Anytime Dynamic A*: An Anytime, Replanning Algorithm

Maxim Likhachev[†], **Dave Ferguson**[†], **Geoff Gordon**[†], **Anthony Stentz**[†], **and Sebastian Thrun**[‡]
[†] School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA
[‡] Computer Science Department
Stanford University
Stanford, CA, USA

Abstract

We present a graph-based planning and replanning algorithm able to produce bounded suboptimal solutions in an anytime fashion. Our algorithm tunes the quality of its solution based on available search time, at every step reusing previous search efforts. When updated information regarding the underlying graph is received, the algorithm incrementally repairs its previous solution. The result is an approach that combines the benefits of anytime and incremental planners to provide efficient solutions to complex, dynamic search problems. We present theoretical analysis of the algorithm, experimental results on a simulated robot kinematic arm, and two current applications in dynamic path planning for outdoor mobile robots.

Introduction

Planning for systems operating in the real world involves dealing with a number of challenges not faced in many simpler domains. Firstly, the real world is an inherently uncertain and dynamic place; accurate models for planning are difficult to obtain and quickly become out of date. Secondly, when operating in the real world, time for deliberation is usually very limited; agents need to make decisions and act upon these decisions quickly.

Fortunately, a number of researchers have worked on these challenges. To cope with imperfect information and dynamic environments, efficient replanning algorithms have been developed that correct previous solutions based on updated information (Stentz 1994; 1995; Koenig & Likhachev 2002b; 2002a; Ramalingam & Reps 1996; Barto, Bradtke, & Singh 1995). These algorithms maintain optimal solutions for a fraction of the computation required to generate such solutions from scratch.

However, when the planning problem is complex, it may not be possible to obtain optimal solutions within the deliberation time available to an agent. Anytime algorithms (Zilberstein & Russell 1995; Dean & Boddy 1988; Zhou & Hansen 2002; Likhachev, Gordon, & Thrun 2003) have shown themselves to be particularly appropriate in such settings, as they usually provide an initial, possibly highly-suboptimal solution very quickly, then concentrate on improving this solution until the time available for planning runs out.

As of now, there has been relatively little interaction between these two areas of research. Replanning algorithms have concentrated on finding a single solution with a fixed suboptimality bound, and anytime algorithms have concentrated on static environments. But the most interesting problems, for us at least, are those that are both dynamic (requiring replanning) *and* complex (requiring anytime approaches). For example, our current work focuses on path planning in dynamic, relatively high-dimensional state spaces, such as trajectory planning with velocity considerations for mobile robots navigating partially-known outdoor environments.

In this paper, we present a heuristic-based, anytime replanning algorithm that bridges the gap between these two areas of research. Our algorithm, *Anytime Dynamic A** (AD*), continually improves its solution while deliberation time allows, and corrects its solution when updated information is received. A simple example of its application to robot navigation in an eight-connected grid is shown in Figure 1.

This paper is organised as follows. We begin by discussing current incremental replanning algorithms, focussing in particular on D* and D* Lite (Stentz 1995; Koenig & Likhachev 2002a). Next, we present existing anytime algorithms, including the recent Anytime Repairing A* algorithm (Likhachev, Gordon, & Thrun 2003). We then introduce our novel algorithm, Anytime Dynamic A*, and provide an example real-world application in dynamic path planning for outdoor mobile robots. We demonstrate the benefits of the approach through experimental results and conclude with discussion and extensions.

Incremental Replanning

As mentioned above, often the information an agent has concerning its environment (e.g. its map) is imperfect or incomplete. As a result, any solution generated using its initial information may turn out to be invalid or suboptimal as it receives updated information through, for example, an onboard or offboard sensor. It is thus important that the agent is able to replan optimal paths when new information arrives.

A number of algorithms exist for performing this replanning (Stentz 1995; Barbehenn & Hutchinson 1995; Ramalingam & Reps 1996; Ersson & Hu 2001; Huim-

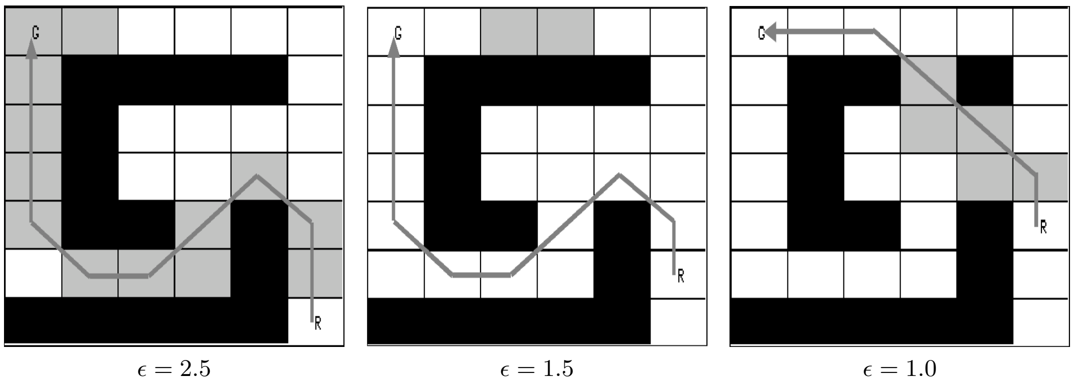


Figure 1: A simple robot navigation example. The robot starts in the bottom right cell and uses Anytime Dynamic A* to quickly plan a suboptimal path to the upper left cell. It then takes one step along this path, improving the bound on its solution, denoted by ϵ, as it moves. After it has moved two steps along its path, it observes a gap in the top wall. Anytime Dynamic A* is able to efficiently improve its current solution while incorporating this new information. The states expanded by Anytime Dynamic A* at each of the first three stages of the traverse are shown shaded.

ing *et al.* 2001; Podsedkowski *et al.* 2001; Koenig & Likhachev 2002a; Ferguson & Stentz 2005). Focussed Dynamic A* (D*) (Stentz 1995) and D* Lite (Koenig & Likhachev 2002a) are currently the most widely used of these algorithms, due to their efficient use of heuristics and incremental updates. D* has been shown to be up to two orders of magnitude more efficient than planning from scratch with A*, and it has been used extensively by fielded robotic systems (Stentz & Hebert 1995; Hebert, McLachlan, & Chang 1999; Matthies *et al.* 2000; Thayer *et al.* 2000; Zlot *et al.* 2002). D* Lite is a simplified version of D* that has been found to be slightly more efficient by some measures (Koenig & Likhachev 2002a). It has been used to guide Segbots and ATRV vehicles in urban terrain (Likhachev 2003). Both algorithms guarantee optimal paths over graphs. D* Lite has also been found useful in other domains. For example, it has been used to construct an efficient heuristic search-based symbolic replanner (Koenig, Furcy, & Bauer 2002).

Both D* and D* Lite maintain least-cost paths between a start state and any number of goal states as the cost of arcs between states change. Both algorithms can handle increasing or decreasing arc costs and dynamic start states. They are both thus suited to solving the goal-directed mobile robot navigation problem, which entails a robot moving from some initial state to one of a number of goal states while updating its map information through an onboard sensor. Because the two algorithms are fundamentally very similar, we restrict our attention here to D* Lite, which has been found to be slightly more efficient for some navigation tasks (Koenig & Likhachev 2002a) and easier to analyze. More details on each of the algorithms can be found in (Stentz 1995) and (Koenig & Likhachev 2002a).

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 the set of states in some finite state space[1]. 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 arc 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 $g(s) < rhs(s)$).

As with 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 an optimal path from state s to s', and needs to satisfy $h(s,s') \leq c^*(s,s')$ 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 priority $key(s')$, denoted $key(s) \dot{<} key(s')$, iff $k_1(s) < k_1(s')$ or both $k_1(s) = k_1(s')$ and $k_2(s) < k_2(s')$. D* Lite expands states from the queue in increasing priority, updating their g-values and the rhs-values of their predecessors, until there is no state in the queue with a key value 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.

If arc costs change after this initial solution has been generated, D* Lite updates the rhs-values of each state immediately affected by the changed arc costs and places those

[1]As mentioned previously, any number of goals can be incorporated. When there is more than one, s_{goal} is the set of goals rather than a single state. The algorithm as presented here remains unchanged.

```
**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 2: **The D* Lite Algorithm (basic version).**

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 key value 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 most efficient 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 is given in Figure 2.

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 arc 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 arc costs increase, it ensures that only those newly-underconsistent states that could potentially invalidate the current cost of the start state are processed.

Anytime Planning

When the planning problem is complex and the time available to an agent for planning is limited, generating optimal solutions can be infeasible. In such situations, the agent must be satisfied with the best solution that can be generated within the available computation time. A

```
**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 minimum key 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 3: **The ARA* Algorithm (backwards version).**

useful set of algorithms for generating such solutions are known as anytime algorithms (Zilberstein & Russell 1995; Dean & Boddy 1988; Zhou & Hansen 2002; Likhachev, Gordon, & Thrun 2003). Typically, these start out by computing an initial, potentially highly suboptimal solution, then improve this solution as time allows.

A*-based anytime algorithms 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 (Bonet & Geffner 2001; Korf 1993; Zhou & Hansen 2002; Edelkamp 2001; Rabin 2000; Chakrabarti, Ghosh, & DeSarkar 1988) at the cost of solution optimality. A* also has the nice property that if the heuristic used is consistent, and the heuristic values are multiplied by an inflation factor $\epsilon > 1$, then the cost of the generated solution is guaranteed to be within ϵ times the cost of an optimal solution (Pearl 1984). Zhou and Hansen take advantage of this property to present an anytime algorithm that begins by quickly producing such an ϵ bounded solution, then gradually improves this solution over time (Zhou & Hansen 2002). However, their algorithm has no control over the suboptimality bound while the initial solution is improved upon. Likhachev, Gordon, and Thrun present an anytime algorithm that performs a succession of A* searches, each with a decreasing inflation factor, where each search reuses efforts from previous searches (Likhachev, Gordon, & Thrun 2003). This approach provides suboptimality bounds for each suc-

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

**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 $(\min_{s \in OPEN}(\text{key}(s)) \dot{<} \text{key}(s_{start})$ 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 4: **Anytime Dynamic A*: ComputeorImprovePath function.**

cessive search and has been shown to be much more efficient than competing approaches (Likhachev, Gordon, & Thrun 2003).

Their algorithm, Anytime Repairing A*, uses the notion of consistency introduced above to limit 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[2]. Once a state has been expanded during a particular search, if it becomes inconsistent 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 fresh priority queue (with new priorities based on the new inflation factor ϵ), 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.

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

```
**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 5: **Anytime Dynamic A*: Main function.**

A simplified (backwards-searching) version of the algorithm is given in Figure 3. 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. Other notation should be consistent with that described earlier.

Anytime Dynamic A*

As shown in the previous sections, there exist efficient algorithms for coping with dynamic environments (e.g. D* and D* Lite), and complex planning problems (ARA*). However, what about when we are facing both complex planning problems and dynamic environments at the same time?

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 may also need to generate suboptimal solutions, as optimality may not be possible if it is subject to limited deliberation time.

Given the strong similarities between D* Lite and ARA*, it seems appropriate to look at whether the two could be combined into a single anytime, incremental replanning algorithm that could provide the sort of performance required in this example.

Our novel algorithm, *Anytime Dynamic A** (AD*), does just this. It 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 with priorities equal to the minimum of their previous key value and their new key value, as with D* Lite. States on the queue are then processed until the current solution is guaranteed to be ϵ-suboptimal.

The Algorithm

The algorithm is presented in Figures 4 and 5. The Main function first sets the inflation factor ϵ to a sufficiently high value ϵ_0, so that an initial, suboptimal plan can be generated quickly (lines $02 - 06$, Figure 5). Then, unless changes in edge costs are detected, the Main function decreases ϵ and improves the quality of its solution until it is guaranteed to be optimal, that is, $\epsilon = 1$ (lines $14 - 20$, Figure 5). 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 that a less optimal solution can be produced quickly (lines 12 - 13, Figure 5). 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 uninflated heuristic values. This means that different key values must be computed for underconsistent states than for overconsistent states (lines 01 - 04, Figure 4).

By incorporating these considerations, AD* is able to handle both changes in edge costs and changes to the inflation factor ϵ. It can also be slightly modified to handle the situation where the start state s_{start} is changing, as is the case when the computed path is being traversed by an agent. To do this, we replace line 07, Figure 5 with the following lines:

```
07a. fork(MoveAgent());
07b. while ($s_{start} \neq s_{goal}$)
```

Here, MoveAgent is another function executed in parallel that shares the variable s_{start} and steps the agent along the current path, at each step allowing the path to be repaired and improved by the Main function:

```
MoveAgent()
01. while ($s_{start} \neq s_{goal}$)
02.     wait until a plan is available;
03.     $s_{start} = \text{argmin}_{s \in Succ(s_{start})}(c(s_{start}, s) + g(s))$;
04.     move agent to $s_{start}$;
```

Here, $\text{argmin}_{s' \in Succ(s)} f()$ returns the successor of s for which function f is minimized, so in line 03 the start state is updated to be one of its neighbors on the current path to the goal. Since the states on the *OPEN* queue have their key values recalculated each time ϵ is changed, processing will automatically be focussed towards the updated agent state s_{start}. This conveniently allows the agent to improve and update its solution path while it is being traversed.

An Example

Figure 6 presents an illustration of each of the approaches addressed thus far on a simple grid world planning problem (the same used earlier to introduce AD*). 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 (each algorithm has been optimized not to expand the agent cell). The resulting paths are shown as dark grey arrows.

The first approach shown is backwards A*, that is, A* with its search focussed from the goal state to the start state. 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 fewer overall cell expansions. 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 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 in-

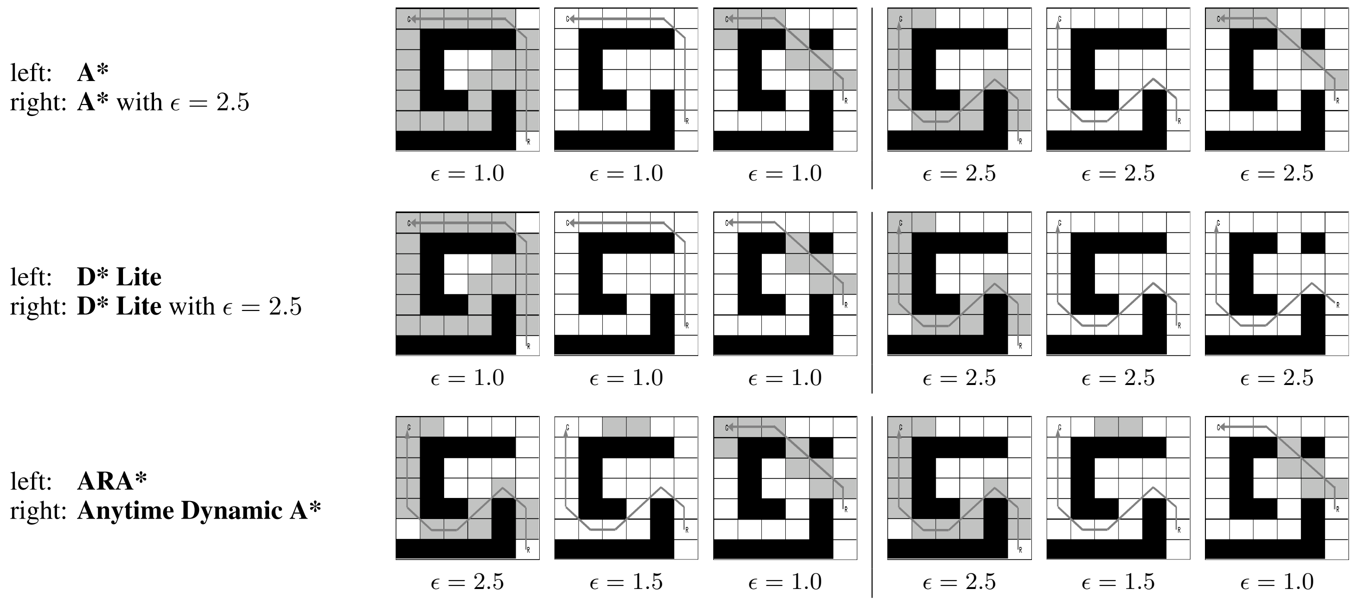

left: **A***
right: **A*** with $\epsilon = 2.5$
$\epsilon = 1.0$ $\epsilon = 1.0$ $\epsilon = 1.0$ $\epsilon = 2.5$ $\epsilon = 2.5$ $\epsilon = 2.5$
left: **D* Lite**
right: **D* Lite** with $\epsilon = 2.5$
$\epsilon = 1.0$ $\epsilon = 1.0$ $\epsilon = 1.0$ $\epsilon = 2.5$ $\epsilon = 2.5$ $\epsilon = 2.5$
left: **ARA***
right: **Anytime Dynamic A***
$\epsilon = 2.5$ $\epsilon = 1.5$ $\epsilon = 1.0$ $\epsilon = 2.5$ $\epsilon = 1.5$ $\epsilon = 1.0$

Figure 6: 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.

flation 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 much 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.

Theoretical Properties

We can prove a number of properties of AD*, including its termination and ϵ-suboptimality. The proofs of the properties below, along with others, can be found in (Likhachev *et al.* 2005). In what follows, we use $g^*(s)$ to denote the cost of an optimal path from s to s_{goal}. Let us also define a *greedy path* from s_{start} to s as a path that is computed as follows: starting from s_{start}, always move from the current state s^c to a successor state s' that satisfies $s' = \text{argmin}_{s' \in Succ(s^c)}(c(s^c, s') + g(s'))$ until $s^c = s$.

Theorem 1 *Whenever the ComputeorImprovePath function exits, for any consistent state* s *with* $key(s) \dot{\leq} \min_{s' \in OPEN}(key(s'))$, *we have* $g^*(s) \leq g(s) \leq \epsilon * g^*(s)$, *and the cost of a greedy path from* s *to* s_{goal} *is no larger than* $g(s)$.

This theorem guarantees the ϵ-suboptimality of the solution returned by the ComputeorImprovePath function, since, when it terminates, s_{start} is consistent and the key value of s_{start} is at least as large as the minimum key value of all states on the *OPEN* queue.

Theorem 2 *Within each call to ComputeorImprovePath() a state is expanded at most twice and only if it was inconsistent before the call to ComputeorImprovePath() or its g-value was altered during the current execution of ComputeorImprovePath().*

This second theorem proves that the ComputeorImprovePath function will always terminate. Since the value of ϵ is decreased after each call to ComputeorImprovePath() unless edge costs have changed, it then follows from Theorem 1 that in the absence of infinitely-changing edge costs, AD* will always eventually publish an optimal solution path.

Theorem 2 also highlights the computational advantage of AD* over D* Lite and ARA*. Because AD* only processes exactly the states that were either inconsistent at the beginning of the current search or made inconsistent during the current search, it is able to produce solutions very efficiently. Neither D* Lite nor ARA* are able to both improve and repair existing solutions in this manner.

Robotic Application

The main motivation for this work was efficient path planning for land-based mobile robots. In particular, those operating in dynamic outdoor environments, where velocity considerations are important for generating smooth, timely trajectories. We can frame this problem as a search over a state space involving four dimensions: the $(x.y)$ position of the robot, the robot's orientation, and the robot's velocity.

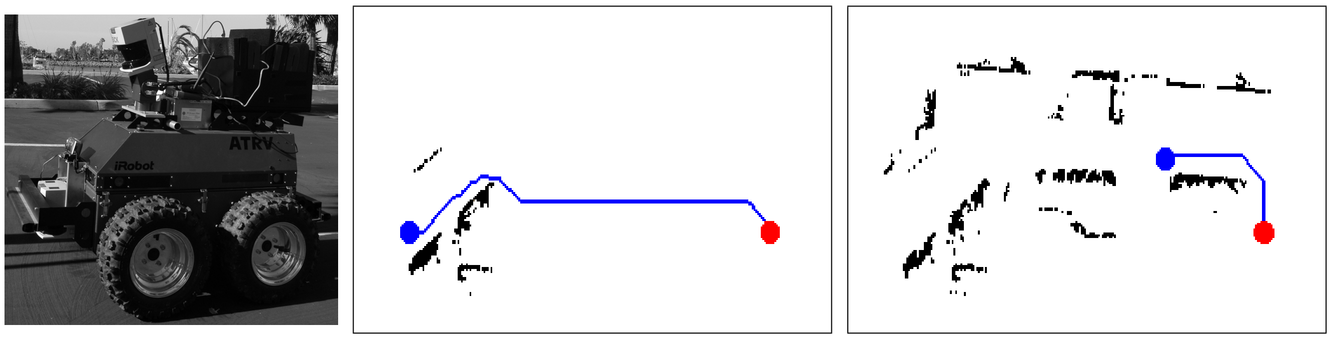


Figure 7: The ATRV robotic platform. Also shown are two images of the robot moving from the left side to the right side of an initially-unknown outdoor environment using AD* for updating and improving its solution path.

Solving this initial 4D search in large environments can be computationally costly, and an optimal solution may be infeasible if the initial processing time of the robot is limited.

Once the robot starts moving, it is highly unlikely that it will be able to replan an optimal path if it discovers changes in the environment. But if the environment is only partially-known or is dynamic, either of which is common in the urban areas we are interested in traversing, changes will certainly be discovered. As a result, the robot needs to be able to quickly generate suboptimal solutions when new information is gathered, then improve these solutions as much as possible given its processing constraints.

We have used AD* to provide this capability for two robotic platforms currently used for outdoor navigation. To direct the 4D search in each case, we use a fast 2D (x, y) planner to provide the heuristic values. Figure 7 shows our first system, an ATRV vehicle equipped with two laser range finders for mapping and an inertial measurement unit for position estimation. Also shown in Figure 7 are two images taken of the map and path generated by the robot as it traversed from one side of an initially-unknown environment to the other. The 4D state space for this problem has roughly 20 million states, however AD* was able to provide fast, safe trajectories in real-time.

We have also implemented AD* on a Segway Robotic Mobility Platform, shown in Figure 8. Using AD*, it has successfully navigated back and forth across a substantial part of the Stanford campus.

Experimental Results

To evaluate the performance of AD*, we compared it to ARA* and D* Lite (with an inflation factor of $\epsilon = 20$) on a simulated 3 degree of freedom (DOF) robotic arm manipulating an end-effector through a dynamic environment (see Figures 9 and 10). In this set of experiments, the base of the arm is fixed, and the task is to move into a particular goal configuration while navigating the end-effector around fixed and dynamic obstacles. We used a manufacturing-like scenario for testing, where the links of the arm exist in an obstacle-free plane, but the end-effector projects down into a cluttered space (such as a conveyor belt moving goods down a production line).

In each experiment, we started with a known map of the end-effector environment. As the arm traversed each step of its trajectory, however, there was some probability $\mathcal{P}^o$ that an obstacle would appear in its path, forcing the planner to repair its previous solution.

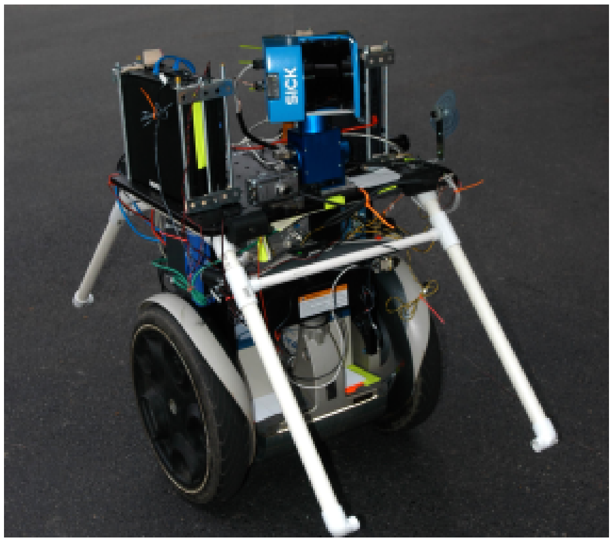


Figure 8: The Segbot robotic platform.

We have included results from two different initial environments and several different values of $\mathcal{P}^o$, ranging from $\mathcal{P}^o = 0.04$ to $\mathcal{P}^o = 0.2$. In these experiments, the agent was given a fixed amount of time for deliberation, $\mathcal{T}^d = 1.0$ seconds, at each step along its path. The cost of moving each link was nonuniform: the link closest to the end-effector had a movement cost of 1, the middle link had a cost of 4, and the lower link had a cost of 9. The heuristic used by all algorithms was the maximum of two quantities; the first was the cost of a 2D path from the current end-effector position to its position at the state in question, accounting for all the currently known obstacles on the way; the second was the maximum angular difference between the joint angles at the current configuration and the joint angles at the state in question. This heuristic is admissible and consistent.

In each experiment, we compared the cost of the path traversed by ARA* with $\epsilon_0 = 20$ and D* Lite with $\epsilon = 20$ to

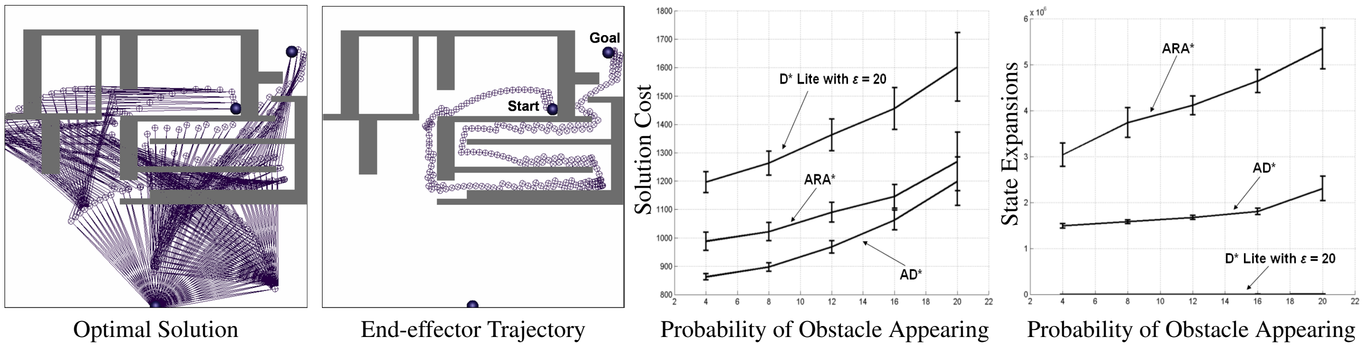


Figure 9: Environment used in our first experiment, along with the optimal solution and the end-effector trajectory (without any dynamic obstacles). Also shown are the solution cost of the path traversed and the number of states expanded by each of the three algorithms compared.

that of AD* with $\epsilon_0 = 20$, as well as the number of states expanded by each approach. Our first environment had only one general route that the end-effector could take to get to its goal configuration, so the difference in path cost between the algorithms was due to manipulating the end-effector along this general path more or less efficiently. Our second experiment presented two qualitatively different routes the end-effector could take to the goal. One of these had a shorter distance in terms of end-effector grid cells but was narrower, while the other was longer but broader, allowing for the links to move in a much cheaper way to get to the goal.

Each environment consisted of a 50×50 grid, and the state space for each consisted of slightly more than 2 million states. The results for the experiments, along with 95% confidence intervals, can be found in Figures 9 and 10. As can be seen from these graphs, AD* was able to generate significantly better trajectories than ARA* while processing far fewer states. D* Lite processed very few states, but its overall solution quality was much worse than either of the anytime approaches. This is because it is unable to improve its suboptimality bound.

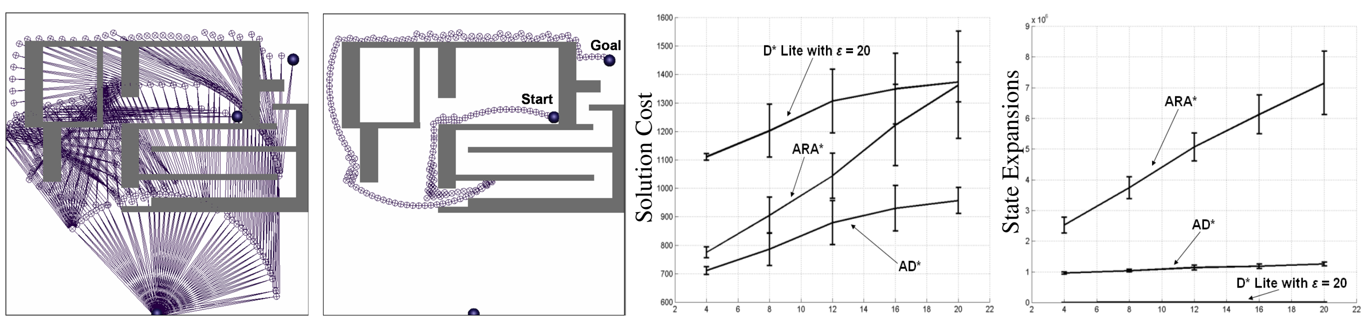
We have also included results focussing exclusively on the anytime behavior of AD*. To generate these results, we repeated the above experiments without any randomly-appearing obstacles (i.e., $\mathcal{P}^o = 0$). We kept the deliberation time available at each step, $\mathcal{T}^d$, set at the same value as in the original experiments (1.0 seconds). Figure 11 shows the total path cost (the cost of the executed trajectory so far plus the cost of the remaining path under the current plan) as a function of how many steps the agent has taken along its path. Since the agent plans before each step, the number of steps taken corresponds to the number of planning episodes performed. These graphs show how the quality of the solution improves over time. We have included only the first 20 steps, as in both cases AD* has converged to the optimal solution by this point.

We also ran the original experiments using D* Lite with no inflation factor and unlimited deliberation time, to get an indication of the cost of an optimal path. On average, the path traversed by AD* was about 10% more costly than the optimal path, and it expanded roughly the same number of states as D* Lite with no inflation factor. This is particularly encouraging: not only is the solution generated by AD* very close to optimal, but it is providing this solution in an anytime fashion for roughly the same total processing as would be required to generate the solution in one shot.

Discussion and Future Work

There are a few details and extensions of the algorithm worth expanding on. Firstly, lines 12 - 13 of the Main function (Figure 5) state that if significant changes in edge costs are observed, then either ϵ should be increased or we should replan from scratch. This is an important consideration, as it is possible that repairing a previous solution will involve significantly more processing than planning over from scratch. Exactly what constitutes “significant changes” is application-dependent. For our outdoor navigation platforms, we look to see how much the 2D heuristic cost from the current state to the goal has changed: if this change is large, there is a good chance replanning will be time consuming. In our simulated robotic arm experiments, we never replanned from scratch, since we were always able to replan incrementally in the allowed deliberation time. However, 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.

Secondly, every time we change ϵ the entire *OPEN* queue needs to be reordered to take into account the new key values of all the states on it (line 17 Figure 5). This can be a rather expensive operation. It is possible to avoid this full queue reorder by extending an idea originally presented along with the Focussed D* algorithm (Stentz 1995), where a *bias* term is added to the key value of each state being placed on the queue. This bias is used to ensure that those states whose priorities in the queue are based on old, incorrect key values are at least as high as they should be in the queue. In



Optimal Solution End-effector Trajectory Probability of Obstacle Appearing Probability of Obstacle Appearing

Figure 10: Environment used in our second experiment, along with the optimal solution and the end-effector trajectory (without any dynamic obstacles). Also shown are the solution cost of the path traversed and the number of states expanded by each of the three algorithms compared.

other words, by adding a fixed value to the key of each new state placed on the queue, the old states are given a relative advantage in their queue placement. When a state is popped off the queue whose key value is not in line with the current bias term, it is placed back on the queue with an updated key value. The intuition is that only a small number of the states previously on the queue may ever make it to the top, so it can be much more efficient to only reorder the ones that do. We can use the same idea when ϵ decreases (from ϵ^o to ϵ^n, say) to increase the bias term by $(\epsilon^o - \epsilon^n) \cdot \max_{s \in OPEN} h(s_{start}, s)$. The key value of each state becomes

$$key(s) = [\min(g(s), rhs(s)) + \epsilon \cdot h(s_{start}, s) + \mathbf{bias}, \\ \min(g(s), rhs(s))].$$

By using the maximum heuristic value present in the queue to update the bias term, we are guaranteeing that each state already on the queue will be at least as elevated on the queue as it should be relative to the new states being added. It is future work to implement this approach but it appears to be a promising modification.

Finally, it may be possible to reduce the effect of underconsistent states in our repair of previous solution paths. With the current version of AD*, underconsistent states need to be placed on the queue with a key value that uses an uninflated heuristic value. This is because they could reside on the old solution path and their true effect on the start state may be much more than the inflated heuristic would suggest. This means, however, that the underconsistent states quickly rise to the top of the queue and are processed before many overconsistent states. At times, these underconsistent states may not have any effect on the value of the start state (for instance when they do not reside upon the current solution path). We are currently looking into ways of reducing the number of underconsistent states examined, using ideas very recently developed (Ferguson & Stentz 2005). This could prove very useful in the current framework, where much of the processing is done on underconsistent states that may not turn out to have any bearing on the solution.

Conclusions

We have presented Anytime Dynamic A*, a heuristic-based, anytime replanning algorithm able to efficiently generate solutions to complex, dynamic path planning problems. The algorithm works by continually decreasing a suboptimality bound on its solution, reusing previous search efforts as much as possible. When changes in the environment are encountered, it is able to repair its previous solution incrementally. Our experiments and application of the algorithm to two real-world robotic systems have shown it to be a valuable addition to the family of heuristic-based path planning algorithms, and a useful tool in practise.

Acknowledgments

The authors would like to thank Sven Koenig for fruitful discussions. This work was partially sponsored by DARPA's MARS program. Dave Ferguson is supported in part by an NSF 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.

Barto, A.; Bradtke, S.; and Singh, S. 1995. Learning to Act Using Real-Time Dynamic Programming. *Artificial Intelligence* 72:81–138.

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).*

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*

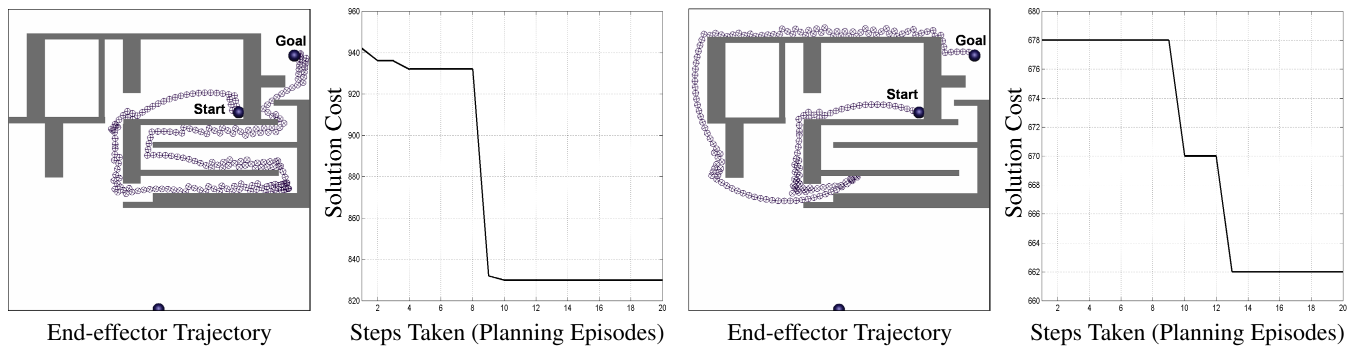


Figure 11: An illustration of the anytime behavior of AD*. Each graph shows the total path cost (the cost of the executed trajectory so far plus the cost of the remaining path under the current plan) as a function of how many steps the agent has taken along its path, for the static path planning problem depicted to the left of the graph. Also shown are the optimal end-effector trajectories for each problem.

IEEE International Conference on Robotics and Automation (ICRA).

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

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.

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

Koenig, S., and Likhachev, M. 2002b. Incremental A*. In *Advances in Neural Information Processing Systems*. MIT Press.

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.

Likhachev, M.; Ferguson, D.; Gordon, G.; Stentz, A.; and Thrun, S. 2005. Anytime Dynamic A*: The Proofs. Technical Report CMU-RI-TR-05-12, Carnegie Mellon School of Computer Science.

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.

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)*.

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

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)*.