D* Lite

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

Incremental heuristic search methods use heuristics to focus their search and reuse information from previous searches to find solutions to series of similar search tasks much faster than is possible by solving each search task from scratch. In this paper, we apply Lifelong Planning A* to robot navigation in unknown terrain, including goal-directed navigation in unknown terrain and mapping of unknown terrain. The resulting D* Lite algorithm is easy to understand and analyze. It implements the same behavior as Stentz' Focussed Dynamic A* but is algorithmically different. We prove properties about D* Lite and demonstrate experimentally the advantages of combining incremental and heuristic search for the applications studied. We believe that these results provide a strong foundation for further research on fast replanning methods in artificial intelligence and robotics.

Introduction

Incremental search methods, such as DynamicSWSF-FP (Ramalingam & Reps 1996), are currently not much used in artificial intelligence. They reuse information from previous searches to find solutions to series of similar search tasks much faster than is possible by solving each search task from scratch. An overview is given in (Frigioni, Marchetti-Spaccamela, & Nanni 2000). Heuristic search methods, such as A* (Nilsson 1971), on the other hand, use heuristic knowledge in form of approximations of the goal distances to focus the search and solve search problems much faster than uninformed search methods. An overview is given in (Pearl 1985). We recently introduced LPA* (Lifelong Planning A*), that generalizes both DynamicSWSF-FP and A* and thus uses two different techniques to reduce its planning time (Koenig & Likhachev 2001). In this paper, we apply LPA* to robot navigation in unknown terrain. The robot could use conventional graph-search methods when replanning its paths after discovering previously unknown obstacles. However, the resulting planning times can be on the order of minutes for the large terrains that are often used, which adds up to substantial idle times (Stentz 1994). Focussed Dynamic A* (D*) (Stentz 1995) is a clever heuristic search method that achieves a speedup of one to two orders of magnitudes(!) over repeated A* searches by modifying previous search results locally. D* has been extensively used on real robots, including outdoor HMMWVs (Stentz & Hebert 1995). It is currently also being integrated into Mars Rover prototypes and tactical mobile robot prototypes for urban reconnaissance (Matthies *et al.* 2000; Thayer *et al.* 2000). However, it has not been extended by other researchers. Building on LPA*, we therefore present D* Lite, a novel replanning method that implements the same navigation strategy as D* but is algorithmically different. D* Lite is substantially 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. These properties also allow one to extend it easily, for example, to use inadmissible heuristics and different tie-breaking criteria to gain efficiency. To gain insight into its behavior, we present various theoretical properties of LPA* that also apply to D* Lite. Our theoretical properties show that LPA* is efficient and similar to A*, a well known and well understood search algorithm. Our experimental properties show that D* Lite is at least as efficient as D*. We also present an experimental evaluation of the benefits of combining incremental and heuristic search across different navigation tasks in unknown terrain, including goal-directed navigation and mapping. We believe that our theoretical and empirical analysis of D* Lite will provide a strong foundation for further research on fast replanning methods in artificial intelligence and robotics.

Motivation

Consider a goal-directed 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 blockage status are traversable. It then follows this path until it reaches the goal cell, in which case it stops successfully, or it observes an untraversable cell, in which case it recomputes a shortest path from its current cell to the goal cell. Figure 1 shows the goal distances of all traversable cells and the shortest paths from its current cell to the goal cell both before and after the robot has moved

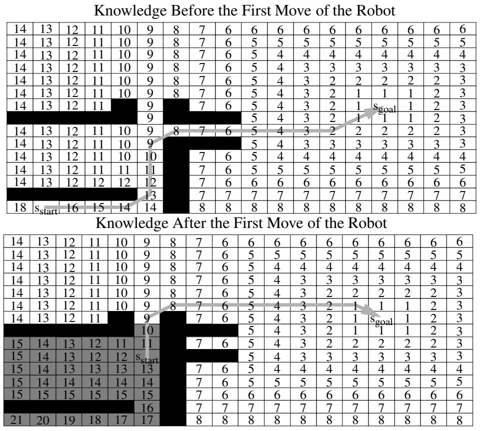

Figure 1: Simple Example.

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 its 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 number of cells with changed goal distances is small and most of the changed goal distances are irrelevant for recalculating a shortest path from its current cell to the goal cell. Thus, one can efficiently recalculate a shortest path from its current cell to the goal cell by recalculating only those goal distances that have changed (or have not been calculated before) *and* are relevant for recalculating the shortest path. This is what D* Lite does. The challenge is to identify these cells efficiently.

Lifelong Planning A*

Lifelong Planning A* (LPA*) is shown in Figure 2. LPA* is an incremental version of A*. It applies to finite graph search problems on known graphs whose edge costs increase or decrease over time (which can also be used to model edges or vertices that are added or deleted). S denotes the finite set of vertices of the graph. $Succ(s) \subseteq S$ denotes the set of successors of vertex $s \in S$. Similarly, $Pred(s) \subseteq S$ denotes the set of predecessors of vertex $s \in S$. $0 < c(s, s') \leq \infty$ 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. We use $g^*(s)$ to denote the start distance of vertex $s \in S$, that is, the length of a shortest path from s_{start} to s. Like A*, LPA* uses heuristics $h(s, s_{goal})$ that approximate the goal distances of the vertices s. The heuristics need to be nonnegative and consistent (Pearl 1985), 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 $s' \in Succ(s)$ with $s \neq s_{goal}$.

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 CalculateKey(s)
return [min(g(s), rhs(s)) + h(s, s_goal); min(g(s), rhs(s))];

procedure Initialize()
U = ∅;
for all s ∈ S rhs(s) = g(s) = ∞;
rhs(s_start) = 0;
U.Insert(s_start, CalculateKey(s_start));

procedure UpdateVertex(u)
if (u ≠ s_start) rhs(u) = min_{s' ∈ Pred(u)}(g(s') + c(s', u));
if (u ∈ U) U.Remove(u);
if (g(u) ≠ rhs(u)) U.Insert(u, CalculateKey(u));

procedure ComputeShortestPath()
while (U.TopKey() <˙ CalculateKey(s_goal) OR rhs(s_goal) ≠ g(s_goal))
  u = U.Pop();
  if (g(u) > rhs(u))
    g(u) = rhs(u);
    for all s ∈ Succ(u) UpdateVertex(s);
  else
    g(u) = ∞;
    for all s ∈ Succ(u) ∪ {u} UpdateVertex(s);

procedure Main()
Initialize();
forever
  ComputeShortestPath();
  Wait for changes in edge costs;
  for all directed edges (u, v) with changed edge costs
    Update the edge cost c(u, v);
    UpdateVertex(v);
```

Figure 2: Lifelong Planning A*.

Lifelong Planning A*: The Variables

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

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

A vertex is called locally consistent iff its g-value equals its rhs-value, otherwise it is called locally inconsistent. Iff all vertices are locally consistent, then the g-values of all vertices equal their respective start distances. In this case one can trace back a shortest path from s_{start} to any vertex u by always transitioning from the current vertex s, starting at u, to any predecessor s' that minimizes $g(s') + c(s', s)$ (ties can be broken arbitrarily) until s_{start} is reached. (This is different from Figure 1, where the goal distances instead of the start distances are used to determine a shortest path and one can follow a shortest path from s_{start} to s_{goal} by always moving from the current vertex s, starting at s_{start}, to any successor s' that minimizes $c(s, s') + g(s')$, until s_{goal} is reached.) However, LPA* does not make all vertices locally consistent after some edge costs have changed. Instead, it uses the heuristics to focus the search and updates only the g-values that are relevant for computing a shortest path. To this end, LPA* maintains a priority queue. The priority queue always contains exactly the locally inconsistent vertices *(Invariant 2)*. These are the vertices whose g-values LPA* potentially needs to update to make them locally consistent. The priority of a vertex in the priority queue is always the same as its key *(Invariant 3)*, which 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 $k_2(s) = \min(g(s), rhs(s))$ $\{01\}$ (numbers in brackets refer to line numbers in Figure 2). The first component of the keys $k_1(s)$ corresponds directly to the f-values $f(s) := g^*(s) + h(s, s_{goal})$ used by A* because both the g-values and rhs-values of LPA* correspond to the g-values of A* and the h-values of LPA* correspond to the h-values of A*. The second component of the keys $k_2(s)$ corresponds to the g-values of A*. Keys are compared according to a lexicographic ordering. For example, a key $k(s)$ is less than or equal to a key $k'(s)$, denoted by $k(s)\dot{\leq}k'(s)$, iff either $k_1(s) < k'_1(s)$ or ($k_1(s) = k'_1(s)$ and $k_2(s) \leq k'_2(s)$). LPA* always expands the vertex in the priority queue with the smallest key (by expanding a vertex, we mean executing $\{10\text{-}16\}$). This is similar to A* that always expands the vertex in the priority queue with the smallest f-value if it breaks ties towards the smallest g-value. The resulting behavior of LPA* and A* is also similar. The keys of the vertices expanded by LPA* are nondecreasing over time just like the f-values of the vertices expanded by A* (since the heuristics are consistent).

Lifelong Planning A*: The Algorithm

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

A locally inconsistent vertex s is called locally overconsistent iff $g(s) > rhs(s)$. When ComputeShortestPath() expands a locally overconsistent vertex $\{12\text{-}13\}$, then it sets the g-value of the vertex to its rhs-value $\{12\}$, which makes the vertex locally consistent. A locally inconsistent vertex s is called locally underconsistent iff $g(s) < rhs(s)$. When ComputeShortestPath() expands a locally underconsistent vertex $\{15\text{-}16\}$, then it simply sets the g-value of the vertex to infinity $\{15\}$. This makes the vertex either locally consistent or overconsistent. If the expanded vertex was locally overconsistent, then the change of its g-value can affect the local consistency of its successors $\{13\}$. Similarly, if the expanded vertex was locally underconsistent, then it and its successors can be affected $\{16\}$. To maintain Invariants 1-3, ComputeShortestPath() therefore updates rhs-values of these vertices, checks their local consistency, and adds them to or removes them from the priority queue accordingly $\{06\text{-}08\}$. ComputeShortestPath() expands vertices until s_{goal} is locally consistent and the key of the vertex to expand next is no less than the key of s_{goal}. This is similar to A* that expands vertices until it expands s_{goal} at which point in time the g-value of s_{goal} equals its start distance and the f-value of the vertex to expand next is no less than the f-value of s_{goal}. 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 transitioning from the current vertex s, starting at s_{goal}, to any predecessor s' that minimizes $g(s') + c(s', s)$ (ties can be broken arbitrarily) until s_{start} is reached. This is similar to what A* can do if it does not use backpointers.

Analytical Results

We now present some properties of LPA* to show that it terminates, is correct, similar to A*, and efficient. All proofs can be found in (Likhachev & Koenig 2001).

Termination and Correctness

Our first theorem shows that LPA* terminates and is correct:

Theorem 1 *ComputeShortestPath() expands each vertex at most twice, namely at most once when it is locally underconsistent and at most once when it is locally overconsistent, and thus terminates. After ComputeShortestPath() terminates, one can trace back a shortest path from s_{start} to s_{goal} by always transitioning 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).*

Similarity to A*

When we described LPA*, we already pointed out strong algorithmic similarities between LPA* and A*. We now show additional similarities between LPA* and A*. Theorem 1 already showed that ComputeShortestPath() expands each vertex at most twice. This is similar to A*, that expands each vertex at most once. Moreover, the next theorem states that the keys of the vertices expanded by ComputeShortest-Path() are monotonically nondecreasing over time. This is similar to the nondecreasing order of f-values of the vertices expanded by A*.

Theorem 2 *The keys of the vertices that ComputeShortest-Path() selects for expansion on line $\{10\}$ are monotonically nondecreasing over time until ComputeShortestPath() terminates.*

The next three theorems show that ComputeShortest-Path() expands locally overconsistent vertices in a way very similar to how A* expands vertices. The next theorem, for example, shows that the first component of the key of a locally overconsistent vertex at the time ComputeShortestPath() expands it is the same as the f-value of the vertex. The second component of its key is its start distance.

Theorem 3 *Whenever ComputeShortestPath() selects a locally overconsistent vertex s for expansion on line $\{10\}$, then its key is $k(s)\doteq[f(s);g^*(s)]$.*

Theorems 2 and 3 imply that ComputeShortestPath() expands locally overconsistent vertices in the order of monotonically nondecreasing f-values and vertices with the same f-values in the order of monotonically nondecreasing start distances. A* has the same property provided that it breaks ties in favor of vertices with smaller start distances.

Theorem 4 *ComputeShortestPath() expands locally overconsistent vertices with finite f-values in the same order as A*, provided that A* always breaks ties among vertices with the same f-values in favor of vertices with the smallest start distances and breaks remaining ties suitably.*

The next theorem shows that ComputeShortestPath() expands at most those locally overconsistent vertices whose f-values are less than the f-value of the goal vertex and those vertices whose f-values are equal to the f-value of the goal vertex and whose start distances are less than or equal to the start distances of the goal vertex. A* has the same property provided that it breaks ties in favor of vertices with smaller start distances.

Theorem 5 *ComputeShortestPath() expands at most those locally overconsistent vertices s with $[f(s);g^*(s)]\dot{\leq}[f(s_{goal});g^*(s_{goal})]$.*

Efficiency

We now show that LPA* expands many fewer vertices than suggested by Theorem 1. The next theorem shows that LPA* is efficient because it performs incremental searches and thus calculates only those g-values that have been affected by cost changes or have not been calculated yet in previous searches.

Theorem 6 *ComputeShortestPath() does not expand any vertices whose g-values were equal to their respective start distances before ComputeShortestPath() was called.*

Our final theorem shows that LPA* is efficient because it performs heuristic searches and thus calculates only the g-values of those vertices that are important to determine a shortest path. Theorem 5 has already shown how heuristics limit the number of locally overconsistent vertices expanded by ComputeShortestPath(). The next theorem generalizes this result to all locally inconsistent vertices expanded by ComputeShortestPath().

Theorem 7 *The keys of the vertices that ComputeShortestPath() selects for expansion on line $\{10\}$ never exceed $[f(s_{goal});g^*(s_{goal})]$.*

To understand the implications of this theorem on the efficiency of LPA* remember that the key $k(s)$ of a vertex s is $k(s)\doteq[\min(g(s),rhs(s))+h(s,s_{goal});\min(g(s),rhs(s))]$. Thus, the more informed the heuristics are and thus the larger they are, the fewer vertices satisfy $k(s)\dot{\leq}[f(s_{goal});g^*(s_{goal})]$ and thus are expanded.

```
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 3: D* Lite.

D* Lite

So far, we have described 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 is shown in Figure 3. It 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. D* Lite can be used to solve the goal-directed navigation problem in unknown terrain (as described in the section on "Motivation"). The terrain is modeled as an eight-connected graph. The costs of its edges are initially one. They change to infinity when the robot discovers that they cannot be traversed. One can implement the robot-navigation strategy by applying D* Lite to this graph with s_{start} being the current vertex of the robot and s_{goal} being the goal vertex.

Search Direction

We first need to switch the search direction of LPA*. The version of LPA* presented in Figure 2 searches from the start vertex to the goal vertex and thus its g-values are estimates of the start distances. D* Lite searches from the goal vertex to the start vertex and thus its g-values are estimates of the goal distances. It is derived from LPA* by exchanging the start and goal vertex and reversing all edges in the pseudo code. Thus, D* Lite operates on the original graph and there are no restrictions on the graph except that it needs to be able to determine the successors *and* predecessors of the vertices, just like LPA*. After ComputeShortestPath() returns, one can then follow a shortest path from s_{start} to s_{goal} by always moving from the current vertex s, starting at s_{start}, to any successor s' that minimizes $c(s, s') + g(s')$ until s_{goal} is reached (ties can be broken arbitrarily).

Heap Reordering

To solve robot navigation problems in unknown terrain, Main() now needs to move the robot along the path determined by CalculatePath(). Main() could recalculate the priorities of the vertices in the priority queue every time the robot notices a change in edge costs after it has moved. Unless the priorities are recalculated, they do not satisfy Invariant 3 since they are based on heuristics that were computed with respect to the old vertex of the robot. However, the repeated reordering of the priority queue can be expensive since the priority queue often contains a large number of vertices. D* Lite therefore uses a method derived from D* (Stentz 1995) to avoid having to reorder the priority queue, namely priorities that are lower bounds on the priorities that LPA* uses for the corresponding vertices. The heuristics $h(s, s')$ now need to be nonnegative and 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 requirement is not restrictive 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. 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 LPA* after the first component of the priorities of LPA* 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 LPA* after the first components of the priorities of LPA* have been increased by the current value of k_m. If $k_{old} \dot{\geq}$CalculateKey(u), then it holds that $k_{old} \doteq$CalculateKey(u) since k_{old} was a lower bound of the value returned by CalculateKey(). In this case, ComputeShortestPath() expands vertex u (by expanding a vertex, we mean executing {15'-20'}) in the same way as LPA*.

```
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)
{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 (rhs(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 4: D* Lite (optimized version).

Optimizations

Figure 4 shows D* Lite with several optimizations. An example is the termination condition of ComputeShortestPath() that can be changed to make ComputeShortestPath() more efficient. As stated, ComputeShortestPath() terminates when the start vertex is locally consistent and its key is less than or equal to U.TopKey() $\{10'\}$. However, ComputeShortestPath() can already terminate when the start vertex is not locally underconsistent and its key is less than or equal to U.TopKey(). To understand why this is so, assume that the start vertex is locally overconsistent and its key is less than or equal to U.TopKey(). Then, its key must be equal to U.TopKey() since U.TopKey() is the smallest key of any locally inconsistent vertex. Thus, ComputeShortestPath() could expand the start vertex next, in which case it would set its g-value to its rhs-value. The start vertex then becomes locally consistent, its key is less than or equal to U.TopKey(), and ComputeShortestPath() thus terminates. At this point in time, the g-value of the start vertex equals its goal distance. Thus, ComputeShortestPath() can already terminate when the start vertex is not locally underconsistent and its key is less than or equal to U.TopKey() $\{10\}$. In this case, the start vertex can remain locally inconsistent after ComputeShortestPath() terminates and its g-value thus may not be equal to its goal distance (but its rhs-value is). This is not a problem since the g-value is not used to determine how the robot should move.

Analytical Results

ComputeShortestPath() of D* Lite is similar to ComputeShortestPath() of LPA* and thus shares many properties with it. For example, ComputeShortestPath() of D* Lite expands each vertex at most twice until it returns. The following theorem shows that ComputeShortestPath() of D* Lite terminates and is correct.

Theorem 8 *ComputeShortestPath() of D* Lite always terminates and one can then follow a shortest path from s_{start} to s_{goal} by always moving from the current vertex s, starting at s_{start}, to any successor s' that minimizes $c(s,s') + g(s')$ until s_{goal} is reached (ties can be broken arbitrarily).*

Experimental Results

We now compare D* and various versions of the optimized version of D* Lite. We implemented all methods using standard binary heaps as priority queues (although using more complex data structures, such as Fibonacci heaps, as priority queues could possibly make U.Update() more efficient). The robot always observed which of its eight adjacent cells were traversable and then moved to one of them. We used the maximum of the absolute differences of the x and y coordinates of any two cells as approximations of their distance. Since all methods move the robot in the same way and D* has already been demonstrated with great success on real robots, we only need to perform a simulation study. We need to compare the total planning time of the methods. Since the actual planning times are implementation and machine dependent, they make it difficult for others to reproduce the results of our performance comparison. We therefore used three measures that all correspond to common operations performed by the methods and thus heavily influence their planning times, yet are implementation and machine independent: the total number of vertex expansions, the total number of heap percolates (exchanges of a parent and child in the heap), and the total number of vertex accesses (for example, to read or change their values). Figure 5 compares D* Lite and D* for goal-directed navigation in unknown terrain (as described in the section on "Motivation") of seven different sizes, averaged over 50 randomly generated terrains of each size whose obstacle density varies from 10 to 40 percent. The terrain is discretized into cells with uniform resolution. The figure graphs the three performance measures of D* as percent difference relative to D* Lite. Thus, D* Lite always scores zero and methods that score above zero perform worse than D* Lite. D* Lite performs better than D* with respect to all three measures, justifying our claim that it is at least as efficient as D*. The figure also shows the corresponding 95 percent confidence intervals to demonstrate that our conclusions are statistically significant. In the following, we study to which degree the combination of incremental and heuristic search that D* Lite implements outperforms incremental or heuristic searches individually. We do this for two different but related tasks, namely goal-directed navigation in unknown terrain and mapping of unknown terrain, using similar setups as in the previous experiment.

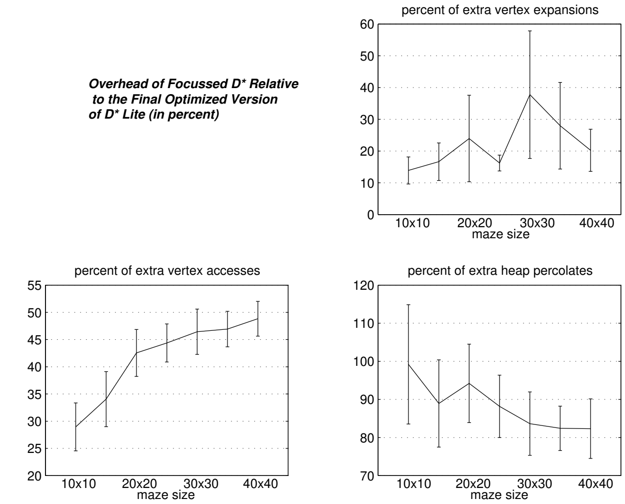

Figure 5: Comparison of D* Lite and D*.

Goal-Directed Navigation in Unknown Terrain

Figure 6 compares D* Lite, D* Lite without heuristic search, and D* Lite without incremental search (that is, A*) for goal-directed navigation in unknown terrain, using the same setup as in the previous experiment. We decided not to include D* Lite without both heuristic and incremental search in the comparison because it performs so poorly that graphing its performance becomes a problem. D* Lite outper-

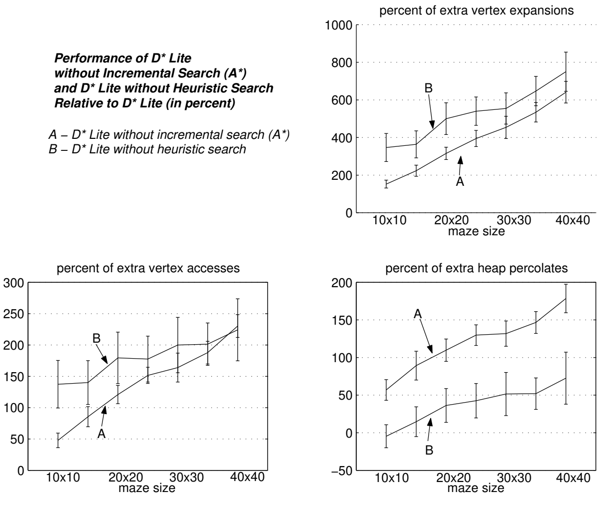

Figure 6: Goal-Directed Navigation (Uniform).

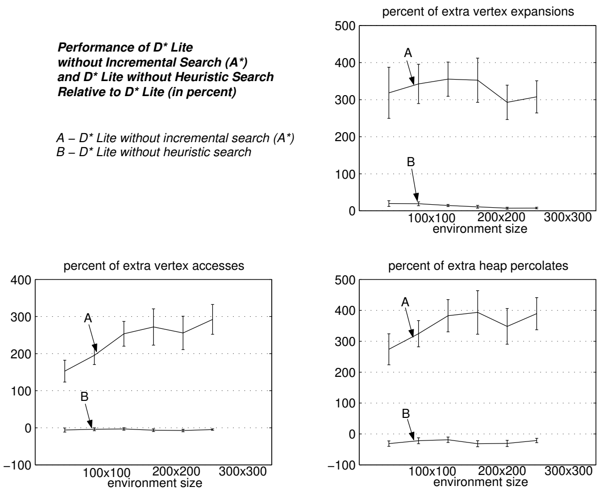

Figure 7: Goal-Directed Navigation (Adaptive).

forms the other two search methods according to all three performance measures, even by more than a factor of seven for the vertex expansions. Moreover, its advantage seems to increase as the terrain gets larger. Only for the number of heap percolates for terrain of size 10 by 10 and 15 by 15 is the difference between D* Lite and D* Lite without heuristic search statistically not significant. These results also confirm earlier experimental results that D* can outperform A* for goal-directed navigation in unknown terrain by one order of magnitude or more (Stentz 1995).

The terrain can also be discretized with nonuniform resolution. Uniform discretizations can prevent one from finding a path if they are too coarse-grained (for example, because the resolution prevents one from noticing small gaps between obstacles) and result in large graphs that cannot be searched efficiently if they are too fine-grained. Researchers have therefore developed adaptive resolution

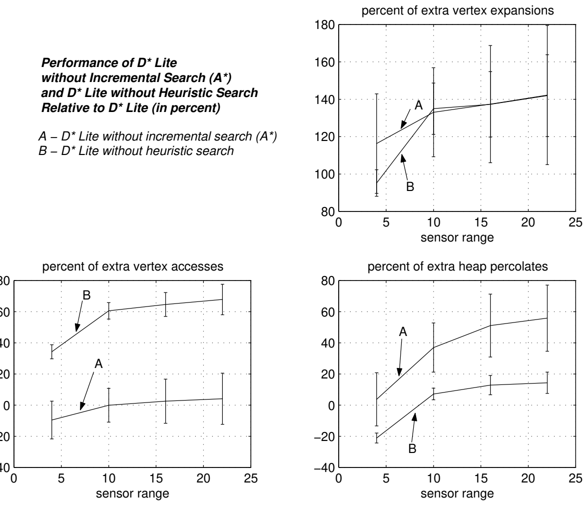

Figure 8: Mapping (Uniform).

schemes (Moore & Atkeson 1995; Yahja *et al.* 1998). We therefore used D* Lite to implement a deterministic version of the parti-game algorithm (Moore & Atkeson 1995) with adaptive discretization that discretizes terrain into cells with nonuniform resolution. In this context, Figure 7 compares D* Lite, D* Lite without heuristic search, and D* Lite without incremental search (that is, A*) for goal-directed navigation in unknown terrain terrains of six different sizes, averaged over 25 randomly generated terrains of each size with an obstacle density of 30 percent each. D* Lite outperforms D* Lite without incremental search (that is, A*) according to all three performance measures, even more than a factor of four for the vertex expansions. On the other hand, different from goal-directed navigation in unknown terrain with uniform discretization, D* Lite and D* Lite without heuristic search perform about equally well.

Mapping of Unknown Terrain

D* Lite can also be used to implement greedy mapping (Koenig, Tovey, & Halliburton 2001), a simple but powerful mapping strategy that has repeatedly been used on mobile robots by different research groups (Thrun *et al.* 1998; Koenig, Tovey, & Halliburton 2001; Romero, Morales, & Sucar 2001). Greedy mapping discretizes the terrain into cells with uniform resolution and then always moves the robot from its current cell to the closest cell with unknown traversability, until the terrain is mapped. In this case, the graph is an eight-connected grid. The costs of its edges are initially one. They change to infinity when the robot discovers that they cannot be traversed. There is one additional vertex that is connected to all grid vertices. The costs of these edges are initially one. They change to infinity once the corresponding grid vertex has been visited. One can implement greedy mapping by applying D* Lite to this graph with s_{start} being the current vertex of the robot and s_{goal} being the additional vertex.

Figure 8 compares D* Lite, D* Lite without heuristic search, and D* Lite without incremental search (that is, A*) for greedy mapping with different sensor ranges, averaging over 50 randomly generated grids of size 64 by 25. The terrain is discretized into cells with uniform resolution. We varied the sensor range of the robot to simulate both short-range and long-range sensors. For example, if the sensor range is four, then the robot can sense all untraversable cells that are up to four cells in any direction away from the robot as long as they are not blocked from view by other untraversable cells. The number of vertex expansions of D* Lite is always far less than that of the other two methods. This also holds for the number of heap percolates and vertex accesses, with the exception of sensor range four for the heap percolates and the number of vertex accesses of D* Lite without incremental search.

Conclusions

In this paper, we have presented D* Lite, a novel fast replanning method for robot navigation in unknown terrain that implements the same navigation strategies as Focussed Dynamic A* (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. 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 easy to understand and extend, yet at least as efficient as D*. We believe that our experimental and analytical results about D* Lite provide a strong algorithmic foundation for further research on fast replanning methods in artificial intelligence and robotics and complement the research on symbolic replanning methods in artificial intelligence (Hanks & Weld 1995) as well as the research on incremental search methods in both algorithm theory (Frigioni, Marchetti-Spaccamela, & Nanni 2000) and artificial intelligence (Edelkamp 1998).

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 fellowship award. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations and agencies or the U.S. government.

References

Edelkamp, S. 1998. Updating shortest paths. In *Proceedings of the European Conference on Artificial Intelligence*, 655–659.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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