Improved Fast Replanning for Robot Navigation in Unknown Terrain

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
GIT-COGSCI-2002/3

Sven Koenig
College of Computing
Georgia Institute of Technology
Atlanta, GA 30312-0280
skoenig@cc.gatech.edu

Maxim Likhachev
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
maxim+@cs.cmu.edu

Abstract

Mobile robots often operate in domains that are only incompletely known, for example, when they have to move from given start coordinates to given goal coordinates in unknown terrain. In this case, they need to be able to replan quickly as their knowledge of the terrain changes. Stentz' Focussed Dynamic A* is a heuristic search method that repeatedly determines a shortest path from the current robot coordinates to the goal coordinates while the robot moves along the path. It is able to replan one to two orders of magnitudes faster than planning from scratch since it modifies previous search results locally. Consequently, it has been extensively used in mobile robotics. In this article, we introduce an alternative to Focussed Dynamic A* that implements the same navigation strategy but is algorithmically different. Focussed Dynamic A* Lite is simple, easy to understand, easy to analyze and easy to extend, yet is more efficient than Focussed Dynamic A*. We believe that our results will make D*-like replanning methods even more popular and enable robotics researchers to adapt them to additional applications.

1 Introduction

Mobile robots often operate in domains that are only incompletely known. In this article, we study a goal-directed navigation problem in unknown terrain where a mobile robot has to move from its current coordinates to given goal coordinates. Robotics researchers have investigated various navigation strategies to solve it, including the well-known bug algorithms [LS87]. In this paper, we study the following navigation strategy: The robot always plans a shortest path from its current coordinates to the goal coordinates under the assumption that unknown terrain is traversable. (It can utilize initial knowledge of the terrain in case it is available.) If it observes obstacles as it follows this path, it enters them into its map and then repeats the procedure, until it eventually reaches the goal coordinates or all paths to them are untraversable. This navigation strategy is an example of sensor-based motion planning [CB94, CB95]. If we model the navigation problem as a navigation problem on an eight-connected grid with edges that are either traversable (with cost one) or untraversable, it must terminate because the robot either follows the planned path to the goal vertex or increases its knowledge about the true edge costs, which can happen only once for each edge.

To implement the navigation strategy, the robot needs to replan a shortest path from its current vertex to the goal vertex whenever it detects that its current path is untraversable. The robot could use conventional graph-search methods. However, the resulting search times can be on the order of minutes for the large graphs that are often used, which adds up to substantial idle times [Ste94]. Focussed Dynamic A* (D*) [Ste95] is probably the most popular solution to this problem at the moment since it combines the efficiency of heuristic and incremental searches, yet still finds shortest paths. It achieves a speedup of one to two orders of magnitudes(!) over repeated A* [Pea85] searches by modifying previous search results locally. D* has been extensively used on real robots. This includes indoor Nomad robots [KTH01] as well as outdoor HMMWVs and the UGV Demo II vehicles as part of the DARPA Unmanned Ground Vehicle program [SH95]. It is currently also being integrated into Mars Rover prototypes and tactical mobile robot prototypes for urban reconnaissance [HMC99, MXH$^+$00, TDD$^+$00]. D* is also used as part of other software, including the GRAMMPS mission planner for multiple robots [BS98].

However, D* is very complex and thus hard to understand, analyze, and extend. For example, while D* has been widely used as a black-box method, it has not been extended by other researchers. Building on our Lifelong Planning A* method [KL01a], we therefore present D* Lite, a novel replanning method that implements the same navigation strategy as D* but is algorithmically different. Lifelong Planning A* is an incremental version of A* and thus very similar to A*. It is efficient and has well-understood properties (for example, we can prove theorems about its similarity to A* and its efficiency). This also allows us to extend it easily, for example, to use inadmissible heuristics and different tie-breaking criteria to gain efficiency. Since D* Lite is based on LPA*, it is simple, easy to understand, easy to analyze and easy to extend. It also inherits all of the properties of LPA* and can be extended in the same way as LPA*. It has more than thirty percent fewer lines of code than D* (without any coding tricks), uses only one tie-breaking criterion when comparing priorities which simplifies the maintenance of the priorities, and does not need nested if-statements with complex conditions that occupy up to three lines each which simplifies the analysis of the program flow. Yet, our experiments show that D* Lite is more efficient than D*. The simplicity of D* Lite is important for optimizing it, integrating it into complete robot architectures, and using it to solve navigation tasks other than goal-directed navigation in unknown terrain. We also provide a mathematically rigorous analysis of D* Lite, probably the most rigorous analysis of any incremental heuristic search method that has been applied to robot navigation in unknown terrain.

In Section 2, we motivate the ideas behind D* Lite. In Section 3, we then introduce LPA* and describe how it works. In Section 4, we use LPA* to develop two versions of D* Lite, the basic version and the final version, and describe how they can be optimized. In Section 5, we illustrate the operation of the two versions of D* Lite with an example. In Section 6 finally, we present experimental results that compare D* Lite against several other search methods, both for goal-directed navigation in unknown terrain and for mapping of unknown terrain. The appendix contains the proofs of the theorems stated in the main text. (As explained in the appendix, the proofs in the appendix are for a version of LPA* that searches from the goal vertex to the start vertex.)

2 Motivation

Consider a robot-navigation task in unknown terrain, where the robot always observes which of its eight adjacent cells are traversable and then moves with cost one to one of them. The robot starts at the start cell and has to move to the goal cell. It always computes a shortest path from its current cell to the goal cell under the assumption that cells with unknown traversability status are traversable. It then follows this path until it reaches the goal cell, in which case it stops successfully, or it observes additional untraversable cells, in which case it recomputes a shortest path from its current cell to the goal cell. Figure 1 illustrates this navigation strategy. Figure 1 (top) shows the terrain in which the robot has to move from cell B1 to cell E3, and Figure 1 (bottom) shows, before each movement of the robot, the untraversable cells that it knows about together with the path that it attempts to follow. White cells are known to be traversable, black cells are known to be untraversable, and gray cells have unknown traversability. The robot starts in cell B1. Since all costs are one, the shortest path from the current cell B1 to the goal cell E3 seems to be via cells C1 and D2. The robot then moves to cell C1 and discovers that cell D2 is untraversable. Now, the shortest path from the current cell C1 to the goal cell E3 seems to be via cells D1 and E2. The robot then follows this path to the goal cell.

Figure 2 shows the beginning of a larger example. It shows the goal distances (that is, the length of a shortest path to a goal cell) of all traversable cells and the shortest paths both before and after the robot has moved along the path and discovered the first untraversable cell it did not know about. Cells whose goal distances have changed are shaded gray. The goal distances are

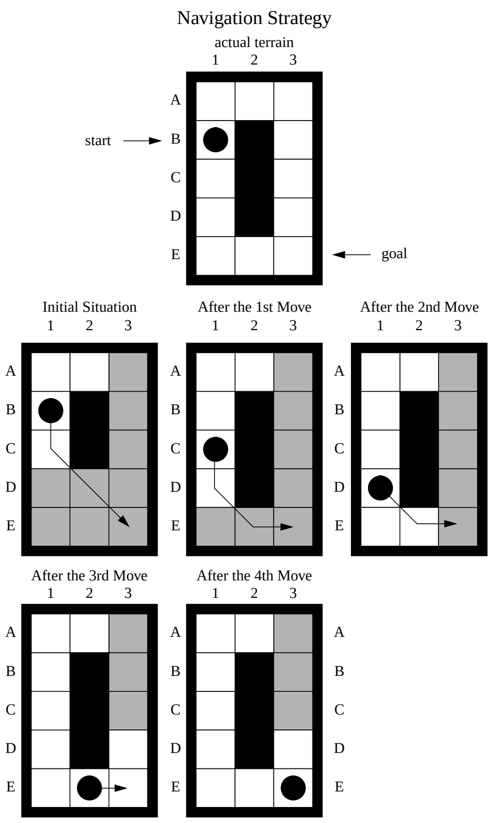


Figure 1: Illustration of the Navigation Strategy.

important because one can easily determine a shortest path from the current cell of the robot to the goal vertex by greedily decreasing the goal distances once the goal distances have been computed. Notice that the goal distances of only about 15 percent of the cells have changed, and most of the changed goal distances are irrelevant for recalculating a shortest path from the current cell of the robot to the goal cell. Thus, one can efficiently recalculate a shortest path from the current cell of the robot to the goal cell by recalculating only those goal distances that have changed (or have not been calculated before) *and* are relevant for recalculating the shortest path. This is what D* Lite does. The challenge is to identify these cells efficiently.

3 Lifelong Planning A*

We first describe Lifelong Planning A* (LPA*) [KL01a], our incremental heuristic search method that repeatedly determines shortest paths between two given vertices as the edge costs of a graph change. We chose its name in analogy to “lifelong learning” [Thr98] because it reuses information from previous searches. We later use LPA* to develop D* Lite.

LPA* is shown in Figure 3. It is an incremental search method that uses heuristics to focus

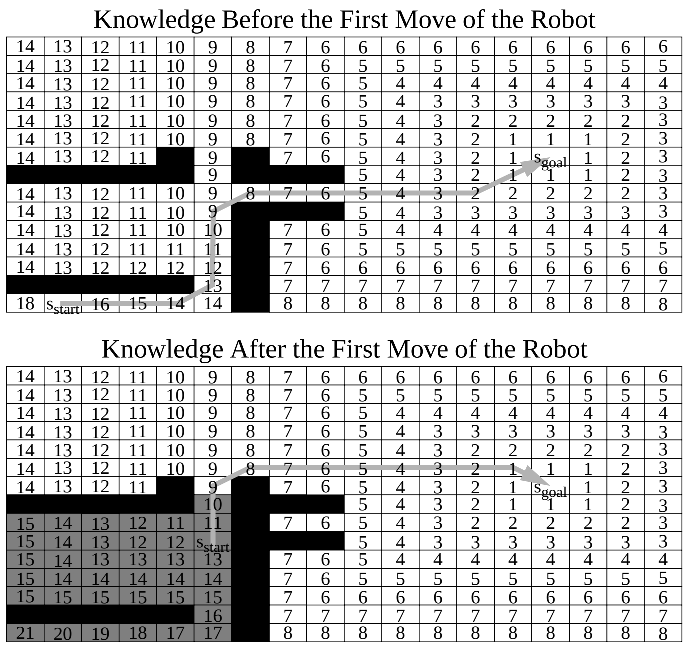

Knowledge Before the First Move of the Robot
s_{start}
s_{goal}
Knowledge After the First Move of the Robot
s_{start}
s_{goal}

Figure 2: Simple Example (Part 1).

the search. An incremental search tends to only recalculate those start distances (that is, distance from the start vertex to a vertex) that have changed (or have not been calculated before) [FMSN00] and a heuristic search tends to only recalculate those start distances that are relevant for recalculating a shortest path from the start vertex to the goal vertex [Pea85]. LPA* thus recalculates only very few start distances.

LPA* is an incremental version of A* [Pea85], probably the most popular search method in artificial intelligence. It applies to the same graph search problems as A* and shares many similarities with it. For example, it uses heuristic knowledge in form of approximations of the goal distances to focus the search. LPA* generalizes A* because its first search is the same as that of A* and it performs an incremental A* search afterwards. LPA* also generalizes a version of DynamicSWSF-FP [RR96a] because it reduces to it if the heuristics are uninformed (that is, zero).

3.1 Lifelong Planning A*: Notation

Lifelong Planning A* (LPA*) applies to finite graph search problems on known graphs whose edge costs increase or decrease over time (which can also be used to model edges or vertices that are added or deleted). Thus, it searches directed graphs, just like A*. S denotes the finite set of vertices of the graph. $Succ(s) \subseteq S$ denotes the set of successors of vertex $s \in S$ in the graph. Similarly, $Pred(s) \subseteq S$ denotes the set of predecessors of vertex $s \in S$ in the graph. $0 < c(s, s') \leq \infty$ denotes the cost of moving from vertex s to vertex $s' \in Succ(s)$. LPA*

The pseudocode uses the following functions to manage the priority queue: U.TopKey() returns the smallest priority of all vertices in priority queue U. (If U is empty, then U.TopKey() returns $[\infty;\infty]$.) U.Pop() deletes the vertex with the smallest priority in priority queue U and returns the vertex. U.Insert(s,k) inserts vertex s into priority queue U with priority k. Finally, U.Remove(s) removes vertex s from priority queue U.

```
procedure CalculateKey($s$)
{01} return $[\min(g(s), rhs(s)) + h(s, s_{goal}); \min(g(s), rhs(s))]$;
procedure Initialize()
{02} $U = \emptyset$;
{03} for all $s \in S$ $rhs(s) = g(s) = \infty$;
{04} $rhs(s_{start}) = 0$;
{05} U.Insert($s_{start}$, CalculateKey($s_{start}$));
procedure UpdateVertex($u$)
{06} if ($u \neq s_{start}$) $rhs(u) = \min_{s' \in \text{Pred}(u)}(g(s') + c(s', u))$;
{07} if ($u \in U$) U.Remove($u$);
{08} if ($g(u) \neq rhs(u)$) U.Insert($u$, CalculateKey($u$));
procedure ComputeShortestPath()
{09} while (U.TopKey() $\dot{<}$ CalculateKey($s_{goal}$) OR $rhs(s_{goal}) \neq g(s_{goal})$)
{10}    $u$ = U.Pop();
{11}    if ($g(u) > rhs(u)$)
{12}       $g(u) = rhs(u)$;
{13}       for all $s \in$ Succ($u$) UpdateVertex($s$);
{14}    else
{15}       $g(u) = \infty$;
{16}       for all $s \in$ Succ($u$) $\cup \{u\}$ UpdateVertex($s$);
procedure Main()
{17} Initialize();
{18} forever
{19}    ComputeShortestPath();
{20}    Wait for changes in edge costs;
{21}    for all directed edges $(u, v)$ with changed edge costs
{22}       Update the edge cost $c(u, v)$;
{23}       UpdateVertex($v$);
```

Figure 3: (Forward Version of) Lifelong Planning A*.

always determines a shortest path from a given start vertex $s_{start} \in S$ to a given goal vertex $s_{goal} \in S$, knowing both the topology of the graph and the current edge costs. We use $gd(s)$ to denote the goal distance of vertex $s \in S$, that is, the cost of a shortest path from s to s_{goal}. Similarly, we use $g^*(s)$ to denote the start distance of vertex $s \in S$, that is, the cost of a shortest path from s_{start} to s. The start distances satisfy the following relationship:

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

3.2 Lifelong Planning A*: The Variables

LPA* maintains an estimate $g(s)$ of the start distance $g^*(s)$ of each vertex s. These values directly correspond to the g-values of an A* search. The initial search of LPA* calculates the g-values of each vertex in exactly the same order as A*. LPA* then carries the g-values forward from search to search. LPA* also maintains a second kind of estimate of the start distances. The rhs-values are one-step lookahead values based on the g-values and thus potentially better informed than the g-values. They always satisfy the following relationship (Invariant 1) according to Theorem 5 in the appendix:

$$rhs(s) = \begin{cases} 0 & \text{if } s = s_{start} \\ \min_{s' \in Pred(s)}(g(s') + c(s', s)) & \text{otherwise.} \end{cases} \tag{2}$$

A vertex s is called locally consistent iff $g(s) = rhs(s)$. This concept is similar to satisfying the Bellman equation for undiscounted deterministic sequential decision problems [Bel57]. A vertex s is called locally inconsistent iff $g(s) \neq rhs(s)$. If all vertices are locally consistent then all of their g-values satisfy

$$g(s) = \begin{cases} 0 & \text{if } s = s_{start} \\ \min_{s' \in Pred(s)}(g(s') + c(s', s)) & \text{otherwise.} \end{cases} \tag{3}$$

A comparison to Equation 1 shows that all g-values are equal to their respective start distances. Thus, the g-values of all vertices equal their start distances iff all vertices are locally consistent. This concept is important because one can then trace back a shortest path from s_{start} to any vertex u by always moving from the current vertex s, starting at u, to any predecessor s' that minimizes $g(s') + c(s', s)$ until s_{start} is reached (ties can be broken arbitrarily). However, LPA* does not make every vertex locally consistent after some of the edge costs have changed. Instead, it uses heuristics $h(s, s_{goal})$ to focus the search and updates only the g-values that are relevant for computing a shortest path. $h(s, s')$ approximates the distance between vertex s and s'. Thus, the heuristics approximate the goal distances of the vertices. The heuristics need to be nonnegative and (forward) consistent [Pea85], that is, obey the triangle inequality $h(s_{goal}, s_{goal}) = 0$ and $h(s, s_{goal}) \leq c(s, s') + h(s', s_{goal})$ for all vertices $s \in S$ and $s' \in Succ(s)$.

LPA* maintains a priority queue, like A*. The priority queue of LPA* always contains exactly the locally inconsistent vertices (Invariant 2) according to Theorem 6 in the appendix. These are the vertices whose g-values LPA* potentially needs to update to make them locally consistent. The keys of the vertices in the priority queue roughly correspond to the f-values used by A*, and LPA* always recalculates the g-value of the vertex ("expands the vertex") in the priority queue with the smallest key. This is similar to A* that always expands the vertex in the priority queue with the smallest f-value. By expanding a vertex, we mean executing {10-16} (numbers in brackets refer to line numbers in Figure 3). The key $k(s)$ of vertex s is a vector with two components:

$$k(s) = [k_1(s); k_2(s)], \tag{4}$$

where $k_1(s) = \min(g(s), rhs(s)) + h(s, s_{goal})$ and $k_2(s) = \min(g(s), rhs(s))$ $\{1\}$. This is the vector that CalculateKey() calculates. The priority of a vertex in the priority queue is always the same as its key (Invariant 3) according to Theorem 7 in the appendix. Keys are compared according to a lexicographic ordering. For example, a key $k(s)$ is smaller than or equal to a key $k'(s)$, denoted by $k(s) \dot{\leq} k'(s)$, iff either $k_1(s) < k'_1(s)$ or ($k_1(s) = k'_1(s)$ and $k_2(s) \leq k'_2(s)$). The first component of the keys $k_1(s)$ corresponds directly to the f-values $f(s) := g^*(s) + h(s, s_{goal})$ used by A* because both the g-values and rhs-values of LPA* correspond to the g-values of A* and the h-values of LPA* correspond to the h-values of A*. The second component of the keys $k_2(s)$ corresponds to the g-values of A*. LPA* always expands the vertex in the priority queue with the smallest k_1-value, which corresponds to the f-value of an A* search, breaking ties in favor of the vertex with the smallest k_2-value, which corresponds to the g-value of an A* search. This is similar to A* that always expands the vertex in the priority queue with the smallest f-value, breaking ties among the vertices on the same branch of the search tree in favor of the vertex with the smallest g-value. The resulting behavior of LPA* and A* is also similar. The keys of the vertices expanded by LPA* are nondecreasing over time according to Theorem 11 in the appendix. This is similar to A* since the f-values of the vertices expanded by A* are also nondecreasing over time (since the heuristics are consistent) and the g-values are also nondecreasing for vertices with the same f-values (since it grows the search tree).

3.3 Lifelong Planning A*: The Method

LPA* is shown in Figure 3. The main function Main() first calls Initialize() to initialize the search problem $\{17\}$. Initialize() sets the initial g-values of all vertices to infinity and sets their rhs-values according to Equation 2 $\{03\text{-}04\}$. Thus, initially s_{start} is the only locally inconsistent vertex and is inserted into the otherwise empty priority queue with a key calculated according to Equation 4 $\{05\}$. This initialization guarantees that the first call to ComputeShortestPath() $\{19\}$ performs exactly an A* search, that is, expands exactly the same vertices as A* in exactly the same order, provided that A* breaks ties among vertices with the same f-values suitably. Note that, in an actual implementation, Initialize() only needs to initialize a vertex when it encounters it during the search and thus does not need to initialize all vertices up front. This is important because the number of vertices can be large and only a few of them might be reached during the search. LPA* then waits for changes in edge costs $\{20\}$. To maintain Invariants 1-3 if some edge costs have changed, it calls UpdateVertex() $\{23\}$ to update the rhs-values and keys of the vertices potentially affected by the changed edge costs as well as their membership in the priority queue if they become locally consistent or inconsistent, and finally recalculates a shortest path $\{19\}$ by calling ComputeShortestPath(), and iterates.

In the following, we give a high-level explanation of how ComputeShortestPath() works, appealing to the intuition of the reader. We prove theorems in the appendix that make our explanations more concrete. ComputeShortestPath() repeatedly expands locally inconsistent vertices in the order of their priorities $\{10\}$. A locally inconsistent vertex s is called locally overconsistent iff $g(s) > rhs(s)$. When ComputeShortestPath() expands a locally overconsistent vertex $\{$12-13$\}$, then it turns out to hold that $rhs(s) = g^*(s)$, which implies that $k(s) = [f(s); g^*(s)]$. During the expansion of vertex s, ComputeShortestPath() sets the g-value of vertex s to its rhs-value and thus its start distance $\{$12$\}$, which is the desired value and also makes the vertex locally consistent. Its g-value then no longer changes until ComputeShortestPath() terminates according to Theorem 13 in the appendix. A locally inconsistent vertex s is called locally underconsistent iff $g(s) < rhs(s)$. When ComputeShortestPath() expands a locally underconsistent vertex $\{$15-16$\}$, then it simply sets the g-value of the vertex to infinity $\{$15$\}$. This makes the vertex either locally consistent or overconsistent. If the expanded vertex was locally overconsistent, then the change of its g-value can affect the local consistency of its successors $\{$13$\}$. Similarly, if the expanded vertex was locally underconsistent, then it and its successors can be affected $\{$16$\}$. To maintain Invariants 1-3, ComputeShortestPath() therefore updates rhs-values of these vertices, checks their local consistency, and adds them to or removes them from the priority queue accordingly $\{$06-08$\}$.

LPA* expands vertices until s_{goal} is locally consistent and the key of the vertex to expand next is no smaller than the key of s_{goal}. This is similar to A* that expands vertices until it expands s_{goal} at which point in time the g-value of s_{goal} equals its start distance and the f-value of the vertex to expand next is no smaller than the f-value of s_{goal}. LPA* expands a vertex at most twice, namely at most once when it is locally underconsistent and at most once when it is locally overconsistent, according to Theorem 16 in the appendix. This property implies that LPA* is guaranteed to terminate after a number of vertex expansions that is at most twice the number of vertex expansions of A* since A* expands each vertex at most once. Thus, even though it is known that there are always cases where incremental search is not more efficient than search from scratch [NK95], LPA* can never be much worse than A* even in situations that are exceptionally bad for LPA*.

If $g(s_{goal}) = \infty$ after the search, then there is no finite-cost path from s_{start} to s_{goal}. Otherwise, one can trace back a shortest path from s_{start} to s_{goal} by always moving from the current vertex s, starting at s_{goal}, to any predecessor s' that minimizes $g(s') + c(s', s)$ until s_{start} is reached (ties can be broken arbitrarily) according to Theorem 16 in the appendix. This is similar to what A* can do if it does not use backpointers. Thus, LPA* does not explicitly maintain a search tree. Instead, it uses the g-values to encode it implicitly.

A more detailed and formal description of LPA* can be found in [KL01b]. This includes its comparison to A* and proofs of its properties, for example that LPA* is efficient because it performs incremental searches and thus calculates only those g-values that have been affected by cost changes or have not been calculated yet in previous searches and that LPA* is also efficient because it performs heuristic searches and thus calculates only the g-values of those vertices that are important to determine a shortest path.

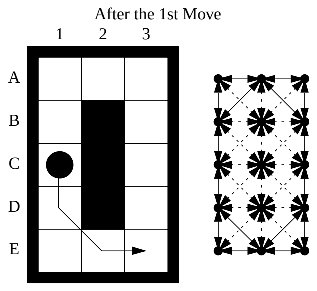


Figure 4: Known Terrain and Corresponding Graph.

4 D* Lite

So far, we have described our LPA*, that repeatedly determines shortest paths between the start vertex and the goal vertex as the edge costs of a graph change. We now use LPA* to develop D* Lite [KL02], that repeatedly determines shortest paths between the current vertex of the robot and the goal vertex as the edge costs of a graph change while the robot moves towards the goal vertex. D* Lite does not make any assumptions about how the edge costs change, whether they go up or down, whether they change close to the current vertex of the robot or far away from it, or whether they change in the world or only because the robot revised its initial estimates. The goal-directed navigation problem in unknown terrain then is a special case of this problem, where the graph is an eight-connected grid whose edge costs are initially one and change to infinity when the robot discovers that they cannot be traversed. Figure 4, for example, shows the untraversable cells that the robot knows about after its first movement for the example from Figure 1 together with the corresponding graph. Solid edges have cost one and dashed edges have infinite cost.

We now first describe a simple version of D* Lite and then a more sophisticated version. Because both versions of D* Lite are based on LPA*, they share many properties with A* and are efficient.

4.1 The Basic Version of D* Lite

We have already argued that many goal distances remain unchanged as the robot moves to the goal vertex and observes obstacles in the process. Thus, we can use a version of LPA* for the goal-directed navigation problem in unknown terrain. We first need to switch the search direction of LPA*. The version presented in Figure 3 searches from the start vertex to the goal vertex and thus its g-values are estimates of the start distances. The version presented in Figure 5 searches from the goal vertex to the start vertex and thus its g-values are estimates of the goal distances. It was derived from the original graph by reversing all edges of the graph and exchanging the start and goal vertex. The heuristics $h(s, s')$ now need to be nonnegative and backward consistent, that is, obey $h(s_{start}, s_{start}) = 0$ and $h(s_{start}, s) \leq h(s_{start}, s') + c(s', s)$

The pseudocode uses the following functions to manage the priority queue: U.TopKey() returns the smallest priority of all vertices in priority queue U. (If U is empty, then U.TopKey() returns $[\infty;\infty]$.) U.Pop() deletes the vertex with the smallest priority in priority queue U and returns the vertex. U.Insert(s,k) inserts vertex s into priority queue U with priority k. Finally, U.Remove(s) removes vertex s from priority queue U.

```
procedure CalculateKey(s)
{01} return [min(g(s), rhs(s)) + h(s_start, s); min(g(s), rhs(s))];
procedure Initialize()
{02} U = ∅;
{03} for all s ∈ S rhs(s) = g(s) = ∞;
{04} rhs(s_goal) = 0;
{05} U.Insert(s_goal, CalculateKey(s_goal));
procedure UpdateVertex(u)
{06} if (u ≠ s_goal) rhs(u) = min_{s' ∈ Succ(u)}(c(u, s') + g(s'));
{07} if (u ∈ U) U.Remove(u);
{08} if (g(u) ≠ rhs(u)) U.Insert(u, CalculateKey(u));
procedure ComputeShortestPath()
{09} while (U.TopKey() <˙ CalculateKey(s_start) OR rhs(s_start) ≠ g(s_start))
{10}   u = U.Pop();
{11}   if (g(u) > rhs(u))
{12}     g(u) = rhs(u);
{13}     for all s ∈ Pred(u) UpdateVertex(s);
{14}   else
{15}     g(u) = ∞;
{16}     for all s ∈ Pred(u) ∪ {u} UpdateVertex(s);
procedure Main()
{17} Initialize();
{18} forever
{19}   ComputeShortestPath();
{20}   Wait for changes in edge costs;
{21}   for all directed edges (u, v) with changed edge costs
{22}     Update the edge cost c(u, v);
{23}     UpdateVertex(u);
```

Figure 5: Backward Version of Lifelong Planning A*.

for all vertices $s \in S$ and $s' \in \text{Pred}(s)$. More generally, since the robot moves and thus changes s_{start}, the heuristics needs to satisfy this property for all $s_{start} \in S$. If $g(s_{start}) = \infty$ after the search, then there is no finite-cost path from s_{start} to s_{goal}. Otherwise, one can follow a shortest path from s_{start} to s_{goal} by always moving from the current vertex s, starting at s_{start}, to any successor s' that minimizes $c(s,s') + g(s')$ until s_{goal} is reached (ties can be broken arbitrarily). To solve the goal-directed navigation problem in unknown terrain, the CalculateKey$()$, Initialize$()$, UpdateVertex$()$, and ComputeShortestPath$()$ functions can remain unchanged. However, the Main$()$ function needs to get extended so that it moves the robot and then recalculates the priorities of the vertices in the priority queue appropriately. This is necessary because the heuristics change when the robot moves, since they are computed with respect to the current vertex of the robot. This only changes the priorities of the vertices in the priority queue but not which vertices are consistent and thus in the priority queue. Figure 6 shows the resulting method, called the basic version of D* Lite.

The main function Main() first calls Initialize() to initialize the search problem $\{17'\}$. Initialize() sets the initial g-values of all vertices to infinity and sets their rhs-values according to the equivalent of Equation 2 $\{03'\text{-}04'\}$. Thus, initially s_{goal} is the only locally inconsistent vertex

The pseudocode uses the following functions to manage the priority queue: U.TopKey() returns the smallest priority of all vertices in priority queue U. (If U is empty, then U.TopKey() returns $[\infty;\infty]$.) U.Pop() deletes the vertex with the smallest priority in priority queue U and returns the vertex. U.Insert(s,k) inserts vertex s into priority queue U with priority k. U.Update(s,k) changes the priority of vertex s in priority queue U to k. (It does nothing if the current priority of vertex s already equals k.) Finally, U.Remove(s) removes vertex s from priority queue U.

```
procedure CalculateKey($s$)
{01’} return $[\min(g(s), rhs(s)) + h(s_{start}, s); \min(g(s), rhs(s))]$;
procedure Initialize()
{02’} $U = \emptyset$;
{03’} for all $s \in S$ $rhs(s) = g(s) = \infty$;
{04’} $rhs(s_{goal}) = 0$;
{05’} U.Insert($s_{goal}$, CalculateKey($s_{goal}$));
procedure UpdateVertex($u$)
{06’} if $(u \neq s_{goal})$ $rhs(u) = \min_{s' \in \text{Succ}(u)}(c(u,s') + g(s'))$;
{07’} if $(u \in U)$ U.Remove($u$);
{08’} if $(g(u) \neq rhs(u))$ U.Insert($u$, CalculateKey($u$));
procedure ComputeShortestPath()
{09’} while (U.TopKey() $\dot{<}$ CalculateKey($s_{start}$) OR $rhs(s_{start}) \neq g(s_{start})$)
{10’}     $u =$ U.Pop();
{11’}     if $(g(u) > rhs(u))$
{12’}         $g(u) = rhs(u)$;
{13’}         for all $s \in$ Pred($u$) UpdateVertex($s$);
{14’}     else
{15’}         $g(u) = \infty$;
{16’}         for all $s \in$ Pred($u$) $\cup \{u\}$ UpdateVertex($s$);
procedure Main()
{17’} Initialize();
{18’} ComputeShortestPath();
{19’} while $(s_{start} \neq s_{goal})$
{20’}     /* if $(g(s_{start}) = \infty)$ then there is no known path */
{21’}     $s_{start} = \arg\min_{s' \in \text{Succ}(s_{start})}(c(s_{start}, s') + g(s'))$;
{22’}     Move to $s_{start}$;
{23’}     Scan graph for changed edge costs;
{24’}     if any edge costs changed
{25’}         for all directed edges $(u,v)$ with changed edge costs
{26’}             Update the edge cost $c(u,v)$;
{27’}             UpdateVertex($u$);
{28’}         for all $s \in U$
{29’}             U.Update($s$, CalculateKey($s$));
{30’}         ComputeShortestPath();
```

Figure 6: D* Lite: Basic Version.

and is inserted into the otherwise empty priority queue with a key calculated according to the equivalent of Equation 4 {05’}. Note that, in an actual implementation, Initialize() only needs to initialize a vertex when it encounters it during the search and thus does not need to initialize all vertices up front. This is important because the number of vertices can be large and only a few of them might be reached during the search. The basic version of D* Lite then computes a shortest path from the current vertex of the robot s_{start} to the goal vertex {18’}. If the robot has not reached the goal vertex yet {19’}, it makes one transition along the shortest path and updates s_{start} to reflect the current vertex of the robot {21’-22’}. (In the pseudocode, we have included a comment on how the robot can detect that there is no path but do not prescribe what it should do in this case. For the goal-directed navigation problem in unknown terrain, for example, it should stop and announce that there is no path since obstacles do not disappear.) It then scans for changes in edge costs {23'}. To maintain Invariants 1-3 if some edge costs have changed, it calls UpdateVertex() {27'} to update the rhs-values and keys of the vertices potentially affected by the changed edge costs as well as their membership in the priority queue if they become locally consistent or inconsistent. Finally, it updates the priorities of all vertices in the priority queue {28'-29'}, recalculates a shortest path {30'}, and iterates.

In the appendix, we prove the correctness of the basic version of D* Lite:

Theorem 1 (= Theorem 17 in the appendix) *ComputeShortestPath() of the basic version of D* Lite expands a vertex at most twice, namely at most once when it is locally underconsistent and at most once when it is locally overconsistent, and thus terminates. One can then follow a shortest path from s_{start} to s_{goal} by always moving from the current vertex s, starting at s_{start}, to any successor s' that minimizes $c(s, s') + g(s')$ until s_{goal} is reached (ties can be broken arbitrarily).*

4.2 The Final Version of D* Lite

The basic version of D* Lite has the disadvantage that the repeated reordering of the priority queue {28'-29'} can be expensive since the priority queue often contains a large number of vertices. The final version of D* Lite, shown in Figure 7, uses a method derived from D* [Ste95] to avoid having to reorder the priority queue. Differences to the basic version of D* Lite are shown in bold. The heuristics $h(s, s')$ now need to be nonnegative and forward-backward consistent, that is, obey $h(s, s'') \leq h(s, s') + h(s', s'')$ for all vertices $s, s', s'' \in S$. They also need to be admissible no matter what the goal vertex is, that is, obey $h(s, s') \leq c^*(s, s')$ for all vertices $s, s' \in S$, where $c^*(s, s')$ denotes the cost of a shortest path from vertex $s \in S$ to vertex $s' \in S$. Theorem 4 in the appendix proves that heuristics with these properties also have the property that heuristics for the basic version of D* Lite need to satisfy. Yet, these properties are not overly restrictive since Theorem 3 in the appendix proves that they are guaranteed to hold if the heuristics were derived by relaxing the search problem, which will almost always be the case and holds for all heuristics used in this article.

The final version of D* Lite uses priorities that are lower bounds on the priorities that the basic version of D* Lite uses for the corresponding vertices. They are initialized in the same way as the basic version of D* Lite initializes them. After the robot has moved from vertex s to some vertex s' where it detects changes in edge costs, the first element of the priorities can have decreased by at most $h(s, s')$. (The second component does not depend on the heuristics and thus remains unchanged.) Thus, in order to maintain lower bounds, D* Lite needs to subtract $h(s, s')$ from the first element of the priorities of all vertices in the priority queue. However, since $h(s, s')$ is the same for all vertices in the priority queue, the order of the vertices in the priority queue does not change if the subtraction is not performed. Then, when new priorities are computed, their first components are by $h(s, s')$ too small relative to the priorities in the

The pseudocode uses the following functions to manage the priority queue: U.TopKey$()$ returns the smallest priority of all vertices in priority queue U. (If U is empty, then U.TopKey$()$ returns $[\infty; \infty]$.) U.Pop$()$ deletes the vertex with the smallest priority in priority queue U and returns the vertex. U.Insert(s, k) inserts vertex s into priority queue U with priority k. Finally, U.Remove(s) removes vertex s from priority queue U.

```
procedure CalculateKey(s)
{01”} return [min(g(s), rhs(s)) + h(s_start, s) + k_m; min(g(s), rhs(s))];
procedure Initialize()
{02”} U = ∅;
{03”} k_m = 0;
{04”} for all s ∈ S rhs(s) = g(s) = ∞;
{05”} rhs(s_goal) = 0;
{06”} U.Insert(s_goal, CalculateKey(s_goal));
procedure UpdateVertex(u)
{07”} if (u ≠ s_goal) rhs(u) = min_{s'∈Succ(u)}(c(u, s') + g(s'));
{08”} if (u ∈ U) U.Remove(u);
{09”} if (g(u) ≠ rhs(u)) U.Insert(u, CalculateKey(u));
procedure ComputeShortestPath()
{10”} while (U.TopKey() <̇ CalculateKey(s_start) OR rhs(s_start) ≠ g(s_start))
{11”}     k_old = U.TopKey();
{12”}     u = U.Pop();
{13”}     if (k_old <̇ CalculateKey(u))
{14”}         U.Insert(u, CalculateKey(u));
{15”}     else if (g(u) > rhs(u))
{16”}         g(u) = rhs(u);
{17”}         for all s ∈ Pred(u) UpdateVertex(s);
{18”}     else
{19”}         g(u) = ∞;
{20”}         for all s ∈ Pred(u) ∪ {u} UpdateVertex(s);
procedure Main()
{21”} s_last = s_start;
{22”} Initialize();
{23”} ComputeShortestPath();
{24”} while (s_start ≠ s_goal)
{25”}     /* if (g(s_start) = ∞) then there is no known path */
{26”}     s_start = arg min_{s'∈Succ(s_start)}(c(s_start, s') + g(s'));
{27”}     Move to s_start;
{28”}     Scan graph for changed edge costs;
{29”}     if any edge costs changed
{30”}         k_m = k_m + h(s_last, s_start);
{31”}         s_last = s_start;
{32”}         for all directed edges (u, v) with changed edge costs
{33”}             Update the edge cost c(u, v);
{34”}             UpdateVertex(u);
{35”}         ComputeShortestPath();
```

Figure 7: D* Lite: Final Version.

priority queue. Thus, $h(s, s')$ has to be added to their first components every time some edge costs change. If the robot moves again and then detects cost changes again, then the constants need to get added up. We do this in the variable k_m $\{30”\}$. Thus, whenever new priorities are computed, the variable k_m has to be added to their first components, as done in $\{01”\}$. Then, the order of the vertices in the priority queue does not change after the robot moves and the priority queue does not need to get reordered. The priorities, on the other hand, are always lower bounds on the corresponding priorities of the basic version of D* Lite after the first component of the priorities of the basic version of D* Lite has been increased by the current value of k_m, that is, lower bounds on the values calculated by CalculateKey() {01"}. We exploit this property by changing ComputeShortestPath() as follows. After ComputeShortestPath() has removed a vertex u with the smallest priority $k_{old} =$ U.TopKey() from the priority queue {12"}, it now uses CalculateKey() to compute the priority that it should have had. If $k_{old} \dot{<}$CalculateKey(u) then it reinserts the removed vertex with the priority calculated by CalculateKey() into the priority queue {13"-14"}. Thus, it remains true that the priorities of all vertices in the priority queue are lower bounds on the corresponding priorities of the basic version of D* Lite after the first components of the priorities of the basic version of D* Lite have been increased by the current value of k_m. If $k_{old} \dot{\geq}$CalculateKey(u), then it holds that $k_{old} \doteq$CalculateKey(u) since k_{old} was a lower bound on the value returned by CalculateKey(). In this case, ComputeShortestPath() performs the same operations for vertex u as ComputeShortestPath() of the basic version of D* Lite {15"-20"}. ComputeShortestPath() performs these operations for vertices in the exact same order as ComputeShortestPath() of the basic version of D* Lite, which implies that the final version of D* Lite shares many properties with the basic version of D* Lite, including its correctness:

Theorem 2 (= Theorem 18 in the appendix) *ComputeShortestPath() of the final version of D* Lite expands a vertex at most twice, namely at most once when it is locally underconsistent and at most once when it is locally overconsistent, and thus terminates. One can then follow a shortest path from s_{start} to s_{goal} by always moving from the current vertex s, starting at s_{start}, to any successor s' that minimizes $c(s, s') + g(s')$ until s_{goal} is reached (ties can be broken arbitrarily).*

5 An Example

We illustrate D* Lite using the two eight-connected grids from Figure 2. To make the search methods comparable, their search always starts at s_{goal} and proceeds towards the current cell of the robot. Furthermore, we consider the current cell of the robot to be expanded by all search methods. We use the maximum of the absolute differences of the x and y coordinates of two cells as an approximation of their distance. Cells expanded by the methods are shaded gray in Figure 8. As the figure shows, the heuristic search outperforms the uninformed searches, and the incremental search outperforms the complete (that is, nonincremental) ones after the first move of the robot (where previous search results are available). The figures also illustrate that the combination of heuristic and incremental search performed by both versions of D* Lite decreases the number of expanded cells even more than either a heuristic search or an incremental search individually. In particular, the initial search of both versions of D* Lite expands exactly the same cells as an A* search if A* breaks ties between vertices with the same f-values suitably. In our example, we broke ties in the most advantageous way for A* and thus both versions of D* Lite and A* expand not exactly the same cells. The second search of both

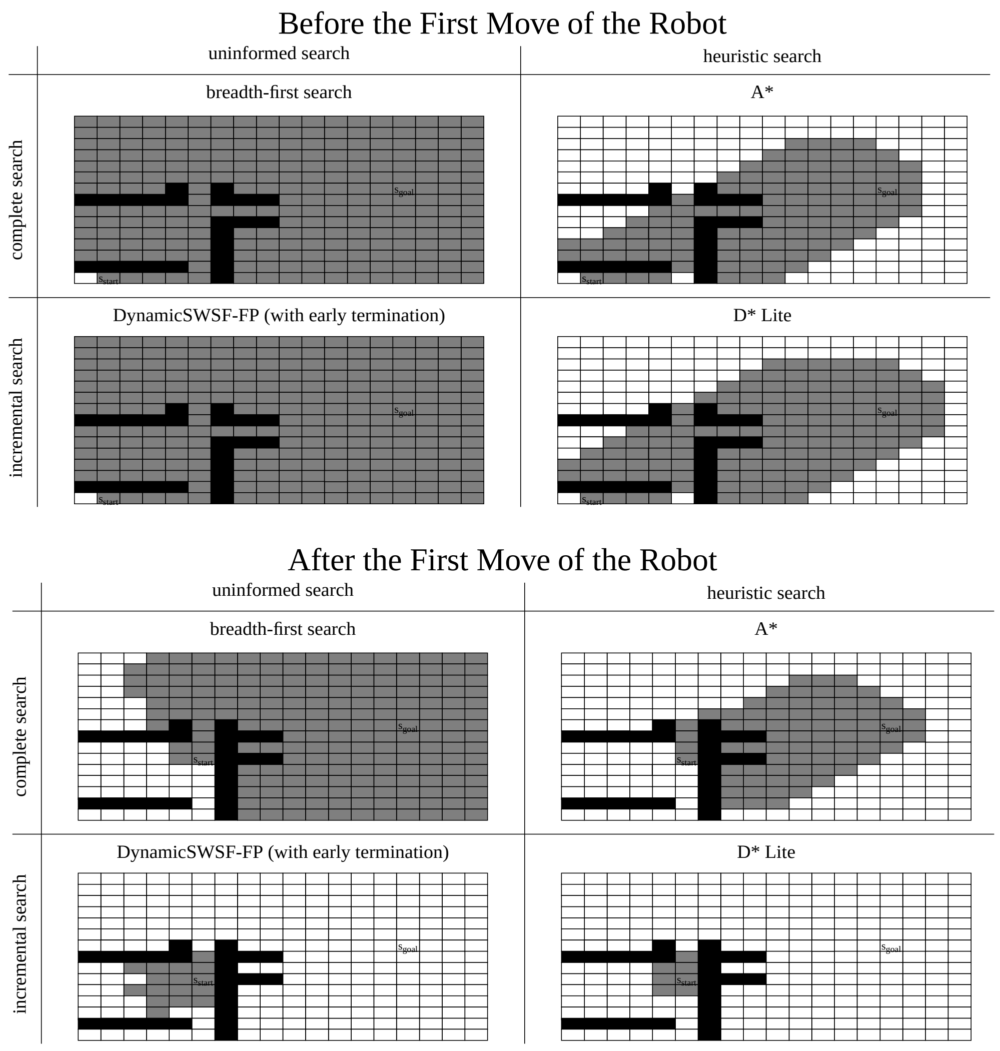


Figure 8: Simple Example (Part 2).

versions of D* Lite expands only a subset of those cells whose goal distances changed (or had not been calculated before). Thus, both versions of D* Lite result in substantial savings over an A* search that replicates most of its previous search.

We now step through the example from Figure 1 to show the operation of both versions of D* Lite. Figure 9 (top) shows the untraversable cells that the robot knows about initially. It also shows the heuristics of the traversable cells, that is, the approximation of the distance from the start cell to the traversable cell given by the maximum of the absolute differences of the x and y coordinates of both cells. Figure 9 (bottom) shows the g-values and rhs-values of the traversable cells and, for locally inconsistent cells, also their priorities. At every point in time exactly the locally inconsistent cells are in the priority queue according to Invariant 2. The locally inconsistent cell with the smallest key has a bold border to indicate that it will be expanded next. The

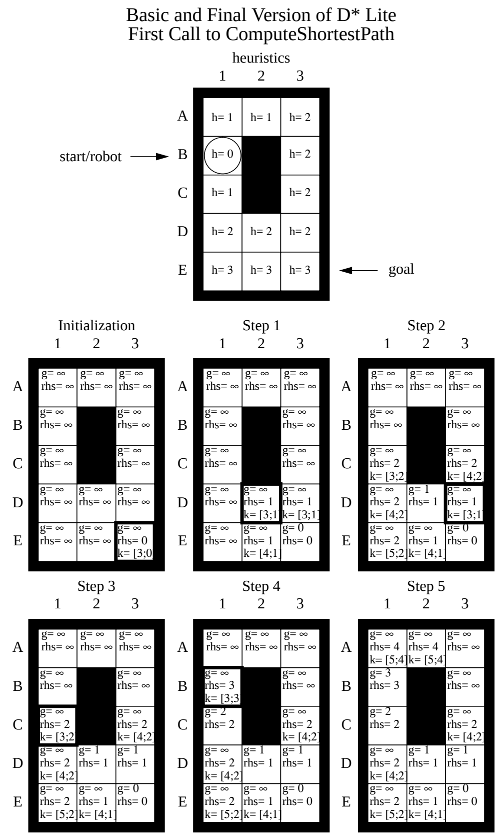
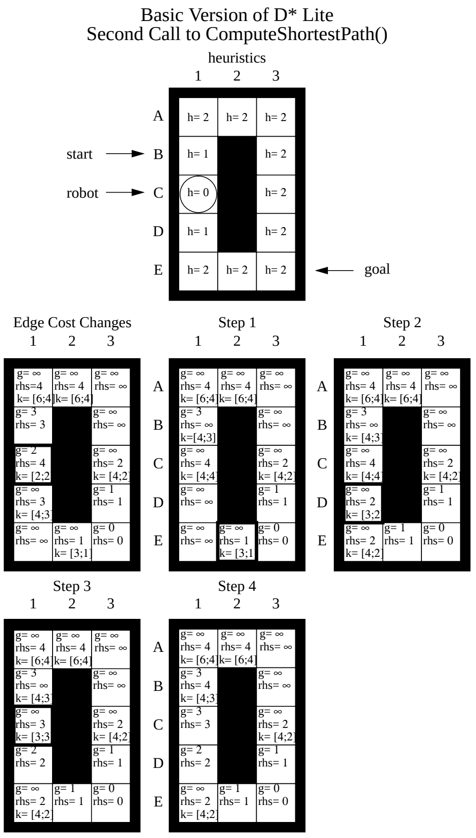


Figure 9: Illustration of the Operation of the Basic and Final Versions of D* Lite.

grid labelled “Initialization” shows the values directly before ComputeShortestPath() is called for the first time. The next grids show the values after each iteration of the first call to ComputeShortestPath(). If the g-value of an expanded cell is larger than its rhs-value, ComputeShortestPath() sets the g-value of the cell to its rhs-value. Otherwise, ComputeShortestPath() sets the g-value to infinity. To maintain Invariants 1-3, ComputeShortestPath() then recalculates the rhs-values of the cells potentially affected by this assignment, checks whether the cells become locally consistent or inconsistent, and (if necessary) removes them from or adds them to the priority queue. It then repeats this process until it is sure that it has found a shortest path, which requires it to recalculate some goal distances but not all of them. The last grid shows the values after ComputeShortestPath() returns. Note that an A* search can expand exactly the same cells in exactly the same order. One can then follow a shortest path from the current cell of the robot to the goal cell by starting at the current cell and always greedily decreasing the goal distance. Any way of doing this results in a shortest path from the current cell to the goal cell. Since all costs are one, this means that the shortest path from the current cell B1 to goal cell E3 is via cells C1 and D2. The robot then moves to cell C1 and discovers that cell D2 is untraversable. Figure 10 shows how the basic version of D* Lite continues and Figure 11 shows how the final version of D* Lite continues. Figure 10 (top) shows the grid as it is now perceived by the robot.



Basic Version of D* Lite
Second Call to ComputeShortestPath()
heuristics
1 2 3
A B C D E
start
robot
goal
h= 2 h= 2 h= 2
h= 1 h= 2
h= 0 h= 2
h= 1 h= 2
h= 2 h= 2 h= 2
Edge Cost Changes
g= ∞ rhs=4 k= [6;4] | g= ∞ rhs= 4 k= [6;4] | g= ∞ rhs= ∞
g= 3 rhs= 3 | g= ∞ rhs= ∞
g= 2 rhs= 4 k= [2;2] | g= ∞ rhs= 2 k= [4;2]
g= ∞ rhs= 3 k= [4;3] | g= 1 rhs= 1
g= ∞ rhs= ∞ | g= ∞ rhs= 1 k= [3;1] | g= 0 rhs= 0
Step 1
g= ∞ rhs= 4 k= [6;4] | g= ∞ rhs= 4 k= [6;4] | g= ∞ rhs= ∞
g= 3 rhs= ∞ k=[4;3] | g= ∞ rhs= ∞
g= ∞ rhs= 4 k= [4;4] | g= ∞ rhs= 2 k= [4;2]
g= ∞ rhs= ∞ | g= 1 rhs= 1
g= ∞ rhs= ∞ | g= ∞ rhs= 1 k= [3;1] | g= 0 rhs= 0
Step 2
g= ∞ rhs= 4 k= [6;4] | g= ∞ rhs= 4 k= [6;4] | g= ∞ rhs= ∞
g= 3 rhs= ∞ k= [4;3] | g= ∞ rhs= ∞
g= ∞ rhs= 4 k= [4;4] | g= ∞ rhs= 2 k= [4;2]
g= ∞ rhs= 2 k= [3;2] | g= 1 rhs= 1
g= ∞ rhs= 2 k= [4;2] | g= 1 rhs= 1 | g= 0 rhs= 0
Step 3
g= ∞ rhs= 4 k= [6;4] | g= ∞ rhs= 4 k= [6;4] | g= ∞ rhs= ∞
g= 3 rhs= ∞ k= [4;3] | g= ∞ rhs= ∞
g= ∞ rhs= 3 k= [3;3] | g= ∞ rhs= 2 k= [4;2]
g= 2 rhs= 2 | g= 1 rhs= 1
g= ∞ rhs= 2 k= [4;2] | g= 1 rhs= 1 | g= 0 rhs= 0
Step 4
g= ∞ rhs= 4 k= [6;4] | g= ∞ rhs= 4 k= [6;4] | g= ∞ rhs= ∞
g= 3 rhs= 4 k= [4;3] | g= ∞ rhs= ∞
g= 3 rhs= 3 | g= ∞ rhs= 2 k= [4;2]
g= 2 rhs= 2 | g= 1 rhs= 1
g= ∞ rhs= 2 k= [4;2] | g= 1 rhs= 1 | g= 0 rhs= 0

Figure 10: Illustration of the Operation of the Basic Version of D* Lite, Continuing Figure 9.

The figure also shows the new heuristics of the traversable cells. To maintain Invariants 1-3, the basic version of D* Lite first updates the rhs-values and keys of the cells adjacent to D2 as well as their membership in the priority queue if they become locally consistent or inconsistent. It also updates the priorities of all cells in the priority queue to reflect the new heuristics. The grid labelled "Edge Cost Changes" shows the values directly before ComputeShortestPath() is called for the second time. The next grids show the values after each iteration of the second call to ComputeShortestPath(). Finally, the last grid shows the values after ComputeShortestPath() returns. The shortest path from the current cell C1 to goal cell E3 is via cells D1 and E2. The robot then follows this path from its current cell to the goal cell without observing additional untraversable cells and thus without further calls to ComputeShortestPath().

5.1 Optimizations

We implemented both the basic and final version of D* Lite, using standard binary heaps as priority queues. There are several ways of optimizing both versions of D* Lite without changing their overall operation, which we discuss in the following in the context of the final version of

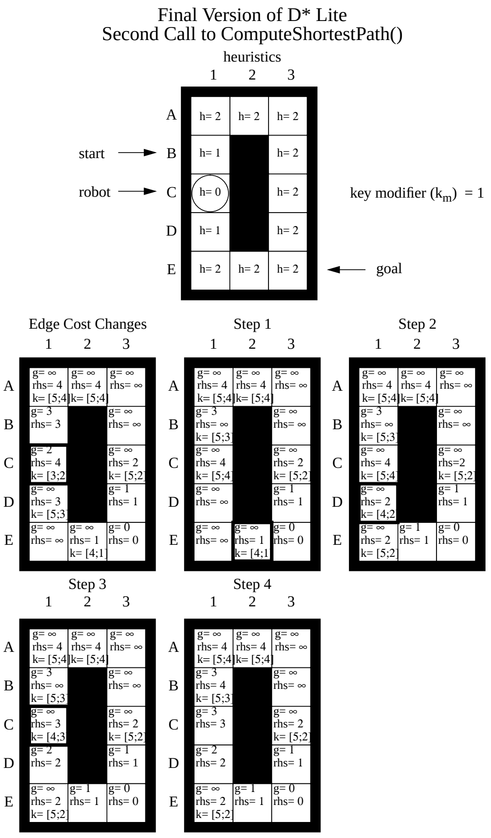


Figure 11: Illustration of the Operation of the Final Version of D* Lite, Continuing Figure 9.

D* Lite, the optimized version of which is shown in Figure 12.

First, the termination condition of ComputeShortestPath() can be changed to make ComputeShortestPath() more efficient. As stated, ComputeShortestPath() terminates when the start vertex is locally consistent and its key is less than or equal to U.TopKey() {10”}. However, ComputeShortestPath() can also terminate when the start vertex is locally overconsistent and its key is less than or equal to U.TopKey(). To understand why this is so, assume that the start vertex is indeed locally overconsistent and its key is less than or equal to U.TopKey(). Then, its key must be equal to U.TopKey() since U.TopKey() is the smallest key of any locally inconsistent vertex. Thus, ComputeShortestPath() could expand the start vertex next, in which case it would set its g-value to its rhs-value. The start vertex then becomes locally consistent according to Theorem 13 in the appendix, its key is less than or equal to U.TopKey(), and ComputeShortestPath() thus terminates. At this point in time, the g-value of the start vertex equals its goal distance. Thus, ComputeShortestPath() can already terminate when the start vertex is not locally underconsistent and its key is less than or equal to U.TopKey() {10”’}. In this case, the start vertex can remain locally inconsistent after ComputeShortestPath() terminates and its g-value thus may not be equal to its goal distance (its rhs-value continues to be equal to its goal distance). This is not a problem since the g-value is not used to determine how the robot should

The pseudocode uses the following functions to manage the priority queue: U.Top$()$ returns a vertex with the smallest priority of all vertices in priority queue U. U.TopKey$()$ returns the smallest priority of all vertices in priority queue U. (If U is empty, then U.TopKey$()$ returns $[\infty; \infty]$.) U.Insert(s, k) inserts vertex s into priority queue U with priority k. U.Update(s, k) changes the priority of vertex s in priority queue U to k. (It does nothing if the current priority of vertex s already equals k.) Finally, U.Remove(s) removes vertex s from priority queue U.

```
procedure CalculateKey(s)
{01'''} return [$\min(g(s), rhs(s)) + h(s_{start}, s) + k_m; \min(g(s), rhs(s))$];
procedure Initialize()
{02'''} $U = \emptyset$;
{03'''} $k_m = 0$;
{04'''} for all $s \in S$ $rhs(s) = g(s) = \infty$;
{05'''} $rhs(s_{goal}) = 0$;
{06'''} U.Insert($s_{goal}$, $[h(s_{start}, s_{goal}); 0]$);
procedure UpdateVertex(u)
{07'''} if ($g(u) \neq rhs(u)$ AND $u \in U$) U.Update($u, CalculateKey(u)$);
{08'''} else if ($g(u) \neq rhs(u)$ AND $u \notin U$) U.Insert($u, CalculateKey(u)$);
{09'''} else if ($g(u) = rhs(u)$ AND $u \in U$) U.Remove($u$);
procedure ComputeShortestPath()
{10'''} while (U.TopKey() $\dot{<}$ CalculateKey($s_{start}$) OR $rhs(s_{start}) > g(s_{start})$)
{11'''}   $u$ = U.Top();
{12'''}   $k_{old}$ = U.TopKey();
{13'''}   $k_{new}$ = CalculateKey($u$));
{14'''}   if($k_{old} \dot{<} k_{new}$)
{15'''}     U.Update($u, k_{new}$);
{16'''}   else if ($g(u) > rhs(u)$)
{17'''}     $g(u) = rhs(u)$;
{18'''}     U.Remove($u$);
{19'''}     for all $s \in$ Pred($u$)
{20'''}       if ($s \neq s_{goal}$) $rhs(s) = \min(rhs(s), c(s, u) + g(u))$;
{21'''}       UpdateVertex($s$);
{22'''}   else
{23'''}     $g_{old} = g(u)$;
{24'''}     $g(u) = \infty$;
{25'''}     for all $s \in$ Pred($u$) $\cup \{u\}$
{26'''}       if ($rhs(s) = c(s, u) + g_{old}$)
{27'''}         if ($s \neq s_{goal}$) $rhs(s) = \min_{s' \in \text{Succ}(s)}(c(s, s') + g(s'))$;
{28'''}       UpdateVertex($s$);
procedure Main()
{29'''} $s_{last} = s_{start}$;
{30'''} Initialize();
{31'''} ComputeShortestPath();
{32'''} while ($s_{start} \neq s_{goal}$)
{33'''}   /* if ($rhs(s_{start}) = \infty$) then there is no known path */
{34'''}   $s_{start} = \arg\min_{s' \in \text{Succ}(s_{start})}(c(s_{start}, s') + g(s'))$;
{35'''}   Move to $s_{start}$;
{36'''}   Scan graph for changed edge costs;
{37'''}   if any edge costs changed
{38'''}     $k_m = k_m + h(s_{last}, s_{start})$;
{39'''}     $s_{last} = s_{start}$;
{40'''}     for all directed edges $(u, v)$ with changed edge costs
{41'''}       $c_{old} = c(u, v)$;
{42'''}       Update the edge cost $c(u, v)$;
{43'''}       if ($c_{old} > c(u, v)$)
{44'''}         if ($u \neq s_{goal}$) $rhs(u) = \min(rhs(u), c(u, v) + g(v))$;
{45'''}       else if ($rhs(u) = c_{old} + g(v)$)
{46'''}         if ($u \neq s_{goal}$) $rhs(u) = \min_{s' \in \text{Succ}(u)}(c(u, s') + g(s'))$;
{47'''}       UpdateVertex($u$);
{48'''}     ComputeShortestPath();
```

Figure 12: D* Lite: Final Version (optimized version).

move. The only difference is that now the rhs-value of the start vertex (instead of its g-value) being infinity indicates that there is no path from the start vertex to the goal vertex.

Second, a vertex sometimes gets removed from the priority queue on line $\{$08”$\}$ and then immediately reinserted on line $\{$09”$\}$. In this case, it is often more efficient to leave the vertex in the priority queue and only update its priority ($\{$07”’$\}$).

Third, when UpdateVertex$()$ on line $\{$17”$\}$ computes the rhs-value for a predecessor of an overconsistent vertex (that is, a vertex whose g-value is larger than its rhs-value) it is unnecessary to take the minimum over all of its respective successors since only the g-value of the overconsistent vertex has changed. Since it decreased, it cannot increase the rhs-values of the predecessors. Thus, it is sufficient to compute the rhs-value as the minimum of its old rhs-value and the sum of the cost of moving from the predecessor to the overconsistent vertex and the new g-value of the overconsistent vertex ($\{$20”’$\}$). A similar optimization can be made for the computation of the rhs-value of a vertex after the cost of one of its outgoing edges has changed ($\{$44”’$\}$).

Fourth, when UpdateVertex$()$ on line $\{$20”$\}$ computes the rhs-value for a predecessor of an underconsistent vertex (that is, a vertex whose g-value is smaller than its rhs-value), the only g-value that has changed is the g-value of the underconsistent vertex. Since it increased, the rhs-value of the predecessor can only get affected if its old rhs-value was based on the old g-value of the underconsistent vertex. This can be used to decide whether the predecessor needs to get updated and its rhs-value needs to get recomputed ($\{$26”’-27”’$\}$). A similar optimization can be made for the computation of the rhs-value of a vertex after the cost of one of its outgoing edges has changed ($\{$45”’-46”’$\}$).

Fifth, there are several small optimizations one can perform. For example, the priority on line $\{$06”$\}$ can be calculated directly ($\{$06”’$\}$), CalculateKey$()$ on lines $\{$13”-14”$\}$ needs to calculate the priority of vertex u only once ($\{$13”’$\}$), and the vertex with the highest priority needs to get removed on line $\{$12”$\}$ only if line $\{$14”$\}$ does not reinsert it again immediately afterwards ($\{$12”’, 15”’, 18”’$\}$).

6 Experimental Results

We now compare (focussed) D* and various versions of the optimized final version of D* Lite. We implemented all methods using standard binary heaps as priority queues (although using more complex data structures, such as Fibonacci heaps, as priority queues could possibly make U.Update() more efficient). Since they move the robot in the same way and D* has already been demonstrated with great success on real robots, we only need to perform a simulation study. We need to compare the total planning time of the methods. Since the actual planning times are implementation and machine dependent, they make it difficult for others to reproduce the results of our performance comparison. We therefore use three measures that all correspond to common operations performed by the methods and thus heavily influence their planning

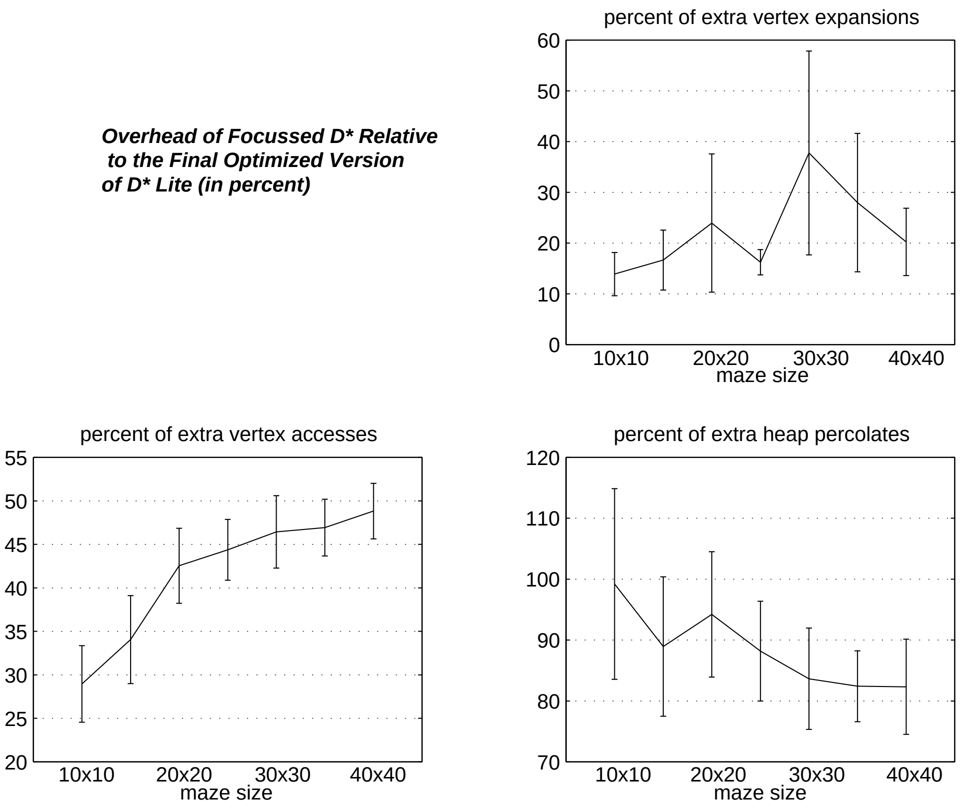


Figure 13: Goal-Directed Navigation in Unknown Terrain (1).

times, yet are implementation and machine independent: the total number of vertex expansions, the total number of heap percolates (exchanges of a parent and child in the heap), and the total number of vertex accesses (for example, to read or change their values).

We run the experiments on the MissionLab robot simulation system [MAC97]. The terrain resembles office environments. The robot always observes which of its eight adjacent cells is traversable and can then move to one of them. We use again the maximum of the absolute differences of the x and y coordinates of any two cells as approximations of their distance.

All figures graph the three performance measures of the other methods as percent difference relative to D* Lite. Thus, D* Lite always scores zero. The figures also show the corresponding 95 percent confidence intervals to demonstrate that our conclusions are statistically significant.

6.1 Goal-Directed Navigation in Unknown Terrain

We first perform experiments for goal-directed navigation in unknown terrain.

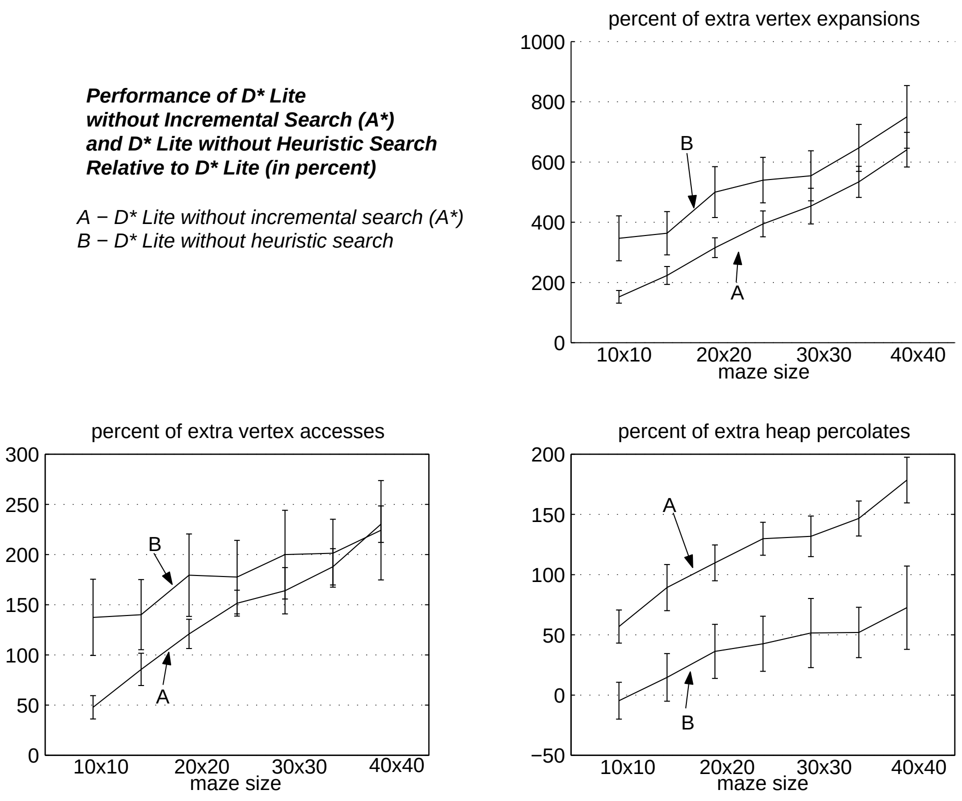


Figure 14: Goal-Directed Navigation in Unknown Terrain (2).

Figure 13 compares D* Lite and D* for goal-directed navigation in terrain of seven different sizes, averaged over 50 randomly generated terrains of each size whose obstacle density varies from 10 to 40 percent. D* Lite performs better than D* with respect to all three measures, justifying our claim that it is more efficient than D*.

We also studied to which degree the combination of incremental and heuristic search that D* Lite implements outperforms incremental or heuristic searches individually. Figure 14 compares D* Lite, D* Lite without heuristic search, and D* Lite without incremental search (that is, A*) for goal-directed navigation, using the same setup. We decided not to include D* Lite without both heuristic and incremental search in the comparision because it performs so poorly that graphing its performance becomes a problem. D* Lite outperforms the other two search methods according to all three performance measures, even up to a factor of seven for the vertex expansions. Moreover, its advantage seems to increase as the terrain gets larger. Only for the number of heap percolates for terrain of size 10 by 10 and 15 by 15 is the difference between D* Lite and D* Lite without heuristic search statistically not significant. These results also confirm earlier experimental results that D* can outperform A* for goal-directed navigation in unknown terrain by one order of magnitude or more [Ste95].

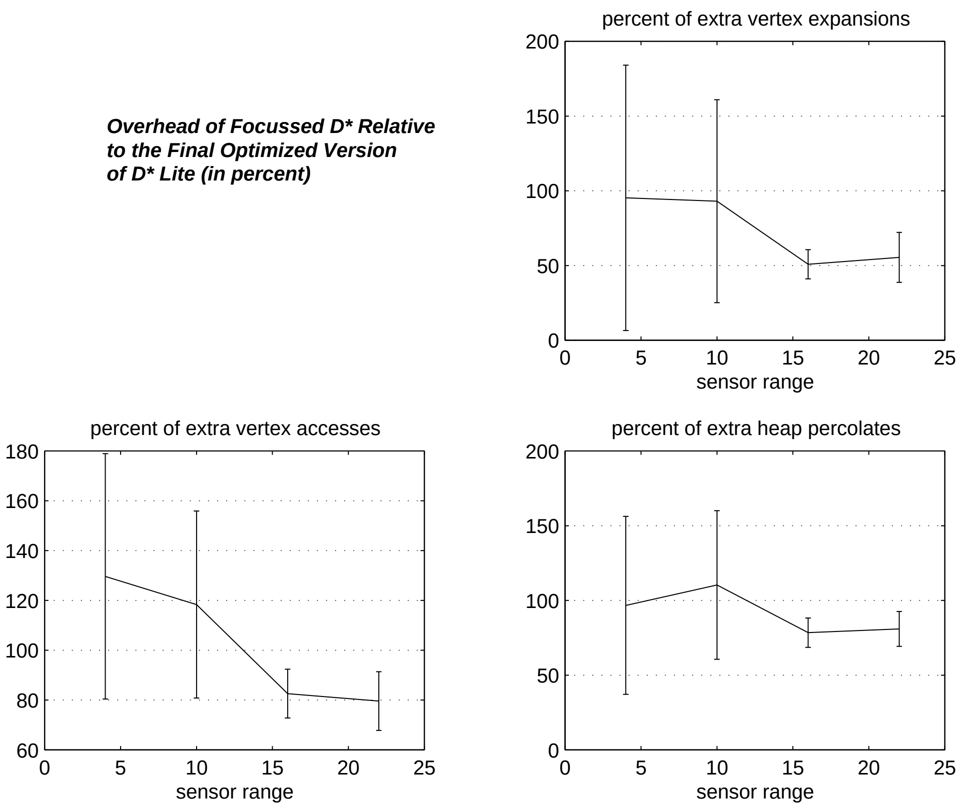


Figure 15: Mapping of Unknown Terrain (1).

6.2 Mapping of Unknown Terrain

D* Lite is versatile because it applies to other robot navigation tasks as well. In particular, it can also be used to implement greedy mapping [KTH01], a simple but powerful mapping strategy that has repeatedly been used on mobile robots by different research groups [TBB$^+$98, KTH01, RMS01]. To improve our understanding of D* Lite, we therefore also perform experiments for mapping for unknown terrain.

Greedy mapping discretizes terrain into cells and then always moves the robot from its current cell to the closest cell with unknown traversability, until the terrain is mapped. In this case, the graph is an eight-connected grid. The costs of its edges are initially one. They change to infinity when the robot discovers that they cannot be traversed. There is one additional vertex that is connected to all grid vertices. The costs of these edges are initially one. They change to infinity once the corresponding grid vertex has been visited. One can implement greedy mapping by applying D* Lite to this graph with s_{start} being the current vertex of the robot and s_{goal} being the additional vertex.

Figure 15 compares D* Lite and D* for mapping of unknown terrain with different sensor ranges, averaging over 50 randomly generated grids of size 64 by 25. We varied the sensor range

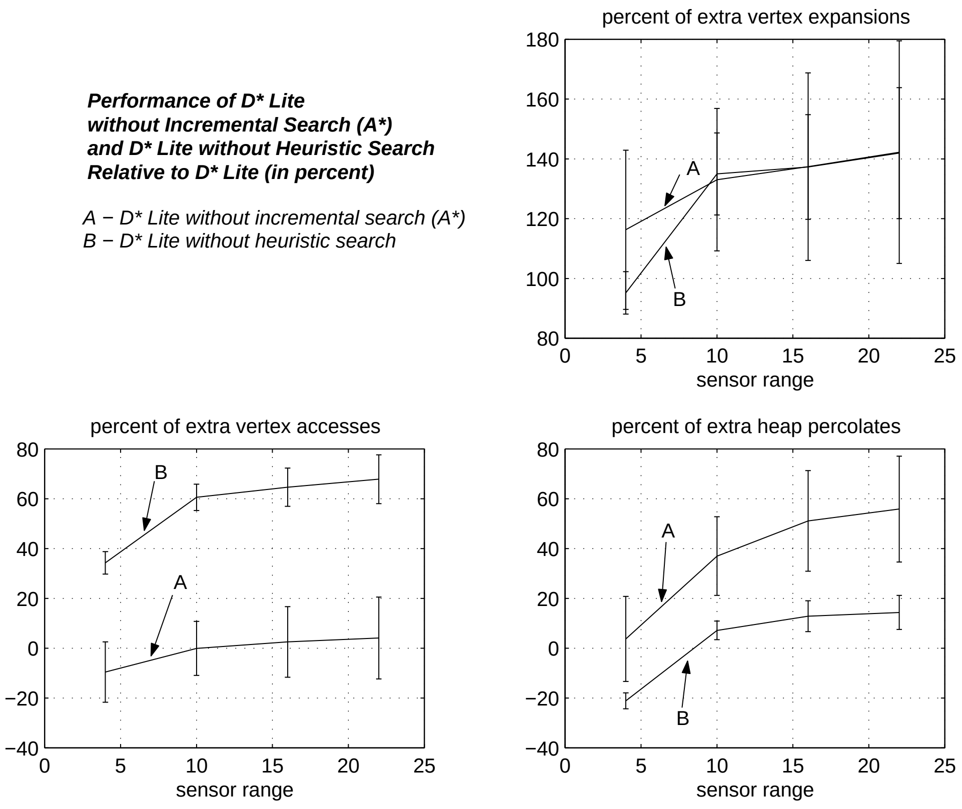


Figure 16: Mapping of Unknown Terrain (2).

of the robot to simulate both short-range and long-range sensors. For example, if the sensor range is four, then the robot can sense all untraversable cells that are up to four cells in any direction away from the robot as long as they are not blocked from view by other untraversable cells. Again, D* Lite performs better than D* with respect to all three measures, demonstrating its advantage across two different tasks.

Figure 16 compares D* Lite, D* Lite without heuristic search, and D* Lite without incremental search (that is, A*) for mapping of unknown terrain, using the same setup. The number of vertex expansions of D* Lite is always far less than that of the other two methods. This also holds for the number of heap percolates and vertex accesses, with the exception of sensor range four for the heap percolates. The advantage of D* Lite over the other two search methods seems to increase as the sensor range increases. This is important since laser scanners tend to be the sensors of choice for mobile robots. Only for the number of vertex accesses is the difference between D* Lite and D* Lite without incremental search statistically not significant.

Overall, our experiments show that D*-like replanning methods apply unchanged to two different robot navigation tasks and performs well for both of them.

7 Related Work

Path planning for goal-directed robot navigation in known terrain has been studied extensively [Lat91]. Path planning for robot navigation in unknown terrain has been studied less frequently. We are interested in those methods that repeatedly determine a shortest path from the current robot coordinates to the goal coordinates while the robot moves along the path. Most methods do not fit this description, including the bug algorithms [LS87]. Those methods that do fit this description face the problem that they have to find shortest paths repeatedly, and researchers have studied how results from previous searches can be used to speed up the current search. Some of the approaches to this problem are based on minimum cost flow problems solved by the network simplex method [EH01]. Other approaches are based on graph search problems solved with either massively parallel search methods [TEFA97] or incremental search methods.

Incremental search methods typically solve dynamic shortest path problems, that is, path problems where shortest paths between a given start and goal vertex have to be determined repeatedly as the topology of a graph or its edge costs change [RR96b]. Examples include [AIMSN91, ES81, EG85, FMS93, FFG01, FMSN96, GSV78, Ita88, KS93, LC90, Roh85, SP75]. They often 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 [FMSN98]. 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 [FMSN00]. An example of an incremental search method for fully dynamic shortest path problems is DynamicSWSF-FP [RR96a], a variant of which has been applied to hierarchical motion planning [BH95]. Incremental search methods are typically uninformed, including all incremental search methods cited so far.

There are only very few incremental heuristic search methods that have been applied to robot navigation in unknown terrain, that is, to take into account that the start vertex changes because the robot moves in the terrain. Some of these methods first identify the perimeter of areas in which the previous movement decisions need to get updated and restart the search from there [Tro90, PNIV01]. More complex and typically more efficient methods discover these areas while updating previous movement decisions. To the best of our knowlege, the only methods that fit this description are focussed D* [Ste94] and the basic and final versions of D* Lite.

All search methods for robot navigation in unknown terrain can be made more efficient by using hierarchical cell decompositions [BH95, YSBS98]. It is future work to study the basic and final versions of D* Lite in this context.

8 Conclusions

In this article, we have presented D* Lite, a novel fast replanning method for goal-directed navigation in unknown terrain that implements the same navigation strategy as (focussed) D*. Both methods search from the goal vertex towards the current vertex of the robot, use heuristics to focus the search, and use similar ways to minimize having to reorder the priority queue. However, D* Lite builds on our LPA*, that has a solid theoretical foundation, a strong similarity to A*, is efficient (since it does not expand any vertices whose g-values were already equal to their respective goal distances) and has been extended in a number of ways. Thus, D* Lite is algorithmically different from D*. It is short, simple, and consequently easy to understand and extend, yet is more efficient than D*. We believe that our results will make D*-like replanning methods even more popular and enable robotics researchers to adapt them to additional applications. More generally, we believe that our experimental and analytical results provide a strong algorithmic foundation for further research on fast replanning methods for mobile robots.

Acknowledgments

We thank Anthony Stentz for his support of this work. The second author worked out the proofs. The Intelligent Decision-Making Group is partly supported by NSF awards under contracts IIS-9984827, IIS-0098807, and ITR/AP-0113881 as well as an IBM faculty partnership award. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations and agencies or the U.S. government.

References

[AIMSN91] G. Ausiello, G. Italiano, A. Marchetti-Spaccamela, and U. Nanni. Incremental algorithms for minimal length paths. *Journal of Algorithms*, 12(4):615–638, 1991.

[Bel57] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[BH95] M. Barbehenn and S. Hutchinson. Efficient search and hierarchical motion planning by dynamically maintaining single-source shortest paths trees. *IEEE Transactions on Robotics and Automation*, 11(2):198–214, 1995.

[BS98] B. Brumitt and A. Stentz. GRAMMPS: a generalized mission planner for multiple mobile robots. In *Proceedings of the International Conference on Robotics and Automation*, 1998.

[CB94] H. Choset and J. Burdick. Sensor based planning and nonsmooth analysis. In *Proceedings of the International Conference on Robotics and Automation*, pages 3034–3041, 1994.

[CB95] H. Choset and J. Burdick. Sensor-based planning, part II: Incremental construction of the generalized Voronoi graph. In *Proceedings of the International Conference on Robotics and Automation*, pages 1643–1649, 1995.

[EG85] S. Even and H. Gazit. Updating distances in dynamic grapphs. *Methods of Operations Research*, 49:371–387, 1985.

[EH01] T. Ersson and X. Hu. Path planning and navigation of mobile robots in unknown environments. In *Proceedings of the International Conference on Intelligent Robots and Systems*, 2001.

[ES81] S. Even and Y. Shiloach. An on-line edge deletion problem. *Journal of the ACM*, 28(1):1–4, 1981.

[FFG01] P. Franciosa, D. Frigioni, and R. Giaccio. Semi-dynamic breadth-first search in digraphs. *Theoretical Computer Science*, 250(1–2):201–217, 2001.

[FMS93] E. Feuerstein and A. Marchetti-Spaccamela. Dynamic algorithms for shortest paths in planar graphs. *Theoretical Computer Science*, 116(2):359–371, 1993.

[FMSN96] 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*, pages 212–221, 1996.

[FMSN98] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Semidynamic algorithms for maintaining single source shortest path trees. *Algorithmica*, 22(3):250–274, 1998.

[FMSN00] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *Journal of Algorithms*, 34(2):251–281, 2000.

[GSV78] S. Goto and A. Sangiovanni-Vincentelli. A new shortest path updating algorithm. *Networks*, 8(4):341–372, 1978.

[HMC99] M. Hebert, R. McLachlan, and P. Chang. Experiments with driving modes for urban robots. In *Proceedings of the SPIE Mobile Robots*, 1999.

[Ita88] G. Italiano. Finding paths and deleting edges in directed acyclic graphs. *Information Processing Letters*, 28(1):5–11, 1988.

[KL01a] S. Koenig and M. Likhachev. Incremental A*. In *Advances in Neural Information Processing Systems 14*, 2001.

[KL01b] S. Koenig and M. Likhachev. Lifelong planning A*. Technical Report GIT-COGSCI-2002/2, College of Computing, Georgia Institute of Technology, Atlanta (Georgia), 2001. (Available online at: http://www.cc.gatech.edu/fac/Sven.Koenig/greedyonline/).

[KL02] S. Koenig and M. Likhachev. Improved fast replanning for robot navigation in unknown terrain. In *Proceedings of the International Conference on Robotics and Automation*, 2002.

[KS93] 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.

[KTH01] S. Koenig, C. Tovey, and W. Halliburton. Greedy mapping of terrain. In *Proceedings of the International Conference on Robotics and Automation*, pages 3594–3599, 2001.

[Lat91] J.-C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, 1991.

[LC90] C. Lin and R. Chang. On the dynamic shortest path problem. *Journal of Information Processing*, 13(4):470–476, 1990.

[LS87] V. Lumelsky and A. Stepanov. Path planning strategies for point automaton moving amidst unknown obstacles of arbitrary shape. *Algorithmica*, 2:403–430, 1987.

[MAC97] D. Mackenzie, R. Arkin, and J. Cameron. Multiagent mission specification and execution. *Autonomous Robots*, 4(1):29–57, 1997.

[MXH$^+$00] L. Matthies, Y. Xiong, R. Hogg, D. Zhu, A. Rankin, B. Kennedy, M. Hebert, R. Maclachlan, C. Won, T. Frost, G. Sukhatme, M. McHenry, and S. Goldberg. A portable, autonomous, urban reconnaissance robot. In *Proceedings of the International Conference on Intelligent Autonomous Systems*, 2000.

[NK95] B. Nebel and J. Koehler. Plan reuse versus plan generation: A theoretical and empirical analysis. *Artificial Intelligence*, 76(1–2):427–454, 1995.

[Pea85] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1985.

[PNIV01] 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.

[RMS01] L. Romero, E. Morales, and E. Sucar. An exploration and navigation approach for indoor mobile robots considering sensor's perceptual limitations. In *Proceedings of the International Conference on Robotics and Automation*, pages 3092–3097, 2001.

[Roh85] 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.

[RR96a] G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms*, 21:267–305, 1996.

[RR96b] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *Theoretical Computer Science*, 158(1–2):233–277, 1996.

[SH95] A. Stentz and M. Hebert. A complete navigation system for goal acquisition in unknown environments. *Autonomous Robots*, 2(2):127–145, 1995.

[SP75] P. Spira and A. Pan. On finding and updating spanning trees and shortest paths. *SIAM Journal on Computing*, 4:375–380, 1975.

[Ste94] A. Stentz. Optimal and efficient path planning for partially-known environments. In *Proceedings of the International Conference on Robotics and Automation*, pages 3310–3317, 1994.

[Ste95] A. Stentz. The focussed D* algorithm for real-time replanning. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1652–1659, 1995.

[TBB+98] S. Thrun, A. Bücken, W. Burgard, D. Fox, T. Fröhlinghaus, D. Hennig, T. Hofmann, M. Krell, and T. Schmidt. Map learning and high-speed navigation in RHINO. In D. Kortenkamp, R. Bonasso, and R. Murphy, editors, *Artificial Intelligence Based Mobile Robotics: Case Studies of Successful Robot Systems*, pages 21–52. MIT Press, 1998.

[TDD+00] S. Thayer, B. Digney, M. Diaz, A. Stentz, B. Nabbe, and M. Hebert. Distributed robotic mapping of extreme environments. In *Proceedings of the SPIE: Mobile Robots XV and Telemanipulator and Telepresence Technologies VII*, volume 4195, 2000.

[TEFA97] M. Tao, A. Elssamadisy, N. Flann, and B. Abbott. Optimal route re-planning for mobile robots: A massively parallel incremental A* algorithm. In *International Conference on Robotics and Automation*, pages 2727–2732, 1997.

[Thr98] S. Thrun. Lifelong learning algorithms. In S. Thrun and L. Pratt, editors, *Learning To Learn*. Kluwer Academic Publishers, 1998.

[Tro90] 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.

[YSBS98] A. Yahja, A. Stentz, B. Brumitt, and S. Singh. Framed-quadtree path planning for mobile robots operating in sparse environments. In *International Conference on Robotics and Automation*, 1998.

A The Proofs – Heuristics

Let $c^*(s,s')$ denote the cost of a shortest path from vertex $s \in S$ to vertex $s' \in S$. We then call heuristics admissible iff they do not overestimate the distance between any two vertices, that is, iff $h(s,s') \leq c^*(s,s')$ for all $s, s' \in S$. (Note that heuristics are often called admissible iff they do not overestimate the goal distance of any vertex. We, on the other hand, call heuristics admissible iff they do not overestimate the goal distance of any vertex, no matter what the goal vertex is.) We call heuristics (forward) consistent iff they satisfy $h(s_{goal}, s_{goal}) = 0$ and $h(s, s_{goal}) \leq c(s,s') + h(s', s_{goal})$ for all vertices $s \in S$ and $s' \in \text{Succ}(s)$. Forward consistent heuristics are important for a (forward) A* search from the start vertex to the goal vertex in a graph. We call heuristics backward consistent iff they satisfy $h(s_{start}, s_{start}) = 0$ and $h(s_{start}, s) \leq h(s_{start}, s') + c(s', s)$ for all vertices $s \in S$ and $s' \in \text{Pred}(s)$. Backward consistent heuristics are important for a (backward) A* search from the goal vertex to the start vertex in the reversed graph, that is derived from the original graph by reversing all edges and exchanging the start and goal vertex. Finally, we call heuristics forward-backward consistent iff they satisfy $h(s,s'') \leq h(s,s') + h(s',s'')$ for all vertices $s, s', s'' \in S$.

Theorem 3 *Heuristics that are obtained by relaxing a graph are admissible and forward-backward consistent.*

Proof: First, the shortest path between any two vertices in the original graph also exists in the relaxed graph. Thus, the distance between any two vertices in the relaxed graph cannot be larger than the distance between the same two vertices in the original graph and the heuristics are admissible. Second, the distance between vertex $s \in S$ and vertex $s' \in S$ in the relaxed graph corresponds to $h(s,s')$ in the original graph. The distances in the relaxed graph satisfy the triangle inequality and the heuristics are thus forward-backward consistent. ■

Theorem 4 *Admissible and forward-backward consistent heuristics are also forward consistent and backward consistent, no matter what the start and goal vertices are.*

Proof: Admissible heuristics satisfy $h(s,s') \leq c^*(s,s')$ and forward-backward consistent heuristics satisfy $h(s,s'') \leq h(s,s') + h(s',s'')$ for all vertices $s, s', s'' \in S$. Consequently, $h(s,s'') \leq h(s,s') + h(s',s'') \leq c^*(s,s') + h(s',s'') = c(s,s') + h(s',s'')$ for all vertices $s \in S$, $s' \in \text{Succ}(s)$, and $s'' = s_{goal}$, no matter what the goal vertex is. Also, $h(s_{goal}, s_{goal}) = 0$ since the heuristics are admissible. Thus, the heuristics are forward consistent, no matter

The g-values are initialized by the user before Main() is called.

The pseudocode uses the following functions to manage the priority queue: U.TopKey() returns the smallest priority of all vertices in priority queue U. (If U is empty, then U.TopKey() returns $[\infty; \infty]$.) U.Pop() deletes the vertex with the smallest priority in priority queue U and returns the vertex. U.Insert(s, k) inserts vertex s into priority queue U with priority k. Finally, U.Remove(s) removes vertex s from priority queue U.

```
procedure CalculateKey(s)
{01} return [min(g(s), rhs(s)) + h(s_start, s); min(g(s), rhs(s))];
procedure Initialize()
{02} U = ∅;
{03} rhs(s_goal) = 0;
{04} for all s ∈ S UpdateVertex(s);
procedure UpdateVertex(u)
{05} if (u ≠ s_goal) rhs(u) = min_{s' ∈ Succ(u)}(c(u, s') + g(s'));
{06} if (u ∈ U) U.Remove(u);
{07} if (g(u) ≠ rhs(u)) U.Insert(u, CalculateKey(u));
procedure ComputeShortestPath()
{08} while (U.TopKey() <̇ CalculateKey(s_start) OR rhs(s_start) ≠ g(s_start))
{09}    u = U.Pop();
{10}    if (g(u) > rhs(u))
{11}       g(u) = rhs(u);
{12}       for all s ∈ Pred(u) UpdateVertex(s);
{13}    else
{14}       g(u) = ∞;
{15}       for all s ∈ Pred(u) ∪ {u} UpdateVertex(s);
procedure Main()
{16} Initialize();
{17} forever
{18}    ComputeShortestPath();
{19}    Wait for changes in edge costs;
{20}    for all directed edges (u, v) with changed edge costs
{21}       Update the edge cost c(u, v);
{22}       UpdateVertex(u);
```

Figure 17: Backward Version of Lifelong Planning A* Used in the Proofs.

what the goal vertex is. Similarly, $h(s, s'') \leq h(s, s') + h(s', s'') \leq h(s, s') + c^*(s', s'') = h(s, s') + c(s', s'')$ for all vertices $s = s_{start}$, $s' \in \text{Pred}(s'')$, and $s'' \in S$, no matter that the start vertex is. Also, $h(s_{start}, s_{start}) = 0$ since the heuristics are admissible. Thus, the heuristics are backward consistent, no matter that the start vertex is. ∎

B The Proofs – Lifelong Planning A*

In the following, we prove theorems for the backward version of Lifelong Planning A* shown in Figure 17. The theorems then also hold for the backward-version of Lifelong Planning A* shown in Figure 5 since it is a special case where initially $g(s) = \infty$ for all vertices s. This initialization allows for a more efficient implementation since the rhs-value of the goal vertex is zero, all other rhs-values are known to be infinity, and the goal vertex is known to be the only locally inconsistent vertex and thus the only vertex in the priority queue. More importantly, this initialization allows Lifelong Planning A* to avoid having to iterate over all vertices in

Initialize$()$ since the goal vertex is the only vertex in the priority queue initially and the other vertices can thus be initialized only after they have been encountered during the search. This is important because the number of vertices can be large and only a few of them might be reached during the search. The theorems also hold for the optimized version of Lifelong Planning A* since the behavior of the unoptimized and optimized versions are identical.

We only prove those theorems that we later use to prove the correctness of the basic and final versions of D* Lite. Additional theorems about the properties of LPA* are proved in [KL01b]. All theorems hold no matter how the g-values are initialized by the user before Main$()$ is called. Unless stated otherwise, all theorems also hold not matter whether the termination condition of line $\{08\}$ or the alternative termination condition "while U is not empty" is used. The heuristics need to be nonnegative and backward consistent, that is, satisfy $h(s_{start}, s_{start}) = 0$ and $h(s_{start}, s) \leq h(s_{start}, s') + c(s', s)$ for all vertices $s, s' \in \text{Pred}(s)$ with $s \neq s_{start}$. Heuristics that are obtained by relaxing the graph are backward consistent according to Theorems 3 and 4.

In the following, we use $k(u)$ as a shorthand to denote the value returned by CalculateKey(u) and call it the key of vertex $u \in S$. We will show that the key of any vertex in the priority queue is its priority. Thus, U.TopKey$()$ returns the vertex in the priority queue with the smallest key. However, the key is defined for all vertices, while the priority is only defined for the vertices in the priority queue. The subscript $b(u)$ denotes the value of a variable directly before vertex u is expanded, that is, directly before line $\{09\}$ is executed. Similarly, the subscript $a(u)$ denotes the value of a variable after vertex u is expanded, that is, directly before line $\{08\}$ is executed again.

Theorem 5 *The rhs-values of all vertices $u \in S$ always satisfy the following relationship:*

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

Proof: Initialize$()$ initializes the rhs-values so that they satisfy the relationship. The right-hand side of the relationship can then change for a vertex only when the cost of one of its outgoing edges changes or the g-value of one of its successor vertices changes. This can happen on lines $\{11\}$, $\{14\}$ and $\{21\}$. In all of these cases, UpdateVertex$()$ updates the potentially affected rhs-values so that they continue to satisfy the relationship. ■

Theorem 6 *The priority queue contains exactly the locally inconsistent vertices every time line $\{08\}$ is executed.*

Proof: Initialize$()$ initializes the priority queue so that it contains exactly the locally inconsistent vertices. The local consistency of a vertex can then only change when its g-value or its rhs-value changes.

The rhs-value can change only on line $\{05\}$. UpdateVertex() then adds the vertex to the priority queue or deletes it from the priority queue, as necessary, immediately afterwards on lines $\{06\text{-}07\}$. Thus, the theorem continues to hold.

The g-value can change only on lines $\{11\}$ and $\{14\}$.

Whenever ComputeShortestPath() updates the g-value of a locally overconsistent vertex on line $\{11\}$, then the g-value of the vertex is set to its rhs-value. The vertex thus becomes locally consistent and is correctly removed from the priority queue. Thus, the theorem continues to hold.

Whenever ComputeShortestPath() updates the g-value of a locally underconsistent vertex on line $\{14\}$, then the local consistency of the vertex can change. ComputeShortestPath() then calls UpdateVertex() immediately afterwards on line $\{15\}$, which adds the vertex to the priority queue or deletes it from the priority queue, as necessary. Thus, the theorem continues to hold. ■

Theorem 7 *The priority of each vertex $u \in U$ is equal to $k(u)$.*

Proof: Whenever a vertex u gets inserted into the priority queue, its priority equals its key $k(u)$. Its key can then change only when its g-value or rhs-value changes. This can happen on lines $\{05\}$, $\{11\}$ and $\{14\}$. Line $\{05\}$ can update the rhs-value of a vertex. If vertex u remains locally inconsistent, it gets re-inserted into the priority queue with priority $k(u)$. Line $\{11\}$ updates the g-value of a vertex but the vertex is no longer in the priority queue. Finally, line $\{14\}$ updates the g-value of a vertex u. Directly afterwards, line $\{15\}$ calls UpdateVertex(u) which updates its rhs-value. If the vertex remains locally inconsistent, it gets re-inserted into the priority queue with priority $k(u)$. Thus, the relationship continues to hold. ■

Theorem 8 *Assume that vertex u has key $k_{b(u)}(u)$ and is selected for expansion on line $\{09\}$. If vertex v is locally consistent at this point in time but locally inconsistent the next time line $\{08\}$ is executed, then the new key $k_{a(u)}(v)$ of vertex v satisfies $k_{a(u)}(v) \dot{>} k_{b(u)}(u)$ the next time line $\{08\}$ is executed.*

Proof: Assume that vertex u has key $k_{b(u)}(u)$ and is selected for expansion on line $\{09\}$. Vertex v is locally consistent at this point in time but locally inconsistent the next time line $\{08\}$ is executed.

The local consistency of vertex v can only change if its g-value changes or its rhs-value changes. Its rhs-value can change only when the cost of one of its outgoing edges changes or the g-value of one of its successor vertices changes. The edge costs do not change in ComputeShortestPath(). The g-value of vertex v does not change either. Only the g-value of vertex u changes and the two vertices must be different since vertex u is initially in the priority queue and thus locally inconsistent whereas vertex v is locally consistent. Consequently, vertex u must be a successor of vertex v, and the rhs-value of vertex v changes when the g-value of vertex u changes. We distinguish two cases:

Case one: Vertex u was locally overconsistent. Thus, $g_{b(u)}(u) > rhs_{b(u)}(u)$. The assignment on line $\{11\}$ decreases the g-value of vertex u since $g_{a(u)}(u) = rhs_{b(u)}(u) < g_{b(u)}(u) \leq \infty$. This can affect the rhs-value of vertex v only if $rhs_{a(u)}(v) = c(v,u) + g_{a(u)}(u)$. In this case, the rhs-value of vertex v decreased. Its rhs-value must now be smaller than its g-value since it was locally consistent before and thus its rhs-value was equal to its g-value, which did not change. Formally, $rhs_{a(u)}(v) < rhs_{b(u)}(v) = g_{b(u)}(v) = g_{a(u)}(v)$. Putting it all together, it holds that

$$\begin{aligned}
k_{a(u)}(v) &\doteq [\min(g_{a(u)}(v), rhs_{a(u)}(v)) + h(s_{start}, v); \min(g_{a(u)}(v), rhs_{a(u)}(v))] \\
&\doteq [rhs_{a(u)}(v) + h(s_{start}, v); rhs_{a(u)}(v)] \\
&\doteq [c(v,u) + g_{a(u)}(u) + h(s_{start}, v); c(v,u) + g_{a(u)}(u)] \\
&\dot{>} [g_{a(u)}(u) + h(s_{start}, u); g_{a(u)}(u)] \\
&\doteq [rhs_{b(u)}(u) + h(s_{start}, u); rhs_{b(u)}(u)] \\
&\doteq [\min(g_{b(u)}(u), rhs_{b(u)}(u)) + h(s_{start}, u); \min(g_{b(u)}(u), rhs_{b(u)}(u))] \\
&\doteq k_{b(u)}(u).
\end{aligned}$$

We used during the derivation the fact that $h(s_{start}, v) + c(v,u) \geq h(s_{start}, u)$ since the heuristics are backward consistent, and the fact that $c(v,u) + g_{a(u)}(u) > g_{a(u)}(u)$ since the edge cost $c(v,u)$ is positive and the g-value $g_{a(u)}(u)$ is finite.

Case two: Vertex u was locally underconsistent. Thus, $g_{b(u)}(u) < rhs_{b(u)}(u) \leq \infty$. The assignment on line $\{14\}$ increases the g-value of vertex u from a finite value to infinity. This can affect the rhs-value of vertex v only if $rhs_{b(u)}(v) = c(v,u) + g_{b(u)}(u)$. In this case, the rhs-value of vertex v increased. Its rhs-value must now be larger than its g-value since it was locally consistent before and thus its rhs-value was equal to its g-value, which did not change. Formally, $rhs_{a(u)}(v) > rhs_{b(u)}(v) = g_{b(u)}(v) = g_{a(u)}(v)$. Putting it all together, it holds that

$$\begin{aligned}
k_{a(u)}(v) &\doteq [\min(g_{a(u)}(v), rhs_{a(u)}(v)) + h(s_{start}, v); \min(g_{a(u)}(v), rhs_{a(u)}(v))] \\
&\doteq [g_{a(u)}(v) + h(s_{start}, v); g_{a(u)}(v)] \\
&\doteq [rhs_{b(u)}(v) + h(s_{start}, v); rhs_{b(u)}(v)] \\
&\doteq [c(v,u) + g_{b(u)}(u) + h(s_{start}, v); c(v,u) + g_{b(u)}(u)] \\
&\dot{>} [g_{b(u)}(u) + h(s_{start}, u); g_{b(u)}(u)] \\
&\doteq [\min(g_{b(u)}(u), rhs_{b(u)}(u)) + h(s_{start}, u); \min(g_{b(u)}(u), rhs_{b(u)}(u))] \\
&\doteq k_{b(u)}(u).
\end{aligned}$$

We used during the derivation the fact that $h(s_{start}, v) + c(v,u) \geq h(s_{start}, u)$ since the heuristics are backward consistent, and the fact that that $c(v,u)+g_{b(u)}(u) > g_{b(u)}(u)$ since the edge cost $c(v,u)$ is positive and the g-value $g_{b(u)}(u)$ is finite. ■

Theorem 9 *If a locally overconsistent vertex u with key $k_{b(u)}(u)$ is selected for expansion on line $\{09\}$, then it is locally consistent the next time line $\{08\}$ is executed and its new key $k_{a(u)}(u)$ satisfies $k_{a(u)}(u) = k_{b(u)}(u)$.*

Proof: Assume that a locally overconsistent vertex u is selected for expansion on line $\{09\}$. Thus, $\infty \geq g_{b(u)}(u) > rhs_{b(u)}(u)$. Its g-value is then set to its rhs-value on line $\{11\}$ ($g_{a(u)}(u) = rhs_{b(u)}(u)$) and it thus becomes locally consistent. If u is not a predecessor of itself, then its rhs-value does not change and it thus remains locally consistent. If u is a predecessor of itself, then the call to UpdateVertex() on line $\{14\}$ does not change its rhs-value either and it thus remains locally consistent. This follows directly from the definition of the rhs-values if vertex u is the goal vertex. Otherwise, it holds that $rhs_{b(u)}(u) = \min_{v\in \text{Succ}(u)}(c(u,v)+g_{b(u)}(v)) = c(u,w)+g_{b(u)}(w)$ for some vertex $w \neq u$. (Otherwise $rhs_{b(u)}(u) = c(u,u)+g_{b(u)}(u) \geq g_{b(u)}(u)$ which would be a contradiction.) Thus, $c(u,u)+g_{a(u)}(u) = c(u,u)+rhs_{b(u)}(u) > rhs_{b(u)}(u) = c(u,w)+g_{b(u)}(w) = c(u,w)+g_{a(u)}(w)$ and consequently $rhs_{a(u)}(u) = \min(c(u,w)+g_{a(u)}(w), c(u,u)+g_{a(u)}(u)) = c(u,w)+g_{a(u)}(w) = rhs_{b(u)}(u) = g_{a(u)}(u)$, which proves the first part of the theorem. Then,

$$\begin{aligned}
k_{a(u)}(u) &\doteq [\min(g_{a(u)}(u), rhs_{a(u)}(u)) + h(s_{start}, u); \min(g_{a(u)}(u), rhs_{a(u)}(u))] \\
&\doteq [rhs_{a(u)}(u) + h(s_{start}, u); rhs_{a(u)}(u)] \\
&\doteq [rhs_{b(u)}(u) + h(s_{start}, u); rhs_{b(u)}(u)] \\
&\doteq [\min(g_{b(u)}(u), rhs_{b(u)}(u)) + h(s_{start}, u); \min(g_{b(u)}(u), rhs_{b(u)}(u))] \\
&\doteq k_{b(u)}(u).
\end{aligned}$$

■

Theorem 10 *Assume that vertex u has key $k_{b(u)}(u)$ and is selected for expansion on line $\{09\}$. If vertex v is locally inconsistent at this point in time and remains locally inconsistent the next time line $\{08\}$ is executed, then the new key $k_{a(u)}(v)$ of vertex v satisfies $k_{a(u)}(v) \dot{\geq} k_{b(u)}(u)$ the next time line $\{08\}$ is executed.*

Proof: Assume that vertex u has key $k_{b(u)}(u)$ and is selected for expansion on line $\{09\}$. Vertex v is locally inconsistent at this point in time and remains locally inconsistent the next time line $\{08\}$ is executed. Since vertex u is expanded instead of vertex v, it holds that $k_{b(u)}(v) \dot{\geq} k_{b(u)}(u)$. We consider four cases:

Case one: The key of vertex v does not change. Then, it holds that $k_{a(u)}(v) \doteq k_{b(u)}(v) \dot{\geq} k_{b(u)}(u)$.

Case two: The key of vertex v changes, and $v = u$. Vertex $u = v$ was locally underconsistent. (Had it been locally overconsistent, then it would have been locally consistent after its expansion according to Theorem 9, which violates our assumptions.) The g-value of vertex $v = u$ is then set to infinity and thus $g_{a(u)}(u) \geq g_{b(u)}(u)$. Since no other g-value changes, The rhs-value can only change if vertex $v = u$ is a predecessor of itself. However, it is guaranteed not to decrease since the g-value does not decrease. Thus, it holds that $rhs_{a(u)}(u) \geq rhs_{b(u)}(u)$. Putting it all together,

$$\begin{aligned}
k_{a(u)}(v) &\doteq k_{a(u)}(u) \\
&\doteq [\min(g_{a(u)}(u), rhs_{a(u)}(u)) + h(s_{start}, u); \min(g_{a(u)}(u), rhs_{a(u)}(u))] \\
&\dot{\geq} [\min(g_{b(u)}(u), rhs_{b(u)}(u)) + h(s_{start}, u); \min(g_{b(u)}(u), rhs_{b(u)}(u))] \\
&\doteq k_{b(u)}(u).
\end{aligned}$$

Case three: The key of vertex v changes, $v \neq u$, and vertex u was locally overconsistent. The g-value of vertex v does not change since $v \neq u$. Thus, $g_{a(u)}(v) = g_{b(u)}(v)$. Since the key of vertex v changes, its rhs-value changes and thus vertex v is a predecessor of vertex u. Vertex u was locally overconsistent and thus $g_{b(u)}(u) > rhs_{b(u)}(u)$. The assignment on line $\{11\}$ decreases the g-value of vertex u since $g_{a(u)}(u) = rhs_{b(u)}(u) < g_{b(u)}(u) \leq \infty$.

This decrease can affect the rhs-value of vertex v only if $rhs_{a(u)}(v) = c(v,u) + g_{a(u)}(u) = c(v,u) + rhs_{b(u)}(u) = c(v,u) + \min(g_{b(u)}(u), rhs_{b(u)}(u))$. This inequality implies both that $rhs_{a(u)}(v) \geq \min(g_{b(u)}(u), rhs_{b(u)}(u))$ (since $c(v,u) > 0$) and $rhs_{a(u)}(v) + h(s_{start}, v) = c(v,u) + \min(g_{b(u)}(u), rhs_{b(u)}(u)) + h(s_{start}, v) \geq \min(g_{b(u)}(u), rhs_{b(u)}(u)) + h(s_{start}, u)$. (We used during the derivation of the last inequality the fact that $h(s_{start}, v) + c(v,u) \geq h(s_{start}, u)$ since the heuristics are backward consistent.) Putting it all together, it holds that

$$\begin{aligned}
&[rhs_{a(u)}(v) + h(s_{start}, v); rhs_{a(u)}(v)] \\
&\quad \dot{\geq} [\min(g_{b(u)}(u), rhs_{b(u)}(u)) + h(s_{start}, u); \min(g_{b(u)}(u), rhs_{b(u)}(u))] \\
&\quad \doteq k_{b(u)}(u).
\end{aligned} \tag{5}$$

It also holds that

$$\begin{aligned}
&[g_{a(u)}(v) + h(s_{start}, v); g_{a(u)}(v)] \\
&\quad \doteq [g_{b(u)}(v) + h(s_{start}, v); g_{b(u)}(v)] \\
&\quad \dot{\geq} [\min(g_{b(u)}(v), rhs_{b(u)}(v)) + h(s_{start}, v); \min(g_{b(u)}(v), rhs_{b(u)}(v))] \\
&\quad \doteq k_{b(u)}(v) \\
&\quad \dot{\geq} k_{b(u)}(u).
\end{aligned} \tag{6}$$

Then,

$$\begin{aligned} k_{a(u)}(v) &\doteq [\min(g_{a(u)}(v), rhs_{a(u)}(v)) + h(s_{start}, v); \min(g_{a(u)}(v), rhs_{a(u)}(v))] \\ &\dot{\geq} k_{b(u)}(u). \end{aligned}$$

This follows directly from Inequality 5 if $g_{a(u)}(v) \geq rhs_{a(u)}(v)$ and from Inequality 6 if $g_{a(u)}(v) \leq rhs_{a(u)}(v)$.

Case four: The key of vertex v changes, $v \neq u$, and vertex u was locally underconsistent. The g-value of vertex v does not change since $v \neq u$. Thus, $g_{a(u)}(v) = g_{b(u)}(v)$. Since the key of vertex v changes, its rhs-value changes and thus it is a predecessor of vertex u. However, its rhs-value is guaranteed not to decrease since the g-value of vertex u is set to infinity on line $\{14\}$ and thus does not decrease. Thus, $rhs_{a(u)}(v) \geq rhs_{b(u)}(v)$. Putting it all together,

$$\begin{aligned} k_{a(u)}(v) &\doteq [\min(g_{a(u)}(v), rhs_{a(u)}(v)) + h(s_{start}, v); \min(g_{a(u)}(v), rhs_{a(u)}(v))] \\ &\dot{\geq} [\min(g_{b(u)}(v), rhs_{b(u)}(v)) + h(s_{start}, v); \min(g_{b(u)}(v), rhs_{b(u)}(v))] \\ &\doteq k_{b(u)}(v) \\ &\dot{\geq} k_{b(u)}(u). \blacksquare \end{aligned}$$

Theorem 11 *The keys of the vertices that are selected for expansion on line* $\{09\}$ *are monotonically nondecreasing over time until ComputeShortestPath*() *terminates.*

Proof: Assume that vertex u is selected for expansion on line $\{09\}$. At this point, its key $k_{b(u)}(u)$ is a smallest key of all vertices in the priority queue, that is, of all locally inconsistent vertices according to Theorem 6. If a locally consistent vertex v becomes locally inconsistent due to the expansion of vertex u, then its new key $k_{a(u)}(v)$ satisfies $k_{a(u)}(v) \dot{>} k_{b(u)}(u)$ according to Theorem 8. If a locally inconsistent vertex v remains locally inconsistent, then its new key $k_{a(u)}(v)$ satisfies $k_{a(u)}(v) \dot{\geq} k_{b(u)}(u)$ according to Theorem 10. Thus, when the next vertex is selected for expansion on line $\{09\}$, its key is at least as large as $k_{b(u)}(u)$. $\blacksquare$

Theorem 12 *Let* k = *U.TopKey*() *during the execution of line* $\{08\}$. *If vertex* u *is locally consistent at this point in time with* $k(u) \dot{\leq} k$, *then it remains locally consistent until ComputeShortestPath*() *terminates.*

Proof by contradiction: If U is empty, then U.TopKey() returns $[\infty; \infty]$ and thus U.TopKey()$\dot{\geq} k(s_{start})$. Also $rhs(s_{start}) = g(s_{start})$ since all vertices are locally consistent. Consequently, the termination condition is satisfied and thus the theorem is trivial. (Similarly, the termination condition is satisfied trivially if the alternative termination condition "while U is not empty" is used.) Thus, we assume that U is not empty.

Assume that vertex u is locally consistent during the execution of line $\{08\}$. Let $g(u)$, $rhs(u)$, and $k(u)$ be the g-value, rhs-value, and key of vertex u (respectively) at this point in time. Then, $g(u) = rhs(u)$ since vertex u is locally consistent. Similarly, $k \doteq$U.TopKey() at this point in time. Assume that $k(u) \dot{\leq} k$ and that u becomes locally inconsistent later during the expansion of some vertex v. When v is chosen for expansion, it must be locally inconsistent since only locally inconsistent vertices get expanded. Thus, $v \neq u$. Then, $k_{a(v)}(u) \dot{>} k_{b(v)}(v)$ according to Theorem 8 and $k_{b(v)}(v) \dot{\geq} k$ according to Theorem 11. Consequently,

$$\begin{aligned}
&[\min(g_{a(v)}(u), rhs_{a(v)}(u)) + h(s_{start}, u); \min(g_{a(v)}(u), rhs_{a(v)}(u))] \\
&\quad \doteq \; k_{a(v)}(u) \\
&\quad \dot{>} \; k_{b(v)}(v) \\
&\quad \dot{\geq} \; k \\
&\quad \dot{\geq} \; k(u) \\
&\quad \doteq \; [\min(g(u), rhs(u)) + h(s_{start}, u); \min(g(u), rhs(u))] \\
&\quad \doteq \; [g(u) + h(s_{start}, u); g(u)]
\end{aligned}$$

and thus $g_{a(v)}(u) \geq \min(g_{a(v)}(u), rhs_{a(v)}(u)) > g(u)$. However, $g_{a(v)}(u) = g(u)$ since vertex u has been locally consistent all the time and thus could not have been assigned a new g-value, which is a contradiction. Consequently, u remains locally consistent until ComputeShortestPath() terminates. ■

Theorem 13 *If a locally overconsistent vertex u is selected for expansion on line $\{09\}$, then it is locally consistent the next time line $\{08\}$ is executed and remains locally consistent until ComputeShortestPath() terminates.*

Proof: If a locally overconsistent vertex u is selected for expansion on line $\{09\}$, then it becomes locally consistent according to Theorem 9. Let $k =$ U.TopKey() during the execution of line $\{08\}$ before u is selected for expansion on line $\{09\}$, and $k' =$ U.TopKey() during the execution of line $\{08\}$ after u is selected for expansion on line $\{09\}$. Then, $k_{a(u)}(u) \doteq k_{b(u)}(u)$ according to Theorem 9, $k_{b(u)}(u) \doteq k$ since u was selected for expansion, $k \dot{\leq} k'$ according to Theorem 11 if the priority queue is not empty during the execution of line $\{08\}$ after u is selected for expansion on line $\{09\}$, and $k \dot{\leq} k'$ if the priority queue is empty since $k' \doteq [\infty; \infty]$. Putting everything together, it holds that $k_{a(u)}(u) \dot{\leq} k'$. To summarize, vertex u is locally consistent during the next execution of line $\{08\}$ after u is selected for expansion on line $\{09\}$ with $k_{a(u)}(u) \dot{\leq} k'$. Consequently, it remains locally consistent until ComputeShortestPath() terminates, according to Theorem 12. ■

Theorem 14 *If line $\{08\}$ is changed to "while U is not empty," then ComputeShortestPath$()$ expands each vertex at most twice, namely at most once when it is locally underconsistent and at most once when it is locally overconsistent. The g-values of all vertices after termination equal their respective goal distances.*

Proof: Assume that line $\{08\}$ is changed to "while U is not empty." Then, ComputeShortestPath$()$ terminates when all vertices are locally consistent. When a locally overconsistent vertex is selected for expansion, it becomes locally consistent and remains locally consistent according to Theorem 13. Thus, every vertex is expanded at most once when it is locally overconsistent. Similarly, when a locally underconsistent vertex is selected for expansion, its g-value is set to infinity and the vertex can thus only be locally consistent or locally overconsistent before it is expanded again. (It cannot be locally underconsistent because its g-value is infinity and cannot be changed before its next expansion.) Thus, if the vertex is expanded again, it must be locally overconsistent. (Locally consistent vertices do not get expanded.) As already shown, it then becomes locally consistent and remains locally consistent. To summarize, every vertex is expanded at most twice before all vertices are locally consistent, namely at most once when it is locally underconsistent and at most once when it is locally overconsistent, and ComputeShortestPath$()$ thus terminates.

When all vertices are locally consistent, then $g(s) = rhs(s) = 0$ if $s = s_{goal}$ and $g(s) = rhs(s) = \min_{s' \in \text{Succ}(s)}(c(s,s') + g(s'))$ otherwise. Thus, the g-values satisfy the definition of the goal distances and thus equal them. ■

Theorem 15 *Let $k = U.\text{TopKey}()$ during the execution of line $\{08\}$. If vertex u is locally consistent at this point in time with $k(u) \dot{\leq} k$, then the g-value of state u equals its minimax goal distance and one can follow a shortest path from u to s_{goal} by always moving from the current vertex s, starting at u, to any successor s' that minimizes $c(s,s') + g(s')$ until s_{goal} is reached (ties can be broken arbitrarily).*

Proof: If U is empty, then the theorem follows from Theorem 14. Thus, we assume that U is not empty.

Assume that vertex u is locally consistent during the execution of line $\{08\}$. Let $g(s)$, $rhs(s)$, and $k(s)$ be the g-value, rhs-value, and key of any vertex s (respectively) at this point in time. Then, $g(u) = rhs(u)$ since state u is locally consistent.

We first show by contradiction that $g(u) < \infty$. Assume that $g(u) = \infty$. Then, $g(u) = rhs(u) = \infty$ since u is locally consistent. Thus, $k(u) \doteq [\min(g(u), rhs(u)) + h(s_{start}, u); \min(g(u), rhs(u))] \doteq [\infty; \infty]$. Consequently, $k \doteq [\infty; \infty]$ since $k(u) \dot{\leq} k$. Let v be a locally inconsistent vertex with key k. Such a vertex exists since we assume that U is not empty. Then, $g(v) = rhs(v) = \infty$. Thus, vertex v must be locally consistent, which is a contradiction. Consequently, it holds that $g(u) < \infty$.

If $u = s_{goal}$ then $g(u) = rhs(u) = 0$ since vertex u is locally consistent and $rhs(u) = 0$ per definition. Thus, $g(u) = gd(u)$. Furthermore, one can trivially follow a shortest path from u to s_{goal} by always moving from the current vertex s, starting at u, to any successor s' that minimizes $c(s, s') + g(s')$ until s_{goal} is reached. Thus, we assume in the following that $u \neq s_{goal}$.

Let w be any successor of vertex u that minimizes $c(u, w) + g(w)$. We now show that vertex w is locally consistent during the execution of line $\{08\}$ with $k(w)\dot{\leq}k$. It holds that $g(u) = rhs(u) = \min_{s' \in \text{Succ}(u)}(c(u, s') + g(s')) = c(u, w) + g(w)$. Thus, $g(w) < g(u)$ since $g(u) < \infty$ and $g(u) < \infty$. Furthermore, $g(w) + h(s_{start}, w) \leq g(u) - c(u, w) + h(s_{start}, u) + c(u, w) = g(u) + h(s_{start}, u)$ since the heuristics are backward consistent and thus $h(s_{start}, w) \leq h(s_{start}, u) + c(u, w)$. Consequently,

$$\begin{aligned}
k(w) &\doteq [\min(g(w), rhs(w)) + h(s_{start}, w); \min(g(w), rhs(w))] \\
&\dot{\leq} [g(w) + h(s_{start}, w); g(w)] \\
&\dot{<} [g(u) + h(s_{start}, u); g(u)] \\
&\doteq [\min(g(u), rhs(u)) + h(s_{start}, u); \min(g(u), rhs(u))] \\
&\doteq k(u) \\
&\dot{\leq} k.
\end{aligned}$$

Thus, $k(w)\dot{<}k$. This shows that vertex w is locally consistent during the execution of line $\{08\}$ with $k(w)\dot{\leq}k$ since k is the smallest key of any locally inconsistent vertex.

We now show that $g(u) = gd(u)$ and $g(w) = gd(w)$ during the execution of line $\{08\}$. Both vertices are locally consistent and their keys are less than or equal to the smallest key of any locally inconsistent vertex. Thus, they remain locally consistent and thus their g-values do not get updated until ComputeShortestPath() terminates even if line $\{08\}$ is changed to “while U is not empty,” according to Theorem 12. Furthermore, the g-values of vertices u and w equal their respective goal distances after termination if line $\{08\}$ is changed to “while U is not empty,” according to Theorem 14. Thus, $g(u) = gd(u)$ and $g(w) = gd(w)$ during the execution of line $\{08\}$. These relationships must also hold for the termination condition actually used by LPA* since the values that LPA* assigns to the g-values of vertices do not depend on the termination condition.

We now show that moving from vertex u to any vertex w that minimizes $c(u, w) + g(w)$ is the beginning of a shortest path from u to s_{goal}. This is indeed the case since $gd(u) = g(u) = c(u, w) + g(w) = c(u, w) + gd(w)$. Finally, we can repeatedly apply this property to show that one can follow a shortest path from u to s_{goal} by always moving from the current vertex s, starting at u, to any successor s' that minimizes $c(s, s') + g(s')$ since vertex w is again locally consistent with $k(w)\dot{\leq}k$. ■

Theorem 16 *ComputeShortestPath() expands a vertex at most twice, namely at most once when it is locally underconsistent and at most once when it is locally overconsistent. After termination, one can follow a shortest path from s_{start} to s_{goal} by always moving from the current vertex s, starting at s_{start}, to any successor s' that minimizes $c(s, s') + g(s')$ until s_{goal} is reached (ties can be broken arbitrarily).*

Proof: ComputeShortestPath() terminates after it has expanded every vertex at most twice, namely at most once when it is locally underconsistent and at most once when it is locally overconsistent according to Theorem 14 if line $\{08\}$ is changed to "while U is not empty." It continues to terminate at least when U is empty even if line $\{08\}$ is not changed because U.TopKey() then returns $[\infty; \infty]$ and thus U.TopKey()$\dot{\geq} k(s_{start})$ and because $rhs(s_{start}) = g(s_{start})$ since all vertices are locally consistent. Thus, the termination condition is satisfied. This means that U will eventually become empty and ComputeShortestPath() will terminate after it has expanded every vertex at most twice, namely at most once when it is locally underconsistent and at most once when it is locally overconsistent, if it does not already terminate earlier.

$k \dot{\geq} k(s_{start})$ and $rhs(s_{start}) = g(s_{start})$ after termination according to the termination condition, where $k =$ U.TopKey() during the execution of line $\{08\}$. Consequently, s_{start} satisfies the conditions of Theorem 15 after termination. The theorem then follows directly from Theorem 15. ■

C The Proofs – Basic Version of D* Lite

In the following, we prove the theorem stated in the main text for the basic version of D* Lite shown in Figure 6. The assumptions are the same as for Lifelong Planning A*. The heuristics again need to be backward consistent, that is, satisfy $h(s_{start}, s_{start}) = 0$ and $h(s_{start}, s) \leq h(s_{start}, s') + c(s', s)$ for all vertices $s, s' \in \text{Pred}(s)$ with $s \neq s_{start}$. Now, however, they need to satisfy this property for all $s_{start} \in S$ since the robot moves and thus changes s_{start}. The proof also shows that many other properties of ComputeShortestPath() transfer from Lifelong Planning A* to the basic version of D* Lite.

Theorem 17 *ComputeShortestPath() of the basic version of D* Lite expands a vertex at most twice, namely at most once when it is locally underconsistent and at most once when it is locally overconsistent, and thus terminates. One can then follow a shortest path from s_{start} to s_{goal} by always moving from the current vertex s, starting at s_{start}, to any successor s' that minimizes $c(s, s') + g(s')$ until s_{goal} is reached (ties can be broken arbitrarily).*

Proof: One can easily show by induction that each time directly before the basic version of D* Lite calls ComputeShortestPath() all variables have the same values that they have directly before Lifelong Planning A* calls ComputeShortestPath() for the first time provided that the graph topology, edge costs, start vertex, goal vertex, heuristics, and g-values of all vertices are identical to those directly before the basic version of D* Lite calls ComputeShortestPath$()$. In particular, in both cases, the rhs-values of all vertices satisfy the relationship of Theorem 5, exactly the locally inconsistent vertices are in the priority queue, and the priority of the vertices in the priority queue satisfy the relationship of Theorem 7. ComputeShortestPath$()$ then behaves identically in both cases and all properties proved in the context of Lifelong Planning A* continue to hold. Theorem 16 then proves the theorem. ■

D The Proofs – Final Version of D* Lite

In the following, we prove the theorem stated in the main text for the final version of D* Lite shown in Figure 7. The heuristics need to be nonnegative, admissible, and forward-backward consistent, that is, satisfy $h(s,s') \leq c^*(s,s')$ and $h(s,s'') \leq h(s,s') + h(s',s'')$ for all vertices $s, s', s'' \in S$. Heuristics that are obtained by relaxing the graph are admissible and forward-backward consistent according to Theorem 3. They are also backward consistent according to Theorem 4 and we can thus continue to use the proofs for Lifelong Planning A* and the basic version of D* Lite. The theorem then also holds for the optimized version of the final version of D* Lite since the behavior of the unoptimized and optimized versions are identical. The proof also shows that many other properties of ComputeShortestPath$()$ transfer from Lifelong Planning A* to the final version of D* Lite.

Theorem 18 *ComputeShortestPath$()$ of the final version of D* Lite expands a vertex at most twice, namely at most once when it is locally underconsistent and at most once when it is locally overconsistent, and thus terminates. One can then follow a shortest path from s_{start} to s_{goal} by always moving from the current vertex s, starting at s_{start}, to any successor s' that minimizes $c(s,s') + g(s')$ until s_{goal} is reached (ties can be broken arbitrarily).*

Proof: We compare the behavior of each call to ComputeShortestPath$()$ of the final version of D* Lite and the first call to ComputeShortestPath$()$ of Lifelong Planning A*. In other words, we consider the state of the system directly before the nth call to ComputeShortestPath$()$ of the final version of D* Lite. We then start Lifelong Planning A* with the same graph topology, edge costs, start vertex, goal vertex, heuristics, and g-values. We also assume that line $\{08\}$ of Lifelong Planning A* is changed to “while U is not empty.”

We say that ComputeShortestPath$()$ of the final version of D* Lite expands a vertex when it executes lines $\{15\text{”-}20\text{”}\}$. These are exactly the same operations that ComputeShortestPath$()$ of Lifelong Planning A* executes when it expands a vertex, except for how CalculateKey$()$ calculates the priorities.

All values in the following refer to the point in time directly before a vertex is selected for expansion by both versions of ComputeShortestPath$()$ (that is, the one of the final version of

D* Lite and the one of Lifelong Planning A*), unless noted otherwise. We use $k(s)$ to denote the priority of vertex s that CalculateKey(s) of Lifelong Planning A* calculates. This is the same as the priority of vertex s in the priority queue of Lifelong Planning A* if the vertex is locally inconsistent and thus in the priority queue, according to Theorems 6 and 7. We use $k'(s)$ to denote the priority of vertex s that CalculateKey(s) of the final version of D* Lite calculates and $k''(s)$ to denote the priority of vertex s in the priority queue of the final version of D* Lite if it is in the priority queue. These two values can be different.

One can easily show by induction that each time directly before the final version of D* Lite calls ComputeShortestPath() the rhs-values of all vertices satisfy the relationship of Theorem 5 and exactly the locally inconsistent vertices are in the priority queue. Since per construction the g-values of all corresponding vertices are identical for both versions of ComputeShortestPath(), the rhs-values of each vertex are identical as well and the priority queues contain the same vertices.

We now prove by induction on n that, directly before both versions of ComputeShortestPath() select the nth vertex for expansion, the g-values and rhs-values of all corresponding vertices are identical and the priority queues contain the same vertices. This statement is true for $n = 1$. Now assume that it is true directly before both versions of ComputeShortestPath() select the nth vertex for expansion and that ComputeShortestPath() of the final version of D* Lite selects vertex $s \in U$ for expansion.

Since ComputeShortestPath() of the final version of D* Lite selects vertex s for expansion, it satisfies $k''(s) \dot{\leq} k''(s')$ for all $s' \in U$ due to the U.Pop() operation on line {12”} and $k'(s) \dot{\leq} k''(s)$ since the condition on line {13”} is false.

We first show that $k(s') \dot{\leq} k'(s')$ for all $s' \in S$. Since $k_m \geq 0$, it holds that

$$\begin{aligned} k(s') &\doteq [\min(g(s'), rhs(s')) + h(s_{start}, s'); \min(g(s'), rhs(s'))] \\ &\dot{\leq} [\min(g(s'), rhs(s')) + h(s_{start}, s') + k_m; \min(g(s'), rhs(s'))] \\ &\doteq k'(s'). \end{aligned}$$

We now show that $k''(s') \dot{\leq} k'(s')$ for all $s' \in U$. Assume that the value of s_{start} was $s^{s'}_{start}$, the value of s_{last} was $s^{s'}_{last}$, and the value of k_m was $k^{s'}_m$ when vertex s' was entered the last time into the priority queue of the final version of D* Lite. Afterwards, neither the g-value nor the rhs-value of vertex s' have changed because, every time one of these values changes, the vertex is removed from the priority queue and thus would have been inserted into the priority queue again. Assume that the values of s_{last}, from the time when vertex s' was inserted into the priority queue until it is selected for expansion with all the values assigned to s_{last} in between, are $s_1 = s^{s'}_{last}, s_2, \ldots, s_n = s_{last}$. Then, $k_m = k^{s'}_m + h(s^{s'}_{last}, s_2) + \ldots + h(s_{n-1}, s_{last})$. Since $s^{s'}_{last} = s^{s'}_{start}$ and $s_{last} = s_{start}$, it holds that $k_m + h(s_{start}, s') = k^{s'}_m + h(s^{s'}_{last}, s_2) + \ldots + h(s_{n-1}, s_{last}) + h(s_{start}, s') = k^{s'}_m + h(s^{s'}_{start}, s_2) + \ldots + h(s_{n-1}, s_{last}) + h(s_{last}, s') \geq k^{s'}_m + h(s^{s'}_{start}, s')$ because the heurisics are forward-backward consistent. Thus, it holds that

$$\begin{aligned}
k''(s') &\doteq [\min(g(s'), rhs(s')) + h(s^{s'}_{start}, s') + k^{s'}_m; \min(g(s'), rhs(s'))] \\
&\dot{\leq} [\min(g(s'), rhs(s')) + h(s_{start}, s') + k_m; \min(g(s'), rhs(s'))] \\
&\doteq k'(s').
\end{aligned}$$

Putting everything together, it holds that $k'(s)\dot{\leq}k''(s)\dot{\leq}k''(s')\dot{\leq}k'(s')$ for all $s' \in U$. Thus,

$$\begin{aligned}
&k'(s)\dot{\leq}k'(s') \\
&[\min(g(s), rhs(s)) + h(s_{start}, s) + k_m; \min(g(s), rhs(s))] \\
&\qquad\dot{\leq}[\min(g(s'), rhs(s')) + h(s_{start}, s') + k_m; \min(g(s'), rhs(s'))] \\
&[\min(g(s), rhs(s)) + h(s_{start}, s); \min(g(s), rhs(s))] \\
&\qquad\dot{\leq}[\min(g(s'), rhs(s')) + h(s_{start}, s'); \min(g(s'), rhs(s'))] \\
&k(s)\dot{\leq}k(s')
\end{aligned}$$

for all $s' \in U$. Thus, the vertex selected for expansion by ComputeShortestPath() of the final version of D* Lite is the one in the priority queue with the smallest priority $k(s)$. Thus, ComputeShortestPath() of the final version of D* Lite always expands the vertex s in the priority queue with the smallest priority $k(s)$, just like ComputeShortestPath() of Lifelong Planning A*. During the expansion, both versions of ComputeShortestPath() execute exactly the same operations, except for how CalculateKey() calculates the priorities. The extra lines {13”-14”} of ComputeShortestPath() of the final version of D* Lite do not change the g-value or rhs-value of any vertex nor which vertices are in the priority queue. They only change the priority of a vertex in the priority queue. Thus, directly before both versions of ComputeShortestPath() select the $(n + 1)$st vertex for expansion, the g-values and rhs-values of all corresponding vertices are identical and the priority queues contain the same vertices.

This concludes the proof that, directly before both versions of ComputeShortestPath() select the a vertex for expansion, the g-values and rhs-values of all corresponding vertices are identical and the priority queues contain the same vertices. Thus, according to Theorem 6, the priority queue of the final version of D* Lite always contains exactly the locally inconsistent vertices. As part of the proof we have also shown that, before one of the versions of ComputeShortestPath() terminates, they both expand the same vertices in the same order.

Assume that line {10”} of the final version of D* Lite is changed to “while U is not empty.” Then, the resulting version of ComputeShortestPath() of D* Lite terminates at the same time as ComputeShortestPath() of Lifelong Planning A* since both priority queues always contain the same vertices. According to Theorem 14, ComputeShortestPath() of the final version of D* Lite then terminates after it has expanded every vertex at most twice, namely at most once when it is locally underconsistent and at most once when it is locally overconsistent since both versions of ComputeShortestPath() expand the same vertices in the same order.

Now assume that line $\{10"\}$ of the final version of D* Lite remains unchanged. ComputeShortestPath() of the final version of D* Lite continues to terminate at least when U is empty because U.TopKey() then returns $[\infty;\infty]$ and thus U.TopKey()$\dot{\geq}$CalculateKey(s_{start}) and because $rhs(s_{start}) = g(s_{start})$ since all vertices are locally consistent. Thus, the termination condition is satisfied. This means that U will eventually become empty and ComputeShortestPath() of the final version of D* Lite will terminate after it has expanded every vertex at most twice, namely at most once when it is locally underconsistent and at most once when it is locally overconsistent, if it does not already terminate earlier.

Now assume that ComputeShortestPath() of the final version of D* Lite terminates but the priority queue is not empty. According to its termination condition, it holds that $k''(s')\dot{\geq}k'(s_{start})$ for all $s' \in U$. Then, $k'(s')\dot{\geq}k''(s')\dot{\geq}k'(s_{start})$ for all $s' \in U$. $k'(s')\dot{\geq}k'(s_{start})$ for all $s' \in U$ in turn implies that $k(s')\dot{\geq}k(s_{start})$ for all $s' \in U$. This is so because $k'(s')$ is identical to $k(s')$ except that its first component is larger by the current value of k_m. The same is true for $k'(s_{start})$ and $k(s_{start})$. Since, according to the termination condition, it also holds that $rhs(s_{start}) = g(s_{start})$, s_{start} satisfies the conditions of Theorem 15. The theorem then follows directly from Theorem 15 since the g-values of all corresponding vertices are identical for both versions of ComputeShortestPath(). ■