Improved Fast Replanning for Robot Navigation in Unknown Terrain*

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 paper, we introduce an alternative to Focussed Dynamic A* that implements the same navigation strategy but is algorithmically different. Focussed Dynamic A* Lite is simpler, easier to understand, easier to analyze and easier to extend than Focussed Dynamic A*, yet is more efficient. We believe that our results will make D*-like replanning algorithms 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 paper, 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. A robot can solve it with the following navigation strategy: It always plans a shortest path from its current coordinates to the goal coordinates under the assumption that unknown terrain is traversable. 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 blocked. 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, this navigation strategy 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 blocked. 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 [13]. Several solutions to this problem have been proposed in the robotics literature [18, 1, 16, 11, 3, 6]. Focussed Dynamic A* (D*) [14] is probably the most popular solution 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* [10] searches by modifying previous search results locally. D* has been extensively used on real robots, including outdoor HMMWVs [15], including UGV Demo II vehicles as part of the DARPA Unmanned Ground Vehicle program. It is currently also being integrated into Mars Rover prototypes and tactical mobile robot prototypes for urban reconnaissance [5, 9, 17]. D* is also used as part of other software, including the GRAMMPS mission planner for multiple robots [2].

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* algorithm [7], 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

*We thank Anthony Stentz for his support of this work. 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.

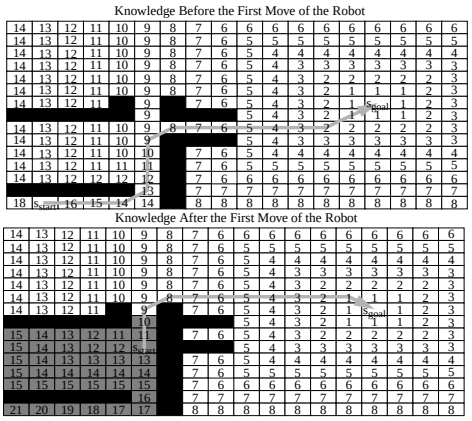

Figure 1: Simple Example (Part 1).

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 solving navigation tasks other than goal-directed navigation in unknown terrain.

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 vertex and has to move to the goal vertex. It always computes a shortest path from its current vertex to the goal vertex under the assumption that cells with unknown blockage status are traversable. It then follows this path until it reaches the goal vertex, in which case it stops successfully, or some edge costs change, in which case it recomputes a shortest path from the current vertex to the goal vertex. It can utilize initial knowledge about the blockage status of cells in case it is available. Figure 1 shows the goal distances of all traversable cells and the shortest paths both before and after the robot has moved along the path and discovered the first blocked cell it did not know about. Cells whose goal distances have changed are shaded gray. The goal distances are important because one can easily determine a shortest path from the current vertex 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 vertex of the robot to the goal vertex. Thus, one can efficiently recalculate a shortest path from the current vertex of the robot to the goal vertex 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 vertices efficiently.

3 Lifelong Planning A*

We first describe Lifelong Planning A* (LPA*) [7], our incremental heuristic search method that repeatedly determines shortest paths between two given vertices as the edge costs of a graph change. LPA* is an incremental search method that uses heuristics to focus 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) [4] 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 [10]. LPA* thus recalculates only very few start distances. We later use LPA* to develop D* Lite.

3.1 Notation

LPA* searches directed graphs, just like A*. We use the following notation. S denotes the finite set of vertices of the graph. $Succ(s) \subseteq S$ denotes the set of successors of $s \in S$ in the graph. Similarly, $Pred(s) \subseteq S$ denotes the set of predecessors of $s \in S$ in the graph. $0 < c(s, s') \leq \infty$ denotes the cost of moving from s to $s' \in Succ(s)$. The task of LPA* is to maintain a shortest path from the start vertex $s_{start} \in S$ to the goal vertex $s_{goal} \in S$, knowing both the topology of the graph and the current edge costs.

3.2 Lifelong Planning A*

LPA* is shown in Figure 2. It maintains an estimate $g(s)$ of the start distance of each vertex s. LPA* also maintains rhs-values, a second kind of estimates 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:

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

A vertex is called consistent iff its g-value equals its rhs-value, otherwise it is called inconsistent. This concept is important because the g-values of all vertices equal their start distances iff all vertices are consistent. However, LPA* does not make every vertex consistent after some of the edge costs have changed. First, we can prove that LPA* does not

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 = ∅;
{03} for all s ∈ S rhs(s) = g(s) = ∞;
{04} rhs(s_start) = 0;
{05} U.Insert(s_start, CalculateKey(s_start));

procedure UpdateVertex(u)
{06} if (u ≠ s_start) rhs(u) = min_{s'∈Pred(u)} (g(s') + c(s', u));
{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_goal) OR rhs(s_goal) ≠ g(s_goal))
{10}     u = U.Pop();
{11}     if (g(u) > rhs(u))
{12}         g(u) = rhs(u);
{13}         for all s ∈ Succ(u) UpdateVertex(s);
{14}     else
{15}         g(u) = ∞;
{16}         for all s ∈ Succ(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(v);
```

Figure 2: Lifelong Planning A* (forward search).

recompute the start distances that have been computed before and have not changed (incremental search) [8], similar to DynamicSWSF-FP [12]. Second, LPA* uses heuristic knowledge (in form of approximations of the goal distances) to focus the search and determine that some start distances need not get computed at all (heuristic search), similar to A* [10]. The heuristics $h(s, s')$ estimate the distance between vertex s and vertex s'. They need to 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 $s' \in Succ(s)$. We can prove that ComputeShortestPath() of LPA* recalculates the g-value of each vertex at most twice and thus terminates. 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). Thus, LPA* does not explicitly maintain a search tree. Instead, it uses the g-values to encode it implicitly.

3.3 Similarity of Lifelong Planning A* and A*

LPA* is an incremental version of A*, the most popular search method in artificial intelligence, and thus shares many similarities with it. For example, both search methods maintain a priority queue. The priority queue of LPA* always contains exactly the inconsistent vertices. These are the vertices whose g-values LPA* potentially needs to update to make them consistent. The priority of vertex s in the priority queue is always

$$k(s) = [k_1(s); k_2(s)], \quad (2)$$

a vector with two components where $k_1(s) = \min(g(s), rhs(s)) + h(s, s_{goal})$ and $k_2(s) = \min(g(s), rhs(s))$. The priorities are compared according to a lexicographic ordering. For example, priority $k(s)$ is smaller than or equal to priority $k'(s)$, denoted by $k(s) \dot{\leq} k'(s)$, iff either $k_1(s) < k'_1(s)$ or ($k_1(s) = k'_1(s)$ and $k_2(s) \leq k'_2(s)$). LPA* recalculates the g-values of vertices in the priority queue ("expands the vertices" by executing lines {10-16}) in the order of increasing first priority components, which correspond to the f-values of an A* search, and vertices with equal first priority components in order of increasing second priority components, which correspond to the g-values of an A* search. Thus, it expands vertices in a similar order as an A* search, that expands vertices in the order of increasing f-values (since the heuristics are consistent) and vertices with equal f-values that are on the same branch of its search tree in order of increasing g-values. A more detailed and formal description of LPA*, its comparison to A*, and proofs of its correctness can be found in [8].

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, 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. We 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 2 searches from the start vertex to the goal vertex and thus its g-values are estimates of the start distances. The version presented

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 3: Lifelong Planning A* (backward search).

in Figure 3 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 version by reversing all edges of the graph and exchanging the start and goal vertex. The heuristics $h(s, s')$ now need to 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 $s' \in Pred(s)$. More generally, since the robot moves and thus changes s_{start}, the heuristics needs to satisfy this property for all $s_{start} \in S$. 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 4 shows the resulting algorithm, called the basic version of D* Lite.

The basic version of D* Lite first calls Initialize() {17'} to initialize the g-values of the vertices to infinity, the rhs-values of the vertices so that they satisfy the equivalent of Equation 1, and the priority queue so that it contains exactly the inconsistent vertices with priorities that satisfy the equivalent of Equation 2. The priority queue only contains the goal vertex since all other vertices are initially consistent. Thus, in an actual implementation, the basic version of D* Lite needs to initialize a vertex only when it is encountered 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

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 = ∅;
{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'} ComputeShortestPath();
{19'} while (s_start ≠ s_goal)
{20'}     /* if (g(s_start) = ∞) then there is no known path */
{21'}     s_start = arg min_{s'∈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 ∈ U
{29'}             U.Update(s, CalculateKey(s));
{30'}         ComputeShortestPath();
```

Figure 4: D* Lite: Basic Version.

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'}. If some edge costs have changed, it updates the edge costs {26'} and calls UpdateVertex() {27'} to update the rhs-values of the vertices potentially affected by the changed edge costs so that they again satisfy the equivalent of Equation 1 and the priority queue so that it again contains exactly the inconsistent vertices with priorities that satisfy the equivalent of Equation 2. The basic version of D* Lite then updates the priorities of all vertices in the priority queue {28'-29'}, recalculates a shortest path {30'}, and iterates.

We can prove a variety of properties of the basic version of D* Lite, including the correctness of its ComputeShortestPath() function which implies the correctness of the basic version of D* Lite (all proofs can be found in [8]):

Theorem 1 *ComputeShortestPath() of the basic version of D* Lite always terminates and one can then follow a short-*

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 5: D* Lite: Final Version.

est 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 can be expensive since the priority queue often contains a large number of vertices. The final version of D* Lite, shown in Figure 5, uses a method derived from D* [14] 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 satisfy $h(s, s') \leq c^*(s, s')$ and $h(s, s'') \leq h(s, s') + h(s', s'')$ for all vertices $s, s', s'' \in S$, where $c^*(s, s')$ denotes the cost of a shortest path from vertex $s \in S$ to vertex $s' \in S$. This property implies the property that heuristics for the basic version of D* Lite need to satisfy. This is not overly restrictive, however, since both properties are guaranteed to hold if the heuristics are derived by relaxing the search problem, which will almost always be the case and holds for the heuristics used in this paper.

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 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. 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). 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 the correctness of ComputeShortestPath() which implies the correctness of the final version of D* Lite:

Theorem 2 *ComputeShortestPath() of the final version of D* Lite always terminates and one can then follow a shortest path from s_{start} to s_{goal} by always moving from the cur-*

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 = ∅;
{03'''} k_m = 0;
{04'''} for all s ∈ S rhs(s) = g(s) = ∞;
{05'''} rhs(s_goal) = 0;
{06'''} U.Insert(s_goal, [h(s_start, s_goal); 0]);

procedure UpdateVertex(u)
{07'''} if (g(u) ≠ rhs(u) AND u ∈ U) U.Update(u, CalculateKey(u));
{08'''} else if (g(u) ≠ rhs(u) AND u ∉ U) U.Insert(u, CalculateKey(u));
{09'''} else if (g(u) = rhs(u) AND u ∈ U) U.Remove(u);

procedure ComputeShortestPath()
{10'''} while (U.TopKey() < 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 < 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 ∈ Pred(u)
{20'''}       if (s ≠ 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) = ∞;
{25'''}     for all s ∈ Pred(u) ∪ {u}
{26'''}       if (rhs(s) = c(s, u) + g_old OR s = u)
{27'''}         if (s ≠ s_goal) rhs(s) = min_{s'∈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 ≠ s_goal)
{33'''}   /* if (g(s_start) = ∞) then there is no known path */
{34'''}   s_start = arg min_{s'∈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 ≠ s_goal) rhs(u) = min(rhs(u), c(u, v) + g(v));
{45'''}       else if (rhs(u) = c_old + g(v))
{46'''}         if (u ≠ s_goal) rhs(u) = min_{s'∈Succ(u)} (c(u, s') + g(s'));
{47'''}       UpdateVertex(u);
{48'''}     ComputeShortestPath();
```

Figure 6: D* Lite: Final Version (optimized version).

rent 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).

4.3 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 D* Lite, the optimized version of which is shown in Figure 6.

First, 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"'}).

Second, 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"'}).

Third, 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"'}).

Fourth, 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"'}).

5 An Example

We illustrate D* Lite using the two eight-connected grids from Figure 1. To make the search algorithms comparable, their search always starts at s_{goal} and proceeds towards the current cell of the robot. 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 algorithms are shaded gray in Figure 7. (We consider the current cell of the robot to be expanded by breadth-first search and A*.) 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 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 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

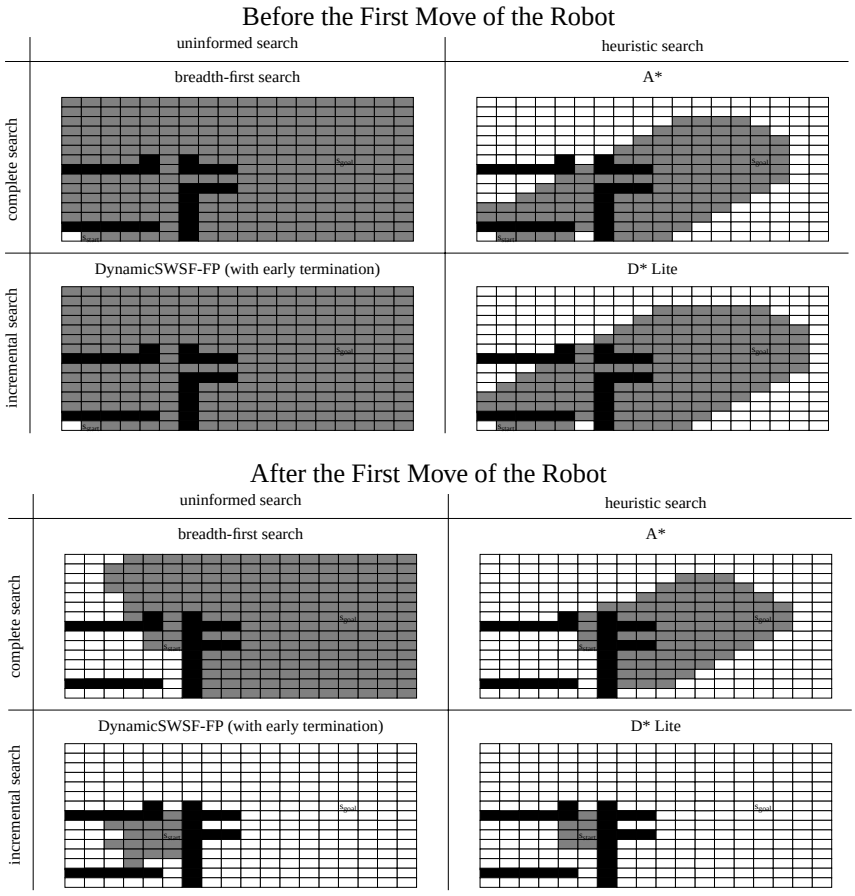

Figure 7: Simple Example (Part 2).

A* and thus D* Lite and A* expand not exactly the same cells. The second search of D* Lite expands only a subset of those cells whose goal distances changed (or had not been calculated before). Thus, D* Lite results in substantial savings over an A* search that replicates most of its previous search.

6 Experimental Results

We now compare focussed D* and the optimized final version of D* Lite. Since both methods move the robot in the same way and focussed D* has already been demonstrated with great success on real robots, we only need to perform a simulation study here. We need to compare the total planning time of the two methods until the robot reaches the goal vertex or recognizes that this is impossible. Since the actual runtimes are implementation-dependent, we instead use three measures that all correspond to common operations performed by the algorithms and thus heavily influence their runtimes: 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).

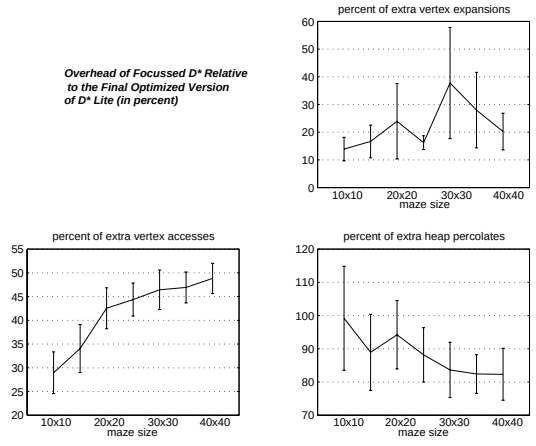

Figure 8: Performance Results for Different Maze Sizes.

We perform experiments with goal-directed navigation tasks in unknown terrain that is modeled as an eight-connected grid. As approximations of the distance between

two vertices we use again the maximum of the absolute differences of their x and y coordinates. Figure 8 shows results for grids of varying sizes, averaging over 50 randomly generated grids of each size, where the robot can always observe which of its eight adjacent cells are traversable. It shows the three measures for the two algorithms as well as the corresponding 95 percent confidence intervals to demonstrate that our conclusions are statistically significant. D* Lite performs better than D* with respect to all three measures, justifying our claim that it is more efficient than D*. We do not present results here that compare D* Lite to repeated A* searches since D* Lite is more efficient than D*, and D* has already been shown to outperform repeated A* searches by one to two orders of magnitude [14].

7 Conclusions

In this paper, 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 algorithms 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 also shorter, simpler, and consequently easier to understand and extend than D*, yet is more efficient. We believe that our results will make D*-like replanning algorithms 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.

References

[1] 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.

[2] 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.

[3] 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.

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

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

[6] Y. Huiming, C. Chia-Jung, S. Tong, and B. Qiang. Hybrid evolutionary motion planning using follow boundary repair for mobile robots. *Journal of Systems Architecture*, 47(7):635–647, 2001.

[7] S. Koenig and M. Likhachev. Incremental A*. In *Proceedings of the Neural Information Processing Systems*, 2001.

[8] M. Likhachev and S. Koenig. Lifelong Planning A* and Dynamic A* Lite: The proofs. Technical report, College of Computing, Georgia Institute of Technology, Atlanta (Georgia), 2001.

[9] 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.

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

[11] 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.

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

[13] 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.

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

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

[16] 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.

[17] 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.

[18] 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.