The Delayed D* Algorithm for Efficient Path Replanning

Dave Ferguson and Anthony Stentz
Robotics Institute
Carnegie Mellon University
Pittsburgh, PA, USA
{*dif, tony*}*@cmu.edu*

Abstract— **Mobile robots are often required to navigate environments for which prior maps are incomplete or inaccurate. In such cases, initial paths generated for the robots may need to be amended as new information is received that is in conflict with the original maps. The most widely used algorithm for performing this path replanning is Focussed Dynamic A* (D*), which is a generalization of A* for dynamic environments. D* has been shown to be up to two orders of magnitude faster than planning from scratch. In this paper, we present a new replanning algorithm that generates equivalent paths to D* while requiring about half its computation time. Like D*, our algorithm incrementally repairs previous paths and focusses these repairs towards the current robot position. However, it performs these repairs in a novel way that leads to improved efficiency.**

I. Introduction

Path planning for mobile robots consists of finding a sequence of state transitions that leads a robot from its initial state to some desired goal state. Typically, the states are robot locations and the transitions represent actions the robot can take, each of which has an associated cost. A path is said to be optimal if the sum of its transition costs (arc costs) is minimal across all possible paths leading from the initial position (start state) to the goal position (goal state). Such paths can be efficiently generated from a map of the environment using focussed algorithms such as A* [1].

However, when operating in real environments, a mobile robot usually does not have complete map information. As a result, any path generated using its initial map may turn out to be invalid or suboptimal as it receives updated map information through, for example, an onboard sensor. It is thus important that the robot is able to update its map and replan optimal paths when new information arrives.

A number of algorithms exist for performing this replanning [2], [3], [4], [5], [6], [7], [8]. Focussed Dynamic A* (D*) [2] and D* Lite [8] are currently the most widely used of these algorithms, due to their efficient use of heuristics and incremental updates. D* has been shown to be one to two orders of magnitude more efficient than planning from scratch with A*, and it has been incorporated into a plethora of real robotic systems [9], [10], [11], [12], [13]. D* Lite is a simplified version of D* that has been found to be slightly more efficient by some measures [8]. It has been used to guide Segbots and ATRV vehicles in urban terrain [14]. Both algorithms guarantee optimal paths over grid-based representations of a robot's environment.

In this paper, we present a new algorithm that solves the same problems as D* and D* Lite yet can be significantly more efficient. We begin by describing the D* and D* Lite algorithms. Next, we introduce our algorithm Delayed D* along with some of the intuition behind it. We then present comparative results on three common navigation scenarios. We conclude with discussion and future work.

II. Focussed Replanning: D* and D* Lite

Both D* and D* Lite maintain least-cost paths between a start state and a goal state as the costs of arcs between states change. Both algorithms can handle increasing or decreasing arc costs and dynamic start states. They are both thus capable of handling the goal-directed mobile robot navigation problem, which entails a robot moving from some initial state to a goal state while updating its map information through an onboard sensor. Because the two algorithms are fundamentally very similar, we restrict our attention here to D* Lite, which has been found to be slightly more efficient for navigation tasks [8]. For more details on each algorithm, see [2] and [8].

D* Lite maintains a least-cost path from a start state $s_{start} \in S$ to a goal state $s_{goal} \in S$, where S is the set of states in some finite state space. To do this, it stores an estimate $g(s)$ of the cost from each state s to the goal. It also stores a one-step lookahead cost $rhs(s)$ which satisfies:

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

where $Succ(s) \in S$ denotes the set of successors of s and $c(s,s')$ denotes the cost of moving from s to s' (the arc cost). A state is called consistent iff its g-value equals its rhs-value, otherwise it is either overconsistent (if $g(s) > rhs(s)$) or underconsistent (if $g(s) < rhs(s)$).

Like A*, D* Lite uses a heuristic and a priority queue to focus its search and to order its cost updates efficiently. The heuristic $h(s,s')$ estimates the cost of moving from state s to s', and needs to satisfy $h(s,s') \leq c^*(s,s')$ and $h(s,s'') \leq h(s,s') + h(s',s'')$ for all states $s, s', s'' \in S$, where $c^*(s,s')$ is the cost associated with a least-cost path from s to s'. The priority queue always holds exactly the inconsistent states; these are the states that need to be updated and made consistent.

The priority $k(s)$ of a state s in the queue is:

$$\begin{aligned} k(s) &= [k_1(s), k_2(s)] \\ &= [min(g(s), rhs(s)) + h(s_{start}, s), min(g(s), rhs(s))]. \end{aligned}$$

A lexicographic ordering is used on the priorities, so that priority $k(s)$ is less than or equal to priority $k(s')$, denoted

$k(s) \dot{\leq} k(s')$, iff $k_1(s) < k_1(s')$ or both $k_1(s) = k_1(s')$ and $k_2(s) \leq k_2(s')$. D* Lite expands states from the queue in increasing priority, updating their g-values and their predecessors' rhs-values, until there is no state in the queue with a priority less than that of the start state. It thus performs similarly to a backwards A* search during its generation of an initial solution path.

When arc costs change, D* Lite updates the rhs-values of each state immediately affected by the changed arc costs and places those states that have been made inconsistent onto the queue. As before, it then expands the states on the queue in order of increasing priority until there is no state in the queue with a priority less than that of the start state. This termination condition guarantees that an optimal path will have been found from the start state to the goal state when processing is finished.

D* Lite is efficient because it uses a heuristic to restrict attention to only those states that could possibly be relevant to repairing the current solution path from a given start state to the goal state. When arc costs decrease, the heuristic ensures that only those newly-overconsistent states that could potentially decrease the cost of the start state are processed. When arc costs increase, it ensures that only those newly-underconsistent states that could potentially invalidate the current cost of the start state are processed.

However, perhaps we can do better by being even more restrictive. It is possible that by using the above approach, we may expand many more states than is necessary. This is because, even if an inconsistent state has a priority less than the start state, its inconsistency may not affect the optimality of the current solution path. Figure 1 illustrates such a scenario. It would be ideal if we could tell whether the inconsistency of a particular state is of consequence to the validity of the current solution without actually propagating this inconsistency.

When we encounter a series of cost decreases that affect states with lower priorities than the start state, there is no simple check that will provide us with this information. When we encounter a similar series of cost increases, however, there *is* such a check: we can guarantee the optimality of our current solution if none of these increases takes place along the solution path. This qualitative difference suggests different treatment for states made underconsistent (due to arc cost increases) and states made overconsistent (due to arc cost decreases).

Motivated by this finding, we have created the algorithm *Delayed D**, which processes underconsistent states much more selectively than overconsistent states.

III. Delayed D*

Delayed D* is a modified version of D* Lite that delays the propagation of cost increases as long as possible. When cost changes occur, the rhs-values of all affected states are updated and the overconsistent states are processed immediately, as in D* Lite. The underconsistent states are ignored. Then, when new values of the overconsistent states have been adequately propagated through the state space, the returned solution path is checked for any

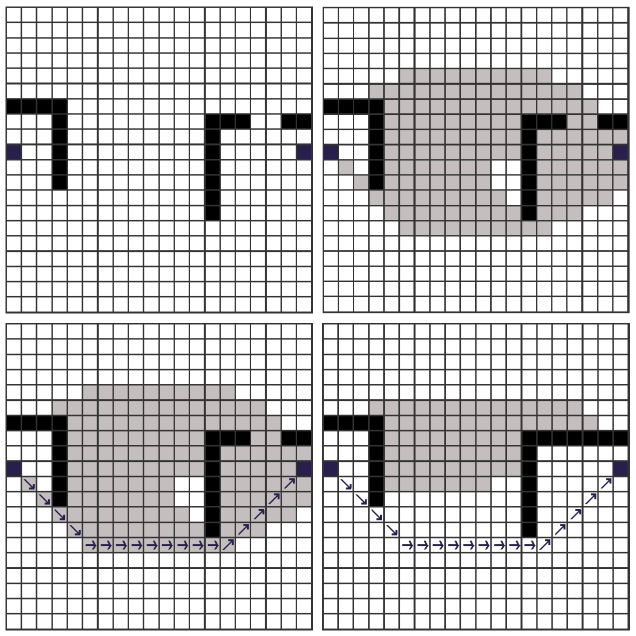

Fig. 1. An eight-connected grid replanning scenario in which a couple of inconsistent states cause D* Lite to make significant repairs. A least-cost path is maintained between state s_{start} and state s_{goal}, the two dark-gray cells on the left and right sides of each grid, respectively. The top-right grid shows the cells expanded by D* Lite (shaded light-gray) during initial planning. The bottom-left grid shows the cells along the initial solution path (dark-gray arrows). The bottom-right grid shows the states re-expanded by D* Lite (shaded light-gray) after information reveals that the gap in the right wall is blocked. Since none of these underconsistent states resided upon the path from s_{start} to s_{goal}, they could have been ignored without jeopardizing optimality.

underconsistent states. All underconsistent states on the path are added to the priority queue and their updated values are propagated through the state space. Because the current propagation phase may alter the solution path, the new solution path needs to be checked for underconsistent states. The entire process repeats until a solution path that contains only consistent states is returned.

By delaying the processing of underconsistent states, Delayed D* holds two advantages over D* Lite. Firstly, we will be able to ignore some underconsistent states entirely, reducing our overall computation. Secondly, even when we have to process underconsistent states, if we have delayed processing them long enough then perhaps the effects of various underconsistent states can be incorporated at the same time, in a single propagation. So, rather than propagating cost increases through the state space every time a new underconsistent state turns up, we wait until some underconsistent state jeopardizes the optimality of the current solution, then do a single propagation of values that may deal with several underconsistent states at once.

A. The Algorithm

We have provided two versions of the Delayed D* algorithm. The first version, given in Figure 2, maintains a least-cost path from a fixed initial state to the goal state. We have presented the algorithm in the same framework as that used by D* Lite [8] to highlight its similarities. In our pseudocode, we have used notation defined earlier as

```
CalculateKey(s)
01. return [$min(g(s), rhs(s)) + h(s_{start}, s); min(g(s), rhs(s))$];

Initialize()
02. $U = \emptyset$;
03. for all $s \in S$
04.   $rhs(s) = g(s) = \infty$;
05. $rhs(s_{goal}) = 0$;
06. Insert($U$, $s_{goal}$, $[h(s_{start}, s_{goal}), 0]$);

UpdateVertex(s)
07. if ($g(s) \neq rhs(s)$)
08.   Insert($U$, $s$, CalculateKey($s$));
09. else if ($g(s) = rhs(s)$) and ($s \in U$)
10.   Remove($U$, $s$);

UpdateVertexLower(s)
11. if ($g(s) > rhs(s)$)
12.   Insert($U$, $s$, CalculateKey($s$));
13. else if ($g(s) = rhs(s)$) and ($s \in U$)
14.   Remove($U$, $s$);

ComputeShortestPathDelayed()
15. while ($U$.MinKey() $\dot{<}$ CalculateKey($s_{start}$) OR $g(s_{start}) \neq rhs(s_{start})$)
16.   $s = U$.Top();
17.   if ($g(s) > rhs(s)$)
18.       $g(s) = rhs(s)$
19.       Remove($U$, $s$);
20.       for all $x \in Pred(s)$
21.         $rhs(x) = min(rhs(x), c(x, s) + g(s))$;
22.         UpdateVertexLower($x$);
23.   else
24.       $g_{old} = g(s)$;
25.       $g(s) = \infty$;
26.       for all $x \in Pred(s) \cup s$
27.         if ($rhs(x) = c(x, s) + g_{old}$ OR $x = s$)
28.           if ($x \neq s_{goal}$)
29.               $rhs(x) = min_{x' \in Succ(x)}(c(x, x') + g(x'))$;
30.         UpdateVertex($x$);

FindRaiseStatesOnPath()
31. $s = s_{start}, raise = false, loop = false, ctr = 0$;
32. while ($s \neq s_{goal}$ AND $loop = false$ AND $ctr < maxsteps$)
33.   $x = argmin_{s' \in succ(s)}(c(s, s') + g(s'))$;
34.   $rhs(s) = c(s, x) + g(x)$;
35.   if ($g(s) \neq rhs(s)$)
36.       UpdateVertex($s$);
37.       $raise = true$;
38.   if ($x = s$)
39.       $loop = true$;
40.   else
41.       $s = x$;
42.       $ctr = ctr + 1$;
43. return $raise$;

Main()
44. Initialize();
45. ComputeShortestPathDelayed();
46. forever
47.   Wait for changes in edge costs;
48.   for all directed edges $(u, v)$ with changed edge costs
49.       $c_{old} = c(u, v)$;
50.       Update the edge cost $c(u, v)$;
51.       if ($c_{old} > c(u, v)$)
52.         $rhs(u) = min(rhs(u), c(u, v) + g(v))$;
53.       else if ($rhs(u) = c_{old} + g(v)$)
54.         if ($u \neq s_{goal}$)
55.             $rhs(u) = min_{u' \in Succ(u)}(c(u, u') + g(u'))$;
56.       UpdateVertexLower($u$);
57.   ComputeShortestPathDelayed();
58.   $raise$ = FindRaiseStatesOnPath();
59.   while ($raise$)
60.       ComputeShortestPathDelayed();
61.       $raise$ = FindRaiseStatesOnPath();
```

Fig. 2. **The Delayed D* Algorithm: Fixed Initial State.**

well as some extra: U is the priority queue, $h(s, s')$ is the heuristic cost from state s to state s', $argmin_{s' \in succ(s)} f()$ returns the successor of s for which function f is minimized, and all variables not already mentioned are local to the respective functions ($raise$, $loop$, ctr, c_{old}, u, etc).

This first version of Delayed D* begins by initializing the g and rhs-values of each state to infinity, and then places the goal state s_{goal} onto the queue (lines 03 - 06). Next, **ComputeShortestPathDelayed()** (CSPD) is called, which computes a least-cost path from s_{start} to s_{goal}. This initial path is computed in exactly the same way as it would be in D* Lite.

Once the initial least-cost path has been found, Delayed D* waits for arc costs to change. When they do, it updates the rhs-value of all immediately affected states and adds only the newly *overconsistent* states to the queue (line 55).

```
CalculateKey(s)
01. return [$min(g(s), rhs(s)) + h(s_{start}, s) + \mathbf{k_m}; min(g(s), rhs(s))$];

Initialize()
02. $U = \emptyset$; $\mathbf{k_m = 0}$;
03. for all $s \in S$
04.   $rhs(s) = g(s) = \infty$;
05. $rhs(s_{goal}) = 0$;
06. Insert($U$, $s_{goal}$, $[h(s_{start}, s_{goal}), 0]$);

ComputeShortestPathDelayed()
07. while ($U$.MinKey() $\dot{<}$ CalculateKey($s_{start}$) OR $g(s_{start}) \neq rhs(s_{start})$)
08.   $s = U$.Top();
09.   $\mathbf{k_{old} = U.TopKey()}$;
10.   $\mathbf{k_{new} = CalculateKey(s)}$;
11.   if ($\mathbf{k_{old} \dot{<} k_{new}}$)
12.       Insert($\mathbf{U, s, k_{new}}$);
13.   else if ($g(s) > rhs(s)$)
14.       $g(s) = rhs(s)$
15.       Remove($U$, $s$);
16.       for all $x \in Pred(s)$
17.         $rhs(x) = min(rhs(x), c(x, s) + g(s))$;
18.         UpdateVertexLower($x$);
19.   else
20.       $g_{old} = g(s)$;
21.       $g_s = \infty$;
22.       for all $x \in Pred(s) \cup s$
23.         if ($rhs(x) = c(x, s) + g_{old}$ OR $x = s$)
24.           if ($x \neq s_{goal}$)
25.               $rhs(x) = min_{x' \in Succ(x)}(c(x, x') + g(x'))$;
26.         UpdateVertex($x$);

Main()
27. $\mathbf{s_{last} = s_{start}}$;
28. Initialize();
29. ComputeShortestPathDelayed();
30. while ($\mathbf{s_{start} \neq s_{goal}}$)
31.   $\mathbf{s_{start} = argmin_{s \in Succ(s_{start})}(c(s_{start}, s) + g(s))}$;
32.   Move to $\mathbf{s_{start}}$ and check for changed edge costs
33.   if any edge costs changed
34.       $\mathbf{k_m = k_m + h(s_{last}, s_{start})}$;
35.       $\mathbf{s_{last} = s_{start}}$;
36.       for all directed edges $(u, v)$ with changed edge costs
37.         $c_{old} = c(u, v)$;
38.         Update the edge cost $c(u, v)$;
39.         if ($c_{old} > c(u, v)$)
40.           $rhs(u) = min(rhs(u), c(u, v) + g(v))$;
41.         else if ($rhs(u) = c_{old} + g(v)$)
42.           if ($u \neq s_{goal}$)
43.               $rhs(u) = min_{u' \in Succ(u)}(c(u, u') + g(u'))$;
44.         UpdateVertexLower($u$);
45.       ComputeShortestPathDelayed();
46.       $raise$ = FindRaiseStatesOnPath();
47.       while ($raise$)
48.         ComputeShortestPathDelayed();
49.         $raise$ = FindRaiseStatesOnPath();
```

Fig. 3. **The Delayed D* Algorithm: Dynamic Initial State.**

It then calls CSPD again to propagate the new values of these overconsistent states through the state space. When it has finished, the solution path is checked, in **FindRaiseStatesOnPath()** (FRSOP), for any underconsistent states. If any exist, they are placed on the queue and their new values are propagated through the state space by another call to CSPD.

When a state is expanded in CSPD (line 16), it is treated differently depending on whether it is overconsistent or underconsistent. If it is overconsistent, it updates its g-value to equal its rhs-value, then uses its new g-value to lower the rhs-values of its predecessors. If this causes some predecessor to become overconsistent, the predecessor is added to the queue. Any predecessor states that are underconsistent are ignored.

If the state to be processed is underconsistent, it sets its g-value to infinity, then updates the rhs-values of all predecessors that use the g-value of the current state for their rhs-values. Any predecessors that are inconsistent are added to the queue.

Since all underconsistent predecessor states are ignored whenever an overconsistent state is expanded, the g-values of states along the solution path returned by CSPD are lower bounds of the optimal costs of the states. Thus, each time CSPD terminates, FRSOP must be called to check

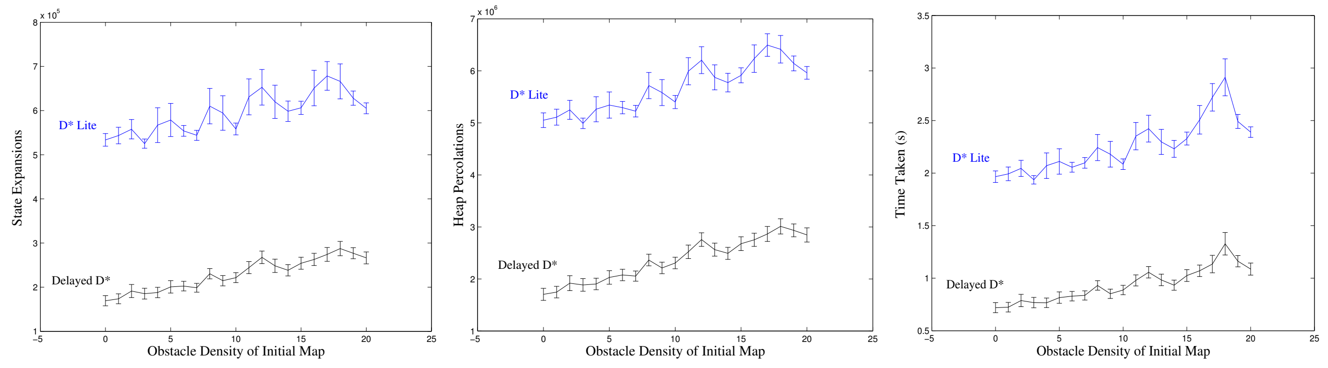

Fig. 4. Results from our first experiment. A least-cost path was maintained between a fixed initial state and a goal state as cells in the environment had their traversability changed. Shown here are the number of states expanded, the number of heap percolations, and the total CPU time taken.

that the g-values of states along the solution path equal their rhs-values (lines 60 - 61). When they do, we can prove that the solution path is optimal (see below).

The second version of Delayed D* allows for a changing initial state. Presented in Figure 3, this algorithm is very similar to both D* and D* Lite in its use of a bias function (k_m) to avoid reordering the queue each time the initial state changes. We have left out of Figure 3 all the functions used by the algorithm that are exactly the same as those in Figure 2 and have highlighted in bold the differences between this more general version of Delayed D* and the version presented above.

This version updates the initial state s_{start} every time the robot moves along the solution path. Because this alters the heuristic value $h(s_{start}, s)$ of each state s on the queue, the stored key values of states on the queue are no longer correct. Updating these key values results in a reordering of the priority queue. However, we avoid reordering the queue every time the robot moves by adopting a technique originally presented in [2]. We compute a lower bound for the new $h(s_{start}, s)$ value by assuming that the robot's movement is directly towards s. So, if the robot moves from s_{last} to s_{start}, we can update the heuristic cost of s to be $h(s_{last}, s) - h(s_{last}, s_{start})$. When s is popped, we then correct its heuristic value to be the true $h(s_{start}, s)$ value and reinsert it onto the queue with an updated key value (lines 09 - 12).

The advantage of this approach is that all states already on the queue have their keys updated by the same $-h(s_{last}, s_{start})$ value, so the order of states on the queue is preserved. Further, we can avoid these updates altogether if we instead *add* the value $h(s_{last}, s_{start})$ to the key of all states to be inserted onto the queue. The remaining computation is simply the reinsertion of popped states whose key values are out of date. In our algorithm description, we maintain the current cumulative adjustment value as k_m (line 34).

Note that, for both versions of Delayed D*, when choosing best successor states in FRSOP and in the **Main()** function, ties can be broken by any method so desired, but the method must be consistent. If a particular successor state is chosen from a state s in FRSOP, then this same successor state should be chosen from s in the **Main()** function. This is important for ensuring that the agent traverses the least-cost path shown to be valid in FRSOP. Processing the successors of each state in a fixed order is one simple way to guarantee this.

We proved the termination and optimality of Delayed D* in an earlier technical report [15].

Theorem 1. *The Delayed D* algorithm always terminates and an optimal solution path can then be followed from s_{start} to s_{goal} by always moving from the current state s, starting at s_{start}, to any successor s' that minimizes $c(s, s') + g(s')$.*

IV. Results

We implemented both versions of Delayed D* and compared them to the optimized version of D* Lite [8] on three common path planning tasks.

The first task was to maintain a least-cost path from the left side of an environment to the right side, as the terrain associated with areas of the environment changed. We generated 1050 random environments of size 500 $\times$ 500: 50 environments with no obstacles (but with varied terrain costs), 50 with 1% obstacle cells, 50 with 2% obstacle cells, and so on, up to 50 with 20% obstacle cells. The terrain cost associated with traversing non-obstacle cells was also randomly generated (with the minimum possible cost being 1.0). We used Euclidean distance for our heuristic h.

For each environment, we first generated an initial optimal path using D* Lite. We then randomly selected 100 cells and flipped their terrain values: traversable cells became untraversable and untraversable cells became traversable. We then used D* Lite and Delayed D* to replan an optimal path given these changes. We repeated the above steps 50 times (for a total of 5000 altered terrain costs) and recorded the performance of each algorithm in replanning.

Figure 4 shows the results of this experiment. We have included three performance measures: the number of states expanded, the number of heap percolations (i.e., the number of times a parent and a child are swapped in our heap priority queue), and the CPU time taken by a P3 1.4 GHz processor. In all our graphs, we have included error bars representing the standard error of the mean. According to all three measures, Delayed D* outperformed D* Lite by roughly a factor of 2.

Note that in this experiment the agent was not moving through the environment; we were maintaining an optimal path from a fixed start state to the goal. Such a situation arises when we have a number of agents traversing from the same initial position to the goal, or in domains such as network routing, where we are interested in maintaining an optimal path between two fixed states in our system.

The second and third tasks we looked at concerned an agent moving through a state space in which arc costs are changing or for which the agent had imperfect initial information. In these scenarios, the agent began with some prior information about the costs of arcs between states and could update its information as it traversed through the state space.

As we are most concerned with robot navigation, we simulated an agent equipped with a sensor that would allow it to detect the terrain value of areas of the environment within some sensor radius of the agent. For our testing, we again used the same series of 500 $\times$ 500 environments but made the common simplification that all traversable cells had the same terrain cost, resulting in a binary map. For each traverse, the agent started at the left side and worked its way to the right.

We first looked at an agent that began with *no* information about its environment, so that its initial map was completely empty and assumed to be everywhere traversable. As the agent moved through the environment, it updated its map to reflect the true nature of the environment, using an omnidirectional sensor with a 30-cell field of view. The results of this experiment are shown in Figure 5.

In these completely unknown environments, Delayed D* showed a slight performance improvement; this improvement increased with the difficulty of the environment. Because unknown environments are assumed to be completely free of obstacles, the number of states expanded when producing the initial solution path in these domains is very small. Thus, in such situations D* Lite performs in a similar manner to Delayed D*: the only states that are processed as a result of observed cost changes are those underconsistent states that reside on or are very close to the solution path.

Our final simulation concerned an agent that began with a complete but inaccurate map of its environment. Here, we used the same series of 500 $\times$ 500 environments, but we randomly flipped the terrain values of 25% of the cells, so that traversable cells became untraversable, and vice versa. The agent then moved through the environment, updating its map information to reflect the observed environment. The results of this experiment are shown in Figure 6.

In these partially known environments, where the number of states with key values less than the start state can be large and costs both increase and decrease, Delayed D* showed a significant improvement in efficiency.

V. Discussion

The results presented above demonstrate that Delayed D* is a valuable extension to the D* Lite algorithm. We believe that the method Delayed D* uses to process underconsistent states is highly beneficial to replanning.

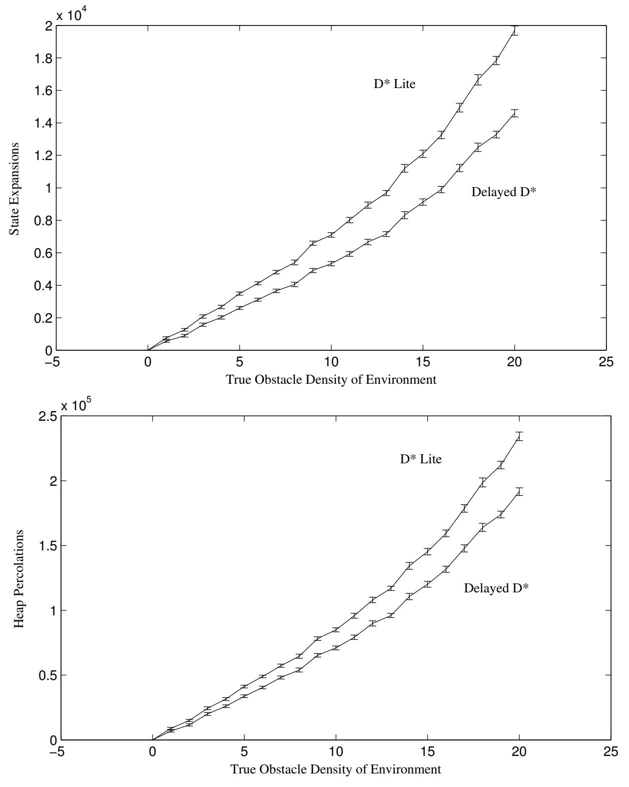

Fig. 5. Results from our first navigation experiment. The agent began with an empty map and updated its map as it traversed the environment. Shown here are the number of states expanded and heap percolations.

It is worth making two points about the version of Delayed D* used for our experiments. Firstly, in our simulations we dealt with random environments. However, real environments navigated by mobile robots may exhibit structure. In such cases, we may want to alter the **FindRaiseStatesOnPath()** function to add to the queue not just the underconsistent states found on the current solution path, but all states adjacent to these states that are also underconsistent. If the agent were to observe a large obstacle in its path, this would prevent it from potentially processing only a small "wedge" of the obstacle each time CSPD is called. Instead, if the entire obstacle was added at once, it could be processed more efficiently.

Secondly, when a cell becomes untraversable (i.e., an obstacle) during our simulations, it is *never* again placed on the queue, even if its least-cost successor is popped as an underconsistent state. Once a cell becomes an obstacle, any dependent predecessors are updated and the cell itself is afterwards ignored. This means that the propagation of underconsistent states can terminate early if obstacles are encountered. This early termination further delays the processing of underconsistent states and contributes to the overall efficiency gain of our Delayed D* implementation.

Looking critically at the Delayed D* algorithm, two possible sources of inefficiency can be found. Firstly, it is possible to construct worst-case scenarios where the processing of underconsistent states changes the solution path several times, each time producing a new path containing underconsistent states. This results in a number of propagation phases, each starting from a new set of underconsistent states, where each propagation phase may process roughly the same area of the state space. This will be less efficient than dealing with all the relevant underconsistent states at once. However, in realistic navigation tasks, worst-case scenarios occur very infrequently. As our results have suggested, Delayed D* is far more efficient on average than D* Lite. In fact, over all our test cases (1050 environments, 3150 total test runs), there was not a single run during which D* Lite expanded fewer states than Delayed D*.

The second possible source of inefficiency concerns the **FindRaiseStatesOnPath()** function. Since Delayed D* ignores underconsistent states when they first appear, the consistency of the current solution path needs to be checked after each propagation phase. This adds a source of computation to the planning task not associated with competing algorithms such as D* Lite. However, the extra processing required to perform this check is only influential in the most trivial of state spaces. Because the solution path always represents a one dimensional slice through the space, in higher dimensional state spaces (where the underconsistent states processed unnecessarily by D* Lite become a real burden) this check requires a relatively insignificant amount of computation. In fact, in light of this, we are currently extending Delayed D* to an anytime algorithm for operating in very large state spaces, where we believe it will be even more effective at reducing the overall computation required to generate solutions.

VI. Conclusion

In this paper we have presented Delayed D*, a new replanning algorithm for maintaining optimal paths through dynamic graphs. We have compared our algorithm to the leading replanning algorithm in several domains and found ours to be significantly more efficient on average. We are currently looking at applying the techniques behind Delayed D* to high-dimensional path planning problems.

VII. Acknowledgements

This work was partially sponsored by the U.S. Army Research Laboratory, under contract "Robotics Collaborative Technology Alliance". The views contained in this document are those of the authors and do not represent the official policies or endorsements of the U.S. Government. Dave Ferguson is supported by a National Science Foundation Graduate Research Fellowship.

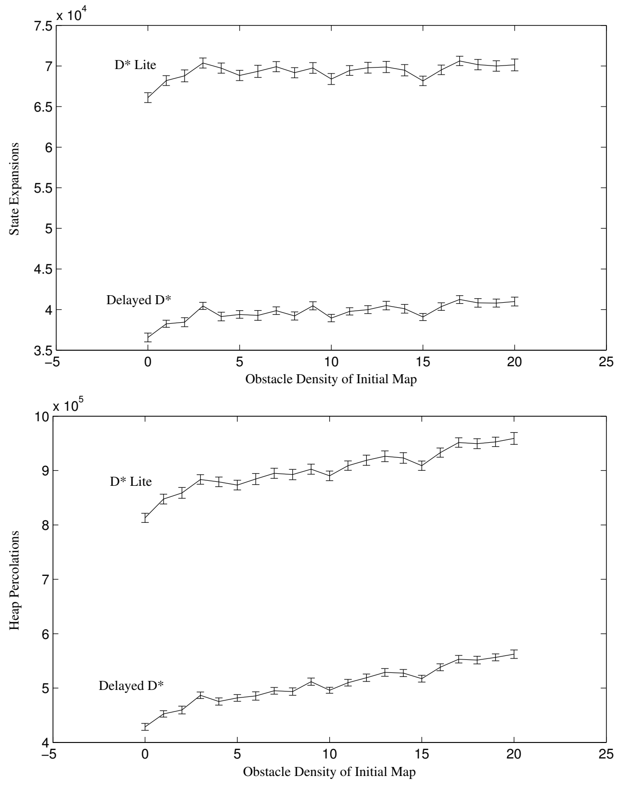

Fig. 6. Results from our second navigation experiment. The agent began with a map that contained incorrect values for 25% of the cells. Shown here are the number of states expanded and heap percolations.

References

[1] N. Nilsson, *Principles of Artificial Intelligence*. Tioga Publishing Company, 1980.

[2] A. Stentz, "The Focussed D* Algorithm for Real-Time Replanning," in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1995.

[3] M. Barbehenn and S. Hutchinson, "Efficient search and hierarchical motion planning by dynamically maintaining single-source shortest path trees," *IEEE Transactions on Robotics and Automation*, vol. 11, no. 2, pp. 198–214, 1995.

[4] 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.

[5] T. Ersson and X. Hu, "Path planning and navigation of mobile robots in unknown environments," in *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS)*, 2001.

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

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

[8] S. Koenig and M. Likhachev, "Improved fast replanning for robot navigation in unknown terrain," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2002.

[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 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 (IAS)*, 2000.

[12] S. Thayer, B. Digney, M. Diaz, A. Stentz, B. Nabbe, and M. Hebert, "Distributed robotic mapping of extreme environments," in *Proceedings of SPIE Mobile Robots*, 2000.

[13] R. Zlot, A. Stentz, M. Dias, and S. Thayer, "Multi-robot exploration controlled by a market economy," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2002.

[14] M. Likhachev, "Search techniques for planning in large dynamic deterministic and stochastic environments," School of Computer Science, Carnegie Mellon University, 2003, thesis proposal.

[15] D. Ferguson and A. Stentz, "Delayed D*: The Proofs," Carnegie Mellon Robotics Institute, Tech. Rep. CMU-RI-TR-04-51, September 2004.