Hierarchical Heuristic Search Revisited

Robert C. Holte, Jeffery Grajkowski, and Brian Tanner

University of Alberta, Computing Science Department,
Edmonton, Alberta, Canada
{holte, grajkows, btanner}@cs.ualberta.ca

Abstract. Pattern databases enable difficult search problems to be solved very quickly, but are large and time-consuming to build. They are therefore best suited to situations where many problem instances are to be solved, and less than ideal when only a few instances are to be solved. This paper examines a technique - hierarchical heuristic search - especially designed for the latter situation. The key idea is to compute, on demand, only those pattern database entries needed to solve a given problem instance. Our experiments show that Hierarchical IDA* can solve individual problems very quickly, up to two orders of magnitude faster than the time required to build an entire high-performance pattern database.

1 Introduction

Pattern databases were introduced [3, 4] as a method for defining a heuristic function to be used by heuristic search algorithms such as A* [9] and IDA* [14]. They have proved very valuable. For example, they are the key breakthrough enabling Rubik's Cube to be solved optimally [15], they have advanced the state of the art of solving the sequential ordering problem [11], and have enabled the length of solutions constructed using a macro-table to be very significantly reduced [10]. They have also proven useful in heuristic-guided planning [6].

A pattern database is defined by a goal state and an abstraction, ϕ, that maps the given state space, S, to an abstract state space $\phi(S)$. The states in $\phi(S)$ are called abstract states or patterns. A pattern database is a lookup table with an entry for each pattern – the entry for pattern P is the distance in $\phi(S)$ from P to the goal pattern, $\phi(goal)$. Given a pattern database, the heuristic value, $h(s)$, for a state $s \in S$ is computed by looking up the entry for $\phi(s)$ in the pattern database. Because ϕ is an abstraction, $h(s)$ is guaranteed to be an admissible, monotone heuristic [12].

A pattern database is built by finding a shortest path to the goal pattern for every pattern in $\phi(S)$. Typically this is done by running a breadth-first search backwards from the goal pattern until $\phi(S)$ is fully enumerated.

Building an entire pattern database as a preprocessing step has two disadvantages. The first is the time it takes to build the pattern database. For example, the "7-8" additive pattern database for the 15-puzzle in [7] takes approximately 3 hours to build and the high-performance pattern database for (17,4)-TopSpin in [8] takes approximately 1 hour to build. The second disadvantage is the size of the pattern database. In solving a single problem, only a tiny fraction of the pattern database is needed, and therefore most of the memory allocated for the pattern database, and the time needed to build it, is wasted.

Both disadvantages disappear, to some extent, if the same pattern database is used to solve many problem instances. For example, if 3 million 15-puzzle instances are solved using the "7-8" pattern database, the majority of its entries would be needed and the time to build the pattern database would amount to less than 10% of the total solution time.

On the other hand, there are circumstances in which the cost of building an entire pattern database cannot reasonably be amortized over a large number of problem instances. The obvious such circumstance is when only one or a few problem instances need to be solved, such as when building a macro-table [10], or when solving multiple sequence alignment problems [17]. In this case the time to build the pattern database will dominate the time to solve the problems.

Another circumstance in which the cost of building a pattern database cannot be amortized is when there are many instances to solve but it is impossible to use the same pattern database to solve them because they have different goals (and no simple transformation is possible), or because the operators or their costs have changed. As will be shown below it can also happen that, even though it is possible to use the same pattern database for all the problem instances, it is advantageous, time-wise, to use a different, custom-selected pattern database to solve different instances.

In this paper we examine a technique - hierarchical heuristic search - that aims to minimize the time and space overhead of using a pattern database by computing only those entries of the pattern database that are actually needed to solve a given problem instance. The idea of on-demand calculation of pattern database entries by hierarchical heuristic search was introduced in [12]. The abstraction technique there was so costly, in terms of both time and space, that it needed to be amortized over a large number of problem instances and therefore offered little or no advantage over pattern databases. The starting point for the present paper is the observation that the abstraction technique used for pattern databases requires negligible space and time, and therefore raises the possibility of realizing the great potential advantages of hierarchical heuristic search over pattern databases for solving individual problem instances. In addition to using a different abstraction technique, the present work also uses IDA* as its basic search procedure, whereas [12] used A*.

This paper reports several experiments with Hierarchical IDA* (HIDA*). The first shows that even if one abstraction, somewhat arbitrarily chosen, is used to solve all problem instances for a given state space, an average instance can be individually solved from scratch by HIDA* in minutes, compared to the one or more hours it takes to build a high-performance pattern database. Subsequent experiments show that in some state spaces a substantial additional speedup can be obtained by using multiple or customized abstractions.

2 The Hierarchical IDA* Algorithm

Pseudocode for Hierarchical IDA* (HIDA*) is given in Figure 1. The lines highlighted in bold font indicate the differences from normal IDA*. To improve performance our actual implementation is somewhat more complex, but this figure captures the central ideas.

The defining characteristic of hierarchical heuristic search is seen in the $h(s, goal)$ function in Figure 1. To compute a heuristic for state s, a recursive call to the search algorithm is made to compute the exact distance between the abstraction of s, $\phi(s)$, and the abstraction of the goal state, $\phi(goal)$. Search at an abstract level is guided by distances computed at an even more abstract level (in the figure the symbol ϕ is used to indicate the function that moves from the current level to the next more abstract level; an alternative notation would have had a different symbol, ϕ_i, for each level).

```
$HIDA*\ (start, goal)$
  $bound \leftarrow h(start, goal)$
  Repeat until $goal$ is found:
    $bound \leftarrow DFS(start, goal, 0, bound)$
  For all states s on the solution path:
    cache[s] ← distance from s to goal
    mark cache[s] as an exact distance
--------------------------------------------------
$DFS(s, goal, g, bound)$
  If $s == goal$: exit with success
  $g \leftarrow g + 1$
  $newbound \leftarrow \infty$
  Iterate over $x \in$ successors($s$):
  // P-g caching
    cache[x] ← max(cache[x],bound-g,h(x,goal))
    f ← g + cache[x]
  // Optimal path caching
    If (f == bound) and (cache[x] is an exact distance):
      exit with success
    If $f \leq bound$: $f \leftarrow DFS(x, goal, g, bound)$
    If $f < newbound$ : $newbound \leftarrow f$
  Return $newbound$
--------------------------------------------------
$h(s, goal)$
  If at the top abstraction level, return 0
  If cache[$\phi$(s)] is not an exact distance:
    HIDA*($\phi$(s),$\phi$(goal))
  Return cache[$\phi$(s)]
```

Fig. 1. Pseudocode for Hierarchical IDA*

Whenever the exact distance from a state to the goal is determined at any level other than the base level (the original search space), it is stored in a cache so that it need not be recomputed if it is required again. This is done by the bold lines of pseudocode in the HIDA* function in Figure 1.

Exact distances to the goal stored in $cache[x]$ for abstract state, x, are actually used for two different purposes. First, $cache[x]$ is used as the value for $h(s, goal)$ for any less abstract state, s, for which $\phi(s) = x$. Second, $cache[x]$ is used in the optimal path caching method that was introduced, for A*, in [12]. If x is reached by a path of length g during a search within x's abstract level and $cache[x]$ is an exact distance to goal, it is treated as if the goal had been reached by a path of length $g + cache[x]$. See the bold lines after the "Optimal path caching" comment in the DFS function in Figure 1.

In addition to storing exact distances to the goal, the cache is also used to store estimated distances to the goal generated by the "P-g" caching technique that was introduced, for A*, in [12]. "P-g" caching improves (increases) the heuristic values for abstract states that are expanded during a search, thereby improving the efficiency of subsequent searches that happen to reach these states. One possible implementation of "P-g" caching for HIDA* is shown by the bold lines after the "P-g caching" comment in the DFS function in Figure 1.

In all experiments the memory used for the cache was limited to 1 Gigabyte. The implementation of the cache in hierarchical search is less efficient than the hash table used to implement pattern databases because it is not known ahead of time which entries, or even how many entries, will be put into the hierarchical search cache[1]. By contrast, the exact set of patterns that will index a pattern database is known ahead of time. This is enormously beneficial in terms of space because a perfect hash function (collision free, no gaps) can be used, meaning that nothing identifying the pattern needs to be stored as part of an entry. Pattern database entries therefore only contain distances, typically needing only one byte per entry. It is not possible to develop a perfect hash function for the hierarchical search cache, so the cache must store a unique identifier for each pattern along with its distance, which increases the size of an entry very substantially (e.g. from one byte to eight for our 15-puzzle implementation). Not having a perfect hash function also slows down access, since collisions can occur and must be detected and resolved.

On a modern PC (AMD Athlon, 2.1GHz) our code generates approximately 4.5 million nodes per second at the base level and 1.5 million nodes per second at the abstract levels. The difference in speed is because the cache operations are done at each abstract level but not at the base level.

[1] This is true only of the lower levels in the abstraction hierarchy. For the upper levels it is virtually certain that almost all possible entries will be generated. For example, in the 15-puzzle experiment in the next section over 90% of the possible entries were generated at each of levels 4-8 in Table 1.

3 State Spaces Used in the Experiments

Four state spaces are used in these experiments: the 15-puzzle, a novel variant called the Macro-15 puzzle, (17,4)-Topspin, and the 14-Pancake puzzle. For each state space 100 randomly generated solvable instances were used for testing.

The 15-puzzle is included in our experiments because it is a standard benchmark for heuristic search methods. It is not actually a good example of the circumstances in which to use HIDA* because a very good, efficiently computed heuristic (Manhattan Distance) is known for it, and with modern search methods individual problems can be solved from scratch very quickly [1].

The Macro-15 puzzle is a novel variation on the 15-puzzle inspired by the fact that in the physical puzzle it takes the same effort to slide any partial row or partial column of tiles one position towards the blank as it takes to slide a single tile. Thus, in the 4x4 Macro puzzle used here there are 6 possible moves in every state (because there are 3 tiles in the same row as the blank and 3 tiles in the same column as the blank, and any tile in the same row or column as the blank can be the endpoint of the group that is moved). We call this state space the Macro-15 puzzle because its additional moves are "macro" moves in the 15-puzzle. Solution lengths in the Macro-15 state space for the 100 standard test problems used for the 15-puzzle [14] range from 27 to 38 with the median and average solution length being 32. By contrast in the normal 15-puzzle these problems' solution lengths range from 41 to 66, with a median and average length of 53. Note that Manhattan Distance is not an admissible heuristic in this space, and additive pattern databases [7] cannot be used for it.

The (N,K)-TopSpin puzzle has N tokens arranged in a ring. The tokens can be shifted cyclically clockwise or counterclockwise. The ring of tokens intersects a region K tokens in length which can be rotated to reverse the order of the tokens currently in the region. In our encoding we ignore the cyclic shifts and only count reversals. Therefore the only moves are to reverse any K adjacent tokens, where adjacency is defined cyclically. We used $N = 17$ and $K = 4$, but the effective number of tokens is only 16 because one of the tokens is used as a fixed reference point and therefore is effectively stationary. [2] shows that all 16! states are reachable. In order to reduce the number of transpositions, if two moves act on non-intersecting sets of positions we force them to be done in a particular order. This reduces the branching factor to 8.

In the N-Pancake puzzle [5] a state is a permutation of N tokens $(0, 1, ...N - 1)$. A state has $N - 1$ successors, with the k^{th} successor formed by reversing the order of the first $k + 1$ positions of the permutation $(1 \leq k < N)$. We used N=14, which has 14! states. Although this space is smaller than the others its much larger branching factor makes it roughly the same difficulty to search.

4 Using One Abstraction Hierarchy for All Problems

Hierarchical search requires an abstraction hierarchy – a sequence of abstractions defining the mappings from one level of abstraction to the next. In our state

Table 1. "Default" Abstraction Hierarchy for the 15-puzzle and the Macro-15 puzzle

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

spaces an abstraction is defined by mapping some of the tiles/tokens to a "don't care" symbol. If the tokens mapped to "don't care" by ϕ_1 are a superset of the tokens mapped to "don't care" by ϕ_2 then the space defined by ϕ_1 is an abstraction of the space defined by ϕ_2. Therefore a sequence of successively more abstract spaces can be defined by partitioning the tokens into groups, $G_1, G_2, ..., G_n$ and defining ϕ_i as the abstraction in which all tokens in groups $G_1, G_2, ..., G_i$ are mapped to "don't care".

In the experiment in this section the same abstraction hierarchy is used for all problem instances of each state space. These "default" abstraction hierarchies were not carefully chosen, they were among our initial thoughts for each state space. The abstraction hierarchy used for the 15-puzzle and the Macro-15 puzzle is shown in Table 1, which is read from bottom to top, because the higher rows represent the higher levels of abstraction. The bottom row ("base" level) indicates which tile(s) each column is referring to. Each other row indicates how the tiles are mapped at a certain level of abstraction, the level being indicated by the number in the first column. For example the row with 1 in the first column shows that the first level of abstraction is defined by mapping tiles 1-7 to "don't care" (indicated by • in the table). The patterns at this abstract level are the possible ways of placing the blank and the 8 remaining tiles (tiles 8-15) in the 16 positions of the 15-puzzle. At the most abstract level (level 8) all the tiles except 15 are mapped to "don't care". The patterns at this abstract level are the possible ways of placing the blank and tile 15 in the 16 positions of the 15-puzzle.

The abstraction hierarchy used for the 14-Pancake puzzle is identical except that it has only 14 tokens and therefore only seven abstract levels. Note that "token 1" has the most volatile home position – the token in that position is changed by every operation. This abstraction therefore abstracts the tokens in volatile positions and retains the identity of tokens that can be placed in their home positions and then left unmoved by a judicious choice of operators.

The abstraction hierarchy for (17,4)-TopSpin starts by abstracting tokens 1-9 to define the first abstraction level and abstracts one token per level thereafter in increasing order (10, then 11, then 12 etc.). Token 0 is the token that is used as a reference and never moves - it is never abstracted.

Table 2. CPU times (in seconds) using the default abstraction hierarchy. "all" means clear every cache between every problem instance. "1-3" means clear levels 1-3 between every instance, never clear the higher levels. "none" means never clear any cache - this is possible only if all entries at all levels fit within the 1 Gigabyte memory limit

State Space	Clear	Avg	Max	Median
15-puzzle	all	642	20,227	93
	1-3	596	17,827	71
Macro-15	all	132	910	84
	1-3	101	959	58
(17,4)-TopSpin	all	766	3,068	680
	none	162	1,875	89
14-Pancake	all	88	405	54
	none	31	326	4

Table 3. Comparison of the average number of cache entries (in thousands) stored by HIDA* to the number of entries (in thousands) in the full pattern database (PDB size) for the first level of abstraction. The last column expresses the average number of Level 1 cache entries as a percentage of the pattern database size

State Space	Total	Level 1	PDB size	%
15-puzzle	10,931	2,657	4,151,347	0.06
Macro-15	7,402	787	4,151,347	0.02
(17,4)-TopSpin	8,143	3,423	57,657	5.9
14-Pancake	1,208	229	17,297	1.3

Table 2 shows the average, maximum, and median CPU times over the 100 test problems for each state space. "All" in the "Clear" column indicates that all caches are cleared completely between each problem instance. This simulates solving a single problem instance in isolation with no preprocessing or prior problem-solving experience with the abstraction. The "median" column in the "all" rows shows that the majority of individual problems can be solved in a few minutes, compared to the hour or more it takes to build high-performance pattern databases. Across the entire experiment only three problem instances, all for the 15-puzzle, take more than an hour to solve.

Table 3 shows the number of cache entries created by HIDA*, on average. "Total" is the total number of cache entries at all levels. "Level 1" is the number of cache entries for the first level of abstraction. The rightmost column shows that this is a small fraction of what would be stored in the pattern database for this abstraction – well under one-tenth of one percent for the 15-puzzle and Macro-15 puzzle.

If a small batch of problem instances with the same goal is to be solved using the same abstraction, HIDA*'s caches need not be cleared between instances: the cache entries created for one instance will be correct for others. The abstractions being used for (17,4)-TopSpin and 14-Pancake are sufficiently coarse-grained that 1 Gigabyte allows all caches at all levels to be made large enough to hold all possible entries. Therefore, it is never necessary to clear any cache. The "none" rows in Table 2 show the CPU times for these puzzles if the 100 test problems are solved as a batch in the random order in which they were generated with no clearing of the caches. Batch-solving substantially reduces the average, median, and the maximum solution times. A batch of 18 average problems can be solved in the time it takes to build a high-performance pattern database for (17,4)-TopSpin.

For the 15-puzzle and Macro-15 puzzle abstractions levels 1, 2, and 3 are sufficiently fine-grained that they must be cleared at some point in order to solve the 100 test problems. The "1-3" rows in Table 2 show the CPU times that result if only the caches at levels 1, 2, and 3 are cleared between each problem instance. This produces a modest reduction in the time to solve problems. Batches of approximately 17 15-puzzle problems and batches of 97 Macro-15 problems can be solved in the time it takes to build a high-performance pattern database for these puzzles.

The fact that HIDA* solves tens of problem instances, on average, in the time required to build a high-performance pattern database does not rule out the possibility that HIDA* would be outperformed by a smaller pattern database when only one or a few problem instances are to be solved. To see why this will not happen, in general, consider the Macro-15 puzzle. To build the complete pattern database for the first level abstraction in Table 2 takes 2.73 hours. The pattern database based on the second level abstraction is much smaller and takes only 452 seconds to build. This is still substantially more than the time it takes HIDA* to solve a single Macro-15 problem on average. A pattern database based on an even coarser abstraction would take fewer than 100 seconds to build but would provide such poor heuristic guidance for the base level search that the time to solve a problem would far exceed HIDA*'s. As a general rule, a pattern database that can be fully computed in a time less than HIDA*'s will offer much weaker guidance than HIDA*'s first level of abstraction and therefore have higher problem-solving runtimes.

5 Multiple Abstractions

[13] shows that for a fixed amount of memory, taking the maximum of several smaller pattern databases outperforms using a single large pattern database. This technique can be applied to hierarchical heuristic search by using multiple abstractions instead of just one at one or more of the abstraction levels. However, it is not obvious if this will lead to improved performance for hierarchical heuristic search because, unlike in the pattern database studies, the time required to calculate the entries for the additional abstractions is now counted as part of the execution time.

Table 4. CPU times (in seconds) using multiple abstractions

State Space	Clear	Avg	Max	Median
15-puzzle	all	131	1,047	78
	none	31	364	7
Macro-15	all	88	392	65
	none	17	162	5
(17,4)-TopSpin	all	982	2,394	919
	none	88	927	50
14-Pancake	all	95	329	72
	none	10	197	2

In this section we define multiple abstractions only at the first level of abstraction. Each of those abstractions then has only one abstraction above it, created by abstracting one additional tile/token, and those abstractions each have only one above them, etc. As in [13], in computing $h(s)$ the calculation of the maximum value given by the different abstractions is aborted if a value is returned that is large enough to ensure that $f(s) = h(s) + g(s)$ exceeds IDA*'s current depth bound.

For (17,4)-TopSpin and the 14-Pancake puzzle, two first-level abstractions are used: the default abstraction from the previous section and a complementary one. For (17,4)-TopSpin the complementary abstraction abstracts tokens 8-16 at the first level (the default abstraction abstracts tokens 1-9) and then abstracts one additional token per level in decreasing order (7, then 6, then 5 etc.). Similarly, the complementary abstraction for the 14-Pancake puzzle abstracts tokens 7-13 (the default abstracts tokens 0-6) and then abstracts one additional token per level in decreasing order. The results are shown in Table 4. The "all" and "none" rows in Table 4 have the same meaning and can be directly compared to the corresponding rows in Table 2. The multiple abstractions in this experiment increase the CPU time for solving individual problems in isolation (the "all" rows) but significantly decrease the time for solving small batches of problems (the "none" rows).

For the 15-puzzle and the Macro-15 puzzle, the default abstractions fill available memory, so there is no room available for additional abstractions. Instead, we use four first-level abstractions that are each considerably smaller than the default. One of them abstracts 8 tiles, the others abstract 9 tiles (the default abstracts only 7 tiles). Comparing the "all" rows in Table 4 to the corresponding rows in Table 2 we see that individual problems are solved much more quickly using multiple abstractions. The average time for solving individual problems, 131 seconds for the 15-puzzle and 88 seconds for Macro-15, is two orders of magnitude less than the time required to build a high-performance pattern database for these puzzles. The multiple abstractions used here are sufficiently coarse-grained that 1 Gigabyte is enough to create perfect hash tables for all caches at all levels. This enables batches of problems to be solved without ever clearing any caches, producing the results shown in the "none" rows in Table 4. In both spaces the entire batch of 100 test problems is solved in well under an hour.

6 Customized Abstractions

The first experiment showed that individual problem instances can be solved quickly using a default abstraction hierarchy. This section shows that this performance can be significantly improved, in some state spaces, by tailoring the abstraction hierarchy to each instance. In this experiment all caches are cleared between each instance.

For the 15-puzzle and Macro-15 puzzle we use a simple method for creating the customized abstraction hierarchy. The key idea is to choose a good order in which the tiles will be abstracted. The first level abstracts the first seven tiles according to the order, and each level after that abstracts the next tile in the ordering. The tile ordering we used is based on each tile's Manhattan distance, i.e. the number of moves required to get the tile from its position in the start state to its goal position. The tiles are sorted in increasing order of this distance, with ties broken arbitrarily.

78 of the 15-puzzle problems are solved more quickly with the customized abstraction than with the default abstraction. The 22 that are slower are all "easy" problem instances. The hardest problem instances have all been sped up substantially by using a customized abstraction; some now run almost 50 times faster than before. The longest-running instance now takes 1,517 seconds. The average time to solve a problem instance is reduced from 642 to 99 seconds, and the median time drops from 93 to 42 seconds. These results are significantly better than the results with generic multiple abstractions (row "all" in Table 4).

The Macro-15 puzzle also benefits significantly from custom abstractions, although not as much as the 15-puzzle. Average solution time is reduced to 99 seconds from 132, and the median drops to 64 seconds from 84.

For (17,4)-TopSpin and the 14-Pancake puzzle numerous methods of custom abstraction were explored. For the 14-Pancake puzzle none outperformed the default abstraction. For (17,4)-TopSpin we identified an abstraction[2] that was significantly better than the default for certain problems. However, there was no obvious rule to decide which abstraction to use on a given problem instance. Our solution was to compute $h(start)$ using each of the abstractions and then use the abstraction that gave the higher value to solve the instance. This is a rather expensive selection rule, because the cache entries created when computing $h(start)$ using the first abstraction have to be cleared in order to compute $h(start)$ using the second abstraction, and then have to be recomputed if the first abstraction is chosen for solving the problem. This overhead must be in-

[2] The first level abstracts tokens 8-16, subsequent levels eliminate one additional token in decreasing order (7 then 6 then 5 etc).

cluded in the total time to solve a problem, and doing so leaves the average and median solution times virtually the same as always using the default abstraction. However, this method does reduce the time to solve the most difficult problem from 3068 to 2304 seconds.

7 Related Work

[17] observes that the pattern database entry for $\phi(s)$ is not needed if

$$d(\phi(s), \phi(goal)) + d'(\phi(start), \phi(s)) > U$$

where $d(x, y)$ is the true distance from x to y, $U \geq d(start, goal)$, and $d'(x, y) \leq d(x, y)$. Given an upper bound, U, on the solution cost in the original space and a function, $d'(x, y)$, that never overestimates distance in the abstract space, [17] runs $A*$ backwards from the abstract goal state until it has enumerated all abstract states $\phi(s)$ with $d(\phi(s), \phi(goal)) + d'(\phi(start), \phi(s)) \leq U$. The resulting table of abstract distances is called a space-efficient pattern database (SEPDB).

If the abstraction used for the SEPDB is used to define HIDA*'s first abstract level and search at this level is guided by the same heuristic in both systems, HIDA*'s first-level cache will always contain a subset of the SEPDB entries. The SEPDB hash table, like HIDA*'s caches, must store pattern identification information along with the distance information when there is not sufficient memory to store all possible entries for the full pattern database. Thus, SEPDB's memory needs cannot be less than HIDA*'s.

To see precisely how SEPDB's memory requirements compare to HIDA*'s we ran SEPDB using the default abstractions for our state spaces. The second abstract level was used as the heuristic to guide SEPDB's A*. The resulting SEPDB is therefore the counterpart of HIDA*'s first level cache. To make the comparison as favourable to SEPDB as possible the upper bound it was given was the actual solution length for each problem instance. The results are shown in Table 5. SEPDB has at least 32% more entries than HIDA*'s first level cache, even when given a perfect upper bound. If this upper bound is increased to be just one larger than the optimal value, the size ratios for Macro-15, (17,4)-Topspin and 14-Pancake increase to 3.52, 1.75, and 4.05 respectively. For the 15-puzzle the next larger meaningful upper bound is two larger than the optimal value. In this case, the average size of the SEPDB rises to 13,515,134, which is 5.08 times larger than HIDA*'s first-level cache.

The CPU times for SEPDB and HIDA* cannot be compared in this experiment because the second level of abstraction is computed by HIDA* but assumed to be given, without computation, by SEPDB. To make a fair time comparison, a hierarchical version of SEPDB would be needed. Hierarchical SEPDB might possibly run faster than HIDA*, but, as this experiment has shown, it would require more memory.

"Reverse Resumable A*" [16], like SEPDB, computes pattern database entries by backwards A* search at the abstract level. Unlike SEPDB, it stops when

Table 5. Comparison of the sizes of HIDA*'s first-level cache and the corresponding SEPDB

Space	HIDA*	SEPDB	Ratio
15-puzzle	2,657,511	6,430,269	2.42
Macro-15	787,664	1,309,100	1.66
(17,4)-TopSpin	3,423,746	4,534,162	1.32
14-Pancake	339,328	467,237	1.38

it closes the abstract start state. If an entry is needed during the base level search that has not been generated, A* search at the abstract level is resumed until the entry is generated. This produces a subset of the SEPDB, and avoids the need for an upper bound on solution length, but requires additional memory for preserving A*'s Open list so that A* can be resumed.

8 Conclusion

This paper has shown that hierarchical heuristic search can solve individual problems very quickly, up to two orders of magnitude faster than building a high-performance pattern database. Hierarchical heuristic search is therefore preferable to pattern databases when only one or a few problem instances with the same goal are to be solved. On the other hand, pattern databases are preferable when many problem instances with the same goal are to be solved. In cases where it is unclear which method to use, the two can be used in parallel. While the pattern database is being built, hierarchical heuristic search can be applied to the problem instances, perhaps with a time limit for each problem instance. Sometimes all the instances will be solved before the pattern database is complete. If this does not happen, the remaining problems can be solved quickly using the pattern database.

Acknowledgments

This research was supported in part by a Discovery grant and a postgraduate scholarship from the Natural Sciences and Engineering Research Council of Canada and an iCORE postgraduate fellowship. We thank Rich Korf for supplying the base IDA* implementation from which our code was developed.

References

1. Andreas Auer and Hermann Kaindl. A case study of revisiting best-first vs. depth-first search. *Proceedings of the Sixteenth European Conference on Artificial Intelligence (ECAI-04)*, pages 141–145, 2004.

2. Ting Chen and Steve Skiena. Sorting with fixed-length reversals. *Discrete Applied Mathematics*, 71(1-3):269–295, 1996. Special volume on computational molecular biology.
3. J. C. Culberson and J. Schaeffer. Efficiently searching the 15-puzzle. Technical report, Department of Computer Science, University of Alberta, 1994.
4. J. C. Culberson and J. Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.
5. Harry Dweighter. Problem e2569. *American Mathematical Monthly*, 82:1010, 1975.
6. S. Edelkamp. Planning with pattern databases. *Proceedings of the 6th European Conference on Planning (ECP-01)*, pages 13–24, 2001.
7. A. Felner, R. E. Korf, and S. Hanan. Additive pattern database heuristics. *Journal of Artificial Intelligence*, 21:1–39, 2004.
8. Ariel Felner, Uzi Zahavi, Jonathan Schaeffer, and Robert Holte. Dual lookups in pattern databases. *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05)*, 2005.
9. P.E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.
10. I. T. Hernádvölgyi. Searching for macro operators with automatically generated heuristics. *Advances in Artificial Intelligence - Proceedings of the Fourteenth Biennial Conference of the Canadian Society for Computational Studies of Intelligence (LNAI 2056)*, pages 194–203, 2001.
11. I. T. Hernádvölgyi. Solving the sequential ordering problem with automatically generated lower bounds. *Proceedings of Operations Research 2003 (Heidelberg, Germany)*, pages 355–362, 2003.
12. R. C. Holte, M. B. Perez, R. M. Zimmer, and A. J. MacDonald. Hierarchical A*: Searching abstraction hierarchies efficiently. *Proc. Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 530–535, 1996.
13. Robert C. Holte, Jack Newton, Ariel Felner, and Ram Meshulam. Mutiple pattern databases. *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS-04)*, pages 122–131, 2004.
14. R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.
15. R. E. Korf. Finding optimal solutions to Rubik's Cube using pattern databases. *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 700–705, 1997.
16. David Silver. Cooperative pathfinding. *Proceedings of the First Annual Conference on Artificial Intelligence and Interactive Entertainment (AIIDE-05)*, 2005.
17. R. Zhou and E. A. Hansen. Space-efficient memory-based heuristics. *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04)*, pages 677–682, 2004.