ID* Lite: IMPROVED D* LITE ALGORITHM

Weiya Yue, John Franco
Computer Science Department
University of Cincinnati
Cincinnati, Ohio 45220, US
weiyayue@hotmail.com,fra nco@gauss.ececs.uc.edu

Weiwei Cao
State Key Laboratory of Information Security
Graduate School of Chinese Academy of Sciences
Beijing China 100049
weiwei.cao@hotmail.com

Hongwei Yue
Information College Zhongkai Univ. of Agriculture and Engineering; Guangdong Univ. of Technology
Guangzhou China 510090
yuehongwei420@163.com

ABSTRACT

Robot navigation has been of great importance especially under unknown or keep-changing environment. In order to solve this kind of problems, many algorithms have been brought up. D* Lite is generally considered as one of the most functional ones. The better performance of D* Lite largely depends on relatively less updating rather than recalculating terrain cost from scratch between robot movements. However, D* Lite still needs updating, i.e. recalculation, every time a terrain change is discovered. In this paper, we give an efficient method to check out when such recalculation can be fully or partially avoided. Experimental results show that it speeds up to 5 times for a variety of benchmarks including a novel and realistic benchmark. Our idea results in an improved version of D* Lite which we call ID* Lite. Moreover, it can be easily embedded into D* Lite variants such as DD* Lite and anytime D* etc.

1. INTRODUCTION

As a demand for extensional development of robot techniques, autonomous vehicles are needed to do replanning work in order to navigate under environment without human involvement, like exploring planets, gathering data and even parking by themselves. For this reason, the efficiency of replanning process is pivotal for practical use. Up to now, the marriage of incremental algorithm and sophisticated search heuristics narrows the search space by exploiting learned terrain information and therefore excels in this respect. D* Lite algorithm [3, 4], a descendant of A* and D* algorithms, represents one of the most functional algorithms of the above marriage idea. It can be efficiently implemented and its "experimental properties show that D* Lite is at least as efficient as D* [1, 2]."

D* Lite algorithm attempts to find a lowest cost path from one point to another under a unknown or keep-changing circumstances. In such circumstances, terrain information is often mathematically modeled as a directed graph $G(V, E)$ with a start vertex v_s, a goal vertex v_g, and one cost function $c : V \times V \mapsto Z^+$ on all possible edges between them. The robot's current position is designated as v_c. Since the movement of robot the old knowledge of edge costs that it is holding probably has changed and the change is unpredictable, so to efficiently determine the lowest cost sequence of transitions from v_s to v_g is far from trivial.

The *length* of a path is the number of edges it contains. The *distance* between two vertices is the minimum length of paths connecting the two vertices. Vertices within *sensor-radius* distance from v_c are said to be in the *view* of v_c and those that are exactly out of *sensor-radius* distance away from v_c are said to be *fringe* vertices. Complete path information can be detected only within the view of v_c.

Any navigation algorithm including D* Lite cannot find global lowest cost sequence of transitionsfrom v_s to v_g. There are two main reasons for this. The first is the unpredictable change of terrain information; the second is that for a robot, the current edges cost can only be updated from v_c, to all possible edges within a fixed distance, i.e. the *sensor-radius*. The sensor-radius limitation means a robot's view cannot cover global edge cost changes but only local ones.

However, some algorithms such as D* Lite can find the lowest cost sequence from v_c to v_g based on known edge 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 within the view then $c_e(w, u) = c(w, u)$ in which $c(w, u)$ is the observed cost by robot at v_c, otherwise $c_e(w, u)$ is assumed unchanged as the existed value. The Shortest path from v_c to v_g based on old knowledge of edge costs is called old shortest path. If no edge cost changes, old shortest path automatically becomes the current shortest path, otherwise, a new current shortest path will come out by D* Lite. More explicitly, D* Lite make a robot advance from v_c in the following two steps. Step one is when no edge cost changes are found, the robot advances along the old shortest path, one transition at one time, and stops if edge cost changes are found within the view of v_c. Step two is when it stops, the robot updates previous edge costs, and recalculates a new current shortest path from v_c to v_g. One round is done when one transition is traversed. D* Lite works round by round until the robot reaches v_g.

The second step of the D* Lite algorithm for recomputing a new shortest path contains two important sub-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 within view and assumed unchanged cost beyond view. One function, $g(v)$, with the same name that is used by the A* algo-

SAC'11 March 21-25, 2011, TaiChung, Taiwan.

rithm to estimate costs from v_c to v but in D* Lite estimates the cost from v to v_g. The other function $rhs(v)$ is one-step lookahead 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. For a vertex v if $g(v) = rhs(v)$, it is said to be locally *consistent*; if $g(v) > rhs(v)$, it is locally *overconsistent*; if $g(v) < rhs(v)$, it is locally *underconsistent*. So each vertex has its g and rhs values as a measure of terrain information.

The shortest path cost from a vertex v_s to v_g is known if and only if vertices on this shortest path are consistent. Then we find the shortest path cost from v_s to v_g by choosing such a successor v so that it can minimize $c(v_s, v) + g(v)$ at every round until v_g is reached. When a vertex v becomes inconsistent because of edge cost changes, D* Lite attempts to update its g value equal to rhs value to make it locally consistent, Moreover, this update propagates out to all neighbors of v, since their g values usually depend on the g value of the vertex v; While when the vertex v keeps consistent, its g value remains the same as the previous round. Intuitively, it is more likely that the shortest path traverses *overconsistent* vertices than *underconsistent* ones, so in some variations of the algorithm, for example delayed D* [5], it will delay updating underconsistent vertices.

There are many other algorithms which are using D* Lite as foundation. For example, DD* Lite [6] combines D* Lite with technique of detecting "dominance relationships" to solve robot navigation problem with global constraints. It calculates dominance relationships to prune search space so that it speed up the efficiency of planing; Anytime D* [7], emphasize performance in the area where only limited time available for planning the next action, and allows that every step maybe not the optimal plan. ID* Lite introduced in this paper can speed up D* Lite and can be used to improve variants of D* Lite such as DD* Lite and Anytime D* etc, since it does not conflict with the techniques used in them.

In [8], a way to find alternative path to avoid calculation has been used. But that means, the new path must be "not shorter" than original path, or no calculation can be saved. In this paper, the improvement of ID* Lite is motivated by the observation that if a new current shortest path shorter than the old shortest path from v_c to v_g can be found, only part of propagative updating recalculation is needed. Furthermore, a new current shortest path whose cost is equal to that of old shortest path may be found without propagative updating recalculation which is caused by edge cost changes and hence can not be avoided in D* Lite. We call such a new path is an alternative of old shortest path. Here we define that one path p_1 is said to be an *alternative* of p_2 if they bear the same start point, end point, and cost value. Experiment shows our improvement can speed the replanning work up to 5 times for various benchmarks.

In Section 2, an overview of ID* Lite will be introduced and pseudo code presented. In Section 3 an example is shown how ID* Lite works. In Section 4 results of experiments on various benchmarks are shown. Section 5 makes an analysis of ID* Lite and gives some theoretical results.

2. IMPROVED D* LITE

In this section, the motivation of development of ID* Lite short for Improved D* Lite are introduced. Since in D* family algorithms, D* Lite is most widely used, thus we choose to develop our ideas based on it. As in D* Lite, ID* Lite searches the optimal path from goal to start; The pivotal step in D* Lite is that when a terrain change, i.e. a edge cost change, is observed, all affected vertices will be reinserted into a priority queue and cause propagative updating recalculation afterwards. However, ID* Lite probably only a part or even none of affected vertices are needed to be reinserted into the priority queue, more important, this reduction directly leads to only a part or even none propagative updating recalculation. The conditions when only a part or none of affected vertices is needed will be given below. A series of theory proofs listed in Section 5 show that reducing affected vertices do not prevent ID* Lite to find a new current shortest path from v_c to v_g needed in every round as D* Lite does. Therefore, upon completion of the algorithm, since many vertices are avoided being reinserted into the priority queue, unnecessary propagative updating recalculation is skipped.

In this paper, use w, u to represent vertices and $e(w, u)$ the edge from w to u. $c_e(w, u)$ or $c(w, u)$ for short both represent the weight of $e(w, u)$. We use $'$ to represent renewed information. $c'_e(w, u)$ or $c'(w, u)$ both refer to the new weight of $e(w, u)$ after updating step. p' means a new optimal path containing the same transitions as p, and Ω' means the corresponding cost value of p' compared with Ω the cost value of p, both obtained by updating in current round. We also use e' to represent the same edge as e but with a changed weight for simplifying notation when necessary.

As described above, when terrain changes are detected at v_c, D* Lite will invoke the updating functions and determine the next point to move onto at step two. ID* Lite follows the same meaning of "one round" as in D* Lite. In $Succ(v)$, all vertices v' satisfying $g(v') + c(v, v') = rhs(v)$ are called *children* of v, and v will be called *parent* of them: i.e., at vertex v, robot can choose an arbitrary vertex from *children* for next move. The vertices in *children* are *siblings* of each other. In order to distinguish different kinds of vertices, one function named *type* is created in ID* Lite.

$$type(v) = x : x \in Z, x \in [-3, \infty)$$

We define a vertex v is unavailable if its (g, rhs) value is affected by terrain changes directly or indirectly. $type(v) = -3$ means v is unavailable because it becomes inconsistent due to some changes on edges directly connected with v. v is therefore also called as *inconsistent source*. $type(v) = -2$ means v is unavailable due to *inconsistent source*, since its (g, rhs) can be affected by some *inconsistent sources*. $type(v) = -1$ means v has never been visited. $type(v) = 0$ means v has been visited but not part of current chosen shortest path. Every vertex is initialized with a -1 type-value. If $type(v) \geq 1$, it means v has been visited and is on current chosen shortest path, and the exact type value is the number of its children whose type value is not -3 or -2.

The most expensive operation of D* Lite algorithm is the propagative updating when edge cost changes occur. In the updating part of ID* Lite, only (g, rhs) values of vertices that have a potentiality to help to find a new current shortest path shorter than the old shortest path are updated. "Potentiality of a vertex" here means that it is possible to generate such a shortest path passing through this "vertex"; If no such vertices exist, then any new current shortest path can be proved to be no better than an alternative of the old one. In this situation, if there are alternative shortest paths

and they are not affected by current edge cost changes, then any one of them can be chosen without updating recalculation as the new current shortest path for current round; If neither of the above two situations applies, we will search the new current shortest path incrementally and it is computed in the same manner as D* Lite dose in worst case. So in this way many propagative recalculation can be avoided as well. We will focus on the two former situations that only partial recalculation or no recalculation is needed respectively: when there is a new path from v_c to v_g that is shorter than old shortest path; when an alternative of old shortest path which is not affected by current edge cost changes. From this point, ID* Lite can potentially outperform D* Lite. More details can be found in Figure 1.

In Figure 1, an outline of ID* Lite is displayed. ID* Lite uses variables and functions of D* Lite for consistency while some are slightly modified to support vertex type function and some are newly created. For example, every time a vertex is added into the priority queue (U), its type value will be reset as 0. Other conditions for modification of type value of a vertex have been shown in outline. The reader is referred to [4] for a description of those functions.

Function *process-changes* works as a job-distributor to determine which function will be called depending on what kind of changes occur. For every changed edge $e =< u, v >$, rhs value of vertex u will be updated. Line 35 means that if a vertex has been changed before, function *getbackvertex* will be called.Lines 37 to 50 help to exclude vertices that have to be inserted into priority queue U and cause unnecessary recalculation in D* Lite. Lines 37 to 41 mean that only if a vertex has the potentiality to cause a new current shortest path shorter than the old shortest path, it will be inserted into U and call *mini-compute* at line 41 to find the shorter one; otherwise it will be temporarily stored in *catch*. If an alternative of old shortest path can be found in *get-alternative*, it is done; otherwise, it comes into a loop from line 43 to 50. Function *compute* at line 43 execute as the D* Lite does, but there is a big difference: the vertices in priority queue U are much less than in D* Lite. Moreover, after line 43, the stored vertices in *catch* will be tested whether they can improve the result of function *compute*, if yes, they will be updated. These actions are shown by lines 44 to 50.

The loop from 42 to 50 is similar as delayed D* to check whether a path is available, but with several differences: 1) ID* Lite adds more inconsistent vertices one time; 2) all paths with the same smallest value are tested in one round; 3) once Ω', the cost of a new current shortest path found to be bigger than the old value Ω in *compute*, more efficient recomputing *mini-compute* is called to see whether this new path can be improved by unconsidered decreased changes.

Function *getbackvertex* is used to get back the vertices available which are abandoned by previous rounds. Function *mini-compute* will only update the vertices which satisfy $rhs < g$ (overconsistent) and the condition at *line 12*, so only better solutions will be found in function *mini-compute*.

get-alternative can 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-like structure 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. At line 04, it is asked to update a vertex $r's$ type value, and the updating will be done according to the definition of type function at Section 2. One thing worthy to being noticed is that in ID* Lite the underconsistent vertices are only needed to be considered here.

```
01) bool get-alternative(vertex v_c)
02)   vertex r = v_c.
03)   while(r ≠ v_g)
04)     update r's type value.
05)     if( type(r) > 0) r = r's one child c
            with type(c) ≠ −3 or −2
06)     else if( type(r) = 0 )
07)       type(r) = −2.
08)       if( r = v_c) return FALSE.
09)       for r's child c
            if type(c) = −3 U.insert(c).
10)       r = parent(r).
11)   return TRUE.

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

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

31)    process-changes()
32)   bool better=FALSE, recompute = FALSE.
33)   For every observed edge ⟨u, v⟩ where
       c_e(u, v) 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 if g(u) > rhs(u)
38)       If h(v_c, u) + rhs(u) ≤ Ω,
39)         better = TRUE, UpdateVertex(u).
40)       else type(u)=-3, catch.add(u).
41)   if (better = TRUE) mini-compute().
42)   while(!get-alternative(v_c))
43)     compute().
44)     if Ω' > Ω
45)       better=FALSE.
46)       for type ≠ 0 vertex in catch
47)         if h(v_c, u) + rhs(u) < Ω',
48)           better = TRUE, UpdateVertex(u).
49)           catch.remove(u).
50)       if (better = TRUE) mini-compute().

51)    move( )
52)    Set Array catch=∅, all type values be −1;
53)    Compute as D* Lite does at beginning.
54)    while v_c ≠ v_g
55)      if(EdgesChanged()) process-changes().
56)      type(v_c) = 0, v_c = one type > 0 child of v_c.
```

Figure 1: Main functions of ID* Lite

The main function *move* is invoked at startup. Its operation is almost the same as that of the *move* function of D* Lite except that on every round it calls *process-changes* to try to avoid some unnecessary updating recalculation.

3. A GRID-WORLD EXAMPLE

This section displays how ID* Lite works by a simple example of 4×4 grid-world which is 4-directional shown in the

left of Figure 2.

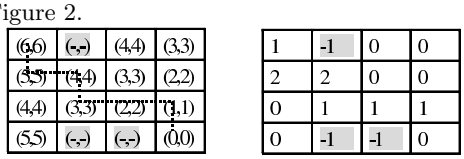

Figure 2: 4×4 4-directional Grid World

In the grid-world where shaded squares are impassable obstacles, a vertex is a square and identified by its row and column. One vertex associated with the i^{th} row and j^{th} column is referred to as $v_{i,j}$, here i and j start from 0. The cost of each edge connecting two unshaded squares is 1, otherwise infinite. v_s is $v_{0,0}$ and v_g is $v_{3,3}$. The $sensor-radius$ is 1. Squares may become shaded or unshaded anytime during movement. Heuristic function is $h(w,u) = 0$. On the left of Figure 2, (g, rhs) values are calculated as in D* Lite, a '-' symbol denotes ∞; there are several shortest path choices with cost $\Omega = 6$, and consider that we use the one marked by the dotted line. The type values in ID* Lite are shown on the right of Figure 2. Notice that, $type(v_{1,0}) = 2$ means that from vertex $v_{1,0}$ there are two choices for next step, that are $v_{1,1}$ and $v_{2,0}$.

Following the chosen shortest path, the robot moves from $v_{0,0}$ to $v_{1,0}$. Then $v_{1,1}$ is found blocked as shown on the left of Figure 3.

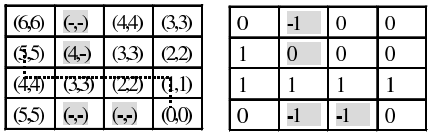

Figure 3: Example of ID* Lite Execution

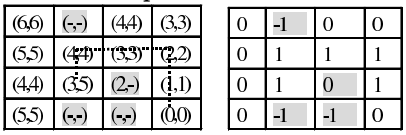

Figure 4: Example of ID* Lite Execution

After updating rhs of vertices, at $v_{1,1}$, $g(v_{1,1}) < rhs(v_{1,1})$, so *mini-compute* will not be executed. Then in *get-alternative*, because $v_{1,1}$ is unavailable due to change, $v_{1,0}$'s type value is 1 now, which means there is only one choice for next move, which is $v_{2,0}$. Repeating iteratively, function *get-alternative* can find a path from v_c to v_g which is shown by dotted line on left of Figure 3 and corresponding type value on right of Figure 3.

Suppose robot is at $v_{2,1}$ and two changes are detected: $v_{1,1}$ unblocked and $v_{2,2}$ blocked shown on the left of Figure 4. Because there are two changes, if $v_{1,1}$ is firstly updated, then (g, rhs) changes from $(4, -)$ to $(4, 4)$. It is turned to be consistent so it does not need to be inserted into priority queue, and Function *getbackvertex* will mark it available again. Actually, suppose $v_{1,1}$ is not consistent after updating, for example from $(5, -)$ to $(5, 4)$, since $h(v_{2,1}, v_{1,1}) + rhs(v_{1,1}) = 0 + 4 = 4 > \Omega = 3$, it still does not need to be inserted into priority queue.

After $v_{2,2}$ is updated, its type value will change from $(2, 2)$ to $(2, -)$, and $v_{2,1}$'s from $(3, 3)$ to $(3, 5)$. Straightforwardly, in next step, the path marked with dotted line on left of Figure 4 can be found by *get-alternative*.

Since D* Lite has covered all the changes of rhs values that ID* Lite has, then it will start a new search in the above example, updating all g and rhs values as usual. Delayed D* operates the same as D* Lite except that it propagates out decreased changes first. But after that it still needs to make an extra check to make sure it gets the shortest path: i.e., there are no inconsistent vertices on its current shortest path. For details please refer to [5].

4. EXPERIMENTS

In this section, ID* Lite is compared experimentally with D* Lite and delayed D* on random grid-world. 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. Here $size$ is the number of columns same as that of rows. Several other parameters are used: 1. $percent$ shows $percent\% * size^2$ of the vertices are selected randomly and blocked; 2. $sensor\text{-}radius$ is the maximum distance observable from the current robot position. We have experiment results for two kinds of benchmarks. The first are results of random rock-and-garden benchmarks: i.e., a blockage found will not move or disappear later. Second is of our novel and realistic benchmarks, which model robot navigation through moving obstacles. Results are measured as average of more than 100 independent examples.

In Figure 5 behaviors of three algorithms for rock-and-garden benchmarks are shown. Heap percolation makes the most significant contribution to time complexity so we can sufficiently compare them by plotting their percolation change. The left shows how *heap percolation* performs when *sensor-radius* increases and parameters *size* and *percent* are fixed; The right shows how *heap percolation* performs when *percent* increases and parameters *size* and *sensor-radius* are fixed.

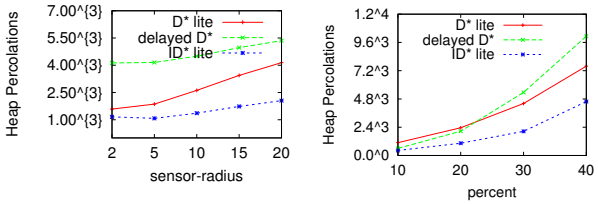

Figure 5: $size = 200, percent = 30$ **and** $size = 200, radius = 20$

In [5], significant delayed D* performance advantages are reported. In particular, delayed D* can perform quite well when there are only a few decreasing changes that cause overconsistent vertices and can successfully find the shortest path through propagation of these vertices. From the right of Figure 5, when *percent* is smaller than 20, i.e. there are not too many increased changes, delayed D* performs good. This gives us a hint in what kind of environment delayed D* can be better than D* Lite. In rock-and-garden benchmarks, there are no decreasing changes so delayed D* does not perform as good as stated in [5].

In addition, we find a second set of benchmarks for which delayed D* performance is also not satisfying because of too many decreasing changes. These benchmarks are intended to model robot navigating through moving obstacles, for example a robot is moving in a parking lot along with other vehicles. We model this situation by initially fixing some percentage of vertices are blocked, then on succeeding rounds, each of the blocked vertices moving to some adjacent vertex with probability 0.5. Since showing the performance results of delay D* would force a change of scale of the plots, we

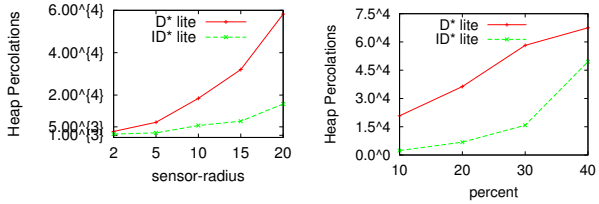

Figure 6: $size = 200, percent = 30$ **and** $size = 200, sensor - radius = 20$

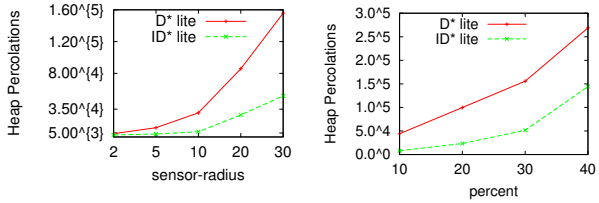

Figure 7: $size = 300, percent = 30$ **and** $size = 300, sensor - radius = 30$

do not show them in Figures 6 and 7. In Figures 6 and 7, the behaviors of *heap percolation* of ID* Lite and D*Lite for the new benchmarks are shown. They can show that ID* Lite can speed up to 5 times than that of D* Lite for various combinations of parameters. With *sensor-radius* increasing, ID* Lite is less sensitive with it. Besides, ID* Lite also performs better with increasing *percent*. We conclude that ID* Lite has a better ability to handle intensely changing environments.

5. ANALYSIS AND THEORETICAL RESULTS

In this section it is shown that on every round ID* Lite can compute a shortest path, if one exists. It is assumed that the heuristic function $h(w, u)$ is the same as that of D* Lite and therefore always bears a lower bound on the minimum cost path from w to u and hence the triangle inequality holds.

LEMMA 1. *In the current round, after propagative updating recalculation if needed, if $e'(w, u)$ is the last changed edge along a shortest path p' from v_c to v_g, then $h(v_c, w)+rhs(w)$ keeps the same value as that before propagative updating recalculation.*

PROOF. Prove it by contradiction. Assume the value of $h(v_c, w) + rhs(w)$ changes, then it means $rhs(w)$ will be affected by some changed edge z'. In the process of updating by D* Lite, p' must include z' and z' must be after e', so z' becomes the last changed edge. This is a contradiction. □

COROLLARY 1. *In the current round, after propagative updating recalculation if needed, if $e'(w, u)$ is the last changed edge along a shortest path p' from v_c to v_g, then the cost value of path p' is $< \Omega$, then before propagative updating recalculation $(h(v_c, w) + rhs(w)) < \Omega$.*

PROOF. If cost value of path p' is $< \Omega$, then after propagative updating recalculation, by property of admissible heuristic function used in ID* Lite, there is $(h(v_c, w) + rhs(w)) < \Omega$. Then the conclusion comes from Lemma 1. □

LEMMA 2. *In the current round, after propagative updating recalculation if needed, if $e'(w, u)$ is the last changed edge along a shortest path p' from v_c to v_g, then before propagative updating recalculation, only w needs to be reinserted into the priority queue in order to get p'.*

PROOF. Without losing generality, assume $z'(v, r)$ is a changed edge along p' before $e'(w, u)$, by property of D* Lite algorithm, if w is reinserted into priority queue, then it will be updated and hence $rhs(r)$ must be propagative affected by w, thus v will also be reinserted and updated. Iteratively, vertices before w on path p' will be subsequently updated until v_c is covered. That means, path p' is generated. □

COROLLARY 2. *In the current round, after propagative updating recalculation if needed, if there exists a new shortest path p' from v_c to v_g with cost less than Ω, then it can be obtained from updating the last changed edge $e'(w, u)$ along p' satisfying $(h(v_c, w) + rhs(w)) < \Omega$.*

PROOF. It follows from Corollary 1 and Lemma 2. □

Corollary 2 means that in order to get a new current shortest path, we only need to locate the last changed edge. But it is hard to locate the last change along a path. So for implementation, such conditions need to be broadened.

PROPOSITION 1. *If there are paths with cost less than Ω, they can be found from propagative updating the changes $e'(w, u)$ with $(h(v_c, w) + rhs(w)) < \Omega$.*

PROOF. It follows straightforwardly from Corollary 2. □

In Proposition 1, it is observed that we do not have to strictly update from "the last change" as required in Corollary 2 to find less-cost paths. That means it broaden the condition on what kind of vertices are available to use. While it is also expected to narrow the condition in Proposition 1 in the future.

LEMMA 3. *In ID* Lite, function* get-alternative *returns $TRUE$ if and only if from v_c to v_g a path whose cost value is $rhs(v_c)$ and satisfying all vertices it traversed are consistent is found.*

PROOF. Firstly function *get-alternative* returns $TRUE$ if and only if a path from v_c to v_g is found. Since ID* Lite operates the same way as in D* Lite to choose the next vertex to move onto, thus no inconsistent vertices will be chosen into the path, and the returned path cost is evidently equal to $rhs(v_c)$. □

LEMMA 4. *In ID* Lite, function* mini-compute *can only update value $rhs(v_c)$ to be less or equal; after execution of* mini-compute, *$rhs(v_c)$ is less than or equal to Ω', which is the cost of the shortest path in D* Lite.*

PROOF. In function *mini-compute*, if a vertex v becomes underconsistent, i.e. $g(v) < rhs(v)$, it will be deleted and hence not cause further propagative updating. It implies that all vertices that are further processed in *mini-compute* are overconsistent. Moreover, propagative updating from such kind of vertices can not increase rhs value of their neighbor vertices including $rhs(v_c)$. That means $rhs(v_c)$ can only be less or equal. The first part of Lemma 4 is done.

Now we prove the latter part of Lemma 4. In D* Lite, no inconsistent vertices is deleted and all inconsistent vertices will be further processed and $rhs(v_c)$ is Ω' which is the cost of the shortest path. So we only need to show that $rhs(v_c)$ in ID* Lite is less than or equal to that in D* Lite. Function *process-changes* in ID* Lite from line 37 to 40 in Figure 1 combined with Proposition 1 mean that all overconsistent vertices causing $rhs(v_c)$ to decrease will be further

processed. So comparing with D* Lite, ID* Lite only process vertices that will cause $rhs(v_c)$ to decrease in D* Lite but does nothing to vertices causing $rhs(v_c)$ to increase in D* Lite. So, finally $rhs(v_c)$ in ID* Lite will definitely be less or equal with that in D* Lite. That also means $rhs(v_c)$ returned by *mini-compute* will be less or equal with Ω'. □

Lemma 4 shows that function *mini-compute* can return cost value of a path less than or equal to that in D* Lite. It can be explained more by the following argument. Assume in D* Lite there is a shortest path p' passing through only one decreased cost change $e' = (w, u)$ and only one increased cost change $z' = (v, r)$. In ID* Lite, if v has $g(v) < rhs(v)$, it will be deleted, so w's values will not be affected by v and $rhs(w)$ can only become less. So finally cost of p' in ID* Lite will be less than that in D* Lite. Furthermore, since the cost value of the shortest path returned by *mini-compute* becomes less than that of p', thus although p' is the shortest path, it can not be returned by *get-alternative*. Actually, in function *get-alternative* this path can be chosen but considered unavailable when v is checked. In this case, we apply the strategy that inserting v into the priority queue and recomputing in *process-changes*. The reason to do so is for efficiency.

Because not all vertices satisfying the equation in Proposition 1 are propagated immediately in *process-changes*, it is possible that some shortest path found by *mini-compute* will be denied when checked by *get-alternative* due to *inconsistency*. In a nutshell, vertices satisfying the equation in Proposition 1 but with $g < rhs$ will not be propagated. To deal with this situation, we have Lemma 5 as follows:

LEMMA 5. *Function* process-changes *can terminate (exit loop from line* 42 *to* 50 *in Figure 1) and a shortest path traversing consistent vertices will be found.*

PROOF. Suppose the new current shortest path's cost value in D* Lite is Ω'. When line 42 is executed the first time, by Lemma 3 if function *get-alternative* return *TRUE*, one path p' passing through consistent vertices can be returned. Suppose its cost value is c. Because Ω' is the value of current shortest path in D* Lite, so $c \geq \Omega'$, while by Lemma 4 $c \leq \Omega'$, hence $c = \Omega'$. It means that if function *get-alternative* return *TRUE*, a shortest path is found.

If *FALSE* is returned by function *get-alternative*, then at line 09 underconsistent vertices with $g < rhs$ will be reinserted into priority queue. From lines 43 to 50 in Figure 1, following the property of function *compute* in D* Lite, those inserted underconsistent vertices will be updated to be consistent. By property of D* Lite and Proposition 1, it can be proved, similarly as proof of Lemma 4, that $rhs(v_c)$ will be less or equal with Ω' although some underconsistent vertices are processed in ID* Lite. Because those are a subset of underconsistent vertices processed by D* Lite, so for an arbitrary vertex, its rhs value will be less than or equal to that in D* Lite. When the loop lines 42 to 50 repeated, if function *get-alternative* can return $TRUE$, with the same proof as in last paragraph, the path found is the shortest.

If *FALSE* is continually returned by function *get-alternative*, more and more underconsistent vertices will be reinserted and updated. Since underconsistent vertices are finite, then in the worst case, all unerconsistent vertices are updated as D* Lite does. According to correctness of D* Lite, all vertices on shortest paths from v_c to v_g will be consistent and then $rhs(v_c)$ will be equal with Ω'. I.e., function *get-alternative* will return *TRUE*. □

LEMMA 6. *Function* process-changes *will find the shortest path in every round if and only if there exists one.*

PROOF. It can be proved from Lemma 5. □

THEOREM 1. *In every round, ID* Lite returns a shortest path from v_c to v_g if and only if one shortest path exists.*

PROOF. It Follows directly from Lemma 6. □

6. CONCLUSION AND NEXT STEP OF WORK

In this paper, an improvement to the D* Lite algorithm for replanning has been introduced and packaged into an algorithm called ID* Lite. The improvement is to update vertices as few as possible and to seek alternative shortest paths when inconsistencies are discovered, rather than recalculate and remove all inconsistencies before finding a new shortest path. It is shown that experiment results are far better performance on random grid benchmarks. Furthermore, the modifications proposed for D* Lite can applied to other D* Lite variants easily. We will combine ID* Lite with D* Lite-based algorithm, such as DD* Lite and anytime D* etc, to test its robustness in various areas in the future.

7. 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, 1994.

[2] Anthony Stentz, "The Focussed D* Algorithm for Real-Time Replanning," *Proceedings of the International Joint Conference on Artificial Intelligence*, Montrĺeal, Quĺebec, Canada, pp. 1652-1659, 1995.

[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, 2002.

[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, 2005.

[6] Mills-Tettey, G. Ayorkor and Stentz, Anthony and Dias, M. Bernardine, "DD* lite: efficient incremental search with state dominance", *AAAI'06: proceedings of the 21st national conference on Artificial intelligence*, Boston, Massachusetts, pp. 1032-1038, 2006.

[7] Likhachev, Maxim and Ferguson, Dave and Gordon, Geoff and Stentz, Anthony and Thrun, Sebastian, "Anytime search in dynamic graphs", *Journal of Artificial Intelligence*, Essex, UK, Volume 172:14, pp. 1613-1643, 2008.

[8] 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.