OPERATOR DECOMPOSABILITY: A NEW TYPE OF PROBLEM STRUCTURE

Richard E. Korf

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213[1]

Abstract

This paper describes a structural property of problems that allows an efficient strategy for solving a large number of problem instances to be based on a small amount of knowledge. Specifically, the property of *operator decomposability* is shown to be a sufficient condition for the effective application of the *Macro Problem Solver*, a method that represents its knowledge of a problem by a small number of operator sequences. Roughly, operator decomposability exists in a problem to the extent that the effect of an operator on each component of a state can be expressed as a function of only a subset of the components, independent of the remaining state components.

1. Introduction

Fundamentally, learning is concerned with situations where we are interested in solving many instances of a problem, rather than just one instance. In that case, it may be advantageous to first learn a general strategy for solving any instance of the problem, and then apply it to each problem instance. This allows the computational cost of the learning stage to be amortized over all the problem instances to be solved. Such an approach will only be useful if there is some structure to the collection of problem instances such that the fixed cost of learning a single strategy plus the marginal cost of applying it to each problem instance is less than the cost of solving each instance from scratch. This paper presents one such structural property, that of *operator decomposability*. Operator decomposability is a sufficient condition for the success of macro problem solving, a learning and problem solving method first described in [2].

2. Macro Problem Solving

We begin with a brief description and example of macro problem solving taken from [2]. The reader familiar with that work may skip this section. Macro problem solving is partly based on the work of Sims [4] and Banerji [1]. Complete details of the problem solving and learning programs can be found in [3].

The Macro Problem Solver is a program that can efficiently solve a number of problems, including the Eight Puzzle, Rubik's Cube, and the Towers of Hanoi, without 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 this case, each subgoal will be to move a particular tile to its goal position, including the blank "tile". The operators to be used are not the primitive operators of the problem space but rather 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. Intermediate states occurring within the application of 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 learned automatically by a macro learning program. They are organized into a two-dimensional matrix called a *macro table*. Table 2-1 shows a macro table for the Eight Puzzle, corresponding to the goal state shown in Figure 2-1. A primitive move is represented by the first letter of Right, Left, Up, or Down. This is unambiguous since only one tile, excluding the blank, can be moved in each direction in a given state. Each column contains the macros necessary to move one tile to its correct position from any possible initial position without disturbing previously positioned tiles. The order of the columns gives the *solution order* or the sequence in which the tiles are to be positioned. The rows of the table correspond to the different possible starting positions for the next tile to be placed. Figure 2-1 also gives the names of the different tile positions.

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

Figure 2-1: Eight Puzzle Goal State

The algorithm used by the Macro Problem Solver is as follows: First, the position of the blank is determined, that position is used as a row index into the first column of the macro table, and the macro at that location is applied. This moves the blank to its goal position, which is the center in this case. 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 goal position and also returns the blank to the center. The macros in the third column move the 2 tile into its goal position while leaving the blank and 1 tiles in their proper places. Similarly, the 3, 4, 5, and 6 tiles are positioned by the macros in their corresponding columns. At this point, tiles 7 and 8 must be in their goal positions or else the puzzle cannot be solved. The lower

---

[1] author's current address: Department of Computer Science, Columbia University, New York, N.Y. 10027

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.

triangular form of the table is due to the fact that as tiles are moved to their goal positions, there are fewer positions that the remaining tiles can occupy.

In addition to the Eight Puzzle, this technique has been applied to the Fifteen Puzzle, Rubik's Cube, the Towers of Hanoi problem, and the Think-A-Dot machine. The key feature that makes this method useful is that only a small number of macros are needed in order to efficiently solve a large number of problem instances. In the case of the Eight Puzzle, all 181,440 problem instances can be solved without any search using only 35 macros. Similarly, for Rubik's Cube, all 4×10^{19} problem instances can be solved without search using only 238 macros.

The remainder of this paper presents the structural property of these problems that makes this savings possible. We begin with an abstract representation of the example problems.

3. State-Vector Representation

A state of a problem is specified by giving the values of a vector of state variables. For example, the state variables for the Eight Puzzle are the nine different tiles of the puzzle, including the blank, and the values are the positions occupied by each tile in a particular state. For Rubik's Cube, the variables are the different individual movable pieces, called *cubies*, and the values encode both the positions of the cubies and their orientation. In the case of the Towers of Hanoi, the variables are the disks, and the values are the pegs that the disks are on.

For each problem, a single goal state is specified by assigning particular values to the state variables, called their goal values. Furthermore, we include in the problem space only those states which are *solvable* in the sense that there exists a path from each state to the goal state. A problem instance then becomes a combination of a problem and a particular initial state.

The operators of the problem space are functions which map state vectors into state vectors. Formally, all operators are total functions in that they apply to all states. We adopt the convention that if a state does not satisfy the preconditions of an operator, then the effect of that operator on that state is to leave it unchanged.

4. Macro Table Definition

When we examine the macro table for the Eight Puzzle (Table 2-1), we notice that the first column contains eight entries. There is one macro for each possible position that the blank could occupy (other than its goal position) in the initial state, or one macro for each possible value of the first state variable. Thus, the choice of what macro to apply first depends only on the value of the first state variable. Another way of looking at this is that for a given value of the first state variable, the same macro will map it to its target value regardless of the values of the remaining state variables. In general, this property would not hold for an arbitrary problem. In fact, in the worst case, one would need a different macro in the first column for each different initial state of the problem. In the second column as well, we only need one macro for each possible value of the second state variable (the positions of the 1 tile). Again, this is due to the fact that its application is independent of the values of all succeeding variables in the solution order. Similarly, for the remaining columns of the table, the macros depend only on the previous state variables in the solution order and are independent of the succeeding variables.

More formally, let S be the set of all states in the problem space, let S_i be the set of all states in which the first $i-1$ state variables equal their goal values, and let S_{ij} be the subset of S_i in which the i^{th} state variable has value j. Then, we can define a macro table as follows:

Definition 1: A *macro table* is a set of macros $\mathbf{m}_{ij}$ such that

$$\forall s \in S_{ij},\ \mathbf{m}_{ij}(s) \in S_{i+1}$$

Operator decomposability is the property that allows macro tables to exist. For pedagogical reasons, we first present a special case of operator decomposability called *total decomposability*.

5. Total Decomposability

Given that each state is a vector of the form $(s_1,s_2,\ldots,s_n)$, we define total decomposability as follows:

Definition 2: A vector function $\mathbf{f}$ is *totally decomposable* if there exists a scalar function f such that

$$\forall s \in S,\ \mathbf{f}(s)=\mathbf{f}(s_1,s_2,\ldots,s_n)=(f(s_1),f(s_2),\ldots,f(s_n)).$$

POSITION	TILES 0	1	2	3	4	5	6
0							
1	UL						
2	U	RDLU					
3	UR	DLURRDLU	DLUR				
4	R	LDRURDLU	LDRU	RDLLURDRUL			
5	DR	ULDRURDLDRUL	LURDLDRU	LDRULURDDLUR	LURD		
6	D	URDLDRUL	ULDDRU	URDDLULDRRUL	ULDR	RDLLUURDLDRRUL	
7	DL	RULDDRUL	DRUULDRDLU	RULDRDLULDRRUL	URDLULDR	ULDRURDLLURD	URDL
8	L	DRUL	RULLDDRU	RDLULDRRUL	RULLDR	ULDRRULDLURD	RULD

Table 2-1: Macro Table for the Eight Puzzle

This property can be illustrated by the operators of Rubik's Cube. Recall that the state variables are the individual cubies and the values encode their positions and orientations. Each operator will affect some subset of the cubies or state variables, and leave the remaining state variables unchanged. However, the resulting position and orientation of each cubie as a result of any operator is solely a function of that cubie's position and orientation before the operator was applied, and independent of the positions and orientations of the other cubies.

Since it can easily be shown that the composition of two totally decomposable functions is also totally decomposable, we state the following lemma without proof:

> **Lemma 3:** A macro is totally decomposable if each of the operators in it are totally decomposable.

Next, we define total decomposability as a property of a problem space as opposed to just a function.

> **Definition 4:** A problem is totally decomposable if each of its primitive operators is totally decomposable.

Finally, we present the main result of this section, that total decomposability is a sufficient condition for the existence of a macro table.

> **Theorem 5:** If a problem is totally decomposable, then there exists a macro table for the problem.

We will omit the formal details of the proof [3] and present only its outline. The basic argument is that for each pair i,j such that S_{ij} is not empty, there must be some macro which maps some element of S_{ij} to the goal state since there is a path from every state to the goal. Since the goal state is an element of S_{i+1} for any i, this macro maps an element of S_{ij} to an element of S_{i+1}. Then, the total decomposability property is used to show that this macro maps *any* element of S_{ij} to an element of S_{i+1}, and hence qualifies as $\mathbf{m}_{ij}$.

6. Serial Decomposability

The small number of macros in the macro table is due to the fact that the effect of a macro on a state variable is independent of the *succeeding* variables in the solution order. However, the effect of a macro on a state variable need not be independent of the *preceding* variables in the solution order, since these values are known when the macro is applied. This suggests that a weaker and more general form of operator decomposability would still admit a macro table. This is the case with the Eight Puzzle, the Think-a-Dot problem, and the Towers of Hanoi problem.

Recall that in the Eight Puzzle, the state variables correspond to the different tiles, including the blank. Each of the four operators (Up, Down, Left, and Right) affect exactly two state variables, the tile they move and the blank. While the *effects* on each of these two tiles are totally decomposable, the *preconditions* of the operators are not. Note that while there are no preconditions on any operators for Rubik's Cube, i.e. all operators are always applicable, the Eight Puzzle operators must satisfy the precondition that the blank be adjacent to the tile to be moved and in the direction it is to be moved. Thus, whether or not an operator is applicable to a particular tile variable depends on whether the blank variable has the correct value. In order for an operator to be totally decomposable, the decomposition must hold for both the preconditions and the postconditions of the operator.

The obvious solution to this problem is to pick the blank tile to be first in the solution order. Then, in all succeeding stages the position of the blank will be known and hence the dependence on this variable will not affect the macro table. The net result of this weaker form of operator decomposability is that it places a constraint on the possible solution orders. The constraint is that the state variables must be ordered such that the preconditions and the effects of each operator on each state variable depend only on that variable and preceding state variables in the solution order. If such an ordering exists, we say that the operators exhibit *serial decomposability*. In the case of the Eight Puzzle, the constraint is simply that the blank must occur first in the solution order.

What follows is a more formal treatment of serial decomposability. The presentation exactly parallels that of total decomposability.

Given that a solution order is a permutation of the state variables, and that the numbering of the state variables coincides with the solution order, we define serial decomposability as follows:

> **Definition 6:** A vector function $\mathbf{f}$ is serially decomposable with respect to a solution order if there exists a set of functions $\mathbf{f}_i$ such that
>
> $$\forall s \in S,\ \mathbf{f}(s) = \mathbf{f}(s_1, s_2, \ldots, s_n) = (\mathbf{f}_1(s_1), \mathbf{f}_2(s_1, s_2), \ldots, \mathbf{f}_n(s_1, s_2, \ldots, s_n))$$

As in the case of total decomposability, if all the operators in a macro are serially decomposable, then the macro is serially decomposable. We define a problem to be serially decomposable if there exists some solution order such that all the primitive operators are serially decomposable with respect to this solution order. Finally, we arrive at the main result of this paper:

> **Theorem 7:** If a problem is serially decomposable, then there exists a macro table for the problem.

The proof is entirely analogous to that for the total decomposability theorem. Note that total decomposability is merely a special case of serial decomposability.

While the Eight Puzzle is the simplest example of a serially decomposable problem, the Think-a-Dot problem exhibits a much richer form of serial decomposability that results in a more complex constraint on the solution order. Think-a-Dot is a toy which involves dropping marbles through gated channels and observing the effects on the gates. Figure 6-1 is a schematic diagram of the device. There are three input channels at the top, labelled A, B, and C, into which marbles can be dropped. When a marble is dropped in, it falls through a set of channels governed by eight numbered gates. Each gate has two states, Left and Right. When a marble encounters a gate, it goes left or right depending on the current state of the gate and then flips the gate to the opposite state. A state of the machine is specified by giving the states of each of the gates. The problem is to get from any arbitrary initial state to some goal state, such as all gates pointing Left.

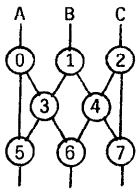

Figure 6-1: Think-a-Dot Machine

The state variables of the problem are the individual gates, and the values are Right and Left. The primitive operators are A, B, and C, corresponding to dropping a marble in each of the input channels. Table 6-1 shows a macro table for the Think-a-Dot problem where the goal state is all gates pointing Left. Note that there are only two possible values for each state variable, one of which is the goal state, and hence only one macro in each column. The last gate in the macro table is gate 6 since once the first seven gates are set, the state of the last gate is determined, due to a situation similar to that of the Eight Puzzle.

	GATES						
	0	1	2	3	4	5	6
Right	A	B	C	AA	CC	AAAA	CCCC
Left							

Table 6-1: Macro Table for Think-a-Dot Machine

Roughly, the effect of an operator on a particular gate can depend on the values of the gates above it. More precisely, the effect of an operator on a particular gate depends only on the values of all of its "ancestors", or those gates from which there exists a directed path to the given gate. Thus, the constraint on the solution order is that the ancestors of any gate must occur prior to that gate in the order. The serial decomposability structure of this problem is directly exhibited by the directed graph structure of the machine, and is based on the effects of the operators rather than their preconditions, since there are no preconditions on the Think-a-Dot operators.

An extreme case of serial decomposability occurs in the Towers of Hanoi problem. Note that in this context the problem is to move all the disks to the goal peg from *any* legal initial state. Table 6-2 shows a macro table for the three-disk Towers of Hanoi problem, where the goal peg is peg C. The state variables are the disks, numbered 1 through 3 in increasing order of size. The values are the different pegs the disks could be on, labelled A, B, and C. There are six primitive moves in the problem space, one corresponding to each possible ordered pair of source peg and destination peg. Since only the top disk on a peg can be moved, this is an unambiguous representation of the operators. The complete set is thus {AB, AC, BA, BC, CA, CB}.

		DISKS		
		1	2	3
PEGS	A	AC	CB AC BC	CA CB AB AC BA BC AC
	B	BC	CA BC AC	CB CA BA BC AB AC BC
	C			

Table 6-2: Macro table for Three Disk Towers of Hanoi Problem

The applicability of each of the operators to each of the disks depends upon the positions of all the smaller disks. In particular, no smaller disk may be on the same peg as the disk to be moved, nor may a smaller disk be on the destination peg. This totally constrains the solution order to be from smallest disk to largest disk. We describe this as a boundary case since it exhibits the maximum amount of dependence possible without violating serial decomposability.

Operator decomposability in a problem is not only a function of the problem, but depends on the particular formulation of the problem in terms of state variables as well. For example, under a dual representation of the Eight Puzzle, where state variables correspond to positions and values correspond to tiles, the operators are not decomposable. The reason is that there is no ordering of the positions such that the effect of each of the operators on each of the positions can be expressed as a function of only the previous positions in the order.

7. Conclusions

Operator decomposability is a newly discovered property of problem spaces that holds for a number of example problems. It is a sufficient condition for the application of macro problem solving and learning, since it allows an efficient solution for a large number of problem instances to be based on a small amount of knowledge, or a small number of macros.

Acknowledgements

I would like to acknowledge many helpful discussions concerning this research with Herbert Simon, Allen Newell, Ranan Banerji, and Jon Bentley. In addition, Walter van Roggen provided helpful criticism on a draft 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., North-Holland, Amsterdam, 1983.

2. Korf, R.E. A program that learns to to solve Rubik's Cube. Proceedings of the National Conference on Artificial Intelligence, Pittsburgh, Pa., August, 1982, pp. 164-167.

3. Korf, R.E. *Learning to Solve Problems by Searching for Macro-Operators.* Ph.D. Th., Department of Computer Science, Carnegie-Mellon University, June 1983.

4. Sims, Charles C. Computational methods in the study of permutation groups. In *Computational Problems in Abstract Algebra*, John Leech, Ed.,Pergamon Press, New York, 1970, pp. 169-183.