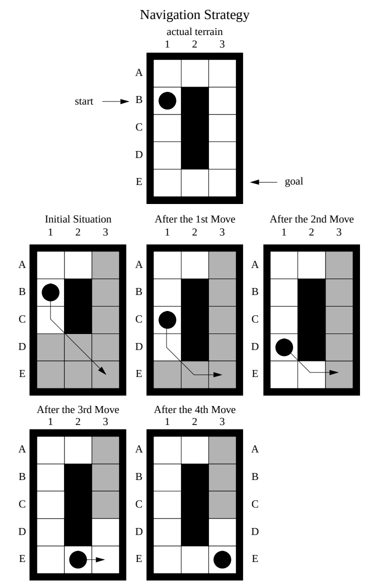
Fast Replanning for Navigation in Unknown Terrain

Sven Koenig, *Member, IEEE,* Maxim Likhachev

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* (D*) 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 faster than planning from scratch since it modifies its previous search results locally. Consequently, it has been extensively used in mobile robotics. In this article, we introduce an alternative to D* that determines the same paths and thus moves the robot in the same way but is algorithmically different. D* Lite is simple, can be rigorously analyzed, extendible in multiple ways, and is at least as efficient as 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.**

Index Terms— **Sensor-Based Path Planning, Replanning, Navigation in Unknown Terrain, D* (Dynamic A*), A*, Search, Planning with the Freespace Assumption**



(Grey cells are cells with unknown blockage status.)

Fig. 1. Illustration of the Navigation Strategy.

I. 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 [1]. 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 [2] [3]. If we model the navigation problem as a graph-traversal 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 but this is inefficient since most edge costs do not change between replanning episodes [4]. The most popular solution to this problem is Focussed Dynamic A* (D*) [5] since it combines the efficiency of heuristic and incremental searches, yet – different from real-time heuristic search methods [6] – still finds shortest paths. It achieves a large speedup over repeated A* [7] searches by modifying previous search results locally. D* has been extensively used on real robots. This includes indoor Nomad robots [8] as well as outdoor HMMWVs and the UGV Demo II vehicles as part of the DARPA Unmanned Ground Vehicle program [9]. It is currently also being integrated into Mars Rover prototypes and tactical mobile robot prototypes for urban reconnaissance [10] [11] [12]. D* is also used as part of other software, including the GRAMMPS mission planner for multiple robots [13]. Finally, versions of D* have been used to implement greedy mapping [14], a particular mapping method [8], and the parti-game method [15], a particular reinforcement-learning method for control [16].

However, D* is 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* (LPA*) method [17], we therefore present D* Lite, a novel replanning method that determines the same paths as D* and thus moves the robot in the same way but is algorithmically different. LPA* is an incremental version of A* that has well-understood properties. For example, we can prove theorems about its similarity to A* and its efficiency. These properties allow us to extend it easily, for example, to use inadmissible heuristics and different tie-breaking criteria to gain efficiency. We apply LPA* to a new domain in this paper, namely to goal-directed

Manuscript received January 20, 2002; revised November 18, 2002.

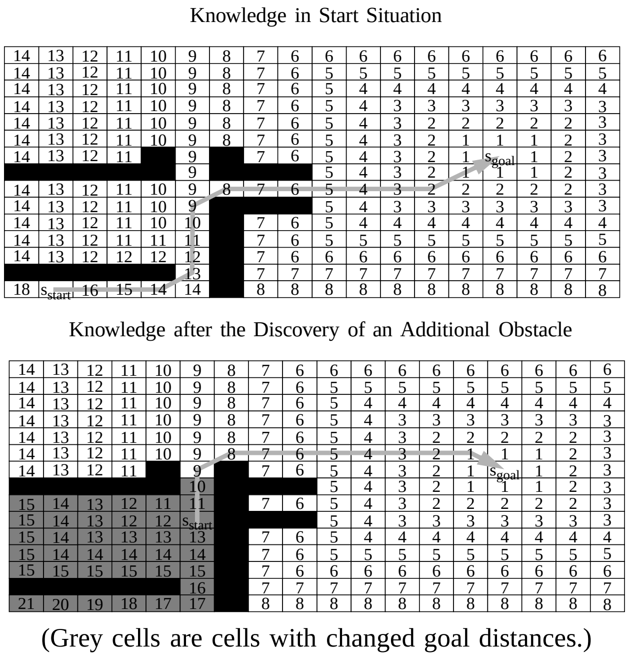

Knowledge in Start Situation
Knowledge after the Discovery of an Additional Obstacle
s_{start}
s_{goal}
(Grey cells are cells with changed goal distances.)

Fig. 2. Simple Example (Part 1).

navigation in unknown terrain. To this end, we introduce D* Lite. D* Lite extends LPA* to the case where the goal of the search changes between replanning episodes. Since D* Lite is based on LPA*, it inherits all of the properties of LPA* and can be extended in the same way as LPA*. It is shorter than D*, 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 at least as efficient as D*. 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 II, we motivate the ideas behind D* Lite. In Section III, we introduce LPA* and describe how it works. In Section IV, we use LPA* to develop two versions of D* Lite and describe how they can be optimized. In Section V, we illustrate the operation of the two versions of D* Lite with an example. In Section VI, we present experimental results that compare D* Lite against several other search methods.

II. 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 either reaches the goal cell, in which case it stops successfully, or 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 grey cells have unknown traversability. The robot starts in cell B1. Since all costs are one, the shortest path from its 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 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. The goal distances of grey cells have changed. The goal distances are important because one can easily determine a shortest path from the current cell of the robot to the goal cell 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 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.

III. Lifelong Planning A*

Lifelong Planning A* (LPA*) [17] generalizes both A* [7] and a version of DynamicSWSF-FP [18]. It is an incremental heuristic search method that repeatedly determines shortest paths between two given vertices as the edge costs of a graph change. An incremental search tends to recalculate only those start distances (that is, distances from the start vertex to a vertex) that have changed or have not been calculated before [19], and a heuristic search tends to recalculate only those start distances that are relevant for recalculating a shortest path from the start vertex to the goal vertex [7]. LPA* thus recalculates only very few start distances. In the following, we describe LPA* briefly. A more detailed and formal description can be found in [20].

A. Lifelong Planning A*: Notation

We use the following notation to describe LPA*: 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* 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.

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.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 CalcKey($s$)
{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($s_{start}$, CalcKey($s_{start}$));
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($u$, CalcKey($u$));
procedure ComputeShortestPath()
{09} while (U.TopKey() $\dot{<}$ CalcKey($s_{goal}$) 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($v$);
```

Fig. 3. Lifelong Planning A*.

B. Lifelong Planning A*: Local Consistency

LPA* maintains two kinds of estimates of the start distance $g^*(s)$ of each vertex s: a g-value $g(s)$ and an rhs-value (that is, right-hand side value, a term borrowed from [18]) $rhs(s)$. The rhs-value of a vertex is based on the g-values of its predecessors and thus potentially better informed than them. It always satisfies the following relationship (Invariant 1) [21]:

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

A vertex s is called locally consistent if $g(s) = rhs(s)$, otherwise it is called locally inconsistent. If all vertices are locally consistent then all of their g-values are equal to their respective start distances, which allows one to find shortest paths from the start vertex to any vertex. 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 from the start to the goal vertex. $h(s, s')$ approximates the distance between vertex s and s'. The heuristics need to be nonnegative and (forward) consistent [7], 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* computes these heuristics as needed. It only needs the heuristics $h(s, s_{goal})$.

C. Lifelong Planning A*: The Priority Queue

LPA* maintains a priority queue that always contains exactly the locally inconsistent vertices (Invariant 2) [21]. These are the vertices whose g-values LPA* potentially needs to change to make them locally consistent. The key $k(s)$ of vertex s in the priority queue is a vector with two components: $k(s) = [k_1(s); k_2(s)]$, where $k_1(s) = \min(g(s), rhs(s)) + h(s, s_{goal})$ and $k_2(s) = \min(g(s), rhs(s))$ {1} (Invariant 3). (Numbers in curly brackets refer to line numbers in the pseudo code.) Keys are compared according to a lexicographic ordering.

D. Lifelong Planning A*: The Method

LPA* is shown in Figure 3. Its 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 1 {03-04}. Thus, initially the start vertex is the only locally inconsistent vertex and is inserted into the otherwise empty priority queue {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.

After calling Initialize(), Main() calls ComputeShortestPath() to find a shortest path from the start to the goal vertex. ComputeShortestPath() repeatedly recalculates the g-values of locally inconsistent vertices ("expands the vertices") in nondecreasing order of their keys {10-16}. 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 holds that $rhs(s) = g^*(s)$, which implies that $k(s) = [f(s); g^*(s)]$, where $f(s) = g^*(s) + h(s, s_{goal})$. During the expansion of the vertex, ComputeShortestPath() sets the g-value of the vertex 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 [21]. 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 the rhs-values of these vertices, checks their local consistency, and adds them to or removes them from the priority queue as needed {06-08}. LPA* expands vertices until the goal vertex is locally consistent and the key of the vertex to expand next is no smaller than the key of the goal vertex. If $g(s_{goal}) = \infty$ after the search, then there is no finite-cost path from the start to the goal vertex. Otherwise, one can find a shortest path from the start to the goal vertex as follows: One always moves from the current vertex s, starting at the goal vertex, to any predecessor s' that minimizes $g(s') + c(s', s)$ until the start vertex is reached (ties can be broken arbitrarily) [21]. This way, one traverses a shortest path from the start to the goal vertex backward.

After calling ComputeShortestPath(), Main() 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() again, and iterates.

IV. 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. Its first search is identical to an A* search but subsequent searches reuse information from previous searches. We now use LPA* to develop D* Lite [22], 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 knowledge of the robot changes. 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.

A. The First 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. The version presented in Figure 3 searches from the start vertex to the goal vertex. Its g-values are estimates of the start distances. We thus need to switch the search direction of LPA* so that the g-values are estimates of the goal distances. Such a version of LPA* searches from the goal vertex to the start vertex and can be derived by reversing all edges of the original graph and exchanging its 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 $h(s_{start}, s) \leq h(s_{start}, s') + c(s', s)$ for all vertices $s \in S$ and $s' \in \text{Pred}(s)$. More generally, since the robot moves and thus changes the start vertex, 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 the start to the goal vertex. Otherwise, one can follow a shortest path from the start to the goal vertex by always moving from the current vertex s, starting at the start vertex, to any successor s' that minimizes $c(s, s') + g(s')$ until the goal vertex is reached (ties can be broken arbitrarily). To solve the goal-directed navigation problem in unknown terrain, the CalcKey(), 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 keys 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 keys of the vertices in the priority queue but not which vertices are locally consistent and thus in the priority queue. Figure 4 shows the resulting search method, called the first version of D* Lite.

```
procedure CalcKey($s$)
{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($s_{goal}$, CalcKey($s_{goal}$));
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($u$, CalcKey($u$));
procedure ComputeShortestPath()
{09'} while (U.TopKey() $\dot{<}$ CalcKey($s_{start}$) 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($s$, CalcKey($s$));
{30'}         ComputeShortestPath();
```

Fig. 4. D* Lite: First Version.

The main function Main() of the first version of D* Lite 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 1 {03'-04'}. Thus, initially the goal vertex is the only locally inconsistent vertex and is inserted into the otherwise empty priority queue with a key calculated according to the equivalent of the formula given earlier {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 first 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.

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

Fig. 5. D* Lite: Second Version.

For the goal-directed navigation problem in unknown static 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 keys of all vertices in the priority queue {28’-29’}, recalculates a shortest path {30’}, and iterates. We can prove the following theorem:

Theorem 1: ComputeShortestPath() of the first 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 the start to the goal vertex by always moving from the current vertex s, starting at the start vertex, to any successor s' that minimizes $c(s, s') + g(s')$ until the goal vertex is reached (ties can be broken arbitrarily).

The proofs of all theorems and other properties mentioned in the text are given in [21].

B. The Second Version of D* Lite

The first 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 second version of D* Lite, shown in Figure 5, uses a search method derived from D* [5] to avoid having to reorder the priority queue. Differences to the first 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 $s' \in S$. Heuristics with these properties also satisfy the property that heuristics need to satisfy for the first version of D* Lite [21]. Yet, they are not overly restrictive since they are guaranteed to hold if the heuristics were derived by relaxing the search problem [21], which will almost always be the case and holds for the heuristics used in this article.

The second version of D* Lite uses keys that are lower bounds on the keys that the first version of D* Lite uses for the corresponding vertices. It initializes them in the same way as the first version of D* Lite. After the robot has moved from vertex s to some vertex s' where it detects changes in edge costs, the first component of the keys 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 component of the keys 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 keys are computed, their first components are by $h(s, s')$ too small relative to the keys in the priority queue. Thus, $h(s, s')$ has to be added to their first components. 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 (that is, key modifier) {30”}. Thus, whenever new keys 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 keys, on the other hand, are always lower bounds on the corresponding keys of the first version of D* Lite after the first component of the keys of the first version of D* Lite has been increased by the current value of k_m, that is, lower bounds on the values calculated by CalcKey() {01”}. We exploit this property by changing ComputeShortestPath() as follows. After ComputeShortestPath() has removed a vertex u with the smallest key k_{old} = U.TopKey() from the priority queue {12”}, it now uses CalcKey() to compute the key that it should have had. If $k_{old}\dot{<}$CalcKey(u) then it reinserts the removed vertex with the key calculated by CalcKey() into the priority queue {13”-14”}. Thus, it remains true that the keys of all vertices in the priority queue are lower bounds on the corresponding keys of the first version of D* Lite after the first components of the keys of the first version of D* Lite have been increased by the current value of k_m. If $k_{old}\dot{\geq}$CalcKey(u), then it holds that $k_{old}\dot{=}$CalcKey(u) since k_{old} was a lower bound on the value returned by CalcKey(). In this case, ComputeShortestPath() performs the

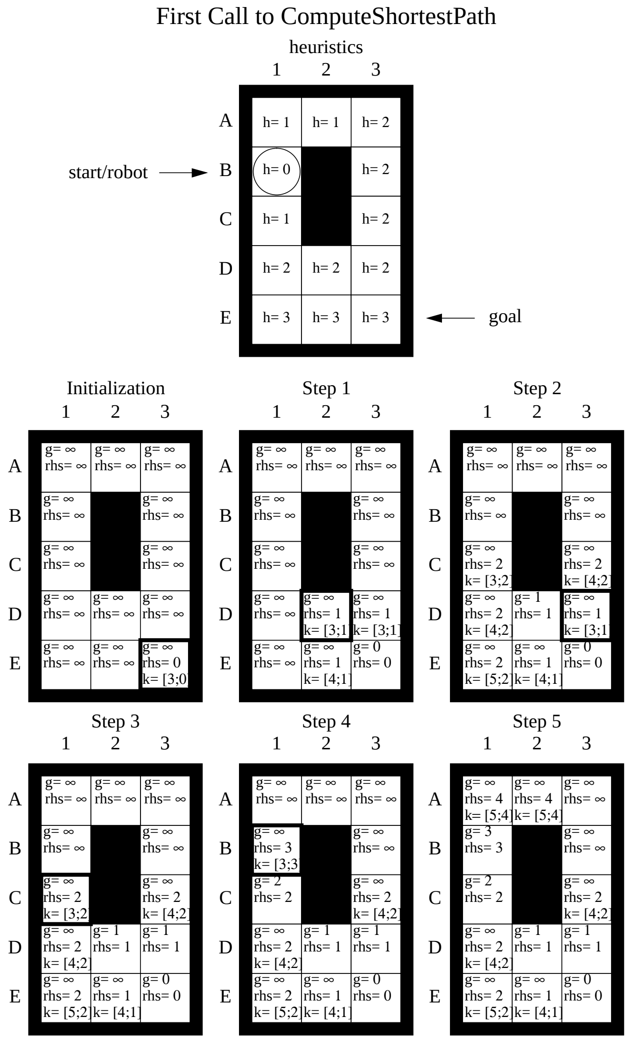


Fig. 6. Operation of the Second Version of D* Lite (Part 1).

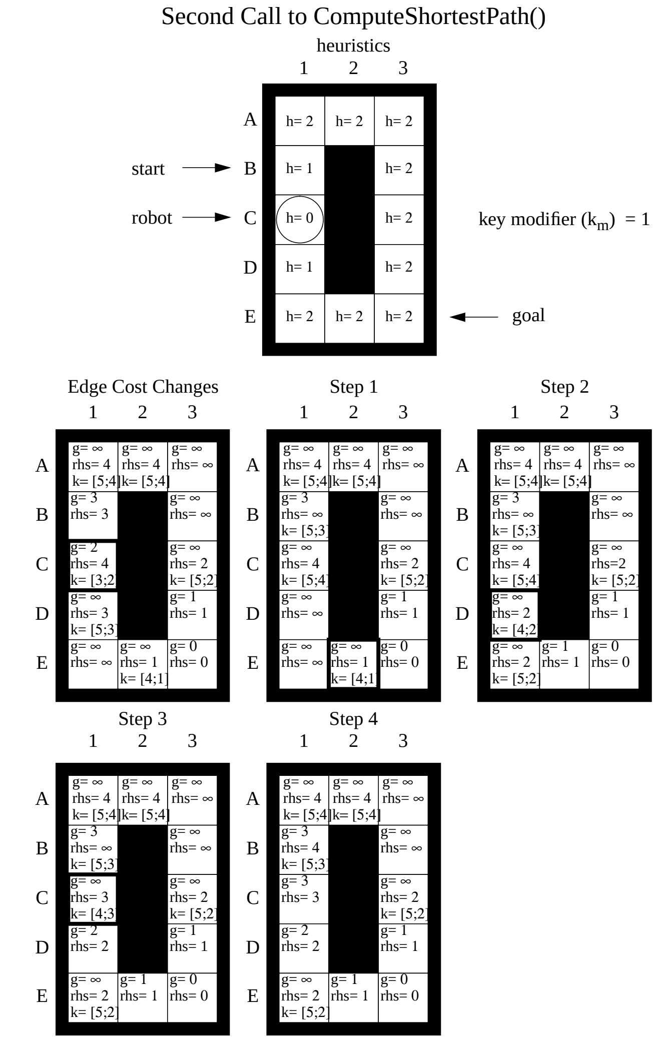


Fig. 7. Operation of the Second Version of D* Lite (Part 2).

same operations for vertex u as ComputeShortestPath() of the first version of D* Lite $\{$15"-20"$\}$. ComputeShortestPath() performs these operations for vertices in the exact same order as ComputeShortestPath() of the first version of D* Lite, which implies that the second version of D* Lite shares many properties with the first version of D* Lite, including its correctness. Formally, we can prove the following theorem:

Theorem 2: ComputeShortestPath() of the second 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 the start to the goal vertex by always moving from the current vertex s, starting at the start vertex, to any successor s' that minimizes $c(s, s') + g(s')$ until the goal vertex is reached (ties can be broken arbitrarily).

V. An Example

We now step through the example from Figure 1 to show the operation of the second version of D* Lite. Figure 6 (top) shows the untraversable cells that the robot knows about initially. It also shows the heuristics of the traversable cells, which approximate the distance from the start cell to a cell with the maximum of the absolute differences of the x and y coordinates of both cells. Figure 6 (bottom) shows the g-values and rhs-values of the traversable cells and, for locally inconsistent cells, also their keys. 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 grid labeled "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 g-values but not all of them. The last grid shows the values after ComputeShortestPath() returns.

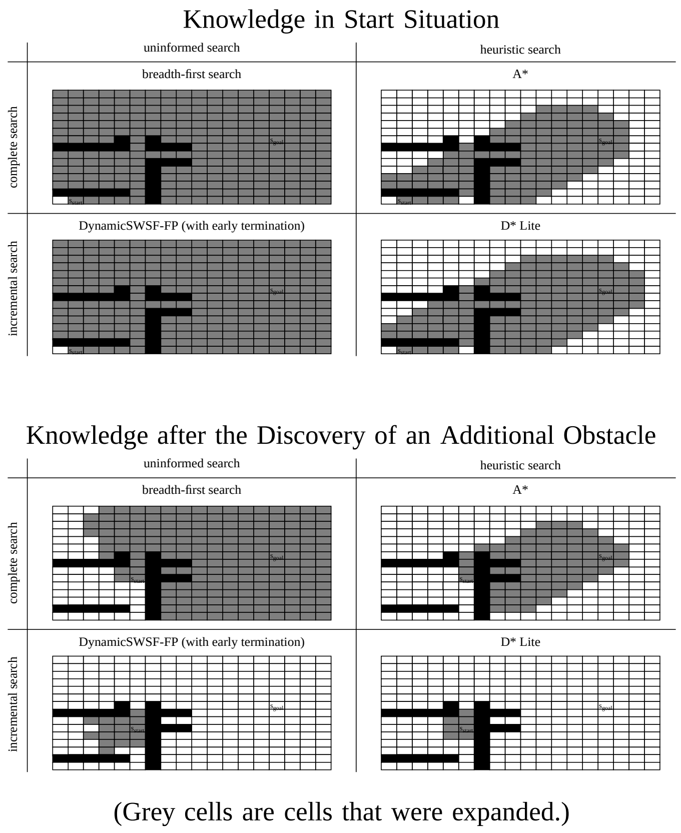

Knowledge in Start Situation
uninformed search
heuristic search
complete search
incremental search
breadth-first search
A*
DynamicSWSF-FP (with early termination)
D* Lite
Knowledge after the Discovery of an Additional Obstacle
uninformed search
heuristic search
complete search
incremental search
breadth-first search
A*
DynamicSWSF-FP (with early termination)
D* Lite
(Grey cells are cells that were expanded.)

Fig. 8. Simple Example (Part 2).

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 7 (top) shows the grid as it is now perceived by the robot. The figure also shows the new heuristics of the traversable cells. The grid labeled "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().

We now use the two eight-connected grids from Figure 2 to compare the second version of D* Lite against several alternatives: complete (that is, nonincremental) uninformed search (breadth-first search), complete heuristic search (A*), and incremental uninformed search (DynamicSWSF-FP, modified to terminate immediately after it has identified a shortest path from the current cell of the robot to the goal cell). To make the search methods comparable, their search always starts at the goal cell 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 a heuristic estimate of their distance. Cells expanded by the methods are shaded grey in Figure 8. In our example, we broke ties in the most advantageous way for A* and thus D* Lite and A* expand not exactly the same cells during the first search. The second search of D* Lite expands only a subset of those cells whose goal distances changed or have not been calculated before. Thus, it is more efficient than an A* search that replicates most of its previous search. More generally, the heuristic searches outperform the uninformed searches during the first and second searches, the incremental searches outperform the complete ones during the second search (where previous search results are available), and the incremental heuristic search decreases the number of expanded cells even more than either a heuristic search or an incremental search individually.

VI. Experimental Results

There are several ways of optimizing both versions of D* Lite without changing their overall operation. These optimizations are similar to the optimizations for LPA* [20]. Figure 9 shows the optimized second version of D* Lite, which we compared against several alternatives: complete uninformed search (breadth-first search), complete heuristic search (A* [7]), incremental uninformed search (DynamicSWSF-FP [18] with early termination, as described earlier), and a different kind of incremental heuristic search (D* [5]). We implemented all priority queues using standard binary heaps, although using more complex data structures (such as Fibonacci heaps) could possibly make U.Update() more efficient. A* broke ties among cells with the same f-value in favor of cells with larger g-values, which tends to be more efficient than the opposite way of breaking ties. In previous work, D* was compared to a version of A* that, like D*, searched from the goal vertex to the current vertex of the robot [5]. It is equally easy, however, to implement a version of A* that searches in the opposite direction. In fact, such forward A* searches can be noticeably faster than backward A* searches, for the following reason: In practice, changes in edge costs are caused by untraversable cells discovered by the robot and thus occur close to the current vertex of the robot. Thus, if the robot has no initial knowledge about which cells are traversable and assumes that all of them are, then its assumption is correct around the goal vertex during the initial searches but not around its current vertex. Thus, the heuristics of most vertices around the goal vertex are totally informed for a forward A* search, which then expands only the vertices on a shortest path to the goal vertex in this region. On the other hand, the heuristics of a backward A* search are not as informed as the heuristics of a forward A* search, and a backward A* search can therefore be expected to expand more vertices than a forward A* search during the initial searches. We therefore present experimental results for both forward and backward A* searches.

Since all search methods studied in this article move the robot in the same way and D* has already been demonstrated

```
procedure CalcKey($s$)
{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($u, CalcKey(u)$);
{08'''} else if ($g(u) \neq rhs(u)$ AND $u \notin U$) U.Insert($u, CalcKey(u)$);
{09'''} else if ($g(u) = rhs(u)$ AND $u \in U$) U.Remove($u$);
procedure ComputeShortestPath()
{10'''} while (U.TopKey() $\dot{<}$ CalcKey($s_{start}$) OR $rhs(s_{start}) > g(s_{start})$)
{11'''}     $u$ = U.Top();
{12'''}     $k_{old}$ = U.TopKey();
{13'''}     $k_{new}$ = CalcKey($u$));
{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'''}             $rhs(s) = \min(rhs(s), c(s, u) + g(u))$;
{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'''}                 $rhs(u) = \min(rhs(u), c(u, v) + g(v))$;
{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();
```

Fig. 9. D* Lite: Second Version (optimized version).

with great success on real robots [9], we only need to perform a simulation study in which we compare the total planning times of the search methods. Since the actual planning times are implementation and machine dependent, we also use two measures that both correspond to common operations performed by the search methods and thus heavily influence their planning times, yet are implementation and machine independent: the total number of cell expansions and the total number of heap percolates (exchanges of a parent and child in the heap). We performed experiments on eight-connected grids of size 129×129, where the start cell of the robot was $(x = 12, y = 12)$ and the goal cell was $(x = 116, y = 116)$. We report the averages over 500 runs on randomly generated grids. All experiments were run on a 1.9 GHz PC under Linux.

In one set of experiments, we used grids whose cells were either traversable or, with forty percent probability, untraversable, where untraversable cells were modeled as cells with no incoming or outgoing edges. The robot initially

TABLE I

EXPERIMENTAL RESULTS – TERRAIN WITH RANDOM OBSTACLES.

Search Algorithm	Planning Time	Cell Expansions	Heap Percolates
Breadth-First Search	302.30 msecs	845,433	4,116,516
Backward A*	10.55 msecs	17,096	276,287
Forward A*	7.29 msecs	8,722	177,476
DynamicSWSF-FP	6.41 msecs	13,962	75,738
(Focussed) D*	4.28 msecs	2,138	79,214
D* Lite	2.82 msecs	2,856	32,988

TABLE II

EXPERIMENTAL RESULTS – FRACTAL TERRAIN.

Search Algorithm	Planning Time	Cell Expansions	Heap Percolates
Breadth-First Search	194.13 msecs	543,408	2,643,916
Backward A*	5.49 msecs	8,680	156,801
Forward A*	4.78 msecs	5,459	124,814
DynamicSWSF-FP	6.26 msecs	13,931	76,703
(Focussed) D*	1.18 msecs	596	19,066
D* Lite	0.97 msecs	393	5,316

assumed that all edges were present with cost one. The robot always deleted those edges entering its neighboring cells that did not exist in the true model and then replanned a shortest path from its current cell to the goal cell. We used the maximum of the absolute differences of the x and y coordinates of any two cells as a heuristic estimate of their distances. Table I compares D* Lite against the other search methods. It shows that incremental heuristic searches outperform both incremental and heuristic searches individually, and that D* Lite is competitive with D*. It is interesting to note that incremental heuristic search is faster than complete search even though each cell expansion takes longer, which confirms earlier experimental results [5].

In a second set of experiments, we used fractal terrain, similar to the one used in [23]. All edges existed but their cost varied from 5 to 14 according to traversal difficulty of the cell they were entering. The robot initially assumed that all edges were present with cost 5. The robot always updated the cost of the edges entering its neighboring cells to correspond to the traversal difficulty of the corresponding neighboring cell and then replanned a shortest path from its current cell to the goal cell. We used five times of the maximum of the absolute differences of the x and y coordinates of any two cells as a heuristic estimate of their distances. Table II presents the same comparison as in the previous experiment. The conclusions of the previous experiment continue to hold in these grids, that might be more realistic models of outdoor terrain.

VII. RELATED WORK

Path planning for goal-directed navigation in known terrain has been studied extensively [24]. Path planning for robot navigation in unknown terrain has been studied less frequently. We are interested in those navigation methods that repeatedly determine a shortest path from the current robot coordinates to the goal coordinates while the robot moves along the path. Most navigation methods do not fit this description, including the bug algorithms [1]. Those navigation 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 [25]. Other approaches are based on graph search problems solved with either massively parallel search methods [26] 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 [27]. Examples include [28], [29], [30], [31], [32], [33], [34], [35], [36], [37], [38], and [39]. 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 [40]. 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 [19], [41], [42]. An example of an incremental search method for fully dynamic shortest path problems is DynamicSWSF-FP [18], a variant of which has been applied to hierarchical motion planning [43]. [44] demonstrates experimentally the benefits of incremental searches over complete ones for repetitive planning tasks. Incremental search methods are typically uninformed, however, including all incremental search methods cited so far.

There exist only very few incremental heuristic search methods that have been applied to navigation in unknown terrain, that is, take into account that the start vertex changes because the robot moves in the terrain. Some of these search methods first identify the perimeter of areas in which the previous movement decisions need to get updated and restart the search from there [45] [46]. Other search methods discover these areas while updating previous movement decisions. To the best of our knowledge, the only search methods that fit this description are (Focussed) D* [4] and the first and second versions of D* Lite.

It is important to understand that D* and the second version of D* Lite share some similarities but work differently. Both search methods search from the destination of the robot to its current vertex. Both search methods use heuristics to focus their search and use the same mechanism to take into account that the heuristics change when the robot moves and the goal of the search thus changes. Both search methods propagate cost changes in two waves: D* uses RAISE and LOWER vertices, whereas D* Lite uses locally underconsistent and overconsistent vertices for similar purposes. Both search methods stop as soon as the smallest key of all vertices in the priority queue is no longer smaller than the key of the current vertex of the robot. Finally, both search methods are often more efficient than A* because the edge costs change around the current vertex of the robot and thus close to the goal of the search, a consequence of the property that sensors on-board of robots sense the terrain in the neighborhood of the robots. However, the two search methods work differently despite these similarities. For example, D* can expand a vertex more than twice during the same replanning episode whereas D* Lite guarantees that every vertex is expanded at most twice. This explains why the two search methods differ in their number of vertex expansions and heap percolates in the exact same terrain.

VIII. Conclusions

In this article, we have presented D* Lite, a novel fast replanning method for goal-directed navigation in unknown terrain that determines the same paths as (Focussed) D* and thus moves the robot in the same way. Both search 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 is algorithmically different from D*. It 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. We showed that D* Lite can be rigorously analyzed and experimentally appears to be even slightly 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 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 offi cial policies, either expressed or implied, of the sponsoring organizations and agencies or the U.S. government.

References

[1] V. Lumelsky and A. Stepanov, "Path planning strategies for point automaton moving amidst unknown obstacles of arbitrary shape," *Algorithmica*, vol. 2, pp. 403–430, 1987.
[2] H. Choset and J. Burdick, "Sensor based planning and nonsmooth analysis," in *Proceedings of the International Conference on Robotics and Automation*, 1994, pp. 3034–3041.
[3] ——, "Sensor-based planning, Part I: Incremental construction of the generalized Voronoi graph," in *Proceedings of the International Conference on Robotics and Automation*, 1995, pp. 1643–1649.
[4] A. Stentz, "Optimal and effi cient path planning for partially-known environments," in *Proceedings of the International Conference on Robotics and Automation*, 1994, pp. 3310–3317.
[5] ——, "The focussed D* algorithm for real-time replanning," in *Proceedings of the International Joint Conference on Artificial Intelligence*, 1995, pp. 1652–1659.
[6] R. Korf, "Real-time heuristic search," *Artificial Intelligence*, vol. 42, no. 2-3, pp. 189–211, 1990.
[7] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1985.
[8] S. Koenig, C. Tovey, and W. Halliburton, "Greedy mapping of terrain," in *Proceedings of the International Conference on Robotics and Automation*, 2001, pp. 3594–3599.
[9] A. Stentz and M. Hebert, "A complete navigation system for goal acquisition in unknown environments," *Autonomous Robots*, vol. 2, no. 2, pp. 127–145, 1995.
[10] M. Hebert, R. McLachlan, and P. Chang, "Experiments with driving modes for urban robots," in *Proceedings of the SPIE Mobile Robots*, 1999.

[11] 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.

[12] 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*, vol. 4195, 2000.

[13] 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.

[14] M. Likhachev and S. Koenig, "Incremental replanning for mapping," in *Proceedings of the International Conference on Intelligent Robots and Systems*, 2002.

[15] ——, "Speeding up the parti-game algorithm," in *Advances in Neural Information Processing Systems 15*, S. Becker, S. Thrun, and K. Obermayer, Eds. MIT Press, 2002.

[16] A. Moore and C. Atkeson, "The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces," *Machine Learning*, vol. 21, no. 3, pp. 199–233, 1995.

[17] S. Koenig and M. Likhachev, "Incremental A*," in *Advances in Neural Information Processing Systems 14*, T. Dietterich, S. Becker, and Z. Ghahramani, Eds. MIT Press, 2002.

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

[19] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni, "Fully dynamic algorithms for maintaining shortest paths trees," *Journal of Algorithms*, vol. 34, no. 2, pp. 251–281, 2000.

[20] S. Koenig, M. Likhachev, and D. Furcy, "Lifelong planning A*," *Artificial Intelligence Journal*, vol. 155, pp. 93–146, 2004.

[21] S. Koenig and M. Likhachev, "Improved fast replanning for robot navigation in unknown terrain," College of Computing, Georgia Institute of Technology, Atlanta (Georgia), Tech. Rep. GIT-COGSCI-2002/3, 2001.

[22] ——, "Improved fast replanning for robot navigation in unknown terrain," in *Proceedings of the International Conference on Robotics and Automation*, 2002, pp. 968–975.

[23] A. Stentz, "Map-based strategies for robot navigation in unknown environments," in *Proceedings of the AAAI Spring Symposium on Planning with Incomplete Information for Robot Problems*, 1996, pp. 110–116.

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

[25] 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, pp. 858–864.

[26] 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*, 1997, pp. 2727–2732.

[27] G. Ramalingam and T. Reps, "On the computational complexity of dynamic graph problems," *Theoretical Computer Science*, vol. 158, no. 1–2, pp. 233–277, 1996.

[28] G. Ausiello, G. Italiano, A. Marchetti-Spaccamela, and U. Nanni, "Incremental algorithms for minimal length paths," *Journal of Algorithms*, vol. 12, no. 4, pp. 615–638, 1991.

[29] S. Even and Y. Shiloach, "An on-line edge deletion problem," *Journal of the ACM*, vol. 28, no. 1, pp. 1–4, 1981.

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

[31] E. Feuerstein and A. Marchetti-Spaccamela, "Dynamic algorithms for shortest paths in planar graphs," *Theoretical Computer Science*, vol. 116, no. 2, pp. 359–371, 1993.

[32] P. Franciosa, D. Frigioni, and R. Giaccio, "Semi-dynamic breadth-first search in digraphs," *Theoretical Computer Science*, vol. 250, no. 1–2, pp. 201–217, 2001.

[33] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni, "Fully dynamic output bounded single source shortest path problem," in *Proceedings of the Symposium on Discrete Algorithms*, 1996, pp. 212–221.

[34] S. Goto and A. Sangiovanni-Vincentelli, "A new shortest path updating algorithm," *Networks*, vol. 8, no. 4, pp. 341–372, 1978.

[35] G. Italiano, "Finding paths and deleting edges in directed acyclic graphs," *Information Processing Letters*, vol. 28, no. 1, pp. 5–11, 1988.

[36] 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*, 1993, pp. 443–451.

[37] C. Lin and R. Chang, "On the dynamic shortest path problem," *Journal of Information Processing*, vol. 13, no. 4, pp. 470–476, 1990.

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

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

[40] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni, "Semidynamic algorithms for maintaining single source shortest path trees," *Algorithmica*, vol. 22, no. 3, pp. 250–274, 1998.

[41] V. King, "Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs," in *Proceedings of the Symposium on Foundations of Computer Science*, 1999, pp. 81–89.

[42] C. Demetrescu and G. Italiano, "Fully dynamic all pairs shortest paths with real edge weights," in *Proceedings of the Symposium on Foundations of Computer Science*, 2001.

[43] 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*, vol. 11, no. 2, pp. 198–214, 1995.

[44] D. Frigioni, M. Ioffreda, U. Nanni, and G. Pasqualone, "Experimental analysis of dynamic algorithms for the single source shortest path problem," *ACM Journal on Experimental Algorithmics*, vol. 3, no. 5, 1998.

[45] 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*, vol. 4, no. 2, pp. 245–268, 1990.

[46] 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*, vol. 34, pp. 145–152, 2001.

PLACE PHOTO HERE

Sven Koenig Sven Koenig is an associate professor in computer science at the University of Southern California. He researches techniques for decision making that enable robots to act intelligently in their environments and exhibit goal-directed behavior in real-time, even if they have only incomplete knowledge of their environment, limited or noisy perception, imperfect abilities to manipulate it, or insufficient reasoning speed. Sven has published over 70 papers in various areas of robotics and artificial intelligence, including papers at many ICRA and IROS conferences. He is the recipient of an NSF CAREER award, an IBM Faculty Partnership Award, a Raytheon Faculty Fellowship Award, the Tong Leong Lim Pre-Doctoral Prize from the University of California at Berkeley, and a Fulbright Fellowship. Several of his students have won awards for their research.

PLACE PHOTO HERE

Maxim Likhachev Maxim Likhachev is a Ph.D. student in the department of Computer Science at Carnegie Mellon University. He is interested in planning with incomplete information for mobile robots. In particular, Maxim develops efficient planning methods for large, partially known and dynamic environments, including stochastic environments. He strives for planning methods that are easy to use and analyzes them theoretically.