FASTER OPTIMAL AND SUBOPTIMAL HIERARCHICAL SEARCH

BY

Michael Leighton

BS, University of New Hampshire (2010)

THESIS

Submitted to the University of New Hampshire
in Partial Fulfillment of
the Requirements for the Degree of

Masters of Science

in

Computer Science

May, 2012

This thesis has been examined and approved.

Thesis director, Wheeler Ruml,
Assistant Professor

Radim Bartos,
Associate Professor of Computer Science

Philip Hatcher,
Professor of Computer Science

Michel Charpentier,
Associate Professor of Computer Science

Date

ACKNOWLEDGMENTS

I would like to thank Professor Wheeler Ruml for his help and guidance throughout my graduate career. Also, the members of the UNH AI group including Ethan Burns for code and plotting software, Brad Larsen for code, Matthew Hatem for scripts and CPU time, and Chris Wilt and Jordan Thayer for advice. Finally, I would also like to thank my wife Kelly and son Charlie for their support and patience.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

FASTER OPTIMAL AND SUBOPTIMAL HIERARCHICAL SEARCH

by

Michael Leighton

University of New Hampshire, May, 2012

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 thesis makes three 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. Finally, we propose a modification to our optimal algorithm that uses multiple additive abstractions in order to improve performance of both optimal and suboptimal hierarchical search on some domains.

CHAPTER 1

Introduction

Hierarchical search is an important area of research that has been studied in domains that do not have well known heuristics or for which the goal state changes between problem instances. Hierarchical search works by replacing one state space by another where search is less costly. Searches in this space are then used to generate heuristic cost-to-go estimates for search in the original space. If the search algorithm used in the abstract space is also informed then its heuristics can be generated using search in an even more abstract space, creating a hierarchy of search. At the top of the hierarchy, the simplest space to search is either fully enumerated or searched using a non-domain-specific heuristic.

Pattern databases (Culberson and Schaeffer, 1996) are an alternative to Hierarchical search that also use abstraction to define a heuristic function to be used by heuristic search algorithms such as A* (Hart et al., 1968) and IDA* (Korf, 1985). A pattern database is constructed by abstracting the problem, enumerating all possible states in the abstract space, then storing their cost values in a table. Later, during search this table is used to look up heuristic values for each state visited. Enumerating all possible states in an abstract version of the original problem can be a costly process but this same table can be used to solve many problems that have the same goal state. The cost of generating this pattern database is then amortized over the number of problems it is used to solve. If however, the goal state changes, the pattern database can not be reused. Thus, in domains where the goal state changes frequently or when only a few problem instances will be solved, hierarchical search is the more attractive approach.

The thesis of this work is:

Hierarchical search contains unique features, not present in typical state space search, that can be exploited to improve performance of both optimal and suboptimal search.

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 usage. One obvious approach is to use a hierarchical search to generate heuristics for sub-optimal algorithms such as Weighted A* (Pohl, 1970). We present an empirical evaluation showing that this approach does not always result in the desired outcome. As we will show below, it is often the case that this will 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 does not need the entire abstraction hierarchy to be kept in memory at one time, reserving resources for search in the original problem. 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.

Current state-of-the-art hierarchical search algorithms use either a single linear abstraction hierarchy or max over multiple abstraction hierarchies (Holte et al., 2005). While theses approaches have been shown to generate powerful heuristics, they can ignore important interactions present in the original problem. Pattern databases which use multiple additive abstractions have been shown by Felner et al. (2004) to produce more accurate heuristics than those that max over many single non-additive abstractions. We propose a hierarchical search algorithm that takes into account more detail of the original problem by using multiple additive abstractions. We show that this approach can improve the performance of existing hierarchical search algorithms on some domains, and that the heuristics generated can work well when suboptimal approaches, such as weighting, are applied to them. We will also discuss some of the limitations to this approach, including the time and memory overhead introduced by searching over multiple abstractions.

We hope that Short Circuit, its variants, 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 the suboptimal algorithms presented in this text illustrates the potential for further research on suboptimal hierarchical search. Taken all together, the results provide convincing evidence that hierarchical search contains unique features, not present in typical state space search, that can be exploited to improve performance of both optimal and suboptimal search.

1.1 Outline

The thesis is organized into the following chapters.

1. Chapter 2 is a survey of the current state-of-the-art in hierarchical heuristic search. We also give a brief review of abstraction and how it can be used to generate heuristics.

2. Chapter 3 describes a proposed algorithm for performing optimal hierarchical search. Short Circuit is a modification to the previous state-of-the-art optimal hierarchical search algorithm Switchback that provides improved tie-breaking in the abstraction hierarchy. An empirical evaluation is presented and shows that Short Circuit reduces node expansion and CPU time when compared to Switchback.

3. Chapter 4 describes a proposed algorithm for performing suboptimal hierarchical search. Switch is a unbounded suboptimal hierarchical search algorithm that places the sub-optimality of the search into the abstraction hierarchy used to generate the heuristic cost-to-go estimates. An empirical evaluation is presented that shows that Switch will solve problems faster than conventional suboptimal search algorithms driven by heuristics generated via hierarchical search.

4. Chapter 5 describes a proposed algorithm for performing optimal and suboptimal hierarchical search using multiple abstractions. We will show how Short Circuit can

be modified to use multiple abstractions and we will discuss some of the shortcomings of such an approach. Finally, we weight the heuristic generated using multiple abstractions and present an empirical evaluation.

5. Chapter 6 offers a conclusion and presents future extensions of our work.

CHAPTER 2

Previous Work

Here we discuss several state-of-the-art approaches to heuristic search that use abstraction-based heuristics. We also give a brief review of the different forms of abstraction that can be found in the literature.

2.1 Abstraction

Abstraction is a way of automatically creating powerful admissible and consistent heuristics. Using abstraction to create heuristics for a given problem requires an abstraction function ϕ that maps a state s in the original space S to a state $\phi(s)$ in the abstract space S'. If the $cost(\phi(s), \phi(r)) \in S'$ of any two states $\phi(s)$ and $\phi(r)$ in the abstraction are less than or equal to the $cost(s, r) \in S$ for states s and r in the original space then the cost in the abstract space can be used as a admissible heuristic $h(s, r) = cost(\phi(s), \phi(r)) \in S'$ for search in the original space. An abstraction is said to be solution-preserving if a solution path in the abstract space exists if and only if a solution path exists in the original problem.

Two major types of abstractions have been studied in the literature. The first type of abstraction is know as an embedding. This type of abstraction maps an abstract state $\phi(s) \in S'$ to a single state $s \in S$ in the original space. Valtorta (1984) proved that if an embedding is used to generated heuristics for an A* search in the original space then the search will expand every state that would have been expanded by a blind search (such as breadth first search or Dijkstra) in the original space. Since the purpose of a heuristic is to improve search performance, using heuristics generated by embedding can be harmful. A heuristic search that uses abstraction to generate its heuristics and expands as many or more states than a blind search is said to have passed the "Valtorta's Barrier".

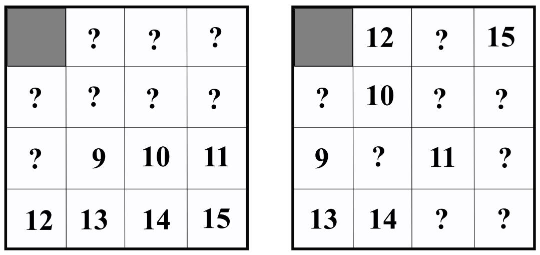


Figure 2-1: An example 15 Puzzle abstraction. Two different states in the abstraction are shown with tiles 1-8 abstracted away and tiles 9-15 as the remaining discrete tiles.

The alternative to embeddings are known as homomorphic abstractions. A homomorphic abstraction is a many-to-one mapping of states in a original space to a state in the abstract space. Informally, a homomorphic abstraction maps a group of states in the original space to a single state in the abstract space. Since homomorphic abstractions are generally smaller than the original space, searching the abstractions to generate heuristics often reduces the amount of search required to solve the original problem. The introduction of homomorphic abstractions made it possible to use abstraction to generate heuristics without passing Valtorta's Barrier.

Many examples of how to create efficient homomorphic abstractions for a variety of different domains can be found in the literature. For example, an abstraction for the 15 Puzzle can be created by treating several of the tiles as if they were indistinguishable rather than discrete. (Readers who are unfamiliar with the 15 Puzzle should read the full description presented in section 3.3.1.) Figure 2-1 shows two separate states in an abstraction that was generated by treating tiles 1-8 as indistinguishable, and tiles 9-15 as discrete. The abstraction still maintains the same cost function, moving the blank costs one unit, but since eight of the tiles are indistinguishable the abstraction is easier to solve.

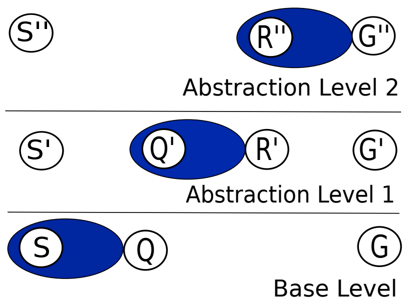


Figure 2-2: An example of the Hierarchical A* algorithm on a 3-level abstraction hierarchy.

2.2 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 2-2 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 the first abstraction level for their heuristic values. To calculate the heuristic value for a given node Q (also known as the query node) 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 $cost(Q', G')$ is then used as a heuristic value for Q. This technique is used recursively to generate heuristic searches in further abstract levels as well, resulting in a hierarchy of abstraction. 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

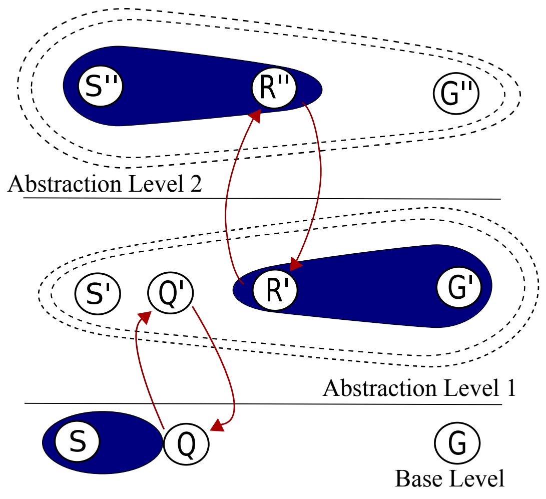


Figure 2-3: An example of the Switchback algorithm running on a 3-level hierarchy.

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.

2.3 Switchback

Switchback (Larsen et al., 2010) is a hierarchical search algorithm that changes the direction of search at every level of the abstraction hierarchy. This technique is used to prevent node re-expansion. Since the search direction alternates with every level, every expansion at an abstract level will contain the optimal path length from the expanded node back to the abstraction of the goal of the level below. Hence, when a heuristic is required from the level below it can be looked up in the cache of previous searches and extra search may not be required. Since Switchback alternates the direction of search, it has the additional requirement of a predecessor function, and it applicable to domains for which such a function is easily computed.

Figure 2-3 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 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. Like HA* a non-domain specific heuristic, such as the ϵ heuristic, is used to drive search in the highest abstraction level.

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 2-3 represent how the search will progress at each level after the goal at that level has been achieved.

2.4 Abstract Solution Refinement

The two most common ways of using abstraction in heuristic search, pattern databases and hierarchical search, use the abstractions solution path as heuristic value to drive an informed search in the original problem. An alternative approach known as refinement (Knoblock, 1994; Minsky, 1963; Sacerdoti, 1974) is to use the abstract solution path as a skeleton for the solution path in the original problem.

In refinement, all searches in the abstraction hierarchy are completed before search in the original problem begins and an abstract path $\langle \phi(s), \phi(q), ..., \phi(g) \rangle$ from the abstract start $\phi(s)$ to the abstract goal $\phi(g)$ has been found. The abstract solution is then refined in the original problem, such that if a state $\phi(q)$ is the successor of $\phi(s)$ along the abstract solution path then a path from s to q is found via unguided search in the original problem. This process is repeated until a full path from s to g has been found. Informally, refinement treats each step of the abstract solution path as a subgoal for search in the original problem. Since refinement attempts to follow an abstract solution, it is not guaranteed to find optimal solution, but it has been shown to find solutions rather quickly.

Refinement is guaranteed to be complete if the abstraction used is *solution preserving*; a solution path in the abstract space exists if and only if a solution path exists in the original problem. One such abstraction is known as the "STAR" abstraction and has been applied to domains similar but smaller than the ones presented in this thesis. In order to create the "STAR" abstraction however, every state in the original problem must be enumerated, and thus the size of the domains presented here make generating the abstraction intractable.

Holte et al. (1994) show that approaches that use refinement and those that use abstraction to produce admissible heuristics, hierarchical search, are actually similar to each other. Both approaches guide search by using the abstract solution path and differ only in how they behave when encountering a state that was not seen in the abstract search. Refinement will ignore all states generated in the original problem that do not map directly to a state on the abstract solution path, these states are given a a heuristic value of infinity. Hierarchical search will perform more search to find an admissible heuristic value for these states. Our suboptimal approach to hierarchical search presented in Chapter 3 uses a similar technique to trade speed for sub-optimality.

CHAPTER 3

Optimal Hierarchical 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. Improving the state-of-the-art optimal hierarchical search algorithm will increase the number of instances that can be solved before a pattern database becomes useful. Here we explore reducing the CPU time required to solve problems optimally using hierarchical search.

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

$$f(fmin) \leq f(n)$$

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

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. □

3.1 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 search in the abstraction hierarchy. 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 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 $g(Q)/(f(fmin) - h(Q)) \leq 1 = f(fmin) = f(Q)$, 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 3-1 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

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

```
ShortCircuit()
01. $open \leftarrow$ array of length $height_\phi$ of empty open lists
02. $closed \leftarrow$ array of length $height_\phi$ of empty closed lists
03. for $i \leftarrow 0$ 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 $g(s_{start}) \leftarrow 0; h(s_{goal}) \leftarrow 0$
08.         insert $\phi(i, s_{goal})$ into $open_i$
09. $result \leftarrow$ 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 $closed_i$ and not in $open_i$ then return $s$
13. while $open_i$ is not empty do
14.     if $s$ is in $open_i$
15.         $fmin \leftarrow$ node from $open_i$ with lowest $f$
16.         if $g(s)/(f(fmin) - h(s)) \leq 1$ then return $s$
17.     $n \leftarrow$ remove node from $open_i$ with lowest $f$
18.     if $i$ is even then $children \leftarrow succs(n)$
19.     else $children \leftarrow$ 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.             $g(c) \leftarrow g(n) + cost(n, c)$
24.             if $c$ is not $open_i$ then insert $c$ onto $open_i$
25.             continue
26.         $h(c) \leftarrow$ Heuristic($i$, $open$, $closed$, $c$)
27.         $g(c) \leftarrow g(n) + cost(n, c)$
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 $i = height_\phi - 1$ then return $\epsilon(s)$
32. $n \leftarrow$ lookup$\phi(i + 1, s)$ in $closed_{i+1}$
33. if $n \neq$ NULL then return $g(n)$
34. $r \leftarrow$ Resume($i + 1$, $open$, $closed$, $\phi(i + 1, s)$)
35. if $r =$ NULL then return $\infty$ else return $g(r)$
```

Figure 3-1: Pseudo-code for the Short Circuit algorithm.

implementation of Switchback is fairly straightforward.

3.2 Tie Breaking

Short Circuit uses A*, and A* must expand all nodes with an $f(n) \leq f^*(goal)$, thus Short Circuit must expand all nodes that Switchback would expand at a given abstraction level with the exception of the last f layer. 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. In domains where Switchback is not required to expand many nodes in this last f layer 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.

To see how many extra nodes Switchback may need to expand in the last f layer of an abstraction hierarchy, we experimented with 50 instances of the 15 Puzzle that could be solved by both algorithms using 8 GB of RAM and no time bound. After a solution to a given instance was found, we continued search at the first abstraction level until the last f layer was fully enumerated. We found that Switchback was required to expanded 99.8% of these nodes before a solution was found while Short Circuit expanded less than 1%. Clearly, on this domain Switchback must expand many nodes in this last layer to find a solution and thus Short Circuit performs well.

3.3 Evaluation

In order to gain a better understanding of the performance of Short Circuit, we implemented it in C++ and compared to Switchback on six separate domains. 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 3-2 shows a scatter plot for each of the domains tested. The x and y axes represent the log_2 of the nodes expanded (including all nodes expanded in the abstraction

	Nodes Expanded		CPU Time	
Domain	Speedup	Std	Speedup	Std
15 Puzzle	5.51	2.07	6.86	2.96
Macro 15 Puzzle	3.01	0.45	3.37	0.59
Glued 15 Puzzle	3.36	1.46	5.3	2.3
Glued Two 15 Puzzle	3.98	1.85	4.4	2.2
17-4 Topspin	2.02	0.41	2.41	0.59
14-Pancakes	1.45	0.31	1.47	0.38

Table 3-1: Comparison of the average node expansion and CPU time speedup ratio for Short Circuit over Switchback.

hierarchy) 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 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.

Figure 3-3 shows scatter plots that are similar to that of Figure 3-2. In these plots however the x and y axes represent the log_2 of the CPU time required to solve each instance by Short Circuit and Switchback respectively. Both plots are nearly identical indicating that Short Circuit does not require significantly more time than Switchback to expand a single node.

3.3.1 15 Puzzle

The first domain is the standard 15 Puzzle. In this puzzle there are 16 location that form a 4x4 grid and 15 tiles, numbered 1-15 and one blank. Any tile that is adjacent to the blank can be moved into the blank's position for a cost of one. The goal of the puzzle is to arrange the tiles so that they are in order form lowest to highest with the blank in top

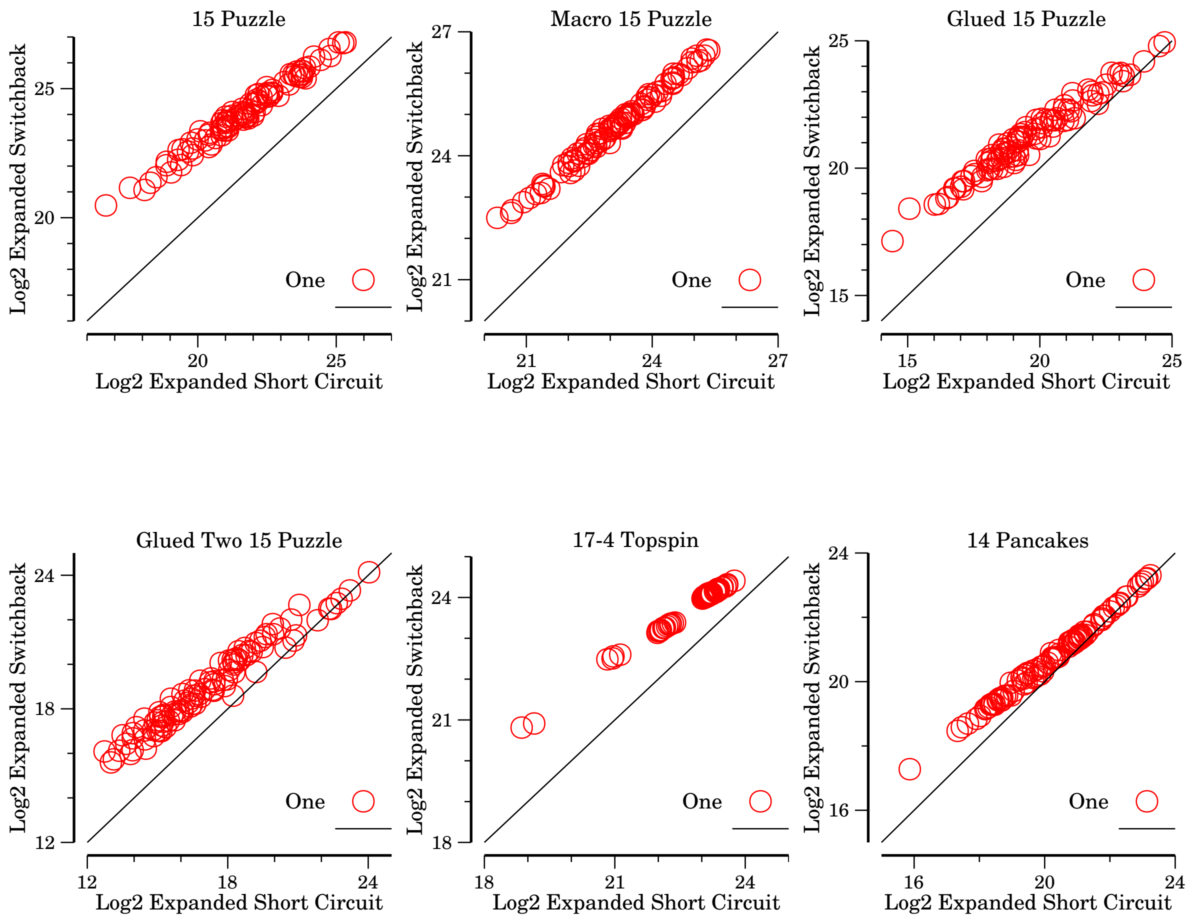


Figure 3-2: Node expansion plots for the 15 Puzzle, Macro 15 Puzzle, Glued 15 Puzzle, Glued Two 15 Puzzle, 17-4 Topspin, and 14 Pancakes.

left location of the grid. This domain is not well suited for hierarchical search since the Manhattan distance heuristic is reasonably informative and less costly to compute. It is offered here because it has become a standard benchmark in the literature.

We used the same 9-level instance specific abstraction hierarchy first presented in Holte et al. (2005). First we compute the Manhattan distance values for all tiles in the start state, then sort them in decreasing order. To create the first abstraction hierarchy we chose to make the first nine of these ordered tiles discrete, the second level uses the first eight, and we continue this up the hierarchy. At the highest level of abstraction we use the ϵ heuristic.

The 100 instances first presented by Korf (1985) were used for all experiments, all

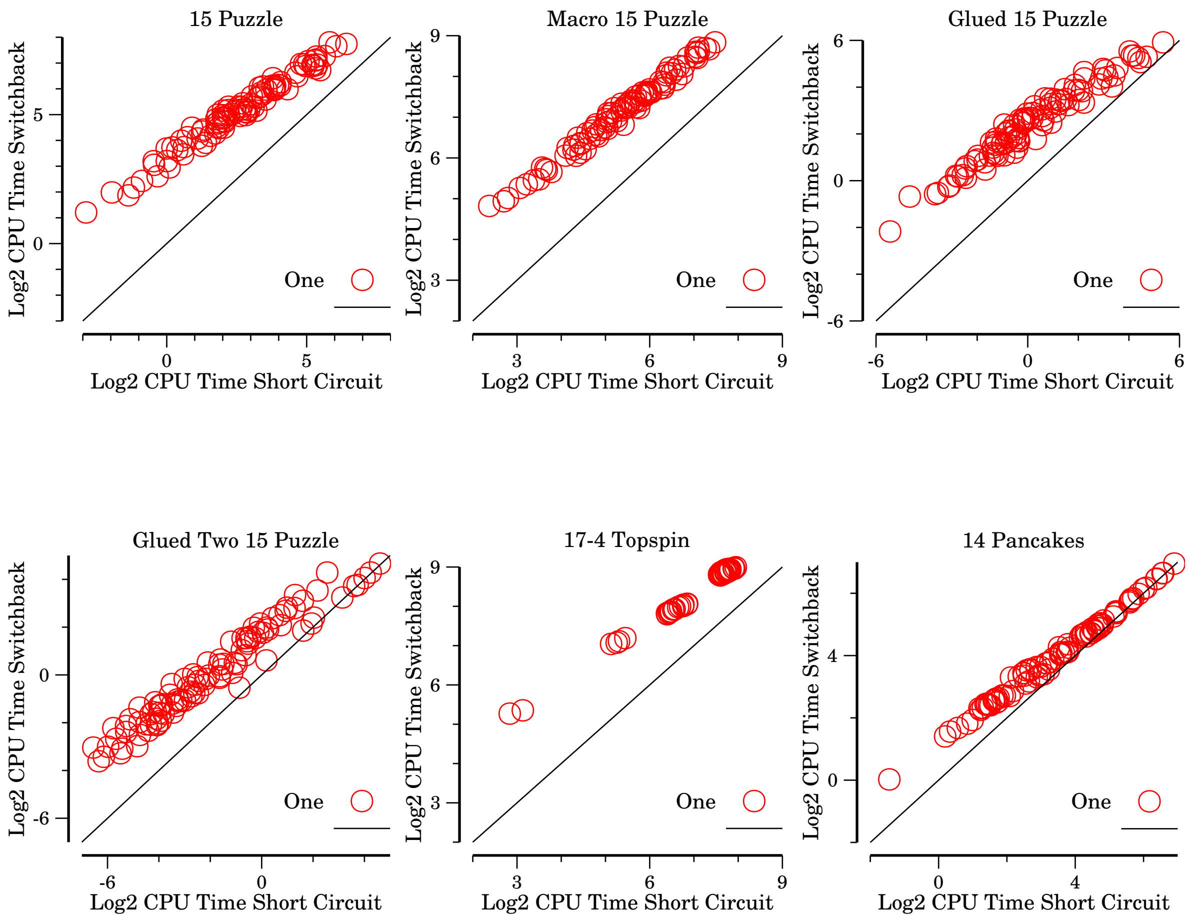


Figure 3-3: CPU Time plots for the 15 Puzzle, Macro 15 Puzzle, Glued 15 Puzzle, Glued Two 15 Puzzle, 17-4 Topspin, and 14 Pancakes.

instances where solved by both algorithms. The top left panel of 3-2 show that Short Circuit expanded fewer nodes than Switchback on every instance tested. The top left panel of 3-3 shows similar results for CPU times. Row 1 of Table 3-1 shows that Switchback expanded nearly 5 times more nodes than Short Circuit on average, and was nearly 7 times slower in terms of average CPU time.

3.3.2 Macro 15 Puzzle

The Macro 15 Puzzle is a variation of the 15 Puzzle that is inspired by the fact that moving multiple physical tiles at once is no harder than moving a single tile. In this puzzle it is possible to move multiple tiles in a row or column in one step, resulting in a larger branching factor and shorter solution length. The puzzle has the same goal configuration as the 15 Puzzle. The abstraction hierarchy used here is identical to the one used for the 15 Puzzle.

The 100 instances presented in Larsen et al. (2010) where used for the evaluation. All instances where solved by each algorithm. The top center panel of Figure 3-2 show the node expansion results for this domain. Short Circuit is the clear winner, expanding fewer node than Switchback on every instance. Figure 3-3 shows similar results for CPU time. Row 2 of Table 3-1 shows that Short Circuit expanded nearly 3 times less nodes than Switchback, and was 3 times faster in terms of CPU time.

3.3.3 Glued 15 Puzzle

The Glued 15 Puzzle is a variant of the standard 15 Puzzle where one tile has been glued to the board in its goal position. The glued tile cannot be moved during the course of the search. The Manhattan distance heuristic is less informative on this domain since it does not take into account the fact that tiles may need to be moved around the glued tile. The abstraction hierarchy used here is identical to that of the one used in the 15 Puzzle.

The 100 instances originally presented in Larsen et al. (2010) where used and fully solved by both algorithms. The top right panel of 3-2 show the node expansion results for this domain. Short Circuit expanded fewer nodes than Switchback on each instance. Row 3 of Table 3-1 shows that Short Circuit expanded 3 times fewer nodes than Short Circuit and was approximately 5 times faster in terms of CPU time.

3.3.4 Glued Two 15 Puzzle

Like the Glued 15 Puzzle, the Glued Two 15 Puzzle is another variant of the standard 15 Puzzle. In this case however two tiles are glued to the board in their goal position. The motivation is that if gluing a single tile to the board in its goal position causes the Manhattan distance heuristic to become inaccurate then gluing multiple tile will increase this effect. The 9-level instance specific abstraction hierarchy was used here as well.

100 instances where randomly generated by a million step walk back from the goal. Both algorithms where able to solve all instances. The bottom left panel of 3-2 shows node expansion results for this algorithm. Again, Short Circuit expanded fewer than Switchback on every instance in the experiment. Similar results can be seen for CPU time in figure 3-3. Row 4 of Table 3-1 shows that Short Circuit expanded approximately 4 times fewer nodes than Switchback and 4 times faster in terms of CPU time.

3.3.5 17-4 Topspin Puzzle

The 17-4 Topspin Puzzle is another permutation puzzle in which 17 tokens, numbered 1-17 are placed on a circular track in a random order along with a turnstile that can reverse the order of four of the tokens at a time. The goal of the puzzle is to get all the tokens in numerical order. In our implementation moving the turnstile has a cost of one and moving the tokens around the track is free. For the first abstraction level we treated token positions 1-8 as if they were indistinguishable and tokens 9-17 as if they where distinct. The remainder of the abstraction hierarchy was created by removing a single token per level starting from position 8. The highest level of abstraction used the ϵ heuristic.

100 instances where randomly generated but neither algorithm could solve more than 72. The center bottom panel of Figure 3-2 shows node expansion results for the 72 instances solved by both algorithms. Short Circuit expanded fewer nodes on each instance. Figure 3-3 shows the CPU time results for both algorithms on the same instances. Again, Short Circuit took less time to solve each instance than Switchback did. According to 5th Row of Table 3-1 Short Circuit expanded approximately half of the nodes on average that were required by Switchback. Short Circuit was also twice as fast in terms of CPU time.

3.3.6 14 Pancakes Puzzle

The last and final domain is the 14-Pancakes puzzle. In this domain 14 pancakes of a different size are stacked on top of each other according to the start configuration of the instance. The goal of the puzzle is to arrange the pancakes so that they are stacked in order form largest to smallest. A move consists of sticking a flipper into the stack and reversing the order of all of the pancakes above it. Hierarchical search is no longer an attractive approach for this domains since the creation of the gap heuristic presented by Helmert (2010) that can be computed quickly and is very accurate. We show it here since it was included in the original evaluation of the Switchback algorithm. The abstraction hierarchy used is similar to the one used in the topspin puzzle. Pancakes at positions 1-8 are treated as if they were distinct. For each level of the abstraction hierarchy we remove one pancake starting at the 8th position. The highest level of abstraction used a search guided by the ϵ heuristic.

The 100 instances originally presented in Larsen et al. (2010) where used and fully solved by both algorithms. The bottom right panel of Figure 3-2 shows the node expansion results for both algorithms. On smaller instances Short Circuit is the clear winner, though as instances get larger Short Circuit starts to behave much like Switchback. Still, Short Circuit expanded fewer node than Switchback on every instance. Figure 3-3 show similar results for CPU times. The last row of Table 3-1 shows that on average Short Circuit expanded 1.45 fewer nodes than Switchback and was 1.47 times faster in terms of CPU time.

3.4 Conclusion

The major contribution of this chapter is the new optimal hierarchical search algorithm, Short Circuit, that exits early from searches in the abstraction hierarchy by breaking ties in favor of the query node. Our empirical results suggest that this algorithm can find solutions with fewer node expansions and less CPU time than the previous state-of-the-art algorithm, Switchback, on several different domains. Since the cost of creating a pattern database can be amortized over many problem instances, it remains the preferred method for solving a large set of problems that share the same goal state. Short Circuit on the other hand can solve individual problems quickly without the computational overhead of constructing a pattern database. Since Short Circuit decreases the time required to solve an individual problem, it makes hierarchical search an even more attractive approach when only a few problem instances must be solved.

CHAPTER 4

Suboptimal Hierarchical Search

In many heuristic search problems finding optimal solutions is prohibitively expensive. In these situations it is often acceptable to abandon optimality for solutions that can be found as quickly as possible. Here we explore quickly finding high quality solutions when using heuristics generated by a hierarchical search.

4.1 Weighting Hierarchical Search

A traditional heuristic search approach used for trading optimality for shorter solving times is known as Weighted A* (WA*) (Pohl, 1970). The node evaluation function f used in A* is modified by placing a weight w on the heuristic h. The resulting evaluation function $f'(n) = g(n) + w * h(n)$ penalizes nodes with larger h values, making the search greedier. WA* may only expand a small portion of the nodes required by A*, which makes search faster. Pohl (1970) also shows that if the heuristic evaluation function used is admissible then w is also a bound on the cost of any solution found by the algorithm.

One obvious approach to creating a suboptimal hierarchical search algorithm would be to replace the optimal A* at the base of Short Circuit with WA*. Since heuristics returned by Short Circuit are admissible, the bound created by WA* would be maintained. 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* (SCWA*).

Figure 4-1 shows the results of applying different weights to SCWA* when solving 63 instances of the 15 Puzzle. The y-axis represents the solution cost relative to optimal and the x-axis is the number of nodes expanded relative to Short Circuit. Each circle placed on

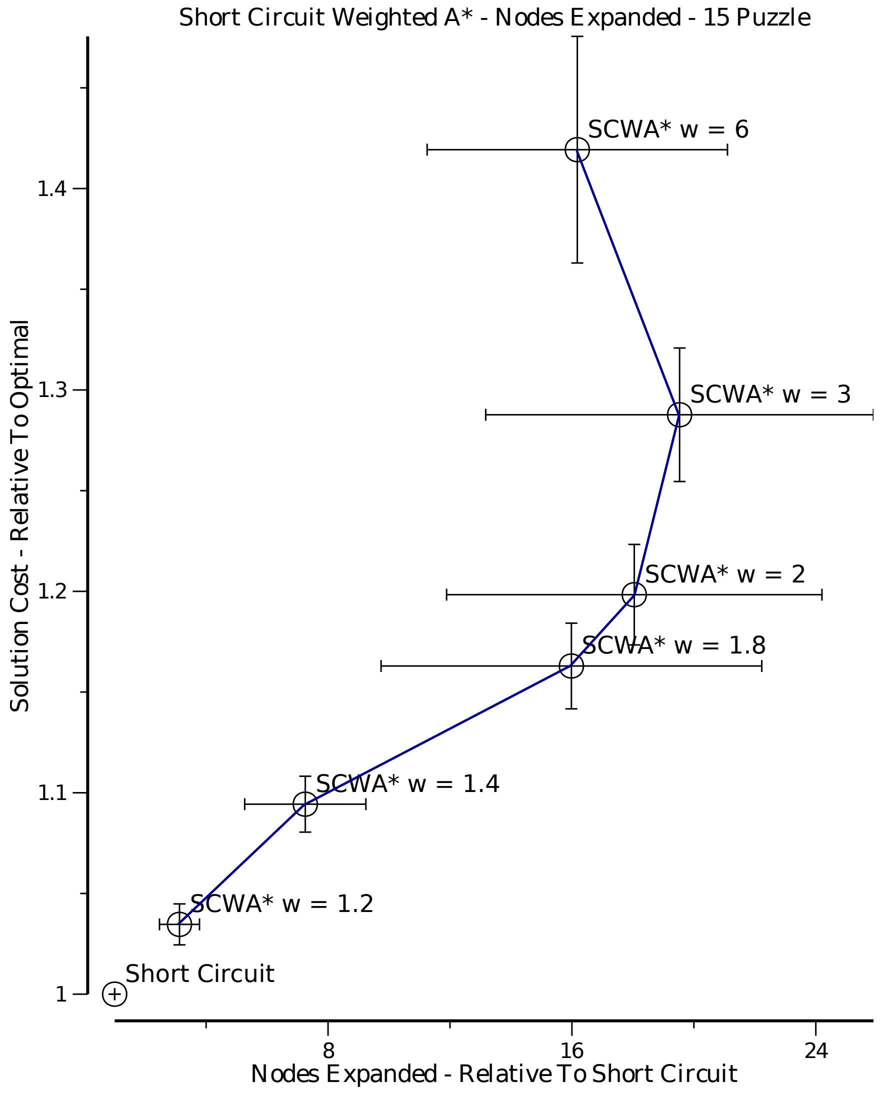


Figure 4-1: A plot showing relative to optimal solution cost and node expansion when wighting the lowest level of a Short Circuit search on 63 instances of the 15 Puzzle solved by all weights.

the plot represents SCWA* with various weights, 95% confidence intervals are also displayed for each axis.

Clearly, this approach does not produce the desired result. As the weight and solution cost increase SCWA* actually expands more nodes than using the optimal A* algorithm at the base. Figure 4-2 shows what happens to searches at the different abstraction levels as

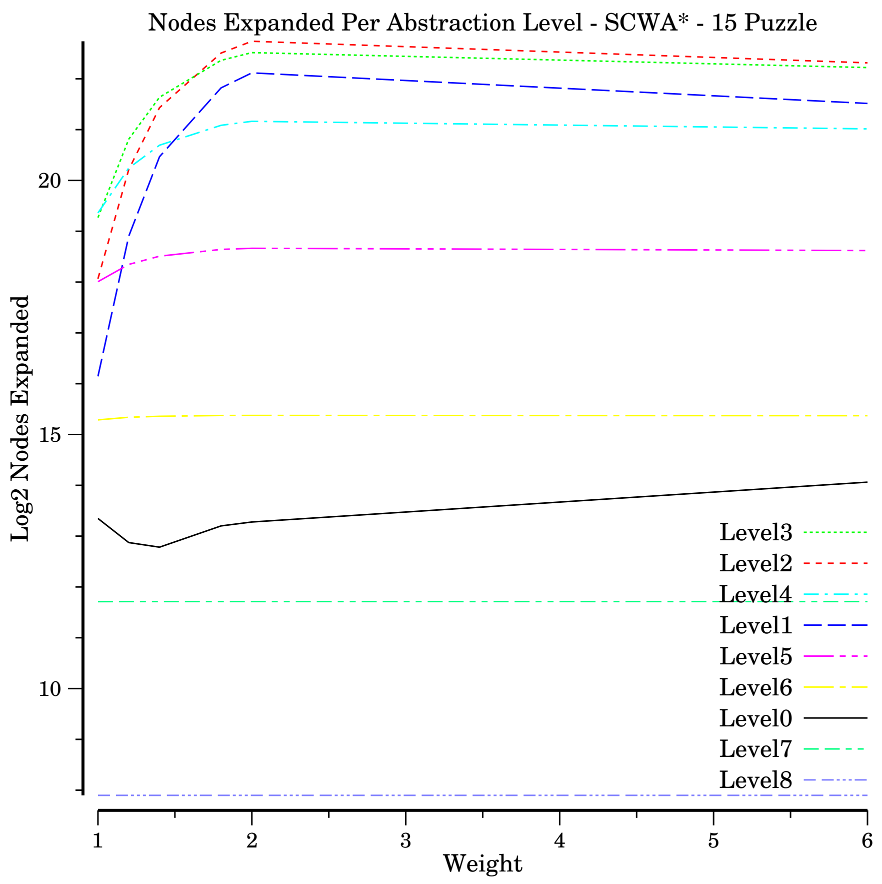


Figure 4-2: A plot showing the number of nodes expanded at each level of the abstraction hierarchy of Short Circuit with a number of different weights applied to the base level.

the weight in SCWA* is increased on the same instances of the 15 puzzle shown in Figure 4-1. The y-axis represents log_2 of the nodes expanded and the x-axis is the log_2 of the weight applied to the algorithm. Each line placed on the plot represents search at a different level of abstraction.

The number of nodes at the base level of SCWA* does indeed decrease when small weights are applied. However this decrease is quickly swamped by an increase of node expansion in the abstraction hierarchy. As mentioned by Larsen et al. (2010), state-of-the-art hierarchical search methods rely on a high level of cache hits to maintain their efficiency. Apparently suboptimal search at the base level worsens the cache behavior that hierarchical search relies upon and hence decreases performance. One possible cause for this behavior

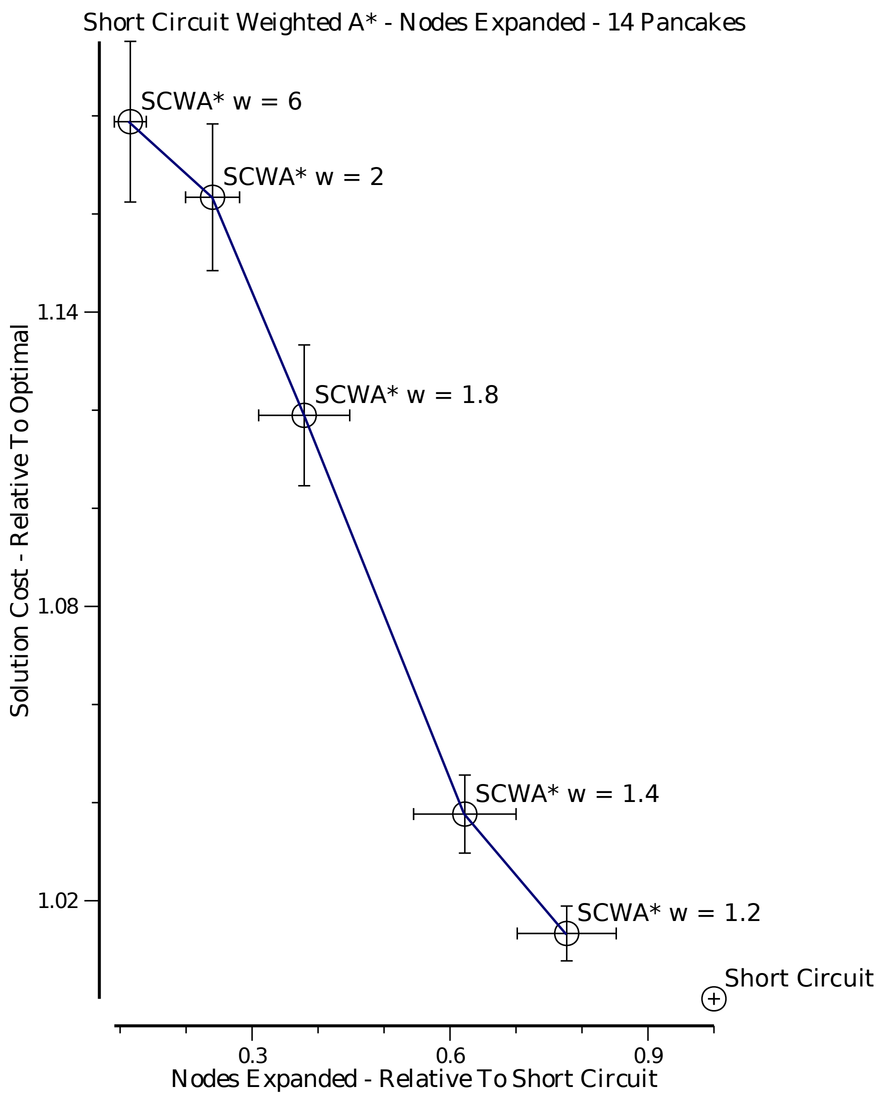


Figure 4-3: A plot showing relative to optimal solution cost and node expansion when wighting the lowest level of a Short Circuit search on 100 instances of the 14 Pancakes puzzle domain.

may be that the heuristic function guides the search into a large local minimum, and forces it to examine a large number of states with the same heuristic value before finding a solution.

The performance of SCWA* changes with both the domain and the abstraction used. Figure 4-3 show SCWA* running on the 14 Pancakes Puzzle. Here the results show that as the weight is increased the number of nodes expanded is decreased which is the opposite of what we see on the 15 Puzzle. Clearly, weighting the base of a hierarchical search does not always result in decreased performance. We have empirically tested the 15, Glued, Glued Two, Macro, 17-4 Topspin, and 14 Pancakes puzzles and found that only Pancakes and Topspin improved in terms of CPU time and node expansion by weighting the base level search using WA*. This result suggest that weighting the base of a hierarchical search will not work well on a broad range of domains.

4.2 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 one. 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. □

4.3 Switch

Using the observation above, we have created Switch, a new unbounded suboptimal search algorithm that places sub-optimality in the hierarchy used to generate heuristics for the search. The algorithm reverses the direction of a Switchback 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 4-4 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 inturn will start a search from S to G. Figure 4-5 shows the pseudo-code for the *FirstSearch* procedure that will be used to perform the first search from start to goal at

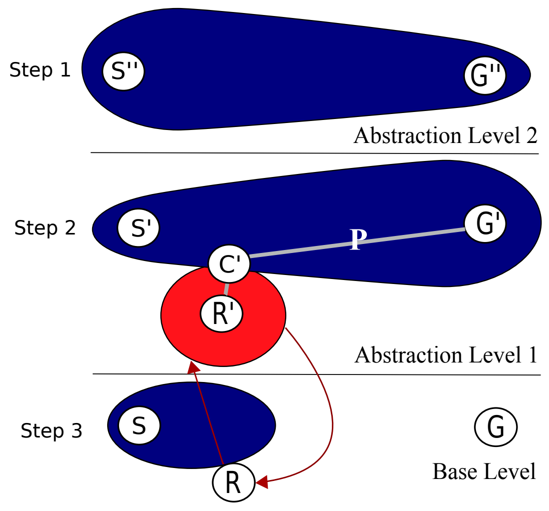


Figure 4-4: An example of the Switch algorithm on a 3-level abstraction hierarchy.

```
FIRSTSEARCH($i$, $closed$, $s$, $g$)
01. if $i \neq height\phi - 1$
02.     return FIRSTSEARCH( $i+1$, $closed$, $\phi(i+1, g)$, $\phi(i+1, s)$)
03. $open \leftarrow$ empty open list
04. insert $s$ $open$
05. while $open$ is not empty do
06.     $n \leftarrow$ remove node from $open$ with lowest $f$
07.     if $i$ is even then $children \leftarrow succs(n)$
08.     else $children \leftarrow$ preds$(n)$
09.     for each $c$ in $children$ do
10.         if $c$ in $closed_i$ then
11.             if $g(c) < g(n) + cost(n, c)$ then continue
12.             $g(c) \leftarrow g(n) + cost(n, c)$
13.             if $c$ is not $open$ then insert $c$ onto $open$
14.             continue
15.         $h(c) \leftarrow$ HEURISTIC($i$, $closed$, $c$)
16.         $g(c) \leftarrow g(n) + cost(n, c)$
17.         insert $c$ into $open$ and $closed_i$
18.     if $n = g$ then
19.         free $closed_{i+1}$ return $n$
20. return NULL
```

Figure 4-5: Pseudo-code for the *FirstSearch* procedure used in the Switch algorithm.

```
SearchBack($i, cache, s$)
01. if $s$ is in $cache_i$ then return $s$
02. $open$ ← empty open list; $closed$ ← empty closed list
03. $q \leftarrow \phi(i, s)$
04. insert $q$ into $open$
05. while $open$ is not empty do
06.     $n$ ← remove node from $open$ with lowest
07.     if $i$ is even then $children \leftarrow succs(n)$
08.     else $children \leftarrow$ preds$(n)$
09.     for each $c$ in $children$ do
10.         if $c$ in $closed$ then continue
11.         $g(c) \leftarrow g(n) + cost(n, c)$; $h(c) \leftarrow 0$
12.         if $c$ in $cache_i$ then
13.             $d \leftarrow cache_i(c)$
14.             $P \leftarrow g(c) + g(d)$
15.             for each $a$ in path from $q$ to $c$
16.                 $g(a) \leftarrow g(c) - g(a) + g(d)$
17.                 insert $a$ into $cache_i$
18.             for each $b$ in $open$ and $closed$
19.                 $g(b) \leftarrow P + g(b)$
20.                 insert $b$ into $cache_i$
21.             $g(q) \leftarrow P$; return $q$
22.         insert $c$ into $open$
23.     insert $n$ into $closed$
24. return NULL
```

Figure 4-6: Pseudo-code for the *SearchBack* procedure used in the Switch algorithm.

each level. Four parameters are required, i is the abstraction level to be searched, *closed* is the closed list (and cache) used for that level, and s and g are the start and goal states respectively.

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. Figure 4-6 displays the pseudo-code that performs the uniform cost search back to the cache. The parameter *cache* is the closed list that was used in the original search and s is the query node.

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. Line 19 of the *SearchBack* procedure frees the appropriate cache when it is no longer required.

In an attempt to reduce the total number of node 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.

The remaining pseudo-code for the Switch algorithm can be found in Figure 4-7 and 4-8. The *Switch* procedure creates the closed lists for each abstraction level. The lists will also be used as the cache in the secondary search. It then calls the *FirstSearch* procedure that will begin that will start search at the highest abstraction level. The *Heuristic* procedure will

```
SWITCH()
01. $closed \leftarrow$ array of length $height_\phi$ of empty closed lists
02. for $i \leftarrow 0$ up to $height_\phi - 1$ do
03. $result \leftarrow$ FIRSTSEARCH(0, $closed$, $s_{start}$, $s_{goal}$)
04. if $result \neq$ NULL then return EXTRACT-PATH($result$)
05. return NULL
```

Figure 4-7: Pseudo-code that will start a Switch search.

```
HEURISTIC($i$, $closed$, $s$)
01. if $i = height_\phi - 1$ then return $\epsilon(s)$
02. $n \leftarrow$ lookup$\phi(i+1, s)$ in $closed_{i+1}$
03. if $n \neq$ NULL then return $g(n)$
04. $r \leftarrow$ SEARCHBACK($i+1$, $closed$, $s$)
05. if $r =$ NULL then return $\infty$ else return $g(r)$
```

Figure 4-8: Pseudo-code for the *Heuristic* procedure used in the Switch algorithm.

look in the cache of the level above for the query node. If found its cost value is returned, otherwise *SearchBack* is called and will perform the uniform cost search back to the cache of the level above.

4.4 Link To Refinement

Solution refinement, as described in section 2.4, uses steps in the abstract solution path as sub-goals for search in the original problem. If a state is expanded in the original problem that does not map to a state on the abstract solution path, it is ignored; given a heuristic value of infinity. Refinement is guaranteed to be complete if the abstraction used is *solution preserving.* The modern instance-specific homomorphism abstraction presented in this thesis do not have this property.

Like refinement, Switch finds a full solution path in the abstraction before search begins in the original problem. However, when a state is expand in the original problem that does not map to a state on the abstract solution path, a non-admissible heuristic value will be returned. Since a suboptimal path in the abstraction was used to generate this

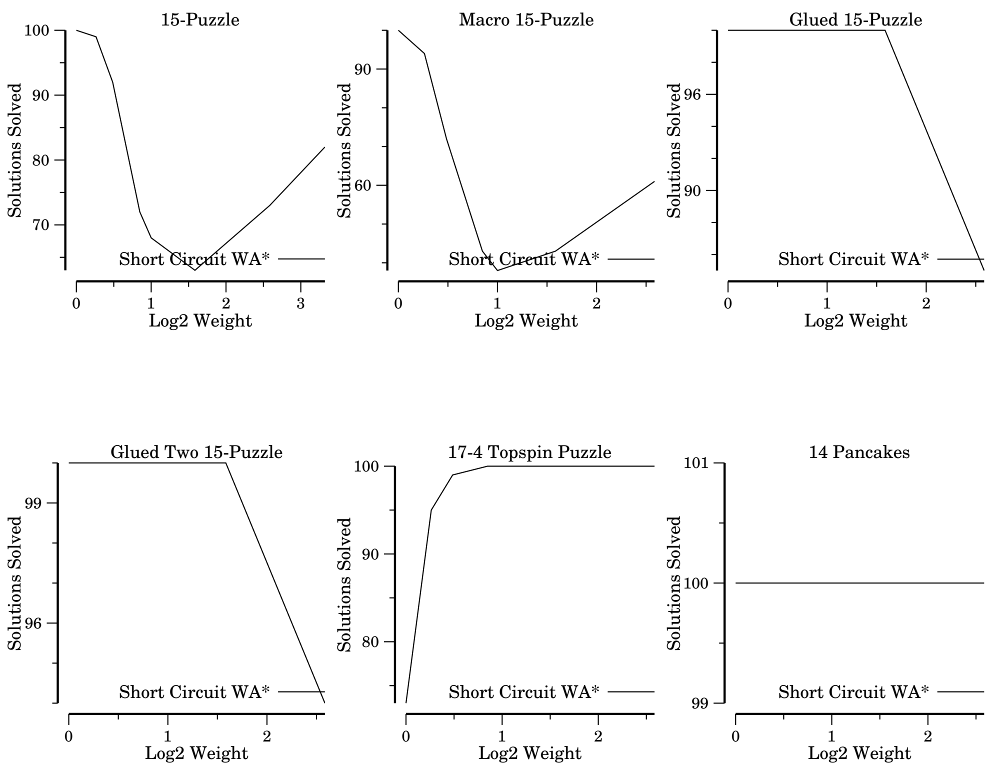


Figure 4-9: Plots for each domain presented in the evaluation that show the number of instances SCWA* could solve with varying weights used at the lowest level of search.

heuristic value, it will likely be larger than those that are on or close to the abstract solution path. This will penalize these nodes, encouraging search in the original problem to follow the abstract solution path. This essentially makes Switch 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.

4.5 Evaluation

In order to gain a better understanding of Switch we implemented the algorithm in C++ and compared it to SCWA* on the same six domains that were used to evaluate Short Circuit. 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 47GB.

Since the performance of Short Circuit often decreased as the weight was increased, all instances of all domains where not solved with each weight. Figure 4-9 displays a line plot for each domain that displays the number of instances that where solved by SCWA* with each weight tested. The y-axis represents the number of instances solved and the x-axis represents the log_2 of the weight that was used. The solid black line placed on each plot is the SCWA* algorithm.

4.5.1 15-Puzzle

The first domain is the 15 Puzzle previously described in section 3.3.1. We used the same abstraction hierarchy described in the previous evaluation for both algorithms. The top left panel of Figure 4-9 shows that while the optimal Short Circuit algorithm was able to solve all 100 instances of this domain SCWA* was not able to with some of the weights in the evaluation. With a weight of 3 for example, SCWA* could only solve 63 instances. Switch was able to solve all 100 of the instances. Figure 4-10 displays the cost and node expansion results for this domain on the 63 instances that SCWA* could solve with all weights. The y-axis represents the solution cost relative to optimal of each algorithm, the x-axis represents the number of nodes expanded relative to Short Circuit. As mentioned previously, as the weight in SCWA* is increased in this domain, the algorithm returns solution with larger solution cost and expands more nodes to do so. The Switch algorithm returned solutions that were an average of 1.3 of optimal and expanded only 2% of the nodes that were expanded by Short Circuit. WA* using Manhattan Distance has also been placed on this plot as a reference. This algorithm expanded significantly fewer nodes than the hierarchical search algorithms at weights larger than 1.8. This however was the expected result since hierarchical search is not particularly well suited for this domain.

4.5.2 Macro 15 Puzzle

The next domain is the Macro 15 Puzzle described in section 3.3.2. We used the same abstraction hierarchy described in the previous evaluation for both algorithms. The top

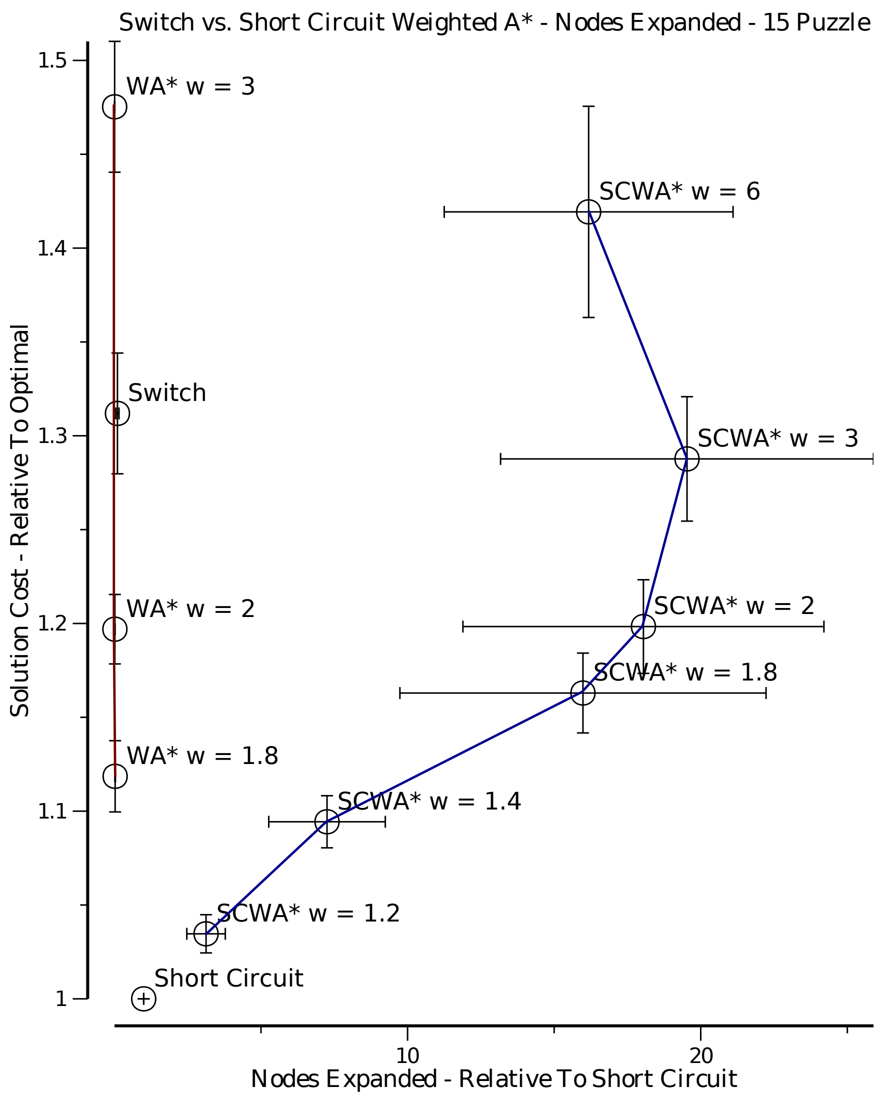


Figure 4-10: A plot showing solution cost and node expansion results for Switch and Short Circuit when weighting the lowest level of search on 63 instances of the 15 Puzzle solved by all weights.

center panel of Figure 4-9 shows that optimal Short Circuit was able to solve all 100 instances while SCWA* with a weight of 2 was only able to solve 38 of the instances. Figure 4-11 shows the cost and node expansion results for these 38 instances. Switch consistently returned solution that were a factor of 1.43 of optimal on average while expanding 3% of the nodes required by Short Circuit. The solution costs returned by SCWA* increased with weight as

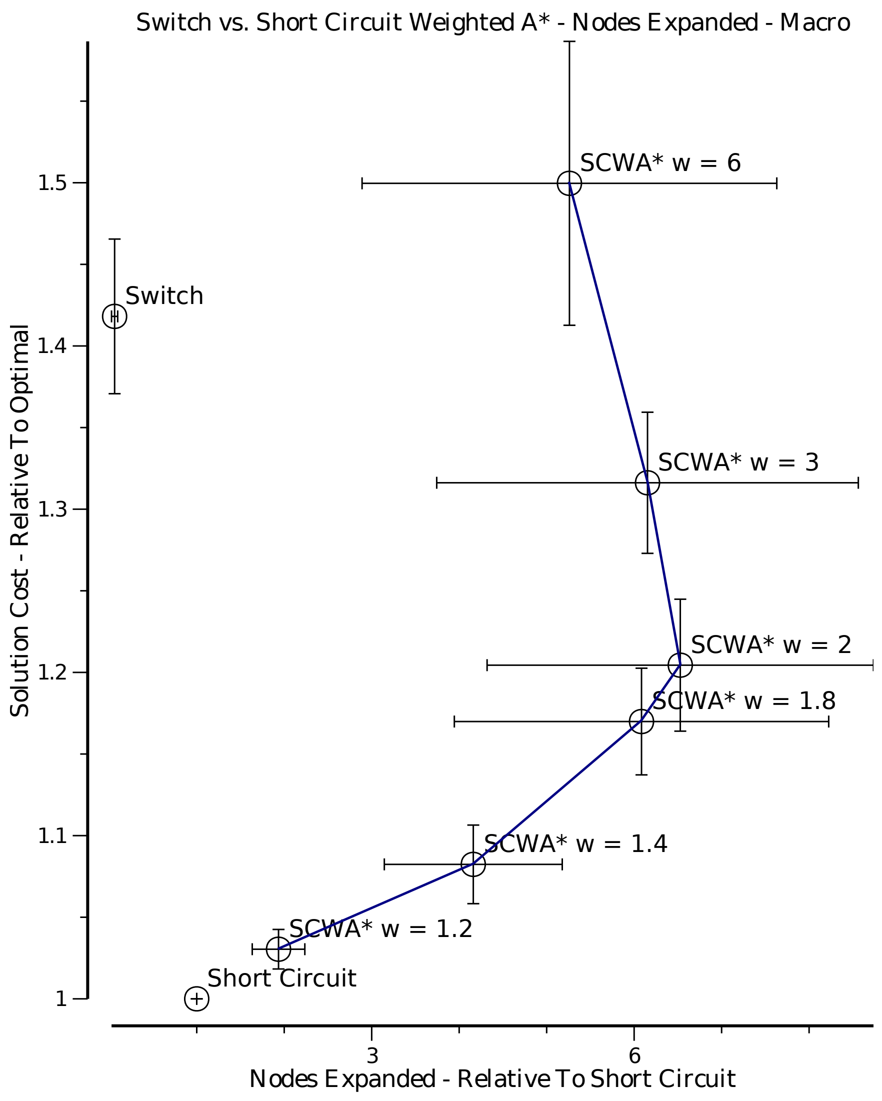


Figure 4-11: A plot showing relative to optimal solution cost and node expansion for Switch and SCWA* on 38 instances of the Macro 15 Puzzle solved by all algorithms and weights.

well as the number of nodes expanded.

Since SCWA* was only able to solve a small number of instances, we compared Switch to Short Circuit directly. The results can be seen in Figure 4-12. The x-axis represents the number of nodes expanded by Short Circuit. The y-axis is the number of nodes expanded by Switch. The line represents y=x. A circle has been plotted for each of the 100 instances. Switch expands fewer nodes than Short Circuit on each instance, ranging from only twice as

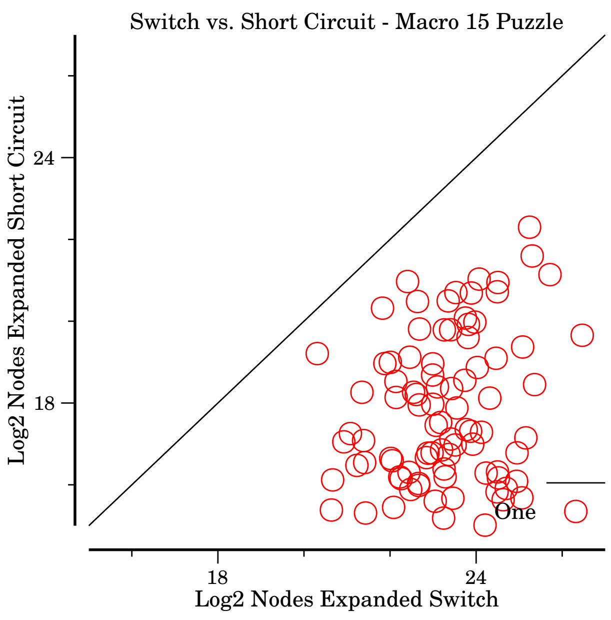


Figure 4-12: A plot that shows node expansion results for Short Circuit and Switch on the Macro 15 Puzzle.

fast to 40 times faster than the optimal algorithm. A closer look at the results shows that Switch returned solutions that were a factor of 1.4 times worse than optimal on average and expanded only 3% of the nodes expanded by Short Circuit.

4.5.3 Glued 15 Puzzle

The same homomorphic instance specific abstraction described in section 3.3.1 was used for all of the instances of the Glued 15 Puzzle domain. The top right panel of Figure 4-9 displays results for the number of instances solved by SCWA* with varying weights. Switch was able to solve all 100 instances, while SCWA* with a weight of 6 was able to solve only 85. Figure 4-13 shows the cost and node expansion results for this domain. Switch was able to solve all instances within a factor 1.84 of optimal with 27% of the nodes expanded by Short Circuit on average. SCWA* failed to solve all instances when using a weight of 3 or larger and always expanded more nodes than required to solve the instance optimally by Short Circuit.

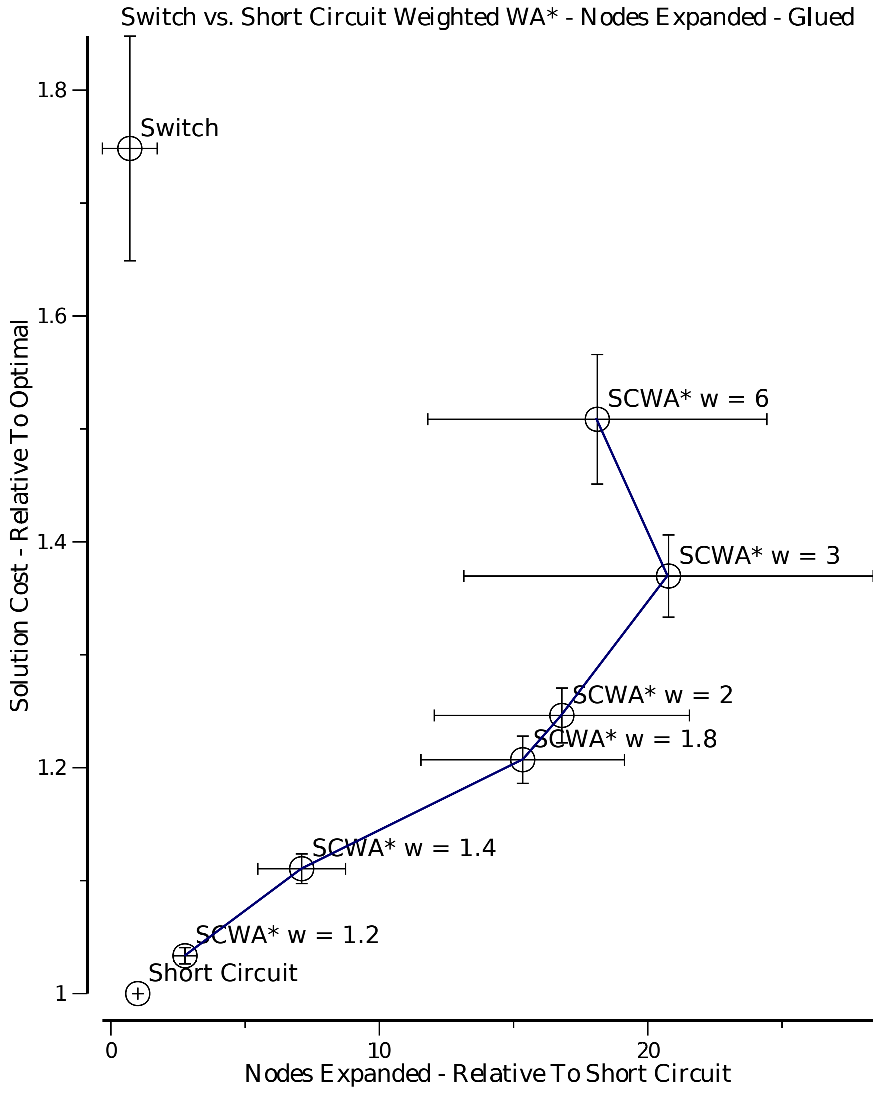


Figure 4-13: A plot that shows solution cost and node expansion results for Switch and Short Circuit WA* with various weights on the Glued 15 Puzzle.

4.5.4 Glued Two 15 Puzzle

The Glued Two 15 puzzle used the same abstraction described in section 3.3.4. The bottom left panel of Figure 4-9 displays the number of instances that were solved by SCWA* with various weights on this domain; Switch was able to solve all 100 instances while SCWA* with a weight of 6 could only solve 92. Figure 4-14 shows solution cost and node expansion results for this domain. The solutions returned by Switch were a factor of 1.9 of optimal

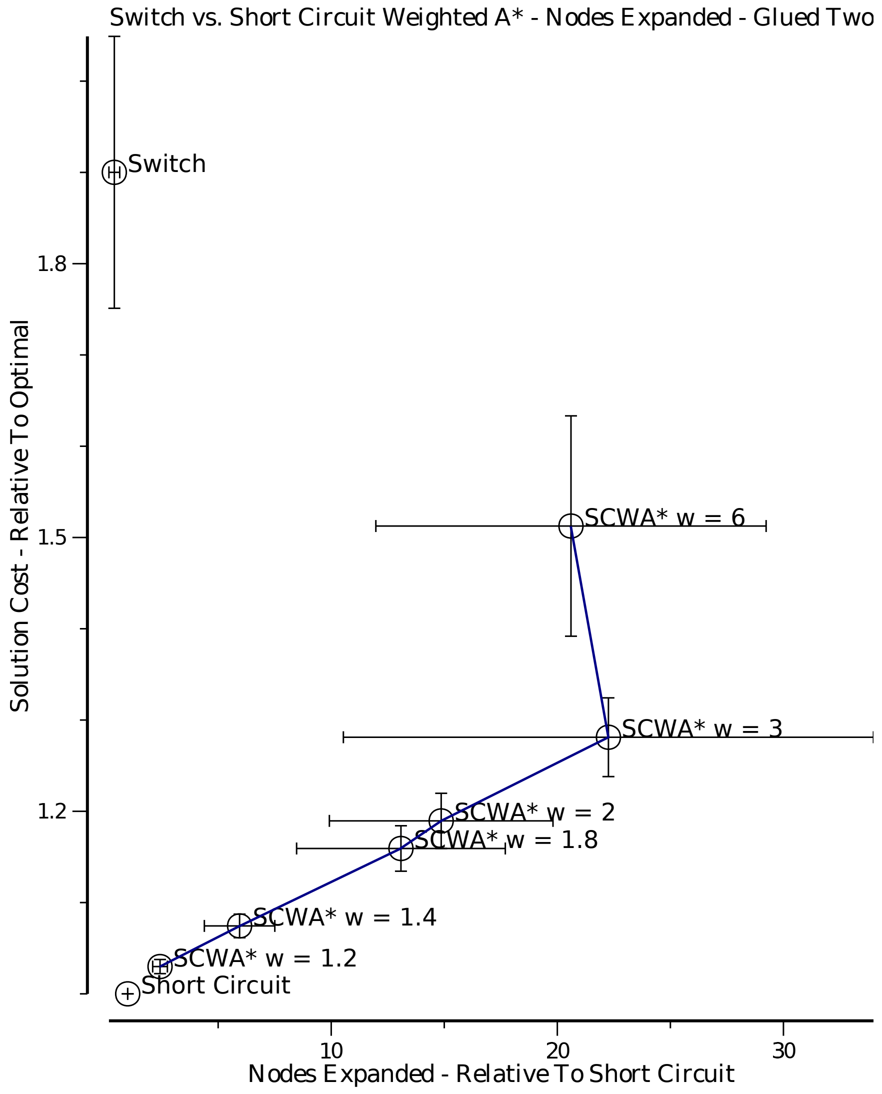


Figure 4-14: Plot that shows solution cost and node expansion results for Switch and SCWA* with various weights on the Glued Two 15 Puzzle.

and the algorithm expanded 31% of the nodes expanded by Short Circuit on average.

4.5.5 14 Pancakes Puzzle

For the 14 Pancakes Puzzle domain we used the same cost function and abstraction hierarchy that is described in section 3.3.4. Figure 4-9 is a plot that shows that SCWA* was able to solve all 100 instances of this puzzle with all weights; Switch was also able to solve all 100

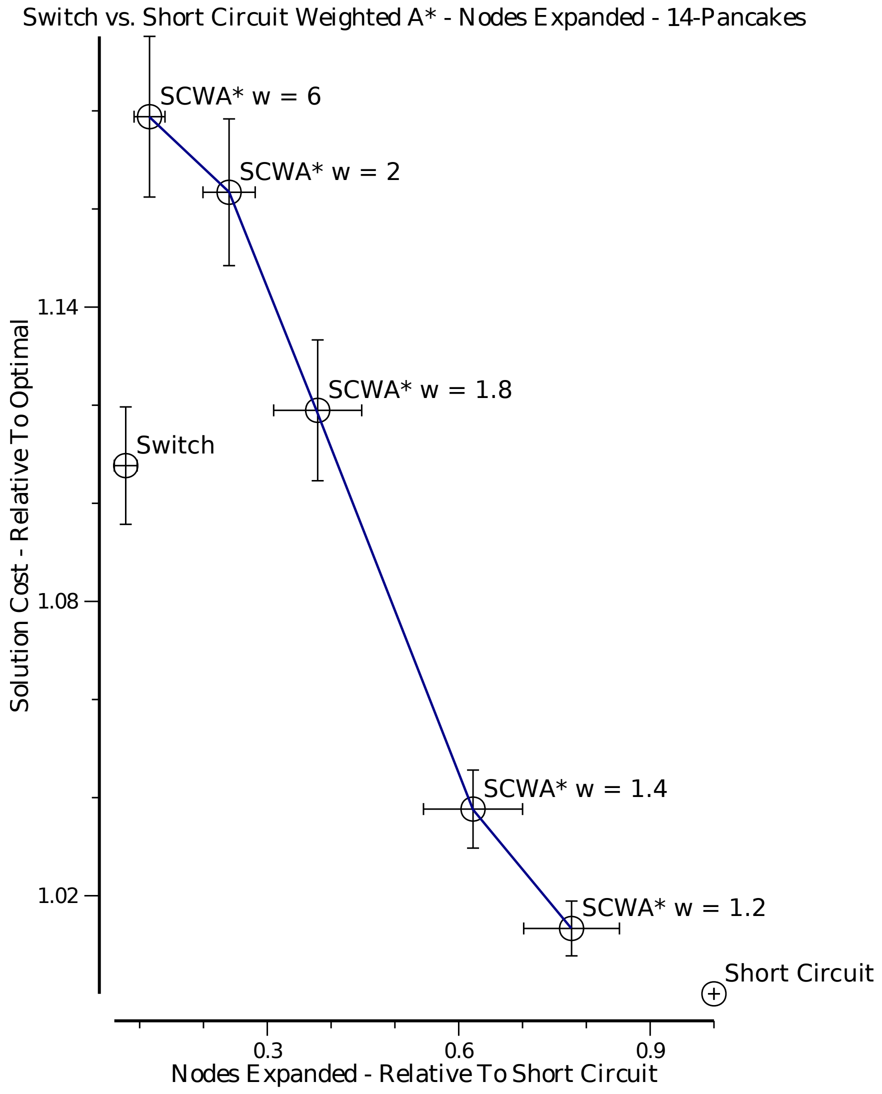


Figure 4-15: A plot showing relative to optimal solution cost and node expansion for Switch and SCWA* on 100 instances of the 14 Pancakes Puzzle.

instances. On this domain the number of nodes expanded by SCWA* decreased as weight was increased as shown in Figure 4-15. SCWA* with a weight of 6 expanded fewer nodes than Switch but the solution cost returned by the algorithm were larger. Switch returned a solution cost factor of 1.12 on average. SCWA* with a weight of 6 was 1.18. Clearly, Switch is superior to SCWA* on this domain even though node expansion results improved with weight.

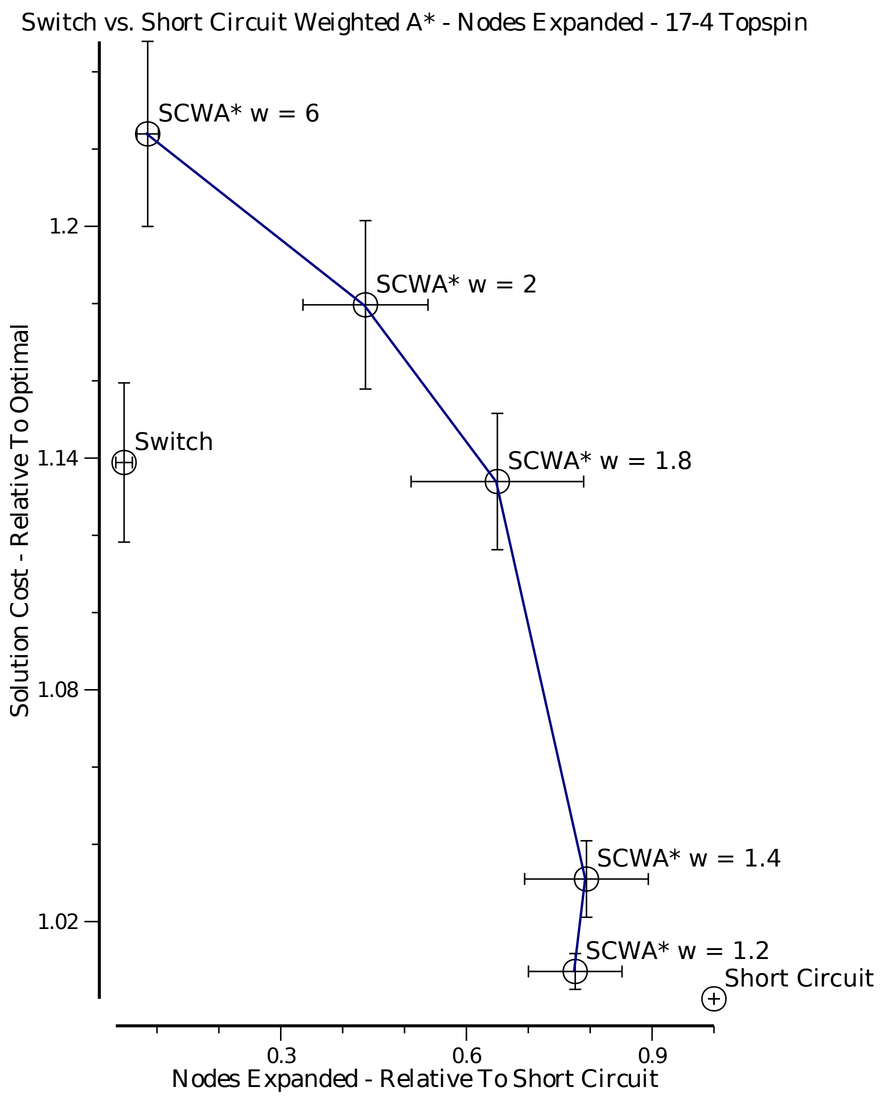


Figure 4-16: A plot showing relative to optimal solution cost and node expansion for Switch and SCWA* on 73 instances of the Macro Tiles Puzzle solved by all algorithms and weights.

4.5.6 17-4 Topspin Puzzle

We used the same cost function and abstraction hierarchy described in section 3.3.5. Figure 4-9 shows that optimal Short Circuit was only able to solve 73 of the instances but SCWA* with a weight of 1.2 and above was able to solve all of the instances. According to Figure 4-16 Switch still expanded fewer nodes than SCWA* at any weight; expanding 85% percent of SCWA* with a weight of 6. The solutions returned by Switch were within a

Domain	Algorithms	Nodes Expanded (100 K)			CPU Time (sec)		
		Mean	Std Dev	% Opt	Mean	Std Dev	Sub Opt
15 Puzzle	SCWA* w=1	87	191	100	20.7	58.3	1
	Switch	2.2	6.6	2.5	0.5	1.9	1.3
Macro 15 Puzzle	SCWA* w=1	137	132	100	76.9	85.9	1
	Switch	4.7	7.7	3.4	1.9	3.5	1.4
Glued 15 Puzzle	SCWA* w=1	22	43	100	4.8	11.5	1
	Switch	8	23.3	36	1.8	5.3	1.7
Glued Two 15 Puzzle	SCWA* w=1	8.3	22.9	100	1.9	5.9	1
	Switch	2.6	13	31	0.59	3.1	1.9
17-4 Topspin	SCWA* w=6	5.6	2.4	6.3	8.1	4.2	1.2
	Switch	3.7	4.6	4.2	7.8	10.9	1.1
	SC	89	42	100	207	110	1
14 Pancakes	SCWA* w= 6	1.1	0.6	7	0.9	0.7	1.2
	Switch	1.4	2.1	7	2.2	3.7	1.1
	SC	20	21	100	26	35	1

Table 4-1: A summary table that shows node expansion, CPU time, and sub-optimality for Switch and SCWA* with the weight that expanded the fewest nodes. On domains for which SCWA* with a weight of 1 was not the best weighted variant Short Circuit is also displayed.

factor of 1.13 of optimal. SCWA* with a weight of 6 was 1.22. SCWA* with a weight lower than 2 returned solutions that were on average closer to optimal than switch. However it took significantly more node expansions to get these solutions.

4.5.7 Summary of Results

Table 4-1 shows a summary of results for Switch and SCWA* for all domains [1] in the evaluation. Column 2 displays the weighted variant of SCWA* that expanded the fewest

[1]The row for the 17-4 Topspin Puzzle only show results for the 73 instances that could be solved optimally.

nodes overall. SCWA* with a weight of 1 expanded the fewest nodes on all 15 puzzle variants, a weight of 6 performed the best on Topspin and Pancakes [2]. Column 3 and 4 displays the mean and standard deviation of nodes expanded by each algorithm. Switch expanded the fewest nodes on five of the domains, SCWA* with a weight of 6 expanded fewer on Pancakes. The column entitled "% Opt" displays the percent of nodes expanded relative to optimal and shows similar results. Column 6 and 7 displays the mean and standard deviation of CPU time. Switch was faster on the first five domains and slower on Pancakes. The last column titled 'Sub Opt' displays the average sub-optimality factor returned by each algorithm. On four of the domains, SCWA* with a weight of 1 expanded the fewest nodes, and thus returned optimal solutions. On the Topspin and Pancakes domain Switch returned the lowest cost solution on average.

4.6 Conclusion

In this chapter we have presented a new algorithm, Switch, specifically designed to find suboptimal solutions quickly using hierarchical search. Switch is essentially a softened version of abstract solution refinement that can use modern homomorphic abstractions. Like abstract solution refinement Switch attempts to follow the abstract solution path. Unlike refinement however, states that do not map to states on or near the abstract solution path are not given a heuristic value of infinity, they are penalized with larger suboptimal heuristic values. This places the sub-optimality in the abstraction hierarchy rather then the base level search. Our empirical results suggest that this approach finds higher quality solutions faster than simply using a standard suboptimal algorithm at the base of a hierarchical search.

[2]Short Circuit (SC) results are shown in the table for the 17-4 Topspin and 14 Pancakes puzzles since SCWA* with a weight of one was not the best performing weighted variant.

CHAPTER 5

Multiple Abstractions

Current hierarchical search approaches have either used a single level abstraction hierarchy, similar to the ones used in the sections above, or have taken the maximum over the heuristics returned by multiple abstraction hierarchies. Korf and Felner (2002) show that it is possible to add heuristics generated from different abstractions together and still maintain optimality as long as the abstractions are disjoint. Here we discuss using this approach in hierarchical search.

5.1 Multiple Additive Abstractions

Korf and Felner (2002) have shown that more accurate admissible heuristic functions can be generated by adding together the result of multiple heuristic values created from disjoint abstractions. When applied to pattern databases this approach significantly improved search performance. To create a set of these abstractions each abstraction must be a disjoint subset, such that, each operator only affects sub-goals in one subset. Informally, we must construct each of the abstractions so that we don't double count the moves of an operator. The linear abstractions described in the previous sections can only take in account the interactions of distinct items, and its not possible to make each item distinct without solving the original problem. Using multiple abstractions allows us to make each item distinct in at least one abstraction which can produce more accurate heuristic values.

In many domains a non-additive abstraction can be turned into an additive abstraction by simply changing the cost function and ensuring that the distinct items are disjoint in each abstraction. For example, a non-additive abstraction for the 15 Puzzle can be modified so that it is additive by changing the cost function so that moves of indistinguishable tiles

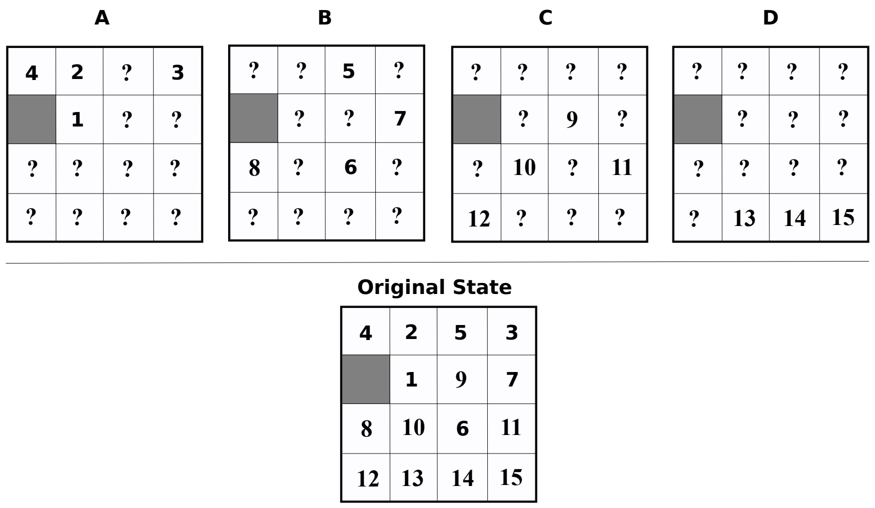


Figure 5-1: An example 15 Puzzle state that has been mapped to four different additive abstractions.

have a cost of 0 and no tile is distinct in more than one abstraction. Figure 5-1 gives an example 15 Puzzle state that has been mapped to four different disjoint abstractions. Each abstraction has been labeled A-D. In abstraction A tiles 1-4 are distinct, in B 5-8 are distinct, in C 9-12 are distinct, and in abstraction D 13-15 are distinct. If the heuristics generated by each of these abstractions are added together the resulting heuristic will still be admissible.

Creating additive abstractions by modifying the cost function so that only moves of distinct tiles are counted will not work on domains were an operator can move more than one distinct item at a time. For example, in the 17-4 Topspin Puzzle the turnstile may move multiple tiles, some indistinguishable and some distinct, with a single move at a cost of 1. Because of this, there is no easy way to create the abstractions so that a move in one abstraction is not counted in another. A technique known as *Cost Splitting* was presented by Yang et al. (2008) to solve just this problem. This technique works by splitting the cost of a move by the ratio of distinct tiles over the total number of tiles moved. For example, if two distinct tiles and two indistinguishable tiles were moved by the turnstile in a 17-4 Topspin Puzzle, we would give the move a cost of $1 * 2/4 = 0.5$. This general approach

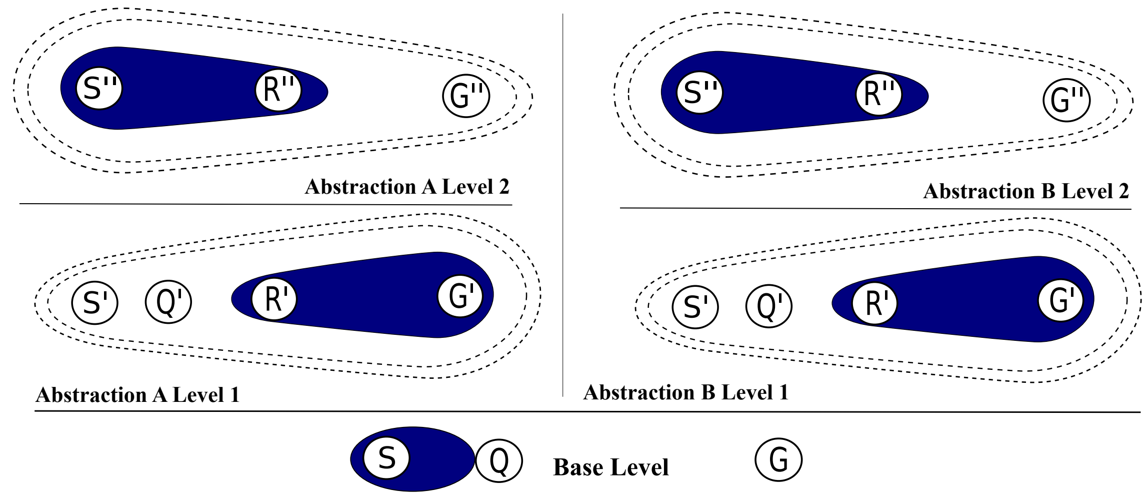


Figure 5-2: An example of SCMA using two separate abstraction hierarchies each of which has 2 levels.

makes it possible to create additive abstractions for a large number of domains.

5.2 Short Circuit With Multiple Abstractions

We propose a modification to the Short Circuit algorithm that uses multiple disjoint abstractions to create admissible heuristics for an A* search. If multiple abstractions are used to generate the heuristic for a Short Circuit search we refer to the algorithm as SCMA.

Figure 5-2 gives an example of SCMA using two abstraction hierarchies that each have two levels. Search begins in the original problem using A*. The start node is expanded and Q is generated. To create the heuristic for Q, searches are started sequentially in abstraction hierarchy A then abstraction hierarchy B. These searches are identical to the searches that would happen in the first level of abstraction in a typical Short Circuit search. The direction of search is changed at each level of the abstraction hierarchy, each abstraction level search is driven by a single abstraction above, and the highest level of search is driven by the ϵ heuristic.

Figure 5-3 shows how the *Heuristic* procedure can be modified to use multiple abstractions. A closed list is kept for each level of each different abstraction used. If a heuristic

```
Heuristic($i$, $open$, $closed$, $s$, $abs$)
01. if $i = height_\phi - 1$ then return $\epsilon(s)$
00. if $i = 0$ then
00.     $sum = 0$
00.     for $a$ in $abstractions$
00.         $n \leftarrow$ lookup$\phi(i+1, s, a)$ in $closed_{a_{i+1}}$
03.         if $n \neq$ NULL then $sum+ = g(n)$
04.         $r \leftarrow$ Resume($i+1$, $open_{abs}$, $closed_a$, $\phi(i+1, s, a)$)
05.         if $r =$ NULL then return $\infty$ else $sum+ = g(r)$
00.     return $sum$
02. $n \leftarrow$ lookup$\phi(i+1, s, abs)$ in $closed_{abs_{i+1}}$
03. if $n \neq$ NULL then return $g(n)$
04. $r \leftarrow$ Resume($i+1$, $open_{abs}$, $closed_{abs}$, $\phi(i+1, s, abs)$)
05. if $r =$ NULL then return $\infty$ else return $g(r)$
```

Figure 5-3: Pseudo-code for the *Heuristic* procedure used in SCMA.

value is required for the base level, each closed list at abstraction level one is queried, the results are summed, and the heuristic is returned. When a value is not found in the closed lists of one of the abstraction hierarchies, a search is started at that abstraction hierarchy. If a heuristic is required for any other level it is looked up as normal.

5.3 Overhead

Using multiple abstractions to drive search in the original problem has significant overhead. Overall CPU time is impacted since multiple hierarchies must perform search every time a heuristic is required. Memory is also affected since the open and closed lists for each abstraction level of each abstraction must be kept in memory. The size and number of abstractions used in SCMA can have a significant impact on its performance. Larger abstractions will likely produce more accurate heuristics since they account for more of the important interactions present in the original problem. These abstractions however can introduce additional overhead since they themselves are harder to search and require a larger abstraction hierarchy. This effect will be compounded in SCMA since there are many abstraction hierarchies.

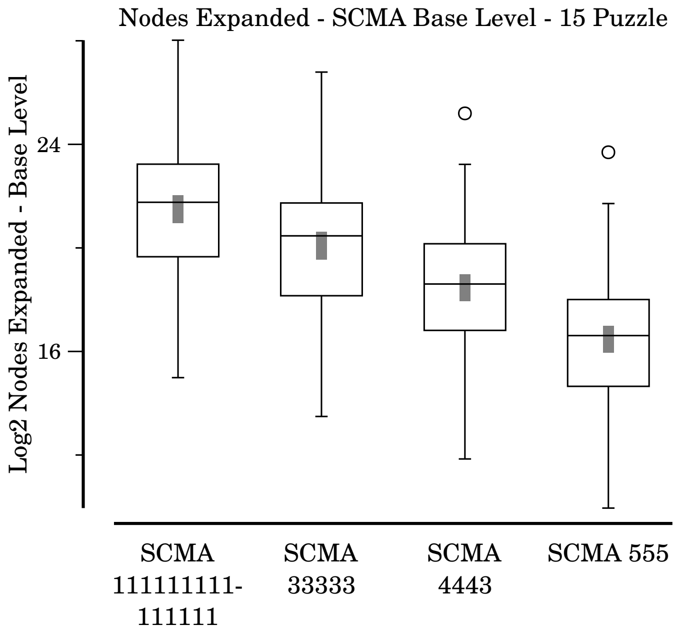


Figure 5-4: A box plot showing the number of nodes expanded in the base level of SCMA for a number of different abstractions.

Individual additive abstractions will produce heuristics that are less accurate than their non-additive counterparts of the same size and may require more memory when searching. This comes from the fact that additional information in each state must be stored for expansion but can not add to the heuristic value generated. For example, when generating the 555 abstraction for the 15 Puzzle, five tiles are discrete in each abstraction and 10 are indistinguishable. Each state in the abstraction must also keep track of the blank tile so that the expand function is able to generate its successors. The position of the blank tile, however, will not contribute to the heuristics generated since moves of indistinguishable tiles have a cost of 0. Two states with all five discrete tiles in the same positions but blanks in different positions will have an identical cost value and thus produce an identical heuristic value. This is not the case for non-additive abstraction where the cost of the indistinguishable tiles is non-zero. This also means that the number of states cached in the closed list of the abstraction hierarchy could actually be larger than an equivalent pattern database since it would ignore locations of the blank.

Figure 5-4 shows the number of nodes expanded by the base level of SCMA on 100 instances of the 15 Puzzle for a number of different abstraction hierarchies. The y-axis represents the log_2 number of nodes expanded at the base level of an SCMA search. The x-axis displays a box plot for each abstraction evaluated. The additive abstractions were create by splitting the puzzle into a number disjoint pieces. For example, to create the 4443 additive abstraction we split the puzzle into 4 disjoint pieces, tiles 1-4 were distinct in the first abstraction, 5-8 in the next, then 9-12, and finally 13-15 in the last. As the size of the abstractions used is increased, the number of nodes expanded at the base is decreased. The abstraction that used 15 1 tile abstractions, essentially the Manhattan distance heuristic, expanded the most nodes, while the 555 abstractions expanded the fewest. On this domain larger abstractions do indeed produce more accurate heuristics and reduce the number of nodes expanded at the base.

Figure 5-5 shows the total number of nodes expanded, including nodes expanded in the abstraction hierarchy, for the same instances and abstraction hierarchies used in Figure 5-4. The y-axis represents the log_2 number of nodes expanded. The x-axis shows the a box plot for each abstraction hierarchy. As the abstraction hierarchies increase in size, the total number of nodes expanded also increases. Evidently, the decrease in node expansion at the base can not overcome the increased node expansion in the abstraction hierarchy when larger abstractions are used. The performance of SCMA is then undoubtedly tied to the size and number of abstractions used.

5.4 Optimal Evaluation

In order to gain a better understanding of how the use of multiple additive abstractions would affect search performance, we implemented SCMA in C++ and compared it to Short Circuit with a single abstraction on several domains. Like our previous evaluations, each instance was given 47 GB of RAM and unlimited running time on a dual quad-core Xeon running at 2.66 GHz.

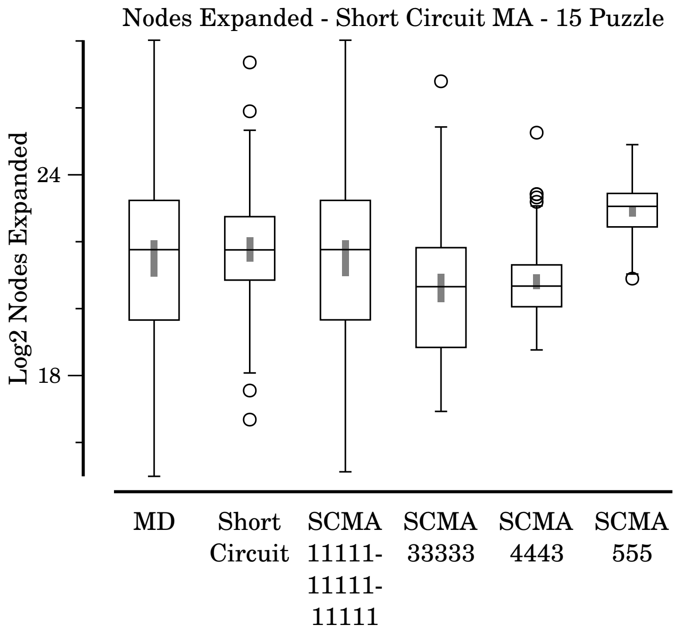


Figure 5-5: A box plot showing node expansion results for several abstraction hierarchies on 100 instances of the 15 Puzzle.

5.4.1 15 Puzzle

The first domain is the standard 15 puzzle described in section 3.3.1. Here Short Circuit used the standard instance specific homomorphic abstraction previously described. We evaluated SCMA on several different additive abstractions. The additive abstractions were create by splitting the puzzle into a number of disjoint pieces as described in section 5.3. Figure 5-5 shows the results for the 33333, 4443, and 555 additive abstractions for all 100 instances of the 15 puzzle used in the previous evaluations. The y-axis represents the log_2 average nodes expanded by each abstraction including all node expansions in the abstraction hierarchy. The x-axis has a box-plot for each version of the additive abstractions; Short Circuit with a standard linear abstraction, and A* with the Manhattan distance heuristic (titled MD on the plot). The 4443 additive abstraction expanded fewer nodes on average than the other abstractions and A* with the Manhattan distance heuristic.

The left panel of Figure 5-6 shows node expansion results for each instance of the 15 Puzzle for Short Circuit with linear and additive abstraction hierarchy. The x-axis

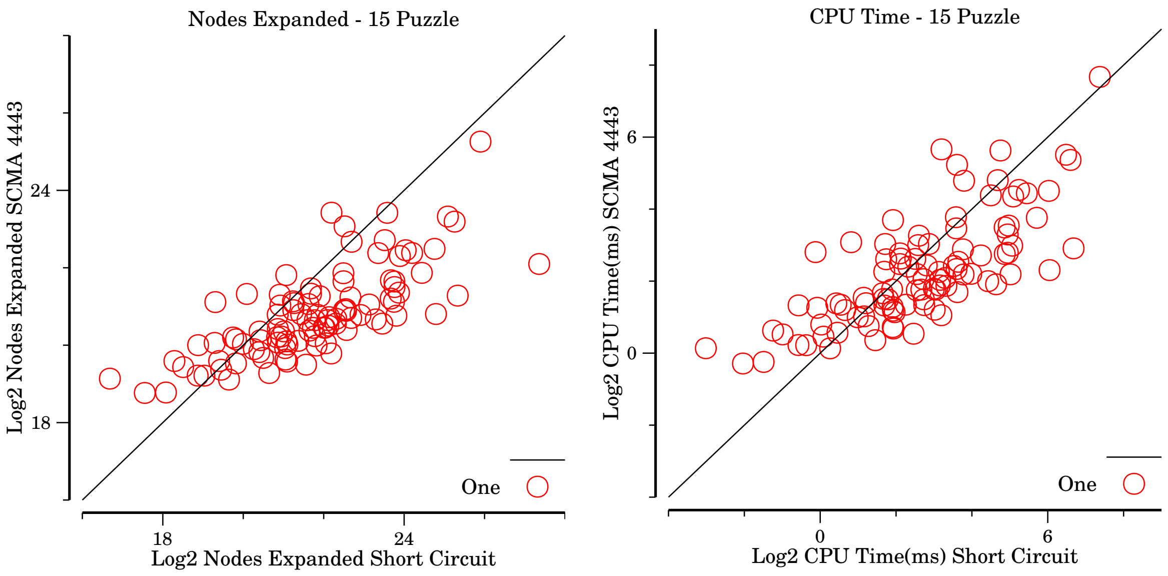


Figure 5-6: Scatter plot that shows node expansion and CPU time results for Short Circuit with a linear abstraction hierarchy and with an additive abstraction hierarchy on 100 instances of the 15 Puzzle.

represents the log_2 nodes expanded by Short Circuit with a linear abstraction hierarchy. The y-axis represents the log_2 nodes expanded by SCMA 4443. While SCMA 4443 expands fewer nodes on average, many of the simpler instances are solved by the linear abstraction hierarchy with fewer node expansions. The right panel of Figure 5-6 shows a similar plot with CPU time results rather than node expansion results. While the results are similar, SCMA 4443 was slower than basic Short Circuit on a few more instances. SCMA 4443 expanded fewer nodes on the hardest instance but took longer to do so in terms of CPU time. This suggests, as expected, that there is an additional overhead to using additive abstractions with hierarchical search.

5.4.2 Glued 15 Puzzle

The next domain is the Glued 15 Puzzle described in section 3.3.3. Short Circuit used the same linear abstraction described in the previous evaluation. We evaluated SCMA on several different additive abstractions. Each was created by using the same method used for

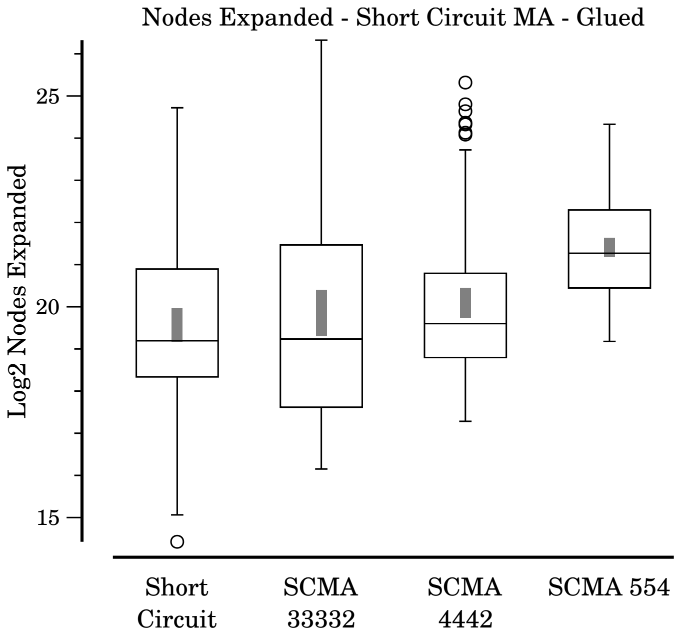


Figure 5-7: A box plot showing node expansion results for several abstraction hierarchies on 100 instances of the Glued 15 Puzzle.

the 15 Puzzle. Figure 5-7 shows the nodes expansion results for each abstraction. The y-axis represents the log_2 nodes expanded and the x-axis shows a box plot for each abstraction in the evaluation. Short Circuit using a single linear abstraction expanded the fewest number of nodes on average.

The left panel of Figure 5-8 shows the node expansion results for Short Circuit with a linear abstraction hierarchy and SCMA with the additive abstraction 33332 (that expanded the fewest nodes on average out of all additive abstractions). The y-axis represents the number of nodes expanded by SCMA 33332 and the x-axis represents the number of nodes expanded by Short Circuit with a linear abstraction. Approximately half of the 100 instances was solved with fewer nodes expansions than Short Circuit. These instances however were some of the easiest of the set. This result suggest that Short Circuit scales better than SCMA on this domain. The right panel of Figure 5-8 shows nearly identical results for CPU times. A few more instances were solved faster by Short Circuit even though it expanded more nodes. This indicates, as expected, that each node expansion takes slightly

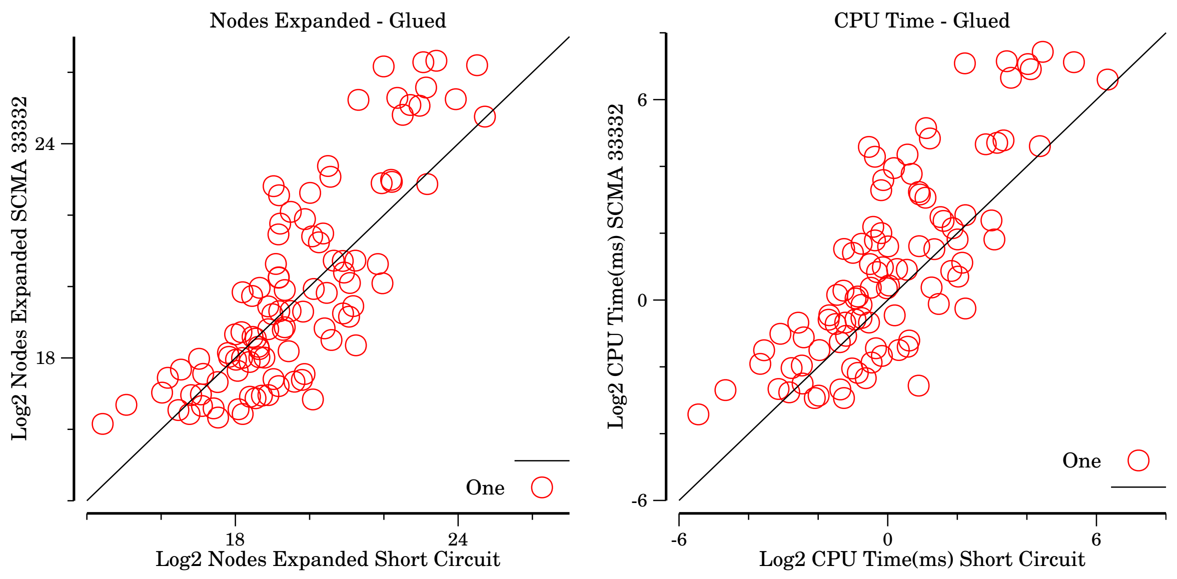


Figure 5-8: Scatter plot that shows node expansion and CPU time results for Short Circuit and SCMA 33332 on 100 instances of the Glued 15 Puzzle.

longer in SCMA.

5.4.3 Glued Two 15 Puzzle

The next domain is the Glued Two 15 Puzzle described in section 3.3.4. Short Circuit used the single linear abstraction hierarchy previously described. We evaluated SCMA on several different additive abstractions that were created in the same way as those created for the 15 Puzzle. Figure 5-9 shows results for each abstraction tested. The y-axis represents the log_2 nodes expanded and the x-axis show a box-plot for each abstraction. Short Circuit using the single linear abstraction expanded fewer nodes on average than any of the additive abstractions.

The left panel of Figure 5-10 shows the node expansion results for Short Circuit with a linear abstraction hierarchy and with the additive abstraction 4441 (that expanded the fewest nodes on average out of the all additive abstractions). The y-axis represents the number of nodes expanded by SCMA 4441 and the x-axis represents the number of nodes expanded by Short Circuit with a linear abstraction. SCMA 4441 expanded fewer nodes on

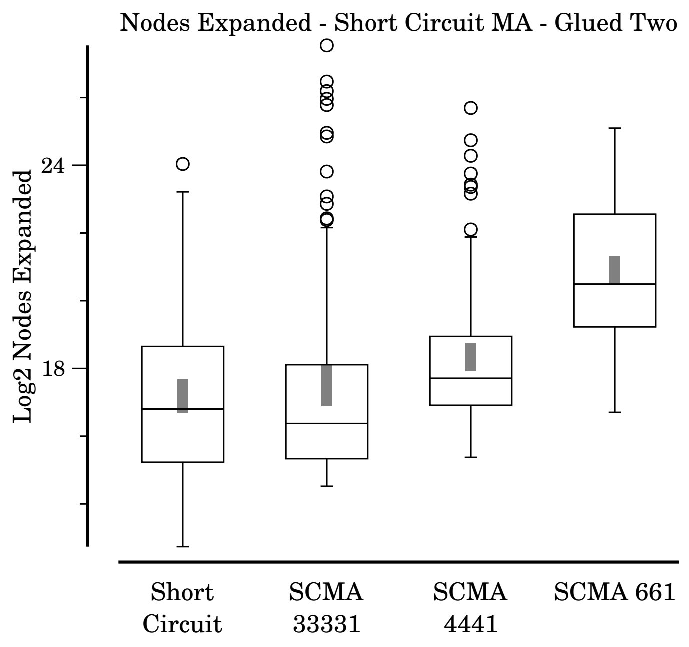
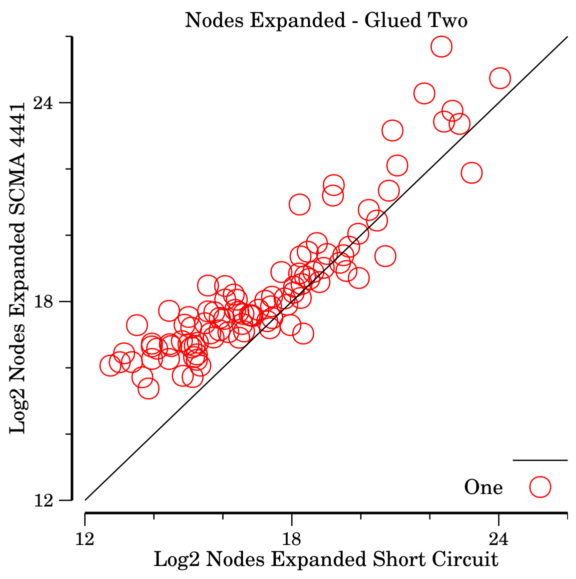


Figure 5-9: A box plot showing node expansion results for several abstraction hierarchies on 100 instances of the Glued Two 15 Puzzle.

a couple of instances but generally expanded more. The right panel of Figure 5-10 shows nearly identical results for CPU times. Again, the overhead of SCMA is not worth the more accurate heuristic.

5.4.4 17-4 Topspin Puzzle

The next domain is the 17-4 Topspin puzzle described in section 3.3.5. To generate the additive abstractions we split the puzzle into several disjoint pieces and modified the cost function used in the abstraction using the *Cost Splitting* technique described in section 5.1. We tested SCMA using several different abstractions and found that only a few could solve any of the 100 instances. SCMA 33333 was able to solve 4 instances and SCMA 44444 was able to solve 15. Short Circuit with a linear abstraction hierarchy was able to solve 72 instances for this domain. Yang et al. (2008) shown that pattern databases using the *Cost Splitting* technique can significantly reduce node expansion on this domain. However, the abstractions that they report are large and cause SKUA to run out of memory; too many



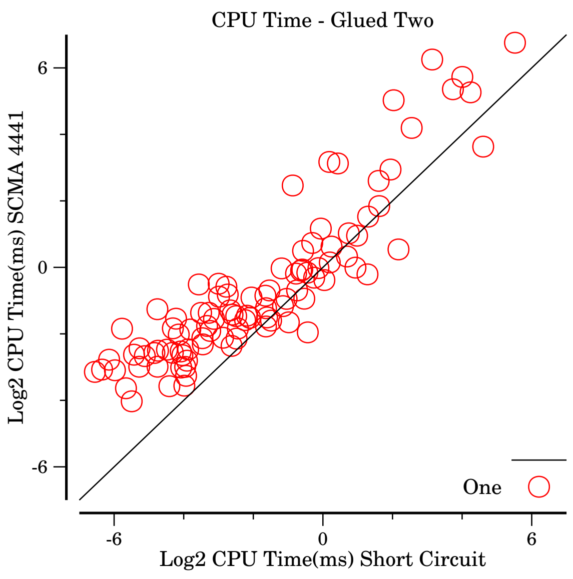


Figure 5-10: Scatter plot that shows node expansion and CPU time results for Short Circuit with a linear abstraction hierarchy and with an additive abstraction hierarchy on 100 instances of the Glued Two 15 Puzzle.

nodes are expanded in the abstraction hierarchy. Unfortunately, the overhead present in SAMOA causes it to fail when applied to this domain.

5.5 Suboptimal Search With Multiple Abstractions

Since using multiple additive abstractions has been shown to produce more powerful heuristics when used for optimal search with pattern databases, one might assume that these heuristics will produce more powerful heuristic guidance when weighted as well. Rather then using A* at the base of SCMA we have created a variant that uses WA*. This algorithm uses abstractions that are identical to the one used in SCMA and then weights the heuristics that they return. Since WA* is used, the algorithms solution cost is bounded by the weight used.

As we noticed in Section 4.1, weighting the heuristics returned by a optimal hierarchical search algorithm does not always result in the desired outcome. One possible explanation could be that the heuristic guides the search at the base level into a large local minima and it must examine most of the states in the minimum before proceeding. A simple example on the 15 puzzle using a non-additive 8 tile abstraction shows why this could be the case. The weighted search at the base level will quickly place all 8 tiles present in the abstraction into their goal pistons, then the search has no additional heuristic guidance to inform it where the other 9 tiles should be placed. Every move will result in an identical heuristic of 1 since all distinct tiles are in their goal position but the state is not the goal. By using additive abstraction we can account for the position of each and every tile in the original problem and should avoid large local minima.

We found that this approach can work well on some domains but the overhead required by SCMA causes it to perform worse on others. It is likely that using larger additive abstractions would indeed provide better guidance for suboptimal heuristic search algorithms. Searching these large abstractions with SCMA however causes increased node expansion in the abstraction hierarchy which reduces overall performance.

As shown in chapter 4, Switch can produce good quality solutions for only a fraction of the node expansion required by other algorithms. Switch however does not produce any guarantee on the cost of the solution that will be returned. We decided that we would also make a variant of SCMA that is more greedy by adding together the heuristics generated via non-additive abstractions. This algorithm is referred to as SCMA Non-Additive in the plots found in the evaluation section below.

5.6 Suboptimal Evaluation

Here we evaluate using WA* at the base of SCMA rather than A*. We also show the results of adding together non-additive abstractions. Both algorithms were implemented in C++ and compared against Switch on several domains. All algorithms in this evaluation were given unlimited running time and 47GB of RAM.

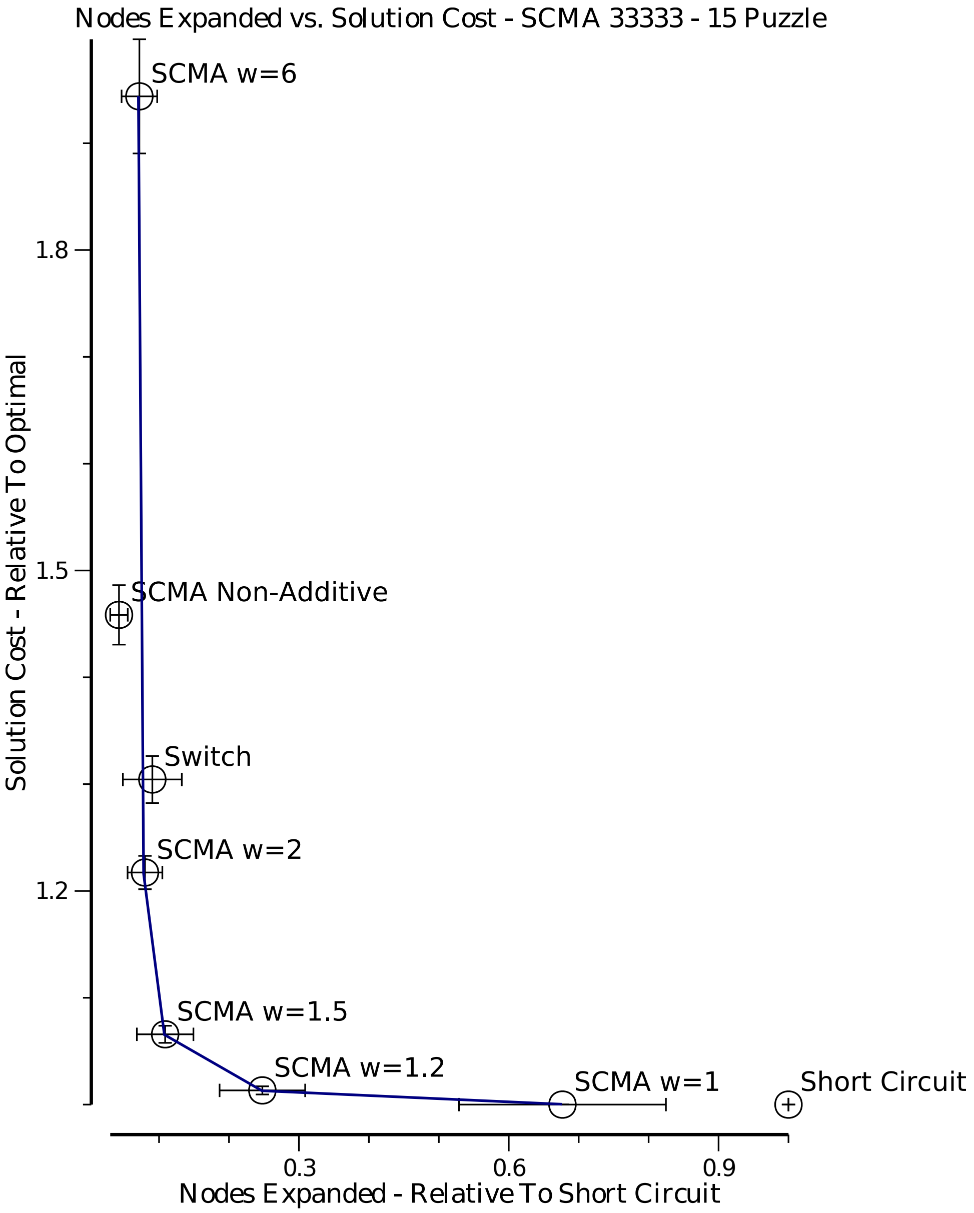


Figure 5-11: A plot showing the number of nodes expanded and solution cost for Switch, SCMA, and SCMA with non-additive abstractions on 100 instances of the 15 Puzzle.

5.6.1 15 Puzzle

The first domain is the standard 15 Puzzle described in section 3.3.1. Several different abstractions were evaluated with WA* and it was found that the 333333 abstraction expanded the fewest number of nodes when weighted. The same 100 instances used in previous evaluations were used and solved by all algorithms. Figure 5-11 shows the node expansion and solution cost results for this domain. The x-axis is represents the nodes expanded relative to Short Circuit. The y-axis is the solution cost relative to optimal. SCMA with a weight larger than 1.5 was able to expanded fewer nodes on average than the Switch algorithm, and with a weight of 2 produced solutions that were shorter. SCMA, driven by non-additive abstractions expanded fewer nodes than any of the SCMA weighted variants and Switch, though its solution cost was larger than Switch and SCOT* with a weight of 2 or less. If it was the case that weighted search using heuristics generated via hierarchical search performed poorly because of large local minima, then using multiple abstractions clearly solves the problem.

5.6.2 Glued 15 Puzzle

Next we evaluated our algorithms on the Glued 15 Puzzle variant described in section 3.3.3. Short Circuit used the same linear abstraction hierarchy described in the previous evaluation. SCMA WA* was evaluated with a number of different abstractions. The 33332 abstraction expanded the fewest nodes on average. Figure 5-12 shows the node expansion and solution cost results for the 100 instances of this domain. The x-axis represents the number of nodes expanded relative to Short Circuit. The y-axis is the solution cost relative to optimal. A circle is placed on the plot for each of the algorithms and weights evaluated. On this domain weighted SCMA performs well, expanding fewer nodes on average than Switch with weights larger than 1.5. SCMA 33332 with a weight of 1 expanded about twice as many nodes as Short Circuit. SCMA with non-additive abstraction solved all instances with fewer node expansions than any of the other algorithms. It also produced higher quality solutions than both Switch or weighted SCMA with weights larger than 2. Apparently, the 33332 abstraction provides extremely good guidance when weighted or when ignoring the additive constraint. This prevents the overhead in SCMA from decreasing overall performance and makes it a good choice for this domain.

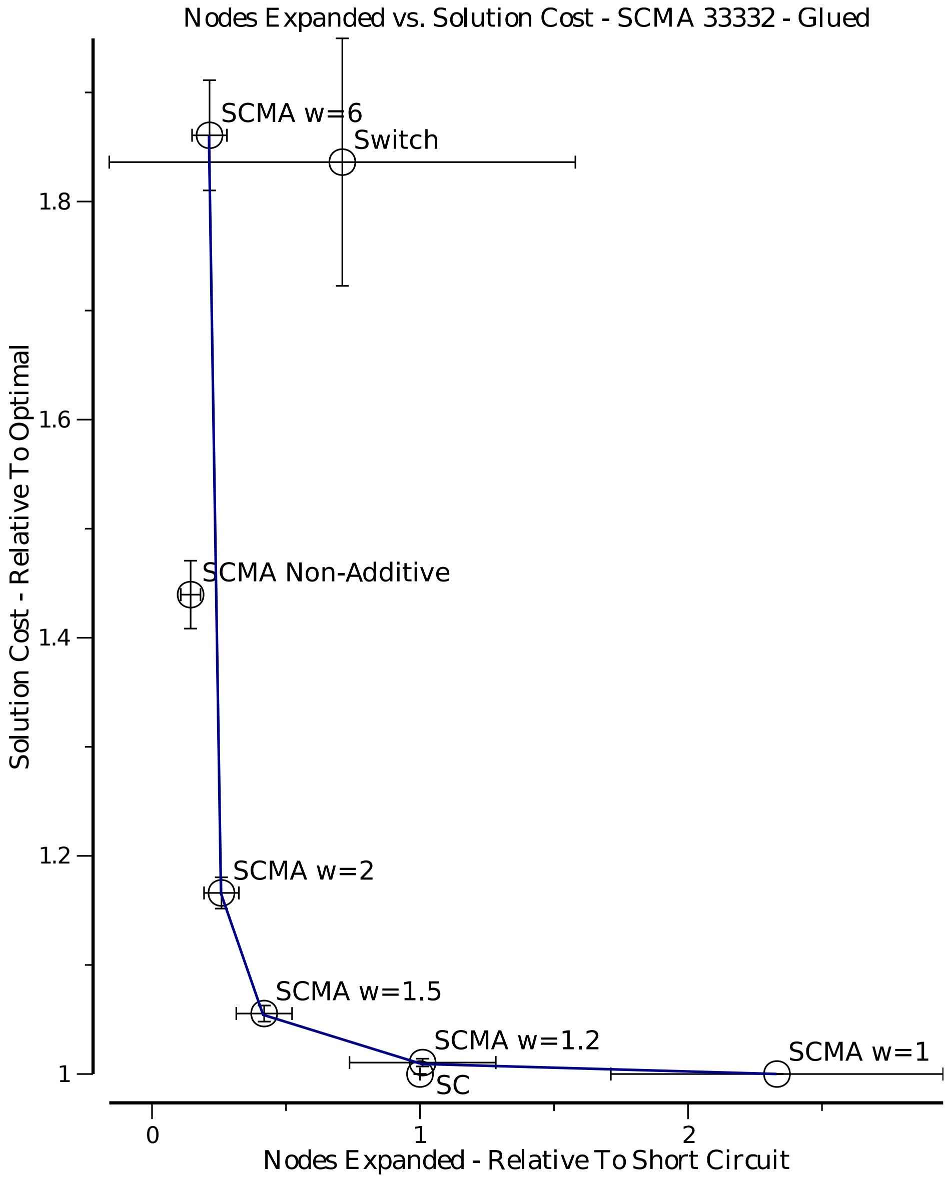


Figure 5-12: A plot showing the number of nodes expanded and solution cost for Switch, SCMA Non-Additive, and the weighted version of SCMA for 100 instances of the Glued 15 Puzzle.

5.6.3 Glued Two 15 Puzzle

The next domain is the Glued Two 15 Puzzle variant described in section 3.3.4. SCMA with WA* at the base was evaluated with a number of different abstractions; The 33331 abstraction expanded the fewest nodes on average. Figure 5-13 shows the node expansion

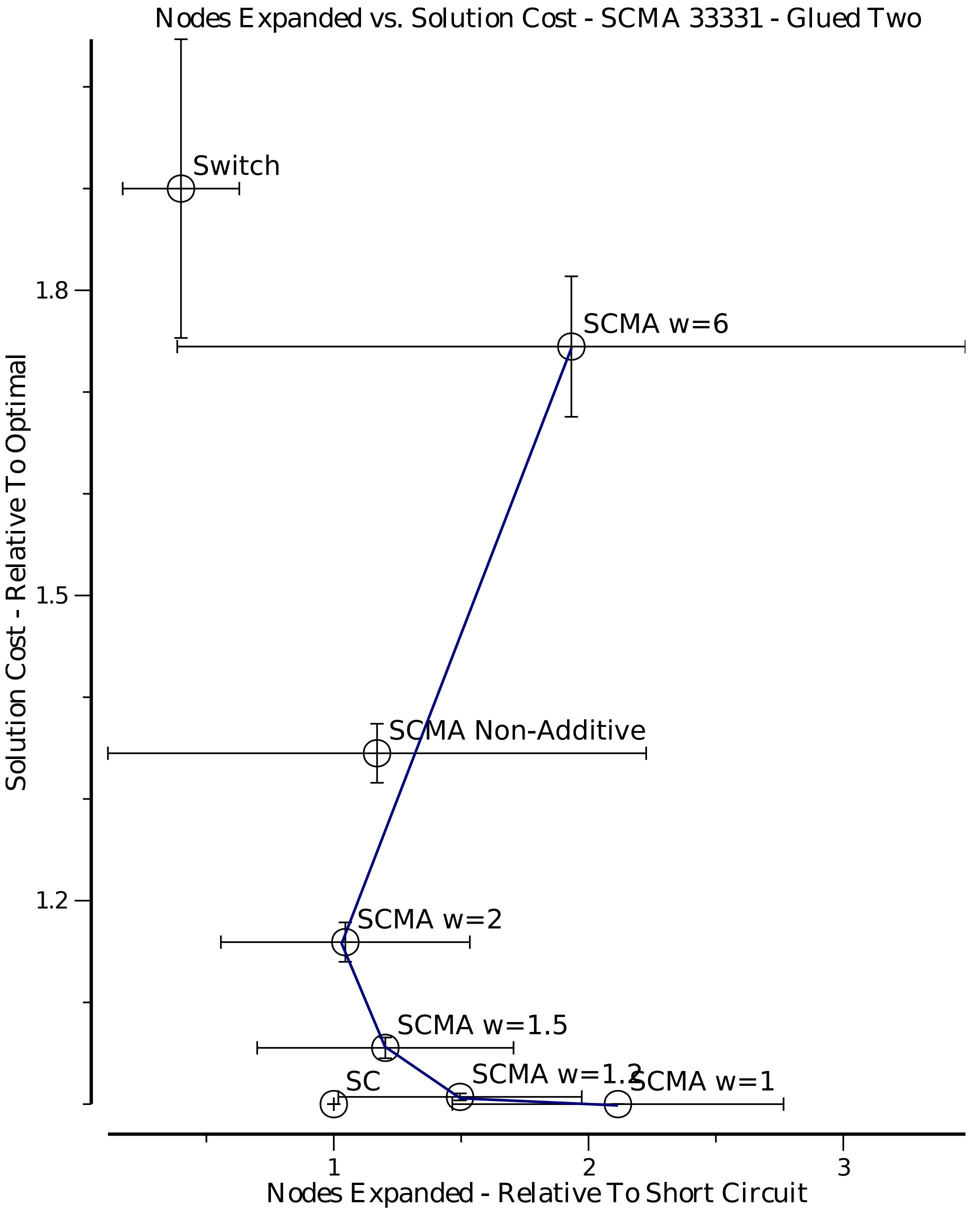


Figure 5-13: A plot that shows the number of nodes expanded and solution cost for Switch, SCMA Non-Additive, and the weighted version of SCMA on the Glued Two 15 Puzzle.

and solution cost results for the 100 instances of this domain. The x-axis represents the number of nodes expanded relative to Short Circuit. The y-axis is the solution cost relative to optimal. A circle is placed on the plot for each of the algorithms and weights evaluated. SCMA starts to decrease the number of nodes expanded as small weights are applied, how-

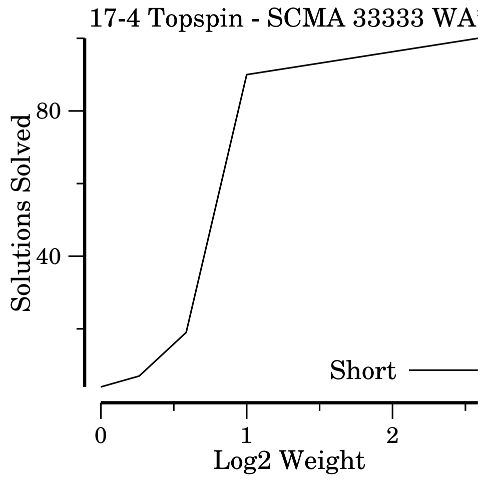


Figure 5-14: A plot that shows the number of solutions solved when weighting the base of SCMA 33333.

ever when a weight of 6 is used expansion is worse than optimal. This behavior is similar to the results seen when using a single linear abstraction. SCMA with non-additive abstractions expanded 1.2 times more nodes than optimal Short Circuit on average. Switch was the only algorithm that expanded fewer nodes on average than Short Circuit. Again, this result is likely because of the overhead present in SCMA. Each of the additive abstractions were to small to account for the important interactions in the problem and a larger set of abstractions could not be used without exhausting memory.

5.6.4 17-4 Topspin

The next domain is the 17-4 Topspin Puzzle. Switch used the same linear abstraction described in Section 3.3.5. Several abstractions were tested with SCMA; the 33333 additive abstractions described in section 5.4.4 expanded the fewest nodes on average. Not all weights used with SCMA were able to solve all instances of the problem. Figure 5-14 shows the number of instances solved by each weight. The y-axis is the number of solutions solved. The x-axis is the weight used. As the weight was increased more instances were solved, with a weight of 6 SCMA was able to solve all 100 instances. The non-additive version of

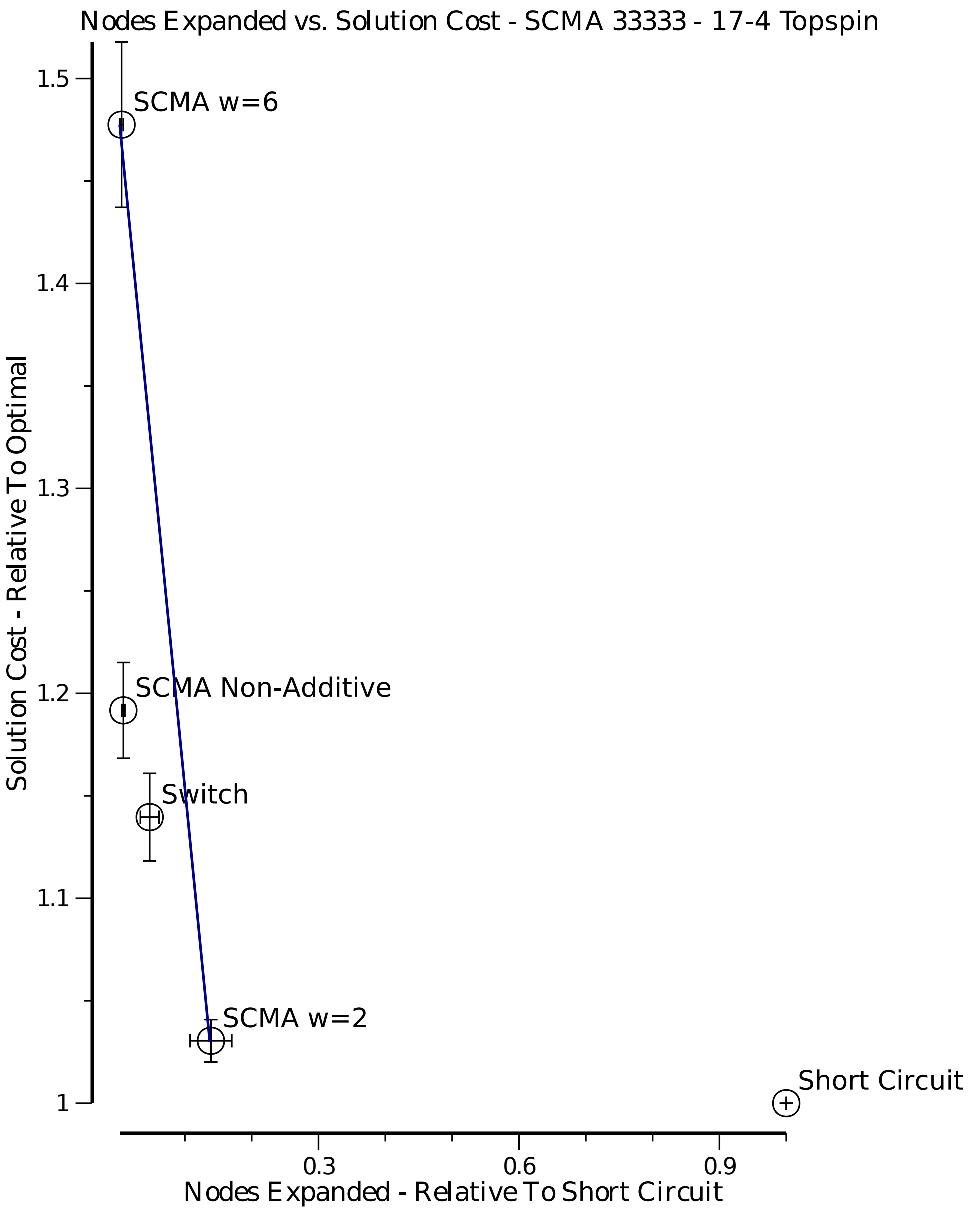


Figure 5-15: A plot showing the node expansions and solution cost results for Switch, SCMA WA*, and SCMA with non additive abstractions on 73 instances of the 17-4 Topspin Puzzle domain.

SCMA and Switch were also able to solve all of the instances as well.

Figure 5-15 shows node expansion and solution cost results for SCMA with its two best weights and the non-additive version as well as Switch on 73 instances of the Topspin

		Nodes Expanded (100 K)		CPU Time (sec)		
Domain	Algorithms	Mean	Std Dev	Mean	Std Dev	Sub Opt
15 Puzzle	SCMA w=2	1.5	0.4	0.25	0.1	1.2
	SCMA NA	0.88	0.27	0.14	0.1	1.45
	Switch	2.2	6.6	0.5	1.9	1.3
Glued	SCMA w=6	1.15	1.6	0.2	0.5	1.86
	SCMA NA	0.8	0.98	0.14	0.26	1.43
15 Puzzle	Switch	8	23	1.8	5.4	1.83
Glued Two	SCMA w=2	24.11	128	9.3	51	1.16
15 Puzzle	SCMA NA	17.4	46	7.16	47	1.34
	Switch	2.6	13	0.59	3.1	1.9
17-4 Topspin	SCMA w=6	0.31	0.36	3.53	4.1	1.5
	SCMA NA	0.7	0.8	7.2	8.3	1.19
	Switch	3.7	4.6	7.8	10.9	1.1

Table 5-1: A summary table that shows node expansion, CPU time, and sub-optimality for Switch, weighted SCMA, and SCMA with a non-additive abstraction.

puzzle. The x-axis is the solution cost relative to optimal. The y-axis is the number of nodes expanded relative to Short Circuit. The results are similar to that of Short Circuit with a single linear abstraction. SCMA was able to trade solution quality for CPU time by weighting the base. Switch however still produced higher quality solutions with less effort than weighted SCMA and it non-additive variant.

5.6.5 Summary of Results

Table 5-1 shows a summary of results for Switch, weighted SCMA, and SCMA with a non-additive abstraction (titled SCMA NA in the table). Column 2 displays the weighted variant of SCMA that was the closest to Switch in terms of solution cost. SCMA with a weight of 2 was the closest for the 15 Puzzle variants. A weight of 6 was the closest on the 17-4 Topspin Puzzle. Column 3 and 4 displays the mean and standard deviation of nodes expanded by each algorithm. SCMA using non-additive abstractions expanded the fewest nodes on the 15 Puzzle. Switch expanded the fewest on the Glued Two Puzzle, and SCMA with a weight of 6 expanded the fewest on the Topspin Puzzle. Column 5 and 6 display the mean and standard deviation of CPU time. The results are similar to the node expansion results. The last column titled Sub-Opt displays the average sub-optimality factor returned by each algorithm. Weighted SCMA returned the lowest cost solutions on the 15 Puzzle variants, while Switch returned the lowest on Topspin.

5.7 Conclusion

Our major contribution in this chapter was the creation of optimal and suboptimal version of Short Circuit capable of using multiple abstractions. On some domains our optimal variant was able to solve problems more quickly than Short Circuit using a single linear abstraction. On other domains, however, the overhead introduced by using multiple abstractions could not be overcome by the more accurate heuristic. On domains where *Cost Splitting* is required to create the additive abstractions, SCMA performed notably worse. Similar results can be seen for our suboptimal algorithm. On the 15 and Glued tiles puzzles the stronger heuristic value was able to produce better guidance allowing SCMA to perform well. On the remaining domains however, this was not the case. Using SCMA with the non-additive version of the abstractions showed promising results, but again failed to surpass the performance of Switch on some domains. These results further motivate the need for an algorithm that is specifically designed to perform suboptimal hierarchical search, such as Switch. Increasing the performance of suboptimal hierarchical search is not as simple as just using multiple additive abstractions and weighting the base of the search.

CHAPTER 6

Conclusion

Here we give a brief conclusion of the hierarchical search approaches discussed in this paper and some potential extensions of our work.

6.1 Future Work

The cost of solutions returned by suboptimal hierarchical search using multiple abstractions are surprisingly low compared to the bound presented by weighted A*. An algorithm known as Optimistic Search presented by (Thayer and Ruml, 2008) takes advantage of just this fact by running Weighted A* with a much larger weight and then proving the bound specified by the user. It is likely that using Optimistic Search at the base of a SCMA search would significantly improve search performance.

As described above, using multiple additive abstractions can produce more accurate heuristics but also can add significant overhead in terms of CPU time and memory usage. A common approach used to reduced the amount of memory required by additive pattern databases is to exploit the symmetry that is present in many domains (Felner et al., 2004). In certain domains such as the 15 Puzzle and Topspin, it is possible to simulate multiple additive abstractions with a single pattern database by applying a state transformation before looking up a heuristic value (Yang et al., 2008). This technique could be applied to SCMA by searching a single abstraction hierarchy and then performing multiple symmetrical lookups for heuristic values. This would likely reduce the amount of memory required to solve each instance. CPU time, however, could be increased since additional work is required to perform a transformation of the state before lookups can be completed.

$P + g$ 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.

In some cases, using large multiple additive abstractions can result in unneeded overhead because of the number of searches in each level and the memory required to hold the open and closed lists. Using smaller abstractions, however, may produce a heuristic that is too weak to solve the original problem. One possible solution may be to use the initial Manhattan distance or other estimates to dynamically decide the size of the abstraction that should be used to solve the problem. If the estimates were accurate, deciding the correct abstraction size may reduce the amount of overhead caused by large abstractions and still produce heuristics powerful enough to solve the original problem.

6.2 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. Finally, we have presented a hierarchical search algorithm that is able to use multiple additive abstractions in hierarchical search. While this approach does add additional overhead, the heuristics that it generates are powerful enough to increase optimal and suboptimal performance on some domains. 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) and CARP CST program (grant N10AP20029).

BIBLIOGRAPHY

Kenneth Anderson, Robert Holte, and Jonathan Schaeffer. Partial pattern databases. In *Symposium on Abstraction, Reformulation and Approximation*, pages 20–34, 2007.

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

Ariel Felner, Richard E. Korf, and Sarit Hanan. Additive pattern database heuristic. *JAIR*, 22:279–318, 2004.

Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, SSC-4(2):100–107, July 1968.

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

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

R. C. Holte, Perez B. Drummond, M. Zimmer, and MacDonald A. Searching With Abstractions: A Unifying Framework and New High-Performance Algorithm. In *Proceedings of the 10th Canadian Conference on Artificial Intelligence*, pages 263–270, 1994.

R.C. Holte, M.B. Perez, R.M. Zimmer, and A.J. MacDonald. The tradeoff between speed and optimality in hierarchical search. Technical Report TR95-19, University of Ottawa, Ottawa, Canada, 1999.

Robert C. Holte, Ariel Felner, Jack Newton, Ram Meshulam, and David Furcy. Maximizing over multiple pattern databases speeds up heuristic search. In *Artificial Intelligence*, volume 170, pages 1123–1136, 2006.

C.A Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68:243–302, 1994.

R.E. Korf and A. Felner. Disjoint pattern database heuristics. *Artificial Intelligence*, 134: 9–22, 2002.

Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

Bradford Larsen, Ethan Burns, Wheeler Ruml, and Robert C. Holte. Searching Without a Heuristic: Efficient Use of Abstraction. In *Proceedings of (AAAI-10)*, pages 114–120, 2010.

Michael J. Leighton. Faster optimal and suboptimal hierarchical search. In *Proceedings of (SoCS 2011)*, pages 92–99, 2011.

Marvin Minsky. *Steps Toward Artificial Intelligence*. McGraw-Hill, 1963.

Ira Pohl. Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, 1: 193–204, 1970.

E. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135, 1974.

Jordan T. Thayer and Wheeler Ruml. Faster than weighted A*: An optimistic approach to bounded suboptimal search. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling*, Fall 2008.

M Valtorta. A result on the computational complexity of heuristic estimates for the a* algorithm. *Information Sciences*, 34:48–59, 1984.

Fan Yang, Joseph Culberson, Robert Holte, Uzi Zahavi, and Ariel Felner. A general theory of additive state space abstractions. *JAIR*, 32:631–662, 2008.