9-1-2008

Anytime search in dynamic graphs

Maxim Likhachev
University of Pennsylvania, maximl@central.cis.upenn.edu

Dave Ferguson
Carnegie Mellon University

Geoff Gordon
Carnegie Mellon University

Anthony Stentz
Carnegie Mellon University

Sebastian Thrun
Stanford University

Copyright 2008 Elsevier. Reprinted from:
Maxim Likhachev, Dave Ferguson, Geoff Gordon, Anthony Stentz, Sebastian Thrun, Anytime search in dynamic graphs, Artificial Intelligence, Volume 172, Issue 14, September 2008, Pages 1613-1643, ISSN 0004-3702.
DOI: 10.1016/j.artint.2007.11.009.
URL: http://www.sciencedirect.com/science/article/B6TYF-4SFS0P8-1/2/281d52a575110d97111291fbaa9210d11

Anytime Search in Dynamic Graphs

Maxim Likhachev [a], Dave Ferguson [b,c], Geoff Gordon [c], Anthony Stentz [c], and Sebastian Thrun [d]

[a]*Computer and Information Science, University of Pennsylvania, Philadelphia, PA, USA*

[b]*Intel Research Pittsburgh, Pittsburgh, PA, USA*

[c]*School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA*

[d]*Computer Science Department, Stanford University, Stanford, CA, USA*

Abstract

Agents operating in the real world often have limited time available for planning their next actions. Producing optimal plans is infeasible in these scenarios. Instead, agents must be satisfied with the best plans they can generate within the time available. One class of planners well-suited to this task are anytime planners, which quickly find an initial, highly suboptimal plan, and then improve this plan until time runs out.

A second challenge associated with planning in the real world is that models are usually imperfect and environments are often dynamic. Thus, agents need to update their models and consequently plans over time. Incremental planners, which make use of the results of previous planning efforts to generate a new plan, can substantially speed up each planning episode in such cases.

In this paper, we present an A*-based anytime search algorithm that produces significantly better solutions than current approaches, while also providing suboptimality bounds on the quality of the solution at any point in time. We also present an extension of this algorithm that is both anytime and incremental. This extension improves its current solution while deliberation time allows and is able to incrementally repair its solution when changes to the world model occur. We provide a number of theoretical and experimental results and demonstrate the effectiveness of the approaches in a robot navigation domain involving two physical systems. We believe that the simplicity, theoretical properties, and generality of the presented methods make them well suited to a range of search problems involving large, dynamic graphs.

Keywords: planning, replanning, anytime planning, A*, search, anytime search, heuristic search, incremental search.

1 Introduction

In this paper we present search algorithms for planning paths through large, dynamic graphs. Such graphs can be used to model a wide range of problem domains in AI and robotics. A number of graph-based search algorithms have been developed for generating paths through graphs. A* search [1] and Dijkstra's algorithm [2] are two commonly used and extensively studied approaches that generate optimal paths. These algorithms are very efficient. In fact, they process the minimum number of states possible while guaranteeing an optimal solution when no other information besides the graph and heuristics (in the case of A*) is provided [3]. Realistic planning problems, however, are often too large to solve optimally within an acceptable time. Moreover, even if an optimal plan is found initially, the model used to represent the problem is unlikely to be perfect and changes may occur in the environment, and therefore an agent may find discrepancies in its model while executing its plan. In such situations, the agent needs to update its model and re-plan. Finding an optimal plan every time it needs to re-plan would make the agent stop execution too often and for too long. Anytime planning [4, 5] presents an appealing alternative. Anytime planning algorithms try to find the best plan they can within the amount of time available to them. They quickly find an approximate, and possibly highly suboptimal, plan and then improve this plan while time is available. In addition to being able to meet time deadlines, many of these algorithms also make it possible to interleave planning and execution: while the agent executes its current plan, the planner works on improving the plan.

In the first part of the paper we present an anytime version of A* search called Anytime Repairing A* (ARA*). This algorithm has control over a suboptimality bound for its current solution, which it uses to achieve the anytime property: it starts by finding a suboptimal solution quickly using a loose bound, then tightens the bound progressively as time allows. Given enough time it finds a provably optimal solution. While improving its bound, ARA* reuses previous search efforts and, as a result, is very efficient. We demonstrate this claim empirically on a motion planning application involving a simulated robotic arm with several degrees of freedom.

While anytime planning algorithms are very useful when good models of the environment are known a priori, they are less beneficial when prior models are not very accurate or when the environment is dynamic. In these situations, the agent may need to update its world model frequently. Each time its world model is updated, all of the previous efforts of the anytime planners are invalidated and they need to start generating a new plan from scratch. This is especially troublesome when one tries to interleave planning with execution: all the efforts spent on improving a plan during execution become wasted after a single update to the model, even though the update may be minor. For example, in mobile robot navigation a robot may start out knowing the map only partially, plan assuming that all unknown areas are safe to traverse, and then begin executing the plan. While executing the plan, it senses the environment around it and as it discovers new obstacles it updates the map and constructs a new plan (e.g., [6,7]). As a result, the robot has to plan frequently during its execution. Anytime planners are not able to provide anytime capability in such scenarios, as they are constantly having to generate new (highly-suboptimal) plans from scratch.

A class of algorithms known as replanning, or incremental, algorithms are effective in such cases as they use the results of previous planning efforts to help find a new plan when the problem has changed slightly. Two such algorithms, Dynamic A* (D*) and Lifelong Planning A* (LPA*), have been particularly useful for heuristic search-based replanning in artificial intelligence and robotics. These algorithms work by performing an A* search to generate an initial solution. Then, when the world model is updated, they repair their previous solution by reusing as much of their previous search efforts as possible. As a result, they can be orders of magnitude more efficient than replanning from scratch every time the world model changes. However, while these replanning algorithms substantially speed up a series of searches for similar problems, they lack the anytime property of ARA*: once they find a solution they stop and do not improve the solution even if more planning time is available. These algorithms can only be pre-configured either to search for an optimal solution or to search for a solution bounded by a fixed suboptimality factor.

We address this limitation in section 5 by presenting Anytime D* (AD*), a search algorithm that is both anytime and incremental. The algorithm re-uses its old search efforts while simultaneously improving its previous solution (as with ARA*) as well as re-planning if necessary (as with D*/LPA*). Besides merely speeding up planning, this combination allows one to interleave planning and execution more effectively. The planner can continue to improve a solution without having to discard all of its efforts every time the model of the world is updated. To the best of our knowledge, AD* is the first search algorithm that is both anytime and incremental, and just like ARA* and D*/LPA*, AD* also provides bounds on the suboptimality of each solution it returns. In section 5 we experimentally demonstrate the advantages of AD* over search algorithms that are either anytime or incremental (but not both) on the problem of motion planning for a simulated robot arm. In section 6 we demonstrate how AD* enables us to plan smooth paths for mobile robots navigating through partially-known environments.

The development of the ARA* and AD* algorithms is due to a simple alternative view of A* search that we introduce in section 4.2 and an extension of this view presented in section 5.2. We hope that this interpretation of A* will inspire research on other search algorithms, while the simplicity, generality and practical utility of the presented algorithms will contribute to the research and development of planners well-suited for autonomous agents operating in the real world.

2 Background

In this paper we concentrate on planning problems represented as a search for a path in a known finite graph. We use S to denote the finite set of states in the graph. $succ(s)$ denotes the set of successor states of state $s \in S$, and $pred(s)$ denotes the set of predecessor states of s. For any pair of states $s, s' \in S$ such that $s' \in succ(s)$ we require the cost of transitioning from s to s' to be positive: $0 < c(s, s') \leq \infty$. (In case of an infinite graph, the cost would also have to be bounded from below by a (small) positive constant.)

Given such a graph and two states s_{start} and s_{goal}, the task of a search algorithm is to find a path from s_{start} to s_{goal}, denoted by $\pi(s_{\text{start}})$, as a sequence of states $\{s_0, s_1, \ldots, s_k\}$ such that $s_0 = s_{\text{start}}$, $s_k = s_{\text{goal}}$ and for every $1 \leq i \leq k$, $s_i \in succ(s_{i-1})$. This path defines a sequence of valid transitions between states in the graph, and if the graph accurately models the original problem, an agent can execute the actions corresponding to these transitions to solve the problem. The cost of the path is the sum of the costs of the corresponding transitions $\sum_{i=1}^{k} c(s_{i-1}, s_i)$. For any pair of states $s, s' \in S$ we let $c^*(s, s')$ denote the cost of a least-cost path from s to s'. For $s = s'$ we define $c^*(s, s') = 0$.

The goal of shortest path search algorithms such as A* search is to find a path from s_{start} to s_{goal} whose cost is minimal, i.e. equal to $c^*(s_{\text{start}}, s_{\text{goal}})$. Suppose for every state $s \in S$ we knew the cost of a least-cost path from s_{start} to s, that is, $c^*(s_{\text{start}}, s)$. We use $g^*(s)$ to denote this cost. Then a least-cost path from s_{start} to s_{goal} can be re-constructed in a backward fashion as follows: start at s_{goal}, and at any state s_i pick a state $s_{i-1} = \arg\min_{s' \in pred(s_i)}(g^*(s') + c(s', s_i))$ until $s_{i-1} = s_{\text{start}}$ (ties can be broken arbitrarily). We will call this a *greedy* path based on g^*-values.

Consequently, algorithms like A* search try to compute g^*-values. In particular, A* maintains g-values for each state it has visited so far, where $g(s)$ is always the cost of the best path found so far from s_{start} to s. If no path to s has been found yet then $g(s)$ is assumed to be ∞ (this includes the states that have not yet been visited by the search). A* starts by setting $g(s_{\text{start}})$ to 0 and processing (expanding) this state first. The expansion of state s involves checking if a path to any successor state s' of s can be improved by going through state s, and if so then setting the g-value of s' to the cost of the new path found and making it a candidate for future expansion. This way, s' will also be selected for expansion at some point and the cost of the new path will be propagated to its children. Thus, the g-values are always the costs of paths found and therefore are always upper bounds on the corresponding g^*-values. Moreover, if the g-values of states on one of the least-cost paths from s_{start} to s_{goal} are exactly *equal* to the corresponding g^*-values, then a path from s_{start} to s_{goal} reconstructed in the greedy fashion described earlier, but based on g-values, is guaranteed to be a least-cost path. From now on, unless specified otherwise the term "greedy path" will refer to a greedy path constructed based on g-values of states.

(Note that the fact that the path is constructed in a greedy fashion by no means implies that the algorithm used to compute g-values was a greedy algorithm.)

The challenge for shortest path search algorithms is to minimize the amount of processing required to guarantee that the g-values of states on one or more of the least-cost paths from s_{start} to s_{goal} are exactly equal to the corresponding g^*-values. A* expands all states (up to tie-breaking) whose g- plus h-values (i.e., $g(s) + h(s)$) are less than or equal to $g(s_{\text{start}})$, where h-values estimate the cost of a least-cost path from s to s_{goal}. This focusses the search on the states through which the *whole* path from s_{start} to s_{goal} looks promising. It can be much more efficient than expanding all states whose g-values are smaller than or equal to $g(s_{\text{start}})$, which is required by Dijkstra's algorithm to guarantee that the solution it finds is optimal.

3 Related Work

Anytime planning algorithms find an initial, possibly highly suboptimal solution very quickly and then continually work on improving the solution until planning time is exhausted. The idea of anytime planning was proposed in the AI community some time ago [5], and much work has since been done on the development of anytime planning algorithms (for instance, [8–12]).

However, much less work has been done on anytime graph-based searches. A simple and quite common way of transforming an arbitrary search algorithm into an anytime search algorithm is to iteratively increase the region of the state space searched through. To begin with, a small region of the state space surrounding the current state of an agent is searched for a solution that looks most promising based on goal distance estimates for the states on the fringe of the region and the costs of reaching these states (such searches are commonly referred to as real-time [13] or agent-centered searches [14, 15]). After this initial solution is returned, the region can then be iteratively increased until either the time available for planning expires or the region has grown to the whole state space. Such searches can usually exhibit good anytime behavior in any domain. In particular, such searches are advantageous in domains where producing *any* complete path is hard within the provided time window and executing a partial path is an acceptable choice. Unfortunately, such algorithms typically provide no bounds on the quality of their solutions and may even return plans that lead to failures in domains that have states with irreversible conditions (e.g. a one-way road that leads to a dead-end).

The most closely related approach to the work we present in this paper is a complete anytime heuristic search algorithm called Anytime A* [16, 17]. Anytime A* relies on the fact that in many domains inflating the heuristic by some constant $\epsilon > 1$ can drastically reduce the number of states A* has to examine before it can produce a solution [16, 18–24]. An additional nice property of inflating heuristics is that the cost of the solution found for an inflation factor ϵ is no larger than ϵ times the cost of an optimal solution [25]. When obtaining an initial solution, Anytime A* inflates heuristics by a large ϵ. After the first solution is found, the algorithm continues to process the states whose f-values ($g(s) + h(s)$, where $h(s)$ is un-inflated) are less than or equal to the cost of the best solution found so far. Similarly to ARA*, the algorithm we present in section 4, Anytime A* provides a bound on the suboptimality of the solutions it returns. Providing suboptimality bounds is valuable as it allows one to judge the quality of the current plan, decide whether to continue or preempt search based on the current suboptimality bound, and evaluate the quality of past planning episodes to allocate time for future planning episodes accordingly. Unlike ARA*, however, Anytime A* does not have any control over its suboptimality bound, except for the selection of the inflation factor of the first search. Such control helps in adjusting the tradeoff between computation and plan quality. ARA*, in addition to providing suboptimality bounds for all its solutions, allows one to control these bounds. In the domains we experimented with, this control allows ARA* to decrease the bounds much more gradually. [1] Another advantage of ARA* is that it guarantees to examine each state at most once during its first search, unlike Anytime A*. This property is important because it provides a theoretical bound on the amount of work before ARA* produces its first plan. Nevertheless, as described in section 4, Anytime A* incorporates a number of interesting ideas that are also applicable to ARA*.

A few other anytime heuristic searches that return complete solutions have been developed [26–29]. Unlike ARA*, however, they all share the property of not being able to provide any non-trivial suboptimality bounds on their solutions. Some of these algorithms are variants of depth-first search (e.g. Depth-first Branch-and-Bound search [29] and Complete Anytime Beam search [28]) and may use much less memory than A* and its variants, but may also process states exponentially many times. Others are variants of breadth-first search (e.g. Beam-Stack search [26] and ABULB [27]) and guarantee completeness. However, these too may process states many times before producing even an initial solution. Our anytime algorithm guarantees that states are never processed more than once while working on *any* single solution. On the other hand, these algorithms are directed towards bounding memory usage and as a result, may scale up to larger domains than ARA*, if memory becomes a bottleneck in obtaining and improving a solution within the provided time window.

Incremental planning algorithms efficiently repair previous solutions when changes are made to the model. A variety of such algorithms have been developed, both by artificial intelligence researchers and by theory researchers [30–50]. Many of these algorithms were developed in the field of symbolic planning rather than graph-based planning. They usually provide no suboptimality bounds on their solutions.

[1] In fact, the latest work on Anytime A* shows that it is possible to incorporate the idea of controlling ϵ into Anytime A* search as well [17].

The rest are graph searches but are primarily concerned with optimal solutions. A number of these algorithms are (or can be viewed as) incremental heuristic searches. The Lifelong Planning A* (LPA*) [41] and Dynamic A* (D*) [40] algorithms directly generalize A* search to an incremental search and are efficient in the sense that they only update states that need to be updated. As we explain later in this paper, the anytime incremental planning algorithm we present can be viewed as an anytime extension of the LPA* and D* algorithms.

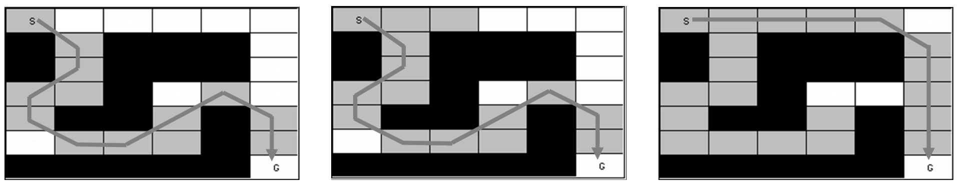
Very few algorithms have been developed that are both anytime *and* incremental and, to the best of our knowledge, all are outside of search-based planning. In particular, the ones we know of have been developed in the framework of symbolic planning. The CASPER system [51], for example, is capable of always returning a plan and constantly works on improving and repairing the plan as changes in the environment are detected. This is achieved, however, at the expense of potentially returning plans that are only partial. A planner described in [52] uses local subplan replacement methodology to quickly repair and then gradually improve a plan whenever changes in the environment invalidate the current plan. Similarly to the anytime incremental search described in section 5 of this paper, it also always returns a complete plan rather than a partial plan. In contrast to our algorithm, however, it provides no guarantees on the suboptimality of its solutions.

4 ARA*: An Anytime A* Search Algorithm with Provable Bounds on Suboptimality

In this section we present the ARA* algorithm. We begin by showing how successive weighted A* searches can be used to produce a simple, naïve anytime algorithm. Next, we discuss a novel formulation of A* that enables us to reuse previous search results. We use this formulation to develop ARA* – an efficient, anytime heuristic search algorithm.

4.1 Using Weighted A Searches to Construct an Anytime Heuristic Search with Provable Suboptimality Bounds*

Normally, the h-values used by A* search are consistent and therefore do not overestimate the cost of paths from states to the goal state. In many domains, however, A* search with inflated heuristics, known as Weighted A* search [53, 54], can drastically reduce the number of states examined before a solution is produced [16, 18–24]. In our framework this is equivalent to processing states in order of $g(s) + \epsilon * h(s)$, rather than $g(s) + h(s)$. While the path the search returns can be suboptimal, the search also provides a bound on this suboptimality, namely, the ϵ by which the heuristic is inflated [25]. (We will often refer to it as ϵ-suboptimality.)



$\epsilon = 2.5$ $\epsilon = 1.5$ $\epsilon = 1.0$ (optimal search)

Fig. 1. A* searches with inflated heuristics

Thus, setting ϵ to 1 results in standard A* with an uninflated heuristic and the resulting path is guaranteed to be optimal. For $\epsilon > 1$ the cost of the returned path is no larger than ϵ times the cost of the optimal path.

For example, Figure 1 shows the operation of the A* algorithm with a heuristic inflated by $\epsilon = 2.5$, $\epsilon = 1.5$, and $\epsilon = 1$ (no inflation) on a simple grid world. In this example we use an eight-connected grid with black cells denoting obstacles. We can extract a graph from this grid by assigning a state to each cell and defining the successors and predecessors of a state to be its adjacent states. S denotes the start state, while G denotes the goal state. The cost of moving from one cell to its neighbor is one. The heuristic used here is the larger of the x and y distances from the cell to the goal. The cells that were expanded are shown in grey. (Our version of A* search stops as soon as it is about to expand a goal state. Thus, the goal state is not shown in grey.) The paths found by these searches are shown with grey arrows. The A* searches with inflated heuristics expand substantially fewer cells than A* with $\epsilon = 1$, but their solutions are suboptimal.

To construct an anytime algorithm with suboptimality bounds, one could run a succession of these A* searches with decreasing inflation factors, just as we did in this example. This naïve approach results in a series of solutions, each with a suboptimality bound equal to the corresponding inflation factor. This approach has control over the suboptimality bound, but wastes a lot of computation since each search iteration duplicates most of the efforts of the previous searches. One could try to employ incremental heuristic searches (e.g. [41]), but the suboptimality bounds for each search iteration would no longer be guaranteed. In the following subsections we introduce the ARA* (Anytime Repairing A*) algorithm, which is an *efficient* anytime heuristic search that also runs a series of A* searches with inflated heuristics but reuses search efforts from previous executions while ensuring that the suboptimality bounds are still satisfied. By not re-computing the state costs that have been correctly computed in previous iterations, the algorithm achieves substantial savings in computation.

```
The pseudocode below assumes the following :
  (1) $g(s_{\text{start}}) = 0$ and $g$-values of the rest of the states are set to $\infty$ (the initialization can also occur whenever Com-
      putePath encounters new states);
  (2) $OPEN = \{s_{\text{start}}\}$.
1 procedure ComputePath()
2 while($s_{\text{goal}}$ is not expanded)
3   remove $s$ with the smallest $f(s)$ from OPEN;
4   for each successor $s'$ of $s$
5     if $g(s') > g(s) + c(s, s')$
6       $g(s') = g(s) + c(s, s')$;
7       insert/update $s'$ in OPEN with $f(s') = g(s') + h(s')$;
```

Fig. 2. A* Search: ComputePath function

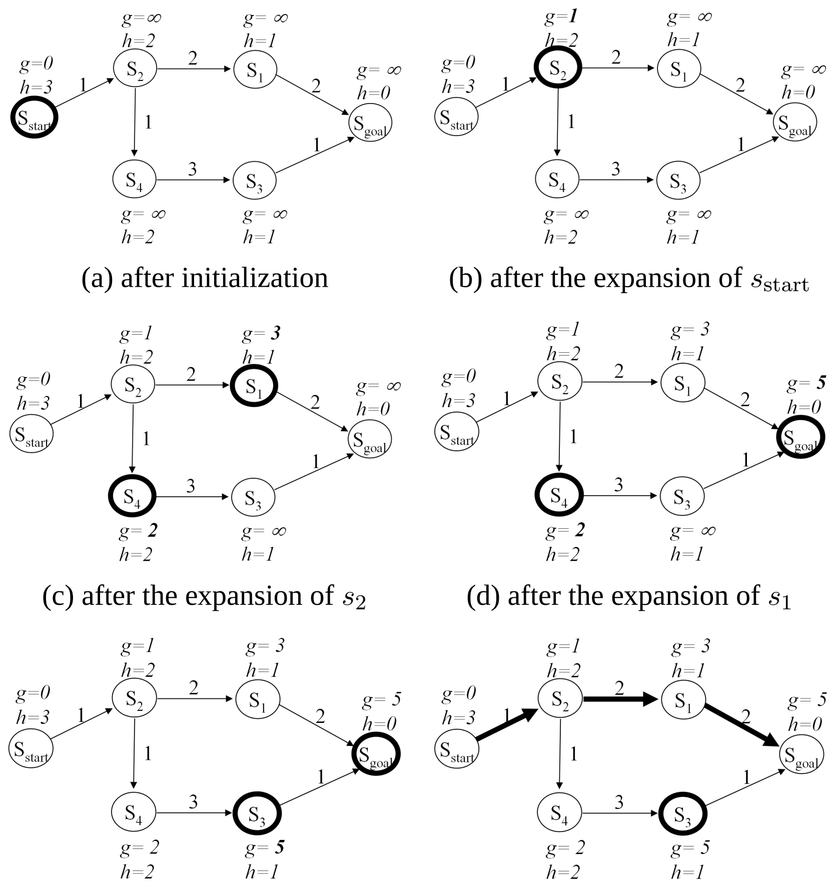
4.2 Reformulating and Generalizing A* Search

We can re-formulate A* search to reuse the search results of its previous executions quite easily. To do this, we define the notion of an inconsistent state (initially introduced in [41]) and then formulate A* search as the repeated expansion of inconsistent states. This formulation can reuse the results of previous executions simply by identifying all of the states that are inconsistent. We will also generalize the priority function that A* uses to any function satisfying certain restrictions. This generalization leads us towards the ARA* algorithm.

4.2.1 Reformulation of A* Search Using Inconsistent States

A standard formulation of A* search is provided in Figure 2. In this pseudocode, *OPEN* is a priority queue containing states to be processed. The priorities according to which states are chosen from *OPEN* are their f-values, the sum of g- and h-values. Since $g(s)$ is the cost of the best path from s_{start} to s found so far, and $h(s)$ estimates the cost of the best path from s to s_{goal}, $f(s)$ is an estimate of the cost of the best path from s_{start} to s_{goal} via state s. If the h-values are admissible, that is, never overestimate the cost of the least-cost path from s to s_{goal}, then A* is guaranteed to find an optimal path. If the h-values are also consistent, that is, for any two states $s, s' \in S$ such that $s' \in succ(s)$, $h(s) \leq c(s, s') + h(s')$, then no state is expanded more than once. The term *expansion* of state s usually refers to the update of g-values of the successors of s (lines 4 through 7). These updates decrease the g-values of the successors of s whenever it is possible to do so using $g(s)$. Once the search finishes, the solution is given by the greedy path. Figure 3 demonstrates the operation of A* search on a simple example. We will later use the same example to show the operation of our alternative formulation of A* search (Figure 7).

We now introduce a new variable, called $v(s)$. The introduction of v-values will not affect the operation of a one-time A* search. However, as we will show later, v-values will make it very easy to extend A* search so that it can reuse the results of previous searches. Intuitively, v-values will also be estimates of start distances, just as the g-values. However, while $g(s)$ is always the cost of the best path found



(a) after initialization (b) after the expansion of s_{start}

(c) after the expansion of s_2 (d) after the expansion of s_1

(e) after the expansion of s_4 (f) after the expansion of s_{goal}

Fig. 3. An example of how A* search operates. The states that have bold borders are in *OPEN*. The g-values that have just changed are shown in bold. After s_{goal} is expanded, a greedy path is computed and is shown in bold.

The pseudocode below assumes the following :

(1) **v-values of all states are set to** ∞, $g(s_{\text{start}}) = 0$ and the g-values of the rest of the states are set to ∞ (the initialization can also occur whenever ComputePath encounters new states);

(2) *OPEN* $= \{s_{\text{start}}\}$.

```
procedure ComputePath()
while($s_{\text{goal}}$ is not expanded)
  remove $s$ with the smallest $f(s)$ from OPEN;
  $\mathbf{v(s) = g(s)}$;
  for each successor $s'$ of $s$
    if $g(s') > g(s) + c(s, s')$
      $g(s') = g(s) + c(s, s')$;
      insert/update $s'$ in OPEN with $f(s') = g(s') + h(s')$;
```

Fig. 4. A* Search: ComputePath function with v-values

so far from s_{start} to s, $v(s)$ is always equal to the cost of the best path found at the time of the last expansion of s. Thus, every v-value is initially set to ∞, as with the corresponding g-value (except for $g(s_{\text{start}})$), and then it is reset to the g-value of the state when the state is expanded. The new pseudocode that uses these v-values is given in Figure 4, with differences from the original version shown in bold.

Since we set $v(s) = g(s)$ at the beginning of the expansion of s, $v(s)$ remains equal to $g(s)$ while s is being expanded (lines 5 through 8). The only way $v(s)$ could become different from $g(s)$ is if $g(s)$ changed during the expansion of s. This is impossible, however, because for this to happen s needs to be a successor of itself with $g(s)$ larger than $g(s) + c(s, s)$ in order to pass the test on line 6. This makes $c(s, s)$ a negative edge cost which is inconsistent with our assumption that all edge costs are positive. As a result, the execution of line 7 is equivalent to setting $g(s') = v(s) + c(s, s')$, and since v-values are updated only for states that are being expanded, one benefit of introducing v-values is the following invariant that A* always maintains: for every state $s' \in S$,

$$g(s') = \begin{cases} 0, & \text{if } s' = s_{\text{start}} \\ \min_{s'' \in pred(s')}(v(s'') + c(s'', s')), & \text{otherwise} \end{cases} \tag{1}$$

More importantly, however, it turns out that *OPEN* contains exactly the states s for which $v(s) \neq g(s)$. This is the case initially, when all states except for s_{start} have both v- and g-values infinite and *OPEN* only contains s_{start} which has $v(s_{\text{start}}) = \infty$ and $g(s_{\text{start}}) = 0$. Afterwards, every time a state is selected for expansion it is removed from *OPEN* (line 3) and its v-value is set to its g-value on the very next line. Finally, whenever the g-value of any state is modified (line 7) it has been decreased and is thus strictly less than the corresponding v-value. After each modification of the g-value, the state is added to *OPEN* if it is not already there (line 8).

Let us call a state s with $v(s) \neq g(s)$ *inconsistent* and a state with $v(s) = g(s)$ *consistent*. Thus, *OPEN* always contains exactly those states that are inconsistent. Consequently, since all the states for expansion are chosen from *OPEN*, A* search expands only inconsistent states.

Here is an intuitive explanation of the operation of A* in terms of inconsistent state expansions. Since at the time of expansion a state is made consistent by setting its v-value equal to its g-value, a state becomes inconsistent as soon as its g-value is decreased and remains inconsistent until the next time the state is expanded. That is, suppose that a consistent state s is the best predecessor for some state s': $s = \arg\min_{s'' \in pred(s')}(v(s'') + c(s'', s'))$. Then

$$g(s') = \min_{s'' \in pred(s')}(v(s'') + c(s'', s')) = v(s) + c(s, s') = g(s) + c(s, s')$$

Thus, the g-value of s is consistent with the g-value of s' in the following sense: the cost of the found path from s_{start} to s' via state s, given by $g(s) + c(s, s')$, can not be used to decrease the g-value of s' any further, $g(s')$ is already equal to it.

Now suppose $g(s)$ decreases. It then becomes strictly smaller than $v(s)$ and therefore $g(s')$ becomes strictly larger than $g(s) + c(s, s')$. In other words, the decrease in $g(s)$ introduces an inconsistency between the g-value of s and the g-value of its successor s' (and possibly other successors of s). Whenever s is expanded, on the other hand, this inconsistency is corrected by setting $v(s)$ to $g(s)$ and re-evaluating the g-values of the successors of s. This in turn may potentially make the successors of s inconsistent. In this way the inconsistency is propagated to the children of s via a series of expansions. Eventually the children no longer rely on s, none of their g-values are lowered, and none of them are inserted into the *OPEN* list.

The operation of this new formulation of A* search is identical to the version in Figure 2. The variable v just makes it easy for us to identify all the states that are inconsistent: these are all the states s with $v(s) \neq g(s)$. In fact, in this version of the ComputePath function, the g-values only decrease, and since the v-values are initially infinite, all inconsistent states have $v(s) > g(s)$. We will call such states *overconsistent*. In later versions of the algorithm we will encounter states s that are *underconsistent*, with $v(s) < g(s)$. Such states will appear in problems with increasing edge costs that lead to increasing g-values.

4.2.2 Generalizing Priorities

A* search uses one possible state expansion ordering: it expands states in the order of increasing f-values. For any admissible heuristic, this ordering guarantees optimality. However, we can generalize A* search to handle more general expansion priorities as long as they satisfy certain restrictions. These restrictions will allow the search to guarantee suboptimality bounds even when heuristics are inadmissible. We first introduce a function key(s) that returns the priority of a state s used for ordering expansions. (For example, key$(s) = g(s)$ corresponds to an uninformed optimal search such as Dijkstra's, key$(s) = g(s) + h(s)$ corresponds to A* search, key$(s) = g(s) + \epsilon * h(s)$ corresponds to Weighted A* search, etc.) Thus, in Figure 4, in line 3 $f(s)$ is replaced with key(s) and line 8 is replaced with

insert/update s' in *OPEN* with key(s');

The rest of the pseudocode remains the same. We restrict key() to be any function satisfying the following restriction

for any two states $s, s' \in S$ *such that* $c^*(s', s_{\text{goal}}) < \infty$, $v(s') \geq g(s')$, $v(s) > g(s)$ *and* $g(s') > g(s) + \epsilon * c^*(s, s')$, *it must hold that key*$(s') >$ *key*(s)

Fig. 5. key restriction

The suboptimality factor ϵ can be any finite real value greater than or equal to one. A simple example illustrates the need for this restriction. Imagine we have two states: state s' that can potentially belong to a path from s_{start} to s_{goal} (i.e., $c^*(s', s_{\text{goal}}) < \infty$) and an overconsistent, and therefore a candidate for expansion, state s. We need to know whether state s' has been computed correctly. In particular, whether the cost of the found path from s_{start} to state s' is no more than ϵ times the cost of an optimal path. The condition $g(s') > g(s) + \epsilon * c^*(s, s')$, however, implies that the g-value of state s' might potentially overestimate the cost of an optimal plan from s_{start} to state s' by more than a factor of ϵ based on the g-value of s. Hence, we can not guarantee that s' has been correctly computed yet, and state s needs to be expanded first so that the path through it can be propagated to s' if it really is a cheaper path. This can be ensured by having key(s) smaller than key(s'). Note that omitting in the key restriction the conditions that involve v-values would also result in a valid restriction but it would be more restrictive. Thus, the key restriction we give does not apply to state s if $v(s) = g(s)$. This is so because the consistency of s means that the path through it has already been propagated, and therefore the expansion of s can not result in finding a cheaper path from s_{start} to s'.

If the restriction in Figure 5 is satisfied then the cost of a greedy path after the search finishes is at most ϵ times larger than the cost of an optimal solution [55]. It is easy to see that the restriction is satisfied by the prioritization of uninformed optimal searches such as Dijkstra's algorithm, the prioritization of A* with consistent heuristics, and the prioritization of A* with consistent heuristics inflated by some constant. For example, in the case of an uninformed optimal search, $g(s') > g(s) + \epsilon * c^*(s, s')$ for any two states $s, s' \in S$ and $\epsilon = 1$ implies that $key(s') = g(s') > g(s) + \epsilon * c^*(s, s') = key(s) + \epsilon * c^*(s, s') \geq key(s)$ since costs cannot be negative. Thus, the solution is optimal. In the case of A* search with consistent heuristics inflated by ϵ, $g(s') > g(s) + \epsilon * c^*(s, s')$ for any two states $s, s' \in S$ implies that

$$\begin{aligned} key(s') = {} & g(s') + \epsilon * h(s') > g(s) + \epsilon * h(s') + \epsilon * c^*(s, s') \geq \\ & g(s) + \epsilon * h(s) = key(s) \end{aligned}$$

where we used the fact that $h(s) \leq c^*(s, s') + h(s')$ when heuristics are consistent [53]. In fact, it can be shown in the exact same way that the restriction holds for $key(s) = g(s) + h(s)$, where heuristics are *any* values satisfying ϵ-consistency [56]: $h(s_{\text{goal}}) = 0$ and for any two states $s, s' \in S$ such that $s' \in succ(s)$, $h(s) \leq \epsilon * c(s, s') + h(s')$. Many different heuristics are ϵ-consistent for a suitable ϵ including consistent heuristics, consistent heuristics inflated by ϵ, the summation of consistent heuristics (as often used in heuristic search-based symbolic planning) and general inadmissible heuristics with bounds on how much they under- and overestimate the true values [56].

In general, when heuristics are inconsistent A* may re-expand states multiple times. However, if we restrict the expansions to no more than one per state, then the algorithm is still complete and possesses ϵ-suboptimality if the heuristic is ϵ-consistent [55]. We restrict the expansions using the set *CLOSED* (Figure 6) in the same way it is often used in A*: initially, *CLOSED* is empty; afterwards, every state that is being expanded is added to this set (line 4) and no state that is already

```
The pseudocode below assumes the following:
  (1) the function key($s$) satisfies the restriction in Figure 5;
  (2) $v-$ and $g-$values of all states are initialized in such a way that $v(s) \geq g(s) = \min_{s' \in pred(s)}(v(s') + c(s', s)) \, \forall s \neq s_{\text{start}}$ and $v(s_{\text{start}}) \geq g(s_{\text{start}}) = 0$ (the initialization can also occur whenever ComputePath
      encounters new states);
  (3) $CLOSED = \emptyset$ and $OPEN$ contains the overconsistent states (i.e., states $s$ whose $v(s) > g(s)$).
1 procedure ComputePath()
2 while(key($s_{\text{goal}}$) $> \min_{s \in OPEN}$ (key($s$)))
3   remove $s$ with the smallest key($s$) from $OPEN$;
4   $v(s) = g(s)$; $CLOSED = CLOSED \cup \{s\}$;
5   for each successor $s'$ of $s$
6     if $g(s') > g(s) + c(s, s')$
7       $g(s') = g(s) + c(s, s')$;
8       if $s' \notin CLOSED$
9         insert/update $s'$ in $OPEN$ with key($s'$);
```

Fig. 6. A* search with a generalized priority function and generalized overconsistent initialization: ComputePath function

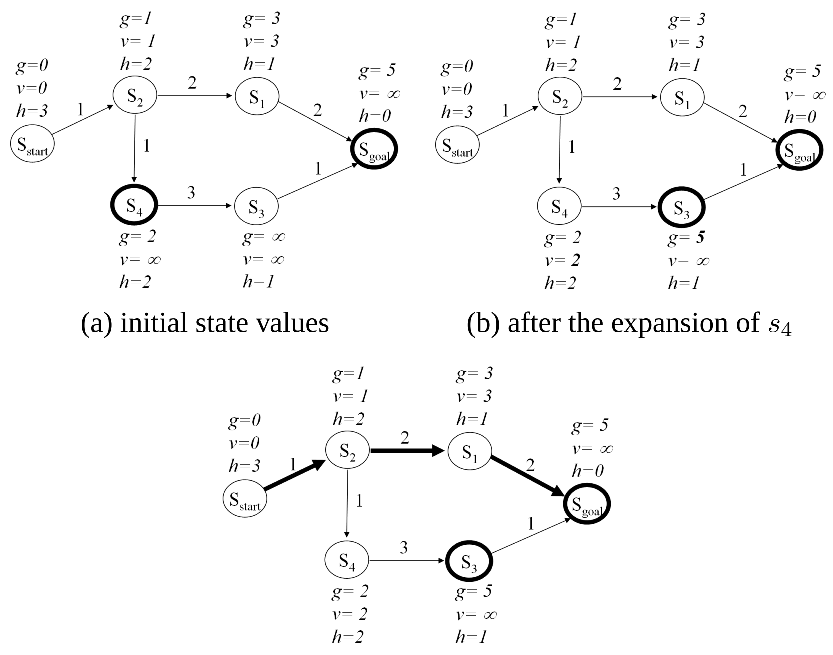
in *CLOSED* is inserted into *OPEN* to be considered for expansion (line 8).

4.2.3 Generalizing to Arbitrary Overconsistent Initialization

In the versions of A* presented so far, all states had their g- and v-values initialized at the outset. We set the v-values of all states to infinity, we set the g-values of all states except for s_{start} to infinity, and we set $g(s_{\text{start}})$ to 0. We now remove this initialization step and the only restriction we make is that no state is underconsistent and all g-values satisfy equation 1 except for $g(s_{\text{start}})$ which is equal to zero. This arbitrary overconsistent initialization will allow us to re-use previous search results when running multiple searches.

The pseudocode under this initialization is shown in Figure 6. It uses the key(s) priority function as described in the previous section. The only change necessary for the arbitrary overconsistent initialization is the terminating condition (line 2) of the while loop. The loop now terminates as soon as key(s_{goal}) becomes less than or equal to the key of the state to be expanded next, that is, the smallest key in *OPEN* (we assume that the $\min$ operator on an empty set returns ∞). The reason for this addition is that under the new initialization s_{goal} may never be expanded if it was already correctly initialized. For instance, if all states are initialized to be consistent, then *OPEN* is initially empty, and the search terminates without a single expansion. This is correct, because when all states are consistent and $g(s_{\text{start}}) = 0$, then for every state $s \neq s_{\text{start}}$, $g(s) = \min_{s' \in pred(s)}(v(s') + c(s', s)) = \min_{s' \in pred(s)}(g(s') + c(s', s))$, which means that the g-values are equal to the corresponding g^*-values and no search is necessary—the greedy path is an optimal solution.

In Figure 7 we show the operation of this version of A* search. Some of the initial state values are already finite. These values, for example, could have been generated by previous searches. Such will be the case with the ARA* algorithm below, which executes the ComputePath function repeatedly, gradually improving its solution. Because some states are already consistent, the search in the example needs to



(a) initial state values (b) after the expansion of s_4

(c) after the computation of a greedy path

Fig. 7. An example of how the ComputePath function operates under an arbitrary overconsistent initialization. States are expanded in the order of f-values (i.e. $key(s) = g(s) + h(s)$). All initially overconsistent states need to be in *OPEN*. The states in *OPEN* are shown with bold borders. The g- and v-values that have just changed are shown in bold. After the search terminates, a greedy path is computed and is shown in bold. Note that the computed greedy path and all g-values are equivalent to those generated by regular A* search (Figure 3).

expand only one state to obtain an optimal path.

This version of A* search has a number of nice properties. The central property of the search is that it maintains the following invariant after each expansion.

Theorem 1 *At line 2, for any state s with ($c^*(s, s_{\text{goal}}) < \infty \wedge$ key$(s) \leq$ key(u) $\forall u \in$ OPEN), it holds that $g^*(s) \leq g(s) \leq \epsilon * g^*(s)$.*

In other words, every state s that may theoretically be on a path from s_{start} to s_{goal} ($c^*(s, s_{\text{goal}}) < \infty$) and whose key is less than or equal to the smallest key in *OPEN* has a g-value that is at worst ϵ-suboptimal and therefore does not have to be processed anymore. Since the g-value of s is the cost of the best path found so far from s_{start} to s, this path is at most ϵ-suboptimal. Given this property and the terminating condition of the algorithm (line 2), it is clear that after the algorithm terminates, $g(s_{\text{goal}}) \leq \epsilon * g^*(s_{\text{goal}})$ and the greedy path from s_{start} to s_{goal} is at most ϵ-suboptimal.

Theorem 2 *When the ComputePath function exits the following holds: $g^*(s_{\text{goal}}) \leq g(s_{\text{goal}}) \leq \epsilon * g^*(s_{\text{goal}})$ and the cost of a greedy path from s_{start} to s_{goal} is no larger* *than* $\epsilon * g^*(s_{\text{goal}})$.

As with A* search with consistent heuristics, this version guarantees no more than one expansion per state.

Theorem 3 *No state is expanded more than once during the execution of the ComputePath function.*

Additionally, the following theorem shows that when the search is executed with a non-trivial initialization of states, such as state values from previous searches, the states with the v-values that cannot be lowered are not expanded. This can result in substantial computational savings when using this search for repeated planning as discussed in the next section.

Theorem 4 *A state s is expanded only if $v(s)$ is lowered during its expansion.*

*4.3 An Efficient Version of Anytime Search with Provable Suboptimality Bounds: ARA**

The formulation of A* search presented in Figure 6 allows for the results of previous searches to be used in successive executions of the algorithm. As explained, the search only expands the states that are inconsistent (in fact, a subset of them) and tries to make them consistent. Therefore, if we begin with a number of consistent states due to some previous search efforts, these states need not be expanded again unless they become inconsistent during the search. Consequently, to reuse previous search efforts we only need to make sure that before each execution of the ComputePath function *OPEN* contains *all* the inconsistent states. Since the ComputePath function restricts each state to no more than one expansion during each search iteration, *OPEN* may not contain all inconsistent states during the execution of ComputePath. In fact, *OPEN* contains only the inconsistent states that have not yet been expanded. We need, however, to keep track of *all* inconsistent states since they will be used to initialize *OPEN* in future searches. We do this by maintaining a set *INCONS* of all the inconsistent states that are not in *OPEN*. Or, in other words, *INCONS* is a set of all the inconsistent states that are in *CLOSED*. Thus, the union of *INCONS* and *OPEN* is exactly the set of all inconsistent states, and can be used as a starting point for the inconsistency propagation before each new search iteration.

Figure 8 presents the ComputePath function of Anytime Repairing A* (ARA*). ARA* executes A* multiple times, starting with a large ϵ and decreasing this value prior to each execution until $\epsilon = 1$. Each search reuses the results of previous searches by maintaining an *INCONS* set as mentioned above. Apart from the maintenance of this set, the ComputePath function of ARA* is almost identical to the ComputePath function of A* search as presented in Figure 6. The only other dif-

```
 1 procedure ComputePath()
 2 while(key($s_{\mathrm{goal}}$) $> \min_{s \in OPEN}$(key($s$)))
 3   remove $s$ with the smallest key($s$) from OPEN;
 4   $v(s) = g(s)$; CLOSED $=$ CLOSED $\cup \{s\}$;
 5   for each successor $s'$ of $s$
 6     if $s'$ was never visited by ARA* before then
 7       v($s'$) = g($s'$) = $\infty$;
 8     if $g(s') > g(s) + c(s, s')$
 9       $g(s') = g(s) + c(s, s')$;
10       if $s' \notin$ CLOSED
11         insert/update $s'$ in OPEN with key($s'$);
12       else
13         insert $s'$ into INCONS;
```

Fig. 8. ARA*: ComputePath function. ARA* specific changes as compared with A* search as formulated in Figure 6 are shown in bold.

ference is the explicit initialization of states as they are encountered by ARA*. Note that each state is initialized once per ARA* execution and *not* every time ComputePath encounters it for the first time during its current search. The key() function used by ComputePath is a summation of the state's g-value and its h-value inflated by the current value of ϵ, as given in Figure 9.

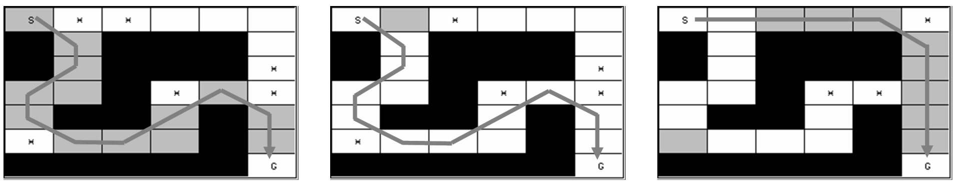
Figure 9 also presents the Main function of ARA*, which performs a series of search iterations. It first initializes the search and then repeatedly calls the ComputePath function with a series of decreasing values of ϵ. Before each call to the ComputePath function, however, a new *OPEN* list is constructed by moving to it the contents of the set *INCONS*. Consequently, *OPEN* contains all inconsistent states before each call to ComputePath. Since the *OPEN* list has to be sorted by the current key-values of states, it is re-ordered between calls to ComputePath (line 12). [2]

```
The pseudocode below assumes the following:
   (1) heuristics are consistent: $h(s) \leq c(s, s') + h(s')$ for any successor $s'$ of $s$ if $s \neq s_{\mathrm{goal}}$ and $h(s) = 0$ if $s = s_{\mathrm{goal}}$.
 1 procedure key($s$)
 2 return $g(s) + \epsilon * h(s)$;

 3 procedure Main()
 4 $g(s_{\mathrm{goal}}) = v(s_{\mathrm{goal}}) = \infty$; $v(s_{\mathrm{start}}) = \infty$;
 5 $g(s_{\mathrm{start}}) = 0$; OPEN = CLOSED = INCONS = $\emptyset$;
 6 insert $s_{\mathrm{start}}$ into OPEN with key($s_{\mathrm{start}}$);
 7 ComputePath();
 8 publish current $\epsilon$-suboptimal solution;
 9 while $\epsilon > 1$
10   decrease $\epsilon$;
11   Move states from INCONS into OPEN;
12   Update the priorities for all $s \in$ OPEN according to key($s$);
13   CLOSED $= \emptyset$;
14   ComputePath();
15   publish current $\epsilon$-suboptimal solution;
```

Fig. 9. ARA*: key and Main functions

[2] At least in our domains, the reordering operation tends to be inexpensive in comparison to the overall search. If necessary, however, one could also employ the optimization discussed in [40, 57] in the context of the D* and D* Lite algorithms. This avoids the reordering operation altogether.



initial search ($\epsilon = 2.5$) second search ($\epsilon = 1.5$) third search ($\epsilon = 1.0$)

Fig. 10. ARA* search

After each call to the ComputePath function we get a solution that is suboptimal by at most a factor of ϵ. Similarly to how it is done in [17], a suboptimality bound for the solution returned by ARA* can also be computed as the ratio between $g(s_{\text{goal}})$, which gives an upper bound on the cost of an optimal solution, and the minimum un-weighted f-value of any inconsistent state, which gives a lower bound on the cost of an optimal solution:

$$\frac{g(s_{\text{goal}})}{\min_{s \in OPEN \cup INCONS}(g(s) + h(s))} \tag{2}$$

This is a valid suboptimality bound as long as the ratio is greater than or equal to one. Otherwise, $g(s_{\text{goal}})$ is already equal to the cost of an optimal solution. Thus, the actual suboptimality bound, ϵ', for each solution ARA* publishes can be computed as the minimum between ϵ and this new bound.

$$\epsilon' = \min(\epsilon, \frac{g(s_{\text{goal}})}{\min_{s \in OPEN \cup INCONS}(g(s) + h(s))}). \tag{3}$$

At first, one might think of using this actual suboptimality bound for deciding how to decrease ϵ between search iterations (e.g., setting ϵ to ϵ' minus a small delta). This can lead to large jumps in ϵ, however, whereas based on our experiments decreasing ϵ in small steps seems to be more beneficial. The reason for this is that a small decrease in ϵ often results in the improvement of the solution, despite the fact that the actual suboptimality bound of the previous solution was already substantially less than the value of ϵ. A large decrease in ϵ, on the other hand, may often result in the expansion of many states during the next search, resulting in a large computation time for the search.

Another useful suggestion from [17], which we have not implemented in ARA*, is to prune *OPEN* so that it never contains a state whose un-weighted f-value is larger than or equal to $g(s_{\text{goal}})$. This may turn out to be useful in domains with very high branching factors, where the expansion of one state may involve inserting into *OPEN* a large number of states that will never be expanded due to their large f-values.

Within each execution of the ComputePath function, computation is saved by not re-expanding the states whose v-values were already correct before the call to ComputePath. For example, Figure 10 shows a series of calls to the ComputePath function on the same example used in Figure 1. States that are inconsistent at the end of an iteration are shown with an asterisk. While the first call ($\epsilon = 2.5$) is identical to the A* call with the same ϵ, the second call to the ComputePath function ($\epsilon = 1.5$) expands only 1 cell. This is in contrast to 15 cells expanded by A* search with the same ϵ. For both searches the suboptimality factor ϵ decreases from 2.5 to 1.5. Finally, the third call to the ComputePath function with ϵ set to 1 expands only 9 cells. The solution is now optimal, and the total number of expansions is 23. Only 2 cells are expanded more than once across all three calls to the ComputePath function. Even a single optimal search from scratch expands 20 cells. As shown in the example, ARA* is an efficient way of computing a series of solutions that satisfy gradually decreasing sub-optimality bounds. This property of the algorithm makes it well-suited for planning under time constraints, when one needs to find the best solution possible within a particular time.

If we are interested in interleaving search with the execution of the current best plan, then we need to address the scenario where the state of the agent, and hence s_{start}, is changing. One way to deal with this problem is to perform the search backwards. That is, the goal state of the agent becomes the start of the search, s_{start}, while the current state of the agent becomes the goal of the search, s_{goal}. This way, the start of the search does not change when the agent moves and the existing g-values remain valid. The search can still be performed on directed graphs by reversing the direction of all the edges in the original graph. Since heuristics estimate the distances to the goal of the search, then in this backward search they estimate the distances from the current state of the agent to states in question. As a result, the heuristics change as the agent moves. This in turn alters the priorities of the states in *OPEN*. Since ARA* reorders *OPEN* after each iteration anyway, however, we can recompute the heuristic values of the states in *OPEN* during the reorder operation (line 12 in Figure 9).

*4.4 Theoretical Properties of ARA**

ARA* inherits all of the properties of the version of A* presented in section 4.2.3. We now list two of the most important of these properties. For the proofs of these and other properties of the algorithm please refer to [58]. The first theorem states that, for any state s with a key smaller than or equal to the minimum key in *OPEN*, we have computed a greedy path from s_{start} to s whose cost is within a factor of ϵ of the least-cost path.

Theorem 5 *Whenever the ComputePath function exits, for any state s with $key(s) \leq \min_{s' \in \text{OPEN}}(key(s'))$, we have $g^*(s) \leq g(s) \leq \epsilon * g^*(s)$, and the cost of a greedy path from s_{start} to s is no larger than $g(s)$.*

The correctness of ARA* follows from this theorem. Each execution of the ComputePath function terminates when key(s_{goal}) is no larger than the minimum key in *OPEN*. This means that the greedy path from start to goal is within a factor ϵ of optimal. Since ϵ is decreased before each iteration, ARA* gradually decreases the suboptimality bound and finds new solutions to satisfy the bound.

Theorem 6 *Each call to ComputePath() expands a state at most once and only if its v-value is lowered during its expansion.*

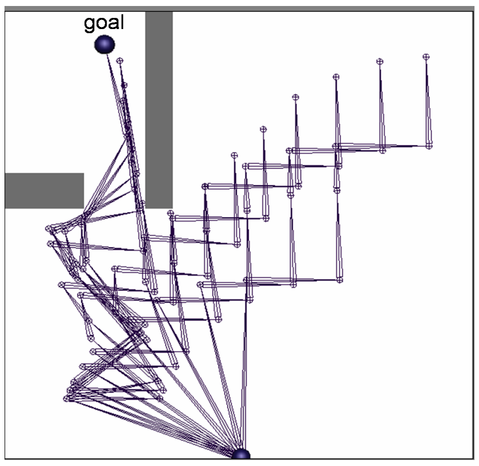
The second theorem formalizes how ARA* saves computation. A usual implementation of A* search with inflated heuristics performs multiple re-expansions of many states. Each search iteration in ARA*, on the other hand, is guaranteed to expand each state at most once. Also, it does not expand states whose v-values before the current call to the ComputePath function have already been correctly computed. This theorem is important because the processing done by the ComputePath function usually dominates all other processing and in particular the insertion of *INCONS* into *OPEN* and the re-ordering of *OPEN*. In the worst-case, however, the re-ordering of *OPEN* can be on the order of $O(|S|\,log(|S|))$.

*4.5 Experimental Analysis of the Performance of ARA**

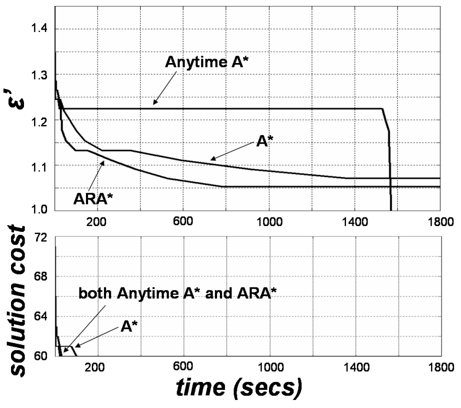
In this section we evaluate the performance of ARA* on simulated 6 and 20 degree of freedom (DOF) robotic arms (Figures 11 and 12) and compare it against other anytime heuristic searches that can provide suboptimality bounds, namely, Anytime A* [17] and a succession of A* searches with decreasing ϵ values (as described in section 4.1). The base of the arm is fixed, and the task is to move its end-effector to a goal position while navigating around obstacles (indicated by grey rectangles). An action is defined as a change of a global angle of any particular joint (i.e., the next joint further along the arm rotates in the opposite direction to maintain the global angle of the remaining joints.) We discretize the workspace into a grid of 50 by 50 cells. A single (and nearly instantaneous) execution of a 2D version of Dijkstra's algorithm is used to compute a distance from each cell to the cell containing the goal. This distance measure takes into account that some cells are occupied by obstacles and is used as our heuristic. In environments where all the obstacles are connected to the boundary, this heuristic directs the arm in an approximately correct direction.[3] This property allows all the three algorithms we compare to provide anytime behavior.

In order for the heuristic not to overestimate true costs, the actions are discretized

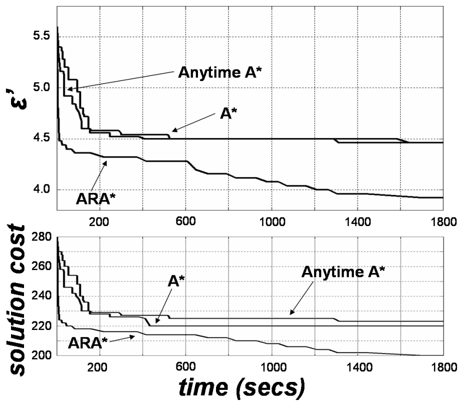
[3] In environments with obstacles floating in the air, these 2D heuristics may not guide well. For example, the heuristics may advocate that the robot arm should move above an obstacle placed in the middle of the workspace, while the robot arm is short enough that it can only move below the obstacle.



(a) 6D arm trajectory for $\epsilon = 3$



(b) uniform cost results



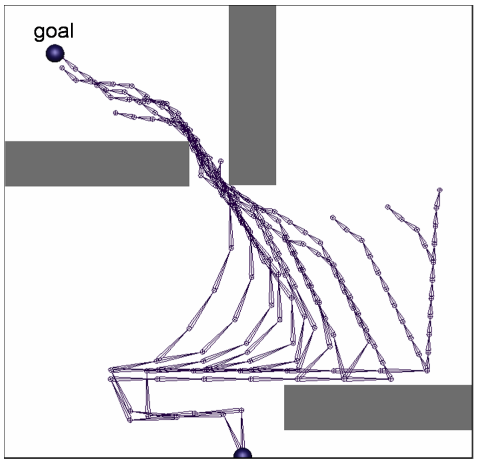
(c) non-uniform cost results

Fig. 11. 6D robot arm experiments. Anytime A* is the algorithm presented in [16].

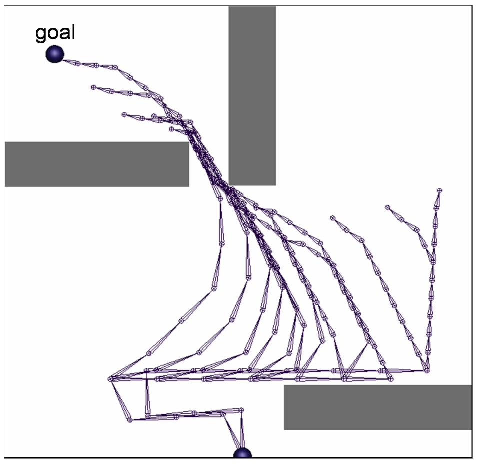
so as to never move the end-effector by more than one cell. The resulting state space is over 3 billion states for a 6 DOF robot arm and over 10^{26} states for a 20 DOF robot arm. Memory for the states is dynamically allocated.

Figure 11(a) shows the planned trajectory of the robot arm after the initial search of ARA* with $\epsilon = 3.0$. This search takes about 0.05 secs. The plot in Figure 11(b) shows that ARA* improves both the quality of the solution and the bound on its suboptimality faster and in a more gradual manner than either a succession of Weighted A* searches or Anytime A* [17].[4] In this experiment ϵ is initially set to 3.0 for all three algorithms. For all these experiments ϵ is decreased in steps of 0.02 (2% suboptimality) for ARA* and the succession of Weighted A* searches. Anytime A* does not control ϵ, and in this experiment its suboptimality bound decreases sharply at the end of its search. On the other hand, it reaches the optimal solution first. To evaluate the expense of the anytime property of ARA* we also ran

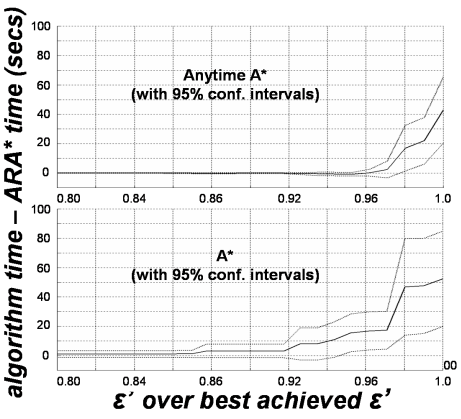
[4] The latest experimental analysis of Anytime A* and ARA*, however, suggests that the actual difference in performances between the two algorithms may depend on the properties of the domain [17].



(a) Anytime A* and A* after 90 seconds



(b) ARA* after 90 seconds



(c) performance results

Fig. 12. 20D robot arm experiments (the trajectories shown are downsampled by 6). Anytime A* is the algorithm presented in [17]. The costs of the solutions shown in (a) and (b) are 682 and 657, respectively.

ARA* and an optimal A* search in an environment with a gap between the obstacles large enough for the optimal A* search to become feasible (this environment is not shown in the figures). Optimal A* search required about 5.3 mins (2,202,666 states expanded) to find an optimal solution, while ARA* required about 5.5 mins (2,207,178 states expanded) to decrease ϵ in steps of 0.02 from 3.0 until a provably optimal solution was found. This represents an overhead of 4%. In other domains such as path planning for robot navigation, though, we have observed the overhead to be up to 30%. While decreasing ϵ, it is often the case that a search iteration expands no states. The termination criterion for the while loop (line 2 in Figure 8) is satisfied as soon as the ComputePath function is entered. The overhead of such iterations is then purely due to reordering the heap. For the domains we have experimented with, the computational expense of this operation is usually substantially less than the computational expense of efforts spent on expanding states.

In the experiment shown in Figure 11(b) all the actions have the same cost. We also experimented with non-uniform costs, to represent the scenario where changing a joint angle closer to the base is more expensive than changing a joint angle further away. Because of the non-uniform costs, our heuristic becomes less informative, and so all searches are much more expensive. In this experiment we start with $\epsilon = 10$, and run all algorithms for 30 minutes. At the end, ARA* achieves a solution with a substantially smaller cost (200 vs. 220 for the succession of A* searches and 223 for Anytime A*) and a better suboptimality bound (3.92 vs. 4.46 for both the succession of A* searches and Anytime A*). Also, since ARA* controls ϵ it decreases the cost of the solution gradually. The results of this experiment are shown in Figure 11(c). ARA* reaches a suboptimality bound $\epsilon' = 4.5$ after about 59,000 expansions and 11.7 secs, while the succession of A* searches reaches the same bound after 12.5 million expansions and 27.4 minutes (corresponding to a 140-fold speedup by ARA*) and Anytime A* reaches it after over 4 million expansions and 8.8 minutes (corresponding to a 44-fold speedup by ARA*). Similar results hold when comparing the amount of work each of the algorithms spend on obtaining a solution of cost 225. While Figure 11 shows execution time, the comparison of states expanded (not shown) is almost identical. Additionally, to demonstrate the advantage of ARA* expanding each state no more than once per search iteration, we compare the first searches of ARA* and Anytime A*: the first search of ARA* performs 6,378 expansions, while Anytime A* performs 8,994 expansions, mainly because some of the states are expanded up to seven times before the initial solution is found.

Figures 12(a-c) show the results of experiments performed on a 20 DOF robot arm, with actions that have non-uniform costs. All three algorithms start with $\epsilon = 30$. Figures 12(a) and 12(b) show that in 90 seconds of planning the cost of the trajectory found by ARA* and the suboptimality bound it can guarantee are substantially smaller than for the other algorithms. For example, the trajectory in Figure 12(a) contains more steps and also makes one extra change in the angle of the third joint from the base of the arm (despite the fact that changing lower joint angles is very expensive) in comparison to the trajectory in Figure 12(b). The graph in Figure 12(c) compares the performance of the three algorithms on twenty randomized environments similar to the environment in Figure 12(a). The environments had random goal locations, and the obstacles were slid to random locations along the boundary. The graph shows the additional time the other algorithms require to achieve the same suboptimality bound reached by ARA*. To make the results from different environments comparable, we normalize the bound by dividing it by the maximum of the best bounds that the algorithms achieve before they run out of memory. Averaging over all environments, the time for ARA* to achieve the best bound was 10.1 secs. Thus, the difference of 40 seconds at the end of the Anytime A* graph corresponds to an overhead of about a factor of 4.

5 Anytime D*: An Anytime Incremental A* Search Algorithm with Provable Bounds on Suboptimality

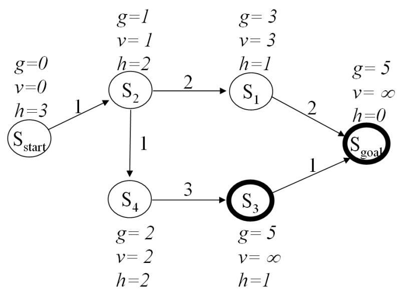
The ARA* algorithm efficiently provides anytime performance when the graph remains unchanged. However, in common real-world applications it is rarely the case that the initial graph perfectly models the planning problem. For example, if a robot navigating to its goal begins without a perfect map of the environment, or the environment is dynamic, then the robot will have to update its graph over time and re-plan. In this section, we present an algorithm that is able to improve its solutions over time *and* repair its solutions when changes are made to any part(s) of the graph. We begin by discussing how changes to the graph violate key properties of ARA*. Next, we discuss how incremental planners are able to repair their solutions when the graph changes. We then combine the major ideas behind ARA* and incremental planners to develop Anytime D* – an efficient anytime, incremental search algorithm.

5.1 The Effect of Cost Changes

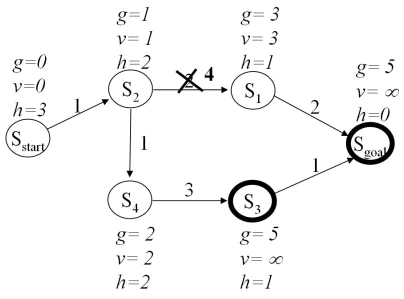
In ARA* the ComputePath function is executed multiple times for different values of ϵ, but always on the same graph. The gradual decrease of ϵ provides the anytime behavior of the algorithm. Unfortunately, ARA* is not able to handle arbitrary edge cost changes. This is because its ComputePath function assumes that all states are either consistent or overconsistent, and this assumption can be violated when edge costs change. To satisfy the requirement that all g-values are one step look-ahead values based on the v-values of the predecessors, that is, for any state $s \neq s_{\text{start}}$, $g(s) = \min_{s' \in pred(s)}(v(s') + c(s', s))$, we need to update the g-values of states for which the costs of incoming edges have changed, and these g-values may become bigger or smaller than their corresponding v-values.

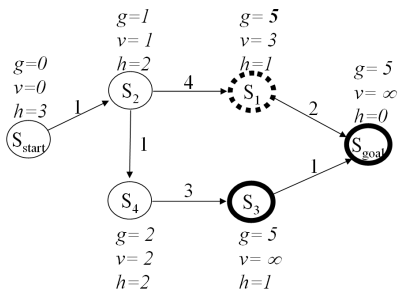
Let us first consider the simpler scenario where edge costs can only decrease. Then, as we update the g-values of the states for which the costs of incoming edges have changed, they also can only decrease. This means that if a state s was not underconsistent before some edge cost $c(s', s)$ decreased, then it cannot become underconsistent due to the edge cost decrease either. This means that all the assumptions of the ComputePath function will still be satisfied. Thus, the same ComputePath function of ARA* can be used to handle decreasing edge costs.

The case of increasing edge costs is more troublesome. As we update the g-values of states for which the costs of incoming edges have increased, these values may also increase. As such they may become larger than the corresponding v-values, and states may become underconsistent. An example demonstrating this is shown in Figure 13. The initial state values are the same as in Figure 7 after the ComputePath



(a) state values after the previous execution of ComputePath (Figure 7(c))





(b) after the cost $c(s_2, s_1)$ changes (c) after the g-value of s_1 is updated

Fig. 13. An example of how an underconsistent state is created as a result of increasing the cost of an edge. Overconsistent states are shown with solid bold borders. Underconsistent states are shown with dashed bold borders. The g-values that have just changed are shown in bold.

function terminated. We now, however, change the cost of the edge from s_2 to s_1 and update the g-value of s_1 accordingly. This results in s_1 becoming an underconsistent state. Unfortunately, the presence of this underconsistent state violates the assumption that no state is underconsistent before a call to the ComputePath function.

Lifelong Planning A* (LPA*) is an incremental version of A* that computes a shortest path repeatedly, updating edge costs in the graph in between each execution [41]. In the following section we briefly explain the ComputePath function of LPA* in a similar manner to how the ComputePath function of ARA* was presented. We then show how it can be combined with ARA* to provide an anytime incremental search.

*5.2 The ComputePath function of LPA**

Unlike the ComputePath function of ARA*, the ComputePath function of LPA* can operate even when underconsistent states exist in the graph. The way the ComputePath function of LPA* handles these states is based on a simple idea: every underconsistent state s can be made either consistent or overconsistent by setting its v-value to ∞. However, by setting $v(s) = \infty$ for each underconsistent state s, the g-values of successors of s may increase, making these successors undercon-

```
 1 procedure FixInitialization()
 2 $Q = \{s \mid v(s) < g(s)\}$;
 3 while($Q$ is non-empty)
 4   remove any $s$ from $Q$;
 5   $v(s) = \infty$;
 6   for each successor $s'$ of $s$
 7     if $s' \neq s_{\text{start}}$
 8       $g(s') = \min_{s'' \in pred(s')} v(s'') + c(s'', s')$;
 9       if $(v(s') < g(s')$ AND $s' \notin Q)$
10         insert $s'$ into $Q$;
```

Fig. 14. Pseudocode that forces all states to become either consistent or overconsistent

```
The pseudocode below assumes the following:
  (1) the function key($s$) satisfies the restriction in Figure 5 and the restriction in Figure 16;
  (2) $v-$ and $g-$ values of all states are initialized in such a way that all the $v$-values are non-negative, $g(s_{\text{start}}) = 0$
      and for every state $s \neq s_{\text{start}}$ $g(s) = \min_{s' \in pred(s)}(v(s') + c(s', s))$ (the initialization can also occur whenever
      ComputePath encounters new states);
  (3) initially, $CLOSED = \emptyset$ and $OPEN$ contains exactly all inconsistent states (i.e., states $s$ whose $v(s) \neq g(s)$).
 1 procedure UpdateSetMembership($s$)
 2 if $v(s) \neq g(s)$
 3   if ($s \notin CLOSED$) insert/update $s$ in $OPEN$ with key($s$);
 4 else
 5   if ($s \in OPEN$) remove $s$ from $OPEN$;

 6 procedure ComputePath()
 7 while(key($s_{\text{goal}}$) $> \min_{s \in OPEN}$(key($s$)) OR $v(s_{\text{goal}}) < g(s_{\text{goal}})$)
 8   remove $s$ with the smallest key($s$) from $OPEN$;
 9   if $v(s) > g(s)$
10     $v(s) = g(s)$; $CLOSED = CLOSED \cup \{s\}$;
11     for each successor $s'$ of $s$
12       if $g(s') > g(s) + c(s, s')$
13         $g(s') = g(s) + c(s, s')$; UpdateSetMembership($s'$);
14   else //propagating underconsistency
15     $v(s) = \infty$; UpdateSetMembership($s$);
16     for each successor $s'$ of $s$
17       if $s' \neq s_{\text{start}}$
18         $g(s') = \min_{s'' \in pred(s')} v(s'') + c(s'', s')$; UpdateSetMembership($s'$);
```

Fig. 15. ComputePath function that expands both overconsistent and underconsistent states

sistent. Thus, these successors need to have their v-values set to ∞ also. Figure 14 provides the pseudocode implementing this idea.

This is a simple way of forcing all states to be either consistent or overconsistent. The computational expense of the pseudocode in Figure 14 though, can become a burden since *every* underconsistent state in the graph is fixed. LPA* incorporates this method of fixing states into the ComputePath function itself and therefore does it only for the states that need to be fixed rather than *all* underconsistent states.

The ComputePath function of LPA* that achieves this is shown in Figure 15. The version we show can handle inflated heuristics, just like the ComputePath function of ARA*. Notice that its second assumption does not require that there are no underconsistent states. The function fixes underconsistent states by expanding them (lines 15 through 18). This means that *OPEN*, the list of candidates for expansion, now needs to contain both underconsistent and overconsistent states. The function UpdateSetMembership inserts inconsistent states into *OPEN* unless they have already been expanded as overconsistent (i.e., they are in *CLOSED*) and removes states from *OPEN* that are consistent. This function is called every time a g- or v-value is modified except for at line 10, where s is consistent and has just been removed from *OPEN*. Initially, *OPEN* must contain *all* inconsistent states, regardless of whether they are overconsistent or underconsistent (this is the third assumption in Figure 15).

To ensure that when an overconsistent state s' is expanded its g-value is no more than ϵ times its g^*-value, we need to make sure that all the states that can possibly belong to the current greedy path from s_{start} to s' are fixed so that they are not underconsistent. To do this, we require that all underconsistent states that could belong to a path from s_{start} to s' are expanded before s' is expanded. This places a second constraint on the state priorities in *OPEN*:

for any two states $s, s' \in S$ *such that* $c^*(s', s_{\text{goal}}) < \infty$, $v(s') \geq g(s')$, $v(s) < g(s)$ *and* $g(s') \geq v(s) + c^*(s, s')$, *it holds that* $\textit{key}(s') > \textit{key}(s)$

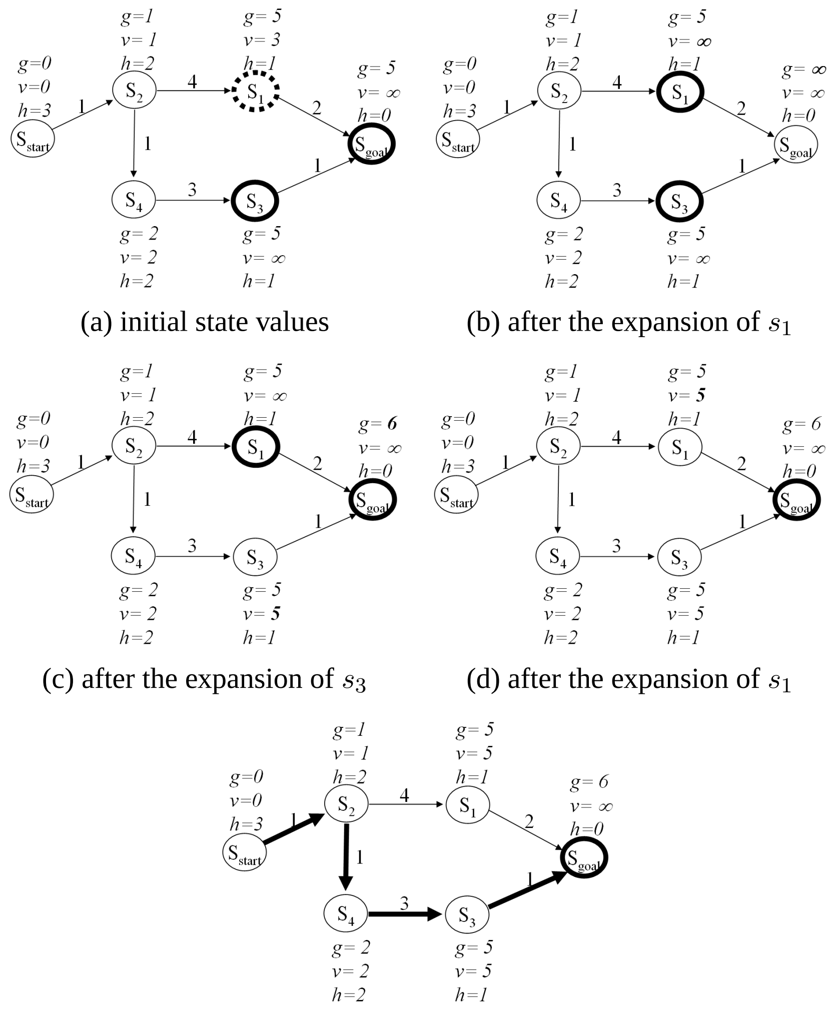
Fig. 16. additional key restriction

Unlike the first restriction, shown in Figure 5, this restriction places the constraints on the priorities of s' and s when s is underconsistent. In the first restriction, s was overconsistent. The new restriction can be described as follows. Given an overconsistent or consistent state s' that can potentially belong to a path from s_{start} to s_{goal} (i.e., $c^*(s', s_{\text{goal}}) < \infty$) and an underconsistent state s, the current path from s_{start} to s' may potentially contain s if $g(s') \geq v(s) + c^*(s, s')$. Therefore, s needs to be expanded first and so key(s) needs to be strictly smaller than key(s'). If there exists no underconsistent state s such that $g(s') \geq v(s) + c^*(s, s')$ then there is no underconsistent state on the current greedy path from s_{start} to s'.

The function also makes sure that when the search terminates s_{goal} itself is not underconsistent. The second part of the terminating condition, namely, $v(s_{\text{goal}}) < g(s_{\text{goal}})$ ensures that the search continues to expand states until s_{goal} is either consistent or overconsistent. Figure 17 demonstrates the operation of the ComputePath function of LPA* when some states are initially underconsistent. The initial state values are the same as in Figure 13(c).

While the pseudocode in Figure 15 is correct, there remains one significant optimization. The re-evaluation of g-values in line 18 is an expensive operation as it requires us to iterate over all predecessors of s'. We can decrease the number of times this re-evaluation is done if we notice that it is invoked when state s is expanded as underconsistent and therefore its v-value is increased to ∞. Therefore, only those successors of s whose g-values depend on s can be affected. To keep track of these states we maintain back-pointers. For state $s' = s_{\text{start}}$, $bp(s') = \textbf{null}$. For all other states generated by search,

$$bp(s') = \arg \min_{s'' \in pred(s')} v(s'') + c(s'', s') \qquad (4)$$



(a) initial state values

(b) after the expansion of s_1

(c) after the expansion of s_3

(d) after the expansion of s_1

(e) after the computation of a greedy path

Fig. 17. An example of how the ComputePath function operates under an arbitrary initialization. The example uses the prioritization function $key(s) = [\min(g(s), v(s)) + h(s); \min(g(s), v(s))]$. This function satisfies the required restrictions. All inconsistent states need to be in *OPEN* initially. Overconsistent states are shown with solid bold borders. Underconsistent states are shown with dashed bold borders. The g- and v-values that have just changed are shown in bold. After the search terminates, a greedy path is computed and is shown in bold. The computed greedy path and all the g-values are the same as those generated by A* search (provided it broke ties in a certain manner when selecting states with the same f-values for expansion).

This is similar to how A* search maintains back-pointers to reconstruct the solution. Whenever a state is expanded, in addition to updating the g-values of its successors, we now also need to update their backpointers so that equation 4 holds. In fact, if a back-pointer is set first, then the g-value is just set based on the new state the back-pointer points to (lines 18 and 26 in Figure 18). The optimization is that in case of an underconsistent state expansion, the re-evaluation of a g-value is now only done for the state whose back-pointer points to the state being expanded (line 24 in Figure 18). In addition, a greedy path, and hence the solution, can now be reconstructed in a backward fashion by just following back-pointers from s_{goal} to s_{start}. We will refer to the path re-constructed in this way as the path defined by back-pointers.

*5.3 An Efficient Anytime Incremental Search Algorithm: Anytime D**

Thus far, we have explained an anytime search algorithm suitable for solving complex planning problems with limited deliberation time (ARA*) and an incremental search algorithm suitable for planning in dynamic domains (LPA*). We now combine these two algorithms into a single anytime incremental search algorithm, which we call Anytime D* (where D* stands for Dynamic A*, as in [40]). We will often refer to Anytime D* simply as AD*. AD* can plan under time constraints, just like ARA*, but is also able to reuse previous planning efforts in dynamic domains.

Both ARA* and LPA* re-use their previous search efforts when executing the ComputePath function. The difference is that before each call to the ComputePath function ARA* changes the suboptimality bound ϵ, while LPA* changes one or more edge costs in the graph. The Anytime D* algorithm should be able to do both types of changes simultaneously, so that it can improve a solution by decreasing ϵ even when the model of a problem changes slightly as reflected in the edge cost changes.

It turns out that the version of the ComputePath function that we have described in section 5.2 is already sufficient to handle both of these types of changes. (The ComputePath function of the original LPA* [41] can not handle changes in ϵ as it can only search for optimal solutions.) Just like the ComputePath function of ARA* it can handle consistent and overconsistent states. In addition, it can also handle underconsistent states which can be created when some edge costs are increased (as discussed in section 5.1). Consequently, the version of the ComputePath function described in section 5.2 is a generalization of the ComputePath function used by ARA* and can be executed when changes in ϵ and edge costs occur at the same time, the scenario that Anytime D* needs to be able to handle.

The pseudocode of Anytime D* is shown in Figures 18 and 19. The code for the ComputePath function is almost the same as the one described in section 5.2. The differences, shown in bold, are that we maintain the *INCONS* list to keep track of all inconsistent states (lines 4 and 7, Figure 18), just like we did it in ARA* and we explicitly initialize the states that Anytime D* (not just the current execution of the ComputePath function) has not seen before (lines 14-15 and 22-23, Figure 18). The *INCONS* list is used to restore *OPEN*, so that it contains all inconsistent states, before each call to the ComputePath function.

The key() function that Anytime D* uses is given in Figure 19. It is straightforward to show that it satisfies the constraints on the key function (the first assumption in

```
procedure UpdateSetMembership($s$)
if ($v(s) \neq g(s)$)
  if ($s \notin$ CLOSED) insert/update $s$ in OPEN with key($s$);
  else if ($s \notin$ INCONS) insert $s$ into INCONS;
else
  if ($s \in$ OPEN) remove $s$ from OPEN;
  else if ($s \in$ INCONS) remove $s$ from INCONS;

procedure ComputePath()
while(key($s_{\mathrm{goal}}$) $> \min_{s \in OPEN}$(key($s$)) OR $v(s_{\mathrm{goal}}) < g(s_{\mathrm{goal}})$)
  remove $s$ with the smallest key($s$) from OPEN;
  if $v(s) > g(s)$
    $v(s) = g(s)$; CLOSED $=$ CLOSED $\cup \{s\}$;
    for each successor $s'$ of $s$
      if $s'$ was never visited by AD* before then
        $v(s') = g(s') = \infty$; $bp(s') = \mathbf{null}$;
      if $g(s') > g(s) + c(s, s')$
        $bp(s') = s$;
        $g(s') = g(bp(s')) + c(bp(s'), s')$; UpdateSetMembership($s'$);
  else //propagating underconsistency
    $v(s) = \infty$; UpdateSetMembership($s$);
    for each successor $s'$ of $s$
      if $s'$ was never visited by AD* before then
        $v(s') = g(s') = \infty$; $bp(s') = \mathbf{null}$;
      if $bp(s') = s$
        $bp(s') = \arg\min_{s'' \in pred(s')} v(s'') + c(s'', s')$;
        $g(s') = v(bp(s')) + c(bp(s'), s')$; UpdateSetMembership($s'$);
```

Fig. 18. Anytime D*: ComputePath function. The changes as compared with the ComputePath described in section 5.2 are shown in bold.

```
The pseudocode below assumes the following:
  (1) heuristics are consistent: $h(s) \leq c(s, s') + h(s')$ for any successor $s'$ of $s$ if $s \neq s_{\mathrm{goal}}$ and $h(s) = 0$ if $s = s_{\mathrm{goal}}$.
procedure key($s$)
if ($v(s) \geq g(s)$)
  return [$g(s) + \epsilon * h(s); g(s)$];
else
  return [$v(s) + h(s); v(s)$];

procedure Main()
$g(s_{\mathrm{goal}}) = v(s_{\mathrm{goal}}) = \infty$; $v(s_{\mathrm{start}}) = \infty$; $bp(s_{\mathrm{goal}}) = bp(s_{\mathrm{start}}) = \mathbf{null}$;
$g(s_{\mathrm{start}}) = 0$; OPEN $=$ CLOSED $=$ INCONS $= \emptyset$; $\epsilon = \epsilon_0$;
insert $s_{\mathrm{start}}$ into OPEN with key($s_{\mathrm{start}}$);
forever
  ComputePath();
  publish $\epsilon$-suboptimal solution;
  if $\epsilon = 1$
    wait for changes in edge costs;
  for all directed edges $(u, v)$ with changed edge costs
    update the edge cost $c(u, v)$;
    if ($v \neq s_{\mathrm{start}}$ AND $v$ was visited by AD* before)
      $bp(v) = \arg\min_{s'' \in pred(v)} v(s'') + c(s'', v)$;
      $g(v) = v(bp(v)) + c(bp(v), v)$; UpdateSetMembership($v$);
  if significant edge cost changes were observed
    increase $\epsilon$ or re-plan from scratch (i.e., re-execute Main function);
  else if $\epsilon > 1$
    decrease $\epsilon$;
  Move states from INCONS into OPEN;
  Update the priorities for all $s \in$ OPEN according to key($s$);
  CLOSED $= \emptyset$;
```

Fig. 19. Anytime D*: key and Main functions

Figure 15). One can also design other key functions that satisfy these constraints and are better suited for certain domains (see section 7 for some examples).

The Main() function of Anytime D* (Figure 19) first sets ϵ to a sufficiently high value ϵ_0, so that an initial, possibly highly suboptimal, solution can be generated quickly, and performs the initialization of states (lines 7 through 9) so that the assumptions of the ComputePath function (assumptions listed in Figure 15) are satisfied. It then generates and publishes an initial solution (lines 11 and 12). Afterwards, unless changes in edge costs are detected, the Main function decreases ϵ (line 23) and improves the quality of its solution by re-initializing *OPEN* and *CLOSED* properly (lines 24 through 26) and re-executing the ComputePath function.This process is identical to how the function Main() in ARA* works: before each execution of ComputePath the *OPEN* list is made to contain exactly all inconsistent states by moving *INCONS* into *OPEN* and *CLOSED* is emptied.

If changes in edge costs are detected, then Main() updates the bp- and g-values (lines 18 and 19) of immediately affected states so that the second assumption of the ComputePath function in Figure 15 is satisfied. If edge cost changes are widespread, then it may be computationally expensive to repair the current solution to regain or improve ϵ-suboptimality. In such a case (detected in line 20), one alternative for the algorithm is to increase ϵ so that a less optimal solution can be produced quickly. In some cases, however, this may be a good time to release all the currently used memory and just re-execute the Main() function with the initial value of ϵ. While we do not give a specific strategy for deciding whether the changes in edge costs are large enough to plan from scratch, in section 6 we give an example of a strategy that works well for mobile robot navigation. If the changes in edge costs are not substantial and are unlikely to cause expensive re-planning efforts, Main() can decrease ϵ (line 23), so that it both repairs and improves the solution in a single execution of the ComputePath function.

The suboptimality bound for each solution Anytime D* publishes is the same as for ARA*:

$$\epsilon' = \min(\epsilon, \frac{g(s_{\text{goal}})}{\min_{s \in OPEN \cup INCONS}(g(s) + h(s))}). \tag{5}$$

If the second term inside the $\min$ function is less than one then $g(s_{\text{goal}})$ is already equal to the cost of an optimal solution.

When interleaving planning with execution using Anytime D*, the agent executes the best plan it has so far while Anytime D* works on fixing and improving the plan. As with ARA*, it can be useful to perform the search backwards (see section 4.3). Consequently, the heuristics change as the agent moves and we can recompute the heuristic values of the states in *OPEN* during the reorder operation (line 25 in

Figure 19).[5]

5.4 Anytime D Example*

Figures 20 and 21 illustrate the approaches discussed in this article 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. As before (Figure 1), we can extract a graph from this grid by assigning a state to each cell and defining the successors and predecessors of a state to be its adjacent states. The cell marked R denotes the position of an agent navigating this environment towards the 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. All of the algorithms search backwards from the goal cell to the agent cell. The cells expanded by each algorithm for each subsequent agent position are shown in grey. The resulting paths are shown as dark grey arrows.

The first row in each figure shows the operation of backwards A*, with $\epsilon = 1$ in Figure 20 and with $\epsilon = 2.5$ in Figure 21. The initial search performed by A* with $\epsilon = 1$ provides a provably optimal path for the agent. In contrast, the initial search by inflated A* with $\epsilon = 2.5$ produces a suboptimal solution but it produces this solution much more quickly. 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 the corresponding A* search to generate a new path to the goal. While both paths happen to be the same, and optimal, they are only guaranteed to be ϵ-suboptimal by each search. In total, A* with $\epsilon = 1$ performed 31 expansions, while inflated A* performed 19 expansions.

The second row shows the operation of optimal LPA* in Figure 20 and LPA* with a constant inflation factor of $\epsilon = 2.5$ in Figure 21. The bounds on the quality of the solutions returned by these approaches are equivalent to those returned by the first two versions of A*. However, because LPA* reuses previous search results, it is able to produce its solutions with fewer overall cell expansions. LPA* without an inflation factor expands 27 cells (almost all in its initial solution generation) and always maintains an optimal solution, and LPA* with an inflation factor of 2.5 expands 13 cells but produces ϵ-suboptimal solutions.

The last row in Figure 20 shows the results of planning with ARA* and the last row in Figure 21 shows the results of planning with AD*. Each of these approaches be-

[5] The heap reorder operation might become expensive when the heap is large. An optimization based on the idea in [40] can be done to avoid heap reordering. This is discussed in [59].

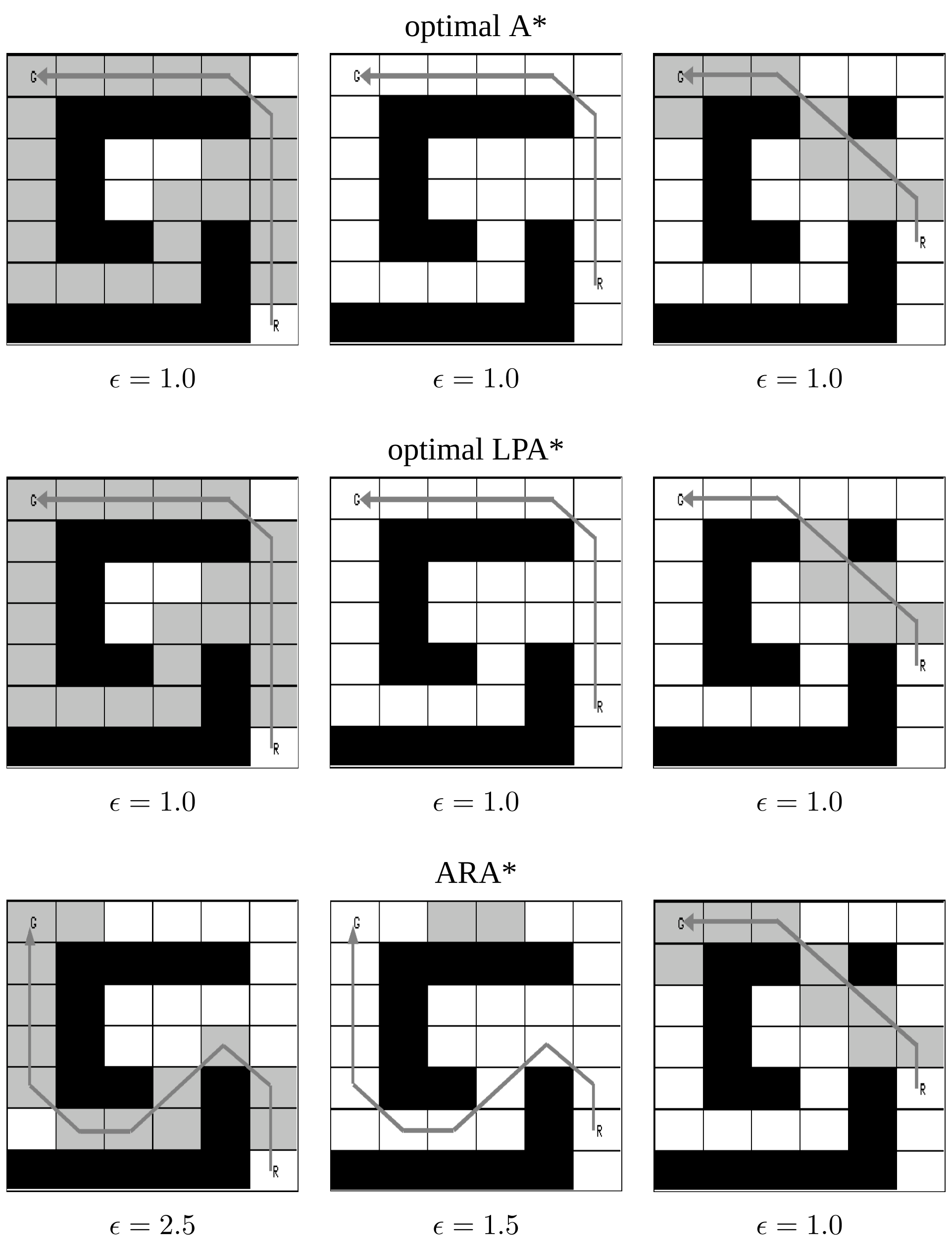


Fig. 20. An example of planning with optimal A*, optimal LPA*, and ARA*. Each algorithm directed its searches from the goal state G to the agent state R. The states expanded by the algorithms are shown in grey. Note that after the third planning episode each of the algorithms can guarantee solution optimality ($\epsilon = 1.0$).

gins 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 agent moves one more step and finds out the top wall is broken, each approach reacts differently. Because ARA* cannot incorporate edge cost

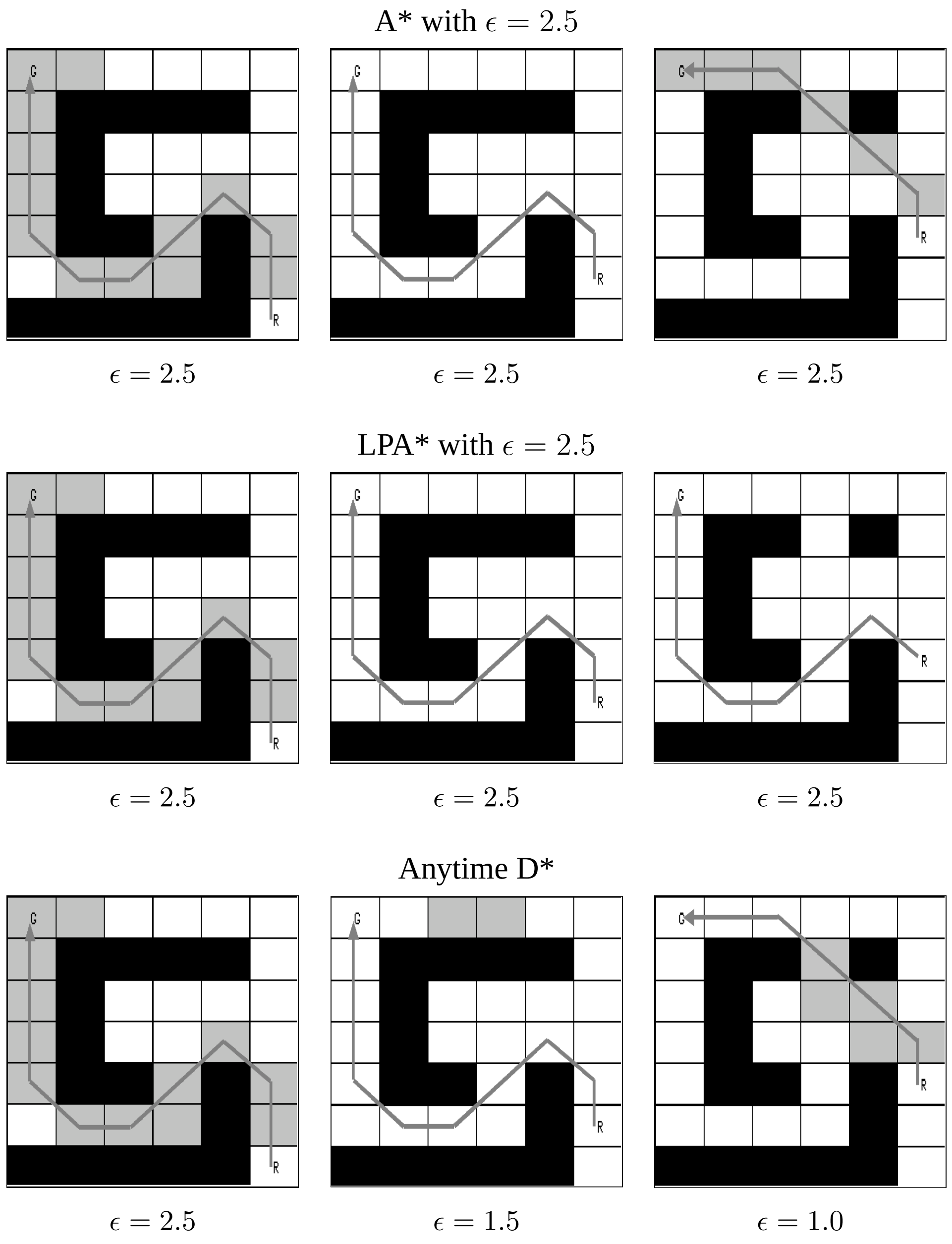


Fig. 21. An example of planning with A* with an inflation factor $\epsilon = 2.5$, LPA* with an inflation factor $\epsilon = 2.5$, and AD*. Each algorithm directed its searches from the goal state G to the agent state R. The states expanded by the algorithms are shown in grey. Note that after the third planning episode only Anytime D* can guarantee solution optimality ($\epsilon = 1.0$).

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 costs are 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 optimal LPA*. Because AD* reuses previous solutions in the same way as ARA* and repairs invalidated solutions in the same way as LPA*, it is able to efficiently provide anytime solutions in dynamic environments.

*5.5 Theoretical Properties of Anytime D**

In [60] we prove a number of properties of Anytime D*, including its termination and ϵ-suboptimality. Here we state the most important of these theorems.

Theorem 7 *When the ComputePath function exits, the following holds for any state* s *with* ($c^*(s, s_{\text{goal}}) < \infty \wedge v(s) \geq g(s) \wedge \textit{key}(s) \leq \min_{s' \in OPEN}(\textit{key}(s'))$)*:* $g^*(s) \leq g(s) \leq \epsilon * g^*(s)$*, and the cost of the path from* s_{start} *to* s *defined by back-pointers is no larger than* $g(s)$*.*

This theorem guarantees ϵ-suboptimality of the solution returned by the ComputePath function, because when it terminates $v(s_{\text{start}}) \geq g(s_{\text{start}})$ and the key value of s_{start} is at least as large as the minimum key value of all states in the *OPEN* queue. The following theorems relate to the efficiency of Anytime D*.

Theorem 8 *No state is expanded more than twice during the execution of the ComputePath function. A state can be expanded at most once as underconsistent and at most once as overconsistent.*

According to the next theorem no state is expanded needlessly. A state is expanded only if it was inconsistent before the ComputePath was invoked or if it needs to propagate the change in its v-value.

Theorem 9 *A state* s *is expanded by ComputePath only if either it is inconsistent initially or its* v*-value is altered by ComputePath at some point during its execution.*

*5.6 Experimental Analysis of the Performance of Anytime D**

To evaluate the performance of AD*, we compared it to ARA* and LPA* on a simulated 3 degree of freedom (DOF) robotic arm manipulating an end-effector through a dynamic environment (see Figures 22 and 23). 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

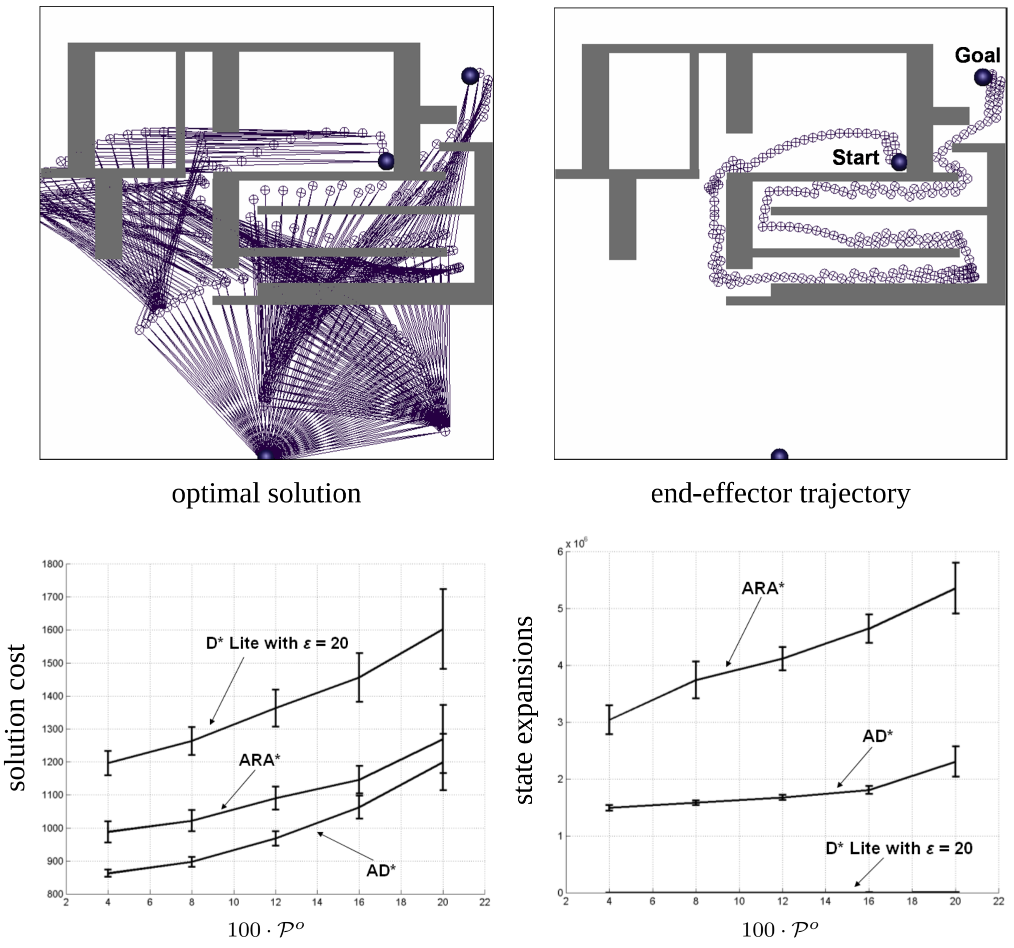


Fig. 22. Environment used in our first Anytime D* experiment, along with the optimal solution and the end-effector trajectory (without any dynamic obstacles). The links of the arm exist in an obstacle-free plane (and therefore in the shown view from the top they look as if intersecting obstacles). The end-effector projects down into a cluttered space. Also shown are the solution cost of the path traversed and the number of states expanded by each of the three algorithms compared. D* Lite is an extension of LPA* to a moving agent case.

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 its trajectory, however, at each step there was some probability $\mathcal{P}^o$ that an obstacle would appear in its current path, forcing the planner to repair its previous solution.

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 non-uniform: the link closest to the end-effector had a movement cost of 1, the middle link had a cost of

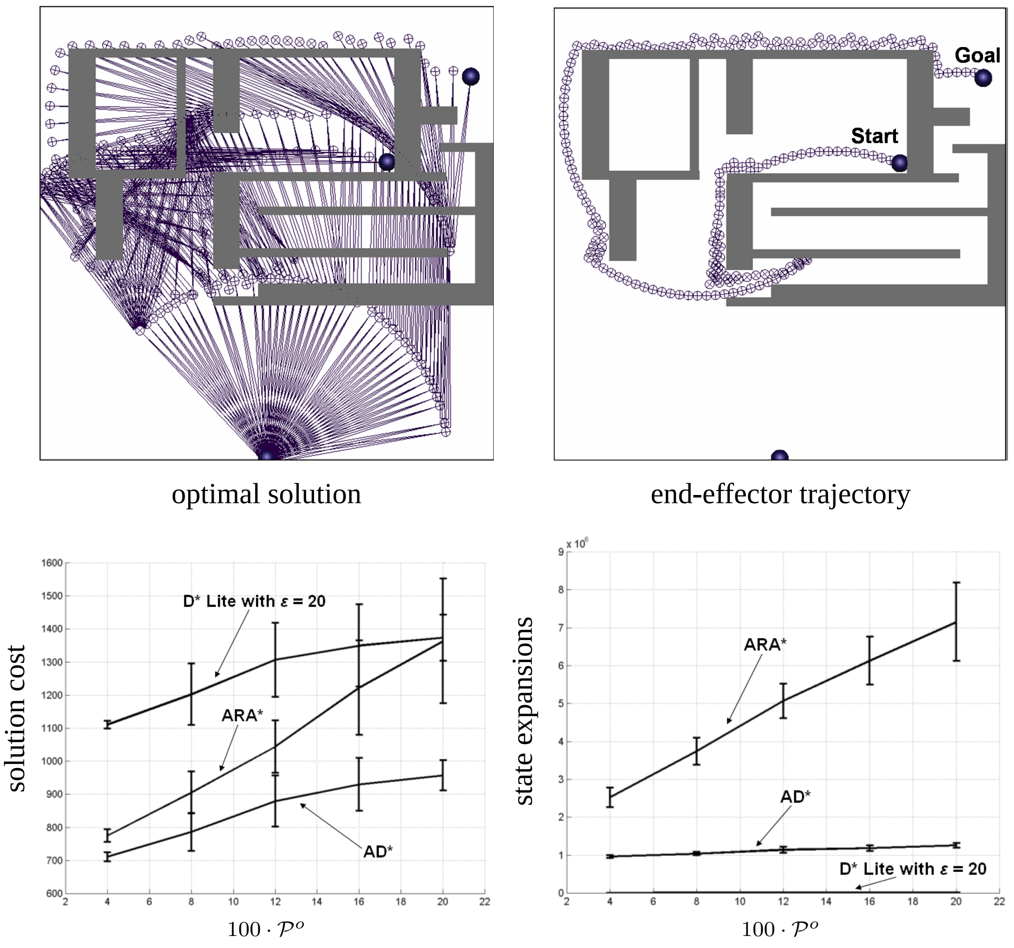
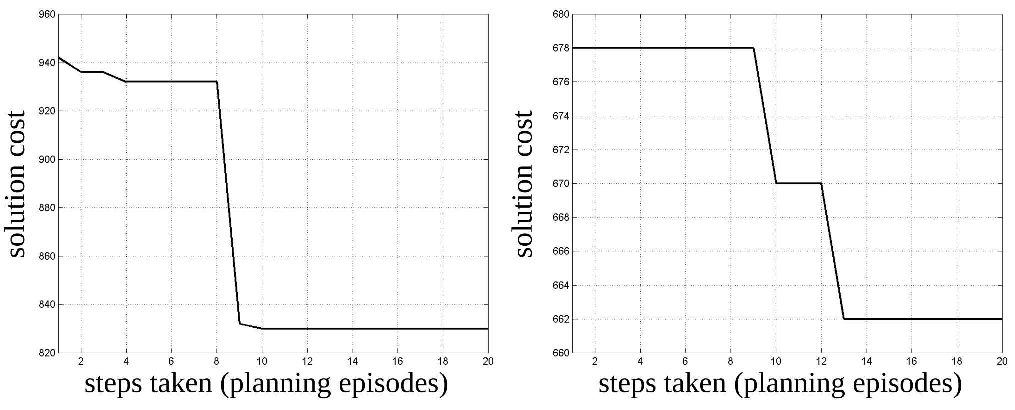


Fig. 23. Environment used in our second experiment, along with the optimal solution and the end-effector trajectory (without any dynamic obstacles). Also shown are the solution cost of the path traversed and the number of states expanded by each of the three algorithms compared.

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 LPA* with $\epsilon = 20$ to that of AD* with $\epsilon_0 = 20$, as well as the number of states expanded by each approach.[6] 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

[6] We used an extension of LPA* designed for the case where the agent is moving, known as D* Lite [57].



environment in Figure 22 environment in Figure 23

Fig. 24. 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.

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 $\times$ 50 grid, and the state space for each consisted of slightly more than 2 million states. The results of the experiments, along with 95% confidence intervals, can be found in Figures 22 and 23. As can be seen from these graphs, AD* was able to generate significantly better trajectories than ARA* while processing far fewer states. LPA* processed very few states, but its overall solution quality was much worse than that of 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 24 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 LPA* 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 roughly 10% more costly than the optimal path, and it expanded roughly the same number of states as LPA* with no inflation

(a) segbot robot (b) ATRV robot (c) 3D Map constructed by ATRV

Fig. 25. Robotic platforms that used AD* for planning

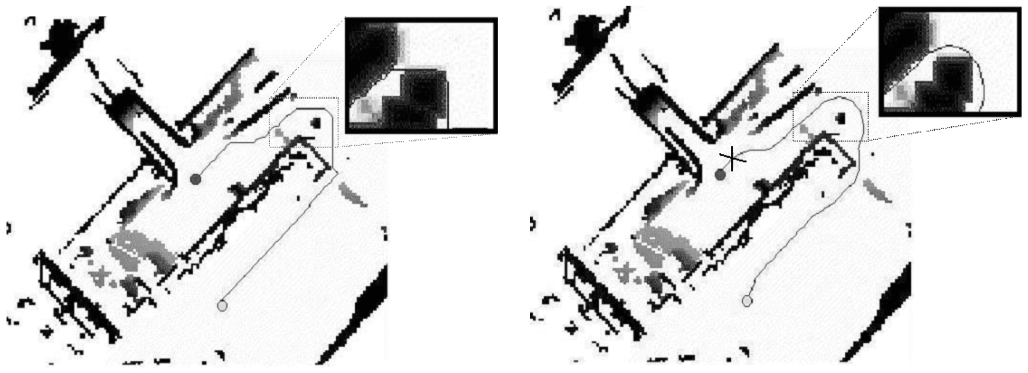
factor. This is particularly encouraging: not only is the solution generated by AD* very close to optimal, but it is providing this solution in an anytime fashion for roughly the same total amount of processing as would be required to generate the solution in one shot.

6 Application of Anytime D* to Outdoor Mobile Robot Navigation

The motivation for the planning algorithms presented in this paper was in part the development of more efficient path-planning for mobile robots, such as the ones in Figure 25. Robots often operate in open, large and poorly modelled environments. In open environments, optimal trajectories involve fast motion and sweeping turns at speed. So, it is particularly important to take advantage of the robot's momentum and find dynamic rather than static plans.

To do this we plan over a four dimensional state space, where each state is characterized by an xy-position, the orientation of the robot, and the translational velocity of the robot. The task of the planner is to generate a plan that minimizes execution time given the constraints of the robot. For example, the robot's inertial constraints prevent the planner from coming up with plans where a robot slows down faster than its maximum deceleration permits. 2D planners that only consider the xy-position of the robot are usually unable to take into account these constraints in a general and systematic way. Perhaps more importantly, constraints on the rotational velocity of the robot limit how much the robot can turn given its current translational velocity. 2D planners assume that the robot can make arbitrarily sharp turns, and therefore in practice a robot controller that executes a plan generated by such planner must drive the robot slowly and may have to stop when the robot has to turn.

As an example, Figure 26(a) shows the optimal 2D plan, and Figure 26(b) shows the optimal 4D plan through an outdoor environment. The map of the environment was constructed from 3D data gathered by an ATRV robot (see Figure 25(c) [61]). Shown in black are obstacles in the environment. The size of the environment is 91.2 by 94.4 meters discretized into cells of 0.4 by 0.4 meters. The robot's initial



(a) optimal 2D search with A*   (b) optimal 4D search with A*

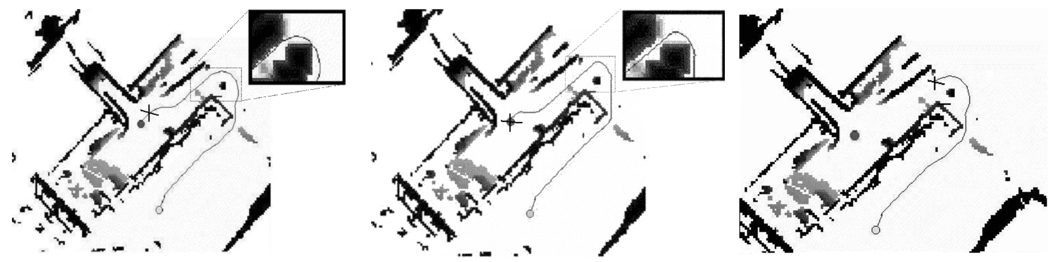
Fig. 26. The comparison of optimal 2D plan with optimal 4D plan.

state is the dark circle to the left, while its goal is the light circle to the right. To ensure the safe operation of the robot we created a buffer zone around each obstacle with high costs. The squares in the upper-right corners of the figures show a magnified fragment of the map with grayscale proportional to cost. As the fragments show, the optimal 2D plan makes a 90 degree turn when going around the obstacles, requiring the robot to come to a complete stop. The optimal 4D plan, on the other hand, results in a wider turn, and the velocity of the robot remains high throughout the whole trajectory.

Unfortunately, higher dimensionality combined with large environments results in very large state spaces for the 4D planner. Moreover, in poorly modelled environments, the planning problem changes often as we discover new obstacles or as modelling errors push us off of our planned trajectory. As a result, the robot needs to re-plan its trajectory many times on its way to the goal, and it needs to do this quickly while moving. Anytime D* is very well-suited for performing this planning task.

We built a two-level planner for this navigation problem: we combined a 4D planner that uses Anytime D* with a 2D (x and y) planner that performs A* search and whose results are used to initialize the heuristics for the 4D planner. (This approach of using a lower-dimensional search to derive heuristics for a higher-dimensional search is closely related to the approach of using pattern databases [62].) The 4D planner searches backward from the goal state to the robot state, while the 2D planner searches forward. This way the 4D planner does not have to discard the search tree every time the robot moves. The 2D planner, on the other hand, is very fast and can be re-run every time the robot moves without causing any delay.

The 4D planner continuously runs Anytime D* until the robot reaches its goal. Initially, Anytime D* sets ϵ to a high value (to be specific, 2.5) and comes up with a plan very quickly. While the robot executes this plan, the plan is improved and repaired if new information about the environment is gathered. Every 500 milliseconds, the robot updates its plan to the most recent solution. Thus, at any point of



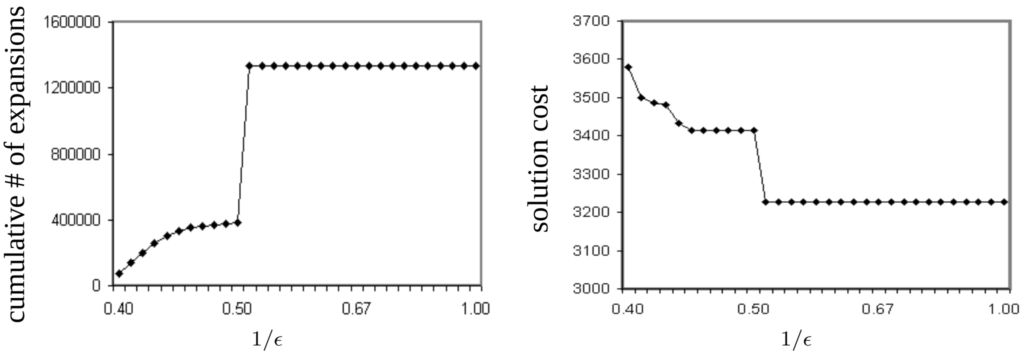
(a) optimal 4D search with A* after 25 secs (b) 4D search with AD* after 0.6 secs ($\epsilon = 2.5$) (c) 4D search with AD* after 25 secs ($\epsilon = 1.0$)

Fig. 27. The comparison of planning with A* and Anytime D* for outdoor robot navigation (cross shows the position of the robot). Both 4D searches used 2D search to compute heuristics when necessary as described in the text. In this example no information inconsistent with the initial map was observed during execution (i.e. no edge cost changes occurred).

time, the robot has access to a 4D plan and does not have to stop. In between each call to ComputePath, the goal state of the search, s_{goal}, is set to the current robot state, so that the plan corresponds correctly to the position of the robot.

In most of our experiments, initially the robot only knows what it can observe from its starting location. As the robot moves it senses obstacles and adds them to the map. When no new information about the environment is observed, Anytime D* decreases ϵ in between calls to ComputePath to provide improved solutions. When new information about the environment is gathered, Anytime D* has to re-plan. As discussed in section 5.3, before calling the ComputePath function, however, it has to decide whether to continue improving the solution (i.e., to decrease ϵ), whether to quickly re-compute a new solution with a looser suboptimality bound (i.e., to increase ϵ), or whether to plan from scratch by discarding all search efforts so far and resetting ϵ to its initial, large value. We chose to make this decision based on the solution computed by the 2D planner. If the cost of the 2D path remained the same or changed little after the 2D planner finished its execution, then the 4D planner decreased ϵ before the new call to ComputePath. In cases when the cost of the 2D path changed substantially, on the other hand, the 4D planner always re-planned from scratch by clearing all the memory and resetting ϵ. In our implementation we chose to never increase ϵ without discarding the current search tree. Because the robot was moving through the environment, a large number of previously computed states quickly became irrelevant. By clearing the memory, we were able to ignore these irrelevant states and make room for those that were relevant.

Using our approach we were able to build a robotic system that can plan and re-plan in outdoor environments while navigating at relatively high speed. The system was deployed on two real robotic platforms: the Segway Robotic Mobility Platform shown in Figure 25(a) and the ATRV vehicle shown in Figure 25(b). Both used laser range finders (one on the Segway and two on the ATRV) for mapping and inertial measurement units combined with global positioning systems for position estimation.



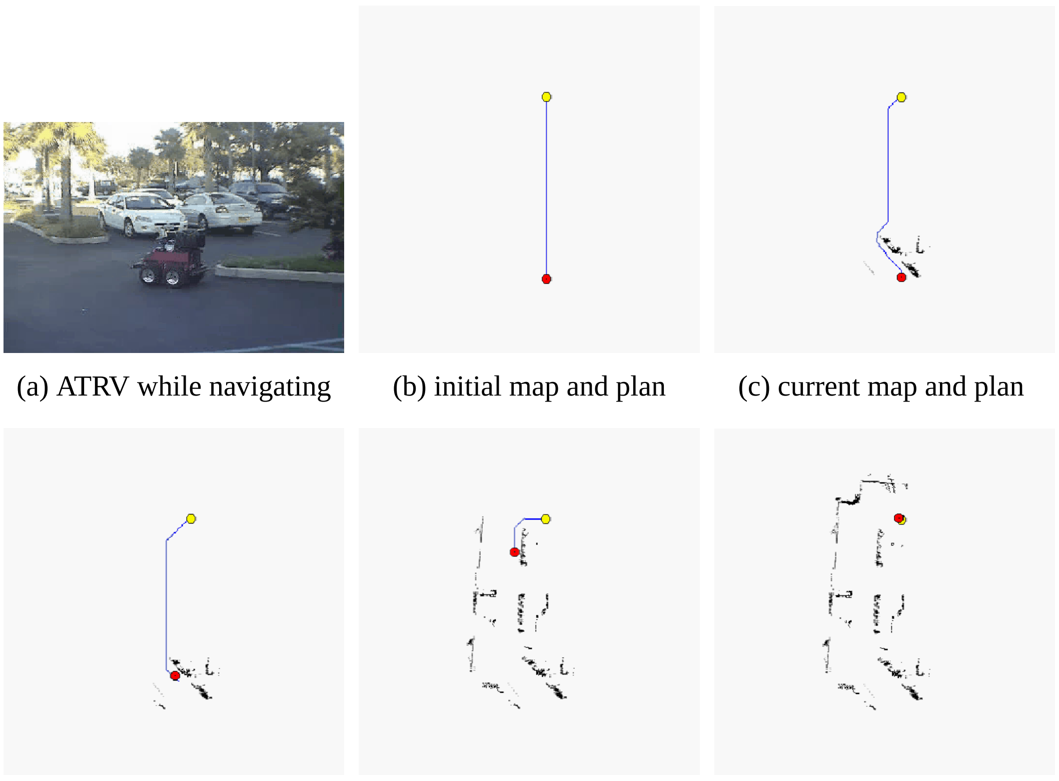
(a) cumulative number of expansions (b) solution cost

Fig. 28. The performance of AD* planner in the same environment as in Figure 27 but for a different configuration of goal and start locations (a harder scenario) and for a fixed robot position (i.e., the robot does not move).

As mentioned before, the size of the environment in the example in Figure 26 is 91.2 by 94.4 meters and the map is discretized into cells of 0.4 by 0.4 meters. Thus, the 2D state space consists of 53,808 states and the 4D state space has over 20 million states. As a result, the 4D state space is too large for efficient planning and re-planning optimally. In Figure 27 we show the advantage of the anytime capability of AD* in this environment. For the sake of easier analysis, this figure shows execution in simulation on the map that is fully-known in advance. We will show execution on a real-robot with a map that is initially completely unknown in Figure 29.

Figure 27(b) shows the initial plan computed by the 4D planner running Anytime D* starting at $\epsilon = 2.5$. In this suboptimal plan, the trajectory is much smoother and therefore can be traversed much faster than the 2D plan (Figure 26(a)). It is, however, somewhat less smooth than the optimal 4D plan (Figure 27(a)). The time required for the optimal 4D planner was 11.196s, whereas the time required for the 4D planner that runs Anytime D* to generate its initial plan was 556 ms. (The planning for all experiments was done on a 1 GHz Pentium processor.) As a result, the robot that runs Anytime D* can start executing its plan much earlier. The cross in Figure 27(a) (close to the initial robot location) shows the location of the robot after 25 seconds from the time it receives a goal location. In contrast, Figure 27(c) shows the position of the robot running Anytime D* after the same amount of time. The robot using Anytime D* has advanced much further, and its plan by now has converged to optimal and thus is no different from the one in Figure 27(a).

In Figure 28(a) and Figure 28(b) we show the cumulative number of states expanded and the cost of the path found so far, as a function of $1/\epsilon$. This experiment was done in the same environment as before but for a different configuration of start and goal states, so that the optimal path is longer and harder to find. We also kept the start state fixed to more easily analyze the performance of the algorithm. Initially, the number of states expanded is small (about 76 thousand). The resulting path is about 10% suboptimal. For each subsequent call to ComputePath the



(a) ATRV while navigating (b) initial map and plan (c) current map and plan

(d) current map and plan (e) current map and plan (f) ATRV at its goal

Fig. 29. A run by an ATRV robot in an initially unknown environment. Figure (a) shows the ATRV navigating in the environment, which was a parking lot full of cars. Figure (b) shows the initial map and the initial plan AD* constructs. Figures (c-e) show the updated map and the plan generated by AD* at different times. Figure (f) shows the map constructed by the robot by the time it reaches its goal.

number of states expanded continues to be small (sometimes less than ten thousand) until one particular invocation of ComputePath. During that iteration, over 952 thousand states are expanded. At exactly this iteration the solution drastically changes and becomes optimal. There are no states expanded during the rest of the iterations despite ϵ decreasing. The overall number of states expanded over all iterations is about 1.3 million. To compare, the number of states expanded by the optimal planner would have been over 953 thousand. Thus, over all iterations about 30 percent more states are expanded by Anytime D* but a solution that is roughly 10% suboptimal was obtained for only 8% of the state expansions performed by the optimal approach. It is important to remember though that the number of expansions Anytime D* performs before it converges to a provably optimal solution (that is, $\epsilon = 1$) is at least the number of expansions performed by an optimal A* search.

In the example we have just seen the environment was consistent with the initial map and thus during execution there were no edge cost changes. In contrast, in the example in Figure 29 the ATRV robot navigates to its goal location in an entirely unknown environment (Figure 29(b)). In this experiment the robot navigates a parking lot full of cars (Figure 29(a)). The robot assumes that all unknown area is traversable (the 'freespace' assumption). Under this assumption the robot performs 4D planning using Anytime D*. While executing the plan it uses its two laser range finders to gather new information about the environment, updates its map accordingly and repairs and improves its plan. Figures 29(b) through (f) show how the robot progresses towards its goal while building the map. This process involves a substantial amount of re-planning as the map updates are often substantial and the plan needs to be re-computed after every map update. Nevertheless, the Anytime D* based planner was able to provide the robot with safe 4D plans at any point in time and allowed the robot to navigate unknown and partially known environments at speeds up to 1.5 meters/sec.

Most recently, we have used Anytime D* to build a 4D (x, y, orientation and translational velocity) planner for performing complex maneuvers with a full-size autonomous SUV. Implemented for Carnegie Mellon University's robotic entry into the 2007 DARPA Urban Challenge, this planner is used to plan and re-plan dynamically feasible motion trajectories for the vehicle operating in parking lots, in off-road scenarios and in on-road situations requiring non-trivial avoidance of obstacles or execution of U-turns. These trajectories involve complex maneuvers such as backing up, parking and moving through dense fields of irregular obstacles. This planner has been tested in environments of sizes up to 500 meters by 500 meters and with speeds up to 5 meters/second.

7 Discussion and Extensions

The anytime behavior of ARA* and AD* strongly relies on the properties of the heuristics used. In particular, it relies on the assumption that a sufficiently large inflation factor ϵ substantially expedites the planning process. While in many domains this assumption is true, 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. In general, the key to obtaining anytime behavior in ARA* is finding heuristics for which the difference between the heuristic values and the true distances these heuristics estimate is a function with only shallow local minima. Note that this is not the same as just keeping small the magnitude of the differences between the heuristic values and the true distances. Instead, the difference will have shallow local minima if the heuristic function has a shape similar to the shape of the true distance function. 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). The size of the obstacle determines how many states weighted A*, and consequently ARA* and AD*, will have to visit before getting out of the minimum. The conclusion is that with ARA* (and AD*), the task of developing an anytime (re-)planner for various hard planning domains becomes the problem of designing a heuristic function that contains shallow local minima. In many cases (although certainly not always) the design of such a heuristic function can be a much simpler task than the task of designing from scratch a whole new anytime (re-)planning algorithm for solving the problem at hand. [7]

The memory requirements of the presented algorithms are also strongly related to the heuristic function used. If the heuristic function guides searches well and the branching factor is not very large, then ARA* and AD* can handle very large graphs. For example, in our experiments on the motion planning for a 20 DOF robot arm, the state-spaces contained up to 10^{26} states. The branching factor, however, was limited to 40 (only one joint angle was changed at a time). Thus, ARA* could decrease ϵ from 10 to less than 4 without running out of memory. Trying to compute a solution for a much smaller ϵ, however, would result in ARA* running out of memory. One way to prevent this is not allow for the planner to decrease ϵ to values smaller than some value (e.g., 1.5). A better way perhaps, would be to query the amount of remaining free memory and use this information to decide whether the planner can be allowed to decrease ϵ or not in real-time. There also exist many search problems that have a very high branching factor. Retaining all the generated states for such problems becomes infeasible. To address this, a number of heuristic searches have been developed that control the amount of memory they consume at the expense of computational efficiency [64–66]. It would be valuable to investigate whether the ideas behind these searches can be incorporated into ARA* and AD*.

Incremental searches in general, and AD* in particular, are very effective for replanning in the context of mobile robot navigation. Typically, in such scenarios the changes to the graph are occurring close to the robot (through the robot's observations). Their effects are therefore usually limited and much of the previous search efforts can be reused if the search is performed backwards from the goal state towards the state of the robot. Using an incremental replanner such as AD* in such cases 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 goal of the search (the state of the robot in the robot navigation problem), it is possible for AD* to be even *less* efficient than weighted A* with the heuristics inflated by the same constant. Mainly, this is because it is possible for AD* 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 AD*, and one that illustrates this pos-

[7] It would also be interesting to extend ARA* and AD* to be able to search for partial paths (in the same way that agent-centered searches only search few steps ahead). This would guarantee that the algorithms can provide a plan at any point in time in *any* domain, no matter how hard it is to find a complete plan for it. This property, though, would come at the expense of not being able to provide bounds, other than polynomial in the total number of states [63], on the suboptimality of the solution found.

sibility, is when changes are being made to the graph in the vicinity of the start of the search. Similarly, AD* can also be less efficient than weighted A* if there are a lot of edge cost changes. It is thus advisable for systems using AD* to abort the replanning process and plan from scratch whenever either major edge cost changes are detected or some predefined threshold of replanning effort is reached. The discussion in section 6 gives one method for deciding when to plan from scratch, at least in the domain of robot navigation using 4D planning.

In general, it is important to note that planning from scratch every so often has the benefit of freeing up memory from the states that are no longer relevant. This is especially so in cases when the agent moves and the regions where it was before are no longer relevant to its current plan. If, however, replanning from scratch needs to be minimized as much as possible then one might consider limiting the expense of re-ordering *OPEN* as well as inserting and deleting states from it by splitting *OPEN* into a priority queue and one or more unordered lists containing only the states with large priorities. The states from these lists need only be considered if the priority of the goal state becomes sufficiently large. Therefore, we only need to maintain the minimum priority among states on the unordered lists (or even some lower bound on it) which can be much cheaper than leaving them on the priority queue. 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 [40]. Its implementation for ARA* and AD* is discussed in [59].

There also exist a few other optimizations to the algorithms presented here. For example, Delayed D* [67] tries to postpone the expansion of underconsistent states in LPA*. This seems to be quite beneficial in the domains where edge cost changes can occur in arbitrary locations rather than close to the agent. This optimization is directly applicable to AD*. As another example, in domains with a very high branching factor, ARA* and AD* can be sped up by pruning states from *OPEN* that are guaranteed not to be useful for improving the current plan [17]. These and other optimizations are described more thoroughly in [68].

A series of other optimizations concern the key function in AD*. The key function we give in this paper is a two-valued function presented in Figure 19. A number of other key functions, however, can also be designed that satisfy the restrictions on the state priorities (the restrictions in Figure 5 and Figure 16). These functions are suited better for certain domains. For example, it is usually desirable to decrease the expense of maintaining *OPEN* as much as possible. While in general *OPEN* can be implemented as a heap, it can be quite expensive to maintain it as such. In cases when the number of distinct priorities is small, *OPEN* can instead be implemented using buckets. To this end, one can achieve a significant decrease in the number of distinct priorities by setting $key(s) = [g(s) + \epsilon * h(s); 1]$ if s is not underconsistent and $key(s) = [v(s) + h(s); 0]$ otherwise. In some domains this key function can decrease the number of distinct priorities to a number small enough for *OPEN* to be implemented using buckets. [56] presents a number of other valid key functions including one that breaks ties among the candidates for expansions with the same f-values towards the states with the larger g-values. This tie-breaking criterion has been known to be important in domains where many optimal solutions exist and we want to avoid exploring all of them.

8 Conclusions

Planners used by agents operating in the real world must be able to provide plans within limited deliberation time. In addition, world models used for planning are often imperfect and so these models and the plans generated using these models need to be updated as agents receive new information about the world. The combination of these requirements makes planning for real-world tasks a challenging area of research.

In this paper we contribute to this research in three ways. First, we present a novel formulation of the well-known and widely-used A* search algorithm as a search that expands inconsistent states. This formulation provides the basis for incremental execution of an A* search: the search can be executed with an arbitrary initialization of states as long as all inconsistent states in the graph are identified beforehand. The search will then concentrate on correcting only the inconsistent states and will ignore the consistent states whose values are already correct.

Next, we use our formulation of A* search to construct an anytime heuristic search, ARA*, that provides provable bounds on the suboptimality of any solution it produces. As an anytime algorithm it finds a feasible solution quickly and then continually works on improving this solution until the time available for planning runs out. While improving the solution, ARA* reuses previous search efforts and, as a result, is significantly more efficient than other anytime search methods. ARA* is an algorithm well-suited for operation under time constraints. We demonstrate this through experiments on a simulated high-dimensional robot arm and a complex path planning problem for an outdoor mobile robot.

Based on our formulation of A* search, we also develop Anytime D*, an algorithm that is both anytime and incremental. Anytime D* produces solutions of bounded suboptimality in an anytime fashion. It improves the quality of its solution until the available search time expires, at every step reusing previous search efforts. When updated information regarding the underlying graph is received, the algorithm can simultaneously improve and repair its previous solution. It thus combines the benefits of anytime and incremental planners and provides efficient solutions to complex, dynamic planning problems under time constraints. We demonstrate its effectiveness on a simulated robot arm and the problem of complex path planning for robots navigating in partially-known outdoor environments. To the best of our knowledge, Anytime D* is the only heuristic search algorithm that is both anytime and incremental.

All the algorithms presented here are simple to implement and extend, are theoretically well-founded and are very useful in practice. As such, we hope they will contribute to and motivate other researchers developing search algorithms for real world applications.

9 Acknowledgements

The authors would like to thank Sven Koenig for highly fruitful discussions and the anonymous reviewers for their valuable comments on the manuscript. This research was sponsored in part by the Defense Advanced Research Projects Agency (DARPA), in particular its MICA and MARS programs, and the National Science Foundation (NSF) under a Graduate Research Fellowship for Dave Ferguson. The views and conclusions expressed in this publication are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, or the U.S. government.

References

[1] P. E. Hart, N. J. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE Transactions on Systems, Science, and Cybernetics SSC-4 (2) (1968) 100–107.

[2] E. W. Dijkstra, A note on two problems in connexion with graphs, Numerische Mathematik 1 (1959) 269–271.

[3] R. Dechter, J. Pearl, Generalized best-first search strategies and the optimality af A*, Journal of the Association for Computing Machinery 32 (3) (1985) 505–536.

[4] S. Zilberstein, S. Russell, Approximate reasoning using anytime algorithms, in: Imprecise and Approximate Computation, Kluwer Academic Publishers, 1995.

[5] T. L. Dean, M. Boddy, An analysis of time-dependent planning, in: Proceedings of the National Conference on Artificial Intelligence (AAAI), 1988.

[6] H. Moravec, Certainty grids for mobile robots, in: Proceedings of the NASA/JPL Space Telerobotics Workshop, 1987.

[7] S. Koenig, Y. Smirnov, Sensor-based planning with the freespace assumption, in: Proceedings of the International Conference on Robotics and Automation (ICRA), 1997.

[8] J. Ambite, C. Knoblock, Planning by rewriting: Efficiently generating high-quality plans, in: Proceedings of the National Conference on Artificial Intelligence (AAAI), 1997.

[9] S. Zilberstein, S. Russell, Anytime sensing, planning and action: A practical model for robot control, in: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), 1993, pp. 1402–1407.

[10] N. Hawes, An anytime planning agent for computer game worlds, in: Proceedings of the Workshop on Agents in Computer Games at The 3rd International Conference on Computers and Games (CG'02), 2002.

[11] D. K. Pai, L.-M. Reissell, Multiresolution rough terrain motion planning, IEEE Transactions on Robotics and Automation 14 (1) (1998) 19–33.

[12] H. Prendinger, M. Ishizuka, APS, a prolog-based anytime planning system, in: Proceedings 11th International Conference on Applications of Prolog (INAP), 1998.

[13] R. E. Korf, Real-time heuristic search, Artificial Intelligence 42 (2-3) (1990) 189–211.

[14] P. Dasgupta, P. Chakrabarti, S. DeSarkar, Agent searching in a tree and the optimality of iterative deepening, Artificial Intelligence 71 (1994) 195–208.

[15] S. Koenig, Chapter 2 of goal-directed acting with incomplete information: Acting with agent-centered search, Ph.D. thesis, Carnegie Mellon University (1997).

[16] R. Zhou, E. A. Hansen, Multiple sequence alignment using A*, in: Proceedings of the National Conference on Artificial Intelligence (AAAI), 2002, Student abstract.

[17] E. A. Hansen, R. Zhou, Anytime heuristic search., Journal of Artificial Intelligence Research (JAIR) 28 (2007) 267–297.

[18] J. G. Gaschnig, Performance measurement and analysis of certain search algorithms, Ph.D. thesis, Carnegie Mellon University (1979).

[19] B. Bonet, H. Geffner, Planning as heuristic search, Artificial Intelligence 129 (1-2) (2001) 5–33.

[20] R. E. Korf, Linear-space best-first search, Artificial Intelligence 62 (1993) 41–78.

[21] S. Rabin, A* speed optimizations, in: M. DeLoura (Ed.), Game Programming Gems, Charles River Media, Rockland, MA, 2000, pp. 272–287.

[22] S. Edelkamp, Planning with pattern databases, in: Proceedings of the European Conference on Planning (ECP), 2001.

[23] A. Bagchi, P. K. Srimani, Weighted heuristic search in networks, Journal of Algorithms 6 (1985) 550–576.

[24] P. P. Chakrabarti, S. Ghosh, S. C. DeSarkar, Admissibility of AO* when heuristics overestimate, Artificial Intelligence 34 (1988) 97–113.

[25] H. W. Davis, A. Bramanti-Gregor, J. Wang, The advantages of using depth and breadth components in heuristic search, in: Z. W. Ras, L. Saitta (Eds.), Methodologies for Intelligent Systems, 3, North-Holland, New York, 1988, pp. 19–28.

[26] R. Zhou, E. A. Hansen, Beam-stack search: Integrating backtracking with beam search, in: Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS), 2005, pp. 90–98.

[27] D. Furcy, Chapter 5 of speeding up the convergence of online heuristic search and scaling up offline heuristic search, Ph.D. thesis, Georgia Institute of Technology (2004).

[28] W. Zhang, Complete anytime beam search, in: Proceedings of the National Conference on Artificial Intelligence (AAAI), 1998, pp. 425–430.

[29] V. Kumar, Branch-and-bound search, in: S. C. Shapiro (Ed.), Encyclopedia of Artificial Intelligence, New York, NY: Wiley-Interscience, 1992, pp. 1468–1472.

[30] R. Simmons, A theory of debugging plans and interpretations, in: Proceedings of the National Conference on Artificial Intelligence (AAAI), 1988, pp. 94–99.

[31] A. Gerevini, I. Serina, Fast plan adaptation through planning graphs: Local and systematic search techniques, in: Proceedings of the International Conference on Artificial Intelligence Planning and Scheduling (AIPS), 2000, pp. 112–121.

[32] J. Koehler, Flexible plan reuse in a formal framework, in: C. Bäckström, E. Sandewall (Eds.), Current Trends in AI Planning, IOS Press, 1994, pp. 171–184.

[33] M. Veloso, Planning and Learning by Analogical Reasoning, Springer, 1994.

[34] R. Alterman, Adaptive planning, Cognitive Science 12 (3) (1988) 393–421.

[35] S. Kambhampati, J. Hendler, A validation-structure-based theory of plan modification and reuse, Artificial Intelligence 55 (1992) 193–258.

[36] S. Hanks, D. Weld, A domain-independent algorithm for plan adaptation, Journal of Artificial Intelligence Research 2 (1995) 319–360.

[37] S. Edelkamp, Updating shortest paths, in: Proceedings of the European Conference on Artificial Intelligence, 1998.

[38] K. Trovato, Differential A*: An adaptive search method illustrated with robot path planning for moving obstacles and goals, and an uncertain environment, Journal of Pattern Recognition and Artificial Intelligence 4 (2).

[39] L. Podsedkowski, J. Nowakowski, M. Idzikowski, I. Vizvary, A new solution for path planning in partially known or unknown environment for nonholonomic mobile robots, Robotics and Autonomous Systems 34 (2001) 145–152.

[40] A. Stentz, The focussed D* algorithm for real-time replanning, in: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), 1995.

[41] S. Koenig, M. Likhachev, Incremental A*, in: T. G. Dietterich, S. Becker, Z. Ghahramani (Eds.), Advances in Neural Information Processing Systems (NIPS) 14, Cambridge, MA: MIT Press, 2002.

[42] G. Ausiello, G. Italiano, A. Marchetti-Spaccamela, U. Nanni, Incremental algorithms for minimal length paths, Journal of Algorithms 12 (4) (1991) 615–638.

[43] S. Even, H. Gazit, Updating distances in dynamic graphs, Methods of Operations Research 49 (1985) 371–387.

[44] D. Frigioni, A. Marchetti-Spaccamela, U. Nanni, Fully dynamic output bounded single source shortest path problem, in: Proceedings of the Symposium on Discrete Algorithms, 1996.

[45] S. Goto, A. Sangiovanni-Vincentelli, A new shortest path updating algorithm, Networks 8 (4) (1978) 341–372.

[46] C. Lin, R. Chang, On the dynamic shortest path problem, Journal of Information Processing 13 (4) (1990) 470–476.

[47] G. Ramalingam, T. Reps, An incremental algorithm for a generalization of the shortest-path problem, Journal of Algorithms 21 (1996) 267–305.

[48] H. Rohnert, A dynamization of the all pairs least cost path problem, in: Proceedings of the Symposium on Theoretical Aspects of Computer Science, 1985, pp. 279–286.

[49] R. Ahuja, S. Pallottino, M. G. Scutell, Dynamic shortest paths minimizing travel times and costs, Networks 41 (2003) 197–205.

[50] D. Frigioni, A. Marchetti-Spaccamela, U. Nanni, Fully dynamic shortest paths in digraphs with arbitrary arc weights, Journal of Algorithms 49 (1) (2003) 86–113.

[51] S. Chien, R. Knight, A. Stechert, R. Sherwood, G. Rabideau, Using iterative repair to improve the responsiveness of planning and scheduling, in: Proceedings of the International Conference on Artificial Intelligence Planning and Scheduling (AIPS), 2000.

[52] G. J. Ferrer, Anytime replanning using local subplan replacement, Ph.D. thesis, University of Virginia (2002).

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

[54] I. Pohl, First results on the effect of error in heuristic search, Machine Intelligence 5 (1970) 219–236.

[55] M. Likhachev, Search-based planning for large dynamic environments, Ph.D. thesis, Carnegie Mellon University (2005).

[56] M. Likhachev, S. Koenig, A generalized framework for Lifelong Planning A*, in: Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS), 2005, pp. 99–108.

[57] S. Koenig, M. Likhachev, D* Lite, in: Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI), 2002.

[58] M. Likhachev, G. Gordon, S. Thrun, ARA*: Formal analysis, Tech. Rep. CMU-CS-03-148, Carnegie Mellon University, Pittsburgh, PA (2003).

[59] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, S. Thrun, Anytime Dynamic A*: An anytime, replanning algorithm, in: Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS), 2005, pp. 262–271.

[60] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, S. Thrun, Anytime Dynamic A*: The proofs, Tech. Rep. CMU-RI-TR-05-12, Carnegie Mellon University, Pittsburgh, PA (2005).

[61] D. Haehnel, Personal communication (2003).

[62] J. C. Culberson, J. Schaeffer, Pattern databases, Computational Intelligence 14 (3) (1998) 318–334.

[63] S. Koenig, R. Simmons, Easy and hard testbeds for real-time search algorithms, in: Proceedings of the National Conference on Artificial Intelligence (AAAI), 1996.

[64] R. E. Korf, Depth-first iterative-deepening: an optimal admissible tree search, Artificial Intelligence 27 (1) (1985) 97–109.

[65] R. E. Korf, W. Zhang, Divide-and-conquer frontier search applied to optimal sequence alignment, in: Proceedings of the National Conference on Artificial Intelligence (AAAI), 2000, pp. 910–916.

[66] R. Zhou, E. A. Hansen, Sweep A*: Space-efficient heuristic search in partially ordered graphs, in: Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence (ICTAI), 2003, p. 427.

[67] D. Ferguson, A. Stentz, The delayed D* algorithm for efficient path replanning, in: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), 2005.

[68] D. Ferguson, M. Likhachev, A. Stentz, A guide to heuristic-based path planning, in: Proceedings of the Workshop on Planning under Uncertainty for Autonomous Systems at The International Conference on Automated Planning and Scheduling (ICAPS), 2005.