Selecting a Greedy Search Algorithm
Technical Report 10-07

Christopher Wilt, Jordan Thayer, and Wheeler Ruml
Department of Computer Science
University of New Hampshire
Durham, NH 03824 USA
wilt, jtd7, ruml at cs.unh.edu

June 6, 2011

Abstract

There are many algorithms that are capable of solving the shortest path problem. Given a specific shortest path problem, it is not clear which of the myriad algorithms should be used. Based upon an empirical evaluation of six benchmark domains, we create a decision tree that considers domain features and approximate time/memory budget constraints to decide which algorithm should be used given a domain with known attributes and given time/memory budget. The decision tree also helps identify open questions regarding what information is needed to predict how well a given algorithm will perform.

1 Introduction

This paper builds upon the work in Wilt et al. [2010] and constructs a decision tree that can be used to decide which algorithm should be used in a given situation. Wilt et al. [2010] consider the traveling salesman problem [Pearl and Kim, 1982], dynamic robot path finding [Likhachev et al., 2003], the sliding tile puzzle [Korf, 1985], a derivative of the pancake puzzle [Holte et al., 2005], and a vacuum-robot domain [Russell and Norvig, 2010].

Wilt et al. [2010] consider weighted A* [Pohl, 1970], A^*_ϵ [Pearl and Kim, 1982], Window A* [Aine et al., 2007], multi-state commitment k weighted A* [Furcy and Koenig, 2005b], greedy best-first search [Doran and Michie, 1966], enforced hill climbing [Hoffmann and Nebel, 2001], LSS-LRTA* [Koenig and Sun, 2008], beam search [Rich and Knight, 1991, Bisiani, 1992], beam-stack search [Zhou and Hansen, 2005], and BULB [Furcy and Koenig, 2005a]. They show that the performance varies across the different benchmark domains, but they also point out that a number of domain features are behind the observed variation. Among the relevant features are dead ends, cost function, and whether or not searches with an unbounded open list can find solutions without running out of memory.

These features are important predictors of which algorithm will be the strongest performer, but they are not the only features that matter. The decision tree created does is not perfect even for the test data, and explaining exactly why remains an open question.

1.1 Selecting the Best Algorithm By Domain

Wilt et al. [2010] show that across all of the domains under consideration, the most successful algorithms in aggregate are weighted A*, A_ϵ^*, beam search, and LSS-LRTA*. For this reason, we shall only consider those algorithms when evaluating algorithms across the various domains.

The subsequent plots all use the same measurements on both the x and y axis. On the x axis is the log of cpu time. On the y axis is an aggregate of solution quality and success rate. Finding the best solution earns a score of 1. Finding a solution that is lower quality earns a score proportional to the solution quality with higher quality solutions scoring closer to 1. Failing to find any solution earns a score of 0. All algorithms were terminated after five minutes. An algorithm could fail to find a solution because of inherent incompleteness or a result of timing out. All algorithms were run on Dell Optiplex 960 Core2 duo E8500 3.16 GHz machines with 8 Gb of RAM. The machines were running 64 bit Linux. On all domains, five minutes was not enough time to run out of memory, with the exception of 48 puzzle, where algorithms were able to exhaust memory in approximately 1 minute. If an algorithm ran out of memory, it was considered to have failed to find a solution.

The lines are constructed by varying the underlying parameter of an algorithm. For beam search, the parameter in question is the beam width. For LSS-LRTA*, the parameter is the look-ahead expansion budget. LSS-LRTA* and beam search were run with 50000, 10000, 5000, 1000, 500, 100, 50, 10, 5, and 3 for parameters. For weighted A* and A_ϵ^*, the parameter is the weight. A* and A_ϵ^* were run with parameters of 100000, 100, 50, 20, 15, 10, 7, 5, 4, 3, 2.5, 2, 1.75, 1.5, 1.3, 1.2, 1.15, 1.1, 1.05, 1.01, 1.001, 1.0005 and 1.

For example, in Figure 1, the green line representing weighted A* shows that weighted A* initially finds a low quality solution when using a high weight, and as the weight decreases, the solution quality increases. As the weight continues to shrink, weighted A* starts to fail to find solutions, so the aggregate solution quality decreases substantially.

1.2 Traveling Salesman Problem

The results of running the four algorithms in question on the traveling salesman problem can be seen in Figure 1. For finding solutions quickly, the algorithm of choice is A_ϵ^*, but if the highest quality solutions are desired, beam search is the algorithm of choice.

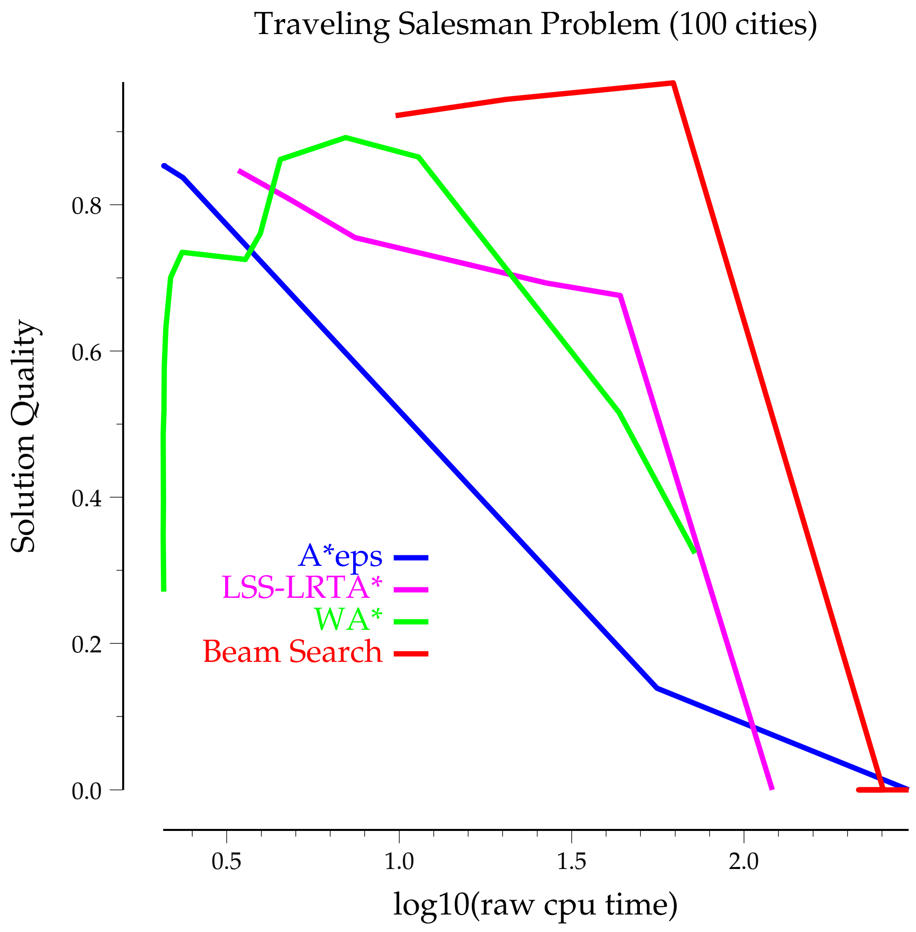

Figure 1: Algorithm performance on the traveling salesman problem

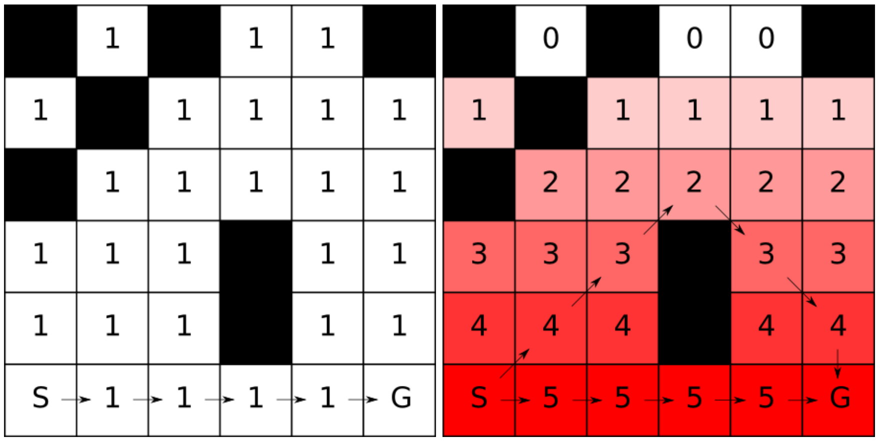

Figure 2: Example of grid path planning problems

1.3 Grid Path Planning

For the grid path planning domain, we considered two varieties of grid path planning problems. The first was basic grid path planning with unit cost, where each move costs the same amount. We also considered a derivative of grid path planning where the cost of a move is proportional to the cell's Y coordinate. In this situation, the shortest path and the best path are often not the same. Examples of each kind of problem can be seen in Figure 2.

The results of running the four algorithms in question on the grid path planning problem can be seen in Figure 3 and 4. In unit costs, weighted A* is better than all of the alternative algorithms, although beam searches converge on high quality solutions slightly faster.

In grid path planning with life costs, A^*_ϵ is slightly faster, although it has terrible scaling behavior as compared to weighted A*. In addition to that, the solution returned is of poor quality relative to that of weighted A*. Despite these drawbacks, A^*_ϵ does provide the fastest solutions. The reason A^*_ϵ is so fast stems from the fact A^*_ϵ expands nodes according to d when the weight is extraordinarily high.

1.4 Vacuum Robot Path Planning

The vacuum robot path planning domain was inspired by Russell and Norvig [2010]. In this domain, there is a robot that inhabits a grid world where there are three kinds of cells: dirty cells, clean cells, and blocked cells. When at a dirty cell, the robot may clean the dirty cell, turning it into a clean cell. The goal is to clean all dirty cells. The domain contains aspects of grid path planning, since the robot has to navigate from one place to another, as well as elements of the traveling salesman problem, since the dirty cells can be viewed as cities that must be visited in the best order possible so as to minimize travel distance.

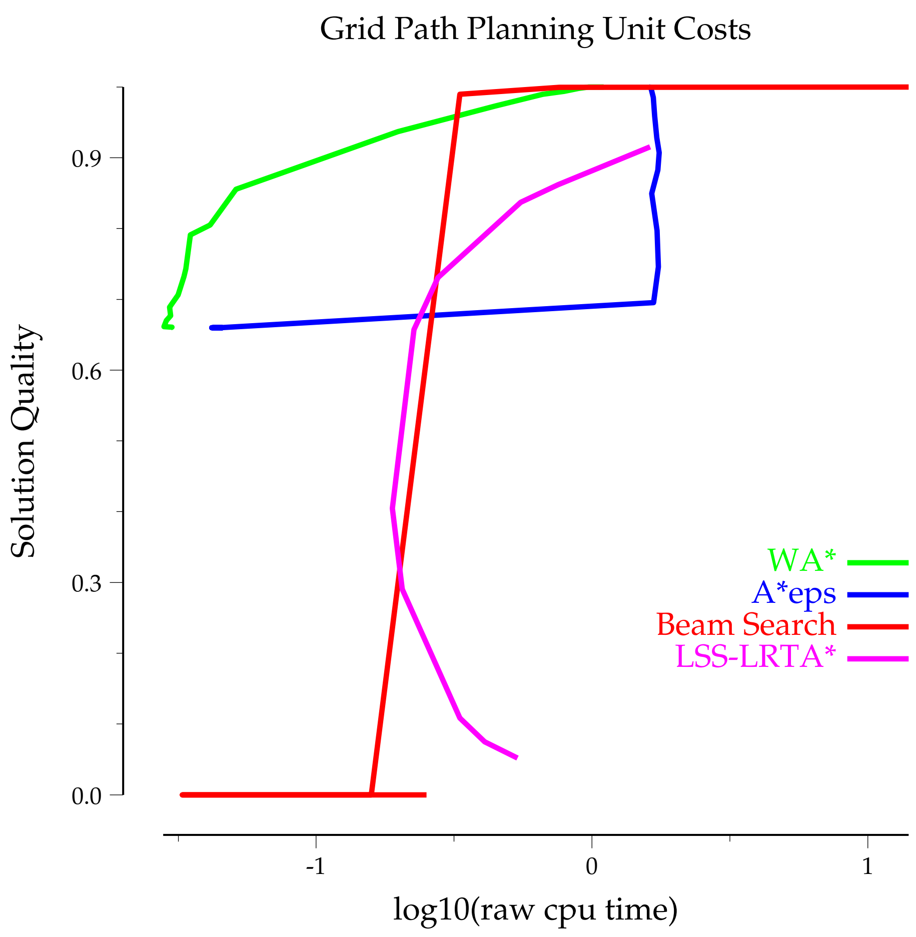

Figure 3: Algorithm performance on grid path planning with unit cost

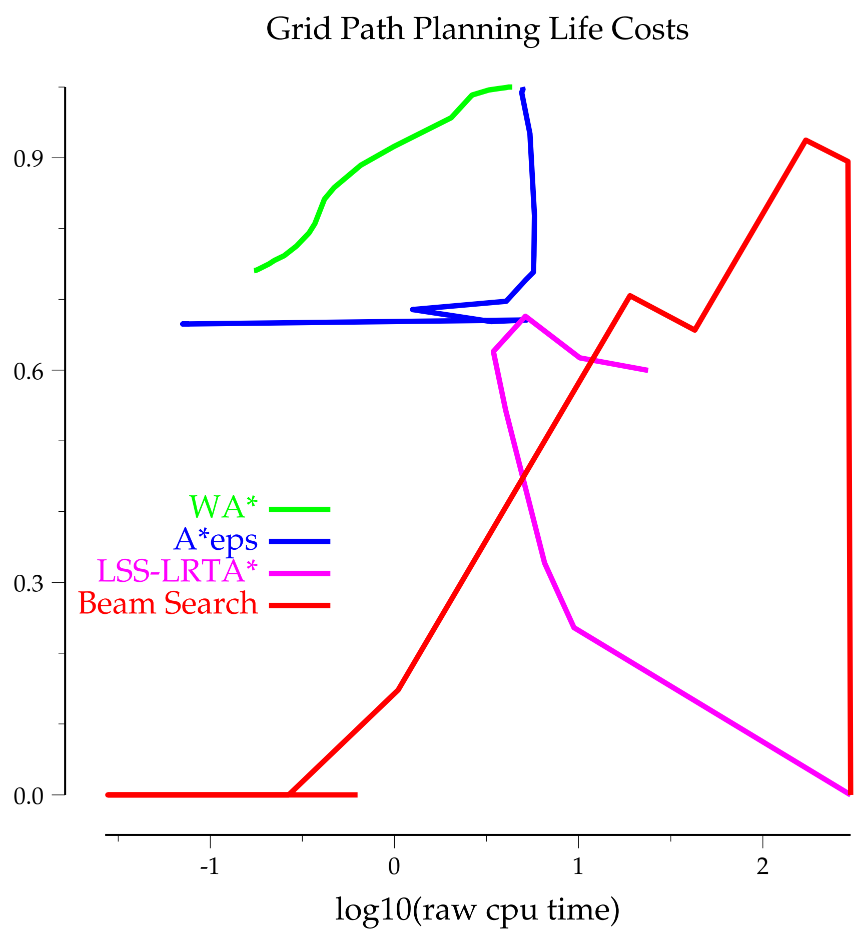

Figure 4: Algorithm performance on grid path planning with unit cost

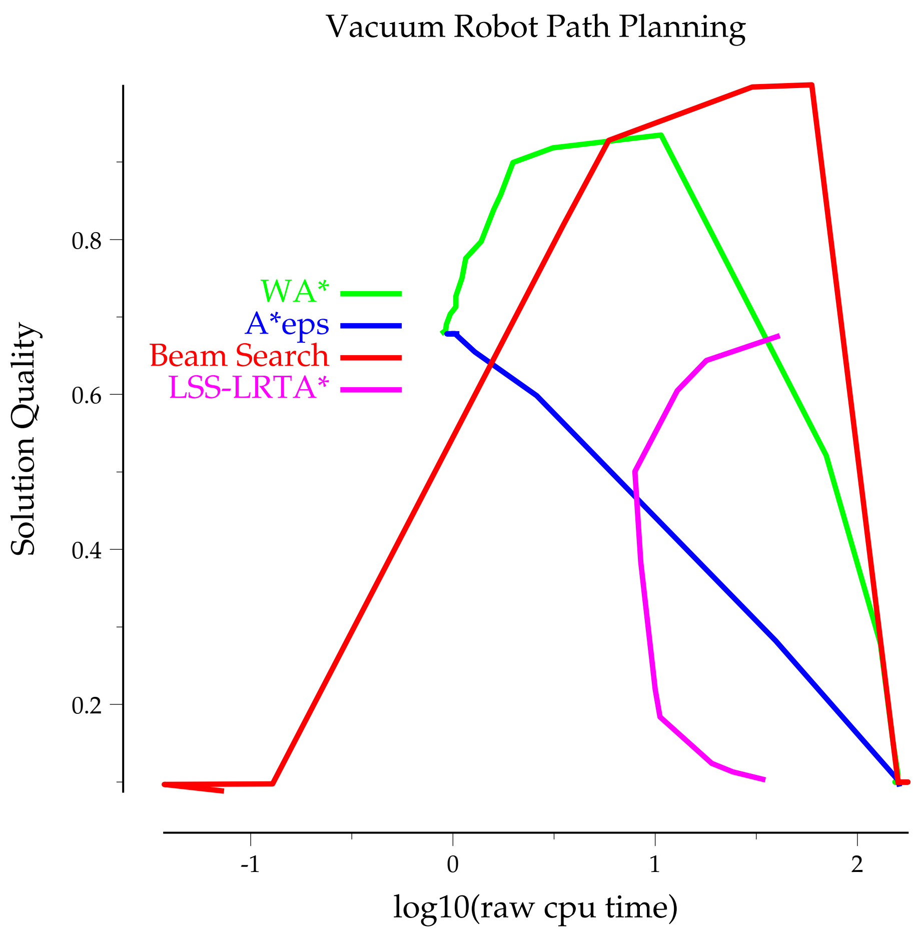

Figure 5: Algorithm performance on the vacuum robot path planning domain

The results of running the four algorithms in question on the vacuum robot path planning problem can be seen in Figure 5. For finding solutions quickly, the algorithm of choice is weighted A*, but if the highest quality solutions are desired, beam search is the algorithm of choice.

1.5 Dynamic Robot Path Planning

In the dynamic robot path planning domain, the goal is to drive a robot from an initial heading/location/speed to a goal heading/location/speed by manipulating the robot's heading and speed. In addition to driving the robot to the correct configuration, there are obstacles that must be avoided.

In this domain, weighted A* provides the highest quality solutions, while A^*_ϵ finds solutions faster.

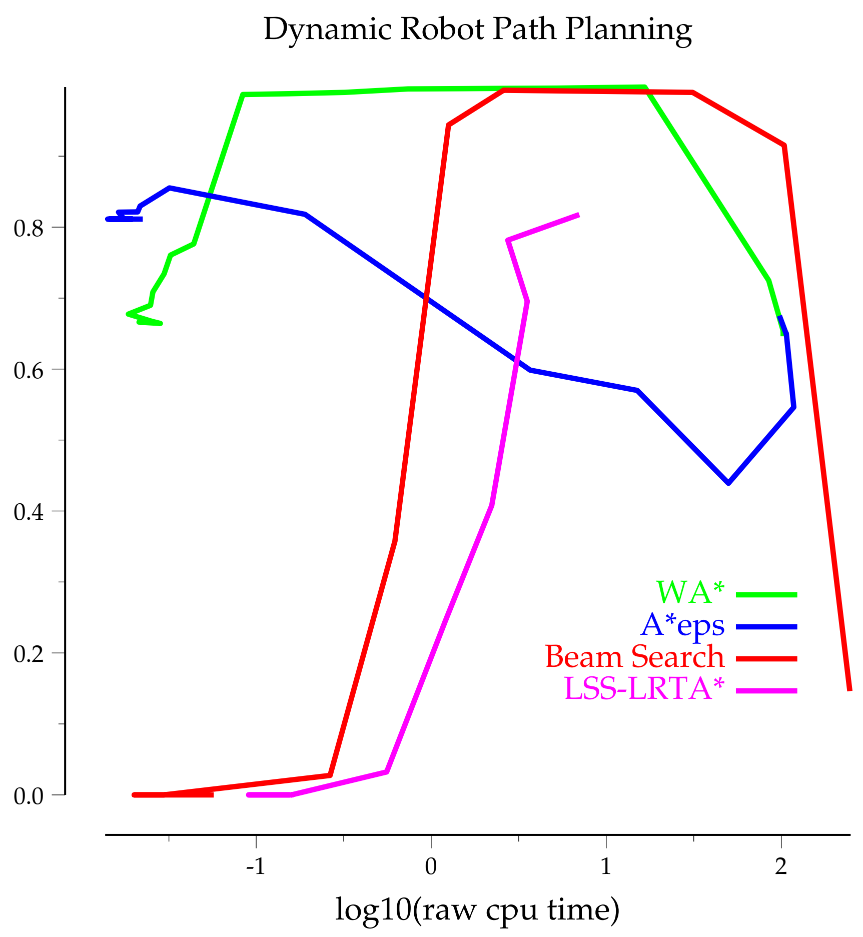

Figure 6: Algorithm performance on dynamic robot path planning

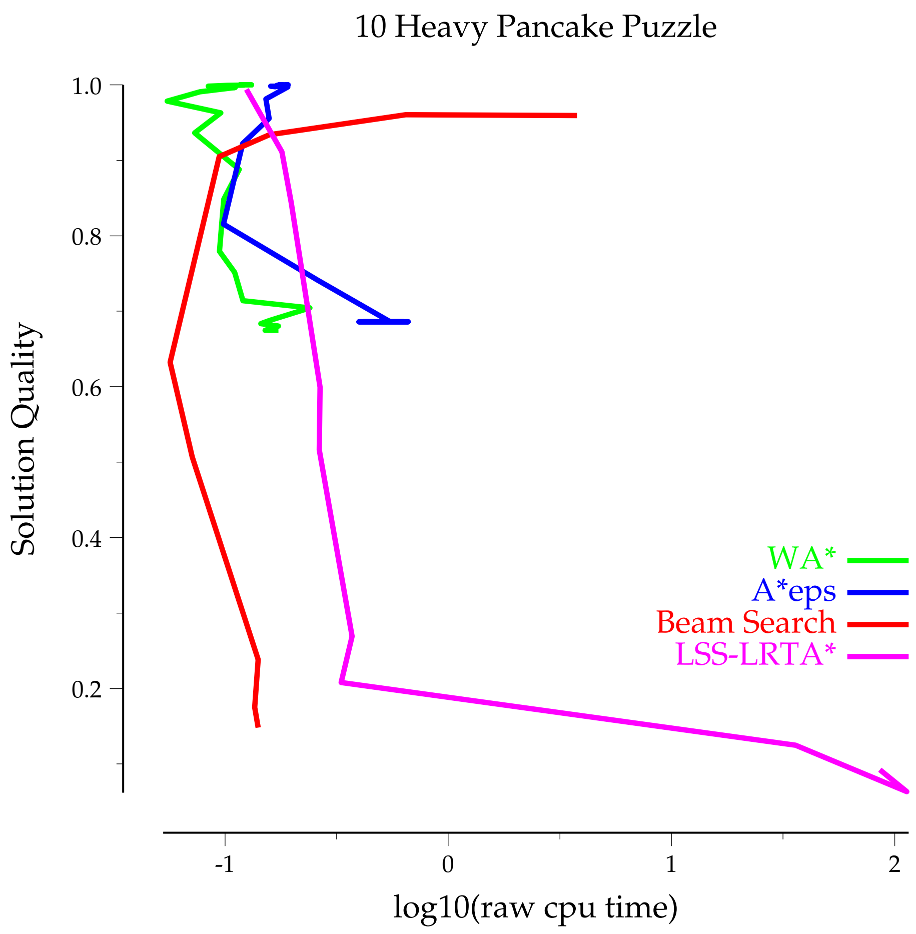

Figure 7: Algorithm performance on the heavy pancake puzzle

1.6 Heavy Pancake Puzzle

In the heavy pancake puzzle, there is a stack of pancakes that must be put in the correct order. The only operation is to take the top k pancakes and flip them. In the standard pancake puzzle, each flip costs 1. In the heavy pancake puzzle, each pancake is assigned an index, the cost of each flip is proportional to the sum of the indexes pancakes that are being flipped.

As can be seen in Figure 7, weighted A* is capable of finding solutions before any other algorithm, and finds higher quality solutions at the same time.

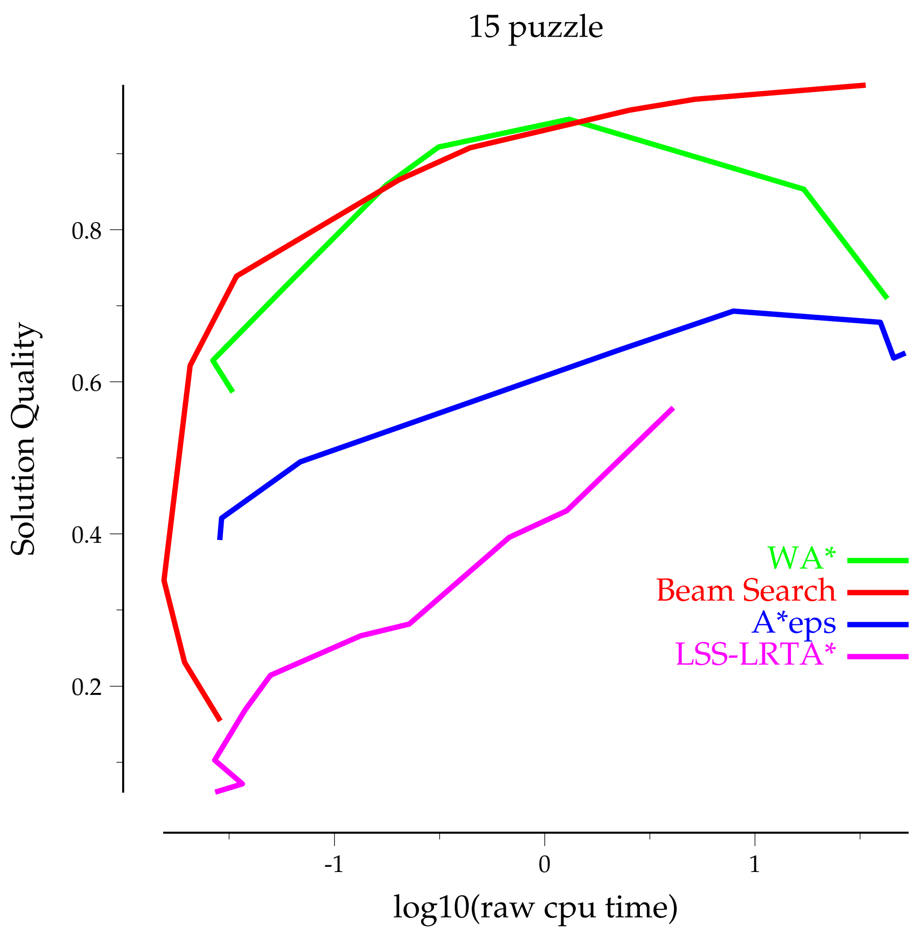

Figure 8: Algorithm performance on the 15 puzzle

1.7 Sliding Tile Puzzle

On the sliding tile puzzle, beam search finds both the fastest and the highest quality solutions. This is even more pronounced on the 48 puzzle, where the other algorithms either fail almost all the time as is the case with weighted A* and A_ϵ^*, or find very low quality solutions, as is the case with LSS-LRTA*.

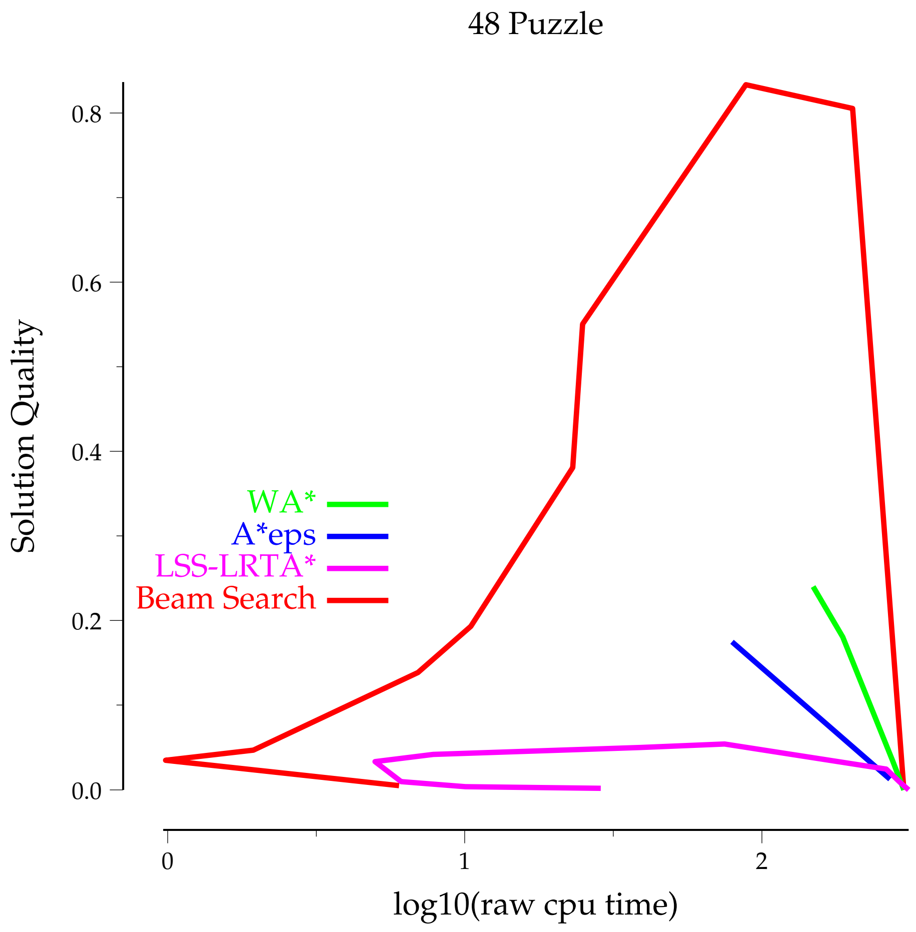

Figure 9: Algorithm performance on the 48 puzzle

2 Selecting an Algorithm - What Matters?

2.1 Time versus Quality

One fact that was apparent from the previous section is that which algorithm is best is a somewhat subjective question. For example, if we are trying to decide how to route paper through a printer, we need to have a solution quickly, and it is practical to trade solution quality for a shorter planning time. Conversely, if we are trying to figure out how route ships or trains around the planet, we can afford additional computational time in order to find a higher quality solution.

Since the definition of "best" fundamentally varies, we consider the two extreme ends of the time-solution quality trade-off. In one situation, we value time over quality, and place the most importance on finding solutions quickly. In the other situation, we value solution quality over time, only considering time when two or more algorithms ultimately find solutions of the same quality.

2.2 Dead Ends

Wilt et al. [2010] show that dead ends cause problems for beam search due to the nature of inadmissible pruning. Since there is no way a priori to tell whether pruning a given node will cause the search to be unable to find solutions, beam searches often fail when run with small beam width on problems with dead ends, as is the case with the vacuum robot navigation, grid path planning, and dynamic robot path planning domains. Although dead ends cause problems for beam searches with small beam width, beam searches can eventually find solutions if the beam width is made large enough, as can be seen in dynamic robot path planning (Figure 6), grid path planning (Figures 4 and 3), and vacuum robot planning (Figure 5), where the beam searches with wide beams do manage to find solutions.

In domains with dead ends, beam searches take longer to find solutions due to the fact that the small beams fail to find a solution. In general, beam searches rely upon small beams to find solutions quickly, and large beams to find high quality solutions. If the small beams are failing, then beam searches are going to find solutions later.

2.3 Unit versus Non-Unit Cost

Beam searches perform significantly better in domains with unit cost as compared to domains with non-unit cost. For example, changing the cost function on grid path planning from unit to life causes a substantial decrease in the performance of beam searches. In addition to that, changing the cost function in the sliding tile puzzle from unit to the square root of the tile moved also causes a major reduction in the effectiveness of beam search, as shown in Figure 10. Just as beam searches are able to overcome problems with dead ends by using a sufficiently large beam, beam searches are able to overcome non-unit cost if the beam width is set sufficiently high, as is the case in the square root 15 puzzle and the heavy pancake puzzle (Figure 7).

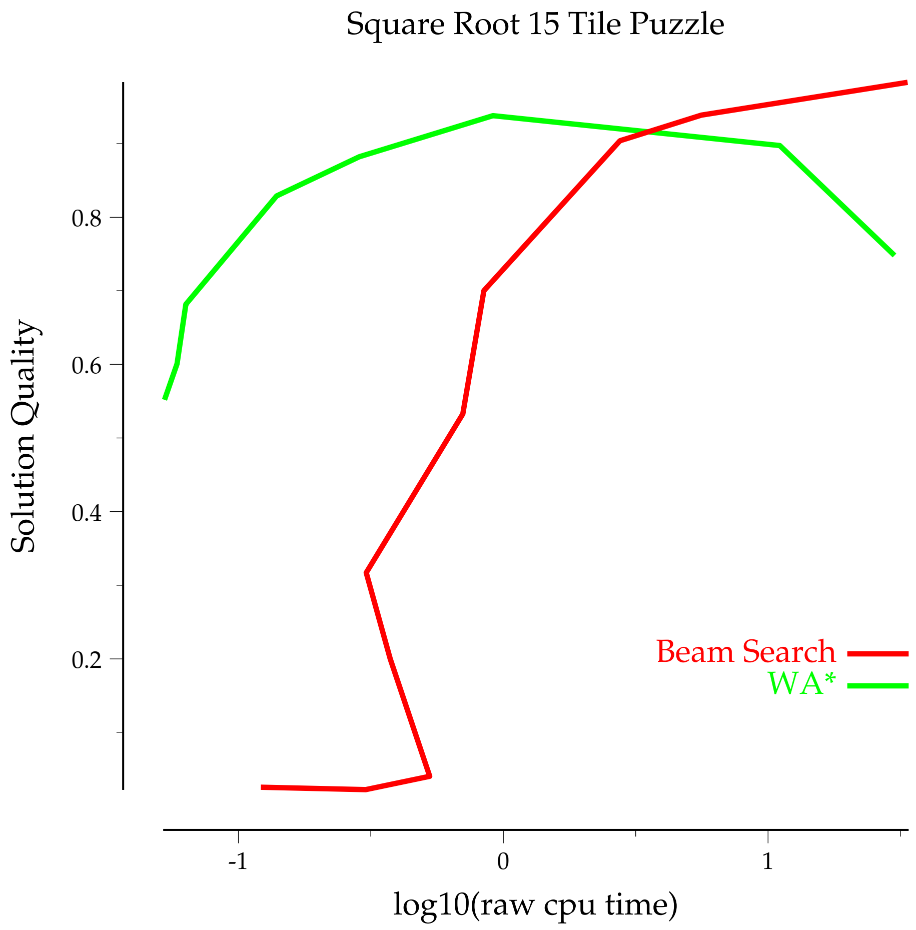

Figure 10: Beam search and weighted A* on the square root tile puzzle

2.4 Is d the Same as h?

$d(n)$ is a heuristic that estimates how many nodes are between the node n and the nearest (or possibly cheapest) goal.

The first and most important thing to consider here is whether there is any fundamental difference between d and h. If the domain is a unit cost domain, estimating how many nodes there are on the path to the nearest solution and estimating the cost of the path to the nearest solution produce the same result, so d provides no additional information, with one possible exception. The d heuristic does not have to be admissible, so in some domains we can use a more accurate, but not admissible, d heuristic.

A_ϵ^* relies upon the heuristic estimate d to ascertain how many nodes there are along the optimal path to the goal from that node. As one might naturally expect, domains in which the d heuristic provides additional information are much better suited to A_ϵ^*, and A_ϵ^* performs much better in domains with highly accurate d heuristics.

In some domains, it is trivial to find out how far away a goal is. For example, in the traveling salesman problem, this number can be calculated with perfect accuracy by counting the number of cities that have not yet been visited. In other domains, the d function makes the same abstraction assumptions as h, so d is no more informed than h is. In grid path planning, both h and d make the free space assumption, optimistically assuming all of the cells are empty. Despite this similarity, the d heuristic is much more accurate in unit cost grids as opposed to in life cost grids. We can observe this empirically if we calculate the average percent error in the two heuristics. h has an average percent error of 27, while d has an average percent error of 18. In the pancake puzzle, both h and d come from pattern databases, which abstract away some of the pancakes. In this domain, d has an average percent error of eight, while h has an average percent error of 38.

Since the vacuum robot path planning domain is a unit cost domain, d and h functions we used always return the same value. The sliding tile puzzle is unit cost, so h and d return the same number in this domain as well.

Domains where there is a substantial difference between h and d are the traveling salesman problem, dynamic robot path planning, and grid path planning with life costs, as well as the heavy pancake problem. In the traveling salesman problem, h comes from a minimum spanning tree of the remaining cities, whereas d can be calculated with perfect accuracy by counting the unvisited cities in the state. In dynamic robot path planning, h assumes that the robot can accelerate and decelerate instantly, as well as drive in a circle with an infinitely small radius, so it is often impossible to follow the path suggested by h. d points to a legal (albeit suboptimal) path in dynamic robot path planning, and is therefore more accurate. In grid life and heavy pancake, we used empirical analysis to show that there was a quantitative difference between h and d.

2.5 Does Weighted A* Fail?

In some of our benchmark domains, weighted A* fails if it is run with a weight that is too small. For example, if we use a weight that is too small A* can't solve traveling salesman problems or vacuum robot navigation problems. The same phenomenon can also be observed on the dynamic robot navigation problems if the weight is sufficiently small, although there does not appear to be any reason to run weighed A* with a weight that small, since the medium weights find the best solutions that can be found by any algorithm under consideration.

Unfortunately, predicting whether or not weighted A* is likely to fail remains an open question.

2.6 Domain Summary

Table 1 shows the different domains under consideration. This table also shows what value each domain takes for the attributes under consideration.

Domain	Dead Ends	Unit Cost	Unique States	WA* fail?	h more accurate?
TSP	No	No	6×10^{33}	Yes	Yes
Grid Unit	Yes	Yes	2×10^{6}	No	No
Vacuum	Yes	Yes	6×10^{11}	Yes	No
Grid Life	Yes	No	2×10^{6}	No	Yes
Pancake	No	No	3×10^{6}	No	Yes
Robot	Yes	No	2×10^{11}	No	Yes
15 Puzzle	No	Yes	6×10^{11}	Yes	No
48 Puzzle	No	Yes	3×10^{62}	Yes	No

Table 1: Domain Attributes by Domain

3 Selecting an Algorithm

3.1 Decision Tree

Figure 11 shows a decision tree for selecting an algorithm. The first branch point is selecting to prefer quality tie breaking on time, or to select time tie breaking on quality. As was evident from the plots, it was possible to trade time for quality, and depending which is preferred, the algorithm of choice varied.

The next branch points are domain features. The first is the cost function, since beam searches perform much better on domains with unit cost compared to domains with non-unit cost. The next branch point is whether or not the domain has dead ends. Domains with dead ends also cause problems for beam searches. If the domains has no dead ends and unit cost, beam search is the algorithm of choice for both finding solutions quickly and finding high quality solutions.

If we value solution quality over solving time and have either non-unit costs or a domain with dead ends, the last branch point to consider is whether or not weighted A* with a low weight will fail. If weighted A* is a reasonable option, then we should use weighted A*. If weighted A* fails, then we would be better off using beam search, since beam searches offer better scaling behavior.

If we value time over solution quality and have a domain with dead ends or non-unit cost, we have to decide whether we should use weighted A* or A^*_ϵ. To that end,

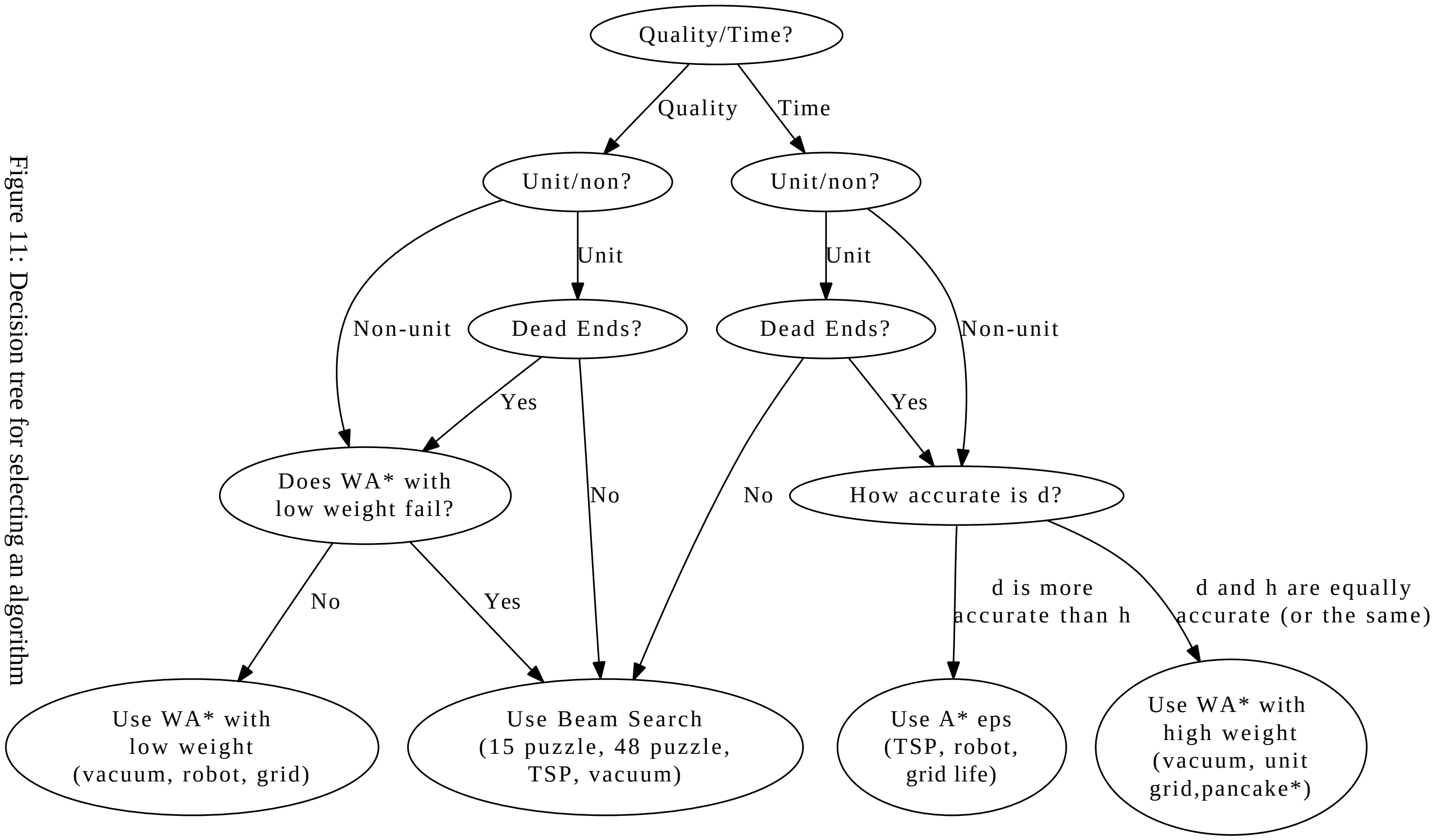
Quality/Time?
Quality
Time
Unit/non?
Unit/non?
Unit
Unit
Non-unit
Non-unit
Dead Ends?
Dead Ends?
Yes
Yes
No
No
Does WA* with low weight fail?
How accurate is d?
No
Yes
d is more accurate than h
d and h are equally accurate (or the same)
Use WA* with low weight (vacuum, robot, grid)
Use Beam Search (15 puzzle, 48 puzzle, TSP, vacuum)
Use A* eps (TSP, robot, grid life)
Use WA* with high weight (vacuum, unit grid,pancake*)

Figure 11: Decision tree for selecting an algorithm

* The pancake domain is problematic and does not follow the decision tree's predictions.

we consider whether or not the d function provides a more accurate estimate than h. If d is more accurate than h, as is the case in some domains, then we should select A^*_ϵ, otherwise we would be better off using weighted A*. Of the domains considered, this rule correctly predicts what to do with grid world with life costs the dynamic robot path planning, but incorrectly selects A^*_ϵ over weighted A* for the heavy pancake puzzle. Determining exactly when to used A^*_ϵ over weighted A* remains an open question.

Under no circumstances was LSS-LRTA* the algorithm of choice. This is hardly surprising, since the algorithm was not designed to solve the shortest path problem with the real-time constraint on actions removed.

4 Conclusion

We analyzed the performance of four effective greedy search algorithms, weighted A*, A^*_ϵ, beam search, and LSS-LRTA*. Our results showed that which algorithm could be considered "best" depended on user preferences based upon value placed on time and solution quality, as well as domain features. We also observed that which algorithm worked the best depended on what kind of hardware was available, since more powerful machines can use weighted A* which may not be a reasonable option on a less powerful machine due to either time or memory.

We compile these results into a decision tree, and use that tree to help us decide which algorithm should be run in a given situation.

References

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

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

J. E. Doran and D. Michie. Experiments with the graph traverser program. In *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, pages 235–259, 1966.

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

David Furcy and Sven Koenig. Scaling up WA* with commitment and diversity. In *International Joint Conference on Artificial Intelligence*, 2005b.

Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

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

Sven Koenig and Xiaoxun Sun. Comparing real-time and incremental heuristic search for real-time situated agents. In *Journal of Autonomous Agents and Multi-Agent Systems*, pages 18(3):313–341, 2008.

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

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

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

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

Elaine Rich and Kevin Knight. *Artificial Intelligence*. McGraw-Hill, Incorporated, 1991.

Stuart Russell and Peteer Norvig. *Artificial Intelligence: A Modern Approach*. Third edition, 2010.

Christopher Wilt, Jordan Thayer, and Wheeler Ruml. A comparison of greedy search algorithms. In *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.