Machine Learning 3: 285–317, 1989

A Heuristic Approach to the Discovery of Macro-operators

GLENN A. IBA (GI01@GTE.COM)
GTE Laboratories, Incorporated, 40 Sylvan Road, Waltham, MA 02254, U.S.A.

(Received: September 30, 1986)
(Revised: November 30, 1988)

Keywords: Macro-operators, search, problem solving, composition, empirical learning.

Abstract. This paper describes a heuristic approach to the discovery of useful macro-operators (macros) in problem solving. The approach has been implemented in a program, MACLEARN, that has three parts: macro-proposer, static filter, and dynamic filter. Learning occurs during problem solving, so that performance improves in the course of a single problem trial. Primitive operators and macros are both represented within a uniform representational framework that is closed under composition. This means that new macros can be defined in terms of others, which leads to a definitional hierarchy. The representation also supports the transfer of macros to related problems. MACLEARN is embedded in a supporting system that carries out best-first search. Experiments in macro learning were conducted for two classes of problems: peg solitaire (generalized "Hi-Q puzzle"), and tile sliding (generalized "Fifteen puzzle"). The results indicate that MACLEARN's filtering heuristics all improve search performance, sometimes dramatically. When the system was given practice on simpler training problems, it learned a set of macros that led to successful solutions of several much harder problems.

1. Introduction

In many domains, problem solving involves considering alternative sequences of *operators*. For such tasks, it is often profitable to gather certain subsequences into clusters or chunks, since it enables a speedier and more efficient search for a correct or desired solution. Such subsequences of operators are called *macro-operators*, or *macros*; each macro may be treated as a whole and may be regarded as just another operator to be used in problem solving.

There are two general approaches to improving search. One is to decrease the branching factor of the search tree, the other is to shorten the effective length of the solution path. Macro-operators implement the second idea by allowing the application of operator sequences as single steps. However, defining new macros also increases the branching factor of the search, which suggests that some selectivity should be exercised in their acquisition.

The discovery of macros may be regarded as a form of learning by composition, as in Iba (1986). The present paper explores macro discovery in the context of puzzle solving, a class of domains in which it proves very useful. A heuristic approach to the discovery of macro-operators has been implemented in MACLEARN, a program that solves problems by searching for operator sequences that achieve a desired goal. These operators may be primitive (e.g., the basic moves in board puzzles) or they may be macros defined by the system itself.

MACLEARN's macros are abstracted versions of compiled move sequences, which it proposes using variants of the peak-to-peak heuristic described in Iba (1985). The system also employs several types of static filtering, including checks for redundancy (equivalence) of macro-operators, limits on the allowable expanded lengths of macros, and a test on the post-conditions of macros. In addition, dynamic filtering deletes those macros which are never used as part of a solution. By default, MACLEARN exhibits *within-trial* learning, since macros are discovered in the course of a problem-solving trial, and macros learned early on may prove useful at later stages of the search. In an alternate mode, the system does *post-trial* learning, in which newly defined macros are not used during the current trial, but only on subsequent problem trials.

In order to test this approach to macro learning, MACLEARN was applied to two classes of puzzles: peg solitaire and tile sliding. The principal results were that the system successfully learned macros which enabled it to solve the full "Hi-Q" puzzle (peg solitaire) and the "Fifteen puzzle" (tile sliding), as well as variations; that some simple heuristics for static and dynamic filtering improved search for macros; and that within-trial learning was better than post-trial learning. MACLEARN was able to solve difficult Hi-Q puzzles after learning macros while solving simpler training puzzles. Without those macros, it could not solve the difficult problems directly. Thus, the system demonstrates a kind of development of expertise, in which training on simpler problems leads to macros that are useful in solving more difficult problems. That is, MACLEARN can take advantage of *transfer* to other problems.

Subsequent sections of this paper describe details of the learning system, and experiments carried out to test its effectiveness. Section 2 describes the general framework for learning macro-operators. Section 3 presents its application to and the results for peg solitaire, including the controlled studies of various filtering heuristics, and a comparison of within-trial and post-trial learning. Section 4 explores the generality of the approach by examining its application to the domain of tile sliding. Section 5 discusses the approach used, compares it with related work, and provides a summary of conclusions.

2. A general framework for learning macro-operators

MACLEARN learns by defining new macros and adding them to the search system's operator set. The objective is to improve that set so the system can solve problems faster or more economically. Viewing problem solving as a search in the space of operator sequences, macros let the program consider much longer sequences by representing them as shorter ones. This is equivalent

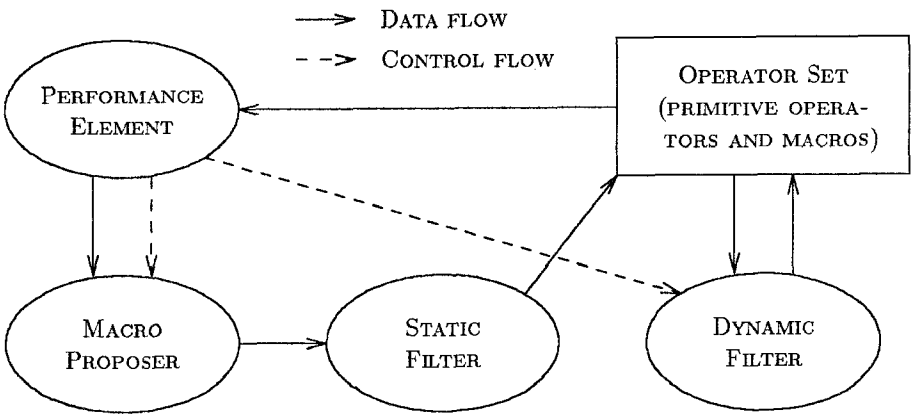

Figure 1. Performance element and learning model.

to decreasing the effective distance to the goal in the search space. MACLEARN also attempts to improve its performance by deleting less useful operators from the set, thus reducing the branching factor of the search.

Figure 1 summarizes the relations between the system's performance element and its three learning components – the macro proposer, the static filter, and the dynamic filter. A more detailed description of each system component follows below.

2.1 The performance element: Best-first search

The *performance element* employs best-first search to find a sequence of operators that transforms a given starting state into a goal state. Problem solving proceeds by expanding nodes of the search tree. A node consists of a problem state (position) paired with the operator sequence leading to that state. In expanding a node, each available operator is applied in turn to the node's state, generating a list of successor nodes. In best-first search an evaluation function determines the order in which the search tree is expanded, assigning a value to each node of the tree according to the estimated promise of its current position. Higher values are assigned to nodes that appear more promising. At each step of the search, an unexpanded node with maximal value is chosen for expansion, hence the term "best-first" search.

Several pieces of information must be supplied to the performance element:

- an evaluation function;
- a specification of the initial state;
- a specification of the goal;
- a specific initial operator set (the *primitive operators*); and
- a uniform mechanism for applying operators to states.

The details of these specifications differ according to the domain, and examples of each are given in Sections 3 and 4.

2.2 Representation of states, goals, and operators

For a problem-solving system to search, it must first be able to represent the states it will consider. This representation should allow easy matching against operators' conditions and easy computation of the states that result from operator application. Moreover, it should be easy to ascertain the equivalence of states. In the experiments reported in this paper, states are represented as two-dimensional arrays. The equivalence test is based on an element-by-element comparison of two arrays, taking into account rotations and reflections.

A *goal* is represented as a condition that must be satisfied by a state in order to qualify it as a *goal state*. One very simple condition is membership in a set of explicitly designated desirable states. In general, the goal condition may be any predicate, which implicitly defines the goal states as those for which the predicate is true.

In order to generate new states, the performance element must apply operators. Following Vere (1978), operators are represented as *relational productions*, which specify the conditions under which an operator can be applied (the *before* part), and the changes that result from application (the *after* part). Sometimes an operator may be applied to a state in more than one way; for example, if an operator jumps one piece over another, in some states more than one piece may be able to make such a jump, or the same piece may be able to jump in more than one direction. To resolve this ambiguity, *binding* information is paired with the operator to specify how it should be applied to a state. The operator, together with its binding information, is termed an *operator instance*. An operator instance is *legal* in a given state when the before part of the operator matches the partial current state specified by the binding information. Application involves substituting the after part of the operator for the partial state matched by the before part, thus generating a new state.

MACLEARN's operator set initially includes just the primitive operators, but it can expand this set by adding new macros, which it defines in terms of primitive operators or other macros. Macros are represented in the same way as primitive operators, but in addition to a before and after part, they have an *expansion* component, which defines the macro in terms of other operators. This leads to a hierarchy of macro definitions. By recursive expansion, any macro can be expressed as a sequence of primitive operators. The uniform representation of primitive operators and macros means that MACLEARN can use the same procedures to test and apply both of them. Thus the system can invoke a new macro immediately after adding it to the operator set.

2.3 Proposing new macros

The process of proposing macro-operators includes the following four steps:

1. triggering of macro proposal during search;
2. delimiting a sequence of operator instances;
3. composing and abstracting the delimited sequence; and
4. defining a new macro and passing it to the static filter.

Macro proposal is triggered during the search process whenever the system

detects a *peak* in the evaluation function along a path of the search tree. A peak is defined (relative to a given path) as a node whose value is greater than each of the two adjacent nodes along the given path. The following sections describe the processes of delimitation, abstraction, and composition which are employed in the macro proposer.

2.3.1 Delimiting operator sequences by the peak-to-peak heuristic

The macro proposer uses a *peak-to-peak* heuristic (Iba, 1985) to delimit sequences of operator instances for defining new macros. Recall that the macro proposer is invoked when the search system notices a peak at the end of the current search path (the one from the root to the most recently expanded node). This search path is now traversed backwards toward the root of the tree until the immediately preceding peak is located. If no such peak is found, then the root of the tree itself is used as a peak. The two peaks thus identified serve to delimit a specific sequence of operator instances. This sequence is then abstracted and composed to define a new macro, as described in Section 2.3.2.

The motivation behind the peak-to-peak heuristic is to smooth the search space by remembering operator sequences that helped in getting past "valleys". A good set of macros will tend to make the fixed evaluation function more monotonic; i.e., guide search on more of a steady uphill climb, with fewer descents into valleys. If the evaluation function is already monotonic, then the peak-to-peak heuristic will never be invoked, and no macros will be learned.

2.3.2 Composition and abstraction of operator sequences

The delimited sequence of operator instances is next *composed* into a single relational production. This is accomplished by defining before and after conditions that are equivalent to the transformation carried out by the entire sequence. Vere (1978) has described a very general method for composition of relational productions. His method has the important property of *closure* – that the composition of a sequence of relational productions is itself a relational production. This property is desirable so that MACLEARN can test the legality of macros and apply them using the same mechanisms as for primitive operators. Similar approaches to composition have been discussed by Dawson and Siklóssy (1977), Lewis (1987), and Neves and Anderson (1981) in the context of production systems.

MACLEARN takes a composition method as one of its procedural parameters. The experiments of this paper used simplified composition methods that were tied to the specific domains of peg solitaire and tile sliding. These were used for the sake of efficiency, but a more general mechanism would have served as well in principle.

The operator instance sequence, in addition to being composed, is abstracted by generalizing the associated binding information for each of the operator instances. Variables can be introduced for constants where that is appropriate, as in the tile-sliding procedures of Section 4.1. The abstraction process produces both a generalized operator sequence, and a generalized composite relational production. The resulting macro will apply in many more settings than the one in which it was proposed.

The process of composing and abstracting a sequence of operator instances is similar to that of goal-regression in explanation-based generalization (Mitchell, Keller, & Kedar-Cabelli, 1986), in that it constitutes a search for the weakest preconditions under which a given transformation is possible. The instantiated operator sequence plays the role of the example that drives generalization. The domain theory is the model of the operators and the state transformations they accomplish.

Once composition and abstraction are complete, the macro proposer defines a new macro. The before and after parts of the new macro are simply the before and after parts of the generalized composite relational production, whereas the abstracted operator sequence becomes the expansion part of the macro. The new macro is given a unique name and passed on to the static evaluator.

2.4 Static filtering of macros

As already mentioned, it is important to be selective in generating new macros, since each one increases the branching factor of the search. Therefore MACLEARN passes each new macro through its static filter, which performs a heuristic analysis of the macro description to decide whether the macro is likely to be useful. If the decision is positive, the macro is retained; otherwise it is discarded. Retained macros are immediately added to the operator set in the case of within-trial learning, whereas they are placed on a reserve list in post-trial learning.

The static filter uses three tests. First, a redundancy check screens out any macros that are equivalent to primitive operators or to previous macros. Since any given macro may be proposed more than once, this criterion helps to keep the operator set from growing needlessly. Next, a length test eliminates any macro whose expanded length exceeds a given threshold. The *expanded length* of a macro is just the number of primitive operators in its full expansion. This test is motivated by the intuition that a macro with greater expanded length will tend to have more complex preconditions and thus would be less often applicable. Finally, a domain dependent test allows for the specification of static criteria that depend on the specific domain and features of the chosen representation. Section 3.5 discusses a simple domain-dependent criterion used in peg solitaire, which is related to the evaluation function used in that domain. No domain dependent test was used in the tile-sliding domain.

2.5 Dynamic filtering of macros

The dynamic filter serves as an empirical counterpart to the static filter. The latter may occasionally pass macros that are not useful. The dynamic filter detects such cases by examining statistics on how macros have been used. Currently, the static filter pays attention only to whether or not a macro has appeared in the solution sequence in any of the solved problems. Credit for appearing in solution sequences is only assigned to the top-level macros in the solution; it is not inherited by macros that appear in the definitions of those macros. Thus, when a macro is removed from the operator set by the dynamic filter, its definition is retained in case it is needed to expand the definitions

of other macros still in use. Primitive operators are never removed from the operator set, since this could make certain problems unsolvable.

At present the dynamic filter is invoked manually, typically after a training sequence has been completed. This is to insure that macros have a reasonable opportunity to help solve several problems before being tested. If the filter were automatically invoked after every trial, it might remove potentially useful macros before they had an opportunity to prove their usefulness on other problems.

3. Application to peg solitaire

The first domain on which MACLEARN was tested is peg solitaire, a class of problems that includes the board puzzle sold commercially as Hi-Q. This is an appropriate domain not only because it includes some difficult problems, but also because it provides a useful point of comparison with the macro-learning technique of Korf (1982, 1983, 1985a, 1985b), which requires that a problem be *operator decomposable*.[1] Since peg solitaire is not operator decomposable, it demonstrates that MACLEARN's heuristic approach can be applied to a larger class of problems than Korf's method. Section 3.1 presents an example to illustrate the nature of peg-solitaire puzzles and how the system deals with them. Sections 3.2 through 3.5 describe how various parts of the MACLEARN system are specialized for application to peg solitaire. Section 3.6 describes experiments with peg solitaire and presents the results for this domain.

3.1 MACLEARN on a simple peg-solitaire problem

The following simple example serves to introduce peg-solitaire puzzles, and illustrates how macros can aid the solution process by shortening the effective solution path. The initial state is:

```
o o o .
. o o .
. o o .
```

Here and subsequently "o" represents a peg and "." represents a hole. The single operator (jump) consists of moving one peg horizontally or vertically over an adjacent peg into a hole. The peg jumped over is then removed from the board. The goal is to reach a board state that contains only one peg. Since each jump removes one peg, any solution has exactly six primitive steps.

The primitive jump operator can be represented as a relational production:

```
o o .   ⟶   . . o
```

This operator can be applied in any location of the board and in any of four directions, subject to obvious boundary constraints. The binding information

[1] A problem is operator decomposable if its states can be represented as vectors of state variables in such a way that the effect of each operator on any state variable depends only on the value of that state variable, and not on the values of any other state variables (see Korf, 1983).

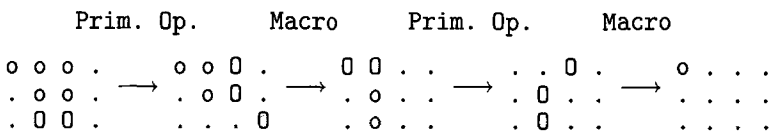

Figure 2. Solution of a simple peg-solitaire problem.

in an operator instance specifies the position and orientation in which the operator is matched against the board. In the initial state above, there are five legal operator instances.

Shortly into its search on this puzzle, MACLEARN encounters a peak that triggers macro proposal. Together with a previous peak, this delimits a sequence of two moves or primitive operator instances. The macro proposer defines a macro with the following before and after parts:

```
          o - -                  . - -
Before:   . o o        After:    . . .
          . - -                  o - -
```

The dashes "-" mean *"don't care"*; they are needed because macros are represented using rectangular arrays. The system uses this macro twice in the first solution it finds, which appears in Figure 2. In this figure, uppercase "O"'s indicate the pegs to which an operator is applied. The advantage of using the macro is that the solution sequence is only four steps long instead of six. This means that the search takes less time, even though more operators are applied at every move.

3.2 Representation and matching in peg solitaire

Board states for the peg-solitaire domain are represented as rectangular arrays. The elements of a board array are either pegs, holes, or voids. Voids represent locations of the array where pegs are not permitted, such as the corners of the Hi-Q board (see Figure 3). The before and after parts of operators (both primitives and macros) are also represented as rectangular arrays. The elements of these arrays can be pegs, holes, or don't-cares. Voids do not appear in operator representations since they are generalized to don't-cares. The before and after arrays of any given operator must have identical dimensions. Furthermore, don't-cares always appear in the same positions in the before and after arrays.

The legal move generator finds all legal instances for each operator in the current operator set. The binding information of an operator instance specifies its position and orientation (including possible reflection) relative to the board. Each operator is paired with every possible binding specification to obtain the set of candidate operator instances, and each candidate instance is then tested for legality. An operator instance is legal if its before part matches the board according to the position and orientation specified by its binding

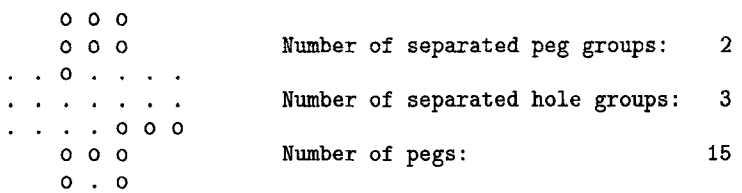

Figure 3. A board state whose evaluation function value is (–2, –3, –15).

information. The match with the board is just an element-by-element comparison of corresponding elements in the arrays, and is successful only if each of the individual elements match. Don't-cares match against anything in a board array, including voids.

The best-first search system applies legal operators to board positions to generate successor positions. Applying a legal operator instance consists simply of copying the after array of the operator into the part of the board array specified by the operator binding. Don't-cares are ignored during this copy process; i.e., nothing is changed at the corresponding board location.

3.3 The evaluation function for peg solitaire

The peg-solitaire version of MACLEARN uses the *component-hole-peg* metric. Briefly, this is a three-element vector whose elements specify the number of peg groups, the number of hole groups, and the number of pegs. Groups are defined here by horizontal and vertical adjacency, with voids being treated as holes. The additive inverse of each score is used, since in each case smaller absolute values are considered more desirable. The first criterion prefers states with fewer groups of pegs, the second prefers those with fewer groups of holes, and the third prefers fewer pegs. Figure 3 shows a sample state and its evaluation. Note that the voids in the lower right corner contribute a third hole group.

Evaluation vectors are ordered by lexicographic comparison. If the first elements of two vectors are unequal, those elements determine the order of the vectors, and all subsequent elements are ignored. If the first elements are equal, then the second elements are compared, and so on, until a difference is encountered. The first difference always determines the ordering of the vectors. If no difference is found, then the vectors are equal.

3.4 The macro-proposer for peg solitaire

In peg solitaire, MACLEARN invokes the macro proposer when the current node (the one being expanded) becomes a peak because one of its child nodes has a lower value. This is referred to as the *possible-peak* triggering condition. It contrasts with the more restrictive *selected-peak* condition used in earlier work (Iba, 1985), in which one proposes a macro only when the previous node is a peak; i.e., when the search was forced to take a downhill step

from the previous node. The possible-peak criterion is less restrictive and leads to the proposal of more macros, thus placing a greater burden on the filtering mechanisms, which must weed out the larger number of poor macros that are generated. The possible-peak condition was chosen for peg solitaire in order to provide a greater test of the filtering heuristics. Once invoked, the macro-proposer uses the peak-to-peak heuristic discussed in Section 2.3.1 to delimit a sequence of operator instances. This sequence is composed and abstracted by the procedure described below.

MACLEARN uses a domain-specific procedure to accomplish composition and abstraction in the peg-solitaire domain. This procedure runs much more efficiently than the more general procedures referenced in Section 2.3.2. The steps are as follows:

1. Find the smallest rectangular window that includes all locations referenced by any operator instance in the sequence being composed.
2. Create a dummy array with the dimensions of this window, and initialize it entirely with don't-cares.
3. Renumber or *relativize* the bindings of each operator instance so that they are relative to the dummy array rather than to the larger board.
4. Copy the after array of each (relativized) operator instance into the dummy array in the position and orientation that are specified by the relativized operator instance bindings. This occurs in the order in which the operator instances appear in the sequence. The result becomes the after part of the macro.
5. Transform (a copy of) the after array by making backwards moves in the reverse order of the sequence. A backwards move is accomplished by reversing the roles of the before and after arrays. The result becomes the before part of the macro.
6. Set the expansion of the macro-operator to be the relativized sequence of operator instances. This sequence transforms the before array into the after array.

The composition part of this procedure merges the individual before and after conditions. The resultant after array is a compilation of the individual after arrays, and the before array is a reverse compilation of the individual before arrays.

Generalization takes place in two ways. By narrowing the window of reference and using don't-cares for unreferenced locations within the window, the procedure only retains those conditions that are necessary for applying the operator sequence. The second form of generalization occurs across positions and orientations. This is supported explicitly by narrowing the window and implicitly by the way operators are applied (automatic translation, rotation, and reflection). Since macros can apply with different bindings than the original sequence from which they were generated, the relativization of bindings is necessary to facilitate correct expansion of macro instances at a later time.

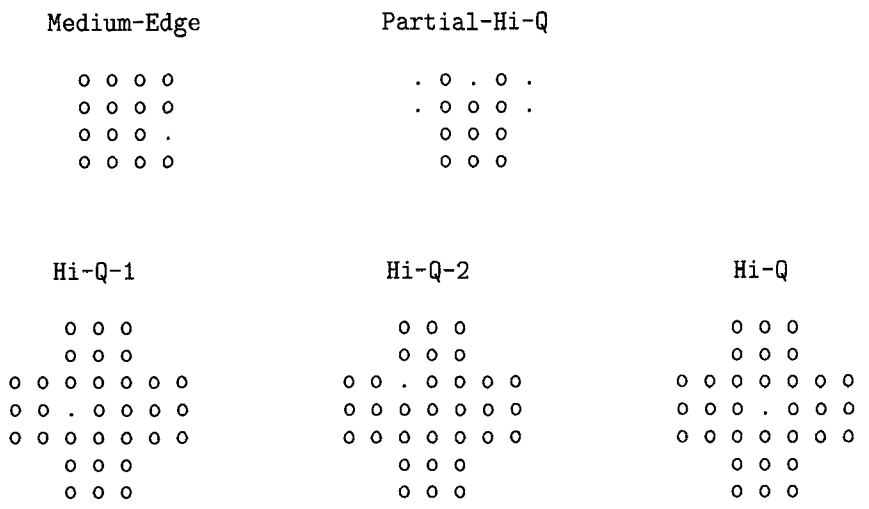

Figure 4. Peg-solitaire problems used in the experiments.

3.5 The static and dynamic filters in peg solitaire

The static filter for peg solitaire includes all three types of tests discussed in Section 2.4. In the redundancy check, two operators are considered equivalent if there is some rotation and reflection for which the before and after arrays correspond exactly. In all of the peg-solitaire experiments, the threshold for the length test was seven.

The domain-specific criterion used for peg solitaire is a *connectedness* test, which requires that the after side of a proposed macro have its pegs connected in just a single component. The motivation is that the evaluation function places highest emphasis on keeping the number of connected components of pegs to a minimum. The notion is that macros leaving their affected pegs in a single connected component will be more likely to contribute positively to such an evaluation function. The experiments reported in Section 3.6 seem to bear this out.

The dynamic filter applies the tests discussed in Section 2.5, filtering out any macro that never appears in a solution sequence. Such unused macros are removed from the operator set, but their definitions are retained.

3.6 Experiments with peg solitaire

In order to evaluate the behavior of MACLEARN in the peg-solitaire domain, a number of experiments were carried out. The first experiment compared problem-solving performance on individual test problems with and without the learning of macros. The second study explored the value of cumulative experi-

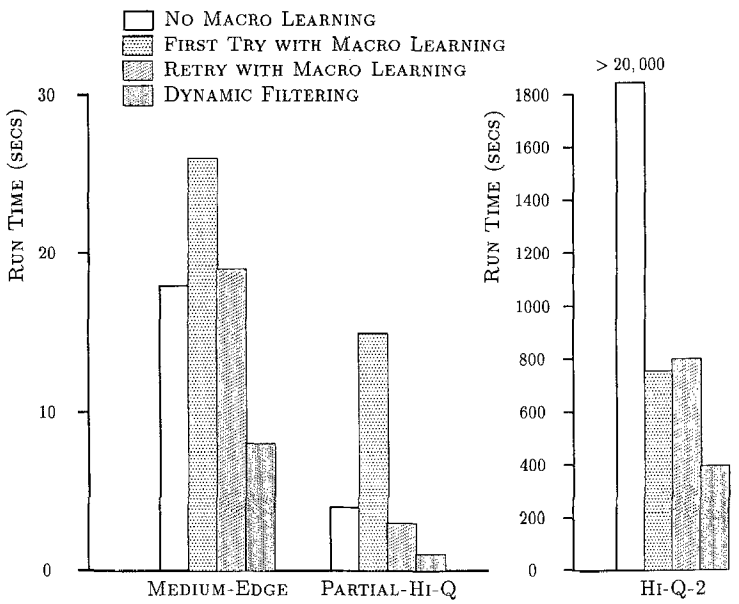

Figure 5. Run-time comparisons with and without macro learning (Experiment 1).

ence over a sequence of problem-solving trials. The third experiment compared the value of alternative static filtering conditions, and the fourth examined the value of dynamic filtering. The final experiment compared within-trial and post-trial learning.

The primary performance measures used to evaluate MACLEARN's problem-solving behavior were total run time and success at solving problems. Other data collected during the experiments include the number of nodes expanded, the number of operator instances considered, the number of macros proposed, the number of macros passed by the static filter, and the length of the solution sequence (in both primitive and macro steps). A derived statistic is the *average branching factor*, which is the total number of generated nodes divided by the total number of expanded nodes.

Figure 4 shows the initial states for the peg-solitaire problems used in the experiments. In each case, the goal is to reduce the board to a single peg. In Experiment 1 these problems were each attempted separately. In the remaining experiments the problems were presented in sequence (Medium-Edge, Partial-Hi-Q, Hi-Q-1, Hi-Q-2, Hi-Q) as a series of training problems, roughly graded from easier to more difficult.

All static filtering was done using the complete static filter (redundancy, length, and domain-specific tests), except for Experiment 3, in which the various static filtering tests were compared. When the length test was used in static filtering, the length threshold was set to seven. The domain-specific

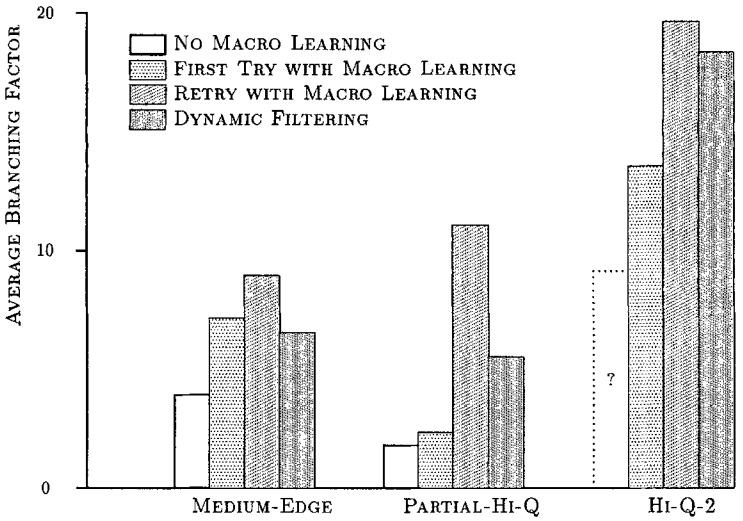

Figure 6. Average branching factors with and without macro learning (Experiment 1).

test for peg solitaire was the connectedness test described in Section 3.5. The default for macro learning was within-trial learning, except for Experiment 5, which compared post-trial and within-trial learning.

3.6.1 Experiment 1: Testing the value of macro learning

This experiment compared search performance *with* macro learning to the baseline case *without* macro learning. Each of the five peg-solitaire problems was attempted first without macro learning and then with macro learning. When the problem was successfully solved with macro learning, the set of learned macros was used to re-solve the problem. This allowed macros proposed in later stages of the first search to be available from the beginning of the search. It also gave all the operators a fair chance to appear in a solution before the dynamic filter was invoked. The final step in this experiment was to apply the dynamic filter to the operator set resulting from the retry attempt. The resulting filtered operator set was used to solve the problem one last time.

Two of the problems (Hi-Q-1 and Hi-Q) were not solved either with or without macros, so no results are reported for them. Of the remaining problems, Hi-Q-2 without learning ran for 20,000 seconds without finding a solution. All the other searches were successful, and the run-time measures appear in Figure 5. The main results are that macro learning led to initially increased run time for the easier problems (Medium-Edge and Partial-Hi-Q), but that for the harder Hi-Q-2 problem, learning led to a solution whereas none was found without macros. Even on the easier problems, the retry attempts led to run times comparable to those without macros, and the application of the dynamic filter led to significant run-time improvements, roughly halving the time spent on the retry attempt.

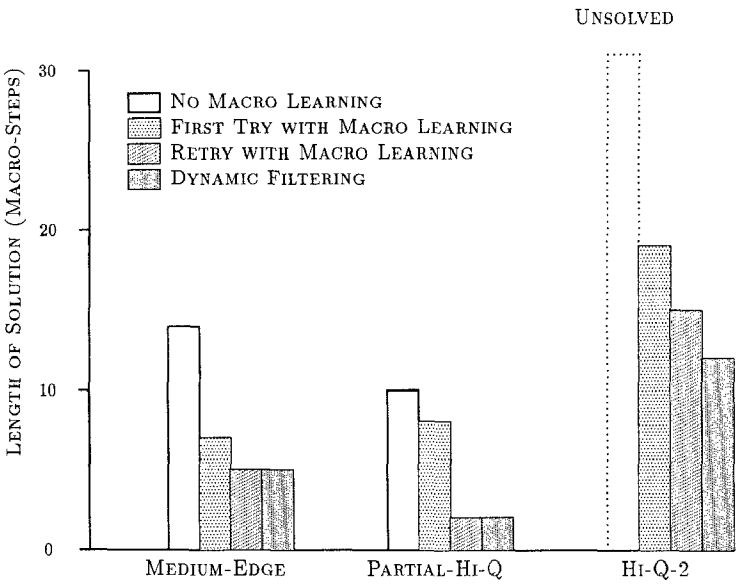

Figure 7. Lengths of solutions found with and without macro-operator learning (Experiment 1).

As discussed earlier, one view of macro learning is that it trades breadth against depth in the search tree, accepting an increase in breadth in order to achieve a reduction in depth. One measure of a tree's breadth is its average branching factor. Figure 6 graphs the average branching factors for each condition on the Medium-Edge, Partial-Hi-Q, and Hi-Q-2 problems. As expected, the branching factor increases as more macros are learned. The dynamic filter, by eliminating certain macros, leads to a decrease in branching factor. Figure 7 graphs the macro solution lengths for each condition on the same problems.[2] As expected, the acquisition of macros consistently leads to shorter solutions.

3.6.2 Experiment 2: Cumulative learning – training and transfer

Although Experiment 1 suggests that macro learning can lead to improvement over search without macros, the method was not sufficient to solve the Hi-Q-1 and Hi-Q problems. One approach to these harder problems is to take advantage of macros learned while solving simpler problems. Experiment 2 explores the issues of training and transfer by treating the peg-solitaire problems as a five-step training sequence, with macros learned on earlier problems being available to help solve later problems. During the first pass through the training sequence, no dynamic filtering was invoked. At the end of this pass, the dynamic filter was invoked on the final operator set, causing 15 of the 26 macros to be discarded. The remaining operators are shown in Figure 8.

[2]Note that although MACLEARN was unable to solve Hi-Q-2 without macros, all solutions of this problem take exactly 31 primitive steps.

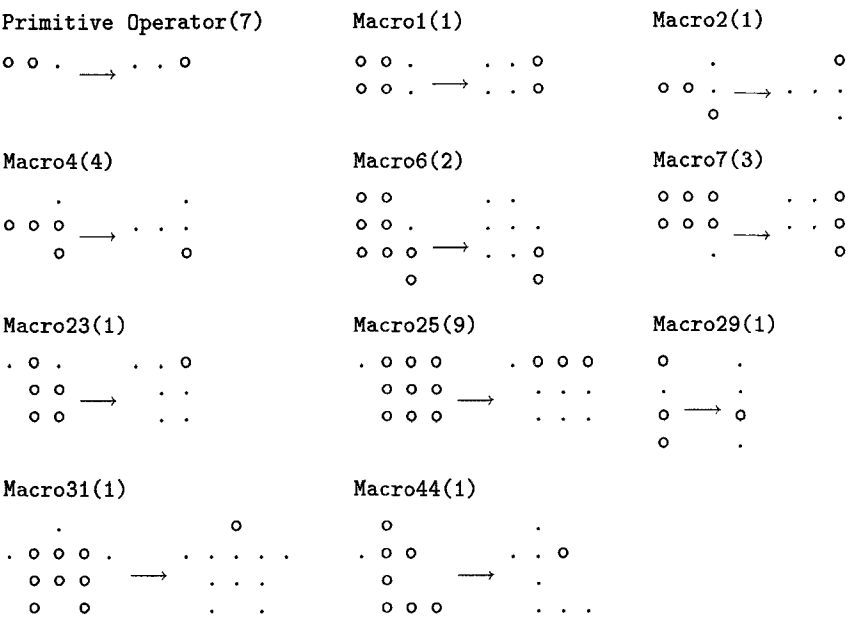

Figure 8. Operators left after dynamic filtering. Numbers in parentheses indicate how many times operators have appeared in solutions. Don't-cares have been omitted in the operator descriptions.

Starting with this filtered set of 11 operators, a second pass was made through the training sequence. This time, dynamic filtering was invoked between each problem trial in the sequence.

Figure 9 shows the run-time comparisons on each problem with no macro learning, the first pass through the training sequence (macro learning with experience accumulating over trials), and the second pass through the training sequence (using the dynamic filter before each trial). The main result is that cumulative learning led to successful solutions for all the problems, including Hi-Q-1, Hi-Q-2, and Hi-Q, which went unsolved without learning. A somewhat negative result was that the transfer of macros from Medium-Edge to Partial-Hi-Q actually slowed down the search. The run time of 179 seconds when experience was transfered is much greater than the 15 second run time of Experiment 1, in which Partial-Hi-Q was solved using macro learning but without any transferred experience. On the second pass, dynamic filtering led to a dramatic reduction in run time for the Partial-Hi-Q problem, and to roughly a halving of run time on the remaining problems. On all problems, the run time after dynamic filtering was less than when no macros were learned at all. Figure 10 shows the solutions to the Hi-Q-1, Hi-Q-2, and Hi-Q problems that MACLEARN found during the dynamic filtering pass through the training sequence. The searches for Hi-Q-1 and Hi-Q-2 were completely *monotonic*, meaning that no backtracking was required and the solution was found in a

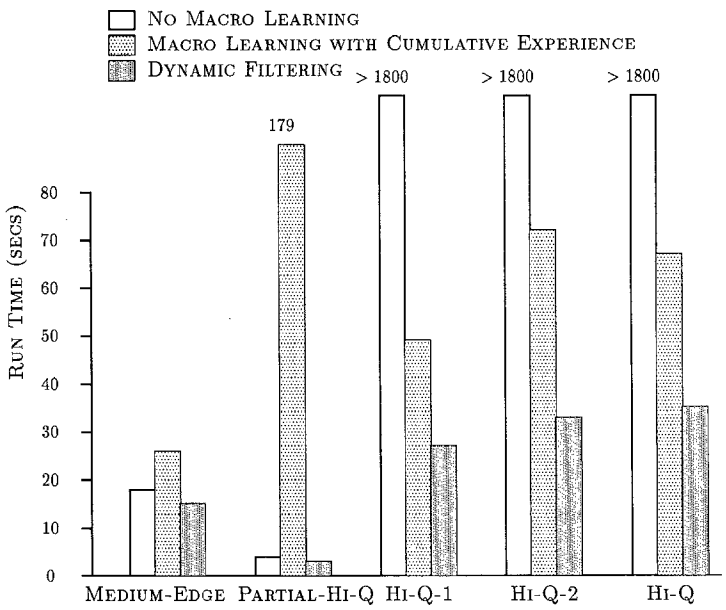

Figure 9. Run times for learning with cumulative experience (Experiment 2).

straight-line fashion. The Hi-Q search was nearly monotonic, having just a single instance of backtracking; eight nodes were expanded to find the final seven-step solution.

3.6.3 Experiment 3: Testing the static filtering heuristics

Section 3.5 described the three static filtering tests employed in MACLEARN. In order to assess the value of each of these tests, the following five combinations of tests were evaluated as static filters:

1. No tests
2. Redundancy test
3. Redundancy + domain-dependent (connectedness test)
4. Redundancy + expanded length
5. Redundancy + expanded length + domain-dependent (connectedness test)

This list includes all combinations of the tests except length, domain, and length + domain-dependent tests. These were omitted to simplify the study, since the redundancy test was so obviously desirable. Note that in the "no tests" condition, all macros pass the static filter by default. In this experiment, each combination of tests was evaluated by using it as the static filter while doing cumulative learning on the entire five-problem training sequence.

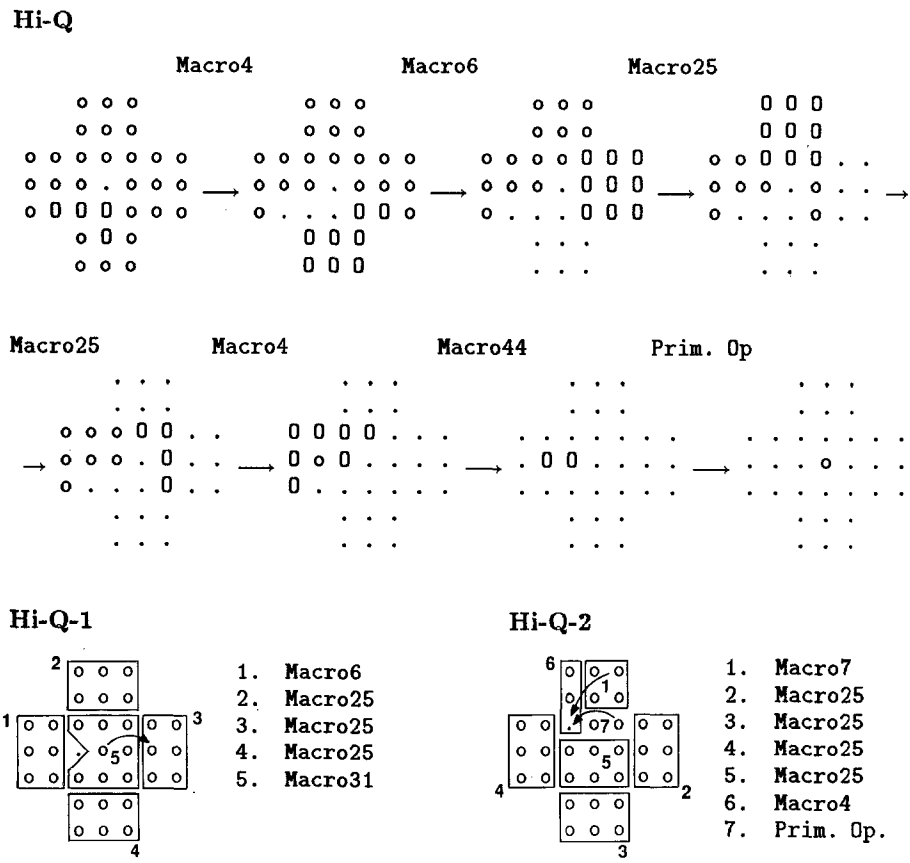

Figure 10. Solutions found with dynamically filtered macros.

Figure 11 shows the *cumulative* run-time data[3] over the training sequence for each of the static filters. The data show a consistent improvement as additional tests are included in the static filter, with the complete static filter performing best of all. The Hi-Q problem was only solved with the third and fifth combinations. In the other conditions (1, 2, and 4), an examination of the Hi-Q search tree revealed that the search started off on a bad track because of some unhelpful macros, which increased the branching factor without usefully advancing the search. In filters 3 and 5 the domain-dependent test excluded these macros, leading to a successful solution for Hi-Q.

Figure 12 shows how the various static filters selectively limited growth of the operator set. Again the complete static filter provided the greatest selectivity, leading to the slowest growth. The slower growth of the operator set with stronger filtering conditions accounts for the improvements in run time. An examination of search statistics shows that very often the same number of

[3]See Minton (1988) for a discussion of this dependent measure.

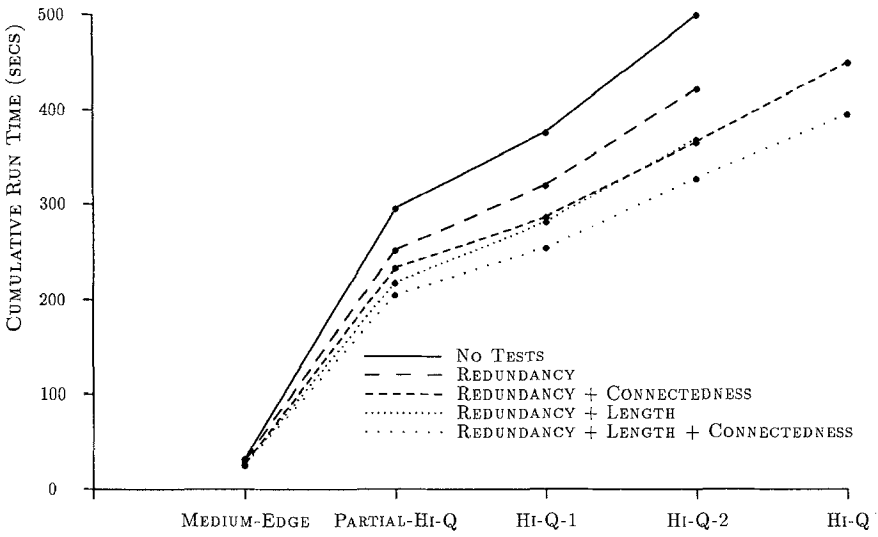

Figure 11. Cumulative run-time comparison of different static filter tests (Experiment 3).

nodes were expanded in corresponding searches, but the reduced branching factor resulted in faster search.

3.6.4 Experiment 4: Dynamic filtering

The first two experiments gave some indication of the value of dynamic filtering. In order to further assess dynamic filtering, a separate experiment was conducted in which MACLEARN was first trained on the five-problem training sequence, and then the training sequence was solved again. The purpose of this was to give each operator ample opportunity to demonstrate its usefulness by appearing in solution sequences. A final pass was made through the training sequence, invoking the dynamic filter before each problem trial. The design of this experiment combines the "retry before dynamic filtering" approach of Experiment 1 with cumulative learning over a training sequence, as in Experiments 2 and 3.

Figure 13 shows the cumulative run-time graphs for the three passes through the training sequence. There is a noticeable improvement for the retry pass, which is due almost entirely to the dramatic reduction in solution time for the Partial-Hi-Q problem. No additional macros were learned during the retry phase, but two new macros were used in solutions and thus earned the right to survive the dynamic filter. The dynamic filter removed 13 operators, compared with 15 at the comparable point of Experiment 2. Due to the consequent reduction in branching factor, run times during the dynamic filtering phase were further reduced on all problems.

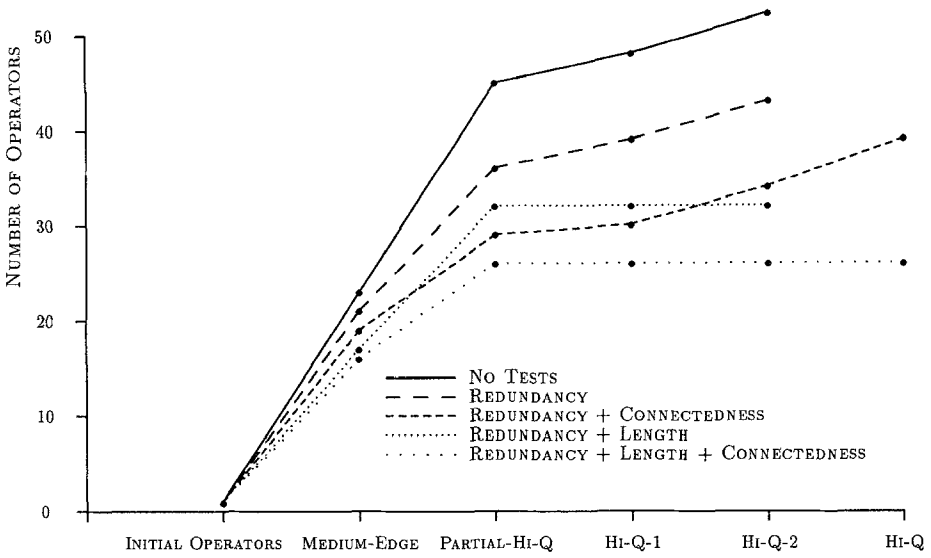

Figure 12. Growth of the operator set for different static filters (Experiment 3).

3.6.5 Experiment 5: Within-trial and post-trial learning

In order to test the value of within-trial learning, the training sequence of five problems was solved using post-trial learning. In this paradigm, macros were proposed as before, but they were not available for use until a subsequent problem was worked on. The cumulative run-time graphs of Figure 14 compare post-trial and within-trial learning on these problems. As the figure shows, within-trial learning resulted in consistently lower run times. This suggests that the value of having macros immediately available for use is greater than the disadvantage of increased branching factor.

Note that within-trial learning also resulted in slower growth of the operator set, as illustrated in Figure 15. This can be explained by the fact that the search trees were consistently smaller in the within-trial condition, and so there were fewer opportunities for additional macros to be proposed. Also note that all the growth occurred on the first two problems, with the operator set remaining constant thereafter.

An interesting feature of within-trial learning is that a single problem-solving trial can lead MACLEARN to define macros in terms of other macros, taking advantage of the hierarchical nature of macro definition. Figure 16 shows the definition ' macro hierarchy generated on the Medium-Edge problem using within-trial learning and the complete static filter. The specifications for some of these macros (numbers 1, 2, 4, 6, and 7) are shown in Figure 8, and it is easy to see how each is defined in terms of simpler macros in accordance with the hierarchy of Figure 16.

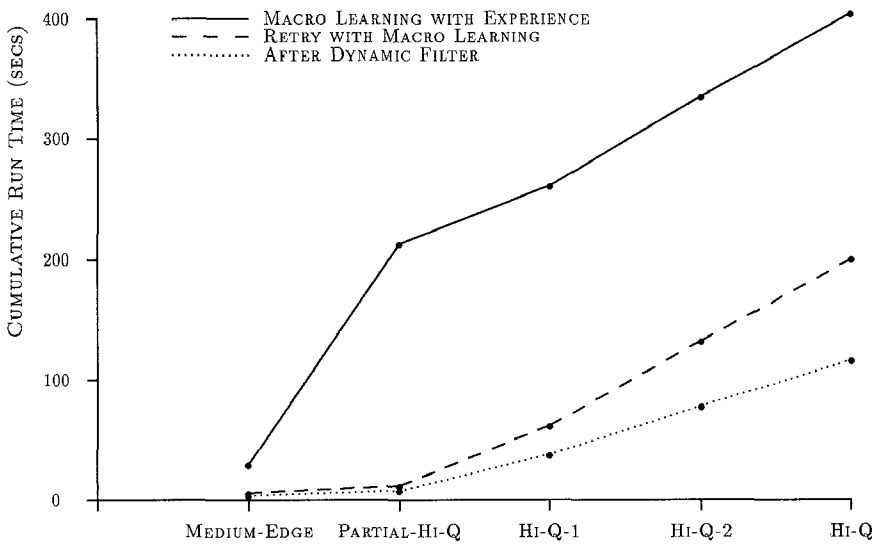

Figure 13. Effect of the dynamic filter on cumulative run time (Experiment 4).

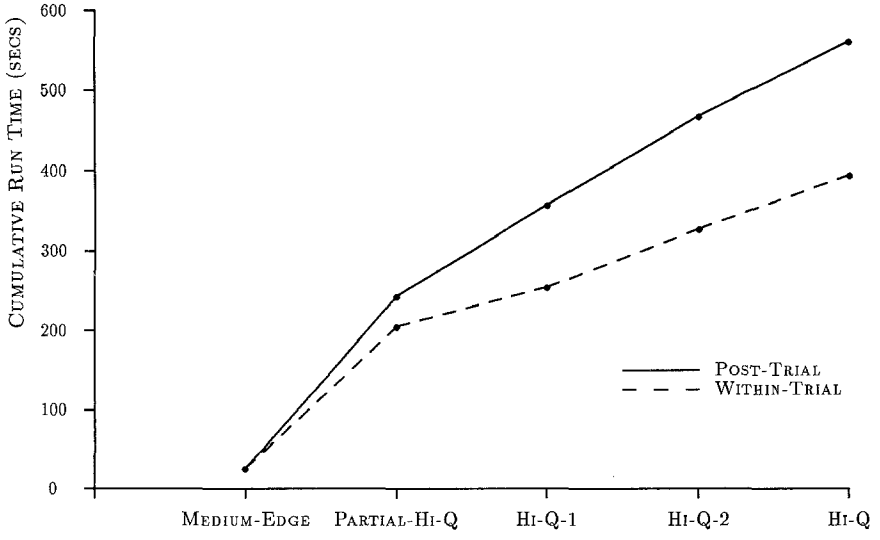

Figure 14. Cumulative run-time comparison of within-trial and post-trial learning (Experiment 5).

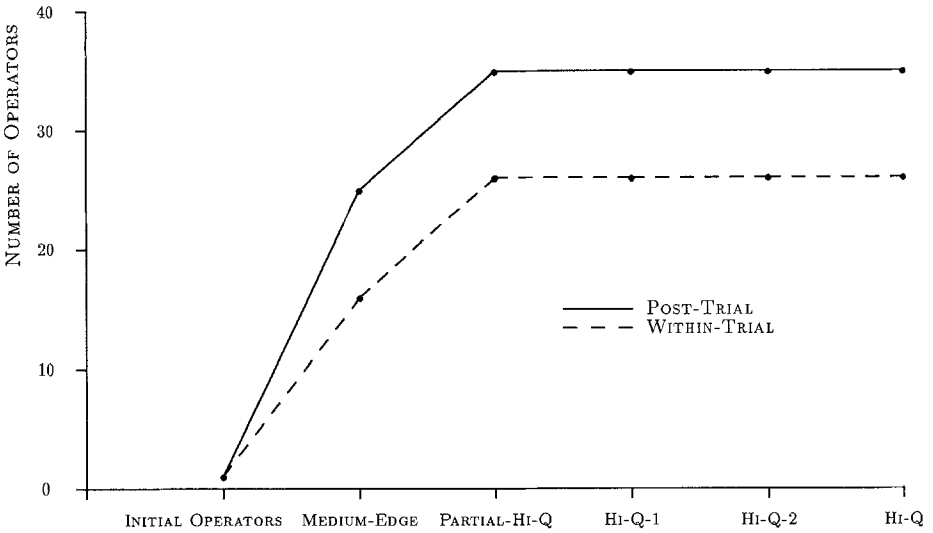

Figure 15. Growth of the operator set in within-trial and post-trial learning (Experiment 5).

4. Application to tile sliding

In order to test the generality of the methods and heuristics embodied in MACLEARN, it was applied to a second domain called *tile sliding*, which is a generalization of the well-known "Eight puzzle" and "Fifteen puzzle". In tile-sliding problems a set of numbered tiles are arranged in a rectangular grid, such that exactly one grid space remains blank. If a tile is adjacent to the blank, it can be moved by sliding it into the blank space. Typically the tiles start out scrambled, and the puzzle is to unscramble them into some canonical pattern (e.g., numerically increasing order). Both Korf (1985a, 1985b), and Laird, Rosenbloom, and Newell (1986) have studied macro-learning in this domain.

Figure 17 shows the set of problems used in the tile-sliding experiments reported in Section 4.4. These initial states were chosen at random, but were then fixed so that comparative studies could be undertaken. The problems are ordered by difficulty (Simple, Eight puzzle, Fifteen puzzle, Twenty-four puzzle), and the most successful learning occurred when the problems were solved in that order, with the accumulation of macros over the whole sequence of trials. The following sections describe the application of MACLEARN to the tile-sliding tasks, along with the results of experiments in this domain.

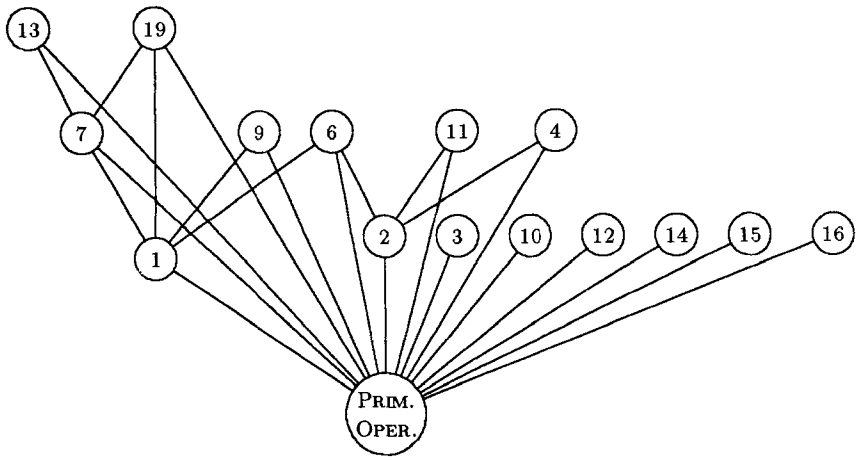

Figure 16. Hierarchy of macro definitions resulting from within-trial solving of Medium-Edge with complete static filter. Macros are represented only by their number to save space.

4.1 Representation and matching in tile sliding

As with peg solitaire, states in tile sliding are represented by rectangular arrays. Instead of pegs and holes, the elements in tile-sliding arrays are numbered tiles, with each tile having a unique number. The array also contains exactly one special element that represents the blank. Rectangular arrays are used to represent the before and after parts of primitive operators and macros.

In peg solitaire there is no distinction between one peg and another, and so it is sufficient to represent primitive operators and macros in terms of before and after arrays whose operators are simply pegs and holes. In tile sliding the identities of tiles are distinct, and so it is necessary to keep track of their individual positions. This is done by introducing *pattern variables* into the representation of both primitive operators and macros. Variables bind to specific tiles, and the movements of tiles are reflected in the movements of variables between the before and after arrays of the operators. Tiles whose positions remain unchanged by an operator are treated as *don't-cares* (signified by "_" in the figures). Figure 18 shows the primitive operator for tile sliding and some useful macros.

Except for the addition of variables that match to any tile (but not the blank), the generation of legal operator instances is the same as in peg solitaire (Section 3.2). Also as in peg solitaire, the mechanism for applying operator instances automatically handles translations, rotations, and reflections. In the application of an operator instance, when the after part is being copied into the board state, each variable in the after part is replaced by the tile of the board state which matches that variable in the before part.

Simple Puzzle

3	4	
2	5	1

Eight Puzzle

5	7	3
4		2
6	8	1

Fifteen Puzzle

5	7	14	10
4	13	12	3
9		2	6
8	15	11	1

Twenty-Four Puzzle

	10	5	20	9
12	7	8	4	1
2	22	21	6	16
11	19	17	3	13
24	15	18	14	23

Figure 17. Initial states for tile-sliding problems. For each problem, the goal is to arrange the tiles in numerical order with the blank in the lower right.

4.2 The evaluation function for tile sliding

The evaluation function used in tile sliding takes two arguments, a current state and a goal state, each represented as a two-dimensional rectangular array. It returns a vector with the following three components:

1. The number of consectutive tiles already in their goal locations. Counting starts in the upper left-hand corner, and proceeds row by row until the first mismatch is encountered.
2. Minus one times the Manhattan distance[4] between the current and goal positions for the next tile to be placed; i.e., the tile belonging in the first grid position not matching the goal.
3. Minus one times the Manhattan distance of the blank from the next tile to be placed.

Informally, these three components encode a set of subgoals to guide the search. The first component of this evaluation function corresponds to the subgoal of placing each tile in its correct position in row-major order. The second component corresponds to the subgoal of getting the target tile (the next tile to be placed) closer to its goal position. In order to be able to move the target tile, the third component represents a subgoal of getting the blank adjacent to that tile. As an illustration of the evaluation function, the initial state for the Simple puzzle in Figure 17 evaluates to $(0, -3, -1)$. As in peg solitaire, the vectors returned by the evaluation function are compared lexicographically.

[4]The *Manhattan* distance between two points is just the number of grid steps it takes to go from one to the other. Formally, $MD(x_1, y_1, x_2, y_2) = |x_1 - x_2| + |y_1 - y_2|$.

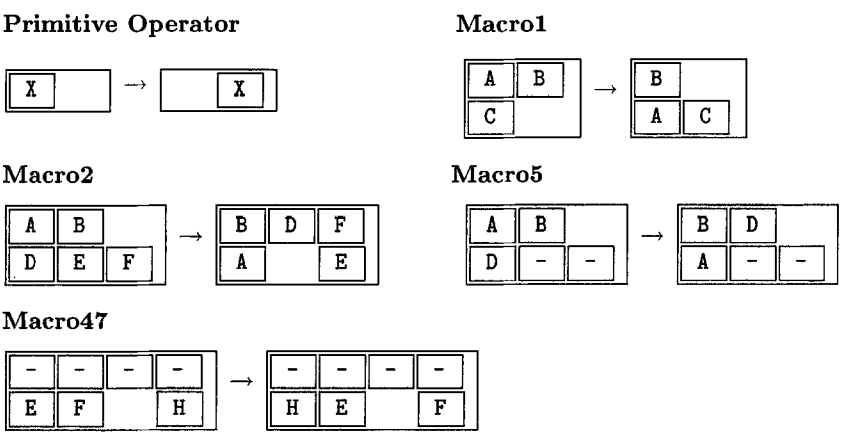

Figure 18. The primitive operator and some useful macros for tile sliding. Letters bind to any tile; the symbol "–" denotes a tile unchanged by a macro.

4.3 Proposing and filtering macros in tile sliding

In tile sliding, macros were again proposed according to the peak-to-peak heuristic. In order to compare the possible-peak and selected-peak conditions (see Section 3.4) for determining when to propose macros, each was tried under identical conditions.

The processes of composition and abstraction are almost identical to those described for peg solitaire. Composition proceeds using fixed identities of tiles, then abstraction replaces the fixed tiles with pattern variables so the final macro can match different configurations of tiles. Strictly speaking, don't-cares are not needed in tile-sliding operators, since they can always be represented by a variable which is left in the same position. MACLEARN introduces them merely to improve readability.

The static filter used only the redundancy and length tests. No domain dependent test was used in tile sliding. Redundancy checking was done as in peg solitaire, with the additional feature of checking for correspondence of pattern variables. Throughout the tile-sliding experiments, the length test of the static filter used a threshold of 30 for the maximum allowable expanded length of macros.

4.4 Experiments with tile sliding

In Experiments 6 and 7, MACLEARN was run on each of the tile-sliding problems in Figure 17 under the following three conditions:

1. No macro learning (just problem solving using the primitive operator);
2. Macro learning without prior experience; and
3. Macro learning with cumulative experience from previous problems.

Experiment 6 tested the possible-peak proposal method for each of these con-

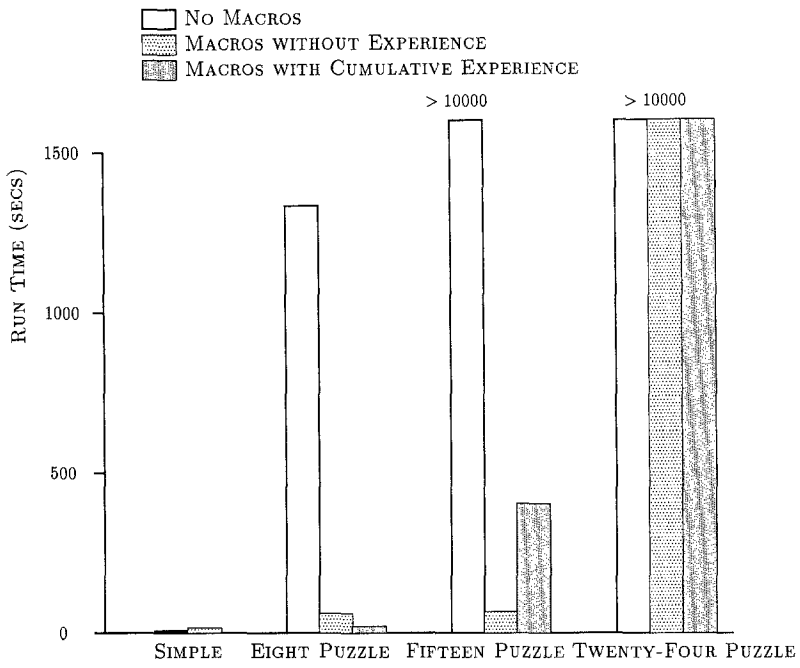

Figure 19. Run-time comparisons with and without macro learning in the tile-sliding domain using the possible-peak macro proposer (Experiment 6).

ditions. Without macro learning, this version of MACLEARN solved only the Simple and Eight-puzzle problems. With macro learning but no prior experience, it was also able to solve the Fifteen puzzle, but it still failed on the Twenty-four puzzle. Solving the problems as a training sequence with cumulative learning did not provide much help.

Figure 19 shows the run-time results for this experiment. Learning macros led to consistently lower run times than problem solving without macros. Prior experience on simpler training problems led to a slight reduction in the run time on the Eight puzzle compared with macro learning without prior experience, but it led to a significant increase in the run time for the Fifteen puzzle. Experiments with other starting states for the Fifteen puzzle suggest that the short run time encountered in this experiment (for the "no prior experience" condition) was serendipitous and not very representative. None of the conditions led to successful solutions for the Twenty-four puzzle within the time limit of 10,000 seconds.

Experiment 7 repeated the format of Experiment 6 but used the selected-peak method rather than the possible-peak technique. Note that since the selected-peak method defers proposal of macros until a lower-valued node is actually selected as the best to expand, it is more restrictive and results in fewer proposals. This could be viewed as an additional form of selectivity, over and above the static filter.

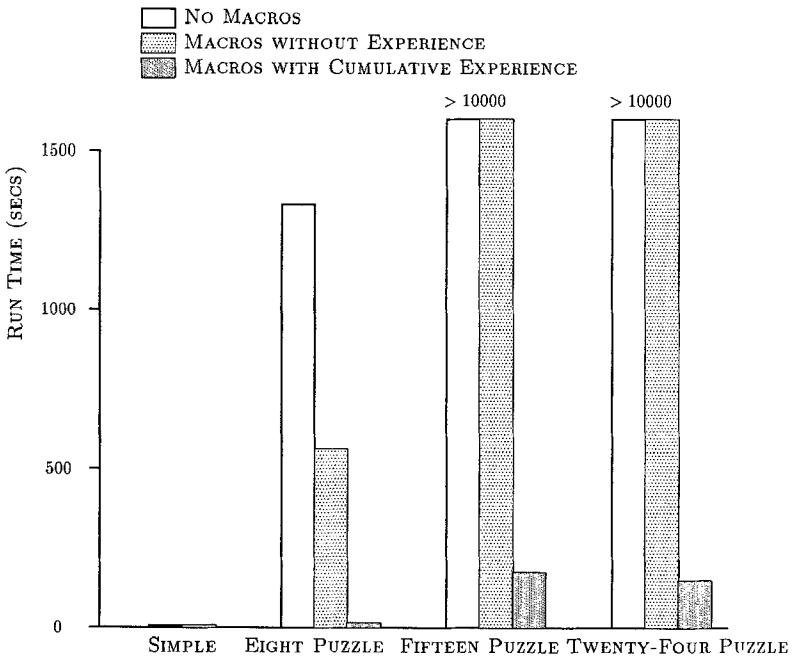

Figure 20. Run-time comparisons with and without macro learning in the tile-sliding domain using the selected-peak macro proposer (Experiment 7).

Figure 20 shows the run-time data for this experiment. The Simple puzzle and Eight puzzle were solved in all three conditions. The more difficult Fifteen puzzle and Twenty-four puzzle were solved only in the case of macro discovery with prior experience from earlier trials. The failure to solve Fifteen puzzle without prior experience is more representative than the success in the previous experiment.

A comparison of the first two conditions (no learning and learning without prior experience) illustrates the value of macro discovery within the course of a single problem-solving trial. The data indicate that macro learning leads to consistently greater search efficiency. The only disadvantage seems to be a tendency towards longer (expanded) solutions in the macro conditions. Nevertheless, the macro solutions are shorter (in macro steps) and can be viewed as a higher-level (chunked) representation of these solutions.

Macro learning with experience accumulating over problem trials provided the most dramatic success. Only in this case was MACLEARN able to solve the more difficult problems. In fact, by the time the system had solved the Fifteen puzzle, it had discovered a complete set of macros and no additional ones were needed to solve the Twenty-four puzzle. Note that the time to solve the Twenty-four puzzle was actually less than for the Fifteen puzzle, since the solution was found in monotonic fashion, with no backtracking. The number of nodes expanded was equal to the macro-length of the solution sequence.

The 11 operators in the final operator set which accomplished this are shown in Figure 21. These macros were also sufficient to solve twenty randomly generated variations of the Twenty-four puzzle (with the same size board but different starting configurations). In each of these additional problems, the goal was still to get the tiles arranged in numerically increasing order with the blank in the lower right.

A comparison of the results of Experiments 6 and 7 indicates that selected-peak did better on these problems. This is particularly true for the cumulative learning case,in which the Twenty-four puzzle was solved so successfully and in which the learned operator set was able to solve additional random problems.

5. Discussion

MACLEARN represents a combination of empirical and analytic techniques for discovery of macro-operators. The empirical component arises from the interplay of the macro proposer with the static and dynamic filters, and can be viewed as a generate-and-test search applied to the space of macro-operators. The macro-proposer does the generation, whereas the static and dynamic filters carry out the testing. The ultimate test is whether the behavior of the performance element is improved. This can also be viewed as a credit-assignment problem, in which macros receive credit or blame for their influence on system performance. The dynamic filter eventually removes those macro-elements that do not receive sufficient credit over time. The analytic component lies in the methods by which macros are composed, abstracted, and defined in terms of before and after conditions. These analytic methods are essentially equivalent to the weakest pre-condition analysis that is commonly used in explanation-based approaches.

5.1 Advantages and disadvantages of the approach

The approach to macro discovery embodied in MACLEARN has several advantages. Learning and performance are integrated, proceeding hand in hand so that macros learned in the early stages of problem solving may aid the later stages. The approach to representation (closure under composition of sequences into pre- and post-conditions) enables the learning of new macros in terms of previously learned macros. The representation also has the important advantage that it lets the system use macros without reference to their actual expansion. Macros function in a kind of planning capacity, suppressing unnecessary details until a solution plan is found. The generalization of macros supports transfer to larger and more difficult problems.

These advantages were born out in the experiments on both peg solitaire and tile sliding. Macro-learning generally outperformed solving without macros, especially on more difficult problems. Within-trial learning proved very beneficial, leading to shorter run times and slower growth of the operator set. Static and dynamic filtering were useful in selectively limiting the growth of the operator set, resulting in greater search efficiency. Cumulative learning over a sequence of increasingly more difficult training problems also proved to be worthwhile.

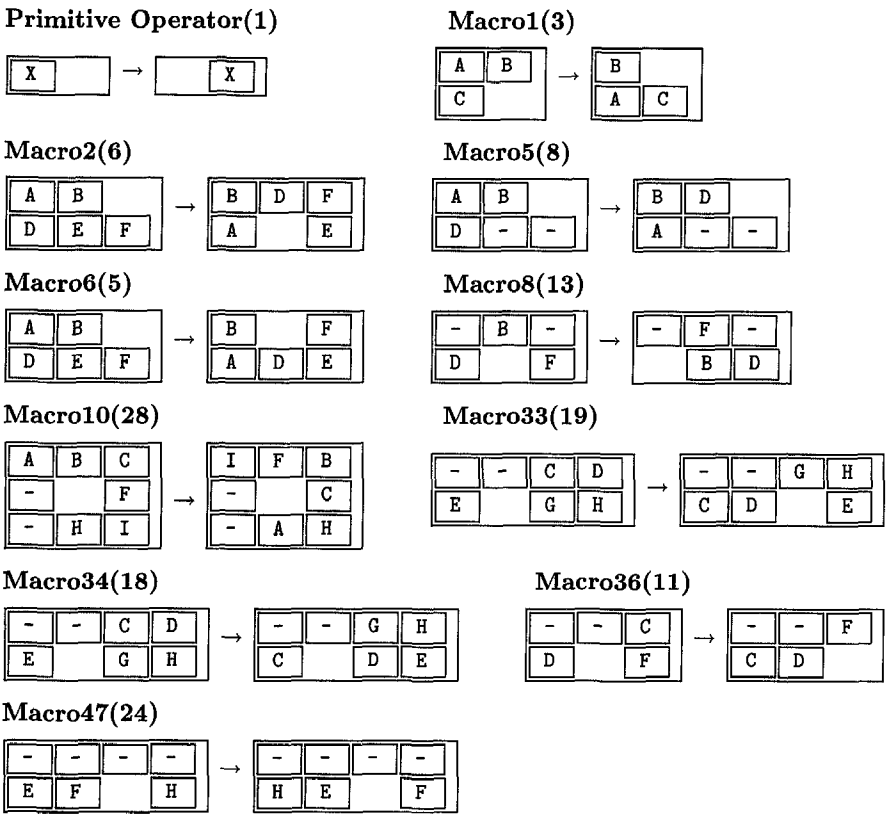

Figure 21. Full tile-sliding operator set after training on first three problems. Numbers in parentheses indicate the expanded length of each macro. Letters bind to any tile; the symbol "–" denotes a tile unchanged by a macro.

The principal difficulty encountered in these applications of MACLEARN was due to the choice of best-first search as the performance element. Since best-first search applies every available operator when expanding a node, the branching factor of the search increases as more macros are learned. A second difficulty is that the search is not goal-directed, except in the sense that goals are implicit in the evaluation function. Thus the system's behavior exhibits a lack of focus, illustrated in what might be termed a *mesa phenomenon*; in some cases, the program spends considerable time exhaustively searching a large number of nodes, all having equal values, before considering any nodes with lower values. This partly explains the importance of training on simpler examples. For smaller problems, the combinatorics of searching mesas are more manageable. This lets the program reach the point where lower-valued nodes must be selected and propose macros that may be useful in similar (but combinatorially more complex) situations.

Both these issues – increased branching factor and lack of search focus – can be addressed by moving to a goal-based performance element such as means-ends analysis (Newell & Simon, 1963). The introduction of explicit subgoals should provide a focus of attention, while the table of connections should reduce the branching factor of the search, insuring that operators are applied only when they are relevant.

5.2 Generality of the learning methods

Several issues arise with respect to the generality of MACLEARN. The first involves the range of problems to which the system can be applied within the domain of problem solving. The current implementation, which uses best-first search, should handle any problem that can be specified in terms of initial state, goal condition, and operators, provided these operators can be composed to form macros represented in the same format as primitive operators. An evaluation function must also be supplied. This paper has focused on the domains of peg solitaire and tile sliding, but MACLEARN has also been applied to Tower of Hanoi, slide jump, and telecommunications network control.

A second issue involves the value of the system's learning component when used in conjunction with different performance elements. Goal-directed search has already been mentioned as one important alternative. With a new performance system, it may be necessary to employ different heuristics for proposing and filtering macros. Further work is needed to explore alternative performance systems and to determine heuristics that are effective with them.

The learning mechanisms of MACLEARN can be abstracted and viewed as embodying a general empirical approach to the task of learning by composition, which is a type of chunking. These mechanisms might be incorporated into any system that uses *elements* to generate behavior, provided those elements can be composed to define *macro-elements*. The learning task is then to discover a set of macro-elements that improves performance. The notion of a macro-element proposer combined with static and dynamic filters may prove to be a powerful one.

Problem solving is one domain that fits within this general framework. A search system generates performance and the composable elements are the operators. Other domains that exhibit a similar compositional structure include theorem proving, automatic programming, design, and natural language parsing. In theorem proving, the basic elements are axioms and composition involves assembling axioms (and theorems) into new theorems. A theorem prover serves as the performance element. Once a theorem is proved, it can function as though it were an axiom, as far as the theorem prover is concerned (O'Rorke, 1987). The other domains fit within this compositional framework in a similar fashion. Future work should explore the extent to which MACLEARN can be extended to apply across these domains.

5.3 Related work

It is useful to compare MACLEARN to related work on the discovery of macro-operators. Fikes, Hart, and Nilsson (1972) carried out some of the earliest work in this area, applying their STRIPS problem solver to the domain of robot

planning. Macros (which they called 'Macrops') were represented in terms of triangle tables, which abstracted robot plans to their most general pre- and post-conditions. In this respect, MACLEARN's representation is similar. One difference is that triangle tables simultaneously abstract all sub-sequences of a plan (sequence of steps), whereas the current system's macros abstract only entire sequences. The latter approach leads to a simpler representation and a reduced branching factor relative to triangle tables, but it does not capture as many possibilities. Another difference is that STRIPS was a goal-directed problem solver, whereas MACLEARN uses best-first search.

Korf (1982, 1983, 1985a, 1985b) has also done important work on macro learning. The learning in his program takes place during an initial phase of explicit search for macros to fill in a table of connections, whereas MACLEARN's discovery of macros is integrated (interleaved) with problem solving. As noted in Section 3, Korf's approach requires operator decomposability, which is not a limitation for MACLEARN. Another difference is that his macros are simple sequences of primitive operators that apply only in particular circumstances, whereas the ones used here are abstracted so as to apply in more general settings. Moreover, Korf's macros do not build on others to form a hierarchy. One can view his approach as learning a large number of rather special-purpose macros, whereas MACLEARN aims for a smaller number of more general-purpose macros. However, Korf's method has the important advantage of being algorithmic; one can prove that a macro table exists under certain assumptions, and that the learning algorithm will eventually find this table. This contrasts with the heuristic approach taken in MACLEARN.

Another appealing aspect of Korf's method is that once a macro table has been learned, it can be used to solve all variants of the puzzle having the same size and goal state. However, if the goal changes, a new macro table is required. In contrast, MACLEARN's macros are not tied to specific goals. Another limitation is that Korf's macros do not generalize naturally to related problems of different size. For example, the macros that solve the $2 \times 2 \times 2$ Rubik's Cube do not help solve the $3 \times 3 \times 3$ version of the cube, nor do macros learned for the Eight puzzle help solve the Fifteen puzzle. The present approach overcomes this difficulty through the process of composing and abstracting macros, which lets the macros apply in new circumstances, such as larger-sized problems with similar structure. This occurred in both the tile-sliding and peg-solitaire domains, where macros learned from simpler problems proved useful in solving harder problems.

Laird, Rosenbloom, and Newell (1986) have taken yet another approach in their work on learning by chunking in SOAR. Both the current system and SOAR use within-trial learning, so that macros from early stages of search can lead to improvements later on. One difference is that SOAR can use goal-directed search, whereas MACLEARN uses best-first search. Another difference is that SOAR explicitly chunks control knowledge having to do with subgoals and methods, whereas MACLEARN's control knowledge (the evaluation function) is separate from the representation of macros. MACLEARN merely discovers macros that one *can* apply, but makes no suggestions about *when* to apply them. This latter question is certainly important, but it can be treated

as a separate issue, rather than having it bound up with the macro discovery process. There is a suggestive analogy with the discovery of theorems in mathematics, which are similar in many respects to the macros discovered here. A theorem states that it is *possible* to draw a conclusion from premises, but it makes no commitment as to when the conclusion *should* be drawn. A mathematician certainly develops control knowledge that suggests when to use a theorem, but this knowledge can stand separately from the statement of the theorem itself. A final difference between the two systems is that SOAR does not address the issues of selectivity in what should be learned and what should be forgotten. These issues are directly addressed in MACLEARN by means of the static and dynamic filters. SOAR has been applied to a number of tasks, including the Eight puzzle and Tic-Tac-Toe.

Minton (1985) has explored another interesting approach to macro discovery in his MORRIS system, which uses a combination goal-directed and best-first search. MORRIS and MACLEARN are similar in that they both use heuristics that explicitly address the need for selectivity. Minton's system learns macros of two types – S-macros and T-macros. The S-macros (script macros) are created from common sub-sequences shared by different solution paths. MORRIS places a limit on the number of allowable S-macros. When the number of S-macros exceeds this limit, the system determines which ones have been used least often, and removes them one by one until the number of macros is within the bound. This stratagem is similar to dynamic filtering. MORRIS is also selective in forming T-macros (trick macros), which are only proposed when a solution path contains a step or steps that run counter to the evaluation function. This is similar to the peak-to-peak heuristic, but his system applies it only to solution paths, rather than to partial search paths. Thus MORRIS does not learn within a single trial, but only from examination of solution sequences once a problem has been solved. Minton's system does not seem to have a mechanism for selectively forgetting T-macros once they have been learned. Tests of MORRIS were conducted in the domain of robot planning.

In other work, Cheng and Carbonell (1986) have studied the discovery of iterative and conditional macros in their FERMI system. This is an important and novel extension of previous macro-learning work. FERMI employs a representation that is an extended form of production rules, in which right-hand sides of productions may have special rule sets called *buckets*. A bucket is essentially a subroutine of rules that is repeatedly called until none of its rules applies, thus achieving iteration. There can be more than one bucket on the right-hand side of a rule, leading to conditionality. As in SOAR, the FERMI system incorporates control knowledge as part of its learned macros, rather than keeping control knowledge separate. Also, since the system does not compile macros into pre- and post-conditions, it can only predict the effect of a macro by actually executing it in expanded form. Learning in FERMI involves analyzing solution traces, so the system learns only after it has found a complete solution, rather than during a problem trial. This contrasts with both SOAR and MACLEARN, which learn within a single trial. However, one should be able to extend FERMI so it proposes macros when it satisfies subgoals, and thus achieve within-trial learning.

5.4 Summary and conclusions

This paper has described a heuristic approach to the discovery of macro-operators. The program MACLEARN, which implements the approach, was tested in the two problem-solving domains of peg solitaire and tile sliding. The tests demonstrated successful learning, leading to the solution of some difficult problems. Experiments with several selective filtering mechanisms showed improvements of overall search efficiency. Within-trial learning proved to be beneficial in limiting growth of the operator set and in discovering solutions more efficiently.

Perhaps the greatest problem with the current work is that the best-first search applies every available operator at each node of the search tree. Thus, as more macros are learned, the search process slows down because of the increased branching factor. A goal-directed search might remedy this defect, since it would selectively consider only operators that are relevant to the current subgoal.

Although the basic framework has proved useful for discovering macros in problem solving, it should also apply to other problem categories. Future work should examine domains such as theorem proving and automatic programming, in order to test its full generality.

Acknowledgements

Chuck Anderson, Jaime Carbonell, Wayne Iba, Pat Langley, Bernard Silver, Rich Sutton, Oliver Selfridge, and John Vittal all provided numerous valuable suggestions for improving drafts of this paper. Special thanks go to Oliver Selfridge for his support and encouragement, as well as his valuable assistance in revising and formatting; and to Pat Langley, who gave a careful reading to each of several drafts, making numerous valuable suggestions for improvements, and provided continuing support and encouragement throughout.

References

Cheng, P., & Carbonell, J. G. (1986). The FERMI system: Inducing iterative macro-operators from experience. *Proceedings of the Fifth National Conference on Artificial Intelligence* (pp. 490–495). Philadelphia, PA: Morgan Kaufmann.

Dawson, C., & Siklóssy, L. (1977). The role of preprocessing in problem solving systems. *Proceedings of the Fifth International Joint Conference on Artificial Intelligence* (pp. 465–471). Cambridge, MA: Morgan Kaufmann.

Fikes, R., Hart, P., & Nilsson, N. (1972). Learning and executing generalized robot plans. *Artificial Intelligence*, *3*, 251–288.

Iba, G. A. (1985). Learning by discovering macros in puzzle solving. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* (pp. 640–642). Los Angeles, CA: Morgan Kaufmann.

Iba, G. A. (1986). Learning by composition. In T. M. Mitchell, J. G. Carbonell, & R. S. Michalski (Eds.), *Machine learning: A guide to current research.* Boston, MA: Kluwer.

Korf, R. E. (1982). A program that learns to solve Rubik's Cube. *Proceedings of the National Conference on Artificial Intelligence* (pp. 164–167). Pittsburgh, PA: Morgan Kaufmann.

Korf, R. E. (1983). Operator decomposability: A new type of problem structure. *Proceedings of the National Conference on Artificial Intelligence* (pp. 206–209). Washington, DC: Morgan Kaufmann.

Korf, R. E. (1985a). Macro-operators: A weak method for learning. *Artificial Intelligence*, *26*, 35–77.

Korf, R. E. (1985b). *Learning to solve problems by searching for macro-operators*. Boston, MA: Pitman.

Laird, J. E., Rosenbloom, P. S., & Newell, A. (1986). Chunking in SOAR: The anatomy of a general learning mechanism. *Machine Learning*, *1*, 11–46.

Lewis, C. (1987). Composition of productions. In D. Klahr, P. Langley, & R. Neches (Eds.), *Production system models of learning and development*. Cambridge, MA: MIT Press.

Minton, S. (1985). Selectively generalizing plans for problem-solving. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* (pp. 596–599). Los Angeles, CA: Morgan Kaufmann.

Minton, S. (1988). Quantitative results concerning the utility of explanation-based learning. *Proceedings of the Seventh National Conference on Artificial Intelligence* (pp. 564–569). St. Paul, MN: Morgan Kaufmann.

Mitchell, T. M., Keller, R. M., & Kedar-Cabelli, S. T. (1986). Explanation-based generalization: A unifying view. *Machine Learning*, *1*, 47–80.

Neves, D. M., & Anderson, J. R. (1981). Knowledge compilation: Mechanisms for the automatization of cognitive skills. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition*. Hillsdale, NJ: Lawrence Erlbaum.

Newell, A., & Simon, H. A. (1963). GPS, A program that simulates human thought. In E. A. Feigenbaum & J. Feldman (Eds.), *Computers and thought*. New York: McGraw-Hill.

O'Rorke, P. (1987). LT revisited: Experimental results of applying explanation-based learning to the logic of Principia Mathematica. *Proceedings of the Fourth International Workshop on Machine Learning* (pp. 148–159). Irvine, CA: Morgan Kaufmann.

Vere, S. A. (1978). Inductive learning of relational productions. In D. Waterman & F. Hayes-Roth (Eds.), *Pattern-directed inference systems*. New York: Academic Press.