ANYTIME SOLVING BY SOLUTION REFINEMENT

BY

Chris Sexton

BS in Computer Science, Indiana University, 2008

THESIS

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

Masters of Science

in

Computer Science

December, 2012

This thesis has been examined and approved.

Thesis director, Wheeler Ruml,
Associate Professor of Computer Science

Radim Bartoš,
Chair, Associate Professor of Computer Science

Philip J. Hatcher,
Professor of Computer Science

Michel Charpentier,
Associate Professor of Computer Science

Date

ACKNOWLEDGMENTS

Completing a thesis is impossible to achieve without support. I would like to thank Professor Wheeler Ruml, my thesis advisor, for his guidance and support through the process. His influence in instruction, group research, and personal research is invaluable. I would like to give a special thanks to Ethan Burns who provided the thoughtfully crafted search framework that I used in my code implementations. For their ideas and comradery, I would like to thank the entire UNH AI group, both past and present. Finally, I would like to thank my lovely wife, Madeleine, who has supported me and put up with my absence throughout my time as a graduate student.

TABLE OF CONTENTS

LIST OF FIGURES

Solving problems with large state spaces is a difficult task to undertake. For most problems that scale in size, there is a scale at which state of the art optimal solving fails. There are many suboptimal methods that provide solutions at the expense of solution length, but these also have an eventual point of failure.

This thesis investigates problems at a size at which even the most popular suboptimal solvers fail. Domains often have domain-specific constructive heuristics that quickly provide poor solutions, even for large problems. We present solution refinement over constructed solutions as a method of improving a poor quality but quickly obtained solution by searching in the solution path for shortcuts. These shortcuts are found incrementally, leading to improved solutions on problems where traditional suboptimal algorithms fail to find any solution.

Several improvements to solution refinement will be presented in order to improve the quality of solutions found and the speed at which they are obtained. Suboptimal search will be used to allow search to find shortcuts between nodes more distant. Next, incrementally changing the search distance will provide quality solutions faster than statically set distances. Finally, offset search will provide a more complete coverage of the search space in order to maximize the number of shortcuts found.

CHAPTER 1

Introduction

In the field of heuristic search, it is useful to explore domains which can be scaled in size and complexity. This property gives access to more difficult problems, pushing the state of the art in solving methods. For example, in the sliding tile domain, the size of the puzzle can be scaled to increase the complexity of solving the puzzle. Similarly, the towers of Hanoi domain offers scaling in the number of discs that need to be sorted in order to solve the puzzle. When scaled to even a modest size, these problems quickly become intractable for optimal search algorithms like A* (Hart et al., 1968). Suboptimal search algorithms such as weighted A* (Pohl, 1970) may be used beyond this point, but even these will fail on very large problems.

This thesis presents a method of solving large state space search problems for which traditional search methods have undesirable performance. When searching for solutions in large state spaces, the traditional approach for finding solutions is to craft heuristics which provide a high level of guidance for the problem at hand and search sub-optimally. By preferring the heuristic value, $h(n)$, instead of the underestimate of the solution length, $f(n) = g(n) + h(n)$, beam search (Bisiani, 1992) and weighted A* search can find solutions to problems where A* fails. IDA* (Korf et al., 2001) offers only linear memory usage, but exponential time complexity. Like A*, weighted A* has worst-case exponential memory complexity, but offers better search guidance, potentially expanding many fewer nodes on the path to a solution and therefore is able to solve more difficult problems. However, weighted A* only works up to a certain point due to its space complexity. Beyond this, beam search can be used, but it provides no guarantee on solution optimality, and can produce very long paths, but has a small memory footprint.

In order to solve problems where other search methods falter, one option is to use a domain-specific constructive solution. Using a hand-crafted solver, we can find solutions to problems in linear time, with some downsides. The first downside is that the domain used must be solvable in this manner. Problems which have recursive solutions are ideal in this situation. The second downside is that the length of the solutions acquired through these methods are unlikely to be short.

A similar method exists in planning with scheduling problems. In order to reduce the makespan of a plan, each action can be quickly planned to be taken sequentially. Once a satisfacting plan is obtained, actions which are independent may be rescheduled to run in parallel, shortening the makespan of the entire plan.

In this thesis, a method of improving solutions returned by deterministic solvers will be presented, exploring the space between hand-crafted solvers and heuristic search. The improvement is achieved by using the deterministic solution path as guidance for heuristic search, finding shortcuts between nodes. Refinement is a generic process for solving a class of problems in which solutions can be obtained and heuristic guidance between any two nodes can be computed.

This thesis makes two conclusions. First, a refinement method will be shown be shown to reduce the length of constructive solutions in the tiles puzzle by up to 40% on small 15-puzzles puzzles and 10% on large 80-puzzles. In the blocks world domain, similar though not as extensive shortcuts will be found. Second, these refinement solvers will produce solutions orders of magnitude shorter than those produced by beam search and will always return a path, even for very difficult problems. Where weighted A* and beam search fail, solution refinement will produce anytime results on every instance. This work suggests an avenue for future work in augmenting or complementing constructive heuristics, using instead of trying to replace them with general-purpose heuristic search.

1.1 Domains

We will primarily examine two domains in this thesis, the N-tile puzzle and blocks world. These will be detailed in order to understand the reasons why they are complex problems.

1.1.1 Sliding Tiles

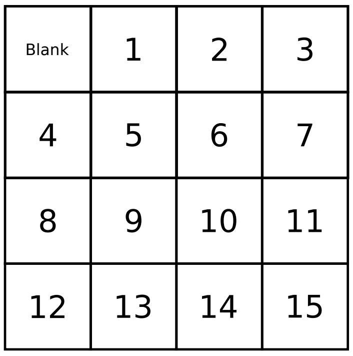

Figure 1-1: A 15 tile puzzle in its goal configuration

The sliding tile puzzle is very well known in the field of heuristic search. A set of N tiles are laid out in a square grid n tiles wide and n tiles tall, where $n = \sqrt{N+1}$. There is one blank space in the puzzle used to move tiles. Any tile adjacent to the blank can be swapped with the blank, moving both the blank and the tile. A simple goal configuration is presented in Figure 1-1.

The sliding tile puzzle presents a very large state space even at small puzzle sizes. The puzzle can be scaled up simply by adding rows and columns. Each puzzle has N tiles and increases the complexity of the state space. It has been shown that solving the N puzzle optimally is NP-Hard (Ratner and Warmuth, 1986).

The first optimal solver for the 24 puzzle used iterative-deepening-A* (IDA*) (Korf and Taylor, 1996) and generated between 8 billion and 8 trillion nodes before finding the solution. The state space of an nxn puzzle is $(n^2)!/2$ (Korf and Taylor, 1996; Sadikov and Bratko, 2007). For a 15 puzzle, this is 1.0×10^{13} reachable states, for the 80 puzzle, 2.8×10^{120}. While a major point of heuristic search is to avoid searching every node in a

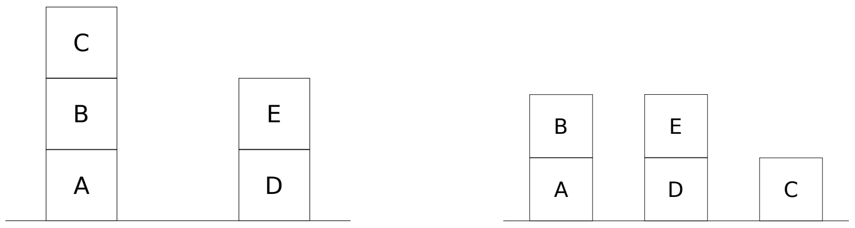

Figure 1-2: Sample blocks world states

state space, we can still see the increase in complexity as the puzzle size is increased.

Several heuristics are available for the tiles domain. Manhattan distance is an additive measure of how far each tile is from its goal position. This heuristic is a mainstay for sliding tile solving because it is easy to implement and is sufficient for guiding search to solve many puzzles. An upgrade to Manhattan distance is linear conflict (Hansson et al., 1992). If two tiles are in conflict with one another, then they must be in reverse order of their goal configuration and therefore to solve the puzzle, must move past each other. The interaction between tiles means that the linear conflict heuristic can add the number of moves it takes to swap tile positions in a row or column to the Manhattan distance and maintain admissibility.

Pattern databases (PDBs) provide a precomputed abstract heuristic for a part of a puzzle. The heuristic solves a simplified version of the puzzle where some elements are ignored and considered free to move. In the tiles puzzle, this means that a number of tiles will be erased and the blank is still considered. If two PDBs are disjoint, then no tile appears in both PDBs. This partitioning allows the PDB values for a state to be added together maintaining admissibility (Felner et al., 2004). The 6-6-6-6 partitioning pattern database was the first heuristic to solve the 24 puzzle optimally (Korf and Felner, 2002).

1.1.2 Blocks World

Blocks world is a well studied, famous planning domain. The domain consists of a finite number of blocks and a table on which they sit. A start and goal state is given by the configuration of the blocks sitting in towers on the table. Any block on the top of a tower

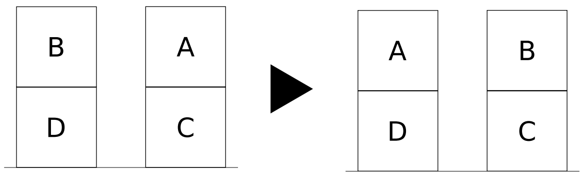

Figure 1-3: A start and goal state with blocks A & B deadlocked

may be moved to the top of any other tower or onto the table.

An optimal blocks world plan consists of the fewest number of moves necessary to transform the start state into the goal state. Figure 1-2 shows two blocks world states. The left state can be thought of as an initial state, and the right state is the outcome of moving block C to the table. Optimal blocks world planning is an NP-hard problem (Chenoweth, 1991).

Blocks world heuristics include counting the number of blocks that must be moved and PDBs. The sum of the number of blocks that must be moved can be improved by finding which blocks are deadlocked. That is, which blocks must be moved more than once in order to be put into a goal position. The left side of Figure 1-3 shows a situation where either block A or block B must me moved twice in order to achieve the goal position on the right side of the figure. If A is chosen to move twice, then A must be moved to the table, B moved on top of C, and then A would be moved on to D.

1.2 Outline

In Chapter 2, we will investigate related work including suboptimal search methods used to find solutions on difficult search domains, domain-specific constructive solvers used to quickly find suboptimal solutions, and methods of improving the paths returned by search algorithms. Chapter 3 will discuss refinement using local search over a solution path and investigate how this method behaves when used on problems that can scale in difficulty. Chapter 4 will introduce two extensions to this method and evaluate their effectiveness. Finally, Chapter 5 will discuss drawbacks, future work, and conclusions of the effectiveness of our method.

CHAPTER 2

Previous Work

In order to understand the reasons why solution refinement over consructively created solutions is a useful method of solving state space search problems, we will discuss several methods of solving large problems suboptimally, present domain-specific solvers for sliding tiles and blocks world, and introduce two previous methods of solution refinement.

2.1 Suboptimal Search Methods

Large state space search problems are often solved by using suboptimal search. Suboptimal algorithms may be bounded or unbounded. Bounded suboptimal search algorithms guarantee that the cost of their solutions are within some factor of the optimal solution. Unbounded suboptimal search makes no such guarantee. The most standard and simple to implement bounded suboptimal search algorithm is weighted A*, but many other options exist. For instances so big that memory becomes an issue, beam search and its variants are used to provide solutions where bounded suboptimal search fails. These search algorithms include beam search, BULB, and beam stack search. All three beam search variants prune unpromising search nodes in favor of nodes with low $h(n)$ value.

2.2 Weighted A*

Weighted A* search is a bounded suboptimal variant of A* search. Like A*, nodes are stored on a priority queue and expanded in a best first order according to the function, $f(n)$. Using the function $f(n) = g(n) + w * h(n)$ where $w \geq 1$, weighted A* focuses on the heuristic value instead of the $g(n)$ cost to reach the current node. This focus adds greed and biases the search towards nodes that appear to be closer to the goal. However, the heuristic is not guaranteed to be an accurate cost to the goal, so weighted A* can be mislead by local minima in the heuristic function. In practice, weighted A* expands fewer nodes along the path to the goal thus making it faster than A*. This comes at the expense of larger solution costs due to inadmissible choices during search.

Weighted A* is guaranteed to find a solution with cost within a factor of w of the optimal solution. This guarantee gives an upper bound on the solution cost, but solutions may be much closer to optimal than the given bound. Therefore, using weighted A* to find solutions quickly can lead to reasonably low cost solutions.

While weighted A* may not be able to solve large problems, we will show that it can be useful for search within suboptimal solutions of large problems.

2.3 Beam Search

Beam search is a search strategy used for solving in domains where bounded suboptimal search fails due to memory constraints. When attempting to solve large problems, beam search may be one of the few algorithms capable of producing a solution and is therefore useful as a benchmark. Beam search expands nodes in a breadth first search order and keeps only a limited set of nodes. These nodes are put into a priority queue ordered on their f value. At each depth layer, only a fixed number of nodes are kept. Search continues by expanding the next node on the OPEN list until a goal is found. Memory is saved by expanding only promising nodes at each depth layer.

Beam search relies on a width parameter, B. This parameter influences the amount of memory used during search and how many nodes the search can consider. Using a small B, beam search can be used to solve instances larger than that which weighted A* can handle. For example, Figure 2-1 shows the behaviors of beam search and weighted A* search across several sizes of tiles puzzles. This plot shows a $\log_{10}$ value of solution length versus the value of N for each N-Tile puzzle size tested. A line for each algorithm shows the average solution length for each increasing puzzle size. The implementations of greedy search and

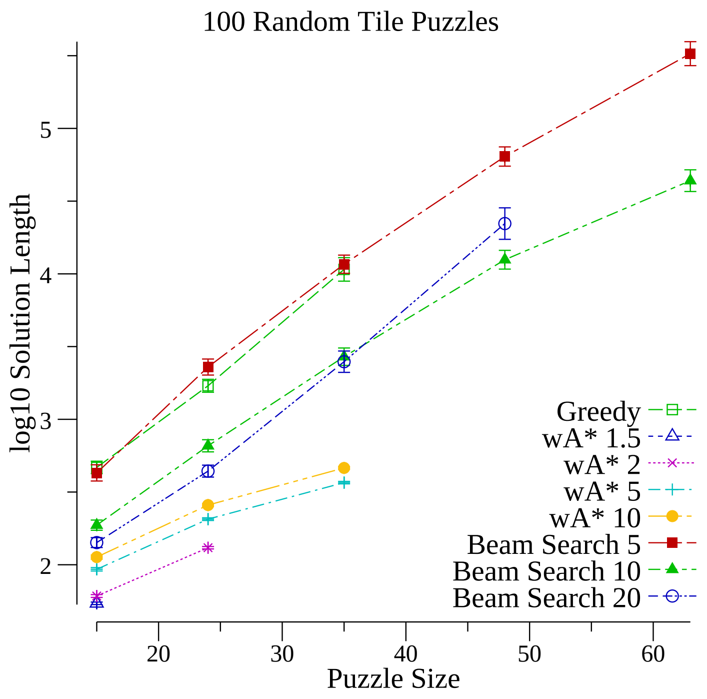

Figure 2-1: Beam and weighted A* search with varying beam widths and weights respectively.

weighted A* were given by Burns et al. (2012), and beam search was implemented in the same code base. While weighted A* finds good solutions on small puzzles, beam search is able to solve much larger puzzles, up to the 63 tile puzzle. Using a large B can result in better solutions since more nodes can be considered, but may introduce memory or time issues on hard instances.

When considering the memory usage of beam search, it is important to consider the closed list. Using a closed list eliminates duplicates in the search space and this has the effect of preventing search from re-expanding any nodes that would result in cycles. For any experiment performed in this thesis, we assume cycles are a problem and use a closed list. Fortunately, the closed list only needs to store a single copy of each search state, which in many cases is very small. In the tiles domain, using a closed list results in a dramatic difference in nodes expanded as compared to no closed list. In many cases, omitting the closed list results in no solution.

If a node is pruned which was the only path to the goal, then beam search may never find a solution. In this situation, beam search will exhaust the search space in other directions until no nodes are left on the open list. With duplicate detection, search will not reopen the pruned nodes that lead to the goal. In domains where dead ends exist, search can be led towards a dead end and not be able to exit. As such, beam search is not guaranteed to find a solution to any given problem. This unpredictability is a downside to the algorithm, but in practice it may not cause a problem in some domains. In the tiles domain, beam search is able to solve many problems that are impossible to solve with search without pruning.

Best First Beam Search

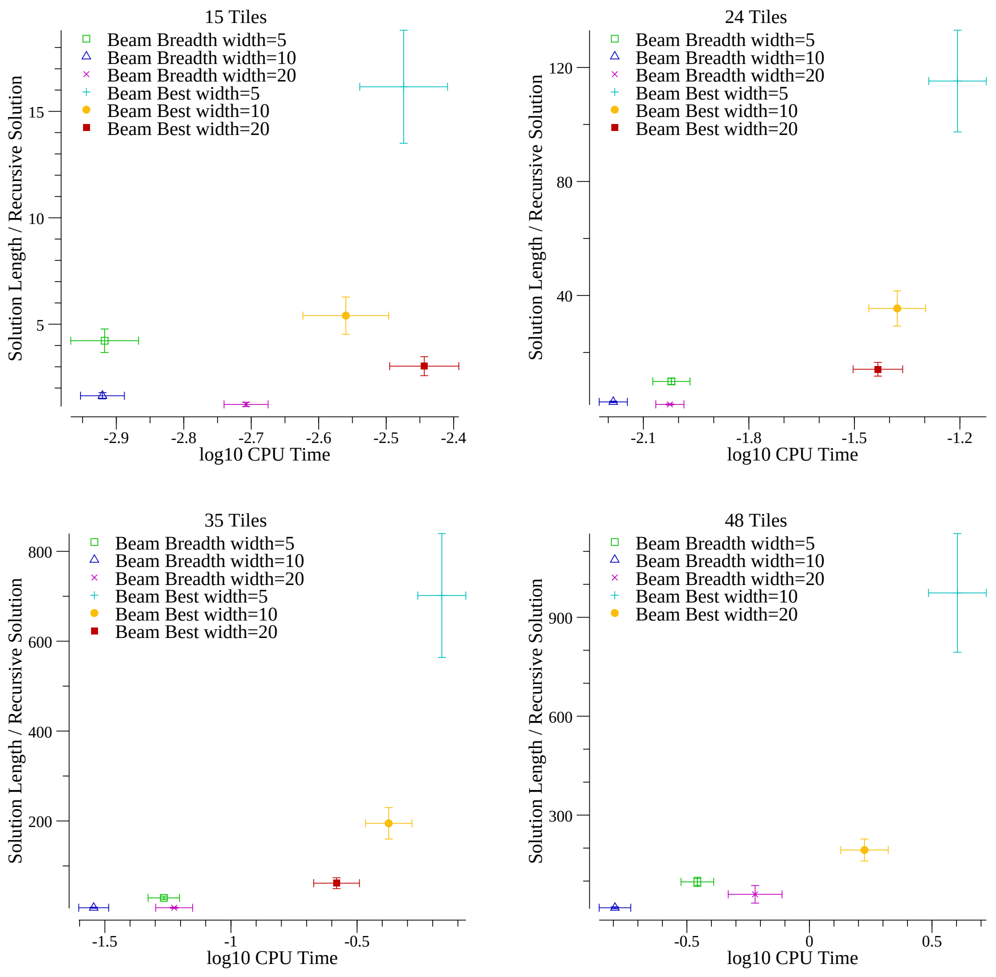

Figure 2-2: Best first vs breadth first beam search

Another variation of beam search is a best first beam search. This search behaves much closer to A* search with a limited open list than the breadth first variety. Nodes are expanded in a best-first order and any nodes which are unpromising are pruned from the open list until the open list is the size of the beam width, B.

While best first beam search can provide solutions to the same problem sets that breadth-first beam search solves, it uses more time to do so and returns longer solutions. Figure 2-2 shows breadth first dominating best first beam search over 100 randomly generated tile instances for the 15, 24, 35, and 48-tile puzzles. For every domain size, best first beam search produces poorer quality solutions and takes much more time to do so. On the 48-puzzle, it fails to return a solution with a beam width of 5 on 57 of the 100 instances.

2.3.1 BULB

Limited discrepancy beam search (Furcy and Koenig, 2005), or BULB, is an algorithm extending beam search to provide limited discrepancy backtracking, preventing pruned nodes in the tree from being lost.

The basic premise for BULB is that it behaves just as beam search does until it reaches a maximum depth and runs out of memory. At this point, BULB employs a backtracking strategy similar to that of limited discrepancy search. This allows BULB to generate solutions with much higher quality and guarantees that a solution will be found if one exists.

While beam search expands each layer of the search tree, BULB expands slices of nodes, B in size for each layer. These slices can be uniquely identified since nodes on each layer can be uniquely ordered. When search fails, backtracking occurs and the next slice of the layer in which the discrepancy is taken is expanded. BULB requires a maximum search depth parameter to trigger its backtracking functionality. This can be implemented with as a memory limit or a limit on the size of the hash table which keeps track of nodes.

A comparison of greedy search strategies by Wilt et al. (2010) shows that BULB can be used to solve 48-tile puzzles, but the sub-optimality of the solutions is unbounded and potentially very large. This study showed that other potential algorithms such as weighted

A*, $A*_\epsilon$ and LSS-LRTA* are all either unable to find solutions or are dwarfed in solution quality as compared to BULB on the 48 puzzle.

2.3.2 Beam Stack Search

Another approach to backtracking in beam search is beam stack search (Zhou and Hansen, 2005). Beam stack search introduces a data structure called a beam stack. The beam stack contains an entry for each depth layer and contains information about which nodes have been expanded at each given layer based on their ordering, similar to BULB's slices.

While traversing the search tree, beam stack search sets depth bounds based on found solutions. This anytime functionality limits the depth at which the algorithm must look to find solutions and guarantees that beam stack search will find an optimal solution if given enough time.

2.4 Constructive Solvers

For many domains, it is not unreasonable to write hand-crafted, domain-specific solvers which provide suboptimal solutions very quickly. These solvers may take advantage of a recursive property of the domain or take advantage of other properties of the domain. With a solver able to produce a valid solution to any size problem, it is possible to leverage the solution as domain knowledge and guide local search using the sub-optimal solutions given by the constructive solver.

We will briefly describe the constructive solver for each domain used in this thesis.

2.4.1 Recursive Decomposition of Sliding Tiles Puzzles

Parberry (1995) presents an algorithm that models the methods a human might use to solve sliding tile puzzles. To generate a solution, the algorithm exploits the recursive nature of the tiles puzzle. The solver operates by recursively solving only the outer L of the problem as seen in Figure 2-3. This outer L is defined by the row and column farthest from the blank space in the goal. Once in place, tiles in the outer L do not need to be moved in order

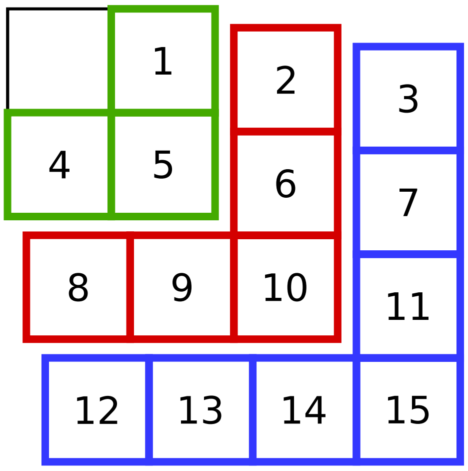

Figure 2-3: The decomposition of the 15 puzzle.

to solve any positions in the inner space left to solve. The solver can then focus inward to the next unsolved outermost L of tiles until the unsolved section of the puzzle is reduced to a 4x4 puzzle. This final step can be solved using a small database lookup. By doing this, it is possible to quickly solve 20x20 (399 tile) puzzles. The solutions found by this algorithm are guaranteed to be no more than 5 times optimal in worst case configurations. The solver is linear time in the length of the solution.

In Sadikov and Bratko (2007), this recursive solver is presented as a pessimistic heuristic. Pessimistic heuristics are heuristics guaranteed to never underestimate the cost to the goal, and these heuristics are shown to be useful in real-time search. Since the suboptimal solution to the entire puzzle is given by the domain-specific solver, this is never an underestimate on the cost-to-go. The domain-specific solver is fast enough to be used as a heuristic in RTA* search.

One benefit of this recursive approach to the tiles puzzle is that the solutions produced are very similar in cost. Figure 2-4 shows the behavior of the recursive decomposition solver for several puzzle sizes. In each instance, the decomposition produces solutions that are much shorter than beam and greedy search and have very little variance. In addition, this method of solving produces solutions very quickly.

As previously stated, the work presented in this thesis will be improving upon the

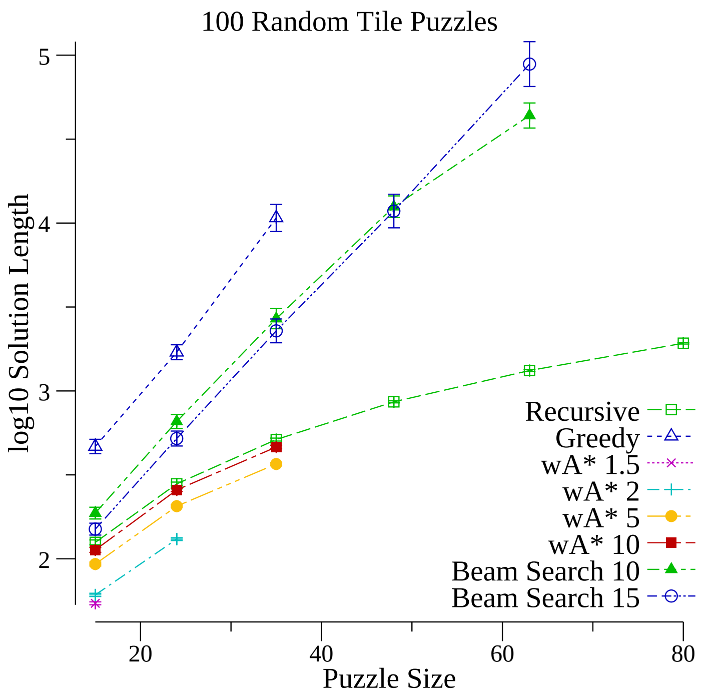

Figure 2-4: A constructive solver for tiles

domain-specific solver. This means that not only are the solutions to the domain-specific solver orders of magnitude better than beam search, but we will find even better solutions than these domain-specific solvers. Beam search is considered a useful tool when solving these big puzzles, but it simply can not compete with the domain-specific solver in terms of solution length and execution time. Not only does beam search produce poor results as the problem gets harder, but it also runs into limitations when attempting to solve the largest problems.

2.4.2 Blocks World

Blocks world benefits from a very simple constructive algorithm known as Unstack, Stack (US), introduced in Gupta et al. (1991). There are also two improvements on US known as GN1 and GN2, named after their original authors, Gupta and Nau (Gupta and Nau, 1992).

In the US algorithm, each misplaced block is picked up from the tower of blocks it rests upon and put onto the table. When all blocks are on the table, a stacking phase puts the blocks in their goal positions. Since each misplaced block must be moved at least once in an optimal solution, this algorithm produces solutions that cost at most twice that of the optimal solution.

The US algorithm can be improved by preferring constructive moves to unstacking. The GN1 algorithm eliminates the two phase system by always constructing goal towers if possible and only moving a block to the table if no constructive move can be made. This is guaranteed to provide solutions at least as short as the US algorithm given that it shortcuts the unstacking only when constructive moves can be made instead. During the stack phase of US, constructive moves are available until the algorithm finishes.

The third algorithm for blocks world relies on deadlocks. A deadlock is a situation which requires a particular block to be moved twice in order to break a loop, as previously seen in Figure 1-3. GN2 aims to handle these deadlocks by moving deadlocked blocks to the table before non- deadlocked blocks. We will focus on the US solver to illustrate how suboptimal solutions for blocks world problems can be improved.

Slaney and Thiébaux (2001) presents these algorithms and an optimal blocks world solver. The US, GN1, and GN2 algorithms are shown to have linear time complexity. While they present an optimal domain-specific solver, the authors argue that blocks world is an effective benchmark when using instances with a reasonable distribution of hardness.

2.5 Joint and LPA*

Ratner and Pohl (1986) present a method of combining the speed of constructive algorithms and the power of heuristic search. Local Path A* (LPA*) is an algorithm that takes a solution provided by a sub-optimal algorithm and executes local A* searches within that solution path. Each local search between nodes on the solution path returns an optimal path between the nodes, short-cutting any missteps or cycles in the original solution path.

The algorithm selects beginning and ending nodes based on a parameter, d_{max}. Local search proceeds to replace the original path with optimal solutions to each sub-path along the original solution path. Even if the path returned by local search is the same length as the segment in the global solution path, the segment is replaced to provide future search new nodes to search between. The point of this behavior is to improve the chance that new paths can be found shortening the global path.

Each local search uses A* to find a path between local start nodes and local goal nodes. This guarantees that any local solution is no longer than the original path between those two nodes. Globally, the path found by LPA* is no longer than the path produced by the constructive algorithm.

The Joint algorithm defines the nodes which act as beginning and ending nodes of each of the local searches as joint nodes. These nodes are the remanents of the original global solution since the start and goal states of local search do not change.

After performing LPA* on the global path, new local search problems are formulated around these original path nodes of the global solution path in order to find shortcuts around the joint nodes. Since LPA* replaces all nodes between the joints of the global solution with optimal sub-paths, the joint nodes are potentially the only nodes left from the original solution. This makes them interesting targets for further improvement.

LPA* forms a basis for the research done in this thesis. We will expand LPA* by using suboptimal search and changing its start and goal node selections. In addition, we will be evaluating LPA* when used with domain- specific solvers as input paths. These changes will allow LPA* to solve difficult problems with good anytime performance.

2.6 Iterative Tunneling Search with A*

Furcy (2006) introduces an algorithm that performs local search on the path given by a suboptimal search algorithm such as BULB. ITSA* builds a neighborhood out of the given solution path by incrementally adding all nodes that can be reached within one step of the neighborhood. This neighborhood is expanded in a breadth-first fashion until memory is filled. With a neighborhood graph in memory, A* is used to search the neighborhood for a shorter path from the start node to the goal.

Figure 2-5 shows an example of a neighborhood expansion for ITSA* between a start and goal node shown in green. The original path is shown in red. This path nodes surrounding

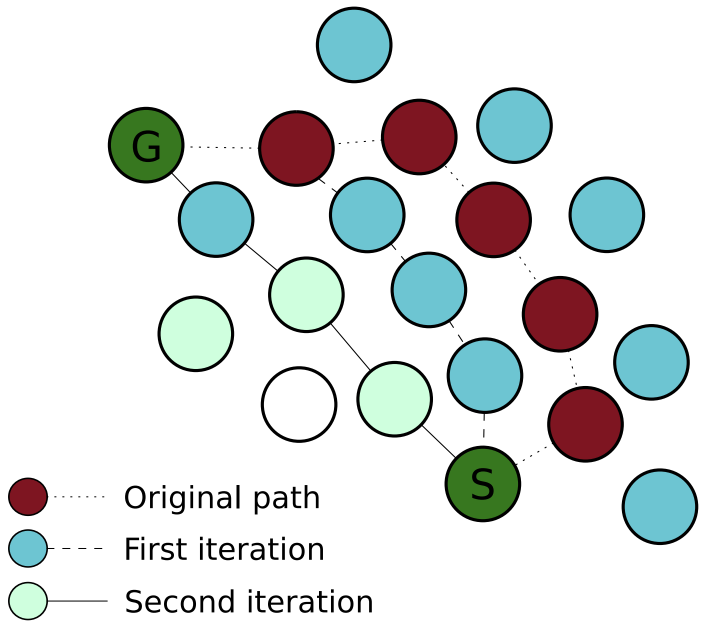

Figure 2-5: ITSA* Expansion

the original path are expanded to find a shorter path in cyan. On a second iteration, the optimal path is found in mint green. The lowest cost path for each iteration is shown by the node connections. We can see that the first iteration includes many nodes not on the optimal path, and one node that is on the final optimal path. On the second iteration, the optimal path is expanded. The paths are selected among all neighborhoods previously expanded.

In Anytime BULB search, solutions found by BULB are used as search depth cutoffs. One variant use of ABULB is to use ITSA* to improve the solution of each solution before a restart. While LPA* is bounded by distance, ITSA* is bounded by its memory usage, and attempts to shorten an entire solution rather than a path within the solution.

CHAPTER 3

Anytime Solution Refinement

Solution refinement considers a class of algorithms that perform local search guided by the solution given by a suboptimal solver. This is particularly interesting in situations where it is possible to obtain a fast domain-specific solver for a domain which variants of bounded suboptimal search are unable to solve.

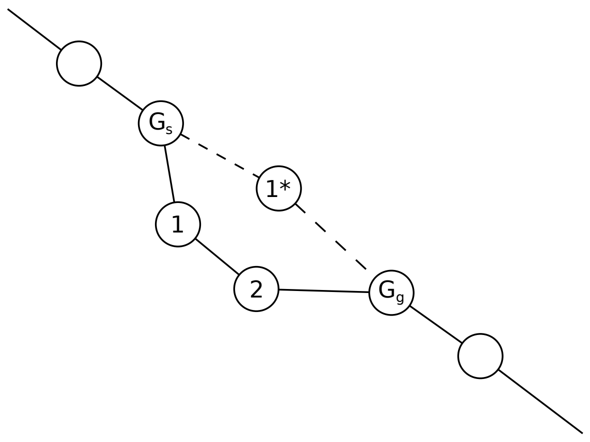

Figure 3-1: A shortcut in the subpath from G_s to G_g shown by the dotted lines and the node, $1*$

Solutions returned by a constructive solver are represented as a path G and are considered a global solution path. A local solution path from the global start node, G_s, to a local goal node, G_g is considered. This forms a local search window increment, *incr*, of length $g(G_g) - g(G_s)$ equivalent to d_{max} in LPA*. The solution returned by local search is then stitched into the global solution. Once all nodes on the solution path are considered, the solution returned by refinement will be used as a new global solution. This solution may then be considered as incumbents for further refinement in an anytime approach.

The goal of refinement is to find shortcuts in a suboptimal solution path. Shortcuts eliminate poor choices and cycles that may be present in a poor quality solution.

To find shortcuts, we first obtain an incumbent solution using a suboptimal solver. The solution is then analyzed to find shortcuts by formulating new, smaller searches based on pairs of nodes in the incumbent solution path, illustrated by Figure 3-1. Each smaller search problem can be bounded by the distance, d, in the path between the two nodes selected in the incumbent solution. Thus, if two nodes are d steps away in the incumbent solution, we know that no search beyond depth d is needed to find either a shortcut or the original path. A search using Dijkstra's algorithm would provide the shortest path between any two nodes near each other in $O(|V|^2)$ time where V is the number of verticies in the graph. Using heuristic search, we can extend the reach of these intermediate searches to find shortcuts between nodes farther apart.

```
1. Refinement($problem$, $incr$):
2.  $soln$ = Solver($problem$)
3.  $length$ = Size($soln$)
4.  while true
5.      $soln$ = PerformRefinement($soln$, $incr$)
6.      if size($soln$) == $length$
7.          break
8.      else
9.          $length$ = Size($soln$)
10.         Output($soln$)

11 PerformRefinement($path$, $incr$):
12. for $i$,$j$ = 0, $incr$; $j \leq$Size($path$); $i$++,$j$++
13.     $subpath$ = $path[i..j]$
14.     $result$ = Astar($subpath$)
15.     $path[i..j]$ = $result$
16. return $path$
```

Figure 3-2: Pseudocode for solution refinement

The pseudocode for anytime solution refinement is given in Figure 3-2. The algorithm begins on line 2 by finding a solution to the problem using a domain-specific solver. Lines

12-16 handle searching the path incrementally from a global start node to a global goal node. In lines 5-10, the solution is continually improved and output. If the solution length does not change between any two iterations, then refinement stops.

Now that we have introduced the simplest form of this algorithm, we will consider the implications of choosing a new search strategy and how the size of *incr* affects the solutions given by refinement. Finally, we will see the results of using refinement on tiles and blocks world.

3.1 Local Search Strategy

While the algorithm providing an initial solution is important to overall running time, the search strategy chosen to find shortcuts within the global path is crucial for refinement to produce an improved path. Using an optimal search algorithm guarantees that paths between selected nodes are optimal, but does not guarantee global optimality over the solution. However, this does guarantee that if there is any suboptimal choice made between two nodes in the incumbent solution, refinement will produce a shorter solution.

Due to the nature of this approach using local search many times through a solution, it is important to choose an algorithm for local search with appropriate behavior in terms of time and space complexity. A* search has a worst case space complexity of $O(b^m)$ where b is the branching factor and m is the maximum search depth. A* runs out of memory quickly on large domains. A* is useful in the refinement step as long as start and goal nodes are near each other. On the opposite end of the spectrum, IDA* has only linear memory consumption, but exponential search time (Hart et al., 1968). On a domain such as the 15-tile puzzle, which has very fast node generation, IDA* can often speed through a search tree and deliver an optimal solution quickly without running out of memory. On larger problems, this becomes less feasible both due to the branching factor and search depth.

Heuristics

Creating sub-problems within a global solution path introduces a limitation in the selection of heuristics for the local search. The nodes G_s and G_g change with each increment of the search window and therefore heuristics which depend on one of these states staying constant are not usable without significant modification.

This limitation makes the use of PDB heuristics complicated. Since a PDB heuristic relies on a single goal state, the use of PDB heuristics would require a different heuristic for each location of the blank, and a mapping of the tiles to the PDB tiles. This introduces a memory overhead in the number of PDBs needed. For a 15-tile puzzle using a set of two PDBs that split the puzzle in half, 30 PDBs would be necessary. Felner et al. (2004) shows that each 7-8 PDB takes 576,575 KB of memory, leading our modified PDB usage to take 8,648,625 KB. For the experiments done in this thesis, we are limited to 7.5GB, so PDB heuristics cannot be used.

Without PDB heuristics local search much less powerful than the state of the art for 15 and 24 tile solvers or blocks world. Instead, we depend upon the short length of the local search to simplify the problem.

3.1.1 Suboptimal Search

Suboptimal search is relevant in refinement solving since we are addressing problems which are very difficult to solve using A*. For instance, the 80-tile puzzle provides a situation in which A* may run out of memory and fail even for relatively short distances between nodes in the incumbent solution path. Node sizes and the number of nodes that must be expanded to locate a goal are limiting factors as puzzles get more difficult. With a primary objective of searching longer distances within the path in order to close large loops, A* may fail to give any results at all in terms of shortening the solution. This opens the door for suboptimal search to find better solutions than the incumbent path.

Weighted A* search is useful for these local searches on large puzzles. Often, the weight in weighted A* can be used as a knob to control how quickly a solution is returned, but it is defined as the bound on solution quality. In refinement, we are interested in solutions that are very close to optimal in order to find shorter paths than the incumbent path. Therefore, weighted A* with low, near-optimal weights is a good fit for refinement.

In LPA*, all local search paths returned are used to mend the global path whether or not they are shorter. This gives variety to the path and allows future search to explore different nodes than were available during the current refinement. Using suboptimal search means that we cannot always replace the path in the global solution with a new path since the local search may produce a path longer than the incumbent path.

Stopping

Applying suboptimal search to the refinement can also provide improvement. Since there is a bound on the length of a solution in hand, any suboptimal search that misses the optimal choice and provides a lower quality solution than the incumbent path can be stopped. This saves time by eliminating irrelevant search effort.

3.2 Node Selection

In order to find the biggest gains and avoid searching between nodes that result in challenging searches, we must carefully choose nodes to connect within the path. A poor choice can result in a search for shortcuts that will not return in time or a search for shortcuts between two nodes that are already represented in the solution with optimal path between each other.

3.2.1 Sliding Windows

A simple approach is to select nodes that are d steps apart. Thus a solution of length 100 split by $incr = 10$ would have 10 subpath searches between nodes. This method is part of the earliest experiments performed with solution refinement. More advanced approaches might include varying the $incr$ value or looking through the nodes on the path for h values between some node, n_1 and n_2. Two nodes far apart on the incumbent solution with low h value between nodes would be ideal targets for shortcuts.

Finally, keeping all nodes in a list of goals during search may allow for new shortcuts to be found that are not involved in the current search. These could be in the form of finding the global goal or finding a state farther down in the incumbent solution path with a shorter g value than the node in memory.

3.3 Evaluation

In order to illustrate the behavior of solution refinement, we will examine its performance on the tiles puzzle and blocks world. Each algorithm and domain were implemented in a fast C++ solver and run on 3.16GHz Intel Core 2 Duo machines. The solvers were given a maximum of 7.5GB of ram.

An equivalent implementation to LPA* was run on 15, 24, 35, 48, 63, and 80-tile puzzles and 15, 25, and 50-blocks world instances. For these experiments, each solver was given several *incr* parameters to control a window size. The solver was allowed to replace each length until it reached the end of the global solution and then return a path and start again using the refined solution as a new incumbent solution.

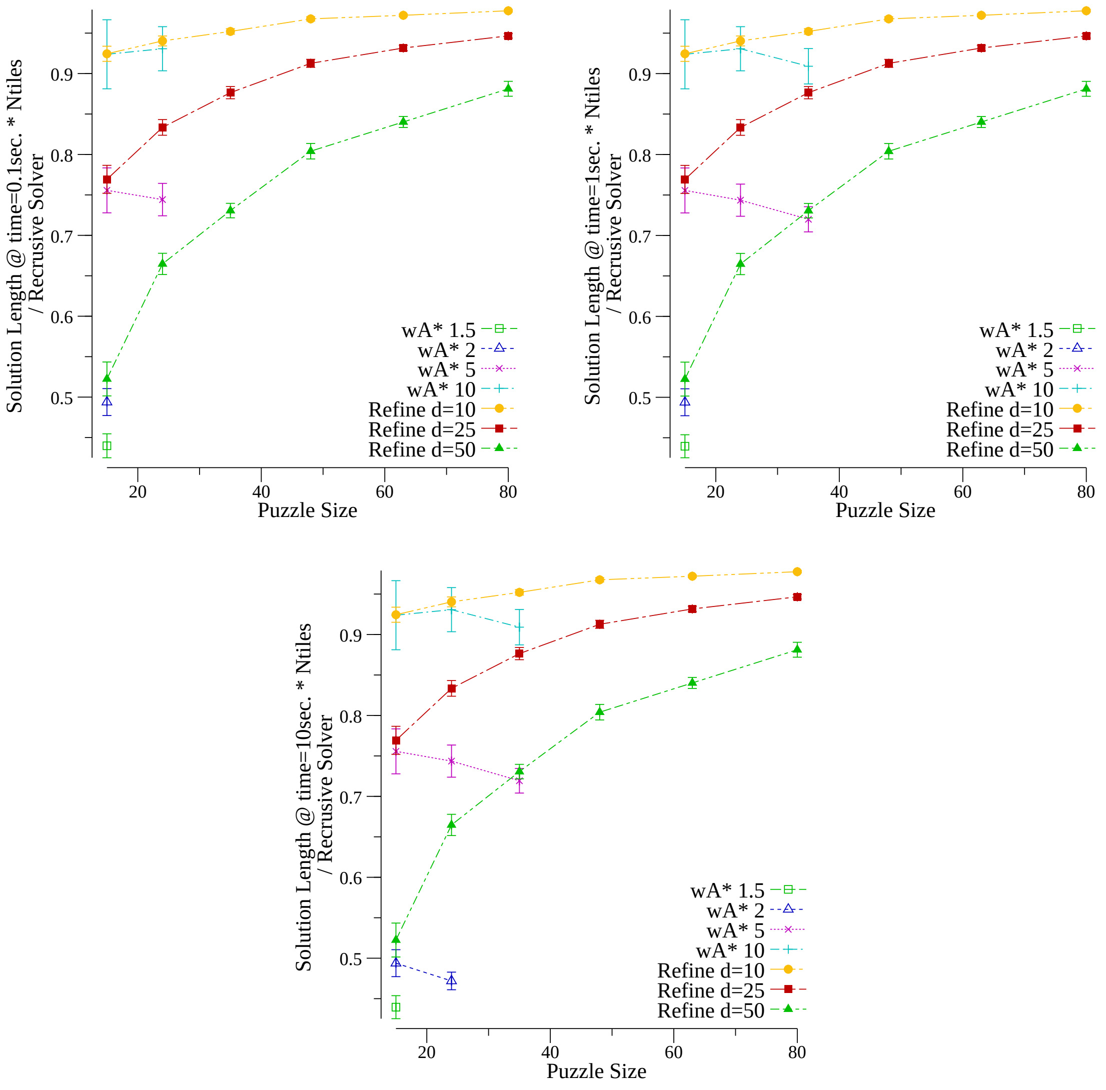

Figure 3-3: Refinement using A* over several puzzles

3.3.1 Tiles

Figure 3-3 shows the performance of the solver sampled at several cut off times relative to the size of each puzzle. This provides a view of the solution length at each time slice and shows how each algorithm behaves over all sizes of the puzzle. The time slices are scaled by the number of tiles in the puzzle. This means that for the first plot, the 15-puzzle was given 1.5 seconds and the 80-puzzle was given 8 seconds. These time slices were chosen in units that reveal how long weighted A* takes to find solutions for various puzzle sizes. Each data point along the lines represents the average best solution at the cut off

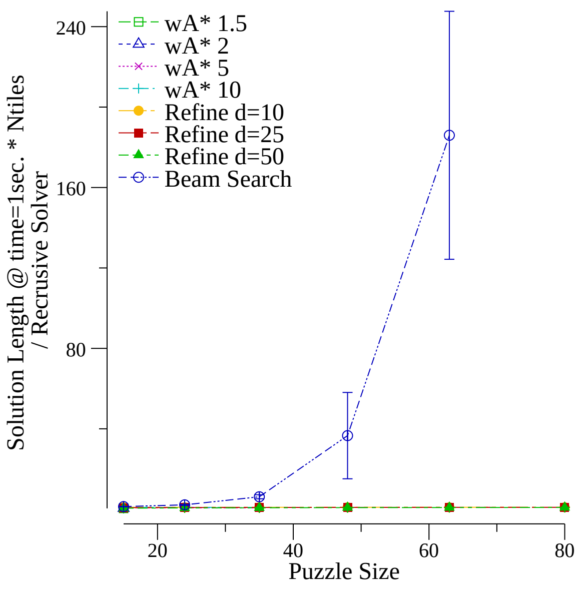

Figure 3-4: Refinement versus beam search

time over 100 randomly generated puzzles. The solution lengths were then paired to the recursive decomposition based solver to show how each algorithm performs in relation to the domain-specific solution.

These plots show that larger *incr* values result in shorter solutions across all puzzles, and that while the overall improvement amount lessens as problems become more difficult, the refinement is still able to find improvement on problems where other algorithms fail at finding any solution. Weighted A* is able to find a solution for the 15-puzzle very quickly for all weights sampled, but takes more time to find solutions for larger puzzles and ultimately cannot solve puzzles larger than 35 tiles.

Figure 3-4 presents the same results with beam search shown for perspective. As previously shown, the recursive solver dominates beam search in performance on the tile puzzle, and refinement can do no worse than the algorithm which provides its initial solution.

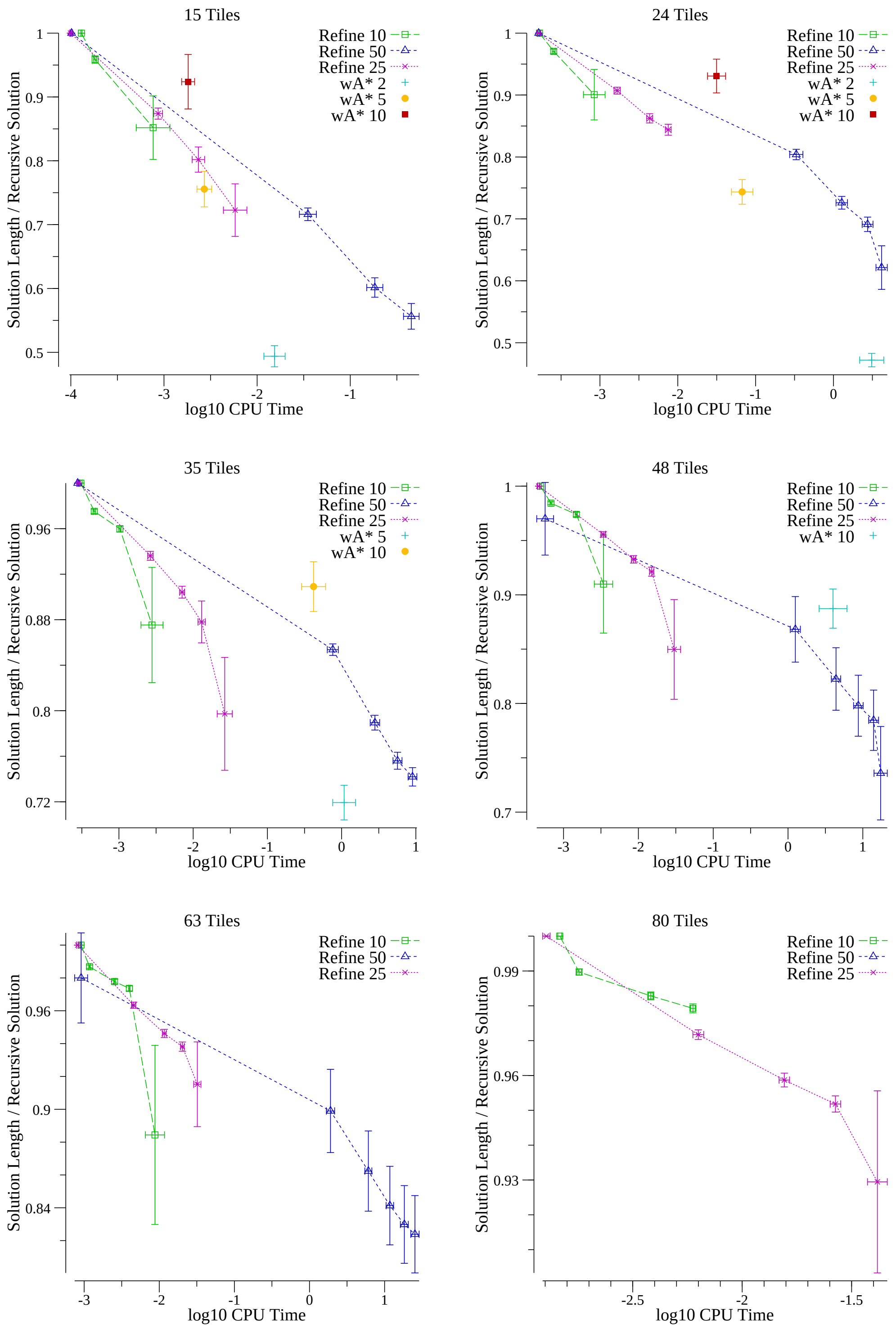

Figure 3-5: Time profile of refinement using A* over several puzzle sizes.

Figure 3-5 shows the anytime profile of the solver. Each data point is again the average of 100 random puzzles. For the refinement solver, each point on the line represents the average time and length of a solution reported at the end of a cycle through the global solution. This shows the improvement over time that the solver achieves as new solutions are reported. A point is drawn only if 95% of the solutions produce a solution for the given iteration. Therefore, many instances may have achieved even better performance with more runs through the data, but the average solution did not iterate through the solution more than the number of points plotted.

These plots demonstrate how the window size affects when the first solution is returned and how much improvement can be gained from search over that particular window size. Smaller windows tend to produce small improvements quickly while large windows can find more shortcuts. This gives motivation to find methods of improving the local search to expand the window sizes. The 80 tile puzzle does not only have a need for larger window sizes, but local search often fails when *incr* is set to 50.

3.3.2 Blocks

In order to have a better understanding of the behavior of solution refinement, an experiment with blocks world was created similar to the experiments with the tile domains. In these experiments, the number of blocks were varied between 15, 25, and 50. The blocks world instances were generated by the work from Slaney and Thiébaux (2001).

The performance in terms of time and memory usage of weighted A* and beam search on blocks world is not as high as that of the tile domains, so *incr* is limited to 30, a value that still produces shortcuts for the given problems without running out of memory. This implementation of blocks world is not highly optimized and its heuristic is a simple count of misplaced blocks, giving the number of single block moves necessary to solve the problem.

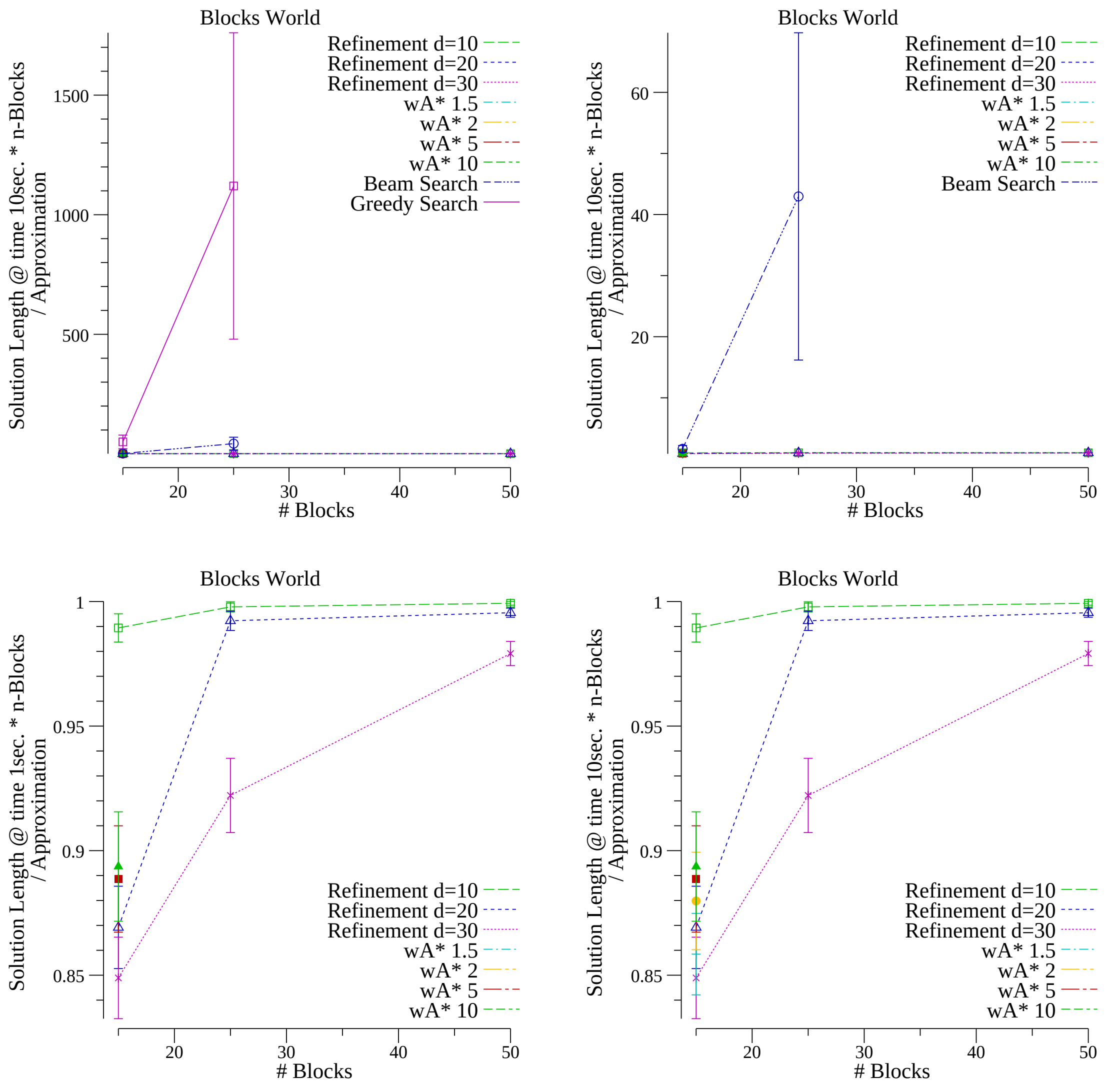

Figure 3-6: Refinement on blocks world

Figure 3-6 shows the refinement solver using A* on blocks world. In the top left chart, greedy seacrh is shown against beam search, weighted A*, and refinement methods. Unlike in tiles, greedy search does a very poor job here. The top right plot removes greedy search and shows beam search in comparison to the refinement methods. This is shown separately instead of using a log scale in an effort not to further complicate the Y-axis. The plot has a similar pattern to the tiles results. In this case, beam search is able to find reasonable solutions to the small 15-block problem, but finds very long paths on the 25-block problem. It is unable to find solutions with 50 blocks. As with tiles, beam search and greedy search

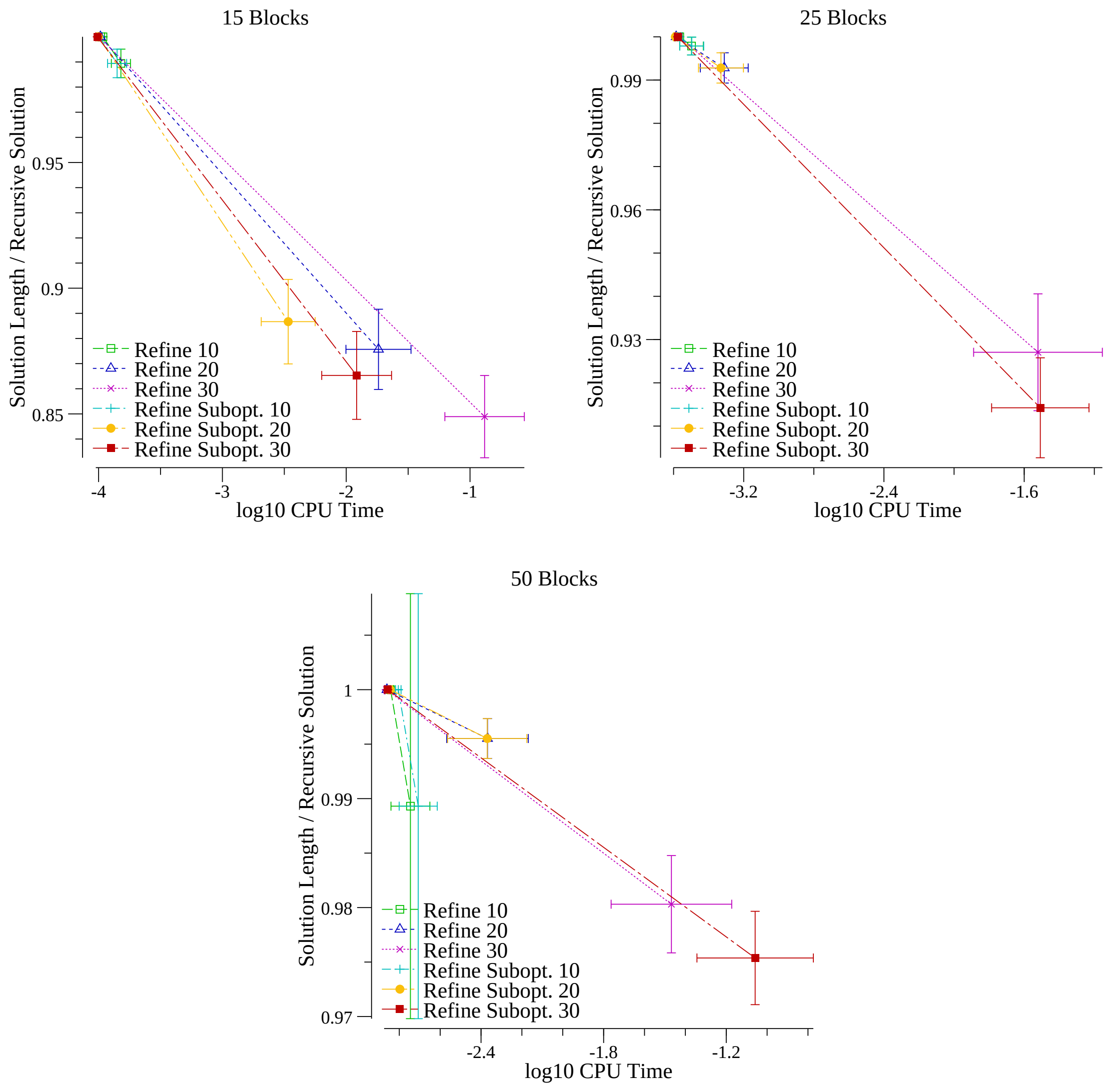

Figure 3-7: Time profile of refinement using A* and wA* (w=1.5) over several blocks world domains

both produce very poor solutions and cannot solve the largest problems. Weighted A* fails after 15-block problems. Refinement is shown in the bottom two plots with two time slices that have identical data. Like tiles, it is most effective when the window size is large, and it can be competitive with weighted A* for some weights. Finally, refinement is able to find shortcuts in the solution for small and medium size problems, but finds little improvement on large problems. In this case, refinement is less useful than in tiles, but in the mid-range number of blocks, performs well when compared to beam search.

Figure 3-7 shows the anytime performance of blocks world with optimal and suboptimal solvers used in refinement. The suboptimal solver is weighted A* with a weight of 1.5. This suboptimal behavior allows more of the local search refinement steps to complete instead of running out of memory, leading to better results when *incr* = 30. With such a low weight, the suboptimal solver behaves very similarly to the optimal version in many instances. This is a desirable trait, as we want to have the shortest local paths possible instead of wasting time by finding paths longer than the window size. The suboptimal solver improves on the optimal solver in terms of time for problems with 15 blocks and is able to find more shortcuts without running out of memory on problems with 50 blocks. On the mid-range 25-block problem, it finds slightly shorter solutions on average. This is a positive improvement for all three situations. These plots also show the reason that the refinement did not change between the $1 * $*n-Blocks* and $10 * $*n-Blocks* values in Figure 3-6: there was only one refinement step on average for each puzzle.

Since the blocks world domain is not highly optimized, its node sizes are much larger than that of the 80 tile domain, and its node expansion rate is slower. These limitations prevent weighted A* and beam search from performing particularly well, but are also the reasons for the limit on *incr*.

There is another phenomenon happening with blocks world worth discussing: the solutions constructed by Unstack-Stack are very regular in pattern. The algorithm has two modes of operation and always executes them in the same order. On small problems, this is often very close to the optimal solution allowing refinement only a few shortcuts. On larger problems, meaningful shortcuts require sizes of *incr* that tend to run out of memory. While beam search finds solutions within a reasonable range of the constructive solver on the 15-blocks world, it cannot solve large instances, results vary widely, and all attempts to solve 25 and 50-block problems with weighted A* fail. This gives evidence that when a simple constructive solver can be created, refinement can be used on the solutions returned in order to find paths that are closer to optimal in length.

With suboptimal refinement, the goal is to allow local search to use a larger *incr* and also to get results faster than A*. Using suboptimal search forgoes the guarantee that the path returned by the local search will be no longer than the path given in the global solution. However, using weighted A* with low weights yields solutions that are generally close to their A* counterparts. On blocks world, we can see that this does make some difference. On worlds with more blocks, the weighted A* solver is able to shorten the solution by a marginal amount, and on smaller worlds it operates faster than the optimal local search.

3.4 Conclusion

In this chapter, we have explored the basics of finding shortcuts in solution paths using solution refinement. This includes the ideas of searching with the guidance of a constructive solution, the effects of various window sizes, and how suboptimal search can be used to improve its behavior when the scale of a problem grows. We have seen that solution refinement can be used to find solutions in cases where bounded suboptimal search fails, but constructive algorithms exist.

While refinement is easily beaten on domains where weighted A* does not run out of memory, it produces higher quality solutions than beam search in all instances and can be used to provide solutions very quickly if necessary. We have shown that refinement can be used with domain-specific solvers to find solutions much shorter than those given by beam search with problems that are too difficult for weighted A*. We have also shown that for large *incr* values, weighted A* can find better solutions since it does not run out of memory when looking for shortcuts.

CHAPTER 4

Windowing

We have so far leveraged suboptimal solution refinement with constructive solvers to solve very large search problems. In this chapter, we will investigate two methods of improving the anytime profile of the algorithm by changing the ways in which windows are selected using iterative and offset windowing.

4.1 Iterative Windowing

Selecting a window size for refinement can be a non-trivial task. Too small a window size will provide very little improvement in the solution, while too large a window may lead to large local searches which eventually fail. Not only does this present an issue for finding window sizes that function, but the overall performance changes with the window sizes as well. A small window that produces small gains may finish quickly and provide new solutions while a window twice its size may take an unacceptable time to return its early solutions.

Taking notice that small windows return solutions quickly, but eventually cannot improve the solution, we now present iterative windowing. In iterative windowing, we begin refinement with small windows and expand the size when no improvement is possible.

```
1.  IterativeSolver($problem$, $incr$, $maxincr$)
2.    $soln$ = Solver($problem$)
3.    $length$ = size($soln$)
4.    $n = incr$
5.    while true
6.        $soln$ = PerformRefinement($soln[o..length]$, $n$)
7.        if Size($soln$) == $length$ and $n + incr \leq maxincr$
8.            $n$ += $incr$
9.        else
10.           $length$ = Size($soln$)
11.           Output($soln$)
```

Figure 4-1: Pseudocode for incremental windows

Incremental windows are shown in 4-1. The window, n, is set to *incr*, as an initial size. Each time a full walk through the solution provides no improvement, lines 7 and 8 add *incr* to n to expand the size until n grows to the size of *maxincr*.

4.2 Offset Windows

Offset windows provide another approach to finding shorter paths by selecting nodes which are not on the same path as the original nodes. In Joint (Ratner and Pohl, 1986), a pass through the global solution is made and refinement is performed around each joint to remove any original nodes from the path. We argue that this behavior is achieved naturally by the nature of the shortening of the solutions and iterative windowing. When refinement returns a shorter path, the next run through the solution will give different nodes at each point where the window begins and ends. When this is not true due to the solution not being resized at all, a new *incr* is chosen and shortening continues this behavior. However, there are corner cases which must be addressed in order for nodes do be selected which are not in the original solution path. Specifically, if the first joint stands unrefined due to suboptimal search returning too long a path, then the first local goal node may never have been updated. This holds true for any case where the path before the window has been unchanged.

A simple solution to repairing these paths is to perform local search on the path offset

```
1.  OffsetSolver($problem$, $incr$, $offset$)
2.  $soln$ = Solver($problem$)
3.  $length$ = size($soln$)
4.  $o = 0$
5.  $n = incr$
5.  while true
6.      $soln$ = PerformRefinement($soln[o..length]$, $n$)
7.      if Size($soln$) == $length$ and $o = 0$
8.          $o = offset$
9.      else if Size($soln$) == $length$
10.         $n$ += $incr$
11.     else
12.         $length$ = Size($soln$)
13.         Output($soln$)
```

Figure 4-2: Pseudocode for offset windows

from the first node. Figure 4-2 defines the algorithm as follows; set the *offset* parameter to some value ≥ 0, and repeatedly perform solution refinement on the incumbent global solution until no improvements can be found. When all solutions are exhausted, lines 7 and 8 set the start node, G_s to G_{offset} in the path and continue refinement using the new offset start node through the end of the path (line 6, *o..length*). If refinement returns without improvement and an offset is already in place, grow the window size as done in Figure 4-1.

4.3 Evaluation

We will now examine how these improvements affect the performance of refinement using the same problems seen in the previous chapter. As before, all processes were given 7.5GB of memory on 3.16GHz machines. For incremental refinement, a minimum window size of 10 steps and a maximum window size of 50 steps was selected for tiles and a maximum of 30 steps for blocks world. Each time no improvement was found, the window size was increased by 10 steps. Offset refinement was performed in the same way except that before adding 10 to the window size, the solution was offset by 5 steps and refinement was run until no improvement could be found. These values were chosen to compare to the previous results for suboptimal refinement in Chapter 3.

As with previous plots showing the anytime profile, data points are shown only when 95% of the instances have a solution at that iteration. Many instances may have more refinements producing even shorter paths.

4.3.1 Tiles

Figure 4-3 shows a plot for each of the N-tiles domains running refinement with incremental windows and offset incremental windows. The incremental window returns solutions with small improvements quickly and then moves on to the larger increments finding more and more improvement. This behavior makes the anytime output more regular in terms of timing and still finds similar levels of improvement as the non-incremental version with a window size of 25. Unfortunately, the incremental solver does not find the high quality solutions that non-incremental search finds at larger window sizes.

Offset windows generally allow the incremental search to move farther through the solution and produce a greater number of full solutions, but still may not shorten the solution as far as static window sizes. On the 80-puzzle, we see that not only does it find similar solution lengths to the suboptimal solver, but the variance is much lower towards the final refinements. In general, the increase in local searches tends to cause offset search to take more time to produce solutions.

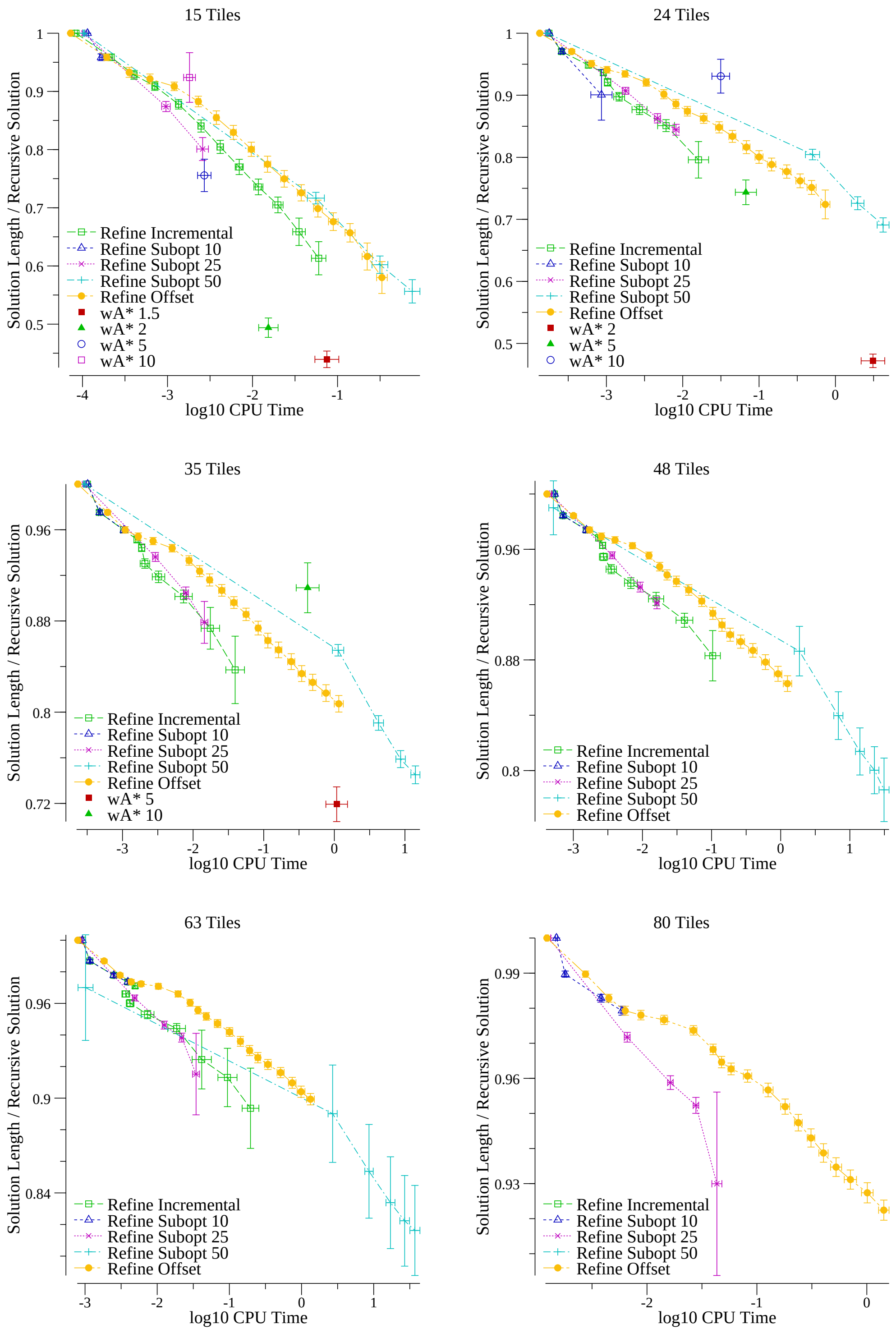

Figure 4-3: Time profile of sliding tile refinement using iterative window sizes.

4.3.2 Blocks

Figure 4-4 shows the results of refinement on blocks world. In this instance, incremental refinement leads to highly improved solutions as compared to static window sizes. On the

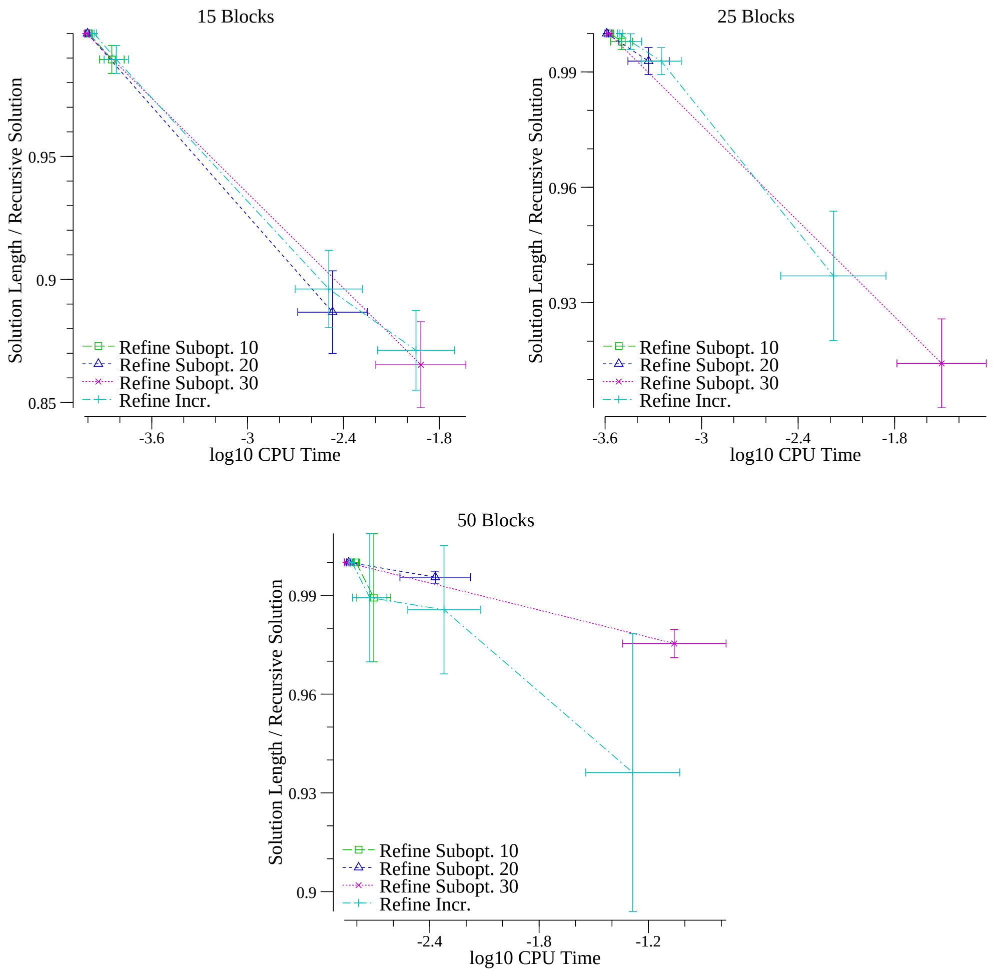

Figure 4-4: Time profile of blocks world refinement using iterative window sizes.

50-blocks problem, refinement is able to take more steps and return solutions 5% shorter on average than the US algorithm.

The behavior of incremental refinement on blocks world shows interesting changes from the behavior on the sliding tile puzzle. Here, we see the incremental window search finding similar solutions for each window size on the 15-block world. With 25 blocks, it returns a slightly longer solution in less time than a fixed window size of 30. Finally, with 50 blocks, the incremental search finds higher quality solutions than its counterparts. It is possible that this is due to a rearrangement of actions in the middle of the plan by short windows that were then improved upon when the window size increased, which a static window size would be unable to detect.

4.4 Conclusion

This chapter has introduced two methods of improving the output of refined solutions by changing the methods of selecting start and goal nodes for local search. Incremental windows allow faster returns of local search solutions with shorter paths than small windows, and offset search allows more iterations to run. In some cases, this can produce shorter paths than search without offsetting, and generally produces more refined solutions.

CHAPTER 5

Discussion

We have seen that solution refinement can be used with domain-specific solvers to find solutions to large problems and that modifications to the windowing method that improve its anytime profile. In this chapter, we will discuss the limitations of the method, ideas for future work to improve it, and conclusions for the thesis.

5.1 Limitations

Solution refinement shortening paths and producing final solutions many times shorter than beam search solving the same problems, but there are cases where solution refinement fails to give meaningful shortcuts over the initial solution. Some situations in which the method performs poorly include high end sliding tile puzzles, domains where optimal or very near optimal domain- specific solutions exist. In addition to this, the method is highly dependent on the structure of both the domain and the solution given by the domain- specific solver.

On sliding tile puzzles, we have seen that the 63 and 80-puzzles consistently give only small amounts of improvement. Although the improvement in relation to the initial solution is small, it may still represent many hundreds fewer steps in the solution. Part of this behavior may be caused by the ratio between window size and the size of the problem. Some instances in the 15-tile domain may be solved optimally in 50 steps or the size of our largest selected window size. On the other hand, no 80-puzzle is solvable in this number of moves. The same behavior is seen at the higher end of blocks world as well, and the distance between nodes selected in the global solution must be closer in order to prevent search from failing.

In blocks world, we have access to better domain-specific solvers than the US solver. In general, the closer to optimal that the initial solution is, the less chance for improvement exists, and time spent searching for improvements may not be well spent. In this particular domain, an optimal solver exists that can solve all of the problems seen in the experiments shown in this thesis. While this is true, the US algorithm still allows us to show how refinement works in a situation where there is some structure to the underlying problem that causes problems for refinement.

As discussed in Section 3.3, there is an underlying structure to the problem which prevents windowed refinement from finding new optimal paths. If the optimal path for a global solution does not exhibit the same behavior that the constructive solution has, and the constructive solution has optimal subpaths within its own solution, then refinement will be unlikely to give meaningful benefits. In refinement, we are looking for shortcuts along a suboptimal path, but few shortcuts may exist.

5.2 Future Work

In all versions of this work, we have focused on selecting local search windows that are chosen only by their distance apart. Leveraging the fact that we have an upper bound on the length of the path given to us by the distance between nodes in the solution path, choosing start and goal nodes based on heuristic value may lead to finding valuable shortcuts. A scheme for implementing this may be to take the length of the path between two nodes in the incumbent solution, L, and look along the path to find areas where the heuristic value between the start node and current node is less than L. Finally, we would choose the biggest disparities and search between these nodes.

In addition to making the choice of start and goal nodes intelligently, searching with multiple goals may also lead to better short cuts. All nodes on the incumbent solution path can be seen as potential shortcuts during local search. Therefore, if local search expands a node on the incumbent path, then a shortcut may have been found.

Due to the independent nature of the local search operations on an incumbent path, this problem is embarrassingly parallel. One expansion on this work would be to distribute local search work among several machines in order to return refined solutions faster than the parallel version. The overhead of organizing workers and transferring data should be manageable, making the speedup largely dependent on the longest running local search and number of CPUs allocated. Each search run by the scheduler is potentially a very large search problem, so parallel execution on a single machine would require a large amount of memory. Though this is not unreasonable, an 8-core machine would need 60GB of memory to repeat the experiments used in this thesis using 8 workers.

One limiting factor for the local search processes is search guidance. PDB heuristics are much more powerful than the heuristics used in this research. An approach to providing more powerful heuristic information in a domain where the goal is not static is to use an algorithm based on hierarchical search such as ShortCircuit or Switch (Leighton et al., 2011). These algorithms provide heuristic values based on abstraction and can potentially give local search more guidance and a longer reach through the incumbent solution paths.

5.3 Conclusion

Search in large state spaces is a complex task involving trade-offs between solution lengths and time spent on search effort. These domains present complex search spaces where optimal and suboptimal solvers such as A*, IDA*, and weighted A* fail. In these problems, algorithms such as beam search and real-time search are the primary means of finding solutions via search. These algorithms tend to return very long solutions and potentially take an unacceptable amount of time doing so.

Hand crafting suboptimal constructive solvers on complicated domains can be difficult but can lead to quickly found, valid solutions. Solutions returned by these algorithms can be orders of magnitude better than the results of beam search and therefore BULB search, assuming a solution can be found with the amount of memory available to beam search. With the extra time between returning a solution via recursive or constructive means, it is possible to improve the quality of the solution, yielding a closer to optimal path on domains in which A* cannot be used.

We have explored the behavior of solution refinement, showing that exploiting the speed of hand-crafted solvers can act as an engine for domain knowledge. In addition to the use of an incumbent solution as domain knowledge, we have used these solutions as bounds on the final solution length and used those bounds to control suboptimal search, finding better and better solutions as time allows. Using suboptimal search, we are able to obtain shorter solutions with large windows. By varying the window sizes and offsets, we are able to improve the anytime profile of the search, giving more solutions at a regular pace. We can conclude that solution refinement on solutions from domain-specific solvers is a viable alternative to beam search on very large puzzles.

BIBLIOGRAPHY

Roberto Bisiani. *Encyclopedia of Artificial Intelligence*, volume 2, chapter Beam Search, pages 1467–1468. John Wiley and Sons, 2 edition, 1992.

E. Burns, M. Hatem, M.J. Leighton, and W. Ruml. Implementing fast heuristic search code. In *Proceedings of The Fifth Annual Symposium on Combinatorial Search (SoCS-12)*, 2012.

S.V. Chenoweth. On the np-ardness of blocks world. In *Proceedings of the Ninth National Conference on Artificial intelligence (AAAI-91)*, pages 623–627. AAAI Press, 1991.

A. Felner, R.E. Korf, and S. Hanan. Additive pattern database heuristics. *J. Artif. Intell. Res. (JAIR)*, 22:279–318, 2004.

D. Furcy and S. Koenig. Limited discrepancy beam search. In *International Joint Conference on Artificial Intelligence*, volume 19, page 125, 2005.

David Furcy. ITSA*: Iterative tunneling search with A*. In *AAAI 2006 Workshop on Learning for Search*, 2006.

N. Gupta and D.S. Nau. On the complexity of blocks-world planning. *Artificial Intelligence*, 56(2):223–254, 1992.

N. Gupta, D.S. Nau, et al. Complexity results for blocks-world planning. In *Proceedings of AAAI-91*, volume 629, 1991.

O. Hansson, A. Mayer, and M. Yung. Criticizing solutions to relaxed models yields powerful admissible heuristics. *Information Sciences*, 63(3):207–227, 1992.

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.

R.E. Korf and A. Felner. Disjoint pattern database heuristics. *Artificial intelligence*, 134 (1-2):9–22, 2002.

R.E. Korf and L.A. Taylor. Finding optimal solutions to the twenty-four puzzle. In *Proceedings of the thirteenth national conference on Artificial intelligence Volume 2*, pages 1202–1207. AAAI Press, 1996.

Richard E. Korf, Michael Reid, and Stefan Edelkamp. Time complexity of iterative-deepening-a*. *Artificial Intelligence*, 129:199–218, 2001.

Michael Leighton, Wheeler Ruml, and Robert Holte. Faster optimal and suboptimal hierarchical search. In *Proceedings of The Fourth Annual Symposium on Combinatorial Search (SoCS-11)*, 2011.

Ian Parberry. A real-time algorithm for the (n 2 1)-puzzle. *Inf. Process. Lett*, 56:23–28, 1995.

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

D. Ratner and I. Pohl. Joint and lpa: Combination of approximation and search. In *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, 1986.

D. Ratner and M. Warmuth. Finding a shortest solution for the n$\times$ n extension of the 15-puzzle is intractable. In *Proceedings of AAAI*, volume 8, pages 168–172, 1986.

Aleksander Sadikov and Ivan Bratko. Solving 20x20 puzzles. In *Computer games workshop 2007, Amsterdam, June 15-17, 2007*, pages 157–164, Amsterdam, The Netherlands, The Netherlands, 2007.

J. Slaney and S. Thiébaux. Blocks world revisited. *Artificial Intelligence*, 125(1-2):119–153, 2001.

Christopher Wilt, Jordan Thayer, and Wheeler Ruml. A comparison of greedy search algorithms. In *Proceedings of the Third Symposium on Combinatorial Search*, July 2010.

Rong Zhou and Eric A. Hansen. Beam-stack search: Integrating backtracking with beam search. In *Proceedings of ICAPS-05*, 2005.