Searching Without a Heuristic: Efficient Use of Abstraction

Bradford Larsen, Ethan Burns, Wheeler Ruml
Department of Computer Science
University of New Hampshire
Durham, NH 03824 USA
blarsen, eaburns, ruml at cs.unh.edu

Robert C. Holte
Department of Computing Science
University of Alberta
Edmonton, Alberta T6G 2E8 Canada
holte at cs.ualberta.ca

Abstract

In problem domains where an informative heuristic evaluation function is not known or not easily computed, abstraction can be used to derive admissible heuristic values. Optimal path lengths in the abstracted problem are consistent heuristic estimates for the original problem. Pattern databases are the traditional method of creating such heuristics, but they exhaustively compute costs for all abstract states and are thus usually appropriate only when all instances share the same single goal state. Hierarchical heuristic search algorithms address these shortcomings by searching for paths in the abstract space on an as-needed basis. However, existing hierarchical algorithms search less efficiently than pattern database constructors: abstract nodes may be expanded many times during the course of a base-level search. We present a novel hierarchical heuristic search algorithm, called Switchback, that uses an alternating direction of search to avoid abstract node re-expansions. This algorithm is simple to implement and demonstrates superior performance to existing hierarchical heuristic search algorithms on several standard benchmarks.

Introduction

It is easy to find search problems for which no easily computable, informative heuristic is known. For example, the classic Manhattan distance heuristic for the sliding tile puzzle becomes significantly less effective if one merely specifies that a single tile in the puzzle cannot move. While trivial, this example captures the fact that the elegant combinatorial structure underlying real-world problems is often obscured by annoying application-specific constraints.

For such problems, informative heuristics can often be derived by solving an *abstraction* of the problem optimally and then using the solution costs in the abstraction as heuristic estimates for the original problem. An *abstraction* of a search problem is a simplification of that problem such that (1) the minimum the cost between any two nodes in the abstraction is no greater than the minimum cost between the corresponding nodes in the original problem, and (2) all goal states in the original problem are goal states in the abstracted problem (Prieditis 1993).

The traditional way of utilizing abstraction in heuristic search is to construct a *pattern database* (Culberson and Schaeffer 1998). A pattern database is a look-up table that maps abstract problem states to admissible heuristic cost-to-go estimates for the original problem states. These estimates can then be used to guide a heuristic search algorithm such as A* (Hart, Nilsson, and Raphael 1968). While this technique can provide powerful heuristic values for many domains, pattern databases have significant drawbacks. Pattern databases are typically generated by performing an exhaustive uniform-cost search in the abstract space in the reverse direction, starting from the goal state. This preprocessing not only limits the database to a single goal state (or an easily enumerable number of goal states), but it also may take hours to compute for a sufficiently fine-grained abstraction. Another drawback to pattern databases is that they can consume large amounts of memory, which will no longer be available for storing search nodes. Typically, only a tiny fraction of the total pattern database is consulted during a search, while the remainder merely contributes to memory overhead (Holte, Grajkowski, and Tanner 2005).

Hierarchical heuristic search is an alternative to pattern databases that, in effect, computes the look-up table lazily (Holte et al. 1996). That is, when a hierarchical heuristic search computes the heuristic estimate between a new node and the goal node, the nodes are abstracted and a heuristic search is executed at the abstract level. The cost of the optimal solution in the abstract space is then used as the heuristic estimate at the lower level. This abstract-and-search technique can be applied recursively until some maximum level of abstraction is reached, where any search space is so small that an uninformed search algorithm can be used. Because heuristic values are computed only as needed, hierarchical heuristic search can solve many instances in the time it takes to construct an entire high-performance pattern database. Therefore, hierarchical heuristic search is more appropriate than pattern databases when only a few instances must be solved or if the goal changes between instances (Holte, Grajkowski, and Tanner 2005).

A naive implementation of hierarchical heuristic search is hopelessly inefficient, as each required heuristic estimate computation executes a search at the next level of abstraction. Furthermore, because many searches at any abstract level may be conducted, nodes at an abstract level may be expanded many times during the course of the base-level search. In order to make hierarchical heuristic search practical, several caching techniques have been devised (Holte et al. 1996; Holte, Grajkowski, and Tanner 2005). However, previously proposed caching techniques can be difficult to implement efficiently and do not eliminate the problem of abstract node re-expansions.

In this paper, we introduce a new hierarchical heuristic search algorithm called Switchback that expands nodes at most once. Switchback searches in alternating directions at each level of the abstraction hierarchy using the A* algorithm. This new algorithm is complete and admissible, and is both simpler to implement and has less overhead than previous approaches. It is applicable to domains in which the predecessors of a node can be easily computed and in which useful (i.e., informative and relatively easy to compute) abstractions can be defined. We empirically evaluate the effectiveness of this algorithm on several benchmark domains, finding that it consistently expands fewer nodes than previous techniques and expands nodes at a faster rate, leading to an overall speed-up of up to a factor of four compared to existing hierarchical search algorithms.

Previous Work

Hierarchical A* In Hierarchical A* (HA*), a forward search algorithm based on A* is used at each level of the abstraction hierarchy (Holte et al. 1996). It uses three caching schemes to improve performance, all of which depend on the fact that the goal node at each abstract level remains the same for any single base-level search.

First, the heuristic estimate function $h(n)$ checks a cache to see if the heuristic value for n is already known; if not, a search is executed at the next level of abstraction, and the cost of the solution found is stored in the cache and returned. If the search is at the highest level of abstraction the heuristic is computed using the function $\epsilon(n)$, which evaluates to zero if n is the goal, and the cost of the cheapest operator applicable to n otherwise. The second caching technique is called *optimal path caching*, where the goal node is fast-tracked onto the open list if an optimal path is known for the node whose heuristic is being computed.

The third caching technique is *P-g caching*. Let P be the cost of an optimal path to the goal. Then for any n, $P \leq g(n) + h^*(n)$. This can be rearranged to derive the admissible heuristic $P - g(n)$. Using $P - g(n)$, the cached value of each node that was expanded during the search can then possibly be increased.[1] (Note that for nodes along an optimal path to the goal, $P - g(n) = h^*(n)$.) Until the true cost-to-goal $h^*(n)$ is known, n may be expanded during multiple searches at its abstraction level.

Hierarchical IDA* Hierarchical IDA* (HIDA*) is a variation of HA* that uses iterative deepening A* (Korf 1985) at each level (Holte, Grajkowski, and Tanner 2005). Like HA*, HIDA* also uses optimal path caching and P-g caching. HIDA* also includes full duplicate detection, which only adds a constant additional memory overhead beyond the other caching techniques.

Holte, Grajkowski, and Tanner (2005) demonstrated that HIDA* can solve dozens of instances in the amount of time required to construct a single high-performance pattern database. When only a few instances need to be solved or if the goal state may change between instances, it is more effective to use hierarchical heuristic search. However, pattern databases have efficiency advantages over HA* and HIDA*, because they can be constructed using retrograde analysis, which expands each node at most once. HA* and HIDA* may expand nodes multiple times, due to the occurrence of many forward searches at abstract levels of the hierarchy.

Hierarchical Cooperative A* Hierarchical Cooperative A* (HCA*) is a two-level hierarchical heuristic search algorithm for multi-agent grid path planning in dynamic worlds (Silver 2005). For its abstraction, HCA* allows agents to pass through each other and assumes they take up no space.

At its abstract level, HCA* uses Reverse Resumable A* (RRA*), a variant of A* that searches backward from the goal node. RRA* is initialized by inserting the abstract goal node into its open list. When a heuristic value for a node n at the base level is requested, RRA*'s closed set is checked for the abstraction of n. If found, its g value is returned. Otherwise, RRA* resumes its search, expanding nodes in f order until the abstraction of n is expanded.

Because a consistent heuristic causes A* to expand nodes in cheapest-path-first order (Nilsson 1980), the first time a node is expanded by RRA*, the node's optimal cost to the goal is known. RRA*'s strategy of checking the closed list will still result in optimal solutions to the base-level search. HCA* uses the Manhattan distance heuristic for its RRA* search, which is admissible and consistent for two-dimensional grid path planning.

Instance-Specific Pattern Databases Felner and Adler (2005) describe instance-dependent pattern databases. Unlike a standard PDB, an instance-dependent pattern database is generated specifically for a given problem instance using only a subset of the abstract states that will be required for an A* search. To accomplish this, RRA* is used, pausing when the start node is expanded to allow search at the ground-level to continue. If the forward search ever reaches a state that has no entry in the PDB, then the RRA* search is resumed until the desired abstract state is expanded or a memory limit is reached.

Similar work from Zhou and Hansen (2004) deals with instance-specific pattern databases, but without the use of RRA*, instead using *focused heuristics* (i.e. abstraction functions) that are computed only for states that are abstractions of states that could be reached in A* in the original problem. By using a hierarchy of focused heuristics, Zhou and Hansen (2004) demonstrated that multiple sequence alignment problems can be solved up to 100 times faster than with the traditional pairwise heuristic.

[1] Merely backing up the minimum f value of a node's children leads to incorrect results in undirected state spaces because the start state changes.

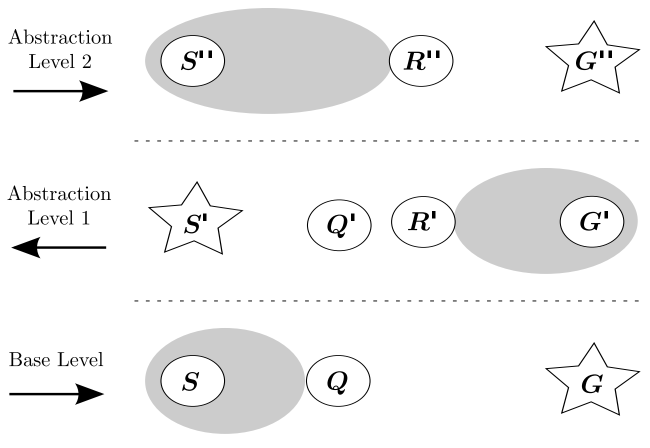

Figure 1: An example of the operation of the Switchback algorithm in a 3-level abstraction hierarchy.

The Switchback Algorithm

Switchback is a new hierarchical heuristic search algorithm that addresses the node re-expansion shortcomings of HA* and HIDA*. It does this by alternating forward and backward searches from one level to the next, which is analogous to switchbacks on a mountain path. Figure 1 illustrates a simple example of a three-level abstraction hierarchy—the base level and two abstract levels. The task is to find an optimal path from node S to node G. At the initialization phase, node S is inserted into the open list at the base level; node G', the abstraction of the base-level goal node, is inserted into the open list at level 1; and S'', the second-level abstraction of the base-level start node, is inserted into the open list at level 2.[2]

Switchback uses A* at each level of the abstraction hierarchy. Suppose the search at the base level is progressing, but a path to the goal has not yet been found. (The gray ovals represent the closed list at each level.) To generate node Q, we need to compute its heuristic estimate, $h(Q, G)$. To do this, we abstract the node, giving Q'. We then look at the next abstraction level in the hierarchy—level one—and continue the search there until Q' is found in the level one closed list. Note that the search at level one proceeds in the opposite direction from the base level, from G' towards S'. Once Q' is expanded at level one, its g value, $g(G', Q')$, is used as the heuristic estimate for Q at the base level. That is, $h(Q, G) = g(G', Q')$.

During the search at abstraction level one, suppose the search encounters a node R' for which it needs to compute the heuristic estimate, $h(R', S')$. To compute this, the node R' is abstracted, yielding R'', and another search is invoked at the next level in the hierarchy from S'' to G''. When R'' makes it into the closed list of the search at level two, its g value is used as the heuristic estimate at level one. That is, $h(R', S') = g(S'', R'')$. In this example, abstraction level two is the highest level of abstraction, and so there we use

[2]In the case that the goal at a level is an easily enumerable set of nodes rather than a single node, the abstractions of all the goal nodes would be inserted into the open list of the next higher level.

```
SWITCHBACK()
01. $open$ ←array of length $height_\phi$ of empty open lists
02. $closed$ ←array of length $height_\phi$ of empty closed lists
03. for $i$ ←0 up to $height_\phi - 1$ do
04.     if $i$ is even then
05.         $g(s_{start}) \leftarrow 0$; $h(s_{start}) \leftarrow 0$
06.         insert $\phi(i, s_{start})$ into $open_i$
07.     else $g(s_{goal}) \leftarrow 0$; $h(s_{goal}) \leftarrow 0$
08.         insert $\phi(i, s_{goal})$ into $open_i$
09. $result$ ←RESUME(0, $open$, $closed$, $s_{goal}$)
10. if $result \neq$ NULL then return EXTRACT-PATH($result$)
11. return NULL

RESUME($i$, $open$, $closed$, $s$)
12. if $s$ is in $closed_i$ then return $s$
13. while $open_i$ is not empty do
14.     $n$ ←remove node from $open_i$ with lowest $f$
15.     if $i$ is even then $children$ ←$succs(n)$
16.     else $children$ ←$preds(n)$
17.     for each $c$ in $children$ do
18.         if $c$ is in $closed_i$ then
19.             if $g(c) < g(n) + cost(n, c)$ then continue
20.             else $g(c) \leftarrow g(n) + cost(n, c)$
21.                 if $c$ is not in $open_i$ then insert $c$ onto $open_i$
22.                 continue
23.         $h(c)$ ←HEURISTIC($i$, $open$, $closed$, $c$)
24.         $g(c) \leftarrow g(n) + cost(n, c)$
25.         insert $c$ into $open_i$ and $closed_i$
26.     if $n = s$ then return $n$
27.     return NULL

HEURISTIC($i$, $open$, $closed$, $s$)
28. if $i = height_\phi - 1$ then return $\epsilon(s)$
29. $n$ ←lookup $\phi(i + 1, s)$ in $closed_{i+1}$
30. if $n \neq$ NULL then return $g(n)$
31. $r$ ←RESUME($i + 1$, $open$,$closed$, $\phi(i + 1, s)$)
32. if $r =$ NULL then return $infinity$ else return $g(r)$
```

Figure 2: Pseudo-code for the Switchback algorithm.

the epsilon heuristic (i.e. $\epsilon(n) = 0$ if n is a goal, and the cost of the cheapest operator applicable to n otherwise).

The benefit of Switchback's alternating direction of search is that every single expansion at an abstract level will contain the optimal path length from the expanded node back to the abstraction of the goal node of the level below. If this value is ever required it can be simply retrieved by a table lookup rather than requiring further search in the abstract space.

In Detail

Pseudocode for Switchback is given in Figure 2. In this algorithm, ϕ is an abstraction function that takes an integer $i \in 0..height_\phi - 1$ and a search state s, and returns s abstracted to the level i. (Note that $\phi(0, s) = s$.) As the algorithm executes, it maintains a set of $height_\phi$ open and closed lists, one for each level of the hierarchy. Initially, the closed lists are all empty, and the open lists each contain a single node, chosen to coincide with the alternating forward/backward direction of the search (lines 1–8). At level 0 (the base level), Switchback performs a forward A* search from the start node s to the goal node g.

The RESUME procedure, when searching for a node s, first checks its closed list for s. If s appears in the closed list, it is returned without searching further (line 12). Otherwise, the search is resumed until s is expanded (lines 13–27). In this latter case, the level at which the search is currently executing is used to determine whether the successors or the predecessors of n are generated, again, to coincide with the alternating forward/backward direction of the search. Because of the alternating search direction, Switchback is only appropriate for domains where both the predecessors and the successors of a node can be easily computed. As we will discuss later, Switchback needs to expand a node at most once. Note, however, that we check for duplicates at node generation time, so it may be necessary to update the best known path to a node on *open* (lines 19–22).

The HEURISTIC procedure implements the heuristic estimate used during the search. If the search is occurring at the highest level of the abstraction hierarchy, the ϵ heuristic is used (line 28). If the search is occurring at a level other than the highest, the closed list for the next level is checked for the next abstraction of the given node. If it is found in the closed list, its g value is used for the heuristic estimate (line 30). (In other words, this is a cache hit.) Otherwise, the search is continued at the next higher level, and the result of that search is used as the heuristic estimate (lines 31–32).

It is important to note that goal testing (line 26) is done in RESUME only after a node has been expanded (lines 15–22). This is necessary to ensure completeness of the algorithm. It is necessary to expand a node before testing whether it is the goal node because it is possible that further search at an abstract level may be required after the goal of any individual RESUME invocation has been expanded. If goal testing were done before expansion, then when expanding a node, it could be closed before its successors or predecessors (depending on the direction of the search) are generated, and hence, those nodes might be lost.

Optimality and Efficiency

Here we show how the alternating direction of search at each level of the Switchback algorithm allows the g value of a node (i.e., the cost from the root of the current search to the node) at any level to be used as an admissible heuristic estimate at the level below.

First, we show in Lemma 1 (modified from Holte et al. (1996)) that homomorphic abstractions can be used to derive consistent heuristics.

Lemma 1 *If distances between nodes in a ground space are no smaller than distances between corresponding nodes in an abstract space (i.e., given an abstraction), a heuristic estimate derived by optimal search in the abstract space will be a consistent and admissible heuristic estimate in the ground space.*

Proof: Consistency requires $h(a) - h(b) \leq c^*(a, b)$, where $c^*(a, b)$ is the cheapest path from a to b. If $\phi(x) = x'$, this can be rewritten as $c^*(a', g') - c^*(b', g') \leq c^*(a, b)$ or equivalently as $c^*(a', g') \leq c^*(a, b) + c^*(b', g')$. To show this, note that $c^*(a', g') \leq c^*(a', b') + c^*(b', g')$ because they are shortest paths. Because the abstraction does not inflate distance, $c^*(a', b') \leq c^*(a, b)$ and we have $c^*(a', g') \leq c^*(a, b) + c^*(b', g')$ as desired. □

Theorem 1 *The heuristic used by Switchback is consistent and admissible at every level of the abstraction hierarchy.*

Proof: The proof is by induction on the level l.

Base case: The search at the top level, $l = height_\phi - 1$, uses the consistent and admissible epsilon heuristic.

Inductive case: Make the inductive hypothesis that the heuristic used at level $l+1$ is both consistent and admissible. Because A* is used at each level, and because A* using a consistent and admissible heuristic computes optimal g values for nodes when they are first expanded (Nilsson 1980), paths found between the start node and any expanded node at level $l + 1$ are optimal.

At level l, to compute the heuristic estimate $h(n, t)$, the value $g(\phi(l + 1, t), \phi(l + 1, n))$ is used. Because the search at level $l + 1$ occurs in the opposite direction of the search at level l, and because g values at level $l + 1$ are optimal, we can apply Lemma 1, concluding that the heuristic used at level l is both consistent and admissible.

By induction, a consistent and admissible heuristic is used by Switchback at every level of the abstraction hierarchy. □

Corollary 1 *Assuming that a solution exists and that all cycles in the search graph have non-negative cost, Switchback terminates with an optimal solution at every level of the abstraction hierarchy.*

Proof: This follows from the fact that Switchback uses A* with an admissible heuristic at each level, using the proof of A*'s optimality (Nilsson 1980). □

Because the heuristic used in the A* search at each level is consistent, a node at any level will be expanded at most once for any single base-level search. The optimal path and P-g caching techniques required for reasonable performance of both HA* (Holte et al. 1996) and HIDA* (Holte, Grajkowski, and Tanner 2005) are unnecessary for an efficient Switchback implementation.

Corollary 2 *At any level of the abstraction hierarchy, Switchback expands a node at most once.*

Proof: Because the heuristic is consistent and admissible, the A* search at every level will first expand a node through an optimal path (Nilsson 1980), and therefore it will never reopen a node that was previously expanded. □

Experimental Evaluation

To gain a more concrete sense of the behavior of the Switchback algorithm, we implemented Switchback along with HA*, HIDA*, and A* and tested them on four benchmark domains: the Macro 15-Puzzle, the Glued 15-Puzzle, the 14-Pancake Puzzle, and the traditional 15-Puzzle. These are standard test domains or are variants of standard ones, and have varying characteristics: the 15-Puzzle has deep solutions with low branching factor, the Pancake Puzzle has shallow solutions with high branching factor, and the Glued 15-Puzzle has asymmetries.

In each domain, the same abstraction hierarchy—derived using modern homomorphic abstraction techniques—was used by all three hierarchical algorithms. For the sliding tile puzzle variants, we used the 9-level instance-specific custom abstraction hierarchy proposed by Holte, Grajkowski, and Tanner (2005): the tiles are sorted in increasing order of how close to their goal positions they are, as measured by Manhattan distance. At the first abstraction level, the first 7 such tiles have their identities obscured; for the next 7 levels, an additional tile is obscured. This attempts to preserve the most relevant information when abstracting the problem.

All experiments were run a dual quad-core Intel Xeon X5550 machine with 48 GB of memory, using no time limit and a 47 GB memory limit. All our code was implemented in single-threaded C++, and built with optimization using the GNU toolchain.

The results from these experiments are shown in Table 1. The column labeled "Solved" shows the number of instances (out of 100) that were solved. The statistics in the remaining columns were all computed over instances that were solved by every algorithm in a major row. We performed a Wilcoxon signed-rank test to determine significance; entries in the table that are in bold represent values whose means were not significantly different ($p \geq 0.05$) from the best value in the column. The next seven columns show the mean, maximum and median CPU time in seconds, and the mean, maximum and median number of nodes generated, followed by the number of nodes generated per second, all in millions.

The column labeled "Stored (M)" shows the mean peak number of nodes held in memory, in millions. This column is indicative of the total memory usage of the algorithm, as all four algorithms tested cache a copy of every unique node generated during the search.

All three of the hierarchical algorithms potentially perform search at the next level when computing the heuristic for a node. If search is avoided by using a cached value, we record it as a cache hit. The final two columns report cache hit information for the hierarchical algorithms, at the base level, and averaged over the entire abstraction hierarchy. The cache hit ratio is an indicator of how many times hierarchical search was required: all other things being equal, a higher cache hit ratio is better. Table 1 shows that Switchback had a substantially higher cache hit ratio than the other hierarchical heuristic search algorithms.

Macro 15-Puzzle

The macro sliding tile puzzle is like the traditional sliding tile puzzle except that multiple tiles in the row or column where the blank is located may be moved in one step. We used the standard 100 solvable instances due to Korf (1985).

The Manhattan distance heuristic is inadmissible in the Macro 15-Puzzle domain, as a single step can move up to three tiles. With A*, we used Manhattan distance heuristic divided by three as the heuristic. Using this heuristic, A*

			CPU Time (s)			Nodes Generated (M)					Cache Hit Rate	
Domain	Algorithm	Solved	Mean	Max	Median	Mean	Max	Median	Num/s	Stored (M)	Base	Total
Macro 15-Puzzle	Switchback	100	**161.77**	**1127.51**	**110.97**	**182.52**	**948.44**	**145.25**	1.31	55.84	**100**	**100**
	HA*	100	708.19	4647.71	386.68	236.70	1074.32	187.98	0.43	21.76	44	84
	HIDA*	100	223.39	1202.47	154.84	350.66	1746.42	251.88	**1.62**	**20.34**	21	5
Glued 15-Puzzle	Switchback	98	**5.52**	**70.33**	**2.12**	**6.33**	**76.99**	**2.43**	**1.14**	4.96	**99**	**100**
	HA*	98	137.67	3286.74	15.88	11.73	99.05	5.58	0.36	**4.33**	8	73
	HIDA*	98	20.91	172.35	8.26	19.93	131.72	9.96	**1.20**	4.77	12	3
	A*	98	71.33	927.59	2.33	54.77	597.66	3.52	**1.31**	44.63	-	-
14-Pancake Puzzle	Switchback	100	**22.47**	**113.91**	**14.57**	**28.64**	**122.65**	**20.40**	**1.35**	12.82	**100**	**100**
	HA*	100	167.80	1772.24	87.36	37.66	156.41	29.26	0.36	**12.36**	23	85
	HIDA*	100	89.33	400.66	66.99	86.57	311.13	69.77	1.07	22.56	3	1
15-Puzzle	Switchback	100	82.98	1422.49	34.92	64.58	713.98	36.10	1.06	42.72	**99**	**100**
	HA*	100	831.38	12034.20	343.77	109.87	1221.97	68.08	0.21	**19.65**	6	71
	HIDA*	100	194.44	2563.26	107.81	196.72	2158.00	117.13	1.11	22.39	27	7
	A*	100	**30.55**	**815.15**	**4.29**	**30.56**	**532.36**	**7.72**	**1.59**	**27.12**	-	-

Table 1: Results from the Macro 15-Puzzle, Glued 15-Puzzle, 14-Pancake Puzzle, and the 15-Puzzle.

was unable to solve any instance within the 47 GB memory limit, so we have not included it in the results table. The heuristics derived through abstraction and used by the hierarchical algorithms proved to be far more effective.

Table 1 shows these results in the first major row. Here, we see Switchback using the least mean, median, and max CPU time, and generating the fewest nodes. HA* was much slower than either Switchback or HIDA*. HIDA* had the fastest node generation rate.

Glued 15-Puzzle

The Glued 15-Puzzle is a variation of the traditional 15-Puzzle in which a random tile is glued to the board in its goal position. During the search it is not possible to move this tile; hence, the solution path must work its way around an extra constraint. While the Manhattan distance heuristic is admissible for the glued tiles puzzle, it can be less informative than on the traditional 15-Puzzle.

The instances were generated by randomly selecting a tile to be glued to the board and then performing a one-million-step random walk backward from the goal position. The average solution cost for the 98 instances that were solved is 53.18 with a standard deviation of 9.34. The average difference between the solution cost and the Manhattan distance estimate of the initial state is 21.27. In contrast, for the normal 15-Puzzle, this mean difference is only 16.

Table 1 shows these results in the second major row. Here, Switchback is superior in all measurements except for node generation rate and mean peak number of nodes stored. The CPU time measurements for Switchback are nearly 4 times faster than HIDA*, the next-fastest hierarchical search algorithm. On average, A* takes longer and requires more search effort to solve these problems than Switchback does; however, A*'s performance sits somewhere between HIDA*'s and HA*'s. In this domain, the mean peak number of nodes held in memory by the hierarchical algorithms are statistically indistinguishable, but all much less than the number retained by A*.

14-Pancake Puzzle

The Pancake Puzzle is a permutation puzzle where a sequence of numbers (pancakes) must be arranged in increasing order by reversing prefixes of the sequence. Because flipping the single top pancake does not change the sequence, the branching factor is $N - 1$, where N is the total number of pancakes. For the following experiments, we used $N = 14$. Since all states in the pancake puzzle are reachable, the 100 instances used in the following experiments were generated by simply selecting random permutations of the numbers $1..14$.

Table 1 shows the results for the 14-Pancake Puzzle in the third major row. Switchback was superior in terms of CPU time and nodes generated. HIDA* held the most nodes in memory, on average, in this domain.

We used the heuristic $h(n) = 0$ with A* for this domain because we did not know of a better admissible heuristic. A* was unable to solve any of the Pancake Puzzle instances within the 47 GB memory limit using this heuristic, and so we did not include its results in the table.

15-Puzzle

We also experimented with the standard 15-puzzle domain. The hierarchical algorithms all used the same abstraction hierarchy that was used with the previous tiles puzzle variants, and A* used the Manhattan distance heuristic.

This is a domain where hierarchical search—at least using the custom abstraction hierarchies introduced by Holte, Grajkowski, and Tanner (2005)—is not a good candidate because the Manhattan distance heuristic is rather informative on the 15-Puzzle for how cheaply it is computed. Nonetheless, it is a staple in the literature, and so we include its results here.

The results are found in the last major row of Table 1. A* performed best, and we were surprised to see that all 100 instances could be solved within the 47 GB memory limit. Switchback performed better than HIDA* and HA*, and had the least maximum node generations, but, on average, held the highest peak number of nodes in memory.

Problem Difficulty

Figure 3 compares the hierarchical algorithms on an instance-by-instance basis in each domain. The x-axis in each plot shows the solution length of a given instance, which is an algorithm-independent proxy for problem difficulty. The y-axis shows the difference in CPU times between the plotted algorithms and Switchback, in seconds. Except for the Macro 15-Puzzle plot, the y-axis is shown in a logarithmic scale. From this set of plots, we can see that the Switchback algorithm has an increasing advantage over HA* and HIDA* as problem difficulty increases. On the Macro 15-Puzzle, the advantage of Switchback over HIDA* is less obvious due to the relatively poor performance of HA*. Although HIDA* was slightly faster on a few instances, Switchback's advantage generally follows an increasing trend.

Discussion

In our experiments, HA* expanded nodes slower than any of the other algorithms. To determine the cause, we conducted experimental runs on a few instances using Naive HA*, i.e., HA* with P-g caching and optimal path caching disabled. We observed a factor of three increase in the node generation rate. Caching overhead proves to be significant.

To confirm that HA* and HIDA* suffer from re-expansion of abstract nodes, we modified our implementations of these two algorithms to keep track of the number of times each node was expanded. We chose five random instances from each of the four domains, and ran both algorithms on these instances. In all but one case, the percentage of expansions that were re-expansions were well above fifty percent for both algorithms. Abstract node re-expansion is indeed a problem for those two algorithms.

Because Switchback essentially computes a pattern database as needed, traditional pattern database techniques, such as maximizing over multiple pattern databases (Holte et al. 1996), using additive state space abstractions (Yang et al. 2008; Felner, Korf, and Hanan 2004), or exploiting sym-

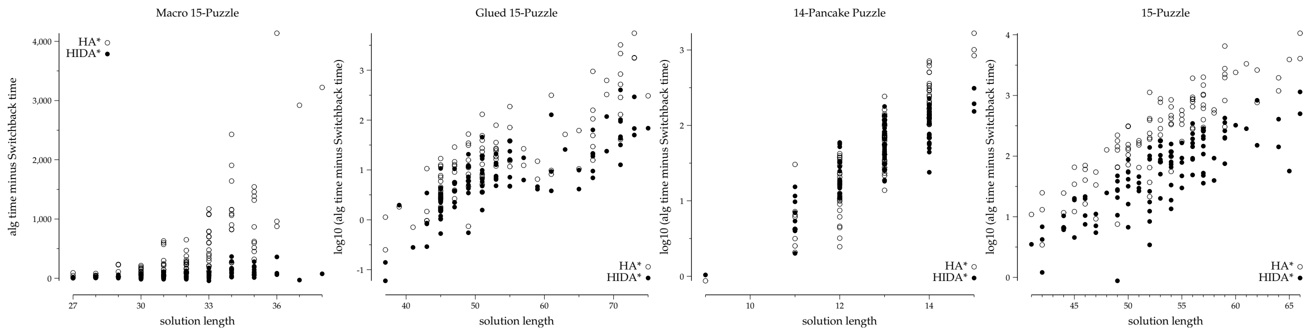

Figure 3: The advantage of Switchback as problem difficulty increases.

metry in abstractions (Culberson and Schaeffer 1998), could be applied to Switchback as well.

HIDA* is able to maintain its cache information between base-level instances with the same goal (Holte, Grajkowski, and Tanner 2005). Switchback, in contrast, would only be able to re-use half its cache information. If the start node changed, the cache information at each even (i.e. forward) search level would need to be discarded. Alternatively, if the goal node changed, the cache information at the odd (i.e. backward) search levels would need to be discarded.

We have described Switchback as an algorithm that performs a forward search at the base level. This is not strictly necessary—one could just as well start in the reverse direction. Kaindl and Kainz (1997) give a method of probing to determine which search direction appears more promising. This technique could be used with Switchback to dynamically decide upon the direction of search at the base level.

Conclusion

We have addressed a shortcoming of existing hierarchical heuristic search algorithms—namely, that abstract nodes can be expanded many times during one base-level search—through a novel generalization of backward search to a multi-level hierarchical setting. The resulting algorithm, Switchback, is simple to implement, depending upon no sophisticated caching techniques such as P-g caching or optimal path caching. Furthermore, Switchback outperforms both HIDA* and HA* on four domains, by up to a factor of four. In settings where an effective heuristic is not available, but a natural abstraction suggests itself and backward search is feasible, Switchback is a good choice due to its fast and efficient search performance.

Acknowledgments

We gratefully acknowledge support from the U.S. NSF (grant IIS-0812141), the DARPA CSSG program (grant HR0011-09-1-0021), and the Canadian NSERC.

References

Culberson, J. C., and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence* 14(3):318–334.

Felner, A., and Adler, A. 2005. Solving the 24-puzzle with instance dependent pattern databases. In Zucker, J.-D., and Saitta, L., eds., *Abstraction, Reformulation and Approximation*, volume 3607 of *Lecture Notes in Computer Science*, 248–260. Springer.

Felner, A.; Korf, R. E.; and Hanan, S. 2004. Additive pattern database heuristics. *Journal of Artificial Intelligence Research* 22(1):279–318.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics SSC-4* 2:100–107.

Holte, R. C.; Perez, M. B.; Zimmer, R. M.; and MacDonald, A. J. 1996. Hierarchical A*: Searching abstraction hierarchies efficiently. In *Proceedings of the National Conference on Artificial Intelligence*, 530–535.

Holte, R. C.; Grajkowski, J.; and Tanner, B. 2005. Hierarchical heuristic search revisited. In Zucker, J.-D., and Saitta, L., eds., *Abstraction, Reformulation and Approximation*, volume 3607 of *Lecture Notes in Computer Science*, 121–133. Springer.

Kaindl, H., and Kainz, G. 1997. Bidirectional heuristic search reconsidered. *Journal of Artificial Intelligence Research* 7:283.

Korf, R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27:97–109.

Nilsson, N. J. 1980. *Principles of Artificial Intelligence*. Tioga Publishing Co.

Prieditis, A. 1993. Machine discovery of effective admissible heuristics. *Machine Learning* 12:117–141.

Silver, D. 2005. Cooperative pathfinding. In Young, M., and Laird, J., eds., *Proceedings of the First Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2005)*, 117–122. AAAI Press.

Yang, F.; Culberson, J. C.; Holte, R.; Zahavi, U.; and Felner, A. 2008. A general theory of additive state space abstractions. *Journal of Artificial Intelligence Research* 32:631–662.

Zhou, R., and Hansen, E. A. 2004. Space-efficient memory-based heuristics. In *Proceedings of the Nineteenth National Conference on Artifical Intelligence (AAAI-04)*, 677–682. AAAI Press / The MIT Press.