Avoiding Unnecessary Calculations in Robot Navigation

Weiya Yue, John Franco *

Abstract— **For solving problems of robot navigation over unknown and changing terrain, many algorithms have been invented. For example, D* Lite, which is a dynamic, incremental search algorithm, is the most successful one. The improved performance of the D* Lite algorithm over other replanning algorithms is largely due to updating terrain cost estimates rather than recalculating them between robot movements. However, the D* Lite algorithm performs some recalculation every time a change in terrain is discovered. In this paper, it is shown that recalculation is often not necessary, particularly when several optimal solutions (shortest paths) exist, and an efficient test for determining this is presented. These ideas are packaged in a modified version of D* Lite which we call ID* Lite for Improved D* Lite. We present experimental results that show the speedups possible for a variety of benchmarks. Also, a novel realistic benchmark is described.**

Keywords: robot navigation, uncertainty, planning

1 Introduction

Advances in robot replanning have made possible the development of serious autonomous vehicles that may be used to explore other planets, gather data in areas considered too dangerous for humans, and even park themselves without human involvement. Notable among these advances is the marriage of incremental search algorithms with sophisticated search heuristics that exploit learned terrain information to narrow the search space and thereby speed up the replanning process. The D* Lite algorithm [3, 4] represents the state-of-the-art in such replanning algorithm development. A descendant of the A* and D* [1, 2] algorithms, D* Lite is easily implemented and its "experimental properties show that D* Lite is at least as efficient as D*." It has been used successfully in a variety of roles.

The terrain information that is used by D* Lite is represented abstractly as a directed graph $G(V, E)$ with distinguished start vertex v_s, goal vertex v_g, and positive integer costs $c : V \times V \mapsto Z^+$ on edges. A "robot" initially occupies v_s and moves along edges to v_g. On every movement through a single edge, called a *transition*, edge costs can change. The cost of a robot's path from v_s to v_g is the sum of the costs of the edges traversed *when they are traversed*. D* Lite attempts to determine the lowest cost sequence of transitions that will take a robot from v_s to v_g. The problem is complicated by the fact that edge cost changes are not predictable.

*Department of Computer Science, University of Cincinnati, Cincinnati OH, 45220. Email: weiyayue@hotmail.com, franco@gauss.ececs.uc.edu

It is unlikely that D* Lite will find the lowest cost sequence of transitions that advances the robot from v_s to v_g because it never has complete information about edge cost changes until the last transition. Moreover, current edge costs are known to D* Lite only within the robot's *view* which consists of the edges out to vertices that are within a fixed distance, called the *sensor-radius*, from the current position of the robot, which we will always designate as v_c below. But D* Lite can always find the lowest cost sequence from v_c to v_g based on the known edge costs within the view and assuming that current estimates of other edge costs are their actual costs. We will use the term *shortest path* to refer to such a sequence and we will use $c_e(w, u)$ to represent the estimated cost of any edge $\langle w, u\rangle$: if $\langle w, u\rangle$ is in the view then $c_e(w, u) = c(w, u)$, otherwise $c_e(w, u)$ does not change from round to round. When a shortest path is computed, the algorithm moves the robot along that path, one transition per round, until the edge cost assumption is violated at some vertex v_x that is within the view and on the path. At that point D* Lite recomputes a new shortest path from v_c to v_g and repeats the above two steps until the robot occupies v_g.

D* Lite is assisted by two functions which take a vertex v as input and return the cost of the shortest path from v to v_g based on current edge costs and estimated costs. One function, $g(v)$, is identical to the function of the same name that is used by the A* algorithm to estimate costs from v_s but in this role estimates the cost to v_g. The other, $rhs(v)$, is a more informed, one-level-lookahead function whose output is expressed as

$$rhs(v) = \min_{v' \in Succ(v)} g(v') + c_e(v, v')$$

if $v \neq v_g$ and otherwise $rhs(v_g) = 0$, where $Succ(v)$ is the set of all vertices, referred to as *successors* below, that are reachable from v through a single edge. A vertex v for which $rhs(v) = g(v)$ is said to be locally *consistent*. If $g(v) > rhs(v)$ then v is locally *overconsistent* and if $g(v) < rhs(v)$ then v is *underconsistent*. Shortest path costs from all vertices to v_g are known precisely if and only if all vertices are locally consistent. In that case, shortest paths can be computed by following minimum edge costs, ties being broken arbitrarily.

When a vertex becomes locally inconsistent due to edge cost changes D* Lite attempts to eagerly update $g(v)$ values to make all vertices locally consistent. During the update, the algorithm propagates changes in $g(v)$ to all neighbors of v until a new shortest path has been found; it will not update a vertex if it remains consistent from the previous round. In some variations, for example delayed D* [5], it will delay updating

underconsistent vertices since, intuitively, it is more likely that the shortest path traverses overconsistent vertices.

The improvement to D* Lite that is proposed here is motivated by the observation that in many replanning problems there are typically more than one shortest path from v_c to v_g. The improvement is to find one of the *alternative* shortest paths that is not affected by the terrain change which invoked the replanning step, if one exists. If the cost of an alternative shortest path is no greater that the current shortest path, a switch to the new path may be made without recomputing any values of g. Also, we will show that if the new path is shorter, only part of the changes need to be propagated.

In Section 2 an overview of the proposed modification to D* Lite will be introduced and pseudo code presented. In Section 3 an example will be discussed. In Section 4 the results of some experiments on random benchmarks will be shown. Section 5 contains an analysis and some theoretical results.

2 Improved D* Lite

This section contains the motivation for and description of an algorithm we call ID* Lite which is short for Improved D* Lite. ID* Lite follows D* Lite in searching from goal to start and in selecting the next edge to traverse. ID* Lite also uses rhs, and g as D* Lite does and it maintains a priority queue as D* Lite does. As in the case of D* Lite, ID* Lite traverses a current shortest path to v_g until an inconsistency is detected. However, in computing a new shortest path, ID* Lite may only calculate part of the changes or even skip all the recalculation. The conditions that allow recalculations to be skipped will be given later along with a proof that they do not prevent ID* Lite from finding a shortest path to v_g. Upon completion of the algorithm it may be the case that at least some skipped $g(v)$ were never recomputed. Because of this, ID* Lite can potentially outperform D* Lite, especially if there are many shortest paths to v_g, or the new path will not be longer.

Some definitions are needed prior to discussing ID* Lite. In $Succ(v)$, all the vertices v' whose $g(v') + c(v, v') = rhs(v)$ are called *children* of v, and v will be called v''s $parent$. I.e., on vertex v, robot can choose arbitrary vertex in *children* of it for next move. A *type* in the form of a number will be assigned to every vertex during execution of ID* Lite as follows:

-3: The vertex is temporarily unavailable because it is not locally consistent and caused by the changes. This is also called *inconsistent source*.

-2: The vertex is temporarily unavailable, but caused by *inconsistent source*. Because the shortest path between it and v_g must cross a type -3 vertex.

-1: The vertex has never been searched. I.e., never appears in priority queue before.

0: The vertex has been visited but is not in the current shortest path.

≥ 1: The vertex has been visited, is in the current shortest path, and the type number is the number of children whose type value is not -3 and -2 it has.

The *length* of a path is the number of edges it contains. The *distance* between two vertices is the minimum length of paths connecting those two vertices. All vertices that are no further than distance *sensor-radius* away from v_c are said to be in the *view* of v_c and those that are exactly distance *sensor-radius* away from v_c are said to be *fringe* vertices. Complete path information is always known within the view of v_c. The cost of the current shortest path will be designated as Ω.

The goal of ID* Lite, when an inconsistency is discovered at a vertex w, is to consider replacing the current shortest path with an alternative shortest path from v_c to v_g which passes through some vertex u on the path from v_c to w with priority in increasing order of distance from u to w. If there exists such a path with cost less than Ω, that path will replace the current shortest path. Otherwise, if the current shortest path is unaffected by the inconsistency, it will remain the shortest path into the next round. If neither of the above applies a new shortest path will have to be computed in the same manner that a new shortest path is computed by D* Lite. In other words, there are two cases where a new shortest path is partially and fully recomputed and vertex information is recomputed to be made consistent respectively: 1) when a path to v_g that is shorter that the current shortest path is found; and 2) when all shortest paths from brothers that are successors to v_c are affected by the inconsistency.

Reduced processing time for ID* Lite depends on the ability to find the shorter path and unaffected alternative shortest paths; one of the new shortest paths will be found if there are more than one. The shorter path can be found by partially computing where only the potential vertices which may lead to a better solution will be processed. Alternative paths can be located with a simple and efficient test and, if determined to exist, they can be efficiently computed by traversing a chain of vertices according to vertex type, possibly changing the type of some vertices during the traversal. The test is merely to determine whether $c_e(w, u)$ has changed for some w and u vertices in the view. There may be several such changes on a round and all may be taken into account when looking for a shortest path to v_g. This is different than for D* Lite and its variants: they will always eagerly recompute g and rhs to remove inconsistencies and then compute a new shortest path based on the new values. If ID* Lite is not forced to recompute g and rhs values it will not do; that presents the opportunity to seek and investigate alternative shortest paths. If recompute cannot be avoided, similar to delayed D*lite [5], ID*lite will try to only update part of the changes. But unlike delayed D*lite, the need to test whether to recompute more than once in every round to guarantee optimality is avoided by ID*lite.

An outline of the action of ID* Lite is displayed in Figure 1. ID* Lite uses the variables and functions of D* Lite but some have been modified slightly to support vertex types. For ex-

```
01) bool get-alternative(vertex p)
02)   vertex R = p.
03)   while(R ≠ v_g)
04)     update R's type value.
05)     if( type(R) > 0) R = one child y of R
06)     else if( type(R) = 0 )
07)       type(R) = -2.
08)       if( R = v_c) return FALSE.
09)       R = parent(R).
10)   return TRUE.

11) bool mini-compute()
12)   while (U.TopKey() < key(v_c))
13)     u = U.Top(), k_old=U.TopKey(), k_new=CalculateKey(u).
14)     if(k_old < k_new) U.Update(u,k_new).
15)     else if(g(u) > rhs(u))
16)       g(u) = rhs(u).
17)       U.Remove(u).
18)       for all s ∈ Pred(u)
19)         if( s ≠ v_g ) rhs(s) = min(rhs(s), c(s,u)+g(u)).
20)         if(s ∈ catch) catch.Remove(s).
21)         UpdateVertex(s).
22)     else U.Remove(u).

23) getbackvertex(vertex p)
24)   if(p ≠ NULL and type(p) < 0)
25)     if(rhs(p) ≠ g(p))
26)       return;
27)     type(p) = 0;
28)     p=parent(p);
29)     getbackvertex(p);

30)   process-changes()
31)   boolean better=FALSE, recompute = FALSE.
32)   For every observed edge ⟨u, v⟩ where
33)   c_e(u, v) has changed since the previous round:
34)     Update u's rhs value.
35)     if(type(u) = −3) getbackvertex(u).
36)     if( rhs(u) = g(u)) type(u)=0.
37)     else
38)       If h(v_c, u) + rhs(u) < Ω,
39)         better = TRUE, UpdateVertex(u).
40)       else catch.add(u), type(u)=-3.
41)   if (better = TRUE) mini-compute().
42)   recompute=!get-alternative(v_c).
43)   if recompute = TRUE
44)     move type ≠ 0 vertices in catch to U
45)     set all ≠ 0, −1 type value to be 0.
46)     compute shortest path as in D*lite.

47)   move( )
48)   Set Array catch=∅, and all type values to be −1;
49)   Initialize and compute as D* Lite does at beginning.
50)   while v_c ≠ v_g
51)     if (EdgesCostChanged()) process-changes().
52)     type(v_c) = 0, v_c = one type > 0 child of v_c.
```

Figure 1: Main functions of ID* Lite

ample, every time a vertex is added into priority queue (U), its type value will be set as 0, the other modifications of type value have been shown in outline. The reader is referred to [4] for a description of those functions.

The function that determines what happens on a round is **process-changes**. For every changed edge $e =< u, v >$, rhs value of vertex u will be updated. In line 35, if a vertex had been changed before, function **getbackvertex** will be called. We will explain this function later. In line 38, if a change may cause a shorter path, line 39 will executed, or this change will be temporarily stored in *catch*. Lines 41 and 42 will find a path if the length of the new shortest path is less than or equal to the length of the old one. If it fails, the last three lines of **process-changes** are invoked to perform a D* Lite recomputation. Function **getbackvertex** is used to get back the vertices which are abandoned by a previous change. **mini-compute** will only propagate the vertices for which $rhs < g$ (over-consistent), so only better solutions can be found in function **mini-compute**; if there is vertices with $rhs > g$ (underconsistent), it must has been catched, so we can delete it directly. **get-alternative** will find a path from p to v_g if and only if there exists one. The main function is **move** which is invoked one time, at startup. Its operation is similar to that of the **move** function of D* Lite except that on every round it calls **process-changes** to try to avoid recomputation of g and rhs values.

As a note, in line 05, when a child of R is chosen, it is better to get the one with $type > 0$ if possible. This way, if the old shortest path can be still used, it will be found with priority. Upon termination of **process-changes**, only part of vertices in *catch* need to be transferred to U with the order of increasing key value. More vertices in *catch* will be moved into U if and only if no shortest path has been found. It follows that the information in *catch* and U should be synchronized. However, for simplicity we chose not to do this in running our experiments. Also because we are investigating a new method which can even skip *catch*. For space limit, this wont be discussed here.

3 An example

Figure 2 presents a simple example of grid world that shows how ID* Lite can avoid recomputation that is required by D* Lite and delayed D*. It is a 3×5 grid-world where shaded squares are impassable obstacles. A vertex is a square and is identified by its row and column position; the vertex associated with the i^{th} row and j^{th} column is referred to as $v_{i,j}$. It is 4-directional. The cost of each edge connecting two unshaded squares is 1. The start vertex is $v_{1,0}$ and v_g is $v_{1,4}$. The sensor-radius is 1. Squares may become shaded or unshaded at any time during movement. The heuristic function is $h(w, u) = 0$. The left of it shows (g, rhs) values calculated as in D*lite, a '–' symbol denotes ∞; there are several shortest path choices with cost $\Omega = 5$, consider that we use the one marked by doted line. Then the type values in ID*lite are shown as the right of Figure 2.

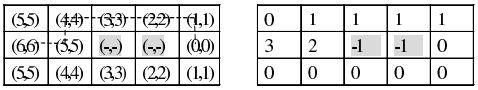

Figure 2: 3×5 4-directional Grid World

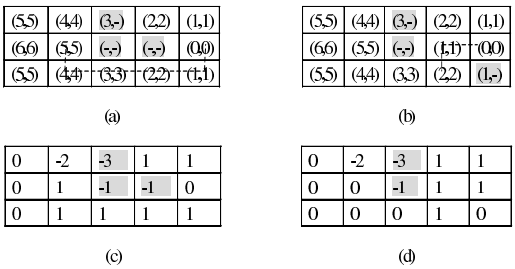

Figure 3: Example of ID* Lite Execution

Now the robot moves from $(1,0)$ to $(1,1)$, and $(0,2)$ is found to be blocked as shown in Figure 3(a). After updating rhs of those vertices, $h((1,1),(0,2))+rhs((0,2)) = 0+\infty > \Omega$, so **mini-compute** is not needed. In **get-alternative**, going from $(1,1)$ to $(0,1)$, because of the blockage of $(0,2)$, $(0,1)$'s type value is 0, so its type value is updated to be -2 and the algorithm backtracks to $(1,1)$. Then $(1,1)$'s type value is updated to be 1 now, and the only child is $(2,1)$. The new shortest path is shown by the dotted line in Figure 3(a). Figure 3(c) is the type graph corresponding to Figure 3(a). In D*lite, recomputation is needed: both $(0,2),(0,1)$ will be inserted into priority queue to propagate the change. Here, if the robot is still on $(1,1)$ and finds $(0,2)$ is unblocked, **getbackvertex** will be used: vertices $(0,1)$ and $(0,2)$ will be set available again.

Now, considering robot is on vertex $(2,3)$, two changes are observed: $(1,3)$ is unblocked and $(2,4)$ is blocked as shown in Figure 3(b) with the old path weight $\Omega = 2$. After updating rhs, because $h((2,3),(1,3))+rhs((1,3)) = 0+1 < \Omega$, vertex $(1,3)$ will be inserted into priority queue and **mini-compute** is triggered. Similarly, as $(0,2)$ in Figure 3(a), it is not necessary to insert $(2,4)$ into the priority queue. After execution of **mini-compute**, the (g, rhs) value will be as in Figure 3(b). Then **get-alternative** is called. The corresponding type value is shown in Figure 3(d), and the new shortest path is shown as the dotted line in Figure 3(b). In D*lite, it is necessary for both changed vertices to be inserted into the priority queue to propagate.

D* Lite would have found all the changes to rhs values that ID* Lite did. Then it would have begun a new search, updating all g and rhs values as usual. Algorithm delayed D* would have operated as D* Lite except that it would need to make an extra check to make sure it has the shortest path: i.e., there is no inconsistent vertices on its current path. This step is necessary to be sure the path found by delayed D* is shortest.

4 Experiments

In this section, the performance of ID* Lite is compared experimentally to the D* Lite and delayed D* algorithms on random grid world terrains. In each experiment the initial terrain is a blank square 8-direction grid world of $size^2$ vertices, where v_s and v_g are chosen randomly. Several other parameters are used: 1. *percent* is used for exactly $percent\% * size^2$ of the vertices are selected randomly and blocked; 2. *sensor-radius* is used as the maximum distance to a node that is observable from the current robot position. First, results of random rock-and-garden benchmarks is given: i.e., a blockage is found it will not move or disappear later. Then, experiments are run on a collection of benchmarks that model robot navigation through changing terrain. The results are average of more than 100 independent runs of each algorithm.

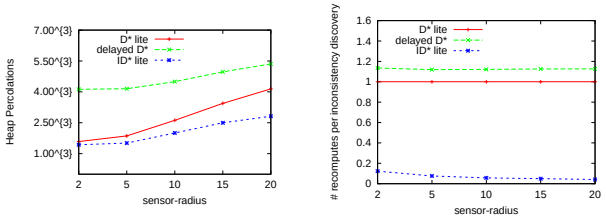

Figure 4: $size = 200$ and $percent = 30$

In Figure 4, the results in rock-and-garden benchmarks are shown: the graph on the left shows the relation between number of *heap operations* and *sensor-radius* given the other parameters. Heap operations make the most significant contribution to time complexity in such algorithms and the plots show only the heap operations in recomputation: i.e., they do not count the number of operations used when initializing a shortest path from v_s to v_g, because all of the family of algorithms discussed here do the same as the A* algorithm in this phase. On the right side of the figure is a graph showing the ratio of the number of recomputations to the number of changes observed. The D*lite, curve is flat at 1 because every time an inconsistency is observed, exactly one recomputation must be performed. The delayed D* curve is always above 1 because at least one recomputation must be performed for every round in order to guarantee local optimality [5]. The curve for ID*lite stays much below 1 since recomputations are skipped when alternatives are found. We note that the numbers plotted in the figure include calls to mini-compute.

To some extent, the graph explains why ID*lite can outperform the other algorithms. The right graph of Figure 4 shows that ID* lite can save almost 90% of the recomputations that would be done by D* Lite. However, this does not mean that a corresponding savings applies for heap operations since changes are transferred from catch to the priority queue every time a full recomputation occurs.

Since there are less recomputations, more changes are processed in each recomputation and many changes with big key values are not propagated. The more vertices affected by such changes, the more heap operations can be saved. Generally

speaking, decreasing changes can affect more vertices than increasing changes. In rock-and-garden benchmarks there are only increasing changes and in the benchmarks below there are many decreasing changes.

In [5] significant delayed D* performance advantages are reported. In particular, delayed D* performs well when there are only a few decreasing changes that cause overconsistent vertices. But in rock-and-garden benchmarks, there are no decreasing changes so the performance of delayed D* is not as good as the other algorithms. Delayed D* performance also suffers in the terrain changing benchmarks since half of the inconsistent vertices are overconsistent. This is especially true when robot movement is blocked. Since showing the performance results of delay D* would force a change of scale of the plots, we do not show them.

The second set of benchmarks is intended to model robot navigating in terrain changes, for example if a robot is moving in a parking lot along other vehicles. In these benchmarks a fixed percentage of vertices are initially blocked and, on succeeding rounds, each of the blocked vertices moves to some adjacent vertex with probability 0.5, the particular target vertex being chosen randomly from all available adjacent vertices. The experiments are done in the same way as the rock-and-garden experiments except we also plot, in Figure 7, the effect of changing the percentage of blocked vertices for fixed sensor-radius.

Figures 5 to 6, left, show that ID* Lite uses fewer heap operations to compute a path from v_s to v_g. In the right plot of Figure 6 the ratio of the number of recalculations to the number of changes tends to 1 as sensor-radius is increased. But we note that significantly many recomputations are calls to *mini-compute* which complete faster than a full recomputation. From Figure 7, we can see that the number of recomputations done in ID*Lite remains below that of D*Lite for a wide range of blocking percentages. We conclude that for these benchmarks ID*lite has a better ability to handle intensely changing environments. Because ID* lite can skip recomputations and then update all the changes at one time, it is more efficient to update changes per round.

In the case of the terrain changing benchmarks, although all three algorithms find and traverse optimal cost paths in every round, they can find different final paths if they use different ways to break ties when there is more than one child to consider. Because ID* Lite exploits alternative shortest paths which try to avoid the area with more intense changes, ID* lite has a better ability to avoid crashes with sliding obstacles.

5 Analysis and theoretical results

In this section it is shown that on every round, given a current shortest path from v_c to v_g, ID* Lite computes a new shortest path, if one exists: that is, a path whose cost is the minimum over all paths P from v_c to v_g of the sum of costs c_e of edges in P. It is assumed that the cost of any edge with at most one endpoint in the view does not conflict with the g values of any of its endpoints that are not in the view. Finally, it

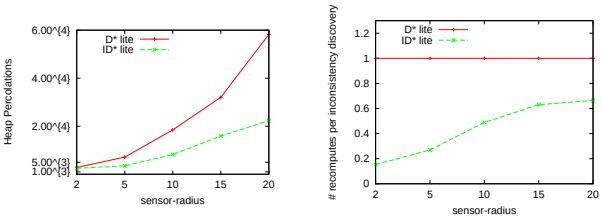

Figure 5: $size = 200$ and percent vertices blocked = 30

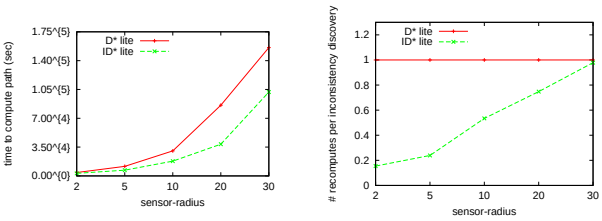

Figure 6: $size = 300$ and percent vertices blocked = 30

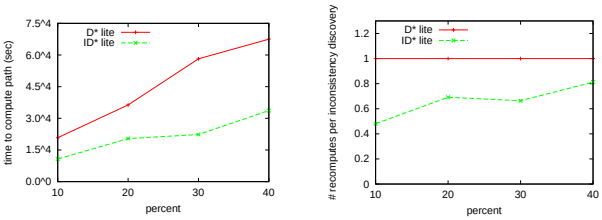

Figure 7: $size = 200$ and sensor-radius = 20

is assumed that the heuristic function $h(w, u)$ is the same as that of D* Lite and is therefore always a lower bound on the minimum cost path from w to u using c_e costs and is such that the triangle inequality holds. We will use $e = \langle w, u\rangle$ to denote an edge in **process-changes** of ID* Lite that is in the view, whose cost $c_e(w, u)$ has changed to be $c'_e(w, u)$ since the previous round.

Lemma 5.1 *If e is greater than it was in the previous round, then any path passing through e without decreased edges has a cost that is greater than Ω which is the cost of previous round shortest path.*

proof 1 *Assume there is a path p of cost less than or equal to Ω and passing through e without a decreased edge. It is straightforward to see that the cost of p in the previous round is less than Ω. This contradicts the hypothesis that Ω is the shortest path of the previous round.* □

Lemma 5.2 *$e = \langle w, u\rangle$'s cost has been decreased. If after updating rhs of w, u, $h(v_c, w) + rhs(w)$ is no less than Ω of previous round, then any path passing through e, without other decreased edges after e along the path, has a cost that is not less than Ω.*

proof 2 *Suppose an arbitrary path p passing through e, then the cost of p, η, has the property: $\eta \geq h(v_c, w) + c'_e(w, u) + g(u)$. By definition of rhs, we have $\eta \geq h(v_c, w) + rhs(w)$. I.e., $\eta \geq h(v_c, w) + rhs(w) \geq \Omega$.* □

Lemma 5.3 *If in $e = \langle w, u\rangle$, w's type value is -1, then any path passing through e without other decreased edges after e along that path has a cost that is not less than Ω which is the cost of previous round shortest path.*

proof 3 *If w's type value is -1, i.e., w has not been inserted into the priority queue, then $key(u) \geq \Omega$. Therefore, any path p passing through e will have cost $\eta \geq h(v_c, w) + c'_e(w, u) + g(u) \geq h(v_c, u) + g(u) \geq \Omega$.* □

Theorem 5.4 *Only the e, whose cost is less than it was on the previous round and after updating rhs has $h(v_c, w) + rhs(w) < \Omega$ may cause a shorter path then the shortest path in previous round.*

proof 4 *By Lemma 5.1, 5.2 and 5.3, we can see that the only non-trivial condition needed to be proved is whether one path p having more than one decreased edge may be shorter. Without loss of generality, supposing p has $e = \langle w, u\rangle$ as the last decreased edge with the direction from v_c to v_g. Then e satisfies the precondition of 5.2, i.e., $p \geq \Omega$.* □

Lemma 5.5 *In ID* lite, only vertices which may cause shorter path than the shortest path in previous round will be propagated in function* mini-compute.

proof 5 *By theorem 5.4 and line $38, 39, 41$ in Figure 1, this lemma can be proved straightforwardly.* □

Lemma 5.6 *Function* get-alternative *returns $TRUE$ if and only if a shortest consistent path from v_c to v_g is found.*

proof 6 get-alternative *returns $TRUE$ if and only if a path from v_c to v_g has been found. By the way the next step is chosen, no abandoned (i.e., inconsistent) vertices will be chosen, so the path must be one of the shortest consistent paths.* □

Theorem 5.7 *Function* **process-changes** *will find the shortest path in every round if and only if there exists one.*

proof 7 *If the new shortest path is shorter than the one in the previous round then, by Lemma 5.5, all the shorter paths will be found by function* mini-compute *and will be updated to be consistent. If the new shortest path is as long as the one in the previous round, then the shortest path in the previous round is not affected by changes and will still be consistent. By Lemma 5.6, all shortest paths can be found correctly.*

*Therefore, lines 43 to 46 in Figure 1 will be executed if and only if the new shortest path is greater than the one in previous round. In this case ID*Lite behaves like D* lite so the correctness of the new shortest path follows from the correctness of D* lite.* □

Lemma 5.8 *The changes which have been skipped in a round when the shortest path's cost is Ω, will not affect the result anymore unless the new shortest path is greater than Ω.*

proof 8 *For a change of $e = \langle w, u\rangle$, if it was skipped, then the low bound η of any path passing through it has $\eta \geq \Omega$. If the new shortest path's length Ω' is not greater than Ω, then we have $\eta \geq \Omega'$. So it will not affect the new shortest path.* □

Theorem 5.9 *ID* Lite finds a shortest path from v_c to v_g on a round, if and only if one exists.*
proof 9 *Follows directly from theorem 5.7 and Lemma 5.8.* □

The following theorem explains, in part, the relative efficiency of searching for alternative shortest paths.

Proposition 5.10 *After a full recomputation, all the new shortest paths will be found.*
proof 10 *Since ID* Lite uses the same data structures as D* Lite, if one child of a vertex has been updated to be consistent, then all the children of it will be updated to be consistent. So if one shortest has been found, supposing an arbitrary path p from v_c to v_g is also shortest, then the child of v_c on p must be updated and consistent too. Iteratively, all the vertices on p are updated and consistent. I.e., it has been found.*

6 Conclusion

A modification to the D* Lite algorithm for planning has been introduced and packaged as an algorithm called ID* Lite. The modification is to update vertices as less as possible and to seek alternative shortest paths when inconsistencies are discovered, rather than recompute to remove all inconsistencies before finding a new shortest path. It is shown that the modification results in far better performance on random grid problems. The modifications proposed for D* Lite can coexist with other D* Lite variants such as delayed D* Lite easily.

References

[1] Anthony Stentz,"Optimal and Efficient Path Planning for Partially-Known Environments," *IEEE International Conference on Robotics and Automation*, San Diego,CA, pp. 3310–3317, 5/94.

[2] Anthony Stentz, "The Focussed D* Algorithm for Real-Time Replanning," *Proceedings of the International Joint Conference on Artificial Intelligence*, Montral, Qubec, Canada, pp. 1652–1659, 8/95.

[3] Sven Koenig, Maxim Likhachev, "Improved Fast Replanning for Robot Navigation in Unknown Terrain," *IEEE International Conference on Robotics and Automation*, Washington DC, pp. 968–975, 8/02.

[4] Sven Koenig, Maxim Likhachev, "D*lite," *Eighteenth national conference on Artificial intelligence*,Menlo Park, CA, USA, pp. 476–483, 2002.

[5] Dave Ferguson, Anthony Stentz, "The Delayed D* Algorithm for Efficient Path Replanning," *IEEE International Conference on Robotics and Automation*, Washington DC, pp. 968–975, 4/05.