Learning Problem Solving Strategies Using Refinement and Macro Generation

H. Altay Güvenir
Computer Engineering and Information Sciences Department, Bilkent University, Ankara, Turkey

George W. Ernst
Computer Engineering and Science Department, Case Western Reserve University, Cleveland, OH 44106, USA

ABSTRACT

In this paper we propose a technique for learning efficient strategies for solving a certain class of problems. The method, RWM, makes use of two separate methods, namely, refinement and macro generation. The former is a method for partitioning a given problem into a sequence of easier subproblems. The latter is for efficiently learning composite moves which are useful in solving the problem. These methods and a system that incorporates them are described in detail. The kind of strategies learned by RWM are based on the GPS problem solving method. Examples of strategies learned for different types of problems are given. RWM has learned good strategies for some problems which are difficult by human standards.

1. Introduction

Search is the basic technique underlying most computer problem solving methods. Exhaustive search methods explore all possible paths to a goal state during the problem solving process. However, such methods are not feasible for problems with combinatorially large state spaces, because there are always practical limits on the amount of time and storage available. For many tasks it is possible to state principles or rules of thumb, so-called heuristics, to help reduce the search. Any such technique used to speed up the search depends on special information about the problem. Heuristics are used in different ways in different problem solving techniques; e.g., test functions in hill-climbing [22], estimate functions in A* [20]. However, Korf showed that for many difficult problems such as Rubik's Cube, no such heuristic functions are of any direct

Artificial Intelligence **44** (1990) 209–243
0004-3702/90/$03.50

use [9]. On the other hand, the heuristics used by the General Problem Solver (GPS) are based on "differences" between problem states [3]. These differences and the way they are used constitute a strategy for solving the problem. *Refinement with macros* (RWM) is a method for essentially learning the heuristic information that GPS needs to know to solve a problem.

The kind of strategy that the system learns is typified by the one it learned for the $2 \times 2 \times 2$ Rubik's Cube, which is difficult by human standards. The goal of this puzzle is to make all eight cubies have the same color on their adjacent facelets.[1] The primitive operators are the 90° counterclockwise rotations of front (F), upper (U) and right (R) halves of the cube. The following strategy was learned by RWM:

(1) Make the two lower left cubies have the same color on each pair of their adjacent facelets.
(2) Make the lower front right cubie and the lower left cubies have the same color on each pair of adjacent facelets.
(3) Make the cubies on the lower half have the same color on each pair of adjacent facelets.
(4) Make upper left front cubie and the cubies on the lower half have the same color on each pair of adjacent facelets.
(5) Make upper right back cubie, upper left front cubie and the cubies on the lower half have the same color on each pair of adjacent facelets.
(6) Make upper right front cubie, upper left back cubie and the rest of the cubies have the same color on the adjacent facelets.

Implicit in this strategy is that the goals are solved in the given order and once a goal is achieved it is not violated later in the solution process. For this reason, RWM also learns good moves for solving each subproblem. A *move* is either a *primitive operator* or a sequence of operators, which is called a *macro*. For example, (UFFFRF) is one of the moves learned for the third stage. A useful property of this macro is that it is safe over the goals of the first two subproblems in the sense that if these goals are satisfied before applying this macro, they will remain satisfied after applying the macro.

This strategy is similar in some respect to those used by humans (see [6], for example). Such strategies divide the problem into a number of subproblems such as getting a portion of the cube in its correct position. In solving these subproblems the goals of the previous subproblems are not violated. This is accomplished by the way that subproblems are solved: each subproblem is solved by a search process which only uses moves that are safe over all of the goals of the previous subproblems. Thus, the order of the subproblems is very important because the moves that are learned for a subproblem will have such safety properties.

[1] The terminology used here is standard in the literature of Rubik's Cube [6].

Another example is the Mod-3 problem, which we will use throughout the paper since it is easier to visualize than Rubik's cube. The Mod-3 Puzzle[2] is played on a 3×3 board. Each square can take any value between 0 and 2. A move consists of playing on a square, which will increment the value of each square that is in the same row or the same column as the square played on. Each increment adds 1 mod 3 to the number. The goal is to have the same value on every square. An initial state has arbitrary values, e.g.,

2 0 1

0 1 2

1 2 0

Although this problem appears to be easy, it is not so obvious how one should go about solving it. The reader is invited to try solving the puzzle from the initial state given above. Usually one starts by making the numbers in the first row equal. However, one soon realizes that it is difficult to make the numbers in the second row equal to the ones in the first row without changing the numbers in the first row. The nature of the difficulty is that lots of squares have to be made equal, and when making a square equal to the others, the relevant operators usually violate the equality of some of the squares which were previously made equal. Also it is not clear in what order the squares should be made equal. An exhaustive search is not reasonable because usually it takes more than 8 operators to solve the problem, and the branching factor is 9.

Our approach is to use a strategy in which each step makes a few squares equal while maintaining the accomplishments of the previous stages. The strategy learned by the RWM method for the Mod-3 Puzzle is shown in Fig. 1.

1. Make $s_{11} = s_{12}$ using moves o_{21} and/or o_{22}.
2. Make $s_{23} = s_{33}$ using moves $(o_{21}o_{22})$ and/or $(o_{31}o_{32})$.
3. Make all the squares in the first row equal using moves o_{23} and $(o_{21}o_{32})$.
4. Make $s_{21} = s_{31}$ using moves $(o_{23}o_{22}o_{31})$ and/or $(o_{33}o_{21}o_{32})$.
5. Make $s_{22} = s_{32}$ using moves $(o_{23}o_{21}o_{32})$ and/or $(o_{33}o_{22}o_{31})$.
6. Make $s_{22} = s_{23}$ using moves o_{12} and/or o_{13}.
7. Make $s_{11} = s_{21}$ using move $(o_{12}o_{13})$.
8. Make all the squares equal using o_{11}.

Fig. 1. The strategy learned for the Mod-3 Puzzle.

[2] This is essentially the "One to Five" puzzle given in [21].

The first stage of the strategy is to get the squares s_{11} and s_{12} equal; s_{ij} denotes the value in row i and column j. The problem solver solves this subproblem by searching for a state in which s_{11} and s_{12} have equal values. During this search only moves o_{21} and o_{22} are used, where o_{ij} denotes the move on square s_{ij}. For example, o_{22} and $o_{21}o_{21}o_{22}$ are two of the possible solutions that the search considers. In the second stage squares s_{23} and s_{33} will be made equal while maintaining what was done in the first stage. The macro moves $(o_{21}o_{22})$ and $(o_{31}o_{32})$ will be used to accomplish this. The macro move $(o_{21}o_{22})$ denotes first playing on square s_{21} and then on square s_{22}. Notice that although playing on s_{21} will change the equality of s_{11} and s_{12}, the goal of the first stage, it will be restored when s_{22} is played on next. Therefore, the macro move $(o_{21}o_{22})$ is "safe" over the goal of the first stage since it does not affect the equality of the squares s_{11} and s_{12}. And, since it affects the equality of s_{23} and s_{33}, the goal of the second stage, it is used for that stage. The moves o_{23} and $(o_{21}o_{32})$ are safe over the goals of the first two stages, and effective in making s_{13} and the other squares of the first row equal. Thus, they are the moves in the third stage of the strategy which makes the squares in the first row the same while maintaining the previous stages. A similar situation holds for the remaining stages. The problem solver continues to use this process one stage at a time until a state satisfying the goal statement of the last stage is found; such a state also satisfies the goal of the whole problem.

1.1. GPS

The General Problem Solver (GPS) implements the problem solving technique called *means-ends analysis*. Means-ends analysis refers to the process of comparing what is given or known to what is desired, and on the basis of this comparison, selecting a "reasonable" thing to do next [5].

GPS is designed to work on state space problems. A state space problem consists of an *initial state*, a set of *goal states* and a set of *operators*. Each operator is a partial function on the set of states. A *solution* to a problem is a sequence of operators which transforms the initial state into a goal state. Each intermediate state produced by one of these operators must be in the domain of the next operator in the sequence.

If the initial state is not a goal state, GPS detects differences between them, and then attempts to reduce the largest of these differences. To do this GPS selects an operator which is relevant to the largest difference and applies it to the initial state. This results in a new state, and the process is repeated by comparing it to the goal state and detecting the differences. If the initial state is not in the domain of an operator, the goal of reducing the largest difference between the initial state and the domain of that operator is created. Operators which are relevant to this difference are used to reduce it. This will produce a state in the domain of the operator which can then be applied.

Information about differences is a problem-dependent parameter to GPS; its purpose is to make the search more efficient. Some of these differences are more difficult to remove than others, and thus they are ordered according to their difficulty. GPS employs the heuristic of removing differences in the order of their difficulty, the most difficult first. In the process of reducing a difference, a previously removed difference must not be reintroduced. Any operator will be relevant to removing some differences but not others. Only the operators which are relevant to a difference are used to reduce it. This use of the difference ordering and operator relevance restricts the number of operators used to remove a difference to some fraction of the total number of operators.

Note that an operator which is relevant to a difference is not guaranteed to remove the difference. It is also possible that the search for reducing a difference may be unsuccessful; in that case backtracking must take place.

RWM is designed to learn the differences, their ordering, and the moves relevant to removing each difference for a given problem. One way to learn such a strategy could be to look for orthogonal groupings of moves such that once a set of differences is satisfied, no move outside the set will modify it. However, for many interesting problems such a method would not result in an efficient strategy. For example, the moves of the Mod-3 Puzzle can be grouped only into two sets, $\{o_{12}, \ldots, o_{33}\}$ and $\{o_{11}\}$. The strategy corresponding to this grouping is: first get $s_{11} = s_{12} = s_{13} = s_{21} = s_{31}$ and $s_{22} = s_{23} = s_{32} = s_{33}$ using the moves $\{o_{12}, \ldots, o_{33}\}$, then get $s_{11} = s_{22}$ using o_{11}. However this strategy is not efficient since the first stage is almost as hard as the Mod-3 Puzzle itself. Therefore, more powerful techniques are needed for learning efficient strategies.

1.2. Learning GPS-based strategies

The aim of our research is to develop methods for learning GPS-based efficient strategies. There were several early attempts to learn differences and difference orderings for GPS. The proposal by Newell, Simon and Shaw in [14] is conceptually interesting, but was never implemented nor is its implementation straightforward. The method by Newell [15] did not take into account the way differences interact with one another, and did not consider using macros. Thus, it cannot learn the kinds of strategies that RWM learns. A nice discussion of these issues is given in [16].

There are two successful methods that also learn GPS-based strategies; Ernst and Goldstein's [4] DGBS discovery system and Korf's [9] learning method for MPS, which is closely related to GPS. In order to learn a strategy, DGBS first finds the basic invariants, which are the properties that are left invariant when applying an operator. Such invariants are found by matching the input state to the output state of an operator. Those basic invariants are then combined to

form high-level invariants (differences). A high-level invariant is implied by the goal statement, in addition to being invariant over some operators. The relationship between the high-level invariants and the operators are stored in a boolean matrix. An element of this matrix is 0 if the operator of its column is invariant over the high-level invariant of its row, 1 otherwise. The final step in learning a strategy is to make permutations and to combine rows and columns to triangularize this matrix. Such a triangular matrix constitutes a strategy for solving the problem using GPS.

As described in the previous section, GPS uses an ordered set of differences, or subgoals, and a set of operators that are relevant to removing a difference and do not reintroduce previously removed differences. A set of subgoals with this property is called *serializable*. On the other hand, there are problems that cannot be solved sequentially by any ordering of subgoals. A good example of this is the Rubik's Cube puzzle. Korf has developed a problem solving method called *macro problem solving* (MPS), and a method for learning strategies based on MPS. Korf used macro operators to overcome the problem of *nonserializable* subgoals. A strategy given to MPS is a *macro table*, which is a table whose column headings are state components (differences in GPS), and whose row headings are the values of state components. The table entries are the macro operators. At any point in solving a problem the macro m_{ij} will be applied if the first $i-1$ state components have their goal values and if the current value of component i is j. That is, the macros in the ith column are used to remove the difference on the state component i. Korf has written programs that generate macro tables for a number of problems. The basic idea is to "work backwards" from a known goal state by applying the inverse of the "primitive" operators. Given the ordering of the state components, this space is searched for a state s that has goal values in its first $i-1$ components and some non-goal value j in its ith component. Then, the sequence of operators on the path from the goal state to s is the inverse of a macro which belongs to row i, column j of a macro table. A macro table is learned by searching from the goal state for all possible values for i and j.

This previous work on mechanical discovery systems is extended by our research. The main extension to Korf's work is the way that RWM learns a good ordering for subproblems. RWM can also learn strategies for multiple goal states. The strategies learned by Korf's method will be compared with the ones learned by RWM later in the discussion. The primary extension to Ernst and Goldstein's work is the use of macro operators in the strategies learned. The strategies learned by all of these methods are similar to ones used by GPS. Other researchers are working on the machine learning of other kinds of strategies and heuristics, e.g., Pearl [20], Amarel [1] and Mitchell [13]. Like Korf's method, RWM uses some of the techniques developed by Sims [23] and Banerji [2]. Minton [12] has developed an explanation-based learning system, called PRODIGY, which learns meta-level concepts for efficient search con-

trol. Laird et al. [11] have developed a learning mechanism, called chunking, for the rule-based general problem solving architecture, SOAR. Chunking acquires rules and macro operators from goal-based experiences. SOAR is designed to solve problems at different levels of abstraction, each of which corresponds to a different problem space. These problem spaces are given to SOAR. The strategies learned by RWM can be given to SOAR as abstract problem spaces.

There are four main sections to the paper. The next section introduces the basic concepts and the relevant terminology. Examples of the concepts are given using the Mod-3 puzzle. Section 3 presents RWM with its two methods, refinement and macro generation. A step-by-step trace of the refinement of the Mod-3 problem is given. Section 4 deals with the complexity of the RWM method. It also presents empirical results and strategies that RWM has learned for some difficult problems. Section 5 compares RWM with other techniques.

2. Basic Concepts and Terminology

Before describing the RWM method, we define the basic concepts on which it is based. Examples will be given for motivation using the Mod-3 problem.

A problem will be represented as a quadruple $P = \langle I(s), G(s), M, S \rangle$. S is the set of states for the problem. $G(s)$ is the *goal statement* which specifies the goal states. $G(s)$ will be true if and only if s is a goal state. The goal statement of the Mod-3 Puzzle is "Every square in state s has the same value." $I(s)$ is the *initial statement* which will be true for any initial state. In the Mod-3 Puzzle $I(s)$ is true since the initial state can be any state, whereas $I(s)$ for the subproblem at the second stage is $\{s_{11} = s_{12}\}$ because the first stage produces such states. M is the set of moves to be used in solving the problem. In the Mod-3 Puzzle there are 9 moves available, $M = \{o_{11}, o_{12}, \ldots, o_{33}\}$, namely one move for each square.

A *problem instance* p is a pair $\langle P, s_{\text{init}} \rangle$ of problem P and a particular *initial state* $s_{\text{init}} \in S$, for which $I(s_{\text{init}})$ is true.

In this work, an *atomic statement* is a binary predicate with two arguments. This representation is due to the fact that every n-ary atomic statement can be written using binary predicates [10], which are easier to process. Arguments of an atomic statement can be constants or *state components*. State components are the values manipulated by the operators. For example, s_{11} through s_{33} are the state components of the Mod-3 Puzzle. A *statement* is a set of atomic statements with the implied AND connective in between them. For instance, in the Mod-3 Puzzle, $s_{11} = s_{12}$ is an atomic statement. The goal statement is the statement $\{s_{11} = s_{12}, s_{11} = s_{13}, s_{11} = s_{21}, \ldots, s_{32} = s_{33}\}$. Every statement $Q(s)$ represents the set of problem states $\{s \mid Q(s)\}$. Therefore, statements will be used to describe the sets of states. An empty statement is true, and represents

S, the set of all problem states. The union of two statements represents the intersection of the sets of states represented by these statements. If $Q(s)$ is a subset of $R(s)$, then every problem state s that satisfies $R(s)$ also satisfies $Q(s)$, that is, $R(s)$ logically implies $Q(s)$. This class of statements does not exhaust all possibilities because disjunctions are not used; only conjunctions are used. Although this is a limitation which simplifies the computations performed by RWM, most of the problems we have looked at, and their associated subproblems can be naturally represented by this special class of statements.

A *move* is represented by a pair $\langle PC(s), A \rangle$, where $PC(s)$ is the *precondition statement*, possibly empty, and A is the set of *assignments* which describe the result of the move. Formally, a move $m : \{s | PC(s)\} \rightarrow S$, where S is the set of all problem states. If $PC(s)$ is true for all s, then m is a total function from S to S.

Each assignment defines the value that a state component will have after the application of the move. An assignment $a_i \in A$ has the form: $a_i : s_i \leftarrow t_i$, where s_i is a state component and t_i is a term. A term is a constant, a state component, or a function applied to one or more terms.[3] As an example, the o_{11} move of the Mod-3 Puzzle has the following form:

$$\begin{aligned} o_{11} = \langle \emptyset,\ & (s_{11} \leftarrow inc_3(s_{11})) \\ & (s_{12} \leftarrow inc_3(s_{12})) \\ & (s_{13} \leftarrow inc_3(s_{13})) \\ & (s_{21} \leftarrow inc_3(s_{21})) \\ & (s_{31} \leftarrow inc_3(s_{31}))\rangle\ , \end{aligned}$$

where $inc_3(x)$ is $(x + 1) \bmod 3$.

The moves given in the problem description are called *primitive moves* (or *operators*). A *macro move* (or *macro* for short) is a finite sequence of operators. *Move* will be used as a general term for both operators and macros. In both cases, $m(s)$ will denote the state obtained by applying move m to state s.

A move m is *safe* over state $Q(s)$ if, when move m is applied to a state s for which $Q(s)$ is true, the resulting state will also satisfy $Q(s)$; i.e., $Q(m(s))$ will also be true. Formally, this is

$$\forall s\ \{Q(s) \Rightarrow Q(m(s))\}\ .$$

For example, the moves o_{13} and $(o_{21}o_{22})$ are safe over the statement $\{s_{11} = s_{12}\}$.

[3] In the implementation of RWM we used only unary functions.

A move m is *irrelevevant* to going from $Q(s)$ to $R(s)$ if m is safe over $Q(s)$ and when applied to a state that satisfies $Q(s)$ but not $R(s)$ the result state $m(s)$ will never satisfy $R(s)$; i.e., $R(m(s))$ will never be true. In other words, m is also safe over $Q(s)\&\neg R(s)$. For example, the move o_{13} is irrelevant to going from $\{s_{11} = s_{12}\}$ to $\{s_{23} = s_{33}\}$.

A move m is *relevant* to going from $Q(s)$ to $R(s)$ if m is safe over $Q(s)$ and not irrelevant to going from $Q(s)$ to $R(s)$. That is, there is a chance that $R(m(s))$ will be true if $Q(s)$ is true and $R(s)$ is false. For example, the move $(o_{21}o_{22})$ is relevant to going from $\{s_{11} = s_{12}\}$ to $\{s_{23} = s_{33}\}$.

A move m is *potentially applicable* to a state that satisfies $Q(s)$ if the precondition statement $PC(s)$ of m does not conflict with $Q(s)$, that is,

$$\exists s \{Q(s) \ \& \ PC(s)\} .$$

Moves m_i and m_j have the *same effect* with respect to statement $Q(s)$ if

$$\forall s \{Q(m_i(s)) \Leftrightarrow Q(m_j(s))\} .$$

For instance, the moves o_{21} and o_{31} have the same effect with respect to the statement $\{s_{11} = s_{12}\}$.

A *strategy* for the problem $P = \langle I(s), G(s), M, S\rangle$ is a sequence of subproblems $P_1, P_2, \ldots, P_n$, which we call *stages*. Let $P_i = \langle I_i(s), G_i(s), M_i, S\rangle$; then the strategy must satisfy the following conditions:

$$I_1(s) \Leftrightarrow I(s) , \tag{1}$$

$$I_i(s) \Leftrightarrow I_{i-1}(s) \ \& \ G_{i-1}(s) \quad \text{for } 1 < i < n , \tag{2}$$

$$G(s) \Leftrightarrow G_1(s) \ \& \ G_2(s) \ \& \cdots \& \ G_n(s) , \tag{3}$$

$$M_i \neq \emptyset , \qquad M_i \subseteq M,$$

$$\forall m \in M_i, \ m \text{ is relevant to going from } I_i(s) \text{ to } G_i(s) . \tag{4}$$

The first condition says that the initial statement of the first subproblem must be the same as the initial statement of the given problem. The initial statement for any other subproblem is equal to the conjunction of the initial statement and the goal statement of the preceding subproblem, as stated in (2). For instance, the initial statement of the third subproblem, $I_3(s)$ in Fig. 1 is $\{s_{11} = s_{12}, s_{23} = s_{33}\}$. The conjunction of the goal statements of all subproblems must be the same as the goal statement of the given problem, as formulated in (3). The last condition guarantees that for every subproblem there are some relevant moves. Normally such subproblems are easier than the main problem because the subproblem goals are part of the main goal.

3. The RWM Method

In the first section we gave two example strategies which are similar to the ones learned and used by human problem solvers. In this section we will explain how these strategies can be learned mechanically using the RWM method. The step-by-step process of learning the strategy shown in Fig. 1 for the Mod-3 Puzzle will be given for motivation. RWM method consists of two separate processes, namely *refinement* and *macro generation*. These two methods and the way they are used will be explained in detail. The result of RWM is a GPS-based strategy. How this strategy maps onto GPS, and is used by the problem solver, will be explained in detail. The result of RWM is a GPS-based strategy. How this strategy maps onto GPS, and is used by the problem solver, will be explained in detail.

3.1. A trace of RWM

Having defined the basic concepts used in RWM, we can now walk through the steps of learning the strategy for Mod-3 Puzzle shown in Fig. 1. RWM first tries to refine the given problem into a sequence of easier subproblems.

Step 1. The first step of the refinement process is to determine the relevant moves for each atomic statement of the goal. Some of the 36 atomic statements of the goal of the Mod-3 Puzzle are shown in Table 1 with their relevant moves.

Step 2. At this step, the atomic statements that have exactly the same set of relevant moves are grouped into one statement. This results in 15 statements; some of them are given in Table 2.

Step 3. The goal of the first subproblem will be the statement with the largest number of safe moves over it. Therefore, for each of the above statements, the

Table 1
Atomic statements and their relevant moves in the Mod-3 problem

Atomic statement	Relevant moves
$s_{11} = s_{12}$	$\{o_{21}, o_{22}, o_{31}, o_{32}\}$
$s_{11} = s_{13}$	$\{o_{21}, o_{23}, o_{31}, o_{33}\}$
$s_{11} = s_{21}$	$\{o_{12}, o_{13}, o_{22}, o_{23}\}$
$s_{11} = s_{22}$	$\{o_{11}, o_{13}, o_{22}, o_{23}, o_{31}, o_{32}\}$
$s_{11} = s_{23}$	$\{o_{11}, o_{12}, o_{22}, o_{23}, o_{31}, o_{33}\}$
...	...
$s_{23} = s_{31}$	$\{o_{11}, o_{13}, o_{22}, o_{23}, o_{31}, o_{32}\}$
$s_{23} = s_{32}$	$\{o_{12}, o_{13}, o_{21}, o_{23}, o_{31}, o_{32}\}$
$s_{23} = s_{33}$	$\{o_{21}, o_{22}, o_{31}, o_{32}\}$
$s_{31} = s_{32}$	$\{o_{11}, o_{12}, o_{21}, o_{22}\}$
$s_{31} = s_{33}$	$\{o_{11}, o_{13}, o_{21}, o_{23}\}$
$s_{32} = s_{33}$	$\{o_{12}, o_{13}, o_{22}, o_{23}\}$

Table 2
Statements and their relevant moves in the Mod-3 problem

Statement	Relevant moves
$\{s_{11} = s_{12}, s_{23} = s_{33}\}$	$\{o_{21}, o_{22}, o_{31}, o_{32}\}$
$\{s_{11} = s_{13}, s_{22} = s_{32}\}$	$\{o_{21}, o_{23}, o_{31}, o_{33}\}$
$\{s_{11} = s_{21}, s_{32} = s_{33}\}$	$\{o_{12}, o_{13}, o_{22}, o_{23}\}$
$\{s_{11} = s_{22}, s_{13} = s_{32}, s_{32} = s_{31}\}$	$\{o_{11}, o_{13}, o_{22}, o_{23}, o_{31}, o_{32}\}$
...	...
$\{s_{13} = s_{33}, s_{21} = s_{22}\}$	$\{o_{11}, o_{12}, o_{31}, o_{32}\}$

moves that are safe over that statement will be determined. There are 10 statements with the highest number of 5 safe moves, and the other 5 statements have 3 safe moves each. Any one of the former is eligible to be the goal statement of the first subproblem; the one chosen is $\{s_{11} = s_{12}, s_{23} = s_{33}\}$. The moves that are safe over this statement are $\{o_{11}, o_{12}, o_{13}, o_{23}, o_{33}\}$, and these are the only moves that will be used in solving the remainder of the problem.

Step 4. To determine the adequacy of these 5 moves we will check whether each of the remaining 34 atomic statements of the goal statement, $G(s) - \{s_{11} = s_{12}, s_{23} = s_{33}\}$, has at least one move relevant to it. Since this is true, we have discovered the first subproblem of the strategy. Its moves are those which are relevant to its goal; its initial statement is true since it is the first stage of the strategy and the initial statement of the problem is true.

This refinement process will continue on the rest of the problem until no more refinement is possible.

Step 1′. This time we will use the moves $\{o_{11}, o_{12}, o_{13}, o_{23}, o_{33}\}$ since those are the only moves that will not violate the goal of the first subproblem. The new goal is the original goal excluding the goal of the first subproblem; i.e., $G(s) - \{s_{11} = s_{12}, s_{23} = s_{33}\}$. The relevant moves for each atomic statement of this new goal are determined as in Step 1. Some of these 34 atomic statements and their relevant moves are shown in Table 3.

Table 3
Atomic statements and their relevant moves during the refinement of the rest of the Mod-3 problem

Atomic statement	Relevant moves
$s_{11} = s_{13}$	$\{o_{23}, o_{33}\}$
$s_{11} = s_{21}$	$\{o_{12}, o_{13}, o_{23}\}$
$s_{11} = s_{22}$	$\{o_{11}, o_{13}, o_{23}\}$
$s_{11} = s_{23}$	$\{o_{11}, o_{12}, o_{23}, o_{33}\}$
...	...
$s_{31} = s_{33}$	$\{o_{11}, o_{13}, o_{23}\}$
$s_{32} = s_{33}$	$\{o_{12}, o_{13}, o_{23}\}$

Table 4
Statements and their relevant moves during the refinement of the rest of the Mod-3 problem

Statement	Relevant moves
$\{s_{11} = s_{13}, s_{12} = s_{13}, s_{21} = s_{31}, s_{22} = s_{32}\}$	$\{o_{23}, o_{33}\}$
$\{s_{11} = s_{21}, s_{12} = s_{21}, s_{13} = s_{31}, s_{23} = s_{32}, s_{32} = s_{33}\}$	$\{o_{12}, o_{13}, o_{23}\}$
$\{s_{11} = s_{22}, s_{12} = s_{22}, s_{13} = s_{32}, s_{23} = s_{31}, s_{31} = s_{33}\}$	$\{o_{11}, o_{13}, o_{23}\}$
$\{s_{11} = s_{23}, s_{11} = s_{33}, s_{12} = s_{23}, s_{12} = s_{33}, s_{21} = s_{32}, s_{22} = s_{31}\}$	$\{o_{11}, o_{12}, o_{23}, o_{33}\}$
$\{s_{11} = s_{31}, s_{12} = s_{31}, s_{13} = s_{21}, s_{22} = s_{23}, s_{22} = s_{33}\}$	$\{o_{12}, o_{13}, o_{33}\}$
$\{s_{11} = s_{32}, s_{12} = s_{32}, s_{13} = s_{22}, s_{21} = s_{23}, s_{21} = s_{33}\}$	$\{o_{11}, o_{13}, o_{33}\}$
$\{s_{13} = s_{23}, s_{13} = s_{33}, s_{21} = s_{22}, s_{31} = s_{32}\}$	$\{o_{11}, o_{12}\}$

Step 2′. These atomic statements are grouped into 7 statements such that each atomic statement of a group has the same set of relevant moves as in Table 4.

Step 3′. For each of these statements the safe moves are computed. The statement with the largest number of safe moves will be selected as the goal of the second subproblem. Of the 7 statements, 2 have 3, 4 have 2 and 1 has only 1 safe move. The statement $\{s_{11} = s_{13}, s_{12} = s_{13}, s_{21} = s_{31}, s_{22} = s_{32}\}$ with the safe moves $\{o_{11}, o_{12}, o_{13}\}$ is chosen as a candidate for the goal of the second subproblem.

Step 4′. Since one of these safe moves is relevant to each of the 30 atomic statements in the rest of the goal, we have discovered the second subproblem. Its moves are the ones that are safe over its initial statement which is the goal of the first stage and relevant to its goal.

We continue this process on the rest of the goal, $G(s) - \{s_{11} = s_{12}, s_{23} = s_{33}, s_{11} = s_{13}, s_{12} = s_{13}, s_{21} = s_{31}, s_{22} = s_{32}\}$, using the moves $\{o_{11}, o_{12}, o_{13}\}$ in the same way as above, which results in two more subproblems. The result of the refinement process is the four-stage strategy shown in Fig. 2. The atomic statements that are implied by the ones in the figure are not shown; e.g.,

Subproblem P_1:	Goal :	$\{s_{11} = s_{12},\ s_{23} = s_{33}\}$
	Moves :	$\{o_{21},\ o_{22},\ o_{31},\ o_{32}\}$
Subproblem P_2:	Goal :	$\{s_{11} = s_{13},\ s_{21} = s_{31},\ s_{22} = s_{32}\}$
	Moves :	$\{o_{23},\ o_{33}\}$
Subproblem P_3:	Goal :	$\{s_{11} = s_{21},\ s_{22} = s_{23}\}$
	Moves :	$\{o_{12},\ o_{13}\}$
Subproblem P_4:	Goal :	$\{s_{11} = s_{22}\}$
	Moves :	$\{o_{11}\}$

Fig. 2. The strategy for the Mod-3 Puzzle obtained after the first refinement.

P_{11}:	Goal :	$\{s_{11} = s_{12}\}$
	Moves :	$\{o_{21}, o_{22}\}$
P_{12}:	Goal :	$\{s_{23} = s_{33}\}$
	Moves :	$\{(o_{21}o_{22}), (o_{31}o_{32})\}$

Fig. 3. Stage 1 after refinement with new moves.

$s_{12} = s_{13}$ is not shown in the second stage, because it is implied by $s_{11} = s_{12}$ and $s_{11} = s_{13}$. The first stage of the strategy in Fig. 2 corresponds to the first two stages of the one in Fig. 1.

Sometimes subproblems are too difficult to solve directly. In these cases it is desirable to learn finer-grain strategies for their solution; e.g., the first stage of Fig. 2. Let us try to further refine that stage. Since each of the moves o_{21}, o_{22}, o_{31} and o_{32} are relevant to both $s_{11} = s_{12}$ and $s_{23} = s_{33}$, this stage cannot be refined with the given set of moves. What is needed is some moves that are safe over a part of its goal and relevant to going from that part to the rest of the goal. In order to discover such moves, we generate new macro moves by combining two operators. At this point we know that moves $\{o_{21}, o_{22}, o_{31}, o_{32}\}$ are relevant to going from any state to each of the atomic statements in $\{s_{11} = s_{12}, s_{23} = s_{33}\}$, and moves $\{o_{11}, o_{12}, o_{13}, o_{23}, o_{33}\}$ are irrelevant. During the macro generation, either two relevant moves or, one irrelevant and one relevant move will be combined. For the first stage of the strategy in Fig. 2, 30 such macro moves are generated; 28 of them are found to be relevant for this subproblem.

Armed with these new moves, we can further refine this subproblem by the refinement process described above. The goal statement is $\{s_{11} = s_{12}, s_{23} = s_{33}\}$, and the moves are $\{o_{21}, o_{22}, o_{31}, o_{32}, (o_{11}o_{21}), (o_{11}o_{22}), (o_{11}o_{31}), (o_{11}o_{32}), (o_{12}o_{21}), (o_{12}o_{22}), (o_{12}o_{31}), (o_{12}o_{32}), (o_{13}o_{21}), \ldots\}$. The refinement process results in the two subproblems shown in Fig. 3.

If several relevant moves have the same effect with respect to the goal statement of a subproblem, only the shortest one is reported. For example, in the first subproblem, o_{31}, $(o_{11}o_{21})$, $(o_{21}o_{23})$, $(o_{21}o_{33})$, $(o_{22}o_{22})$, $(o_{23}o_{31})$ and $(o_{31}o_{33})$ are not reported since they have the same effect as o_{21} with respect to $\{s_{11} = s_{12}\}$. The first stage of the strategy in Fig. 2 is replaced by the two subproblems found as its refinement, resulting in a five-stage strategy for the Mod-3 Puzzle. If the same process of macro generation and refinement is applied to the third and fourth stages, the strategy given in Fig. 1 will be obtained.

3.2. Relation of RWM to GPS

RWM produces a list of subproblems $P_1, P_2, \ldots, P_n$. This result is a GPS-based strategy to solve a problem P. The goal statement $G_i(s)$ of each

subproblem $P_i = \langle I_i(s), G_i(s), M_i, S \rangle$ corresponds to a difference for GPS. The ordering of differences is defined by the order of the subproblems, i.e., P_1 corresponds to the most difficult difference and P_n to the easiest difference. The problem solver solves a problem P by first solving the subproblem P_1, then P_2, and so on. When solving a subproblem P_i, the problem solver conducts a search using only the moves in M_i to find the states that satisfy $G_i(s)$, which corresponds to removing that difference in GPS nomenclature. Thus, the moves in M_i are the ones that GPS considers "relevant" to this difference. Backtracking may occur, since an initial state satisfying $I_i(s)$ may be a dead-end state for this subproblem. In that case, another state satisfying $I_i(s)$ is chosen, and the search continues. Of course, such a state would correspond to another solution to P_{i-1} which may be generated by backtracking in previous stage.

The moves in the set M_i are safe over the initial statement $I_i(s)$ by the definition of a stage in (1)–(4). Therefore, these moves are also safe over the goal statements of the previous stages, that is, they do not reintroduce any previously removed differences. Furthermore, the moves in M_i are relevant (not irrelevant) to going from $I_i(s)$ to $G_i(s)$.

The solutions to the last subproblem are also solutions to the problem as a whole. Since those states satisfy the initial statement of the last stage $I_n(s) = I(s)\ \&\ G_1(s)\ \&\ \cdots\ \&\ G_{n-1}(s)$ and its goal statement $G_n(s)$, they satisfy the goal of the problem P.

3.3. Refinement

Refinement is our method for learning differences and their ordering along with the set of relevant moves for each difference. In more formal terms, refinement is a method for generating subproblem goals $G_1, \ldots, G_n$ from the goal statement $G(s)$ of a given problem $P = \langle I(s), G(s), M, S \rangle$ and finding the sets of relevant moves $M_1, M_2, \ldots, M_n$, such that $P_1, P_2, \ldots, P_n$ satisfy the conditions required to be a strategy for P in (1)–(4), where $P_i = \langle I_i(s), G_i(s), M_i, S \rangle$. If such a strategy is found, then the sequence $P_1, P_2, \ldots, P_n$ is returned. If no refinement is possible with the given set of moves, problem P is returned unchanged. The formal description of the refinement method is shown in Fig. 4.

The first step of the refinement process is to find the relevant moves for each atomic statement of the goal. Whether or not a move is relevant to an atomic statement depends on the *domain-dependent knowledge* (e.g., properties of inc_3 function in Mod-3 Puzzle) and how it is determined will be explained in the next section. If there is an atomic statement for which no relevant moves are found, the refinement method considers this problem as "unsolvable." Assuming that every atomic statement of the goal has some relevant moves, this situation will not occur. On the other hand, since the refinement method is defined recursively, this heuristic is used to estimate whether or not the rest of the problem after the creation of a subproblem is solvable.

refine $(< I(s), G(s), M, S >)$:

1. For each atomic statement $g_i(s) \in G(s)$, find the set of moves, M_i, that are relevant to going from $I(s)$ to $\{g_i(s)\}$. If there is any atomic statement $g_i(s)$ with no relevant moves, return "unsolvable."
2. Form statements, $G_i(s)$, by grouping the atomic statements with the same set of relevant moves into one statement. If there is only one statement, return $< I(s), G(s), M, S >$.
3. For each statement $G_i(s)$, determine the set of moves, MS_i, that are safe over and potentially applicable to $I(s) \cup G_i(s)$, and form a candidate $< G_i(s), M_i, MS_i >$. Sort the list of candidates so that $|MS_i| \geq |MS_{i+1}|$.
4. While the list of candidates is not empty do:

   Choose the first candidate $< G_1(s), M_1, MS_1 >$.

   Let *rest* be **refine** $(< I(s) \cup G_1(s), G(s) - G_1(s), MS_1, S >)$.

   If *rest* is not "unsolvable" then return $< I(s), G_1(s), M_1, S >$ followed by *rest*,

   else remove the first candidate from the list of candidates.

   End of while. Return $< I(s), G(s), M, S >$.

Fig. 4. The refinement method.

The atomic statements $g_i(s)$ that have exactly the same set of relevant moves are grouped together to form statements $G_i(s)$ in the second step. The heuristic used here is that if a set of atomic statements have exactly the same set of relevant moves, then there is a high amount of interaction between them, and therefore they should be satisfied at the same time. If all the moves in M are relevant to every atomic statement in $G(s)$, then no more refinement is possible, and the refinement process terminates by returning the problem P unchanged.

In the third step, the set of moves MS_i that are safe over and potentially applicable to both $I(s)$ and $G_i(s)$ are calculated for each $G_i(s)$. During this process all the moves in M are tested. This is to determine the moves that can be used in the latter stages, if the statement $G_i(s)$ is selected to be the goal of the first stage. For each statement $G_i(s)$, a candidate $\langle G_i(s), M_i, MS_i \rangle$ is formed. Then this list of candidates is sorted on the size of MS_i, so that the candidate with the largest number of safe moves is the first in the list.

The actual ordering of the stages takes place in the fourth step. The first candidate $G_1(s)$ in the list is selected. In order for this candidate to be the first stage of the refinement, the rest of the problem should be solvable. The rest of the problem has as its goal the original goal $G(s)$ with $G_1(s)$ removed. Its moves are the moves that are safe over (and potentially applicable to) its initial

statement which is $G_1(s)$ added to $I(s)$. The test of solvability is done by trying to refine the rest of the problem recursively. If the result is a message indicating that the rest is unsolvable, then the next candidate in the list is tried. Otherwise the result of the refinement is a list whose first element is the subproblem representing the selected candidate and the rest of the list is the refinement found for the rest of the problem. If all the candidates are exhausted, the refinement process terminates unsuccessfully returning the problem P unchanged.

3.4. Macro generation

As seen in the Mod-3 example, more relevant moves are needed to further refine a "difficult" subproblem. The macro generation method of RWM described below is designed to find such moves for a given problem (or a subproblem) $P = \langle I(s), G(s), M, S \rangle$.

Since only such moves are used in solving P, the new moves needed must be safe over $I(s)$. The macro generation method uses all the moves in MS, the set of moves that are safe over $I(s)$. The following theorem guarantees that macro moves generated by composing two moves from MS are also safe over $I(s)$.

Theorem 3.1. *Given moves m and m' that are safe over $Q(s)$, a macro move (mm'), generated by composing m and m', is also safe over $Q(s)$.*

Proof. Since m is safe over $Q(s)$, $Q(s)$ implies $Q(m(s))$ for all $s \in S$. Similarly $Q(s)$ implies $Q(m'(s))$ for all $s \in S$. Therefore, $Q(m(s))$ implies $Q(m'(m(s)))$, which can be rewritten as $Q((mm')(s))$. □

During the macro generation, two moves from MS are composed to produce a new macro move. Such a macro move looks like just another move to the refinement process and the problem solver.

Two kinds of moves are needed. Primarily, we need moves that are safe over a subset of the goal statement and relevant to going from that part to the rest of the goal. Such moves are required to further refine the given problem. Secondarily, we need moves that have a different effect, than the other moves in M, with respect to the goal statement $G(s)$. Such moves enlarge the set of states from which a goal state can be reached. Therefore, these moves increase the efficiency of the strategy by reducing the amount of backtracking during the problem solving process.

An important issue for efficiency is the choice of the moves to be used. The moves in MS can be separated into two classes; M is the set of moves that are relevant to $G(s)$, and $MI = MS - M$ being the set of moves that are irrelevant to each of the atomic statements in $G(s)$. The following theorem shows that a macro move generated by composing two irrelevant moves is also an irrelevant move, and therefore need not be considered.

Theorem 3.2. *Let m and m' be irrelevant to going from $Q(s)$ to $R(s)$. Then (mm') is also irrelevant to going from $Q(s)$ to $R(s)$.*

Proof. Since m and m' are both irrelevant to going from $Q(s)$ to $R(s)$, they both are safe over $Q(s)$ & $\neg R(s)$. By Theorem 3.1, (mm') is also safe over $Q(s)$ & $\neg R(s)$. Hence it is irrelevant to going from $Q(s)$ to $R(s)$. □

That is, if m and m' are in MI, then (mm') is also irrelevant to going from $I(s)$ to $G(s)$. Hence such compositions do not yield moves that we are looking for.

Let us now consider a macro move (mm') which is generated by composing a relevant move $m \in M$ followed by an irrelevant move $m' \in MI$. Because of the way refinement procedure groups the atomic statements, m' is irrelevant to every atomic statement $g_i(s) \in G(s)$. Hence, (mm') is safe over $I(s)$ & $g_i(s)$ and relevant to going to $g_i(s)$ only if m has the same property. Therefore, (mm') does not help in the refinement of the given problem. Furthermore, for all s satisfying $I(s)$ if $(mm')(s)$ satisfies the goal $G(s)$ then $m(s)$ satisfies $G(s)$ as well. That is, (mm') is not any better than m for the problem solver either. Hence such moves need not be generated. Therefore, during macro generation for a given problem P, either two relevant moves, or one irrelevant move[4] followed by a relevant move are composed. This selective generation of macros helps speed up the macro generation process and reduces the number of moves to be tested for relevancy.

Another issue that requires attention is the precondition statements of moves. If either of the two moves used has a precondition, then the composition will have a precondition as well. If $PC_m(s)$ is the precondition statement of m, and $PC_{m'}(s)$ is that of m', then the precondition statement of the composition (mm') is $PC_m(s)$ & $PC_{m'}(m(s))$. That is, it is the precondition of the first move and the precondition of the second which is modified to reflect the effect of the first move. If this statement turns out to be always false, then m' can not be applied after m, and (mm') is not generated.

Above we have described a single level of macro generation. In some cases, this does not produce moves which would allow the given subproblem to be refined. In this case, another level of macro generation is needed which has as input all the safe moves including the ones generated by the first level. Since each level of macro generation can double the length of macros, the length of macros generated by the second level may be four times as long as those input to the first level. Also the number of macros generated grows at the same exponential rate. This is the reason why it is important to focus on only those compositions that are likely to yield relevant moves; the above theorems allow a large number of macros to be removed from consideration. The number of

[4] In our implementation of RWM, MI is saved for each stage along with M.

levels of macro generation to be used is determined by the user of the system because this gives the user direct control over the exponential part of the process. Even though this process tries to eliminate as many macros as possible, this reduction may not sufficiently reduce the exponential growth of the moves. In some problems, such as Pyraminx the available memory was exhausted before all the difficult subproblems were refined. For such problems, only the moves that have a different effect with respect to the goal of the stage, instead of all relevant moves, were saved. This decision is also under direct user control.[5]

3.5. Creation of subproblems for preconditions

In some problems moves may have preconditions, and such a move can only be applied if its precondition statement is satisfied first. In order to use a move m with a precondition in solving a subproblem P_i, not only must it be relevant to going from $I_i(s)$ to $G_i(s)$, but also it must be potentially applicable to $I_i(s)$. That is, $I_i(s)$ and the precondition statement $PC_m(s)$ must not conflict. RWM creates a separate subproblem for using such a move in a particular stage. The goal of this subproblem is the precondition statement of the move, and the initial statement is the same as the initial statement of the stage for which the move is relevant. Such subproblems are treated the same way as the main problem and can be refined into sequences of subproblems. During the problem solving process, the problem solver has to solve this subproblem to find a state satisfying the precondition of the move, and then it can apply the move to the resulting state. GPS also provides for this by generating differences between a state and the domain of an operator.

A good example of subproblem creation can be seen in the Tower of Hanoi problem [3, 22]. A strategy learned by RWM for solving the 3-disk Tower of Hanoi problem is shown in Fig. 5. The goal statement is $\{s_1 = C, s_2 = C, s_3 = C\}$, where s_i is the position of the ith disk. o_{ijk} denotes the move of disk i from

P_1:	Goal :	$\{s_3 = C\}$
	Moves :	$\{o_{3AC}, o_{3BC}\}$
P_2:	Goal :	$\{s_2 = C\}$
	Moves :	$\{o_{2AC}, o_{2BC}\}$
P_3:	Goal :	$\{s_1 = C\}$
	Moves :	$\{o_{1AC}, o_{1BC}\}$

Fig. 5. A strategy for the 3-disk Tower of Hanoi.

[5] This option was used for Pyraminx and the Trillion Puzzle.

peg j to peg k. All moves except o_{1BA}, o_{1CA}, o_{1AB} and o_{1CB} are safe over $\{s_1 = C\}$.

All disk-1 and disk-3 moves, o_{2AC} and o_{2BC} are safe over $\{s_2 = C\}$; and all disk-1 and disk-2 moves, o_{3AC} and o_{3BC} are safe over $\{s_3 = C\}$. Among the moves that are safe over $\{s_1 = C\}$ only o_{2AB}, o_{2BA}, o_{3AB} and o_{3BA} are also potentially applicable to $\{s_1 = C\}$. All disk-1 moves, o_{3AB} and o_{3BA} (a total of 8 moves), are both safe over and potentially applicable to $\{s_2 = C\}$. However, there are 12 moves, all disk-1 and disk-2 moves, that are both safe over and potentially applicable to $\{s_3 = C\}$. Since $\{s_3 = C\}$ yields the largest number of moves that can be used to solve the rest of the problem, it is selected first. The rest of the goal statement is $\{s_1 = C, s_2 = C\}$. The moves that can be used to achieve the rest of the goal are all disk-1 and disk-2 moves. Since the rest of the goal is solvable, in the sense that each of the atomic statements in the rest of the goal has a relevant move among these moves, $\{s_3 = C\}$ becomes the goal of the first stage of the strategy. There are two moves that can be used in the first stage; they are o_{3AC}, which moves disk-3 from peg A to peg C, and o_{3BC}, which moves disk-3 from peg B to peg C. The problem solver selects the right move to apply according to the position of disk-3 in the initial state. That is, it selects o_{3AC} since disk-3 is on peg A; o_{3BC} is selected if it is on peg B. Similarly, $\{s_2 = C\}$ is the goal of the second subproblem because the preconditions of the disk-1 moves do not involve the position of disk-2. To be more precise, all disk-1 moves are both safe over and potentially applicable to $\{s_2 = C\}$ whereas only o_{2AB} and o_{2BA} are safe over and potentially applicable to $\{s_1 = C\}$. Thus, more moves are safe over and potentially applicable to $\{s_2 = C\}$ which is the reason it is selected as the goal of the second subproblem. The remaining statement $\{s_1 = C\}$ becomes the goal of the final subproblem.

The precondition statement for the move o_{3AC} is $\{s_1 = B, s_2 = B, s_3 = A\}$. A separate subproblem for finding a state that satisfies this precondition is created. The refinement of this subproblem results in the strategy shown in Fig. 6.

P_1:	Goal :	$\{s_3 = A\}$
	Moves :	$\{o_{3BA}, o_{3CA}\}$
P_2:	Goal :	$\{s_2 = B\}$
	Moves :	$\{o_{2AB}, o_{2CB}\}$
P_3:	Goal :	$\{s_1 = B\}$
	Moves :	$\{o_{1AB}, o_{1CB}\}$

Fig. 6. A strategy for solving the precondition of o_{3AC} in the first stage of the Tower of Hanoi problem.

The stages of this strategy are ordered in the same way as in the strategy for the main problem. The first stage deals with disk-3 because all of the disk-1 and disk-2 moves are safe and potentially applicable after solving this stage. Similarly, only disk-1 moves, o_{3CA} and o_{AC} can be used after $\{s_2 = B\}$, and only o_{2CA}, o_{2AC}, o_{3CA}, o_{3AC} are safe over and potentially applicable to $\{s_1 = B\}$. Note that, the first stage of this strategy is redundant, since o_{3AC} will be selected only when disk-3 is on peg A, in which case the initial state satisfies the goal of the first subproblem.

Similarly, separate subproblems are created for the other operator preconditions and each is refined in a similar manner. For example, the precondition of o_{2AB} is refined into two subproblems whose goals are $\{s_2 = A\}$ and $\{s_1 = C\}$. Like P_1 in Fig. 6, the first stage will always be satisfied since the operator is only selected by the problem solver when disk-2 is on peg A. Note that disk-3 operators are not used in this strategy for the precondition of o_{2AB} because the disk-3 operators are not safe over P_1 in Fig. 5. Thus, they are not eligible for solving P_2 and P_3 in Fig. 5 and subproblems arising from the preconditions of the operators used to solve them.

It is instructive to compare this strategy with the one learned by Korf's [9] method for the 3-disk Tower of Hanoi problem. The first stage of the Korf's strategy is to move disk-1 (the smallest disk) to the goal peg. The next step is to move disk-2 to the goal peg. Of course, this step requires disk-1 to be removed from the goal peg and then to be moved back. The next step is to move disk-3 to the goal peg which is accomplished by removing disk-1 and disk-2 from the goal peg and then returning them again. Although this behavior seems wasteful, it is necessary for Korf's method because this is the only ordering of state components which results in serial decomposability. Korf also allows for intermediate target values which are different than goal values. This is a form of subgoaling which reduces the kind of wasteful behavior described above, but unfortunately is not sufficiently powerful to eliminate it. The subproblems that RWM generates for the Tower of Hanoi give rise to a classical solution in which the correct operator is selected at each point in solving the problem. This is the same solution GPS found [3], but GPS was given the strategy and RWM learns it mechanically. DGBS also learned the same strategy [4].

3.6. The RWM system

The RWM system for learning strategies is based on taking a problem and refining it into a sequence of "easier" subproblems, which collectively constitute a strategy for solving the given problem. In order for a subproblem to be easier than the given problem, the goal statement of the subproblem must be easier to satisfy than the goal of the problem. The statement $Q(s)$ will be considered easier to satisfy than the statement $R(s)$ if $R(s)$ implies $Q(s)$. For

instance, in the Mod-3 Puzzle the statement $\{s_{11} = s_{12}, s_{23} = s_{33}\}$ is easier to satisfy than the goal statement.

The RWM system first applies the refinement method to the given problem. This should result in a sequence of subproblems whose goal statements are easier than that of the problem in hand. Some of these subproblems may still be difficult to solve themselves. Such a subproblem cannot be refined further, since all the moves of that subproblem are relevant to all the atomic statements of its goal. More relevant moves need to be found first. Therefore, RWM generates new macro moves for such a subproblem. These new moves are tested for relevancy. The ones that are relevant are added to the move set of the subproblem. Some of these newly generated moves may be relevant to the further stages of the strategy as well. Therefore, after generating a set of moves for stage i, the move sets of all stages $j \geq i$ are updated with these moves.

When generating moves for a difficult stage, it is possible that no new moves are found. In that case, RWM generates moves for the previous stage. This backtracking continues until some new moves for the difficult stage are found.

The block diagram of the RWM system is shown in Fig. 7. The refiner and the macro generator directly implement the refinement and the macro generation methods described earlier.

In various stages of the process of learning a strategy, questions such as whether a move is safe over a statement, or if it is relevant to going from one statement to another are raised. In order to answer such questions, problem domain-related knowledge is needed. This knowledge is provided to the RWM system along with the description of a problem, and stored in the *domain-dependent knowledge base* (DDKB). The domain-related knowledge can be thought of as a set of general rules that describe the effect of the functions used in the moves and the predicates used in the statements. DDKB also contains

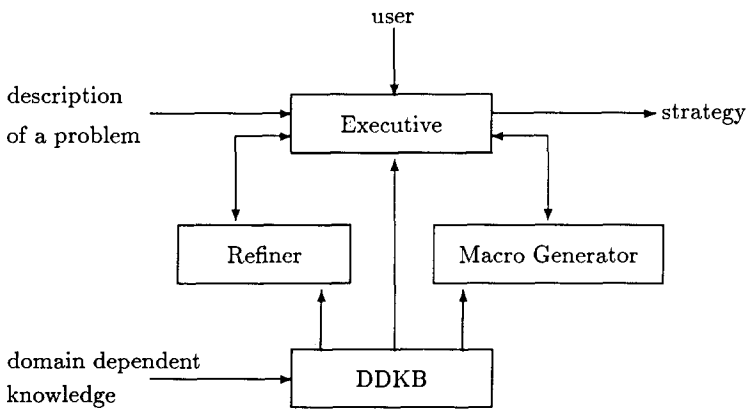

Fig. 7. Block diagram of the RWM system.

information about compositions of functions. For example, the DDKB for Mod-3 problem includes the fact that $inc_3(inc_3(inc_3(x))) = x$. For problems that use the same set of predicates and functions the same DDKB can be used. For instance, the same DDKB is used for Rubik's Cube, Pyraminx and the Eight Puzzle.

DDKB is designed to answer questions of the form,

"Does $Q(s)$ imply $r(m(s))$?"

where $Q(s)$ is a set of atomic statements and $r(m(s))$ is an atomic statement. DDKB answers "yes" to such a question if it can infer $r(m(s))$ from $Q(s)$ using the domain dependent knowledge provided. Otherwise, the answer is simply "don't know." That is, the input to DDKB is a statement $Q(s)$, an atomic statement $r(s)$ and a move m; and its output is "yes" or "don't know." For example, the safety of move m over an initial statement $I(s)$ is determined by asking the question,

"Does $I(s)$ imply $i_j(m(s))$?"

for each atomic statement $i_j(s)$ in $I(s)$. If the answer is "yes" for all atomic statements in $I(s)$, then m is safe over $I(s)$.

If the move m has a precondition statement $PC_m(s) = \{pc_{m,1}(s), \ldots, pc_{m,k}(s)\}$, then m also has to be potentially applicable to the initial statement $I(s)$. This is determined by asking the class of questions,

"Does $I(s)$ & $pc_{m,1}(s)$ & $\cdots$ & $pc_{m,i-1}(s)$ imply $\neg pc_{m,i}(s)$?"

for $1 \leq i \leq k$. If the answer is "yes" for any i, then m is not applicable to $I(s)$, otherwise it is potentially applicable. For instance, to determine the applicability of o_{3AC}, whose precondition statement is $\{s_1 = B, s_2 = B, s_3 = A\}$, to the statement $\{s_2 = C\}$ in the Tower of Hanoi problem, the refiner first asks DDKB the question,

"Does $\{s_2 = C\}$ imply $s_1 \neq B$?"

Since the answer is "don't know", the refiner asks the second question,

"Does $\{s_2 = C, s_1 = B\}$ imply $s_2 \neq B$?"

Because s_2 is equal to C, which is different from B, the answer is "yes." Therefore, o_{3AC} is not potentially applicable to $\{s_2 = C\}$, i.e., there cannot be any state satisfying both $\{s_2 = C\}$ and $\{s_1 = B, s_2 = B, s_3 = A\}$.

During the refinement process, moves that are relevant to going from an

initial statement $I(s)$ to an atomic statement $g_i(s)$ of a goal are sought. In order to determine the relevancy of a move m, the refiner asks the question,

"Does $I(s)$ & $\neg g_i(s)$ imply $\neg g_i(m(s))$?"

If the answer is "yes," then the move is irrelevant, otherwise it is considered as relevant to $g_i(s)$. For example, in the second stage of the Mod-3 Puzzle, $(o_{21}o_{22})$ is relevant to $s_{23} = s_{33}$, since the answer to the question,

"Does $\{s_{11} = s_{12}, s_{23} \neq s_{33}\}$ imply $inc_3(inc_3(s_{23})) \neq s_{33}$?"

is "don't know." On the other hand, o_{13} is irrelevant at the same stage, because the answer to the question,

"Does $\{s_{11} = s_{12}, s_{23} \neq s_{33}\}$ imply $inc_3(s_{23}) \neq inc_3(s_{33})$?"

is "yes." That is, if s_{23} and s_{33} are not equal before the application of o_{13}, they will remain unequal afterwards as well.

Note that since the effect of predicates and functions are separated from the RWM as domain-dependent knowledge, RWM can be used with any kind of predicates and functions.

The executive's task is to handle the interaction between the various parts of the RWM system and the user. It first reads the description of a problem along with the relevant DDKB. Then the problem is refined by the refiner, and the problem is replaced with its refinement. This refinement constitutes a raw strategy for solving the problem. If the user "estimates" that some subproblem is difficult, then new moves are generated by the macro generator for this subproblem. The moves of the current and the latter stages are updated by adding the newly generated moves that are safe over their initial statements. Then the difficult subproblem is tried for further refinement.

The user may choose to create subproblems for some moves with preconditions. In that case a separate subproblem whose goal statement is the precondition statement of a move selected by the user is created. Then this subproblem is refined to obtain a strategy for the precondition of this move.

Through the executive, the user has a control over the course of the process of learning a strategy. The user can choose to refine a subproblem, generate moves for a given subproblem, or create a new subproblem for the precondition of a move at a given stage. Since the decision on whether a subproblem is difficult or whether to create subproblems for moves is left to the user, the RWM system can be used to learn strategies that satisfy certain user criteria. Although RWM is a man–machine system, the user's role is somewhat indirect. Never does the user give RWM direct information about the strategy like the goal of a particular stage should be a particular statement or like the

order of different stages. However, the user has some control over such things as the number of stages in a strategy and the length of the macros used in the strategy.

4. Complexity of RWM and Empirical Results

In this section, the refinement and the macro generation processses will be analyzed in terms of the time required to refine a problem and the time required to generate macro moves for a given problem, respectively.

4.1. Time complexity of the refinement

Let a problem $P = \langle I(s), G(s), M, S \rangle$ be input to the refinement procedure shown in Fig. 4. The first step is to determine the relevant moves for each atomic statement $g_i(s)$ in $G(s)$. The relevancy of a move m, where m is safe over $I(s)$, to an atomic statement $g_i(s)$ is determined by asking the following question to the DDKB (domain-dependent knowledge base),

"Does $I(s)$ & $\neg g_i(s)$ imply $\neg g_i(m(s))$?"

Let r be the response time of DDKB to such a question. Then, the time required for determining the relevant moves for all atomic statements in $G(s)$ is $|G(s)| \cdot |M| \cdot r$, where $|G(s)|$ denotes the number of atomic statements in $G(s)$. Solvability of a problem is tested by checking that the set of relevant moves for each atomic statement of $G(s)$ is non-empty. Since the response time of a DDKB question is much longer than the time required for any list comparison, the response time is the dominant factor in the time complexity of the refinement process. Also, r is constant for a given DDKB. Therefore, the complexity of the refinement method will be calculated in terms of the number of questions that will be asked to the DDKB. Thus, the time complexity of the first step is $O(|G(s)| \cdot |M|)$.

The complexity of the second step can be omitted because it does not involve asking any question to the DDKB. In the third step, the moves that are safe over $I(s)$ & $G_j(s)$ are determined for each group. Since all the moves in M are known to be safe over $I(s)$, only their safety over $G_j(s)$ needs to be checked. Safety of a move m over $G_j(s)$ is determined by asking the question,

"Does $I(s)$ & $G_j(s)$ imply $g_{j,k}(m(s))$?"

for each $g_{j,k}(s)$ in $G_j(s)$, which takes $|G_j(s)| \cdot r$ time. If m has a precondition $PC_m(s)$, then it has to be potentially applicable to $G_j(s)$ as well, which requires $|PC_m(s)| \cdot r$ time in the worst case. Therefore, the time complexity of determining the moves that are safe over and potentially applicable to $I(s)$ & $G_j(s)$ for a

group G_j is

$$O((|G_j(s)| + n_p) \cdot |M|) ,$$

where n_p is the average number of atomic statements in a precondition of a move. This will be repeated for each group G_j. Let n_g be the number of groups resulting from the second step of refinement process. Since $\Sigma_j |G_j(s)| = |G(s)|$, the time complexity of the third step is

$$O((|G(s)| + n_g n_p) \cdot |M|) .$$

The total complexity of the first three steps is

$$\begin{aligned} &O(|G(s)| \cdot |M| + |G(s)| \cdot |M| + n_g n_p \cdot |M|) \\ &\quad = O((2|G(s)| + n_g n_p) \cdot |M|) . \end{aligned}$$

The complexity of ordering the groups in the fourth step is omitted because it does not require asking any questions to the DDKB.

In order to determine the first stage of the strategy, the first three steps can be executed at most n_g times, which in turn requires

$$O((2|G(s)| + n_g n_p) \cdot n_g \cdot |M|)$$

calls to the DDKB. In the worst case, $n_g = |G(s)|$, the complexity of finding the first stage is

$$O(|G(s)|^2 \cdot (2 + n_p) \cdot |M|) .$$

If the total number of stages that will result is n_s, then the total time complexity of the refinement process is

$$O(|G(s)|^2 \cdot (2 + n_p) \cdot |M| \cdot n_s) .$$

In the case of $|G(s)|$ stages, the worst case in terms of time complexity, the total number of calls to the DDKB during the refinement process is

$$O(|G(s)|^3 \cdot (2 + n_p) \cdot |M|) .$$

As an example, experimentally measured times required for the refinement of the Mod-3 Puzzle played on 3×3, 4×4 and 5×5 boards are shown in Table 5. The strategies obtained by refining the 4×4 and 5×5 Mod-3 Puzzles are given in [7]. For each size of the Mod-3 Puzzle, the *rest of the problem* is found

Table 5
Measured times for refining the Mod-3 Puzzle on different boards sizes

Board size	$\|G(s)\|$	$\|M\|$	n_s	Measured time (sec.)
3×3	36	9	4	6.2
4×4	120	16	8	76
5×5	300	25	12	458

to be solvable for each stage, that is, the number of questions asked to the DDKB is a function of $|G(s)|$ rather than $|G(s)|^2$. Also $n_p = 0$ since the moves do not have preconditions. According to the calculations given above, the maximum number of questions that will be asked to the DDKB in the Mod-3 Puzzle is $2 \cdot |G(s)| \cdot |M| \cdot n_s$. The response time of the DDKB used for the Mod-3 Puzzle is about 2 milliseconds. Therefore, the time for the refinement of the 3×3 Mod-3 Puzzle should have been 5.2 seconds. The difference betweeen the calculated and the measured times is due to the time spent for grouping and sorting in the second and the fourth steps. The time for the refinement of the Mod-3 Puzzle on a 4×4 board is measured to be 12.2 times longer than that of a 3×3 board. This agrees very well with the analytical results. Also the refinement time for the 5×5 Mod-3 Puzzle is measured to be 69.4 times that of 3×3 Mod-3 Puzzle which is predicted by the analysis.

4.2. The time complexity of macro generation

When generating macro moves for a given subproblem, either two relevant moves or an irrelevant move and a relevant move are composed. Therefore, the time complexity of the macro generation process is $O((|M| + |MI|) \cdot |M|)$, where $|M|$ is the number of relevant moves and $|MI|$ is the number of irrelevant moves of the subproblem. However, the macro generation normally takes place more than once because one set of new moves might not lead to the refinement of the problem at hand. Therefore, the number of macros generated increases exponentially. This is because the total number of newly generated moves is the total number of all old safe moves times the number of old relevant moves.

Another issue here is the length of the macros. Since two moves are composed to generate a new macro, the length of some of the newly generated macros is twice the length of the old moves, and each level of macro generation has this length doubling potential. Therefore, the macros generated using this method may be longer than the optimal macros for the same task. The generation of macros is the main bottle-neck in the RWM method because of its exponential nature. This is also the case with Korf's method even though its macro generation process is very different than that of RWM. The difficulty is very problem dependent because good strategies for some problems require

macros that are relatively long and there are exponentially many macros of this required length. This leads to difficulties in both the time and the space complexity.

4.3. Empirical results

A program implementing the RWM method has been written in LISP and runs on a VAX 11/780. We now sketch some test problems and the empirical results of RWM on these problems. The detailed description of these problems can be found in [7] along with the strategies learned.

Mod-3 Puzzle

The strategy learned for the Mod-3 Puzzle played on a 3×3 board is shown in Fig. 1. During the process of learning this strategy 57 macros were generated. It took RWM 13 seconds to learn this strategy.

$2 \times 2 \times 2$ Rubik's Cube

This is a $2 \times 2 \times 2$ version of the $3 \times 3 \times 3$ Rubik's Cube puzzle. A six-stage strategy, shown in Section 1, for solving this puzzle was learned by RWM in 397 seconds. The first stage of the strategy is to make the adjacent facelets of the two bottom left cubies the same color. The following two stages make the lower half of the cube have the same colors on their adjacent facelets. The rest of the strategy completes the puzzle. This strategy is also similar to the strategies used by humans. During the process of learning this strategy a total number of 1695 macro moves were generated. The longest macro contains 91 operator applications, and only swaps the cubies in the upper front right and the upper back left corners. One reason that some of the macros are so long is that only three operators are given in the problem specification; thus a 270-degree rotation is accomplished by a macro move consisting of three applications of a 90-degree rotation.

Pyraminx

The puzzle, commercially available under the name of Pyraminx, can be thought of as a pyramidal or tetrahedral version of the Rubik's Cube. There are two kinds of rotations possible at each of four corners. Counterclockwise rotations are defined in the operators. A clockwise rotation is accomplished by two counterclockwise rotation. A strategy of twelve stages is learned in 138 seconds. During this process 285 macros are generated, by saving only the moves that have a different effect with respect to the goal statement of each stage. The longest macro is 75 operators long, but again some of this length is due to the use of two operators to do a simple clockwise rotation. In the first four stages the four small cubies are rotated until they have the same colors as the adjacent big cubies. The following five stages make all the adjacent facelets

of the cubies in the front layer of the pyramid, the same color. The rest of the strategy completes the puzzle. This strategy is very similar to the one learned and used by humans for this puzzle [18].

Tower of Hanoi problem

The refinement of the 3-disk Tower of Hanoi problem results in a three stage strategy. In the first stage of the strategy the largest disk, in the second stage the medium, and in the third stage the smallest disk are moved to the goal peg. Moving the largest disk in the first stage requires that a precondition involving the other smaller disks be satisfied. Similarly, to move the medium disk a precondition involving the smallest disk must be satisfied. Separate subproblems are created for the preconditions of the operators used in the first two stages. These subproblems are refined individually which results in separate strategies for solving the preconditions. This process took 16 seconds, and no macros were generated.

Eight Puzzle

The Eight Puzzle has been extensively studied in the artificial intelligence literature [24]. The refinement of this puzzle results in an eight-stage strategy. At each stage one tile is moved to its goal position. Macros are generated for the first six stages to enrich their move sets. A strategy for the Eight Puzzle was learned in 631 seconds; and 711 macros were generated. The longest macro is of length fourteen.

Monkey and Bananas problem

Although it is simple, the well-known Monkey and Bananas problem [22] has several interesting properties. First, most of its operators have preconditions. Second, the representation of this problem given to RWM is not serially decomposable as is required by Korf's method [9]. Third, the goal statement of the problem has only one atomic statement, therefore it cannot be directly refined. This problem is used to show how different strategies can be learned by either only creating subproblems for operator preconditions, or only generating macros, or both.

The first strategy was learned by only generating macros. It was learned in 7 seconds and 7 macros were generated. The longest macro was of length 4. This strategy has only one stage, which contains a macro for each possible position of monkey and the box so that monkey can grab the bananas. The second strategy was learned by creating subproblems for operator preconditions only, and then refining them. RWM learned a two-stage strategy for grab, and one-stage strategies for push and climb operators. This strategy was learned in 4 seconds. In the third strategy both techniques were used. This strategy was learned in 5 seconds and 6 macros were generated. The longest macro was only 2 operators long. In learning this strategy, only a subproblem for the precondi-

tion of the grab operator was created and refined into a two-stage strategy. Then, length-2 macros were generated for its stages.

Rubik's Magic

This puzzle is commercially available under the name Rubik's Magic; it consists of eight squares connected to each other by very thin cords. Each square is attached to two others by these cords. The squares are marked by colored arcs. The goal is to arrange the squares so that the overall picture is three whole rings linked together. However, the shape of this goal figure is not given. A three-stage strategy for solving this puzzle was learned in 115 seconds by generating 22 macros. The longest macro was of length 3. The first stage is to have the same colors on the adjacent edges of the two lower left squares. In the second stage, upper left three and lower left two squares arranged so that they make a complete picture. The third stage brings the puzzle to its goal state. This strategy is also very similar to the one learned by humans for this puzzle [19].

Trillion Puzzle

This is a commercially available puzzle which is quite difficult to solve. It consists of four blue, four green, four red and one white chips placed on four concentric circles and one center point. The circles are marked red, yellow, green and blue, going outwards. The chips can be moved by three concentric transparent rings and a slide lever, Fig. 8.

Moving the outer ring rotates the chips on the blue circle. The chips on the green and the yellow circles are rotated by the middle ring. The inner ring rotates the chips on the red circle. The slide lever pushes all the chips on the

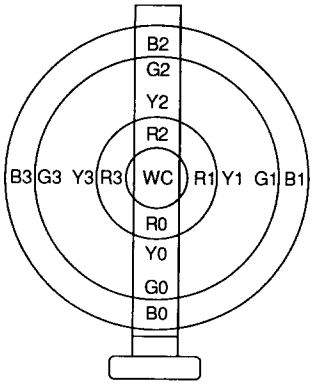

Fig. 8. The Trillion Puzzle.

vertical line up one position. While the lever is pushed the inner or the middle rings can be rotated but not the outer ring. The goal is to get the chips of one color on the circle of that color, and do this for all four colors.

RWM has learned a 14-step strategy for this problem in 10 minutes [8]. For learning this strategy 2719 macros were generated. Some of the macros are quite long (the longest one is 118 operators long); each clockwise rotation is represented by three counterclockwise rotations which are primitive. The first four stages put the blue colored chips on the outer ring. The next two stages put the two green chips on the G1 and G3 positions, postponing the other green chips to the eighth and tenth stages. The seventh stage puts a yellow chip on the Y2 position. The other yellow chips are put in their goal positions in the ninth, eleventh and twelfth stages. The thirteenth stage puts a red chip on the R2 position. The other red chips and the white chip are put on their goal positions in the last stage. After a small amount of experimentation, it is obvious that the blue chips should be done first. However, the rest of the solution is not obvious because there is a lot of interaction between the remaining positions. The ordering of the green and yellow chips is difficult to discover, which is an important part of the strategy learned by the RWM. Using the strategy results in long solutions; i.e., solutions that contain many primitive operators. But it is also the only way we know of to solve this problem. There may be a better way to solve it than to use the RWM strategy, but we are quite sure there is no easy way to solve it.

The strategies learned by RWM have been tested on a large number of problem instances, and a solution was found for all the initial states. Most of these strategies seem to be efficient in the sense that the amount of search required for each stage is relatively small. Also these strategies are quite similar to the strategies used by humans. Humans often learn a number of moves for each stage of their strategy, and apply a search using these moves in solving a stage. This is quite different than the strategies learned by Korf's method in which there is one macro learned for each possible next step in the *macro problem solving* (MPS) process. Of course, Korf's strategies are very efficient because no search is required, but also quite different from human strategies.

Only the unique operators of a problem are given in its description. For example, only counterclockwise rotations of the front layer of the Rubik's Cube is defined, since a 180-degree turn of the front layer is equal to two counterclockwise rotations of the same layer. Similarly a clockwise rotation is equal to three counterclockwise turns. It is left to the macro generator to generate these moves if they are really useful. However, this makes the macros appear to be longer than they really are, because for example, a clockwise rotation shows up as three separate rotations in the macro.

5. Discussion

RWM is a method that learns efficient strategies for some problems such as the Mod-3 Puzzle and Tower of Hanoi problem, but does not work for some others such as the Fool's Disk.[6] Hence, an important issue is how to characterize the problems for which RWM is able to learn good strategies. Before attempting to characterize the class of problems applicable to RWM, we will look at the characterization of Korf's method.

Korf [9] has been able to characterize the class of problems for which his method is applicable. These problems, which he calls *serially decomposable*, have the property that the value of the ith component of the result of an operator is a function of only the first i components of the state to which the operator is applied. MPS only applies macro m_{ij} to a state in which the first $i-1$ components have their goal values and the ith has j as its value. Since the first i components of the input are known, these components of the output are also known because of the serial decomposability. Thus, m_{ij} performs the desired function because of the way it is generated. The key property is that state components other than the first i components have no effect on the first i components of m_{ij}'s output.

This is an important characterization because if a problem does not have this property then his method should not be considered, otherwise the method will definitely learn a macro table to solve the problem. However it is important to realize that decomposability is defined relative to an ordering on the state components, and hence for some orders a problem may be serially decomposable while for others it is not. Moreover, this characterization is valid under the assumption that every legal state is solvable in the sense that there is a path from it to the goal state. However, many problems do not have this property because there are "dead-end" states for which solutions do not exist. This shows up in the macro table as empty slots. For such problems, even though there may be solutions, Korf's method will fail because it does not backtrack when it finds an empty entry in a macro table.

Another difficulty with such a characterization is that it does not say anything about the efficiency of the strategy that will be learned. That is, even if a problem is serially decomposable, the strategy may not be efficient. This difficulty can be illustrated by the strategy learned for the Tower of Hanoi problem, which is very inefficient as explained in Section 3.5.

The difficulty with the characterization of the problem solving and the strategy learning methods is to capture the features such as efficiency. In fact, this difficulty is present in all artificial intelligence techniques such as A* and resolution-based strategies [17] because there are very few theoretical results which characterize the efficiencies of such methods. Although the theory of resolution theorem proving has many results on the completeness of different

[6] A strategy for solving this puzzle has been discovered by Ernst and Goldstein [4].

search strategies, very little is known about their efficiency. All these strategies allow many legal inferences to be pruned from their search spaces, but they are all inefficient on some theorems because their use increases the length of the proofs to the point where much more search is required. Thus, the study of the efficiency of strategies has been almost entirely limited to empirical investigations, and RWM is no exception. However, it is instructive to attempt to characterize the class of problems for which RWM is useful in an informal way.

Because of the difficulties explained above, instead of giving a concrete characterization (such as decomposability) of the problems that are suitable for RWM, we will discuss the RWM method in terms of its restrictions and efficiency in general.

5.1. Restrictions due to representation

RWM is designed for state space problems. The goal states of the problem space should be represented by a statement which is a conjunction of lower-level (atomic) statements about the values of state components. This is necessary for the success of the refinement method since it tries to partition the goal statement into a sequence of subgoals. However, this is not a requirement for the RWM method in general. A good example of this is the Monkey and Bananas problem, whose goal statement is $\{Monkey's\ hand = Bananas\}$. Although the problem itself cannot be refined because its goal contains only one atomic statement, new subproblems can be created for the operator preconditions.

There are no requirements on the operators, such as decomposability. For example, the Monkey and Bananas problem as given to the RWM is not serially decomposable. However, only unary functions are allowed in the assignments of an operator. This restriction is imposed solely for the sake of simplicity of the particular implementation developed in this research. A different implementation could remove this restriction.

5.2. Safety of moves over portions of the goal

In order to refine a problem, its moves must be safe over some portions of the goal statement and relevant to going from that portion to the rest. If a move is safe over the whole goal statement, then this move is not going to be useful in solving the problem. For example, a move that rotates the whole cube in the Rubik's Cube puzzle cannot be used to solve that problem. On the other hand, if a problem has no moves that are safe over only a portion of the goal in terms of its atomic statements, then the refinement process will fail to refine that problem, and return it unchanged. However, this is not a requirement for the RWM method as a whole. In such cases, new macro moves that are safe over portions of the goal statement can be generated. A good example of this situation is a subproblem found as a stage after the refinement. All the moves of a stage are unsafe over any portion of its goal. After generating new macro

moves for that subproblem, some of these moves may be safe over portions of its goal. This situation can be true for the problem itself. Therefore, such a requirement on the safety of the primitive operators over portions of the goal statement does not constitute a characterization for the applicability of the RWM method, but it is desirable for a problem to have this property. The major requirement for RWM is that there exist macros, which are not "too long," which are safe over some portions of the goal but not others. The difficulty is that this information is not explicit in the problem specification and must be inferred from it by a process such as RWM.

5.3. Subproblems for move domains

In order for a move to be used in a stage, it must be "potentially" applicable to the initial statement of that stage. That is, the initial state does not have to satisfy the precondition of that move. However, GPS allows giving separate strategies for solving the problem of finding a state in the domain of such a move. In such cases, GPS solves that new problem by using the strategy provided for the domain of that move. Those separate strategies are needed for the efficiency of the overall strategy. A good example of this is the strategy learned for the Tower of Hanoi problem, where separate strategies are learned for each move.

Generating separate strategies for move domains may not always be the best strategy. For problems whose moves are relevant to each other's precondition this may lead to inefficient strategies or no strategy at all. In such cases, generating macro moves may yield more efficient strategies. A good example of this case is the Eight Puzzle, where each move is relevant to the precondition of some other moves. In the strategy learned for the Eight Puzzle only macro moves are used; no subproblems for transforming a state into the domain of an operator are used.

A combination of separate strategies for move domains and macro generation is also possible. An example of such a strategy is given for the Monkey and Bananas problem. The strategies learned only by generating separate strategies for move domains may have many stages while the strategies with only macro moves may contain large number of moves. Therefore, a hybrid of these two techniques seems to be appropriate for some problems.

6. Conclusions

RWM is a method for mechanically learning GPS based problem solving strategies. It incorporates two separate methods, namely, *refinement* and *macro generation*. The former is to partition a given problem into a sequence of easier subproblems which constitutes a raw strategy to solve the problem. The latter is to learn more relevant moves, so that a difficult subproblem can be further refined.

The RWM method is applicable to many state-space problems such as those described in Section 4.3. However, its implementation explained in this document has some restrictions over the representation of problems. For example, only unary functions are allowed in the assignments of the operators. The use of the available memory was not efficient; e.g., a complete strategy for the $3 \times 3 \times 3$ Rubik's Cube could not be found because of memory limitations. A more serious deficiency is that refinement only generates subproblems whose goals are subsets of the main goal. Thus, it could not learn a good strategy for the Fool's Disk [4].

The RWM method has been used to learn strategies for a number of problems. The strategies learned have been tested on a large number of problem instances, and a solution was found for all the initial states tested. Most of these strategies seem to be efficient in the sense that the amount of search required for each stage is quite small. Also these strategies are in some respects similar to the strategies used by humans.

Although RWM is a man–machine system, the user's role is quite indirect. Never does the user give RWM direct information about the strategy, like the goal of a particular stage should be a particular statement, or like the order of different stages. However, the user has some control over such things as the number of stages in a strategy and the length of the macros used in the strategy.

Most of the research discussed in this paper is concerned with learning GPS-based strategies. In addition to RWM, the methods of Korf [9] and Ernst and Goldstein [4] have mechanically learned strategies of this type. Each method has its strengths and weaknesses. For example, Ernst and Goldstein's discovery system does not use macro moves that are shown here to be very useful in strategies. Korf's technique is not designed to find an ordering of subproblems which constitutes a very crucial part of a strategy, and does not allow multiple goal states. However, these techniques have learned some clever and non-obvious strategies for different types of problems. The RWM method is based on the ideas in these previous learning methods; it shows how they can be combined and expanded. Although the resulting method can learn some strategies that the previous methods could not, the former cannot learn all of the strategies learned by the latter. Thus something has been lost in the combination and extension, but more importantly something has been gained. Each of these research efforts contributes to the overall understanding of this kind of learning, which is somewhat limited, but collectively the results of this research are quite impressive.

ACKNOWLEDGEMENT

This research was conducted at Case Western Reserve University. We would like to thank Leon Sterling and David Helman for many useful comments. David Davenport and the referees provided many helpful and greatly appreciated comments that improved the previous version of this paper.

REFERENCES

1. S. Amarel, Expert behaviour and problem representations, Laboratory for Computer Science Research, Rutgers University, New Brunswick, NJ (1982).
2. R.B. Banerji, GPS and the psychology of the Rubik cubist: A study in reasoning about actions, in: A. Elithorn and R. Banerji, eds., *Artificial and Human Intelligence* (North-Holland, Amsterdam, 1983).
3. G.W. Ernst and A. Newell, *GPS: A Case Study in Generality and Problem Solving* (Academic Press, New York, 1969).
4. G.W. Ernst and M.M. Goldstein, Mechanical discovery of classes of problem-solving strategies, *J. ACM* **29** (1982) 1–23.
5. G.W. Ernst, Means-ends analysis, *Encyclopedia of Artificial Intelligence* (Wiley, New York, 1987).
6. A.H. Frey Jr and D. Singmaster, *Handbook of Cubik Math* (Enslow, Hillside, NJ, 1982).
7. H.A. Güvenir, Learning problem solving strategies using refinement and macro generation, Tech. Rept. CES-87-22, Computer Engineering and Science Department, Case Western Reserve University, Cleveland, OH (1987).
8. H.A. Güvenir and G.W. Ernst, A method for learning problem solving strategies, in: *Proceedings AAAI Spring Symposium Series*, Stanford University, Stanford, CA (1988).
9. R.E. Korf, Macro-operators: A weak method for learning, *Artificial Intelligence* **26** (1985) 35–77.
10. R. Kowalski, *Logic for Problem Solving* (North-Holland, New York, 1979).
11. J.E. Laird, P.S. Rosenbloom and A. Newell, Chunking in Soar: The anatomy of a general learning mechanism, *Machine Learning* **1** (1986).
12. S. Minton, Quantitative results concerning the utility of explanation-based learning, in: *Proceedings AAAI-88*, St. Paul, MN (1988) 564–569.
13. T.M. Mitchell and R. Keller, Goal-directed learning, in: *Proceedings International Machine Learning Workshop*, Urbana-Champaign, IL (1983).
14. A. Newell, H.A. Simon and J.C. Shaw, A variety of intelligent learning in a general problem solver, in: M.C. Yovits and S. Cameron, eds. *Self-Organizing Systems: Proceedings of an Interdiciplinary Conference* (Pergamon Press, Oxford, 1960) 153–189.
15. A. Newell, Learning, generality and problem solving, in: *Proceedings IFIP Congress* (North-Holland, Amsterdam, 1963) 407–412.
16. A. Newell and H.A. Simon, *Human Problem Solving* (Prentice-Hall, Englewood Cliffs, NJ, 1972) 435–438.
17. N.J. Nilsson, *Principles of Artificial Intelligence* (Tioga, Palo Alto, CA, 1980).
18. J.G. Nourse, *The Simple Solutions to Cubic Puzzles* (Bantam Books, New York, 1981).
19. J.G. Nourse, *Simple Solutions to Rubik's Magic* (Bantam Books, New York, 1986).
20. J. Pearl, On the discovery and generation of certain heuristics, *AI Mag.* **4** (1983) 23–33.
21. H. Renko and S. Edwards, *Awesome Games for your Atari Computer* (Addison-Wesley, Reading, MA, 1984).
22. E. Rich, *Artificial Intelligence* (McGraw-Hill, New York, 1983).
23. C.C. Sims, Computational methods in the study of permutation groups, in: J. Leech, ed., *Computational Problems in Abstract Algebra* (Pergamon, New York, 1970).
24. P. Schofield, Complete solution of the eight puzzle, in: N.L. Collins and D. Michie, eds., *Machine Intelligence* **1** (Oliver & Boyd, Edinburgh, 1967).

Received September 1988; revised version received April 1989