A New Way to Reduce Computing in Navigation Algorithm

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 one of the most successful. 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, or only a partial recalculation is needed and an efficient test for determining these 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. Besides, a novel realistic benchmark is described.**

Keywords: robot navigation, uncertainty, planning, incremental

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.

It is unlikely that any navigation algorithm such as D* Lite will find the global 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. Moreover, current edge costs are known 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 some algorithms such as D* Lite assuming can always find the lowest cost sequence from v_c to v_g based on the known edge costs within the view 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 $c_e(w, u) = c(w, u)$ in which $c(w, u)$ is the observed cost by robot in v_c, otherwise $c_e(w, u)$ is assumed and does not change from round to round. When a shortest path is found, the algorithm moves the robot along that path, one transition per round, until the edge cost assumption is violated with observations within the view of v_c, i.e., some changes have been found. 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 returns 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

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

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

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 is 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. This work was partially published in [6], here is an extension of it.

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 as D* and 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 in every round. 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. When changes are found, D* Lite will need to recompute, this will be called "one round". Although in ID* Lite recomputing may be skipped, it will still be called "one round". 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 an arbitrary vertex from *children* for next move. The vertices in *children* are *siblings* of each other. 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 due to the changes. This is also called *inconsistent source*.

-2: The vertex is temporarily unavailable, due to *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., has not appeared previously in the priority queue.

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 its children whose type value is not -3 or -2.

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.

For convenience, from now on, e will be used to represent one edge and also its weight. No explanation will be commented unless there is a confusion. w, u are used for vertices and $e = (w, u)$ means the edge from w to u. Sometimes, $c_e(w, u)$ or $c(w, u)$ will be used to represent the weight of $e = (w, u)$. $c'_e(w, u)$ or $c'(w, u)$ means the same thing but just after updating: i.e, the symbol attached with $'$ represents that it shows the newest information. Use the same format on a path p, then p' means the same route with p but after propagating changes in the current round. Also, Ω will be used for the value of one optimal solution before some changes have been found. Ω' will be used for the optimal-solution-value in the current round. Similarly, if e has been changed, then e' will be used for the new edge.

The explanation below is in order to give a rough idea of ID* Lite . The goal of ID* Lite, when an inconsistency is discovered at a vertex w, is to consider replacing the current shortest path with an new 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, the first one found will replace the current shortest path. In this case only those changes which have potentialities to cause such a path will be propagated. "Potentiality of a change" here means that it is possible to generate one shortest path passing through this "change". Otherwise, if the current shortest path or one of its alternatives 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 old shortest paths from v_c to v_g 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 so; that presents the opportunity to seek and investigate alternative shortest paths. If recomputation 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 example, every time a vertex is added into

```
01) bool get-alternative(vertex p)
02)   vertex r = p.
03)   while(≠ 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

the 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 be executed, otherwise 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 to find such a path in previous lines, the last three lines of *process-changes* are

invoked to perform a D* Lite style recomputation. From here, we can see that function *process-changes* works like a distributor to drive other functions to work. Function *getbackvertex* is used to get back the vertices which are abandoned by previous changes. Notice that one edge may be changed more than one time among rounds. In D* Lite, once one edge has been changed the first time, it will be reinserted into priority-queue for updating to be consistent, but in ID* Lite, one change on an edge may be skipped and marked by $type = -3$. When that edge is changed one more time, it is possible that that edge can be used again. But when it becomes unavailable, it may cause many other vertices to become unavailable and marked by $type = -2$ in function *get-alternative*. Such vertices will not be involved further in the recomputation of a round but when necessary a call to *getbackvertex* after the round is over may make them all available once again. *mini-compute* will only propagate the vertices for which $rhs < g$ (overconsistent) and satisfy the condition in *line 12*, so only better solutions can be found in function *mini-compute*; if there is vertex with $rhs > g$ (underconsistent), it must be *catched*, so we can delete it directly, the reason to do this and its benefits will be discussed in Section 5. Here notice that only part of $rhs < g$ (overconsistent) vertices will be propagated. The condition in *line 12* is very important. It makes sure that this function will be terminated when it is impossible to get a better solution than the old one. In delayed D* [5], there is a function *computeShortestPathDelayed* where all $rhs < g$ (overconsistent) vertices are propagated at first. Its looping condition is also different from ours by appending with "*or* $g(v_c) \neq rhs(v_c)$". This can cause a problem in some seldom occurring condition in delayed D*. Now assuming $rhs(v_c) > g(v_c)$ due to an increased change around v_c, then that looping condition surely is satisfied. If that change can be updated, it will make v_c to be consistent to get out of the loop. But remember that delayed D* will try to propagate $rhs < g$ (overconsistent) vertices first, that means the increased change can not get processed when function *computeShortestPathDelayed* is called the first time so this function can not terminate until the priority queue is empty. This problem can be solved by inserting such kind of increased changes which can affect v_c's values directly into the priority queue as decreased changes when function *computeShortestPathDelayed* is called first time.

get-alternative will find a path from p to v_g if and only if there exists one. The rough idea is: all the solutions construct a tree rooted at v_c, so from v_c a depth-first search can find one solution if and only if there exists one. It is straightforward to see the complexity of this method is linear in the number of all vertices on shortest paths. In *line 04*, it is asked to update a vertex $r's$ type value, this will be done according to the definition at the beginning of Section 2. So if one vertex can be counted into a positive number for $type(r)$, more than consistence is required. 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 first priority, i.e. the old shortest path is searched firstly than the others. Upon termination of *process-changes*, only some 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 but to transfer all of them one time. Currently we are looking for a method to skip memorizing vertices as is done in *catch*. For space limit, this won't be discussed here.

3 An example

Figure 2 presents a simple example of grid-world that shows how ID* Lite works. 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 diagram on the left 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 the dotted line. Then the type values in ID* Lite are shown as the right of Figure 2.

(5,5)	(4,4)	(3,3)	(2,2)	(1,1)
(6,6)	(5,5)	(-,-)	(-,-)	(0,0)
(5,5)	(4,4)	(3,3)	(2,2)	(1,1)

0	1	1	1	1
3	2	-1	-1	0
0	0	0	0	0

Figure 2: 3×5 4-directional Grid World

(5,5)	(4,4)	(3,-)	(2,2)	(1,1)
(6,6)	(5,5)	(-,-)	(-,-)	(0,0)
(5,5)	(4,4)	(3,3)	(2,2)	(1,1)

(a)

(5,5)	(4,4)	(3,-)	(2,2)	(1,1)
(6,6)	(5,5)	(-,-)	(1,1)	(0,0)
(5,5)	(4,4)	(3,3)	(2,2)	(1,-)

(b)

0	-2	-3	1	1
0	1	-1	-1	0
0	1	1	1	1

(c)

0	-2	-3	1	1
0	0	-1	1	1
0	0	0	1	0

(d)

Figure 3: Example of ID* Lite Execution

Suppose 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 *getalternative*, going from $(1,1)$ to $(0,1)$, because of the blockage of $(0,2)$, $(0,1)$'s type value is 0, 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, 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 the 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 and vertices $(0,1)$ and $(0,2)$ will be set available again.

Now, consider the 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 the 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 *getalternative* 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 propagate decreased changes first. But after that it needs to make an extra check to make sure it has the shortest path: i.e., there are no inconsistent vertices on its current path. This step is necessary to be sure the path found by delayed D* is shortest. For details please refer to [5].

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.

In Figure 4 and Figure 5, the results in rock-and-garden benchmarks are shown. In Figure 4, 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, but because delayed D* may finish one round with less time that it can still perform better than D* Lite in many conditions [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*. In Figure 5, the graphs show the same meanings as corresponding graphs in Figure 4 but with *sensor-radius* fixed and different *blockage percent*.

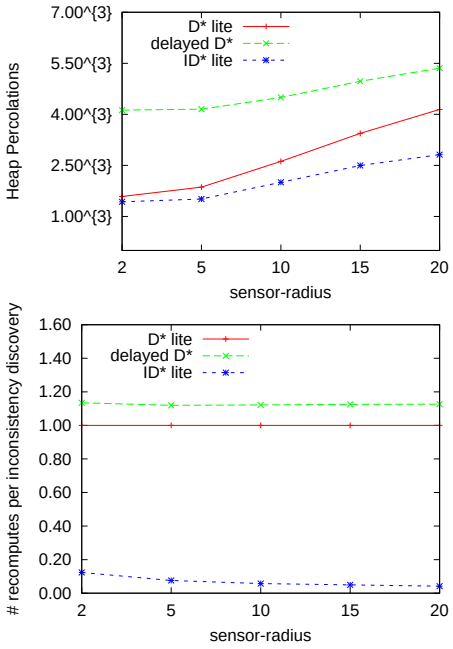

Figure 4: $size = 200$ and $percent = 30$

To some extent, the graph explains why ID* Lite can outperform the other algorithms. The right graph of Figure 4 and Figure 5 show 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

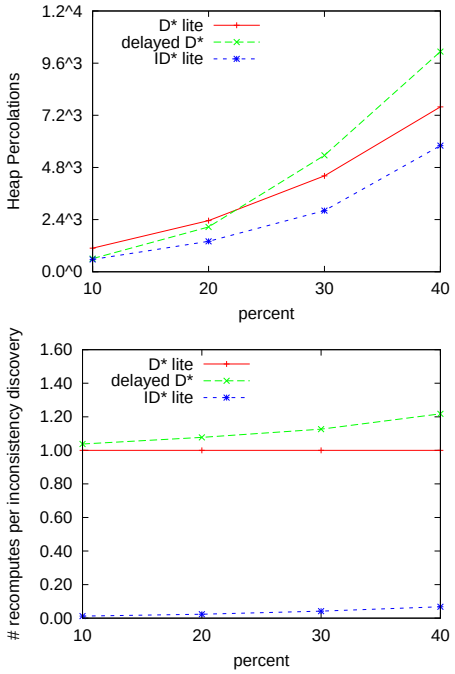

Figure 5: $size = 200$ and $radius = 20$

a full recomputation occurs. To reduce the transferring will be our next step to improve ID* Lite in the future. 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. From the strategy used by delayed D*, it performs good when there are not too many increased changes. In particular, delayed D* can perform quite well when there are only a few decreasing changes that cause overconsistent vertices. In rock-and-garden benchmarks, there are no decreasing changes so the performance of delayed D* is not as good as the other algorithms. From Figure 5, we can see that, when there are not too many increased changes, delayed D* performs better than D* Lite. This gives us a hint in what kind of environment delayed D* can be better than D* Lite. But when the blockage *percent* increases which means there will be more increased changes, delayed D* performs worse. Delayed D* performance also suffers in the terrain changing benchmarks, that will be introduced below, 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 8, the effect of changing the percentage of blocked vertices for fixed sensor-radius.

Figures 6 to 7, 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 7 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 8, 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. This is deserved to be searched more to combine domain knowledge to get a better global path.

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. In D* or D* Lite, changes are all considered again and so lead to updating of values of affected vertices. In delayed D* [5], some kind of changes will be updated first then others. This is more like a kind of heuristic strategy. In this section, we will show how it is possible to get deterministic improvement with ID* Lite. It is assumed that the cost of any edge with at most one endpoint in the view has values which are still consistent. Finally, it 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.

$e = \langle w, u \rangle$ is used to denote an edge in *process-changes* of ID* Lite that is in the view, whose cost $c_e(w, u)$ has

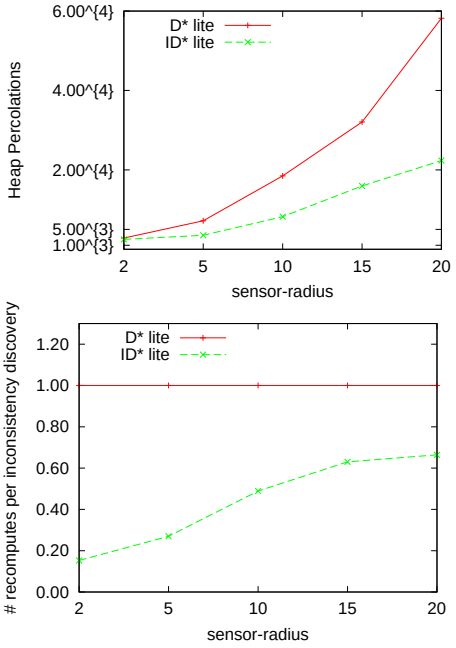

Figure 6: $size = 200$ and percent vertices blocked $= 30$

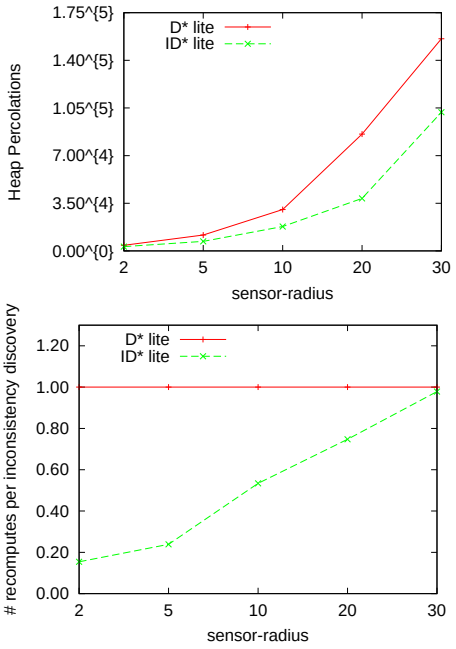

Figure 7: $size = 300$ and percent vertices blocked $= 30$

changed to be $c'_e(w,u)$ since the previous round. For convenience, from now on, e will be used to represent one edge and also its weight. w, u are used for vertices and $e = (w,u)$ means one directed edge from w to u as noted by $e = \langle w,u\rangle$. Sometimes, $c_e(w,u)$ or $c(w,u)$ will be used to represent the weight of $e = (w,u)$. $c'_e(w,u)$ or $c'(w,u)$ means the same thing but just after changing. Use the same format on a path p, p' means the same route but in the current round. Also, Ω will be used for the value of one optimal solution before some changes have been found. Ω' will be used the the optimal-solution-value in the current round. If e has been changed, then e' will

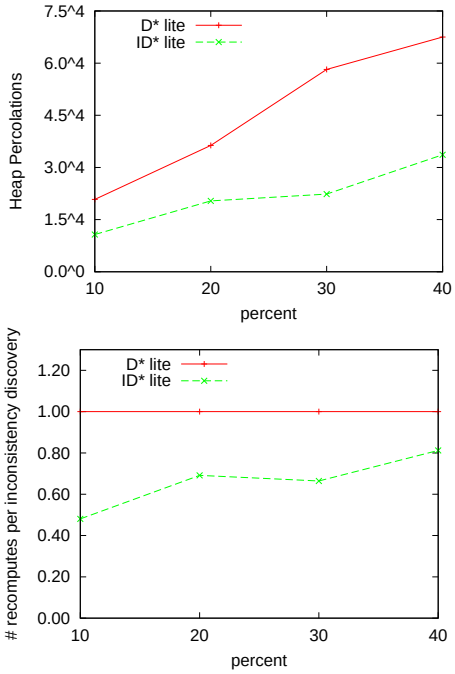

Figure 8: $size = 200$ and sensor-radius $= 20$

be used for the new edge or its weight: i.e., $'$ represents newer information. Given a change of an edge-weight in the graph, it can be an increased change or a decreased change, i.e. $e' > e$ or $e' < e$ respectively.

Observation 5.1 *If one change is an increased change, i.e. $e' > e$, then any path passing through e' without decreased edges has a cost that is greater than Ω, 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.* □

Although Observation 5.1 is obvious, it shows that we can divide changes by how they can infect solutions. By comparing with the previous round, there are three kinds of solutions: better($\Omega' < \Omega$), equal($\Omega' = \Omega$) and worse($\Omega' > \Omega$). From the statements above, it may be guessed that the increased changes will cause worse results and decreased changes will cause better results. Then the new way to divide changes will be the same as the method by change of weight-value. The observation below will show that this is not the exact situation.

Observation 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, $'\eta'$, has the property: $\eta \geq h(v_c, w) + c'_e(w, u) + g(u)$. By definition of rhs, we have $\eta \geq h(v_c, w) + rhs(w)$. Surely, the later is bigger than Ω. I.e., $\eta \geq h(v_c, w) + rhs(w) \geq \Omega$.* □

In Observation 5.2, functions h and rhs have the same meaning as in D* Lite. This observation shows that some kind of decreased changes can not take a better solution and can be determined before recomputing. Until now, we may notice that all the statements above require that "without other decreased edges...". But with the results below, we can see that such conditions can be skipped safely.

Lemma 5.1 *In the current round, after propagating changes(*i.e. recomputing*), if there are two changed edges $e'(w, u)$ and $z'(v, r)$ on path p' which is the shortest path passing through e' and z', and z' is after e' along path p', then $(h(v_c, w) + rhs(w)) \geq (h(v_c, v) + rhs(v))$.*

proof 3 *If p' is the shortest path passing through e' and z' and z' is after e' along path p', then $(h(v_c, w) + rhs(w)) \geq (h(v_c, w) + h(w, v) + rhs(v)) \geq (h(v_c, v) + rhs(v))$ by property of consistent heuristic function.* □

Lemma 5.2 *In the current round, if there is a path which is the shortest one passing through a set of changes and $e'(w, u)$ is the last one along the path before propagating changes, $h(v_c, w)+rhs(w)$ is correct, i.e. the same as after propagating changes.*

proof 4 *If the value of $h(v_c, w) + rhs(w)$ is not correct, that means $rhs(w)$ will be affected by some changes z', then the shortest path passing through this set of changes must include z' too and z' will be after e' along the path. This is a contradiction.* □

corollary 1 *Suppose, after propagating changes, there is a path p' passing through a set of changes with the last change at edge $e'(w, u)$ along p'. Then, the cost value of path p' is $< \Omega$ if and only if before propagating changes $(h(v_c, w) + rhs(w)) < \Omega$.*

proof 5 *This comes straightforwardly from Observation 5.2, Lemma 5.1 and Lemma 5.2.* □

Lemma 5.3 *In the current round, if after propagating changes, there is one path p' passing through a set of changes and the last change is $e'(w, u)$ along p', then before propagating changes, only e' needs to be reinserted for propagation in order to get path p'.*

proof 6 *Without losing generality, assume that $z'(v, r)$ is the change along p' before e', then if e' is reinserted and propagated, $rhs(v)$ must be affected by e'. I.e., z' will be reinserted and propagated too. Similarly, all the changes on path p' will be processed. Straightforwardly, path p' can be generated.* □

corollary 2 *In the current round, if there will be one better path p' than the old one, then it can be obtained by propagating the change $e'(w, u)$ which is the last change along a path p'. The cost of the path satisfies $(h(v_c, w) + rhs(w)) < \Omega$.*

proof 7 *This comes straightforwardly from Corollary 1 and Lemma 5.3.* □

The conclusion of Corollary 2 *looks good but can not be executed conveniently*, because it is hard to say whether one change is the last one along a path. So for implementation, such conditions need to be broadened as below.

Proposition 5.4 *If there are paths which are better than the old one, they can be computed by propagating the changes such as $e'(w, u)$ with $(h(v_c, w) + rhs(w)) < \Omega$.*

proof 8 *This comes straightforwardly from Corollary 2.*

Proposition 5.4 differs from Corollary 2 in two ways: 1) "last change" is not required anymore; 2) this one is not obvious: in Lemma 5.2, the equation in it will be correct when the change is the last one. But if not, the correctness can not be guaranteed anymore. That means the change $z'(v, r)$ whose correct value $(h(v_c, w) + rhs(w)) \geq \Omega$ may be reinserted and propagated by a wrong $(h(v_c, w)+rhs(w))$ value. I.e., the vertices considered are a super-set of the vertices in Corollary 1. This is expected to be improved to find a stricter condition in the future. Proposition 5.4 also solved the question about existence of condition "without other decreased edges...", because it does not use any assumption about it at all.

Until now, the statements above are concerned with getting the optimal solution if the new solution is better than that of the previous round. But it is also possible that the new optimal solution is no better than the old one. Now we will discuss what can happen if the new solution is as good as(equal) old one. Here "equal" does not means exactly the same but equivalently good by comparison with some rules. For example, the values of weight are equivalent. So we can also get another conclusion: the old one may be not available anymore. Then in current round, the optimal solutions come in two different ways. 1) the ones in the previous round and have not been affected by changes; 2) the ones which are newly generated

in current round caused by changes, i.e., containing one or more changes. For the first kind, because they are not affected by changes, the values on them are still correct, i.e. they can be found directly. For the second kind, we can get them through a small change of previous results as below. Notice that *in Corollary 1* $\geq \Omega$ *can replace* $> \Omega$ *without loss.* To change Corollary 2 and Proposition 5.4 similarly, we can get:

corollary 3 *In the current round, if there will be a path no worse path* p' *than the old one, then it can be obtained by propagating the change* $e'(w,u)$ *which is the last change along a path* p' *and also satisfy* $(h(v_c,w) + rhs(w)) \leq \Omega$.

proof 9 *proof is similar to that for Corollary 2.*

Proposition 5.5 *If there are paths which are no worse than the old one, they can be obtained by propagating changes such as* $e'(w,u)$ *with* $(h(v_c,w) + rhs(w)) \leq \Omega$.

proof 10 *proof is similar to that for Proposition 5.4.*

In Figure 1, we can do this by replacing $\leq$ with $=$ in *line 38.* This performs better when there are many solutions of the second kind. Our experiments show that the difference is not too big. Both methods are optimal. Because if no better or equal solution is found, a whole recomputation will be ran as D* Lite. The difference is that in current code, not all the equal solution can be found. But both of them act the same on better solutions.

One might suppose that if the new solution is equal to the old one the round might be terminated successfully. But this is not the case. The conclusions above are correct and have been proved, but there may be some "fake" paths which can not be used anymore. All these "fake" paths are caused by the fact that many changes have not been propagated. For example, path p' including one unpropagated change $e'(w,u)$ can be considered to be one optimal solution. Here the change e' must be an increased change, or it will satisfy the formula in Proposition 5.5 and has been propagated. Then we can see that if e' is propagated, p' must have a bigger weight, i.e., it is not a real optimal solution. This requires us to have a way to find one correct optimal solution quickly. This seems similar to delayed D*, but there is a major difference. In delayed D*, the processing is an iteration of finding rising vertices and updating changes: once rising vertices have been found, they will be propagated then repeat checking rasing vertices again. This processing is used to guarantee the optimality of found solution. For more details refer to [5]. But here no updating will be taken until one solution has been found or no such solution can be found. That is because all the optimal solutions (if they are no worse than old one) are calculated already: then if one of them can be located, we are done. Now, recall that there are three kinds of solutions comparing with previous round: better, equal and worse. The only one which has not been discussed is "worse". If we find the current condition is "worse", then we have to update all the changes as D* Lite.

From now on, we will prove some properties of the pseudo code of ID* Lite in Figure 1. With all the results above, it will be relatively straightforward.

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

proof 11 *Function* 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.* □

In Function *get-alternative*, the rough idea is: all the solutions construct a flow with source at v_c and destination at v_g, so from v_c a depth-first style searching can find one solution if and only if there exists one. It is straightforward to see this method is very efficient.

Lemma 5.7 *Function* mini-compute *can only generate a path whose weight is better than* Ω.

proof 12 *By the condition in* line 12*, if one vertex's key value is bigger than* Ω*, then it will not be processed and the function terminates. So if one path is generated, then it must have a weight value better than* Ω.

It is necessary to notice that: if there is a vertex with $rhs > g$ (underconsistent), it will be deleted from priority queue. It must be *catched*, then we can still get it back if we need it later. From this, we can see that what Function *mini-compute* does is different from Proposition 5.4. In *mini-compute*, it is possible that some shortest path can not be found. For example, there is one shortest path p' passing through one decreased change $e' = (w,u)$ and one increased change $z' = (v,r)$ along the path. Then when z' is propagated, it may be deleted because of underconsistency, or cause termination of this function. Then, path p' can not be generated. This is the only possibility that a shortest path *can not* be found by *mini-compute.* But the last result will be correct because without propagation of z', $rhs(w)$ is less than its actual value, so the value of path p' generated by *mini-compute* is less than its actual value too. That means, in function *get-alternative*, this path will be considered unavailable and a whole recomputing will be called to get the correct results. The direct reason why to do so is

for *efficiency*. As we found that such kind of conditions seldom appear, i.e., an increased change normally causes a worse path. Only in some special conditions, where the increased value will be canceled out by a decreased change, it can appear on a shorter path. But in such a condition, probably a path passing through e' and then switching to another route around z' exists and is better than p'.

Theorem 5.8 *Function* process-changes *will find the shortest path in every round if and only if there exists one.*

proof 13 *By Proposition 5.4, Lemma 5.6 and Lemma 5.9, if in* line 43*, the* recompute = false*, a correct shortest path must have been found. Or, the last three line in Figure 1 will be executed like D* Lite, so the correctness of the new shortest path follows from the correctness of D* Lite.* □

Lemma 5.9 *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 14 *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.* □

And if the new shortest path is greater than Ω, a full recomputation as D* Lite must have been executed, that means all the *catched* vertices will be transferred into the priority queue.

Theorem 5.10 *In every round, ID* Lite will return a shortest path from v_c to v_g if and only if at least one shortest path exists.*

proof 15 *Follows directly from theorem 5.8 and Lemma 5.9.* □

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

Proposition 5.11 *After a full recomputation, all the new shortest paths will be found.*

proof 16 *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 and Next Step of Work

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 little 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.

All the observations in this paper are not related closely with any domain-knowledge in navigation, that means *it is possible these observations can be generalized into incremental algorithm.* We will formalize a general frame to do so and probably one or two applications of this frame to different areas will be tested.

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.

[6] Weiya Yue, John Franco, "Avoiding Unnecessary Calculations in Robot Navigation," *Lecture Notes in Engineering and Computer Science: Proceedings of The World Congress on Engineering and Computer Science 2009, WCECS 2009*, 20-22 October, 2009, San Francisco, USA, pp. 718–723.