Faster Optimal and Suboptimal Hierarchical Search

Michael J. Leighton and **Wheeler Ruml**
Department of Computer Science
University of New Hampshire
Durham, NH 03824 USA
mjr58, ruml at cs.unh.edu

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

Abstract

In problem domains for which an informed admissible heuristic function is not available, one attractive approach is hierarchical search. Hierarchical search uses search in an abstracted version of the problem to dynamically generate heuristic values. This paper makes two contributions to hierarchical search. First, we propose a simple modification to the state-of-the-art algorithm Switchback that reduces the number of expansions (and hence the running time) by approximately half, while maintaining its guarantee of optimality. Second, we propose a new algorithm for suboptimal hierarchical search, called Switch. Empirical results suggest that Switch yields faster search than straightforward modifications of Switchback, such as weighting the heuristic or greedy search. The success of Switch illustrates the potential for further research on specifically suboptimal hierarchical search.

Introduction

Hierarchical search is used to generate informed admissible heuristics via abstraction for problems where a good heuristic does not exist or is not known. It works by first creating an abstract, easier to solve, version of the original problem. Then when a heuristic is required for a state in the original problem, a search is executed in the abstraction until the distance of the abstract state to the abstract goal is known. The distance in the abstraction is then returned as the heuristic value for the state in the original problem. This same approach can be used to generate heuristics for the search at the abstract level, creating a hierarchy of abstractions that can be used to generate informed heuristics.

One alternative to using hierarchical search is to use a pattern database (Culberson and Schaeffer 1996). The difference is that pattern databases enumerate all possible abstract states before search and store them in a table (the pattern database). Then, generating a heuristic value for the original problem consists of a table look-up. One of the shortcomings of using pattern databases is that generating every state in the abstraction requires a considerable amount of pre-computation. However, Holte, Grajkowskic, and Tanner (2005) shared that only a small fraction of these states are required when solving an individual problem. In addition, the database must be recomputed each time the goal state changes. They must also be stored in memory during the search in order to produce acceptable performance. This means that fewer resources can be dedicated to solving the original problem.

Switchback (Larsen et al. 2010) is a state-of-the-art hierarchical search algorithm. We propose a simple modification to the Switchback algorithm, called Short Circuit, that reduces the number of node expansions required at each abstract level search. We present an empirical evaluation of Short Circuit that shows that on average it increases the performance of Switchback by approximately a factor of two.

Hierarchical search algorithms proposed to date have focused on finding optimal solutions. However, it is often the case that time or memory is limited such that no optimal solution can be found. In such cases one is sometimes willing to sacrifice solution quality for decreased CPU time or memory use. One obvious approach is to use a hierarchical search to generate heuristics for sub-optimal algorithms such as Weighted A* (Pohl 1973). We present an empirical evaluation showing that this approach does not result in the desired outcome. As we will show below, in many cases this can actually increase the amount of CPU time required to solve the problem.

We offer an alternative approach, that we call Switch, that uses suboptimal searches during the creation of the heuristic to boost performance. This approach only requires that a portion of the abstraction hierarchy be kept in memory at any one time, reserving resources for the base level search. Evidence suggests that this approach not only improves performance but also returns shorter solutions than simply using hierarchical search to generate a heuristic for a conventional suboptimal search algorithm.

We hope that Short Circuit and Switch will further popularize hierarchical search as a useful alternative when the pre-computation costs of a full pattern database are prohibitive. The success of Switch illustrates the potential for further research on suboptimal hierarchical search.

Previous Work

Because Short Circuit and Switch build on previous work, we briefly review relevant algorithms.

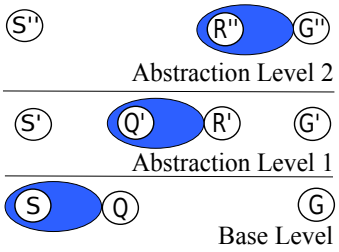

Figure 1: Hierarchical A* on a 3-level hierarchy.

Hierarchical A* Hierarchical A* (HA*) is a forward hierarchical search algorithm that performs an A* search at each level in the abstraction hierarchy (Holte et al. 1994). It requires an abstraction function $\phi_i(s)$ that maps a state s at level $i-1$ to a state at abstraction level i. Figure 1 illustrates the process. Conventional A* search is run on the original problem at the base level by removing a node with minimum f on open, generating its children, and querying their heuristic values. To calculate the heuristic value for a given node Q at the base level, an A* search is started from $Q' = \phi_1(Q)$ at the next level in the abstraction hierarchy. This search is terminated when the abstract goal state $G' = \phi_1(G)$ is reached. The length of the path found by this search is used as the heuristic value for Q. This technique is used recursively to generate heuristic searches in further abstract levels as well, resulting in an hierarchy of abstraction levels. The highest level of the hierarchy is a trivial space where search can be driven by the ϵ heuristic, which returns 0 for the goal state, and the cheapest cost operator otherwise.

Because a new A* search is started each time a heuristic for the base level search is required, nodes in the abstraction hierarchy could potentially be expanded many times. To prevent as much node re-expansion as possible, HA* uses three caching techniques. (We will later modify and use some of these techniques in the creation of our suboptimal algorithm.) The first caching technique is to store heuristic values for every node for which a full abstraction level search has been completed. This value will be returned if the node's heuristic is ever requested again. Second, when a solution from a given query node to the goal is found, the entire optimal path is placed into the cache. This technique is known as *optimal path caching*. Third, every node n that is expanded on the way from the query node to the goal node that does not lie along the optimal path is placed into the cache with the value of $P - g(n)$, where P is the optimal path length. This technique comes from the fact that, because P is optimal, $P \leq g(n) + h^*(n)$ for all nodes and hence $h(n) \geq P - g(n)$. This can potentially increase the heuristic value of a given node.

Switchback Switchback (Larsen et al. 2010) is a hierarchical search algorithm that changes the direction of search at every level of the hierarchy. This technique is used to prevent node re-expansion. Figure 2 gives an example of Switchback on a three level abstraction hierarchy. Search begins in the original problem using A*. The start node is expanded and Q is generated. To create a heuristic value

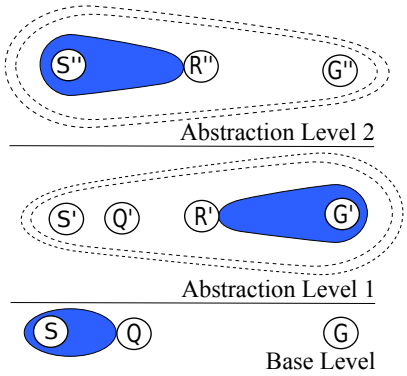

Figure 2: Switchback on a 3-level hierarchy.

for Q, an A* search at abstraction level one begins. This search proceeds in the opposite direction, from the goal state $G' = \phi_1(G)$ to $S' = \phi_1(S)$. R' is generated and a heuristic is required. This leads to a search at the second abstraction level from $S'' = \phi_2(S')$ to $G'' = \phi_2(G')$. Once R'' has been expanded, the optimal distance from S'' to R'' is known and used as the heuristic value for R'. When Q' is achieved, its distance from G' is used as the heuristic value for Q. This will continue until the original search expands G, and the solution is known.

The closed list at each level of a Switchback search will have optimal g values and search will alternate in direction as the abstraction level increases. Because g values are used as the heuristic for the level below, the entire closed list can be used as a cache. Once an abstraction level has completed a search from start to goal (or goal to start), it may need to continue searching if additional query nodes are requested and not found in the cache. Heuristic values always guide the search to the goal at the given level. After that goal has been achieved search continues in an uninformed manner. This will cause the search to expand f layers until the query node is reached. The dotted lines in Figure 2 represent how the search will progress at each level after the goal at that level has been achieved.

Optimal Search

One approach to solving problems optimally without a known heuristics is to use a pattern database. Holte, Grajkowskic, and Tanner (2005) show that hierarchical search can solve many problem instances before an initial pattern database can be constructed. For example, creating a "7-8" additive pattern database for the 15-puzzle takes approximately 3 hours. Hierarchical search algorithms can solve many problem instances in that time. This makes hierarchical search an attractive approach when only a few problem instances will be solved for a given goal state.

The purpose of search at abstract levels is to determine g values. At node generation time of an A* search, many nodes have their optimal g values. These nodes can be used to expand the size of the cache, and therefore decrease the number of subsequent searches. These nodes can also be used to exit early from, or "Short Circuit", abstract level

searches. The Short Circuit algorithm we introduce below uses a special case of the following theorem.

Theorem 1 *Assume $fmin$ is the node at the front of open and n is a goal node that has been generated but not expanded. Assume also that a consistent heuristic is being used. Then the cost of a solution returned by an A* search that stops when a goal is generated rather than waiting for it to be expanded is bounded by a sub-optimality factor of $g(n)/f(fmin) - h(n)$.*

Proof: Let $g^*(n)$ be the optimal g value for n. The sub-optimality of a goal node n can be measured using $g(n)/g^*(n)$. If an A* search exits during node generation only when $g(n)/g^*(n) \leq b$ then the solution is clearly within b of optimal. Given our assumptions and properties of A*, we have

$$\begin{aligned} f(fmin) &\leq f(n) \\ f(fmin) &\leq g^*(n) + h(n) \quad \text{by admissibility} \\ f(fmin) - h(n) &\leq g^*(n) \end{aligned}$$

Hence $f(fmin) - h(n)$ is a lower bound on $g^*(n)$ and $g(n)/f(fmin) - h(n)$ is an upper bound on the solution sub-optimality of a A* search that returns a solution as soon as it is generated. □

Short Circuit

Short Circuit is a simple modification of the Switchback algorithm. The difference between the two algorithms is in deciding when to return from abstract searches. Switchback does not return until the query node has been expanded, while Short Circuit returns as soon as the query node is known to have its optimal g value. This is done by checking to see if the query node has been found during node generation rather than node expansion. Theorem 1 gives a method for bounding the sub-optimality of a solution returned during node generation of an A* search. Short Circuit uses a special case of Theorem 1 where the bound is 1. Assume that Q is the query node and $fmin$ is the node with minimum f on open. Then, when $g(Q)/(f(fmin) - h(Q)) \leq 1 \equiv f(fmin) = f(Q)$, Q has its optimal g value. When this holds, search is stopped and $g(Q)$ is returned as the heuristic value.

This technique can also be used to add additional caching to Switchback. In a normal Switchback search, the closed list is checked for the query node before search is restarted. In Short Circuit the open list is checked as well. If the query node is found in open then the same optimality checks performed above can be used to prove that it has an optimal g and thus prevent additional search. Otherwise, search will be restarted as usual. Since Short Circuit returns only optimal values, the admissibility and consistency guaranteed by Switchback are maintained.

Figure 3 shows the pseudo-code for Short Circuit. The code is nearly identical to the original Switchback code with the exception of lines 14–16.[1] Adding Short Circuit to an implementation of Switchback is fairly straightforward.

[1]Line 12 fixes an error in the published Switchback pseudo-code, which omitted the check against *open*.

```
SHORTCIRCUIT()
01. open ← array of length height_φ of empty open lists
02. closed ← array of length height_φ of empty closed lists
03. for i ← 0 up to height_φ − 1 do
04.   if i is even then
05.     g(s_start) ← 0; h(s_start) ← 0
06.     insert φ(i, s_start) into open_i
07.   else g(s_start) ← 0; h(s_goal) ← 0
08.     insert φ(i, s_goal) into open_i
09. result ← RESUME(0, open, closed, s_goal)
10. if result≠ NULL then return EXTRACT-PATH(result)
11. return NULL

RESUME(i, open, closed, s)
12. if s is in closed_i and not in open_i then return s
13. while open_i is not empty do
14.   if s is in open_i
15.     fmin ← node from open_i with lowest f
16.     if g(s)/(f(fmin) − h(s)) ≤ 1 then return s
17.   n ← remove node from open_i with lowest f
18.   if i is even then children ← succs(n)
19.   else children ← preds(n)
20.   for each c in children do
21.     if c in closed_i then
22.       if g(c) < g(n) + cost(n, c) then continue
23.       g(c) ← g(n) + cost(n, c)
24.       if c is not open_i then insert c onto open_i
25.       continue
26.     h(c) ← HEURISTIC(i, open, closed, c)
27.     g(c) ← g(n) + cost(n, c)
28.     insert c into open_i and closed_i
29.   if n = s then return n
30. return NULL

HEURISTIC(i, open, closed, s)
31. if i = height_φ − 1 then return ε(s)
32. n ← lookupφ(i + 1, s) in closed_{i+1}
33. if n ≠ NULL then return g(n)
34. r ← RESUME(i + 1, open, closed, φ(i + 1, s))
35. if r = NULL then return ∞ else return g(r)
```

Figure 3: Pseudo-code for Short Circuit

Evaluation

In order to gain a better understanding of the performance of Short Circuit, we modified the original Switchback code from Larsen et al. (2010) and compared the two algorithms on the four domains used in their work. Each instance was tested on a dual quad-core Xeon running at 2.66 GHz with 48 GB of RAM. All instances were given unlimited running time and a memory limit of 47 GB.

Figure 4 shows a scatter plot for each of the domains tested. The x and y axes represent the Log_2 of the nodes expanded for each instance solved by Short Circuit and Switchback respectively. Each point on the chart represents a single problem instance. A solid black line is drawn on each plot to indicate the line $x = y$. Points to the left of this

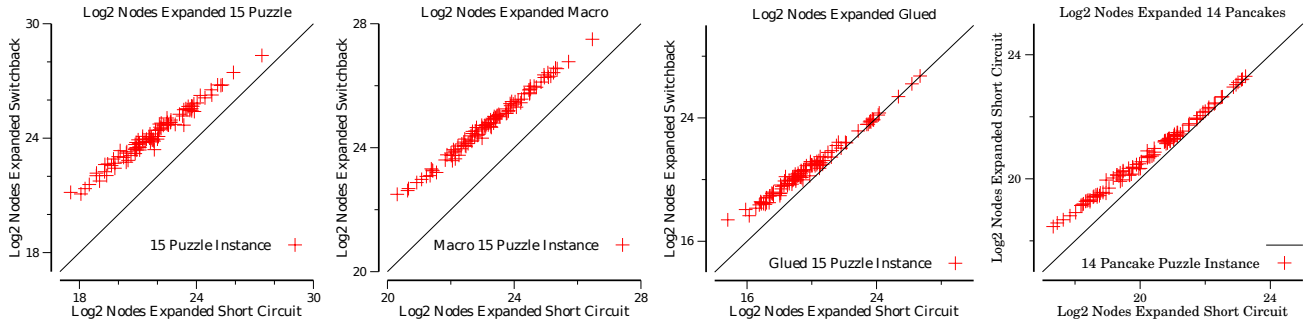

Figure 4: Log_2 of Node expansions on the 15 puzzle, glued 15 puzzle, macro 15 puzzle, and 14-pancake domains.

	Nodes Expanded		CPU Time	
Domain	Speedup	Std	Speedup	Std
15 Puzzle	5.49	2.05	5.18	1.68
Macro 15	2.98	0.47	3.27	0.41
Glued 15	2.11	0.85	1.86	0.54
14-Pancakes	1.45	0.31	1.46	0.29

Table 1: Average node expansion and CPU time speedup ratio for Short Circuit over Switchback.

line indicate that Short Circuit solved the instance with less node expansions than Switchback, points to the right indicate the opposite. Points that lie directly on this line indicate that there is no difference between either algorithm on the specific instance. CPU time plots (omitted for space) show nearly identical results.

Table 1 displays the average node expansion and CPU time speedup of the two algorithms on each domain. Column 2 shows speedup in terms of average nodes expanded ; average nodes expanded by Switchback over average nodes expanded by ShortCircuit for all instances in each domain. Column 4 shows a similar result for CPU time. The standard deviation for each distribution is shown in columns 3 and 5.

15 Puzzle The first domain is the standard 15-Puzzle. This domain is not well suited for hierarchical search since Manhattan distance is reasonably informative and less costly to compute. It is offered here because it has become a standard benchmark in the literature. We tested on the standard 100 instances first presented by Korf(1985).

A 9-level instance-specific abstraction hierarchy that was first presented by Holte, Grajkowskic, and Tanner (2005) was used for all puzzle instances. To create the first level of abstraction, the 7 tiles that are closest to their goal position (according to Manhattan distance) are selected. The identity of these tiles are then removed. The second level is created by selecting 8 tiles and removing their identities. This continues until the complete hierarchy has been created. The epsilon heuristic is used at the highest level. An identical abstraction hierarchy was used in the macro 15-puzzle and glued 15-puzzle domains.

The left most panel of Figure 4 shows the node expansion results for this domain. The results are clear; Short Circuit expands fewer nodes on every instance compared to Switchback. The first row of Table 1 shows that, on average, node expansion results improved by approximately a factor of five.

Macro 15 Puzzle The macro 15-Puzzle is a variant of the classic sliding tile puzzle. Multiple tiles in a row or column can be moved in one step, resulting in a larger branching factor and shorter solution length. The same standard instances from Korf (1985), used for the 15-Puzzle, were applied to this domain. The median solution length is 32 with an average solution length of 32.06 and a standard deviation of 2.3.

The second panel of Figure 4 shows node expansion results. On all instance, Short Circuit offered superior performance. According to row 2 of Table 1, Switchback expanded nearly three times more nodes than Short Circuit on average.

Glued 15 Puzzle The glued 15-Puzzle is another variation of the standard puzzle that glues a tile to the board. This tile cannot be moved during the entire search process. The 100 instances reported by Larsen et al. (2010) were used for the experiment. The median solution length is 51. The average solution length is 53.56 with a standard deviation of 9.61. This domain is well suited for hierarchical search since the standard Manhattan distance heuristic does not account for the glued tile.

The third panel of Figure 4 shows the results. In this domain, Short Circuit did not offer substantial speedup on every instance. Short Circuit increased performance the most on instances that required the fewest number of node expansions. As problem difficulty increased, both algorithms performed approximately the same.

14-Pancake Puzzle The last domain in the evaluation is the N-Pancake puzzle. In this domain, N pancakes of different size stack on top of one another. The goal is to arrange the pancakes according to their size, largest pancake on the bottom of the stack, smallest on the top. A legal move consists of picking a position in the stack and then flipping all pancakes above it. We used the 100 instances reported in Larsen et al. (2010) where N=14 for all instances. The median solution length is 13 and the average solution length is 12.91 with a standard deviation of 1.1. Hierarchical search

is no longer an attractive approach for this domain since the gap heuristic presented by Helmert (2010) is an informative and less computationally expensive alternative.

In this domain, Short Circuit offered the smallest performance increase. The fourth row of Table 1 shows an average speedup of 1.45 over all instances. The far right panel of Figure 4 shows node expansion results. While substantial reductions are not reported, Short Circuit always expanded fewer nodes than Switchback.

Suboptimal Search

When resources are limited, or time is critical, optimal solutions may be traded for suboptimal ones in order to solve larger problems. Two well-known approaches to finding solutions sub-optimally include weighted search (Pohl 1973) and greedy search (Doran and Michie 1966). In weighted search, the node evaluation function is made greedier by applying a weight $w \geq 1$ to the heuristic value. If the unweighted heuristic is admissible, then weighted search will return a solution guaranteed to cost no more than a factor of w greater than optimal. When we use Short Circuit to compute heuristic values for a weighted search at the base level, we call the resulting algorithm Short Circuit Weighted A*. In greedy search, we ignore the g value associated with each node and always expand the node with the smallest heuristic value. Solutions returned by greedy search are not guaranteed to have any bound on solution cost. When using Short Circuit to generate heuristic values that drive a greedy search at the base level, we call the resulting algorithm Short Circuit Greedy.

Tables 2 and 3 compare optimal Short Circuit to the suboptimal variants. (The first row of each table pertains to the Switch algorithm, which we present below.) The column entitled "Solved" displays how many of the 100 instances each algorithm was able to solve with the 47GB memory limit. The Sub-Optimality column shows the average ratio of solution length returned by a given algorithm over optimal solution length. The last two major columns display statistics for node expansion and CPU time. Mean in each one of these columns represents the geometric mean of the distributions. These results were derived from a completely new C++ implementation, disjoint from the one used above.[2] Table 2 shows results on the same 15-Puzzle instances as used previously and Table 3 shows results on a more difficult variant of the glued 15-Puzzle in which two tiles cannot move.

In the Glued Two domain, 100 problem instances were generated by randomly selecting two adjacent tiles to be glued, then performing a random walk back of one million steps from the goal. The median solution length is 47 with an average solution length of 52.76 and a standard deviation of 21.83. This domain is more suitable for hierarchical search than the standard 15-Puzzle since the Manhattan distance heuristic does not take into account the fact that 2 of the tile pieces are not able to move.

The results are surprising: suboptimal search using heuristics from hierarchical search is even slower than optimal search. A more detailed look at the results (omitted for space) shows that the number of nodes expanded at the base level does decrease, as one would expect, but that this decrease is swamped by an increase in the number of nodes expanded at higher levels of the search. Apparently suboptimal search worsens the cache behavior that hierarchical search relies on for efficiency. A new approach using hierarchical search to generate suboptimal solutions is needed.

[2]This is the reason for the absence of the Macro puzzle, glued 15 puzzle, and 14-Pancakes domains in these results.

A Closer Look at Switchback

To motivate a new approach to suboptimal hierarchical search, we first examine the behavior of Switchback in more detail. Larsen et al. (2010) suggest that Switchback is effective because it expands each node at most once. However, Switchback's effectiveness also hinges on the assumption that subsequent nodes in the search are likely to already be present in the large caches built up during previous searches. This is a reasonable assumption for nodes that lie roughly between the start and goal, as these are exactly the nodes that needed to be expanded to compute previous heuristic values. And indeed, Larsen et al. (2010) present high cache hit rates for Switchback. However, note that optimal heuristic search must also expand many nodes that are not directly between the start and goal, in fact, it must expand any node n for which $f(n) \leq f^*(opt)$. This means that Switchback must determine heuristic values for nodes that lie significantly to the 'opposite' side of the start or goal. Note that these additional searches are still guided by a heuristic focused on the original goal, not one focused on the current query nodes. How can Switchback be efficient in the face of these additional expansions? The following theorem shows that, in fact, any new query node must be close to the existing cache, so the amount of additional search is limited.

Theorem 2 *Assume an abstraction ϕ that preserves structure, that is, if r is a child of q, then either $\phi(r) = \phi(q)$ or $\phi(r)$ is a child of $\phi(q)$. Assume also that the search space is bidirectional (r is a child of q implies q is child of r) and that all actions cost 1. Let node q at level i of Switchback be a node for which we have already computed a heuristic value. Then, for a child r of q, $f(\phi(r)) \leq f(\phi(q)) + 2$.*

Proof: From our assumptions and properties of the abstraction we know that if $\phi(r)$ is not in the cache, then $\phi(r) \neq \phi(q)$ and $\phi(r)$ is a child of $\phi(q)$. This means that $g(\phi(r)) \leq g(\phi(q)) + 1$. From Theorem 1 of Larsen et al. (2010) we know that h is admissible and consistent. From consistency, we have

$$\begin{aligned} h(\phi(r)) &\leq h(\phi(q)) + c(\phi(r), \phi(q)) \\ h(\phi(r)) &\leq h(\phi(q)) + 1 \end{aligned}$$

Then, $f(\phi(r)) = g(\phi(r)) + h(\phi(r)) \leq g(\phi(q)) + h(\phi(q)) + 2 = f(\phi(q)) + 2$ as desired. □

Switch

Using the observation above, we have created Switch, a new unbounded suboptimal search algorithm that places suboptimality in the hierarchy used to generate heuristics for the search. The algorithm reverses the direction of a Switchback

Algorithm	Weight	Solved	Sub-optimality	Nodes Expanded (100K) Mean	Nodes Expanded (100K) Median	CPU Time (s) Mean	CPU Time (s) Median
Switch	-	**100**	1.31	**0.78**	**0.84**	**0.27**	**0.30**
Short Circuit	-	99	**1**	34	34	28	28
Short Circuit WA*	1.1	99	1.01	53	58	44	49
” ”	1.2	99	1.04	105	121	92	106
” ”	2	52	1.20	200	238	182	217
” ”	5	52	1.37	159	215	143	199
” ”	10	68	1.50	192	243	173	219
Short Circuit Greedy	-	58	8	321	430	276	375

Table 2: Switch, Short Circuit Greedy, and Short Circuit WA* on the 15 Puzzle.

Algorithm	Weight	Solved	Sub-optimality	Nodes Expanded (100K) Mean	Nodes Expanded (100K) Median	CPU Time (s) Mean	CPU Time (s) Median
Switch	-	100	1.13	**0.29**	**0.20**	**0.10**	**0.06**
Short Circuit	-	100	**1**	1.39	0.91	0.89	0.55
Short Circuit WA*	1.1	100	1.01	1.69	1.28	1.10	0.81
” ”	1.2	100	1.03	2.39	1.89	1.59	1.24
” ”	2	100	1.24	6.34	6.77	4.46	5.00
” ”	5	100	1.44	6.03	8.06	4.22	5.75
” ”	10	100	1.53	5.51	6.34	3.84	4.52
Short Circuit Greedy	-	100	5.17	8.25	8.69	5.63	6.00

Table 3: Switch, Short Circuit Greedy, and Short Circuit WA* on the glued two 15 Puzzle.

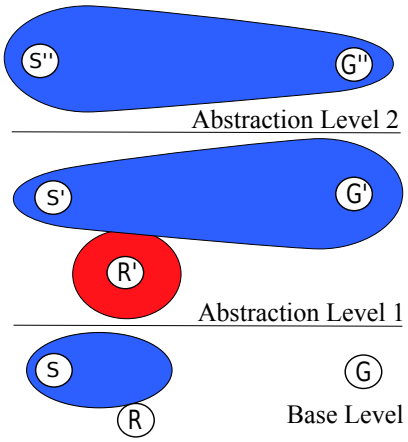

Figure 5: Switch on a 3-level abstraction hierarchy.

search after one complete search from start to goal (or goal to start depending on the direction of search) at each level. Once the first search at each level is complete, all nodes remaining in the open and closed list are moved to a cache for that level, even though their g values are not necessarily optimal.

Figure 5 illustrates the search procedure on a three level hierarchy. Switch starts at the highest level of the abstraction hierarchy with an A* search from $S'' = \phi_2(S)$ to $G'' = \phi_2(G)$. This search is driven by the ϵ heuristic. Once the goal at this level is reached, all nodes on the open and closed lists are placed into the cache. This is indicated by the oval encapsulating S'' and G''. The search then progresses down the hierarchy to abstraction level one. Abstraction level one starts an A* search in the opposite direction from G' to S' using heuristics from abstraction level two. Once S' is expanded search moves to the base. The base in-turn will start a search from S to G.

At some point during the base level search a heuristic for node R will be required. Node R will be abstracted to $R' = \phi_1(R)$ which may not exist in the cache of abstraction level one. If it doesn't, we take comfort from knowing that it is probably not far away. A uniform cost search will be started from R' back to the cache in abstraction level one. Once the search intercepts the cache, the cost from R' to G' will be calculated and returned as a heuristic value for R. A uniform cost search back to the cache will be performed for any node that is not found in the cache for the remainder of the base level search. It should be noted that this search checks for cache intersections during time of generation and not time of expansion.

In Switchback and Short Circuit, every level of the abstraction hierarchy is used for the duration of the search. This is not the case for Switch. Since subsequent searches are uninformed they do not use heuristics from the level above. Once the initial search is completed at a given level, the levels above it will never be used again. This allows memory resources to be reclaimed as the initial search progresses down the hierarchy.

In an attempt to reduce the total number of nodes expansions, Switch adopts two caching techniques taken from HA*. First, all nodes in a secondary search that lie along the path from the query node R' to the cache are moved into

the cache. This is similar to the optimal path caching that is used in HA* but in this case the path is sub-optimal.

The second caching technique used is a variant of $P - g$ caching where nodes generated during the secondary search that are not along the suboptimal path (indicated by the circle surrounding R') are also placed into the cache. These nodes however are cached with the much larger value of $P + g$, where P is the cost of the suboptimal path and g is the nodes g value in the secondary search. If at any point during search a heuristic for one of these nodes is requested from a lower level search, the value $P + g$ will be returned. Since secondary searches are uninformed, these values are only used to guide the lower levels, and for cache intersections. They are not used to guide the search at a given abstraction level. This is an attempt to force the lower level search to stay within the bounds of the abstract solution. In other words, we are trying to build a wall around the initial search.

Evaluation

The first two lines in Tables 2 and 3 show, the suboptimal Switch algorithm find solutions much faster than the optimal Short Circuit algorithm, while suffering only a modest increase in solution cost. In the Plain 15-Puzzle, the solutions returned by Switch were on average within 33% of optimal. Short Circuit expanded 30 times more nodes than Switch. In the glued two 15-Puzzle domain, Switch was able to solve all 100 instances with fewer resources than the other algorithms. Switch was nearly five times faster than Short Circuit in terms of node expansion and four times faster in terms of CPU time.

Discussion

Our evaluation revealed that a simple modification to the state-of-the-art optimal hierarchical heuristic search algorithm can result in a significant speedup on many domains. Short Circuit offered equivalent or enhanced performance on every instance of every domain presented. In some domains, speedups of more than a factor of five are reported.

Short Circuit gains its speedup because of how Switchback behaves when it reaches the goal at a given abstraction level. It has the potential to continuously expand f layers in the abstraction hierarchy until the node in question is found. Short Circuit does not completely solve this problem but it does reduce its effect on search performance. Exiting during node generation and using part of the open list as a cache has the effect of tie breaking in favor of the query node. This can prevent a complete f layer expansion. In domains where Switchback does not need to restart abstraction level searches often, the speedup of this technique will be limited. It should be noted however that Short Circuit requires almost no additional overhead; hence performance on such domains would likely be similar to the original algorithm.

We have also shown that placing sub-optimality in the heuristic evaluation function of a hierarchical search leads to better solutions faster than applying a standard suboptimal algorithm at the base level. Changing the direction of the search takes advantage of the fact that nodes not found

```
SWITCH()
01. closed ← array of length height_φ of empty closed lists
02. for i ← 0 up to height_φ − 1 do
03. result ← FIRSTSEARCH(0, closed, s_start, s_goal)
04. if result≠ NULL then return EXTRACT-PATH(result)
05. return NULL

FIRSTSEARCH(i, closed, s, g)
06. if i ≠ heightφ − 1
07.    return FIRSTSEARCH( i + 1, closed, φ(i + 1, g), φ(i + 1, s))
08. open ← empty open list
09. insert s open
10. while open is not empty do
11.    n ← remove node from open with lowest f
12.    if i is even then children ← succs(n)
13.    else children ← preds(n)
14.    for each c in children do
15.       if c in closed_i then
16.          if g(c) < g(n) + cost(n, c) then continue
17.          g(c) ← g(n) + cost(n, c)
18.          if c is not open then insert c onto open
19.          continue
20.       h(c) ← HEURISTIC(i, closed, c)
21.       g(c) ← g(n) + cost(n, c)
22.       insert c into open and closed_i
23.    if n = g then
24.       free closed_{i+1} return n
25. return NULL

SEARCHBACK(i, cache, s)
26. if s is in cache_i then return s
27. open ← empty open list; closed ← empty closed list
28. q ← φ(i, s)
29. insert q into open
30. while open is not empty do
31.    n ← remove node from open with lowest
32.    if i is even then children ← succs(n)
33.    else children ← preds(n)
34.    for each c in children do
35.       if c in closed then continue
36.       g(c) ← g(n) + cost(n, c); h(c) ← 0
37.       if c in cache_i then
38.          d ← cache_i(c)
39.          P ← g(c) + g(d)
40.          for each a in path from q to c
41.             g(a) ← g(c) − g(a) + g(d)
42.             insert a into cache_i
43.          for each b in open and closed
44.             g(b) ← P + g(b)
45.             insert b into cache_i
46.          g(q) ← P; return q
47.       insert c into open
48.    insert n into closed
49. return NULL
```

Figure 6: Pseudo-code for Switch

HEURISTIC(i, *closed*, s)
50. if $i = height_\phi - 1$ then return $\epsilon(s)$
51. $n \leftarrow \text{lookup}\phi(i+1, s)$ in $closed_{i+1}$
52. if $n \neq$ NULL then return $g(n)$
53. $r \leftarrow$ SEARCHBACK($i+1$, *closed*, s)
54. if $r =$ NULL then return ∞ else return $g(r)$

in the Switchback cache are likely only a few steps away. In nonuniform cost domains this may not be the case and performance may degrade. Also, Switch attempts to constrain the base level search to follow a solution found by the abstraction. It may be the case that no such solution exists in the original problem. This would force the base level search to leave the constraints of the abstract solution and could negatively effect search performance. Switch also has another limitation in that it is unbounded and does not offer the ability to control the tradeoff between solution quality and CPU time.

Switch uses $P + g$ caching to return large heuristic values for states that are outside of the initial Switchback search. Holte et al. (1994) present a method known as Refinement that is similar. In Refinement, all searches in the abstraction hierarchy are completed before search at the base level begins. Then when search at the base level proceeds, only states that map to states on the solution path of the abstraction are considered. All other states are given a heuristic value of infinity. Refinement is complete as long as a path can be found in the original problem space that follows the solution path found in the abstraction. To guarantee that this is the case, Refinement uses the "star" method of abstraction (Mkadmi 1993). This method creates the abstraction by enumerating the original problem space. Switch is essentially a softened version of refinement that maintains completeness while using modern instance-specific homomorphic abstraction techniques that do not require enumeration of the complete problem space.

$P + g$ caching is a first attempt at creating a wall of abstraction around an initial Switchback search. It is essentially an upper bound on the true cost of the state in question. An alternative approach would be to execute a Dijkstra search as soon as the cache was intersected. This search could then update all node values in the subsequent search with their actual distance from the suboptimal path. This approach could potentially increase search performance by returning more accurate heuristic values to the level below.

Switch also has similarities to the work of Zhou and Hansen (2004), who use a suboptimal search to help construct an instance-specific pattern database. Their scheme is targeted at optimal search, however.

Conclusion

We presented a simple modification to the state-of-the-art optimal hierarchical heuristic search algorithm. This modification is simple to implement and resulted in a significant speedup across all tested domains, and in some cases expanded less than one fifth of the nodes. We also presented and evaluated a new suboptimal hierarchical search algorithm that places sub-optimality in the hierarchy rather than at the base of the search. Our results have shown that adaptations of existing suboptimal search techniques may not take advantage of the fact that a hierarchy of search is being used to generate heuristic values and can actually use more resources than their optimal counterparts to solve the same problem. This work opens up a new area of investigation in hierarchical search for the common case in which one wishes to solve a problem so large that optimal search is not feasible.

Acknowledgments

We gratefully acknowledge support from NSF (grant IIS-0812141), DARPA CSSG program (grant N10AP20029) and the Canadian NSERC. We would also like to thank Brad Larsen, Ethan Burns and Matthew Hatem for their help and advice.

References

Culberson, J. C., and Schaeffer, J. 1996. Searching with pattern databases. In *Proceedings of Canadian Conf on AI*, 402–416.

Doran, J. E., and Michie, D. 1966. Experiments with the Graph Traverser Program. In *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 235–259.

Helmert, M. 2010. Landmark heuristics for the pancake problem. In *Proceedings of (SoCS 2010)*, 109–110.

Holte, R. C.; Drummond, P. B.; Zimmer, M.; and A., M. 1994. Searching With Abstractions: A Unifying Framework and New High-Performance Algorithm. In *Proc of the 10th Canadian Conf on AI*, 263–270.

Holte, R.; Grajkowskic, J.; and Tanner, B. 2005. Hierachical heuristic search revisitied. In *Symposium on Abstracton Reformulation and Approximation*, 121–133.

Larsen, B.; Burns, E.; Ruml, W.; and Holte, R. C. 2010. Searching Without a Heuristic: Efficient Use of Abstraction. In *Proceedings of (AAAI-10)*, 114–120.

Mkadmi, T. 1993. Speeding Up State-Space Search by Automatic Abstraction. Master's thesis, University of Ottawa.

Pohl, I. 1973. The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computation issues in heuristic problem solving. In *Proceedings of IJCAI-73*, 12–17.

Zhou, R., and Hansen, E. A. 2004. Space-efficient memory-based heuristics. In *Proceedings (AAAI-04)*, 677–682.