TRAVERSER: A HEURISTIC PROBLEM SOLVER

USER DOCUMENTATION & PROGRAM LISTINGS

BRIAN AUSTIN TATE MARCH 1972

Extracts from "THE DEVELOPMENT AND USE OF A HEURISTIC PROBLEM SOLVER".
A project report from third year studies for the B.A. degree in the
Computer Studies Department at the University of Lancaster.

APPENDIX A - USER DOCUMENTATION

TRAVERSER DOCUMENTATION	AUSTIN TATE	20 JAN 1972
A POP-2 HEURISTIC SEARCH PROGRAM	LANCASTER UNIV.	

1. INTRODUCTION

1.1 Terminology

Keywords are in Capital Letters

The GRAPH TRAVERSER program provides a framework for the heuristic growth of search trees. It is applicable to any problem in which it is possible to reach a required problem state by a finite number of applications of any of a set of OPERATORS (which implicitly specify the GRAPH of the problem). Each operator given to the Graph Traverser will be applicable to some of the states in the problem graph and in such cases will produce a new state, otherwise a standard fail state will be their result. A solution to a problem will involve finding a sequence of operators which will transform one problem state to another required state. This is directed by an EVALUATION FUNCTION which will assign to each state a numerical value, which is an estimate of its distance from a GOAL state. Such an evaluated state will be added to the PARTIAL SEARCH TREE together with information about the operator used to generate the state and the PARENT of the state as a NODE of the graph.

At the beginning of a search only the initial state is explicitly known to the program and is capable of being evaluated and DEVELOPED. Development proceeds by applying in succession appropriate operators until a state is produced that is estimated to be closer to a goal state, or until the value assigned originally to a state is eroded sufficiently by LOCAL SMOOTHING (a constant re-evaluating of a states "worth" in the light of the operators applied to it and their results) to make another node on the partial search tree look more promising for development. A node is labeled PARTIALLY DEVELOPED if some but not all operators have been applied to it. Whenever a new state is found it is evaluated and added to the partial search tree and will be a NEIGHBOURING NODE of the node (its parent) from which by application of an operator (an ARC of the graph) it was produced. At each stage a check is made to avoid adding to the partial search tree any node whose state is already present in the tree. After all appropriate operators have been applied to a node it is labeled FULLY DEVELOPED, and will then not be chosen again for the application of any operator. The Graph Traverser will always develop the lowest valued node (closer to goal) on the partial search tree at each stage, 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 solution path is retraced from the lowest valued developable node to the ROOT of the tree. Then starting at the root a calculated number (based on a parameter and the depth of the best node in the tree) of nodes on this path are added to the solution path being constructed, and that part of the partial search tree is erased, preserving only the part which is dependant on the last node to be added to the path. If it so happens that the most promising node is not very deep in the search tree then the tree is CURTAILED (the least promising nodes on the tree which have not had any operators applied to them are erased). After such PRUNING the 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. However if a goal state is located, a path across the graph between the initial state and the goal state has been found and the nodes on this path are left as a result of the Traverse.

1.2 Traverser Package

The TRAVERSER Package provides a wide range of facilities for conducting heuristically controlled searches over a problem graph in many modes. These facilities are extended by the TRAVERSER EXTRA Package. The terminology of this document and the names of the package functions have been chosen to be in line with other Graph Traverser literature (Machine Intelligence series of volumes). Section 1.1 is based on a description of GT4 (Michie and Ross MI 5).

2. DATA STRUCTURES

2.1 Nodes

A Node of the Graph is held as a POP-2 record of 5 components with specification [0 0 0 12 12]. Dataword is "GENODE".

a) stateof — The representation of the problem state(simple or complex).
b) parentof — The parent node, either another node or "undef" if the node is at the root of a partial search tree.
c) valueof — The value of the state as given by the evaluation function and possibly modified by Local Smoothing.
d) usage — The number (an integer >=1 and =<NUMBEROP+1) of the last operator applied to this node +1.
e) opused — The number of the operator which was applied to the parent to reach this state.
f) consnode and destnode are the constructor and destructor for nodes.

2.2 Trees

The Search Tree represents the section of the problem graph being dealt with by the program and is held as 3 parts:-

The part of the tree still developable is an ordered collection of nodes held as a list, the current best node is at the head of the list (termed TREE).

The nodes of the tree which have been fully developed are held in no particular order as a list (termed DEADTREE).

The nodes which have been chosen during pruning to be on the path to the goal are held as a list with the initial node of the search at its head (termed PATH).

3. HOW TO USE THE PACKAGE

The package can be compiled by typing on the console:-

```
COMPILE(LIBRARY([TRAVERSER]));
```

The manner in which the package can then be used is indicated in the following sections.

3.1 Operators

The elements of an operator set must be functions which map from one problem state to another. If an operator fails to produce a new state when applied it should return "undef" as a result (the standard fail state). The operators applicable to a problem are handed over to the Traverser in an array via the identifier OPLIST. NUMBEROP must be set to the total number of operators in the array.

To give a simple example, if states are integers>=0 and the operators are to add one to the state or minus one from a state this could be set up thus:-

```
newarray([1 2],lambda x; undef end) -> OPLIST; 2 -> NUMBEROP;
lambda x; x+1 end -> OPLIST(1);
lambda x; if x>0 then x-1 else undef close end -> OPLIST(2);
```

The size of the operator set may be large (>4000) but it is suggested that most potentially useful operators are towards the beginning of the array.

3.2 Operator Selection

Each time an operator is to be applied to a node state (each iteration of the function DEVELOP) the node being developed is handed over to the function SELECTOP which is expected to return as its result the index number of the next operator to be applied to the stateof the node. Index numbers range from 1 to NUMBEROP. There is thus the opportunity of redefining function SELECTOP so that the state of the node can be checked and the index number incremented over operators which may not be useful for that state, or of taking into account global variables on the state of the search. The result of SELECTOP can at maximum be NUMBEROP but even if the function decides no operators are applicable to a state, substantial savings could still occur with large operator sets. If the operator with index number NUMBEROP is applied to a node, the node is marked fully developed and will not be chosen to DEVELOP again. Thus no node with USAGE>NUMBEROP will be handed over to SELECTOP. If the operator whose index number is given by SELECTOP when applied to the node fails to give a new state, DEVELOP will be re-iterated, and so on until either the node becomes fully developed or produces a new state.

SELECTOP by default is set to the USAGE selector function of the node. Thus at each development the next operator in OPLIST not already applied to that node is tried until USAGE(node)>NUMBEROP. If SELECTOP is altered by the user he may reset it by executing:- USAGE -> SELECTOP; Note that when a new node is added to the tree its usage is set to 1, thus the default setting for SELECTOP will apply all operators to a state if it is called repeatedly.

3.3 State Selection

For purposes of comparison and ordering of states within the Traverser the user must supply two functions.

One is used to test for equality between states. The identifier EQSTATES is used to pass this function to the Traverser. So for +ve integer states
NONOP = -> EQSTATES; would be sufficient. EQSTATES is used by NOTADD to see if each state produced by a TRAVERSE should be added to the tree. NOTADD may be altered by the user if special checks on a state are required or if it is known that say, all states produced will be unique and legal (see sec. 5.1.8).

The second is a function to estimate whether one state is "better" than another, it also embodies the goal recognizer. This function is handed over to the Traverser via the identifier EVALUATE. This evaluation function must (to simplify - see below) take as arguments two state representations and produce a number>=0 as a result. If the first argument is equivalent to the second the result should be 0, otherwise a +ve numerical result should be produced which gives a measure of the closeness of the two parameters. Lower values should be associated with states which are closer.

Suppose we wish to use the Graph Traverser to take an integer>=0 and transform it to another integer using the operators in section 3.1. It would be sufficient for the evaluation function to provide the difference between two integers thus:-
lambda state1 state2; abs(state1-state2) end -> EVALUATE;
for further details of evaluation functions see section 4.

N.B. On entry to the Graph Traverser via the entry function TRAVERSE, the actual evaluation function used in the search is one produced by partially applying the function EVALUATE to the goal parameter handed over to TRAVERSE. I.e., EVALUATE(%GOAL%) -> EVAL; In TRAVERSE the GOAL parameter is not used for anything else, it is thus possible to have no definite goal and treat the 2nd parameter as a dummy (or anything else), in such a case the evaluation function can be thought of as a cost function producing a numerical value of

the "cost" of the state rather than the promise of the state in comparison with another to which it is hoped the first can be transformed. However if a bi-directional search is being carried out (see section 3.6) the evaluation function used is the two parameter EVALUATE function in order to re-direct the search to the "closest" states on opposite trees.

3.4 Search Parameters

If the functions called for above are inserted the Graph Traverser can be applied to a problem using the entry functions (see sections 3.5 and 3.6), but there are various parameters which may first be altered to control the mode of search performed. All these parameters have default settings as indicated below in brackets. These parameter settings were chosen for 8-puzzle solutions which represent a moderately complex problem solving situation.

3.4.1 FULLDEV (false) The switch for full development or partial development. If the switch is set false development of a node proceeds by applying operators to the state of a node until a "valid" new state (not a repeated state or undef) is produced. This is added to the tree and the parent node checked to see if any further operators can be applied to it, before choosing a new node for development (in this mode Local Smoothing is applicable). If the switch is set true development proceeds by applying all operators which have index numbers greater than the value produced by the operator selection function (SELECTOP) to the node stateof, adding all "valid" new states to the tree, and then marking the node as fully developed (in this mode Local Smoothing is not applicable).

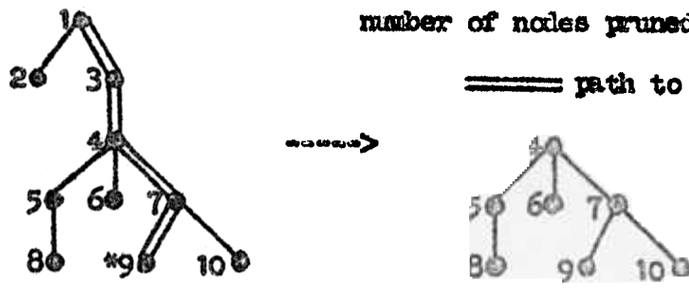
3.4.2 MAXSECS (60) This is the number of seconds to be spent on a search before giving up if a goal has not been found. This specifies the default resignation criterion for the search as given by the LIMIT function (see section 5.1.7). If the Traverser does use up this time "RESIGN" is printed (see sections 3.5 and 3.6).

3.4.3 MAXTREE (25) MAXTREE, MAXPRUNE, MAXCURTAIL and CURTPART together control the pruning of the search tree. MAXTREE is the maximum number of nodes to be added to the partial search tree before an attempt to prune is made. If MAXTREE is made sufficiently large no pruning will occur.

3.4.4 MAXPRUNE (6) This parameter is used during pruning to decide the depth to which pruning of the tree should be taken. The number of nodes along the path to the current best node, to be pruned off is determined by calculating:-

$$\text{intof}\left(\frac{\text{length of path to current best node} * \text{MAXPRUNE}}{\text{MAXTREE}}\right)$$

thus MAXPRUNE will be dependant on the relation between the number of descendants likely to be produced for each node chosen for development and thus on the type of search (partial or full development). So if MAXTREE=10 and MAXPRUNE=5 then the tree below would be pruned thus (9 is current best node)



3.4.5 MAXCURTAIL (2) If the figure calculated as above to give the number of nodes on the path to the current best node to be pruned from the tree is small or zero, few or no nodes will be removed from the tree during normal pruning. In such a case it may be desirable to get rid of nodes of high value in the tree (not very promising nodes). This is done by the CURTAIL function. The number calculated from MAXPRUNE is compared with MAXCURTAIL and if it is less than or equal to MAXCURTAIL the function CURTAIL is entered.

3.4.6 CURTPART (0.4) This is the proportion of the bottom of the TREE list which is checked for curtailing. If a node has USAGE=1 (i.e., has had no operators applied to it) and is in the lower CURTPART of the TREE it is removed from the TREE by CURTAIL.

3.4.7 W (0.1) This is the control parameter for the Local Smoothing scheme. If this is set to 0 there is no local smoothing effect. Local smoothing only applies in partial development mode. It is a process whereby the value of a node can be changed in the light of the evaluated values of the nodes generated from it. Thus allowing nodes given a low value by the evaluation function and then found to be consistantly producing nodes of only high value to be put aside in favour of others or allowing high valued nodes to be upgraded in a similar way. If local smoothing operates, each time a node is generated from a parent node the VALUEOF the parent node is altered as follows:-

$$\frac{\text{present VALUEOF node} + W * \text{VALUEOF generated node}}{1 + W}$$

3.4.8 SELECTOP (usage) See section 3.2.

3.5 Single Tree Searches

3.5.1 Traverse

The entry function TRAVERSE takes as parameters an initial state in any representation and a goal representation (in a form suitable for partial application to the function EVALUATE - the user has the choice, as a goal to TRAVERSE need not be a state representation). TRAVERSE produces as a result a list containing in sequence all NODES on the path between the initial state and the state evaluated with a zero value if this is found. Alternative results (when GROWTREE (see section 5.1.2) has not been assigned to - see section 3.5.2) are:-

a) In the event of the resignation criterion (specified by the function LIMIT - see section 5.1.7) being reached before a goal is found RESIGN is printed and the result is a list of all nodes on the path between the initial state and the current best state.

b) If all applicable operators are applied to all available states in the partial search tree without producing any new state UNSOLVE is printed and the result of TRAVERSE is "nil". This could occur if the problem is unsolvable in the representation given or if pruning is too extensive.

To find information about the search after it is terminated we could use the primitive doublets used to access components of the node (see section 2) and the MAPLIST and APPLIST facilities of POP-2. For this and BITRAVERSE (the other entry function to the Graph Traverser facilities) the date and time is printed at the beginning and end of the search.

3.5.2 Traverse for a Best State

A function to enable searches for a "best" state to be made is available on the Auxiliary Package obtainable, so long as the Traverser Package has already been compiled, by typing on the console:-

```
COMPILE(LIBRARY([TRAVERSER EXTRA]));
```

The TRAVERSE entry function is used for such a search, but GROWTREE must be set to the BESTGROW function available on the auxiliary package. This can be done by typing:- BESTGROW -> GROWTREE;

The BESTGROW function during operation holds a pointer to the best node found as evaluated by the evaluation function (note this may not be the current best node), but continues searching until a node of no cost (cost=0) is found or until the resignation criterion is reached. This is thus a means of searching when no definite goal may be found or when the resignation criterion is expected to be passed. On a RESIGN or UNSOLVE the path given as a result of the Traverse will still give the route to the best node found during a search. In this case RESIGN will indicate a normal termination, no word printed will indicate a zero valued state has been found. Pruning is not applicable to searches for a "best" state, and it should be noted that MAXTREE in this case is not used.

When using BESTGROW in this manner PRUNE MAKEROOT CURTAIL BITRAVERSE PATRNODE and BIGROWTREE may be Cancelled.

3.6 Double-tree Searches

Bitraverse

This entry function is available on the auxiliary package obtainable, provided the Traverser package has already been compiled, by typing on the console:-

```
COMPILE(LIBRARY([TRAVERSER EXTRA]));
```

BITRAVERSE allows a heuristic search to proceed in two directions. Two partial search trees are grown, one with initial node the initial state representation, and the other with initial node the goal representation which must be in the form of another state representation for bi-directional tree searches. The evaluation function used must take as its parameters two state representations and give a measure of the "distance" between them, 0 for equivalent states. It is handed over via the identifier EVALUATE. The search continues by developing nodes on both trees, using as its internal evaluation function the EVALUATE function with the second parameter the "best" state on the opposite tree at each development, until a node with the same state is put on both trees.

At each development a switch is set to indicate which tree is being developed. This is held in the global variable DIRECTION. If DIRECTION=1 the forward tree is being developed, otherwise the reverse tree is being developed and DIRECTION=0. All user defined search functions may take account of this. Thus if for instance there are different operators for the forward and reverse trees (and a further complication there are 4 forward operators and 2 reverse ones), then we could set up OPLIST as follows using the POP-2 facility of treating arrays and functions in a similar manner (FOPS holds the forward operators, ROPS the reverse ones).

```
4 -> NUMBEROP;
lambda i;  if DIRECTION then FOPS(i)
           elseif i=<2 then ROPS(i)
           else lambda i; undef end close
end -> OPLIST;
```

SELECTOP (the operator selection function) can be similarily defined, as for the above case it may be wise to do so. However if it is wished all user defined functions may be as for a normal TRAVERSE, not taking into account the DIRECTION of a search. STILIADD before each entry to DEVELOP is zeroed so that if this is set on exit from DEVELOP a new node has been added to a tree and the corresponding tree count can be incremented (FGNTREE or RGNTREE) and a test for a repeat on the opposite tree made.

In BITRAVERSE only partial development is available (only one or no nodes are added to the tree at each development). Also Pruning is not available as a standard feature - but could be implemented by the user using the BIGROWTREE function (see section 6.1.2). Once again MAXTREE is not used. BITRAVERSE takes as parameters two state representations and tries to produce a list of the nodes on the path between the two states. On RESIGN or UNSOLVE result is "nil". OPUSED of the nodes will not indicate the difference between forward and reverse operators.

When using BITRAVERSE in this manner TRAVERSE GROWTREE PRUNE MAKEROOT CURTAIL and BESTGROW may be Cancelled.

3.7 Recursive Calls of Traverser

3.7.1 Problem Reduction

Consider the case where problem states are algebraic equations to be integrated. Complex equations could be reduced to a combination of simpler integratable equations e.g."the integral of a sum is the sum of the integrals" would be a useful rule to be able to use for this task. An operator when applied to a suitable state may, for such a problem, split up the equation sums and try to integrate each sub-equation separatly. Traverser can be called recursively to help solve this sort of problem. The method is outlined in section 3.7.2. There are many different ways such sub-problems can be manipulated within Traverser. For instance it would be possible to have a "top-level" Traverser application to break a state into its simplest sub-states by using a set of reduction operators, and then when a simplest sub-state is found it could be operated on by Traverser in the normal way using another set of operators. Or both types of operators could co-exist in one system. The sub-problem solution methods using recursive calls may be useful in finding applicable operators in SELECTOP and in a whole variety of special problem solving methods.

3.7.2 Recursive Call Method

To use Traverser (or BiTraverser) within an outer call of itself, it is necessary to create new instances of several variables global to the search to be made, but not at the same program level as any other active Traverser call (i.e., if Traverser is to be called from within an operator of an outer search, the variables should be declared local to the operator). The variables which must be redeclared in this manner are of 3 types:-

a) the variables used to hold information necessary to the searching mechanism. These must be re-declared for inner calls or the outer search will be upset. These are for
   TRAVERSE - PATH STILIADD EVAL INTIME
   BITRAVERSE - STILIADD EVAL INTIME, FULLDEV (if fulldev=1 elsewhere)

b) the variables which contain information for the user for his use during or after a search. If any of this information is needed for an outer search they must be re-declared. These are for
   TRAVERSE - TREE DEADTREE GNTREES OPNUM NEW BEST
   BITRAVERSE - TREE DEADTREE RTREE RDEADTREE DIRECTION GNTREES OK OPNUM NEW BEST

c) the search parameter and guidance variables. If these are not re-declared the search will use the last global instance declared of a variable. However they may be re-declared and reset if other values are required for the search to be made. These are for
TRAVERSE - FULLDEV MAXSECS MAXTREE MAXPRUNE MAXCURTAIL CURTBART W SELECTOP OPLIST NUMBEROP EVALUATE EQSTATES LIMIT NOTADD
BITRAVERSE - MAXSECS SELECTOP OPLIST NUMBEROP EVALUATE EQSTATES W LIMIT NOTADD

Once these re-declarations are made, Traverser may be called within the scope of the variables. Information about operators applied during sub-problem solutions may be extracted and held globally, or using the Pruning facilities of Traverser added directly to the PATH of the main outer search.

4. EVALUATION FUNCTIONS

The Evaluation Function is crucial for the success of heuristic searches. This function should pick on any particular features of the state representation in order to estimate the difference between two states or a state and a goal.

4.1 8-Puzzle Evaluation Function

As an example consider the 8-puzzle for which a full application package for the Graph Traverser in various modes is given later (see section 8). A puzzle has nine cells all of which hold a block except one which is empty. The cells are numbered thus

1	2	3
4	5	6
7	8	9

a particular configuration is held as a 9 element POP-2 strip with the empty block=0.

So

1	4	7
8		6
3	2	5

would be represented by '147806325'. The blocks of the puzzle are re-arranged to form a specified configuration.

In order to construct an evaluation function for searches involving the 8-puzzle we must look for meaningful features of a state representation. Such a feature may be the number of blocks not in their correct cell as represented by a goal state, which could be preset but we will allow to be any other state (A_i - where i is the block number). Thus a check must be made for each block of a configuration to see if the same block is in the same cell of the goal being compared. The distance (in terms of shortest number of slides if no other blocks interfered) a block is away from its "goal" cell may also be important (B_i). We may wish to ignore the space (i=0) from these considerations. Therefore we want to find a suitable combination of ΣA_i and ΣB_i for i=1 to 8 to use as an evaluation function. This must be done by trial and error or using evaluation function parameter optimisation methods with the Graph Traverser itself (see Michie and Ross MI 5). A useful combination for the 8-puzzle is $5\Sigma_1^8 A_i + 7\Sigma_1^8 B_i$. So the evaluation function could be constructed in POP-2 as follows:-

```
lambda config goal; vars heurval i dis blockval;
   9 -> i;  0->heurval;
  L: subscr(i,config) -> blockval; comment 'find the block in the i th cell';
     if blockval then DISTANCE(i,WHEREIS(blockval,goal)) -> dis;
             comment 'if block≠0 then find distance between where
                      block is and where it should be';
             if dis then heurval+5+7*(dis-3) -> heurval close;
     close; i-1 -> i; if i then goto L close;
   heurval
end -> EVALUATE;
```

The auxiliary functions, DISTANCE and WHEREIS, and the rest of the 8-Puzzle application package are described in section 8.

4.2 Searches with a Preference for Short Paths

As Pohl (MI 5) the evaluation function may not just purely give a value for a particular state but may take into account its depth in the search tree. The search thus being guided by a combination of the heuristic value as given by an evaluation function say f, and the depth of the state generated in the tree, say d. Then for $0 =< q < 1$ the search is guided by $(1-q)d + q*f$ where q is a variable parameter.

A state could be represented as say, a pair, whose front is the normal state representation and whose back is 1 plus the integer in the back of the state it was generated from (where the initial state of a search is a pair whose back is set to 0). Then in EVALUATE this could be easily dealt with by letting the state back represent d. The operators of such a system would have to produce a pair, incrementing the back of a state pair to produce the new state back.

4.3 Evaluating Features

As mentioned in section 4.1 the evaluation function should pick on features of a state description to estimate the distance a state is from a goal. The idea of section 4.2 is to add the depth of the search as another feature for evaluation. Obviously a state description could be a Data Structure representing any feature thought to be relevant and may in fact just contain a number of such features and not an explicit problem state. Operators in this case would have to apply to the feature descriptions altering them into new descriptions. Such a state description is called an "IMAGE". This representation may form the basis of problem solving systems which can sensibly choose operators according to the features of a state which they alter and in the light of those problem features not like the corresponding feature in a goal. This is the "differences" evaluation basis of GPS (Ernst and Newell 1969). It may be useful to realize that several variables which contain information about a particular state are global to the EVALUATE function when it is called (see notes 5.2.5 and 5.2.6).

5. TRAVERSER

5.1 Functions available to user

5.1.1 TRAVERSE ε state representation, goal representation => path;
This function is an entry function for a single goal directed heuristic search as explained previously in section 3.5.

5.1.2 GROWTREE ε tree, deadtree => deadtree, tree;
The tree held in TREE, DEADTREE and global PATH is grown using EVAL as its single parameter distance from goal estimator, under the constraints of the search parameters. On exit global PATH is left altered and the two results hold the rest of the grown tree.

5.1.3 INSERT ε node, tree => tree;
Inserts a node into the ordered collection of nodes on the tree according to its VALUEOF. ONTREES is not altered by INSERT.

5.1.4 HADBEFOR ε state, tree => truth value;
Produces true if the state is a repeat of the STATEOF any node on the tree, otherwise produces false.

5.1.5 MAKEROOT ε rootnode,tree,deadtree => deadtree,tree;
Makes rootnode the new root of the tree represented by tree and deadtree, making PARENTOF(rootnode)="undef" and deleting all nodes in the tree which cannot be retraced to the new root.

5.1.6 CURTAIL ε tree => tree;
Removes all nodes in the lower CURTPART of the list TREE with USAGE=1.

5.1.7 LIMIT ε () => truthvalue;
Returns true if the resignation criterion for search is reached.

5.1.8 NOTADD ε state => truthvalue;
Returns true if state should not be added to the tree as a new node.

5.1.9 The Functions of the TIMEPACK library package:-
DATETIME ε () => strip; SECSINCE ε integer => integer;
DATETIME produces a strip with the date and time.
SECSINCE produces the number of seconds since an integer representation of the time initialised by SECSINCE(0)

5.2 Useful variables

5.2.1 INTIME the integer rep. of the time an entry function was entered.

5.2.2 TREE,DEADTREE and PATH hold the corresponding parts (see section 2.2) of the search tree on exit from Traverser.

5.2.3 ONTREES initialized on entry to Traverse it holds the number of NODES generated during the search.

5.2.4 NEW holds the result (a state or undef) of the last operator applied in a DEVELOP. On exit from an entry function this holds the last state generated during a search (it will be the junction state of the two trees in a BITRAVERSE).

5.2.5 BEST a global variable which holds a pointer to the node whose STATEOF was used to apply operators to in the last call of DEVELOP.

5.2.6 OPNUM holds the index number of the last operator applied to a state.

5.2.7 STILLADD holds the number of nodes which can still be added to the partial search tree before pruning is attempted. MAXTREE-STILLADD is the number of nodes on a partial search tree.

5.2.8 other global variables CONSNODE DESTNODE STATEOF PARENTOF VALUEOF USAGE OPUSED W MAXTREE EVAL MAXSECS EQSTATES EVALUATE OPLIST DEVELOP PRUNE NUMBEROP MCGIVE MAXPRUNE FULLDEV FINDPATH SELECTOP CURTPART

N.B. The Traverser package occupies 1.6K words when compiled on an ICL 1900.

6. TRAVERSER EXTRA

6.1 Functions available to user

6.1.1 BITRAVERSE ε state representation, state representation => path;
This function is an entry function for a bi-directional heuristic search as explained previously in section 3.6.

6.1.2 BIGROWTREE & tree,deadtree,rtree,rdeadtree =>
rdeadtree,rtree,deadtree,tree;
This function grows the two trees, represented by the parameters, using EVALUATE as an evaluation function to measure the distance between the two trees.

6.1.3 BESTGROW & tree,deadtree -> deadtree,tree;
This function grows the tree, represented by the parameters, and holds a pointer to the best state produced. The best node pointer is added to the head of the tree list on exit from BESTGROW in all cases. BESTGROW may be assigned to GROWTREE for a TRAVERSE search.

6.1.4 FNTRNODE & state, tree => node;
finds the node with the state given by its parameter in the tree, such a node must exist.

6.2 Useful variables

INTIME ONTREES NEW and OPNUM are as for Traverser.

6.2.1 FONTREE holds the number of nodes on the forward growing tree during a BITRAVERSE.

6.2.2 RONTREE is the reverse tree equivalent of FONTREE.

6.2.3 DIRECTION holds 1 if the development of a BITRAVERSE is on the forward tree or 0 for a development on the reverse tree.

6.2.4 OK is set to "true" after a bitraverse if the search terminated normally, otherwise OK is set "false".

6.2.5 RTREE and RDEADTREE holds globally the tree grown in the reverse direction during a BITRAVERSE, TREE and DEADTREE hold the corresponding forward grown tree parts.

N.B. The Traverser Extra package occupies 0.5K words when compiled on ICL 1900.

7. REFERENCES

graph traverser description, pruning	AN APPROACH TO AUTOMATIC PROBLEM-SOLVING DORAN MI 1
evaluation function optimisation, operator selection methods, local smoothing, GT4	EXPERIMENTS WITH THE ADAPTIVE GRAPH TRAVERSER MICHIE and ROSS MI 5
heuristic search algorithms, shortest path searches	FIRST RESULTS ON THE EFFECTS OF ERROR IN HEURISTIC SEARCH POHL MI 5
double tree searches	BI-DIRECTIONAL SEARCH POHL MI 6
problem reduction problem solver	GPS: A CASE STUDY IN GENERALITY AND PROBLEM SOLVING ERNST and NEWELL ACADEMIC PRESS 1969
heuristic and full search methods, problem reduction, state space	PROBLEM SOLVING METHODS IN ARTIFICIAL INTELLIGENCE NILSSON McGRAW-HILL 1971

8. 8-PUZZLE EXAMPLE APPLICATION PACKAGE AND RUN

For details of the 8-puzzle state representation and the way the evaluation function works, see section 4.1. The Graph Traverser will be used to find a sequence of moves which will transform one 8-puzzle configuration to another.

```
COMPILE(LIBRARY([TRAVERSER]));

lambda strip1 strip2; vars i; 9 -> i;
  L:  if subscrc(i,strip1)=subscrc(i,strip2) then i-1 -> i;
          if i then goto L close; true  else false  close
end -> EQSTATES;

function WHEREIS x config; vars i; 9 -> i;
      comment'find the subscrc number of the cell in config, the block
               x is in|;
    L: if subscrc(i,config)=x then i exit; i-1 -> i; goto L
end;

function DISTANCE i j; vars quotj remj quoti remi;
      comment'find the shortest move distance between cells i and j|;
      (i-1)//3 -> quoti -> remi; (j-1)//3 -> quotj -> remj;
      abs(quoti-quotj)+abs(remi-remj)
end;

lambda config goal; vars heurval i dis blockval;
    9 -> i; 0 -> heurval;
  L:  subscrc(i,config) -> blockval;
      if blockval then DISTANCE(i,WHEREIS(blockval,goal)) -> dis;
          if dis then heurval+5+7*(dis*3) -> heurval  close
      close; i-1 -> i; if i then goto L close;
    heurval
end -> EVALUATE;

function NEWCNFIG oldcn movsp; vars nxtcn spacat spacto;
   WHEREIS(0,oldcn) -> spacat; spacat+movsp -> spacto;
   if spacto>9 or spacto<1 or abs(erase((spacat-1)//3)-erase((spacto-1)//3))>1
   then undef exit; comment'check no attempt to move space past the end or
                            physically through more than one space in OLDCN|;
   copy(oldcn) -> nxtcn; subscrc(spacto, nxtcn) -> subscrc(spacat, nxtcn);
   0 -> subscrc(spacto, nxtcn); nxtcn
end;
```

NEWCNFIG is the general operator, it represents an attempt to produce a new state (NXTCN) from a state (OLDCN) by moving the space in OLDCN (in cell SPACAT) by the number of spaces in MOVSP through the strip representation of the configuration.

```
newarray([1 4],lambda x; undef end) -> OPLIST; 4 -> NUMBEROP;
NEWCNFIG(% 1 %) -> OPLIST(1);          Operator to move space right.
NEWCNFIG(% -1 %) -> OPLIST(2);         Operator to move space left.
NEWCNFIG(% 3 %) -> OPLIST(3);          Operator to move space down.
NEWCNFIG(% -3 %) -> OPLIST(4);         Operator to move space up.
```

```
function DOPROBLEM; vars i config goal solution;
   1 -> i; inite(9) -> config;
  ORIGIN:  .itemread -> subscrc(i,config); i+1 -> i;
           if i=<9 then goto ORIGIN close;
   1 -> i; inite(9) -> goal;
  GOALIN:  .itemread -> subscrc(i,goal); i+1 -> i;
           if i=<9 then goto GOALIN close;
   TRAVERSE(config,goal) -> solution;
   applist(solution,lambda x; 1.nl; pr(STATEOF(x));
                    pr(OPUSED(x)); pr(VALUEOF(x)) end)
end;

0 -> W;  comment 'no local smoothing|;

.DOPROBLEM;
1  5  2  8  0  3  4  7  6
1  2  3  4  5  6  7  8  0

 20JAN71 AT 10.53.28

 20JAN71 AT 10.53.33

'152803476|  0  133.00
'152083476|  2  84.000
- - - - - - - - - - etc.
'123456780|  3  0

CANCEL PRUNE CURTAIL MAKEROOT;
COMPILE(LIBRARY( [TRAVERSER EXTRA]));   CANCEL BESTGROW;
```

Partial development for BITRAVERSE, FULLDEV reset on entry to BITRAVERSE Use DOPROBLEM for entry to BITRAVERSE, although VALUEOF may have little meaning when printed. Put BITRAVERSE in place of TRAVERSE in DOPROBLEM. The same operators can be used for both forward and reverse trees in this case.

```
BITRAVERSE -> TRAVERSE;

.DOPROBLEM;
1  5  2  8  0  3  4  7  6
1  2  3  4  5  6  7  8  0

 20JAN72 AT 10.54.05

 20JAN72 AT 10.54.10

'152803476|  0  133.00
'152083476|  2  84.000
- - - - - - - - - - etc.
'123456780|  4  133.00
```

We could at this stage find where the forward and reverse grown trees joined by printing the state held in global variable NEW (only applies if the search terminated "OK").

APPENDIX B - PROGRAM LISTINGS

[TRAVERSER]

```
COMMENT' AUSTIN TATE..TRAVERSER..7 DEC 1971 !;

VARS W MAXTREE EVAL MAXSECS DESTNODE CONSNODE INTIME STATEOF PARENTOF
     EQSTATES EVALUATE OPLIST NUMBEROP SELECTOP MAXCURTA NEW CURTPART
     USAGE OPUSED MCGIVE VALUEOF ONTREES MAXPRUNE PATH FULLDEV OPNUM
     TREE DEADTREE BEST STILLADD;

RECORDFNS("GTNODE",[0 0 0 12 12]) -> OPUSED -> USAGE -> VALUEOF
          -> PARENTOF -> STATEOF -> DESTNODE -> CONSNODE;
60 -> MAXSECS; 25 -> MAXTREE; 6 -> MAXPRUNE; 2 -> MAXCURTA; 0.1 -> W;
USAGE -> SELECTOP; 0 -> FULLDEV; 0.4 -> CURTPART;

APPLY('83000$01HA03PA040$02HA06PA07[H001,
     CONSWORD(21,21,45,33,35,38,53,46,8).VALOF  -> MCGIVE;

FUNCTION DATETIME; '00000000 AT 00000000!.MCGIVE END;

FUNCTION SECSINCE ORSECS; VARS TS NEWSECS; .DATETIME -> TS;
   (SUBSCRC(16,TS)*10+SUBSCRC(17,TS))*60
       +SUBSCRC(19,TS)*10+SUBSCRC(20,TS)-ORSECS -> NEWSECS;
   IF NEWSECS<0 THEN NEWSECS+3600 -> NEWSECS CLOSE; NEWSECS
END;


FUNCTION FINDPATH NODE; VARS PATH; NIL -> PATH;
  L: NODE::PATH -> PATH; PARENTOF(NODE) -> NODE
     IF NODE=UNDEF THEN PATH EXIT;  GOTO L
END;


FUNCTION LIMIT;
   IF SECSINCE(INTIME)>MAXSECS THEN TRUE ELSE FALSE CLOSE
END;


FUNCTION HADBEFOR NODESTAT TREE;
  L: IF TREE.NULL THEN FALSE EXIT;
     IF EQSTATES(NODESTAT,HD(TREE).STATEOF) THEN TRUE EXIT;
     TL(TREE) -> TREE;  GOTO L
END;


FUNCTION NOTADD ST;
   IF ST=UNDEF OR EQSTATES(ST,STATEOF(BEST))
         OR HADBEFOR(ST,TREE) OR HADBEFOR(ST,DEADTREE)
   THEN TRUE ELSE FALSE CLOSE
END;
```

```
FUNCTION CURTAIL TREE; VARS NEWTREE LIMIT HEAD; NIL -> NEWTREE;
   INTOF(LENGTH(TREE)*CURTPART) -> LIMIT;REV(TREE) -> TREE;
  L: IF LIMIT THEN DEST(TREE) -> TREE -> HEAD;
     IF USAGE(HEAD)>1 THEN HEAD::NEWTREE -> NEWTREE
                      ELSE STILLADD+1 -> STILLADD CLOSE;
     LIMIT-1 -> LIMIT; GOTO L CLOSE; REV(TREE)<>NEWTREE
END;


FUNCTION MAKEROOT ROOT TREE DEADTREE;
   FUNCTION PRUNETREE TREE; VARS NEWTREE HEAD LAST PAR;
      NIL -> NEWTREE;
     L1: IF TREE.NULL THEN REV(NEWTREE) EXIT;
         DEST(TREE) -> TREE -> HEAD; HEAD -> LAST;
        L2: PARENTOF(LAST) -> PAR;
            IF PAR=UNDEF THEN
                 IF LAST=ROOT THEN HEAD::NEWTREE -> NEWTREE;
                 ELSE STILLADD+1 -> STILLADD CLOSE; GOTO L1
            CLOSE; PAR -> LAST; GOTO L2
   END; UNDEF -> PARENTOF(ROOT); PRUNETREE(DEADTREE),PRUNETREE(TREE
END;


FUNCTION PRUNE DEADTREE TREE; VARS PPATH OFFP PATHOFF;
   FINDPATH(HD(TREE)) -> PPATH;
   INTOF(LENGTH(PPATH)*MAXPRUNE/MAXTREE) -> OFFP;
   IF OFFP=<MAXCURTA THEN CURTAIL(TREE) -> TREE;
        IF OFFP=0 THEN DEADTREE,TREE EXIT CLOSE; PPATH -> PATHOFF;
  L: OFFP-1 -> OFFP; IF OFFP THEN TL(PPATH) -> PPATH; GOTO L CLOSE;
   HD(TL(PPATH)); NIL -> TL(PPATH);
   PATH<>PATHOFF -> PATH; MAKEROOT(TREE,DEADTREE)
END.
```

```
FUNCTION INSERT X TREE;
   VARS NODEVAL TREE1; NIL -> TREE1; VALUEOF(X) -> NODEVAL;
  L: IF TREE.NULL OR NODEVAL<VALUEOF(HD(TREE)) THEN
           REV(TREE1)<>(X::TREE) EXIT;
     DEST(TREE) -> TREE; ::TREE1 -> TREE1; GOTO L
END;


FUNCTION DEVELOP DEADTREE TREE; VARS STAT HEURVAL;
   DEST(TREE) -> TREE -> BEST; STATEOF(BEST) -> STAT;
  L: SELECTOP(BEST) -> OPNUM; OPNUM+1 -> USAGE(BEST);
     STAT,OPLIST(OPNUM).APPLY -> NEW;
     IF NOTADD(NEW) THEN IF OPNUM<NUMBEROP THEN GOTO L CLOSE;
     ELSE EVAL(NEW) -> HEURVAL; ONTREES+1 -> ONTREES;
          INSERT(CONSNODE(NEW,BEST,HEURVAL,1,OPNUM),TREE) -> TREE;
          STILLADD-1 -> STILLADD CLOSE;
   IF OPNUM=NUMBEROP THEN BEST::DEADTREE,TREE EXIT;
   IF FULLDEV THEN GOTO L CLOSE;
   (VALUEOF(BEST)+W*HEURVAL)/(1+W) -> VALUEOF(BEST);
   DEADTREE,INSERT(BEST,TREE)
END;


FUNCTION GROWTREE TREE DEADTREE; DEADTREE;
  L: IF TREE.NULL THEN PR("UNSOLVE");
     ELSEIF LIMIT() THEN PR("RESIGN")
     ELSEIF VALUEOF(HD(TREE)) THEN
         IF STILLADD=<0 THEN PRUNE(TREE) -> TREE CLOSE;
         DEVELOP(TREE) -> TREE; GOTO L CLOSE; TREE
END;


FUNCTION TRAVERSE STATE GOAL;
   2.NL; PRSTRING(.DATETIME,CUCHAROUT); 2.NL;
   SECSINCE(0) -> INTIME;  EVALUATE(% GOAL %) -> EVAL;
   NIL -> PATH; 1 -> ONTREES; MAXTREE-1 -> STILLADD;
   GROWTREE(CONSNODE(STATE,UNDEF,EVAL(STATE),1,0)::NIL,NIL)
                    -> TREE -> DEADTREE;
   IF TREE.NULL THEN NIL ELSE PATH<>FINDPATH(HD(TREE)) CLOSE;
   2.NL; PRSTRING(.DATETIME,CUCHAROUT); 2.NL;
END;


PRSTRING(°
TRAVERSER READY FOR USE
!,CUCHAROUT)
:
```

```
[TRAVERSER EXTRA]


COMMENT' AUSTIN TATE..TRAVERSER EXTRA..9 DEC 1971 !;
VARS FONTREE RONTREE DIRECTION RTREE RDEADTREE OK;

FUNCTION BESTGROW TREE DEADTREE; VARS VAL PNTRBEST BESTVAL;
   10!9 -> BESTVAL; DEADTREE;
  L: IF TREE.NULL THEN PR("UNSOLVE")
     ELSEIF LIMIT() THEN PR("RESIGN")
     ELSE VALUEOF(HD(TREE)) -> VAL;
          IF VAL<BESTVAL THEN HD(TREE) -> PNTRBEST;
                              VAL -> BESTVAL CLOSE;
          IF VAL THEN DEVELOP(TREE) -> TREE; GOTO L CLOSE CLOSE;
   PNTRBEST::TREE
END;

FUNCTION PNTRNODE STAT TREE;
  L: IF EQSTATES(STAT,STATEOF(HD(TREE)   THEN HD(TREE) EXIT;
     TL(TREE) -> TREE; GOTO L
END;


FUNCTION BIGROWTREE TREE DEADTREE RTREE RDEADTREE;
  L: IF LIMIT() THEN PR("RESIGN") GOTO OUT
     ELSEIF TREE.NULL THEN IF RTREE.NULL THEN PR("UNSOLVE") GOTO OUT
                           ELSE DEADTREE; GOTO RD CLOSE CLOSE;
     IF RTREE.NULL THEN RDEADTREE
     ELSEIF FONTREE>RONTREE THEN TREE; GOTO RD CLOSE; RTREE;

     .HD.STATEOF -> FROZVAL(1,EVAL); 0 -> STILLADD; 1 -> DIRECTION;
     DEVELOP(DEADTREE,TREE) -> TREE -> DEADTREE;
     IF STILLADD THEN IF HADBEFOR(NEW,RTREE)
          OR HADBEFOR(NEW,RDEADTREE) THEN 1 -> OK; GOTO OUT CLOSE;
     FONTREE+1 -> FONTREE CLOSE; GOTO L;

 RD: .HD.STATEOF -> FROZVAL(1,EVAL); 0 -> STILLADD; 0 -> DIRECTION;
     DEVELOP(RDEADTREE,RTREE) -> RTREE -> RDEADTREE;
     IF STILLADD THEN IF HADBEFOR(NEW,TREE) OR HADBEFOR(NEW,DEADTREE)
       THEN 1 -> OK; GOTO OUT CLOSE;
     RONTREE+1 -> RONTREE CLOSE; GOTO L;
 OUT: RDEADTREE,RTREE,DEADTREE,TREE
END;


FUNCTION BITRAVERSE STATE1 STATE2; VARS EV;
   2.NL; PRSTRING(.DATETIME,CUCHAROUT); 2.NL; 2 -> ONTREES;
   EVALUATE(%STATE2%) -> EVAL; 1 -> FONTREE; 1 -> RONTREE;
   SECSINCE(0) -> INTIME; 0 -> FULLDEV; 0 -> OK; EVAL(STATE1) -> EV;
   CONSNODE(STATE1,UNDEF,EV,1,0)::NIL, NIL,
   CONSNODE(STATE2,UNDEF,EV,1,0)::NIL, NIL.BIGROWTREE
              -> TREE -> DEADTREE -> RTREE -> RDEADTREE;
   IF OK THEN FINDPATH(PNTRNODE(NEW,TREE<>DEADTREE))<>
              TL(REV(FINDPATH(PNTRNODE(NEW,RTREE<>RDEADTREE))))
   ELSE NIL CLOSE; 2.NL; PRSTRING(.DATETIME,CUCHAROUT); 2.NL;
END;

PRSTRING('
TRAVERSER EXTRA READY FOR USE
!,CUCHAROUT)
```