A Selective Macro-learning Algorithm and its Application to the $N \times N$ Sliding-Tile Puzzle

Shaul Markovitch and Lev Finkelshtein
Computer Science Department
Technion, Haifa 32000
Israel
shaulm@cs.technion.ac.il

Abstract

One of the most common mechanisms used for speeding up problem solvers is macro-learning. Several methods for acquiring macros have been devised. The major problem with these methods is the vast number of macros that are available for learning. To make macro learning useful, a program must be selective in acquiring and utilizing macros. This paper describes a general method for selective acquisition of macros. Solvable training problems are generated in increased difficulty. The only macros acquired are those that take the problem solver out of a local minimum to a better state. The utility of the method is demonstrated in several domains, including the domain of $N \times N$ sliding-tile puzzles. After learning on small puzzles, the system is able to solve puzzles of any size.

1 Introduction

The goal of a learning system is to improve the potential performance of a problem solver, with respect to some given criterion. The learning system does so by using its experience to generate knowledge for use by the problem solver. A major portion of the machine learning community is concerned with improving the classification accuracy of classifiers based on classified examples. A smaller part of the community is concerned with improving the speed of search programs based on problem solving experience. This type of learning is commonly termed speedup learning.

One of the most common methods of speedup learning is the acquisition of macro-operators [10, 7, 2, 12, 14, 16, 22]. Given the traditional definition of a search space with a set of states and a set of basic operators that connect states, a macro-operator is a finite sequence of basic operators. Macro-operators are acquired during problem solving and are used in the same manner as basic operators. The process of acquiring macros is relatively simple. A system needs only to solve training problems and pass the search tree to the acquisition procedure, which, in turn, can add any sub-path of the tree to its macro knowledge-base. However, the acquired macros carry costs as well as benefits. When the costs outweigh the benefits, we face a phenomenon called the utility problem [17, 15].

To obtain a macro set with high utility, a learning program must be selective. The goal of this research is to demonstrate that a simple macro-learning technique combined with the right selection mechanisms can lead to a powerful and general, yet simple, speedup learning algorithm.

Section 2 defines selective macro-learning and identifies the choices to be made when designing a macro-learning system. Section 3 describes the general Micro-Hillary\(^1\) algorithm for learning macro-operators. Section 4 describes experiments with Micro-Hillary in various domains, mainly the eight-puzzle domain. Section 5 describes an extension of Micro-Hillary that is able to learn a family of domains that are differentiated by a single numeric parameter and describes the application of the parameterized algorithm to the family of $N \times N$ puzzle domains. Finally, Section 6 concludes.

\(^1\)Micro-Hillary is a simplified version of a system called Hillary[3]. The program was named after Sir Edmund Hillary who was the first human to reach the summit of Mount Everest. We did not notice that the name Hillary has been already taken by Iba, Wogulis and Langley[8]. We apologize for that.
2 Background: Selective macro-learning

Let \( S \) be a finite set of states. Let \( O \) be a set of operators where each operator \( o \in O \) is a function \( o : S \rightarrow S \cup \{ \emptyset \} \). If \( o(s) = \emptyset \), we say that \( o(s) \) is undefined. A problem is a pair of states \( \langle s_i, s_g \rangle \), \( s_i \in S \) is called the initial state and \( s_g \in S \) is called the goal state. A solution is a sequence of operators \( \langle o_1, \ldots, o_k \rangle \) such that \( o_k \ldots o_1 (s_i) = s_g \).

A search node is a pair \( \langle (o_1, \ldots, o_k), s \rangle \) where \( o_i \in O \) and \( s \in S \). Given a problem \( p = \langle s_i, s_g \rangle \) and a set of operators \( O \), a search is a sequence of search nodes defined as follows:

1. \( \langle \emptyset, s_i \rangle \) is a search.
2. Let \( C \) be a search. Let \( \langle (o_1, \ldots, o_k), s \rangle \in C \) and \( o \in O \). Then \( C \| \langle (o_1, \ldots, o_k), o \rangle, o(s) \rangle \) is also a search.

A search \( C = \langle c_1, \ldots, c_n \rangle \) is complete\(^2\) if \( c_n = \langle \ldots, s_g \rangle \). A search strategy \( \varphi \) is a function that takes a problem \( p \), a search \( C \) and a set of operators \( O \) and decides which operator to apply to which node, i.e., \( \varphi(p, C, O) = \langle c_i, o \rangle \) where \( o \in O \) and \( c_i \in C \). A strategy is acyclic if it never creates a cycle. Given a solvable problem \( p = \langle s_i, s_g \rangle \) and a set of operators \( O \), an acyclic search strategy \( \varphi \) defines a complete search \( C_{\varphi}(\langle s_i, s_g \rangle, O) = \langle c_1, \ldots, c_n \rangle \) whereas if \( p = \langle c_1, \ldots, c_n, O \rangle = \langle O, s \rangle \) then \( c_{i+1} = c_i \| \langle O,s \rangle, o(s) \rangle \).

A macro-operator is a sequence of operators \( m = \langle o_1, \ldots, o_m \rangle \). The application of the macro-operator \( m \) to a state \( s \) in \( S \) is defined as:

\[
m(s) = \begin{cases} o_m (\ldots o_1 (s) \ldots) & \forall i \leq m [o_i (\ldots o_1 (s) \ldots) \neq \emptyset] \\ \emptyset & \text{otherwise} \end{cases}
\]

Given a solvable problem \( p \), a strategy \( \varphi \) and a set of operators \( O \), we define the cost of solving \( p \) using \( \varphi \) and \( O \) as the length of \( C_{\varphi}(p, O) \). We define the utility of a set of macros \( M \) with respect to a problem \( p \), a search strategy \( \varphi \) and a set of operators \( O \) as

\[
U_{p, \varphi, O}(M) = \text{Cost}(C_{\varphi}(p, O)) - \text{Cost}(C_{\varphi}(p, O \cup M))
\]

When the problems are drawn from some fixed distribution \( D \), we eliminate \( p \) from the above equation and use expectation values for a problem randomly drawn from \( D \). In general, the utility of knowledge depends on the criterion used for evaluating the performance of the problem solver [15]. Equation 1 assumes a speedup learner, i.e., a learner whose goal is to increase the speed of solving problems. Note that the utility of macros depends on the particular strategy used for search. Macro learning has a negative utility when used for optimizing search [13].

Most of the macro-learning systems perform learning by experimentation [18]. The program solves training problems and acquires sequences of operators applied during the search. Given a set of operators \( O \), a distribution of problems \( D \) and a strategy \( \varphi \), the goal of a macro-learning system is to acquire a set of macro-operators \( M \), such that \( U_{\varphi, O}(M) \) is positive, and as high as possible. However, it is quite feasible that the utility of \( M \) is negative due to the increase in the branching factor as a result of the added macros. In order to increase the utility of macro-learning, we should consider the following issues:

- What training examples give rise to a macro set with high utility?
- What sub-paths of the search graph should be converted into macros?
- What macros should not be added to an existing macro set?
- What macros should be deleted from an existing macro set?
- What subset of the macro set should be used for a particular problem?

Markovitch and Scott [15] introduced the information filtering framework which identifies five logical types of selection processes in learning systems: selective experience, selective attention, selective acquisition, selective retention and selective utilization. The framework views learning programs as information processing systems where information flows from the experience space through some attention mechanism, through an acquisition procedure to the knowledge base and finally from there to the problem solver. The five filters are defined with respect to their logical location within the information flow.

Let \( E \) be the set of possible training examples. In the context of macro-learning, the information filters can be specified more precisely:

\(^{2}\)For simplicity, we do not deal with a search that may not terminate after getting to the goal state (such as A*).
The architecture of an off-line macro-learning system is shown in Figure 1. Markovitch and Scott argue that the art of building successful learning systems is many times in selecting the right combination of filters. A careful examination of existing macro-learning systems, reveals that those containing sophisticated filtering mechanisms performed the most successful learning. Maclearn [7], for example, employs three filtering mechanisms that increase the utility of its macro knowledge-base: an attention filter, an acquisition filter and a retention filter. The most interesting of the three is the peak-to-peak attention filter that selects only subsequences which connect states whose heuristic values are peaks in the solution path. The idea behind this approach is to learn how to get out of local minima. The problem with this approach is that the learned route leads into another local minimum. In the following section we present an alternative attention filter that avoids this fault.
Figure 2: An example of a sequence of heuristic values of states along a solution path to a fifteen-puzzle problem.

3 Micro-Hillary

In this section we discuss Micro-Hillary—a particular instantiation of the architecture described above that is very simple, and yet is powerful enough to perform efficient learning in a large class of problem domains.

Micro-Hillary, like all macro-learners, works with *satisficing* [24] search programs, i.e., programs whose goal is to find solutions as fast as possible regardless the length of the solution found. Such programs typically use heuristic functions which serve as preference predicates—the search strategy prefers states with a lower value. Many of these heuristic functions order states reasonably well except in a small number of local minima. For example, Figure 2 illustrates the sequence of heuristic values of states along the solution-path of a fifteen-puzzle problem. The basic idea of Micro-Hillary is to acquire macro-operators for escaping from local minima. Micro-Hillary learns macros that lead from a local minimum to a state with a better heuristic value. Thus, while Iba uses the *peak-to-peak* filtering method, we can call our method *peak-to-better* filtering. Let $h$ be a heuristic function that estimates the distance of a state to the goal, let $\epsilon = (s_{init}, s_g)$ be a training problem, and let $T = C_{\varphi}(\epsilon, O)$ be a search that solves $\epsilon$. Let $s_j = o_i \cdot s_{j-1} \ldots o_i \cdot (s_{init}))$. The peak-to-better attention filter can be defined as:

$$
\langle o_{i_j}, \ldots, o_{i_2} \rangle \in f_{att}(T) \Rightarrow \\forall o \in O[h(s_{j_1}, s_g) \leq h[o(s_{j_1}), s_g)] \wedge \forall y[j_1 < y < j_2 \Rightarrow h(s_{j_1}, s_g) \leq h(s_y, s_g)] \wedge h(s_{j_2}, s_g) < h(s_{j_1}, s_g)
$$

When a sufficient number of such macros are acquired, the search program may be able to advance toward the goal state without ever being trapped in local minima.

3.1 The Micro-Hillary algorithm

Micro-Hillary works by generating training problems and solving them. The goal of Micro-Hillary is to detect local minima and to learn how to escape from them. The best way to detect local minima is to use a search method that is susceptible to local minima such as hill-climbing. When trapped in a local minimum, we search for a way out to a state with a better heuristic value than the local minimum and record this escape route as a macro-operator. Micro-Hillary quits when it does not encounter local minima for a given duration (we call such a condition *quiescence*.) Figure 3 lists the Micro-Hillary algorithm. To complete the algorithm we need to specify a method for generating training problems and a method for escaping from local minima.
procedure Micro-Hillary(BasicOps,h)
    Macros ← {}
    loop until quiescence
        ⟨si,sg⟩ ← GenerateTrainingProblem()
        s ← si
        loop until s = sg
            sbest ← s
            loop for o ∈ BasicOps ∪ Macros
                if h(o(s), sg) < h(sbest, sg) then sbest ← o(s)
            if sbest = s then
                O ← FindEscapeRoute(s, sg)
                Macros ← Macros ∪ {O}
            else
                s ← sbest
            Return Macros

FindEscapeRoute(s, sg): A procedure that finds a sequence of operators O = o1,...,ok such that h(o_k(...o_1(s, sg))) < h(s, sg).

Figure 3: The Micro-Hillary algorithm

3.2 Selective experience: Generating solvable problems in increasing level of difficulty

The basic architecture of a macro-learning system requires a mechanism for generating training problems and a method for filtering them (or, alternatively, a method for ordering them). For Micro-Hillary we have developed a problem-generation technique with the following properties:

- Training problems are solvable. When solving training problems we want to be sure that no state is a global minimum except the goal state. One way to achieve this is to train with solvable problems.

- Training problems are generated in increasing order of difficulty. The reason for that is to save learning resources by acquiring as much knowledge as possible using easy problems.

The optimal way for implementing such a problem-generation methodology is to require a domain-specific generator for each domain. However, to make Micro-Hillary more general, we supply a default problem generator that can be used in many domains. Our default problem generator assumes the following:

- The specification of the problem domain includes an algorithm for generating random goal states (or, alternatively, there is only one goal state).

- All the operators are reversible: ∀s ∈ S ∀o ∈ O ∃o' ∈ O [o'(o(s)) = s]. Alternatively, the specification of the domain includes a list of reverse operators.

Micro-Hillary generates a problem by applying a random sequence of operators to a randomly generated goal state. The problems are generated in order of increasing difficulty by increasing the length of the random sequence.

3.3 Escaping from local minima

There are several possible ways to escape from a local minimum. The safest way is to employ an exhaustive search method such as Breadth First, or Iterative Deepening. However, an exhaustive search consumes too many computation resources for any non-trivial escape task. We can try to make some use of the heuristic function by calling A* or IDA*. However, both methods use significant efforts in making sure that the route to the goal state is optimal.
We are looking for a search method that finds solutions quickly if the heuristic function is sufficiently accurate, but still guarantees to find a solution even if the heuristic function is poor. We came up with a search method that we call Iterative Limited BFS (ILB). The basic procedure is Limited BFS which gets an initial state, a goal predicate, a breadth limit $B$ and a depth limit $D$. It performs a BFS search to depth $D$. Whenever the number of nodes kept in memory exceeds $B$, it deletes the node with the worst heuristic value. This search is different from Beam-search. Limited BFS keeps the $B$ best nodes of the current depth while Beam-search keeps the best $B$ nodes regardless of their depth. When $B \to \infty$, Beam-search becomes Best-first while Limited BFS becomes BFS. Iterative Limited BFS calls Limited BFS iteratively until either a full BFS is performed or a solution is found. The breadth limit $B$ is increased for each iteration and is set to the value $k + b^i$ where $k$ is some constant, $b$ is the branching factor and $i$ is the iteration number. This scheme guarantees that ILB performs full BFS after $D$ iterations.

The particular escape method is not an inherent part of Micro-Hillary. In domains where the heuristic function does not give indication regarding the direction of the escape route, Iterative Limited BFS may reduce to full BFS and require too much memory. In such cases, using Iterative Deepening as an escape algorithm may be more appropriate.

One important issue to consider when designing an escape procedure is whether acquired macros are used during the escape search. In the experiments performed we found that it is more beneficial to search for an escape route using only basic operators. However, it is quite possible that in some domains using macros for the escape search is beneficial.

### 3.4 Using Micro-Hillary for a specific domain

To use Micro-Hillary for learning in a specific domain, the following domain-specific information should be supplied:

1. A function that generates a random goal state.
2. A set of operators that take a state and return either another state or $\emptyset$.
3. A heuristic function that estimates the distance between two states.

A Common Lisp version of the basic Micro-Hillary algorithm is listed in the appendix.

### 4 Experimenting with Micro-Hillary

To test the effectiveness of the Micro-Hillary algorithm, we experimented with it in various domains. Most of the experiments were done in the eight-puzzle domain. The basic operators in this domain are $\text{Up}$, $\text{Down}$, $\text{Left}$ and $\text{Right}$, indicating the direction that the empty tile "moves". The heuristic function is similar to the one used by Iba$^7$. The function contains three factors that are ordered in lexicographical order: the number of tiles that are not ordered yet in a row-by-row order, the Manhattan distance of the first tile that is not in place to its destination, and the Manhattan distance of that tile from the empty square. This function essentially tells the problem solver to try and order the tiles row by row$^3$. We start by describing the experimental methodology used, and continue with a description of the experiments performed.

#### 4.1 Experimental Methodology

Most of the experiments described here consist of a learning phase and a testing phase. The training problems are generated by Micro-Hillary. We let every learning phase run until Micro-Hillary decides to stop due to quiescence. Since the problems are generated by random sequences, we repeat each learning session 100 times. We test each resulted macro set by allowing the problem solver to use the macros for solving a set of 100 test problems. A random test-problem is generated in the same manner as a training problem. The only difference is the length of the random sequence of operators applied to the goal state. For testing we used random sequences of length 1,000,000 to ensure that the problems will be hard enough.

---

$^3$ Section 5 specifies the exact definition of this function for the general $N \times N$ case.
4.1.1 Dependent variables

The goal of a learning system is to improve the potential performance of a problem solver with respect to a certain criterion. Hence, the principle dependent variables are those associated with the performance of the problem solver:

**Problem solving speed:** Micro-Hillary is a speedup learner. Its main goal is to increase the speed of the problem solver. The most obvious candidate for measuring problem solving speed is CPU time spent during search. However, such a measurement is effected too much by irrelevant factors such as hardware, software and programming quality. A common alternative measurement for search programs is the number of nodes expanded. Nevertheless, this measurement might be misleading in the context of macro-learning since the branching factor increases when acquiring macros. Also, it is more expensive to apply a macro-operator since it involves several applications of basic operators. Therefore, the principle dependent variable that we use to measure the performance of the problem solver is the number of applications of a basic operator to a state.

**Solution quality:** Macro-learning is not a suitable technique when the macros are used by an optimizing search procedure[13]. However, we are still interested in the quality of the solution obtained. We measure the quality of the solution by the ratio between the solution length obtained by Micro-Hillary and the length of the optimal solution. For domains where computing the shortest solution is too expensive, we use a lower bound on the solution cost to compute an upper bound on the ratio.

In addition we measure various aspects of the learning process. Dependent variables associated with the learning process are:

**Total learning resources:** The amount of resources consumed during the learning process until the learning system decides that it has learned enough. The common way for measuring learning resources is the number of training problems. The problem with this dependent variable is that it ignores the time invested in the search, and the time invested in generating training problems. Since the basic operation that is used both in search and in problem generation is the application of a basic operator to a state, we use the total number of operator applications as the basic measurement for the learning resources consumed. Note that the quiescence parameter effects the learning resources significantly. A higher value for this parameter increases the amount of resources consumed, but also decreases the likelihood that the problem solver will encounter a local minimum during testing.

**Number of macros:** The total number of macros acquired during the learning session.

**Length of macros:** The mean and maximal length of a macro.

1.2 Independent variables

There are many independent variables that effect the performance of the learning system. For the experiments described here we look at the following variables:

**Heuristic-function:** In addition to the row-by-row heuristic, we have tried the known sum-of-Manhattan-distances heuristic, and two variations of the row-by-row heuristic: the reduction heuristic which leads to placing of tiles in the first row, then the last column, then the second row, then the second to last column etc., and the spiral heuristic which leads to placing the tiles in a spiral order from outside in.

**Domain:** Most of the experiments described here were performed in the eight-puzzle domain. However, we have tried Micro-Hillary on several other domains:

- **15-puzzle:** 4 × 4 sliding tile puzzle.
- **10-cannibals:** The famous cannibals and missionaries problem. To make it non-trivial, instead of using 3 cannibals and 3 missionaries, we use 10 cannibals and 10 missionaries.
- **10-stones:** A puzzle that appeared in Nilsson’s book [19]. Instead of using 3 black stones and 3 white stones we use 5 black stones and 5 white stones.
- **5-Hanoi:** The famous Towers of Hanoi problem with 5 rings.
- **Grid:** A grid of 50 × 50 with random parallel walls inserted to make the Manhattan-distance heuristic inaccurate.
4.2 Experiment 1: Learning to solve problems in the eight-puzzle domain.

<table>
<thead>
<tr>
<th>Op applications</th>
<th>CPU seconds</th>
<th>Training problems</th>
<th>Macros learned</th>
<th>Maximal macro length</th>
</tr>
</thead>
<tbody>
<tr>
<td>266872 (154712)</td>
<td>34.7 (9.8)</td>
<td>66.3 (12.9)</td>
<td>10.39 (0.8)</td>
<td>11 (8)</td>
</tr>
</tbody>
</table>

Table 1: The learning resources for the eight-puzzle domain averaged over 100 learning sessions. The numbers in parenthesis are standard deviations.

The basic experiment tests the ability of Micro-Hillary to learn the eight-puzzle domain. Table 1 shows the learning resources averaged over the 100 learning sessions. We also tested the utility of the learned knowledge by comparing the performance of the problem solver that uses the learned macros to the following problem solvers:

1. Micro-Hillary without the learned knowledge and with learning turned off.

Table 2 shows the results of this experiment.

<table>
<thead>
<tr>
<th>Search Method</th>
<th>Op Applications</th>
<th>CPU seconds</th>
<th>Solution Length</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>A*</td>
<td>8342</td>
<td>32</td>
<td>22.5</td>
<td>1</td>
</tr>
<tr>
<td>Best First [Manhattan]</td>
<td>1826</td>
<td>4.46</td>
<td>127.3</td>
<td>5.66</td>
</tr>
<tr>
<td>Micro-Hillary before learning</td>
<td>16031</td>
<td>10.8</td>
<td>46.8</td>
<td>2.1</td>
</tr>
<tr>
<td>Micro-Hillary after learning</td>
<td>212</td>
<td>0.065</td>
<td>49.4</td>
<td>2.2</td>
</tr>
</tbody>
</table>

Table 2: The utility of the learned macros. The resources required for solving the testing set compared with various non-learning problem solvers.

Micro-Hillary solves problems much faster than the other algorithms. It is 40 times faster than A* with solutions that are twice as long as those found by A*. It is 9 times faster than Best-First with solutions that are 3 times shorter than those found by Best-First.

4.3 Experiment 2: The effect of the heuristic function on learning

The Micro-Hillary algorithm is based upon the availability of a “generally good” heuristic function to start with. The heuristic function should be able to lead the search program towards the goal, except in a small number of local minima where exhaustive search and learning take place. The next experiment tests the effect of the heuristic function on the performance of Micro-Hillary. We have conducted 100 learning sessions and testing sessions with each of the heuristic functions described above. Table 3 shows the results obtained. It is obvious that the performance of Micro-Hillary with the sum-of-Manhattan-distance heuristic is much worse than with the other heuristic functions.

One possible explanation is the difference in the number of alternative solution paths for each problem. The first three heuristic functions significantly restrict the structure of a solution to a problem – a rigid order of placing the tiles is required. Therefore the set of possible solutions is small and it is likely that similar patterns appear while solving different problems. The Manhattan heuristic, on the other hand, accepts any order of tile placement, allowing a much larger set of alternative solutions. Therefore, the probability that a macro learned while solving one problem will be useful for solving another problem is low. This experiment points out a major weakness of the Micro-Hillary algorithm: its sensitivity to the heuristic function used.
In the previous section we showed how Micro-Hillary can be used to solve the eight-puzzle and the fifteen-puzzle domains. Is it possible to use Micro-Hillary to solve the general $N \times N$ sliding-tile puzzle? The eight-puzzle and the fifteen-puzzle have always been favorite problems among mathematicians and among AI researchers [6, 11, 20, 23]. Ratner [20] proved that finding the shortest path between two $N \times N$ puzzle states is NP-complete. AI researchers have used the sliding-tile domain as a test bed for heuristic search methods. However, the attempts to find even non-optimal solutions for the puzzle using weak methods were not very successful.

A major advance in finding non-optimal solutions for the sliding-tile puzzle was achieved when macro-learning techniques were applied [6, 10, 12, 22]. Korf [10] built a macro table that could handle any $3 \times 3$ puzzle. Another (rather large) macro table could handle any $4 \times 4$ puzzle. Iba’s Maclearn [7] made further progress by being able to transfer macros that were acquired while solving $4 \times 4$ puzzles to solving $5 \times 5$ puzzles. Stepping-Stone [21] was the

<table>
<thead>
<tr>
<th>Heuristic</th>
<th>Op - Learning</th>
<th>CPU seconds</th>
<th>Problems</th>
<th>Macros</th>
<th>Max length</th>
<th>Op - Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row-by-row</td>
<td>265,872 (184,712)</td>
<td>34.7 (8.83)</td>
<td>66.3 (12.9)</td>
<td>10.39 (0.8)</td>
<td>11 (8)</td>
<td>212.7 (7.5)</td>
</tr>
<tr>
<td>Reduction</td>
<td>303,827 (164,173)</td>
<td>48.07 (10.12)</td>
<td>70.7 (12.6)</td>
<td>13.8 (0.37)</td>
<td>13 (8)</td>
<td>222.3 (7.2)</td>
</tr>
<tr>
<td>Spiral</td>
<td>296,488 (87,523)</td>
<td>48.17 (9.8)</td>
<td>70 (10.8)</td>
<td>13.96 (0.19)</td>
<td>13 (8)</td>
<td>214.2 (5.6)</td>
</tr>
<tr>
<td>Manhattan</td>
<td>7,175,086 (2,657,633)</td>
<td>1458 (315)</td>
<td>3425 (75)</td>
<td>104 (5.2)</td>
<td>23 (8)</td>
<td>1432 (341)</td>
</tr>
</tbody>
</table>

Table 3: Summary of learning sessions in the eight-puzzle domain using various heuristic functions. Each number represents the mean over 100 learning sessions. The standard deviation is given in parenthesis.

<table>
<thead>
<tr>
<th>Heuristic</th>
<th>15-puzzle</th>
<th>10-Cannibals</th>
<th>10-stones</th>
<th>5-Hanoi</th>
<th>Grid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op-learning</td>
<td>492945 (84,643)</td>
<td>216572 (98,467)</td>
<td>144303 (1653)</td>
<td>377671 (80,968)</td>
<td>1956972 (1,191,383)</td>
</tr>
<tr>
<td>CPU seconds</td>
<td>685 (224)</td>
<td>14.2 (6.0)</td>
<td>8.4 (1.9)</td>
<td>35.2 (3.97)</td>
<td>58.8 (35)</td>
</tr>
<tr>
<td>Problems</td>
<td>60.6 (6.0)</td>
<td>63.9 (12.0)</td>
<td>52 (8.28)</td>
<td>76.2 (9.55)</td>
<td>185 (60.2)</td>
</tr>
<tr>
<td>Macros</td>
<td>14.3 (6.83)</td>
<td>2.16 (0.39)</td>
<td>1.2 (0.4)</td>
<td>11.47 (0.71)</td>
<td>17.5 (3.14)</td>
</tr>
<tr>
<td>Max macro</td>
<td>18 (8)</td>
<td>3.98 (0.2)</td>
<td>2 (8)</td>
<td>16 (8)</td>
<td>16.36 (1.49)</td>
</tr>
<tr>
<td>Testing Best-first</td>
<td>*</td>
<td>3.74</td>
<td>157.9</td>
<td>1518</td>
<td>436</td>
</tr>
<tr>
<td>Testing - before</td>
<td>224529</td>
<td>57</td>
<td>206.5</td>
<td>10993</td>
<td>1159</td>
</tr>
<tr>
<td>Testing - after</td>
<td>704 (23)</td>
<td>31.6 (4.43)</td>
<td>126.6 (8.1)</td>
<td>156.2 (9.44)</td>
<td>368.9 (35)</td>
</tr>
</tbody>
</table>

Table 4: Summary of learning sessions performed in various domains. Each number represents the mean over 100 learning sessions. The standard deviation is given in parenthesis. We were not able to complete a best-first search in the 15-puzzle domain (due to memory problems).

### 4.4 Experiment 3: Learning macros in other domains

We have applied Micro-Hillary to the other domains specified in Subsection 4.1. Table 4 shows the mean results for 100 learning sessions. Micro-Hillary was able to reach quiescence in all the domains. The 10-stones and 10-cannibals are very simple. One or two macros were sufficient to reach quiescence. Note that we used the same quiescence parameter, 50 problems, for all the domains. After solving each problem, Micro-Hillary increases by 100 the length of the random sequence used for generating a training problem. Therefore, Micro-Hillary spends 125,000 operator applications just to make sure that there is nothing new to learn. In the simple domains, that spends most of the resources during learning.

It is interesting to look at the macros learned in the grid domain. Most of the macros have a structure of $SSS...SNN...NNN$ where $S$ stands for south, $W$ for west, $N$ for north and the number of $S$ and $N$ is equal. Such macros are used to make detours around walls that block the search.

Micro-Hillary was able to improve the performance of problem solving in each of the domains. The most notable improvement is in the 15-puzzle domain where the performance after learning is 300 times better than the performance before learning.

### 5 Solving the general $N \times N$ puzzle

In the previous section we showed how Micro-Hillary can be used to solve the eight-puzzle and the fifteen-puzzle domains. Is it possible to use Micro-Hillary to solve the general $N \times N$ sliding-tile puzzle? The eight-puzzle and the fifteen-puzzle have always been favorite problems among mathematicians and among AI researchers [6, 11, 20, 23]. Ratner [20] proved that finding the shortest path between two $N \times N$ puzzle states is NP-complete. AI researchers have used the sliding-tile domain as a test bed for heuristic search methods. However, the attempts to find even non-optimal solutions for the puzzle using weak methods were not very successful.

A major advance in finding non-optimal solutions for the sliding-tile puzzle was achieved when macro-learning techniques were applied [6, 10, 12, 22]. Korf [10] built a macro table that could handle any $3 \times 3$ puzzle. Another (rather large) macro table could handle any $4 \times 4$ puzzle. Iba’s Maclearn [7] made further progress by being able to transfer macros that were acquired while solving $4 \times 4$ puzzles to solving $5 \times 5$ puzzles. Stepping-Stone [21] was the
first program that was able to solve $6 \times 6$ puzzles. The most significant advance in solving the $N \times N$ sliding-tile puzzle was achieved independently by EASE [22] that solved puzzles of $20 \times 20$, and by Hillary [3], that solved puzzles of $32 \times 32$.

5.1 Using Micro-Hillary in scalable domains

Some domains can be scaled up or down by adjusting a certain parameter, such that the same operators are applicable in the scaled domain (maybe by using the parameter in their definition). For example, we can define a family of domains called the $N \times N$ puzzle, where each $N$ defines a different puzzle domain. When $N = 3$ we get the eight-puzzle domain, when $N = 4$ we get the fifteen-puzzle domain etc. All the domains in this family use the same operators: Up, Down, Left and Right.

We have developed a learning algorithm called Parametric Micro-Hillary that uses Micro-Hillary in scalable domains. The algorithm sets the parameter to its lowest value and calls Micro-Hillary. When Micro-Hillary returns (due to quiescence), the parameter is increased and Micro-Hillary is called again. The algorithm stops when no new macros are added. The idea behind this algorithm is to extend the strategy of generating training problems of increasing difficulty. Micro-Hillary increases the length of the random sequences to generate more complex problems, while Parametric Micro-Hillary also increases the domain parameter.

5.2 Applying Parametric Micro-Hillary to the $N \times N$ puzzle domains.

Table 5 lists the exact parameters used for Micro-Hillary in the $N \times N$ puzzle domain. To make the table more concise, we use the following definitions and notations. A puzzle state is a permutation of the sequence $\langle 0,1,\ldots,n^2-1 \rangle$. Each of the elements of the puzzle-state is called a tile. 0 is called the empty tile. For convenience we will represent a puzzle state by an $N \times N$ row major matrix. Let $s$ be a state, let $i, j \leq n$, we define $tile_s(i, j)$ to be the tile located in row $i$ column $j$ of $s$ when representing it as a row-major $n \times n$ matrix. For every state $s$ and tile $t \in s$ we define the tile location $loc_s(t) = (i, j)$ where $tile_s(i, j) = t$. Let $t_1 = (i_1, j_1)$ and $t_2 = (i_2, j_2)$ be two locations. The distance between $t_1$ and $t_2$ is defined as $d = \mid i_1 - i_2 \mid + \mid j_1 - j_2 \mid$. Let $s = \langle t_1, \ldots, t_{n^2} \rangle$ be a state. Let $s_g = \langle g_1, \ldots, g_{n^2-1}, 0 \rangle$ be the goal state. The number of placed tiles is the largest $p$ such that $t_i = g_i$ for all $i \leq p$. The expression of $p + 1$ in the matrix notation is called the next location and is marked as $NextLoc(s)$. $g_p$ is called the next tile and is marked as $NextTile(s)$.

<table>
<thead>
<tr>
<th>Generate-goal</th>
<th>Let $n$ be the value of the complexity parameter. Let $t_1, \ldots, t_{n^2-1}$ be a random permutation of $1, \ldots, n^2-1$. Generate-goal returns $\langle t_1, \ldots, t_{n^2-1}, 0 \rangle$.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic-operators</td>
<td>${u, d, l, r}$. Let $loc_s(0) = (i_0, j_0)$. $u(s)$ is defined as:</td>
</tr>
<tr>
<td>$tile_{u(s)}(i, j)$</td>
<td>$tile_{u(s)}(i, j) = \begin{cases} tile_s(i_0 - 1, j_0) : &amp; i = i_0, j = j_0 \ 0 : &amp; i = i_0 - 1, j = j_0 \ tile_s(i, j) : &amp; \text{otherwise} \end{cases}$</td>
</tr>
<tr>
<td>Heuristic-function</td>
<td>$h(s, s_g) = 4n^3 \left( n^2 - \text{placed}(s, s_g) \right) + 2nd \left( NextLoc(s, s_g), Loc_s(NextTile(s, s_g)) \right)$</td>
</tr>
<tr>
<td>$d(loc_s(0), Loc_s(NextTile(s, s_g)))$</td>
<td>$d(loc_s(0), Loc_s(NextTile(s, s_g)))$</td>
</tr>
</tbody>
</table>

Table 5: The definitions of the parameters that were used to apply Micro-Hillary to the $N \times N$ puzzle domains.

5.3 Experimenting with Parametric Micro-Hillary in the $N \times N$ puzzle domains.

We have applied the Parametric Micro-Hillary algorithm to the $N \times N$ puzzle domains, with the domain parameter set to an initial value of 3. Table 6 shows the summary of 100 learning sessions. Micro-Hillary learned most of its macros by solving 3 by 4 by 4 puzzles. It typically learned one or two macros while solving 5 by 5 puzzles, and reached quiescence while solving 6 by 6 problems without learning any new macros. Note that 200 out of the 232 training problems and at least half of the learning time (operator applications) were used just for reaching quiescence.
Table 6: The summary of 100 learning session performed by Parametric Micro-Hillary in the $N \times N$ puzzle domain.

<table>
<thead>
<tr>
<th>Op Applications</th>
<th>CPU seconds</th>
<th>Training problems</th>
<th>Macros learned</th>
<th>Maximal macro length</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Massive</td>
<td>27,0.7</td>
<td>232.8 (13)</td>
<td>14.87 (0.83)</td>
<td>18 (8)</td>
</tr>
</tbody>
</table>

Table 7: The average performance of Micro-Hillary in the testing sessions for each of the 100 learned macro sets. The standard deviation is given in parenthesis.

<table>
<thead>
<tr>
<th>Op Applications</th>
<th>CPU seconds</th>
<th>Solution length</th>
<th>Optimality upper bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Massive</td>
<td>15.64 (0.85)</td>
<td>30 (188)</td>
<td>4.60 (0.4)</td>
</tr>
</tbody>
</table>

We tested the resulted macro sets on 100 test problems of $10 \times 10$. Table 7 shows the results. Note that, unlike the eight-puzzle experiment, here we cannot compute the exact ratio between the solution length and the optimal solution. Instead, we use the sum of Manhattan distances as a lower bound on the length of the optimal solution to compute an upper bound on the ratio.

A second test was performed with one of the macro sets. We generated sets of 10 testing problems for various sizes up to $50 \times 50$ and solved these problems using the macro set. Figure 4 shows the mean number of operator applications as a function of the puzzle size. Micro-Hillary solved the whole test set. It looks as if the program can now solve any $N \times N$ solvable problem. We have indeed succeeded to prove that the set of macros learned is sufficient to solve any solvable $N \times N$ puzzle, and that it can do it in $O(N^3)$. The proof is given in appendix B. The proof is for a specific macro set but can be repeated for any other macro set. We do not claim that Micro-Hillary will always perform successful learning. It is feasible that Micro-Hillary will quit learning due to quiescence but will miss essential macros. However, the probability of such an event diminishes with an increase in the size of the quiescence parameter.

The left graph of Figure 5 shows the mean solution length as a function of the puzzle size. The right graph of Figure 5 shows the upper bound on the optimality as a function of the puzzle size. It is interesting to note that the graph is flattened at a value of 6. Ratner and Warmuth [20] sketch a (quite complicated) hand-crafted approximation algorithm for the $N \times N$ puzzle. Looking at Figure 5 it is quite possible that Micro-Hillary was able to find such an algorithm by itself. The proof for the completeness of the solution found by Micro-Hillary for the $N \times N$ puzzle also proves that the length of the solution is $O(N^3)$. Recall that we compute the upper bound as the length of the solution divided by the sum of Manhattan distances. The actual ratio is lower. Korf [9] lists the length of the optimal solution for $100 \times 4 \times 4$ puzzles found using IDA*. We applied Micro-Hillary to this set of problems and compared the length of its solutions to the optimal length. The solutions found by Micro-Hillary were 3.19 times longer on average (the upper bound computed above for $4 \times 4$ puzzles is 3.72). However, IDA* generated, on average, 503,326 times the nodes generated by Micro-Hillary. This looks like a very attractive trade-off: losing a factor of 3 in quality in return for a factor of 500,000 in performance.

5.4 Experimenting with Parametric Micro-Hillary in other scalable domains.

We tested Parametric Micro-Hillary on other parameterized domains. In the N-Cannibals and N-Stones domains, Parametric Micro-Hillary learned all the macros with the minimal value of the parameter (3). The test was done using problems with the parameter equal to 20 in both domains. Micro-Hillary’s performance indeed improved in both domains and problem solving proceeded without encountering local minima.

The N-Hanoi domain family is recursive in nature and we did not expect Parametric Micro-Hillary to “solve” these domains. The length of the macros should grow with the growth in the number of rings, therefore Micro-Hillary should not reach quiescence in these domains. We were surprised to find out that Parametric Micro-Hillary achieved quiescence after solving problems of 7 or 8 rings (713 on average). This wrong decision is a result of the domain-independent training-problems generator. The probability that the largest ring is moved from its target location after a random sequence of moves is very low. Indeed, when we increased the length of the sequences used for generating training problems and increased the quiescence parameter, learning continued but Micro-Hillary still reached quiescence after solving problems with 9 rings. In both cases the macros learned were not sufficient for solving problems with a
Figure 4: The mean number of operator applications as a function of the puzzle size. The error bars are one standard deviation away on either side of the mean. The right graph shows the results for all the sizes up to 50. In the left graph we show the results for sizes up to 20.

Figure 5: The left graph shows the solution length (in basic moves) as a function of the puzzle size. The right graph shows the upper bound on the ratio between the solution length and the shortest solution as a function of the puzzle size. In both graphs the error bars are one standard deviation away on either side of the mean.

parameter that is larger than the values encountered during training. For solving domain families such as N-Hanoi, we should extend Micro-Hillary and endow it with the capability of generating recursive macros. To avoid the problem of Parametric Micro-Hillary quitting prematurely, we can modify the problem generator of Micro-Hillary to use random sequences with length that is based on the domain parameter. Alternatively, we can use domain-specific problem generators.

The results of this experiment are summarized in Table 8.

Another related experiment that we performed involved transfer of knowledge between two similar domains (but not parameterized as the domains above). We generated a random grid, different than the one used for the experiments above, and performed a testing session with Micro-Hillary, using macros that were learned in the first grid. Using the macros improved Micro-Hillary's performance — from 1529 operator applications without macros down to 594 with macros.

6 Discussion

The Micro-Hillary algorithm presented in this paper, despite its simplicity, was able to solve the $N \times N$ puzzle as well as several other domains. In this section, we would like to compare Micro-Hillary to other macro-learning algorithms and discuss its strengths and weaknesses.

Most of the existing macro-learning programs are based on the notion of subgoaling: the learner tries to acquire
<table>
<thead>
<tr>
<th></th>
<th>N-Cannibals</th>
<th>N-Stones</th>
<th>N-Hanoi</th>
</tr>
</thead>
<tbody>
<tr>
<td>op-learning</td>
<td>331,183 (57,723)</td>
<td>27,852 (3,382)</td>
<td>2,427,186 (558,533)</td>
</tr>
<tr>
<td>CPU seconds</td>
<td>20.84 (8.8)</td>
<td>11.53 (2.7)</td>
<td>579 (176)</td>
</tr>
<tr>
<td>Problems</td>
<td>112.5 (8.5)</td>
<td>103 (8.6)</td>
<td>370 (43.7)</td>
</tr>
<tr>
<td>Max Parameter</td>
<td>4 (0)</td>
<td>4 (0)</td>
<td>7.13 (8.38)</td>
</tr>
<tr>
<td>Macros</td>
<td>4.4 (0.31)</td>
<td>1.22 (0.41)</td>
<td>16.08 (0.9)</td>
</tr>
<tr>
<td>Max macro</td>
<td>4 (0)</td>
<td>2 (0)</td>
<td>33.2 (6.3)</td>
</tr>
<tr>
<td>Testing - before</td>
<td>150</td>
<td>3671</td>
<td>171,956 (168,338)</td>
</tr>
<tr>
<td>Testing - after</td>
<td>105.5 (10.1)</td>
<td>2009 (155)</td>
<td>2913 (16,539)</td>
</tr>
</tbody>
</table>

Table 8: Summary of learning sessions performed in various domains. Each number represents the mean over 100 learning sessions. The standard deviation is given in parenthesis.

macros that achieve some subgoal without undoing previously satisfied subgoals [10, 12, 21, 22, 26, 27]. Micro-Hillary, unlike MACLEARN[6], does not assume subgoaling, but assumes the existence of a heuristic function. EASE [22] combines subgoaling with a heuristic function to guide the search for the current subgoal. The subgoal-oriented macro-learners use various methods to guard the previously achieved subgoals. Micro-Hillary is much simpler — it only requires that the macro acquired leads from a local minimum to a state with a better heuristic value.

Unlike some speedup learners that provide us with either statistical or theoretical guarantees [4, 1, 27, 5, 25], Micro-Hillary has a heuristic nature and does not provide us with any guarantee. Indeed, while it performs very well in some domains, it fails to completely solve other domains such as the N-Hanoi. To handle such domains we would have to endow Micro-Hillary with the capability of learning parameterized recursive macros. A related weakness of Micro-Hillary is the sensitivity of the algorithm to the heuristic function available. We have shown an example where Micro-Hillary has extreme difficulties with a seemingly good heuristic (the sum of Manhattan distances for the eight-puzzle) while learning easily with another one.

We can say that Micro-Hillary performs well when there is a “small” set of operator sequences that completely smooth the heuristic function to eliminate any local minimum. However, we did not find a method to characterize such domains. Even when the domain and the heuristic function have the “smoothability” characteristic, Micro-Hillary is not guaranteed to perform well due to the distance from the local minimum to a state with better heuristic value. If the heuristic function does not give any indication toward the direction of the escape route, Micro-Hillary will resort to an exhaustive search. If the escape routes are long, such a search may require too many computation resources.

Another weakness of Micro-Hillary is its sensitivity to the quiescence parameter. This parameter tells Micro-Hillary when to quit learning. For Parametric Micro-Hillary it is used to determine when to increase the domain parameter for training. Setting the quiescence parameter to a low value can make Micro-Hillary quit prematurely before it learned all the necessary macros. Setting it to a high value can lead to a great waste of learning resources. Indeed, for the simple domains such as the N-Cannibals and N-Stones, Micro-Hillary spent most of its learning time making sure that there is nothing new to learn.

While the above weaknesses are significant, we should also consider the strengths of the algorithm. Micro-Hillary is extremely simple, yet, it is able to solve many known problems. The ability of such a simple learning algorithm to solve the general $N \times N$ puzzle problem shows the potential in using selective learning for speeding up problem solvers.

**Appendix A: The code for Micro-Hillary**

Although it is not common for AI papers to include the actual code of programs, we decided to include the code for Micro-Hillary (Fig. 6) and the $N \times N$ puzzle (Fig. 6) in order to emphasize the simplicity of the algorithm, and to allow the readers to experiment with the system.
Appendix B: A proof of completeness

This section contains a proof that a specific set of macros learned by Micro-Hillary is sufficient for solving any \( N \times N \) puzzle. The proof can be easily generalized to specify sufficient conditions for a set of macros to be complete.

**Lemma 1** Let \( s = (t_1, \ldots, t_N) \) be a puzzle state. If \( d(loc_i(0), loc_{nextT ile}(s))) > 1 \), then there exists a basic operator \( o \in B \) such that \( h(o(s), s_g) < h(s, s_g) \).

**Proof:**

Let \( loc_{nextT ile}(s) = (i_p, j_p), loc_{0}(0) = (i_0, j_0) \) and \( NextLoc(s) = (i_b, j_b) \). The following basic operators will decrease the value of \( h \):

\[
o = \begin{cases}
  j_p > j_b & r \\
  j_p = j_b & \begin{cases}
    i_p > i_b & u \\
    i_p < i_b & d
  \end{cases} \\
  j_p < j_b & \begin{cases}
    i_p > i_b & d \\
    i_p < i_b & l
  \end{cases}
\end{cases}
\]

(2)

**Lemma 2** Let \( M \) be the following set of macros:

\[
M = \{ fur, ru, uld, rudd, allur, arrul, urudd, urludd, udrrulud, urlurruulhd, udldrruldrdr, uradrululd, udldruldrdr, ulldrulldrdr \}
\]
Let $O = B \cup M$. Then, there is always an operator $o \in O$ such that $h(o(s)) < h(s)$.

**Proof:**

By Lemma 1, we need to prove only for the cases where $d(\text{loc}(s), \text{loc}(N\text{extT}\text{ile}(s))) = 1$. There are four possible cases. Table 9 shows which operator can be applied in each of the cases in order to decrease the value of $h$. To make the proof simpler, the table assumes $N > 4$.

**Theorem 1** Any $N \times N$ puzzle is solvable using the Micro-Hillary algorithm. The complexity of the algorithm is $O(N^3)$.

**Proof**

By Lemma 2, we can move to each state we can move to a state with a lower heuristic value. Therefore, after a finite number of states, we will reach the state $s$ with $h(s) = 0$, which is the goal state. $N^2$ tiles are moved into their goal location. The maximal distance from the goal location is $2(N - 1)$. To move a tile $t$ to its goal the blank tile is moved toward $t$ using $2(N - 1)$ moves at most, then a sequence of macros and basic operators is applied. The blank tile stays in distance of at most 4 from $t$ after a macro application. The maximal length of a macro is 18. Thus the total number of moves is bounded by $N^2[2(N - 1) + 2(N - 1)(18 + 4)] = 46N^3 - 46N^2$.

**References**


\[ j_p = j_0 \]
\[ i_p > i_0 \]
\( (i_p = i_0 + 1) \]
\[ \text{d} \]

\[ j_p < j_0 \]
\[ (i_p = i_0 - 1) \]
\[ j_p > j_0 \]
\[ \text{lur} \]

\[ j_p < j_0 \]
\[ j_p = j_0 \]
\[ \text{ru} \]

\[ j_p = j_0 \]
\[ j_p < N \]
\[ \text{ruul} \]

\[ j_p > j_0 \]
\[ (j_p = j_0 + 1) \]
\[ i_p = i_0 \]
\[ j_p > j_0 \]
\[ \text{r} \]

\[ j_p < j_0 \]
\[ i_p < N \]
\[ \text{drrul} \]

\[ i_p = N \]
\[ i_k < N - 1 \]
\[ \text{urru} \]

\[ i_p = N \]
\[ i_k = N - 1 \]
\[ j_k - j_p < 1 \]
\[ \text{urru} \]

\[ i_p = N \]
\[ i_k = N - 1 \]
\[ j_k - j_p > 1 \]
\[ \text{urdrull} \]

\[ j_p = j_0 \]
\[ j_0 > 1 \]
\[ \text{lurru} \]

\[ j_p < j_0 \]
\[ (j_p = j_0 - 1) \]
\[ i_p = i_0 \]
\[ j_p > j_0 \]
\[ i_k < N \]
\[ i_p < N \]
\[ \text{dllur} \]

\[ i_k < N \]
\[ i_p = N \]
\[ \text{uld} \]

\[ i_k = N \]
\[ j_0 < N \]
\[ \text{lurdrull} \]

\[ i_k = N \]
\[ j_0 = N \]
\[ j_k < N - 2 \]
\[ \text{ulldurrul} \]

\[ i_k = N \]
\[ j_0 = N \]
\[ j_k \geq N - 2 \]
\[ \text{Unsolvable puzzle} \]

\[ j_p < j_0 \]
\[ \text{l} \]

\[ j_p = j_t \]
\[ \text{uld} \]

Table 9: The macro used for each of the conditions on the tile indices. The right column shows an example for a state before and after the macro application.