A Comparison of Greedy Search Algorithms

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

Abstract

We discuss the relationships between three approaches to greedy heuristic search: best-first, hill-climbing, and beam search. We consider the design decisions within each family and point out their oft-overlooked similarities. We consider the following best-first searches: weighted A*, greedy search, A^*_ϵ, window A* and multi-state commitment k-weighted A*. For hill climbing algorithms, we consider enforced hill climbing and LSS-LRTA*. We also consider a variety of beam searches, including BULB and beam-stack search. We show how to best configure beam search in order to maximize robustness. An empirical analysis on six standard benchmarks reveals that beam search and best-first search have remarkably similar performance, and outperform hill-climbing approaches in terms of both time to solution and solution quality. Of these, beam search is preferable for very large problems and best first search is better on problems where the goal cannot be reached from all states.

Introduction

In many applications of heuristic search finding optimal solutions is prohibitively expensive. Despite abandoning optimality, we strive to find solutions of high quality as quickly as possible. In this paper, we consider the three major families of algorithms that are suitable for solving shortest path problems: best-first, hill-climbing, and beam searches.

Algorithms like weighted A* (Pohl 1970) systematically explore the search space in 'best' first order. 'Best' is defined by a node ranking function which typically considers the cost of arriving at a node, g, as well as the estimated cost of reaching a goal from a node, h. Some algorithms, such as A^*_ϵ (Pearl and Kim 1982) also consider the distance of a node from the goal, d. Hill-climbing algorithms are less deliberative; rather than considering all open nodes, they expand the most promising descendant of the most recently expanded node until they encounter a solution. Beam searches exist in between these families of algorithms, exploring a fraction of the space and pruning regions that appear less promising.

Although these algorithms can all be applied to the shortest path problem, we are unaware of any studies that directly compare them. The main contribution of this paper is the first cross-family empirical comparison that includes such a wide variety of algorithms. Our study considers six standard benchmark domains: pathfinding in grids (Thayer, Ruml, and Bitton 2008), the traveling salesman problem (Pearl and Kim 1982), dynamic robot pathfinding (Likhachev, Gordon, and Thrun 2003), the sliding tile puzzle (Korf 1985), the pancake puzzle (Holte, Grajkowskic, and Tanner 2005), and a vacuum-robot domain (Russell and Norvig 2010). Our evaluation shows that the strength of hillclimbing algorithms is their amenablity to realtime settings, not the quality of solutions they return or the time they require to find them. We find that despite their differences, best-first search and beam search frequently provide a comparable time-solution quality trade off, with best-first searches having an advantage on problems where the goal cannot be reached from all states, and beam searches having better scaling behavior due to their bounded memory consumption. This improved understanding is a necessary first step toward choosing algorithms appropriate to a given problem and toward designing a better greedy search.

We begin by comparing various algorithms within each family, and selecting the best algorithms in each family to consider in the cross-family evaluation we then perform. We use these results to make recommendations about when to use each algorithm family.

Best-First Search

In best-first search nodes are expanded one at a time according to some definition of best. The simplest search algorithms order all nodes on a single criterion, typically an estimate of the cost of a solution passing through that node. More sophisticated best first searches like A^*_ϵ (Pearl and Kim 1982), Window A* (Aine, Chakrabarti, and Kumal 2007), and multi-state commitment k weighted A* (Furcy and Koenig 2005b) define a privileged set of nodes from which nodes are expanded.

Of the three major algorithm families, the best-first search family seems least oriented towards solving a problem as quickly as possible. These algorithms are complete, or in the case of anytime window A* can be made to be complete, and in some cases they expend effort to prove the quality of their solutions. The completeness of these algorithms is the result of their having an open list of unbounded size, and the fact that they only terminate when they have found a

solution or they have emptied the entire open list. Despite the fact that the time and space complexity of a best-first search is bounded only by the size of the underlying search space, the completeness of these algorithms allows best-first searches to avoid problems created by heuristic error and local extrema. We now introduce the best-first searches we consider.

Weighted A* is a simple and effective bounded suboptimal search. The traditional node evaluation function of A* (Hart, Nilsson, and Raphael 1968) is $f(n) = g(n) + h(n)$. In weighted A*, this function is modified to place additional emphasis on the heuristic evaluation function, resulting in $f'(n) = g(n) + w \cdot h(n)$. The weight, w, increases the importance of h, the estimated cost to go, relative to g, the cost already incurred, in the node ordering function.

Weighted A* has the property that when it finds a solution, its cost is within a factor w of the optimal solution cost. Proving the solution meets some quality bound might seem expensive, especially when no such requirement exists. A prerequisite of maintaining a quality bound is paying attention to g, and this can help an algorithm avoid being misled by overly optimistic heuristics.

Greedy best-first search (Doran and Michie 1966) is the logical extreme of weighted A*. Rather than scaling h relative to g, greedy search ignores g completely. As a result, there is no way to request a fixed quality solution from greedy search; the quality of the solution returned may be determined after the fact by comparing its cost with a lower bound on the optimal solution cost for the problem which can either be derived from the initial state or a linear scan of all the nodes on the open list when the goal was returned.

$\mathbf{A}^*_\epsilon$ maintains two orderings on the nodes, allowing two sources of information to be considered when deciding which nodes to expand. The open list of A^*_ϵ sorts nodes as they would be by A*. A^*_ϵ uses the open list to form a subset of nodes to consider for expansion called the focal list. The focal list contains all nodes such that $f(n) \leq w \cdot f(f_{min})$, where f_{min} is the node with minimum $f(n)$. The focal list is sorted in increasing order of d. At each step, A^*_ϵ expands the node with minimum d from the focal list. In some sense it behaves much like a greedy search on d, but rather than comparing all of the d values globally, it singles out a subset of the nodes to be considered.

Window A* is an incomplete search in which A* is run on a sliding window of size s instead of considering the entire open list. Each node considered for expansion is compared to the depth of the deepest node found thus far. If the deepest node is more than s nodes deeper than the current node, the current node is pruned. The sliding window forces the search algorithm to dive deeper and deeper into the search space by pruning nodes that are high in the search tree. Although window A* prunes nodes like a beam search, there is no bound on the size to which its open list can grow. For this reason, we include window A* with the best-first searches.

Anytime window A* is an extension of window A* where window A* is run with iteratively increasing window sizes. By starting with a small window, and increasing the size when an iteration terminates, anytime window A* can find increasingly improving solutions, eventually terminating with the optimal solution. When evaluating window A*, we look at the first solution returned by anytime window A*, rather than any particular setting of window size. This neatly avoids the problem of hand tuning the parameter for this particular search.

Multi-State Commitment Search Maintaining a set of privileged nodes is not only useful for improving search order, it also helps algorithms scale, as shown by Furcy and Koenig (2005b) in their paper on multi-state commitment k-weighted A*(MSC-kwA*). Like A^*_ϵ and window A*, MSC-kwA* maintains a privileged set of nodes to be considered for expansion, the commit list. MSC-kwA* has three parameters. The first is w, identical in form and function to that used by weighted A*. The second parameter defines the size of the commit list. The third parameter is k. Each time the algorithm is ready to proceed, it removes the k best nodes from the commit list and expands all of these nodes. The children are put back into the commit list. If there is no room in the commit list, the worst node on the commit list is removed and placed onto another list called the reserve list. If the commit list is ever smaller than the maximum possible size, it is replenished from the reserve list.

Comparison of Best-First Algorithms

Window A*, $\mathbf{A}^*_\epsilon$ and MSC-kwA* are similar The intuition behind all three algorithms is that the size of the relevant search space can be reduced by defining a subset of the most promising nodes, and expanding only those nodes.

Ignoring Duplicates If the state space forms a graph, there are many ways a search may arrive at the same state. When using inconsistent heuristics or algorithms other than weighted A*, ignoring duplicate states may relax the bound (Ebendt and Drechsler 2009). In this evaluation the bound is of no consequence, so all algorithms should avoid re-expanding duplicates as this typically results in fewer node expansions (and solutions of lower quality).

Empirical Results All algorithms were implemented in Objective Caml, compiled to native binaries, and run on 3.16 GHz Intel Core 2 duo E8500 systems with 8 Gb RAM under 64-bit Linux. Algorithms with suboptimality bounds were sampled at 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 while algorithms with integer parameters were sampled at 50000, 10000, 5000, 1000, 500, 100, 50, 10, 5, and 3. All runs were limited to 5 minutes of cpu time. Any algorithm not able to find a solution within that time was considered to have failed. Algorithms were terminated after 300 seconds. This time limit was such that memory was not an issue, except on the 7x7 sliding tile puzzle. In that domain, we defined a failure as either running out of time or memory.

The first domain considered is the dynamic robot path planning domain (Likhachev, Gordon, and Thrun 2003). Here the object is to control a robot's heading and velocity to drive it from one place to another. We also consider vacuum robot path planning, which is a domain inspired by (Russell and Norvig 2010) in which a robot must clean all dirty squares on a grid. We also consider path planning on grids as it is a simplification of the vacuum robot path planning problem. Another domain we consider is the sliding

Domain	Cycles	Dead Ends	Unit Cost	Depth Bound	Branching Factor
TSP	None	No	No	Yes	1–99
Vacuum	Short	Yes	Yes	No	0–3
Grid Life	Short	Yes	No	No	0–3
Pancake	Short	No	No	No	8-9
Robot	Long	Yes	No	No	0–240
Tiles	Long	No	Yes	No	1–3

Table 1: Domain Attributes

tile puzzle (Korf 1985). In addition to that, we also consider the heavy pancake puzzle, which is a derivative of the standard pancake puzzle in which the cost of flipping a stack of pancakes is proportional to the indexes of the pancakes that are to be flipped. For the between-family comparison, we also consider the traveling salesman problem.

Domains can be classified in many ways. We consider five domain attributes, cycles, dead ends, cost function, depth bound, and branching factor. The domains we selected provide a wide range of combinations of these attributes, summarized in Table 1.

In Table 2 the results of the different algorithms under consideration are shown for our selected benchmark domains. We show the quality of a solution as between 0 and 1. We calculate this by dividing the cost of the best solution found for the problem by the cost of the solution returned by an algorithm. Failing to find a solution earns a score of 0 and finding the best solution over all algorithms earns a score of 1. The number shown in the Q column is the average of this quality metric across all instances. The T column is the average run time in seconds, including failed runs. For each domain, bold denotes the best algorithm valuing quality and tie breaking on time. Italics denotes the best algorithm valuing run them and tie breaking on quality. When applicable, considerations are made for failure rate.

In dynamic robot path planning, the algorithm of choice is either weighted A* or A^*_ϵ, since anytime window A* fails almost all the time, greedy best-first search finds poor quality solutions and MSC-kwA* takes significantly longer to find solutions. In the vacuum robot path planning domain, weighted A*, greedy search, and A^*_ϵ can all be argued to be the algorithm of choice, depending on the desired solution quality-time trade-off. Anytime window A* fails too often and although MCS-kWA* returns high quality solutions, it takes longer than weighted A* with a weight of 2, so neither of these algorithms are competitive in this domain. All of the algorithms under consideration performed well on grids, with the exception of anytime window A*, which was once again plagued by a high failure rate. On the sliding tile puzzle, weighted A* with a weight of 5 finds a solution of reasonable quality almost instantly, whereas weighted A* with a weight of 2 offers the best solution quality. Anytime window A*, greedy best-first search, and MSC-kwA* all find solutions quickly, but the solutions are of lower quality than those of weighted A*. In the last domain, the heavy pancake puzzle, the clear choice is weighted A* with a weight of 2, since it finds the best solutions in under a tenth of a second.

In all five benchmark domains considered, weighted A* was always capable of producing the highest quality solutions. When other algorithms were able to produce solutions of comparable quality, the other algorithms were never able to produce a run time that was significantly different than the run time of weighted A*. On some domains A^*_ϵ was able to find solutions significantly faster than weighted A*, so we also consider A^*_ϵ when evaluating best-first searches against the other algorithm families. MSC-kwA* was competitive in grid path planning, but its performance was never an improvement over weighted A*, and was often substantially worse in terms of either time or solution quality. The same can be said for greedy search. For this reason, we shall only consider weighted A* and A^*_ϵ further.

Hill-climbing

A basic hill climbing algorithm expands one node at a time beginning with the root, and each time expands only the best child of the previously expanded node. One might reasonably conclude that these algorithms do no more work than they absolutely have to do; no effort is expended to guarantee solution quality or completeness. Consequently, we might reasonably expect hill climbing algorithms to find solutions with very little effort, allowing them to be extremely competitive in this setting. Hill-climbing algorithms have two major drawbacks: some lack a tunable parameter so that we can select an appropriate trade-off between speed and quality, and the algorithms are brittle, frequently failing to find solutions. We now introduce the hill-climbing algorithms considered.

Enforced Hill-climbing Local extrema are a common challenge for hill-climbing. In these situations, the algorithm must move through a potentially large area of worse states before it reaches the next best state. We could choose to just terminate, or behave randomly until we find a new decreasing value. Both return poor quality solutions, since termination returns no solution at all and it could be a while before a random walk discovers a way out of the local extrema.

There are better tactics than random behavior. Hoffmann and Nebel (2001) suggest that when facing a local extrema, we search systematically outwards using a breadth-first search until we find a node which is better than our current position, at which point we move to the better location and continue our hill-climbing. This approach performs far better in practice than behaving randomly, but is not without pitfalls. Performing this breadth-first search often proves prohibitively costly, as shown in the comparatively high failure rate of enforced hill climbing.

Real-time Search We place real-time search algorithms into the hill-climbing family due to the common trait of committing to actions before the search has completed. In real-time search this is motivated by a time constraint on taking actions in the real world. Although that constraint does not apply in our evaluation, real-time algorithms such as LSS-LRTA* (Koenig and Sun 2008) can still effectively solve shortest path problems. LSS-LRTA* searches outward from its current state using A* for a fixed duration. Next it commits to the best looking state on the frontier. Last, it runs Dijkstra's algorithm to improve the heuristic values. The al-

Alg.	Robot			Vacuum			Grid			Tiles			Pancake		
	F%	Q	T	F%	Q	T	F%	Q	T	F%	Q	T	F%	Q	T
WA*(2)	**0**	**1.0**	**0.1**	**0**	**1.0**	**3.3**	**0**	**1.0**	**0.7**	**0**	**1.0**	**0.6**	***0***	***1.0***	***0.0***
WA*(5)	0	0.7	0.0	0	0.8	1.6	0	0.9	0.3	*0*	*0.7*	*0.0*	0	0.8	0.0
A*eps(2)	*0*	*0.8*	*0.0*	65	0.3	197	0	0.7	5.3	9	0.7	34.0	0	1.0	0.1
A*eps(5)	*0*	*0.8*	*0.0*	0	0.7	1.2	*0*	*0.7*	*0.1*	0	0.5	0.0	0	0.7	0.0
Greedy	0	0.1	0.1	*0*	*0.7*	*1.1*	0	0.8	0.2	0	0.2	0.0	0	0.6	0.0
Anytime Window A*	97.5	0.0	293	63	0.3	220	85	0.1	278	0	0.2	0.0	0	0.2	2.0
MSC-kwA*(10, 5, 2)	0	1.0	0.5	0	1.0	4.1	**0**	**1.0**	**0.7**	0	0.3	0.0	0	0.5	0.0

Table 2: Mean performance of best-first searches across our benchmarks
(F% = failure rate, Q = quality, T = time in seconds, italicized denotes competitive on time, bold competitive on quality)

gorithm repeats this process until a goal is found or there are no children to expand.

Depth-First Branch and Bound We do not consider algorithms from this family because they work poorly when the search space does not have a depth bound. Of our six benchmark domains only the TSP has bounded depth. Similarly, genetic algorithms rely upon quickly and easily finding feasible, or close to feasible candidate solutions to breed together. We are not aware of any studies that consider genetic algorithms for implicitly represented large graphs, so we do not consider that family of algorithms here.

Comparison of Hill-Climbing Algorithms

Pitfalls of Early Commitment Hill-climbing algorithms are extremely brittle on some common benchmark domains, such as the dynamic robot path planning domain. In this domain, there are many nodes that have no children. For example, nodes with the robot moving very quickly towards a wall generate zero nodes since the robot crashes no matter what control action is applied. While complete algorithms such as weighted A* can easily recover by considering a node from before the fatal decision, hill-climbing algorithms and real-time algorithms can end up in situations where they can no longer solve the problem, the result of early commitment. At every step, hill-climbing algorithms commit to a node, which sometimes causes early termination.

Comparing LSS-LRTA* with Hill Climbing Table 3 shows the relative performance of enforced hill-climbing and LSS-LRTA*. Enforced hill-climbing is only competitive with LSS-LRTA* on the pancake puzzle and vacuum robot planning. The time to solution quality trade-off of enforced hill-climbing on dynamic robot path planning and tiles is abysmal due to a high failure rate. On grid path planning, enforced hill-climbing is dominated by LSS-LRTA*. In vacuum robot path planning and the pancake puzzle enforced hill-climbing is competitive with LSS-LRTA*, but the performance is comparable. The failure of enforced hill-climbing on the dynamic robot domain and the sliding tile domain, combined with the fact that the performance is otherwise comparable to that of LSS-LRTA* justifies its exclusion from further consideration.

Beam Search

In this section, we first discuss the different kinds of beam searches, and important design decisions that must be made when implementing a beam search. Beam searches can be grouped into two general categories. In a best-first beam search, the beam search is run just like a best-first search, except that when the open list grows beyond a predetermined size limit, the lowest quality nodes are removed from the open list until the open list is within its size bound (Rich and Knight 1991). The other kind of beam search is a breadth-first beam search. These are the same as a breadth-first search, except at each depth layer a fixed number of nodes are expanded; the rest are pruned (Bisiani 1992).

Although both variants were used for experiments, best-first beam searches performed poorly when compared to breadth-first beam searches, as can be seen in the left pane of Figure 1. The points on the scatter plots are connected with lines that show the relationship between points as a function of the parameter being used by the algorithm (weight, beam width, or look ahead). The y-axis represents solution quality.

As can be seen in the left pane of Figure 1 breadth-first beam searches return better solutions than best-first beam searches in less time, so there is no reason one would want to use a best-first beam search over a breadth-first beam search. Consequently, all beam searches presented in the remainder of this paper are breadth-first beam searches.

The poor performance of best-first beam search on the sliding tile puzzle can be attributed to the admissibility of the Manhattan distance heuristic. In a best-first beam search there are nodes of different depths competing for entry into the beam. With an admissible heuristic, f values typically increase with depth. Consequently, it is often the case that shallow nodes look more promising than deep nodes, even if the deeper node is, in reality, the superior node.

To test this hypothesis, we examined a search on an 8-puzzle in detail, where we know h^*, the true cost-to-go. Best-first beam search pruned 468 nodes. 397 of the prunings eliminated a node with high g in favor of one with lower g. Unfortunately, 33% of the time replacing the deeper node was detrimental to the search, since the deeper node had a lower $g + h^*$ than the new node. On the same problem, breadth-first beam search was much more successful, only pruning the better node 11% of the time. The improved discrimination of breadth-first beam search makes a huge difference in solution quality. The solution found by the breadth-first beam search was optimal, whereas the best-first beam search found a solution with cost approximately 10 times that of optimal.

Closed Lists When implementing a beam search, one must

Alg.	Robot			Vacuum			Grid			Tiles			Pancake		
	F%	Q	T	F%	Q	T	F%	Q	T	F%	Q	T	F%	Q	T
EHC	65	0.4	196	***2***	***0.9***	***9.9***	0	0.7	12.0	37	0.5	246.7	*0*	*0.7*	*0.0*
LSS-LRTA*(50)	100	0.1	0.3	0	0.2	18.6	0	0.3	10.5	*0*	*0.4*	*0.0*	0	0.2	0.7
LSS-LRTA*(500)	52.5	0.4	2.3	0	0.6	11.8	*0*	*0.8*	*4.3*	0	0.6	0.2	0	0.6	0.6
LSS-LRTA*(5000)	***5***	***1.0***	***5.2***	0	0.9	17.3	**0**	**1.0**	**6.0**	**0**	**0.8**	**0.9**	**0**	**0.9**	**0.6**

Table 3: Mean performance of hill-climbing searches across our benchmark domains

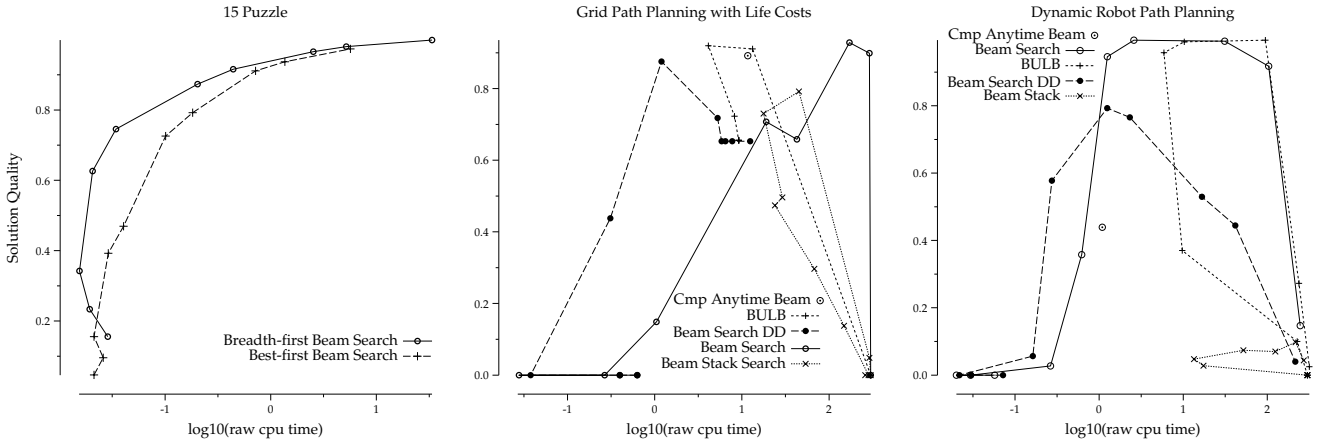

Figure 1: Beam searches on 15-puzzle, life cost grid path planning, and dynamic robot navigation

decide how (if at all) to use a closed list. We can choose not to maintain one, to ignore all duplicate states encountered, or to re-expand duplicates that have better g values than the previous expansion. In some domains, beam searches perform very poorly when not using a global closed list. On one 15-puzzle instance beam search without a closed list kept expanding the same 2,467 nodes forever, while the same search augmented with a closed list solved the problem by generating 5,247 nodes and detecting 65 duplicates. On the pancake puzzle and the TSP, the heuristic is sufficient to guide beam search away from cycles (or in the case of the TSP there are no cycles), but on grid path planning, dynamic robot path planning, vacuum robot planning, and sliding tiles, something must be done to break cycles.

Closed lists allow an algorithm to detect duplicate nodes, so that they may be effectively dealt with. One option is to prune the duplicate node. Another choice is to compare the duplicate node to the previous expansion of it and compare the g value to see if the latest expansion represents a better path to that state. The latter option is extremely important for the performance of beam search in domains with many duplicates and non-unit cost.

The center and right panes of Figure 1 show a surprising trend which is the direct result of duplicate handing policy. Here we see several different kinds of beam search with different closed list policies. "Beam Search DD" immediately prunes any duplicate node, while "Beam Search" (Bisiani 1992) allows the duplicate nodes to be re-expanded if the new expansion has a better g value. BULB, Complete Anytime Beam, and Beam Stack Search are other kinds beam search that will be discussed later. These plots show that re-expanding duplicates allows the algorithm to find higher quality solutions. The beam search that drops duplicates also has the unusual attribute that increasing the beam width decreases the solution quality, while simultaneously increasing the time to the first solution.

The unusual scaling behavior seen in Figure 1 can be seen in domains with non unit cost. An example of this is grid path finding with life costs, where the cost of moving out of a cell is proportional to the cell's y value (Thayer and Ruml 2008). If there are many paths to the goal, the breadth-first beam searches will only consider the most promising paths. As the beam width increases, the number of less promising paths considered also increases. Beam searches with smaller beams greedily follow longer solutions, whereas their counterparts with larger beams find more expensive but shallower solutions as a result of their larger beams. Although the wider beam widths find shallower solutions, the increased beam widths cause the algorithm to take longer than when run with a smaller beam.

Incompleteness of Beam Search A consequence of inadmissible pruning is that it is possible to prune all nodes that lead to goals. There are two approaches to remedy this problem. Zhang (1998) suggests restarting the search using a wider beam. An alternative that does not increase the memory consumption is backtracking, which is the approach followed by Furcy and Koenig (2005a) with BULB and Zhou and Hansen (2005) with beam-stack search.

Zhou and Hansen (2005) define the beam-stack algorithm to begin with some initial upper bound found by some approximation algorithm. Consequently, the first solution found by beam-stack search is whatever solution the approximation algorithm found. As a result, beam-stack does not directly apply in the evaluation context used here, but it can be modified to take infinity as the upper bound, which is what we did. BULB and beam-stack search both drop dupli-

cates on expansion without regard to quality. The result of this behavior is that BULB exhibits unusual scaling behavior, shown in Figure 1. Beam stack search runs into timeout problems instead, failing to find solutions when the beam width is too small or too large.

Comparison of Beam Search Algorithms For the inter-family algorithm comparison, we selected a breadth-first beam search that re-expands duplicates over the alternatives. Best-first beam search was not selected because it does not perform as well as its breadth-first counterpart. Complete beam searches alleviate the problems of incompleteness from a theoretical perspective, but within the time limit considered both BULB and beam-stack search often fail to find solutions if the beam width is too small, just like their incomplete cousins. The only domains where breadth-first beam searches encountered premature termination problems that complete beam searches could solve are grid path planning, vacuum robot planning, and dynamic robot navigation. Figure 1 shows that the breadth first beam search outperforms the other beam searches in dynamic robot path planning, an artifact of duplicate handling. The results are mixed on grid path planning, where BULB is able to find low quality solutions faster, but suffers from the same scaling problem that duplicate dropping beam searches encounter. The balance of the evidence points to the conclusion that the best beam search is a breadth-first beam search.

Comparison of Search Algorithm Families

We now discuss the results of the cross-family comparison. We group the results by underlying phenomenon that are responsible for ordering the performance of the various algorithms considered.

Dead End Nodes The right pane of Figure 2 demonstrates the problems that dead-ends can produce for search algorithms. When beam search or LSS-LRTA* fails to find a solution it can be attributed to one of two things happening. Either the algorithm ran out of nodes to expand, or the algorithm ran out of time. As can be seen in Figure 2, when the beam searches and LSS-LRTA* fail to find a solution, they fail quickly which shows that in this domain, the dead end nodes are causing early termination.

In addition to domains that have nodes beneath which no solution exists, there are other domains with no natural dead ends that can easily be partitioned into subspaces. If some of those subspaces have no goal nodes, this is topologically equivalent to a dead end. Beam searches must use a closed list to find solutions, but restricting re-expansions to nodes with improved g can divide the search space into multiple components, some with no goals. Consider the grid path planning problem in the left pane of Figure 3. If B or B' get put on the closed list, the space is partitioned into two regions, one of which has no goals. The collection of nodes B" can also partition the space. If nodes are only being expanded in a partition with no goals, the search algorithm will probably not solve the problem. To escape a partition, at least one node defining the partition has to be re-expanded, but that requires finding a better path to that node. This is unlikely in most domains, and impossible in others.

Similarity of Beam Search to Breadth-First Search One of the pitfalls of breadth-first search is the excessive use of resources. Inadmissible pruning helps alleviate this problem, but breadth-first beam searches still bear a striking resemblance to breadth-first search. In a problem with a perfect heuristic, best-first searches expand only nodes along a path to the goal. A beam search will expand beam width w nodes at each depth. This can be a major disadvantage, as beam search can potentially expand w times as many nodes as a best first search on the same problem. Other algorithms expand one node at a time, so they frequently expand only a few nodes at each level. Beam searches always expand a full beam worth of nodes, even if the nodes are of low quality.

The result of this is that beam searches sometimes expand more nodes than other search algorithms. The right pane of Figure 3 shows the log of the nodes generated on the x-axis, with solution quality on the y-axis. Beam search is expanding an order of magnitude more nodes than the best-first searches. Despite this, the beam search is still completing the search in about the same amount of time as the best-first searches, as can be seen in the center pane of Figure 3. In the pancake domain this is not a big deal, since the time-solution quality trade-off is still reasonable. This is not the case in all domains. In the 100 city TSP, shown in Figure 2, beam search with a beam width of 3 must generate about 15000 nodes in order to reach the bottom of the search tree, which takes about 10 seconds. For comparison, weighted A* is able to generate the minimum 5000 to solve the problem in about 2 seconds.

Not all node generations are equivalent. Depending on the heuristic used, the time it takes to calculate can vary by more than an order of magnitude. For example, the heuristic we used for the TSP and vacuum robot path planning are both based on a minimum spanning tree, and consequently takes much more time to compute when there are a large number of places left still to be visited. Beam searches expand all the nodes near the root until the beam fills up, but a best-first search will expand nodes only one at a time. Best first searches (particularly weighted best first searches) often expand only a few nodes at each level near the root. For the experiments used to generate the left panel of Figure 2 the heuristic involved calculating a minimum spanning tree which requires more time to calculate near the root where there are still a large number of places to visit.

Scaling To Large Instances A major application of unbounded suboptimal search is solving very large problems that can't be solved by A* due to memory constraints, and where IDA* may take too long. Even weighted A* becomes impractical on the 15-puzzle when the weight is small enough. Using a weight of 1.1, the failure rate is 30% within 8 Gb of memory, so we only consider the relatively high weights of 3 and 5.

Table 4 shows the performance of these algorithms on the 15-puzzle and the 48-Puzzle. Relative to the other problems considered, the 48 puzzle has a significantly larger search space, combined with the fact that it is not a fixed-depth combinatorial optimization problem so finding even a poor quality solution is not trivial. Of the algorithms considered, beam search is the clear algorithm of choice. LSS-LRTA*

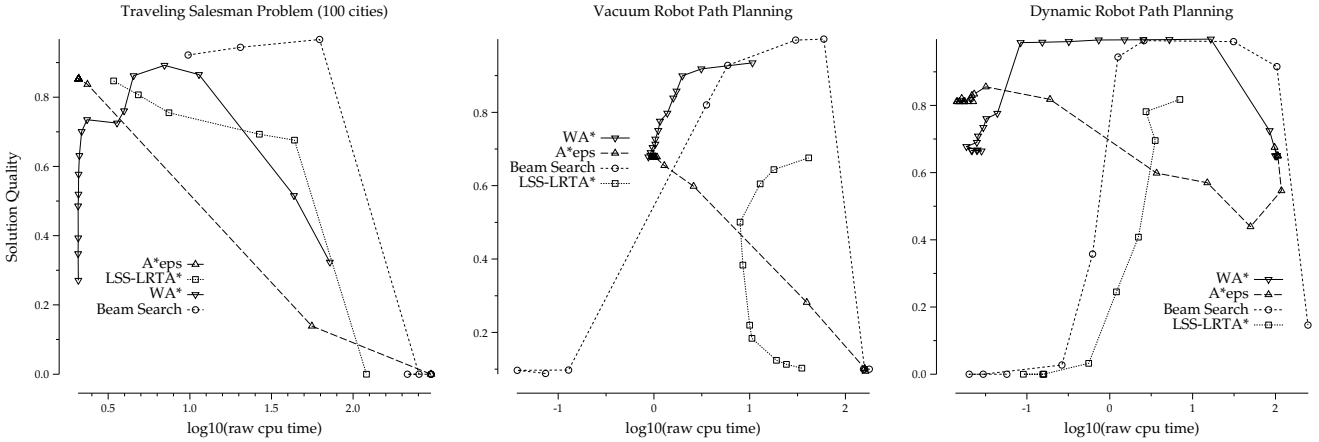

Figure 2: Performance on TSP, Vacuums, and Grid Pathfinding

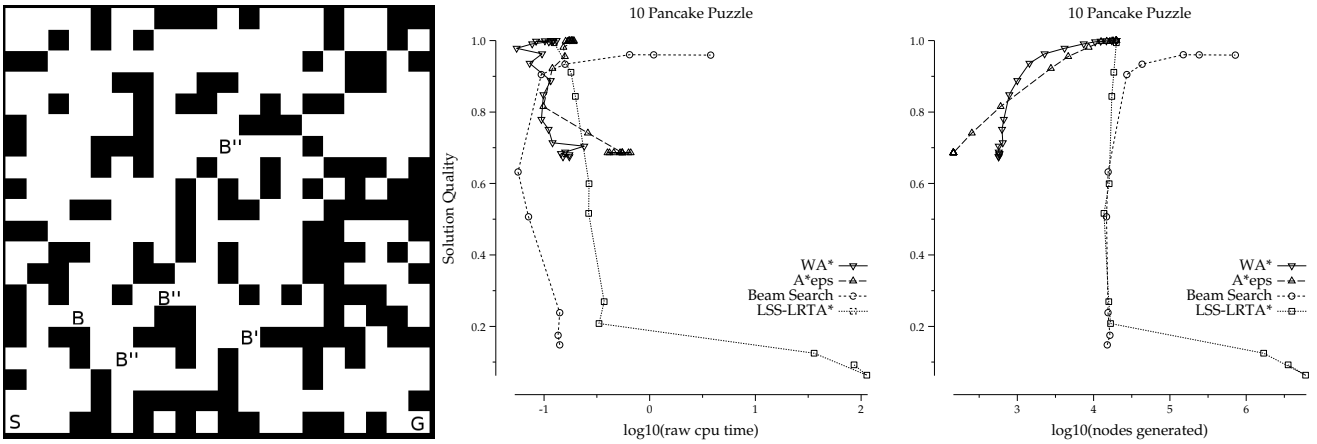

Figure 3: A grid path planning instance and performance on 10 heavy pancakes puzzle

Alg.	15			48		
	F%	Q	T	F%	Q	T
WA*(5)	0	0.8	0.0	66	0.3	228
WA*(3)	0	0.9	0.1	96.1	0.0	291
A*eps(5)	0	0.5	0.0	58.3	0.3	188
A*eps(3)	0	0.6	0.3	98.1	0.0	295
Beam(10)	0	0.4	0.0	*0*	*0.2*	*3.3*
Beam(100)	***0***	***0.9***	***0.0***	**12.6**	**0.5**	**52.2**
LSS-LRTA*(10)	0	0.1	0.0	0	0.1	10.3
LSS-LRTA*(100)	0	0.3	0.1	0	0.2	11.8
Anytime Window A*	0	0.2	0.0	4.9	0.1	27.2

Table 4: Mean performance of selected algorithms on the 15 and 48 sliding tile puzzles

with a look-ahead of 100 can provide solutions of similar quality, but this comes at a cost of increased average run time. Compared to beam search with a beam width of 10, anytime window A* returns higher cost solutions while taking more time, as well as sometimes failing to find any solution. Weighted A* and A^*_ϵ fail more than half the time thus have low average quality with a very high average run time. Table 4 shows that beam search is the algorithm of choice for solving large sliding tile puzzles.

Summary

One of the most important things we can gain from a study such as this one is a set of rules which dictate in a very general sense when each family of algorithm is preferable to one another. From the results of our empirical analysis, we draw the following set of rules.

Weighted A* and A^*_ϵ are the best best-first algorithms. In all our benchmark domains, weighted A* was able to find the highest quality solutions, while A^*_ϵ was often able to find solutions of slightly lower quality in substantially less time. In certain domains MSC-kwA* was competitive, but the performance of this algorithm was erratic when considered across our benchmark domains, and its performance was never significantly better than that of weighted A*.

LSS-LRTA* outperforms enforced hill-climbing. Results in our benchmark domains show that LSS-LRTA* is much more robust than enforced hill-climbing.

Breadth-first beam searches with duplicate re-expansion is the most effective beam search. We considered several variants of beam search, and showed that a breadth-first beam search that re-expands duplicates with better g is the best choice. We found that breadth-first beam searches outperform best-first beam searches due to the nature of error in admissible heuristics. We also found that beam searches

function best when they have a closed list and are allowed to re-expand duplicate nodes. We also observed very unusual scaling behavior if duplicates were pruned due to the fact that in some non-unit cost domains, short and expensive paths can terminate in very high cost solutions. Overall, it was clear that a breadth-first beam search that re-expands duplicates with better g offered the best combination of robustness, solution quality, and scaling behavior of all the beam searches considered.

Dead ends are trouble for hill-climbing and beam search. Since both beam search and LSS-LRTA* commit to including certain states, if either algorithm commits to only states that do not lead to a goal, no solution will be found. Best-first searches considered never commit to including any state, so they are naturally immune to dead ends. Due to the use of a closed list to restrict re-expansions, beam search can run into the same problem if the closed list partitions the space, which can create artificial dead ends.

Massive search spaces require beam search. When scaling up the size of any domain, best-first searches that use an unbounded open list will eventually fail due to either time or memory constraints. Both beam search and LSS-LRTA* can continue to find solutions long after best-first search becomes impractical, but beam search offers a better time-solution quality trade-off. The problems associated with best-first search become clear when we consider the largest problem we ran on, the 48-puzzle. In this domain, best-first searches that maintain a full open list often fail to find solutions because the cost of maintaining an open list becomes prohibitive which causes timeouts.

Beam Search and best-first search are often comparable. Despite radically different performance in certain situations, beam searches and best-first searches often provide time-solution trade-offs that are comparable. In particular, on the 15 puzzle, grid path planning, and the heavy pancake puzzle the performance of the beam search and the best-first searches algorithms was comparable. On domains where the performance is similar, the domain is small enough that the best-first searches can find solutions, and the topology is such that beam search does not get stuck in regions without goals, so we see both algorithms returning comparable time-solution quality curves.

Conclusion

We presented the first comprehensive comparison across algorithm families. This comparison showed us that real-time algorithms, though applicable, should not be used for solving shortest path problems unless there is a need for real-time action. This comparison also showed us that best-first searches are preferable in domains where the goal cannot be reached from all states, and that beam search has the most robust scaling behavior.

This provides the first step towards a clearer understanding of how to deploy greedy search algorithms and how to better design more robust methods for greedy search.

Acknowledgements

We gratefully acknowledge support from NSF (grant IIS-0812141) and the DARPA CSSG program (grant HR0011-09-1-0021). We also thank David Furcy for helpful discussions regarding MSC-kwA* and BULB.

References

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

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

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

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

Furcy, D., and Koenig, S. 2005a. Limited discrepancy beam search. In *Proceedings of IJCAI-05*, 125–131.

Furcy, D., and Koenig, S. 2005b. Scaling up WA* with commitment and diversity. In *IJCAI-05*, 1521–1522.

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.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *JAIR* 14:253–302.

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

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

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*: 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.

Rich, E., and Knight, K. 1991. *Artificial Intelligence*. McGraw-Hill, Incorporated.

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

Thayer, J. T., and Ruml, W. 2008. Faster than weighted A*: An optimistic approach to bounded suboptimal search. In *Proceedings of ICAPS-08*.

Thayer, J. T.; Ruml, W.; and Bitton, E. 2008. Fast and loose in bounded suboptimal heuristic search. In *Proceedings of STAIR-08*.

Zhang, W. 1998. Complete anytime beam search. In *Proceedings of AAAI-98*, 425–430.

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