16

Experiments with the Adaptive Graph Traverser

Donald Michie and Robert Ross
Department of Machine Intelligence and Perception
University of Edinburgh

Abstract

A formal description is given of GT4, a revised and extended version of the Graph Traverser. Methods are described whereby GT4 can improve its performance at run time (a) by automatic optimization of parameters used by the evaluation function and (b) by dynamic re-ordering of operators. Neither method depends upon there being any successful searches in the program's past experience of a given problem. The essential feasibility of both approaches has been validated in experimental tests using sliding block puzzles. Two planned extensions, 'local smoothing' and 'regionalization' are described.

INTRODUCTION

The Graph Traverser (Doran and Michie 1966), and subsequent work based on it, represents an attempt to adapt game-playing methods, particularly those of Samuel (1959), to automatic problem-solving. The design objective is not the simulation of human problem-solving as a study in psychology, but rather to provide an efficient general-purpose search procedure appropriate to non-numerical problem domains. There is a parallel with the development of direct search techniques for numerical function minimization, for example pattern search (Hooke and Jeeves 1961), simplex (Spendley, Hext and Himsworth 1962, Nelder and Mead 1965). These now form a staple constituent of well-stocked program libraries.

The main topic of the present paper is the proposal that general-purpose search procedures should be able to use accumulating experience of a given problem to improve search efficiency at run time. One of the mechanisms to be described involves optimization of numerical parameters used to control the search, reminiscent of the 'learning by generalization' used in Samuel's (1959) checkers-playing program. In our experiments it was convenient to call the Graph Traverser recursively for this purpose; in the

lower-level calls the program implemented the pattern-search method of Hooke and Jeeves. Thus classical numerical methods can be viewed as special cases of more general search procedures which make no assumptions of continuity or dimensionality.

Some of the formulations of heuristic search which we shall employ were first coined in the GPS work (Newell, Shaw and Simon 1957, Ernst and Newell 1969). The Graph Traverser diverges from GPS in motivation and method; it may be useful to list, as we have done in table 1, some of the contrasts.

	GPS	Graph Traverser
A. Method of selecting next state	Current state*	Most promising state on stored lookahead tree, selected by evaluation function
B. Method of selecting next operator	Operator-difference table	Re-trace from most promising state
C. Action if operator inapplicable	Recursive call to solve the sub-problem: 'find state to which operator *is* applicable'	Try next operator
D. Action when memory is full	Search abandoned	Dynamic pruning of lookahead tree
E. Adaptive features	—	(a) Automatic optimization of evaluation function (cf. Samuel 1959, 1967) (b) Automatic re-ordering of operators (cf. Quinlan 1969)

* but lookahead and back-tracking to remembered states can occur when GPS calls itself recursively.

Table 1. A comparison of selected features of GPS and the Graph Traverser. The present paper is concerned with the last item (adaptive features).

A programme of work for an adaptive Graph Traverser was outlined at the first Machine Intelligence Workshop (Michie 1967) in terms which can be related to the scheme of table 1 as follows:

(1) The program must be able to improve its performance before, not after, its first successful search in the problem domain. It must become capable of solving problems for which its initial unimproved form would be inadequate. There is a difference here from the work of Quinlan (1969) which,

although similar in some other respects, depends upon solution of problems for the learning feature to work.

(2) The learning mechanism's ultimate goal is to render the Graph Traverser mode of operation redundant. The chess master, Reti, was once asked 'How many moves do you look ahead?' and is said to have replied 'One: the right one!'. The Graph Traverser represents a particular scheme for managing the lookahead process. Ideally, self-improvement under categories E(a) and E(b) should allow the program continually to reduce the lookahead tree and to narrow the choice among operators at each stage. In the limit the program would be operating on the current state only, having synthesized for itself a complete strategy in the form of a table of states and operators. We are not suggesting attainment of this limit as a practical goal of unaided heuristic search. Methods will be needed, which go outside the present Graph Traverser framework, for automatically abstracting strategically meaningful features so as to set up an 'image space' (e.g., Sandewall 1969) in which higher-level search can proceed, indexed, for example, by a table of state-categories and operator-categories. In the meantime we have the limited aim for an adaptive heuristic search method that it should make effective use of lookahead trees to generate and implement a strategy (a function mapping from states onto operators), and partially condense it into tabular form.

INFORMAL DESCRIPTION OF THE SEARCH METHOD

The graph of a particular problem is specified to the program implicitly by a list of operators. Each operator will be applicable to some, but not normally all, nodes. When an operator is applied successfully a neighbouring node of the problem graph will be produced. The search for a solution is directed by an *evaluation function* that assigns to each node a numerical value, which is an estimate of its distance from the nearest goal node. At the beginning of a search only the initial node is explicitly known to the program and is capable of being evaluated and *developed.* Development proceeds by applying in succession appropriate operators until a neighbouring node is produced that is estimated to be closer to a goal node. This node is then developed in a similar fashion. At each stage a check is made to avoid adding to the *partial search tree* any node that is already on it. If the most promising node fails to produce a descendant of lower value after all operators have been applied it is labelled as 'fully developed'. The program will always develop the lowest valued node on the partial search tree ignoring any that are fully developed. The search proceeds iteratively in this manner until either a goal is located or the partial search tree reaches a specified size. In the latter case a *partial path* is traced from the lowest-valued developable node to the root of the tree. Then, starting at the root, a specified number (typically one) of nodes on this path are printed out, and that part of the tree is erased, preserving only the part which is dependent from the last node to be printed. If it so happens that the most promising node is also the root then the program will

trace a partial path from the next most promising node and print out the appropriate number of nodes on this path. After *dynamic pruning*, growth of the search tree is resumed. Pruning therefore makes it possible for the search to continue indefinitely. In practice, a *resignation criterion* is specified and when it is met the search is abandoned.

FORMAL DESCRIPTION

The Graph Traverser's domain includes any problem which can be represented as a graph, $G \equiv (X, \Gamma)$, where

> X is a set of nodes (corresponding to the discrete states of the problem) and
> Γ: $X \rightarrow 2^X$ is the successor function over the set (corresponding to the 'rule-book' of the problem).

Given a node $s \in X$ (initial state) and a goal-recognizing unary predicate, P, it is required to find a sequence of nodes $x_0, x_1, \ldots, x_k$ such that $x_0 = s$, $P(x_k)$ and $x_{i+1} \in \Gamma(x_i)$ for $0 \leqslant i \leqslant k-1$. There may be some additional requirement, for example that k be a minimum.

We now introduce an operator set $\Gamma' = \{\Gamma'_1, \Gamma'_2, \ldots \Gamma'_m\}$, on which we impose an ordering, so that the notation Γ'_i denotes the ith element in Γ'. An operator is a (possibly partial) function, Γ'_i: $X \rightarrow X$, corresponding to a 'move' or 'action' to be performed upon states of the problem. The relation between Γ' and Γ is: let $X_i = \Gamma(x_i)$ and let $X'_i = \bigcup_{a=1}^{m} \Gamma'_a(x_i)$; then $x_j \in X'_i$ if $x_j \in X_i$. Thus for all i, $X_i \subseteq X'_i$. We avoid asserting $X_i = X'_i$ in order to allow for modes of search in which *compound moves* are synthesized and added to Γ', supplementing the set of *simple moves* given by the rule book.

We can now represent the path $x_0, x_1, \ldots, x_k$ in terms of a sequence of operator-applications, thus: $\Gamma'_a(x_0)$, $\Gamma'_b(\Gamma'_a(x_0))$, . . ., $\Gamma'_t(\Gamma'_s(\ldots(\Gamma'_b(\Gamma'_a(x_0)))\ldots))$. To allow for the case where $\Gamma'_r(x_i)$ is undefined for some r and i (i.e., some actions not applicable to some states) we extend X to incorporate an additional node representing the value *undefined*. This allows the set of immediate successors of a problem state to include not only all those states which can be generated from it by application of legal moves taken from the rule book, but also a generalized 'error state' which results from the application of any other move. Observe that $\Gamma(\textit{undefined}) = \{\textit{undefined}\}$ and $\Gamma'_i(\textit{undefined}) = \textit{undefined}$ for all i.

Γ is a function which, with X, can be used to generate all paths within the problem graph. The operators in Γ' on the other hand are used in conjunction with a search algorithm to generate a search tree. In the original version of the Graph Traverser, which we might here call GT1, Γ was used as the move-generator of the search algorithm, so that when a node was developed all its immediate successors were produced. 'Partial development' was implemented by Doran (1968) in ALGOL versions which we shall collectively call GT2,

and by Marsh (1969) in the POP-2 library program GT3. We call the version described in the present paper GT4.

There is a function *strategy*: $X \rightarrow \Gamma'$ which can be applied iteratively to generate a path, one node at a time after each successive call of *strategy*. To specify it so as to give the action of GT4 we consider a tree, T, which can be represented as a set of 4-tuples of the form

$t = \langle stateof(t), j, parentof(t), i \rangle$ where $stateof(t) \in X$ and *parentof*: $T \rightarrow T$ is defined so that for all t in T
$\Gamma'_i(stateof(parentof\ (t))) = stateof(t)$ except when $t = t_0$, the root node, in which case $parentof(t) = undefined$. Note that i is used to preserve the index number of the operator used to generate $stateof(t)$ from the state component of the predecessor node. In like manner, j is 1 plus the index number of the most recent operator to have been applied to $stateof(t)$.

Growth of the tree proceeds by successive enlargements $T_0, T_1, T_2, \ldots$ where $T_0 = \{undefined, t_0\}, T_1 = \{undefined, t_0, t_1\}, T_2 = \{undefined, t_0, t_1, t_2\}$ etc. Observe that t_0 is constructed from x_i, $0 \leqslant i < k$, where x_i represents some member of the path $x_0, x_1, \ldots x_k$ and is the argument of the current call of *strategy*. *undefined* is the parent of the root node. It is also the immediate successor obtained from any node in T by applying an operator corresponding to an action inapplicable to that state. Tree growth is guided by

(1) an evaluation function f: $X \rightarrow R^+$ (the set of non-negative reals) which estimates the distance of any $x \in X$ along the minimal path to the nearest x satisfying P,

(2) an operator-selection function c: $T \rightarrow \Gamma'$ which uses the j-component of the node to which it is applied to ensure that the new operator selected is not a member of the sequence of operators $\Gamma'_1, \Gamma'_2, \ldots, \Gamma'_j$ which have already been applied to the state component of this node during the current call of *strategy*.

Application of *strategy* to a state, x, causes the following chain of events. Recall that m is the size of the operator set Γ'.

1. Create sets $S = \{\langle x, 1, undefined, undefined \rangle\}$, $S' = \{\langle undefined, undefined, undefined, undefined \rangle\}$; note that $S \cup S' = T$ at every stage
2. Select $t_{min} \in S$ such that $f(stateof(t_{min}))$ is a minimum
3. If $P(stateof(t_{min}))$ or if size of $S \cup S' =$ limit then *retrace* (t_{min}) and exit
4. Apply c to t_{min} to obtain Γ'_r, $j \leqslant r \leqslant m$
   (in the study to be described $r = j$ always)
5. Assign $j+1$ to j in t_{min}
6. If there is no t' in $S \cup S'$ such that $\Gamma'_r(stateof(t_{min})) = stateof(t')$ then create a new node $\langle \Gamma'_r(stateof(t_{min})), 1, t_{min}, r \rangle$ and place it in S
7. If $j > m$ then move t_{min} from S into S'
8. Go to step 2.

The retrace function is evaluated by repeated use of *parentof*. When *undefined* is finally produced, *retrace* gives as its result the member of Γ' which corresponds to the first arc of the retraced path. In the operator-reordering mode to be described later it is at this point that promotion and demotion of operators occur. We arrange that when the argument of *retrace* is the root, i.e., $t_{min} = t_0$, t_{min} is transferred into S' and control returns to step 2.

To summarize the action of the search algorithm as a whole, if the following is a POP-2 function (globals defined as previously):

```
FUNCTION TRANSFORM STATE; VARS OPERATOR;
LOOP: PRINT(STATE);
  IF P(STATE) THEN EXIT;
  STRATEGY(STATE)—> OPERATOR;
  OPERATOR(STATE)—> STATE;
  GOTO LOOP
END;
```

then the call TRANSFORM(X0); will cause the sequence $x_0, x_1, \ldots x_k$ to be generated and printed out provided that a solution path exists and is found.

The above is a satisfactory description of an idealized version of the Graph Traverser. It would, however, be seriously inefficient if implemented in the form described, and the following modifications are needed to bring it into line with the actual program.

(1) Once a goal node is located, during step 3 above, the complete path is immediately retraced and printed out and the program halts. Provided that the condition 'if $P(a)$ and not $P(b)$ then $f(a) < f(b)$' is satisfied then the final result is unaffected.

(2) The procedure as described re-grows the complete lookahead tree with every fresh call of *strategy*. Most of this represents needless duplication of work. To eliminate this, step 1 above is omitted, and S and S' (which together comprise T) are held as globals and hence the lookahead tree is preserved between successive calls.

In order to get rid of the part of the tree which is *not* re-grown under the previous form of the algorithm it is necessary to introduce a pruning routine into the TRANSFORM function which deletes the root node of the lookahead tree and all side-branches dependent from it. Under standard conditions it can be arranged that the end result is the same under either form of the algorithm, although this is not the case if certain further efficiency-promoting changes are made.

We shall not pursue implementation details, since they are irrelevant to the main topic. It will, on the other hand, prepare the ground and help fix ideas if we state at this point that the 'learning' features which we will describe operate solely by allowing the program at run time to modify

(a) the elements of a global list of numerical parameters, thus modifying the effects of applying f,

(b) the ordering of a global list of integers used to index Γ', thus modifying the effects of applying *c*.

Implementation of (a) is through recursive call of GT4, while implementation of (b) is through a side-effect of *retrace* immediately before exit from *strategy*.

'LEARNING' EXPERIMENTS WITH GT4

Program-modification of the effects of *f* and *c* is restricted to making changes in global structures used by the respective evaluation procedures: no modification of corresponding procedure bodies is involved. The structure used to evaluate *f* is a parameter list analogous to the list of coefficients used in Samuel's program to evaluate his 'scoring polynomial'. In the case of *c* the relevant structure is a permutation on the indices by which *c* orders its selection of operators from Γ'. Two separate series of experiments were conducted, one to examine the feasibility of automatic optimization of the evaluation process, and the other to test the effect of automatic re-ordering of operators. For these experiments GT4 was implemented as a POP-2 program and run on an ICL 4130 computer.

Parameter optimization: method

The evaluation function, *f*, is defined as an estimator of the minimum distance from a node of the problem graph to the nearest state satisfying *P*. More generally we may define a distance estimator $d: X \times X \rightarrow R^+$ which estimates the minimum distance between any two nodes of *G*. A given distance estimator over *G* may be transformed into an evaluation function for a particular problem associated with *G* by requiring $f(x) = d(x, x_k)$ where x_k is chosen from the goal set so as to minimize the true distance from *x*, which we denote $\delta(x, x_k)$. Since in practice δ is unknown there is no general way of obtaining *f* from *d*, except in the special case where the goal set contains only a single node, x^* say. In this case, in the terminology of POP-2 programming, *f* may be produced from *d* by partially applying *d* to x^*, i.e., D(%XSTAR%)—>F. This condition obtained in all the experimental work to be reported, and the program did in fact construct *f* from *d* on entry in the manner described. We can usefully list at this point the data objects which the user has to supply to this version of GT4 in order to specify a problem to it and set it to run in self-optimizing mode.

1. Γ' in the form of a POP-2 list of functions of one variable,
2. x^* in the form of a POP-2 data-structure of appropriate type,
3. *d* in the form of a POP-2 function,
4. a list of parameters used by the numerical optimizer.

1 and 2 comprise the specification of the problem, whereas in principle items 3 and 4 could be supplied by default settings. By substituting different combinations of values in the global parameter list used by *d* different estimators can be produced whose utility the program must assess. The basis of this assessment, suggested by Doran (1967), is:

(1) that the utility of a given d can be measured by the closeness with which $d(x_i, x_j)$ approximates $\delta(x_i, x_j)$ over $X \times X$; actually a weaker measure is sufficient, namely that the *rank order* of the distances estimated by d should approximate the rank order of the true distances.
(2) that in the absence of direct knowledge of δ an estimate of this rank correlation can be formed from comparison of estimated with actual distances over the partial search tree, T; for example distances from terminal nodes to the root node. T corresponds to a connected subgraph, and hence in some sense to a 'local sample', of G. The success of such an approach clearly depends upon the manner in which this local sample has been formed, and hence upon how good an estimator d is in the first place. It also depends upon d possessing a certain homogeneity over G, or at least over sufficiently large connected sub-graphs, with respect to the features to which the entries in the global parameter list correspond. This is the same as saying that rank correlations of d with δ computed over different local samples of G should tend to agree.

The entries in the parameter list can thus be regarded as independent variables of a 'goodness of match' function which it is desired to maximize: specifically this function is the rank correlation coefficient computed over T as outlined above. GT4's search for an optimal combination of settings is carried out recursively after a specified number of searches, the program acting as a direct search numerical optimizer, in a fashion which will be described in detail in the next section.

The optimization technique investigated was proposed by Hooke and Jeeves (1961) and is known as 'pattern search'. While experimenting with the method we produced a modification (Michie and Ross 1969) which resulted in significant improvements in search efficiency as measured by the number of function evaluations. When, however, comparisons were made on a strict real-time basis it was found that such gains could only be maintained for functions involving unusually heavy computational work to evaluate: for the four test functions used in our experiments the unmodified algorithm was faster. It was further found that the algorithm of Powell (1964), which is conspicuously sparing in the number of function evaluations, fared no better than pattern search on the real-time criterion. On the basis of these and subsequent experiments we therefore adopted pattern search as the numerical optimizer to be used in parameter optimization. Pattern search also possessed the advantage that it could be readily represented in a form suitable for execution by the Graph Traverser itself.

DESCRIPTION OF PATTERN SEARCH

The search proceeds from an initial exploration phase through an alternation of pattern phases and further exploration phases, the latter being followed by an interposed rescue phase where appropriate.

Exploration phase

An initial base point $\mathbf{b}_1$, is defined by the starting values for x_1, x_2 (for simplicity we shall envisage a function of only two variables) and the function is evaluated at this point. x_1 is then incremented by an amount Δ and the function's value at the resulting point is compared with that at $\mathbf{b}_1$. If no improvement is obtained another exploratory move is defined by decreasing x_1 by 2Δ. If this move also fails x_1 is reset to its value at $\mathbf{b}_1$. Perturbation of x_2 is carried out in a similar manner about the point resulting from the previous exploration.

If the exploratory phase results in an improvement in the value of the function a new base point, $\mathbf{b}_2$, is set and the pattern phase is entered. Failure triggers a decrease in step size. We will now describe the details of this after first explaining the role of the pattern move.

Pattern phase

As soon as $\mathbf{b}_1$ and $\mathbf{b}_2$, or more generally $\mathbf{b}_n$ and $\mathbf{b}_{n+1}$, exist, a pattern move can be specified as the geometric production of the line joining $\mathbf{b}_n$ and $\mathbf{b}_{n+1}$ to point whose distance from $\mathbf{b}_{n+1}$ is equal to the distance from $\mathbf{b}_{n+1}$ to $\mathbf{b}_n$. This point is a 'candidate' to become a new base-point, $\mathbf{b}_{n+2}$. After each pattern move the exploratory phase is entered and is aimed at revising the pattern move. If there is an improvement over $\mathbf{b}_{n+1}$ in the value of the function either at the candidate point or after the exploratory moves have been performed from this point then a new base point is set, defining a new pattern move. Otherwise the rescue phase is entered.

Rescue phase

Failure, as defined above, results in a return to the current base point, $\mathbf{b}_{n+1}$, where the exploratory phase is again entered in an attempt to define a fresh pattern move. Further failure causes Δ to be decayed by the factor ρ after which the exploratory process is again carried out. Subsequent failure results in Δ being reduced still further. When Δ falls below a limiting value, δ, the search is abandoned.

APPLICATION OF THE GRAPH TRAVERSER TO NUMERICAL FUNCTION OPTIMIZATION

The Graph Traverser may be employed as a numerical optimizer if we let problem states correspond to n-tuples whose components are the variables of the function to be optimized. This function is used as the evaluation function (compare the use of the cost function of the Travelling Salesman problem as an evaluation function in solution by the Graph Traverser (Doran 1968)). The choice and ordering of the operators will depend on the particular optimization technique that is being implemented. In the case of pattern search it will consist of two operators which are applied to base points and which produce, when successful, new base points. They are:

(1) make a pattern move from the current base point and then carry out an exploratory search

(2) carry out an exploratory search from the current base point.

To complete the adaptation of the Graph Traverser we must inhibit backtracking, a search being terminated when both operators have been applied unsuccessfully to the current base point, i.e., when a Shen Lin search (Lin 1965) has been completed. At this point the step length is reduced and a fresh search is initiated from the current base point.

The similarity between the Graph Traverser and pattern search can be extended to the exploratory phase. Thus we have the Graph Traverser as one of its own operators. To effect this extension we must first define another list of operators consisting of $2n$ exploratory moves, a positive and a negative one forming a pair which is associated with each of the n variables of the function being optimized, i.e.,

$$[+\Delta_1, -\Delta_1, +\Delta_2, -\Delta_2, \ldots +\Delta_n, -\Delta_n].$$

Each subscript identifies the independent variable to which the associated exploratory move is applied. When a new best-valued node is generated the usual procedure (see earlier formal description) is to choose, as an operator to apply to it, the next in order on the list after the last one applied to it. To complete the transformation we must prevent this initialization and arrange that the first operator to be applied to the new best node is the successor of the one that produced it. The Graph Traverser is now adapted for pattern search function optimization.

EXPERIMENTAL SCHEME

As explained, the rank correlation coefficient, r, provides an approximate assessment of merit which can be made independently of any successful searches. By substituting different combinations of parameter settings we compute different values of r over T, finding those settings which maximize r. In the experiments the recursive optimization routine was entered after the first partial search of each problem and on every 20th succeeding partial search.

Optimization consisted of three stages.

(1) The formation of a list of pairs, each pair consisting of the depth in the tree of a terminal node (its *tree distance*) together with its *estimated distance* from the root.

(2) The tree distances were then rank correlated with the estimated distances and the resulting coefficient maximized by the Graph Traverser operating in *pattern search* mode. The optimized values of the parameters formed a *current estimate* of the optimum.

(3) The current estimate was then incorporated in a *cumulative mean estimate*, obtained by averaging all past estimated optima and the original values of the parameters. The mean estimate was then used by the evaluation function on subsequent searches.

PARAMETER OPTIMIZATION: RESULTS

Preliminary trials were made with the Eight-puzzle (*see* Schofield 1967), in which 8 sliding blocks must be re-arranged in a bounded 3×3 array so as to form a specified goal configuration,

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

where the zero stands for the empty square. The operator set consisted of all movements of a block made by sliding it unit distance into the space. The evaluation function was the $P+wS$ expression used by Doran and Michie, whose results indicated that the optimum for the adjustable parameter w lies on a rather flat surface in the region $\frac{3}{2} \leqslant w \leqslant 9$. The size of the partial search tree was 200. Four replicate runs, each with 25 randomly generated puzzles, were done, with w initialized to zero. The step length, Δ, was 0·1. The final values for w were 0·9, 2·9, 2·1, 1·6 respectively; 98 out of the 100 searches terminated successfully, as compared with uniform failure when the optimizing call was suppressed so that w remained fixed at 0.

With this preliminary encouragement, the Fifteen-puzzle was substituted with a search space of 16!/2 as compared with 9!/2. As evaluation function we used Doran and Michie's $\sum_{i=1}^{15} h_i^a p_i^b + cR$, and initial parameter settings $a=1$, $b=1$, $c=50$. Step lengths were 0·1, 0·2 and 4. Other conditions were as before, with the exception that the number of tree-distance/estimated-distance pairs used to compute r was limited to 50 (the number for an average tree in the Eight-puzzle runs was about 75).

Table 2 shows the results of the first run. The values to which a, b and c settled down are well within the ranges judged optimal from independent evidence. A more critical test was set up, in which GT4 was run on either 0, 1, or 2 Fifteen-puzzles in optimizing mode, and the values of a, b, and c were then frozen. Performance was then tested on a new set of 8 Fifteen-puzzles, with results shown in table 3. Evidently very few optimizations have proved sufficient to confer as much adaptive improvement as can be got within this limited framework.

A criticism is that the starting values are arbitrary, and might be said to embody external knowledge of the problem; for example, they are all positive. A natural 'default setting' for initializing each parameter in the *total* absence of externally acquired knowledge would perhaps be zero, and we are now studying the effects of imposing this condition. In the meantime we can summarize by saying that given a very little external knowledge, quite inadequate in itself for solving the problem, the program can very rapidly improve upon it.

puzzle	number of optimizations	average parameter settings		
		a	b	c
1	0	1·000	1·000	50·00
(unsolved)	1	1·025	2·100	42·00
	2	0·725	2·367	40·67
	3	0·575	2·292	32·67
2	4	0·435	2·532	46·27
(unsolved)	5	0·368	2·365	38·93
	6	0·326	2·179	40·65
	7	0·263	2·279	47·40
3	8	0·257	2·335	42·07
(unsolved)	9	0·222	2·255	42·07
	10	0·204	2·746	36·61
	11	0·229	2·779	36·28
4	12	0·222	2·733	34·44
(solved)	13	0·218	2·711	34·72

Table 2. Fifteen-puzzle: optimization of the parameters of the Doran–Michie evaluation function using a random sample of 5 puzzles. The 5th puzzle was solved before the first partial search had been completed. Lookahead tree of 200 nodes.

OPERATOR-SELECTION: METHODS

In step 4 of the *strategy* function c is applied to t_{min} to obtain Γ'_r.

The action of c with which we have experimented is restricted to choosing the jth operator in Γ' so that $r=j$. If some operators are more frequently useful than others when searching a given problem graph, we would like the preferred operators to come high on the list. Ideally we would like to associate different orderings with different *categories* of node (different 'images' in Sandewall's (1969) notation), arriving at these orderings automatically during the search process, and perhaps discarding, in course of time, all but a few favourites occupying the top places in each such ordering. We have not aimed so high, but have restricted ourselves to investigating whether

training regime	number of optimizations	parameter settings a	b	c	number of puzzles solved	total number of states generated
no training	0	1·000	1·000	50·00	0	4,000
puzzle 1 (unsolved)	2	0·725	2·367	40·67	3	3,833
puzzle 2 (unsolved)	3	0·592	0·950	50·00	0	4,000
puzzle 1 then puzzle 2 (both unsolved)	6	0·326	2·179	40·65	6	2,480
puzzle 2 then puzzle 1 (both unsolved)	5	0·563	2·574	42·13	5	3,226

Table 3. Fifteen-puzzle: performance on 8 randomly generated puzzles using parameter settings that were obtained after varying degrees of training. Lookahead tree of 200 nodes.

dynamic re-ordering can be feasible and profitable even for a single global ordering of operators. A positive answer would suggest *a fortiori* that self-improvement on a more impressive scale should be attainable for problem-spaces which have been subdivided into state-categories.

The method we have adopted is to promote an operator whenever it is selected by *strategy* and to demote it in the ordering whenever it *could* have been selected (in the sense that it was actually applied during the given call of strategy) but was not selected. In our experiments a demotion was always by one place, and a promotion by one less than the number of immediate descendants of the root node. This scheme is essentially the 'path popularity ratio' proposal of Michie (1967) modified along lines suggested by work of Longuet-Higgins and Ortony (1968). The 'path popularity ratio' is a measure of the frequency with which an operator directs the lookahead along a line which is subsequently adopted, as opposed to directing it away from the main path. Table 4 shows this statistic tabulated for the six symmetry classes into which 48 operators for the Eight-puzzle can be grouped. These

operators comprise all the macro-moves which can be formed from concatenations of the elementary moves, subject to the conditions:
(1) the problem is re-formulated so that X comprises only the 8!/2 states of the board in which the centre square is empty; it follows that every operator is the concatenation of an even number of moves, at least 4 in number. The identity operator is disallowed.
(2) no operator is formed by the concatenation of more than 8 moves.

category of operator	average frequency on final path / frequency off final path (path popularity ratio)			rank position of average path popularity ratio		
	sample A	sample B	average	sample A	sample B	average
	0·119	0·101	0·110	1	2	2
	0·075	0·084	0·080	3	3	3
	0·108	0·143	0·126	2	1	1
	0·036	0·034	0·035	6	6	6
	0·070	0·077	0·074	4	4	4
	0·055	0·046	0·051	5	5	5

Table 4. Eight-puzzle: path popularity statistics obtained with random development using the 48 macro-moves. Each sample consisted of 50 randomly generated puzzles. Lookahead tree of 50 nodes.

It should be added that for the purpose of the experiment summarized in table 4, the function c was redefined to give a random selection of the next operator from Γ'.

Table 4 shows that marked differences do indeed exist between the different categories of operator on the criterion of path popularity. This was sufficient encouragement to run a test of the promotional scheme previously outlined.

In terms of the formal description, *retrace* is extended so as to reduce the index of the selected operator by *size* (X'_{root}) − 1 and to increment by one the indices of all other operators leading from the root node.

Table 5 shows the results obtained using the same two samples of randomly constructed puzzles as those of table 4. The initial ordering of the 48 operators was random in each case. The average final ordering, shown in table 5, agrees exactly with that of the path popularity ratios.

category of operator	rank position after promotion		
	sample A	sample B	overall average
	2	1	2
	3	3	3
	1	2	1
	5	6	6
	4	4	4
	6	5	5

Table 5. Eight-puzzle: results of promotional experiment. Each sample consisted of 50 randomly generated puzzles. Lookahead tree of 50 nodes.

At this point the operator ordering was frozen and split into a top 24 and a bottom 24. Duplicate re-runs were then done using the two reduced operator sets with the promotional mechanism inhibited. The results, shown in table 6, demonstrate that the program has sorted the 48 operators into a 'good' and a 'bad' subset; use of the two sets is associated with a marked difference in performance, whether assessed on the proportion of problems solved or on the computational cost per search.

selection procedure	sample	percentage of puzzles solved	average number of states generated / puzzle
top 24 operators on list after promotion	A	100%	62·4
	B	100%	72·9
	average	100·0%	67·7
bottom 24 operators on list after promotion	A	24%	173·3
	B	30%	166·4
	average	27·0%	169·9

Table 6. Eight-puzzle: results of a promotional scheme for operator selection. Lookahead tree of 50 nodes.

DYNAMIC SHRINKING

This result provoked the idea that the program might improve its own performance and economize its effort from the start, discarding an operator which was already at the bottom of the list if it became a candidate for further demotion. This policy of dynamic shrinking of Γ' was given a preliminary trial, with a bound set on the shrinking process so that the size of Γ' could not fall below 24. The results were satisfactory, and as far as they went indicated convergence to steady state conditions similar to those of table 6.

DISCUSSION

Our results establish the feasibility of automatic optimization procedures in heuristic search with respect both to state-evaluation and preference-ordering of operators. Improvement of performance is not dependent, under either heading, on having problems sufficiently tractable to be within reach of the program in its non-adapted state. Optimization proceeds through modifying

the action of f and c respectively without modifying their structure, i.e., through trial-and-error modifications of global parameters.

Substantial further gains in performance should accrue from two extensions which we are now pursuing:

1. *Local smoothing.* Much of the program's *time* is consumed in the application of large numbers of operators (typically the whole set) to a small number of unproductive nodes, while most of its *progress* is achieved through a larger number of nodes, on each of which only a little development work is expended. The 'obstructive' nodes are those to which f has assigned a misleadingly low value; in other words $d(x, x^*) \ll \delta(x, x^*)$. The program ought, but in its present form does not, gradually re-value such a node in the light of accumulating evidence from its high-valued immediate descendants. A simple approach, reminiscent of the use in genetics of progeny-testing to improve the assessment of an individual's genotypic value, would be to use f' rather than f to guide the search, where

$$f'(x) = \frac{f(x) + w\sum_{i=1}^{j-1} f(\Gamma_i'(x))}{1 + w(j-1)}$$

and w is a weighting coefficient. Preliminary results are positive.

2. *Regionalization.* We do not have any systematic method for partitioning problem spaces in general into regions for the purpose of associating independent promotional processes with these regions. Such partitioning, even if done crudely, can be surprisingly effective. Samuel's (1967) 'signature tables'; and Michie and Chambers' (1968) 'boxes' are relevant cases in point. In the experience of human solvers the Eight-puzzle exhibits marked interaction between different features of problem states and the relative utilities of different operators. It should therefore provide a good testing ground for regionalized promotional systems, in which the features used for partitioning into regions are in the first instance supplied to the program from outside. The possibility to which we attach most importance arises from the use of dynamic shrinking in the independent regional promotional processes. This could result in the action of *strategy* being gradually re-structured during search until only a small specialized operator set was associated with each region, forming the basis of small and economical lookahead trees. A self-mediated transformation of this kind would constitute a step towards the compression of lookahead-type strategies into tabular form, a process which seems to play a role in the acquisition of skill by human solvers.

Acknowledgements

This work was done as part of a programme of research into machine simulation of learning, cognition and perception supported by the Science Research Council. One of us (R.R.) is in receipt of an S.R.C. Research Studentship which is here gratefully acknowledged.

REFERENCES

Doran, J.E. Michie, D. (1966) Experiments with the Graph Traverser program. *Proc. R. Soc. A*, **294**, 235–59.

Doran, J. (1967) An approach to automatic problem-solving. *Machine Intelligence 1*, pp. 105–23 (eds Collins, N.L. & Michie, D.). Edinburgh: Oliver and Boyd.

Doran, J. (1968) New developments of the Graph Traverser. *Machine Intelligence 2*, pp. 119–35 (eds Dale, E. & Michie, D.). Edinburgh: Oliver and Boyd.

Ernst, G.W. & Newell, A. (1969) *GPS: A Case Study in Generality and Problem Solving*. New York and London: Academic Press.

Hooke, R. & Jeeves, T.A. (1961) 'Direct Search' solution of numerical and statistical problems. *J. Ass. comput. Mach.*, **8**, 212–29.

Lin, S. (1965) Computer solutions of the Travelling Salesman problem. *Bell System Tech. J.*, **44**, 2245–69.

Longuet-Higgins, H.C. & Ortony, A. (1968) The adaptive memorization of sequences. *Machine Intelligence 3*, pp. 311–22 (ed. Michie, D.). Edinburgh: Edinburgh University Press.

Marsh, D.L. (1969) LIB GRAPHTRAVERSER. *Multi-POP Program Library Documentation*. Edinburgh: Department of Machine Intelligence and Perception, University of Edinburgh.

Michie, D. (1967) Strategy-building with the Graph Traverser. *Machine Intelligence 1*, pp. 135–52 (eds Collins, N.L. & Michie, D.). Edinburgh: Oliver and Boyd.

Michie, D. & Chambers, R.A. (1968) BOXES: an experiment in adaptive control. *Machine Intelligence 2*, pp. 137–52 (eds Dale, E. & Michie, D.). Edinburgh: Oliver and Boyd.

Michie, D. & Ross, R. (1969) A comparison of Powell's general purpose function optimizing algorithm with that of Hooke and Jeeves, using a real-time criterion. *Research Memorandum MIP–R–36*. Edinburgh: Department of Machine Intelligence and Perception, University of Edinburgh.

Nelder, J.A. & Mead, R. (1965) A simplex method for function minimization. *Computer Journal*, **7**, 308–13.

Newell, A., Shaw, J.C. & Simon, H.A. (1957) Preliminary description of general problem solving program – I (GPS-1), *CIP Working Paper No. 7*. Pittsburgh: Carnegie Institute of Technology.

Powell, M.J.D. (1964) An efficient method for finding the minimum of a function of several variables without calculating derivatives. *Computer Journal*, **7**, 155–62.

Quinlan, J.R. (1969) A task-independent experience-gathering scheme for a problem solver. *Proc. International Joint Conference on Artificial Intelligence*, pp. 193–7 (eds Walker, D.E. & Norton, L.M.).

Samuel, A.L. (1959) Some studies in machine learning using the game of checkers. *IBM J. Res. Dev.*, **3**, 211–29.

Samuel, A.L. (1967) Some studies in machine learning using the game of checkers, 2 – recent progress. *IBM J. Res. Dev.*, **11**, 601–17.

Sandewall, E.J. (1969) A planning problem solver based on look-ahead in stochastic game trees. *J. Ass. comput. Mach.*, **16**, 364–82.

Schofield, P.D.A. (1967) Complete solution of the 'Eight-puzzle'. *Machine Intelligence 1*, pp. 125–33 (eds Collins, N.L. & Michie, D.). Edinburgh: Oliver and Boyd.

Spendley, W., Hext, G.R. & Himsworth, F.R. (1962) Sequential application of simplex designs in optimization and evolutionary operation. *Technometrics*, **4**, 441–61.