<table>
<thead>
<tr>
<th>$j_p = j_0$</th>
<th>$i_p &gt; i_0$</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i_p &lt; i_0$</td>
<td>$j_p &gt; j_b$</td>
<td>lur</td>
</tr>
<tr>
<td>$j_p &lt; j_b$</td>
<td>$j_p &lt; N$</td>
<td>ruuld</td>
</tr>
<tr>
<td>$j_p = j_b$</td>
<td>$j_p &lt; N$</td>
<td>ruuld</td>
</tr>
<tr>
<td>$j_p &gt; j_0$</td>
<td>$i_p = i_0$</td>
<td>r</td>
</tr>
<tr>
<td>$j_p &lt; j_b$</td>
<td>$i_p &lt; N$</td>
<td>drrul</td>
</tr>
<tr>
<td>$j_p = j_0$</td>
<td>$j_0 &gt; 1$</td>
<td>lurruuld</td>
</tr>
<tr>
<td>$j_p &lt; j_0$</td>
<td>$i_p = i_0$</td>
<td>lurruuld</td>
</tr>
<tr>
<td>$i_p &lt; N$</td>
<td>$j_0 &lt; N$</td>
<td>udl</td>
</tr>
<tr>
<td>$i_p = N$</td>
<td>$j_0 = N$</td>
<td>lurrdrrldrrrd</td>
</tr>
<tr>
<td>$j_0 = N$</td>
<td>$j_0 &lt; N$</td>
<td>lurrdrrldrrrd</td>
</tr>
<tr>
<td>$j_0 = N$</td>
<td>$j_0 = N$</td>
<td>lurrdrrldrrrd</td>
</tr>
<tr>
<td>$j_0 = N$</td>
<td>$J_0 &gt; N - 2$</td>
<td>Unsolvable puzzle</td>
</tr>
<tr>
<td>$j_p &lt; j_b$</td>
<td>$J_b &lt; N - 2$</td>
<td>lurrdrrldrrrd</td>
</tr>
<tr>
<td>$j_p = j_b$</td>
<td>$J_b &lt; N - 2$</td>
<td>lurrdrrldrrrd</td>
</tr>
</tbody>
</table>

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.


blank tile is moved toward \( t \) (using \( 2(N - 1) \) moves at most), and a sequence of macros and basic operators is then applied. The blank tile remains at a distance of no more than 4 from \( t \), after a macro application. In the worst case, the algorithm may try all the macros before finding the one that reduces the heuristic value. The sum of lengths of all the macros is \( 124 \). Thus the total number of operators applied during the search is bounded by 
\[ N^3 \left[ 2(N - 1) + 2(N - 1)(124 + 4) \right] = 258N^3 - 258N^2. \]
The maximal length of a macro is 18. Therefore the length of the solution is bounded by 
\[ N^3 \left[ 2(N - 1) + 2(N - 1)(18 + 4) \right] = 46N^3 - 46N^2. \]

We can use the same lemmas to prove that Micro-Hillary can solve any solvable problem in \( O(N^3) \) even without learning by performing BFS search in each place where a macro is used in the original proof. However, instead of using 124 in the above formula, we will use \( 4^{18} \) (18 is the length of the longest macro I have several minor complaints with the paper that I believe should be corrected before publication. macro which can be used as an upper bound on the depth of the search), resulting in a bound of \( 137,438,953,480(N^3 - N^2) \), which is 532,709,122 times larger than the constant for Micro-Hillary after learning. While in the pure sense of computational complexity both bounds belong to the same complexity class, the huge constant makes this fact meaningless for any practical purpose.

References


is not guaranteed to perform well because of the distance from the local minimum to a state with a better heuristic value. If the heuristic function does not give any indication of the direction of the escape route, Micro-Hillary resorts 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 cause Micro-Hillary to quit prematurely before learning when to quit learning. For Parametric Micro-Hillary it is used to determine when to increase the domain parameter for 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 learn to efficiently solve problems in a number of known domains. The ability of such a simple learning algorithm to find an efficient procedure for solving general \( N \times N \) puzzle problems shows the potential in using selective learning for speeding up problem solvers.

**Appendix: The efficiency of solving \( N \times N \) puzzles by Micro-Hillary**

This section contains a proof that a specific set of macros learned by Micro-Hillary is complete with respect to the heuristic function \( h \) defined in Table 5, and that a problem solver that uses these macros can solve any solvable \( N \times N \) puzzle problem in \( O(N^3) \) with a reasonable constant. The proof can be easily generalized to specify sufficient conditions for a set of macros to be complete.

**Lemma 1** Let \( h \) be the heuristic function and \( B \) the set of basic operators defined in Table 5. Let \( s_g \) be a goal state of \( N \times N \) puzzle. Let \( s \in \{t_1, \ldots, t_{N^2}\} \) be a puzzle state different from \( s_g \) such that \( \text{solvable}(s, s_g) \).

If \( d(\text{loc}_s(0), \text{loc}_s(\text{nextTile}(s))) > 1 \), then there exists a basic operator \( o \in B \) such that \( h(o(s), s_g) < h(s, s_g) \).

**Proof:**
Let \( \text{loc}_s(\text{nextTile}(s)) = \langle i_p, j_p \rangle \), \( \text{loc}_s(0) = \langle i_0, j_0 \rangle \) and \( \text{nextLoc}(s) = \langle i_h, j_h \rangle \). The following basic operators will decrease the value of \( h \):

\[
o = \begin{cases} 
  j_p > j_0 & r \\
  j_p = j_0 & \begin{cases} 
    i_p > i_0 & u \\
    i_p < i_0 & d 
  \end{cases} \\
  j_p < j_0 & \begin{cases} 
    i_p < i_0 & d \\
    i_p > i_0 & l 
  \end{cases}
\end{cases}
\]

**Lemma 2** Let \( h \), \( B \), \( s_g \), \( s \) be defined as in Lemma 1. Let \( M \) be the following set of macros:

\[
M = \{ \text{har, rul, uld, ruldr, allhar, arrul, urrdhdr, urrdhdr, lurrulldrurr, lurdrulldrurr, urldlturdrdr} \}
\]

Then the set \( M \) is a complete set of macros, i.e., there is always an operator \( o \in B \cup M \) such that \( h(o(s), s_g) < h(s, s_g) \).

**Proof:**
By Lemma 1, we need to prove only for the cases where \( d(\text{loc}_s(0), \text{loc}(\text{nextTile}(s))) = 1 \). There are four possible cases. Table 9 shows which operator can be applied in each of the cases to decrease the value of \( h \). To make the proof simpler, the table assumes \( N > 4 \).

**Theorem 1** Let \( h \), \( M \) be defined as in Lemma 2. Micro-Hillary, using \( M \), can solve any solvable \( N \times N \) puzzle problem performing no more than \( 256(N^3 - N^2) \) basic operator applications. The length of the solution is bounded by \( 6(N^3 - N^2) \).

**Proof:**
By Lemma 2, we can move from each state to a state with a lower heuristic value. Therefore, after a finite number of states, we will reach the state \( s \) with \( h(s, s_g) = 0 \), 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
Figure 5: The left graph shows the solution length (in basic moves) as a function of the puzzle size. The right graph shows the ratio between the solution length and the heuristic lower-bound on 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.

<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>331,185 (59,723)</td>
<td>277,852 (3384)</td>
<td>2,427,186 (588,933)</td>
</tr>
<tr>
<td>CPU seconds</td>
<td>20.84 (3.8)</td>
<td>11.53 (2.7)</td>
<td>579 (176)</td>
</tr>
<tr>
<td>Problems</td>
<td>112.5 (8.3)</td>
<td>163 (6.6)</td>
<td>570 (43.7)</td>
</tr>
<tr>
<td>Max Parameter</td>
<td>4 (0)</td>
<td>4 (0)</td>
<td>7.13 (0.36)</td>
</tr>
<tr>
<td>Macros</td>
<td>4.4 (0.51)</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 (6.5)</td>
</tr>
<tr>
<td>Testing - before</td>
<td>150</td>
<td>3671</td>
<td>171,956 (188,338)</td>
</tr>
<tr>
<td>Testing - after</td>
<td>105.5 (10.1)</td>
<td>2009 (155)</td>
<td>2913 (16,538)</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 parentheses.

6 Discussion

Despite its simplicity, the Micro-Hillary algorithm presented in this paper was able to learn macros that allow efficient solution of any solvable $N \times N$ puzzle problem. It also performed well in a number of other domains. In this section, we 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 macros that achieve some subgoal without undoing previously satisfied subgoals [10, 12, 21, 22, 26, 27]. Micro-Hillary, like 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, requiring only that the macro acquired lead 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 in 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 minima. 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
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 parentheses.

<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>1529 (1020)</td>
<td>15.64 (0.85)</td>
<td>3628 (268)</td>
<td>4.60 (0.4)</td>
</tr>
</tbody>
</table>

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 results up to size 50. In the left graph we show the results for sizes up to 20.

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

We tested Parametric Micro-Hillary in 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 performed using problems with a parameter of 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 find a complete macro set for these domains. The length of the macros should grow with the number of rings; therefore, Micro-Hillary should not reach quiescence in these domains. We were surprised to find that Parametric Micro-Hillary achieved quiescence after solving problems of 7 or 8 rings (7.13 on average). This error was caused by the domain-independent training-problems generator. The probability that the largest ring will be 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 parameter that is larger than the values encountered during training. The test was performed with problems of 6 rings. 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 involved transfer of knowledge between two similar domains (but not parameterized as the domains above). We generated a random grid, different from 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.
Let $n$ be the value of the domain parameter. Let $t_1, \ldots, t_{n^2-1}$ be a random permutation of $1, \ldots, n^2-1$. Generate-goal returns $(t_1, \ldots, t_{n^2-1}, 0)$.

Basic-operators
\[ \{u, d, l, r\}. \]

Let $loc_x(0) = (i_0, j_0)$. $u(s)$ is defined as:
\[
tile_u(i, j) = \begin{cases} 
  tile_x(i - 1, j_0) : & i = i_0, j = j_0 \\
  0 : & i = i_0 - 1, j = j_0 \\
  tile_x(i, j) : & \text{otherwise}
\end{cases}
\]

Heuristic-function
\[
h(s, s_g) = 4n^2 (n^2 - \text{placed}(s, s_g)) + 2nd(\text{NextLoc}(s), \text{Loc}_s(\text{NextTile}(s)))
\]

\[
d(loc_x(0), \text{Loc}_s(\text{NextTile}(s)))
\]

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

<table>
<thead>
<tr>
<th>Op Applications</th>
<th>CPU seconds</th>
<th>Training problems</th>
<th>Macros learned</th>
<th>Maximum macro length</th>
</tr>
</thead>
<tbody>
<tr>
<td>1084571 (124338)</td>
<td>270.7 (62.7)</td>
<td>232.8 (13)</td>
<td>14.87 (0.83)</td>
<td>18 (0)</td>
</tr>
</tbody>
</table>

Table 6: The summary of 100 learning sessions performed by Parametric Micro-Hillary in the $N \times N$ puzzle domain.

We tested the resulting macro sets on 100 test problems of $10 \times 10$. Table 7 shows the results. Note that here, unlike in the eight-puzzle experiment, 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 arbitrarily selected the one that was learned first). 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 in reasonable time. It looks as if the program can now efficiently solve any $N \times N$ solvable problem. We have indeed succeeded in proving that the set of macros learned is complete and that Micro-Hillary can solve any solvable problem in $O(N^3)$ with a reasonable constant (258). The proof is given in the appendix. The proof is for a specific macro set, but can be repeated for the other macro sets as well. A variation of that proof shows that Micro-Hillary can solve any solvable problem in $O(N^3)$ even without learning. However, the constant in this upper limit is prohibitively high ($10^{12}$). We do not claim that Micro-Hillary will always perform successful learning. It is possible that Micro-Hillary will reach quiescence and quit learning, having missed 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 could have found such an algorithm by itself. In the appendix, we prove that the length of the solutions found by Micro-Hillary for $N \times N$ puzzle problems 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 $100 \times 4$ 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. Also, computational limitations restrict IDA* to small puzzles (up to $4 \times 4$), while Micro-Hillary can efficiently solve puzzles as large as $50 \times 50$.  

\[ \text{Table 5: The definitions of the parameters that were used to apply Micro-Hillary to the } N \times N \text{ puzzle domains.} \]

\[ \text{Table 6: The summary of 100 learning sessions performed by Parametric Micro-Hillary in the } N \times N \text{ puzzle domain.} \]

\[ \text{Table 7: The results of the tests performed.} \]
5 Solving the general $N \times N$ puzzle

In the previous section we showed how Micro-Hillary can be used to efficiently solve problems in the eight-puzzle and the fifteen-puzzle domains. Is it possible to use Micro-Hillary to efficiently solve general $N \times N$ sliding-tile puzzle problems? The eight-puzzle and the fifteen-puzzle problems have long been popular among mathematicians and AI researchers [6, 11, 20, 25]. Ratner [26] 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 general heuristic search methods. However, the attempts to find even non-optimal solutions for the puzzle using weak methods were successful only for small puzzles. It is not difficult to design a special purpose algorithm that solves the $N \times N$ puzzle efficiently, but applying a weak method such as best-first search with the sum-of-Manhattan-distances heuristic on puzzles as small as $5 \times 5$ fails because of the extensive computation resources required.

A major advance in using weak methods to find 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 with the ability to solve $5 \times 5$ puzzles using macros acquired while solving puzzles of $4 \times 4$. Stepping-Stone [21] was able to solve $6 \times 6$ puzzles. EASe [22] solved puzzles of $20 \times 20$, and Hillary [3] solved puzzles of $32 \times 32$. All these programs not only solved the puzzle problems using weak methods, but also acquired this ability by learning.

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 (perhaps 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 $(0, 1, \ldots, n^2 - 1)$. 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 $i, j < n$. We define $tile(i, j)$ to be the tile located in row $i$ column $j$ of $s$, where $s$ is represented by a row-major $n \times n$ matrix. For every state $s$ and tile $t \in s$, we define the tile location $loc(t) = (i, j)$ where $tile(i, j) = t$. Let $l_1 = (i_1, j_1)$ and $l_2 = (i_2, j_2)$ be two locations. The distance between $l_1$ and $l_2$ is defined as $d(l_1, l_2) = |i_1 - i_2| + |j_1 - j_2|$. Let $s = (s_1, \ldots, s_{n^2})$ be a state. Let $s_y = (g_1, \ldots, g_{n^2 - 1}, 0)$ be the goal state. The number of placed tiles is the largest $p$ such that $l_i = g_i$ 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 $NextTile(s)$.

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 $3 \times 3$ and $4 \times 4$ puzzles. It typically learned one or two macros while solving $5 \times 5$ puzzles, and reached quiescence while solving $6 \times 6$ problems without learning any new macros. Note that on average at least 85% of the training problems and at least half of the learning time (operator applications) were used just for reaching quiescence.
function on the performance of Micro-Hillary. We have conducted 100 learning sessions and testing sessions with each of the heuristic functions described in Section 4.1.2. 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 and a rigid order of tile placement 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.

4.4 Experiment 3: Learning macros in other domains

<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>209,872 (184,712)</td>
<td>34.7 (9.83)</td>
<td>66.3 (12.9)</td>
<td>10.39 (0.8)</td>
<td>11 (0)</td>
<td>212.7 (7.5)</td>
</tr>
<tr>
<td>Reduction</td>
<td>303,827 (184,173)</td>
<td>48.07 (12.12)</td>
<td>70.7 (12.6)</td>
<td>13.8 (0.37)</td>
<td>15 (0)</td>
<td>222.3 (7.3)</td>
</tr>
<tr>
<td>Spiral</td>
<td>266,488 (87,523)</td>
<td>48.17 (9.8)</td>
<td>70 (10.8)</td>
<td>13.96 (0.19)</td>
<td>13 (0)</td>
<td>214.2 (5.6)</td>
</tr>
<tr>
<td>Manhattan</td>
<td>7,175,066 (2,657,433)</td>
<td>145.8 (31.5)</td>
<td>342.5 (72.3)</td>
<td>104 (5.2)</td>
<td>23 (0)</td>
<td>1432 (841)</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 parentheses.

We have applied Micro-Hillary to the other domains specified in Section 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 domains 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, this amounts to most of the resources used by Micro-Hillary.

It is interesting to look at the macros learned in the grid domain. Most of the macros have a structure of `S...S` or `W...W`, where `S` stands for south, `W` for west, `N` for north, and `S` and `N` are equal in number. 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.
• **Domain:** Most of the experiments described here were performed in the eight-puzzle domain. However, we have tried Micro-Hillary in several other domains:

- **15-puzzle:** 4 × 4 sliding tile puzzle.
- **10-cannibals:** The cannibals and missionaries problem [19]. To make it non-trivial, we use 10 cannibals and 10 missionaries instead of the usual 3 and 3. The boat can be used by one or two persons.
- **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 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>Maximum macro length</th>
</tr>
</thead>
<tbody>
<tr>
<td>265872 (104712)</td>
<td>34.7 (9.8)</td>
<td>66.3 (12.9)</td>
<td>10.39 (1.8)</td>
<td>11 (0)</td>
</tr>
</tbody>
</table>

Table 1: The learning resources for the eight-puzzle domain averaged over 100 learning sessions. The numbers in parentheses 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</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, after learning, 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. During the testing phase, Micro-Hillary never encountered local minima. Thus, according to definition 1, all the sets of macros learned by Micro-Hillary were complete. This was also the case with the rest of the experiments described in this paper.

### 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 an exhaustive search and learning take place. The next experiment tests the effect of the heuristic
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 reaches quiescence and halts. Since the problems are generated by random sequences, we repeat each learning session 100 times. We test each resulting 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 sufficiently difficult. When in testing mode, we do not allow Micro-Hillary to acquire new macros.

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 principal 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 overly affected 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 may 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, as several applications of basic operators are required. Therefore, the principal 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. Note that we count every application, including those which fail.

- **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 resources consumed during the learning process until the learning system decides that it has learned enough. The common measure for 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 significantly affects the learning resources. A higher value for this parameter increases resources consumption, 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 maximum length of a macro.

4.1.2 Independent variables

There are many independent variables that affect 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.
All the operators are **reversible**. If there is an operator connecting state \( s_1 \) to state \( s_2 \) then there is also an operator connecting \( s_2 \) to \( s_1 \): \( \forall s_1, s_2 \in S \exists o \in O [s_2 = o(s_1)] \rightarrow \exists o' \in O [s_1 = o'(s_2)] \). 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. Exhaustive search methods, such as Breadth First (BFS) or Iterative Deepening (ID), guarantee finding an escape route if such a route exists. 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 waste much effort to ensure that the route to the goal state is optimal.

We are looking for a search method that will find solutions quickly if the heuristic function is sufficiently accurate, but also guarantees finding 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 \( B \rightarrow \infty \), Beam-search becomes Best-first, while Limited BFS becomes BFS. ILB 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 \( k + bi \), where \( k \) is some constant, \( b \) is the branching factor and \( i \) is the iteration number. This scheme guarantees that ILB will perform 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 indicate the direction of the escape route, ILB 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 it is beneficial to use macros.

### 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 accept a state and return either another state or \( \emptyset \).
3. A heuristic function that estimates the distance between two states.

### 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 Up, Down, Left and Right, indicating the direction in which the empty tile "moves". The heuristic function is similar to the one used by Iba[7]. The function contains three factors that are ordered lexicographically: the total number of tiles minus the count of tiles consecutively placed in 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 ordering the tiles row by row\(^3\). We begin by describing the experimental methodology used, and continue with a description of the experiments performed.

\(^3\) Section 5 specifies the exact definition of this function for the general \( N \times N \) case.
predefined limit. We say that Micro-Hillary is in *quiescence* when it solves $Q$ consecutive training problems without acquiring any new macro. Micro-Hillary quits when it detects quiescence.

When used for solving externally supplied problems, Micro-Hillary uses the union of the basic operators and learned macros. When encountering a local minimum, it searches for an escape route but does not acquire macros. An alternative version can perform on-line learning.

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.

```plaintext
procedure Micro-Hillary(BasicOps, h)
Macros ← {}
q ← 0
loop until q > Q
    ⟨sᵢ, sₒ⟩ ← GenerateTrainingProblem()
    q ← q + 1
    s ← sᵢ
    loop until s = sₒ
        sₘᵦᵢₐₓ ← s
        loop for o ∈ BasicOps ∪ Macros
            if $h(o(s), sₒ) < h(sₘᵦᵢₐₓ, sₒ)$ then $sₘᵦᵢₐₓ ← o(s)$
        if $sₘᵦᵢₐₓ = s$ then
            $O ← $ FindEscapeRoute($s, sₒ$)
            Macros ← Macros ∪ {O}
            q ← 0
        else
            s ← sₘᵦᵢₐₓ
        Return Macros

FindEscapeRoute($s, sₒ$): A procedure that finds a sequence of operators $O = o₁, \ldots, o_k$ such that
$h(o_k(\ldots o₁(s)\ldots), sₒ) < h(s, sₒ)$.
```

Figure 3: The Micro-Hillary algorithm

### 3.2 Selective experience: Generating solvable problems with increasing levels 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 other than the goal state is a global minimum. One way to achieve this is to train with solvable problems.
- **Training problems are generated in increasing order of difficulty.** This saves learning resources by acquiring as much knowledge as possible using easy problems.

One good 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 (unless there is only one goal state).**
Micro-Hillary, like most macro-learners, works with satisifying [24] search programs, i.e., programs whose goal is to find solutions as fast as possible regardless of 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[7] 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 \( e = \langle s_{\text{init}}, s_g \rangle \) be a training problem, and let \( T = \langle o_1, \ldots, o_n \rangle \) be a solution to \( e \). Let \( s = o_t(o_{t-1}(\ldots o_1(s_{\text{init}}))) \). The peak-to-better attention filter can be defined as:

\[
\langle o_j, o_{j+1}, \ldots, o_{j+k} \rangle \in f_{\text{db}}(T) \quad \Leftrightarrow \quad \forall o \in O [h(s_j, s_g) \leq h(o(s_j), s_g)]
\]

\[
\land \forall y [0 < y < k \rightarrow h(s_j, s_g) \leq h(s_{j+y}, s_g)]
\]

\[
\land h(s_{j+k}, s_g) < h(s_j, s_g).
\]

Thus, the filter passes any sub-sequence of the solution path whose first state is a local minimum, and whose last state is the first state in the sequence with a better heuristic value. 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.

**Definition 1** Let \( S \) be a set of states and \( O \) a set of operators. Let \( s_g \) be a goal state. Let \( h \) be a heuristic function. Let \( M \) be a set of macro-operators. We say that \( M \) is complete with respect to \( h \) and \( s_g \) if and only if

\[
\forall s \in S - \{s_g\}[\text{solvable}(s, s_g) \rightarrow \exists o \in O \cup M [h(o(s), s_g) < h(s, s_g)]].
\]

A complete macro set “smooths” the heuristic function by eliminating 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. A good way to detect local minima is to use a search method such as hill-climbing, which is susceptible to local minima. When trapped in a local minimum, Micro-Hillary searches for a way out to a state with a better heuristic value, and records this escape route as a macro-operator. Let \( Q \) be a
The attention filter: Let $G(e)$ be the set of all operator subsequences in the search tree generated when solving $e$. $f_{att}$ is a function that selects a subset of $G(e)$.

The acquisition filter, $f_{acq}$, is a function that decides whether to add a newly generated macro $m$ to the macro set $M$. $f_{acq}(m, M) \in \{M, M \cup \{m\}\}$. The definition can be extended to a new set of macros.

The retention filter, $f_{ret}$, is a function that deletes a subset of the macro set. $f_{ret}(M) \subseteq M$.

The utilization filter, $f_{util}$, is a function that gets the current state $s$ and the current problem $p$, and decides what part of the macro set $M$ should be passed on to the search strategy $\varphi$. $f_{util}(s, p, M) \subseteq M$.

Let $M$ be the current macro set and $E$ the set of training examples. The macro set $f_{util}(s, p, f_{ret}(f_{acq}(f_{att}(G(f_{exp}(E)), M))))$ is the set available to the problem solver at state $s$, while solving problem $p$, after having learned the next training example.

The general 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 often 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 sub-sequences connecting 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, however, is that the learned route leads into another local minimum. In the following section we present an alternative attention filter that avoids this fault.

3 Micro-Hillary

In this section we discuss Micro-Hillary – a particular instantiation of the architecture described above that is both very simple, yet powerful enough to perform efficient learning in a large class of problem domains.
2 Background: Selective macro-learning

Let $S$ be a finite set of states. Let $O$ be a finite 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 $(s_i, s_g)$, where $s_i \in S$ is called the initial state and $s_g \in S$ is called the goal state\(^2\). A solution is a sequence of operators $(o_1, \ldots, o_k)$, such that $o_k(\ldots o_1(s_i)\ldots) = s_g$. A problem $p = (s_i, s_g)$ is solvable if there exists a solution for $p$ (specified by $\text{solvable}(s_i, s_g)$). A search strategy $\varphi$ is a function that takes a problem $p$ and a set of operators $O$ and repeatedly applies operators to states until it either finds a solution to $p$ or fails because of resource limitation or exhaustion of the space.

A macro-operator is a sequence of operators $m = (o_1, \ldots, o_m)$. A macro-operator $m$ is applied to a state $s \in S$ by applying its basic operators in a sequence:

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

Given a solvable problem $p$, a search strategy $\varphi$ and a set of operators $O$, we define $\text{Cost}(\varphi(p, O))$, the cost of solving $p$ using $\varphi$ and $O$, as the number of operator applications performed by $\varphi$ until a solution is found. The utility of a set of macros $M$, with respect to a problem $p$, a search strategy $\varphi$ and a set of operators $O$, is defined as

$$U_{\varphi, O}(M) = \text{Cost}(\varphi(p, O)) - \text{Cost}(\varphi(p, O \cup M)).$$

Thus, the utility of a set of macro-operators with respect to a given problem and search strategy is the time saved by using these operators. 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 criteria 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 the search. Macro learning has a negative utility when used for optimizing search [3].

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 $M$ will be negative, as the added macros also increase the branching factor. To increase the utility of macro-learning, we should consider the following questions:

- 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 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:

The experience filter, $f_{exp}$, is a function that returns the next training problem $e \in E$.

\(^2\)The formalization could have used a goal predicate that returns true for states that are goal states. For simplicity we assumed a single specified goal state.
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 increasing order of 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 efficiently solve puzzles of any size.

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. Most 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 them, a macro-operator is defined as 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 speedup learning algorithm that is powerful and general, yet simple as well.

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 discusses and sums up Micro-Hillary's strengths and weaknesses, in comparison to other macro-learning algorithms.

---

*This is a revised version of report CS9617

\(^1\) 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 had already been used by Iba, Wogulis and Langley\([8]\).