A Guide to Heuristic-based Path Planning

Dave Ferguson, Maxim Likhachev, and Anthony Stentz
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA

Abstract

We describe a family of recently developed heuristic-based algorithms used for path planning in the real world. We discuss the fundamental similarities between static algorithms (e.g. A*), replanning algorithms (e.g. D*), anytime algorithms (e.g. ARA*), and anytime replanning algorithms (e.g. AD*). We introduce the motivation behind each class of algorithms, discuss their use on real robotic systems, and highlight their practical benefits and disadvantages.

Introduction

In this paper, we describe a family of heuristic-based planning algorithms that has been developed to address various challenges associated with planning in the real world. Each of the algorithms presented have been verified on real systems operating in real domains. However, a prerequisite for the successful general use of such algorithms is (1) an analysis of the common fundamental elements of such algorithms, (2) a discussion of their strengths and weaknesses, and (3) guidelines for when to choose a particular algorithm over others. Although these algorithms have been documented and described individually, a comparative analysis of these algorithms is lacking in the literature. With this paper we hope to fill this gap.

We begin by providing background on path planning in static, known environments and classical algorithms used to generate plans in this domain. We go on to look at how these algorithms can be extended to efficiently cope with partially-known or dynamic environments. We then introduce variants of these algorithms that can produce suboptimal solutions very quickly when time is limited and improve these solutions while time permits. Finally, we discuss an algorithm that combines principles from all of the algorithms previously discussed; this algorithm can plan in dynamic environments *and* with limited deliberation time. For all the algorithms discussed in this paper, we provide example problem scenarios in which they are very effective and situations in which they are less effective. Although our primary focus is on path planning, several of these algorithms are applicable in more general planning scenarios.

Our aim is to share intuition and lessons learned over the course of several system implementations and guide readers in choosing algorithms for their own planning domains.

Path Planning

Planning consists of finding a sequence of actions that transforms some initial state into some desired goal state. In path planning, the states are agent locations and transitions between states represent actions the agent can take, each of which has an associated cost. A path is *optimal* if the sum of its transition costs (edge costs) is minimal across all possible paths leading from the initial position (start state) to the goal position (goal state). A planning algorithm is *complete* if it will always find a path in finite time when one exists, and will let us know in finite time if no path exists. Similarly, a planning algorithm is optimal if it will always find an optimal path.

Several approaches exist for computing paths given some representation of the environment. In general, the two most popular techniques are deterministic, heuristic-based algorithms (Hart, Nilsson, & Rafael 1968; Nilsson 1980) and randomized algorithms (Kavraki *et al.* 1996; LaValle 1998; LaValle & Kuffner 1999; 2001).

When the dimensionality of the planning problem is low, for example when the agent has only a few degrees of freedom, deterministic algorithms are usually favored because they provide bounds on the quality of the solution path returned. In this paper, we concentrate on deterministic algorithms. For more details on probabilistic techniques, see (LaValle 2005).

A common technique for robotic path planning consists of representing the environment (or configuration space) of the robot as a graph $G = (S, E)$, where S is the set of possible robot locations and E is a set of edges that represent transitions between these locations. The cost of each edge represents the cost of transitioning between the two endpoint locations.

Planning a path for navigation can then be cast as a search problem on this graph. A number of classical graph search algorithms have been developed for calculating least-cost paths on a weighted graph; two popular ones are Dijkstra's algorithm (Dijkstra 1959) and A* (Hart, Nilsson, & Rafael 1968; Nilsson 1980). Both algorithms return an optimal path (Gelperin 1977), and can be considered as special forms of

```
**ComputeShortestPath**()
01. while (argmin$_{s \in OPEN}(g(s) + h(s, s_{goal})) \neq s_{goal}$)
02.     remove state $s$ from the front of *OPEN*;
03.     for all $s' \in Succ(s)$
04.       if ($g(s') > g(s) + c(s, s')$)
05.           $g(s') = g(s) + c(s, s')$;
06.           insert $s'$ into *OPEN* with value ($g(s') + h(s', s_{goal})$);

**Main**()
07. for all $s \in S$
08.     $g(s) = \infty$;
09. $g(s_{start}) = 0$;
10. *OPEN* $= \emptyset$;
11. insert $s_{start}$ into *OPEN* with value ($g(s_{start}) + h(s_{start}, s_{goal})$);
12. ComputeShortestPath();
```

Figure 1: **The A* Algorithm (forwards version).**

Figure 2: D* and its variants are currently used for path planning on several robotic systems, including indoor planar robots (Pioneers) and outdoor robots operating in more challenging terrain (E-Gators).

dynamic programming (Bellman 1957). A* operates essentially the same as Dijkstra's algorithm except that it guides its search towards the most promising states, potentially saving a significant amount of computation.

A* plans a path from an initial state $s_{start} \in S$ to a goal state $s_{goal} \in S$, where S is the set of states in some finite state space. To do this, it stores an estimate $g(s)$ of the path cost from the initial state to each state s. Initially, $g(s) = \infty$ for all states $s \in S$. The algorithm begins by updating the path cost of the start state to be zero, then places this state onto a priority queue known as the *OPEN* list. Each element s in this queue is ordered according to the sum of its current path cost from the start, $g(s)$, and a heuristic estimate of its path cost to the goal, $h(s, s_{goal})$. The state with the minimum such sum is at the front of the priority queue. The heuristic $h(s, s_{goal})$ typically underestimates the cost of the optimal path from s to s_{goal} and is used to focus the search.

The algorithm then pops the state s at the front of the queue and updates the cost of all states reachable from this state through a direct edge: if the cost of state s, $g(s)$, plus the cost of the edge between s and a neighboring state s', $c(s, s')$, is less than the current cost of state s', then the cost of s' is set to this new, lower value. If the cost of a neighboring state s' changes, it is placed on the *OPEN* list. The algorithm continues popping states off the queue until it pops off the goal state. At this stage, if the heuristic is *admissible*, i.e. guaranteed to not overestimate the path cost from any state to the goal, then the path cost of s_{goal} is guaranteed to be optimal. The complete algorithm is given in Figure 1.

It is also possible to switch the direction of the search in A*, so that planning is performed from the goal state towards the start state. This is referred to as 'backwards' A*, and will be relevant for some of the algorithms discussed in the following sections.

Incremental Replanning Algorithms

The above approaches work well for planning an initial path through a known graph or planning space. However, when operating in real world scenarios, agents typically do not have perfect information. Rather, they may be equipped with incomplete or inaccurate planning graphs. In such cases, any path generated using the agent's initial graph may turn out to be invalid or suboptimal as it receives updated information. For example, in robotics the agent may be equipped with an onboard sensor that provides updated environment information as the agent moves. It is thus important that the agent is able to update its graph and replan new paths when new information arrives.

One approach for performing this replanning is simply to replan from scratch: given the updated graph, a new optimal path can be planned from the robot position to the goal using A*, exactly as described above. However, replanning from scratch every time the graph changes can be very computationally expensive. For instance, imagine that a change occurs in the graph that does not affect the optimality of the current solution path. Or, suppose some change takes place that does affect the current solution, but in a minor way that can be quickly fixed. Replanning from scratch in either of these situations seems like a waste of computation. Instead, it may be far more efficient to take the previous solution and repair it to account for the changes to the graph.

A number of algorithms exist for performing this repair (Stentz 1994; 1995; Barbehenn & Hutchinson 1995; Ramalingam & Reps 1996; Ersson & Hu 2001; Huiming *et al.* 2001; Podsedkowski *et al.* 2001; Koenig & Likhachev 2002). Focussed Dynamic A* (D*) (Stentz 1995) and D* Lite (Koenig & Likhachev 2002) are currently the most widely used of these algorithms, due to their efficient use of heuristics and incremental updates. They have been used for path planning on a large assortment of robotic systems, including both indoor and outdoor platforms (Stentz & Hebert 1995; Hebert, McLachlan, & Chang 1999; Matthies *et al.* 2000; Thayer *et al.* 2000; Zlot *et al.* 2002; Likhachev 2003) (see Figure 2). They have also been extended to provide incremental replanning behavior in symbolic planning domains (Koenig, Furcy, & Bauer 2002).

D* and D* Lite are extensions of A* able to cope with changes to the graph used for planning. The two algorithms are fundamentally very similar; we restrict our attention here to D* Lite because it is simpler and has been found to be slightly more efficient for some navigation tasks (Koenig & Likhachev 2002). D* Lite initially constructs an optimal solution path from the initial state to the goal state in exactly the same manner as backwards A*. When changes to the planning graph are made (i.e., the cost of some edge is altered), the states whose paths to the goal are immediately affected by these changes have their path costs updated and are placed on the planning queue (*OPEN* list) to propagate the effects of these changes to the rest of the state space. In this way, only the affected portion of the state space is processed when changes occur. Furthermore, D* Lite uses a heuristic to further limit the states processed to only those states whose change in path cost could have a bearing on the path cost of the initial state. As a result, it can be up to two orders of magnitude more efficient than planning from scratch using A* (Koenig & Likhachev 2002).

In more detail, D* Lite maintains a least-cost path from a start state $s_{start} \in S$ to a goal state $s_{goal} \in S$, where S is again the set of states in some finite state space. To do this, it stores an estimate $g(s)$ of the cost from each state s to the goal. It also stores a one-step lookahead cost $rhs(s)$ which satisfies:

$$rhs(s) = \begin{cases} 0 & \text{if } s = s_{goal} \\ \min_{s' \in Succ(s)}(c(s,s') + g(s')) & \text{otherwise,} \end{cases}$$

where $Succ(s) \in S$ denotes the set of successors of s and $c(s,s')$ denotes the cost of moving from s to s' (the edge cost). A state is called consistent iff its g-value equals its rhs-value, otherwise it is either overconsistent (if $g(s) > rhs(s)$) or underconsistent (if $g(s) < rhs(s)$).

Like A*, D* Lite uses a heuristic and a priority queue to focus its search and to order its cost updates efficiently. The heuristic $h(s,s')$ estimates the cost of moving from state s to s', and needs to be admissible and (backward) consistent: $h(s,s') \leq c^*(s,s')$ and $h(s,s'') \leq h(s,s') + c^*(s',s'')$ for all states $s, s', s'' \in S$, where $c^*(s,s')$ is the cost associated with a least-cost path from s to s'. The priority queue *OPEN* always holds exactly the inconsistent states; these are the states that need to be updated and made consistent.

The priority, or *key value*, of a state s in the queue is:

$$\begin{aligned} key(s) &= [k_1(s), k_2(s)] \\ &= [\min(g(s), rhs(s)) + h(s_{start}, s), \\ &\quad \min(g(s), rhs(s))]. \end{aligned}$$

A lexicographic ordering is used on the priorities, so that priority $key(s)$ is less than or equal to priority $key(s')$, denoted $key(s) \dot{\leq} key(s')$, iff $k_1(s) < k_1(s')$ or both $k_1(s) = k_1(s')$ and $k_2(s) \leq k_2(s')$. D* Lite expands states from the queue in increasing priority, updating their g-values and their predecessors' rhs-values, until there is no state in the queue with a priority less than that of the start state. Thus, during its generation of an initial solution path, it performs in exactly the same manner as a backwards A* search.

To allow for the possibility that the start state may change over time D* Lite searches backwards and consequently focusses its search towards the start state rather than the goal state. If the g-value of each state s was based on a least-cost path from s_{start} to s (as in forward search) rather than from s to s_{goal}, then when the robot moved every state would have to have its cost updated. Instead, with D* Lite only the heuristic value associated with each inconsistent state needs to be updated when the robot moves. Further, even this step can be avoided by adding a bias to newly inconsistent states being added to the queue (see (Stentz 1995) for details).

```
**key**($s$)
01. return $[min(g(s), rhs(s)) + h(s_{start}, s); min(g(s), rhs(s)))]$;

**UpdateState**($s$)
02. if $s$ was not visited before
03.     $g(s) = \infty$;
04. if $(s \neq s_{goal})$ $rhs(s) = min_{s' \in Succ(s)}(c(s,s') + g(s'))$;
05. if ($s \in$ OPEN) remove $s$ from OPEN;
06. if $(g(s) \neq rhs(s))$ insert $s$ into OPEN with key($s$);

**ComputeShortestPath**()
07. while ($\min_{s \in OPEN}$(key($s$)) $\dot{<}$ key($s_{start}$) OR $rhs(s_{start}) \neq g(s_{start})$)
08.     remove state $s$ with the minimum key from OPEN;
09.     if $(g(s) > rhs(s))$
10.         $g(s) = rhs(s)$;
11.         for all $s' \in Pred(s)$ UpdateState($s'$);
12.     else
13.         $g(s) = \infty$;
14.         for all $s' \in Pred(s) \cup \{s\}$ UpdateState($s'$);

**Main**()
15. $g(s_{start}) = rhs(s_{start}) = \infty$; $g(s_{goal}) = \infty$;
16. $rhs(s_{goal}) = 0$; OPEN $= \emptyset$;
17. insert $s_{goal}$ into OPEN with key($s_{goal}$);
18. forever
19.     ComputeShortestPath();
20.     Wait for changes in edge costs;
21.     for all directed edges $(u, v)$ with changed edge costs
22.         Update the edge cost $c(u, v)$;
23.         UpdateState($u$);
```

Figure 3: **The D* Lite Algorithm (basic version).**

When edge costs change, D* Lite updates the rhs-values of each state immediately affected by the changed edge costs and places those states that have been made inconsistent onto the queue. As before, it then expands the states on the queue in order of increasing priority until there is no state in the queue with a priority less than that of the start state. By incorporating the value $k_2(s)$ into the priority for state s, D* Lite ensures that states that are along the current path and on the queue are processed in the right order. Combined with the termination condition, this ordering also ensures that a least-cost path will have been found from the start state to the goal state when processing is finished. The basic version of the algorithm (for a fixed start state) is given in Figure 3[1].

D* Lite is efficient because it uses a heuristic to restrict attention to only those states that could possibly be relevant to repairing the current solution path from a given start state to the goal state. When edge costs decrease, the incorporation of the heuristic in the key value (k_1) ensures that only those newly-overconsistent states that could potentially decrease the cost of the start state are processed. When edge costs increase, it ensures that only those newly-underconsistent states that could potentially invalidate the current cost of the start state are processed.

In some situations the process of invalidating old costs

[1]Because the optimizations of D* Lite presented in (Koenig & Likhachev 2002) can significantly speed up the algorithm, for an efficient implementation of D* Lite please refer to that paper.

may be unnecessary for repairing a least-cost path. For example, such is the case when there are no edge cost decreases and all edge cost increases happen outside of the current least-cost path. To guarantee optimality in the future, D* Lite would still invalidate portions of the old search tree that are affected by the observed edge cost changes even though it is clear that the old solution remains optimal. To overcome this a modified version of D* Lite has recently been proposed that delays the propagation of cost increases as long as possible while still guaranteeing optimality. Delayed D* (Ferguson & Stentz 2005) is an algorithm that initially ignores underconsistent states when changes to edge costs occur. Then, after the new values of the overconsistent states have been adequately propagated through the state space, the resulting solution path is checked for any underconsistent states. All underconsistent states on the path are added to the *OPEN* list and their updated values are propagated through the state space. Because the current propagation phase may alter the solution path, the new solution path needs to be checked for underconsistent states. The entire process repeats until a solution path that contains only consistent states is returned.

Applicability: Replanning Algorithms

Delayed D* has been shown to be significantly more efficient than D* Lite in certain domains (Ferguson & Stentz 2005). Typically, it is most appropriate when there is a relatively large distance between the start state and the goal state, and changes are being observed in arbitrary locations in the graph (for example, map updates are received from a satellite). This is because it is able to ignore the edge cost increases that do not involve its current solution path, which in these situations can lead to a dramatic decrease in overall computation. When a robot is moving towards a goal in a completely unknown environment, Delayed D* will not provide much benefit over D* Lite, as in this scenario typically the costs of only few states outside of the current least-cost path have been computed and therefore most edge cost increases will be ignored by both algorithms. There are also scenarios in which Delayed D* will do more processing than D* Lite: imagine a case where the processing of underconsistent states changes the solution path several times, each time producing a new path containing underconsistent states. This results in a number of replanning phases, each potentially updating roughly the same area of the state space, and will be far less efficient than dealing with all the underconsistent states in a single replanning episode. However, in realistic navigation scenarios, such situations are very rare.

In practise, both D* Lite and Delayed D* are very effective for replanning in the context of mobile robot navigation. Typically, in such scenarios the changes to the graph are happening close to the robot (through its observations), which means their effects are usually limited. When this is the case, using an incremental replanner such as D* Lite will be far more efficient than planning from scratch. However, this is not universally true. If the areas of the graph being changed are not necessarily close to the position of the robot, it is possible for D* Lite to be *less* efficient than A*. This is because it is possible for D* Lite to process every state in the environment twice: once as an underconsistent state and once as an overconsistent state. A*, on the other hand, will only ever process each state once. The worst-case scenario for D* Lite, and one that illustrates this possibility, is when changes are being made to the graph in the vicinity of the goal. It is thus common for systems using D* Lite to abort the replanning process and plan from scratch whenever either major edge cost changes are detected or some predefined threshold of replanning processing is reached.

Also, when navigating through completely unknown environments, it can be much more efficient to search forwards from the agent position to the goal, rather than backwards from the goal. This is because we typically assign optimistic costs to edges whose costs we don't know. As a result, areas of the graph that have been observed have more expensive edge costs than the unexplored areas. This means that, when searching forwards, as soon as the search exits the observed area it can rapidly progress through the unexplored area directly to the goal. However, when searching backwards, the search initially rapidly progresses to the observed area, then once it encounters the more costly edges in the observed area, it begins expanding large portions of the unexplored area trying to find a cheaper path. As a result, it can be significantly more efficient to use A* rather than backwards A* when replanning from scratch. Because the agent is moving, it is not possible to use a forwards-searching incremental replanner, which means that the computational advantage of using a replanning algorithm over planning from scratch is reduced.

As mentioned earlier, these algorithms can also be applied to symbolic planning problems (Koenig, Furcy, & Bauer 2002; Liu, Koenig, & Furcy 2002). However, in these cases it is important to consider whether there is an available predecessor function in the particular planning domain. If not, it is necessary to maintain for each state s the set of all states s' that have used s as a successor state during the search, and treat this set as the set of predessors of s. This is also useful when such a predecessor function exists but contains a very large number of states; maintaining a list of just the states that have actually used s as a successor can be far more efficient than generating all the possible predecessors.

In the symbolic planning community it is also common to use inconsistent heuristics since problems are often infeasible to solve optimally. The extensions to D* Lite presented in (Likhachev & Koenig 2005) enable D* Lite to handle inconsistent heuristics. These extensions also allow one to vary the tie-breaking criteria when selecting states from the *OPEN* list for processing. This might be important when a problem has many solutions of equal costs and the *OPEN* list contains a large number of states with the same priorities.

Apart from the static approaches (Dijkstra's, A*), all of the algorithms that we discuss in this paper attempt to reuse previous results to make subsequent planning tasks easier. However, if the planning problem has changed sufficiently since the previous result was generated, this result may be a burden rather than a useful starting point.

For instance, it is possible in symbolic domains that altering the cost of a single operator may affect the path cost of a huge number of states. As an example, modifying the cost of the *load* operator in the rocket domain may completely change the nature of the solution. This can also be a problem when path planning for robots with several degrees of freedom: even if a small change occurs in the environment, it can cause a huge number of changes in the complex configuration space. As a result, replanning in such scenarios can often be of little or no benefit.

Anytime Algorithms

When an agent must react quickly and the planning problem is complex, computing optimal paths as described in the previous sections can be infeasible, due to the sheer number of states required to be processed in order to obtain such paths. In such situations, we must be satisfied with the best solution that can be generated in the time available.

A useful class of deterministic algorithms for addressing this problem are commonly referred to as *anytime* algorithms. Anytime algorithms typically construct an initial, possibly highly suboptimal, solution very quickly, then improve the quality of this solution while time permits (Zilberstein & Russell 1995; Dean & Boddy 1988; Zhou & Hansen 2002; Likhachev, Gordon, & Thrun 2003; Horvitz 1987). Heuristic-based anytime algorithms often make use of the fact that in many domains inflating the heuristic values used by A* (resulting in the weighted A* search) often provides substantial speed-ups at the cost of solution optimality (Bonet & Geffner 2001; Korf 1993; Zhou & Hansen 2002; Edelkamp 2001; Rabin 2000; Chakrabarti, Ghosh, & DeSarkar 1988). Further, if the heuristic used is consistent[2], then multiplying it by an inflation factor $\epsilon > 1$ will produce a solution guaranteed to cost no more than ϵ times the cost of an optimal solution. Likhachev, Gordon, and Thrun use this property to develop an anytime algorithm that performs a succession of weighted A* searches, each with a decreasing inflation factor, where each search reuses efforts from previous searches (Likhachev, Gordon, & Thrun 2003). Their approach provides suboptimality bounds for each successive search and has been shown to be much more efficient than competing approaches (Likhachev, Gordon, & Thrun 2003).

Likhachev et al.'s algorithm, Anytime Repairing A* (ARA*), limits the processing performed during each search by only considering those states whose costs at the previous search may not be valid given the new ϵ value. It begins by performing an A* search with an inflation factor ϵ_0, but during this search it only expands each state at most once[3]. Once a state s has been expanded during a particular search, if it becomes inconsistent (i.e., $g(s) \neq rhs(s)$) due to a cost change associated with a neighboring state, then it is not reinserted into the queue of states to be expanded. Instead, it is placed into the *INCONS* list, which contains all inconsistent states already expanded. Then, when the current search terminates, the states in the *INCONS* list are inserted into a

[2]A (forwards) heuristic h is consistent if, for all $s \in S$, $h(s, s_{goal}) \leq c(s, s') + h(s', s_{goal})$ for any successor s' of s, and $h(s_{goal}, s_{goal}) = 0$.

[3]It is proved in (Likhachev, Gordon, & Thrun 2003) that this still guarantees an ϵ_0 suboptimality bound.

```
key(s)
  01. return $g(s) + \epsilon \cdot h(s_{start}, s)$;

ImprovePath()
  02. while ($\min_{s \in OPEN}$(key(s)) < key($s_{start}$))
  03.   remove $s$ with the smallest key($s$) from OPEN;
  04.   $CLOSED = CLOSED \cup \{s\}$;
  05.   for all $s' \in Pred(s)$
  06.     if $s'$ was not visited before
  07.       $g(s') = \infty$;
  08.     if $g(s') > c(s', s) + g(s)$
  09.       $g(s') = c(s', s) + g(s)$;
  10.       if $s' \notin CLOSED$
  11.         insert $s'$ into OPEN with key($s'$);
  12.       else
  13.         insert $s'$ into INCONS;

Main()
  14. $g(s_{start}) = \infty$; $g(s_{goal}) = 0$;
  15. $\epsilon = \epsilon_0$;
  16. $OPEN = CLOSED = INCONS = \emptyset$;
  17. insert $s_{goal}$ into OPEN with key($s_{goal}$);
  18. ImprovePath();
  19. publish current $\epsilon$-suboptimal solution;
  20. while $\epsilon > 1$
  21.   decrease $\epsilon$;
  22.   Move states from INCONS into OPEN;
  23.   Update the priorities for all $s \in OPEN$ according to key($s$);
  24.   $CLOSED = \emptyset$;
  25.   ImprovePath();
  26.   publish current $\epsilon$-suboptimal solution;
```

Figure 4: **The ARA* Algorithm (backwards version).**

fresh priority queue (with new priorities based on the new ϵ inflation factor) which is used by the next search. This improves the efficiency of each search in two ways. Firstly, by only expanding each state at most once a solution is reached much more quickly. Secondly, by only reconsidering states from the previous search that were inconsistent, much of the previous search effort can be reused. Thus, when the inflation factor is reduced between successive searches, a relatively minor amount of computation is required to generate a new solution.

A simplified, backwards-searching version of the algorithm is given in Figure 4[4]. Here, the priority of each state s in the *OPEN* queue is computed as the sum of its cost $g(s)$ and its inflated heuristic value $\epsilon \cdot h(s_{start}, s)$. *CLOSED* contains all states already expanded once in the current search, and *INCONS* contains all states that have already been expanded and are inconsistent.

Applicability: Anytime Algorithms

ARA* has been shown to be much more efficient than competing approaches and has been applied successfully to path planning in high-dimensional state spaces, such as kinematic robot arms with 20 links (Likhachev, Gordon, & Thrun 2003). It has thus effectively extended the applicability of

[4]The backwards-searching version is shown because it will be useful when discussing the algorithm's similarity to D* Lite.

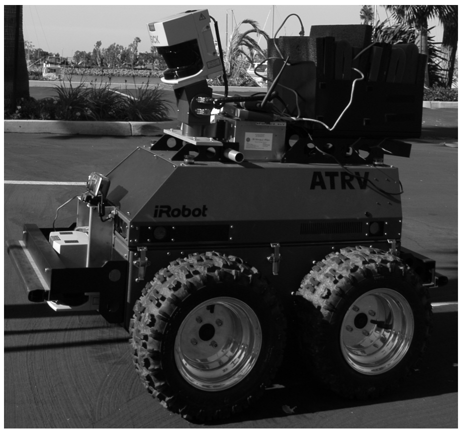

Figure 5: The ATRV robotic platform.

deterministic planning algorithms into much higher dimensions than previously possible. It has also been used to plan smooth trajectories for outdoor mobile robots in known environments. Figure 5 shows an outdoor robotic system that has used ARA* for this purpose. Here, the search space involved four dimensions: the (x, y) position of the robot, the robot's orientation, and the robot's velocity. ARA* is able to plan suboptimal paths for the robot very quickly, then improve the quality of these paths as the robot begins its traverse (as the robot moves the start state changes and therefore in between search iterations the heuristics are recomputed for all states in the *OPEN* list right before their priorites are updated).

ARA* is well suited to domains in which the state space is very large and suboptimal solutions can be generated efficiently. Although using an inflation factor ϵ usually expedites the planning process, this is not guaranteed. In fact, it is possible to construct pathological examples where the best-first nature of searching with a large ϵ can result in much longer processing times. The larger ϵ is, the more greedy the search through the space is, leaving it more prone to getting temporarily stuck in local minima. In general, the key to obtaining anytime behavior with ARA* is finding a heuristic function with shallow local minima. For example, in the case of robot navigation a local minimum can be a U-shaped obstacle placed on the straight line connecting a robot to its goal (assuming the heuristic function is Euclidean distance) and the size of the obstacle determines how many states weighted A*, and consequently ARA*, will have to process before getting out of the minimum.

Depending on the domain one can also augment ARA* with a few optimizations. For example, in graphs with considerable branching factors the *OPEN* list can grow prohibitively large. In such cases, one can borrow an interesting idea from (Zhou & Hansen 2002) and prune (and never insert) the states from the *OPEN* list whose priorities based on un-inflated heuristic are already larger than the cost of the current solution (e.g., $g(s_{goal})$ in the forwards-searching version).

However, because ARA* is an anytime algorithm, it is only useful when an anytime solution is desired. If a solution with a particular suboptimality bound of ϵ_d is desired, and no intermediate solution matters, then it is far more efficient to perform a weighted A* search with an inflation factor of ϵ_d than to use ARA*.

Further, ARA* is only applicable in static planning domains. If changes are being made to the planning graph, ARA* is unable to reuse its previous search results and must replan from scratch. As a result, it is not appropriate for dynamic planning problems. It is this limitation that motivated research into the final set of algorithms we discuss here: anytime replanners.

Anytime Replanning Algorithms

Although each is well developed on its own, there has been relatively little interaction between the above two areas of research. Replanning algorithms have concentrated on finding a single, usually optimal, solution, and anytime algorithms have concentrated on static environments. But some of the most interesting real world problems are those that are both dynamic (requiring replanning) *and* complex (requiring anytime approaches).

As a motivating example, consider motion planning for a kinematic arm in a populated office area. A planner for such a task would ideally be able to replan efficiently when new information is received indicating that the environment has changed. It would also need to generate suboptimal solutions, as optimality may not be possible given limited deliberation time.

Recently, Likhachev et al. developed Anytime Dynamic A* (AD*), an algorithm that combines the replanning capability of D* Lite with the anytime performance of ARA* (Likhachev *et al.* 2005). AD* performs a series of searches using decreasing inflation factors to generate a series of solutions with improved bounds, as with ARA*. When there are changes in the environment affecting the cost of edges in the graph, locally affected states are placed on the *OPEN* queue to propagate these changes through the rest of the graph, as with D* Lite. States on the queue are then processed until the solution is guaranteed to be ϵ-suboptimal.

The algorithm is presented in Figures 6 and 7[5]. AD* begins by setting the inflation factor ϵ to a sufficiently high value ϵ_0, so that an initial, suboptimal plan can be generated quickly. Then, unless changes in edge costs are detected, ϵ is gradually decreased and the solution is improved until it is guaranteed to be optimal, that is, $\epsilon = 1$. This phase is exactly the same as for ARA*: each time ϵ is decreased, all inconsistent states are moved from *INCONS* to *OPEN* and *CLOSED* is made empty.

When changes in edge costs are detected, there is a chance that the current solution will no longer be ϵ-suboptimal. If the changes are substantial, then it may be computationally expensive to repair the current solution to regain ϵ-suboptimality. In such a case, the algorithm increases ϵ so

[5]As with D* Lite the optimizations presented in (Koenig & Likhachev 2002) can be used to substantially speed up AD* and are recommended for an efficient implementation of the algorithm.

```
key($s$)
01. if ($g(s) > rhs(s)$)
02.     return [$min(g(s), rhs(s)) + \epsilon \cdot h(s_{start}, s); min(g(s), rhs(s))$];
03. else
04.     return [$min(g(s), rhs(s)) + h(s_{start}, s); min(g(s), rhs(s))$];

UpdateState($s$)
05. if $s$ was not visited before
06.     $g(s) = \infty$;
07. if ($s \neq s_{goal}$) $rhs(s) = min_{s' \in Succ(s)}(c(s, s') + g(s'))$;
08. if ($s \in$ OPEN) remove $s$ from OPEN;
09. if ($g(s) \neq rhs(s)$)
10.     if $s \notin$ CLOSED
11.         insert $s$ into OPEN with key($s$);
12.     else
13.         insert $s$ into INCONS;

ComputeorImprovePath()
14. while ($\min_{s \in OPEN}$(key($s$)) $\dot{<}$ key($s_{start}$) OR $rhs(s_{start}) \neq g(s_{start})$)
15.     remove state $s$ with the minimum key from OPEN;
16.     if ($g(s) > rhs(s)$)
17.         $g(s) = rhs(s)$;
18.         CLOSED = CLOSED $\cup \{s\}$;
19.         for all $s' \in Pred(s)$ UpdateState($s'$);
20.     else
21.         $g(s) = \infty$;
22.         for all $s' \in Pred(s) \cup \{s\}$ UpdateState($s'$);
```

Figure 6: **Anytime Dynamic A*: ComputeorImprovePath function.**

```
Main()
01. $g(s_{start}) = rhs(s_{start}) = \infty; g(s_{goal}) = \infty$;
02. $rhs(s_{goal}) = 0; \epsilon = \epsilon_0$;
03. OPEN = CLOSED = INCONS = $\emptyset$;
04. insert $s_{goal}$ into OPEN with key($s_{goal}$);
05. ComputeorImprovePath();
06. publish current $\epsilon$-suboptimal solution;
07. forever
08.     if changes in edge costs are detected
09.         for all directed edges $(u, v)$ with changed edge costs
10.             Update the edge cost $c(u, v)$;
11.             UpdateState($u$);
12.     if significant edge cost changes were observed
13.         increase $\epsilon$ or replan from scratch;
14.     else if $\epsilon > 1$
15.         decrease $\epsilon$;
16.     Move states from INCONS into OPEN;
17.     Update the priorities for all $s \in$ OPEN according to key($s$);
18.     CLOSED = $\emptyset$;
19.     ComputeorImprovePath();
20.     publish current $\epsilon$-suboptimal solution;
21.     if $\epsilon = 1$
22.         wait for changes in edge costs;
```

Figure 7: **Anytime Dynamic A*: Main function.**

that a less optimal solution can be produced quickly. Because edge cost increases may cause some states to become underconsistent, a possibility not present in ARA*, states need to be inserted into the *OPEN* queue with a key value reflecting the minimum of their old cost and their new cost. Further, in order to guarantee that underconsistent states propagate their new costs to their affected neighbors, their key values must use admissible heuristic values. This means that different key values must be computed for underconsistent states than for overconsistent states.

By incorporating these considerations, AD* is able to handle both changes in edge costs and changes to the inflation factor ϵ. Like the replanning and anytime algorithms we've looked at, it can also be slightly modified to handle the situation where the start state s_{start} is changing, as is the case when the path is being traversed by an agent. This allows the agent to improve and update its solution path while it is being traversed.

An Example[6]

Figure 8 presents an illustration of each of the approaches described in the previous sections employed on a simple grid world planning problem. In this example we have an eight-connected grid where black cells represent obstacles and white cells represent free space. The cell marked R denotes the position of an agent navigating this environment towards a goal cell, marked G (in the upper left corner of the grid world). The cost of moving from one cell to any non-obstacle neighboring cell is one. The heuristic used by each algorithm is the larger of the x (horizontal) and y (vertical) distances from the current cell to the cell occupied by the agent. The cells expanded by each algorithm for each subsequent agent position are shown in grey. The resulting paths are shown as grey arrows.

The first approach shown is (backwards) A*. The initial search performed by A* provides an optimal path for the agent. After the agent takes two steps along this path, it receives information indicating that one of the cells in the top wall is in fact free space. It then replans from scratch using A* to generate a new, optimal path to the goal. The combined total number of cells expanded at each of the first three agent positions is 31.

The second approach is A* with an inflation factor of $\epsilon = 2.5$. This approach produces an initial suboptimal solution very quickly. When the agent receives the new information regarding the top wall, this approach replans from scratch using its inflation factor and produces a new path (which happens to be optimal). The total number of cells expanded is only 19, but the solution is only guaranteed to be ϵ-suboptimal at each stage.

The third approach is D* Lite, and the fourth is D* Lite with an inflation factor of $\epsilon = 2.5$. The bounds on the quality of the solutions returned by these respective approaches are equivalent to those returned by the first two. However, because D* Lite reuses previous search results, it is able to produce its solutions with far fewer overall cell expansions.

[6]This example and the ensuing discussion are borrowed from (Likhachev *et al.* 2005).

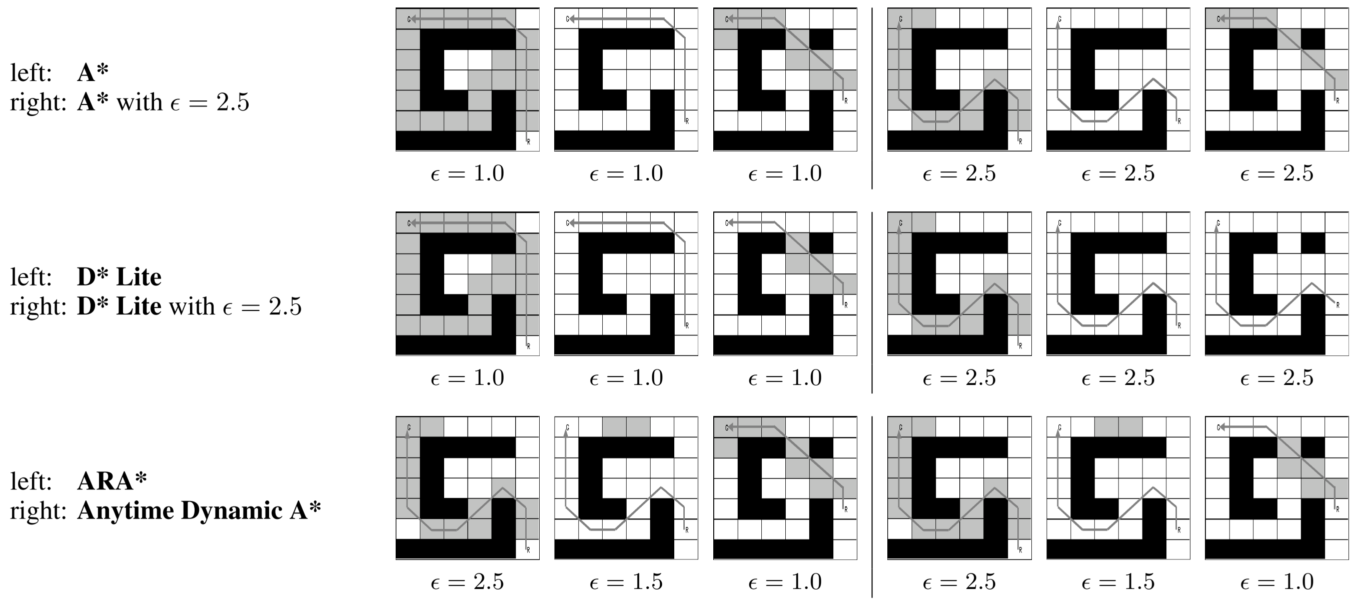

Figure 8: A simple robot navigation example. The robot starts in the bottom right cell and plans a path to the upper left cell. After it has moved two steps along its path, it observes a gap in the top wall. The states expanded by each of six algorithms (A*, A* with an inflation factor, D* Lite, D* Lite with an inflation factor, ARA*, and AD*) are shown at each of the first three robot positions. Example borrowed from (Likhachev *et al.* 2005).

D* Lite without an inflation factor expands 27 cells (almost all in its initial solution generation) and always maintains an optimal solution, and D* Lite with an inflation factor of 2.5 expands 13 cells but produces solutions that are suboptimal every time it replans.

The final row of the figure shows the results of (backwards) ARA* and AD*. Each of these approaches begins by computing a suboptimal solution using an inflation factor of $\epsilon = 2.5$. While the agent moves one step along this path, this solution is improved by reducing the value of ϵ to 1.5 and reusing the results of the previous search. The path cost of this improved result is guaranteed to be at most 1.5 times the cost of an optimal path. Up to this point, both ARA* and AD* have expanded the same 15 cells each. However, when the robot moves one more step and finds out the top wall is broken, each approach reacts differently. Because ARA* cannot incorporate edge cost changes, it must replan from scratch with this new information. Using an inflation factor of 1.0 it produces an optimal solution after expanding 9 cells (in fact this solution would have been produced regardless of the inflation factor used). AD*, on the other hand, is able to repair its previous solution given the new information and lower its inflation factor at the same time. Thus, the only cells that are expanded are the 5 whose cost is directly affected by the new information and that reside between the agent and the goal.

Overall, the total number of cells expanded by AD* is 20. This is 4 less than the 24 required by ARA* to produce an optimal solution, and substantially less than the 27 required by D* Lite. Because AD* reuses previous solutions in the same way as ARA* and repairs invalidated solutions in the same way as D* Lite, it is able to provide anytime solutions in dynamic environments very efficiently. The experimental evaluation on a simulated kinematic robot arm performed in (Likhachev *et al.* 2005) supports these claims and shows AD* to be many times more efficient than ARA*, to be able to operate under limited time constraints (an ability that D* Lite lacks), and finally to consistently produce significantly better solutions than D* Lite with inflated heuristics.

Applicability: Anytime Replanning Algorithms

AD* has been shown to be useful for planning in dynamic, complex state spaces, such as 3 DOF robotic arms operating in dynamic environments (Likhachev *et al.* 2005). It has also been used for path-planning for outdoor mobile robots. In particular, those operating in dynamic or partially-known outdoor environments, where velocity considerations are important for generating smooth, timely trajectories. As discussed earlier, this can be framed as a path planning problem over a 4D state space, and an initial suboptimal solution can be generated using AD* in exactly the same manner as ARA*.

Once the robot starts moving along this path, it is likely that it will discover inaccuracies in its map of the environment. As a result, the robot needs to be able to quickly repair previous, suboptimal solutions when new information is gathered, then improve these solutions as much as possible given its processing constraints.

AD* has been used to provide this capability for two robotic platforms currently used for outdoor navigation: an ATRV and a Segway Robotic Mobility Platform (Segway RMP) (see Figure 9) (Likhachev *et al.* 2005). To direct the 4D search in each case, a fast 2D (x, y) planner was used to provide the heuristic values.

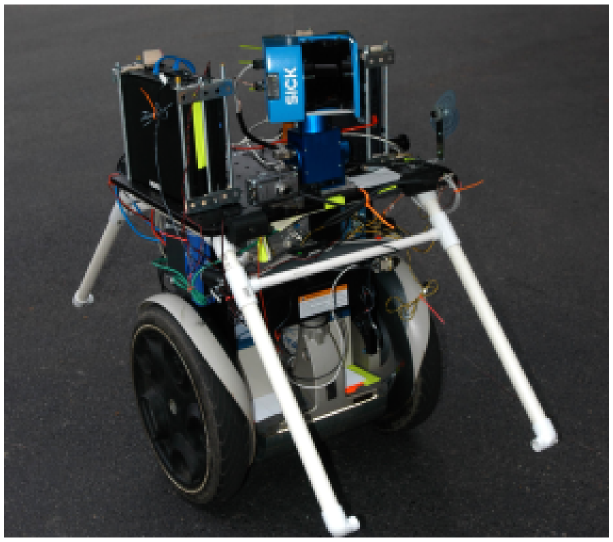

Figure 9: The Segway Robotic Mobility Platform.

Unfortunately, AD* suffers the drawbacks of both anytime algorithms *and* replanning algorithms. As with replanning algorithms, it is possible for AD* to be more computationally expensive than planning from scratch. In fact, this is even more so for AD*, since the version presented here and in (Likhachev *et al.* 2005) reorders the *OPEN* list every time ϵ is changed. It is thus important to have extra checks in place for AD* to prevent trying to repair the previous solution when it looks like it will be more time consuming than starting over (see lines 12 - 14 in Figure 7). For the outdoor navigation platforms mentioned above, this check is based on how much the 2D heuristic cost from the current state to the goal has changed based on changes to the map: if this change is large, there is a good chance replanning will be time consuming. In general it is worth taking into account how much of the search tree has become inconsistent, as well as how long it has been since we last replanned from scratch. If a large portion of the search tree has been affected and the last complete replanning episode was quite some time ago, it is probably worth scrapping the search tree and starting fresh. This is particularly true in very high-dimensional spaces where the dimensionality is derived from the complexity of the agent rather than the environment, since changes in the environment can affect a huge number of states.

There are also a couple optimizations that can be made to AD*. Firstly, it is possible to limit the expense of reordering the *OPEN* list each time ϵ changes by reducing the size of the queue. Specifically, *OPEN* can be split into a priority queue containing states with low key values and one or more unordered lists containing the states with very large key values. The states from the unordered lists need only be considered if the element at the top of the priority queue has a larger key value than the state with minimum key value in these lists. We thus need only maintain the minimum key value (or some lower bound for this value) for all states in the unordered lists. Another more sophisticated and potentially more effective idea that avoids the re-order operation altogether is based on adding a bias to newly inconsistent states (Stentz 1995) and is discussed in (Likhachev *et al.* 2005).

Conclusions

In this paper, we have discussed a family of heuristic algorithms for path planning in real world scenarios. We have attempted to highlight the fundamental similarities between each of the algorithms, along with their individual strengths, weaknesses, and applicable problem domains. A common underlying theme throughout this discussion has been the variable value of previous solutions. When the problem being solved does not change significantly between invocations of our planner, it can be highly advantageous to take advantage of previous solutions as much as possible in constructing a new one. When the problem being solved does change, previous solutions are less useful, and can even be detrimental to the task of arriving at a new solution.

Acknowledgments

The authors would like to thank Sven Koenig for fruitful discussions. This work was sponsored by DARPA's MARS program and the U.S. Army Research Laboratory, under contract "Robotics Collaborative Technology Alliance". The views contained in this document are those of the authors and do not represent the official policies or endorsements of the U.S. Government. Dave Ferguson is partially supported by a National Science Foundation Graduate Research Fellowship.

References

Barbehenn, M., and Hutchinson, S. 1995. Efficient search and hierarchical motion planning by dynamically maintaining single-source shortest path trees. *IEEE Transactions on Robotics and Automation* 11(2):198–214.

Bellman, R. 1957. *Dynamic Programming*. Princeton University Press.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1-2):5–33.

Chakrabarti, P.; Ghosh, S.; and DeSarkar, S. 1988. Admissibility of AO* when heuristics overestimate. *Artificial Intelligence* 34:97–113.

Dean, T., and Boddy, M. 1988. An analysis of time-dependent planning. In *Proceedings of the National Conference on Artificial Intelligence (AAAI).*

Dijkstra, E. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik* 1:269–271.

Edelkamp, S. 2001. Planning with pattern databases. In *Proceedings of the European Conference on Planning*.

Ersson, T., and Hu, X. 2001. Path planning and navigation of mobile robots in unknown environments. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS).*

Ferguson, D., and Stentz, A. 2005. The Delayed D* Algorithm for Efficient Path Replanning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*.

Gelperin, D. 1977. On the optimality of A*. *Artificial Intelligence* 8(1):69–76.

Hart, P.; Nilsson, N.; and Rafael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE trans. Sys. Sci. and Cyb.* 4:100–107.

Hebert, M.; McLachlan, R.; and Chang, P. 1999. Experiments with driving modes for urban robots. In *Proceedings of SPIE Mobile Robots*.

Horvitz, E. 1987. Problem-solving design: Reasoning about computational value, trade-offs, and resources. In *Proceedings of the Second Annual NASA Research Forum*.

Huiming, Y.; Chia-Jung, C.; Tong, S.; and Qiang, B. 2001. Hybrid evolutionary motion planning using follow boundary repair for mobile robots. *Journal of Systems Architecture* 47:635–647.

Kavraki, L.; Svestka, P.; Latombe, J.; and Overmars, M. 1996. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation* 12(4):566–580.

Koenig, S., and Likhachev, M. 2002. Improved fast replanning for robot navigation in unknown terrain. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*.

Koenig, S.; Furcy, D.; and Bauer, C. 2002. Heuristic search-based replanning. In *Proceedings of the International Conference on Artificial Intelligence Planning and Scheduling*, 294–301.

Korf, R. 1993. Linear-space best-first search. *Artificial Intelligence* 62:41–78.

LaValle, S., and Kuffner, J. 1999. Randomized kinodynamic planning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*.

LaValle, S., and Kuffner, J. 2001. Rapidly-exploring Random Trees: Progress and prospects. *Algorithmic and Computational Robotics: New Directions* 293–308.

LaValle, S. 1998. Rapidly-exploring Random Trees: A new tool for path planning. Technical report, Computer Science Dept., Iowa state University.

LaValle, S. 2005. *Planning Algorithms*. In progress - see http://msl.cs.uiuc.edu/planning/.

Likhachev, M., and Koenig, S. 2005. A Generalized Framework for Lifelong Planning A*. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*.

Likhachev, M.; Ferguson, D.; Gordon, G.; Stentz, A.; and Thrun, S. 2005. Anytime Dynamic A*: An Anytime, Replanning Algorithm. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*.

Likhachev, M.; Gordon, G.; and Thrun, S. 2003. ARA*: Anytime A* with provable bounds on sub-optimality. In *Advances in Neural Information Processing Systems*. MIT Press.

Likhachev, M. 2003. Search techniques for planning in large dynamic deterministic and stochastic environments. Thesis proposal. School of Computer Science, Carnegie Mellon University.

Liu, Y.; Koenig, S.; and Furcy, D. 2002. Speeding up the calculation of heuristics for heuristic search-based planning. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 484–491.

Matthies, L.; Xiong, Y.; Hogg, R.; Zhu, D.; Rankin, A.; Kennedy, B.; Hebert, M.; Maclachlan, R.; Won, C.; Frost, T.; Sukhatme, G.; McHenry, M.; and Goldberg, S. 2000. A portable, autonomous, urban reconnaissance robot. In *Proceedings of the International Conference on Intelligent Autonomous Systems (IAS)*.

Nilsson, N. 1980. *Principles of Artificial Intelligence*. Tioga Publishing Company.

Podsedkowski, L.; Nowakowski, J.; Idzikowski, M.; and Vizvary, I. 2001. A new solution for path planning in partially known or unknown environments for nonholonomic mobile robots. *Robotics and Autonomous Systems* 34:145–152.

Rabin, S. 2000. A* speed optimizations. In DeLoura, M., ed., *Game Programming Gems*, 272–287. Rockland, MA: Charles River Media.

Ramalingam, G., and Reps, T. 1996. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms* 21:267–305.

Stentz, A., and Hebert, M. 1995. A complete navigation system for goal acquisition in unknown environments. *Autonomous Robots* 2(2):127–145.

Stentz, A. 1994. Optimal and efficient path planning for partially-known environments. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*.

Stentz, A. 1995. The Focussed D* Algorithm for Real-Time Replanning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.

Thayer, S.; Digney, B.; Diaz, M.; Stentz, A.; Nabbe, B.; and Hebert, M. 2000. Distributed robotic mapping of extreme environments. In *Proceedings of SPIE Mobile Robots*.

Zhou, R., and Hansen, E. 2002. Multiple sequence alignment using A*. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*. Student Abstract.

Zilberstein, S., and Russell, S. 1995. Approximate reasoning using anytime algorithms. In *Imprecise and Approximate Computation*. Kluwer Academic Publishers.

Zlot, R.; Stentz, A.; Dias, M.; and Thayer, S. 2002. Multi-robot exploration controlled by a market economy. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*.