A PROGRAM THAT LEARNS TO SOLVE RUBIK'S CUBE

Richard E. Korf

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213

Abstract

This paper describes a program which learns efficient strategies for solving problems such as Rubik's cube and the eight puzzle. It uses a new general problem solving method based on macro-operators. The strategies learned by the program are equal to or superior to strategies used by humans on these problems, in terms of number of moves required for solution.

1. Introduction: A limitation of GPS

This paper describes research aimed at extending the range of problems that can be solved by general problem solving methods. Currently, the most powerful such method is the combination of means-ends analysis and operator subgoaling described by Newell and Simon in [6], referred to here as the GPS (general problem solver) paradigm. GPS utilizes a set of differences or subgoals and a total ordering among them to arrive at a solution by achieving the subgoals one at a time. GPS can effectively solve a wide range of problems, many with a minimum number of moves.

However, there exist problems that cannot be solved by the GPS formalism. The reason is that GPS requires a set of subgoals that can be solved sequentially such that once a subgoal is achieved, it never has to be violated in order to complete the solution of the problem. For some problems, such as Rubik's cube, no such set of subgoals is known. Every known strategy for the cube involves at least temporarily violating previously established subgoals in order to achieve new subgoals. Note that if we select a set of subgoals of the form "decrease the distance to the goal by one," then these subgoals can be solved sequentially. However, the only known way of computing the distance to the goal for an arbitrary state is exhaustive search. Hence, GPS is of little help in solving Rubik's cube.

The class of problems that are outside the domain of GPS is large and of considerable practical importance. For example, one subclass is the collection of NP-hard problems. For these problems, there are no known sets of subgoals that can be solved strictly sequentially. Furthermore, no efficient strategies are known for solving these problems.

However, for some problems beyond the reach of GPS, such as Rubik's cube, efficient solution strategies are known. Another example of such a problem is the well known eight-puzzle. It is interesting to note that while efficient strategies are known for these problems, there are no efficient strategies for finding minimal-move solutions. This paper is concerned with this class of problems. The two questions to be addressed are:

1. What is the structure of these efficient strategies?

2. How can these strategies be learned or acquired?

2. MPS: Macro Problem Solver

This section describes a problem solving program, called the *Macro Problem Solver*, that can solve problems such as Rubik's cube and the eight-puzzle without doing any search. For simplicity, we will consider the eight-puzzle as an example. The problem solver starts with a simple set of ordered subgoals. In the case of the eight-puzzle, each subgoal will be of the form "move the N tile to the correct position," for N between 1 and 8, plus the blank "tile". The operators to be used are not the primitive operators of the problem space but sequences of primitive operators called *macro-operators* or *macros* for short. Each macro has the property that it achieves one of the subgoals of the problem without disturbing any subgoals that have been previously achieved. Note that intermediate states occurring within a macro may violate prior subgoals, but by the end of the macro all such subgoals will have been restored, and the next subgoal achieved as well.

The macros are organized into a two dimensional table, called a *Macro Table*, which is analogous to the difference table of GPS. A macro table for the eight-puzzle is shown in Table 1, while Figure 1 shows the corresponding goal state for the puzzle. A primitive move is represented by the first letter of Right, Left, Up, or Down. Note that this is unambiguous since only one tile, other than the blank, can be moved in each direction. Each column contains the macros necessary to move one tile to the correct position without disturbing previously

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, and monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539. The views and conclusions in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

positioned tiles. The headings of the columns give the solution order or sequence in which the tiles are to be positioned. Note that the first subgoal is to position the blank. The algorithm for selecting the rest of the solution order will be described below. The rows of the table correspond to the current position of the next tile to be positioned.

```
1 2 3
8   4
7 6 5
```

Figure 1: Goal state for the eight-puzzle

The algorithm employed by the macro problem solver works as follows: First the blank is located, its current position is used as a row index into the first column, and the macro at that location is applied. This moves the blank to the center position. Next, the number 1 tile is located, its position is used as a row index into the second column, and the corresponding macro is applied. This moves the 1 tile to its correct position and also leaves the blank in the center.The macros in the third column move the 2 tile into its correct position while leaving the blank and 1 tiles in their proper places and similarly for the 3, 8, 7, and 4 tiles. At this point, tiles 5 and 6 must be in their correct positions or the puzzle cannot be solved. The lower triangular form of the table is due to the fact that as tiles are positioned, there are fewer positions that the remaining tiles can occupy.

A similar macro table has been built for the Rubik's cube, however space limitations prohibit its inclusion here. There are twenty individual movable "cubies" (1x1x1 cubes), divided into twelve edge (two-sided) and eight corner (three-sided) cubies. In addition, each edge cubie can be in one of two orientations and each corner cubie can be in one of three possible orientations. Each subgoal is to place a particular cubie in its correct position and orientation. There are eighteen primitive moves corresponding to 90 degree clockwise, 90 degree counterclockwise, and 180 degree twists, for each of six faces. Each column of the table contains the macros required to move a particular cubie to the correct position and orientation, from each possible position and orientation that the cubie could be in, while leaving the cubies previously solved in their correct positions. The entire table contains 238 macros. The lengths of the macros range from one to sixteen primitive moves.

The algorithm used to select the solution order is the following: First pick a component which is affected by the least number of primitive operators, in other words, a corner tile or an edge cubie. Remove those operators from the set of operators. At each step, pick the component which maximizes the number of primitive operators remaining which do not affect the previous goals. Ties are resolved by choosing components adjacent to those already selected. For Rubik's cube, this results in intermediate stages of solving an edge cubie, a 2x2x2 subcube, a 2x2x3 rectangular box, two 3x3x1 planes, and finally the entire cube.

Table 2 shows that the macro strategy for the eight-puzzle requires about the same number of primitive moves on the average as a human problem solver and that the Rubik's cube macro strategy is more efficient than the strategies used by most people.

STRATEGY	EIGHT-PUZZLE	RUBIK'S CUBE
Optimal (brute force)	22 [7]	$\sim 18^2$
Macro Problem Solver	39	90
Average Human	38 [2]	125^3

Table 2: Average number of primitive moves in solution path generated by different strategies

	B	1	2	3	8	7	4
B							
1	LU						
2	U	RDLU					
3	RU	DLURRDLU	DLUR				
8	L	DRUL	RULLDDRU	RDLULDRRUL			
7	LD	RULDDRUL	DRUULDRDLU	RULDRDLULDRRUL	RULD		
4	R	LDRURDLU	LDRU	RDLLURDRUL	LURRDL	URDLLURDRULD	
5	RD	LURDLDRURDLU	LURDLDRU	LDRULURDDLUR	ULDRURDL	URDLULDRRULD	LURD
6	D	URDLDRUL	ULDDRU	LDRUULDRDLUR	URDL	LDRRUULDRDLLUR	ULDR

Table 1: Macro table for the eight-puzzle

3. Learning Strategies

This section addresses the issue of learning the strategies to be used by the macro problem solver. The problem is one of finding the macros to fill the macro table.

Given an arbitrary sequence of primitive operators (a macro) and a solution order, we define the *invariance* of the macro as follows. The macro is applied to the goal state of the problem and then the number of components which are in their goal position and orientation are counted, until a component is reached in the solution order which is not in its goal state. For example, if the goal state of the eight-puzzle is represented by the vector [B 1 2 3 4 5 6 7 8], the solution order is (B 1 2 3 8 7 4 5 6), and the state resulting from the application of some particular macro to the goal state is [B 1 2 3 6 5 7 4 8], then the invariance of the macro is five, because the first five tiles (including the blank) in the solution order are in their goal positions and the sixth (the 7 tile) is not. The invariance of a macro determines its column in the macro table. The row of a macro is determined by the position and orientation of the component that occupies the position immediately following the invariant components in the solution order. In the above example, the row of the macro would be the one labelled 4 because the 4 tile occupies the sixth position in the solution order, or the 7 position in the puzzle.

The simplest learning scheme is to perform a breadth-first search of the problem space starting with the goal state, and for each macro generated, insert it into the macro table if the corresponding position is empty, until the table is filled. Note that a breadth-first search ensures the shortest possible macro for each position in the table. This is the algorithm employed to generate the eight-puzzle macro table. It was also used to produce a macro table for a 2x2x2 version of Rubik's cube.

However, the combinatorics of the full 3x3x3 Rubik's cube render this technique ineffective. The technique used in this case is a type of bidirectional search. Consider two macros which map two corresponding cubies to the same position and orientation when applied to the goal state. The effect of the inverse of either macro, obtained by replacing each operator by its inverse operator and reversing the order of the operators, would be to map the cubie back to its original position and orientation. Hence, if the inverse of the second macro is appended to the first macro, the result is a macro which leaves invariant the particular cubie in question. If the states resulting from two macros match in (at most) the first N cubies of the solution order, the composition of one with the inverse of the other is a macro with invariance N. Thus, by storing the macros that are generated, and comparing each new macro to the stored ones, the macro table can be generated by searching to only half the depth of the longest macro required.

[2]This estimate is based on the depth in the search tree at which the number of nodes exceeds the number of possible states

[3]Based on a random sample of 10 graduate students

Unfortunately, in addition to requiring a great deal of space, a bidirectional search requires as much time as a unidirectional search if each new state must be compared to each stored state. This is avoided by hashing each macro using the cubies in an initial subsequence of the solution order. If macros are hashed according to the first N cubies in the solution order, then only macros with invariance greater than or equal to N will be found. However in general, as the invariance increases, the length of the corresponding macros also increases. Thus, in a breadth-first bidirectional search, the macros to fill the low invariance columns of the macro table will be found fairly early, and subsequent effort can be focused on macros with greater invariance, allowing a more effective hashing function. The algorithm maintains an invariance threshold, which is the minimum invariance for which the corresponding column in the macro table is not yet completely filled. As the invariance threshold increases, the entire table is rehashed using a hash function which takes advantage of the higher invariance.

This algorithm is sufficient to find all macros up to length eleven for the Rubik's cube, before the available memory is exhausted. This still leaves several slots empty in the macro table. These final macros are found by composing the macros with the greatest invariance. Note that the composition of two macros with invariance N necessarily results in another macro with invariance N or greater. There is some psychological plausibility to this technique in that many human cube solvers use compositions of shorter macros to accomplish the final stages in their solution strategies.

The learning program for the Rubik's cube is written in C and runs under Unix on a VAX-11/780. The time required to completely fill the macro table is about 15 minutes, and the memory required is about 200K words.

4. Related Work

The Strips [4] program was one of the first programs to learn and use macro-operators (MACROPS) in problem solving. However, the robot problem solving domain used by Strips and other programs is one that is amenable to a GPS treatment.

Goldstein [3] wrote a program which automatically constructed triangular difference tables for GPS. The program worked on a variety of tasks, such as Towers of Hanoi and Instant Insanity, for which effective differences are known.

Furst et al [5] have demonstrated an algorithm for learning strategies for permutation puzzles. Both the Rubik's cube and the eight-puzzle can be embedded in larger problem spaces which are permutation groups. The structure of the macro table is from [5]. The macro composition technique described above is also from [5] and is the sole technique they use for learning macros. The running time of their algorithm is of order N^6, where N is the number of individual

components of the puzzle. In the case of Rubik's cube, N is 48, one for each individual movable square face of the cube. Unfortunately, the Furst algorithm is impractical for problems as large as the cube (48^6 ~ 12 billion) and would generate solutions that are inefficient in terms of number of primitive moves (about 250). However, the most significant difference between the approach exhibited here and that of [5] is that the running time of their algorithm is related to the size of the problem, whereas the running time of this method is related to the length of the longest macro needed to solve the problem.

Banerji [1] has made the observation that GPS fails to solve problems such as Rubik's cube and the fifteen puzzle, and suggests the technique of using macros to bridge the gaps between points of recognizable progress. His work was independent of and occurred at about the same time as this research.

5. Further Work

There are several areas that are currently being investigated further in this work. The first problem to be addressed is the identification of heuristics to reduce the amount of computation required to learn the macros. One approach is to characterize the amount of disorder in the puzzle state and to minimize this disorder in the search for macros. The second problem is reducing the number of moves required for a solution. One approach to this problem is to allow dynamic flexibility in the solution order to allow utilization of the shortest macros at each stage. Another approach is to satisfy more than one subgoal simultaneously. Thirdly, the number of macros in the complete strategy can be reduced by several means. One is to generate subgoals which are preconditions for the effective use of macros from a smaller set. Another is to parameterize macros to reduce the number of similar macros. The effect of different solution orders on these three problems also requires study.

While the programs for the eight-puzzle and Rubik's cube were written separately, efforts are underway to generalize these programs to produce a single learning program that can handle this class of problems, and to characterize the range of problems for which the technique is useful. Finally, any realistic model of problem solving in these domains must admit a compound strategy composed of ordinary difference reduction plus the application of macros to jump between local maxima of the evaluation function.

6. Conclusions

There are three conclusions that can be drawn from this work. One is that GPS is not useful for solving problems such as Rubik's cube. The second is that a generalization of GPS, called the macro problem solver, is capable of solving these problems deterministically without search. Finally, the learning of the macros required by the macro problem solver can be effectively automated. The resulting strategies are comparable to or superior to typical human strategies in terms of number of primitive moves required for solution.

Acknowledgments

I would like to acknowledge many helpful discussions concerning this research with Herbert Simon, Allen Newell, Merrick Furst, Ranan Banerji, and Glenn Iba. In addition, Dave McKeown, Kemal Oflazer, Bruce Lucas, and Greg Korf read and provided helpful comments on drafts of this paper.

References

1. Ranan B. Banerji. GPS and the psychology of the Rubik cubist: A study in reasoning about actions. In *Artificial and Human Intelligence*, A. Elithorn and R. Banerji, Eds., , 1982.

2. K. Anders Ericsson. *Approaches to Descriptions and Analysis of Problem Solving Processes: The 8-puzzle.* Ph.D. Th., University of Stockholm, March 1976.

3. Ernst, George W., and Michael M. Goldstein. "Mechanical discovery of classes of problem-solving strategies." *J.A.C.M. 29*, 1 (January 1982), 1-23.

4. Fikes, Richard E., Peter E. Hart, and Nils J. Nilsson. "Learning and executing generalized robot plans." *Aritificial Intelligence 3* (1972), 251-288.

5. Furst, Merrick, John Hopcroft, and Eugene Luks. Polynomial-time algorithms for permutation groups. 21st Annual Symposium on Foundations of Computer Science, IEEE, Syracuse, New York, October, 1980, pp. 36-41.

6. Newell, A. and H. A. Simon. *Human Problem Solving.* Prentice-Hall, Englewood Cliffs, N.J., 1972.

7. Schofield, P. Complete solution of the eight puzzle. In *Machine Intelligence*, N. L. Collins and D. Michie, Eds., American Elsevier, New York, 1967.