A New Anytime Dynamic Navigation Algorithm

Weiya Yue, John Franco, Weiwei Cao*, Qiang Han

***Abstract*—Dynamic navigation algorithm has been an important component in planning. Navigation algorithm is required to find out an optimal solution to its goal. However, under some environments where time is more critical than optimality, a sub-optimal solution is acceptable. Therefore for practical applications it is useful to find a high sub-optimal solution in limited time. The dynamic algorithm Anytime D*(AD*) is currently the best anytime algorithm which aims to return a high sub-optimal solution with short corresponding time and control of sub-optimality. In this paper, a new algorithm named Improved Anytime D*(IAD*) is introduced. Experiment comparison is made to show IAD* better outperforms Anytime D* in various random benchmarks.**

***Index Terms*—Planning, Dynamic Navigation Algorithm, Anytime, Incremental**

I. Introduction

With development of techniques, it is highly possible to develop autonomous vehicles, intelligent agents, etc. Because of this, navigation algorithm has been more and more important. In navigation algorithm, an agent is required to find out one optimal solution to its goal under changing environment. Many algorithms have been developed to solve this problem and gained big success [1], [2], [3], [4], [5], [6], [7]. But under some environments, where there is no sufficient resources for the agent to find out an optimal solution, a sub-optimal solution is acceptable. In this paper, we will focus on the environment where time is more critical than optimality.

Time-limited search algorithms, called *anytime planning* algorithms, have been developed to fit in time critical environment. The basic idea is to find one solution as soon as possible, then to progressively replace a stored path with a better path when one is discovered during search until the time available for search expires [8], [9], [10]. Then the stored, probably sub-optimal path is the one that is used. In [8], [11], [12], it has been demonstrated that a so-called weighted A* algorithm variant which uses "inflated" heuristics (described below) can expand fewer vertices than the normal A* algorithm.

In A*, the vertices in $OPEN$ are sorted by their values $f = g + h$. By assuming h is admissive, in A* algorithm if we use $f = g + \epsilon \cdot h$, then the returned path can be guaranteed to be ϵ sub-optimality, i.e. $g(v_g) \leq \epsilon \cdot g^*(v_g)$ [13]. This strategy is called inflated heuristics, and the benefit is the control of ϵ sub-optimality. The A* algorithm using inflated heuristics is named weighted A* algorithm. In [8], one general method, named Anytime Weighted A*, to transform heuristic search algorithms to anytime algorithms is proposed. Anytime Weighted A* is a anytime planning algorithm which returns one sub-optimal solution as soon as possible then whenever allowed continues to improve current solution until one optimal solution returned.

Anytime Weighted A* does initialization on most variants as A* algorithm, and as Weighted A*, the heuristic function h used is admissive. Different from normal A* algorithm, in Anytime Weighted A* p is used to record current returned path which may be improved later; $ERROR$ is used to estimate how far away current solution is from optimal path; ϵ is the parameter used to inflate h; in one vertex $v \in OPEN$, the stored values are $\langle g(v), f'(v)\rangle$ instead of $\langle g(v), f(v)\rangle$, in which $f'(v) = g(v)+\epsilon \cdot h(v)$. I.e., in Anytime Weighted A* in priority queue $OPEN$, the vertices are sorted by f' value instead of f value in normal A* algorithm. Although Anytime Weighted A* uses inflated heuristic value f' to sort vertices, normal f values are also recorded, which is used to prune searching space [9].

Under the changing environment, to search a path between a fixed pair of vertices within limited time, incremental anytime algorithm Anytime Repairing A*(ARA*) algorithm [14] was developed to mitigate this problem. ARA* runs weighted A* algorithm many times. Every time changes observed, ARA* needs to run weighted A* to find the new sub-optimal path. Most important to the performance of ARA* is that it reuses previously calculated information to avoid duplicating computation. This is done in accordance with ideas taken from [15], [16]. By observing that in weighted A* when h is admissive, if every vertex is allowed to be expanded only once, the returned path is still ϵ sub-optimal, every time ARA* needs to recalculate, ARA* will only update a vertex at most one time. ARA* starts with a large value for the so-called *inflated parameter* ϵ and then reducing ϵ on each succeeding round until either $\epsilon = 1$ or available time expires. ARA* performs similarly as Anytime Weighted A* algorithm and give the ability to control the sub-optimality ϵ.

D* Lite algorithm can be treated a dynamic version of lifelong A* algorithm [15], [16]. As D* algorithm [3], [4], D* lite searches backward from v_g to v_s. This is likely the critical point to the success of D* and its descendants because the g value of every node is exactly the path cost from that node to the goal v_g and can be used *after* the agent moves to its next position. The function rhs is defined by

$$rhs(v) = \begin{cases} \min_{v' \in succ(v)} g(v') + c(\langle v, v'\rangle) & v \neq v_g \\ 0 & \text{otherwise,} \end{cases}$$

The "more informed" rhs function assists in making better vertex updates during expansion. Call vertex v *locally consistent* if $rhs(v) = g(v)$, *locally overconsistent* if $rhs(v) < g(v)$, and *locally underconsistent* if $rhs(v) > g(v)$. In the latter two cases v is said to be *inconsistent*. A "best" path can be found if and only if, after expansion of v_s, all vertices on the path are locally consistent and can be computed

Manuscript received August 18, 2012.

W. Yue and J. Franco and Q. Han are with the Department of Computer Science, University of Cincinnati, Cincinnati, OH, 45220 USA e-mail: weiyayue@hotmail.com, franco@gauss.ececs.uc.edu, hanqg@ucmail.uc.edu.

W. Cao is with Institute of Information Engneering, Chinese Academy of Science, Beijing, China, 100093 e-mail: weiwei.cao@hotmail.com.

ISBN: 978-988-19251-6-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

by following the maximum-g-decrease-value vertices one by one from target v_g. If some changes that have been made since the last round cause a vertex v to become inconsistent then D* Lite will update $g(v)$ to make v locally consistent by setting $g(v) = rhs(v)$. Because D* Lite algorithm will only propagate inconsistent vertices to update partial vertices' (g, rhs) values instead of updating all vertices' (g, rhs) values, D* Lite can perform much better than other navigation algorithms.

Anytime D* [17] intends for dynamic navigation applications where optimality is not as critical as response time. It may be thought of as a descendant of both the Anytime Repairing A*(ARA*) and D* Lite algorithms. It may re-calculate a best path more than once in a round with decreasing ϵ-suboptimality until $\epsilon = 1$ or time has run out. Thus, Anytime D* will try to give a relatively good, available path quickly and, if time allows, will try to improve the path incrementally as is the case for Anytime A*.

In [5], [6], D* Lite algorithm has been improved by avoiding unnecessary calculations further, in which if the original optimal path is still available and can not be improved by changes observed, that path will be chose without computing. In [7], the ID* Lite algorithm uses a threshold number which is estimated to control the propagation of inconsistent vertices. Every time only changes may contribute a path whose weight is less or equal with the threshold number are propagated. The threshold number is increased until an optimal path found or all inconsistent vertices have been updated.

In this paper, we will combine the techniques used in ID* Lite to improve Anytime D* algorithm and get an algorithm names Improved Anytime D* algorithm, and IAD* for abbreviation. In Section II, pseudo code of IAD* is listed and described. Then in Section III, IAD* and AD* are compared in various random benchmarks. At last, we concludes and discuss the next step of work.

II. Improved Anytime D* Algorithm

In this section at first we explain how IAD* algorithm works and then give its pseudo code. As Anytime D* algorithm, when changes observed, IAD* will try to update inconsistent vertices to get a new sub-optimal date. Every time, after recalculating, it is required to return a sub-optimal path whose weight is no bigger than $\epsilon' \cdot g^*$ in which ϵ' is a preset parameter to control sub-optimality and g^* is the weight of current optimal path.

Every time, when recalculating needed, the sub-optimality parameter ϵ is reset to be ϵ' which is relatively big. By doing this, we expect to return a path as soon as possible [8], [12], [11]. In [18], it is recommended that in weighted A* ϵ can be set to be bigger than ϵ' and experiments show that the first path can be returned faster. This technique can be combined with any weighted A* algorithm easily. In order to speed up returning the first path, Anytime D* allow one vertex to be expanded at most one time which will be explained later. But it has been shown that in some benchmarks this may delay propagation of some critical vertices and slow down the algorithm [8]. In Section III, we will run experiments to compare these results.

After the first path returned, as any other anytime algorithm, IAD* will try to improve current path until time runs out or an optimal path has been found. In Anytime D*, in order to do so, ϵ is decreased to look for a new path until $\epsilon = 1$ which means the returned path is optimal. When ϵ is decreased, in Anytime D* all inconsistent are inserted into priority queue to be updated. As in ID* Lite [7], IAD* will not do any recalculating at all. Given a ϵ, at first, IAD* will try to find one consistent ϵ' sub-optimal path which is not affected by changes observed, and if such a path exists, it is returned immediately as the first path found. If no such a path found, IAD* will only choose part of overconsistent vertices whose propagation may lead to a ϵ sub-optimal path.

The pseudo code of IAD* is listed in Figure 1. Function **Initialize()** defines and initializes ϵ, and the priority queues OPEN, CLOSED, and INCONS, and initialize values for g, rhs and $type$ values of vertices. The initial value of ϵ_0 is relatively large in order to make sure some path is returned quickly. And vertex v_g and its key is inserted into priority queue OPEN.

In Function **key**(s), under-consistent vertices have their key-values updated as $g(s) + h(s)$ which is smaller than $\mathbf{key}(v_s)$. This processing can guarantee such kind of increasing changes can be propagated. Function **UpdateVertex**(s) updates one vertex in the same way as Anytime D* by using INCONS to store some of the inconsistent vertices and making sure that one vertex is expanded at most once in one execution of **ComputeOrImprovePath**(). After doing this, there is still returned solution generated satisfies ϵ-suboptimality [17].

Functions **ComputeorImprovePath**(), **MiniCompute**() and **GetBackVertex(v)** are the same as in ID* Lite. Function **GetAlternativePath**(v_c) returns TRUE if and only if there is a path from v_c to v_g and, if it returns TRUE, it has changed type values on vertices so that a least cost path from v_c to v_g can be traversed by visiting neighboring vertices of lowest positive type until v_g is reached. Different from ID* Lite, at line 05, instead of choosing a child of r, one successor y of r with $rhs(y) + c(r, y) \leq rhs(r)$ is chose. The reason is that here we only need a suboptimal path.

Different from ID* Lite, here the returned path may be not optimal, but is guaranteed to be ϵ' suboptimal. It worths to notice that if a path returned, and on which there are overconsistent vertices, then the path returned is better than ϵ' suboptimal. If no path can be returned by function **GetAlternativePath**(v_c), every vertex $c \in C$ is updated by function **UpdateVertex**. Observe that c is underconsistent and that this is the only place in the code where underconsistent vertices are placed in OPEN as in ID* Lite [7]. This is because only increased changes will cause underconsistent vertices, and increased changes are only inserted here. Decreased changes have been inserted before **GetAlternativePath**(v_c) was called.

ProcessChanges acts similarly as ID* Lite. But differently, at line 07 and 17, to test whether a overconsistent vertex should be put in OPEN to propagate, $\epsilon * h(v_c, u) + rhs(u) < t$ is used instead of $h(v_c, u) + rhs(u) < t$. Functions **Main**() and **MoveAgent**() are the same as Anytime D*.

We end this Section with the Theorem of correctness of IAD*.

Theorem 2.1: In Improved Anytime D* algorithm, the

ISBN: 978-988-19251-6-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

Procedure Initialize()
01. OPEN = CLOSED = INCONS = catch=$\emptyset$;
02. for all $v \in V$, $rhs(v) = g(v) = \infty$; **type**$(v) = -1$;
03. $rhs(v_g) = g(v_g) = $ **type**$(v_g) = 0$; $\epsilon = \epsilon_0$
04. OPEN.insert($[v_g,\ [h(v_g),\ 0]]$);

Procedure key(s):
01. if $(g(s) > rhs(s))$
02. return $[rhs(s) + \epsilon \cdot h(s),\ rhs(s)]$;
03. else
04. return $[g(s) + h(s),\ g(s)]$;

Procedure UpdateVertex(s):
01. if s has not been visited
02. $g(s) = \infty$;
03. if $(s \neq v_g) rhs(s) = \min_{s' \in succ(s)}(c(\langle s, s' \rangle) + g(s'))$;
04. if ($s \in$ OPEN) OPEN.remove(s);
05. if $(g(s) \neq rhs(s))$
06. if ($s \in$ CLOSED)
07. OPEN.insert($[s,$ **key**$(s)]$);**type**$(v) = 0$;
08. else
09. insert s into INCONS;

Procedure ComputeOrImprovePath():
01. while (OPEN.TopKey() $<$ **key**(v_s) OR $rhs(v_s) \neq g(v_s)$)
02. s = OPEN.Top(), OPEN.remove(s);
03. if $(g(s) > rhs(s))$
04. $g(s) = rhs(s)$;
05. CLOSED.insert(s);
06. for all $s' \in pred(s)$ **UpdateVertex**(s');
07. else
08. $g(s) = \infty$;
09. for all $s' \in pred(s) \cup \{s\}$ **UpdateVertex**(s');

Procedure MiniCompute( )
01. while (OPEN.TopKey() $<$ **key**(v_c))
02. u = OPEN.Top(), OPEN.remove(u);
03. if $(g(u) > rhs(u))$
04. $g(u) = rhs(u)$;
05. CLOSED.insert(s);
06. for all $s' \in pred(s)$ **UpdateVertex**(s');
07. else
08. OPEN.Remove(u);

Procedure GetAlternativePath(v_c)
01. Vertex $r = v_c$;$C = \emptyset$
02. while $(r \neq v_g)$
03. update r's type value;
04. if (**type**$(r) > 0$)
05. $r =$ one successor y of r with $rhs(y) + c(r, y) \leq rhs(r)$ *and* **type**$(y) \neq -3$ and **type**$(y) \neq -2$;
06. else if (**type**$(r) == 0$)
07. **type**(r) = -2;
08. if $(r == v_c)$
09. for every vertex $c \in C$ **UpdateVertex**(c);
10. return FALSE;
11. $C = C \cup r's$ **type** value -3 children; $r = parent(r)$;
12. return TRUE.

Procedure GetBackVertex(v)
01. if ($v \neq$ NULL and **type**$(v) < 0$)
02. if $(rhs(p) \neq g(p))$
03. return;
04. **type**(v) = 0;
05. $v = parent(v)$;
06. **GetBackVertex**(v);

Procedure ProcessChanges()
01. Boolean $better$=FALSE, $recompute$ = FALSE, $t = rhs(v_c)$.
02. for every edge $e = \langle u, v \rangle$ where $c(e)$ has changed since the previous round:
03. Update $rhs(u)$;
04. if (**type**$(u) = -3$) **GetBackVertex**(u);
05. if $(rhs(u) == g(u))$ **type**$(u) = 0$;
06. else
07. if $(g(u) > rhs(u))$ and $\epsilon * h(v_c, u) + rhs(u) < t$
08. $better$ = TRUE, **UpdateVertex**(u);
09. else
10. $catch$.add(u), **type**$(u) = -3$;
11. if ($better ==$ TRUE) **MiniCompute**();
12. while (!**GetAlternativePath**(v_c))
13. $t_{old} = t$, **ComputeShortestPath**(), $t = rhs(v_c)$;
14. if $t > t_{old}$
15. $better$=FALSE;
16. for every $u \in catch$ such that **type**$(u) \neq 0$
17. if ($\epsilon * h(v_c, u) + rhs(u) < t$ and $g(u) > rhs(u)$)
18. $better$ = TRUE, **UpdateVertex**(u);
19. $catch$.remove(u).
20. if ($better ==$ TRUE) **MiniCompute**().

Procedure Main():
01. **Initialize**();
02. **ComputeOrImprovePath**();**GetAlternativePath**(v_c);
03. publish current ϵ-suboptimal solution;
04. repeat the following:
05. for all directed edges $\langle u, v \rangle$ with changed edge costs
06. Update the edge cost $c(\langle u, v \rangle)$;
07. **UpdateVertex**(u);
08. if significant edge cost changes were observed
09. increase ϵ or replan from scratch;
10. else if $(\epsilon > 1)$
11. decrease ϵ;
12. CLOSED = $\emptyset$;
13. **ProcessChanges**();
14. publish current ϵ-suboptimal solution;
15. if $(\epsilon == 1)$
16. wait for changes in edge costs;

Procedure MoveAgent():
01. while $(v_s \neq v_g)$
02. wait until a plan is available;
03. Set **type**$(v_c) = 0$;
04. $v_c = u$ where u is a child of v_c and **type**$(u) == 0$;
05. Move the agent to v_c;
06. $v_s = argmins_{s \in succ(v_s s)}(c(\langle v_s, s \rangle) + g(s))$;
07. move agent to v_s;

Fig. 1. Main functions of IAD*

returned path between v_c and v_g has its cost no larger than $\epsilon * g'(v_c)$ in which $g'(v_c)$ is the cost of optimal path between v_c and v_g.

Proof: This Theorem follows the correctness of ID* Lite algorithm and Anytime D* algorithm. ■

III. Experiments and Analysis

In this section, the performance of IAD* is compared experimentally with Anytime D* on random grid world terrains. In each experiment the terrain is a square, 8-direction grid world of $size^2$ vertices. Special vertices v_s and v_g are chosen randomly from the terrain. Initially $percent\% * size^2$ of the vertices are selected randomly and blocked, $percent$ being a controlled parameter. The parameter *sensor-radius* is used to set the maximum distance to a vertex that is observable from the current agent position. Before navigation, the traveling agent has a old map, in which an obstacle may be wrongly considered to be blank with a fifty percent possibility.

ISBN: 978-988-19251-6-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

To compare the anytime dynamic navigation algorithms, we control the sub-optimality by setting parameter ϵ. I.e. solutions returned by the algorithm is ϵ sub-optimal. One criterion of anytime algorithm is the time of returning the first sub-optimal path. In this condition, the less operations used, the better is the algorithm. We will run two kinds of random benchmarks to compare IAD* algorithm and Anytime D* algorithm with constant sub-optimality parameter ϵ. The first set of results are on random rock-and-garden benchmarks. That is, a blockage is set initially and will remain for the entire experiment. The second set of results are on a collection of benchmarks that model agent navigation through changing terrain, named as Parking-lot benchmarks. I.e., a blockage may move to its neighborhood randomly during the navigation.

Form Figure 2 to 5, AD* and IAD* are compared on rock-and-garden benchmarks. From Figures 2 to 5, the results of $percent = 10$ and $sensor - radius = 0.1 * size$ are presented for $size = 300$ and $size = 500$ respectively. These figures show the number of heap operations and also the time consumed as a function of sub-optimality EPSILON(ϵ). From the figures, we can see that IAD* gains more than one order of speeding up than AD* in some cases. About the consumed time, IAD* needs extra time to calculate the alternative path besides the time of applying heap operations which is different from AD*. From the figures we can see that in IAD* the time to calculate alternatives cost only takes a very small percentage of the whole time, which means it does not affect the speed of IAD* much. It also shows that heap operations consist the main time complexity in navigation algorithms. Compared with the speeding up of ID* Lite for D* Lite [7], the speed up performance of IAD* for AD* is much better. There are two main reasons, the first one is that there are more alternatives because of sub-optimality. In order to find an alternative of path P_1, in ID* Lite, only pathes with the same cost as P_1 can be used as alternatives and there are no pathes with cost smaller than P_1. But in IAD*, all pathes with cost $\leq P_1$ can be used as alternatives. The second one is that AD* needs to reorder its priority queue OPEN every time of recalculating. If IAD* can avoid the recalculating, then the reordering of priority queue can be skipped. The method used in [3] to avoid reordering priority queue can also be used in AD* by modifications introduced in [17]. Unfortunately, the heuristic used in AD* is not consistent, hence that method will cause a lot of reinsertion of key values of vertices because of updating. In [17], the authors choose to reorder the priority queue when recalculating and consider this operation is bearable for time.

The second set of benchmarks, Parking-lot benchmarks, is intended to model agent navigating in the presence of terrain changes. Compared with rock-and-garden benchmarks, we are more interested in parking-lot benchmarks. The reason is the later can simulate the practical environment better. Hence we will give more comparisons between IAD* and AD* on this kind of benchmarks. Terrain changes are commonly encountered by autonomous vehicles of all kinds and may represent the movement of other vehicles and structures in the agent's environment. A number of *tokens* equal to a given fixed percentage of vertices are initially created and distributed over vertices in the grid, at most one token

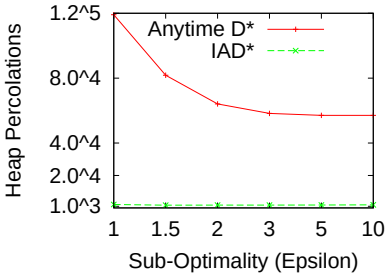

Fig. 2. $size = 300$, $percent = 10$, $sensor - radius = 30(rock - and - garden)$

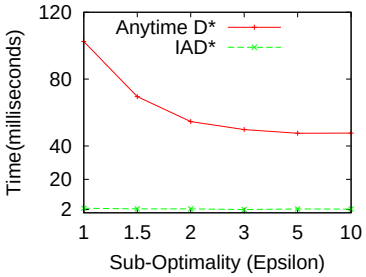

Fig. 3. $size = 300$, $percent = 10$, $sensor - radius = 30(rock - and - garden)$

covering any vertex. As an agent moves from vertex to vertex through the grid tokens move vertex to vertex as well. Tokens are never destroyed or removed from the grid and the rules for moving tokens do not change: on each round a token on vertex v moves to a vertex adjacent to v with probability 0.5 and the particular vertex it moves to is determined randomly and uniformly from the set of all adjacent vertices that do not contain a token when the token is moved. Tokens are moved sequentially so there is never more than one token on a vertex. Any vertex covered by a token at any point in the simulation is considered blocked at that point which means all edge costs into the vertex equal ∞. A vertex with no

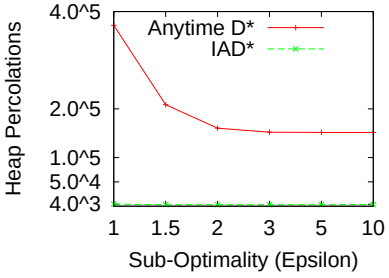

Fig. 4. $size = 500$, $percent = 10$, $sensor - radius = 50(rock - and - garden)$

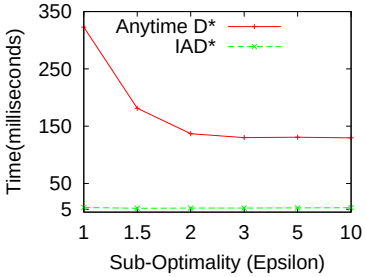

Fig. 5. $size = 500$, $percent = 10$, $sensor - radius = 50(rock - and - garden)$

ISBN: 978-988-19251-6-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

token is unblocked and edge costs into it are not ∞.

From Figure 6 to 9, IAD* and AD* are compared similarly as from Figures 2 to 5. We can see that IAD* can also get one order of speeding up, but not as good as in rock-and-garden benchmarks. The reason is that in parking-lot benchmarks, more recalculating are needed by IAD*. This is similar as ID* Lite compared with D* Lite [7]. As we have seen, the time has similar plot as heap percolation, hence below we only show the heap percolation plot because of the limited space.

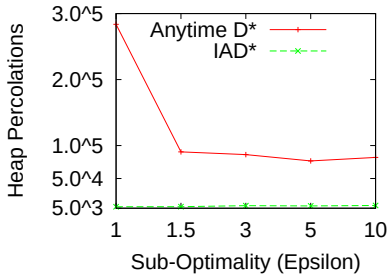

Fig. 6. $size = 300,\ percent = 10,\ sensor-radius = 30(Parking-lot)$

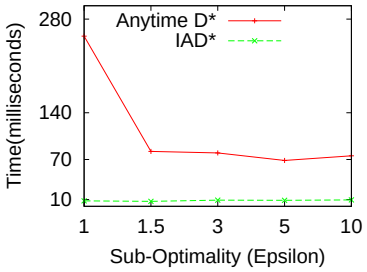

Fig. 7. $size = 300,\ percent = 10,\ sensor-radius = 30(Parking-lot)$

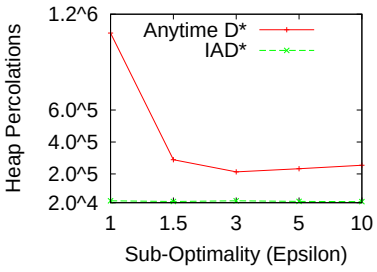

Fig. 8. $size = 500,\ percent = 10,\ sensor-radius = 50(Parking-lot)$

In Figures 10 and 11, AD* and IAD* are compared similarly as in Figures 6 and 8, but with $percent = 20$. This is to show whether IAD* can perform well in dramatically changing environments. We can see IAD* can still achieve up to one order times of speeding up than AD*. Also, from such figures we can see that the plot of AD* decreases faster than IAD*. Hence we can say that with other conditions made certain, the smaller the sub-optimality required, the better IAD* performs.

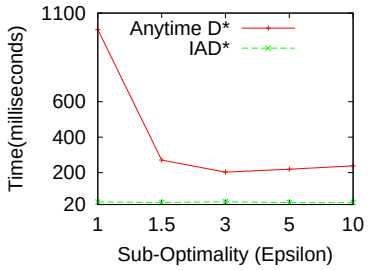

Fig. 9. $size = 500,\ percent = 10,\ sensor-radius = 50(Parking-lot)$

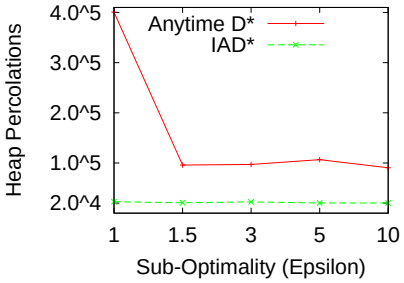

Fig. 10. $size = 300,\ percent = 20,\ sensor - radius = 30(Parking - lot)$

In Figure 12, results are compared with different sensor-radius given $size = 300, percent = 3$ and $\epsilon = 3$. We can see the heap operations increases with $sensor-raduis$. The reason is that in parking-lot benchmarks the blockages are keeping moving, hence a larger sensor-radius means more changes observed which may cause more updating.

In Figure 13, results are compared with different $percent$ given $size = 300, sensor - radius = 30$ and $\epsilon = 3$. We can see that IAD* has heap operations increased faster than AD* when percent is increased. When percent is increased, in order to find an alternative, more recalculations are needed

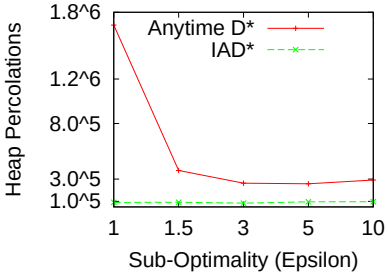

Fig. 11. $size = 500,\ percent = 20,\ sensor - radius = 50(Parking - lot)$

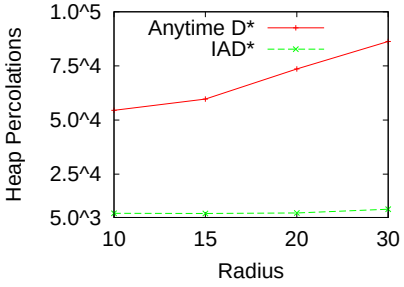

Fig. 12. $size = 300,\ percent = 10,\ \epsilon = 3(Parking - lot)$

ISBN: 978-988-19251-6-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

in IAD*, which affects its performance. Hence, in parking-lot kind of environments, if the changing of terrain is relatively light, IAD* can have a better performance than in heavily changing environments. From the figure, we can also see that, even with $percent = 30$, IAD* can still get about two times speeding up than AD*.

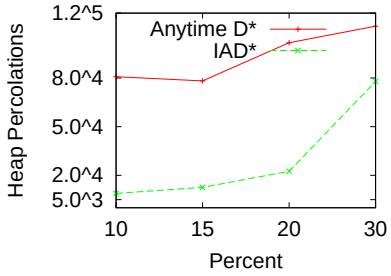

Fig. 13. $size = 300,\ sensor - radius = 30,\ \epsilon = 3(Parking - lot)$

At last in Figure 14, results are compared with different $size$ given $percent = 10, sensor - radius = 30$ and $\epsilon = 3$. The results show that IAD* is very scalable in parking-lot benchmarks.

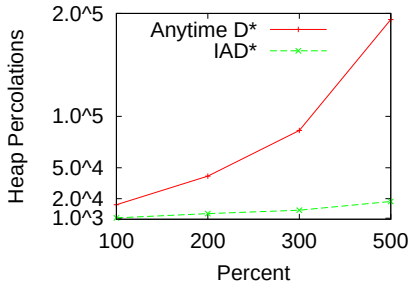

Fig. 14. $percent = 10,\ sensor - radius = 30,\ \epsilon = 3(Parking - lot)$

From the results above, we can see IAD* returns the first qualified sub-optimal path in a shorter time than AD*. Also when other conditions unchanged and in rock-and-garden benchmarks ϵ from 1 to 1.5 IAD* can save about 25 percent of calculations; but in parking-lot benchmarks, the calculations of IAD* varies a little. So IAD* has the ability of returning a high sub-optimal path especially in parking-lot style of environments, and when something emergency happens, for example a lot of changes observed, the first sub-optimal path can be returned fast; after that, IAD* can continue to improve current path until time runs out. I.e., a desired sub-optimal path can be guaranteed to be found in a short time with a high possibility, which also means more time can be used to improve the firstly returned path. Hence, we can conclude that IAD* can return the first sub-optimal path faster than AD* in various random benchmarks, from which IAD* gains a better potentiality to return high sub-optimal path within limited time.

IV. Conclusion and Next Step of Work

In this paper, we propose a new dynamic anytime algorithm IAD*. IAD* improves AD* following the similar strategy as that ID* Lite improves D* Lite. That is, IAD* will try to find an alternative of original path instead of recalculating immediately as in AD*. Moreover, if an alternative is not available, in order to avoid a full recalculation IAD* will try to propagate changes part by part with the help of a threshold until a new sub-optimal path found. Experimental results show that IAD* can achieve up to one order of speeding up in various random benchmarks. There is still much work can be done in the next step. For example, We will consider when there is an upper bound of heap operations allowed if there are limited resource and how to make the algorithm to achieve a better sub-optimality; As discussed in this paper IAD* has a better potential to support high sub-optimal path, so we can compare this aspect of AD* and IAD* by experiment; Anytime Weighted A*(AWA*) [8] is demonstrated performing better in some benchmarks to achieve a higher sub-optimality than other algorithms, so combining IAD* with AWA* to get a new faster algorithm is also an interesting work we will do.

Acknowledgement: The work of this paper was supported by the Strategic Priority Research Program of Chinese Academy of Sciences under Grant XDA06010702, the State Key Laboratory of Information Security, Chinese Academy of Sciences, the National Natural Science Foundation of China (Grants 61070172 and 10990011).

References

[1] S. Koenig and M. Likhachev, "D*lite," in *Eighteenth national conference on Artificial intelligence 2002*, pp. 476–483.
[2] ——, "Improved fast replanning for robot navigation in unknown terrain," 2002, pp. 968–975.
[3] A. Stentz, "The focussed d* algorithm for real-time replanning," in *Proceedings of the International Joint Conference on Artificial Intelligence 1995*, pp. 1652–1659.
[4] ——, "Optimal and efficient path planning for partially-known environments," *The Kluwer International Series in Engineering and Computer Science*, vol. 388, pp. 203–220, 1997.
[5] W. Yue and J. Franco, "Avoiding unnecessary calculations in robot navigation," in *Proceedings of World Congress on Engineering and Computer Science 2009*, pp. 718–723.
[6] ——, "A new way to reduce computing in navigation algorithm," *Journal of Engineering Letters*, vol. 18(4), 2010.
[7] W. Yue, J. Franco, W. Cao, and H. Yue, "Id* lite: improved d* lite algorithm," in *Proceedings of 26th Symposium On Applied Computing 2011*, pp. 1364–1369.
[8] E. A. Hansen and R. Zhou, "The heuristic search under conditions of error," *Journal of Artificial Intelligence Research*, vol. 28, pp. 267–297, 2007.
[9] L. Harris, "The heuristic search under conditions of error," *Artificial Intelligence*, vol. 5(3), pp. 217–234, 1974.
[10] R. Zhou and E. Hansen, "Multiple sequence alignment using anytime a*," in *Proceedings of Conference on Articial Intelligence 2002*, pp. 975–976.
[11] R. Korf, "Linear-space best-first search," *Artificial Intelligence*, vol. 62(1), pp. 41–78, 1993.
[12] I. Pohl, "Heuristic search viewed as path finding in a graph," *Artificial Intelligence*, vol. 1(3), pp. 193–204, 1970.
[13] H. Davis, A. Bramanti-Gregor, and J. Wang, "The advantages of using depth and breadth components in heuristic search," *Methodologies for Intelligent Systems*, vol. 3, pp. 19–28, 1988.
[14] M. Likhachev, G. Gordon, and S. Thrun, "Ara*: anytime a* with provable bounds on sub-optimality," *Advances in Neural Information Processing Systems*, 2003.
[15] S. Koenig and M. Likhachev, "Incremental a*," *Advances in Neural Information Processing Systems*, pp. 1539–1546, 2002.
[16] S. Koenig, M. Likhachev, and D. Furcy, "Lifelong planning a*," *Artificial Intelligence Journal*, vol. 155(1-2), pp. 93–146, 2004.
[17] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun, "Anytime dynamic a*: An anytime, replanning algorithm," in *Proceedings of the International Conference on Automated Planning and Scheduling 2005*.
[18] J. T. Thayer and W. Ruml, "Faster than weighted a*: An optimal approach to bounded suboptimal search," in *Proceedings of the International Conference on Automated Planning and Scheduling 2008*.