8

NEW DEVELOPMENTS OF THE GRAPH TRAVERSER

JAMES DORAN
EXPERIMENTAL PROGRAMMING UNIT
DEPARTMENT OF MACHINE INTELLIGENCE AND PERCEPTION
UNIVERSITY OF EDINBURGH

INTRODUCTION

This paper describes some recent experiments with a computer program which is capable of useful, or at least interesting, application to a number of different problems. The program, the Graph Traverser, has been described in detail in a previous paper (Doran & Michie 1966). However, we shall here need to view the basic algorithm from a rather more general standpoint, corresponding to an actual extension in the flexibility of the program, so that a restatement of what the program can do is desirable.

The Graph Traverser, which is written in Elliott 4100 Algol, is potentially applicable to problem situations which can be idealised in the following way (see for comparison Newell and Ernst 1965). There is given a set of 'states', which are connected by a set of 'transformations', or, as I shall call them, 'operators'. An operator will be applicable to some, but not necessarily all, of the states and two distinct operators applied to either the same or distinct states may each give the same state as end-product. Most of the concepts to be used here which are related to the use of operators were discussed in a paper by Michie (1967).

This type of problem situation is represented in Fig. 1 by a graph (in the mathematical sense) to which have been added various labels. In this representation states correspond to nodes of the graph, and operators to labelled arcs—*a*, *b*, *c* in this quite arbitrary case. Notice that associated with each node (or state) is a triad of integers. These represent the way in which the

program handles the 'structure' that is typically associated with a state, for with each node it holds an integer matrix, the dimensions of which will depend upon the particular problem. It is this structure which embodies the additional information which the program uses to carry out a heuristically directed search for a solution rather than a systematic search.

What 'idealised' problems can be posed in this context? It is convenient to recognise two types (for a fuller classification see Doran & Michie 1966). These we may call informally 'state-search' problems and 'path-search' problems. In a state-search problem we wish to find a state with some particular property. For example, and this will be relevant later on, we might seek the state for which some associated cost was smallest. In Fig. 1 we might seek

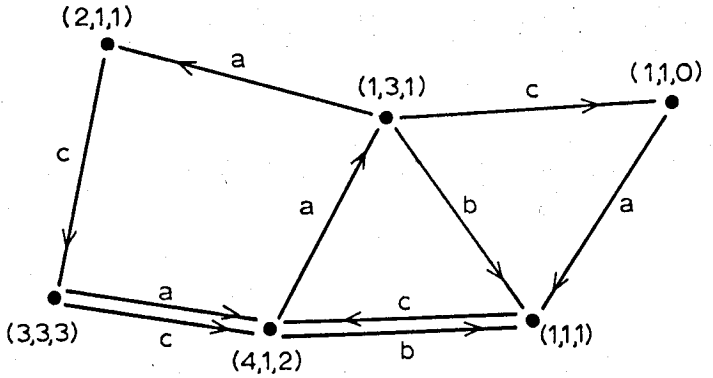

FIG. 1. An example of a problem graph. The nodes correspond to problem states, and the arcs to operators (transformations).

the node for which the sum of the triad of integers was least. In a path-search problem we seek a path between two states, that is, a sequence of operators which take us from one state to another. Referring again to Fig. 1, we might require a path from the node (3, 3, 3) to the node (1, 1, 0). A solution would be (*a*, *a*, *c*). Note that the sequence of states (3, 3, 3), (4, 1, 2), (1, 3, 1), (1, 1, 0) does not immediately give the corresponding operators (though the converse is true) and thus is a solution of a weaker kind. To appreciate this point the reader must realise that although in Fig. 1 he can see the whole of the problem graph laid out before him, the program must work from the definitions of the states and the operators, and the application of the latter to the former.

Before describing the Graph Traverser search method, it must be emphasised that these problems are always to be regarded from a practical rather than a theoretical standpoint. Actual solutions must be found to actual problems within a reasonable search time. A further point is that in a given problem situation the way in which the states and operators are defined is likely to be of great importance. Although this aspect of the matter is outside the present program schema, it must not be overlooked. In particular, for a state-search problem, the definition of the operator set may be an important part of the method of solving a problem, rather than of the problem situation itself.

THE GRAPH TRAVERSER SEARCH METHOD

The program proceeds by selecting an initial state either as directed or at random or, in a path-search problem, from the specification of the problem, and then by applying operators to this state and to the new states it thus generates, until the fragment of the problem graph which it has built up provides the information needed to find the solution. The structure which is constructed by the program is a 'tree', since although the program may generate a state more than once, it will only store it once, and will only remember one path to it.

When growing its search tree the program repeatedly encounters two problems. The first is to decide to which state next to apply operators. The second is, having chosen a state, to decide which operators actually to apply. The program's reaction to the first problem involves the 'evaluation function', and to the second the 'develop procedure'.

Briefly, the evaluation function must assign to each state encountered a value based on its structure. When a state must be selected for development (i.e., application of operators) that with the smallest value (smallest rather than largest since in path-search problems the value may often be identified with estimated distance from the 'goal') is chosen, avoiding states which have already been fully developed.

When a state has been chosen for development the develop procedure must decide two things, (i) which operators of those applicable actually to apply (this is the process of *operator selection*) and then having applied them (ii) whether or not to mark the state as fully developed. If a state is marked as fully developed then it will never be chosen for development again. If it is not so marked then the program leaves itself the option of taking up development of that state again. A state such as this which is neither quite undeveloped nor fully developed we call *partially developed.*

Perhaps the simplest thing to do when developing a state is to apply all possible operators and be done with it. The earliest work with the Graph Traverser used this approach, which does not involve partial development. At the other extreme, at most one operator may be applied in a development, the order of application being fixed but arbitrary, and the state being considered as partially developed until the last operator has been applied. This approach, which has an important application to be discussed later in the paper, is illustrated in Fig. 2, which shows the program searching the problem graph of Fig. 1. The arrows are reversed corresponding to the actual structure of the search tree. The task is to find the node the sum of whose integers is least. The program arbitrarily selects (4, 1, 2) as its starting point. As indicated above it applies just one operator during a development, and will not mark a node fully developed until all the operators which are applicable there have actually been applied. The order of application is always *a, b, c.* The value assigned to a node by the evaluation function is just the sum of its integers. The letters U, P, F indicate the status of a node—undeveloped, partially developed, or fully developed.

It is a consequence of this algorithm that the program carries out repeated developments of a node until either a "descendant" node of lower value is found, or no more applicable operators are available. In the latter event a "disconnected" development will ensue. Notice in (*f*) of Fig. 2 that when a

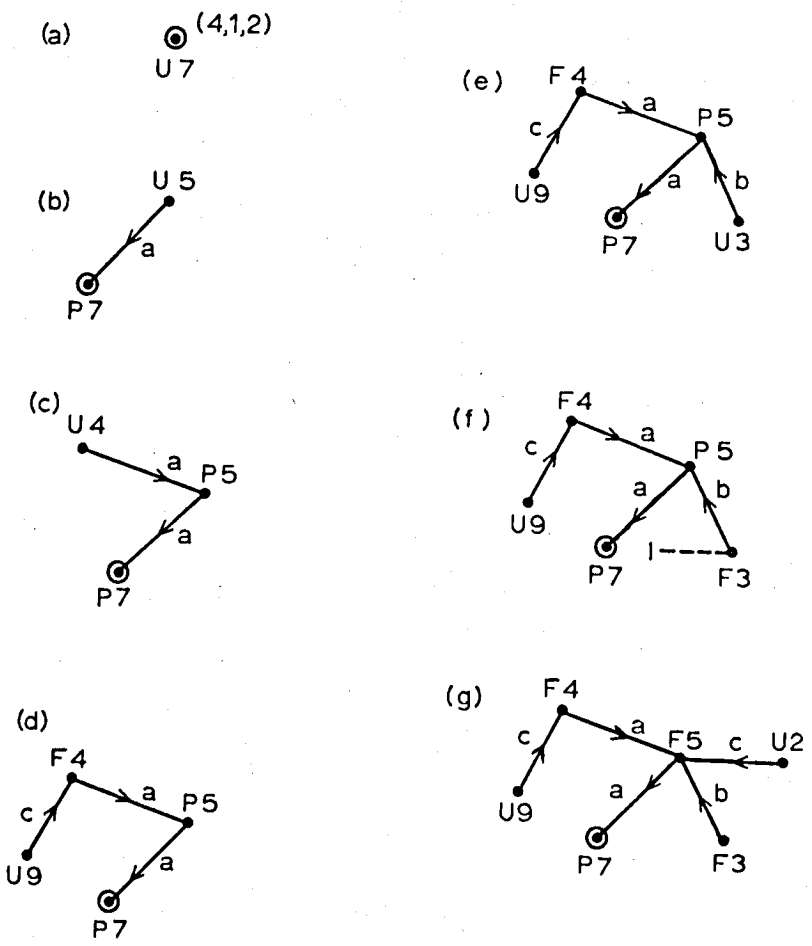

FIG. 2. The Graph Traverser searching the graph of Fig. 1. Successive diagrams show successive developments. The node from which the search starts is ringed for clarity. The letters U, P, F indicate the status of a node—undeveloped, partially developed, or fully developed. The figures give the values calculated for the nodes. The arcs are reversed since in the search tree the program stores a pointer to the ancestor of a node, not pointers to its descendants.

node already on the tree is generated its regeneration is ignored. Had the new encounter provided a shorter path to the node than that already known, the tree would have been modified accordingly. The search ultimately succeeds, but whether the best node would be found in practice depends upon the particular rule used to decide when to terminate the search.

This brings us to the most important point that these various decisions which the program must make depend heavily upon guidance from the user in a particular problem situation, although ways of making the program itself improve its decisions have been suggested elsewhere (Doran 1967, Michie 1967). In practice the develop and evaluate procedures, and any other routines which need to be respecified, are programmed for each application and embedded within the program proper.

THE TRAVELLING SALESMAN PROBLEM

The Travelling Salesman is one of the best known problems of operations research. It is very simply stated. A salesman, starting at a given city must visit $n-1$ other cities, each just once, before returning to base. There is a fixed distance to travel between any particular pair of cities and he wishes the whole tour to be as short as possible. The problem is to specify an efficient way for him to find the shortest tour. The word 'efficient' here is crucial. One can find the shortest tour by calculating the length of all possible tours. But if there are n cities, there are $\frac{(n-1)!}{2}$ possible tours, so that this approach is usually much too laborious.

This version of the problem corresponds to marking a score or so points on a piece of paper, and connecting them up so that the tour goes through each just once, and is as short as possible in length. More generally, we stop thinking in terms of cities or points on a plane surface, and simply say that a cost is assigned to travelling between each pair of 'cities' (independent of direction) in some way which does not concern us.

Linear programming and dynamic programming methods can be applied to the Travelling Salesman problem but only with difficulty, particularly if the number of cities involved is more than 20, say. Yet the problem is one of real importance, and in modified and disguised forms occurs repeatedly in industrial and economic situations. We may, for example, think of ships visiting ports, or less obviously, of an automatic press punching a series of holes in specified locations in pieces of sheet metal.

Shen Lin (1965) and G. M. Roe (1966) have recently explored with considerable success heuristic search methods which provide near-optimal tours, but avoid spending much time trying to turn a near-optimal tour into an optimal tour. By an 'optimal tour' I mean any tour with least possible cost.

Lin's approach, which is the one I shall discuss here (Roe's is broadly similar), defines for the Travelling Salesman problem a certain set of, in our terminology, operators. These are called (using Roe's terminology now) 'cuts'. The set of all n-cuts is the set of all possible ways of breaking any n links of a given tour and putting it together again, as a single loop, in some other way. In particular, a 2-cut involves reversing the direction in which a sequence of the cities is visited, and a 3-cut involves removing a sequence of cities from the tour and then reinserting this sequence, possibly reversed, at the same or some other point. If there are, say, 20 cities in a tour then there are 170 possible

2-cuts, and about 5000 possible 3-cuts. Lin uses 3-cuts (his experiments indicate that 3 is the best value of n for this purpose) to convert the set of all possible tours into a problem graph in the Graph Traverser sense, and then adopts an algorithm now to be described.

Lin's algorithm involves a number of 'searches'. In each search an initial tour is selected at random, and the 3-cuts are systematically checked until one which will generate a new tour with lower cost has been found. This is then applied. The process is repeated with the new tour until an improvement

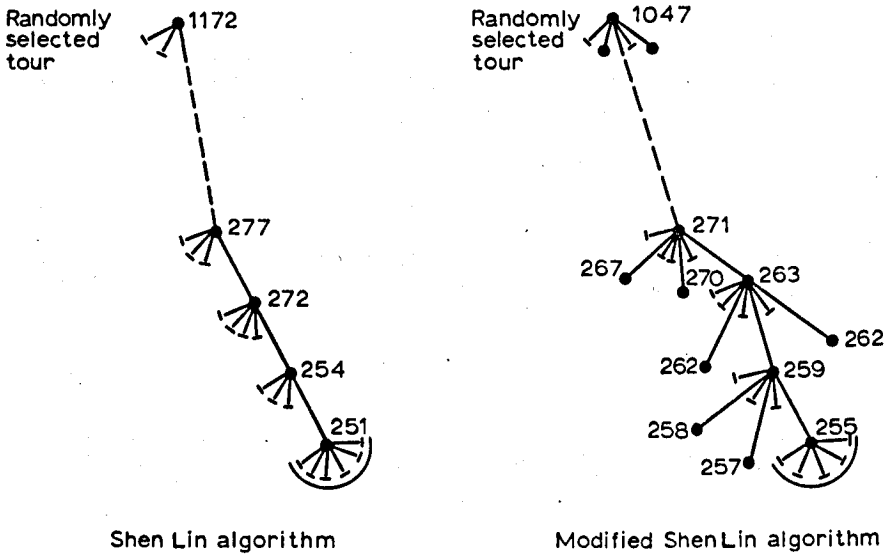

FIG. 3. Shen Lin's algorithm for solving the Travelling Salesman problem, and a modification of this to permit three new tours to be generated at each development. The nodes correspond to possible tours, and the figures give the corresponding costs. The blocked arcs indicate 3-cuts which do not reduce the cost. Both algorithms terminate when a tour is encountered which no 3-cut improves (a 3-opt tour).

is again obtained, and so on until ultimately a tour is reached which no 3-cut improves (Fig. 3). This tour is then the 'locally optimal tour' or '3-opt' tour associated with the initial random starting configuration. In all tens of searches will be made. Experiment shows that for all except the smallest problems, not all, and possibly none, of the 3-opt tours obtained will be optimal, nor in general will they be all distinct.

However, as Lin points out, after some experience with the algorithm it is possible to estimate in advance for an unsolved problem of particular size the probability that a search will give an optimal tour. It is then possible to calculate how many searches are necessary to reduce to a level low enough to be negligible the probability that an optimal tour still has not been discovered. In this situation it is not necessary to push on with an individual search until an optimal tour has definitely been located—a process likely to be both

inconclusive and very time consuming. Lin's program is for an IBM 7094 II, and is very fast, most relevant times being measured in seconds.

Lin also employs various techniques to use information collected in the earlier searches to cut computation in the later. These techniques will not concern us here but are interesting and important.

APPLICATION OF THE GRAPH TRAVERSER TO THE TRAVELLING SALESMAN PROBLEM

The Graph Traverser has been applied to the Travelling Salesman problem with two main objectives:

(i) to establish that it *is* applicable to this situation, and that all the basic results obtained by Lin can be obtained using the Graph Traverser, though in practice at a slower rate since the Graph Traverser aims at generality rather than speed; and

(ii) to use the flexibility and facilities of the Graph Traverser to explore the worth of other search methods more complex than that used by Lin.

The actual mapping process which takes Lin's algorithm into that of the Graph Traverser has already been indicated. A state of the problem graph corresponds to a possible complete tour (*not* to a city), and an operator to a 3-cut as already described. All the 3-cuts are applicable to any tour, but the develop procedure when called only actually applies one 3-cut—the first one which it considers which will, it calculates, generate a cheaper tour. The cuts are considered in an arbitrary but fixed order. To implement Lin's strategy, the develop procedure then marks the tour as fully developed. The effect of this is that when a 3-opt tour is encountered, so that no new undeveloped tour is generated before the current tour is declared fully developed, the search must terminate (Fig. 3).

Using the Lin algorithm which is an example of a 'conditional choice strategy' (see Michie 1967) the Graph Traverser was applied to randomly generated problems involving up to 20 cities, and to two published problems, the 20-city problem of Croes (1958), and the 25-city problem of Held and Karp (1962). The results obtained were consistent with those obtained by Lin. For the Croes problem Lin quotes 13 out of 40 for the proportion of successful searches, and for the Held and Karp problem 26 out of 40. The corresponding figures using the straightforward Graph Traverser implementation of Lin's algorithm were 3 out of 10 and 13 out of 16. A typical search on the 20-city problem involved developing about 45 tours, and on the 25-city problem about 70 tours. These results indicated that the first research objective had been achieved. An Elliot 4120 processor was used to obtain these and the other results described in this paper.

MODIFICATION OF SHEN LIN'S ALGORITHM

How can Lin's algorithm be improved? From the standpoint of the Graph Traverser program it is a rather simple strategy, and presents a number of opportunities for modification.

The first modification which suggested itself was to require the develop procedure to produce, where possible, more than one cheaper tour. Since the cost of a tour is being used as its value, the program will always select the cheapest descendant tour for the next development. This trades the extra computation time required for each development against, hopefully, a more successful search. In practice, three descendant tours were generated at each development (see Fig. 3). Table 1, which refers to the Croes problem, shows

TABLE 1

Costs of 3-opt tours obtained with the Graph Traverser for a particular Travelling Salesman problem. The three algorithms used are described in the text. In every case tours with the same cost are identical.

CROES 20-CITY PROBLEM
OPTIMAL TOUR HAS COST 246

Random starting tour	Lin algorithm	Modified Lin algorithm	Graph Traverser algorithm
A	246	246	246
B	251	246	251
C	252	252	246
D	252	252	246
E	252	255	246
F	246	252	246
G	246	252	246
H	252	246	246
I	252	252	246
J	251	252	246
Occurrences of optimal tour	3	3	9
Approximate time for each search in minutes	15-20	20	30

that each search took rather longer with no noticeable improvement in the results. This agrees with Lin's remark that 'Attention should be directed to finding improvements with a minimum amount of computation rather than to making the maximum improvement possible at each step'. What seems to be happening is that there are only a very few 3-opt tours, all quite similar, and which of these is encountered during any particular search depends only upon the final 'direction' of approach to the critical region. What happens earlier is quite irrelevant.

The mutual proximity of the 3-opt tours, in the sense that one can be converted into another by a very few 3-cuts (not, of course, by just one) suggests that terminating a search as soon as a 3-opt tour is encountered may be premature, and that a certain amount of additional investigation might be worthwhile—even though such additional investigation is likely to be very costly in time. Roe suggests something along these lines.

It seems appropriate to invoke the Graph Traverser's ability to partially develop tours, leaving them available for further development at a later time.

This process was illustrated in detail in Fig. 2. Specifically, instead of the develop procedure applying the first 3-cut it encounters which will reduce the tour cost, and then marking the tour as fully developed, this latter step is omitted unless there exists no unused beneficial operator at all. In consequence, when a 3-opt tour is found, the search does *not* terminate, but continues by restarting the development of the tour from which the 3-opt

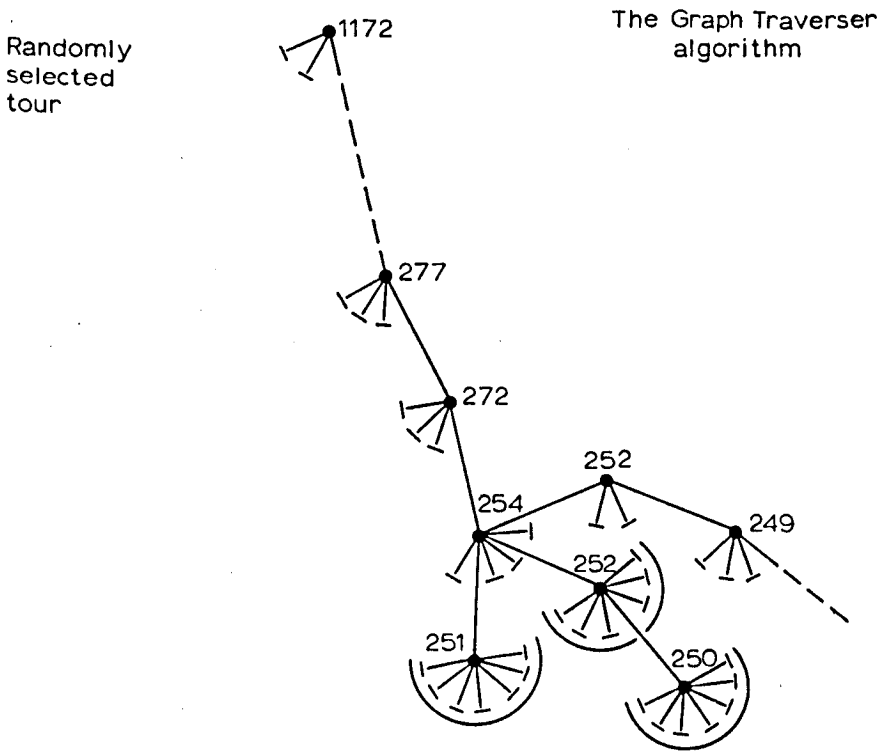

FIG. 4. Shen Lin's algorithm augmented by the use of the Graph Traverser's partial development facility. A search is continued beyond the discovery of a 3-opt tour by continuing the development of the best partially developed tour. The branches are explored from left to right, and the search is terminated when the program has been forced to 'retreat' a specified number of times.

tour was itself generated (Fig. 4). The search is continued in this way until some fixed number of tours has been fully developed. Since each full development corresponds to the program finding itself forced to 'retreat' to a tour of greater cost, this is a convenient and reasonable way to terminate a search. Of course, the 'result' of such a search is the cheapest tour encountered, not merely the last encountered.

EXPERIMENTAL RESULTS

Table 1 shows results obtained by this revised algorithm. For this problem, of which Lin remarks that it 'seems harder to solve than most 20 city prob-

lems', it almost uniformly succeeds in finding the best tour when the Lin algorithm fails. A search was terminated at the fourth retreat, which roughly doubled the time required for it.

Thus, although the average effectiveness of the searches has been increased, the time required for each has also risen. To decide whether there has been an overall improvement in performance we look at the matter in terms of the question 'How long must we run the program on this problem to achieve some selected probability that we have obtained an optimal tour?' Using the Lin algorithm the best estimate of the probability that a particular randomly initiated search will produce an optimal tour is clearly $p_1 = 0{\cdot}3$. It follows that the number of searches required to ensure a probability of 0·05 or less, say, that an optimal tour has not been found is found by solving the inequality in N_1

$$(0{\cdot}7)^{N_1} \leqslant 0{\cdot}05$$

This gives $N_1 \geqslant 9$. Supposing unit time for each search, the time required is 9 units.

Using the Graph Traverser strategy the estimated probability of a search succeeding is $p_2 = 0{\cdot}9$. The corresponding calculation gives $N_2 \geqslant 2$. Since the search time doubles, the time required is 4 units. Thus the time required is reduced by a factor of more than 2. The reader will easily see that this factor is broadly independent of the probability of error initially specified.

Thus the conclusion to be drawn from this admittedly very small quantity of evidence is that the partial development extension of Lin's algorithm is worthwhile.

Further investigation of this conclusion required results gathered from a range of problems. However this led to a snag. Fairly *difficult* problems were required, for otherwise Lin's basic strategy leaves little room for improvement. Suitably difficult problems were too large, however, for the limited machine store size immediately available.

An experiment was therefore carried out using 15 city problems defined by randomly generated cost matrices, and using *2-cuts* as operators rather than 3-cuts. The 2-cuts are a smaller, less drastic set of operators which Lin found to be less powerful in practice even allowing for the sharp reduction in the time required for each search when using them. Thus small problems which would otherwise be too easy are made suitably difficult by reducing the effectiveness of the operator set.

Eleyen random 15-city problems were generated, and ten searches were carried out on each, each search being continued until the 30th retreat occurred. Table 2 summarises the results obtained. The following points are important:

1. Column 2 gives the average number of cuts applied or considered for application by the program before reaching the nth retreat in the search, where n is indicated by the figure in column 1. Just over 300 operations are needed to reach the first retreat—the point at which Lin's search would

terminate—and thereafter just under 100 are on average needed to reach each successive retreat. These figures are effectively a measure of *time*.

2. Column 3 gives the mean cost of the best tour obtained up to the nth retreat and column 4 the observed probability that an optimum cost tour has already been found. What *was* the cheapest cost was decided by inspection of all the results obtained for the given problem. In no case was the matter in doubt. We see that the probability that a Lin search would be successful is about 0·09.

Note that there appears to be a sharp improvement if the search is carried only a little beyond the point where Lin terminates.

TABLE 2

Results obtained by applying the Graph Traverser algorithm to eleven randomly generated 15-city problems. The means are calculated over 110 searches in all. The final column refers to a 'typical' problem of this type and envisages several searches each terminating after a fixed number of retreats. The basic Lin algorithm would halt at the first retreat.

Retreat at which search ends	Mean number of operations per search	Mean cost of cheapest tour found in search	Observed probability that optimal-cost tour has been found in search	Total number of operations needed to give 0·99 probability of optimal tour
1	317	201	0·09	15316
2	416	193	0·22	7783
3	517	190	0·31	6439
4	605	187	0·34	6795
5	704	186	0·36	7172
6	798	185	0·36	8130
7	902	184	0·38	8636
8	995	183	0·43	8221
9	1090	182	0·47	7842
10	1176	181	0·49	8021
15	1651	179	0·55	9643
20	2105	178	0·60	10579
25	2570	177	0·65	11413
30	3025	177	0·68	12165

3. The last column attempts to combine columns 2 and 4 to discover if the loss in time is balanced by the gain in power. The figures give the number of operations, that is the number of inspections or applications of 2-cuts, needed to give a probability of 0·99 that an optimum tour has been located, assuming that we are faced with a new 15-city problem generated in the same way. The total number of operations required with termination on the rth retreat is $-2i_r/\log(1-p_r)$ where p_r is the estimated probability that an optimum tour will be found during a search thus terminated (column 4), and where i_r is the observed average number of inspections up to the rth retreat (column 2). These figures are artificial in two ways. First they assume that a non-integral number of searches is meaningful, in effect some smoothing has

been carried out, and secondly, and more important, it is assumed that all the problems generated by the random process are of equal difficulty. This latter assumption is certainly false. However, there is no reason to think that this will cause the figures to be misleading for our purposes.

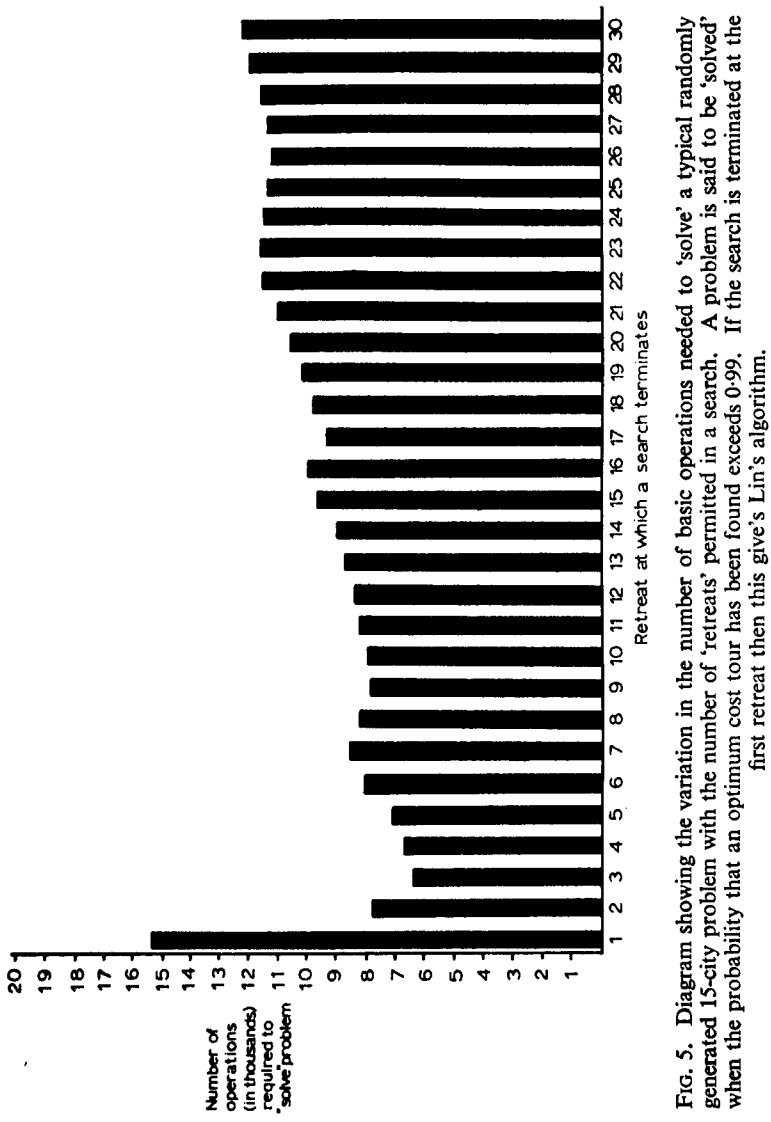

FIG. 5. Diagram showing the variation in the number of basic operations needed to 'solve' a typical randomly generated 15-city problem with the number of 'retreats' permitted in a search. A problem is said to be 'solved' when the probability that an optimum cost tour has been found exceeds 0·99. If the search is terminated at the first retreat then this give's Lin's algorithm.

Fig. 5 shows the required number of operations plotted against the length of search permitted. Again there is striking evidence in favour of continuing searches beyond the point at which Lin would terminate, and we see that the best results are obtained when a search is terminated at the third retreat. The apparently rhythmic fluctuation has no significance.

To sum up, although work is required on larger and more varied problems, it does appear that the Graph Traverser search method is capable of improving significantly Lin's already very successful algorithm. The improvement is obtained by continuing a search beyond the moment at which the first retreat occurs, that is, spending time searching the neighbourhood of locally optimal tours and proportionately reducing the total number of searches carried out.

Put another way, which may be more illuminating, the modification to Lin's algorithm is to start a proportion of the searches (as he defines them) from tours already known to be good, rather than always to start searches from randomly generated tours. This follows from the fact that when a retreat occurs, and development of an already partially developed tour is restarted, this is virtually equivalent to beginning a new search from that tour.

ON-LINE COLLABORATION WITH THE GRAPH TRAVERSER

It seems probable that in the foreseeable future problem-solving programs will be designed to interact with the user, rather than operate in isolation, when they are really intended to be useful. Such an arrangement permits the user to support the program in those aspects of the problem-solving procedure where it cannot cope adequately by itself. I shall therefore end this paper with a discussion of a version of the Graph Traverser adapted for such collaboration in the context of two simple sliding block puzzles, the Eight-puzzle and the Fifteen-puzzle, and with some more general remarks.

The Eight- and Fifteen-puzzles have been described in detail elsewhere (e.g., Doran & Michie 1966, Schofield 1967). Briefly, the Eight-puzzle consists of eight unit square pieces in a 3×3 frame, the pieces being numbered from 1 to 8, and there being one empty cell. The problem is to convert one configuration of the pieces into another by sliding the pieces around. Thus we might seek a sequence of *moves*, where a move means sliding one piece into the adjacent space, which converts

7	5	4		1	2	3
1	0	8	into	8	0	4
2	6	3		7	6	5

Provided that the two configurations regarded as permutations of the digits 0 to 8 have the same parity, at most 30 moves will be required.

The Fifteen-puzzle is the same puzzle scaled up to a 4×4 frame, and with the pieces numbered from 1 to 15. For this puzzle, although the same parity rule holds, the maximum number of moves needed to convert one configuration into another is not known.

Extensive results have already been published from the application of the Graph Traverser to these puzzles (Doran & Michie 1966, Doran 1967, Michie 1967). The problem is of the 'path-search' variety, unlike the Travelling Salesman application, and the evaluation function plays a much more important role, for it has the difficult task of estimating how close con-

figurations generated in the search are to the target configuration. As a result of this earlier work, fairly good evaluation functions are available for the puzzles.

A very simple way to bring a human solver into the scheme is to arrange that he takes over the function of the develop procedure, and this has been implemented. Every time the program wishes to develop a configuration the

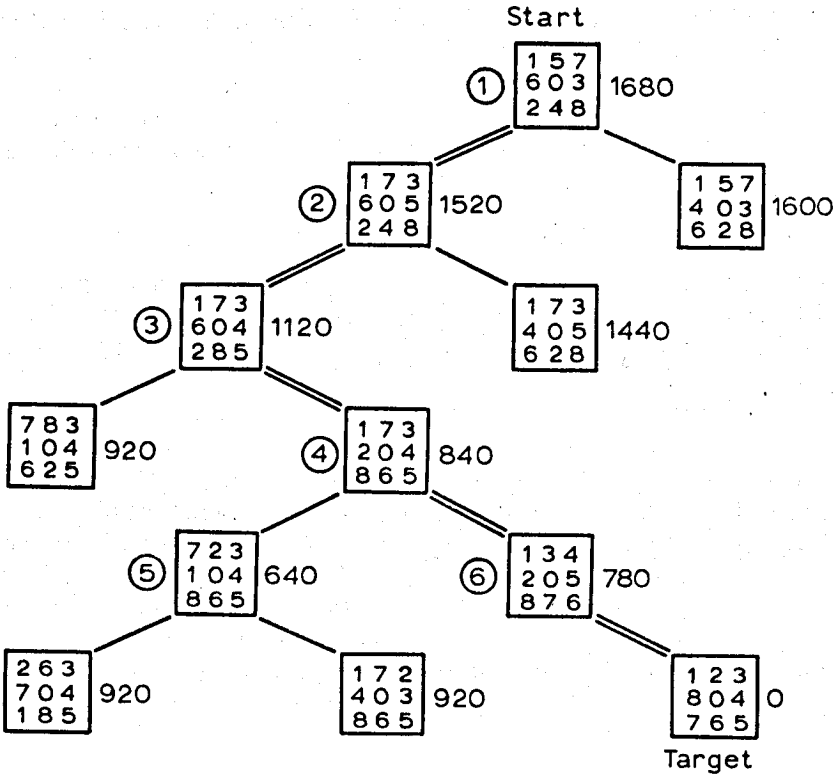

FIG. 6. The search tree constructed collaboratively by the Graph Traverser and the author to solve a particular Eight-puzzle configuration. The ringed figures give the order of development, and the unringed figures the values assigned by the evaluation function. Notice that after the fifth development the program rejects both the macro-moves suggested by the author in favour of a continuation previously itself rejected.

human solver is presented with it, and asked to specify sequences of moves (operators) which he suggests be applied. Once he has suggested all the moves or move sequences he thinks reasonable, he indicates this, the moves are implemented, and the configuration is henceforth regarded as fully developed. Thus there is no partial development.

From the user's point of view the sequence of events is as follows. The program asks whether it is the Eight-puzzle or the Fifteen-puzzle which is to be solved. Given the answer that it is the Eight-puzzle, say, the program then types out the standard goal configuration and asks the user to type the

starting configuration. This done, the program asks the solver what moves he suggests for this configuration. The solver might suggest three possible sequences of moves, and then type ALL. The program implements the moves, decides which of the three resulting configurations it favours, and presents this to the user for more moves to be suggested. This continues until either the goal is located, when the program will output the solution path with relevant statistics, or the solver types RESIGNS. The program will ignore impossible moves or typing errors.

The configuration selected by the program for 'human' development will not necessarily be the outcome of one of the moves or move sequences last suggested by the solver. As always with the Graph Traverser, disconnected developments are possible. That is, if none of the current suggestions of the solver is sufficiently attractive, the program will return to an earlier possible continuation not thought the best at the time. This point is illustrated in Fig. 6, which shows the search tree built up by the author and the 'on-line' Graph Traverser to solve a particular Eight-puzzle configuration. The nodes correspond to configurations and are labelled with the value assigned to them by the program, and with an integer indicating the order in which the developments occurred. The final solution path involved 26 basic moves, compared with a minimum possible of 22. In a parallel attempt on this configuration *not* making use of the program the author required 36 moves to solve it. Indeed, although no serious experimentation has been attempted, the author usually seems to benefit from the program's assistance, although whether the converse is true is less clear!

DISCUSSION

This example of a collaborative use of the Graph Traverser is of no great importance in itself, but it does serve to introduce some further ideas. Dr Donald Michie has pointed out to me that Travelling Salesman problems could be solved by an analogous collaborative approach. The user would be presented with a tour to which he would suggest modifications, and the program would next present either a tour generated by one of the suggested modifications, or a tour generated earlier in the search, but not then judged sufficiently attractive to be followed up.

More generally, a version of the program might be prepared in which the function of both the develop procedure *and* the evaluate procedure were taken over by an on-line user. Since all other aspects of the program's behaviour can be controlled by setting parameters, which even in the current version of the program can be done on-line, such a program would be capable of being applied, *without additional programming*, to a wide range of problems certainly including both of the puzzles mentioned and the Travelling Salesman. However, although this would certainly achieve generality, so much of the work load would have been transferred to the user and progress would be so slow, that the arrangement is unattractive.

What facilities would an on-line user of the Graph Traverser ask for?

Certainly he will not wish to do any preparatory programming. He will wish to enter the program, to specify to it the task, to give it such guidance as he can, and then either to withdraw entirely or to remain on hand to answer questions which the program may ask. He will often wish to monitor the program's progress and to change the problem specification, or to give additional guidance where appropriate. In Graph Traverser terms this means that the user must at least be able initially to specify the operator set, that is the develop procedure, and the evaluation function, and then to change his specification at will. Some kind of on-line problem-oriented programming language, however primitive, seems essential. This could be provided within Algol, but neither easily nor efficiently. It seems obvious that such a version of the Graph Traverser should be wholly written in a language designed for on-line usage. POP-1, created by R. J. Popplestone and described elsewhere in this volume, is attractive given the ease with which it permits the construction of a library of sub-routines.

Other papers already mentioned have suggested ways in which the Graph Traverser might itself improve its search apparatus, usually by some form of parameter optimisation, and thus be a more 'intelligent' problem-solver. D. Vigor, in his stimulating paper included in this volume, emphasises another aspect of intelligence by stressing the importance not of the way in which a program *attacks* a problem (or general game) situation, but of the way in which the program *represents* the situation. He suggests that very often this is the most important step in the solution process, and outlines a method for seeking the most efficient problem representation. I mention this because it indicates an aspect of problem-solving which, although unrecognised by the present Graph Traverser, must surely be important in future work.

ACKNOWLEDGEMENTS

The research described in this paper is sponsored by the Science Research Council. I am greatly indebted to Dr Donald Michie for many discussions on the topics covered by this paper and, in particular, for his original suggestion that the Graph Traverser might be applicable to the Travelling Salesman in the manner described.

REFERENCES

Croes, G. A. (1958). A method for solving Traveling Salesman problems. *Operations Research*, **6**, 791-812.

Doran, J. E. (1967). An approach to automatic problem-solving. *Machine Intelligence* 1, pp. 105-123. Collins, N. L., and Michie, D. (eds.). Edinburgh: Oliver & Boyd.

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

Held, M., & Karp, R. M. (1962). A dynamic programming approach to sequencing problems. *J. Soc. Ind. Appl. Math.* **10**, No. 1, 196-210.

Lin, S. (1965). Computer solutions of the Travelling Salesman problem. *Bell System Tech. J.*, **44**, 2245-2269.

Michie, D. (1967). Strategy-building with the Graph Traverser. *Machine Intelligence* 1, pp. 135-152. Collins, N. L., and Michie, D. (eds.). Edinburgh: Oliver and Boyd.

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

Roe, G. M. (1966). Three fast, suboptimal procedures for the Traveling Salesman problem. *General Electric* report No. 66-C-051.

Schofield, P. (1967). Complete solution of the Eight-puzzle. *Machine Intelligence* 1, pp. 125-133. Collins, N. L., and Michie, D. (eds.). Edinburgh: Oliver and Boyd.