Search-based Planning for Large Dynamic Environments

Maxim Likhachev

September 2005

CMU-CS-05-182

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

This research was sponsored in part by the Defense Advanced Research Projects Agency (DARPA), in particular, its MICA and MARS programs. The views and conclusions expressed in this publication are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, or the U.S. government.

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

Abstract

Agents operating in the real world often have to act under the conditions where time is critical: there is a limit on the time they can afford to spend on deliberating what action to execute next. Planners used by such agents must produce the best plans they can find within the amount of time available. The strategy of always searching for an optimal plan becomes infeasible in these scenarios. Instead, we must use an anytime planner. Anytime planners operate by quickly finding a highly suboptimal plan first, and then improving it until the available time runs out.

In addition to the constraints on time, world models used by planners are usually imperfect and environments are often dynamic. The execution of a plan therefore often results in unexpected outcomes. An agent then needs to update the model accordingly and re-execute a planner on the new model. A planner that has a replanning capability (a.k.a. an incremental planner) can substantially speed up each planning episode in such cases, as it tries to make use of the results of previous planning efforts in finding a new plan.

Combining anytime with replanning capabilities is thus beneficial. For one, at each planning episode it allows the planner to produce a better plan within the available time: both in finding the first plan as well as in improving it, the planner can use its replanning capability to accelerate the process. In addition, the combination allows one to interleave planning and execution effectively. While the agent executes the current plan, the planner can continue improving it without having to discard all of its efforts every time the model of the world is updated.

This thesis concentrates on graph-based searches. It presents an alternative view of A* search, a widely popular heuristic search in AI, and then uses this view to easily derive three versions of A* search: an anytime version, an incremental version and a version that is both anytime and incremental. Each of these algorithms is also able to provide a non-trivial bound on the suboptimality of the solution it generates. We believe that the simplicity, the existence of suboptimality bounds and the generality of the presented methods contribute to the research and development of planners well suited for systems operating in the real world.

Acknowledgements

First of all, I would like to thank my advisors, Geoff Gordon and Sebastian Thrun, for letting me pursue the research I was most interested in and providing guidance whenever needed. I would also like to thank the other members of my thesis committee, Manuel Blum and Sven Koenig, for being very supportive throughout my Ph.D. studies. In particular, I am very grateful to Sven Koenig for introducing me to search-based planning and then being always ready to help with my research in this area. In addition, I would like to thank the other academic mentors that I worked with during my graduate studies, including Ronald C. Arkin, Chris Atkeson and Zoubin Ghahramani.

I would also like to acknowledge a number of colleagues I had worked with throughout my graduate studies, both at CMU and Georgia Tech. This list includes Rahul Biswas, Tom Collins, Yoichiro Endo, David Furcy, Dirk Haehnel, Michael Kaess, Zsolt Kira, Yaxin Liu, H. Brendan McMahan, Mike Montemerlo, Nick Roy, Rudy Triebel and Daniel Wilson. I would like to thank separately my colleague at CMU, Dave Ferguson, as our fruitful collaborations had direct influence on the work presented in this thesis.

Finally, and most importantly, I would like to thank my family. This work would not have been done without the love and invaluable help of my parents in every aspect of my life. I am grateful to my wife, Alla, for her moral support and devotion to me during my graduate studies and to my beautiful daughter, Alexandra, for making sure I did not work long hours.

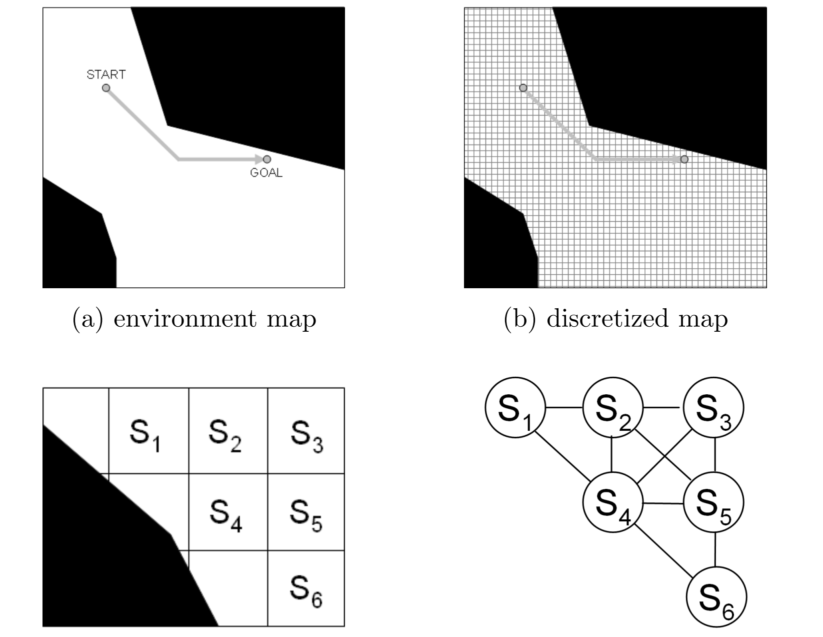
Contents

Chapter 1

Introduction

The mechanism for deciding what to do next is one of the key components in an autonomous agent. For example, consider one of the most common problems in robotics that is also simple to illustrate: an autonomous robot that controls in which direction to move needs to navigate from its current position to a goal point (see figure 1.1(a)). The robot has some information about the environment, such as the areas that are traversable and those that are not, and based on this information needs to decide in which direction it should move next. The robot must take moves that make progress in reaching the goal point without causing catastrophic failures or situations that make impossible to reach the goal (e.g., falling off a cliff or moving into a one-way road that leads to a dead-end). A common approach that ensures this, at least when no critical information about the environment is missing, is to generate some plan of reaching a goal - a path shown in figure 1.1(a), for example - based on the currently available information, start executing the plan and re-plan if the environment changes or the agent gathers new information.

One of the widely-popular ways to generate a plan is graph-based search. Graph-based searches are theoretically well-grounded, extensively studied and general enough to be applicable in many domains. For instance, to generate a plan for our robot navigation example we first discretize the environment (figure 1.1(b)). We then construct a weighted graph where each state encodes the position of a cell, each directed edge between two states encodes an action of moving between two unobstructed cells that correspond to these states, and the weight of an edge is the length of the corresponding move (figure 1.1(c,d)). We then search the graph for a path from the



(a) environment map (b) discretized map

(c) a segment of discretized map (d) the graph corresponding to (c)

Figure 1.1: An example of planning for the robot navigation problem. The robot is initially at START and needs to navigate to GOAL. Untraversable areas are shown in black, traversable in white. (a) - the map of the environment as known to the robot; (b) - the map after discretization; (c) - a small segment of discretized map; (d) - the graph that corresponds to the map in (c) and is used by search to find a path.

state that corresponds to the position of the cell the robot is in to the state that corresponds to the position of the cell the goal is in. The found path is then the plan that the robot can follow. In particular, it can start executing the first action along this path. A uniform discretization that we used is, perhaps, one of the simplest ways to construct a graph. A number of other more complex but often more efficient alternatives exist such as graphs constructed based on non-uniform discretizations of environment (e.g., [82]), based on trapezoidal cell decomposition for environments with polygonal obstacles (e.g., [70]), based on random sampling of environment (e.g., [41]) and based on other methods (the description of many of them can be found in [55]).

Given a graph, the main question then becomes what search algorithm to use for finding a path of a reasonable cost in it. Dijkstra's algorithm is an efficient search algorithm for finding a *least-cost* path in a graph when no other information besides the graph is given. A* search [64] - a popular search algorithm in AI community - extends Dijkstra's algorithm by incorporating lower bounds on the costs of paths from states to the goal state. It uses these bounds to avoid the evaluation of states that have the bounds too high to belong to an optimal path from the start state to the goal state. While A* search also returns an optimal solution, it can often be several magnitudes faster than Dijkstra's algorithm when non-trivial lower bounds are provided. In figure 1.1 for instance, a simple Euclidean distance between a cell and the goal cell can be used as an effective and cheap to compute lower bound on the length of a shortest path from the corresponding state to the goal state.

Unfortunately, in reality the problems can often be too large for finding an optimal plan within an acceptable time. Moreover, even when an optimal plan is found initially, the model of the problem is rarely perfect, the world can rarely be predicted well, and therefore while executing the plan an agent may often find discrepancies in the model. It then needs to update the model and re-plan. Finding an optimal plan every time it needs to re-plan would make the agent stop execution for too long and too often. Anytime planning [14, 87] presents an appealing alternative. Anytime planning algorithms try to find the best plan they can within the amount of time available to them. They operate by quickly finding an approximate and possibly highly suboptimal plan first, and then improving it until the available time runs out. In addition to being able to meet time deadlines, many of such algorithms make it also possible to interleave planning and execution: while the agent executes its current plan, the planner can improve it. To this end, chapter 2 of this thesis develops an anytime heuristic search algorithm, called Anytime Repairing A* (ARA*). The algorithm has control over a suboptimality bound of the solution it produces, 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 significantly more efficient than the other few existing anytime search methods. The chapter demonstrates this empirically on the problem of motion planning for a high degree-of-freedom simulated robot arm.

Despite the usefulness of anytime planning algorithms for systems operating in the real world, the problem arises when the model is not close to being perfect or the environment is dynamic, and both cases do happen often. In these cases the agent needs to update the model often. Unfortunately, the updates to the model invalidate all of the previous efforts of the planner and it has to start working on a new plan from scratch. This is especially unfortunate 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 since the problem becomes different even though the differences can often be small. For example, in our robot navigation problem the robot may start out knowing the map only partially, plan assuming that all unknown space is traversable 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., [51, 63]). As a result, the robot has to plan frequently during its execution. Replanning algorithms are helpful in such cases as they use the results of previous planning efforts in finding a plan for a problem that has slightly changed. Chapter 3 of this thesis develops an incremental heuristic search algorithm, called Lifelong Planning A* (LPA*). The first search of LPA* is the same as that of A* search, but all subsequent searches of LPA* are usually much faster because it tries to reuse as much of its previous search effort as possible.

While LPA* speeds up substantially a series of searches for similar problems it lacks the anytime property of ARA*, namely, once it finds a solution it stops and does not improve the solution even if more planning time is available. LPA* can only be pre-configured either as a search for an optimal solution or as a search for a solution bounded by a given suboptimality factor. Chapter 4 addresses this by developing a search algorithm, anytime D* (AD*), that combines the anytime and incremental properties together. The algorithm re-uses its old search efforts while simultaneously improving its previous solution (ARA* capability) as well as re-planning if necessary (LPA* capability). 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 LPA*, AD* also provides bounds on the suboptimality of each solution it returns. In chapter 4 we experimentally demonstrate the advantages of AD* over the searches that are either anytime or incremental but not both on the problem of motion planning for a simulated robot arm.

In chapter 5 we demonstrate how AD* enables us to successfully solve the problem of efficient path-planning for mobile robots that takes into account the dynamics of the robot. Optimal trajectories in large outdoor environments involve fast motion and sweeping turns at speed. In such environments it is particularly important to take advantage of the robot's momentum and find dynamic rather than static plans. To address this we extend the problem of planning in a 2D state space, figure 1.1 example, to the problem of planning in a 4D state space: xy position, orientation, and velocity. High dimensionality and large environments result in very large state spaces for the planner, however. They make it computationally infeasible for the robot to plan optimally every time it discovers new obstacles. To solve this problem we built a planner that uses AD* in its core for planning and re-planning. Chapter 5 describes the planner and demonstrates its behavior. The planner has been successfully used for navigating several robotic systems in large outdoor environments that were initially completely or partially unknown.

The development of the three main algorithms in this thesis, namely anytime, incremental, and both anytime and incremental versions of A*, are all mainly due to the simple alternative view of A* search that we develop in section 2.1 and the extension of this view presented in section 3.2. We hope that this interpretation of A* search will inspire research on other search-based 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.

1.1 Background

In this thesis 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 $s \in S$, whereas $pred(s)$ denotes the set of predecessor states of state s. For any pair of states $s, s' \in S$ such that $s' \in succ(s)$ we require the cost between the two to be positive: $0 < c(s, s') \leq \infty$.

Every time a search algorithm is executed it is given a graph that represents the problem and two states, s_{start} and s_{goal}. The task of the 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 thus defines a sequence of valid transitions between states in the graph, and an agent can therefore execute the corresponding actions that will place the agent in the desired goal state if the planning model is correct. The cost of the path is the summation of the costs of the corresponding transitions, that is, $\sum_{i=1}^{k} c(s_{i-1}, s_i)$. 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, that is 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, $c^*(s_{\text{start}}, s)$. Let us denote such cost by $g^*(s)$. 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). To see that a path $\pi(s_{\text{start}}) = \{s_{\text{start}} = s_0, s_1, \ldots, s_{i-1}, s_i, \ldots, s_k = s_{\text{goal}}\}$ re-constructed this way will be a least-cost path just note that the state s' preceding any state s_i in any least-cost path from s_{start} to s_i must be such that the summation of the cost of a least-cost path from s_{start} to it and the cost of a transition from s' to s_i is minimal. Put mathematically, $s_{i-1} = \arg\min_{s' \in pred(s_i)}(g^*(s') + c(s', s_i))$.

Consequently, algorithms like A* search try to compute g^*-values. In particular, A* maintains g-values for each state it has visited so far. $g(s)$ 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 been visited by search yet). Thus, the g-values are always upper bounds on the g^*-values. The simple pseudocode in figure 1.2 maintains the g-values this way. This code is not A* search but rather is its generalization.

```
1 $OPEN = \{s_{\text{start}}\}$;
2 $g(s_{\text{start}}) = 0$;
3 while($s_{\text{goal}}$ is not expanded)
4   remove some state $s$ from $OPEN$;
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       insert $s'$ into $OPEN$;
```

Figure 1.2: Computation of the g-values via repetitive expansions

The code starts by setting $g(s_{\text{start}})$ to 0 and inserting s_{start} into *OPEN*. Given the assumption that all the g-values are initially ∞ (initialization can usually be done during the first time a state is visited by search), the g-values before the loop are all upper bounds on the corresponding g^*-values. The code then repetitively selects states from *OPEN* and expands them - executes lines 5 through 8 - until s_{goal} is expanded. At any point in time *OPEN* is a set of states that are candidates for expansion. These are also the states to which new paths have been found but have not been propagated to their children yet. As a result, 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 inserting it into *OPEN*. This way, s' will also be selected for expansion at some point and the cost of the found 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.

If, when the algorithm terminates, the g-values of states on at least one of the least-cost paths from s_{start} to s_{goal} are exactly equal to the corresponding g^*-values, then one can re-construct a least-cost path from s_{start} to s_{goal} 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). Let us call this path *greedy*. It is constructed in exactly the same way as a least-cost path constructed using g^*-values except here we use g-values instead. Since the g-values are all upper bounds on the g^*-values and the g-values of states on at least one of the least-cost paths from s_{start} to s_{goal} are exactly equal to the g^*-values, then only those states will be the minimizing states.

The goal then is to expand states in such order as to minimize the number of expansions required to guarantee that the states on at least one of the least-cost paths from s_{start} to s_{goal} are exactly equal to the g^*-values. One simple way to achieve this is to expand states in the order of their g-values, a state with the smallest g-value is always expanded first. *OPEN* becomes a priority queue that sorts states according to their g-values. This makes the algorithm look very much like Dijkstra's algorithm except that we terminate as soon as s_{goal} is expanded. In the case of uniform costs it is equivalent to Breadth-First Search. (The algorithms like this one do not use heuristics to focus their search and are therefore commonly referred to as uninformed searches [75].) The algorithm expands each state no more than once because every time it expands a state, a least-cost path to it has already been found and therefore, a better path to it will never show up later and the state will never be re-inserted into *OPEN*.

A* is another instance of the search algorithm in figure 1.2. One can think of it as a generalization of Dijkstra's algorithm as it expands states in the order of their g- plus h-values (i.e., $g(s) + h(s)$), where h-values estimate the cost of a least-cost path from s to s_{goal}. The h-values must never overestimate, or otherwise A* may return a suboptimal solution. Setting all h-values to zero reduces A* to the search that expands states in the order of their g-values, the uninformed search we have just described. A search with non-zero h-values is often called an informed search [75]. In order for each state not to be expanded more than once, h-values need to be also consistent: $h(s_{\text{goal}}) = 0$ and for any two states $s, s' \in S$ such that $s' \in succ(s)$, $h(s) \leq c(s, s') + h(s')$. If h-values are consistent then once again every time the search expands a state, a least-cost path to it has already been found and therefore a better path to it will never show up later and the state will never be re-inserted into *OPEN*. Ordering expansions based on the summation of g- and h-values makes the search focus expansions on the states through which the *whole* path from s_{start} to s_{goal} looks promising. In contrast, ordering expansions based on just g-values makes the search prefer expanding states that have paths from s_{start} to them of smallest costs, even if these states lead the search away from s_{goal}. The searches that use heuristics can often be tremendously faster and use much less memory than searches that do not.

1.2 Related Work

Anytime planning algorithms work by finding a first, possibly highly suboptimal, solution very fast and then continually working on improving the solution until allocated time expires. The idea of anytime planning has been proposed in AI community a while ago [14], and much work has been done on the development of anytime planning algorithms since then (for instance, [3, 35, 65, 69, 86]). Major fraction of the researched anytime planning algorithms are domain specific, however. For example, the work in [65] constructs an anytime planning algorithm for robot navigation in outdoor environments based on a multi-resolution representation of the environment using wavelets. The initial planning is done quickly in a low-resolution version of the environment. Afterwards, the plan is improved by planning in the higher-resolution versions. ARA*, the anytime planning algorithm described in chapter 2 of this thesis, does anytime planning on a commonly used graph representation of a problem and thus is independent of the domain. It is important to note, however, that ARA* does heavily rely on the existence of informative heuristics the efficient computation of which is usually specific to the domain.

Apart from often being domain specific, currently available anytime planning algorithms rarely provide bounds on the suboptimality of their solutions unless the cost of an optimal solution is already known. Even less often can these algorithms control their suboptimality. Providing suboptimality bounds is valuable, though: it allows one to judge the quality of the current plan, decide whether to continue or preempt search based on the current suboptimality, and evaluate the quality of past planning episodes and allocate time for future planning episodes accordingly. Control over the suboptimality bounds helps in adjusting the tradeoff between computation and plan quality. ARA* both provides suboptimality guarantees for its solutions and allows one to control these bounds. To the best of our knowledge ARA* is the only domain-independent anytime planning algorithm that is able to do this.

Much less work has actually been done on researching anytime graph-based searches. A simple and quite common way of transforming a search into an anytime search is to first search in only a small region of the state space surrounding the current state of an agent (such searches are commonly referred to as agent-centered searches [13,45]), return the best solution found there, and then to iteratively increase the region and return the best solution in the new region 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 coming up with *any* solution is hard within the provided time window and executing a partial path is an acceptable choice. On the negative side, such algorithms typically provide no bounds on the quality of their solutions[1] and may even return plans that lead to failures in domains that have states with irreversible conditions (e.g., a one-way traffic road that leads to a dead-end).

Out of anytime heuristic searches that return complete plans at any point in time, the most closely related to ARA* is anytime heuristic search algorithm called Anytime A* [84]. Both ARA* and Anytime A* rely on the fact that in many domains inflating the heuristic by some constant $\epsilon > 1$ can

[1]There does exist, however, a loose bound on the number of action executions required to reach a goal for an agent that uses an agent-centered search for planning. The bound is polynomial in the total number of states in the graph, and it assumes that the graph is strongly connected [50].

drastically reduce the number of states A* has to examine before it can produce a solution [6,8,11,17,52,71,84]. 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 [66]. When obtaining a first solution, both ARA* and Anytime A* inflate heuristics by a large ϵ. The major difference is in how the algorithms improve the solution afterwards. ARA* decreases ϵ and finds a new solution for this ϵ by efficiently repairing the current search tree. ARA* thus controls its anytime behavior through the control of ϵ. Anytime A*, on the other hand, simply continues to expand and re-expand states after the first solution is found pruning away from *OPEN* the states with f-values ($g(s) + h(s)$, where $h(s)$ is un-inflated) larger than the cost of the best solution found so far. Unlike ARA*, therefore, Anytime A* does not have a control over its suboptimality bound, except for the selection of the inflation factor of the first search. Our experiments show that ARA* is able to decrease its bounds much more gradually and, moreover, does so significantly faster. 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 chapter 2, [84] describes a number of interesting ideas that are also applicable to ARA*.

A few other anytime heuristic searches that return complete solutions have been developed [29, 54, 83, 85]. They all share the property, however, of not being able to provide any non-trivial suboptimality bounds on their solutions. On the other hand, most of them possess other nice properties that perhaps can make them more suitable to certain domains than ARA*. The algorithms such as depth-first branch-and-bound search [54] and Complete Anytime Beam search [83] are variants of Depth-First search that prune away the states that seem unlikely to belong to a good quality path from start to goal (based on their f-values). As such, on the positive side, they tend to use a much more limited amount of memory than A* and its variants, but on the negative side tend to re-expand states exponentially many times and are therefore mainly suited to domains with numerous paths to the goal where there is a high chance of finding a solution of a reasonable quality just by exploring a number of trajectories. Beam-Stack search [85] and ABULB [29], on the other hand, are variants of Breadth-First search that also prune away the states for which a path from start to goal through them looks "bad", but both backtrack to guarantee completeness. They can be thought of as generalizations of Depth-First branch-and-bound algorithms in that they explore more than one trajectory at a time. Consequently, they are also memory-bounded anytime search algorithms, but they can re-expand states many times before producing even an initial solution. This is unlike ARA* which guarantees that states are never re-expanded while working on a solution for a particular suboptimality bound, a similar guarantee that A* with consistent heuristics does. Memory-bounded algorithms may scale up to larger domains than ARA* though, if memory becomes a bottleneck in obtaining and improving a solution within the provided time window.

The field of re-planning has also been investigated extensively in the AI community. Similarly to the case with anytime planning, however, most of the work in re-planning lies outside of search-based planning, the type of planning LPA* algorithm proposed in chapter 3 of this thesis performs. Most notably, re-planning has been heavily developed in the area of symbolic planning. Different from search-based planners, symbolic planning and replanning algorithms use logical representations of states and the effects of actions to avoid the instantiation of every possible state and to direct planning in a potentially more effective way. The examples of symbolic replanning methods include case-based planning, planning by analogy, plan adaptation, transformational planning, planning by solution replay, repair-based planning, and learning search-control knowledge. These replanning methods have been used as part of systems such as CHEF [33], GORDIUS [76], LS-ADJUST-PLAN [30], MRL [44], NoLimit [81], PLEXUS [2], PRIAR [40], and SPA [34]. NoLimit, for example, accelerates a backward-chaining nonlinear planner that uses means-ends analysis, SPA accelerates a causal-link partial-order planner, PRIAR accelerates a hierarchical nonlinear planner, and LS-ADJUST-PLAN accelerates a planner that uses planning graphs. Besides being based on search-based planning LPA* also differs from these algorithms in that it finds solutions that satisfy a user specified suboptimality bound including optimal solutions if the bound is set so. In contrast, the above listed replanning algorithms usually provide no suboptimality bounds on their solutions.

LPA* configured to return only optimal solutions [48] (i.e., a suboptimality bound set to one) falls into the category of incremental search methods. Such methods solve dynamic shortest path problems, that is, path problems where shortest paths have to be determined repeatedly as the topology of a graph or its edge costs change [73]. They differ from symbolic replanning methods in that they find shortest paths. A number of incremental search methods have been suggested in the algorithms literature [5, 18, 19, 24–26, 31, 37, 43, 60, 72, 74, 77] and, to a much lesser degree, the artificial intelligence literature [15]. They are all uninformed, that is, do not use heuristics to focus their search, but differ in their assumptions, for example, whether they solve single-source or all-pairs shortest path problems, which performance measure they use, when they update the shortest paths, which kinds of graph topology and edge costs they apply to, and how the graph topology and edge costs are allowed to change over time [27]. If arbitrary sequences of edge insertions, deletions, or weight changes are allowed, then the dynamic shortest path problems are called fully dynamic shortest path problems [28]. LPA* configured to search for optimal solutions is an incremental search method that solves fully dynamic shortest path problems but, different from the incremental search methods cited above, uses heuristics to focus its search and thus combines two different techniques to reduce its search effort. LPA* as presented in this thesis can also be configured to return solutions for any bound of suboptimality (a particular instance of a general framework developed in [59]). It thus may also be suitable to the domains where optimal planning is infeasible.

A particularly closely related to LPA* is an algorithm called DynamicSWSF-FP [72]. The operation of LPA* configured to search for optimal solutions follows very closely the operation of DynamicSWSF-FP and even borrows such central concepts as state consistency. LPA* differs from DynamicSWSF-FP in the following ways. First and most importantly, LPA* is an informed search and therefore can use heuristics if available to boost drastically its efficiency (as evidenced by our experiments in chapter 3). Second, unlike DynamicSWSF-FP which can only search for optimal solutions, LPA* as presented in this thesis can be configured to trade-off the solution quality for the computational expense of finding it. It is important in domains where a search for an optimal solution is infeasible. Other, smaller differences between LPA* and DynamicSWSF-FP include the fact that LPA* searches forward rather than backward and the fact that LPA* maintains a solution from the start to the goal state rather than *all* shortest paths to the goal.

A few incremental search methods have been developed that are also informed, that is, can make use of heuristics. All of them have been developed in the context of robot navigation. Methods like [68, 80] perform re-planning by identifying the perimeter of areas in which the states may need to be updated, and they restart the search from there. LPA* differs from these in that it finds and updates the states that need to be updated during the search itself, and therefore directly generalizes A* search to an incremental search. To the best of our knowledge, the only other incremental heuristic search method that tries to update *only* the states that need to be updated is D* [78]. D* and LPA*, and in particular, its extension, called D* Lite [47], that was developed specifically for a moving agent, are similar in many respects and are comparable in their performances. The algorithms, however, are nevertheless different. In contrast to D*, LPA* is substantially simpler, has a number of theoretical properties including ones that make it an incremental version of A* and bound the number of times each state can be expanded. LPA* as presented in this thesis also differs from D* and all the other incremental heuristic searches in that it can plan and re-plan solutions for a given suboptimality bound, rather than only optimal solutions. This broadens the spectrum of problems incremental heuristic searches can be used for.

Only few anytime replanning algorithms have been developed 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 [12], for example, is capable of always returning a plan and constantly works on improving the plan and fixing it as changes in the environment are detected. This is achieved, however, at the expense of often returning plans that are only partial. The system may not be suitable in cases when a complete solution is required before the agent starts execution. A planner described in [23] 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, Anytime D*, described in chapter 4 of this thesis, it also always returns a complete plan rather than a partial plan. Different from Anytime D*, however, it provides no guarantees on the suboptimality of a solution.

We know of no heuristic search algorithm that is both anytime and incremental besides Anytime D* we present here. The figure 1.3 perhaps can serve to better visualize different incremental and anytime algorithms developed in the area of search-based planning. In light grey color are shown algorithms that are optimal, that is, can only search for optimal solutions, and in dark grey are algorithms that are able to trade-off optimality for computational efficiency. In bold are shown the algorithms that are able to provide suboptimality bounds on their solutions. Clearly, all optimal algorithms are shown

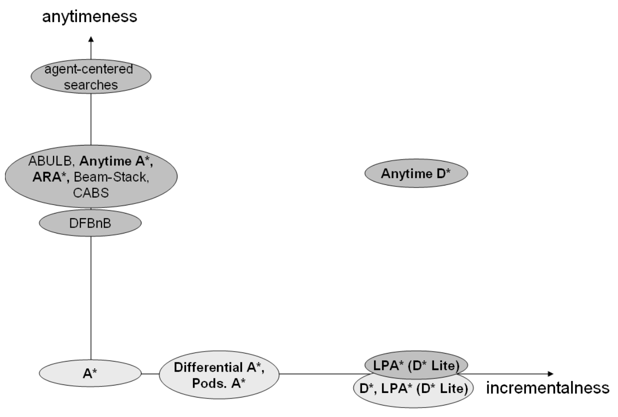

anytimeness
agent-centered searches
ABULB, **Anytime A*, ARA***, Beam-Stack, CABS
DFBnB
Anytime D*
A*
Differential A*, Pods. A*
LPA* (D* Lite)
D*, LPA* (D* Lite)
incrementalness

Figure 1.3: A graphical overview of the research on anytime and incremental heuristic searches. In bold are shown the algorithms that provide suboptimality guarantees on their solutions. In light grey ovals are listed the algorithms that can only search for optimal plans. The algorithms presented in this thesis are ARA*, LPA* (and its extension D* Lite) and Anytime D*. Note that Anytime D* is the only algorithm that lies off both of the axes.

in bold. On the y-axis are different anytime search algorithms ordered by their anytime capability in some loose sense. Thus, the "most anytime" algorithms are agent-centered searches [13, 45] as they can provide a plan in any size and complexity domain since the plan is only partial and can often be just a few first actions. The next group of anytime algorithms going downwards along the y axis includes ABULB [29], Anytime A* [84], ARA* [57] (which is presented in this thesis), Beam-Stack [85], and CABS [83]. These algorithms work on returning a complete plan even initially and therefore may be less anytime in some hard-to-solve domains. Out of those algorithms only Anytime A* and ARA* provide bounds on suboptimality. Next is DFBnB [54] (depth-first branch-and-bound) which is also an anytime algorithm in the domains where there is a high chance of finding a path. On the x-axis are incremental algorithms ordered by their degree of incrementalness in some loose sense. Thus, at the origin is A* algorithm that is neither anytime nor incremental. All of the incremental algorithms are optimal algorithms except for LPA* [59] (and its extension to a moving robot, D* Lite [47]). The graph shows LPA* as both an optimal search and a search that can trade-off optimality for computational efficiency. Originally, LPA* was developed as an optimal search [48]. Later, it was extended to a search that can find solutions for any suboptimality bound [59]. The latter is the version presented in this thesis. Currently and as far as we know, Anytime D* algorithm [56] is the only heuristic search algorithm that lies off the two axis because it is both anytime and incremental. It is also shown in bold as it provides bounds on its solution suboptimality. The graph also shows that there is a need for the future development of algorithms that add incremental planning to the algorithms that are anytime in *any* domain at the expense of being incomplete (e.g., agent-centered searches).

Chapter 2

ARA*: Anytime A* Search with Provable Bounds on Suboptimality

2.1 Single-shot Search under Overconsistent Initialization

In section 2.2 we will develop anytime heuristic search algorithm, ARA*, that provides bounds on its suboptimality. ARA* operates by executing a series of searches with decreasing suboptimality bounds on the solution. It quickly generates a first plan that satisfies the initial suboptimality bound and then improves the plan so that it continues to satisfy the new bounds. ARA* gains efficiency by making each search iteration to reuse the results of its previous search iterations. In this section we will develop a novel formulation of A* search that enables us to reuse the search results of its previous executions quite easily.

In this section we will define the notion of an inconsistent state and then formulate A* search as a repetitive expansion of inconsistent states. It turns out that such search can be made to reuse the results of its previous executions just by identifying all of the inconsistent states beforehand. In this section we will also generalize the priority function that A* uses to any function satisfying certain restrictions. This is necessary because ARA* will use the search with different prioritization functions.

2.1.1 Reformulation of A* Search Using Inconsistent States

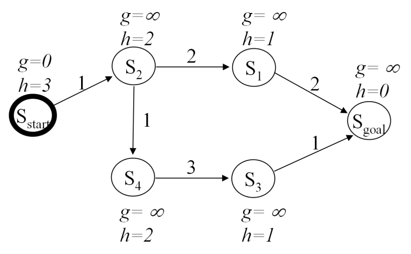
The pseudocode below assumes the following :

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

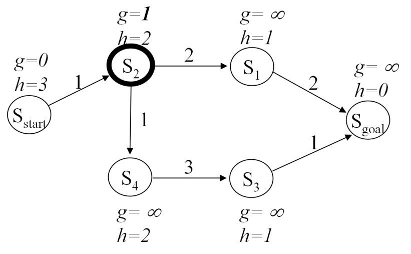
```
1 procedure ComputePath()
2 while($s_{\text{goal}}$ is not expanded)
3   remove $s$ with the smallest $f(s)$ from $OPEN$;
4   for each successor $s'$ of $s$
5     if $g(s') > g(s) + c(s, s')$
6       $g(s') = g(s) + c(s, s')$;
7       insert/update $s'$ in $OPEN$ with $f(s') = g(s') + h(s')$;
```

Figure 2.1: A* Search: ComputePath function

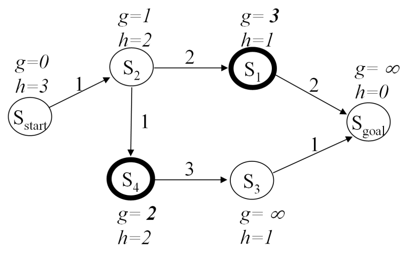
In figure 2.1 we give a standard formulation of A* search. It is a particular instance of the code given in figure 1.2. It specifies that the priorities according to which states are chosen from $OPEN$ for expansion are their f-values, the summation of g- and h-values. Since $g(s)$ is the cost of the 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}, then $f(s)$ is an estimate of the cost of the best path from s_{start} to s_{goal} via state s. If h-values are admissible, that is, never overestimate the cost of the least-cost path from s to s_{goal}, then A* search guarantees to find an optimal path. If h-values are 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 make sure to decrease the g-value of a successor of s, whenever it is possible to do so using $g(s)$. Once the search finishes, the solution is given by the greedy path, the path that is re-constructed backwards 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). Figure 2.2 demonstrates the operation of A* search on a simple example. We will use later the same example to show the operation of our alternative formulation of A* search (figure 2.7).



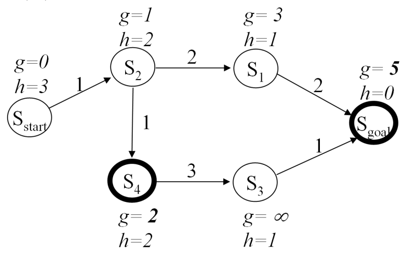
(a) after initialization



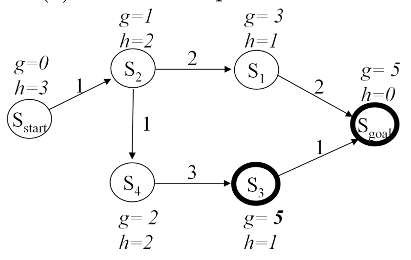
(b) after the expansion of s_{start}



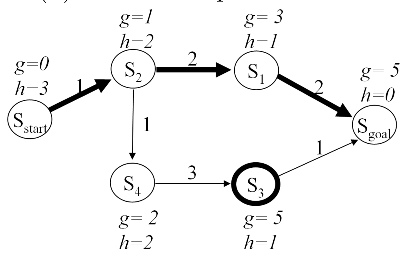
(c) after the expansion of s_2



(d) after the expansion of s_1



(e) after the expansion of s_4



(f) after the expansion of s_{goal}

Figure 2.2: 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.

We will now introduce a new variable, called $v(s)$. Intuitively, these v-values will also be the estimates of start distances, same as the g-values. However, while $g(s)$ is always the cost of the best path found 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 ∞, same as the corresponding g-value, except for $g(s_{\text{start}})$, and then it is always reset to the value of the state at the time it is being expanded. The new pseudocode that involves these v-values is given in figure 2.3. In bold are shown the introduced changes. The pseudocode in figure 2.3 is the exact equivalent of the pseudocode of original A* search in figure 2.1: the v-values are only being set (line 4) and do not participate in the computation of any other values.

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. $OPEN = \{s_{\text{start}}\}$.

```
1 procedure ComputePath()
2 while($s_{goal}$ is not expanded)
3   remove $s$ with the smallest $f(s)$ from $OPEN$;
4   $\mathbf{v(s) = g(s)}$;
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       insert/update $s'$ in $OPEN$ with $f(s') = g(s') + h(s')$;
```

Figure 2.3: A* Search: ComputePath function with v-values

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 6 and 7). The only way how $v(s)$ would 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. One benefit of introducing v-values is the following invariant that A* maintains: for every state $s' \in S$,

$$g(s') = \min_{s'' \in pred(s')} (v(s'') + c(s'', s')). \qquad (2.1)$$

More importantly, what we ask the reader to observe is that only the states s whose $v(s) \neq g(s)$ are ever appear in *OPEN*. It is so 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 being 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 a g-value of any state is modified on line 7 it only decreases and thus becomes strictly less than the corresponding v-value which is either still infinite if the state was not expanded yet or equal to what the g-value of the state was during its last expansion. After each modification of the g-value, the state is made sure to be in *OPEN* (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* is always a set of all and only states that are inconsistent. Consequently, since all the states for expansion are chosen from *OPEN* A* search expands only inconsistent states.

Here is a bit of intuitive explanation of A* operation 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 remains inconsistent every time its g-value is decreased and until the next time the state is expanded (s_{start} remains also initially inconsistent until its first expansion since its g-value is initially zero). That is, suppose that a consistent state s is the best predecessor for some state s': $g(s') = \min_{s'' \in pred(s')}(v(s'') + c(s'', s')) = v(s) + c(s, s') = g(s) + c(s, s')$. Then, if $g(s)$ decreases we get $g(s') > g(s) + c(s, s')$ and therefore $g(s') > \min_{s'' \in pred(s')}(g(s'') + c(s'', s'))$. In other words, the decrease in $g(s)$ introduces an inconsistency between the g-value of s and the g-values of its successors. Whenever s is expanded, on the other hand, the inconsistency of s is corrected by re-evaluating the g-values of the successors of s. This in turn makes 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.

Given this definition of inconsistency it is clear that the *OPEN* list consists of exactly all inconsistent states: every time a g-value is lowered the state is inserted into *OPEN*, and every time a state is expanded it is removed from *OPEN* until the next time its g-value is lowered. Thus, the *OPEN* list can be viewed as a set of states from which we need to propagate inconsistency. The algorithm itself in its operation is still identical to A* search. The variable v just makes it easy 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 chapters we will encounter states that are *underconsistent*, the states s with $v(s) < g(s)$.

2.1.2 Generalizing Priorities

A* search uses one of the possible state expansion orderings. It expands states in the order of increasing f-values. For any admissible heuristics, this ordering guarantees optimality. In this section we generalize A* search as presented in figure 2.3 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. The pseudocode of this generalized version is given in figure 2.4. It is the same pseudocode as before except now the function key() determines the order in which states are expanded, and it can be any function satisfying the restriction in the first assumption of the pseudocode. The variable ϵ can be any finite real value larger than or equal to one. 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 A* search with inflated heuristics, i.e., weighted A* search.

The restriction can be "translated" as follows. We are given two states: an overconsistent state s (and therefore a candidate for expansion) and a *possibly* overconsistent state s'. The condition $g(s') > g(s) + \epsilon * c^*(s, s')$ 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, 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 setting $key(s)$ to a smaller value than $key(s')$.

In section 2.3 we will show that if the restriction 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. It is easy to see that the restriction encompasses the prioritization done by uninformed optimal searches such as Dijkstra's algorithm, A* with consistent heuristics and A* with consistent heuristics inflated by some constant. For example, in 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 $key(s') = g(s') > g(s) + \epsilon * c^*(s, s') = key(s) + \epsilon * c^*(s, s') \geq key(s)$

The pseudocode below assumes the following:

1. key function satisfies the following restriction: for any two states $s, s' \in S$ if $c^*(s', s_{\text{goal}}) < \infty$, $v(s') \geq g(s')$, $v(s) > g(s)$ and $g(s') > g(s) + \epsilon * c^*(s, s')$, then key$(s') >$ key(s);
2. the 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);
3. $OPEN = \{s_{\text{start}}\}$.

```
1 procedure ComputePath()
2 while($s_{\text{goal}}$ is not expanded)
3   remove $s$ with the smallest key($s$) from $OPEN$;
4   $v(s) = g(s)$;
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       insert/update $s'$ in $OPEN$ with key($s'$);
```

Figure 2.4: A* search with a generalized priority function: ComputePath function

since costs cannot be negative. Thus, the solution is optimal. In 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

$$key(s') = g(s') + \epsilon * h(s') > g(s) + \epsilon * h(s') + \epsilon * c^*(s, s') \geq g(s) + \epsilon * h(s) = key(s). \tag{2.2}$$

where we used the fact that $h(s) \leq c^*(s, s') + h(s')$ when heuristics are consistent [66]. 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 [59]: $h(s_{\text{goal}}) = 0$ and for any two states $s, s' \in S$ $h(s) \leq \epsilon * c(s, s') + h(s')$. Many different heuristics are ϵ-consistent for a suitable ϵ including consistent heuristics, consistent heuristics inflated by some constant, 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 [59].

In general, when heuristics are inconsistent the states in A* search may be re-expanded multiple times. As we prove later in section 2.3, if we restrict the expansions to no more than one per state, then the algorithm is still complete and possesses ϵ-suboptimality if heuristics satisfy ϵ-consistency. We restrict the expansions using the set *CLOSED* in the pseudocode shown in figure 2.5 (in the same way it is usually used in A* when re-expansions are not allowed).

The pseudocode below assumes the following:

1. key function satisfies the following restriction: for any two states $s, s' \in S$ if $c^*(s', s_{\text{goal}}) < \infty$, $v(s') \geq g(s')$, $v(s) > g(s)$ and $g(s') > g(s) + \epsilon * c^*(s, s')$, then $\text{key}(s') > \text{key}(s)$;
2. the 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);
3. $CLOSED = \emptyset$ and $OPEN = \{s_{\text{start}}\}$.

```
1 procedure ComputePath()
2 while($s_{goal}$ is not expanded)
3   remove $s$ with the smallest key($s$) from $OPEN$;
4   $v(s) = g(s)$; $CLOSED \leftarrow CLOSED \cup \{s\}$;
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($s'$);
```

Figure 2.5: A* search with a generalized priority function: ComputePath function with at most one expansion per state

Initially, *CLOSED* is empty. Afterwards, every state that is being expanded is added to it (line 4) and no state that is already in *CLOSED* is inserted into *OPEN* and thus considered for expansion (line 8).

2.1.3 Generalizing to Arbitrary Overconsistent Initialization

In previous sections we had a fixed initialization of states. We set the v-values and the g-values of all states except for s_{start} to infinity, we set $v(s_{\text{start}})$ also to infinity and we set $g(s_{\text{start}})$ to 0. In this section we relax this assumption and the only restriction we make is that no state is underconsistent (with v-value strictly smaller than its g-value) and all g-values satisfy the invariant that A* maintains, namely the equation 2.1. This arbitrary overconsistent initialization will allow us to re-use previous search results when running more than one search in a row, by re-using the state values from previous searches.

The pseudocode under this initialization is shown in figure 2.6. It remains exactly the same as before except for the terminating condition (line 2) of the while loop. The loop now terminates as soon as the $key(s_{\text{goal}})$ becomes the same or less than 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 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 in such a way that all of them are consistent, then *OPEN* is initially empty, and the search terminates without a single expansion. This is correct, because when all states are consistent, then for every state s, $g(s) = \min_{s' \in pred(s)}(v(s') + c(s', s)) = \min_{s' \in pred(s)}(g(s') + c(s', s))$, which means that the g-values are equal to corresponding g^*-values and no search is necessary anymore, the greedy path is an optimal solution.

The pseudocode below assumes the following:

1. key function satisfies the following restriction: for any two states $s, s' \in S$ if $c^*(s', s_{\text{goal}}) < \infty$, $v(s') \geq g(s')$, $v(s) > g(s)$ and $g(s') > g(s) + \epsilon * c^*(s, s')$, then $\text{key}(s') > \text{key}(s)$;

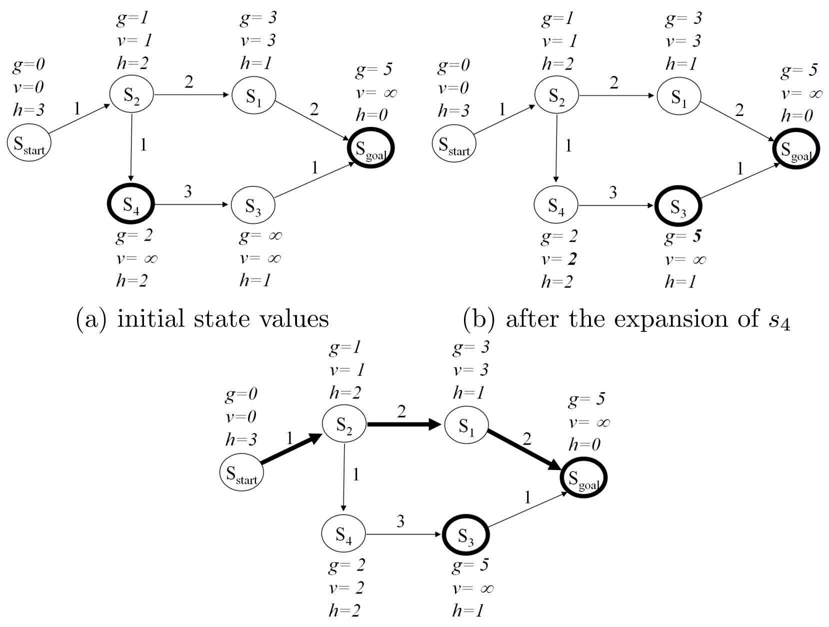
2. $v-$ and $g-$values of all states are initialized in such a way that $v(s) \geq g(s) = \min_{s' \in pred(s)}(v(s') + c(s', s))\ \forall s \in S - \{s_{\text{start}}\}$ and $v(s_{\text{start}}) \geq g(s_{\text{start}}) = 0$ (the initialization can also occur whenever ComputePath encounters new states);

3. *CLOSED* $= \emptyset$ and *OPEN* contains exactly all and only 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   $v(s) = g(s)$; $CLOSED \leftarrow CLOSED \cup \{s\}$;
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($s'$);
```

Figure 2.6: A* search with a generalized priority function and under generalized overconsistent initialization: ComputePath function

In figure 2.7 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 ARA* that will execute the ComputePath function repeatedly, gradually improving



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

(c) after the computation of a greedy path

Figure 2.7: An example of how the ComputePath function operates under an arbitrary overconsistent initialization. States are expanded in the order of f-values (the summation of g- and h-values). All overconsistent states need to be in *OPEN* initially. The states that are 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 the same as what regular A* search would have generated (figure 2.2).

its solution. The prioritization function in the example is the summation of g- and h-values. That is, $key(s) = g(s) + h(s)$.

In section 2.3 we prove the theorem about the correctness, ϵ-suboptimality and the efficiency of this version of A* search. Here, we just give few of the most important of those theorems. The key 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 smaller 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 a g-value 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, it is clear that after the algorithm terminates, it holds that $g(s_{\text{goal}}) \leq \epsilon * g^*(s_{\text{goal}})$ and the found 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* $\epsilon * g^*(s_{\text{goal}})$.

Just like 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, the states with the v-values that cannot be lowered by finding better paths to them are not expanded. This may result in substantial computational savings when using this search for repetitive planning as discussed in the next section when presenting the ARA* algorithm.

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

2.2 Anytime Search

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

Normally, the pseudocode of A* search in figure 2.6 uses $key(s) = g(s)+h(s)$, where h-values 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 can drastically reduce the number of states it has to

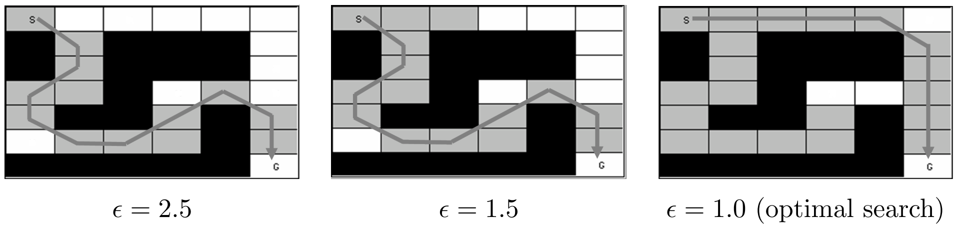


Figure 2.8: A* searches with inflated heuristics

examine before it produces a solution [6,8,11,17,52,71,84]. In our framework this is equivalent to setting $key(s) = g(s) + \epsilon * h(s)$. While the path the search returns can be suboptimal, the search also provides a bound on the suboptimality, namely, the ϵ by which the heuristic is inflated [66][1]. 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 length of the found path is no larger than ϵ times the length of the optimal path [66], while the search can often be much faster than its version with uninflated heuristics.

For example, figure 2.8 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 being obstacles. S denotes a start state, while G denotes a goal state. The cost of moving from one cell to its neighbor is one. The heuristic is the larger of the x and y distances from the cell to the goal. The cells which were expanded are shown in grey. (Our version of A* search stops as soon as it is about to expand a goal state without actually expanding it. 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 solution is suboptimal.

To construct an anytime algorithm with suboptimality bounds, one could run a succession of these A* searches with decreasing inflation factors, just like we did in the example. This naive approach results in a series of solutions, each one with a suboptimality factor 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

[1]Note that as mentioned earlier restricting each state to be expanded at most once still guarantees ϵ-suboptimality of the path. We prove this in section 2.3.

(e.g., [48]), but the suboptimality bounds for each search iteration would no longer be guaranteed. In the next section we propose ARA* (Anytime Repairing A*) algorithm, which is an *efficient* anytime heuristic search that also runs A* with inflated heuristics in succession but reuses search efforts from previous executions in such a way that the suboptimality bounds are still satisfied. As a result, a substantial speedup is achieved by not re-computing the state values that have been correctly computed in the previous iterations.

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

ARA* works by executing A* multiple times (just like in the example in figure 2.8), starting with a large ϵ and decreasing ϵ prior to each execution until $\epsilon = 1$. As a result, after each search a solution is guaranteed to be within a factor ϵ of optimal. Unlike it is done in the example, however, ARA* reuses the results of the previous searches to save computation.

For our formulation of A* search in section 2.1.3 (pseudocode in figure 2.6), the reuse is trivial. 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 have 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 itself. Consequently, to make search reuse previous search efforts we only need to make sure that before each execution of the ComputePath function *OPEN* contains *all* 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 as they will be the starting points for the inconsistency propagation in the future search iterations. We do this by maintaining the set *INCONS* of all the inconsistent states that are not in *OPEN* (lines 12 and 13 in figure 2.9). 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.

Apart from the maintenance of the *INCONS* set the ComputePath function is almost identical to the ComputePath function of A* search that we have presented in section 2.1.3. The only other difference is the explicit initialization of the states as they are encountered by ARA*. Note that the states are initialized once per ARA* execution and *not* every time ComputePath encounters them for the first time during its current search. The function key() used by ComputePath is a summation of g-value and h-value inflated by the current value of ϵ as given in figure 2.10.

```
procedure ComputePath()
while(key($s_{\text{goal}}$) > $\min_{s \in OPEN}$(key($s$)))
  remove $s$ with the smallest key($s$) from $OPEN$;
  $v(s) = g(s)$; $CLOSED \leftarrow CLOSED \cup \{s\}$;
  for each successor $s'$ of $s$
    if $s'$ was never visited by ARA* before then
      $v(s') = g(s') = \infty$;
    if $g(s') > g(s) + c(s, s')$
      $g(s') = g(s) + c(s, s')$;
      if $s' \notin CLOSED$
        insert/update $s'$ in $OPEN$ with key($s'$);
      else
        insert $s'$ into $INCONS$;
```

Figure 2.9: ARA*: ComputePath function. ARA* specific changes as compared with A* search as formulated in figure 2.6 are shown in bold.

The main function of ARA* (figure 2.10) performs a series of search iterations. It does initialization and then repetitively 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 (line 12, Figure 2.10)[2]. Thus, after each call to the ComputePath function we get a solution that is suboptimal by at most a factor of ϵ.

As suggested in [84] a suboptimality bound can also be computed as the ratio between $g(s_{goal})$, which gives an upper bound on the cost of an optimal solution, and the minimum un-weighted f-value of a inconsistent state, which gives a lower bound on the cost of an optimal solution:

[2]At least in our domains, the reordering operation tends to be inexpensive in comparison to the overall search time. If necessary, however, one could also employ the optimization discussed in [47] in the context of D* Lite algorithm. It avoids the reordering operation altogether.

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 $s \neq s_{\text{goal}}$ and $h(s) = 0$ if $s = s_{\text{goal}}$.

```
procedure key($s$)
return $g(s) + \epsilon * h(s)$;

procedure Main()
$g(s_{\text{goal}}) = v(s_{\text{goal}}) = \infty$; $v(s_{\text{start}}) = \infty$;
$g(s_{\text{start}}) = 0$; $OPEN = CLOSED = INCONS = \emptyset$;
insert $s_{\text{start}}$ into $OPEN$ with key($s_{\text{start}}$);
ComputePath();
publish current $\epsilon$-suboptimal solution;
while $\epsilon > 1$
  decrease $\epsilon$;
  Move states from $INCONS$ into $OPEN$;
  Update the priorities for all $s \in OPEN$ according to key($s$);
  $CLOSED = \emptyset$;
  ComputePath();
  publish current $\epsilon$-suboptimal solution;
```

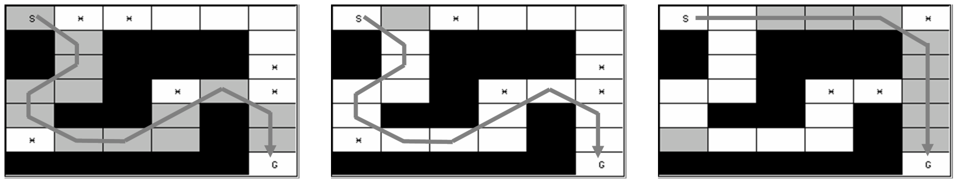
Figure 2.10: ARA*: key and Main functions

$$\frac{g(s_{goal})}{\min_{s \in OPEN \cup INCONS}(\text{g}(s)+\text{h}(s))} \tag{2.3}$$

This is a valid suboptimality bound as long as the ratio is larger than or equal to one. Otherwise, $g(s_{goal})$ 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.

$$\epsilon' = \min(\epsilon, \frac{g(s_{goal})}{\min_{s \in OPEN \cup INCONS}(\text{g}(s)+\text{h}(s))}). \tag{2.4}$$

At first, one may also think of using this actual suboptimality bound in 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 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



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

Figure 2.11: ARA* search

decrease in ϵ, on the other hand, may often result in the expansion of too many states during the next search.

Another useful suggestion from [84], 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 $g(s_{goal})$. This may turn out to be useful in domains with very high branching factor, where an expansion of a 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 we mainly save computation by not re-expanding the states whose v-values were already correct before the call to ComputePath (Theorem 6 states this more precisely). For example, figure 2.11 shows a series of calls to the ComputePath function on the same example we used before (figure 2.8). 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.

If one is interested in interleaving search with the execution of the current best plan, one needs to address the problem that the state of the agent changes. In particular, as the agent moves, the starting state changes. A way to deal with this problem is to perform the search backwards. That is, the actual 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}. 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 to the current state of the agent. As a result, the heuristics change as the agent moves. This, in turn, changes the priorities of the states in the *OPEN* list. Since ARA* algorithm reorders the *OPEN* list after each iteration anyway, however, we can recompute the heuristic values of the states in the *OPEN* list during the reorder operation (line 12 in figure 2.10).

2.2.3 Theoretical Properties of ARA*

Since the ComputePath function of ARA* is essentially the ComputePath function of A* search as we have presented it in section 2.1.3 it inherits all of its properties. We now list some of the theoretical properties of ARA*. For the proofs of these and other properties of the algorithm please refer to section 2.4. 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 which is within a factor of ϵ of optimal.

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 before each iteration ϵ is decreased (and, in case ARA* publishes ϵ' bounds, ϵ, in its turn, is an upper bound on ϵ'), ARA* gradually decreases the suboptimality bound and finds new solutions to satisfy the bound.

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

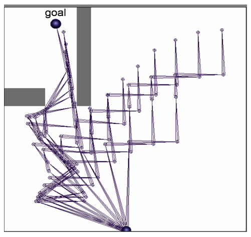
The second theorem formalizes where the computational savings for ARA* search come from. A usual implementation of A* search with inflated heuristics can and does perform multiple re-expansions of many states. Each search iteration in ARA*, on the other hand, is guaranteed not to expand states more than once. Moreover, it also does not expand states whose v-values before a call to the ComputePath function have already been correctly computed by some previous search iteration.

2.2.4 Experimental Analysis of the Performance of ARA*

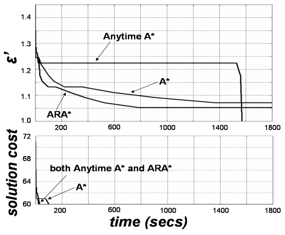
In this section we evaluate the performance of ARA* on simulated 6 and 20 degree of freedom (DOF) robotic arms (Figure 2.12) and compare it against other anytime heuristic searches that can provide suboptimality bounds, namely, Anytime A* [84] and the succession of A* searches with decreasing ϵ values (as described in section 2.2.1). The base of the arm is fixed, and the task is to move its end-effector to the goal 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 50 by 50 cells and compute a distance from each cell to the cell containing the goal while taking into account that some cells are occupied by obstacles. This distance is our heuristic. In the environment where all obstacles are touching walls, this heuristic seems to always direct the robot arm in an approximately correct direction. This property makes all the three algorithms we compare to exhibit a good anytime behavior.

In order for the heuristic not to overestimate true costs, joint angles are discretized so as to never move the end-effector by more than one cell in a single action. 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 states is allocated dynamically.

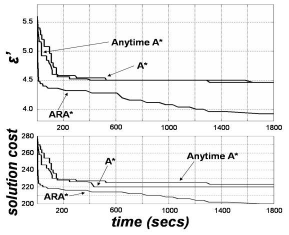
Figure 2.12(a) shows the planned trajectory of the robot arm after the initial search of ARA* with $\epsilon = 3.0$. This search takes about 0.05 secs. The plot in Figure 2.12(b) shows that ARA* improves both the quality of the solution and the bound on its suboptimality faster and in a more gradual manner than either a succession of A* searches or Anytime A* [84]. In this experiment ϵ is initially set to 3.0 for all three algorithms. For all the experiments in this section ϵ is decreased in steps of 0.02 (2% suboptimality) for ARA* and a succession of A* searches. Anytime A* does not control ϵ, and in this experiment it performs a lot of computations that result in a



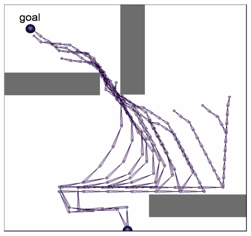
(a) 6D arm trajectory for $\epsilon = 3$



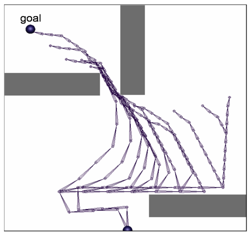
(b) uniform costs



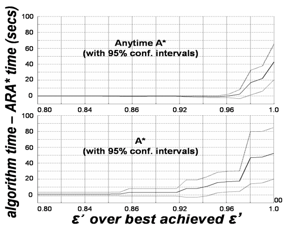
(c) non-uniform costs



(d) both Anytime A* and A*
after 90 secs
cost=682, ϵ'=15.5



(e) ARA*
after 90 secs
cost=657, ϵ'=14.9



(f) non-uniform costs

Figure 2.12: Top row: 6D robot arm experiments. Bottom row: 20D robot arm experiments (the trajectories shown are downsampled by 6). Anytime A* is the algorithm in [84].

large decrease of ϵ at the end. On the other hand, it does reach the optimal solution first. To evaluate the expense of the anytime property of ARA* we also ran ARA* and an optimal A* search in an environment with larger gap between obstacles (for the optimal A* search to be feasible). Optimal A* search required about 5.3 mins (2,202,666 state expanded) to find an optimal solution, while ARA* required about 5.5 mins (2,207,178 state expanded) to decrease ϵ in steps of 0.02 from 3.0 until a provably optimal solution was found (about 4% overhead). In other domains such as path planning for robot navigation, though, we have observed the overhead to be up to 30 percent.

While in the experiment for Figure 2.12(b), all the actions have the same cost, in the experiment for Figure 2.12(c) actions have non-uniform costs. This is because changing a joint angle closer to the base is more expensive than changing a joint angle further away. As a result of the non-uniform costs, our heuristic becomes less informative, and so search is 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. Reading the graph differently, 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 (about 140-fold speedup by ARA*) and Anytime A* reaches it after over 4 million expansions and 8.8 minutes (over 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 2.12 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* performed 6,378 expansions, while Anytime A* performed 8,994 expansions, mainly because some of the states were expanded up to seven times before a first solution was found.

Figures 2.12(d-f) show the results of experiments done on a 20 DOF robot arm, with actions that have non-uniform costs. All three algorithms start with $\epsilon = 30$. Figures 2.12(d) and 2.12(e) show that in 90 seconds of planning the cost of the trajectory found by ARA* and the suboptimality bound it can guarantee is substantially smaller than for the other algorithms. For example, the trajectory in Figure 2.12(d) 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 2.12(e). The graph in Figure 2.12(f) compares the performance of the three algorithms on twenty randomized environments similar to the environment in Figure 2.12(d). The environments had random goal locations, and the obstacles were slid to random locations along the outside walls. The graph shows the additional time the other algorithms require to achieve the same suboptimality bound that ARA* does. 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.

2.3 Proofs for Single-shot Search under Overconsistent Initialization

2.3.1 Pseudocode

The pseudocode for the ComputePath function under overconsistent initialization is given in figure 2.13.

The pseudocode below assumes the following (*Assumption A*):

1. key function satisfies *key-requirement 1*: for any two states $s, s' \in S$ if $c^*(s', s_{\text{goal}}) < \infty$, $v(s') \geq g(s')$, $v(s) > g(s)$ and $g(s') > g(s) + \epsilon * c^*(s, s')$, then $\text{key}(s') > \text{key}(s)$;

2. $v-$ and $g-$values of all states are initialized in such a way that $v(s) \geq g(s) = \min_{s' \in pred(s)}(v(s') + c(s', s))\, \forall s \in S - \{s_{\text{start}}\}$ and $v(s_{\text{start}}) \geq g(s_{\text{start}}) = 0$ (the initialization can also occur whenever ComputePath encounters new states);

3. initially, *CLOSED* $= \emptyset$ and *OPEN* contains exactly all and only overconsistent states (i.e., states s whose $v(s) > g(s)$).

```
1 procedure ComputePath()
2 while($\text{key}(s_{\text{goal}}) > \min_{s \in OPEN}(\text{key}(s))$)
3   remove $s$ with the smallest key($s$) from $OPEN$;
4   $v(s) = g(s)$; $CLOSED \leftarrow CLOSED \cup \{s\}$;
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($s'$);
```

Figure 2.13: Generic search that expands overconsistent states

2.3.2 Notation

A state s is called *inconsistent* iff $v(s) \neq g(s)$, *overconsistent* iff $v(s) > g(s)$, *underconsistent* iff $v(s) < g(s)$ and *consistent* iff $v(s) = g(s)$. $g^*(s)$ denotes the cost of a shortest path from s_{start} to s. We require that $1 \leq \epsilon < \infty$.

Let us also recall the definition of a *greedy path* from s_{start} to s. It is a path that is computed by tracing it backward as follows: start at s, 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 $s_{i-1} = s_{\text{start}}$.

We also assume that the min operation on an empty set returns ∞.

2.3.3 Proofs

In the first section we prove several lemmas about some of the more obvious properties of the search. In the section labeled "Main Theorems" we prove several theorems that constitute the main idea behind the algorithm. Finally, in the section "Correctness" we show how these theorems lead to the correctness of the ComputePath function. In the last section we also present few theorems relevant to the efficiency of the algorithm.

Low-level Correctness

Most of the lemmas in this section simply state the correctness of the program state variables such as v- and g-values, *OPEN*, and *CLOSED* sets. The lemmas also show that $g(s)$ is always an upper bound on the cost of a greedy path from s_{start} to s, and can never become smaller than the cost of a least-cost path from s_{start} to s, $g^*(s)$.

Lemma 7 *At any point in time for any state s, $v(s) \geq g(s)$.*

Proof: The theorem clearly holds before the ComputePath function is called for the first time according to Assumption A.2. Afterwards, the g-values can only decrease (lines 6-7). For any state s, on the other hand, $v(s)$ only changes on line 4 when it is set to $g(s)$. Thus, it is always true that $v(s) \geq g(s)$. ■

Lemma 8 *At line 2, $g(s_{\text{start}}) = 0$ and for $\forall s \neq s_{\text{start}}$, $g(s) = \min_{s' \in pred(s)}(v(s') + c(s', s))$.*

Proof: The theorem holds right after the initialization according to Assumption A.2. The only place where g- and v-values are changed afterwards is on lines 4 and 7. If $v(s)$ is changed in line 4, then it is decreased according to Lemma 7. Thus, it may only decrease the g-values of its successors. The test on line 6 checks this and updates the g-values if necessary. Since all costs are positive and never change, $g(s_{\text{start}})$ can never be changed: it will never pass the test on line 6, and thus is always 0. ■

Lemma 9 *At line 2, OPEN and CLOSED are disjoint, OPEN contains only inconsistent states and the union of OPEN and CLOSED contains all inconsistent states (and possibly others).*

Proof: The first time line 2 is executed the theorem holds according to Assumption A.3 and the fact that all inconsistent states are overconsistent according to Assumption A.2.

During the following execution whenever we decrease $g(s)$ (line 7), and as a result make s inconsistent (Lemma 7), we insert it into *OPEN* unless it is already in *CLOSED*; whenever we remove s from *OPEN* (line 3) we set $v(s) = g(s)$ (line 4) making the state consistent. We never add s to *CLOSED* while it is still in *OPEN*, and we never modify $v(s)$ or $g(s)$ elsewhere. ∎

Lemma 10 *Suppose s is selected for expansion on line 3. Then the next time line 2 is executed $v(s) = g(s)$, where $g(s)$ before and after the expansion of s is the same.*

Proof: Suppose s is selected for expansion. Then on line 4 $v(s) = g(s)$, and it is the only place where a v-value changes. We, thus, only need to show that $g(s)$ does not change. It could only change if $s \in succ(s)$ and $g(s) > g(s) + c(s, s)$. The second test, however, implies that $c(s, s) < 0$. This contradicts to our restriction that costs are positive. ∎

Lemma 11 *At line 2, for any state s, the cost of a greedy path from s_{start} to s is no larger than $g(s)$, and $v(s) \geq g(s) \geq g^*(s)$.*

Proof: $v(s) \geq g(s)$ holds according to Lemma 7. We thus need to show that the cost of a greedy path from s_{start} to s is no larger than $g(s)$, and $g(s) \geq g^*(s)$. The statement follows if $g(s) = \infty$. We thus can restrict our proof to a finite g-value.

Consider a greedy path from s_{start} to s: $s_0 = s_{\text{start}}, s_1, ..., s_k = s$. Then for any i, $k \geq i > 0$, we have $g(s_i) = \min_{s' \in pred(s_i)}(v(s') + c(s', s_i)) \geq \min_{s' \in pred(s_i)}(g(s') + c(s', s_i)) = g(s_{i-1}) + c(s_{i-1}, s_i)$ from Lemma 8, Lemma 7 and the definition of a greedy path. For $i = 0$, $g(s_i) = g(s_{\text{start}}) = 0$ from Lemma 8. Thus, $g(s) = g(s_k) \geq g(s_{k-1}) + c(s_{k-1}, s_k) \geq g(s_{k-2}) + c(s_{k-2}, s_{k-1}) + c(s_{k-1}, s_k) \geq ... \geq \sum_{j=1..k} c(s_{j-1}, s_j)$. That is, $g(s)$ is at least as large as the cost of the greedy path from s_{start} to s. Since the cost cannot be smaller than the cost of a least-cost path we also have $g(s) \geq g^*(s)$. ∎

Main theorems

We now prove two theorems which constitute our main results about the search function in figure 2.13. These theorems guarantee that the algorithm is ϵ-suboptimal (where ϵ is the minimum value for which assumption A.1 holds): when the algorithm finishes its processing, it has identified a set of states for which its cost estimates $g(s)$ are no more than a factor of ϵ greater than the optimal costs $g^*(s)$. Together with Lemma 11 this shows that given such cost estimates the greedy paths that ComputePath finds to these states are suboptimal by at most ϵ (we will prove these corollaries in the next section).

For $\epsilon = 1$, the ComputePath function is essentially equivalent to the A* (or Dijkstra's if zero heuristics are used) algorithm. For intuition, here is a very brief summary of how our proofs below would apply to A*: we would start by showing that the *OPEN* list always contains all inconsistent states. (These states are arranged in a priority queue ordered by their key values.) We say a state s is ahead of the *OPEN* list if $\text{key}(s) \leq \text{key}(u)$ for all $u \in OPEN$. We then prove by induction that states which are ahead of *OPEN* have already been assigned their correct optimal path length. The induction works because, when we expand the state at the head of the *OPEN* queue, its optimal path depends only on states which are already ahead of the *OPEN* list.

The proofs for ComputePath with an arbitrary ϵ are somewhat more complicated than for A* because the *OPEN* list may not contain all inconsistent states. (Some of these states may be in *CLOSED* because they have already been expanded.) Therefore, we will introduce the set Q instead:

Definition 1 $Q = \{u \mid v(u) > g(u) \wedge v(u) > \epsilon * g^*(u)\}$

This set contains all inconsistent states except those whose v-values (and consequently g-values) are already within a factor of ϵ of their true costs.

The set Q takes the place of the *OPEN* list in the next theorem. In particular, Theorem 12 says that all states which are ahead of Q have their g-values within a factor of ϵ of optimal. Theorem 13 builds on this result by showing that *OPEN* is always a superset of Q, and therefore the states which are ahead of *OPEN* are also ahead of Q.

Theorem 12 *At line 2, let Q be defined according to the definition 1. Then for any state s with ($c^*(s, s_{\text{goal}}) < \infty \wedge key(s) \leq key(u)\ \forall u \in Q$), it holds that $g(s) \leq \epsilon * g^*(s)$.*

Proof: We prove by contradiction. Suppose there exists an s such that $c^*(s, s_{\text{goal}}) < \infty \wedge key(s) \leq key(u)\ \forall u \in Q$, but $g(s) > \epsilon * g^*(s)$. The latter implies that $g^*(s) < \infty$. We also assume that $s \neq s_{\text{start}}$ since otherwise $g(s) = 0 = \epsilon * g^*(s)$ from Lemma 8.

Consider a least-cost path from s_{start} to s, $\pi(s_0 = s_{\text{start}}, ..., s_k = s)$. The cost of this path is $g^*(s)$. Such path must exist since $g^*(s) < \infty$. Our assumption that $g(s) > \epsilon * g^*(s)$ means that there exists at least one $s_i \in \pi(s_0, ..., s_{k-1})$, namely s_{k-1}, whose $v(s_i) > \epsilon * g^*(s_i)$. Otherwise,

$$\begin{aligned} g(s) = g(s_k) = \min_{s' \in pred(s)} (v(s') + c(s', s_k)) &\leq \\ v(s_{k-1}) + c(s_{k-1}, s_k) &\leq \\ \epsilon * g^*(s_{k-1}) + c(s_{k-1}, s_k) &\leq \\ \epsilon * (g^*(s_{k-1}) + c(s_{k-1}, s_k)) = \epsilon * g^*(s_k) &= \epsilon * g^*(s) \end{aligned}$$

Let us now consider $s_i \in \pi(s_0, ..., s_{k-1})$ with the smallest index $i \geq 0$ (that is, the closest to s_{start}) such that $v(s_i) > \epsilon * g^*(s_i)$. We will first show that $\epsilon * g^*(s_i) \geq g(s_i)$. It is clearly so when $i = 0$ according to Lemma 8 which says that $g(s_i) = g(s_{\text{start}}) = 0$. For $i > 0$ we use the fact that $v(s_{i-1}) \leq \epsilon * g^*(s_{i-1})$ from the way s_i was chosen,

$$\begin{aligned} g(s_i) = \min_{s' \in pred(s_i)} (v(s') + c(s', s_i)) &\leq \\ v(s_{i-1}) + c(s_{i-1}, s_i) &\leq \\ \epsilon * g^*(s_{i-1}) + c(s_{i-1}, s_i) &\leq \\ \epsilon * g^*(s_i) & \end{aligned}$$

We thus have $v(s_i) > \epsilon * g^*(s_i) \geq g(s_i)$, which also implies that $s_i \in Q$.

We will now show that key$(s) >$ key(s_i), and finally arrive at a contradiction. According to our assumption

$$\begin{aligned} g(s) > \epsilon * g^*(s) &= \\ \epsilon * (c^*(s_0, s_i) + c^*(s_i, s_k)) &= \\ \epsilon * g^*(s_i) + \epsilon * c^*(s_i, s_k) &\geq \\ g(s_i) + \epsilon * c^*(s_i, s) & \end{aligned}$$

Hence, we have $g(s) > g(s_i) + \epsilon * c^*(s_i, s)$, $v(s_i) > \epsilon * g^*(s_i) \geq g(s_i)$, $v(s) \geq g(s)$ from Lemma 7 and $c^*(s, s_{\text{goal}}) < \infty$ from theorem conditions. Thus, from Assumption A.1 it follows that key$(s) >$ key(s_i). This inequality, however, implies that $s_i \notin Q$ since $\text{key}(s) \leq \text{key}(u)\ \forall u \in Q$. But this contradicts to what we have proven earlier. ■

Theorem 13 *At line 2, for any state s with $(c^*(s, s_{\text{goal}}) < \infty \land key(s) \leq key(u)\ \forall u \in OPEN)$, it holds that $g(s) \leq \epsilon * g^*(s)$.*

Proof: Let Q be defined according to the definition 1. To prove the theorem we will show that Q is a subset of *OPEN* and then appeal to Theorem 12.

From the definition of set Q it is clear that for any state $u \in Q$ it holds that u is inconsistent (that is, $v(u) \neq g(u)$).

According to the Assumption A.2 and A.3 when the ComputePath function is called *OPEN* contains all inconsistent states. Therefore $Q \subseteq OPEN$, because as we have just shown any state $u \in Q$ is also inconsistent. Thus, if any state s has $c^*(s, s_{\text{goal}}) < \infty \land \text{key}(s) \leq \text{key}(u)\ \forall u \in OPEN$, it is also true that $c^*(s, s_{\text{goal}}) < \infty \land \text{key}(s) \leq \text{key}(u)\ \forall u \in Q$, and $g(s) \leq \epsilon * g^*(s)$ from Theorem 12. Thus, the first time line 2 is executed the theorem holds.

Also, because during the first execution of line 2 $CLOSED = \emptyset$ according to assumption A.3, the following statement, denoted by (*), trivially holds when line 2 is executed for the first time: for any state $s \in CLOSED$ $v(s) \leq \epsilon * g^*(s)$.

We will now show by induction that the theorem continues to hold for the consecutive executions of the line 2. Suppose the theorem and the statement (*) held during all the previous executions of line 2, and they still hold when a state s is selected for expansion on line 3. We need to show that the theorem holds the next time line 2 is executed.

We first prove that the statement (*) still holds during the next execution of line 2. Since the v-value of only s is being changed and only s is being added to *CLOSED*, we only need to show that $v(s) \leq \epsilon * g^*(s)$ during the next execution of line 2 (that is, after the expansion of s). Since when s is selected for expansion on line 3 $\text{key}(s) = \min_{u \in OPEN}(\text{key}(u))$, we have $\text{key}(s) \leq \text{key}(u)\ \forall u \in OPEN$. According to the assumptions of our induction then $g(s) \leq \epsilon * g^*(s)$. From Lemma 10 it then also follows that the next time line 2 is executed $v(s) \leq \epsilon * g^*(s)$, and hence the statement (*) still holds.

We now prove that after s is expanded the theorem itself also holds. We prove this by showing that Q continues to be a subset of *OPEN* the next time line 2 is executed. According to Lemma 9 *OPEN* set contains all inconsistent states that are not in *CLOSED*. Since, as we have just proved, the statement

(*) holds the next time line 2 is executed, all states s in *CLOSED* set have $v(s) \leq \epsilon * g^*(s)$. Thus, any state s that is inconsistent and has $v(s) > \epsilon * g^*(s)$ is guaranteed to be in *OPEN*. Now consider any state $u \in Q$. As we have shown earlier such state u is inconsistent, and $v(u) > \epsilon * g^*(u)$ according to the definition of Q. Thus, $u \in$ *OPEN*. This shows that $Q \subseteq$ *OPEN*. Consequently, if any state s has $c^*(s, s_{\text{goal}}) < \infty \wedge \text{key}(s) \leq \text{key}(u)\ \forall u \in$ *OPEN*, it is also true that $c^*(s, s_{\text{goal}}) < \infty \wedge \text{key}(s) \leq \text{key}(u)\ \forall u \in Q$, and $g(s) \leq \epsilon * g^*(s)$ from Theorem 12. This proves that the theorem holds during the next execution of line 2, and proves the whole theorem by induction. ■

Correctness

The corollaries in this section show how the theorems in the previous section lead quite trivially to the correctness of ComputePath.

Corollary 14 *When the ComputePath function exits the following holds for any state s with $c^*(s, s_{\text{goal}}) < \infty \wedge key(s) \leq \min_{s' \in OPEN}(key(s'))$: the cost of a greedy path from s_{start} to s is no larger than $\epsilon * g^*(s)$.*

Proof: According to Theorem 13 the condition $c^*(s, s_{\text{goal}}) < \infty \wedge \text{key}(s) \leq \min_{s' \in OPEN}(\text{key}(s'))$ implies that $g(s) \leq \epsilon * g^*(s)$. The proof then follows by direct application of Lemma 11. ■

Corollary 15 *When the ComputePath function exits the following holds: the cost of a greedy path from s_{start} to s_{goal} is no larger than $\epsilon * g^*(s_{\text{goal}})$.*

Proof: According to the termination condition of the ComputePath function, upon its exit $\text{key}(s_{\text{goal}}) \leq \min_{s' \in OPEN}(\text{key}(s'))$. The proof then follows from Corollary 14 noting that $c^*(s_{\text{goal}}, s_{\text{goal}}) = 0$. ■

Efficiency

Several theorems in this section provide some theoretical guarantees about the efficiency of ComputePath.

Theorem 16 *No state is expanded more than once during the execution of the ComputePath function.*

Proof: Suppose a state s is selected for expansion for the first time during the execution of the ComputePath function. Then, it is removed from *OPEN* set on line 3 and inserted into *CLOSED* set on line 4. It can then never be inserted into *OPEN* set again unless the ComputePath function exits since any state that is about to be inserted into *OPEN* set is checked against *CLOSED* set membership on line 8. Because only the states from *OPEN* set are selected for expansion, s can therefore never be expanded second time. ■

Theorem 17 *A state s is expanded only if $v(s)$ is lowered during its expansion.*

Proof: Only the states from *OPEN* can be selected for expansion. Any such state is inconsistent according to Lemma 9. Moreover, for any such state s it holds that $v(s) > g(s)$ from Lemma 7. From Lemma 10 it then follows that $v(s)$ is set to $g(s)$ during the expansion of v and thus is lowered. ■

Theorem 18 *A state s is expanded only if it was already inconsistent before the call to ComputePath() or its g-value was lowered during the current execution of ComputePath().*

Proof: According to Lemma 9 any state s that is selected for expansion on line 3 is inconsistent. If a state s was already inconsistent before the call to ComputePath() then the theorem is immediately satisfied. If a state s was *not* inconsistent before the call to the ComputePath function, then its g- and v- values were equal. Since $v(s)$ can only be changed during the expansion of s, it must have been the case that $g(s)$ was changed, and the only way for it to change is to decrease on line 7. Thus, the g-value of s was lowered during the execution of ComputePath(). ■

2.4 Proofs for Anytime Repairing A*

2.4.1 Pseudocode

The pseudocode for ARA* is given in figures 2.14 and 2.15.

```
procedure ComputePath()
while(key($s_{goal}$) > $\min_{s \in OPEN}$(key($s$)))
  remove $s$ with the smallest key($s$) from $OPEN$;
  $v(s) = g(s)$; $CLOSED \leftarrow CLOSED \cup \{s\}$;
  for each successor $s'$ of $s$
    if $s'$ was never visited by ARA* before then
      $v(s') = g(s') = \infty$;
    if $g(s') > g(s) + c(s, s')$
      $g(s') = g(s) + c(s, s')$;
      if $s' \notin CLOSED$
        insert/update $s'$ in $OPEN$ with key($s'$);
      else
        insert $s'$ into $INCONS$;
```

Figure 2.14: The ComputePath function as used by ARA*. The changes specific to ARA* are shown in bold.

The pseudocode below assumes the following (*Assumption B*):

1. Heuristics need to be consistent: $h(s) \le c(s, s') + h(s')$ for any successor s' of s if $s \neq s_{goal}$ and $h(s) = 0$ if $s = s_{goal}$.

```
procedure key($s$)
return $g(s) + \epsilon * h(s)$;

procedure Main()
$g(s_{goal}) = v(s_{goal}) = \infty$; $v(s_{start}) = \infty$;
$g(s_{start}) = 0$; $OPEN = CLOSED = INCONS = \emptyset$;
insert $s_{start}$ into $OPEN$ with key($s_{start}$);
ComputePath();
publish current $\epsilon$-suboptimal solution;
while $\epsilon > 1$
  decrease $\epsilon$;
  Move states from $INCONS$ into $OPEN$;
  Update the priorities for all $s \in OPEN$ according to key($s$);
  $CLOSED = \emptyset$;
  ComputePath();
  publish current $\epsilon$-suboptimal solution;
```

Figure 2.15: ARA* algorithm

2.4.2 Notation

The notation is exactly the same as we have used in section 2.3. We re-emphasize that ϵ is restricted to finite values larger than or equal to 1.

2.4.3 Proofs

The changes that we have introduced into the ComputePath function (lines 6, 7, 12 and 13 in figure 2.14) do not affect the properties that we have already proven to hold for the ComputePath function in section 2.3, assuming that every time the function is called assumption A holds. Lines 12 and 13 are purely keeping track of states that are both inconsistent and not in *OPEN*. This maintenance does not affect the operation of ComputePath in any way. Lines 6 and 7, on the other hand, are used to do the online initialization of states that have not been seen before. To make the following proofs easier to read we therefore from now on assume that any state s with undefined values (not visited) has $v(s) = g(s) = \infty$.

The goal of the proofs in this section is to show that ARA* algorithm as presented in figure 2.15 ensures that the assumption A is true every time it calls the ComputePath function. Once this is shown, all the properties in section 2.3 apply here and we thus obtain the desired properties about each search iteration in ARA* including its ϵ-suboptimality.

The proofs are based on induction. They begin by showing that assumption A holds the first time the ComputePath function is called (line 7 in figure 2.15). The proofs then continue by showing that if assumption A is satisfied before calling the ComputePath function, then upon its exit moving states from *INCONS* into *OPEN* (line 11) and resetting *CLOSED* (line 13) is sufficient to restore the conditions of assumption A again. These operations are performed before each subsequent execution of the ComputePath function (line 7). By induction, the assumption A thus holds before each call to ComputePath, which makes all of the properties proven in section 2.3 applicable to each execution of the ComputePath function in figure 2.15.

Lemma 19 *For any pair of states s and s', $\epsilon * h(s) \leq \epsilon * c^*(s, s') + \epsilon * h(s')$.*

Proof: According to [66] the consistency property required of heuristics in Assumption B is equivalent to the restriction that $h(s) \leq c^*(s, s') + h(s')$ for *any* pair of states s, s' and $h(s_{\text{goal}}) = 0$. The theorem then follows by multiplying the inequality with ϵ. ■

Theorem 20 *If assumption A holds and INCONS $= \emptyset$ before the execution of ComputePath, then during the execution of ComputePath, at line 2, OPEN and INCONS are disjoint and INCONS contains exactly all the inconsistent states which are also in CLOSED.*

Proof: We will prove the theorem by assuming that assumption A holds and $INCONS = \emptyset$ before the execution of ComputePath. The first time line 2 is executed *OPEN* contains all inconsistent states according to assumption A.2 and A.3, $CLOSED = \emptyset$ according to assumption A.3 and $INCONS = \emptyset$. Thus, the statement of the theorem is not violated.

During the following execution of ComputePath whenever we remove s from *OPEN* (line 3) we set $v(s) = g(s)$ (line 4) making the state consistent and therefore adding this state to *CLOSED* (line 4) does not require us to add this state to *INCONS*; whenever we decrease $g(s)$ (line 9), and as a result make s inconsistent (Lemma 7), we insert it into either *OPEN* if it is not in *CLOSED* or *INCONS* if it is already in *CLOSED*. We never do any other operations that would affect the membership of a state in *CLOSED*, *OPEN* or *INCONS* or an inconsistency of a state. ■

Theorem 21 *Every time the ComputePath function is called from Main function of ARA*, assumption A is fully satisfied prior to the execution of ComputePath.*

Let us first prove assumption A.1. Consider two arbitrary states s' and s such that $c^*(s', s_{\text{goal}}) < \infty$, $v(s') \geq g(s')$, $v(s) > g(s)$ and $g(s') > g(s) + \epsilon * c^*(s, s')$. We need to show that these conditions imply $\text{key}(s') > \text{key}(s)$. Given the definition of the key() function in figure 2.15 we need to show that $g(s') + \epsilon * h(s') > g(s) + \epsilon * h(s)$. We examine the inequality $g(s') > g(s) + \epsilon * c^*(s, s')$ and add $\epsilon * h(s')$ on both sides. Because $h(s') \leq c^*(s', s_{\text{goal}}) < \infty$ due to heuristics consistency we now have $g(s') + \epsilon * h(s') > g(s) + \epsilon * c^*(s, s') + \epsilon * h(s')$ and from Lemma 19 we obtain the desired $g(s') + \epsilon * h(s') > g(s) + \epsilon * h(s)$.

Now, before we prove assumptions A.2 and A.3 let us first prove the following statement denoted by (*): every time the ComputePath function is called $INCONS = \emptyset$. This is true the first time ComputePath is called since *INCONS* is reset to empty on line 5. Then the statement holds because before each subsequent call to ComputePath all the states in *INCONS* are moved into *OPEN* on line 11.

We are now ready to prove that assumptions A.2 and A.3 hold before each execution of ComputePath by induction. Consider the first call to the ComputePath function (line 7). At that point the g- and v-values of all states except for s_{start} are infinite, and $v(s_{start}) = \infty$ and $g(s_{start}) = 0$. Thus, for every state $s \neq s_{\text{start}}$, $v(s) = g(s) = \min_{s' \in pred(s)}(v(s') + c(s', s))$ and for $s = s_{\text{start}}$ $v(s) > g(s) = 0$. Consequently, assumption A.2 holds.

Additionally, the only overconsistent state is s_{start} which is inserted into *OPEN* at line 6. *OPEN* does not contain any other states and *CLOSED* is empty. Hence, assumption A.3 is also satisfied.

Suppose now that the assumptions A.2 and A.3 held during the previous calls to the ComputePath functions (we have already shown that assumption A.1 holds before *each* call to ComputePath). We will now show that the assumptions A.2 and A.3 continue to hold the next time ComputePath is called and thus hold every time ComputePath is called by induction.

After ComputePath exits, according to theorems 7 and 8, for every state s, $v(s) \geq g(s)$ and $g(s) = \min_{s' \in pred(s)}(v(s') + c(s', s))$ for $s \neq s_{\text{start}}$ and $g(s) = 0$ for $s = s_{\text{start}}$. Since in the Main function of ARA* we do not change v- or g-values, the assumption A.2 continues to hold the next time the ComputePath function is called.

Since assumption A holds and $INCONS = \emptyset$ (statement (*)) right before the call to ComputePath, Theorem 20 applies and we conclude that at the time ComputePath exits *INCONS* contains exactly all inconsistent states that are in *CLOSED*. Combined with the Lemma 9 this implies that *INCONS* contains all and only inconsistent states that are not in *OPEN*. Thus, by moving states from *INCONS* into *OPEN* on line 11 we force *OPEN* to contain exactly all inconsistent states. Also, from Theorem 7 it holds that the only inconsistent states are overconsistent states. Thus, the next time ComputePath is executed, *OPEN* contains exactly all overconsistent states while *CLOSED* is empty (due to line 13). Hence, assumption A.3 holds the next time ComputePath is executed. ■

2.5 Summary

In this chapter we have presented an anytime heuristic search algorithm, ARA*, that works by continually decreasing a suboptimality bound on its solution and finding new solutions that satisfy the bound on the way. The anytime property of the algorithm is achieved by initially setting the desired suboptimality bound large and thus finding the solution quickly, and then slowly decreasing the desired suboptimality bound and finding new improved solutions until the available planning time runs out. Each planning episode in ARA* is an A*-like search. Differently from A* search, however, each search in ARA* tries to reuse as much as possible of the results from previous searches.

The key idea that enabled us to develop an A*-like search capable of reusing results from previously executed searches was an observation that A* search can be viewed as a repetitive expansion of overconsistent states. In this chapter we therefore first introduced a notion of a state being overconsistent. We then showed how to re-formulate A* search as a repetitive expansion of overconsistent states. Afterwards, all that became necessary for this search to reuse the results from previous search executions was to identify all overconsistent states and make them candidates for expansion before its execution. We showed how to do this rather simply in the context of ARA*.

In our experiments the efficiency of these search iterations allow ARA* to decrease suboptimality bounds and find cheaper solutions on the way much faster than any of the previously developed anytime searches that could also provide suboptimality bounds on their solutions. In addition, the ability to control suboptimality bounds in ARA* allows one to tune its anytime behavior. For example, in some domains one can execute ARA* with gradually decreasing bounds, while in others one might only be interested in obtaining an optimal solution once a suboptimal one is known.

Chapter 3

LPA*: Incremental A* Search

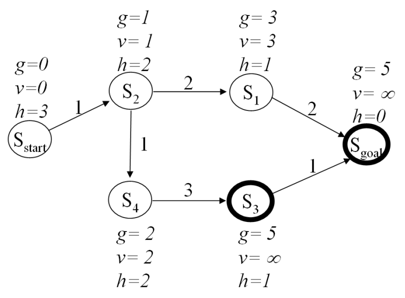
This chapter presents an algorithm that is suitable for planning in domains where the model of a problem changes over time. In such domains the presented algorithm gains efficiency by being able to re-use its previous planning efforts. The model of a problem may change because the environment is dynamic, for instance, if a robot is navigating in a building populated with people or a car route is being re-generated under changing traffic conditions. The model of a problem may also change because it initially is fully or partially unknown and more information about it is gathered during the execution of a plan, for instance, if a robot is navigating with a partially unknown map and updates the map while executing its plan whenever it senses obstacles in front of it.

3.1 The Effect of Cost Changes

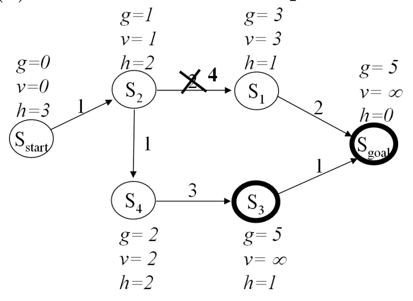
So far, we have only considered situations where the ComputePath function was executed multiple times for a different value of ϵ used to inflate heuristics, but always on the same graph. Since the value of ϵ affects purely the ordering of expansions, changing ϵ does not affect the status of a state. In particular, no state s would become underconsistent (i.e., $v(s) < g(s)$) if it was either consistent (i.e., $v(s) = g(s)$) or overconsistent (i.e., $v(s) > g(s)$) before ϵ was changed. Hence, when constructing the ARA* algorithm, the only thing that we really have to worry about before each call to ComputePath is that *OPEN* contains all inconsistent states and *CLOSED* is empty. All the other assumptions the ComputePath function makes (listed in figure 2.6) remain true within the search itself and remain satisfied outside of the ComputePath function as well. These assumptions are that the function key() satisfies its restriction, no state is underconsistent, and all g-values are exactly one step look-ahead values based on the v-values of the predecessors. Function key() of ARA* stays always the same, and function Main() of ARA* does not change g-values, v-values or edge costs, it only changes ϵ.

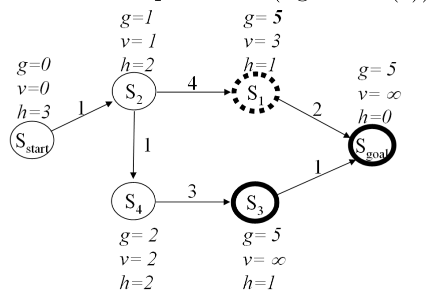
The situation changes, however, when in between two calls to the ComputePath function some edge costs in the graph change. First, to satisfy the requirement that all g-values are one step look-ahead values based on the v-values of the predecessors, that is, for any $s \in S$, $g(s) = \min_{s' \in pred(s)}(v(s') + c(s', s))$, we need to update the g-values of states for which the costs of incoming edges have changed. Let us first consider the simpler scenario when edge costs can only decrease. Then, as we update the g-values of these states, they can only decrease. This means that if a state s was not underconsistent (in other words, s had $v(s) \geq g(s)$) 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 in figure 2.6 will still be satisfied if we decrease some edge costs in the graph and update the g-values correspondingly. This way we can construct a simple incremental search for the case of decreasing edge costs.

The case of increasing edge costs is harder, however. As we update the g-values of states whose incoming edge costs increased, they can increase now. As such they may become larger than the corresponding v-values, and states may become underconsistent (i.e., $v(s)$ may become smaller than $g(s)$). An example demonstrating this is shown in figure 3.1. The initial state values are the same as in figure 2.7 after the ComputePath 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 an underconsistent state violates the assumption that no state is underconsistent before a call to the ComputePath function. This is the main issue this chapter addresses in the next section.



(a) state values after the previous execution of ComputePath (figure 2.7(c))





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

Figure 3.1: 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.

3.2 Single-shot Search under Arbitrary Initialization

3.2.1 An Offline Approach to Correcting Initialization

In this section we show a way to restore the property that no state is underconsistent before the ComputePath function is called. In the next section, however, we will show how to incorporate this process of fixing states into the ComputePath function itself and how to do it only for the states that need to be fixed rather than all underconsistent states.

One way to restore the property that no state is underconsistent before the ComputePath function is called is to set the v-value of every underconsistent state s to ∞. This forces the state to stop being underconsistent (recall that an underconsistent state has $v(s) < g(s)$), but may possibly affect the g-values of the successors of s. As we update their g-values so that they remain to be one-step look-ahead values based on the predecessor v-values, their g-values may now increase. As a result, these states may become underconsistent themselves. If so, we will need to fix their v-values as well. We can continue doing this until no state is underconsistent. The simple pseudocode that does this is shown in figure 3.2.

```
1 procedure FixInitialization()
2 while(there is an underconsistent state)
3   let $s$ be an underconsistent state (i.e., $v(s) < g(s)$);
4   $v(s) = \infty$;
5   for each successor $s'$ of $s$
6     $g(s') = \min_{s'' \in pred(s')} v(s'') + c(s'', s')$;
```

Figure 3.2: A code that forces all states to become either consistent or overconsistent

We process states until there are no underconsistent states left. For every underconsistent state we pick (line 3), we first force it to become either consistent or overconsistent (line 4). We then make sure the g-values of all of its successors are updated to be consistent with the v-values of their predecessors (line 6).

The pseudocode is a simple way of preprocessing the states that ensures the proper state value initialization before the ComputePath function is called. The computational expense of this pre-processing step, though, can become a burden. On the positive side, this pseudocode processes each state only once (once a v-value is set to ∞, the state cannot become underconsistent again) and the processing is limited to only the states that have been expanded in previous searches (otherwise their v-values are infinite). The last property, perhaps, is even more important because it ensures that the pre-processing step will not require more memory than what is currently used. On the negative side, the preprocessing step may still be inefficient in cases when many of these underconsistent states are no longer relevant to the computation of a plan. This can be the case, when the agent moves and changes in edge costs in the regions of the state space where agent used to be and no longer currently is are unlikely to be relevant to the computation of a current plan. In the next section we will incorporate this pre-processing step into the ComputePath function, so that it becomes online processing and fixes only the states that are relevant to the computation of a plan from the current state of the agent.

3.2.2 An Online Approach to Correcting Initialization

When A* search expands any overconsistent state s' there exists at least one path from s_{start} to s' whose cost is at most ϵ times $c^*(s_{\text{start}}, s')$ and all of the states on which have their v- and g-values no more than ϵ times their g^*-values. Hence, when we expand s', $g(s')$ is at worst ϵ-suboptimal since it is equal to $\min_{s'' \in pred(s')}(v(s'') + c(s'', s'))$, and when we set its v-value to its g-value the v-value of s' also becomes at worst ϵ-suboptimal. Now that some states can be initially underconsistent we need to make sure that when an overconsistent s' is being expanded no state on which a path to s' can depend on is underconsistent. In other words, we need to make sure that all the states that can possibly belong to the currently best path from s_{start} to s are pre-processed so that they are not underconsistent. For example, we can pre-process all such states in the same way it was done in the previous section, namely, set their v-values to ∞ and update their successors (lines 4 through 6 in figure 3.2)).

The pseudocode that achieves this is shown in figure 3.3. Notice that its second assumption no longer requires that no state is underconsistent. The first change is that we make this step of setting v-value to ∞ and updating the successors of a state to be an expansion of an underconsistent state (lines 15 through 17 in figure 3.3). This also means that *OPEN*, the list of candidates for expansions, should contain both underconsistent and overconsistent states. We have moved the process of deciding on the membership in *OPEN* into its own function UpdateSetMembership, which inserts an inconsistent state into *OPEN* unless it was already expanded as overconsistent (i.e., in *CLOSED* set) and removes a state from *OPEN* if it is consistent. This function is called every time a g- or v-value is modified except for line 10, where s is consistent and was just removed from *OPEN* anyway. Also, initially *OPEN* needs to contain *all* inconsistent states, independently of whether they are overconsistent or underconsistent (the third assumption in figure 3.3).

The pseudocode below assumes the following:

1. key function satisfies the following restrictions: for any two states $s, s' \in S$ (a) if $c^*(s', s_{\text{goal}}) < \infty$, $v(s') \geq g(s')$, $v(s) > g(s)$ and $g(s') > g(s) + \epsilon * c^*(s, s')$, then key$(s')$ > key(s), and (b) if $c^*(s', s_{\text{goal}}) < \infty$, $v(s') \geq g(s')$, $v(s) < g(s)$ and $g(s') \geq v(s) + c^*(s, s')$, then key$(s')$ > key(s);

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 $s \in S - \{s_{\text{start}}\}$ $g(s) = \min_{s' \in pred(s)}(v(s') + c(s', s))$ (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     $v(s) = g(s)$; $CLOSED \leftarrow CLOSED \cup \{s\}$;
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       $g(s') = \min_{s'' \in pred(s')} v(s'') + c(s'', s')$; UpdateSetMembership($s'$);
```

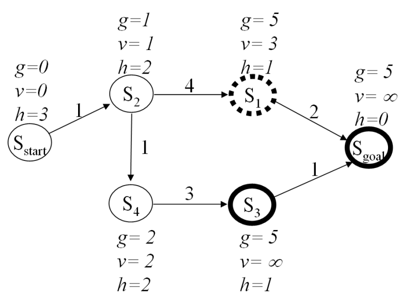
Figure 3.3: ComputePath function that expands both overconsistent and underconsistent states

The second change takes care of making sure that all underconsistent states which can belong to a path from s_{start} to s' are expanded before s' is expanded. This constraint is specified using an additional restriction on the function key() (case (b) in the first assumption in figure 3.3). This additional restriction can be "translated" as follows. Given an underconsistent state s and an overconsistent or consistent state s' that can potentially belong to a path from s_{start} to s_{goal} (i.e., $c^*(s', s_{\text{goal}}) < \infty$), a currently found 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, or in other words, 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 currently best path from s_{start} to s', the property we wanted to achieve.

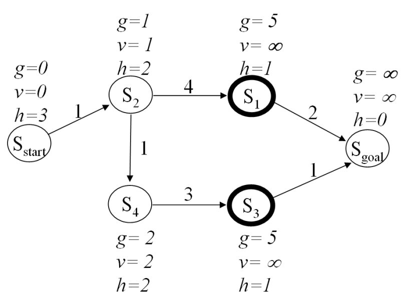
The last small modification we have to make is the terminating condition. We need to make sure that s_{goal} itself is not underconsistent when the search terminates. The second part of the terminating condition, namely, $v(s_{\text{goal}}) < g(s_{\text{goal}})$ ensures that the search forces the process of expansions until s_{goal} is either consistent or overconsistent.

Figure 3.4 demonstrates the operation of the ComputePath function when some states are initially underconsistent. The initial state values are the same as in figure 3.1, where we have changed the cost $c(s_2, s_1)$ after the ComputePath function was already executed. This small example imitates the conditions under which LPA* will be executing the ComputePath function: it will execute the ComputePath function repeatedly, updating edge costs in between. The prioritization function according to which states are chosen for expansion is $\min(g(s), v(s)) + h(s)$ with ties broken towards smaller $\min(g(s), v(s))$. That is, $key(s) = [\min(g(s), v(s)) + h(s); \min(g(s), v(s))]$ and $key(s) > key(s')$ iff $\min(g(s), v(s)) + h(s) > \min(g(s'), v(s')) + h(s')$ or $\min(g(s), v(s)) + h(s) = \min(g(s'), v(s')) + h(s')$ and $\min(g(s), v(s)) > \min(g(s'), v(s'))$. This function satisfies the key function restrictions listed in figure 3.3.

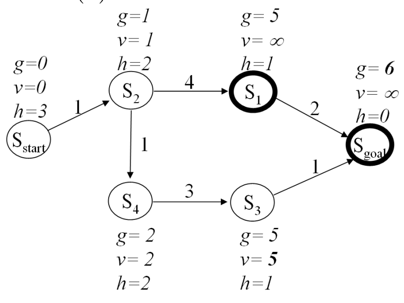
While the pseudocode in figure 3.3 is correct, there remains one significant optimization. The re-evaluation of g-values on line 17 is an expensive operation as it requires us to iterate over all predecessors of s'. We can decrease the number of times this re-evaluation is done if we notice that it is invoked when state s is expanded as underconsistent and therefore its v-value is increased to ∞. Therefore, only a successor of s whose g-value depends on s can be affected. If $s \neq \arg\min_{s'' \in pred(s')} v(s'') + c(s'', s')$ before its expansion, then setting $v(s)$ to ∞ cannot change $g(s')$ and we therefore don't need to re-evaluate $g(s')$. To implement this test we simply maintain back-pointers,



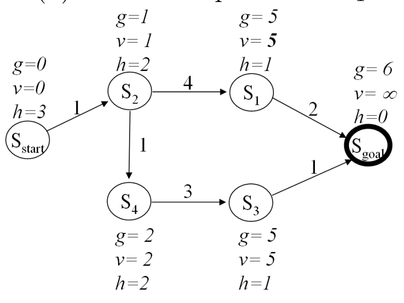
(a) initial state values



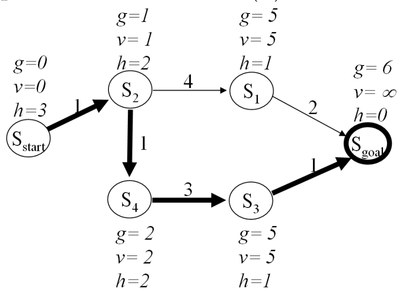
(b) after the expansion of s_1



(c) after the expansion of s_3



(d) after the expansion of s_1



(e) after the computation of a greedy path

Figure 3.4: An example of how the ComputePath function operates under an arbitrary initialization. The example uses the following prioritization function: $key(s) = [\min(g(s), v(s)) + h(s); \min(g(s), v(s))]$. 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 what regular A* search would have generated (provided it broke ties in a certain manner when selecting states with the same f-values for expansion).

so that for any state $s' \in S$

$$bp(s') = \arg \min_{s'' \in pred(s')} v(s'') + c(s'', s'). \tag{3.1}$$

The pseudocode that maintains these back-pointers and uses them to avoid unnecessary re-evaluations of the g-values is given in figure 3.5 (changes are shown in bold). Back-pointers are updated whenever the g-values change (lines 14 and 20). In case of underconsistent state expansions the re-evaluations of the g-values are now only done for states whose back-pointer points to the state being expanded (lines 18 through 20).

The use of back-pointers also simplifies the re-construction of a solution after the search finishes. Recall that previously the solution was given by a greedy path that was computed 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). Now, as we prove in section 3.4 the solution can also be re-constructed using back-pointers, tracing it backwards as follows: start at s_{goal}, and at any state s_i pick a state $s_{i-1} = bp(s_i)$ until $s_{i-1} = s_{\text{start}}$. We will refer to the path re-constructed in this way as the path defined by back-pointers.

3.3 Incremental Search

3.3.1 LPA*

Now, that we have a version of the ComputePath function that can handle an arbitrary initialization of states, that is, states that are consistent, overconsistent and underconsistent, it is a simple exercise to construct an incremental heuristic search algorithm. We call this search Lifelong Planning A* (LPA*) [48][1]. The pseudocode is given in figures 3.6 and 3.7. The code for the ComputePath function is essentially the same as what we have presented in the previous section (the differences are shown in bold). The only differences are that first, we maintain *INCONS* list to keep track of all inconsistent states (lines 4 and 7 in figure 3.6) so that we can restore *OPEN* to contain

[1]The version presented in [48] is an incremental heuristic search that is restricted to finding optimal solutions only. The version we present here is a generalization in that it can trade-off the optimality for computational savings. The algorithm follows closely the generalized LPA* algorithm developed in [59].

The pseudocode below assumes the following:

1. key function satisfies the following restrictions: for any two states $s, s' \in S$ (a) if $c^*(s', s_{\text{goal}}) < \infty$, $v(s') \geq g(s')$, $v(s) > g(s)$ and $g(s') > g(s) + \epsilon * c^*(s, s')$, then key$(s') >$ key(s), and (b) if $c^*(s', s_{\text{goal}}) < \infty$, $v(s') \geq g(s')$, $v(s) < g(s)$ and $g(s') \geq v(s) + c^*(s, s')$, then key$(s') >$ key(s);

2. $v-$, $g-$ and $bp-$ values of all states are initialized in such a way that all the v-values are non-negative, $bp(s_{\text{start}}) =$ **null**, $g(s_{\text{start}}) = 0$ and for every state $s \in S - \{s_{\text{start}}\}$ $bp(s) = \arg\min_{s' \in pred(s)}(v(s') + c(s', s))$ and $g(s) = v(bp(s)) + c(bp(s), s)$ (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($s_{goal}$) > $\min_{s \in OPEN}$(key($s$)) OR $v(s_{goal}) < g(s_{goal})$)
8    remove $s$ with the smallest key($s$) from $OPEN$;
9    if ($v(s) > g(s)$)
10     $v(s) = g(s)$; $CLOSED \leftarrow CLOSED \cup \{s\}$;
11     for each successor $s'$ of $s$
12       if $g(s') > g(s) + c(s, s')$
13         $\mathbf{bp(s') = s}$;
14         $g(s') = g(\mathbf{bp(s')}) + c(\mathbf{bp(s')}, s')$; UpdateSetMembership($s'$);
15   else //propagating underconsistency
16     $v(s) = \infty$; UpdateSetMembership($s$);
17     for each successor $s'$ of $s$
18       if $\mathbf{bp(s') = s}$
19         $\mathbf{bp(s') = \arg\min_{s'' \in pred(s')} v(s'') + c(s'', s')}$;
20         $g(s') = v(\mathbf{bp(s')}) + c(\mathbf{bp(s')}, s')$; UpdateSetMembership($s'$);
```

Figure 3.5: ComputePath function that expands both overconsistent and underconsistent states and uses back-pointers to avoid g-value re-computations.

all inconsistent states before each call to the ComputePath function, and second, we explicitly initialize the states that LPA* (not the ComputePath function) has not seen before on lines 14-15 and 22-23.

The function key() that LPA* uses is given in figure 3.7. It is straightforward to show that it satisfies the restriction on the key function (the first assumption in figure 3.5). One can also design other key functions that satisfy the restriction and are suited better for certain domains. For

```
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($s_{\text{goal}}$) > $\min_{s \in OPEN}$(key($s$)) OR $v(s_{\text{goal}}) < g(s_{\text{goal}})$)
  remove $s$ with the smallest key($s$) from $OPEN$;
  if ($v(s) > g(s)$)
    $v(s) = g(s)$; $CLOSED \leftarrow CLOSED \cup \{s\}$;
    for each successor $s'$ of $s$
      if $s'$ was never visited by LPA* before then
        $v(s') = g(s') = \infty$; $bp(s') =$ 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 LPA* before then
        $v(s') = g(s') = \infty$; $bp(s') =$ 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($s'$);
```

Figure 3.6: LPA*: ComputePath function. LPA* specific changes as compared with the search that can handle arbitrary state initialization (figure 3.5) are shown in bold.

example, in some cases it is more efficient to use buckets for maintaining $OPEN$. Setting $key(s) = [g(s) + \epsilon * h(s); 1]$ if s is not underconsistent and $key(s) = [v(s) + h(s); 0]$ otherwise is more suitable for this case as it will create much fewer buckets than the key function in figure 3.7 and will satisfy the required restriction. For other examples of possibly useful key functions see [59].

The main function Main() in the pseudo code of LPA* first initializes the variables so that the start state is initially the only inconsistent state and is inserted into the otherwise empty priority queue (line 9 in figure 3.7). Main() then calls ComputePlan() to find a path that is ϵ-suboptimal at worst. The path defined by back-pointers is then published as the current solution (line 12). (If $g(s_{start}) = \infty$ after the search, then there is no finite cost path from the start state to the goal state.) Main() then waits for changes in edge costs. If some edge costs have changed, Main() updates the corresponding *bp*- and *g*-values (lines 19 and 20) so that the second assumption of the ComputePath function (figure 3.5) is satisfied[2]. Before it calls the ComputePath function again, Main() makes *OPEN* contain all inconsistent states by moving the contents of the *INCONS* list into it. To fully satisfy the assumption of the ComputePath function it finally resets *CLOSED* (line 22). Main() then re-computes a path that is ϵ-suboptimal at worst by calling ComputePath() again, and iterates.

3.3.2 Theoretical Properties of LPA*

We now give several theoretical properties of LPA*. The proofs of these and other properties are given in section 3.5. Since the ComputePath function of LPA* is essentially the ComputePath function in figure 3.5 they both share the same theoretical properties. The first theorem is similar to the corresponding theorem about ARA*. It says that once ComputePath terminates we have computed a path that is at worst ϵ-suboptimal from s_{start} to every consistent and overconsistent state s that can be on a path from s_{start} to s_{goal} and whose key is no larger than the minimum key in *OPEN*.

Theorem 22 *When the ComputePath function exits the following holds for any state s with* $(c^*(s, s_{\text{goal}}) < \infty \wedge v(s) \geq g(s) \wedge key(s) \leq \min_{s' \in OPEN}(key(s')))$*:* $g^*(s) \leq g(s) \leq \epsilon * g^*(s)$*, and the cost of the path from* s_{start} *to s defined by backpointers is no larger than* $g(s)$*.*

Given that ComputePath terminates when the key of s_{goal} is no larger than the smallest key in *OPEN* and s_{goal} is either consistent or overconsistent, the path defined by backpointers from s_{start} to s_{goal} is also at worst ϵ-suboptimal. The next few theorems talk about the efficiency of LPA*. The first one says that each state can only be expanded at most twice. In ARA*, each state could be expanded only once. Here, a state can first potentially be pre-processed if it ever becomes underconsistent. Since we count this as an expansion, a state can be expanded at most twice: once as underconsistent and once as overconsistent.

[2]In problems that involve large number of edge cost updates one can avoid a large number of g-value re-computations. The optimization is based on the idea described at the end of section 3.2.2. It is given explicitly in [48].

The pseudocode below assumes the following:

1. Heuristics need to be consistent: $h(s) \le c(s, s') + h(s')$ for any successor s' of s if $s \neq s_{\text{goal}}$ and $h(s) = 0$ if $s = s_{\text{goal}}$.

```
procedure key($s$)
if ($v(s) \ge g(s)$)
  return $[g(s) + \epsilon * h(s); g(s)]$;
else
  return $[v(s) + h(s); v(s)]$;

procedure Main()
$g(s_{\text{goal}}) = v(s_{\text{goal}}) = \infty$; $v(s_{\text{start}}) = \infty$; $bp(s_{\text{goal}}) = bp(s_{\text{start}}) = \textbf{null}$;
$g(s_{\text{start}}) = 0$; $OPEN = CLOSED = INCONS = \emptyset$;
insert $s_{\text{start}}$ into $OPEN$ with key($s_{\text{start}}$);
forever
  ComputePath();
  publish $\epsilon$-suboptimal solution;
  wait for changes in edge costs;
  for all directed edges $(u, v)$ with changed edge costs
    update the edge cost $c(u, v)$;
    if $v \neq s_{\text{start}}$
      if $v$ was never visited by LPA* before then
        $v(v) = g(v) = \infty; bp(v) = \textbf{null}$;
      $bp(v) = \arg\min_{s'' \in pred(v)} v(s'') + c(s'', v)$;
      $g(v) = v(bp(v)) + c(bp(v), v)$; UpdateSetMembership($v$);
  Move states from $INCONS$ into $OPEN$;
  $CLOSED = \emptyset$;
```

Figure 3.7: LPA*: key and Main functions

Theorem 23 *No state is expanded more than twice during the execution of the ComputePath function: at most once as underconsistent and at most once as overconsistent.*

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

Theorem 24 *A state s is expanded by ComputePath only if either it was inconsistent initially or its v-value was altered by ComputePath at some point during its execution.*

The last theorem about the efficiency of LPA* states that if LPA* searches for an optimal solution (i.e., $\epsilon = 1$), then no state is expanded if its v-value was already correct, in other words, equal to the cost of a least-cost path from s_{start} to the state, before the ComputePath function was called.

Theorem 25 *If $\epsilon = 1$ then no state s which has a finite $c^*(s, s_{\mathrm{goal}})$ and which had $v(s)$ initialized to $g^*(s)$ before the ComputePath function was called is expanded by it.*

LPA* possesses a number of other interesting properties, in particular, the ones that relate it to A*. For these and other additional properties please refer to [49].

3.3.3 Extensions to LPA*

Moving Agent For many problems an agent may need to re-plan while executing the plan. For example, a robot navigating to its goal in a partially known environment gathers new information about its environment while navigating and this information alters the cost function. In these cases the state of the robot may change from one invocation of ComputePath to another. Just as we have described for ARA* in section 2.2.2, it is common to perform the search backwards, so that the start of the search remains the same and the search tree can be re-used. In this case heuristics estimate distances to the current state of the agent, and therefore change as the robot moves. This means that we need to re-order *OPEN* before each re-planning episode if the robot moved since the last time planning occurred. This can be done right before ComputePath is called in Main() (line 11, figure 3.7). The operation, however, can be expensive, especially, when optimal planning is performed. The work in [47] describes an algorithm, called D* Lite, that extends LPA* by avoiding the heap reordering based on an idea presented in [78].

Minimax LPA* LPA* applies to deterministic domains, that is, to domains where each action has only one possible outcome. In many domains however, an action may have more than one outcome due to either inherent randomness of the action or uncertainty in the model of the world. In [58] we develop a minimax version of LPA*, called Minimax LPA*, that solves nondeterministic domains. Minimax LPA* is an incremental planning algorithm that, instead of returning least-cost deterministic paths, returns plans that minimize the worst-case plan execution cost. Minimax LPA* proved to be highly beneficial in speeding up the Parti-game algorithm [62], a popular algorithm for finding feasible control policies in continuous and high-dimensional state spaces.

Generalizations of LPA* Our work in [59] generalizes LPA* in several ways. Most importantly, it presents alternative key functions that satisfy the key requirements outlined in figure 3.5. The alternative key functions may prove to be useful for certain domains. For example, [59] presents a key function that breaks ties among the candidates for expansions with the same f-values in order of states with larger g-values. This tie breaking criteria is important in domains where numerous optimal solutions exist and we want to avoid exploring all of them.

3.3.4 Experimental Analysis of the Performance of LPA*

In this section we present an analysis of the efficiency of LPA* on the problem of path planning for robot navigation in initially unknown environments. The environments are modelled as gridworlds. We have used a version of LPA* that was optimized for moving agents and presented in [47]. We compared LPA* against several alternatives: complete uninformed search (breadth-first search), complete heuristic search (A*), incremental uninformed search (DynamicSWSF-FP [72] with a modified termination condition that stops each search iteration as soon as a shortest path to the goal state is found), and a different kind of incremental heuristic search (D* [78]). We implemented all priority queues using standard binary heaps, although using more complex data structures (such as Fibonacci heaps) could possibly make U.Update() more efficient. A* broke ties among cells with the same f-value in favor of cells with larger g-values, which tends to be more efficient than the opposite way of breaking ties. Since all of these search methods move the robot in the same way, we only need to perform a simulation study in which we compare the total planning times of the search methods. Since the actual planning times are implementation and machine dependent, we also use two measures that both correspond to common operations performed by the search methods and thus heavily influence their planning times, yet are implementation and machine independent: the total number of cell expansions and the total number of heap percolates (exchanges of a parent and child in the heap).

Search Algorithm	Planning Time	Cell Expansions	Heap Percolates
Breadth-First Search	302.30 msecs	845,433	4,116,516
A*	10.55 msecs	17,096	276,287
Dynamic SWSF-FP	6.41 msecs	13,962	75,738
(Focussed) D*	4.28 msecs	2,138	79,214
LPA*	2.82 msecs	2,856	32,988

Figure 3.8: Experimental Results – Terrain with Random Obstacles.

Search Algorithm	Planning Time	Cell Expansions	Heap Percolates
Breadth-First Search	194.13 msecs	543,408	2,643,916
A*	5.49 msecs	8,680	156,801
Dynamic SWSF-FP	6.26 msecs	13,931	76,703
(Focussed) D*	1.18 msecs	596	19,066
LPA*	0.97 msecs	393	5,316

Figure 3.9: Experimental Results – Fractal Terrain.

We performed experiments on eight-connected grids of size 129×129, where the start cell of the robot was $(x = 12, y = 12)$ and the goal cell was $(x = 116, y = 116)$. We report the averages over 500 runs on randomly generated grids. All experiments were run on a 1.9 GHz PC under Linux.

In one set of experiments, we used grids whose cells were either traversable or, with forty percent probability, untraversable, where untraversable cells were modeled as cells with no incoming or outgoing edges. The robot initially assumed that all edges were present with cost one. The robot always deleted those edges entering its neighboring cells that did not exist in the true model and then replanned a shortest path from its current cell to the goal cell. We used the maximum of the absolute differences of the x and y coordinates of any two cells as a heuristic estimate of their distances. Table 3.8 compares LPA* against the other search methods. It shows that incremental heuristic searches outperform both incremental and heuristic searches individually, and that LPA* is competitive with D*.

In the second set of experiments, we used fractal terrain, similar to the one used in [79]. All pairs of adjacent cells were connected by an edge but its cost varied from 5 to 14 according to the traversal difficulty of the cell it was entering. The robot initially assumed that all edges were present with cost 5. The robot always updated the cost of the edges entering its neighboring cells to correspond to the traversal difficulty of the corresponding neighboring cell and then replanned a shortest path from its current cell to the goal cell. We used five times of the maximum of the absolute differences of the x and y coordinates of any two cells as a heuristic estimate of their distances. Table 3.9 presents the same comparison as in the previous experiment. The conclusions of the previous experiment continue to hold in these grids, that might be more realistic models of outdoor terrain.

3.4 Proofs for Single-shot Search under Arbitrary Initialization

3.4.1 Pseudocode

The pseudocode for the ComputePath function under arbitrary initialization is given in figure 3.10.

3.4.2 Notation

The notation includes the one in section 2.3 plus few additional ones that concern the new variable, $bp(s)$, that we have introduced. $bp(s)$ is a backpointer that points towards a predecessor state of s that lies on the currently computed path from s_{start} to s.

To make the following proofs easier to read we assume that $\arg\min$ operation, used to compute backpointer values, when called on the set consisting of only infinite values returns **null**, and in the computation $g(s) = v(bp(s)) + c(bp(s), s)$ it is assumed that if $bp(s) =$ **null** then $g(s)$ is set to ∞.

Throughout the proofs we sometimes refer to a path from s_{start} to s defined by backpointers. This path is defined as a path that is computed by tracing it backward as follows: start at s, and at any state s_i pick a state $s_{i-1} = bp(s_i)$ until $s_{i-1} = s_{\text{start}}$.

We want to remind that ϵ is restricted to finite values larger than or equal to 1.

The Assumption C, required to execute the ComputePath function in figure 3.10, is an equivalent of the Assumption A that was required to execute ComputePath when no underconsistent states were allowed (figure 2.13). In particular, C.1 extends A.1 by adding an additional requirement on the computation of keys in cases when a state is underconsistent (C.1(b)), C.2 extends A.2 by allowing to have underconsistent states initially, and C.3 extends A.3 by making *OPEN* to contain all inconsistent states rather than all oveconsistent ones as it was in A.3.

The pseudocode below assumes the following (*Assumption C*):

1. key function satisfies *key-requirement 2*: for any two states $s, s' \in S$ (a) if $c^*(s', s_{\text{goal}}) < \infty$, $v(s') \geq g(s')$, $v(s) > g(s)$ and $g(s') > g(s) + \epsilon * c^*(s, s')$, then $\text{key}(s') > \text{key}(s)$, and (b) if $c^*(s', s_{\text{goal}}) < \infty$, $v(s') \geq g(s')$, $v(s) < g(s)$ and $g(s') \geq v(s) + c^*(s, s')$, then $\text{key}(s') > \text{key}(s)$;

2. $v-$, $g-$ and $bp-$ values of all states are initialized in such a way that all the v-values are non-negative, $bp(s_{\text{start}}) = \textbf{null}$, $g(s_{\text{start}}) = 0$ and for every state $s \in S - \{s_{\text{start}}\}$ $bp(s) = \arg\min_{s' \in pred(s)}(v(s') + c(s', s))$ and $g(s) = v(bp(s)) + c(bp(s), s)$ (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 $v(s) \neq g(s)$).

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

procedure ComputePath()
while(key($s_{\text{goal}}$) > $\min_{s \in OPEN}$(key($s$)) OR $v(s_{\text{goal}}) < g(s_{\text{goal}})$)
  remove $s$ with the smallest key($s$) from $OPEN$;
  if ($v(s) > g(s)$)
    $v(s) = g(s)$; $CLOSED \leftarrow CLOSED \cup \{s\}$;
    for each successor $s'$ of $s$
      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 $bp(s') = s$
        $bp(s') = \arg\min_{s'' \in pred(s')} v(s'') + c(s'', s')$;
        $g(s') = v(bp(s')) + c(bp(s'), s')$; UpdateSetMembership($s'$);
```

Figure 3.10: ComputePath function that is able to handle underconsistent states.

3.4.3 Proofs

The structure of the proofs is similar to the structure we used in section 2.3.3. The first section mostly proves that the ComputePath function in figure 3.10 correctly maintains its variables. The section "Main Theorems" is more interesting and proves the key properties of the algorithm. The section labeled "Correctness" uses these key properties to quite trivially derive the

ϵ-suboptimality property of the algorithm. Finally, the last section proves several properties regarding the efficiency of the algorithm.

Low-level Correctness

Most of the theorems in this section simply state the correctness of the program state variables such as v-, g-values and bp-values, and *OPEN* and *CLOSED* sets.

Lemma 26 *At line 7, all v- and g-values are non-negative, $bp(s_{\text{start}}) = \textbf{null}$, $g(s_{\text{start}}) = 0$ and for $\forall s \neq s_{\text{start}}$, $bp(s) = \arg\min_{s' \in pred(s)}(v(s') + c(s', s))$, $g(s) = v(bp(s)) + c(bp(s), s)$.*

Proof: The theorem holds right after the initialization according to Assumption C.2 and the fact that g-values can only be non-negative when all the v-values are non-negative and costs are positive. The only places where g-, v- and bp-values are changed afterwards are on lines 10, 13, 14, 16, 19, and 20.

If $v(s)$ is set to $g(s)$ on line 10, then it is decreased since s is being expanded as overconsistent (i.e., $v(s) > g(s)$ before the expansion according to the test on line 9). Thus, it may only decrease the g-values of its successors. The test on line 12 checks this for each successor of s and updates the bp- and g-values if necessary. Since all costs are positive and never change, $bp(s_{\text{start}})$ and $g(s_{\text{start}})$ can never be changed: it will never pass the test on line 12, and thus is always 0. Since $v(s)$ is set to $g(s)$ it still remains non-negative. Consequently, when the g-values of the successors of s are re-calculated their g-values also remain non-negative.

If $v(s)$ is set to ∞ on line 16, then it either stays the same or increases since it is the "else" case of the test on line 9 (i.e., $v(s) \leq g(s)$ before the expansion of s). (As we will show in later theorems the inequality will always be strict as no consistent state is ever expanded.) Thus, it may only affect the g- and $bp-$ values of the successors of s if their bp-value is equal to s. The test on line 18 checks this for each successor of s and re-computes the bp- and g-values if necessary. Since $bp(s_{\text{start}}) = \textbf{null}$ the test will never pass for s_{start} and therefore $bp(s_{\text{start}})$ and $g(s_{\text{start}})$ can never be changed. Since $v(s)$ is set to ∞ it remains non-negative. Consequently, when the g-values of the successors of s are re-calculated their g-values also remain non-negative. The theorem thus holds. ■

Lemma 27 *At line 7, OPEN and CLOSED are disjoint, OPEN contains only inconsistent states and the union of OPEN and CLOSED contains all inconsistent states (and possibly others).*

Proof: The first time line 7 is executed the theorem holds according to Assumption C.3.

During the following execution whenever we remove s from *OPEN* (line 8) it is either made consistent on line 10 and its addition to *CLOSED* on the same line is therefore valid, or its v-value is set to ∞ on line 16 but its set membership is immediately updated on the same line by calling UpdateSetMembership function. This function makes sure that the state it is called on is consistent with the theorem.

The g-values of states could also be modified on lines 14 and 20, but the set membership of these states is updated immediately by calling UpdateSetMembership function on the same lines. ■

Lemma 28 *Suppose an overconsistent state s is selected for expansion in line 8. Then the next time line 7 is executed $v(s) = g(s)$, where $g(s)$ before and after the expansion of s is the same.*

Proof: Suppose an overconsistent state s (i.e., $v(s) > g(s)$) is selected for expansion. Then line 10 sets $v(s) = g(s)$, and it is the only place where a v-value changes while expanding an overconsistent state. We, thus, only need to show that $g(s)$ does not change. It could only change if $s \in succ(s)$ and $g(s) > g(s) + c(s, s)$ (test on line 12) which is impossible since all costs are positive. ■

Main theorems

We now prove two main theorems about the ComputePath function in figure 3.10. These theorems guarantee the ϵ-suboptimality of the algorithm (where ϵ is the minimum value for which assumption C.1 holds): when the algorithm finishes its processing, it has identified a set of states s for which a path from s_{start} to s as defined by backpointers is guaranteed to be of cost no larger than $g(s)$ which in turn is no more than a factor of ϵ greater than the optimal cost $g^*(s)$.

Just like in the proofs for the ComputePath function that deals only with overconsistent states we introduce here the set Q again but now it additionally contains all underconsistent states:

Definition 2 $Q = \{u \mid v(u) < g(u) \vee (v(u) > g(u) \wedge v(u) > \epsilon * g^*(u))\}$

In other words, the set Q contains all underconsistent states and all those overconsistent states whose v-values are larger than a factor of ϵ of their true costs. As we will show later *OPEN* is always a superset of set Q.

Theorem 29 *At line 7, let Q be defined according to the definition 2. Then for any state s with ($c^*(s, s_{\text{goal}}) < \infty \wedge v(s) \geq g(s) \wedge key(s) \leq key(u)$ $\forall u \in Q$), it holds that (i) $g(s) \leq \epsilon * g^*(s)$, (ii) the cost of the path from s_{start} to s defined by backpointers is no larger than $g(s)$.*

Proof: (i) We first prove statement (i). We prove by contradiction. Suppose there exists an s such that $(c^*(s, s_{\text{goal}}) < \infty \wedge v(s) \geq g(s) \wedge key(s) \leq key(u)\ \forall u \in Q)$, but $g(s) > \epsilon * g^*(s)$. The latter implies that $g^*(s) < \infty$. We also assume that $s \neq s_{\text{start}}$ since otherwise $g(s) = 0 = \epsilon * g^*(s)$ from Lemma 26.

Consider a least-cost path from s_{start} to s, $\pi(s_0 = s_{\text{start}}, ..., s_k = s)$. The cost of this path is $g^*(s)$. Such path must exist since $g^*(s) < \infty$. We will now show that such path must contain a state s' that is overconsistent and whose v-value overestimates $g^*(s')$ by more than ϵ. As a result, such state is a member of Q. We will then show that its key, on the other hand, must be strictly smaller than the key of s. This, therefore, becomes a contradiction since s according to the theorem assumptions has a key smaller than or equal to the key of any state in Q.

Our assumption that $g(s) > \epsilon * g^*(s)$ means that there exists at least one $s_i \in \pi(s_0, ..., s_{k-1})$, namely s_{k-1}, whose $v(s_i) > \epsilon * g^*(s_i)$. Otherwise,

$$\begin{aligned} g(s) = g(s_k) = \min_{s' \in pred(s)} (v(s') + c(s', s_k)) &\leq \\ v(s_{k-1}) + c(s_{k-1}, s_k) &\leq \\ \epsilon * g^*(s_{k-1}) + c(s_{k-1}, s_k) &\leq \\ \epsilon * (g^*(s_{k-1}) + c(s_{k-1}, s_k)) = \epsilon * g^*(s_k) &= \epsilon * g^*(s) \end{aligned}$$

Let us now consider $s_i \in \pi(s_0, ..., s_{k-1})$ with the smallest index $i \geq 0$ (that is, the closest to s_{start}) such that $v(s_i) > \epsilon * g^*(s_i)$. We will first show that $\epsilon * g^*(s_i) \geq g(s_i)$. It is clearly so when $i = 0$ according to Lemma 26 which says that $g(s_i) = g(s_{\text{start}}) = 0$. For $i > 0$ we use the fact that $v(s_{i-1}) \leq \epsilon * g^*(s_{i-1})$ from the way s_i was chosen,

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

$$\begin{aligned} v(s_{i-1}) + c(s_{i-1}, s_i) &\leq \\ \epsilon * g^*(s_{i-1}) + c(s_{i-1}, s_i) &\leq \\ \epsilon * g^*(s_i) & \end{aligned}$$

We thus have $v(s_i) > \epsilon * g^*(s_i) \geq g(s_i)$, which also implies that $s_i \in Q$.

We will now show that key$(s) >$ key(s_i), and finally arrive at a contradiction. According to our assumption

$$\begin{aligned} g(s) > \epsilon * g^*(s) &= \\ \epsilon * (c^*(s_0, s_i) + c^*(s_i, s_k)) &= \\ \epsilon * g^*(s_i) + \epsilon * c^*(s_i, s_k) &\geq \\ g(s_i) + \epsilon * c^*(s_i, s) & \end{aligned}$$

Hence, we have $g(s) > g(s_i) + \epsilon * c^*(s_i, s)$, $v(s_i) > \epsilon * g^*(s_i) \geq g(s_i)$, $v(s) \geq g(s)$ and $c^*(s, s_{\text{goal}}) < \infty$ from theorem assumptions. Thus, from Assumption C.1(a) it follows that key$(s) >$ key(s_i). This inequality, however, implies that $s_i \notin Q$ since key$(s) \leq$ key(u) $\forall u \in Q$. But this contradicts to what we have proven earlier.

(ii) Let us now prove statement (ii). We assume that $g(s) < \infty$ for otherwise the statement holds trivially. Suppose we start following the backpointers starting at s. We need to show that we will reach s_{start} at the cumulative cost of the transitions less than or equal to $g(s)$ (we assume that if we encounter a state with *bp*-value equal to **null** before s_{start} is reached then the cumulative cost is infinite).

We first show that we are guaranteed not to encounter an underconsistent state or a state with *bp*-value equal to **null** before s_{start} is reached. Once we have proven this property, we will be able to show that the cost of the path is bounded above by $g(s)$ simply from the fact that at each backtracking step in the path the g-value can only be larger than or equal to the sum of the g-value of the state the backpointer points to and the cost of the transition. Consequently, the g-value can never underestimate the cost of the remaining part of the path. The property that we are guaranteed not to encounter an underconsistent state or a state with *bp*-value equal to **null** before s_{start} is reached is based on the fact that any such state will have a key strictly smaller than the key of s or have an infinite g-value. The first case is impossible because key(s) is smaller than or equal to the key of any state in Q and this set already contains all underconsistent states. The second case can also be shown to be impossible quite trivially.

We thus first prove by contradiction the property that we are guaranteed not to encounter an underconsistent state or a state with *bp*-value equal to **null** before s_{start} is reached while following backpointers from s to s_{start}. Suppose the sequence of backpointer transitions leads us through the states $\{s_0 = s, s_1, \ldots, s_i\}$ where s_i is the first state that is either underconsistent or has $bp(s_i) = \textbf{null}$ (or both). It could not have been state s since $v(s) \geq g(s)$ from the assumptions of the theorem and $g(s) < \infty$ implies $bp(s) \neq \textbf{null}$ according to Lemma 26 (except when $s = s_{\text{start}}$ in which case the theorem holds trivially). We now show that s_i cannot be underconsistent. Since all the states before s_i are *not* underconsistent and have defined backpointer values we have $g(s) = v(s_1)+c(s_1, s) \geq g(s_1)+c(s_1, s) = v(s_2)+c(s_2, s_1)+c(s_1, s) \geq \ldots \geq v(s_i) + \sum_{k=1..i} c(s_k, s_{k-1}) \geq v(s_i) + c^*(s_i, s)$. If s_i was underconsistent, then we would have had $c^*(s, s_{\text{goal}}) < \infty$, $v(s) \geq g(s)$, $v(s_i) < g(s_i)$ and $g(s) \geq v(s_i)+c^*(s_i, s)$, and the Assumption C.1(b) would imply that $key(s) > key(s_i)$ which means that $s_i \notin Q$ and therefore cannot be underconsistent according to the definition of Q. We will now show that $bp(s_i)$ cannot be equal to **null** either. Since s_i is not underconsistent $v(s_i) \geq g(s_i)$. From our assumption that $g(s) < \infty$ and the fact that $g(s) \geq v(s_i) + c^*(s_i, s)$ it then follows that $g(s_i)$ is finite. As a result, from Lemma 26 $bp(s_i) \neq \textbf{null}$ unless $s_i = s_{\text{start}}$. Hence, as we backtrack from s to s_{start} the path defined by backpointers we are guaranteed to have states that are not underconsistent and whose *bp*-values are not equal to **null** except for s_{start}.

We are now ready to show that the cost of the path from s_{start} to s defined by backpointers is no larger than $g(s)$. Let us denote such path as: $s_0 = s_{\text{start}}, s_1, ..., s_k = s$. Since all states on this path are either consistent or overconsistent and their *bp*-values are defined (except for s_{start}), for any i, $k \geq i > 0$, we have $g(s_i) = v(s_{i-1}) + c(s_{i-1}, s_i) \geq g(s_{i-1}) + c(s_{i-1}, s_i)$ from Lemma 26. For $i = 0$, $g(s_i) = g(s_{\text{start}}) = 0$ from the same theorem. Thus, $g(s) = g(s_k) \geq g(s_{k-1}) + c(s_{k-1}, s_k) \geq g(s_{k-2}) + c(s_{k-2}, s_{k-1}) + c(s_{k-1}, s_k) \geq ... \geq \sum_{j=1..k} c(s_{j-1}, s_j)$. That is, $g(s)$ is at least as large as the cost of the path from s_{start} to s as defined by backpointers. ■

Theorem 30 *At line 7, for any state s with $(c^*(s, s_{\text{goal}}) < \infty \wedge v(s) \geq g(s) \wedge key(s) \leq key(u)\ \forall u \in OPEN)$, it holds that (i) $g(s) \leq \epsilon * g^*(s)$, (ii) the cost of the path from s_{start} to s defined by backpointers is no larger than $g(s)$.*

Proof: Let Q be defined according to the definition 2. To prove the theorem we will show that Q is a subset of *OPEN* and then appeal to Theorem 29. We will show that Q is a subset of *OPEN* by induction. We will first show that it holds initially because *OPEN* contains all inconsistent states initially and set Q is a subset of those. Afterwards, we will show that any state $s \in \mathit{CLOSED}$ always remains either consistent or overconsistent but with $v(s) \leq \epsilon * g^*(s)$. Given that the union of *OPEN* and *CLOSED* contains all inconsistent states, it is then clear that *OPEN* contains at least all those (and possibly other) inconsistent states that are in Q.

We now prove the theorem. From the definition of set Q it is clear that for any state $u \in Q$ it holds that u is inconsistent (that is, $v(u) \neq g(u)$).

According to the Assumption C.3 when the ComputePath function is called *OPEN* contains all inconsistent states. Therefore $Q \subseteq \mathit{OPEN}$, because as we have just said any state $u \in Q$ is also inconsistent. Thus, if any state s has $\text{key}(s) \leq \text{key}(u)\ \forall u \in OPEN$, it is also true that $\text{key}(s) \leq \text{key}(u)$ $\forall u \in Q$. Thus, from the direct application of Theorem 29 it follows that the first time line 7 is executed the theorem holds.

Also, because during the first execution of line 7 $\mathit{CLOSED} = \emptyset$ according to assumption C.3, the following statement, denoted by (*), trivially holds when line 7 is executed for the first time: for any state $v \in \mathit{CLOSED}$ it holds that $g(v) \leq v(v) \leq \epsilon * g^*(v)$ and $g(v) < v(s') + c^*(s', v)\ \forall s' \in \{s'' \mid v(s'') < g(s'')\}$. We will later prove that this statement always holds and thus all states $v \in \mathit{CLOSED}$ are either consistent or overconsistent but ϵ-suboptimal (i.e., $v(v) \leq \epsilon * g^*(v)$).

We will now show by induction that the theorem continues to hold for the consecutive executions of the line 7. Suppose the theorem and the statement (*) held during all the previous executions of line 7, and they still hold when a state s is selected for expansion on line 8. We need to show that the theorem holds the next time line 7 is executed.

We first prove that the statement (*) still holds during the next execution of line 7. Suppose first we select an overconsistent state s to be expanded. Because it is added to *CLOSED* immediately afterwards, we need to show that it does not violate statement (*). Since when s is selected for expansion on line 8 $\text{key}(s) = \min_{u \in OPEN}(\text{key}(u))$, we have $\text{key}(s) \leq \text{key}(u)\ \forall u \in$ *OPEN*. According to the assumptions of our induction then $g(s) \leq \epsilon * g^*(s)$. From Lemma 28 it then also follows that the next time line 7 is executed $g(s) = v(s) \leq \epsilon * g^*(s)$. To show that $g(s) < v(s') + c^*(s', s)\, \forall s' \in \{s'' \mid v(s'') < g(s'')\}$ after s is expanded we show that this is true right before s is expanded and therefore since the v-values of all states except for s do not change during the expansion of s and $g(s)$ does not change either (Lemma 28) it still holds afterwards. To show that the inequality held before the expansion of s we note that according to our assumptions *CLOSED* contained no underconsistent states and they were all therefore in *OPEN* (Lemma 27); from the way s was selected from *OPEN* it then followed that $\text{key}(s) \leq \text{key}(s') \, \forall s' \in \{s'' \mid v(s'') < g(s'')\}$; finally, the fact that s was overconsistent $(v(s) > g(s))$ implies that $g(s) < v(s') + c^*(s', s) \, \forall s' \in \{s'' \mid v(s'') < g(s'')\}$ because otherwise $c^*(s, s_{\text{goal}}) < \infty, v(s) > g(s)$, $v(s') < g(s')$ and $g(s) \geq v(s') + c^*(s', s)$ would imply $\text{key}(s) > \text{key}(s')$ according to the Assumption C.1.(b). As for the rest of the states the statement (*) follows from the following observations: only v-value of s was changed and s is not underconsistent after its expansion (it is in fact consistent according to Lemma 28); since $g(s)$ decreased during the expansion of s the g-values of its successors could only decrease implying that they could not have violated the statement (*); and finally no other changes to either v- or g-values were done and no operations except for insertion of s were done on *CLOSED*.

Suppose now an underconsistent state s is selected for expansion. Because it is not added to *CLOSED*, we only need to show that statement (*) remains to hold true for all the states that were in *CLOSED* prior to the expansion of s. Since only v-value of s has been changed, none of the v-values of states in *CLOSED* are changed. We will now show that none of their g-values could have changed either. Since prior to the expansion of s, s was underconsistent and statement (*) held by our induction assumptions, it was true that for any state $v \in$ *CLOSED*, $g(v) < v(s) + c^*(s, v)$. This means that $bp(v) \neq s$ (Lemma 26) and therefore the test on line 18 will not pass and $g(v)$ will not change during the expansion of s. Finally, we will now show that the newly introduced underconsistent states could not have violated the statement (*) either. The v-values of states that were underconsistent before s was expanded were not changed (v-value of only s was changed and s could not remain underconsistent as its v-value was set to ∞). Suppose some state s' became underconsistent as a result of expanding s. We need to show that after the expansion of s, for any state $v \in$ *CLOSED* it holds that $g(v) < v(s') + c^*(s', v)$. Since s' became underconsistent as a result of expanding s it must be the case that before the expansion of s $v(s') \geq g(s')$ and $bp(s') = s$ (in order for $g(s')$ to change). Consequently before the expansion of s, $v(s') \geq g(s') = v(s) + c(s, s')$. Since before the expansion of s statement (*) held, for any state $v \in$ *CLOSED* $g(v) < v(s) + c^*(s, v)$. We thus had $g(v) < v(s) + c^*(s, v) \leq v(s) + c(s, s') + c^*(s', v) \leq v(s') + c^*(s', v)$. This continues to hold after the expansion of s since neither $v(s')$ nor $g(v)$ changes during the expansion of s as we have just shown. Hence the statement (*) continues to hold the next time line 7 is executed.

We now prove that after s is expanded the theorem itself also holds. We prove it by showing that Q continues to be a subset of *OPEN* the next time line 7 is executed. According to Lemma 27 *OPEN* set contains all inconsistent states that are not in *CLOSED*. Since, as we have just proved, the statement (*) holds the next time line 7 is executed, all states s in *CLOSED* set have $g(s) \leq v(s) \leq \epsilon * g^*(s)$. Thus, any state s that is inconsistent and has either $g(s) > v(s)$ or $v(s) > \epsilon * g^*(s)$ (or both) is guaranteed to be in *OPEN*. Now consider any state $u \in Q$. As we have shown earlier such state u is inconsistent, and either $g(u) > v(u)$ or $v(u) > \epsilon * g^*(u)$ (or both) according to the definition of Q. Thus, $u \in$ *OPEN*. This shows that $Q \subseteq$ *OPEN*. Consequently, if any state s has $c^*(s, s_{\text{goal}}) < \infty \wedge v(s) \geq g(s) \wedge \text{key}(s) \leq \text{key}(u)\ \forall u \in$ *OPEN*, it is also true that $c^*(s, s_{\text{goal}}) < \infty \wedge v(s) \geq g(s) \wedge \text{key}(s) \leq \text{key}(u)\ \forall u \in Q$, and the statement of the theorem holds from Theorem 29. This proves that the theorem holds during the next execution of line 7, and proves the whole theorem by induction. ■

Correctness

The corollaries in this section show how the theorems in the previous section lead quite trivially to the correctness of ComputePath that can handle underconsistent states (figure 3.10).

Corollary 31 *When the ComputePath function exits the following holds for any state s with $(c^*(s, s_{\text{goal}}) < \infty \wedge v(s) \geq g(s) \wedge key(s) \leq \min_{s' \in OPEN}(key(s')))$: the cost of the path from s_{start} to s defined by backpointers is no larger than $\epsilon * g^*(s)$.*

Proof: The corollary follows directly from Theorem 30 after we combine the statements (i) and (ii) of the theorem. ■

Corollary 32 *When the ComputePath function exits the following holds: the cost of the path from s_{start} to s_{goal} defined by backpointers is no larger than $\epsilon * g^*(s_{\text{goal}})$.*

Proof: According to the termination condition of the ComputePath function, upon its exit $(v(s_{\text{goal}}) \geq g(s_{\text{goal}}) \wedge \text{key}(s_{\text{goal}}) \leq \min_{s' \in OPEN}(\text{key}(s')))$. The proof then follows from Corollary 31 noting that $c^*(s_{\text{goal}}, s_{\text{goal}}) = 0$. ■

Efficiency

Several theorems in this section provide some theoretical guarantees about the efficiency of ComputePath in figure 3.10.

Theorem 33 *Once a state is expanded as overconsistent it can never be expanded again (independently of it being overconsistent or underconsistent).*

Proof: Suppose a state s is selected for expansion as overconsistent for the first time during the execution of the ComputePath function. Then, it is removed from *OPEN* set on line 8 and inserted into *CLOSED* set on line 10. It can then never be inserted into *OPEN* set again unless the ComputePath function exits since any state that is about to be inserted into *OPEN* set is checked against membership in *CLOSED* on line 3. Because only the states from *OPEN* set are selected for expansion, s can therefore never be expanded second time. ■

Theorem 34 *No state is expanded more than once as underconsistent during the execution of the ComputePath function.*

Proof: Once a state is expanded as underconsistent its v-value is set to ∞. As a result, unless the state is expanded as overconsistent this state can never become underconsistent again. This is so because for a state to be underconsistent it needs to have its v-value strictly less than its g-value, which implies that the v-value needs to be finite. The only way for a v-value to change its value onto a finite value, on the other hand, is during the expansion of the state as an overconsistent state. However, if the state is expanded as overconsistent then according to Theorem 33 the state is never expanded again. Thus, a state can be expanded at most once as underconsistent. ■

Corollary 35 *No state is expanded more than twice during the execution of the ComputePath function.*

Proof: According to theorems 33 and 34 each state can be expanded at most once as underconsistent and at most once as overconsistent. Since there are no other ways to expand states, this leads to the desired result: each state is expanded at most twice. ■

Theorem 36 *A state s was expanded by ComputePath only if either it was inconsistent initially or its v-value was altered by ComputePath at some point during its execution.*

Proof: Let us pick a state s such that right before a call to the ComputePath function it was consistent and during the execution of ComputePath its v-value has never been altered. Then it means that $v_{afterComputePath}(s) = v_{beforeComputePath}(s) = g_{beforeComputePath}(s)$. Since only states from *OPEN* are selected for expansion and *OPEN* contains only inconsistent states, then in order for s to have been selected for expansion, it should have had $v(s) \neq g(s)$. Because the v-value of s remains the same throughout the ComputePath function execution, it has to be the case that the g-value of s has changed since the beginning of ComputePath. If s is expanded as overconsistent then $v(s)$ is changed by setting it to $g(s)$, whereas if s is expanded as underconsistent then $v(s)$ is increased by setting it to ∞ (it could not have been equal to ∞ before since it was underconsistent, i.e., $v(s) < g(s) \leq \infty$). Both cases contradict to our assumption that $v(s)$ remained the same throughout the execution of ComputePath. ■

In order to state the next theorem we first need to introduce an additional property that the function key() may often satisfy:

Definition 3 key-requirement 3*: for any two states $s, s' \in S$ if $c^*(s', s_{\text{goal}}) < \infty$, $v(s') < g(s')$, $v(s) > g(s)$ and $v(s') > g(s) + c^*(s, s')$, then $key(s') > key(s)$.*

This key-requirement essentially says that given an underconsistent state s' and an overconsistent state s we need to expand s first (i.e., key$(s') >$ key(s)) whenever the expansion of s (setting $v(s) = g(s)$) may potentially decrease $g(s')$ (as the g-values of the successors of s are being updated) and correct its underconsistency. That is, if $v(s') > g(s) + c^*(s, s')$, then it may potentially be possible that during the expansion of s we set $g(s') = v(s) +$

$c^*(s,s') = g(s) + c^*(s,s')$ and therefore state s' ceases to be underconsistent. The next theorem makes use of it, and shows that if the function key() does satisfy this property, then ComputePath can guarantee not to expand the states whose v-values are already correct when searching for an optimal path (i.e., $\epsilon = 1$).

Theorem 37 *If key-requirement 3 holds and Assumption C.1 is satisfied for $\epsilon = 1$ then no state s which has a finite $c^*(s, s_{\text{goal}})$ and which had $v(s)$ initialized to $g^*(s)$ before a call to ComputePath is expanded.*

Proof: We prove the theorem by contradiction. Suppose there is some state s with $c^*(s, s_{\text{goal}}) < \infty$, whose v-value before ComputePath was executed was equal to $g^*(s)$, and which was expanded at least once during the execution of the ComputePath function. Let us consider the first expansion of s. We will show that at that point there must have been another state which was inconsistent and had a key strictly smaller than the key of s. Moreover, since $\epsilon = 1$, it will turn out that any inconsistent state must be in *OPEN* which will lead us to a contradiction since s was selected for expansion as a state with the smallest key among the states in *OPEN*.

First, let us consider the case when s is expanded as overconsistent. Then right before it is selected for expansion on line 8, $c^*(s, s_{\text{goal}}) < \infty$, $v(s) > g(s)$ and $\text{key}(s) \leq \text{key}(u)\,\forall u \in$ *OPEN*. Hence, according to Theorem 30 it holds that the cost of the path from s_{start} to s as defined by backpointers is no larger than $g(s)$. This implies that $g(s) \geq g^*(s)$. Because the v-values of states are only changed when states are expanded and initially $v(s) = g^*(s)$ we then have $g(s) \geq v(s)$. This contradicts to our assumption that s is expanded as overconsistent and rules out the possibility of s being expanded first as overconsistent.

Now let us consider the case when s is expanded as underconsistent. Then right before it is selected for expansion on line 8 $v(s) < g(s)$ and $\text{key}(s) \leq \text{key}(u)\,\forall u \in$ *OPEN*. The first inequality and the fact that before the expansion $v(s)$ is equal to $g^*(s)$ implies that $g^*(s)$ is finite. The same inequality also implies that $s \neq s_{\text{start}}$ from Lemma 26. Let us therefore consider an optimal path from s_{start} to s, $\pi(s_0 = s_{\text{start}}, ..., s_k = s)$. The cost of this path is $g^*(s) < \infty$.

Our assumption that $g(s) > v(s) = g^*(s)$ means that there exists at least one $s_i \in \pi(s_0, ..., s_{k-1})$, namely s_{k-1}, whose $v(s_i) > g^*(s_i)$. Otherwise,

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

$$\begin{aligned} v(s_{k-1}) + c(s_{k-1}, s_k) &\leq \\ g^*(s_{k-1}) + c(s_{k-1}, s_k) &= \\ g^*(s) &= v(s) \end{aligned}$$

Let us now consider $s_i \in \pi(s_0, ..., s_{k-1})$ with the smallest index $i \geq 0$ (that is, the closest to s_{start}) such that $v(s_i) > g^*(s_i)$. We will first show that $g^*(s_i) \geq g(s_i)$. This is clearly so when $i = 0$ according to Lemma 26 which says that $g(s_i) = g(s_{\text{start}}) = 0$. For $i > 0$ we use the fact that $v(s_{i-1}) \leq g^*(s_{i-1})$ from the way s_i was chosen,

$$\begin{aligned} g(s_i) = \min_{s' \in pred(s_i)} (v(s') + c(s', s_i)) &\leq \\ v(s_{i-1}) + c(s_{i-1}, s_i) &\leq \\ g^*(s_{i-1}) + c(s_{i-1}, s_i) &\leq \\ g^*(s_i) & \end{aligned}$$

We thus have $v(s_i) > g^*(s_i) \geq g(s_i)$. We will now show that $s_i \in$ *OPEN*. Since it is inconsistent then according to Lemma 27 s_i is either in *CLOSED* or *OPEN*. Suppose that $s_i \in$ *CLOSED*. In order for this to be true s_i must have been expanded as overconsistent. Since s_i is on an optimal path from s_{start} to s and $g * (s)$ and $c^*(s, s_{\text{goal}})$ are both finite, it then follows that $g * (s_i)$ and $c^*(s_i, s_{\text{goal}})$ are also both finite. Therefore, according to Theorem 30 right before s_i was selected for expansion as an overconsistent state $g(s_i) \leq \epsilon * g^*(s_i) = g^*(s_i)$. From the fact that during the expansion $v(s_i)$ is set to $g(s_i)$ and s_i is never expanded afterwards (Theorem 33), and consequently $v(s_i)$ is never changed, follows that $v(s_i) > g^*(s_i)$ is an impossible assumption for a state that is in *CLOSED*. Hence, $s_i \in$ *OPEN*.

We will now show that key$(s) >$ key(s_i), and finally arrive at a contradiction. According to our assumption

$$\begin{aligned} v(s) = g^*(s) &= \\ c^*(s_0, s_i) + c^*(s_i, s_k) &= \\ g^*(s_i) + c^*(s_i, s_k) &\geq \\ g(s_i) + c^*(s_i, s) & \end{aligned}$$

Hence, we have $c^*(s, s_{\text{goal}}) < \infty$, $v(s) > g(s_i) + c^*(s_i, s)$, $v(s_i) > g(s_i)$ and $v(s) < g(s)$ at the time s is selected for expansion. Since key-requirement 3 holds it then follows that key$(s) >$ key(s_i). This inequality, however, implies that $s_i \notin OPEN$ since otherwise s_i should have been selected for expansion instead of s. But this contradicts to what we have proven earlier, namely that s_i does indeed belong to *OPEN*. This rules out the possibility of having expanded s as underconsistent first. Since we have already proven that s could not have been expanded as overconsistent first s could not have been expanded altogether. ■

3.5 Proofs for Lifelong Planning A*

3.5.1 Pseudocode

The pseudocode for LPA* is given in figures 3.11 and 3.12.

3.5.2 Notation

The notation is exactly the same as we have used in section 3.4. We re-emphasize that ϵ is restricted to finite values larger than or equal to 1.

3.5.3 Proofs

The changes that we have introduced into the ComputePath function (all lines in bold in figure 3.11, i.e., lines 4, 7, 14, 15, 22 and 23) are equivalent to the changes we introduced into the ComputePath function when used by ARA*. Just like there, these changes do not affect the properties that we have already proven to hold for the ComputePath function in section 3.4.3, assuming that every time the function is called assumption C (figure 3.10) holds. Lines 4 and 7 are purely keeping track of states that are both inconsistent and not in *OPEN*. This maintenance does not affect the operation of ComputePath in any way. The other four lines, 14, 15, 22 and 23 are used to do online initialization of states that have not been seen before. For the sake of the proofs simplicity we therefore from now on assume that any state s with undefined values (not visited) has $v(s) = g(s) = \infty$ and $bp(s) = \mathbf{null}$.

Similarly to the proofs of ARA*, the goal of proofs in this section is to show that LPA* algorithm as presented in figure 3.12 ensures that the assumption C is true every time it calls the ComputePath function. Once this is shown, all the properties in section 3.4.3 apply here and we thus obtain

```
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($s_{\text{goal}}$) > $\min_{s \in OPEN}$(key($s$)) OR $v(s_{\text{goal}}) < g(s_{\text{goal}})$)
  remove $s$ with the smallest key($s$) from $OPEN$;
  if ($v(s) > g(s)$)
    $v(s) = g(s)$; $CLOSED \leftarrow CLOSED \cup \{s\}$;
    for each successor $s'$ of $s$
      if $s'$ was never visited by LPA* before then
        $v(s') = g(s') = \infty$; $bp(s') =$ 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 LPA* before then
        $v(s') = g(s') = \infty$; $bp(s') =$ 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($s'$);
```

Figure 3.11: ComputePath function as used by LPA*. Changes specific to LPA* are shown in bold.

the desired properties about each search iteration in LPA* including its ϵ-suboptimality and other properties regarding its efficiency. The flow of the proofs is very similar to the flow of the proofs for ARA* (section 2.4.3).

Lemma 38 *For any pair of states s and s', $\epsilon * h(s) \leq \epsilon * c^*(s, s') + \epsilon * h(s')$.*

Proof: According to [66] the consistency property required of heuristics in Assumption D is equivalent to the restriction that $h(s) \leq c^*(s, s') + h(s')$ for *any* pair of states s, s' and $h(s_{\text{goal}}) = 0$. The theorem then follows by multiplying the inequality with ϵ. ■

Theorem 39 *If assumption C holds and $INCONS = \emptyset$ before the execution of ComputePath, then during the execution of ComputePath, at line 9, OPEN*

The pseudocode below assumes the following (*Assumption D*):

1. Heuristics need to be consistent: $h(s) \leq c(s, s') + h(s')$ for any successor s' of s if $s \neq s_{\text{goal}}$ and $h(s) = 0$ if $s = s_{\text{goal}}$.

```
1  procedure key($s$)
2  if ($v(s) \geq g(s)$)
3    return $[g(s) + \epsilon * h(s); g(s)]$;
4  else
5    return $[v(s) + h(s); v(s)]$;

6  procedure Main()
7  $g(s_{\text{goal}}) = v(s_{\text{goal}}) = \infty$; $v(s_{\text{start}}) = \infty$; $bp(s_{\text{goal}}) = bp(s_{\text{start}}) =$ null;
8  $g(s_{\text{start}}) = 0$; $OPEN = CLOSED = INCONS = \emptyset$;
9  insert $s_{\text{start}}$ into $OPEN$ with key($s_{\text{start}}$);
10 forever
11   ComputePath();
12   publish $\epsilon$-suboptimal solution;
13   wait for changes in edge costs;
14   for all directed edges $(u, v)$ with changed edge costs
15     update the edge cost $c(u, v)$;
16     if $v \neq s_{\text{start}}$
17       if $v$ was never visited by LPA* before then
18         $v(v) = g(v) = \infty; bp(v) =$ null;
19       $bp(v) = \arg\min_{s'' \in pred(v)} v(s'') + c(s'', v)$;
20       $g(v) = v(bp(v)) + c(bp(v), v)$; UpdateSetMembership($v$);
21   Move states from $INCONS$ into $OPEN$;
22   $CLOSED = \emptyset$;
```

Figure 3.12: LPA* algorithm

and INCONS are disjoint and INCONS contains exactly all the inconsistent states which are also in CLOSED.

Proof: We will prove the theorem by assuming that assumption C holds and $INCONS = \emptyset$ before the execution of ComputePath. Thus, the first time line 9 is executed $OPEN$ contains exactly all inconsistent states and $CLOSED = \emptyset$ according to assumption C.3 and $INCONS = \emptyset$. Therefore, the statement of the theorem is not violated at this point.

Let us now examine all the lines where we change v- or g-values of states or their set membership during the following execution of ComputePath. On line 10 we remove a state s from $OPEN$. This operation by itself cannot violate the theorem. On line 12 we insert the state s into $CLOSED$ but since we set $v(s) = g(s)$ on the same line, the state is consistent and still cannot violate the theorem. On all the other lines of ComputePath where we modify either v- or g-values of states except for state initialization (lines 15 and 23) we also call UpdateSetMembership function. The state initialization code leaves a state consistent but since the state was never visited before it correctly does not belong to any set. We thus only need to show that UpdateSetMembership function correctly updates the set membership of a state.

In UpdateSetMembership function if a state s is inconsistent and is not in *CLOSED* it is inserted into *OPEN*, otherwise it is inserted into *INCONS* (unless it is already there). Combined with Lemma 27 that states that *OPEN* and *CLOSED* are disjoint, this procedure ensures that an inconsistent state s does not appear in both *OPEN* and *INCONS* and does appear in *INCONS* if it also belongs to *CLOSED*. If a state s is consistent and belongs to *OPEN*, then it does not belong to *CLOSED* (since these sets are disjoint according to Lemma 27) and consequently does not belong to *INCONS*. If a state s is consistent and does not belong to *OPEN*, then it may potentially belong to *INCONS*. We check this and remove s from *INCONS* if it is there on line 7. ■

Theorem 40 *The function Key() satisfies the key-requirement in assumption C.1.*

Consider first case C.1(a): two arbitrary states s' and s such that $c^*(s', s_{\text{goal}}) < \infty$, $v(s') \geq g(s')$, $v(s) > g(s)$ and $g(s') > g(s) + \epsilon * c^*(s, s')$. We need to show that these conditions imply key$(s') >$ key(s). Given the definition of the key function in figure 3.12 we need to show that $[g(s') + \epsilon * h(s'); g(s')] > [g(s) + \epsilon * h(s); g(s)]$. We examine the inequality $g(s') > g(s) + \epsilon * c^*(s, s')$ and add $\epsilon * h(s')$, which should be finite since $c^*(s', s_{\text{goal}})$ is finite and heuristics are consistent. We thus have $g(s') + \epsilon * h(s') > g(s) + \epsilon * c^*(s, s') + \epsilon * h(s')$ and from Lemma 38 we obtain $g(s') + \epsilon * h(s') > g(s) + \epsilon * h(s)$ that guarantees that the desired inequality holds.

Consider now case C.1(b): two arbitrary states s' and s such that $c^*(s', s_{\text{goal}}) < \infty$, $v(s') \geq g(s')$, $v(s) < g(s)$ and $g(s') \geq v(s) + c^*(s, s')$. We need to show that these conditions imply key$(s') >$ key(s). Given the definition of the key function in figure 3.12 we need to show that $[g(s') + \epsilon * h(s'); g(s')] > [v(s) + h(s); v(s)]$. Since $v(s) < g(s)$, $v(s)$ is finite. Consider now the inequality $g(s') \geq v(s) + c^*(s, s')$. Because $v(s) < \infty$

and costs are positive we can conclude that $g(s') > v(s)$. We now add $\epsilon * h(s')$ to both sides of the inequality and use the consistency of heuristics as follows $g(s')+\epsilon*h(s') \geq v(s)+c^*(s,s')+\epsilon*h(s') \geq v(s)+c^*(s,s')+h(s') \geq v(s)+h(s)$. Hence, we have $g(s') + \epsilon * h(s') \geq v(s) + h(s)$ and $g(s') > v(s)$. These inequalities guarantee that $[g(s') + \epsilon * h(s'); g(s')] > [v(s) + h(s); v(s)]$. ■

Theorem 41 *Every time the ComputePath function is called from the Main function of LPA*, assumption C is fully satisfied prior to the execution of ComputePath.*

We have already proven that assumption C.1 is satisfied in Theorem 40. Before we prove assumptions C.2 and C.3 let us first prove the following statement denoted by (*): every time the ComputePath function is called *INCONS* $= \emptyset$. This is true the first time ComputePath is called since *INCONS* is reset to empty on line 8. Then the statement holds because before each subsequent call to ComputePath all the states in *INCONS* are moved into *OPEN* on line 21.

We are now ready to prove that assumptions C.2 and C.3 hold before each execution of ComputePath by induction. Consider the first call to the ComputePath function. At that point the g- and v-values of all states except for s_{start} are infinite, and $v(s_{start}) = \infty$ and $g(s_{start}) = 0$. Also, the bp-values of all states are equal to **null**. Thus, for every state $s \neq s_{\text{start}}$, $bp(s) =$ **null** and $v(s) = g(s) = \infty$, and for $s = s_{\text{start}}$, $bp(s) =$ **null**, $g(s) = 0$ and $v(s) = \infty$. Consequently, assumption C.2 holds. Additionally, the only inconsistent state is s_{start} which is inserted into *OPEN* at line 9. *OPEN* does not contain any other states and *CLOSED* is empty. Hence, assumption C.3 is also satisfied.

Suppose now that the assumptions C.2 and C.3 held during the previous calls to the ComputePath functions. We will now show that the assumptions C.2 and C.3 continue to hold the next time ComputePath is called and thus hold every time ComputePath is called by induction.

Since assumption C held before ComputePath started, after ComputePath exits, according to Lemma 26, all the v-values are non-negative, for every state $s \neq s_{\text{start}}$, $bp(s) = \arg\min_{s' \in pred(s)}(v(s') + c(s', s))$ and $g(s) = v(bp(s)) + c(bp(s), s)$ and for $s = s_{\text{start}}$ $bp(s) =$ **null** and $g(s) = 0$. The assumption C.2 therefore continues to hold when ComputePath exits. After ComputePath exits, costs may change on line 15. The bp- and g-values, though, are updated correctly on the following lines 19 and 20. The assumption C.2 therefore continues to hold the next time the ComputePath function is called.

Since assumption C holds and $INCONS = \emptyset$ (statement (*)) right before the call to ComputePath Theorem 39 applies and we conclude that at the time ComputePath exits *INCONS* contains exactly all inconsistent states that are in *CLOSED*. Combined with the Lemma 27 it implies that *INCONS* contains all and only inconsistent states that are not in *OPEN*.

We then may introduce new inconsistent states or make some inconsistent consistent by changing g-values of states on line 20. Their set membership, however, is updated by UpdateSetMembership function on the same line. In this function if a state s is inconsistent and is not in *CLOSED* it is inserted into *OPEN*, otherwise it is inserted into *INCONS* (unless it is already there). Because *OPEN* and *CLOSED* are disjoint at this point, this procedure ensures that an inconsistent state s does not appear in both *OPEN* and *INCONS* and does appear in *INCONS* if it also belongs to *CLOSED*. If a state s is consistent and belongs to *OPEN*, then it does not belong to *CLOSED* (since these sets are disjoint) and consequently does not belong to *INCONS*. If a state s is consistent and does not belong to *OPEN*, then it may potentially belong to *INCONS*. We check this and remove s from *INCONS* if it is there in UpdateSetMembership (figure 3.11). Thus, after UpdateSetMembership function exits *INCONS* continues to contain all and only inconsistent states that are not in *OPEN*.

By moving states from *INCONS* into *OPEN* on line 21 we make *OPEN* to contain exactly all inconsistent states. Thus, the next time ComputePath is executed, *OPEN* contains exactly all inconsistent states while *CLOSED* is empty (due to line 22). Hence, assumption C.3 holds the next time ComputePath is executed. ■

Theorem 42 *If LPA* algorithm is used with* $\epsilon = 1$ *(on line 3 in the Main() function), then the key function satisfies key-requirement 3.*

Proof: Suppose LPA* algorithm is used with $\epsilon = 1$. We then need to show that if we are given two arbitrary states s' and s such that $c^*(s', s_{\text{goal}}) < \infty$, $v(s') < g(s')$, $v(s) > g(s)$ and $v(s') > g(s) + c^*(s, s')$, then these conditions imply $\text{key}(s') > \text{key}(s)$. Given the definition of the key function in figure 3.12 we need to show that $[v(s')+h(s'); v(s')] > [g(s)+h(s); g(s)]$. Let us examine inequality $v(s') > g(s) + c^*(s, s')$ and add $h(s')$ to both sides of it. Since $c^*(s, s_{\text{goal}})$ is finite and heuristics are consistent, $h(s')$ is also finite. We therefore have $v(s') + h(s') > g(s) + c^*(s, s') + h(s')$ and from heuristics consistency we get $v(s') + h(s') > g(s) + h(s)$. This is sufficient to guarantee that $[v(s') + h(s'); v(s')] > [g(s) + h(s); g(s)]$ as desired. ■

3.6 Summary

In this chapter we have presented an incremental heuristic search, LPA*. It is a repetitive execution of A*-like searches where in between the searches one or more edge costs in the graph can change. LPA* is therefore suitable to solving dynamic or partially known environments. In these environments the edge costs in the graph that models the environment change as the environment changes or new information about it is received.

While each search iteration in the anytime heuristic search, ARA*, developed in the previous chapter, was based on the observation that A* can be viewed as a repetitive expansion of overconsistent states and we only need to identify these overconsistent states before each new search iteration, it is not sufficient for the operation of LPA* search. In this chapter we have shown that edge cost increases can actually introduce underconsistent states, something that can never happen in ARA* which changes only suboptimality bounds in between its search iterations. In this chapter we therefore first introduced a notion of a state being underconsistent. We then showed how to efficiently fix underconsistent states so that we once again can employ the formulation of A* search that repetitively expands overconsistent states. Once we had a version of search that can handle both underconsistent and overconsistent states, LPA* algorithm only needed to identify all such states in between search iterations. The section 3.3 in this chapter showed how LPA* managed this task.

Chapter 4

Anytime D*: Anytime Incremental A* with Provable Bounds on Suboptimality

4.1 Anytime D*

4.1.1 Combining ARA* with LPA*

Previous chapters have demonstrated an anytime search algorithm suitable for solving complex planning problems under the conditions where time is critical (ARA*) and an incremental search algorithm suitable for planning in domains that require frequent re-planning, for example, dynamic environments (LPA*). This chapter presents an algorithm that combines the two algorithms into a single anytime incremental search algorithm, called Anytime D* (where D* stands for Dynamic A*, same as in [78]). We will often refer to Anytime D* simply as AD*. AD* can plan under time constraints, just like ARA* can, but gains efficiency over it by being able to re-use previous planning efforts in domains where the model of a problem changes over time.

Both ARA* and LPA* re-use their previous search efforts when executing the ComputePath function to re-compute a solution. 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 a graph. 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 LPA* uses is already sufficient to handle both of these types of changes [1]. The reasons for this are as follows. A change in ϵ does not change the v- or g-values of states. Therefore, if there were no underconsistent states (states s with $v(s) < g(s)$) before ϵ was changed, then there would be no underconsistent states afterwards either. Combined with the fact that A* search by itself does not create any underconsistent states if there were none before it was executed, all states that ARA* needs to handle are either consistent of overconsistent. As a result, it only needs to execute the ComputePath function which requires that no state is underconsistent initially (the version developed in section 2.1.3). The ComputePath function used by LPA*, on the other hand, needs to be able to handle all three types of states, consistent, overconsistent and underconsistent, because underconsistent states can be created when some edge costs are increased (as discussed in section 3.1). This version of the ComputePath function (presented in section 3.2.2) is therefore a generalization of the ComputePath function used by ARA*. Consequently, it can be executed even if the changes in ϵ and edge costs occur at the same time, exactly the scenario that Anytime D* needs to be able to handle.

The pseudocode of Anytime D* algorithm is shown in figures 4.1 and 4.2. Just like in case of LPA*, the code for the ComputePath function is essentially the same as the ComputePath function that can handle arbitrary state initialization (figure 3.5). The differences, shown in bold, are that we maintain the *INCONS* list to keep track of all inconsistent states (lines 4 and 7, figure 4.1) so that we can restore *OPEN* to contain all inconsistent states before each call to the ComputePath function, 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).

The function Main() of Anytime D* (figure 4.2) first sets ϵ to a sufficiently high value ϵ_0, so that an initial, possibly highly suboptimal, plan 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 3.5) are satisfied. It then calls the ComputePath function to generate a first plan and publishes an ϵ-suboptimal solution. Afterwards, unless

[1]This is only true of the ComputePath function in generalized LPA* [59]. The ComputePath function of the original LPA* [48] would not be able to handle changes in ϵ as it can only search for optimal solutions.

```
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 if ($s \notin INCONS$) insert $s$ into $INCONS$;
5  else
6    if ($s \in OPEN$) remove $s$ from $OPEN$;
7    else if ($s \in INCONS$) remove $s$ from $INCONS$;

8  procedure ComputePath()
9  while(key($s_{\text{goal}}$) > $\min_{s \in OPEN}$(key($s$)) OR $v(s_{\text{goal}}) < g(s_{\text{goal}})$)
10   remove $s$ with the smallest key($s$) from $OPEN$;
11   if ($v(s) > g(s)$)
12     $v(s) = g(s)$; $CLOSED \leftarrow CLOSED \cup \{s\}$;
13     for each successor $s'$ of $s$
14       if $s'$ was never visited by AD* before then
15         $v(s') = g(s') = \infty$; $bp(s') =$ null;
16       if $g(s') > g(s) + c(s, s')$
17         $bp(s') = s$;
18         $g(s') = g(bp(s')) + c(bp(s'), s')$; UpdateSetMembership($s'$);
19   else //propagating underconsistency
20     $v(s) = \infty$; UpdateSetMembership($s$);
21     for each successor $s'$ of $s$
22       if $s'$ was never visited by AD* before then
23         $v(s') = g(s') = \infty$; $bp(s') =$ null;
24       if $bp(s') = s$
25         $bp(s') = \arg\min_{s'' \in pred(s')} v(s'') + c(s'', s')$;
26         $g(s') = v(bp(s')) + c(bp(s'), s')$; UpdateSetMembership($s'$);
```

Figure 4.1: Anytime D*: ComputePath function. This version is identical to the version used by LPA*. The changes as compared with the search that can handle arbitrary state initialization (figure 3.5), on the other hand, are shown in bold.

changes in edge costs are detected, the Main function decreases ϵ (line 25) and improves the quality of its solution, by re-initializing *OPEN* and *CLOSED* properly (lines 26 through 28) and re-executing the ComputePath function, until the solution is guaranteed to be optimal, that is, $\epsilon = 1$. This part 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 20 and 21) so that the second assumption of the ComputePath function (the second assumption in figure 3.5) is satisfied. These updates

The pseudocode below assumes the following:

1. Heuristics need to be consistent: $h(s) \leq c(s, s') + h(s')$ for any successor s' of s if $s \neq s_{\text{goal}}$ and $h(s) = 0$ if $s = s_{\text{goal}}$.

```
1  procedure key($s$)
2  if ($v(s) \geq g(s)$)
3    return $[g(s) + \epsilon * h(s); g(s)]$;
4  else
5    return $[v(s) + h(s); v(s)]$;

6  procedure Main()
7  $g(s_{\text{goal}}) = v(s_{\text{goal}}) = \infty$; $v(s_{\text{start}}) = \infty$; $bp(s_{\text{goal}}) = bp(s_{\text{start}}) = $ null;
8  $g(s_{\text{start}}) = 0$; $OPEN = CLOSED = INCONS = \emptyset$; $\epsilon = \epsilon_0$;
9  insert $s_{\text{start}}$ into $OPEN$ with key($s_{\text{start}}$);
10 forever
11   ComputePath();
12   publish $\epsilon$-suboptimal solution;
13   if $\epsilon = 1$
14     wait for changes in edge costs;
15   for all directed edges $(u, v)$ with changed edge costs
16     update the edge cost $c(u, v)$;
17     if $v \neq s_{\text{start}}$
18       if $v$ was never visited by AD* before then
19         $v(v) = g(v) = \infty; bp(v) = $ null;
20       $bp(v) = \arg\min_{s'' \in pred(v)} v(s'') + c(s'', v)$;
21       $g(v) = v(bp(v)) + c(bp(v), v)$; UpdateSetMembership($v$);
22   if significant edge cost changes were observed
23     increase $\epsilon$ or re-plan from scratch (i.e., re-execute Main function);
24   else if $\epsilon > 1$
25     decrease $\epsilon$;
26   Move states from $INCONS$ into $OPEN$;
27   Update the priorities for all $s \in OPEN$ according to key($s$);
28   $CLOSED = \emptyset$;
```

Figure 4.2: Anytime D*: key and Main functions

are identical to what LPA* does as well. If edge costs changes are large, then it may be computationally expensive to repair the current solution to regain or even improve ϵ-suboptimality. In such a case, one alternative for the algorithm is to increase ϵ so that a less optimal solution can be produced quickly (line 23). In some cases, however, it might be a good time to release all the currently used memory and just re-execute the Main() function for the initial setting of ϵ (line 23). 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 5.2 we give an example of a strategy that seems to work well for the problem of dynamic path planning. If the changes in edge costs are not substantial and are unlikely to cause expensive re-planning efforts, Main() can decrease ϵ (line 25), so that it re-plans for new edge costs and improves the solution via a single execution of the ComputePath function.

Similarly to how it is done in ARA*, and based on the idea in [84], the suboptimality bound on the solution can also be given by:

$$\frac{g(s_{goal})}{\min_{s \in OPEN \cup INCONS}(\mathrm{g}(s)+\mathrm{h}(s))} \tag{4.1}$$

If the ratio becomes smaller than one then $g(s_{goal})$ is already equal to the cost of an optimal solution. Thus, the actual suboptimality bound, ϵ', for each solution Anytime D* publishes can be computed as the minimum between ϵ and this new bound.

$$\epsilon' = \min(\epsilon, \frac{g(s_{goal})}{\min_{s \in OPEN \cup INCONS}(\mathrm{g}(s)+\mathrm{h}(s))}). \tag{4.2}$$

When interleaving planning with execution with Anytime D* as a planner, the agent executes the best plan it has so far while the planner works on fixing the plan if edge costs change and improving it. Same as with ARA*, however, as the agent moves the start of the search changes. Once again one can deal with this problem by performing the search backward: the start of the search, s_{start}, is the actual goal state of the agent, the goal of the search, s_{goal}, is the current state of the agent and the search is performed on a directed graph by reversing the direction of all edges in the original graph. The heuristics then estimate the distances to the current state of the agent as it is the goal of the search. Consequently, the heuristics change as the agent moves and this changes the priorities of the states in the *OPEN* list. Just like in ARA*, we can recompute the heuristic values of the states in the *OPEN* list during the reorder operation (line 27 in figure 4.2)[2].

4.1.2 Example

Figures 4.3 and 4.4 illustrate the approaches discussed in this thesis on a simple grid world planning problem. In this example we have an eight-

[2]The heap reorder operation might become expensive when the heap grows large. An optimization based on the idea in [78] can be done to avoid heap reordering. This is discussed in [56].

connected grid where black cells represent obstacles and white cells represent free space. The cell marked R denotes the position of an agent navigating this environment towards 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. 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 approach shown is backwards A* with $\epsilon = 1$ (figure 4.3, top row), that is, an optimal A* with its search directed from the goal state to the start state. The initial search performed by A* provides an optimal path for the agent. After the agent takes two steps along this path, it receives information indicating that one of the cells in the top wall is in fact free space. It then replans from scratch using A* to generate a new, optimal path to the goal. The combined total number of cells expanded at each of the first three agent positions is 31.

In contrast to the optimal A* search backwards A* with a constant inflation factor of $\epsilon = 2.5$ (figure 4.4, top row) produces an initial suboptimal solution very quickly. When the agent receives the new information regarding the top wall, this approach replans from scratch using its inflation factor and produces a new path, which happens to be optimal. The total number of cells expanded is only 19, but the solution is only guaranteed to be ϵ-suboptimal at each stage.

The second row in the figures shows the operation of the optimal LPA* (figure 4.3) and LPA* with a constant inflation factor of $\epsilon = 2.5$ (figure 4.4). Both versions were run backwards. The bounds on the quality of the solutions returned by these respective 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 solutions that are suboptimal every time it replans.

The last row in figure 4.3 shows the results of planning with ARA* and the last row in the figure 4.4 shows the results of planning with AD*. Both algorithms were run backwards as well. Each of these approaches begins by computing a suboptimal solution using an inflation factor of $\epsilon = 2.5$.

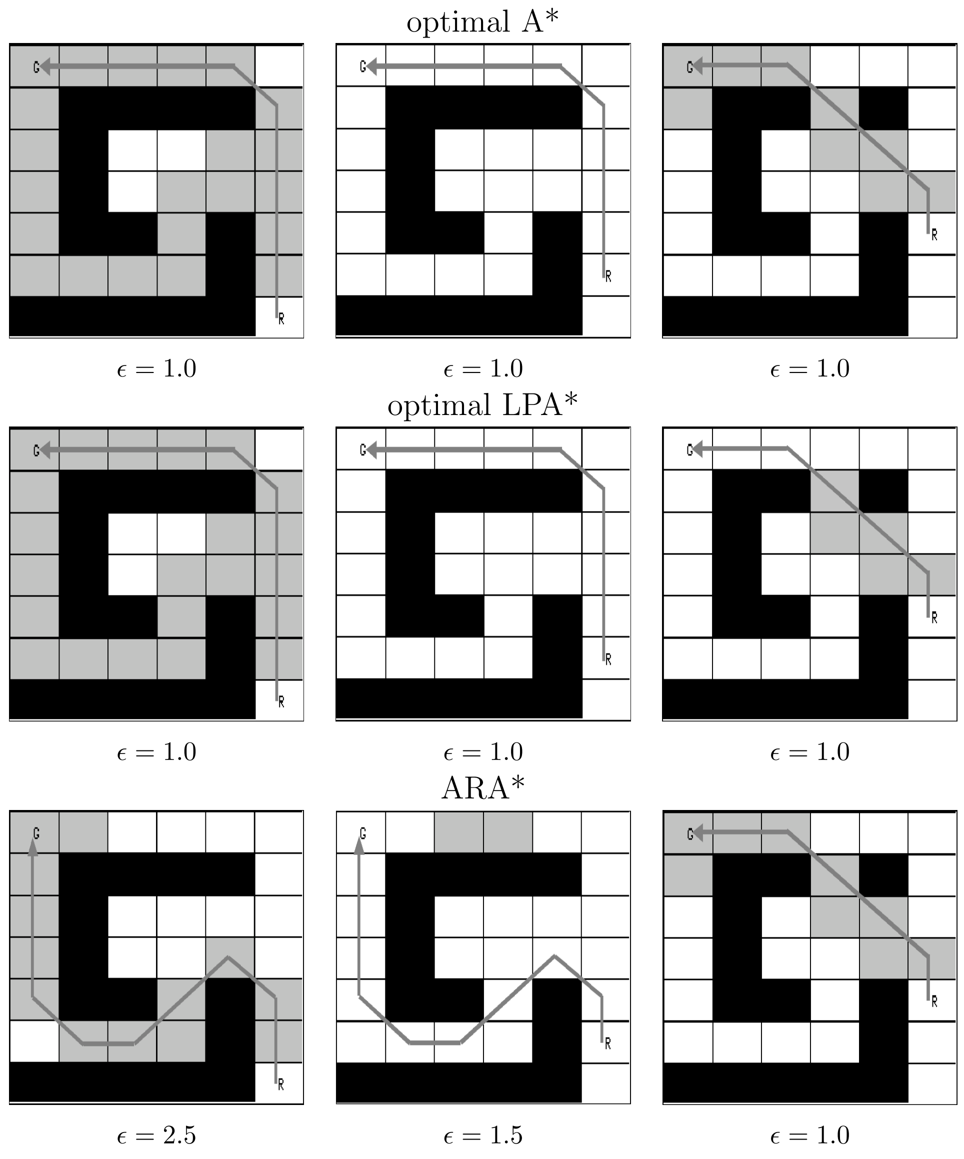


Figure 4.3: An example of planning with optimal A*, optimal LPA*, and ARA*. The states expanded by the algorithms are shown in grey. Note that after the third planning episode each of the algorithms can guarantee solution optimality ($\epsilon = 1.0$).

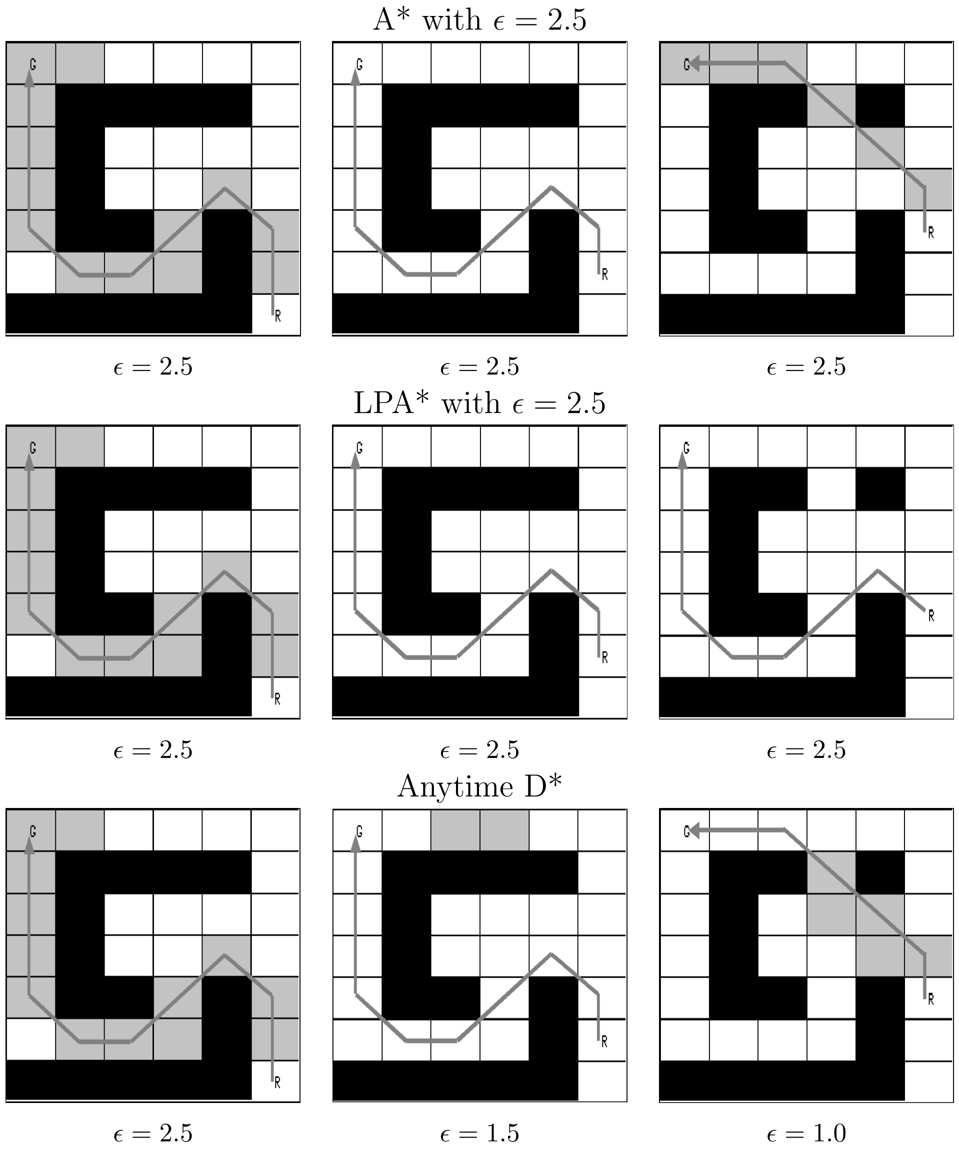


Figure 4.4: An example of planning with A* with an inflation factor $\epsilon = 2.5$, LPA* with an inflation factor $\epsilon = 2.5$ and AD*. The states expanded by the algorithms are shown in grey. Note that after the third planning episode only Anytime D* can guarantee solution optimality ($\epsilon = 1.0$).

While the agent moves one step along this path, this solution is improved by reducing the value of ϵ to 1.5 and reusing the results of the previous search. The path cost of this improved result is guaranteed to be at most 1.5 times the cost of an optimal path. Up to this point, both ARA* and AD* have expanded the same 15 cells each. However, when the robot moves one more step and finds out the top wall is broken, each approach reacts differently. Because ARA* cannot incorporate edge cost changes, it must replan from scratch with this new information. Using an inflation factor of 1.0 it produces an optimal solution after expanding 9 cells (in fact this solution would have been produced regardless of the inflation factor used). AD*, on the other hand, is able to repair its previous solution given the new information and lower its inflation factor at the same time. Thus, the only cells that are expanded are the 5 whose 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 provide anytime solutions in dynamic environments very efficiently.

4.1.3 Theoretical Properties of Anytime D*

In section 4.2 we prove a number of properties of AD*, including its termination and ϵ-suboptimality. In here we only state the most important of these theorems. These properties of Anytime D* are essentially the same as the properties of LPA* since both algorithms have the same ComputePath function.

Theorem 43 *When the ComputePath function exits, the following holds for any state s with ($c^*(s, s_{\text{goal}}) < \infty \wedge v(s) \geq g(s) \wedge key(s) \leq \min_{s' \in OPEN}(key(s'))$): $g^*(s) \leq g(s) \leq \epsilon * g^*(s)$, and the cost of the path from s_{start} to s defined by backpointers 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*. The first one says that the ComputePath function of Anytime D* has the same worst-case bound as LPA* on the number of times each state can be expanded, namely two.

Theorem 44 *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, just like in LPA*, 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 45 *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.*

4.1.4 Experimental Analysis of the Performance of Anytime D*

To evaluate the performance of AD*, we compared it to ARA* and LPA* with an inflation factor of $\epsilon = 20$ (it was actually an extension of LPA* to the case of a moving agent, called D* Lite [47]) on a simulated 3 degree of freedom (DOF) robotic arm manipulating an end-effector through a dynamic environment (see Figures 4.5 and 4.6). In this set of experiments, the base of the arm is fixed, and the task is to move into a particular goal configuration while navigating the end-effector around fixed and dynamic obstacles. We used a manufacturing-like scenario for testing, where the links of the arm exist in an obstacle-free plane, but the end-effector projects down into a cluttered space (such as a conveyor belt moving goods down a production line).

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

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,

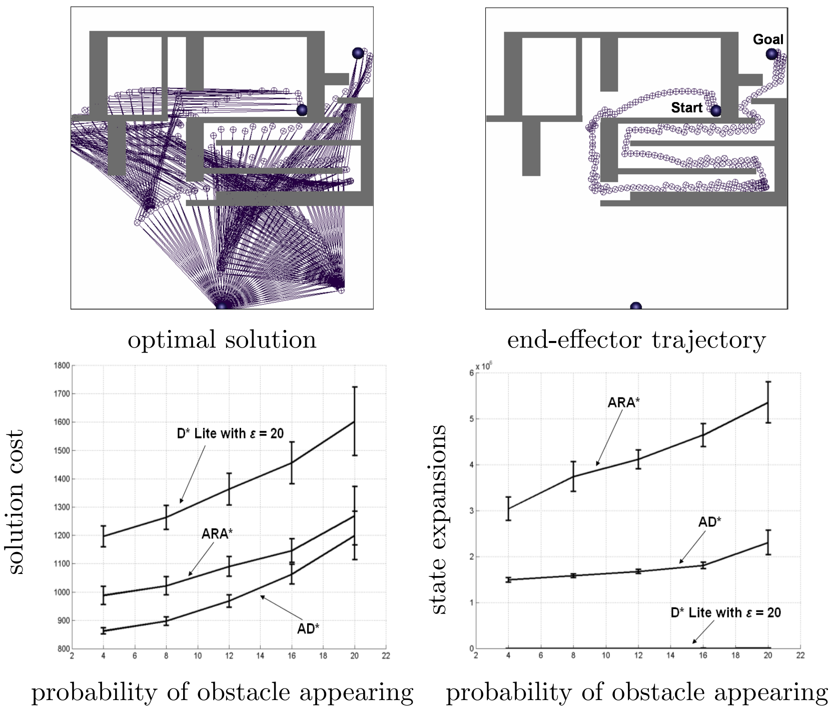


Figure 4.5: Environment used in our first experiment, along with the optimal solution and the end-effector trajectory (without any dynamic obstacles). The links of the arm exist in an obstacle-free plane (and therefore in the shown view from the top they look as if intersecting obstacles). The end-effector 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.

$\mathcal{T}^d = 1.0$ seconds, at each step along its path. The cost of moving each link was nonuniform: the link closest to the end-effector had a movement cost of 1, the middle link had a cost of 4, and the lower link had a cost of 9. The heuristic used by all algorithms was the maximum of two quantities; the first one 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 one 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. Our first environment had only one general route that the end-effector could take to get to its goal configuration, so the difference in path cost between the algorithms was due to manipulating the end-effector along this general path more or less efficiently. Our second experiment presented two qualitatively different routes the end-effector could take to the goal. One of these had a shorter distance in terms of end-effector grid cells but was narrower, while the other was longer but broader, allowing for the links to move in a much cheaper way to get to the goal.

Each environment consisted of a 50×50 grid, and the state space for each one consisted of slightly more than 2 million states. The results of the experiments, along with 95% confidence intervals, can be found in figures 4.5 and 4.6. 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 one 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 4.7 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 about 10% more costly than the optimal path, and it expanded roughly the same number of states as LPA* with no inflation factor. This is particularly encouraging: not only is the solution generated by AD* very close to optimal, but it is providing this solution in an anytime fashion for roughly the same total amount of

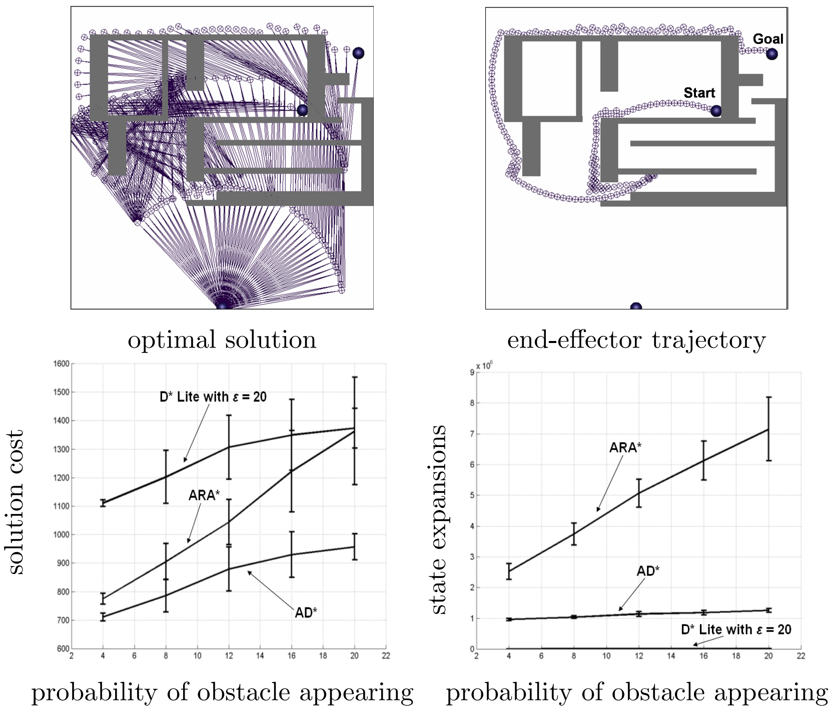
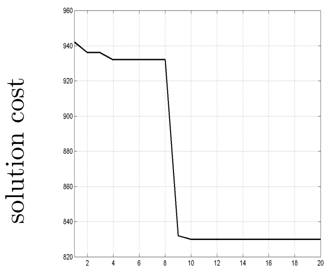
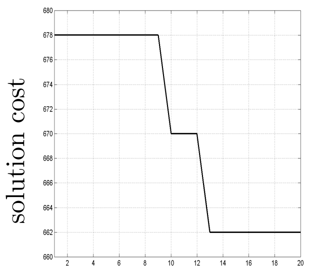


Figure 4.6: Environment used in our second experiment, along with the optimal solution and the end-effector trajectory (without any dynamic obstacles). Also shown are the solution cost of the path traversed and the number of states expanded by each of the three algorithms compared.



steps taken (planning episodes)
environment in figure 4.5



steps taken (planning episodes)
environment in figure 4.6

Figure 4.7: 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.

processing as would be required to generate the solution in one shot.

4.2 Proofs for Anytime D*

4.2.1 Pseudocode

The pseudocode for Anytime D* is given in figures 4.8 and 4.9.

4.2.2 Notation

The notation is exactly the same as we have used in section 3.5. Unlike in the previous algorithms, however, AD* now controls the computational expense of the ComputePath function by increasing or decreasing ϵ in few places inside the function Main. While the actual values of ϵ are not specified as they are domain dependent, they need to be finite and no smaller than 1. This also means that the initial value of ϵ, namely ϵ_0, is restricted to the same range of values.

```
 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 if ($s \notin INCONS$) insert $s$ into $INCONS$;
 5 else
 6   if ($s \in OPEN$) remove $s$ from $OPEN$;
 7   else if ($s \in INCONS$) remove $s$ from $INCONS$;

 8 procedure ComputePath()
 9 while(key($s_{\text{goal}}$) $>$ $\min_{s \in OPEN}$(key($s$)) OR $v(s_{\text{goal}}) < g(s_{\text{goal}})$)
10   remove $s$ with the smallest key($s$) from $OPEN$;
11   if ($v(s) > g(s)$)
12     $v(s) = g(s)$; $CLOSED \leftarrow CLOSED \cup \{s\}$;
13     for each successor $s'$ of $s$
14       if $s'$ was never visited by AD* before then
15         $v(s') = g(s') = \infty$; $bp(s') =$ null;
16       if $g(s') > g(s) + c(s, s')$
17         $bp(s') = s$;
18         $g(s') = g(bp(s')) + c(bp(s'), s')$; UpdateSetMembership($s'$);
19   else //propagating underconsistency
20     $v(s) = \infty$; UpdateSetMembership($s$);
21     for each successor $s'$ of $s$
22       if $s'$ was never visited by AD* before then
23         $v(s') = g(s') = \infty$; $bp(s') =$ null;
24       if $bp(s') = s$
25         $bp(s') = \arg\min_{s'' \in pred(s')} v(s'') + c(s'', s')$;
26         $g(s') = v(bp(s')) + c(bp(s'), s')$; UpdateSetMembership($s'$);
```

Figure 4.8: ComputePath function as used by AD*. Changes specific to AD* are shown in bold.

4.2.3 Proofs

The changes introduced into the ComputePath function (all lines in bold in figure 4.8, i.e., lines 4, 7, 14, 15, 22 and 23) as compared to the ComputePath function in figure 3.10 are identical to the changes we introduced into the ComputePath function when used by LPA* (figure 3.11). Therefore just like there, these changes do not affect the properties that we have already proven to hold for the ComputePath function in section 3.4.3, assuming that every time the function is called assumption C (figure 3.10) holds. Once again lines 4 and 7 are purely keeping track of states that are both inconsistent and not in *OPEN*, while the other four lines we introduced are used to perform online initialization of states that have not been seen before. To simplify

The pseudocode below assumes the following (*Assumption E*):

1. Heuristics need to be consistent: $h(s) \leq c(s, s') + h(s')$ for any successor s' of s if $s \neq s_{\text{goal}}$ and $h(s) = 0$ if $s = s_{\text{goal}}$.

```
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()
$g(s_{\text{goal}}) = v(s_{\text{goal}}) = \infty$; $v(s_{\text{start}}) = \infty$; $bp(s_{\text{goal}}) = bp(s_{\text{start}}) = \textbf{null}$;
$g(s_{\text{start}}) = 0$; $OPEN = CLOSED = INCONS = \emptyset$; $\epsilon = \epsilon_0$;
insert $s_{\text{start}}$ into $OPEN$ with key($s_{\text{start}}$);
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 $v \neq s_{\text{start}}$
      if $v$ was never visited by AD* before then
        $v(v) = g(v) = \infty; bp(v) = \textbf{null}$;
      $bp(v) = \arg\min_{s'' \in pred(v)} v(s'') + c(s'', v)$;
      $g(v) = v(bp(v)) + c(bp(v), v)$; 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($s$);
  $CLOSED = \emptyset$;
```

Figure 4.9: AD* algorithm

the subsequent proofs we therefore assume that any state s with undefined values (not visited) has $v(s) = g(s) = \infty$ and $bp(s) = \textbf{null}$.

Similarly to the proofs for ARA* and LPA*, the task of proofs in this section is to show that AD* algorithm as presented in figure 4.9 ensures that the assumption C is true every time it calls the ComputePath function. Once this is shown, all the properties in section 3.4.3 apply here and we thus obtain the desired properties about each search iteration in AD* including its

ϵ-suboptimality and other properties regarding its efficiency. The flow of the proofs is essentially identical to the flow of the proofs for LPA* (section 3.5.3).

Lemma 46 *For any pair of states s and s', $\epsilon * h(s) \leq \epsilon * c^*(s, s') + \epsilon * h(s')$.*

Proof: According to [66] the consistency property required of heuristics in Assumption E is equivalent to the restriction that $h(s) \leq c^*(s, s') + h(s')$ for *any* pair of states s, s' and $h(s_{\text{goal}}) = 0$. The theorem then follows by multiplying the inequality with ϵ. ■

Theorem 47 *If assumption C holds and INCONS = ∅ before the execution of ComputePath, then during the execution of ComputePath, at line 9, OPEN and INCONS are disjoint and INCONS contains exactly all the inconsistent states which are also in CLOSED.*

Proof: We will prove the theorem by assuming that assumption C holds and *INCONS* = ∅ before the execution of ComputePath. Thus, the first time line 9 is executed *OPEN* contains exactly all inconsistent states and *CLOSED* = ∅ according to assumption C.3 and *INCONS* = ∅. Therefore, the statement of the theorem is not violated at this point.

Let us now examine all the lines where we change v- or g-values of states or their set membership during the following execution of ComputePath. On line 10 we remove a state s from *OPEN*. This operation by itself cannot violate the theorem. On line 12 we insert the state s into *CLOSED* but since we set $v(s) = g(s)$ on the same line, the state is consistent and still cannot violate the theorem. On all the other lines of ComputePath where we modify either v- or g-values of states except for state initialization (lines 15 and 23) we also call UpdateSetMembership function. The state initialization code leaves a state consistent but since the state was never visited before it correctly does not belong to any set. We thus only need to show that UpdateSetMembership function correctly updates the set membership of a state.

In UpdateSetMembership function if a state s is inconsistent and is not in *CLOSED* it is inserted into *OPEN*, otherwise it is inserted into *INCONS* (unless it is already there). Combined with Lemma 27 that states that *OPEN* and *CLOSED* are disjoint, this procedure ensures that an inconsistent state s does not appear in both *OPEN* and *INCONS* and does appear in *INCONS* if it also belongs to *CLOSED*. If a state s is consistent and belongs to *OPEN*, then it does not belong to *CLOSED* (since these sets are disjoint according to Lemma 27) and consequently does not belong to *INCONS*. If a state s is consistent and does not belong to *OPEN*, then it may potentially belong to *INCONS*. We check this and remove s from *INCONS* if it is there on line 7. ∎

Theorem 48 *Key function satisfies the key-requirement in assumption C.1.*

Consider first case C.1(a): two arbitrary states s' and s such that $c^*(s', s_{\text{goal}}) < \infty$, $v(s') \geq g(s')$, $v(s) > g(s)$ and $g(s') > g(s) + \epsilon * c^*(s, s')$. We need to show that these conditions imply $\text{key}(s') > \text{key}(s)$. Given the definition of the key function in figure 4.9 we need to show that $[g(s') + \epsilon * h(s'); g(s')] > [g(s) + \epsilon * h(s); g(s)]$. We examine the inequality $g(s') > g(s) + \epsilon * c^*(s, s')$ and add $\epsilon * h(s')$, which is finite since $c^*(s', s_{\text{goal}})$ is finite and heuristics are consistent. We thus have $g(s') + \epsilon * h(s') > g(s) + \epsilon * c^*(s, s') + \epsilon * h(s')$ and from Lemma 46 we obtain $g(s') + \epsilon * h(s') > g(s) + \epsilon * h(s)$ that guarantees that the desired inequality holds.

Consider now case C.1(b): two arbitrary states s' and s such that $c^*(s', s_{\text{goal}}) < \infty$, $v(s') \geq g(s')$, $v(s) < g(s)$ and $g(s') \geq v(s) + c^*(s, s')$. We need to show that these conditions imply $\text{key}(s') > \text{key}(s)$. Given the definition of the key function in figure 4.9 we need to show that $[g(s') + \epsilon * h(s'); g(s')] > [v(s) + h(s); v(s)]$. Since $v(s) < g(s)$, $v(s)$ is finite. Consider now the inequality $g(s') \geq v(s) + c^*(s, s')$. Because $v(s) < \infty$ and costs are positive we can conclude that $g(s') > v(s)$. We now add $\epsilon * h(s')$ to both sides of the inequality and use the consistency of heuristics as follows $g(s') + \epsilon * h(s') \geq v(s) + c^*(s, s') + \epsilon * h(s') \geq v(s) + c^*(s, s') + h(s') \geq v(s) + h(s)$. Hence, we have $g(s') + \epsilon * h(s') \geq v(s) + h(s)$ and $g(s') > v(s)$. These inequalities guarantee that $[g(s') + \epsilon * h(s'); g(s')] > [v(s) + h(s); v(s)]$. ∎

Theorem 49 *Every time the ComputePath function is called from the Main() function of AD*, assumption C is fully satisfied prior to the execution of ComputePath.*

We have already proven that assumption C.1 is satisfied in Theorem 48. Before we prove assumptions C.2 and C.3 let us first prove the following statement denoted by (*): every time the ComputePath function is called *INCONS* $= \emptyset$. This is true the first time ComputePath is called since *INCONS* is reset to empty on line 8. Then the statement holds because before each subsequent call to ComputePath all the states in *INCONS* are moved into *OPEN* on line 26.

We are now ready to prove that assumptions C.2 and C.3 hold before each execution of ComputePath by induction. Consider the first call to the ComputePath function. At that point the g- and v-values of all states except for s_{start} are infinite, and $v(s_{start}) = \infty$ and $g(s_{start}) = 0$. Also, the bp-values of all states are equal to **null**. Thus, for every state $s \neq s_{\text{start}}$, $bp(s) = \textbf{null}$ and $v(s) = g(s) = \infty$, and for $s = s_{\text{start}}$, $bp(s) = \textbf{null}$, $g(s) = 0$ and $v(s) = \infty$. Consequently, assumption C.2 holds. Additionally, the only inconsistent state is s_{start} which is inserted into *OPEN* at line 9. *OPEN* does not contain any other states and *CLOSED* is empty. Hence, assumption C.3 is also satisfied.

Suppose now that the assumptions C.2 and C.3 held during the previous calls to the ComputePath functions. We will now show that the assumptions C.2 and C.3 continue to hold the next time ComputePath is called and thus hold every time ComputePath is called by induction.

Since assumption C held before ComputePath started, after ComputePath exits, according to Lemma 26, all the v-values are non-negative, for every state $s \neq s_{\text{start}}$, $bp(s) = \arg\min_{s' \in pred(s)}(v(s') + c(s', s))$ and $g(s) = v(bp(s)) + c(bp(s), s)$ and for $s = s_{\text{start}}$ $bp(s) = \textbf{null}$ and $g(s) = 0$. The assumption C.2 therefore continues to hold when ComputePath exits. After ComputePath exits, costs may change on line 16. The bp- and g-values, though, are updated correctly on the following lines 20 and 21. The assumption C.2 therefore continues to hold the next time the ComputePath function is called.

Since assumption C holds and *INCONS* $= \emptyset$ (statement (*)) right before the call to ComputePath Theorem 47 applies and we conclude that at the time ComputePath exits *INCONS* contains exactly all inconsistent states that are in *CLOSED*. Combined with the Lemma 27 it implies that *INCONS* contains all and only inconsistent states that are not in *OPEN*.

We then may introduce new inconsistent states or make some inconsistent consistent by changing g-values of states on line 21. Their set membership, however, is updated by UpdateSetMembership function on the same line. In this function if a state s is inconsistent and is not in *CLOSED* it is inserted into *OPEN*, otherwise it is inserted into *INCONS* (unless it is already there). Because *OPEN* and *CLOSED* are disjoint at this point, this procedure ensures that an inconsistent state s does not appear in both *OPEN* and *INCONS* and does appear in *INCONS* if it also belongs to *CLOSED*.

If a state s is consistent and belongs to *OPEN*, then it does not belong to *CLOSED* (since these sets are disjoint) and consequently does not belong to *INCONS*. If a state s is consistent and does not belong to *OPEN*, then it may potentially belong to *INCONS*. We check this and remove s from *INCONS* if it is there in UpdateSetMembership (figure 4.8). Thus, after UpdateSetMembership function exits *INCONS* continues to contain all and only inconsistent states that are not in *OPEN*.

By moving states from *INCONS* into *OPEN* on line 26 we make *OPEN* to contain exactly all inconsistent states. Thus, the next time ComputePath is executed, *OPEN* contains exactly all inconsistent states while *CLOSED* is empty (due to line 28). Hence, assumption C.3 holds the next time ComputePath is executed. ■

4.3 Summary

In this chapter we have presented an anytime incremental heuristic search algorithm called Anytime D*. To the best of our knowledge, it is currently the only search to combine both anytime and incremental properties together. Similarly to the algorithms presented in the previous chapters, Anytime D* is also a repetitive execution of A*-like searches. In between the searches, however, now both the desired suboptimality bounds on the solutions as well an arbitrary number of edge costs can change. The anytime property of the algorithm is achieved by initially setting the desired suboptimality bound large and thus finding a solution quickly, and then slowly decreasing the desired suboptimality bound and finding new improved solutions until planning time runs out. The incremental property of the algorithm is achieved by updating edge costs and replanning in the updated graphs. Anytime D* is thus suitable to solving complex dynamic or partially known environments that cannot be solved optimally quickly enough and require frequent replanning.

Independently of whether the desired suboptimality bound alone or whether edge costs alone or both have been changed in between search iterations, each search in Anytime D* gains efficiency by reusing the results of its previous executions. Once again, similarly to ARA* and LPA* algorithms, it is based on the observation that A* can be viewed as a repetitive expansion of overconsistent states. Since edge cost increases can introduce underconsistent states the search in Anytime D* deals with them efficiently in the same manner it was done in LPA*.

Chapter 5

Application to Outdoor Robot Navigation

5.1 Problem

The motivation for the planning algorithms in this thesis was in part the development of more efficient path-planning for mobile robots, such as the ones in figure 5.1. 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 address this we decided to use 4D state space for planning, where each

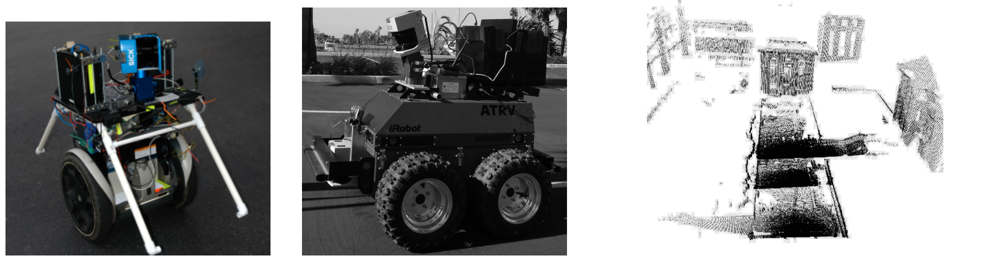
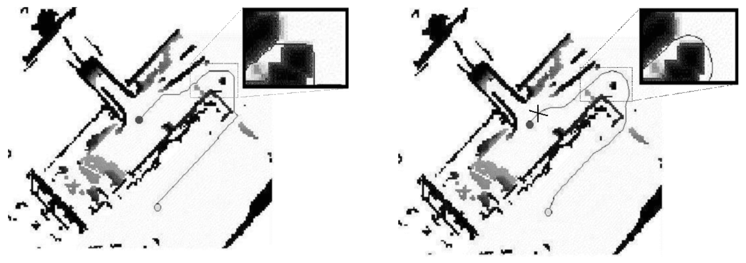

(a) segbot robot (b) ATRV robot (c) 3D Map constructed by ATRV

Figure 5.1: Robotic platforms that used AD* for planning

state was characterized by 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 under the dynamics constraints of the robot. For example, the robot inertia constraints restrict the planner from coming up with plans where a robot slows down faster than its maximum deceleration permits. The 2D planners that only take into account xy-position of the robot usually don't take into account these constrains in a general and systematic way. Perhaps more importantly, the constraints on the rotational velocity of the robot limit how much the robot can turn given its current translational velocity. The 2D planners assume that the robot can make arbitrary sharp turns, and therefore in practice a robot controller that executes a plan generated by such planner must drive the robot slowly and often even stop it when it has to turn.

As an example, figure 5.2(a) shows the optimal 2D plan, and figure 5.2(b) shows the optimal 4D plan for the same environment. The map of the environment was constructed from a scan gathered by an ATRV 3D laser scan [32] shown in figure 5.1(c). In black are shown what a 2D mapping algorithm labeled as obstacles. 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 state is the circle to the left, while its goal is the 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. So, the robot needs to re-plan its trajectory many times on its way to the goal, and it needs to do it fast while moving. Anytime D* is a well-suited planning solution for this task.



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

Figure 5.2: The comparison of optimal 2D plan with optimal 4D plan.

5.2 Our Approach

To solve the problem we built a two-level planner: a 4D planner that uses Anytime D*, and a 2D (x and y) planner that performs A* search and whose results are used to initialize the heuristics for the 4D planner. 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 on every move of the robot without slowing it down.

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 fast. While the robot executes it, on the next iteration of ComputePath the plan is improved and repaired if new information about the environment is gathered. The plan is then sent to the robot as a new plan. Thus, at any point of time, the robot has access to a 4D plan and does not have to stop. 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.

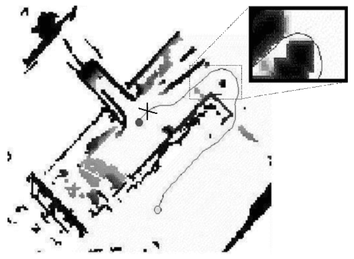
When no new information about the environment is observed, then Anytime D* just decreases ϵ to have a better bound on the suboptimality of the solution the next time it calls ComputePath. When new information about the environment is gathered, Anytime D* has to re-plan. As discussed in section 4.1.1, 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. The strategy that we have chosen was to decide 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 have never chosen to increase ϵ without discarding the current search tree. The reason was that the robot moved and a large number of previously computed states were often becoming irrelevant. By clearing the memory, we made sure it was not getting full with these irrelevant states.

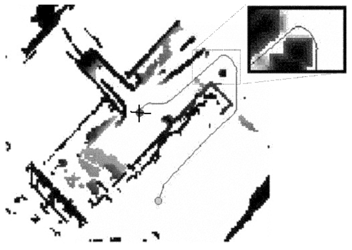
Using our approach we were able to build a real robot system that can plan/re-plan in outdoor environments while navigating at relatively high speed. The system was deployed on two real robotic platforms: the Segway Robotic Mobility Platform shown in figure 5.1(a) and the ATRV vehicle shown in figure 5.1(b). Both used laser range finders (one on the Segway and two on the ATRV) for mapping and inertial measurement unit combined with global positioning system for position estimation.

5.3 Examples

As mentioned before, the size of the environment in the example in figure 5.2 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 becomes too large for planning and re-planning optimally by a moving robot. In figure 5.3 we show the advantage of using our approach that performs 4D planning with Anytime D*.

Figure 5.3(b) shows the initial plan computed by the 4D Planner that runs 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 5.2(a)). It is, however, somewhat less smooth than the optimal 4D plan (figure 5.3(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 the shown plan was 556 ms. (The planning for all experiments was done on a 1GHz Pentium processor.) As a result, the robot that runs


(a) optimal 4D search with A* after 25 secs


(b) 4D search with AD* after 0.6 secs ($\epsilon = 2.5$)

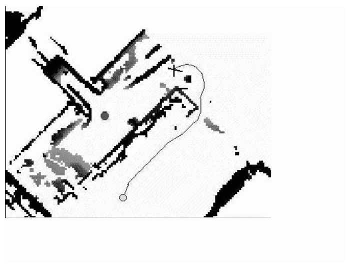
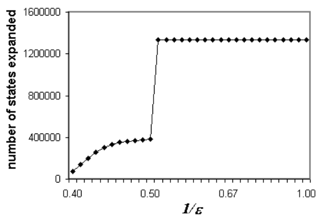
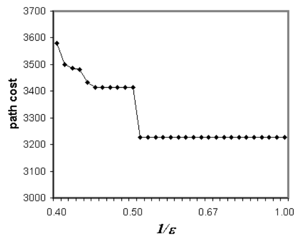

(c) 4D search with AD* after 25 secs ($\epsilon = 1.0$)

Figure 5.3: 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 initial map was observed during execution (i.e., no cost changes).



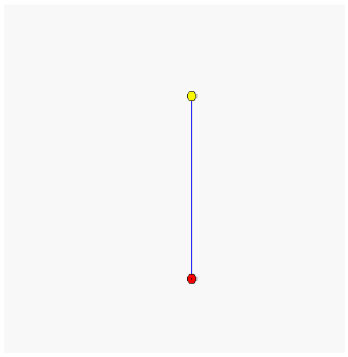
(a) cumulative number of expansions vs. $1/\epsilon$

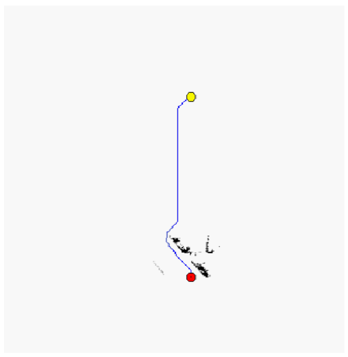


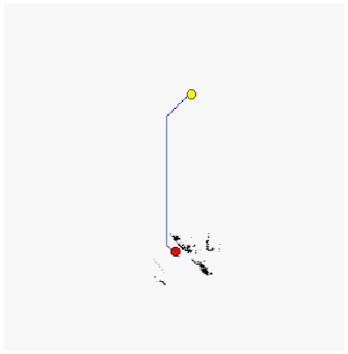
(b) solution cost vs. $1/\epsilon$

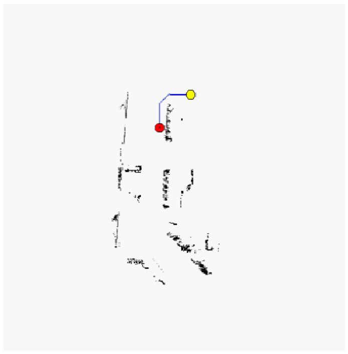
Figure 5.4: The performance of AD* planner in the same environment as in figure 5.3 but for a different configuration of goal and start locations (a harder scenario) and for a fixed robot position (i.e., the robot does not move).

(a) ATRV while navigating


(b) initial map and plan


(c) current map and plan


(d) current map and plan


(e) current map and plan

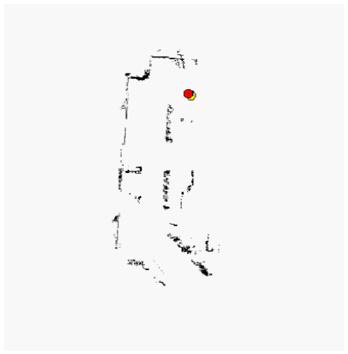

(f) ATRV at its goal

Figure 5.5: A run by an ATRV robot in an initially unknown environment. Figure (a) shows ATRV navigating in the environment which is a parking lot full of cars. Figure (b) shows initial map as known by the robot and the initial plan AD* constructs. Figures (c-e) show the map constructed by the robot and a plan generated by AD* at different time points. Figure (f) shows the map constructed by the robot by the time it reaches its goal.

Anytime D* can start executing a plan much earlier. The cross in figure 5.3(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 5.3(c) shows the position of the robot running Anytime D* after 25 seconds from the time a goal location was given to it. The Anytime D* robot has advanced much further, and its plan by now has converged to optimal and thus is no different from the one in figure 5.3(a).

In figure 5.4(a) and figure 5.4(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 while Anytime D* was executed until convergence. We did this for the sake of easier analysis 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 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 1,334 thousand. To compare, the number of states expanded by the optimal planner would have been over 953 thousand. Thus, over all iterations about 30 percent extra states are expanded but a solution that is suboptimal by about ten percent was obtained very fast.

In the example we have just seen the environment was consistent with the map constructed initially and thus during the agent execution there were no edge cost changes. In contrast, in the example in figure 5.5 the ATRV robot navigates to its goal location in initially completely unknown environment (figure 5.5(b)). The experiment was done in a different environment from the one used previously. In particular, in this experiment the robot navigates a parking lot full of cars (figure 5.5(a)). The robot assumes that all unknown area is traversable (the assumption commonly known as a freespace assumption). Under this assumption the robot performs a 4D planning using Anytime D*. While executing the plan it uses its two laser range finders to gather new information about the environment, updates the map accordingly and fixes and improves the plan. Figures 5.5(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. Nevertheless, the Anytime D* based planner was able to provide the robot with safe 4D plans at any point in time that allowed the robot to navigate unknown and partially known environments at speeds up to 1.5 meters/sec.

Chapter 6

A Brief Guide to Usage

The algorithms presented in this thesis are all suited better to some domains and worse to others. They can also benefit from certain optimizations in some cases. The main purpose of this chapter is to give some intuition as to when the presented algorithms might be useful and when they might not. We will only consider the class of problems in which we are given a start state and one or more goal states and the task is to find a sequence of actions that lead to the goal state while minimizing the cumulative cost of action executions. Such problems are search-based planning problems. Our algorithms have not been designed to be useful for the problems where one is only concerned with finding a state that meets certain goal criteria (for example, graph coloring problems, task scheduling problems, or Boolean satisfiability problems). These problems differ from the ones we are concerned with in this thesis in that the costs of all actions on the path to the final state are zero and only the degree to which the final state meets the goal criteria is important.

In section 6.1 we will try to describe the advantages and disadvantages of few commonly used approaches that do not fall into the category of search-based planning. In section 6.2, on the other hand, we will try to provide some insights into what the factors are that can make the application of one of our algorithms successful. In that section we will also point out some of the existing optimizations described elsewhere that might benefit the algorithms in certain domains. For additional discussion on these topics please refer to [21, 49].

6.1 Search-based Planning versus Other Planning Approaches

The planning algorithms presented in this thesis fall into the class of search-based planning algorithms, whereas there exist a number of other classes of planning algorithms (a thorough description of many of these alternatives can be found in [55]). For example, for high-dimensional state spaces, sampling-based algorithms are a popular class of planning algorithms. This class includes such planners as the Probabilistic RoadMap (PRM) [41], the Probabilistic Roadmap of Trees (PRT) [67], the Rapidly-exploring Random Tree (RRT) [53], the Expansive-Spaces Tree (EST) [36], and a number of similar approaches. They are all based on the high-level idea of generating samples of valid configurations and using local controllers to connect these samples to their neighbors. A planner can then search for a solution in the resulting graph, provided there is a sample that corresponds to the start state and at least one sample that satisfies the goal conditions. Some planners, such as PRMs, construct the graph first and then search it for a solution; others, such as RRTs, simultaneously construct the graph and search for a solution.

Most of the sampling-based planning algorithms are complete provided the sampling is done infinitely long. The algorithms are well-suited to solving high-dimensional problems as the size of a graph they construct is not explicitly exponential in the number of dimensions. They are, however, restricted to solving problems for which good local controllers are available. A common type of problem to which they are applied is motion planning in high-dimensional continuous state spaces; the robot arm motion planning domain from chapter 2 is an example of such a problem. It is hard, however, to develop good local controllers for arbitrary discrete domains or for continuous domains that are cluttered with obstacles and have narrow passages (see figure 6.1(a) for an example). Without good local controllers, the complexity of constructing a graph using samples becomes too large since many of the generated samples cannot be connected to any of their neighbors and are therefore rejected. In contrast, the algorithms presented in this thesis search general graphs and can therefore handle arbitrary domains, including discrete ones. There are other properties, though, that affect the performance of our algorithms as discussed in the next section. Most sampling-based planning algorithms also provide no bounds on their suboptimality due to the random construction of the graph. For the solutions to lower dimensional planning problems such as robot navigation, however, it is usually important to maintain the quality close to optimal. It is therefore quite common to discretize a low-dimensional space of the problem in some consistent manner and then use search-based planning algorithms such as A* and the algorithms presented in this thesis to find solutions with known bounds on suboptimality.

More generally, the searches that we have presented take as an input an arbitrarily generated graph. Therefore even some sampling-based planning methods can take advantage of our algorithms for planning and replanning as long as they generate graphs a priori and as long as an informative heuristic function is available. (The latter requirement can potentially be difficult to satisfy, however, considering the fact that the graphs are generated at random.) Some work has already been done in this direction [20, 39].

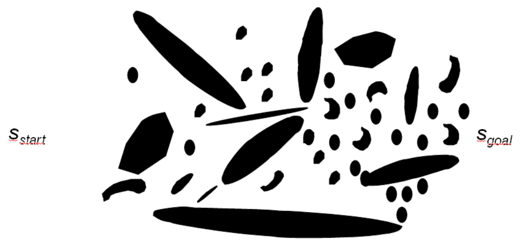
Some of other currently popular classes of planning algorithms include Roadmap algorithms (e.g., [10]) and cell decomposition based algorithms (e.g., [1]). These approaches deterministically pre-compute a sparse graph representation of the environment, taking into account its geometrical properties, and then search for a solution in the graph. They differ substantially in how the graphs are constructed but both use searches to find solutions in the graphs. Similarly to sampling-based planning algorithms, however, these approaches typically do not provide bounds on the suboptimality of their solutions.

A very different class of planning approaches builds on the symbolic representation of problems [75]. Such approaches are well-suited for discrete problems. They use logical representations of states and actions to avoid the instantiation of every possible state and to direct planning in a potentially more effective way than search-based planning algorithms. A drawback of these approaches is that they usually provide no bounds on the suboptimality of their solutions. There has been recent interest, however, in using heuristic searches to solve symbolically represented problems (e.g., [8, 16, 38]). And in fact, the LPA* algorithm has been successfully implemented as a symbolic replanner [46].

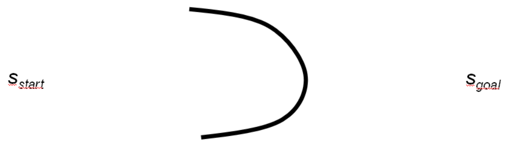
Since we have applied our algorithms to robot navigation problems, it is important to mention that there exist a vast number of popular algorithms for robot navigation that abandon the idea of planning altogether. These algorithms try to move the robot towards the goal greedily and apply different methods of dealing with obstacles on the way. The Bug algorithm [61] and its variants, for example, will make the robot follow the boundary of an obstacle when encountered. Potential fields methods [7, 42] assign repulsive forces to obstacles and attractive forces to goal locations, and the robot always moves in the direction given by the sum of all of these forces. A number of other algorithms have been successfully applied to robot navigation including schema-based navigation [4], vector field histograms [9] and numerous others. These greedy algorithms are very simple to implement and work well in environments with sparse obstacles. Perhaps even more importantly they are well-suited for very large environments since the size of an environment does not have an effect on their complexity. The trajectories generated by these approaches, however, can be drastically inefficient with no non-trivial bounds on suboptimality and can even result in failures in domains with irreversible actions. Many of these approaches also have a hard time dealing with local minima created by obstacles in the environment.

6.2 Applying ARA*, LPA* or AD*

In this section, we will elaborate on the conditions under which the anytime and incremental properties of our algorithms are effective. The anytime behavior of ARA* and AD* strongly relies on the properties of the heuristics they use. 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 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. For example, in figure 6.1(a), where there are only small U-shaped obstacles, all local minima that the search can encounter will be small, and it will not take



(a) a "good" example domain for ARA*



(b) a "bad" example domain for ARA*

Figure 6.1: Domain examples where ARA* and AD* work well (a) and where they do not (b) (assuming the heuristics are Euclidean distances). In contrast, sampling-based methods like PRMs are likely to work well in the domains with sparse obstacles such as (b) and to have a harder time solving cluttered domains such as (a).

long to expand all the states within these minima. Thus, ARA* and AD* will quickly find good paths to the goal. In contrast, figure 6.1(b) has a single large local minimum created by the large U-shaped obstacle. Even the first search of ARA* or AD* will have to expand quite a few states before it can get out of the minimum and proceed with the search.

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 which results in shallow local minima. In many cases (although not always) the design of such a heuristic function can be a much simpler task than the task of designing a whole new anytime (re-)planning algorithm for solving the problem at hand.[1]

Both LPA* and AD* are very effective for replanning in the context of mobile robot navigation. Typically, in such scenarios the changes to the graph are happening close to the robot (through its observations). Their effects are therefore usually limited and much of the previous search efforts can be reused if the search is done backwards: from the goal state towards the state of the robot. Using an incremental replanner such as LPA* and 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 LPA* to be even *less* efficient than A*. Mainly, this is because it is possible for LPA* 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 LPA*, and one that illustrates this possibility, is when changes are being made to the graph in the vicinity of the start of the search. Similarly, LPA* can also be less efficient than A* if there are a lot of edge cost changes. It is thus common for systems using LPA* 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 5.2 gives a particular way for deciding when to plan from scratch, at least in the domain of robot

[1]It would also be interesting to extend 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 algorithm 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 [50], on the suboptimality of the solution found.

navigation using 4D planning.[2]

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 [78]. Its implementation for LPA* is discussed in [47] and for ARA* and AD* in [56].

There also exist a few other optimizations to the algorithms presented here. For example, Delayed D* [22] 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. As another example, in domains with a very high branching factor, ARA* and AD* can be sped up using the idea in [84] that prunes from *OPEN* the states that are guaranteed not to be useful for improving a current plan. These and other potentially useful optimizations are described more thoroughly in [21].

A series of other optimizations concern the key() functions in LPA* and AD*. The key() function we give in this thesis is a two dimensional key function presented in figure 3.7. A number of other key functions, however, can also be designed that satisfy the restrictions on the key (the first assumption in figure 3.5) and are therefore perfectly valid. Some of them 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 prior-

[2]In the future, it would certainly be interesting to see a theoretical analysis of when algorithms such as LPA* and AD* should give up re-planning and plan from scratch.

ities is small, *OPEN* can be implemented using buckets instead. 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 the *OPEN* to be implemented using buckets. [59] presents a number of other valid key() functions including the 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 criteria is important in domains where numerous optimal solutions exist and we want to avoid exploring all of them.

Chapter 7

Summary of Contributions

Planners used by agents operating in the real world must often be able to provide plans under conditions where time is critical. In addition, the world is dynamic and the world models used for planning are often imperfect and are only partially known. Consequently, planners are called to re-plan every time the model is updated based on the incoming agent's sensor information. These conditions make planning for real world tasks a hard problem.

In this thesis we contribute to the research on planning in the following ways. We first 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: it 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 only work on correcting the inconsistent states and will reuse the consistent states whose values are already correct.

We used our formulation of A* search to construct several variants of A*. First, we constructed an anytime heuristic search, ARA*, that provides provable bounds on the suboptimality of any solution it outputs. As an anytime algorithm it finds a feasible solution quickly and then continually works on improving it 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 in different domains. We showed this with experiments on a simulated high-dimensional robot arm and a dynamic path planning problem for an outdoor rover.

We then used our novel formulation of A* search to develop an incremental heuristic search algorithm called LPA*. To the best of our knowledge, LPA* is the first incremental heuristic search that is able to operate for a specified solution suboptimality, and when planning for an optimal path LPA* seems to have about the same efficiency as another commonly used incremental heuristic search algorithm, called D* [78]. LPA*, however, has the benefit of being substantially simpler and more closely related to the well-studied A* algorithm. Experimentally, we showed that LPA* speeds up the process of re-planning substantially.

Finally, based on our formulation of A* search, we also developed an algorithm, called Anytime D*, that is both anytime and incremental. The algorithm 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. The algorithm thus combines the benefits of anytime and incremental planners and provides efficient solutions to complex, dynamic search problems under time constraints. We demonstrated this on a simulated robot arm and the problem of dynamic path planning for a rover navigating in unknown and partially known outdoor environments. To the best of our knowledge, Anytime D* is the only heuristic search algorithm to combine the anytime and incremental properties.

Bibliography

[1] E.U. Acar, H. Choset, A.A. Rizzi, P. Atkar, and D. Hull. Morse decompositions for coverage tasks. *International Journal of Robotics Research*, 21:331–344, 2002.

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

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

[4] R. C. Arkin. Motor schema-based mobile robot navigation. *The International Journal of Robotics Research*, pages 92–112, 1989.

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

[6] A. Bagchi and P. K. Srimani. Weighted heuristic search in networks. *Journal of Algorithms*, 6:550–576, 1985.

[7] J. Barraquand, B. Langlois, and J. Latombe. Numerical potential field techniques for robot path planning. *IEEE Trans. on Systems, Man, and Cybernetics*, 22(2):224–241, 1992.

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

[9] J. Borenstein and Y. Koren. The vector field histogram - fast obstacle avoidance for mobile robots. *IEEE Transactions on Robotics and Automation*, 7(3):278–288, 1991.

[10] J. Canny. Constructing roadmaps of semi-algebraic sets I: Completeness. *Artificial Intelligence*, 37:203–222, 1988.

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

[12] S. Chien, R. Knight, A. Stechert, R. Sherwood, and 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.

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

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

[15] S. Edelkamp. Updating shortest paths. In *Proceedings of the European Conference on Artificial Intelligence*, 1998.

[16] S. Edelkamp. Heuristic search planning with BDDs. In *Proceedings of the Workshop "New Results in Planning, Scheduling and Design" (PuK)*, 2000.

[17] S. Edelkamp. Planning with pattern databases. In *Proceedings of the European Conference on Planning (ECP)*, 2001.

[18] S. Even and H. Gazit. Updating distances in dynamic graphs. *Methods of Operations Research*, 49:371–387, 1985.

[19] S. Even and Y. Shiloach. An on-line edge deletion problem. *Journal of the ACM*, 28(1):1–4, 1981.

[20] D. Ferguson. Personal communication, 2005.

[21] D. Ferguson, M. Likhachev, and 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.

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

[23] G. J. Ferrer. *Anytime Replanning Using Local Subplan Replacement.* PhD thesis, University of Virginia, 2002.

[24] E. Feuerstein and A. Marchetti-Spaccamela. Dynamic algorithms for shortest paths in planar graphs. *Theoretical Computer Science*, 116(2):359–371, 1993.

[25] P. Franciosa, D. Frigioni, and R. Giaccio. Semi-dynamic breadth-first search in digraphs. *Theoretical Computer Science*, 250(1–2):201–217, 2001.

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

[27] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Semidynamic algorithms for maintaining single source shortest path trees. *Algorithmica*, 22(3):250–274, 1998.

[28] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *Journal of Algorithms*, 34(2):251–281, 2000.

[29] D. Furcy. *Chapter 5 of Speeding Up the Convergence of Online Heuristic Search and Scaling Up Offline Heuristic Search.* PhD thesis, Georgia Institute of Technology, 2004.

[30] A. Gerevini and 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)*, pages 112–121, 2000.

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

[32] D. Haehnel. Personal communication, 2003.

[33] K. Hammond. Explaining and repairing plans that fail. *Artificial Intelligence*, 45:173–228, 1990.

[34] S. Hanks and D. Weld. A domain-independent algorithm for plan adaptation. *Journal of Artificial Intelligence Research*, 2:319–360, 1995.

[35] 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.

[36] D. Hsu, J.-C. Latombe, and R. Motwani. Path planning in expansive configuration spaces. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 1997.

[37] G. Italiano. Finding paths and deleting edges in directed acyclic graphs. *Information Processing Letters*, 28(1):5–11, 1988.

[38] R. M. Jensen, R. E. Bryant, and M. M. Veloso. SetA*: An efficient BDD-based heuristic search algorithm. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 668–673, 2002.

[39] F. Kabanza, R. Nkambou, and K. Belghith. Path-planning for autonomous training on robot manipulators in space. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1729–1731, 2005. Poster Paper.

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

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

[42] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *International Journal of Robotics Research*, 5(1):90–98, 1986.

[43] P. Klein and S. Subramanian. Fully dynamic approximation schemes for shortest path problems in planar graphs. In *Proceedings of the International Workshop on Algorithms and Data Structures*, pages 443–451, 1993.

[44] J. Koehler. Flexible plan reuse in a formal framework. In C. Bäckström and E. Sandewall, editors, *Current Trends in AI Planning*, pages 171–184. IOS Press, 1994.

[45] S. Koenig. *Chapter 2 of Goal-Directed Acting with Incomplete Information: Acting with Agent-Centered Search*. PhD thesis, Carnegie Mellon University, 1997.

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

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

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

[49] S. Koenig, M. Likhachev, and D. Furcy. Lifelong planning A*. *Artificial Intelligence Journal*, accepted with minor revisions.

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

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

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

[53] J.J. Kuffner and S.M. LaValle. RRT-connect: An efficient approach to single-query path planning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2000.

[54] V. Kumar. Branch-and-bound search. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 1468–1472. New York, NY: Wiley-Interscience, 1992.

[55] J.-C. Latombe. *Robot Motion Planning.* Boston, MA: Kluwer Academic Publishers, 1991.

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

[57] M. Likhachev, G. Gordon, and S. Thrun. ARA*: Anytime A* with provable bounds on sub-optimality. In *Advances in Neural Information Processing Systems (NIPS) 16*. Cambridge, MA: MIT Press, 2003.

[58] M. Likhachev and S. Koenig. Speeding up the parti-game algorithm. In S. Becker, S. Thrun, and K. Obermayer, editors, *Advances in Neural Information Processing Systems (NIPS) 15*. Cambridge, MA: MIT Press, 2002.

[59] M. Likhachev and S. Koenig. A generalized framework for lifelong planning A*. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 99–108, 2005.

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

[61] V. Lumelsky and A. Stepanov. Path planning strategies for point mobile automation moving amidst unknown obstacles of arbitrary shape. *Algorithmica*, 2:403–430, 1987.

[62] A. Moore and C. Atkeson. The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. *Machine Learning*, 21(3):199–233, 1995.

[63] H. Moravec. Certainty grids for mobile robots. In *Proceedings of the NASA/JPL Space Telerobotics Workshop*, 1987.

[64] N. Nilsson. *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill, 1971.

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

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

[67] E. Plaku, K. E. Bekris, B.Y. Chen, A.M. Ladd, and L. Kavraki. Sampling-based roadmap of trees for parallel motion planning. *IEEE Transactions on Robotics*, 21(4):597–608, 2005.

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

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

[70] F. Preparata and M.I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[71] S. Rabin. A* speed optimizations. In M. DeLoura, editor, *Game Programming Gems*, pages 272–287, Rockland, MA, 2000. Charles River Media.

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

[73] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *Theoretical Computer Science*, 158(1–2):233–277, 1996.

[74] H. Rohnert. A dynamization of the all pairs least cost path problem. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, pages 279–286, 1985.

[75] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Englewood Cliffs, NJ: Prentice-Hall, 1995.

[76] R. Simmons. A theory of debugging plans and interpretations. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 94–99, 1988.

[77] P. Spira and A. Pan. On finding and updating spanning trees and shortest paths. *SIAM Journal on Computing*, 4:375–380, 1975.

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

[79] A. Stentz. Map-based strategies for robot navigation in unknown environments. In *AAAI Spring Symposium on Planning with Incomplete Information for Robot Problems*, 1996.

[80] 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), 1990.

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

[82] A. Yahja, A. Stentz, S. Singh, and B. L. Brumitt. Framed-quadtree path planning for mobile robots operating in sparse environments. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 650–655, 1998.

[83] W. Zhang. Complete anytime beam search. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 425–430, 1998.

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

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

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

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