Using Distance Estimates in Heuristic Search: A Re-evaluation

Jordan T. Thayer and **Wheeler Ruml** and **Jeff Kreis**
Department of Computer Science
University of New Hampshire
Durham, NH 03824 USA
jtd7, ruml, jhg22 at cs.unh.edu

Abstract

Traditionally, heuristic search algorithms have relied on an estimate of the cost-to-go to speed up problem-solving. In many domains, operators have different costs and estimated cost-to-go is not the same as estimated search-distance-to-go. We investigate further accelerating search by using a distance-to-go function. We evaluate two previous proposals: Dynamically weighted A* and A$^*_\epsilon$. We present a revision to dynamically weighted A* which improves its performance substantially in domains where solutions are not at fixed depths. We show how to incorporate search distance to go estimates into weighted A* in order to improve its performance in pathfinding problems. We present a proof showing that weighted A* can ignore duplicate states, leading to large improvements in performance for pathfinding problems.

Introduction

Heuristic search is used to solve a wide variety of problems. If sufficient resources are available, optimal solutions can be found using A* search with an admissible heuristic (Hart, Nilsson, and Raphael 1968). In practical settings we are willing to accept suboptimal solutions in order to reduce the computation required to find an answer. We consider the setting in which one wants the fastest possible search where the solution is guaranteed to be within a bounded factor of the optimal solution. For a given factor w, we call an algorithm w-admissible if it is guaranteed to return a solution that is no more than w times more expensive than an optimal solution.

The purpose of bounded suboptimal search is not to find a solution whose cost is within a given bound of the optimal, but rather, given a desired quality bound produce an acceptable solution as quickly as possible. The hope is that as requirements on solution quality are relaxed, the algorithm will find solutions faster. Many bounded suboptimal algorithms behave more like a greedy search on a cost-to-go heuristic as the bound is loosened. When the heuristics is reasonably accurate, greedy searches find solutions of reasonable but unbounded quality in few expansions. Bounded suboptimal algorithms attempt to make a controlled transition between performing like A* when the bound is near optimal, and greedy search where the bound is lax.

In some benchmark domains all actions have the same cost, and so becoming greedy with respect to the cost of a solution is equivalent to becoming greedy with respect to the length of a solution. This is not always the case. Searches that consider both the estimated cost-to-go, given by a heuristic evaluation function h, as well as the search-distance-to-go, given by the evaluation function d, can perform better in domains where h and d differ. In these domains, a search that is greedy with respect to solution cost may take longer than a search that focuses on finding the nearest solution, especially when the suboptimality bound is generous. Figure 1 demonstrates the extreme case, where no bound is placed on the quality of the returned solution. We see that not only do we find solutions faster when greedily searching on d than when we do on h, but that the performance gap between these two approaches increases with problem size. Although it may not be obvious at first glance, domains where h and d differ are numerous and include temporal planning, the traveling salesman problem, and many variations of pathfinding.

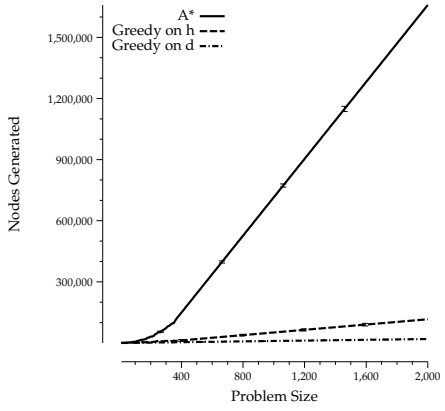

Figure 1: Greedy Search on d vs. Greedy Search on h

We evaluate two previous approaches to incorporating search-distance-to-go information into searches, A$^*_\epsilon$ and dynamically weighted A*, and find that neither algorithm performs particularly well. We show how to improve the performance of dynamically weighted A* with a small revision to the algorithm. We extend weighted A* to include distance to go information and show that this extension can significantly improve the performance of the algorithm.

A^*_ϵ

A^*_ϵ (Pearl and Kim 1982) considers the distance of a node from a goal when deciding which to expand next. It prefers to expand nodes that are as close to a solution as possible while still guaranteeing w-admissibility. To do this, it maintains two ordered lists. The first is identical to that used by A^*, where nodes are ordered according to the cost function $f_{A^*}(n) = g(n) + h(n)$. g is the cost of travelling to a node from the root of the search, and h in the estimated cost to go. All nodes are sorted in this fashion, forming the open list. At the front of the open list is the node with minimum f_{A^*}, f_{min}. In order to select nodes which are close to a goal, A^*_ϵ must maintain a list of nodes sorted on d, called the focal list. The node at the front of focal is the node with minimum d, d_{min}. d_{min} is the node which appears to be closest to the goal. To find a solution, it is expanded, and its children are placed into open until d_{min} is a solution.

To ensure that the solution returned by A^*_ϵ is w-admissible, A^*_ϵ only places those nodes that have f_{A^*} values that are within a factor w of f_{min} onto the focal list. This means that $f_{A^*}(d_{min}) \leq w \cdot f_{A^*}(f_{min})$, given by the construction of the focal list. Since we are using an admissible heuristic h, we know that $f_{A^*}(f_min) \leq f_{A^*}(opt)$ where opt is a solution with optimal cost. Together, we can infer that $f_{A^*}(d_{min}) \leq w \cdot f_{A^*}(opt)$. This property holds for every d_{min}, and d_{min} is the only node ever expanded. Eventually it will be the solution , if one exists, and the quality will be bounded.

An Efficient Implementation

Open and focal are separate lists, at least conceptually. There are several ways in which to build them, but an inefficient implementation will harm the performance of the algorithm. One might consider only maintaining the open list, and iterating through the first handful of nodes on every expansion to select d_{min}. Unfortunately when the bound is loose such an algorithm would be examining every node in open at every expansion. Alternatively, we might keep both open and a list of all nodes ordered on d in memory, iterating back on this d-list until a node near enough to f_{min} is discovered, but again, this is inefficient.

To make A^*_ϵ a practical algorithm, we use a more sophisticated data structure. Nodes in the open list are stored in a balanced binary tree totally ordered by f. In our implementation, we used a red-black tree following Cormen et al., 2001. The subset of nodes within ϵ of the node with minimum f is also stored in a heap ordered on d. We used a binary heap stored in an array, following Sedgewick, 1992. This this arrangement, it takes constant time to identify the node to expand, logarithmic time to remove it from the heap and tree, and logarithmic time to insert each child resulting from the expansion. However, if the node with minimum f changes, then nodes may need to be added or removed from the heap. (All nodes are stored in the tree.) While it is easy to find the nodes whose f values wall between w times the old minimum f and w times the new one (because the tree is ordered on f), there might be many such nodes that need to be added or removed from the heap. Removal is easy because we maintain, in each node, its index in the heap array.

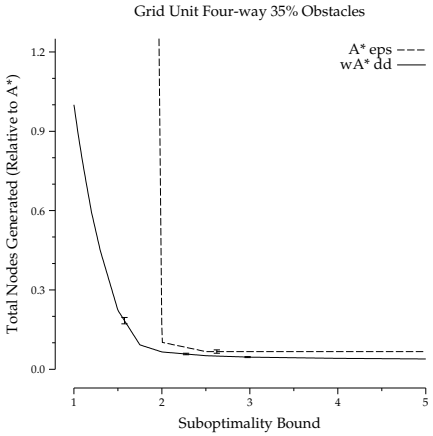

Figure 2: A^*_ϵ has Poor Performance

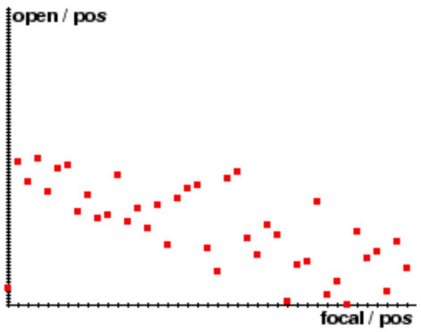

Figure 3: Open and Focal Interact Destructively

Using this more sophisticated data structure speeds up A^*_ϵ in practice by an enormous factor which increases as problem become more difficult.

Flaws

Even with efficient data structures, the performance of A^*_ϵ leaves something to be desired. It also works well for loose bounds, where the algorithm searches greedily on d, but fails to find solutions otherwise.

The reason A^*_ϵ performs poorly in practice is easy to understand. When using an admissible h function, the f values of nodes cannot decrease, and typically increase, as one descends from the root. Along a path to a goal, d will usually decrease. Thus, nodes with low d will often have relatively high f values. An illustration of this during grid pathfinding can be seen in Figure 3. The scatter plot shows, for each node in A^*_ϵ's focal list, its position in both the focal and open lists. There is a clear trend from low d and high h to high d and low f. This creates a phenomenon in which almost all nodes on focal are expanded in sequence, with none of their children making it on to focal, until at last the node with minimum f is expanded and focal is refilled. During each of these phases, little progress is made toward the goal.

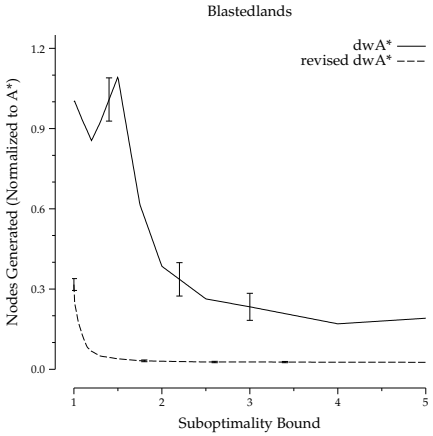

Figure 4: Improving Upon Dynamically Weighted A*

Dynamically Weighted A*

Dynamically weighted A* searches nodes in a best first order as determined by the node evaluation function f_{dwA^*}. It applies a weighting factor w to the heuristic evaluation function h to encourage the search to behave greedily. Further it decreases the size of this factor as the search progresses. More concretely, let D_n be the depth of a node and D_{goal} be the depth of solutions, giving $f_{dwA^*} = g(n) + w \cdot (1 - \frac{D_n}{D_{goal}}) \cdot h(n)$. This reduces the weight applied to the cost-to-go estimate as the search progresses. The depth of goals isn't always known a priori, and in these situations, D_{goal} can be approximated by looking using the search-distance-to-go heuristic d at the root.

Dynamically weighted A* prefers expanding nodes which are nearer to a solution for two reasons. First, the weighting makes it prefer nodes which have lower h values, and these nodes tend to be closer to a solution. Second, dynamically weighted A* explicitly rewards progress. By decreasing the weights used as the nodes become deeper, nodes deep in the search space become far more attractive than those at the root.

Revising Dynamically Weighted A*

Although dynamically weighted A* works well as it was originally proposed in domains where solutions are at a fixed, known depth, it performs poorly in domains without a fixed depth, as figure 4 shows. Decreasing the weight by which the heuristic evaluation function is multiplied as depth increase encourages progress in any direction. Changing the node evaluation function to $f'_{dwA^*} = g(n) + max(1, w\frac{d(n)}{d(root)}) \cdot h(n)$ rewards progress towards a solution. This revision makes the algorithm prefer nodes with low d, which are exactly the nodes that appear to be close to a solution. This alteration, denoted revised dwA* in the plots, dramatically improves the performance of the algorithms.

It should be noted that in figure 4, revised dynamically weighted A* is generating fewer nodes than A* does when finding the optimal solution. Dechter and Pearl (1988) show that among all algorithms with access to the same h function, no algorithm can expand fewer nodes than A*; however, this result does not hold when considering additional sources of information such as d. In this case, simply breaking ties on d allowed revised dynamically weighted A* to solve the problem in a third of the time A* using just h would have taken.

Incorporating d into Weighted A*

Weighted A* is an elegant solution to bounded suboptimal search where traditional node evaluation function of A*, $f_{A^*} = g(n) + h(n)$ is changed to increase the cost to go by a factor w, as in $f_{wA^*} = g(n) + w \cdot h(n)$. This emphasis on the heuristic evaluation function causes weighted A* to behave more like a greedy search on h. While this is an effective strategy where heuristics are accurate and h and d are identical, figure 1 shows that better performance can be obtained by focusing solely on d when bounds are loose.

One of the easiest, and effective, ways to incorporate this information into the search is simply to use d to break ties. When two nodes have identical f_{wA^*} values, order them based on d, the search-distance-to-go estimate. Where h and d are identical, this will have no impact on the search order. When they are different, in can significantly speed up the algorithm, as shown in figure 6.

Ignoring Duplicate States

In pathfinding as well as other domains, it is possible to reach the same problem state by multiple paths. While not affecting the completeness or suboptimality guarantees of search algorithms, these redundant paths can significantly impact performance. Likhachev, Gordon, and Thrun (2003) showed that ignoring search states that have already been expanded can significantly improve the performance of their algorithm, Anytime Repairing A*, without losing its suboptimality guarantee.

We prove an analogous result that applies to weighted A* search where $f(n) = g(n) + w \cdot h(n)$. We show that it can drop duplicate nodes that have already been expanded and still return a w-admissible solution. We will rely on the consistency (and thus admissibility) of h, which states that for any pair of nodes a and b, $h(a) \leq c^*(a, b) + h(b)$ (Pearl 1984). Our proof is quite different than that of Likhachev, Gordon, and Thrun (2003). First, note that there always exists an optimal solution path.

Theorem 1 *There always exists a node p along the optimal solution path that is on open and has $g(p) \leq w \cdot g^*(p)$.*

Proof: The proof is by induction over iterations of search. In the base case, consider the first expansion, that of the root. One of its children is clearly along the optimal path and has its optimal g value. For the induction step, assume there is a w-admissible state p_{i-1} on the path and consider its fate during an iteration of search. It can only be removed from open by being expanded. If its child p_i that lies along the optimal solution path is inserted in open, $g(p_i) = g(p_{i-1}) + c^*(p_{i-1}, p_i) \leq w \cdot g^*(p_{i-1}) + c^*(p_{i-1}, p_i) \leq w \cdot g^*(p_i)$ and the theorem holds. If p_i is discarded, it can only be because

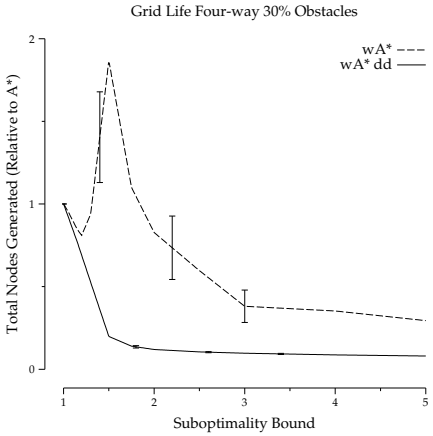

Figure 5: Ignoring Duplicates Helps

it is already on closed after having been generated along another path and subsequently expanded. If p_i is expanded before whichever w-admissible ancestor p_{i-j} was on the open list at that time, this means that $f'(p_i) \leq f'(p_{i-j})$. But then:

$$\begin{aligned} f'(p_i) &\leq f'(p_{i-j}) \\ g(p_i) + wh(p_i) &\leq g(p_{i-j}) + wh(p_{i-j}) \\ &\text{by consistency of } h\text{:} \\ &\leq g(p_{i-j}) + w(c^*(p_{i-j}, p_i) + h(p_i)) \\ &\leq g(p_{i-j}) + wc^*(p_{i-j}, p_i) + wh(p_i) \\ &\text{due to the optimal path:} \\ &\leq w(g(p_{i-j}) + c^*(p_{i-j}, p_i)) + wh(p_i) \\ &\leq wg^*(p_i) + wh(p_i) \\ g(p_i) &\leq wg^*(p_i) \end{aligned}$$

□

The remaining steps are analogous to the proof of weighted A*'s admissibility (Pearl 1984).

Theorem 2 *The solution s returned has $g(s) \leq w \cdot g^*(\text{opt})$.*

Proof: If s is selected from open before p, then:

$$\begin{aligned} g(s) = f'(s) &\leq f'(p) \leq g(p) + w \cdot h(p) \\ &\leq w \cdot g^*(p) + w \cdot h(p) \\ &\leq w \cdot (g^*(p) + h(p)) \\ &\leq w \cdot g^*(opt) \text{ by admissibility of } h \end{aligned}$$

□

The ability to drop duplicates significantly improves the algorithms performance for large grid worlds. Figure 5 shows the difference on one type of pathfinding instance, though similar savings are present in all pathfinding problems of significant size.

Performance

To gain a better understanding of the performance of algorithm using a search-distance-to-go heuristic we tested them on several challenging benchmark problems including grid-world pathfinding, the travelling salesman problem, and the sliding tile puzzle. All algorithms were implemented in Objective Caml, compiled to 64-bit native code executables, and run on a collection of Intel Linux systems. We compare the following algorithms:

weighted A* (wA*) using the desired suboptimality bound as a weight. Weighted A* ignores nodes that are already in the closed list. We found that this improves performance dramatically without sacrificing admissibility (although see Hansen and Zhou, 2007, for another view). Such an approach is only possible with a consistent heuristic. When this feature is on, the algorithm is labeled wA*, dd.

weighted A* with d tie-breaking (wA*, d-tie) using the desired suboptimality bound as a weight. When ignoring duplicate nodes the algorithm is labeled wA*, d-tie, dd.

revised dynamically weighted A* (revised dwA*) Dynamically weighted A*, revised to use search-distance-to-go estimates instead of d to decrease the weight used as search progresses. The proof of admissibility for duplicate dropping relies on the fact that w never changes. As such, it does not hold for dynamically weighted A* or its revised implementation.

We sampled all the algorithms at the following suboptimality bounds: 1, 1.005, 1.001, 1.01, 1.05, 1.1, 1.15, 1.2, 1.3, 1.5, 1.75, 2, 2.5 and 3. All of the plots follow the same layout. The suboptimality bound of the algorithms is on the x-axis. A suboptimality bound of 1 means that the algorithms produced an optimal solution, and a bound of 3 means that the solution returned has cost within a factor of 3 of the optimal solution. The y-axis shows the total amount of CPU time consumed, normalized across machines, and is normalized to the CPU time consumed by A*.

Grid-world Pathfinding

We tested on a variety of grid world finding problems. We ran on boards with uniformly blocked cells, boards where the obstacles took the form of straight lines laid across the board, and several boards from a popular real time strategy game. We show 95% confidence intervals, averaged over 100 instances for game boards, and 20 instances otherwise.

Uniform Distribution We consider 12 varieties of simple path planning problems on a 2000 by 1200 grid, using four-way or eight-way movement, three different probabilities of blocked cells, and two different cost functions. The start state is in the lower left corner and the goal state is in the lower right corner. In addition to the standard unit cost function we tested on 'life', a graduated cost function in which moves along the upper row are free and the cost goes up by one for each lower row. In eight-way movement, diagonal moves cost $\sqrt{2}$ times as much as cardinal directions. Under both cost functions, simple analytical lower bounds are available for the cost h and search-distance d to the cheapest goal. Our results are averaged over 20 instances.

Figure 6 shows the results. In 4-way worlds, where h and d are identical, the performance of weighted A* and weighted A* with tie-breaking on d are nearly identical. Revised dynamically weighted A* does not perform as well as these two algorithms due to its inability to drop duplicates. In 8-way movement, breaking ties on d improves the performance of weighted A* dramatically for tight bounds, but

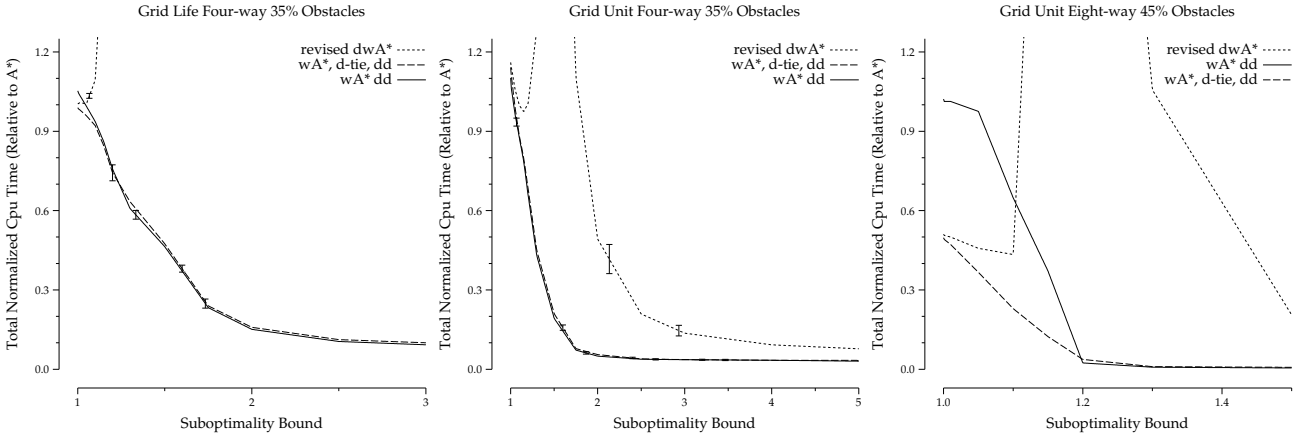

Figure 6: Performance on Grid-world Pathfinding with Uniform Obstacles

it does not improve the greedy behavior of the algorithm. Despite the significant difference in h and d in life boards, weighted A* with tie breaking on d doesn't significantly out perform weighted A*. This is because there are very few nodes with identical values in this domain because of the graduated cost function.

Lines We considered 6 varieties of path planning problems in 2000 by 1200 grids which use lines for obstacles instead of uniform obstacle distribution. The lines used were of lengths ranging between 5% and 25% of the board's diagonal. We ran on boards with 25, 50, and 75 such lines scattered across them. The starting state was in the middle of the left-hand side, and the goal state was in the middle of the right hand side. The lines introduce error into the heuristic functions, but this error is non-uniform.

Figure 7 shows us that in the 4-way movement boards we suffer slightly for breaking ties on d. There is no effect on the node ordering, as ties on f_{wA^*} are being broken on low h, which is identical to the d tie-breaking rule for this movement model and domain. It is because of the cost of calculating two heuristics for every node. Were we to just replicate h and call it d, the algorithms would have identical performance, but this would obscure the overhead of calculating two heuristics. On the 8-way boards, breaking ties on d improves the performance for tight bounds and harms it slightly for higher bounds. Revised dynamically weighted A* failed to solve any of these problems, and is not pictured in the plots.

Game Boards Following Bulitko et al. (2007), we tested on several pathfinding problems from a popular real-time strategy game. The boards allowed for eight-way movement, and start and goal locations were selected at random. Instances were then further grouped by the length of their optimal solution. We show results for the most challenging instances we ran against, which have an optimal solution length somewhere between 160 and 190 steps.

These problems are significantly smaller than the other grid world problems which we experimented on. Although taking special care to efficiently handle duplicate paths

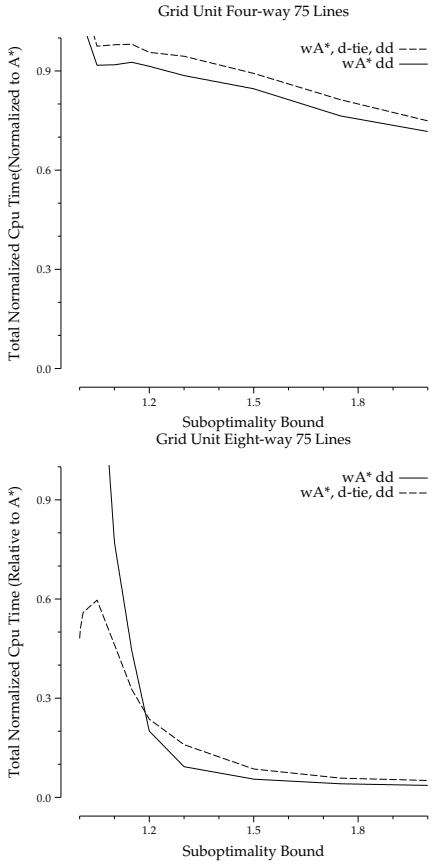

Figure 7: Performance on grid-world path-finding problems with lines

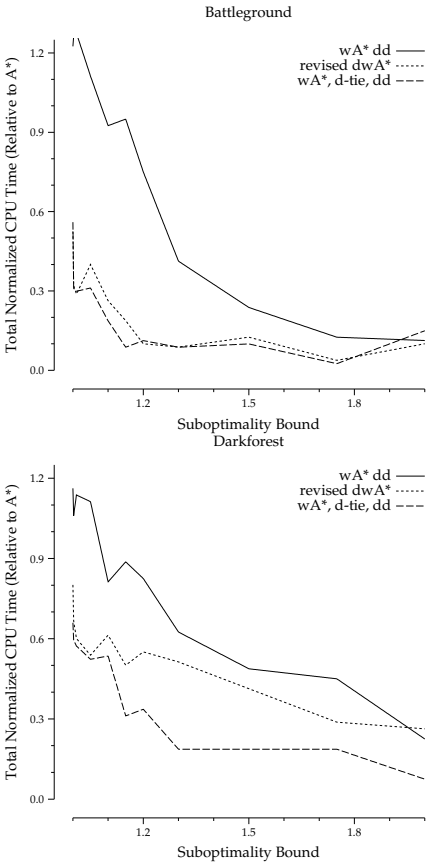

Figure 8: Performance on grid-world path-finding problems for games

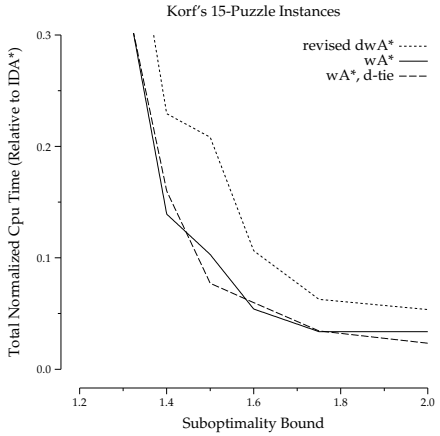

Figure 9: Performance on the 15-Puzzle

through the state is key for larger boards, here it isn't as important important, as evidenced by the performance of revised dynamically weighted A*, shown in figure 8. Even without duplicate dropping, it significantly outperforms weighted A*. When d tie-breaking is added, weighted A* with duplicate dropping performs better.

Sliding Tile Puzzle

We examined algorithm performance on the 15-puzzle, using the instances from the original paper on iterative deepening A* (Korf 1985). We use Manhattan distance for h and d. Since A* fails to solve these instances within a reasonable memory bound, we normalize our data against iterative deepening A*. All the algorithms we tested ran into the same memory limitation as A*, so we show results only for weights of 1.2 and above. We show results averaged over all 100 instances.

As d and h are identical in this domain, there is little difference in the performance of weighted A* with and without tie-breaking. Revised dynamically weighted A* performs surprisingly well in this domain, considering the number of duplicate nodes it must encounter during a search. Its performance is still significantly worse than that of either weighted A*.

Travelling Salesman Problem

Following Pearl and Kim (1982), we test on a straightforward encoding of 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 exact depth remaining in the tree for d. We test on two types of instances, 19 cities placed uniformly in a unit square and 12 cities with distance chosen uniformly at random between 0.75 and 1.25 ('Pearl and Kim Hard'). Both types of problems are symmetric. We average our results over 40 instances in this domain. The confidence intervals are so tight as to not be visible in our results.

Figure 10 shows the performance of the algorithms on this domain. For sake of comparison with the original re-

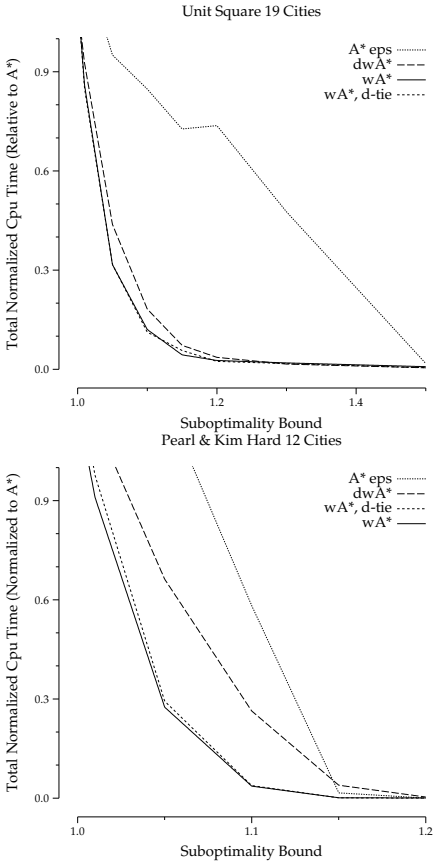

Figure 10: Performance on Travelling Salesman

sults, A^*_ϵ is also included in the plot. Both weighted A* and weighted A* with tie-breaking on d perform better than dynamically weighted A*, which is identical to the revised implementation for this domain. Tie-breaking is of little benefit to weighted A* in this domain, despite the fact that h and d are different. This is due to the fact that there are very few nodes with the same f_{wA^*} value, and so the tie breaking rule is rarely, if ever, used.

Discussion

The revision of dynamically weighted A* significantly improves its performance for domains where goals do not exist at a fixed depth. Unfortunately, this was not enough to make the algorithm competitive with the current state of the art. Its inability to discard duplicate states makes it a poor choice for large pathfinding problems. Hopefully it will demonstrate good performance on large problems without a large number of duplicate states.

Weighted A* with tie-breaking on d is only effective when there are a number of ties to be broken. This is obviously the case in grid-world pathfinding, given the domain and the performance of the algorithm in it. Whenever there are real valued edge costs, tie-breaking on d isn't going to help because there aren't going to be very many ties. The performance gains we saw from tie breaking should transfer to algorithms which rely on weighted A*, such as anytime heuristic search, anytime repairing A*, and optimistic search.

Conclusions

Including a distance-to-go-estimate d can significantly improve the performance of bounded suboptimal heuristic search. When used to guide dynamically weighted A*, it improves the performance of that algorithm considerably in domains without a fixed goal depth. Simply using it for tie-breaking can lead to large performance gains for weighted A*, causing it to find optimal solutions much faster than an algorithm only relying on h. When these improvements do not improve performance, they also do not seem to substantially harm it. We proved that weighted A* is allowed to ignore duplicate states, and that this significantly improves its performance.

Acknowledgments

We gratefully acknowledge support from NSF grant IIS-0812141.

Many thanks to Minh B. Do for the temporal planner used in these experiments and to Richard Korf for publishing his sliding tile instances.

References

Bulitko, V.; Sturtevant, N.; Lu, J.; and Yau, T. 2007. Graph abstraction in real-time heuristic search. *JAIR* 30:51–100.

Cormen, T.; Leiserson, C.; Rivest, R.; and Stein, C. 2001. *Introduction to Algorithms*. Cambridge, MA: MIT Press.

Dechter, R., and Pearl, J. 1988. The optimality of A*. In Kanal, L., and Kumar, V., eds., *Search in Artificial Intelligence*. Springer-Verlag. 166–199.

Hansen, E. A., and Zhou, R. 2007. Anytime heuristic search. *JAIR* 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 IJCAI-85*, 1034–1036.

Likhachev, M.; Gordon, G.; and Thrun, S. 2003. ARA*: Formal analysis. Technical Report CMU-CS-03-148, Carnegie Mellon University School of Computer Science.

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.

Pearl, J. 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Sedgewick, R. 1992. *Algorithms in C*. Boston, MA: Addison-Wesley Professional.