the case of a search conducted, not over a graph, but over an n-dimensional Euclidean space. Specifically we imagine in case 1 that a 'search hypersphere' is grown from the origin until it touches the goal, while in case 2 hyperspheres are grown from both start and goal until they touch each other. Here it can easily be shown that the factor of economy (supposing cost to be represented by the total volume explored)

is 2^{n-1}. As pointed out to us by D. G. Kendall, it is possible that this way of looking at the matter might lead to a useful definition of the effective dimensionality of a graph. The second point is that although a problem type has been strictly specified, in practice fairly minor adjustments to the program will permit its application to a wider range of problem types, for example to a situation where some defining property of a goal configuration is available, but where no particular goal is specified.

A classification of problem types may be obtained by distinguishing three pairs of alternatives: (1) whether it is a *path* or a *node* that must be found; (2) whether the goal node is *fully* specified, or specified up to some property it must have; and (3) whether the graph is *symmetric* or *non-symmetric*.

We now identify the problem types:

Type A: *path* *full* *symmetric*

Sliding block puzzles are of this 'demonstration of equivalence' type. It seems likely that search trees should be grown from 'both ends'. Practical problems that fall naturally into this category seem rather rare, although some problems in algebraic simplification and theorem-proving are of this type.

Type B:	*path*	*full*	*non-symmetric*
	path	*property*	*symmetric*
	path	*property*	*non-symmetric*

The Graph Traverser strategy is most appropriate to these situations, although it does not use the symmetry in the second. Sliding block puzzles with partially specified goal configuration are of this type, a solution consisting in a path from the starting configuration to a terminal configuration satisfying the goal condition. Again practical applications seem rare.

Type C:	*node*	*property*	*symmetric*
	node	*property*	*non-symmetric*

Typically the graph structure is *imposed* as part of the strategy, rather than *given* by the terms of the problem. Operations, or 'moves', provided by the strategy for transforming problem states define arcs in the abstract representation, and thus convert the initially given problem into a connected graph upon which the program can work. In particular, the symmetry or otherwise of the graph is likely to be a matter of definition. A solution must merely satisfy the goal condition. The path is *not* of primary interest. The Graph Traverser strategy is applicable to this type of problem but it is not yet clear how efficient such an application would be. Many practical problems are of this type, for example allocation and timetabling problems.

A more general type of problem is that where a solution is a *set* of nodes. This includes the above problem types, as special cases.

In our work to date with the Graph Traverser we have avoided the use of operator selection, but such techniques can be inserted into the program schema. In its present form the procedure 'develop' produces all immediate descendants of a given node. However, there is no difficulty in constraining it to produce only a

subset of the immediate descendants, corresponding to selection and application of only a subset of the available operators. This topic is discussed in an article by Michie (1966).

Generality versus speed

This paper has been about a general problem-solving program. In the extreme case such a program embodies no more special knowledge about the problem in hand than is required to set in motion a search for solutions. It typically purchases a wide range of application in exchange for speed. For example, the Graph Traverser at best takes about 20 s to solve a difficult eight-puzzle configuration on an Elliott 503, and takes about 15 s to demonstrate that the two expressions given in table 6 are equivalent. Special-purpose programs for performing the same two tasks (Michie 1966; E. W. Elcock, personal communication) are an order of magnitude faster. At first sight this comparison seems so damaging as to prompt the question: why bother at all with general purpose programs?

An immediate reply is that such programs could have a use when 'one-off' problem-solving is required, where the attraction of avoiding the labour of constructing a special program may outweigh the defects of the general one. For 'production runs', however, the balance would normally be tipped in favour of constructing an efficient, fast-running program for the particular problem. The possibility should be borne in mind here, that the construction of such a program may itself be aided by exploratory work using a general purpose program such as the Graph Traverser.

Conclusions

The first stage of this design project has been successful, in the sense:

(1) That the program, in its present restricted form, does solve problems. It has in fact already been found to be a useful tool by a colleague working in a different field (automatic theorem-proving: see Popplestone 1966).

(2) That it seems to meet our criteria for using it as the platform on which to build the next, i.e. 'learning', stage.

The cost of this work was defrayed by a grant from the Science Research Council, which also provided a Junior Research Fellowship held by one of us (J. E. D.). Our thanks are also due to Dr N. T. J. Bailey, Director of the Unit of Biometry, Oxford University, and to Dr M. H. Rogers, Director of the Computer Unit, Bristol University, for generous provision of computing facilities, and to our colleague R. M. Burstall for many helpful criticisms and comments.

References

Alexander, W. P. 1946 *A performance scale for the measurement of practical ability*. Instruction book issued with 'Alexander Performance Scale'. London: Councils and Education Press.

Berge, C. 1962 *The theory of graphs*. English translation by Alison Doig. London: Methuen.

Burstall, R. M. 1966 Computer design of electricity supply networks by a heuristic method. *Computer Journal* (in press).

Doran, J. E. 1966 An approach to automatic problem-solving. In *Machine intelligence*, **1** (ed. N. L. Collins & D. Michie). Edinburgh: Oliver and Boyd. (In press.)

Gardner, M. 1964, 1965*a*, *b*, *c* Mathematical games. *Scient. Am.* **210**, 122–30; **212**, 112–17; **212**, 120–4; **213**, 222–36.

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 Report*, no. 7. Experimental Programming Unit, University of Edinburgh.

Johnson, W. W. & Story, W. E. 1879 Notes on the '15' puzzle. *Am. J. Math.* **2**, 397–404.

Michie, D. 1966 Game playing and game learning automata. Ch. 8 of *Advances in programming and non-numerical computation* (ed. L. Fox), pp. 183–95. London: Pergamon.

Michie, D. 1966 Strategy-building with the Graph Traverser. In *Machine intelligence*, **1** (ed. N. L. Collins & D. Michie). Edinburgh: Oliver and Boyd. (In press.)

Newell, A. & Ernst, G. 1965 The search for generality. In *Information processing* 1965: *Proceedings of IFIP Congress* 1965, vol. 1 (ed. Wayne A. Kalenich), pp. 17–24. Baltimore: Spartan.

Newell, A., Shaw, J. C. & Simon, H. A. 1960 A variety of intelligent learning in a general problem solver. In *Self-organising Systems* (eds. Marshall C. Yovits and Scott Cameron) pp. 153–89. London: Pergamon.

Popplestone, R. J. 1966 Theorem proving by Beth tree methods. In *Machine intelligence*, **1** (ed. N. L. Collins & D. Michie). Edinburgh: Oliver and Boyd. (In press.)

Samuel, A. L. 1960 Programming computers to play games. In *Advances in computers* vol. 1 (ed. Franz L. Alt.), pp. 165–92. London: Academic Press.

Tait, P. G. 1880 Note on the theory of the '15 puzzle'. *Proc. Royal Soc. Edinb.* **10**, 664–5.

Turing, A. M. 1953 Digital computers applied to games, ch. 25 of *Faster than thought* (ed. B. V. Bowden), pp. 286–310. London: Pitman.