Anytime Heuristic Search: Frameworks and Algorithms

Jordan Thayer and **Wheeler Ruml**
⟨ jtd7, ruml ⟩ at cs.unh.edu
The University of New Hampshire
Department of Computer Science
Durham, NH 03824 USA

Abstract

Anytime search is a pragmatic approach for trading solution cost and solving time. It can also be used for solving problems within a time bound. Three frameworks for constructing anytime algorithms from bounded suboptimal search have been proposed: continuing search, repairing search, and restarting search, but what combination of suboptimal search and anytime framework performs best? An extensive empirical evaluation results in several novel algorithms and reveals that the relative performance of frameworks is essentially fixed, with the repairing framework having the strongest overall performance. As part of our study, we present two enhancements to Anytime Window A* that allow it to solve a wider range of problems and hastens its convergence on optimal solutions.

Introduction

Many applications of heuristic search limit the resources available for problem solving. Anytime heuristic search (Boddy and Dean 1989) is particularly appealing when time is limited. An anytime algorithm can be run so that it returns an improving stream of solutions, eventually converging on an optimal solution. This allows the anytime algorithm to gracefully expand to consume all of the allotted time, using it to produce a solution of greater quality than the initial solution.

Many anytime search algorithms can be viewed as general frameworks for extending bounded suboptimal searches into anytime algorithms. Although these conversion techniques can be applied to any bounded suboptimal search, previous evaluations focus on weighted A* (Pohl 1970). Other algorithms deserve consideration, particularly those that outperform weighted A* on common benchmarks. We naturally expect this improved performance to extend to anytime search frameworks.

Previous comparisons of the anytime algorithms focus on a limited range of spaces. In their paper on anytime repairing A* (ARA*), Likhachev, Gordon, and Thrun (2003) restrict their evaluation to two domains, the robotic arm and dynamic robot pathfinding. These benchmarks have very high branching factor and relatively few duplicate states, thus representing a small fraction of the space of possible problems. Other evaluations have erred in the other direction, focusing primarily on domains which are trees of bounded depth where each leaf is a solution (Aine, Chakrabarti, and Kumal 2007). Both types of benchmarks are required for a realistic impression of the performance of an algorithm.

Hansen and Zhou (2007) argued that the approach taken by ARA* is flawed. They note that decreasing weights and delaying the expansion of duplicate states is of limited utility and can be harmful. Delaying duplicates can increase the cost of the solution found during an iteration of ARA*, potentially increasing the time to converge on optimal solutions. While delaying duplicates is only of moderate value in the benchmarks used by Likhachev, Gordon, and Thrun (2003), it is enormously important for some domains such as pathfinding in grids, where duplicate handling determines performance. The implementation of weight schedules in Hansen and Zhou (2007) was different from the one suggested by Likhachev, Gordon, and Thrun (2003). ARA* uses the minimum of the decremented weight and a calculated lower bound, while the evaluation in Hansen and Zhou (2007) only considers the decremented weight.

Richter, Thayer, and Ruml (2009) compare restarting weighted A* against anytime heuristic search and anytime repairing A*, as well as several other state of the art anytime algorithms on a wide variety of benchmarks including planning, the sliding tile puzzle, the robotic arm domain, and a synthetic tree. Unfortunately, this comparison is limited to weighted A* based anytime searches as well as beam searches and anytime window A*.

This paper fills in these holes in the empirical analyses of anytime search algorithms. We test the performance of a wide variety of bounded suboptimal algorithms within all three general anytime search frameworks across eight diverse benchmarks in an attempt to study their general performance. We compare these bounded suboptimal based anytime searches against each other as well as beam stack search and anytime window A*. We introduce two improvements to anytime window A*, allowing it to solve problems it previously could not.

After discussing the three frameworks for converting bounded suboptimal algorithms into anytime searches, we discuss the bounded suboptimal algorithms themselves. We present an evaluation showing that although there are large differences in the underlying search algorithms, the rela-

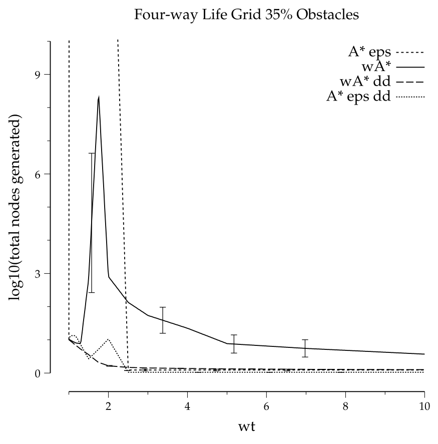


Figure 1: Impact of ignoring duplicates

tive performance of the frameworks is essentially fixed. We compare the framework based algorithms against other state of the art anytime search algorithms such as anytime window A* and beam stack search. On the test domains considered, the repairing framework is best.

Frameworks

There are three previously proposed frameworks that convert bounded suboptimal search algorithms into anytime searches. We may choose to continue the search after the first solution is found without changing anything at all, we may continue after an initial solution is found and reconfigure the search, or we could also choose to simply run several bounded suboptimal search algorithms in succession.

Continuing a bounded suboptimal search beyond the first encountered solution was introduced by Hansen and Zhou (2007). If the search is continued it will produce a stream of ever improving solutions, eventually finding the optimal solution. Continued search is sensitive to the configuration of the underlying bounded suboptimal search. There will naturally be some sensitivity to the underlying algorithm for all frameworks, but unlike repairing or restarting search, continued search never reconsiders the initial configuration of the underlying algorithm. As a result it is very reliant on pruning for performance. Thus, it performs best in domains with strong admissible heuristics where greedy search produces good solutions, and many nodes can be pruned once an incumbent solution is in hand. It has difficulties in domains where there are many cycles because it cannot ignore improved paths to an already visited state. Although the underlying bounded suboptimal algorithms may be able to ignore duplicate states while still respecting a suboptimality bound (Ebendt and Drechsler 2009), ignoring these nodes during a continued anytime search would prevent us from converging on optimal.

Repairing searches differ from continued searches in two ways. First, they have a special way for handling duplicate nodes. When repairing search encounters a better path to a state which it has already expanded, it places this state onto a list of inconsistent nodes rather than immediately re-expanding it. These nodes will not be selected for expansion until the next iteration of repairing search. While this may decrease the quality of the solution found on any iteration of repairing search, it leads to improved performance in domains with many cycles by decreasing the time it takes to find a solution on any iteration, as seen in Figure 1. Here, we show the performance of A^*_ϵ (Pearl and Kim 1982) and weighted A* (Pohl 1970) on a grid pathfinding problem. The y-axis represents the number of nodes generated while finding a solution on a log scale. The x-axis represents the parameter that the algorithm was run with. Algorithms with 'dd' appended do not re-expand duplicate states, instead they ignore duplicate states whenever they are encountered. While this can decrease solution quality, and even quality bounds for some algorithms, ignoring duplicates allows both of these algorithms to solve the problems while generating orders of magnitude fewer nodes. In the event that ignoring duplicate nodes loosens the desired suboptimality bound, as it does in every algorithm but weighted A*, the anytime nature of the framework will ensure that we still converge on an optimal solution, but the speedup will still extend to every iteration of the search.

Second, repairing searches rely on parameter schedules. These are typically constructed by selecting a starting parameter and a decrement for the parameter, although they may also be specified by hand. Every time a new solution is encountered the parameters are updated. Either they are replaced by the next set on the schedule, or they are replaced by a dynamic lower bound on solution quality [1], whichever is tighter. There are now two parameters that need tuning: the starting weight and the decrement. Set the decrement to be too large, and the next iteration may never finish; however, if the decrement is too small, the dynamic bound will always be used for the next iteration, and this may not provide a large enough step towards optimality. Changing the parameters used by the search requires updating the evaluation of every node the search is currently considering. While touching every node will take time, it also allows for the immediate pruning of every node that cannot lead to an improved solution. This considerably reduces the size of the open list and thus reduces overhead.

Restarting is an incredibly straightforward approach to converting bounded suboptimal search into anytime search. The underlying algorithm is run with progressively tightening suboptimality bounds, each time restarting from the initial search state. Heuristic calculations and the best path to a state from the root are stored across iterations. Like repairing search, restarting can choose to ignore duplicates during any iteration, save the last. Restarting allows these anytime algorithms to reconsider decisions made early on in the search and avoid a problem called low-h-bias. The weighting in weighted A* causes nodes with low cost to go estimates to look more desirable than they may actually be. As a result, weighted A* may spend a large amount of time searching portions of the space where there are no or very few good solutions. Not every domain is structured so that low-h-bias causes problems; we observed it in only one of our benchmarks, vacuum planning. Further, not every al-

[1]cost of incumbent divided by the minimum f_{A*} of any node

gorithm is prone to low-h-bias. This property is unique to weighted searches such as weighted A*. These algorithms rely on a single heuristic, h, and their performance hinges on placing more importance on h than on the cost of arriving at a node. Other bounded suboptimal algorithms, A^*_ϵ for example, do not do this, and thus do not suffer in domains with low-h-bias.

Bounded Suboptimal Algorithms

We briefly describe each algorithm and their performance relative to one another both within and without the anytime frameworks. We will see that although the algorithms have very different behaviors as bounded suboptimal searches, this rarely changes the relative performance of the frameworks.

Weighted A* (Pohl 1970) alters the node evaluation function of A* (Hart, Nilsson, and Raphael 1968) by placing additional emphasis on the heuristic evaluation function h, making it appear more important than the cost of arriving at a node, g. That is, the cost function of A*, $f_{A*}(n) = g(n) + h(n)$ becomes $f_{wA*}(n) = g(n) + w \cdot h(n)$. Thus it tends to prefer nodes that appear to be closer to a goal.

Revised Dynamically Weighted A* (Thayer and Ruml 2009) attempts to extend this approach by using two measures of goal proximity: h, and d, the estimated number of steps to the goal. As nodes progress towards the goal, the weight is steadily decreased. Revised dynamically weighted A* sorts nodes in order of $f_{rdwA^*} = g(n) + max(1, w\frac{d(n)}{d(root)})h(n)$. Revised dynamically weighted A* was constructed to correct a flaw in the original implementation of dynamically weighted A* (Pohl 1973). In the original formulation, weight was decreased based on the depth of a node, implicitly assuming that every step away from the root is one towards a goal. While this is true in the traveling salesman problem, the benchmark used in the original paper, it doesn't hold for many common benchmarks including permutation puzzles and pathfinding.

A^*_ϵ (Pearl and Kim 1982) also considers the distance of a node from a goal when deciding which to expand next. Rather than incorporating this information into the evaluation function, A^*_ϵ maintains two orderings on the nodes. In the first ordering nodes are sorted in order of f_{A^*}, forming the open list. The node at the front of the open list is the one with minimum f_{A^*}. In order to select nodes that are close to a goal, A^*_ϵ maintains a list of nodes sorted on d, called focal. Focal contains nodes that we can prove lie on a path to a w-admissible solution, and these are sorted in order of d. The node at the front of focal is expanded until a solution is found.

AlphA* (Reese 1999) also uses the idea of maintaining multiple orderings over the nodes. Nodes are stored using one of two cost functions. Either they are stored with their f_{A*} values or their f_{wA*} value. If a node's parent has an h value greater than the last node expanded, it is stored with f_{wA*}, otherwise it is stored with f_{A*}.

A_ϵ (Ghallab and Allard 1983) is very similar in form to A^*_ϵ. It starts by expanding n such that $argmin_n \; d(n) : f_{A*}(n) \leq w \cdot f_{A*}(best_f)$, exactly the node that A^*_ϵ expands every iteration. A_ϵ commits to this node, not unlike the way realtime search algorithms commit to a node, and repeatedly follows the best child from each expansion until a goal is found or until the best child would not be within the bound. If the best child would be outside of the bound, A_ϵ must either abandon its commitment, expanding n such that $argmin_n \; d(n) : f_{A*}(n) \leq w \cdot f_{A*}(best_f)$, or it may instead choose to persevere, expanding the node with minimum f_{A*} until the desired child is within the bound, continuing along this path once the lower bound on optimal solution cost has been sufficiently raised.

Bound	1.5	1.75	2.	3.	4.	5.
wA*	4.1	3.4	2.8	3.7	3.4	2.4
$A*_\epsilon$	50.4	44.8	28.5	1.8	1.1	**0.6**
Clamped	8.3	10.1	11.6	67.0	85.6	85.8
AlphA*	126.6	140.1	181.6	282.2	309.3	315.0
rdwA*	374.1	316.9	245.1	101.0	84.8	128.1
A_ϵ	911.4	857.7	683.2	624.9	597.4	614.3

Figure 2: CPU usage as a factor of EES

Clamped Adaptive (Thayer, Ruml, and Bitton 2008) is a simple technique for using an inadmissible cost function $\widehat{h}$ to guide our search while maintaining bounded suboptimality. We merely restrict the estimated cost of a solution traveling through a node, $\widehat{f}(n) = g(n) + \widehat{h}(n)$ to never be larger than $w \cdot f_{A*}$ as in $f_{ca}(n) = min((w \cdot f_{A*}), \widehat{f}(n))$. In many domains, clamped adaptive search can be too conservative, so we place additional weight on the inadmissible heuristic to correct for this as in $\widehat{f}'(n) = g(n) + w \cdot \widehat{h}(n)$. Although there are many ways to construct accurate but potentially inadmissible heuristics, we use temporal difference learning (Sutton 1988) to construct $\widehat{h}$ from h.

Explicit Estimation Search (Thayer and Ruml 2010) There is one significant drawback to clamped adaptive search: it cannot make use of search distance information provided by d unless it is used in the construction of $\widehat{h}$. Yet the stated goal of bounded suboptimal search is to find a solution within the bound as quickly as possible, so it seems necessary to rely on d to form an appropriate search order. Explicit Estimation Search (EES) uses $\widehat{f}$ to estimate the true cost of a solution through a node and a corrected version of d to estimate the search effort remaining beneath a node.

Relative Performance

The bounded suboptimal algorithms we just introduced were constructed to take advantage of additional sources of information to address problems with previous search algorithms, or to do both. Their performance differs, sometimes greatly, between domains. We would expect the performance of the bounded suboptimal searches to impact the performance of an anytime algorithm based on them. We now summarize the relative performance of the bounded suboptimal algorithms, on their own and within anytime frameworks.

Figure 2 summarizes the relative performance of these bounded suboptimal algorithms across the domains in the evaluation. Here, we present the performance of all other

	Continued	Restarting	Repairing
wA*	263.38	288.45	298.55
A^*_ϵ	227.48	220.44	260.82
rdwA*	11.06	286.66	298.41
Clamped	279.95	264.46	298.28
EES	227.80	295.81	297.23

Figure 3: Performance score on life grids

bounded suboptimal algorithms as a factor of the performance of EES, with a value of two in a column meaning that an algorithm consumed twice as much CPU time on average across all instances for all domains at that particular suboptimality bound. The number is calculated by comparing the runtime of the algorithm with the runtime of EES on a per instance basis. We then take the average all of the instances within a domain, and then take the average value across all domains. We do this to avoid unfairly weighting any given domain, since we have a different number of instances for each domain in our study. We clearly see that, with the exception of A^*_ϵ with suboptimality bound of 5, all algorithms take more time for all suboptimality bounds than explicit estimation search does. See Thayer and Ruml (2010) for a more thorough comparison of these algorithms.

Despite the potentially large differences in the underlying best first search algorithms, with a given algorithm, the relative performance of the frameworks is only rarely affected by the choice of bounded suboptimal search, as demonstrated in Figure 3, where we see that, without exception, repairing search outperforms all other approaches, and that continued search is almost always the worst approach. In Figure 3 we've reduced the performance curve of an anytime algorithm, in this case the CPU time consumed on versus solution quality, into a single value, the area beneath that curve. Time ranges between 0 and 300 seconds, and the quality of a solutions ranges between 0 and 1. Quality is computed by dividing the cost of the best solution found by any algorithm on an instance by the cost of the current solution of the anytime algorithm. Frameworks are represented as columns, and best first searches as rows. We omit the results of AlphA* and A_ϵ because they fail to find an initial solution to many of the benchmarks. We show the results for Life Grids, but other domains were similar. For all underlying algorithms, the repairing framework performed best. Note that not only did it perform best, but the performance of algorithms within the repairing framework is very similar. For example, clamped adaptive search consumed tens of times more CPU time than EES for any suboptimality bound. However, when we evaluate them as the centerpiece of an anytime algorithm, their performance becomes quite similar. For the domains in our evaluation the continued framework performed worse than both the restarting and the repairing framework, and the repairing framework was always better than the restarting framework.

The ordering of frameworks occasionally changes when using clamped adaptive search as the underlying best first search. Its continued search variant is much stronger when compared to other continued search algorithms. Continued searches do not alter their parameters during the course of a run, but clamped adaptive search learns a correction to the admissible heuristic over the course of the search, effectively altering the algorithm's initial parameter settings. It avoids one of the largest drawback of the continued search framework, and this explains its surprisingly good performance. It still doesn't pay special attention to duplicate nodes, and this is why it is not better than the repairing framework.

Alternate Approaches

There are anytime searches that are not frameworks for extending bounded suboptimal algorithms into anytime searches. These include beam stack search, BULB, anytime window A*, and branch and bound. Branch and bound performs poorly for all of the benchmarks problems presented here excluding the TSP. The traveling salesman problem is the only domain we examined with a fixed depth. As a result of this fixed depth, depth first approaches like branch and bound can find an incumbent solution quickly, and begin pruning starting the process of converging on an optimal solution. When the safety net of a fixed depth is removed, finding any solution with a depth first search is extremely challenging, and converging on an optimal solution may happen, but it will take a remarkably long time. For example for the 4-connected grid pathfinding problems we considered, A* will solve the problem in less than 2 seconds for all instances we considered, while branch and bound fails to find any solution within the first five minutes. This isn't simply a problem with one domain, it happens in every domain in our evaluation save the TSP. As a result, we omit discussion of it, instead focusing on the more general algorithms which can solve problems of bounded and unbounded depth.

Beam Searches

Beam search is a memory limited search where a set number of nodes at each depth are expanded. The beam is typically some form of 'leaky' priority queue, where the best elements that fit within the size limit are held. When a new element is added to the beam, if the beam is at capacity, the worst element is discarded. Since nodes are discarded before a solution is found, the search is incomplete, but it can be extended into a complete anytime search in several ways.

Beam stack search (Zhou and Hansen 2005), keeps track of the elements that are discarded from each beam at each depth. Whenever a node is discarded, we make a note of it. When we have exhausted all of the nodes at a certain depth, backtracking begins. When backtracking to a layer, we see if any nodes were discarded. If no nodes were discarded from the beam, we continue backtracking. If some nodes were discarded, we regenerate the beam by re-expanding all of the nodes in the previous beam. This time, rather than only holding on to the best nodes, we hold on to the best nodes that are at least as bad as the best previously discarded node. When repopulating the beam, we still keep track of the best node that is discarded. Eventually, we will exhaust all beams right up to the root layer, at which point we know that the search has returned an optimal solution.

BULB (Furcy and Koenig 2005) is a blending of limited discrepancy search (Korf 1996) and beam search that aims

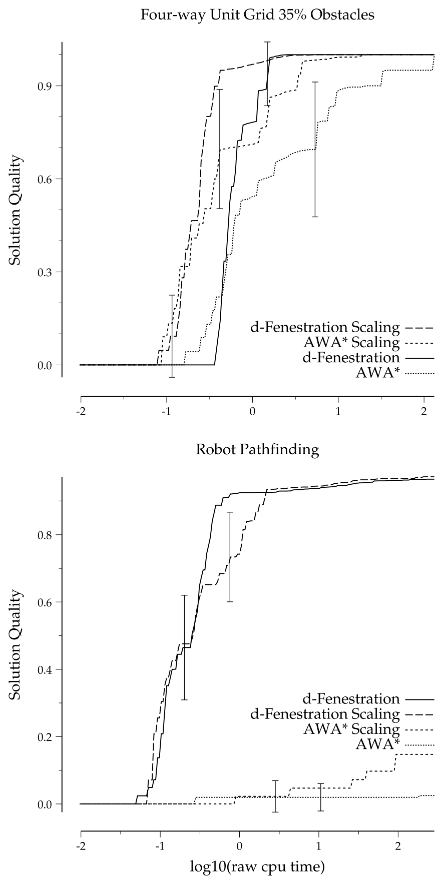


Figure 4: Use distance instead of depth in window A*

to correct the incompleteness of beam search. Limited discrepancy search is a tree based search where we search from the start of the search space towards the leaves but limit the number of times we can choose a node not recommended by the heuristic. Initially, limited discrepancy search will proceed greedily towards a goal, but as the allowed number of discrepancies increases, more of the space is explored until eventually the entire space is considered. It can also be extended to graph search. Rather than maintaining f_{A*}-boundaries as beam stack search, BULB increases the number of discrepancies allowed during an iteration, and eventually it will exhaust the search space. Beam stack search consistently outperformed BULB in our evaluation.

Window A*

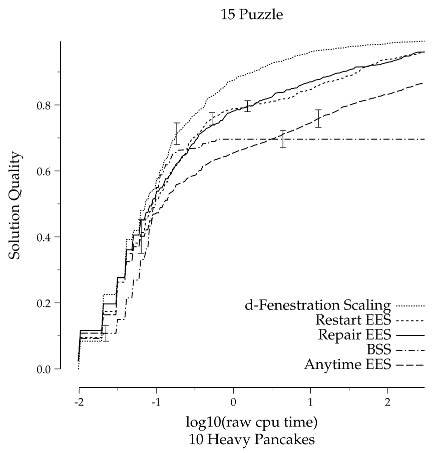
Anytime window A* (Aine, Chakrabarti, and Kumal 2007) is an extension of window A* where window A* is run with iteratively increasing window sizes. Window A* is an incomplete search where A* is run on a sliding window of nodes in the search space, instead of on an open list consisting of every node ever generated but not yet expanded. Restricting the comparisons between nodes to nodes a similar distance away from the root makes the comparisons fairer while searching on a restricted set of nodes typically improves the speed with which we can find solutions.

d-Fenestration When we say that window A* assumes nodes at a similar depth are similarly informed, what we mean is that it assume their heuristics are similarly accurate. Large heuristics belong to nodes that are very far away from the goal, and therefore seem more likely to be inaccurate than nodes with small heuristic values. It has been previously noted that the depth of a node does not directly translate into the distance of that node from a goal, even in best first search (Thayer and Ruml 2009). We use an estimate of distance to goal, d, to form the window of window A* rather than the node depth, a technique we call d-Fenestration. Using d instead of depth requires a minor change to the algorithm. Unlike depth, which grows over the course of a search, d should decrease as new nodes are generated. This may not always be true since d is a heuristic estimate of the distance to a goal for most of the domains in our evaluation. We are interested in the smallest d that the search has ever seen rather than the largest depth. This changes how we determine if a node is within the current window. Nodes are within the window if they have d values that are up to the window size larger than the smallest d we've ever seen, as opposed to up to the window size shallower than the deepest node we have ever seen.

Scaling Windows Selecting an appropriate window size for the iterations of anytime window A* is key in obtaining reasonable performance. For some domains, such as the knapsack problem, window A* is guaranteed to find a solution for any window size. All nodes have solutions beneath them, so it is impossible for the window to only contain nodes with no solution beneath them. There are also no cycles in the standard encoding, so it is impossible for the algorithm to see nodes it has already generated via a better path, meaning the window can never be exhausted. When these properties do not hold there are many window sizes that find no solution. Typically these are smaller windows, so the question of how to grow the window to the appropriate size naturally arises.

To solve this problem, we grow the window rapidly so long as no solution is found, and become more cautious in growing the window as solutions begin to stream in. We maintain two values, a window step size and a current window size, both initialized to 1. At every iteration, we add the window step size to the current window size to produce a new window. In every iteration where no solution is found, the window step size increases by one, but if we do find a solution, the step size is set back to one. So long as no solution is found, the size of the window continues to grow rapidly until the first solution is encountered. Then, we back off and increase the window size slowly until the solution stream dries up. We also considered using a geometric progression for window step size, but found this was too aggressive.

The top panel of Figure 4 compares the performance of the old and new approaches to window search on pathfinding in a gridworld. The x-axis represents the time used by the algorithms in log10 scale. The y-axis represents solution quality, where the quality of a solution is calculated by dividing the cost of the best solution found across all algorithms in that plot by the cost of the solution returned by the algorithm. This was the same performance metric used in the most recent IPC. We show 95% confidence intervals



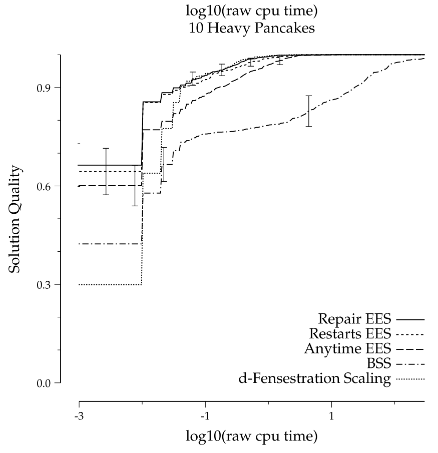


Figure 5: Performance on permutation puzzles

on the quality of the solution. d-fenestration significantly improves the speed at which the algorithm converges. In other domains, such as dynamic robot pathfinding, shown in the lower panel, the difference is not convergence, but if instances can be solved.

Comparison of Anytime Algorithms

We now present an empirical evaluation of the anytime algorithms across a set of common benchmark domains. All algorithms were implemented in Objective Caml and compiled to native binaries on 64-bit Intel Linux systems. Algorithms with initial suboptimality bounds were sampled at weights of 5, 3, and 2 while beam searches were sampled with beam widths of 500, 100, and 50. We show only the best parameter setting in the results below. For clarity, we reduce the number of algorithms shown in each plot. We restrict ourselves to the strongest window search, d-fenestration with scaling, the strongest beam search, beam stack search, and the most robust bounded suboptimal search across the three frameworks, EES. In this case, we are measuring robustness in terms of average performance across all of the benchmarks we considered. As we saw in Figure 2, no bounded suboptimal algorithm outperforms EES, with the exception of a single setting of A_ϵ^*.

Sliding Tile Puzzle We examined algorithm performance on the 15-puzzle using the instances from Korf (1985), using Manhattan distance to guide search. Figure 5 shows the results for this domain. We can see that d-fenestrated window A* is the best algorithm, followed by restarting and repairing anytime algorithms. The window-based approach performs very well in this domain because of the structure of the problem. There are few cycles and no dead-ends in tiles, so it is rare that algorithms based on window A* will fail to find a solution in a given iteration. Restarting and repairing are comparable for two reasons: First, node generation is very cheap in this problem, so the extra effort of restarting search doesn't add up very quickly, and second, solutions are shallow, so there is not much repeated work.

Pancakes Following (Thayer and Ruml 2010), we also performed experiments on another permutation problem, the 10 Heavy Pancake puzzle. Like the original pancake puzzle, the goal is to arrange a permutation of numbers from 1 to N into an ascending sequence. Each pancake has a weight, equal to its index. The cost of a move is the sum of the indexes of pancakes being flipped.

Figure 5 shows the results for the heavy pancake puzzle. Initially there is a large difference between the quality of solutions returned by d-fenestration and those returned by the bounded suboptimal based algorithms, a result of overcommitment by the window A* based algorithm. A best first algorithm will be able to go back and reconsider all previous decisions at any point, while window based approaches must wait until the end of an iteration before being allowed to reconsider a node outside of the window's scope. Beyond this large early gap, the algorithms perform quite similarly, except for beam stack search, which lags behind.

Grid Worlds Following (Thayer and Ruml 2010) we tested on grid pathfinding problems using the "life" cost function as well as the standard unit cost variant. We show results over 20 instances of 2000 by 1200 grids, allowing for movement in each of the cardinal directions. The grids were generated by blocking 35% of the cells randomly. The start is in the lower left, and the goal is in the lower right.

In the leftmost panel of Figure 6 we see that the algorithms separate into roughly two classes, those that eventually converge to an optimal solution, and those that failed to solve the problem optimally within the time limit. Both of the algorithms that converge to optimal reconsider their initial parameter settings. We cannot simply select better initial parameters. Altering the parameter presumably means lowering it, increasing the time taken to find an initial solution and harming the performance of the algorithms. Additionally, the question of how to best set parameters for anytime algorithms is still open. For the unit cost problems, shown in the center panel, the algorithms are still separable into two groups: continued search and everyone else. Continued search is having difficulty handling the large number of duplicate nodes in this space.

Dynamic Robot Navigation This domain follows that used by Likhachev, Gordon, and Thrun (2003). The goal is to find the fastest path from the starting location of the robot to some goal location and heading by altering the heading and velocity of the robot. We perform this search in worlds that are 500 by 500 cells in size. To add challenge to these problems, we scatter 75 lines with random orientations across the domain. Each line is up to 70 cells in length.

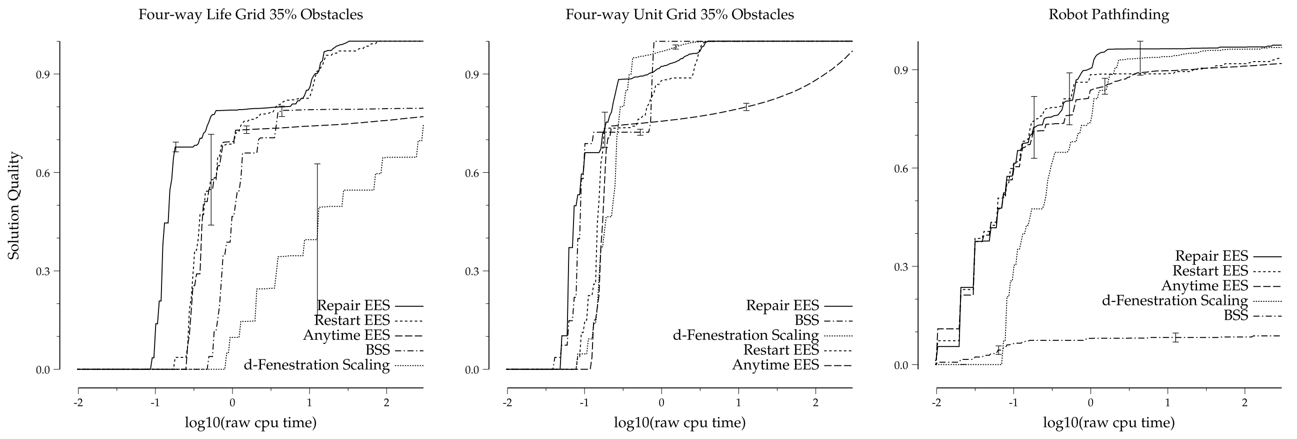


Figure 6: Performance in pathfinding

The dynamic robot navigation problem is fraught with dead ends, and the performance of d-fenestration and beam stack search suffer as we see in the rightmost panel of Figure 6. d-fenestration takes much longer than the other algorithms to find an initial solution, while beam stack search never converges. Despite this initial handicap, d-fenestration does eventually find some of the best solutions.

Traveling Salesman Following Pearl and Kim (1982) we test on the traveling salesman problem. Each node represents a partial tour with each action representing the choice of which city to visit next. We used the minimum spanning tree heuristic for h and the number of cities remaining for d.

We see that d-fenestration performs relatively well in the leftmost and center panels of Figure 7, taking longer to find the first solution, but converging very quickly, finding better solutions than other approaches in the Unit square variant. This is because window A* algorithms lack any form of heuristic focus and the greedy behavior that comes with it. They are still expanding nodes in A* order, they just happen to be working on a subset of the nodes in the space at any given time. As a result, d-fenestration takes much longer to find its first solution than the other algorithms. Once it does, it converges quickly. This suggests that we should look at the expected cutoff time for anytime algorithms as well as the domain properties when selecting which algorithm to use on a particular problem.

Vacuum World In this domain, inspired by the first state space depicted in Russell and Norvig (2010), a robot is charged with cleaning up a grid world. Movement is in the cardinal directions, and when the robot is on top of a pile of dirt, it may remove it. Cleaning and movement have unit cost. We use the minimum spanning tree of the robot and dirt locations plus the number of piles of dirt as an admissible h. Search distance is estimated by finding the length of a greedy solution on a board with no obstacles. We considered making h and d equivalent, but found that this produced poor performance for algorithms which relied on d. We used instances that are 500 cells tall by 500 cells wide, each cell having a 35% probability of being blocked. We place twenty piles of dirt randomly in unblocked cells and ensure that the problem can be solved. We show 95% confidence intervals averaged over 100 instances.

The right panel of Figure 7 shows the results in this domain. The algorithms separate into two groups, those based around best first search and those that are not. The vacuum problem has many cycles and inconsistent heuristics, which window A* and beam based algorithms both find problematic. The continued, repairing, and restarting frameworks have been serendipitously designed to work with inconsistent heuristics. These frameworks have traditionally used weighted A* as the underlying algorithm. Weighted A* works by weighting an admissible, and typically consistent, heuristic. Weighting a consistent heuristic will almost always produce an inconsistent one, and as a result, most evaluations focus on the use of inconsistent information. The beam and window based algorithms have seen less evaluation on domains with inconsistent heuristics, and perform poorly as a result.

Discussion

Anytime searches built from the previously proposed frameworks were always better than the next best approach, d-fenestration, except for the sliding tiles problem, where their performance was comparable. d-fenestration consistently outperforms the original implementation of Anytime Window A* as well as all beam based approaches that we evaluated. Best-first based anytime searches, especially in the repairing framework, are the best choice for anytime search for the wide variety of domains considered in our evaluation.

Restarting was not the strongest performing framework in our evaluation. Only one of the domains in our evaluation exhibit a strong low-h-bias, limiting the effectiveness of the restarting approach in this evaluation. The results we present use EES within each of the frameworks. EES has no low-h-bias to correct for; given a large suboptimality bound, it will attempt to greedily complete the shortest solution. In the restarting framework, EES gets all of the additional work of restarting, but none of the benefits of avoiding low-h-bias.

Our results make the continued framework appear more efficient than it truly is. This is the result of using EES as a base. EES tunes a set of parameters constantly during the search, making reconsidering the initial parameter settings

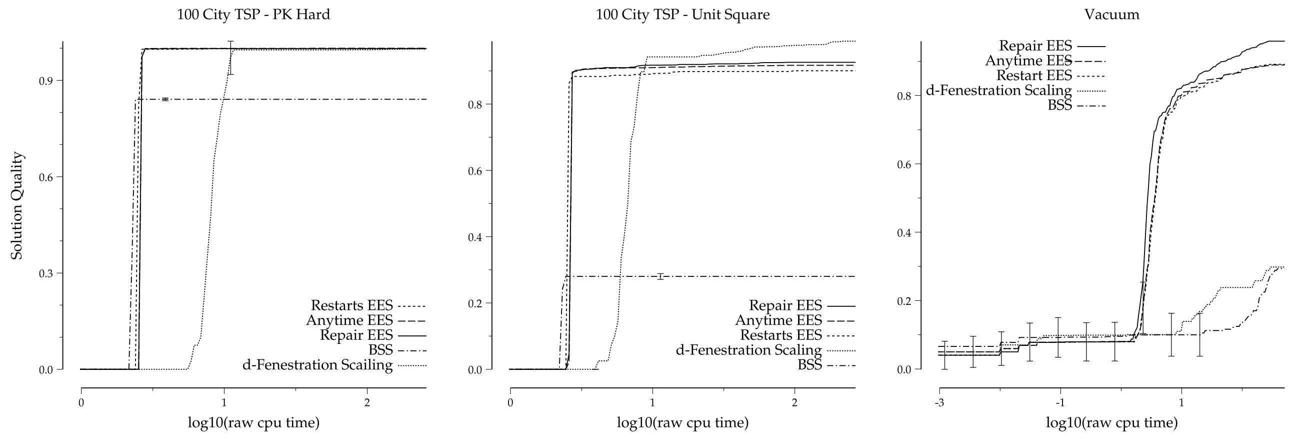


Figure 7: Performance on the traveling salesman and vacuum problems

less important. If we had shown continued A^*_ϵ instead, then continued search would not have appeared to be competitive with either restarting or repairing search.

Conclusions

We presented an extensive analysis of three frameworks for converting bounded suboptimal heuristic search algorithms into anytime algorithms across a wide variety of benchmark domains. We discovered that framework has strong influence on the performance of an anytime algorithm, and that repairing is the strongest framework in general. We introduced two improvements to anytime window A* allowing it to solve problems it previously could not, and improving its convergence behavior. When compared to other state of the art algorithms, including anytime beam search and our improved anytime window A* variant, repairing searches performed best. This is a result of its ability to gracefully handle duplicates and its robustness in the face of domains with dead end nodes.

Acknowledgments

We gratefully acknowledge support from both the NSF grant IIS-0812141 and the DARPA CSSG program (grant HR0011-09-1-0021).

References

Aine, S.; Chakrabarti, P.; and Kumal, R. 2007. AWA* - a window constrained anytime heuristic search algorithm. In *Proceedings of IJCAI-07*.

Boddy, M. S., and Dean, T. 1989. Solving time-dependent planning problems. In *Proceedings of IJCAI-89*, 979–984.

Ebendt, R., and Drechsler, R. 2009. Weighted A* search - unifying view and application. *Artificial Intelligence* 173:1310–1342.

Furcy, D., and Koenig, S. 2005. Limited discrepancy beam search. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 125–131.

Ghallab, M., and Allard, D. 1983. A_ϵ: An efficient near admissible heuristic search algorithm. In *Proceedings 8th IJCAI*.

Hansen, E. A., and Zhou, R. 2007. Anytime heuristic search. *Journal of Artificial Intelligence Research* 28:267–297.

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

Korf, R. E. 1985. Iterative-deepening-A*: An optimal admissible tree search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-85)*, 1034–1036.

Korf, R. E. 1996. Improved limited discrepancy search. In *Proceedings of AAAI-96*, 286–291. MIT Press.

Likhachev, M.; Gordon, G.; and Thrun, S. 2003. ARA*: Anytime A* with provable bounds on sub-optimality. In *Proceedings of the Seventeenth Annual Conference on Neural Information Porcessing Systems (NIPS-03)*.

Pearl, J., and Kim, J. H. 1982. Studies in semi-admissible heuristics. *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-4(4):391–399.

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

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

Reese, B. 1999. AlphA*: An ϵ-admissible heuristic search algorithm. Unpublished, retrieved from http://home1.stofanet.dk/breese/papers.html.

Richter, S.; Thayer, J.; and Ruml, W. 2009. The joy of forgetting: Faster anytime search via restarting. In *Symposium on Combinatorial Search*.

Russell, S., and Norvig, P. 2010. *Artificial Intelligence: A Modern Approach*. Third edition.

Sutton, R. S. 1988. Learning to predict by the methods of temporal differences. *Machine Learning* 3:9–44.

Thayer, J., and Ruml, W. 2009. Using distance estimates in heuristic search. In *Proceedings of ICAPS-2009*.

Thayer, J., and Ruml, W. 2010. Finding acceptable solutions faster using inadmissible information. Technical Report 10-01, University of New Hampshire.

Thayer, J. T.; Ruml, W.; and Bitton, E. 2008. Fast and loose in bounded suboptimal heuristic search. In *Proceedings of the First International Symposium on Search Techniques in Artificial Intelligence and Robotics (STAIR-08)*.

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