University of Cincinnati

Date: 2/12/2013

I, Weiya Yue Ph.D., hereby submit this original work as part of the requirements for the degree of Doctor of Philosophy in Computer Science & Engineering.

It is entitled:

Improving Dynamic Navigation Algorithms

Student's name: **Weiya Yue Ph.D.**

This work and its defense approved by:

Committee chair: John Franco, PhD

Committee member: Raj Bhatnagar, PhD

Committee member: Yizong Cheng, PhD

Committee member: Wen Ben Jone, PhD

Committee member: John Schlipf, PhD

3186

Improving Dynamic Navigation Algorithms

A dissertation submitted to the
Graduate School
of the University of Cincinnati
in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

in the Department of Computer Science
of the College of Engineering and Applied Science
by

Weiya Yue

M.S. SUN YAT-SEN University
June 2006

Committee Chair: John Franco, Ph.D.

Abstract

Navigation algorithms for advanced autonomous vehicles, such as an unmanned automobile or airplane, require improved response times to complete numerous tasks that are still only imagined. Existing navigation algorithms tend to be incremental, do not take full advantage of accumulated information to compute a next move, and tend to be too eager in recomputing much information when a new optimal path must be found. The result is unnecessary per-round state recalculations that slow the algorithms considerably. The formalization of a general framework for dynamic planning algorithms, aimed at eliminating such recalculations by considering the relationship between optimal solutions between rounds, is proposed. The framework is based on our successful work which improved the speed of the well-known D*lite algorithm by up to eight times. The expected direct result of this research is to improve the performance of navigation algorithms in various terrains. As an example, the framework is applied to the Anytime D* algorithm, a variant of D*Lite, to get a new algorithm, called IAD*, which is an order of magnitude faster than Anytime D*. Moreover, the IAD* algorithm and the AWA* algorithm are combined to form another Anytime variant, and another new dynamic anytime algorithm, called DAWA*, the first dynamic anytime algorithm able to utilize time resource continuously. These improvements show the extensibility and robustness of the proposed framework.

Acknowledgement

This dissertation not only presents my academic result submitted for getting my Ph.D. degree, but also reminds me of a most beautiful memory in my life involving trust, support and love. It is no doubt that I should express my truly gratitude to my dearest professors and my precious parents and friends who expand my knowledge and rich my life.

My deepest gratitude is to my advisor Professor John Franco. He is the reason I made up my mind to go abroad and pursued a higher academic goal when I was at a crossroad in life ambivalent about which direction I should step forward. He discussed with me over correspondence on many of his ideas which inspiringly fascinated me. I am very thankful to him for guiding me to the realm of artificial intelligence I am really interested in and supplying me with all his knowledge of the area helping me track the newest idea. Through the years of my graduate study his patience on advising my work strengthens my confidence and his always being enthusiastic on work always inspires me to do better. If ever I could have a chance to conduct research with my student, I would emulate his professional spirit that I learned from him and make every effort to be a good advisor.

I also own my gratitude to Dr. Raj Bhatnagar, Dr. Yizong Cheng, Dr. Wen-Ben Jone, and Dr. John Schlipf for taking their time being my committee of my dissertation. I also feel deep gratitude to Dr. Dieter Schmidt, Dr. Gregory White, Dr. Jerome Paul, Dr. Kenneth Berman, and Dr. Prabir Bhattacharya. Their numerous discussions and lectures, their words for encouragement greatly inspired me.

The smooth going of my dissertation is indispensable from staff and faculty in compute science department, staff in college of engineering, and also staff in ISSO office. Their in-time and helpful reminder and advice when I get problems on paper work or application procedure cleared many puzzles and saved me a lot of effort so that I can mostly focus on my thesis. Special thanks to Graduate Studies Program Director Julie Muenchen for her work on updating and handling my all kinds of documents.

The friendship accrued through the graduation study years greatly riches my life. I will never forget my good buddies: Qiang Han, Chen Lu, Hailong Li, Tao Ma, Dapeng Wu, Sean Weaver, Michael Koril, Qi Zhang, Weichen Ouyang, Nan Wang etc. I appreciate their company when I was happy and especially when I was unhappy. The good memories with them are deeply in my heart.

My family have been who I value the most and I want to bring happiness to. My parents, my elder brother, and my wife. They are always on my side to do what my passion is in, and seldom push me to compromise with things I do not like. Thanks to my parents and my elder brother, their emotional and financial back-up help me get through a lot of tough time. Thanks to my wife Weiwei Cao, being my partner of my research and more a part of my life. She touches my heart and brings me love, making my life complete and happy.

Contents

1 Introduction

Advances in agent 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 [14, 15] represents the state-of-the-art in such replanning algorithm development. A descendant of the A* and D* [31, 32] 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 [1, 37].

The terrain information that is used by D* Lite is represented abstractly as a directed graph $G(V, E)$ with distinguished start or source vertex v_s, goal vertex v_g, and positive integer costs $c : V \times V \mapsto Z^+$ on edges. An "agent" 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 agent'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 an agent from v_s to v_g. The problem is complicated by the fact that edge cost changes are not predictable.

It is unlikely that D* Lite will find the lowest cost sequence of transitions that advances the agent from v_s to v_g because it never has complete information about edge cost changes until the last transition. Moreover, current edge costs are known to D* Lite only within the agent'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 agent, which we will designate below as v_c. But D* Lite can always find the lowest cost sequence from v_c to v_g based on the known edge costs within the view and assuming that current estimates of other edge costs are their actual costs.

Several other planning algorithms have been proposed specifically to deal with an environment where graph changes are expected during agent transit and the view of the agent is limited. The Anytime A* (ARA*) [21,22] returns a suboptimal solution quickly with control over a sub-optimality bound which can be improved during search until time runs out. This algorithm may be useful when the time available for recomputation is limited. A BBD based approach proposed in [39] memoizes and reuses learned information to save

space and recomputation.

Despite orders of magnitude improvements in speed over the years, planning algorithms still have trouble meeting the demands of the time-critical and ever-changing nature of navigation. In prior work we have observed that the relationship between optimal solutions across rounds can be exploited in new ways to further improve the performance of D* Lite, which is generally considered to be an important navigation algorithm [40,41,43]. Recently, we have observed that the technique we used to get this improvement can be generalized to improve various dynamic planning algorithms. We propose to formalize this technique as a general framework and use this framework to improve the performance of various navigation algorithms. This framework will be the main contribution of the thesis. An important by-product of this result will be the immediate improvement of several known navigation algorithms.

To help motivate and understand this framework several navigation algorithms are described in this proposal. Algorithms D* Lite, Anytime A*, and other algorithms that they depend on are described in Section 3. In Section 4, our Improved D* Lite algorithm (ID* Lite) is described. ID* Lite applies new, efficient methods to avoid unnecessary recomputations when graph changes are detected. According to results of our experiments on several kinds of changing graph topologies ID* Lite can be as much as eight times faster than D* Lite. Ideas that ID* Lite is built around naturally lead to a better skeleton for incremental planning algorithms in general because little domain knowledge is used during recomputations. We show that this skeleton can be used to also improve Anytime D*.

2 Background

The mathematical structure that will be the foundation for our framework is the finite weighted directed graph which will be denoted in the usual way for vertex set V and edge set E as $G(V, E)$ with cost function $c : E \mapsto Z^+$ that defines positive edge costs. For a pair of vertices $\{a, b\} \subset V$, an edge that is directed from a to b is denoted $\langle a, b\rangle$ and that edge, if it exists, will be said to be an *out-edge* of a and an *in-edge* of b and a will be said to be the source of that edge and b will be said to be the target of that edge. In the graphs that are used in modeling navigation problems V remains fixed but E may change with time and the mapping c may change with time. In addition, two of the vertices of V are designated as the *starting and goal* vertex, denoted v_s, v_g respectively. A particular collection of values of V, E, v_s, v_g and mapping c define a *state* of the model.

The problem considered here is to find a trajectory of an agent from a given starting vertex to the goal vertex. A trajectory is a set of legal moves of the agent. Agent moves and state changes are assumed to occur at discrete points called epochs. The result of an agent's move is instantaneous and a state change may not happen at every epoch. At an initial state $S = \langle V, E, v_g, v_c, c\rangle$ an agent is placed on some (starting) vertex $v_c = v_s \in V$ and a *trajectory cost* c is assigned the value 0. At any state S, an agent located at vertex $v_c \in V$ may move legally to any vertex w such that $\langle v_c, w\rangle \in E$ or the agent may stay put. The agent must stay put at an epoch if there is no out-directed edge from v_c. If the agent moves to $w \neq v_c$ then w becomes v_c and $c(\langle v_c, w\rangle)$ is added to the trajectory cost as the outcome of the move. When and if $v_c = v_g$ (the agent reaches the goal vertex, wherever it happens to be), the movement of the agent stops and the trajectory cost is the cost of moving the agent from the starting vertex in the initial state to the goal. The problem is to plan a trajectory that will have least cost among all possible trajectories.

The navigation algorithms considered here account for state changes and the movement of an agent through vertices while those changes take place by making updates during a round of computation. A current (best) path to the v_g is maintained from round to round and may change as state changes and the agent is made to move one edge along that path. If the current path is no longer feasible or not optimal, the current best path is replaced by a new feasible best path before moving the agent. Doing so may require an amount of computation that ranges from negligible to extreme. To clarify,

especially for following sections, given a state S, a *path* from vertex v_c to vertex v_g is a sequence $\{e_1, \ldots, e_k\}$ of edges in E where the source of e_1 is v_c, the target of e_k is v_g, and for all $1 \leq i < k$ the target of e_i is the source of e_{i+1}. A path from v_c to v_g in current state S will often be denoted $P^S_{v_c,v_g}$ or P_{v_c,v_g} for brevity. Alternatively, for given S, a path between vertices v_c and v_g may be represented as a sequence $\{v_c, v_1, \ldots, v_k, v_g\}$ of vertices where, for $1 \leq i < k$, v_i is the target of one edge and the source of another edge in E, and edges $\langle v_c, v_1\rangle$ and $\langle v_k, v_g\rangle$ exist in E.

For any vertex v, function c doubles as defining the cost of a path according to $c(P_{v,v_g}) = \sum_{e \in P_{v,v_g}} c(e)$. If $\mathcal{P}_{v,v_g}$ is a set of paths from v to vertex v_g, and if $P^* \in \mathcal{P}_{v,v_g}$ and $c(P^*) \leq c(P)$ for all $P \in \mathcal{P}_{v,v_g}$, then P^* is said to be an optimal or best path between vertices v and v_g. The cost of the optimal path between a vertex v and the goal will be denoted $g^*(v)$. Navigation algorithms presented here will maintain a *current* collection of paths to the goal from round to round. The cost of a *best* path, as currently determined by the algorithm, from any vertex v (not just v_c) will be denoted $g(v)$. An estimate of the cost between two vertices u, w is denoted by $h(u, w)$, and for convenience an estimate of the cost of the best current path from v to v_g will be denoted $h(v)$. Although g^*,g, and h are not shown to be functions that depend on time, it should be emphasized that the vertex v_c and path costs may change over time.

Two functions are defined to express possibilities for agent movement on a round. In state S, denote by $succ(v)$ the subset of all vertices of V of S that are targets in edges whose source vertex is v and by $pred(v)$ the subset of all vertices that are a source in some edge whose target is v.

Algorithm A*

```
01.  Priority Queue:  OPEN = ∅; CLOSED = ∅;
02.  OPEN.insert(v_s, f(v_s));
03.  while OPEN ≠ ∅:
04.      [v, f(v)] = OPEN.top();
05.      OPEN.remove(v); CLOSED.insert(v);
06.      if (v == v_g) PrintPath();
07.      for every vertex u ∈ succ(v):
08.          if (u ∈ CLOSED and g(u) > g(v) + c(⟨v, u⟩))
09.              g(u) = g(v) + c(⟨v, u⟩);
10.              CLOSED.remove(u); OPEN.insert(u, f(u));
11.          else if (u ∈ OPEN)
12.              g(u) =min(g(u), g(v) + c(⟨v, u⟩));
13.              OPEN.update(u, f(u));
14.          else
15.              g(u) = g(v) + c(⟨v, u⟩); OPEN.insert(u, f(u));
```

Figure 1: Pseudo code for the A* algorithm

2.1 The A* algorithm

This section presents the classic A* search algorithm [12] [7] which has been used for decades on graph traversal and pathfinding problems of fixed state due to its performance characteristics and accuracy. This algorithm is the basis for all the other algorithms presented here. Its key property of interest is the use of a distance-plus-cost heuristic function, denoted $f(v)$, which is used to choose the *best* vertex through which to extend the search for a best path to the goal. The function $f(v) = g(v) + h(v)$ where $g(v)$ is cost of reaching v from the start vertex v_s over the current *best* path and $h(v)$ is an estimate, but not over-estimate, of the cost of reaching the goal vertex from v.

A priority queue, which will be called `OPEN` here, is maintained to support the search which progresses in a best-first manner. Every node of `OPEN` contains a vertex v with associated priority $f(v)$. The nodes in `OPEN` are in non-decreasing order of $f(v)$. For convenience of computation, a temporary list of vertices, called `CLOSED` here, holds all vertices that have been considered for expansion in the search: that is, vertices that have been re-

moved from OPEN. A next vertex v for consideration is taken from the top of OPEN: that is, the next vertex is the lowest priority node of OPEN. Each potential successor u of v is tested to see whether $g(u) > g(v) + c(\langle v, u\rangle)$. If so, $g(u)$ is reduced to that amount and u is placed in OPEN with newly reduced, associated $f(u)$, if it was not there already. Pseudo code for the A* algorithm is shown in Figure 1, in which every vertex v is initialized with $g(v) = \infty$ except $g(v_s) = 0$. Function $PrintPath()$ can generate one path when the algorithm is finished. To generate one path $P = \{v_s = v_1, v_2, ..., v_d = v_g\}$, we only need to traverse back from v_g by setting $v_d = v_g$, then $v_i = argmin_{v \in pred(v_{i+1})}(g(v) + c(\langle v, v_{i+1}\rangle), 1 \leq i < d$.

The heuristic function h has a major impact on the performance characteristics and optimality of the A* algorithm. The A* algorithm is guaranteed to find a least-cost path from start vertex to goal if $h(v) \leq g^*(v)$ for all v. In that case, h is said to be *admissible*. If for any vertex $v \neq v_g$, and every $u \in succ(v)$, $h(v) \leq h(u) + c(\langle v, u\rangle)$, and if $h(v_g) = 0$, then A* is optimally efficient: that is, it expands the minimal number of nodes which are necessary to guarantee optimality and completeness. A heuristic function with this property is said to be *consistent*. Most commonly used heuristics are consistent. However, a heuristic that is not consistent can sometimes be employed to reduce search further. For example, the inflated heuristics used in [25, 28] expand fewer vertices than well-known consistent heuristics in some application. The use of inconsistent heuristics has been explored at length in [3, 4]. If h is admissible and $f(v) = g(v) + \epsilon \cdot h(v)$, then the path returned by the A* algorithm is guaranteed to be ϵ-sub-optimal [5]: that is, $g(v_s) \leq \epsilon \cdot g^*(v_s)$. The Anytime A* algorithm [21, 22] has this property, and Anytime Dynamic A* [20], which can be treated as one descendent of D* Lite and Anytime A*, also has this property. Observe that if $h(v) = 0$ for all vertices then the A* algorithm is Dijkstras algorithm.

2.2 D* Lite Algorithm

The D* algorithm [31,32], introduced in 1995, is the first dynamic incremental algorithm to be applied to autonomous navigation with an improvement in performance of two orders of magnitude over previously used algorithms. This was followed in 2002 by the D* lite algorithm [14, 15], a descendant of D*, which is considered to be the most successful variant currently [1]. D* lite improves upon D* by cleverly avoiding full round re-calculations under certain conditions. D* lite is easily implemented, expanded to be combined with various domain knowledge and its "experimental properties show that D* lite is at least as efficient as D*" [15]. It has been used successfully in a variety of roles. For example, a prototype system tested on the Mars rovers Opportunity and Spirit and the navigation system of the winning entry in the DARPA Urban Challenge [37] use the D* lite algorithm.

Whereas the A* algorithm is intended to be used on fixed state models, D* and D* lite have been designed for models where arbitrary state changes can occur. There is no algorithm for solving the general problem of finding the lowest cost path in the presence of state changes and where the agent has a limited view so D* and D* lite can, at best, return an agent trajectory whose cost approximates that of a lowest cost trajectory. To do this some assumptions must be made about the vertices and edges that are outside the view of (and not accessible to) the agent and some next-move-judgment algorithm for applying those assumptions must be designed and implemented. Typically, these assumptions include a fixed agent view whose maximum distance from the agent, in pathlength, is call the *sensor radius*. In D* lite a path that is optimal based on current knowledge and assumptions is computed and that path is followed until an assumption is violated at which time a new optimal path is computed, and so on, until v_g is reached. Every transition from one vertex to a neighboring vertex corresponds to a round of D*lite in which the current path is checked and a new optimal path is computed, if necessary.

The next-move-judgment algorithm that is used by D* lite combines elements of the A* algorithm with elements of the D* algorithm. It uses the functions g and h as described earlier for A*, except with some small differences, plus a function rhs that returns a one-level lookahead cost for a vertex. Function g is now used to estimate the cost of a path from the agent's current position (v_c) to v_g and not from v_s to v_c as is done in A* and function h is used as a heuristic for path cost from v_s to v_c and not from v_c to v_g as is done

in A*. Hence, 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 by following the maximum-g-decrease-value vertices one by one. 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 $g(v) = rhs(v)$. During the update changes in $g(v)$ will be propagated through all neighbors of v. Propagation continues until a new "best" path has been found. During propagation vertices that are found to be consistent are not updated; this is partly why D* Lite is faster than D*.

The pseudo code of D* Lite listed in Figure 2 is exactly the same as in [14] with only minor changes to representations.

Procedure **CalculateKey**(v) returns the priority queue key value of vertex v which is a function of $g(v)$, $rhs(v)$, $h(v)$, and k_m, where k_m is one value used to reduce the number of resorting priority queue introduced in [31]. Procedure **Initialize**() initializes g, rhs values explicitly. Procedure **UpdateVertex**(v) is used to update v's g and rhs values and to keep v in the right position of priority queue $OPEN$.

Procedure **ComputeShortestPath**() is the most important function. As discussed above, the critical point of D* Lite is to make sure all the changes are propagated until one optimal path has been generated with all vertices on it are consistent. If vertex v has been affected by changes of edges, then v can be overconsistent or underconsistent. If v is overconsistent this change can be guaranteed to be propagated out. If v is underconsistent, the procedure must set $g(v) = \infty$ in order to propagate this change out. This corresponds to lines 14-26 in Figure 2. It is important to observe that the keys in OPEN are updated lazily so by the time line 11 is executed, the

priority queue key value of the top of OPEN may be smaller than its actual value as computed by **CalculateKey**(v) [31]. In that case, the vertex at the top of OPEN cannot be the actual lowest priority vertex in OPEN so we avoid searching on this vertex (Lines 14-26) and just do an update on the priorities of vertices in OPEN (Line 13).

Procedure **Main**() gets the agent to move to the next vertex after the "best" path has been calculated, then scans the graph for changes and recomputes vertex values when changes are found. In Figure 2, k_m is used to reduce the time needed to reorder the priority queue. This addition was first introduced in [31].

In some variations of the D* family of algorithms, for example delayed D* [9], updating underconsistent vertices is delayed since, intuitively, it is more likely that the "best" path contains overconsistent vertices. Delayed D* performs well under some conditions, but if there are many overconsistent vertices, its performance suffers. Also, because delayed D* needs to determine whether a path is correct by checking whether there is any underconsistent vertex on it, a recomputation may have to be repeated, possibly resulting in worse performance than D* Lite which explores all inconsistent vertices together. For more details refer to [9].

Despite the clever design of D* Lite we envision a significant improvement in performance is possible for two reasons. First, it can be shown that some calculations that are performed when recomputing a "best" path are unnecessary, and these calculations can be avoided safely. Second, it has been observed that in many problems there is typically more than one "best" solution and, if one of the alternative "best" paths is not affected by changes, it can be found efficiently. If the new alternative "best" path so found is no better than the old one, the alternative can be used in place of the old one without a recomputation and with a guarantee of optimality. Moreover, if the new alternative path is better than the old one, only a partial recomputation is needed. These two reasons are the basis of the research we propose both to improve D* Lite and its other variants.

Procedure CalculateKey(v):
01. return $[\min(g(v), rhs(v)) + h(v) + k_m,\ \min(g(v), rhs(v))]$

Procedure Initialize():
02. OPEN= $\emptyset$; $k_m = 0$;
03. for all $v \in V$, $rhs(v) = g(v) = \infty$;
04. $rhs(v_g) = 0$;
05. OPEN.insert($[v_g,\ [h(v_g), 0]]$);

Procedure UpdateVertex(v):
06. if ($g(v) \neq rhs(v)$ and $v \in$ OPEN) OPEN.update($[v,$ **CalculateKey**$(v)]$);
07. else if ($g(v) \neq rhs(v)$ and $v \notin$ OPEN) OPEN.insert($[v,$ **CalculateKey**$(v)]$);
08. else if ($g(v) == rhs(v)$ and $v \in$ OPEN) OPEN.remove(v);

Procedure ComputeShortestPath():
09. while (OPEN.TopKey() $<$ **CalculateKey**(v_s) or $rhs(v_s) > g(v_s)$)
10. u = OPEN.top();
11. k_{old} = OPEN.TopKey(), k_{new} = **CalculateKey**(u);
12. if ($k_{old} < k_{new}$)
13. OPEN.update($[u,\ k_{new}]$);
14. else if ($g(u) > rhs(u)$)
15. $g(u) = rhs(u)$;
16. OPEN.remove(u);
17. for all $v \in pred(u)$
18. if ($v \neq v_g$)$rhs(v) = \min(rhs(v), c(\langle v, u\rangle) + g(u))$;
19. **UpdateVertex**(v);
20. else
21. $g_{old} = g(u)$;
22. $g(u) = \infty$;
23. for all $v \in pred(u) \cup \{u\}$
24. if ($rhs(v) == c(\langle v, u\rangle) + g_{old}$)
25. if ($v \neq v_g$)$rhs(v) = \min_{s \in succ(v)}(c(\langle v, s\rangle) + g(s))$;
26. **UpdateVertex**(v);

Procedure Main():
27. $v_{last} = v_s$;
28. **Initialize**();
29. **ComputerShortestPath**();
30. while ($v_s \neq v_g$)
31. $v_s = argmin_{v \in succ(v_s)}(g(v) + c(\langle v_s, v\rangle))$;
32. Move to v_s;
33. Scan graph for changes;
34. if changes are found
35. $k_m = k_m + h(v_{last}, v_s)$;
36. $v_{last} = v_s$;
37. for every edge $\langle u, v\rangle$ with changed edge cost
38. $c_{old} = c(\langle u, v\rangle)$;
39. Update edge cost of $\langle u, v\rangle$;
41. if ($c_{old} > c(\langle u, v\rangle)$)
42. if ($u \neq v_g$)$rhs(u) = \min(rhs(u), c(\langle u, v\rangle) + g(v))$;
43. else if ($rhs(u) == c_{old} + g(v)$)
44. if ($u \neq v_g$)$rhs(u) = \min_{s \in succ(u)}(g(s) + c(\langle u, s\rangle))$;
45. **UpdateVertex**(u);
46. **ComputeShortestPath**();

Figure 2: Pseudo code for D* Lite

2.3 Anytime Planning Algorithm

The goal of the algorithms discussed in previous sections is to determine the "best" path for the agent given current, incomplete knowledge of the environment in which the agent must move. In some applications these algorithms are too slow and are unable to return any path at all when path recomputation is necessary. Time-limited search algorithms, called *anytime planning* algorithms, have been developed to be used in such cases. The basic idea is to progressively replace a stored path with a better path when one is discovered during search until the time available for search expires [10, 11, 45, 46]. Then the stored, probably sub-optimal path is the one that is used. In [6, 47], it is shown that this strategy can be made very efficient and in [3, 4, 10, 19, 25, 28, 29], 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 $f = g + h$. After that, many works [5] [18] [19] [23] [26] [27] [28] [29] [30] etc. have been done on researching the effect of weighting g and h, i.e. relating separated parameters with g and 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. $g(v_g) \leq \epsilon \cdot g^*(v_g)$ [5]. 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. Next we will mainly introduce two anytime algorithms introduced in [10] and [21]. The two algorithms both use a similar idea from Weighted A* as [5].

2.3.1 Anytime Weighted A*

In [10], one general method to transform heuristic search algorithms to anytime algorithms is proposed, in which it is shown how to transform Weighted A* [5] to be Anytime Weighted A*, and also the transformation of Recursive Best-First Search algorithm [19] to an anytime version. Here we will only introduce Anytime Weighted A* algorithm. 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. In Figure 3 the pseudo code of Anytime Weighted A* is listed.

Anytime Weighted A* does initialization on most variants as A* algo-

```
    Algorithm Anytime Weighted A*
01. Priority Queue: OPEN = ∅, CLOSED = ∅; PATH p = NULL; ERROR=∞; Constant ε;
02.     OPEN.insert(< v_s, f'(v_s) >);
03.     while OPEN ≠ ∅ and not interrupted by outside:
04.         < v, f'(v) >= OPEN.top(); OPEN.remove(v);
05.         if ( (p == NULL) or (f(v) < c(p)) )
06.             CLOSED.insert(v);
07.             for every vertex u ∈ succ(v) and g(v) + c(v, u) + h(u) < c(p):
08.                 if(v == v_g)
09.                     f(u) = g(u) = g(v) + c(v, u);
10.                     Update p to be the path passing through v; PrintPath();
11.                 else if u ∈ CLOSED and g(u) > g(v) + c(e(v, u))
12.                     g(u) = g(v) + c(e(v, u)); f'(u) = g(u) + ε · h(u); f(u) = g(u) + h(u);
13.                     CLOSED.remove(u), OPEN.insert(< u, f'(u) >);
14.                 else if(u ∈ OPEN)
15.                     g(u) = min(g(u), g(v) + c(e(v, u))); f'(u) = g(u) + ε · h(u); f(u) = g(u) + h(u);
16.                     OPEN.update(< u, f'(u) >);
17.                 else
18.                     g(u) = g(v) + c(e(v, u)); f'(u) = g(u) + ε · h(u); f(u) = g(u) + h(u);
19.                     OPEN.insert(< u, f(u) >);
20.     if (OPEN == ∅) ERROR = 0;
21.     else ERROR = c(P) − min_{v∈OPEN}(f(v));
22.     PrintPath();
```

Figure 3: Pseudo Code of Anytime Weighted A* Algorithm

rithm, 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 [11] [13]. In Figure 3, at *line* 05, there is one judgement about whether to expand one $OPEN$ vertex which does not exist in A*. If $p! = NULL$, $c(p)$ is equal with $f(v_g)$. If current there is one path p, and $f(v) < c(p)$, i.e. v can not reduce value $f(v_g)$ to improve current path, then p will not expanded; at *line* 07, with the same reason, for one $u \in succ(v)$, if its f value is updated by v and can not be smaller than $c(p)$, it is not inserted into $OPEN$.

Here notice that, because Anytime Weighted A* does not have the properties of A* by using consistent heuristic function, it is not guaranteed that

vertices in $OPEN$ have their f values no bigger than the top vertex v's f value. Hence, when v_g is expanded, the returned path can not be guaranteed to be optimal, and some vertices in $OPEN$ can be deleted whose f values are $\geq f(v_g)$. The reason not to remove them after p improved or to remove those vertices at *line* $14-16$ is that we do not need to do extra resorting actions on $OPEN$ until such vertices rise to the top of $OPEN$. This enables us to improve current path as soon as possible. For the same reason, at *line* $08-10$, when a vertex is generated instead of it is expanded in A*, it is tested whether is v_g, and if yes, current path is reported to be improved immediately.

Anytime Weighted A* has been proved its terminability and optimality. In Figure 3, at *line* 20, if $OPEN == \emptyset$, $ERROR$ is set to be 0, which means current solution is optimal; otherwise at *line* 21, $ERROR$ is set to be $c(P) - min_{v \in OPEN}(f(v))$, which gives one upper bound of variance between current path and optimal path, i.e. $ERROR \geq c(p) - g^*(v_g)$ where $g^*(v_g)$ is the weight of optimal path. If the heuristic h is admissive, then Anytime Weighted A* can guarantee ϵ sub-optimality of every returned path. For more details, please refer to [10].

2.3.2 Anytime A* with Control of Sub-optimality

Essential to anytime planning algorithms is a measure of the closeness of the cost of the stored path to the cost of an optimal path. An effective cost measure contributes to making a more precise decision about when to recompute for a better path and choosing a better path. In other words, it provides a quantitative understanding of the error/available-time tradeoff. A simple cost measure exists for the A* algorithm if f is modified slightly: if h is admissive and $f = g + \epsilon \cdot h$, then the recomputed path is guaranteed to be ϵ sub-optimality, that is, $g(v_g) \leq \epsilon \cdot g^*(v_g)$ [5, 10]. This is the basis for one of the weighted A* algorithm variants mentioned above. Obviously, ϵ can initially be relatively large and then incrementally reduced during search. However, doing so can result in significantly many duplicated recomputations that prevent ϵ from reaching a satisfactory value before time expires. The incremental anytime algorithm Anytime Repairing A* (ARA*) algorithm [21, 22] was developed to mitigate this problem.

The pseudo code of ARA* that is shown in Figure 4 is replicated from [21] with slight representation differences. The heuristic function h used in ARA* is assumed to be consistent. The algorithm runs the weighted A* algorithm

many times, starting 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. Since f depends on ϵ, the heuristic function is said to be an *inflated heuristic*. 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 [16, 17]. In A*, all vertices in OPEN are treated as inconsistent vertices. Since such inconsistency must be overconsistency, $\epsilon \cdot h$, $\epsilon > 1$, is not guaranteed to be consistent even if h is consistent. So, a vertex that is expanded from OPEN, and that would be made consistent if $\epsilon = 1$, may have to be re-inserted into OPEN and expanded again, possibly several times, before becoming consistent, if $\epsilon > 1$. This re-expansion may be avoided by forcing a maximum of one expansion per vertex per round and leaving expanded vertices that remain inconsistent for the next round. In ARA*, all inconsistent vertices are also only overconsistent. In Figure 4, function **Fvalue**(v) does the same job as $f(v)$ in the A* algorithm. Function **ImprovePath**() is the key function of ARA*. Because $\epsilon \cdot h$ cannot guarantee consistency, CLOSED is used in ARA* to keep all expanded and consistent vertices. The priority queue INCONS is used in ARA* to keep all expanded but inconsistent vertices. As in A*, OPEN is used to keep all unexpanded and inconsistent vertices. Thus, after a round of computation, OPEN $\cup$ INCONS is the set of all the inconsistent vertices and CLOSED is the set of all expanded and consistent vertices. As a result of reusing previous information, when it is not the first round computation, v_g may be consistent already before computation. Line 02 of ARA*, which is missing from A*, checks this possibility. Function **Main**() executes **ImprovePath**() multiple times until available time expires or $\epsilon = 1$. Before calling **ImprovePath**() on a round, INCONS is dumped into OPEN. It is worth noting the variable ϵ' in function **Main**(), which was first introduced in [10]. The purpose of introducing ϵ' is to estimate a better current bound of sub-optimality.

As the D* Lite algorithm can be treated as a dynamic version of the Lifelong A* algorithm, the Anytime D* algorithm can be thought of as a dynamic version of the Anytime A* algorithm. The Anytime D* algorithm is discussed in detail in the next section.

```
Procedure Fvalue(v):
01.  return g(v) + ε · h(v);

Procedure ImprovePath():
02.  while (Fvalue(v_g) > OPEN.TopValue())
03.      v = OPEN.top(), OPEN.remove(v);
04.      CLOSED.insert(v);
05.      for each vertex s ∈ succ(v)
06.          if s was not visited before then
07.              g(s) = ∞;
08.          if g(s) > g(v) + c(⟨v, s⟩)
09.              g(s) = g(v) + c(⟨v, s⟩);
10.          if s ∉ CLOSED
11.              OPEN.insert([s, Fvalue(s)]);
12.          else
13.              INCONS.insert(s);

Procedure Main():
14.  g(v_g) = ∞, g(v_s) = 0;
15.  OPEN = CLOSED = INCONS = ∅;
16.  OPEN.insert([v_s, Fvalue(v_s)]);
17.  ImprovePath();
18.  ε' = min(ε, g(v_g)/min_{v∈OPEN∪INCONS}(g(v) + h(v)));
19.  Print current ε'-suboptimal solution;
20.  while (ε' > 1)
21.      decrease ε;
22.      Move vertices from INCONS into OPEN;
23.      Update the priorities for all v ∈ OPEN according to Fvalue(v);
24.      CLOSED = ∅;
25.      ImprovePath();
26.      ε' = min(ε, g(v_g)/min_{v∈OPEN∪INCONS}(g(v) + h(v)));
27.      Print current ε'-suboptimal solution;
```

Figure 4: Pseudo code for Anytime A*

2.4 Anytime D*

The preceding sections have discussed several planning algorithms and the motivation for their development. Incremental A* [16, 17] is intended for applications where replanning is done between the same pair of source and destination vertices but under changing circumstances. Anytime A* [21] [22] has been developed for applications where optimality is not as critical as response time, incremental changes in the environment are expected, and replanning due to those changes is done between a fixed pair of vertices. D* [31, 32], and D* Lite [14, 15] are appropriate for dynamic navigation by autonomous vehicles, intelligent robots, etc.

In this section yet another planning algorithm, called Anytime D*, is discussed. This algorithm, introduced in [20], is intended 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 A* 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*. The difference is that in Anytime A* inconsistent vertices are only over-consistent, but in Anytime D*, due to edge cost changes, there may be under-consistent vertices and different rules are employed when propagating information about such vertices. These are stated in the following description of the algorithm.

The Anytime D* algorithm is listed in Figure 5. In function **Main**(), lines 23-24 define and initialize ϵ, and the priority queues OPEN, CLOSED, and INCONS, and initialize values for g and rhs of the source and destination vertices. The initial value of ϵ_0 is relatively large in order to make sure some path is returned quickly by lines 27-28. Lines 28-42 incrementally improve on the initial or current path until an optimal path is computed ($\epsilon = 1$). A new path is computed and state updated when any edge cost changes are observed. If the changes are small then ϵ is decreased (line 35) but when significant changes are observed ϵ is increased or replanning is restarted (line 33). When *epsilon* reaches 1, an optimal solution is obtained and the algorithm suspends itself until additional changes in edge costs are observed.

If edge cost changes are substantial, instead of updating current states, it may be less expensive to increase ϵ or replan from scratch as described in Line 33. Thus, there needs to be some way to determine when changes are substantial and some way to determine whether a restart should be per-

formed.

Consider the measure for determining what a substantial change is first. This measure is application-dependent [20]. If many re-calculations have been made there are many inconsistent vertices (in OPEN) that will not be part of a best path and should be eliminated from OPEN to save unnecessary computation and memory.

Returning to Figure 5, function **ComputeorImprovePath**() is similar to the function of the same name in D* Lite. Function **UpdateVertex**(s) updates one vertex in the same way as Anytime A* 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**(). This guarantees the solution generated satisfies ϵ-suboptimality. Function **key**(s) is different from function **Fvalue**(v) in Figure 4 because in Anytime D* there are under-consistent vertices to consider and in order to propagate information about under-consistent vertices $g(s) + h(s)$, which is guaranteed to be smaller than **key**(v_s), is used instead of $g(s) + \epsilon \cdot h(s)$ in Line 04. Otherwise, not only the path returned is not ϵ-suboptimal, but also some vertices on the path may be inconsistent.

In order to give an agent the ability to navigate, Line 28 in Figure 5 is replaced with **fork**(**MoveAgent**()); while($v_s \neq v_g$); in [20] where **MoveAgent**() is shown in Lines 43-46. Running in parallel with lines 28-42, **MoveAgent**() allows the agent to move along the current best path while the current solution is being improved. However, the case where the vertex that the agent is about to move to is v but the improved best path no longer contains v may present a problem. A solution is to allow the agent to land on v, set $v_s = v$, and recalculate a new best path.

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):
05. if s has not been visited
06. $g(s) = \infty$;
07. if ($s \neq v_g$)$rhs(s) = \min_{s' \in succ(s)}(c(\langle s, s' \rangle) + g(s'))$;
08. if ($s \in$ OPEN) OPEN.remove(s);
09. if ($g(s) \neq rhs(s)$)
10. if ($s \in$ CLOSED)
11. OPEN.insert($[s, \mathbf{key}(s)]$);
12. else
13. insert s into INCONS;

Procedure ComputeOrImprovePath():
14. while (OPEN.TopKey() $<$ **key**(v_s) OR $rhs(v_s) \neq g(v_s)$)
15. s = OPEN.Top(), OPEN.remove(s);
16. if ($g(s) > rhs(s)$)
17. $g(s) = rhs(s)$;
18. CLOSED.insert(s);
19. for all $s' \in pred(s)$ **UpdateVertex**(s');
20. else
21. $g(s) = \infty$;
22. for all $s' \in pred(s) \cup \{s\}$ **UpdateVertex**(s');

Procedure Main():
23. $g(v_s) = rhs(v_s) = \infty, g(v_g) = 0, rhs(v_g) = 0, \epsilon = \epsilon_0$;
24. OPEN = CLOSED = INCONS = $\emptyset$;
25. OPEN.insert(v_g);
26. **ComputeOrImprovePath**();
27. publish current ϵ-suboptimal solution;
28. repeat the following:
29. for all directed edges $\langle u, v \rangle$ with changed edge costs
30. Update the edge cost $c(\langle u, v \rangle)$;
31. **UpdateVertex**(u);
32. if significant edge cost changes were observed
33. increase ϵ or replan from scratch;
34. else if ($\epsilon > 1$)
35. decrease ϵ;
36. Move states from INCONS into OPEN;
37. Update the priorities for all $s \in$ OPEN according to **key**(s);
38. CLOSED = $\emptyset$;
39. **ComputeOrImprovePath**();
40. publish current ϵ-suboptimal solution;
41. if ($\epsilon == 1$)
42. wait for changes in edge costs;

Procedure MoveAgent():
43. while ($v_s \neq v_g$)
44. wait until a plan is available;
45. $v_s = argmins_{s \in succ(v_s s)}(c(\langle v_s, s \rangle) + g(s))$;
46. move agent to v_s;

Figure 5: Pseudo code for Anytime D*

In this paper, we will give a framework to speed up dynamic navigation algorithms including both optimal and sub-optimal searching algorithms. In order to show this, several new algorithms are composed as below:

1. ID* Lite algorithm: [40,41] This algorithm improves D* Lite algorithm with guaranteed no more heap operations used than D* Lite algorithm during searching. Every time when changes observed by the agent, ID* Lite will try to find an alternative instead of applying a full recalculation. In Section 3.1, ID* Lite algorithm is fully introduced and discussed. Also, via experiments, we will see ID* Lite performs better than D* Lite on random benchmarks.

2. IID* Lite algorithm: [43] IID* Lite algorithm is introduced in Section 3.2. This algorithm achieves up to 8 times experimentally speeding up than D* Lite algorithm. As in ID* Lite algorithm, when changes observed, alternative will be searched. But if no alternative available, IID* Lite will try to propagate changes part by part instead of propagating all changes.

3. IAD* algorithm: [42] In Section 4.1, we will introduce IAD* algorithm and doing experiments to compare it with Anytime D* algorithm. The same strategy as in IID* Lite is used. And by experiments, IAD* can achieve up to one order times of speeding up compared with Anytime D* algorithm.

4. DAWA* algorithm: In Section 4.2, DAWA* algorithm, a descendent of IAD* and AWA* algorithm, is introduced and compared with AD* and IAD* on sub-optimality and fail-ratios under given limited time. Results show DAWA* achieves up to one order of lower fail-ratios, and gains better sub-optimality than other algorithms.

By above results we claim that our framework can be combined with various techniques in dynamic navigation algorithm to speed up present navigation algorithms.

3 Improved D* Lite Algorithm

In this section two improvements to the D* Lite algorithm are proposed and analyzed. One improved algorithm is shown analytically and experimentally to produce optimal solutions at least as efficiently as D* Lite. The other performs more efficiently than D* Lite on average but is not guaranteed to be always at least as fast as D* Lite. Both algorithms can find one optimal solution in every round if one exists [40, 41, 43]. Both algorithms intelligently analyze observed environmental changes and use the results to avoid unnecessary re-calculations. These algorithms are especially effective when more than one "best" solution can be computed: this is usually the case for dynamic navigation algorithms [10].

3.1 Finding an alternative "best" path

This section presents the first improvement to D* Lite which we call ID* Lite for Improved D* Lite. A description of ID* Lite has been published [40] and the content of this section is based on that paper. ID* Lite is similar to D* and D* Lite in that it searches backwards from the goal vertex v_g to the start vertex v_c and in the way it selects the next edge to traverse. As in the case of D* Lite, ID* Lite traverses a current shortest path from v_c to v_g until changes in edge costs are detected. When that happens, instead of re-calculating vertex values immediately, as is done in D* Lite, ID* Lite tries to find a single optimal alternative path which is still consistent. To see how, suppose w is a vertex on the current path and suppose w becomes inconsistent after changes are detected. ID* Lite looks for consistent paths from v_c to v_g which contain a vertex u which is also on the current path from v_c to w. There may be several possibilities and each is given a priority proportional to the distance from u to w. The cost of some of those paths may not have been affected by the changes that cause w to become inconsistent. ID* Lite finds those paths quickly, if at least one exists, and, if at least one is found to have cost no greater than the current path, the one with highest priority is used to replace the current path.

ID* Lite can find alternative shortest paths more quickly than D* Lite can re-calculate and that is the reason ID* Lite has been shown empirically to have superior performance. ID* Lite inspects all edges in the view and records all those for which the cost has changed. There may be several such changes in a round and all will be taken into account when looking for a shortest path to v_g. Then ID* Lite makes use of a system of vertex typing to efficiently compute alternative paths by traversing chains of vertices according to type, possibly changing the type of some vertices during the traversal. Details of how this is done are saved for later. This is different than for D* Lite and its variants: they will always eagerly recompute g and rhs values 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 (because no alternative path can be found) it will not do so.

3.1.1 Terminology

When edge cost changes are detected, D* Lite re-calculates all vertex values to make them consistent. This action starts a *round* of the algorithm. A

type(v)	**Meaning**
≥ 1	Vertex v has been visited, is in the current "best" path, and **type**$(v) = \sum_{w:w \in succ(v)} T(w)$ where $T(w) = 1$ if w is available (**type**$(w) > -2$) and is 0 otherwise. That is, $T(w)$ the number of w's children that are available (they are not of type -2 and not of type -3).
0	Vertex v has been visited but is not in the current "best" path.
-1	Vertex v has not been visited. That is, v has not appeared previously in any priority queue.
-2	Vertex v is temporarily unavailable because any "best" path between v and v_g includes a vertex w that is locally inconsistent (**type**$(w) =$ -3).
-3	Vertex v is temporarily unavailable because it is not locally consistent due to changes detected on the current round.

Table 1: A table

round of the ID* Lite algorithm starts at the same point, even if no recalculations are made. If v is a vertex on the current path and $v' \in succ(v)$ with $g(v') + c(\langle v, v' \rangle) = rhs(v)$ then v' is said to be a *child* of v and v is said to be a *parent* of v'. The set of all vertices with parent is v is referred to as the *children* of v and two or more children of v are said to be *siblings*. We say a vertex is *visited* at some point in time if it exists on some path that has been considered in the current round or some previous round. We say a vertex is *available* at some point if it is consistent, has been visited, and is not affected by any of the edge cost changes that have been detected currently. When a vertex becomes unavailable due to detection of edge cost changes we say it has been *abandoned*. Abandoned vertices are *recovered* when they become available again after detecting and while dealing with edge cost changes.

3.1.2 Typing of vertices

Associated with every vertex is a type. The type of a vertex changes as ID* Lite progresses through a round. Types are indicated by numbers and have the meaning shown in Table 1. Vertex types are used to support actions that are unique to ID* Lite. These are described in the rest of this section.

When ID* Lite determines that re-calculation of vertex values can be skipped it must put aside, or abandon, some vertices that may otherwise be inconsistent, so that it may continue searching along paths that are known to be consistent. This is different from the action of D* Lite where, when the cost of edge $\langle u, v \rangle$ has been changed for the first time, u's rhs value is updated, and if u becomes inconsistent, it is inserted into OPEN to be propagated. In ID* Lite, if the update of u is skipped then **type**(u) is set to -3 to denote it is unavailable. Being unavailable, u must also force some other vertices to be unavailable as described in the next to last row of Table 1 and for each such vertex x, **type**(x) is set to -2.

3.1.3 ID* Lite details

Refer to the pseudo code for ID* Lite that is shown in Figures 6. That code uses $h(u, w)$ to denote an estimate of the distance between vertices u and w. Functions **CalculateKey**(v) and **ComputeShortestPath**() are taken without modification from the D* Lite algorithm. Function **Initialize**() defines and initializes all variables and structures common to the D* Lite algorithm and also sets the type of all vertices to 1, and the array *catch* to be empty. Function **UpdateVertex**(v) sets **type**(v) = 0 when vertex v is inserted into OPEN. Function **GetAlternativePath**(v_c) attempts to replace the current "best" path (from v_c to v_g) with a new, consistent path of cost no greater than the current path and returns TRUE if successful or FALSE if it is not successful. This is done using a (linear time) depth-first search from v_c, skipping unavailable vertices. In the process of doing this, it may discover that some vertices need to be forced to become unavailable due to the presence of a type -3 vertex on the path being searched and sets their types to -2. Function **GetBackVertex**(u) is used to recover all the type -2 vertices which are made unavailable on a previous round because u was discovered to be inconsistent and became unavailable.

Function **ProcessChanges**() determines what happens on a round. The following happens for every edge $\langle u, v \rangle$ whose cost is observed to change:

$rhs(u)$ is updated; in Line 46, if u had been discovered to be inconsistent previously, function **GetBackVertex**(u) is called to recover vertices that had also been made unavailable as a consequence; in Line 49-52, if the change may result in a shorter path, *better* is set to TRUE, otherwise u is temporarily stored in *catch* and made unavailable. If no better path has been discovered in the above process, **GetAlternativePath**(v_c) is called. If **GetAlternativePath**(v_c) fails to produce an alternative path, a full D* Lite style re-calculation is performed in Lines 55-57: all inconsistent vertex values are updated, including vertices in *catch* that were put there in previous rounds, and **ComputeShortestPath**() is called.

Function **ProcessChanges**() acts like a distributor to drive other functions and this enables ID* Lite to distinguish different kinds of edge cost changes and apply different actions naturally as will be shown in Section 3.2 when the incremental version of ID* Lite is described.

Function **Main**() is similar to that of D* Lite except that on every round it calls **ProcessChanges**() to try to avoid re-calculation of g and rhs values. Note that in Line 31, when a child of r is chosen, one with type ¿ 0 is preferred. Then, if the old shortest path can be still used, it will have highest priority and be searched first. Intuitively, this will lead to a stabler path to v_g.

3.1.4 An example

This section presents a small example which illustrates how ID* Lite can avoid re-calculations that D* Lite would make. Refer to the bi-directional graph of Figure 8(a). Vertex v_3 is shaded because it is blocked. Assume that the sensor radius is 2, the cost of every edge is 1, and that for all edges $\langle u, w, \rangle$, $h(u, w) = 1$ and $h(u, u) = 0$. Observe that this heuristic function is consistent. Vertex v_s is the start vertex and vertex v_g is the goal vertex. All other vertices are given arbitrary labels. The goal to find a least cost path between v_s and v_g.

At initialization ID* Lite sets, for all v except v_g, $g(v) = rhs(v) = \infty$ and **type**(v) = -1. It also sets $h(v_g) = rhs(v_g) = g(v_g) =$ **type**$(v_g) = 0$, puts v_g in the OPEN priority queue, and sets $k_m = 0$. Then v_c is set to v_s and **ComputeShortestPath**() is called. Since OPEN.TopKey() $= [0, 0] <$ **CalculateKey**$(v_c) = [\infty, \infty]$ and $rhs(v_c) = g(v_c)$, v_g is taken from OPEN. Since $k_{old} = k_{new}$ (see Page 11 for the reason k_{old} might be different from k_{new} in general) and $g(v_g) = \infty > 0 = rhs(v_g)$, v_g is popped from OPEN, rhs values of v_3, v_5, and v_8 become 1, the type values of those vertices become

Procedure CalculateKey(v)
01. return $[\min(g(v), rhs(v)) + h(v) + k_m,\ \min(g(v), rhs(v))]$;

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

Procedure UpdateVertex(v)
06. if ($g(v) \neq rhs(v)$ and $v \in$ OPEN) OPEN.update($[v,$ **CalculateKey**$(v)]$);
07. else if ($g(v) \neq rhs(v)$ and $v \notin$ OPEN) OPEN.insert($[v,$ **CalculateKey**$(v)]$); **type**$(v) = 0$;
08. else if ($g(v) == rhs(v)$ and $v \in$ OPEN) OPEN.remove(v);

Procedure ComputeShortestPath()
09. while (OPEN.TopKey() $<$ **CalculateKey**(v_c) or $rhs(v_c) > g(v_c)$)
10. $u =$ OPEN.top();
11. $k_{old} =$ OPEN.TopKey(), $k_{new} =$ **CalculateKey**(u);
12. if ($k_{old} < k_{new}$)
13. OPEN.update($[u,\ k_{new}]$);
14. else if ($g(u) > rhs(u)$)
15. $g(u) = rhs(u)$;
16. OPEN.remove(u);
17. for all $v \in pred(u)$
18. if ($v \neq v_g$) $rhs(v) = \min(rhs(v), c(\langle v, u\rangle)) + g(u)$);
19. **UpdateVertex**(v);
20. else
21. $g_{old} = g(u)$;
22. $g(u) = \infty$;
23. for all $v \in pred(u) \cup \{u\}$
24. if ($rhs(v) == c(\langle v, u\rangle) + g_{old}$)
25. if ($v \neq v_g$) $rhs(v) = \min_{s \in succ(v)}(c(\langle v, s\rangle) + g(s))$;
26. **UpdateVertex**(v);

Procedure GetAlternativePath(v_c)
27. vertex $r = v_c$;
28. while ($r \neq v_g$)
29. update r's type value;
30. if (**type**$(r) > 0$)
31. $r =$ one child v of r with **type**$(v) \neq -3$ and **type**$(v) \neq -2$;
32. else if (**type**$(r) == 0$)
33. **type**(r) = -2;
34. if ($r == v_c$) return FALSE;
35. $r =$ **parent**(r);
36. return TRUE;

Figure 6: The ID* Lite algorithm

Procedure GetBackVertex(v)
37. if ($v \neq$ NULL and **type**(v) < 0)
38. if ($rhs(p) \neq g(p)$)
39. return;
40. set **type**(v) = 0;
41. $v = parent(v)$;
42. **GetBackVertex**(v);

Procedure ProcessChanges()
43. boolean *better* = FALSE, *recompute* = FALSE;
44. for every edge $\langle u, v\rangle$ where $c(\langle u, v\rangle)$ has changed since the previous round
45. Update u's rhs value;
46. if (**type**(u) == -3) **GetBackVertex**(u);
47. if ($rhs(u) == g(u)$) set **type**(u) = 0;
48. else
49. if ($g(u) > rhs(u)$ and $h(v_c, u + rhs(u)) < \Omega$)
50. *better* = TRUE, **UpdateVertex**(u);
51. else
52. *catch*.add(u), set **type**(u) = -3;
53. if (*better* == FALSE) *recompute* = **!GetAlternativePath**(v_c);
54. if (*recompute* == TRUE)
55. Update v such that **type**(v) $\neq 0$ in *catch*;
56. for all v such that **type**(v) $\neq 0$ set **type**(v) = 0;
57. **ComputeShortestPath**();

Procedure Main()
58. **Initialize**(); $v_{last} = v_c = v_s$;
59. **ComputeShortestPath**(); **GetAlternativePath**(v_c);
60. while ($v_c \neq v_g$)
61. set **type**(v_c) = 0;
62. for some child v of v_c such that **type**(v) > 0 set $v_c = v$;
63. Move the agent to v_c;
64. Scan the graph for changes;
65. if any changes are found
66. $k_m = k_m + h(v_{last}, v_c)$;
67. $v_{last} = v_c$;
68. **ProcessChanges**();

Figure 7: The ID* Lite algorithm continued

0, and those vertices are placed in OPEN with equal priority $[2, 1]$ (Lines 14-19, 07). Now suppose v_3 is at the top of OPEN. Then u is v_3 in Line 10, $k_{old} = k_{new}$ in Line 12, and $g(v_3) = \infty > 1 = rhs(v_3)$ so v_3 is removed from OPEN, $g(v_3)$ is set to 1, and v_2 is added to OPEN with priority $[3, 2]$ with $rhs(v_2)$ set to 2 (the key of v_5 is not updated in Line 06). Now say v_5 is at the top of OPEN. Then, as before, $k_{old} = k_{new}$ and $g(v_5) = \infty > 1 = rhs(v_5)$ so v_5 is removed from OPEN, $g(v_5)$ is set to 1, and v_4 is added to OPEN with priority $[3, 2]$ with $rhs(v_4)$ set to 2 (v_3 and v_8 are not updated in Line 06 because they have unchanged key values). Now v_8 is at the top of OPEN and Lines 15-19 are executed. This results in $g(v_8)$ set to 1, v_8 is removed from OPEN, v_7 is added to OPEN with priority $[3, 2]$ and $rhs(v_7) = 2$. Now say v_2 is at the top of OPEN and Lines 15-19 are executed resulting in $g(v_2) = 2$, v_2 removed from OPEN, v_1 is added to OPEN with priority $[4, 3]$ and $rhs(v_1) = 3$. Now say v_4 is at the top of OPEN and lines 15-19 are executed resulting in $g(v_4) = 2$, v_4 removed from OPEN, v_6 is added to OPEN with priority $[4, 3]$ and $rhs(v_6) = 3$. Now v_7 is at the top of OPEN and lines 15-19 are executed resulting in $g(v_7) = 2$, v_7 removed from OPEN and nothing is added to OPEN. Now say v_1 is at the top of OPEN and Lines 15-19 are executed resulting in $g(v_1) = 3$, v_1 removed from OPEN and v_s is added to OPEN with priority $[4, 4]$ and $rhs(v_s) = 4$. Now v_6 is at the top of OPEN and Lines 15-19 are executed but nothing happens. Finally v_s is at the top of OPEN and execution returns from **ComputeShortestPath**() because v_s is v_c. At this point the g and rhs values for all vertices have been updated and the type values have been set to 0. This information is next going to be used by **GetAlternativePath**(v_c) to find the highest priority shortest path from v_s to v_g.

When **GetAlternativePath**(v_s) is called a cursor vertex r is set to v_s, **type**(r) is set to 2 in Line 29 because v_s has successors v_1 and v_6 and $rhs(v_s) = 4 = g(v_1) + c(\langle v_s, v_1 \rangle)$ and $rhs(v_s) = 4 = g(v_6) + c(\langle v_s, v_6 \rangle)$ (see Page 24). Since **type**(v_s) > 0, r is set to, say, v_1 in Line 31. Then **type**(v_1) is set to 2 in Line 29 because it has two children and r is set to, say, v_2. Then **type**(v_2) is set to 1 in Line 29 because v_3 is v_2's only child. Next r is set to v_3 and **type**(v_3) is set to 1 with v_g as the only child. Finally, r is set to v_g and execution stops. In Figure 8(a) the numbers above the vertices are the type numbers of those vertices after **GetAlternativePath**(v_s) is called. Implicitly, the current "best" path follows the vertices of non-negative type numbers starting at v_s and is shown as the dashed line in the figure.

At this point movement of the agent along the current path starting at

$v_s = v_c$ commenses in Line 61 of function **Main**(). First **type**(v_s) is set to 0. In Line 62 the agent is moved to $v_1 = v_c$. At Line 64, since the sensor radius is 2, v_3 is discovered to be blocked, and all edges incident with it have new costs equal to ∞. In D* Lite, because all changes are propagated, v_2 and v_3 will be expanded. But in ID* Lite, function **ProcessChanges**() is called with $k_m = 1$ (set in Line 66) to test possible changes to costs of edges $\langle v_2, v_3\rangle$, $\langle v_3, v_2\rangle$, $\langle v_5, v_3\rangle$, $\langle v_3, v_5\rangle$, $\langle v_g, v_3\rangle$, and $\langle v_3, v_g\rangle$.

In **ProcessChanges**() suppose edge $\langle v_2, v_3\rangle$ is processed first. At line 45, $rhs(v_2)$ is updated, as in D* Lite, by setting $rhs(v_2) = min(rhs(v) + c(\langle v_2, v\rangle), v \in succ(v_2)$ and since $rhs(v_g) = 0$, $rhs(v_2) = rhs(v_1)+c(\langle v_2, v_1\rangle) = 3 + 1 = 4$. Because **type**($v_2$)=1, Line 46 is not executed and since $rhs(v_2) = 4 \neq 2 = g(v_2)$ the assignment of Line 47 is not executed either. In Line 49, since $g(v_2) < rhs(v_2)$, line 52 is executed, v_2 is temporarily stored in *catch*, and **type**(v_2) is set to -3. Then say edge $\langle v_3, v_2\rangle$ is processed. At Line 45, $rhs(v_3)$ is set to ∞. Because **type**(v_3)=1, Line 46 is not executed and since $rhs(v_3) = \infty \neq 1 = g(v_3)$, Line 47 is not executed. In line 49, because $g(v_3) < rhs(v_3)$, line 52 is executed and v_2 is temporarily stored in *catch* and **type**(v_3) is set to -3. Next say edge $\langle v_5, v_3\rangle$ is processed. At Line 45 $rhs(v_5)$ is set to $rhs(v_g) + c(\langle v_5, v_g\rangle) = 0 + 1 = 1$. Next, since $rhs(v_5) = 1 = g(v_5)$, Line 47 is executed and **type**(v_5) is set to 0. When edge $\langle v_3, v_5\rangle$ is processed, nothing happens. Now say edge $\langle v_g, v_3\rangle$ is processed. At Line 45 $rhs(v_g)$ is set to 0 and hence nothing happens. When edge $\langle v_3, v_g\rangle$ is processed, nothing happens. Having finished considering edge costs, execution procceds to Line 53 where, since *better* = $FALSE$, function **GetAlternativePath**(v_c) is called.

In **GetAlternativePath**(v_c) cursor vertex r is set to v_1, **type**(r) is set to 1 and in Line 29 because v_1 has successor v_4 and $rhs(v_1) = g(v_4) + c(\langle v_1, v_4\rangle)$ (observe since **type**(v_2)=–3, v_2 can not contribute to the type value of v_1). Since **type**(v_1)= 1, r is set to v_4 in Line 31. Then **type**(v_4) is set to 1 in Line 29 because it has one child, v_5. Then r is set to v_5 and **type**(v_5) is set to 1 in Line 29 because v_g is v_5's only child. Then r is set to v_g and execution in **GetAlternativePath**(v_c) stops. At this point, the "best" path from $v_c = v_1$ to v_g is indicated by the dashed line in Figure 8(b).

Returning to **ProcessChanges**(), because function **GetAlternativePath** returns $TRUE$, *recompute* is set to 0 **ProcessChanges**() returns control to **Main**() and the agent is made to move along the dashed line in Figure 8(b) starting at v_1. Thus, compared to D* Lite, ID* Lite saves updating vertices v_2 and v_4 in OPEN. If the sensor radius had been 1 the agent

would not have observed that v_3 is blocked and would have moved to v_2; at that time a new best path would have to be computed by one complete re-calculation. If v_3 becomes unblocked as the agent begins to move out of v_1 then **GetBackVertex**(v_3) is executed in Line 46. This causes v_2 and v_3 to become available and the old optimal path to be recovered.

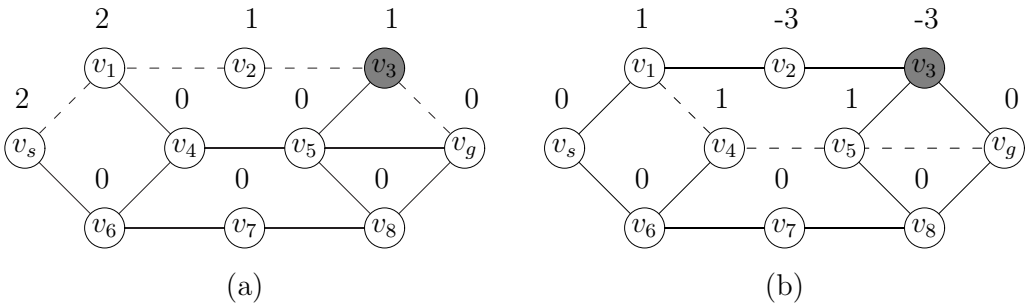


Figure 8: The action of ID* Lite on a small example

3.1.5 Analysis and theoretical results

In this section it is shown that on every round, ID* Lite computes a shortest path, if one exists: that is, a path from the current agent location to the goal vertex whose edge cost sum is minimum over all such paths given the observable environment of the round. It is assumed that the vertex heuristic function $h(w, u)$ is consistent and is therefore always a lower bound on the minimum cost path from w to u using edge costs given by function c and is such that the triangle inequality holds. Ω is used to represent the cost of the old found "best" path, and Ω' represents the cost of the current "best" path after re-calculation.

Observation 1 *If the edge cost of edge e increases in value then any path passing through e with edge costs that have not decreased in value has a cost that is greater than Ω, the cost of previous round shortest path.*

Proof: 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. □

Sponsored by Observation 1, we can divide changes by how they can affect solutions.

Intuitively, in case edge costs are only increased, it is expected that the cost of a "best" path will go up and if edge costs are only decreased, the cost of a "best" path will not increase. Since it is possible that a path includes both increased and decreased changes one might suppose that superposing results that consider increased and decreased changes separately might be a reasonable way to find a "best" path. The following observation shows that this is not always the case.

Observation 2 *If $c(\langle w, u\rangle)$ decreases and if after updating $rhs(w)$, $rhs(v)$, $h(v_c, w) + rhs(w)$ is no less than Ω of the previous round, then any path passing through $\langle w, u\rangle$ without other decreased edges* after *$\langle w, u\rangle$ has a cost that is not less than Ω.*

Proof: Suppose a path p passes through $e = \langle w, u\rangle$. Then the cost η of p has the property: $\eta \geq h(v_c, w) + c(e) + g(u)$. By definition of rhs, we have $\eta \geq h(v_c, w) + rhs(w)$. Surely, the later is greater than Ω. That is, $\eta \geq h(v_c, w) + rhs(w) \geq \Omega$. □

Observation 2 shows that in some cases where edge costs are decreased, the cost of the "best" path may not decrease. Moreover, such cases can be identified prior to re-calculating g and rhs values. The results below show the requirement that no edges after $\langle w, u\rangle$ have decreased edge costs can be relaxed.

Observation 3 *Assume h is consistent. If in $e = \langle w, u\rangle$, w's type value is -1, and $g(u)$ is correct, then any path from v_c to v_g passing through e has a cost that is not less than Ω.*

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

This Observation provides an efficient way to compute a lower bound on the cost of a path such that at least one edge has a decreased cost.

Lemma 4 *In the current round, after propagating changes (i.e. re-calculating), if there are two edges $e' = \langle w, u\rangle$ and $z' = \langle v, r\rangle$ on some path p' which is the shortest path passing through e' and z', and z' is after e' along p', then $h(v_c, w) + rhs(w) \geq h(v_c, v) + rhs(v)$.*

Proof: If p' is the shortest path passing through e' and z' and z' is after e' along 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 the fact that h is consistent. □

Lemma 5 *In the current round, before propagating changes, suppose there is a "best" path containing at least one edge cost change over the previous round and $e' = \langle w, u \rangle$ is the last one that has changed. Then $h(v_c, w) + rhs(w)$ is correct: that is, the sum does not change after propagating changes.*

Proof: If the value of $h(v_c, w) + rhs(w)$ is not correct, then $rhs(w)$ will be affected by some cost change, say in edge z'. The shortest path through the "best" path must include z' too and z' will be after e' along the path. This is a contradiction. □

Corollary 6 *In the current round, after re-calculation if needed, if $e' = \langle w, u \rangle$ is the last edge along a "best" path p' from v_c to v_g whose cost has changed, and the cost of p' is less than Ω, then before re-calculation $h(v_c, w) + rhs(w) < \Omega$.*

Proof: If the cost of p' is less than Ω, then after re-calculation, by the fact that h is admissible, $h(v_c, w) + rhs(w) < \Omega$. The conclusion follows from Lemmas 4 and 5. □

Lemma 7 *In the current round, after re-calculation if needed, if $e' = \langle w, u \rangle$ is the edge along a "best" path p' from v_c to v_g whose cost has changed, then, before re-calculation, only w needs to be reinserted into the priority queue OPEN in order to get p'.*

Proof: Without loss of generality assume $z' = \langle v, r \rangle$ is an edge along p' whose cost has changed and suppose z' appears before $e' = \langle w, u \rangle$ on p'. By an argument taken from [14], if w is reinserted into priority queue, then it will be updated and hence $rhs(r)$ must be affected. Thus v will also be reinserted into OPEN and updated. Iteratively, vertices before w on p' will be subsequently updated until v_c is chose to be updated from OPEN. In other words, path p' has been found. □

Corollary 8 *In the current round, after re-calculation if needed, if there exists a new "best" path p' from v_c to v_g with cost less than Ω, then it can be obtained by updating the last changed edge $e' = \langle w, u \rangle$ along p' satisfying $h(v_c, w) + rhs(w) < \Omega$.*

Proof: Follows from Corollary 6 and Lemma 7. □

The conclusion of Corollary 8 looks good but cannot be implemented conveniently because it is hard to say whether one edge cost change is the last one along a path. So the conditions of the corollary are broadened in the following.

Proposition 9 *If there are paths which are better than the old "best" path they can be computed by propagating only the changes on edges* $\langle w, u\rangle$ *where* $h(v_c, w) + rhs(w) < \Omega$.

Proof: Follows from Corollary 2. □

Proposition 9 differs from Corollary 8 in two ways: 1) the "last change" is not required anymore; 2) the equation in Lemma 5 will be guaranteed correct only when the change is on the last edge with changed cost. That means the change in edge $z' = \langle v, r\rangle$ 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. In other words, the vertices considered are a super-set of the vertices in Corollary 6. We expect eventually to find a more strict condition that will prevent this.

Proposition 9 answers the question concerning "without other decreased edges..." because it does not use any assumption about it at all. One might suppose that if the cost of a new "best" path is equal to the cost of the old one the round might be terminated successfully at that point. This is not the case. The conclusions above are correct and have been proved, but there may be some "fake" paths which cannot 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 un-propagated change $e' = \langle w, u\rangle$ can be considered to be one optimal solution. In that case the cost of the changed e' must have increased or it will satisfy the formula in Proposition 9 and then **ComputeShortestPath**() will be called. Then, if the cost information of e' is propagated, the cost of p' must increase. In other words, p' is not a real optimal solution. This requires a way to find one *correct* optimal solution quickly.

The above results will now be used to prove the correctness and completeness of the pseudo code of ID* Lite (shown in Figures 6 and 7).

Lemma 10 *Assume* g *and* rhs *values are correct.* **GetAlternativePath**(v_c) *returns TRUE if and only if, after it executes, it is possible to traverse a path from* v_c *to* v_g *by always visiting a least positive type neighboring vertex next*

and that path is a consistent least-cost path from v_c to v_g. The cost of that path is $rhs(v_c)$.

Proof: Since **GetAlternativePath**(v_c) does a depth-first search from v_c, visiting each vertex at most once because any vertex that is not available is marked by -2 or -3 and will not be visited again during the search. If there is a path from v_c to v_g then the depth first search will visit all vertices from v_c to v_g along some such path. But all such vertices will have positive type because types are never changed from 0 or negative to positive during the search. Prior to execution of **GetAlternativePath**(v_c) the cost of the optimal path is $rhs(v_c)$. This follows from D* Lite []. Traversing vertices from v_c to v_g by moving to the next adjacent vertex of lowest positive type is the least cost path from v_c to v_g. By definition of child, the total cost of this path is also $rhs(v_c)$ and must be of least cost. By definition of type, a vertex with a positive type is consistent. Hence the path is consistent. □

Since **GetAlternativePath**(v_c) executes a depth first search on the graph its worst-case complexity is linear in the number of vertices. However, the efficiency of this function is due to the fact that simple checks are made when a vertex is visited: that is, arithmetic operations are avoided.

Lemma 11 *The changes which have been skipped in a round when the shortest path's cost is Ω may continue to be skipped on future rounds until a new "best" path has cost that is greater than Ω.*

Proof: Suppose the cost of edge $e = \langle w, u \rangle$ is skipped. Then, using the analysis of A* in [7, 12] which applies here as well, the lower bound η on the cost of any path passing through e has $\eta \geq \Omega$ and this edge cannot be on any "best" path to v_g. Hence the propagation of values due to a change in cost of e can be skipped safely. If the new shortest path has its cost not greater than Ω, edge e cannot be on it too. So the change on e can be skipped until a new "best" path has cost greater than Ω. □

Theorem 12 *Function ProcessChanges() will find the shortest path in every round if and only if there exists one.*

Proof: By Proposition 9, Lemma 10, and Lemma 11 if, in Line 54, *recompute* is FALSE, a correct shortest path must have been found. If *recompute* is TRUE, however, Lines 55-57 will execute just like D* Lite would. Therefore, the correctness and completeness of finding a new shortest path follows from the correctness and completeness of D* Lite. □

Theorem 13 *In every round, ID* Lite returns a shortest path from v_c to v_g if and only if at least one shortest path exists.*

Proof: Follows directly from Theorem 12 and Lemma 11. □

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

Proposition 14 *After a full re-calculation of g and rhs values in Lines 55-57, all vertices which can be part of some shortest path are updated.*

Proof: 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 its children will be updated to be consistent. So, suppose one least-cost path between v_c and v_g has been found, and suppose p is any path from v_c to v_g which is optimal, then the child of v_c on p must be updated and consistent too. Iteratively, all the vertices on p are updated and consistent. □

Proposition 15 *ID* Lite expands no more vertices than D* Lite if the same tie breaking method is used.*

Proof: Because **GetAlternativePath**(v_c) is executed before re-calculating g and rhs values and no vertices are expanded in **GetAlternativePath**(v_c), by Lemma 10, if TRUE is returned, then one round of re-calculation that D* Lite has to make can be avoided.

So we need to show that if several rounds of re-calculations are combined to be one will not expand more vertices than recomputing in every round. Assume in ID* Lite $\{r_1, r_2, ..., r_s\}$ are combined to be one recomputation. Because we have the hypothesis that the same tie breaking method used, when full recomputation executed in ID* Lite, we have that the weight of current optimal path Ω is the same as D* Lite. If in ID* Lite one vertex v is expanded because of the propagation of one change of edge $e = \langle w, u \rangle$, then $\mathbf{key}(v) < \Omega$. If the change is propagated in D* Lite in the round r_s, then v will be expanded; if e is propagated in D* Lite in round r_l, $l < s$, and by Lemma 11, the cost Ω' of an optimal path in r_l is greater than Ω. Hence, v is expanded in the re-calculation of round r_l in D* Lite. □

Proposition 15 shows that, by supposing the agent chooses the same next vertex to move to in Lines 62-63, ID* Lite expands no more vertices than D* Lite. Also, according to the proof of Proposition 15, one vertex v expanded during re-calculation of round r_l, $l < s$ in D* Lite will not be expanded in ID* Lite if $\Omega \leq \mathbf{key}(v) < \Omega'$. This partially explains why ID* Lite can improve on D* Lite.

3.1.6 Experiments and Results

In this section, the performance of ID* Lite is compared experimentally to that of the D* Lite algorithm on random grid world terrains. In each experiment the initial terrain is a blank, 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.

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.

The results on rock-and-garden benchmarks are shown in Figures 9 to 13. In Figures 9 and 10 *size* is 300, and *percent* is 10 and 30 respectively. The left graph of each figure shows the number of heap operations as a function of *sensor-radius*. Heap operations make the most significant contribution to time complexity in dynamic navigation algorithms and that is why those results are presented. The plots show only the heap operations in re-calculations: the number of operations used when initializing a shortest path from v_s to v_g are not counted because all of algorithms of the D* family operate the same way that the A* algorithm does in this phase. The right graph of each figure shows the ratio of the number of re-calculations to the number of edge cost changes observed within the *sensor-radius*. The D* Lite curve is flat at 1 because every time an inconsistency is observed, exactly one re-calculation must be performed.

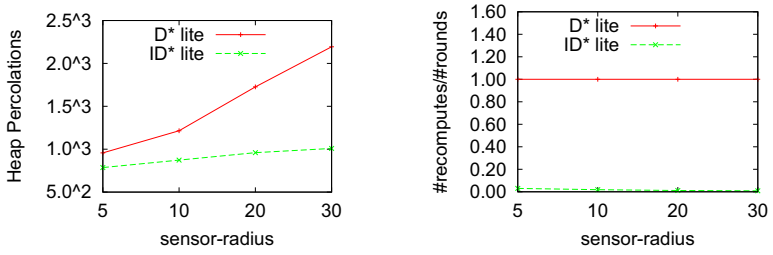


Figure 9: size=300 and percent=10

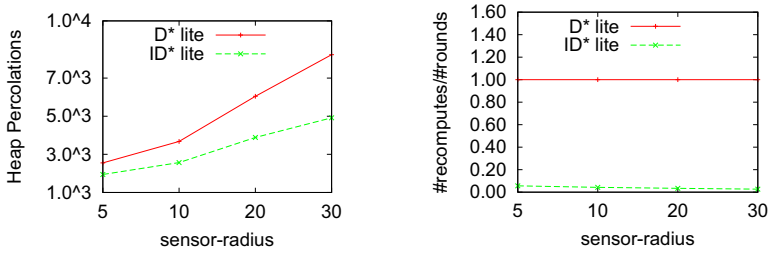


Figure 10: size=300 and percent=30

To some extent, the graphs explain why ID* Lite can outperform D* Lite. The right graphs of Figures 9 and 10 show that ID* Lite can save almost 90% of the re-calculations that would be done by D* Lite. However, this does not mean that a corresponding savings applies for heap operations since edge cost changes are transferred from *catch* to OPEN every time a full re-calculation occurs. Since there are fewer re-calculations in ID* Lite, more changes are processed when re-calculating and many changes with big key values are not propagated. The more vertices that are affected by such changes, the more heap operations can be saved in ID* Lite. Informally speaking, decreasing changes can be propagated more easily than increasing changes. Hence, decreasing changes can affect more vertices than increasing changes. In the rock-and-garden benchmarks there are only increasing changes but in the dynamic agent navigation benchmarks there are many decreasing changes. From Figures 9 and 10 ID* Lite heap percolation is seen to increase more slowly than that of D* Lite as the *sensor-radius* increases. Hence, in rock-and-garden benchmarks, as *sensor-radius* is increased, the time consumed by ID* Lite increases more slowly than the time consumed by D* Lite. This is significant because, as advancing technology admits finer grain terrain models where *size* is increased, *sensor-radius* will correspondingly increase as well.

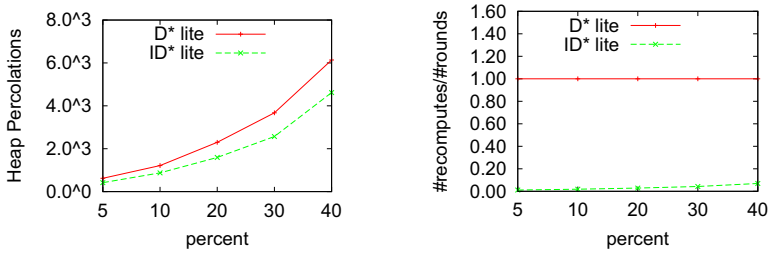


Figure 11: size=300 and sensor-radius=10

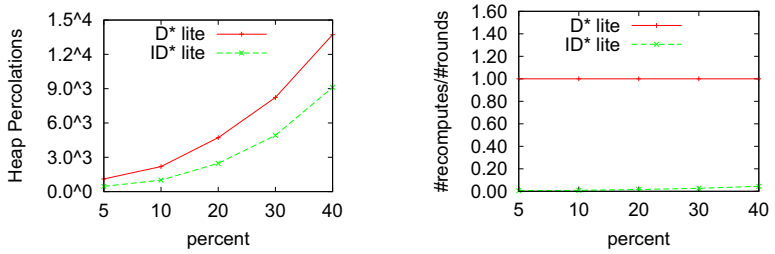


Figure 12: size=300 and sensor-radius=30

Figure 11 and 12 present the data with *sensor-radius* fixed and *percent* changing from 5 to 40 percent.

Figure 13 shows the relationship between heap operations and *size* where *sensor-radius*$= 0.1 * size$ and $percent = 30$. No matter what the *size*, ID* Lite performs better than D* Lite on the rock-and-garden benchmarks.

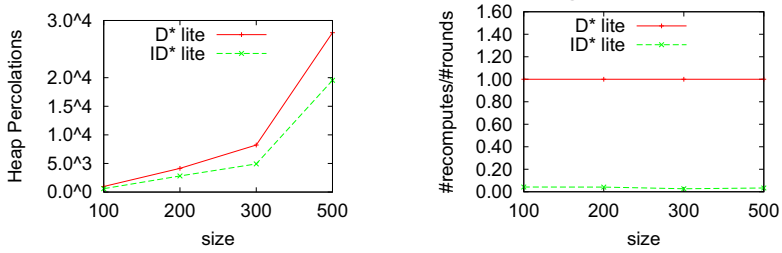


Figure 13: sensor-radius=0.1*size and percent=30

The second set of benchmarks, parking-lot benchmarks, is intended to model agent navigating in the presence of terrain changes. 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 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 token is unblocked and edge costs into it are not ∞.

The experiments are done in the same way as the rock-and-garden experiments. In Figures 14 and 15, and *percent* = 10 and *percent* = 30, respectively. The left graph of each figure shows the relationship between heap operations and *sensor-radius*. Observe that heap percolation in ID* Lite increases more slowly than for D* Lite as the *sensor-radius* increases. So, also for the dynamic navigation benchmarks, as *sensor-radius* is increased, the time consumed by ID* Lite increases more slowly than the time consumed by D* Lite.

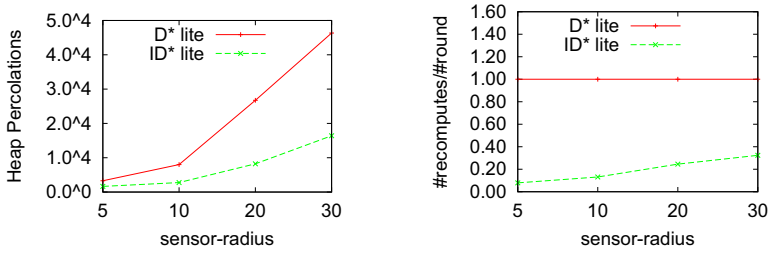


Figure 14: size=300 and percent=10

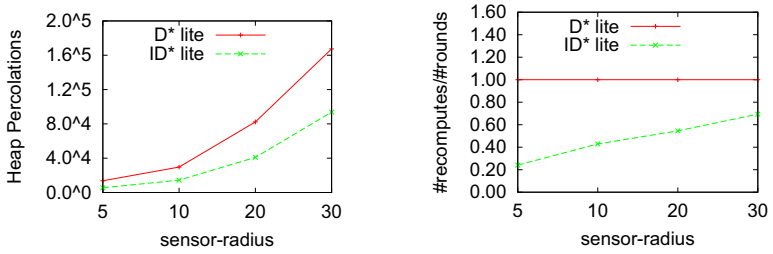


Figure 15: size=300 and percent=30

In Figures 14 and 15, the size is fixed at 300, and *sensor-radius*= 10 and *sensor-radius*= 30, respectively. The left graph of each figure shows the relationship between heap operations and *percent* while the right graph of each figure shows the ratio of the number of re-calculations to the number of edge cost changes observed in the view. Observe that heap percolation

in ID* Lite increases more slowly than in D* Lite as *percent* increases. In other words, for dynamic navigation benchmarks ID* Lite is less sensitive to edge cost changes than is D* Lite. This infers ID* Lite can perform better than D* Lite in environments where edge cost changes are frequent. When *percent* goes to 40, it seems that the conclusion above does not hold: when *percent* is high, there is often no path at all from the agent to the goal and this causes every vertex to be expanded, canceling the feature of ID* Lite that is capable of making it outperform D* Lite and making it perform as D* Lite does.

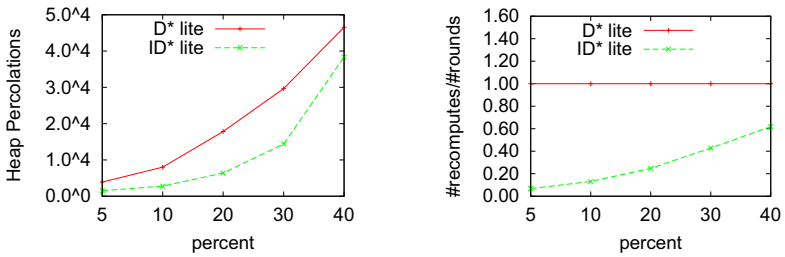


Figure 16: size=300 and sensor-radius=10

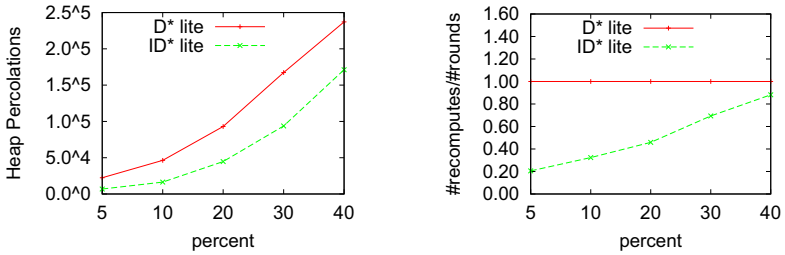


Figure 17: size=300 and sensor-radius=30

Figure 18 shows the relationship between heap operations and *size* when *sensor-radius*$= 0.1 * size$ and $percent = 30$. ID* Lite performs better than D* Lite from $size = 100$ to $size = 500$. In Figures 14 to 18 the ratio of re-calculations to the number of edge cost changes observed for ID* Lite increases with increasing *sensor-radius* and *percent* as one would expect, but the ratio is always below 1.

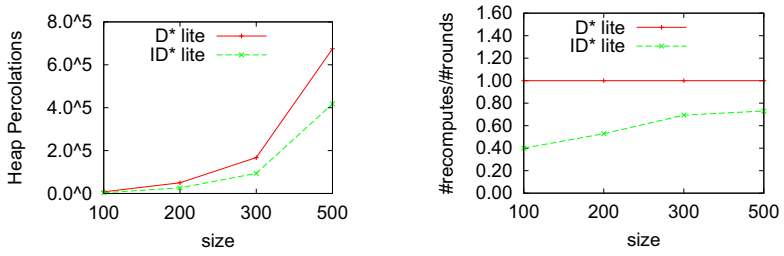


Figure 18: sensor-radius=0.1*size and percent=30

3.1.7 Future Work

In Line 49 of ID* Lite (Figure 7) the expression $h(v_c, u) + rhs(u)$ is a lower bound on path costs through u. If this can be replaced by an expression that produces a better bound then more edge cost changes can be skipped and ID* Lite should run faster. Looking for such an expression is part of our plan for improving on the results presented here. We also expect to find a way to determine whether an increased edge cost change can affect the path or not efficiently. If an observed increase in edge cost does not affect one some path of least cost, this change can be skipped safely. If an observed decrease in edge cost causes the equation of Line 49 not to satisfied, then that change can be skipped as well. If ID* Lite is modified to make these checks we expect its performance will be significantly improved and, if handled deterministically, will still be no worse that the performance of D* Lite.

3.2 Finding an alternative "best" path incrementally

This section presents the second improvement to D* Lite which we call IID* Lite for Incremental Improved D* Lite. IID* Lite can have better average case time performance than D* Lite without a guarantee to produce at least as good a path as D* Lite. The algorithm introduced in [40, 41] updates and expands all overconsistent vertices which may cause a better optimal path than the original one, then uses function **GetAlternativePath**(v_c) to find an alternative best path whenever possible. But if the new "best" path has higher cost than the original, a full re-calculation is executed by calling **ComputeShortestPath**().

IID* Lite aims to reduce the cost of re-calculation by dynamically ordering edge cost changes and propagating changes, in order, until a path to v_g is found or until it is discovered that no such path exists. If a path is found, it is optimal. Since edge cost changes need not be considered after finding the path, this algorithm can potentially run faster than D* Lite. The idea of ordering edge cost changes is compatible with ID* Lite and is added to ID* Lite which is why we call this IID* Lite. IID* Lite can potentially run faster than ID* Lite as well.

The dynamic ordering of edge cost changes uses an increasing estimate t of the cost of an optimal path given the current environment. The ordering is initiated when it is discovered that the current "best" path, of cost ω, is inconsistent and no alternative path of cost Ω is available to replace it. Initially, t is set to be close to Ω and a search is made for a path of cost no greater than t. Only values of any vertex u such that $g(u) > rhs(u)$, $h(v_c, u) + rhs(u) < t$, and u is an end point of an edge whose cost has changed are propagated. No other vertex values are propagated because no other vertices on an edge whose cost has changed can be on a path of cost less than t. In case there is still no consistent path to v_g, t is increased and the values of additional vertices that now satisfy the above conditions are propagated. This continues until either a consistent path is found or until values of all vertices associated with edge cost changes have been propagated in which case no consistent path is possible and the algorithm terminates.

It remains to explain how t is incremented. This is quite an important part of IID* Lite. If t is initially too small or the change in t is too small, then too few vertices are updated. If t is initially too large or the change in t is too large, then too many vertices are updated. If too few vertices are updated, the overhead due to partially re-calculating many times will

cancel the savings due to propagating fewer values. If too many vertices are updated, the savings potential of IID* Lite will be diminished because it will begin to behave more like D* Lite. Given the current position of the agent is v_c, the value of t is set to $rhs(v_c)$, which is an estimate of the cost of a path from v_c to v_g that includes the cost of edges incident to v_c. Then, if there is one path whose cost is less than t, **GetAlternativePath**(v_c) will find it and its cost will be optimal. Otherwise, $rhs(v_c)$ will have increased due to the changes and t can again be set to $rhs(v_c)$ and the process repeated.

3.2.1 Propagating information about overconsistent vertices

From Section 3.1, when a change in edge cost or consistency is discovered in the current "best" path, if there is an alternative "best" path of equal cost, then ID* Lite will try to find it without re-calculating g and rhs values, and if it is unable to do so, a re-calculation is executed, as is done in D* Lite algorithm. Since re-calculations are expensive and to be avoided, if possible, the question whether something more can be done to find an alternative path is raised and discussed in this section. This leads to a proposed algorithm which we call IID* Lite algorithm.

The proposed IID* Lite algorithm is based on the observation that if, after a current optimal path of cost Ω becomes inconsistent after edge cost changes, there exists a path from v_c to v_g with cost $t > \Omega$, it is sometimes possible to propagate values due to only some of the edge cost changes and still determine a new, optimal, consistent path from v_c to v_g. Specifically, if the cost of an edge $e = \langle u, w \rangle$ has changed, but $f(\langle u, w \rangle) \doteq h(v_c, u) + rhs(u) \geq t$ then any path to v_g from v_c that contains e has cost greater than t so changes to the values of u and w need not be propagated. So, when edge cost changes are observed, t is set to Ω and values associated with any edge e of changed cost such that $f(e) < t$ are propagated and a search for a path having cost no greater than t is made. If successful, the path becomes the current "best" path. If not successful, t is increased, more values are propagated because more edges satisfy $f(e) < t$, and another search for a path of cost less than or equal to t is made. This process continues until either a path is found to exist or all edge cost changes have been propagated. The algorithm will always find the "best" path to v_g if a path exists but its efficiency depends on how t is incremented.

The motivation of how to set the threshold number is explained below. Given a threshold number T, if after propagation of changes, there is a path

whose cost is $< T$, that means one better optimal path has been found; otherwise if there is a path whose cost $= T$ then the path is the optimal path. Hence we want to set T where it is not that easy to be improved and also there are paths whose cost values equal T.

Procedure MiniCompute( )
01. while (OPEN.TopKey() < **key**(v_c))
02. u =OPEN.Top(), k_{old} =OPEN.TopKey(), k_{new} =**CalculateKey**(u);
03. if ($k_{old} < k_{new}$)
04. OPEN.Update(u, k_{new});
05. else if ($g(u) > rhs(u)$)
06. $g(u) = rhs(u)$;
07. OPEN.Remove(u);
08. for every $s \in pred(u)$
09. if ($s \neq v_g$)
10. $rhs(s) = min(rhs(s), c(\langle s, u\rangle) + g(u))$;
11. if ($s \in catch$) $catch$.Remove(s).
12. **UpdateVertex**(s);
13. else
14. OPEN.Remove(u);

Figure 19: Propagation of values due to overconsistent vertices

3.2.2 Description of the pseudo code for IID* Lite

Pseudo code for IID* Lite is shown in Figures 19and 20. The following explains the difference between ID* Lite and IID* Lite emphasizing how the threshold t is increased and how changes are propagated.

Most of the functions of IID* Lite are the same as those of ID* Lite. The main differences are in the functions are **GetAlternativePath**(v_c) and **ProcessChanges**(). **GetAlternativePath**(v_c) returns TRUE if and only if there is a consistent 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. If at Line 32 **type**(r) is 0, then by definition of type, r is not on the current least cost path and its type is set to -2 but in Lines 37, children of r such that whose **type** values are -3 are merged into C. If no path can be returned by function

Procedure CalculateKey(v)
01. return $[\min(g(v), rhs(v)) + h(v) + k_m,\ \min(g(v), rhs(v))]$;

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

Procedure UpdateVertex(v)
06. if ($g(v) \neq rhs(v)$ and $v \in$ OPEN) OPEN.update($[v,$ **CalculateKey**$(v)]$);
07. else if ($g(v) \neq rhs(v)$ and $v \notin$ OPEN) OPEN.insert($[v,$ **CalculateKey**$(v)]$); **type**$(v) = 0$;
08. else if ($g(v) == rhs(v)$ and $v \in$ OPEN) OPEN.remove(v);

Procedure ComputeShortestPath()
09. while (OPEN.TopKey() $<$ **CalculateKey**(v_s) or $rhs(v_s) > g(v_s)$)
10. $u =$ OPEN.top();
11. $k_{old} =$ OPEN.TopKey(), $k_{new} =$ **CalculateKey**(u);
12. if ($k_{old} < k_{new}$)
13. OPEN.update($[u,\ k_{new}]$);
14. else if ($g(u) > rhs(u)$)
15. $g(u) = rhs(u)$;
16. OPEN.remove(u);
17. for all $v \in pred(u)$
18. if ($v \neq v_g$) $rhs(v) = \min(rhs(v), c(\langle v, u\rangle) + g(u))$;
19. **UpdateVertex**(v);
20. else
21. $g_{old} = g(u)$;
22. $g(u) = \infty$;
23. for all $v \in pred(u) \cup \{u\}$
24. if ($rhs(v) == c(\langle v, u\rangle) + g_{old}$)
25. if ($v \neq v_g$) $rhs(v) = \min_{s \in succ(v)}(c(\langle v, s\rangle) + g(s))$;
26. **UpdateVertex**(v);

Procedure GetAlternativePath(v_c)
27. Vertex $r = v_c$;$C = \emptyset$
28. while ($r \neq v_g$)
29. update r's type value;
30. if (**type**$(r) > 0$)
31. $r =$ one child y of r with **type**$(y) \neq -3$ and **type**$(y) \neq -2$;
32. else if (**type**$(r) == 0$)
33. **type**(r) = -2;
34. if ($r == v_c$)
35. for every vertex $c \in C$ **UpdateVertex**(c);
36. return FALSE;
37. $C = C \cup r's$ **type** value -3 children; $r = parent(r)$;
38. return TRUE.

Procedure GetBackVertex(v)
39. if ($v \neq$ NULL and **type**(v) < 0)
40. if ($rhs(p) \neq g(p)$)
41. return;
42. **type**(v) $= 0$;
43. $v = parent(v)$;
44. **GetBackVertex**(v);

Procedure ProcessChanges()
45. Boolean *better*=FALSE, *recompute* = FALSE, $t = rhs(v_c)$.
46. for every edge $e = \langle u, v \rangle$ where $c(e)$ has changed since the previous round:
47. Update $rhs(u)$;
48. if (**type**(u) $= -3$) **GetBackVertex**(u);
49. if ($rhs(u) == g(u)$) **type**(u) $= 0$;
50. else
51. if ($g(u) > rhs(u)$) and $h(v_c, u) + rhs(u) < t$)
52. *better* = TRUE, **UpdateVertex**(u);
53. else
54. *catch*.add(u), **type**(u) $= -3$;
55. if (*better* == TRUE) **MiniCompute**();
56. while (!**GetAlternativePath**(v_c))
57. $t_{old} = t$, **ComputeShortestPath**(), $t = rhs(v_c)$;
58. if $t > t_{old}$
59. *better*=FALSE;
60. for every $u \in$ *catch* such that **type**(u) $\neq 0$
61. if ($h(v_c, u) + rhs(u) < t$ and $g(u) > rhs(u)$)
62. *better* = TRUE, **UpdateVertex**(u);
63. *catch*.remove(u).
64. if (*better* == TRUE) **MiniCompute**().

Procedure Main()
65. **Initialize**(); $v_{last} = v_c = v_s$;
66. **ComputeShortestPath**(); **GetAlternativePath**(v_c);
67. while ($v_c \neq v_g$)
68. Set **type**(v_c) $= 0$;
69. $v_c = u$ where u is a child of v_c and **type**(u) > 0;
70. Move the agent to v_c;
71. Scan the graph for changes;
72. if changes are found
73. $k_m = k_m + h(v_{last}, v_c)$;
74. $v_{last} = v_c$;
75. **ProcessChanges**();

Figure 20: Main functions of IID* Lite

GetAlternativePath(v_c), at Line 35, 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. 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.

As mentioned earlier, function **ProcessChanges**() works as a "job distributor" by determining which function will be called depending on what kind of changes occur. Different from ID* Lite, at line 55, if *better* equals TRUE, **MiniCompute**() is called to propagate such changes. This may result in a better least cost path. Because in **Main**() at Line 66 **ComputeShortestPath**() is called regardless of whether **MiniCompute**() is called in **ProcessChanges**(), **GetAlternativePath**(v_c) is invoked at Line 56 to search for a path of cost no greater than Ω. If **GetAlternativePath**(v_c) succeeds, then the resulting vertex types infer a new optimal path in the current round. Otherwise, **ComputeShortestPath**() is called to propagate the underconsistent vertices that were inserted by **GetAlternativePath**(v_c) into OPEN.

ProcessChanges uses a threshold number t to impose an order on the propagation of values due to edge cost changes. At line 45, t is set to $rhs(v_c)$ which is the cost of original optimal path Ω. At Line 57, t_{old} records the value of t, **ComputeShortestPath**() is called to propagate changes, and t becomes v_c's new rhs value. If $t \leq t_{old}$, there must be a path of cost less than the original threshold and **GetAlternativePath**(v_c) is called to find it. Otherwise, t has increased and Lines 59-64 select the edge cost changes that are propagated. Some overconsistent vertices that are stored in *catch* may be placed in OPEN depending on the comparison results at line 61. These vertices were placed in *catch* to delay consideration until they could have a chance to affect discovery of a new "best" path, in this case of cost no greater than t. If there are any such vertices, **MiniCompute**() is called to propagate their changes. This is an iterative process that is executed in the while loop of Lines 56-64 as long as t is not decreasing, an optimal path has not been found, and *catch* is not empty.

It is technical about how to efficiently maintain the *catch*. As we discussed, only overconsistent vertices are needed to be inserted into *catch*. An obvious way is to keep such overconsistent vertices in a min-heap according to their $h + rhs$ values. Of course, k_m can be used to avoid reordering. But we find that as the agent is moving toward to v_g, normally the distance from

v_c to v_g is getting smaller. So if a overconsistent vertex is not reinserted into priority queue OPEN in current round, then probably it is not needed to be reinserted into OPEN in the following rounds. And hence the stored overconsistent vertices is seldom reinserted into priority queue OPEN again. Our experiments show that from v_s to v_g for average less than 10 percent of stored overconsistent vertices are reinserted into OPEN and recalculated; and operations on this min-heap cost about 1 percent of total used time. By the experiment, we can conclude that there is no need to maintain a min-heap for all overconsistent vertices. So a more efficient way is only to keep track of a small part of overconsistent vertices with smallest $h + rhs$ values.

Other algorithms have addressed the question of how to deal with underconsistent vertices. One of these is a variant of D* Lite called Delayed D* [9]. In every round, Delayed D* propagates only values of overconsistent vertices and postpones the propagation of values of underconsistent vertices. In some environments Delayed D* may perform worse than D* Lite since only decreased changes are propagated. Whenever a path is returned, it is checked for underconsistent vertices and, if it has none, the path is considered available and can be chosen as the "best" path between v_c and v_g; or if contains some underconsistent vertices, the path is considered unavailable and all underconsistent vertices on the path are inserted into OPEN. In the latter case, as for D* Lite, such underconsistent vertices are then propagated. This processing is repeated until one available "best" path is found. The reasons the propagation of values of underconsistent vertices are postponed by Delayed D* are that some of these may be able to be ignored later and many underconsistent vertices may be updated at the same time to avoid duplicate expansions of vertices.

IID* Lite propagates decreased changes in a manner similar to that of Delayed D* but with several differences. These are:

1. IID* Lite does not propagate all overconsistent vertices but only those that may result in a path whose cost is less than Ω'

2. IID* Lite propagates more inconsistent vertices in every partial recomputation.

3. All paths whose costs equal threshold number T are tested for availability simultaneously by function **GetAlternativePath**(v_c)

4. Once T is found to be greater than Ω, the more efficient **MiniCompute**() is called to see whether this new path can be improved using

unconsidered decreased changes.

5. IID* Lite considers more vertices for propagation per partial recomputation. It has been shown that under some conditions, for example when a agent is blocked, Delayed D* needs many times of partial recomputation until it finds out there is no path, hence performs worse than propagating all changes in one full recomputation. The performance of IID* Lite can be better under such circumstances. Because IID* Lite propagates more inconsistent vertices by comparing with threshold number T, not too many times of recomputation are needed. Also in IID* Lite, one parameter can be used to measure the possibility that the agent is blocked. As soon as the possibility is great enough, for example, if $T > 2 * \Omega$, all vertices are forced to be updated immediately, and *catch* is cleaned.

The value of these strategies is confirmed by the results of Section 3.2.5.

3.2.3 An example

Figure 21(a) shows a bi-directed graph which will be used to illustrate the operation of IID* Lite, particularly in comparison with the operation of ID* Lite. It is assumed that the cost of every available edge is 1 (that is, $h(u, w) = 1$ if $u \neq w$), there are no self-edges ($h(u, u) = \infty$) and the cost of a blocked edge (indicated by `X`) is ∞. The heuristic function h is then consistent. It is assumed that the *sensor-radius* is 4, that v_2 is initially blocked and that edge $\langle v_4, v_5 \rangle$ is initially blocked as shown in Figure 21(a). The start vertex is v_s and the goal vertex is v_g.

Initialization of the graph proceeds as in ID* Lite: for all v except v_g, $g(v) = rhs(v) = \infty$ and **type**$(v) = -1$; $h(v_g) = rhs(v_g) =$**type**$(v_g) = 0$; v_g is placed in OPEN; k_m is set to 0; v_c is set to v_s; and **ComputeShortestPath**() and **GetAlternativePath**(v_c) are called. Details showing how an initial "best" path is obtained are skipped and instead the reader is referred to Section 3.1.4 and Figure 8 to see how this would be accomplished. After initialization the "best" path is shown as the dashed line in Figure 21(a). The cost of this path is $\Omega = 6$.

The agent now moves along the dashed line from v_s to v_0, and v_c is set to v_0 and Ω becomes 5. Suppose at this point edge $\langle v_4, v_5 \rangle$ becomes unblocked and edges $\langle v_7, v_4 \rangle$ and $\langle v_7, v_8 \rangle$ become blocked as shown in Figure 21(b). This is discovered at Line 71 and causes **ProcessChanges**() to be called at

Line 75. Then at line 45, t is set to be 5. Since the cost of edge $\langle v_4, v_5 \rangle$ is now 1, $g(v_5) = 1$, $rhs(v_5) = 1$, $g(v_4) = \infty$. At Line 47 $rhs(v_4)$ is set to 2. At Lines 51-52, since $h(v_c, v_4) + rhs(v_4) = 1 + 2 < T = 5$, *better* is set to TRUE and v_4 is inserted into OPEN with priority $[3, 2]$. Also, $g(v_7) = 3$ and $rhs(v_7) = 5$ because $g(v_6) + c(\langle v_6, v_7 \rangle) = 4 + 1 = 5$. Since $h(v_c, v_7) + rhs(v_7) = 1 + 5 > T = 5$, v_7 is inserted into *catch* temporarily for later use and **type**(v_7) is set to -3 in Line 54. The g and rhs values of vertices v_5 and v_8 remain unchanged.

Since *better* is TRUE, **MiniCompute**() is called at Line 55. Since v_4 is at the top of OPEN, Lines 05-12 of **MiniCompute**() are executed resulting in the removal of v_4 from OPEN with $g(v_4) = 2$ and $rhs(v_4) = 2$. Vertex v_1 is a predecessor of v_4 and since $g(v_1) = 4$ from Line 06, $rhs(v_1)$ is set to 3 in Line 10. Then **UpdateVertex**(v_1) is called and v_1 is inserted into OPEN with priority $[4, 3]$. Now v_1 is at the top of OPEN and Lines 05-12 of **MiniCompute**() are executed. Thus v_1 is removed from OPEN with $g(v_1) = 3$ and $rhs(v_1) = 3$. Since v_0 is a predecessor of v_1, $g(v_0)$ is set to ∞ and $rhs(v_0)$ is set to 4. **UpdateVertex**(v_0) is called and v_0 is inserted into OPEN with priority $[4, 4]$. Another iteration of the while loop at Line 01 of **MiniCompute**() results in skipping Lines 10-12 and removing v_0 from OPEN. Since OPEN is now empty, execution returns to Line 55 of **ProcessChanges**(). Then function **GetAlternativePath**(v_c) is called to reset vertex types so that the highest priority shortest path from v_c *to* v_g may be found.

When **GetAlternativePath**(v_c) is called a cursor vertex r is set to v_0, **type**(r) is set to 1 in Line 29 because v_0 has successor v_1 and $rhs(v_0) = 4 = g(v_1) + c(\langle v_s, v_1 \rangle)$. Since **type**$(v_0) = 1 > 0$, r is set to v_1 in Line 31. Then **type**(v_1) is set to 1 in Line 29 because it has only one child, v_4, and r is set to v_4. Then **type**(v_4) is set to 1 in Line 29 because v_5 is v_4's only child. Next r is set to v_5 and **type**(v_5) is set to 1 with v_g as the only child, and execution stops. In Figure 21(b) the numbers above the vertices show the vertex type numbers after **GetAlternativePath**(v_0) is called. Implicitly, the current "best" path follows the vertices of non-negative type numbers starting at v_0 and this path is shown as the dashed line in the figure. Since **GetAlternativePath**(v_0) has returned TRUE, t is not incremented and no further edge cost changes are considered.

The agent begins to move along the current path starting from $v_0 = v_c$. Line 68 of **Main**() sets **type**(v_0) to 0. In Line 69, v_c is set to v_1, and in Line 70 the agent is moved to $v_1 = v_c$. Suppose edge $\langle v_4, v_4 \rangle$ becomes blocked and edge $\langle v_7, v_8 \rangle$ becomes unblocked. This new situation is shown in Figure 21(c)

and is detected in Line 71 of **Main**(). Ω is now 3. As above, **Process-Changes**() is called. Then at line 45, t is set to 5. At Line 47, $g(v_7) = 3$ and $rhs(v_7)$ is set to 3. At Line 51, because $h(v_c, v_7) + rhs(v_5) = 1 + 3 > t = 3$, v_7 is not inserted into OPEN and **type**(v_7) is set to -3. Iterating the for loop at Line 46, since **type**(v_7) $= -3$, **GetBackVertex**(v_7) is executed at Line 48, and **type**(v_7) is set to 0 at Line 49. Iterating the for loop, values of vertex v_4 are updated to $g(v_4) = 2$ and $rhs(v_4) = 4$ because $g(v_1) + c(\langle v_1, v_4 \rangle) = 3 + 1 = 4$. Since $h(v_c, v_4) + rhs(v_4) = 1 + 4 > T = 3$, v_4 is stored in *catch* with **type**(v_4) $= -3$. Values of vertices v_5 and v_8 remain unchanged. Since no vertices have been inserted into OPEN, **MiniCompute**() is not called, but **GetAlternativePath**(v_c) is called directly in Line 56. In **GetAlternativePath**(v_c) a cursor vertex r is set to $v_c = v_1$, **type**(r) is set to 0 in Line 29, hence v_4 (with type value -3) is inserted into OPEN. **GetAlternativePath**(v_c) returns FALSE. Hence t will increase and more edge cost changes will be considered. At line 57, t_{old} is set to 3, and after execution of **ComputeShortestPath**, $rhs(v_c)$ is updated to 4, so t becomes 4. In line 58, since $t > t_{old}$, Lines 59-64 are executed. But there is no edge cost change that satisfies the condition of Line 61, so Line 56 is executed again.

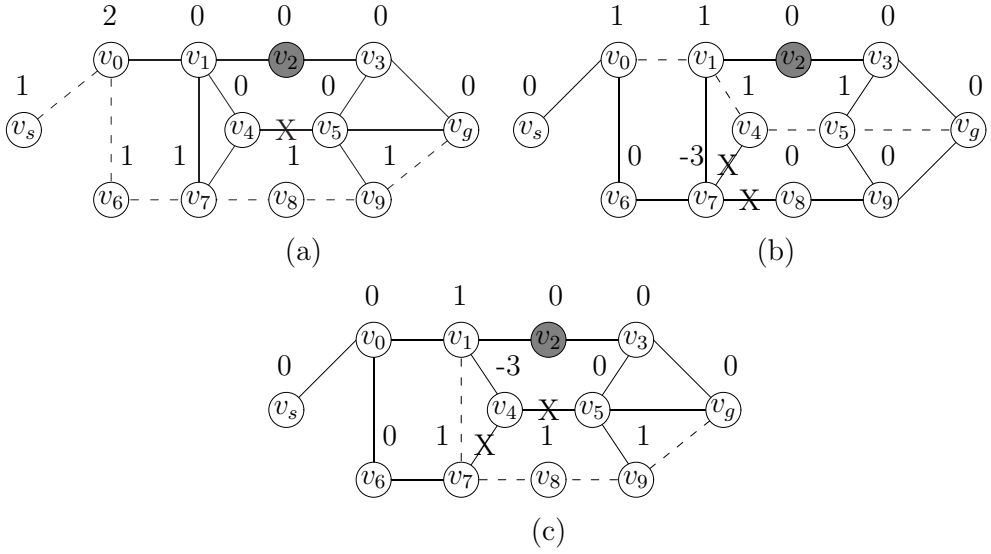


Figure 21: The action of ID* Lite on a small example

When **GetAlternativePath**(v_c) is called the second time a cursor vertex r is set to $v_c = v_1$, **type**(r) is set to 1 in Line 29 because $v_c = v_1$ has successor v_7 and $rhs(v_1) = 4 = g(v_7) + c(\langle v_1, v_7 \rangle)$. Since **type**($v_1$) $= 1 > 0$, r is set to v_7 in Line 31. Then **type**(v_7) is set to 1 in Line 29 because it has only one child v_8 and r is set to v_8. Then **type**(v_8) is set to 1 in Line 29 because v_9 is v_8's only child. Next r is set to v_9, **type**(v_9) is set to 1 with v_g as the only child, and **GetAlternativePath**(v_c) returns TRUE. In Figure 21(c) the numbers above the vertices are the type numbers of those vertices at this point. Implicitly, the current "best" path follows the vertices of non-negative type numbers starting at v_1 and is shown as the dashed line in the figure.

3.2.4 Analysis of IID* Lite

In this section we show that IID* Lite returns a least cost path between v_c and v_g on every round. As with D* Lite, the heuristic function $h(w, u)$ is assumed to be consistent.

Lemma 16 *In IID* Lite, function* **MiniCompute***() can only update value $rhs(v_c)$ to be less than or equal to its formal value and, after execution of* **MiniCompute***(), $rhs(v_c) \leq t$.*

Proof: In **MiniCompute**(), if a vertex v becomes underconsistent, that is, $g(v) < rhs(v)$, it will be deleted and hence not cause further propagative updating. This implies that all vertices that are further processed in **MiniCompute**() are overconsistent. Moreover, values that are propagated from v cannot increase the rhs value of their neighboring vertices including v_c if it is a neighbor. That means $rhs(v_c)$ cannot increase. This proves the first part of the lemma.

To prove the second part, observe that in D* Lite, no inconsistent vertex is deleted and all inconsistent vertices will be further processed and $rhs(v_c)$ will take the value t. So we only need to show that $rhs(v_c)$ in IID* Lite is less than or equal to $rhs(v_c)$ in D* Lite. **ProcessChanges**() in IID* Lite combined with Proposition 9 means that all overconsistent vertices causing $rhs(v_c)$ to decrease will be further processed. So in comparison to D* Lite, IID* Lite only process vertices that in D* Lite will cause $rhs(v_c)$ to decrease but does nothing to vertices causing $rhs(v_c)$ to increase in D* Lite. So, $rhs(v_c)$ in IID* Lite is less than or equal to what it would be in D* Lite. It follows from the correctness of D* Lite that $rhs(v_c)$ returned by **MiniCompute**() is no greater than t. □

Lemma 16 shows that **MiniCompute**() can find paths of cost no greater than those found by D* Lite. The intuitive explanation for this is as follows. Assume in D* Lite there is a shortest path p' passing through only one decreased cost change $e = \langle w, u \rangle$ and only one increased cost change $z' = \langle v, r \rangle$. In IID* Lite, if v has $g(v) < rhs(v)$, v's values will not be propagated immediately, so w's values will not be affected by v and $rhs(w)$ can only become less. Then the cost of p' in IID* Lite will be less than that in D* Lite. Furthermore, since the cost of the least cost path returned by **MiniCompute**() is less than that of p', p' cannot be "found" by **GetAlternativePath**(v_c) even though it is the least cost path. Actually, p' could be "found" by **GetAlternativePath**(v_c) but considered unavailable until v is checked. In that case, the strategy that inserting v into the priority queue and re-calculating in **ComputeShortestPath**() can be applied to find it. The reason for doing this is efficiency. Since not all vertices satisfying the equation in Proposition 9 are propagated immediately in **ProcessChanges**(), it is possible that some shortest path found by **MiniCompute**() will be unavailable when checked by **GetAlternativePath**(v_c) due to inconsistency. To summarize, a vertex v satisfying the equation in Proposition 9 but with $g(v) < rhs(v)$ will not be propagated. To deal with this situation we have the following lemma.

Lemma 17 ProcessChanges() *exits from Line 56 if and only if a least cost path traversing consistent vertices has been found.*

Proof: When function **ProcessChanges**() exits the loop at line 56, suppose the cost of the new least cost path is t. By Lemma 10, when line 56 is executed for the first time, if function **GetAlternativePath**(v_c) returns TRUE, one path p' passing through consistent vertices has been found. Suppose its cost value is c. Since, by assumption, t is the value of current least cost path, $c \geq t$. By Lemma 16 $c \leq t$. Hence $c = t$. Therefore, that if **GetAlternativePath**(v_c) returns TRUE, a least cost path is found.

If **GetAlternativePath**(v_c) returns FALSE then any vertex v that is underconsistent ($g(v) < rhs(v)$) must be inserted into OPEN. From Lines 56-64 in **ProcessChanges**(), following the operation of **ComputeShortestPath**() in D* Lite, the underconsistent vertex values will be updated to make those vertices consistent. From the operation of D* Lite and by Proposition 9, it can be proved that $rhs(v_c) \leq t$, as was done in the proof of Lemma 16, even though some underconsistent vertices are processed in

IID* Lite. That is because the underconsistent vertices propagated by IID* Lite are a subset of underconsistent vertices processed by D* Lite and the overconsistent vertices whose propagation is postponed by IID* Lite cannot be part of any path whose cost is no greater then t. So, for an arbitrary consistent vertex v with $rhs(v) \leq t$, $rhs(v)$ will also be less than the cost of the path returned by D* Lite. When the loop in Lines 56-64 is repeated, if **GetAlternativePath**(v_c) can return TRUE, using the same argument as above, the path found is of least cost.

If FALSE is continually returned by **GetAlternativePath**(v_c), more and more inconsistent vertices will be reinserted into OPEN and updated. Since the number of inconsistent vertices is finite, in the worst case all inconsistent vertices are updated as is done in D* Lite. According to the correctness of D* Lite, all vertices on least cost paths from v_c to v_g will be consistent and $rhs(v_c) = t$. In that case **GetAlternativePath**(v_c) returns TRUE. □

Lemma 17 shows that IID* Lite runs an incremental like search within every round. The rough idea is to search the shortest path incrementally from low value to high value.

Lemma 18 ProcessChanges*() finds the least cost path in every round if and only if there exists one.*

Proof: This follows from Lemma 17. □

Theorem 19 *In every round, IID* Lite returns a least cost path from v_c to v_g if and only if a path exists.*

Proof: This follows directly from Lemma 18. □

The above analysis shows that the improvements proposed to ID* Lite do not require any special property of the terrain information that an agent may encounter. Yet, the proposed improvements potentially speed up D* Lite by avoiding vertex updates until they are shown to be necessary. The improvements can be embedded into any algorithm which uses D* Lite as a foundation.

3.2.5 Experiments and Results

In this section, the performance of IID* Lite is compared experimentally to the performance of D* Lite and delayed D* algorithms on random grid-world terrains. The experiments are the same as the ones described in Section 3.1.6.

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. Recall that parameter *percent* is the fraction of vertices that are initially blocked and *sensor-radius* is the maximum cost of a path from v_c to an observable vertex.

Results on random rock-and-garden benchmarks are given first. The results are averaged over at least 100 independent runs of each algorithm. Because we found that Delayed D* can perform quite differently depending on different instances, the results of Delayed D* are averaged over at least 1000 independent runs.

In Figures 22 and 23 $size = 300$, and $percent = 10$ and $percent = 30$, respectively. The left graph of each figure shows the number of heap operations as a function of *sensor-radius*. As before, the plots show only the heap operations in re-calculations. The right graph of each figure shows the ratio of the number of re-calculations to the number of edge cost changes observed within the *sensor-radius*.

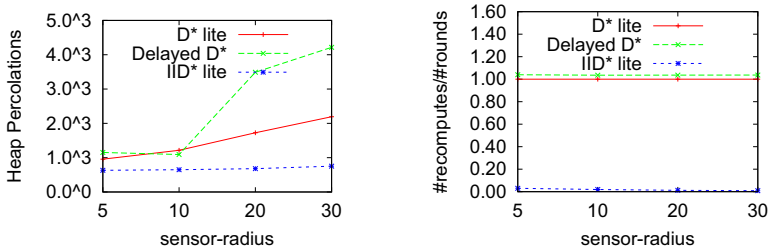


Figure 22: size=300 and percent=10

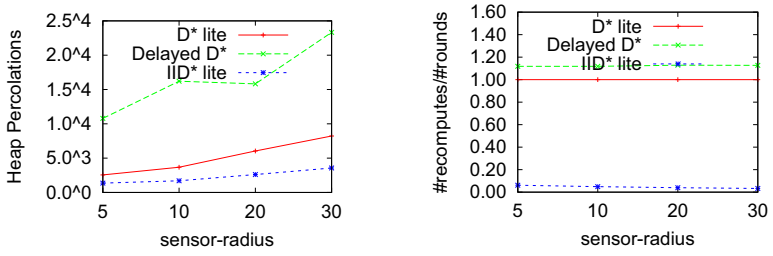


Figure 23: size=300 and percent=30

In [9], a D* Lite variant Delayed D* has been introduced and it is stated that Delayed D* can outperform D* Lite by roughly a factor of 2. In that

paper three kinds of experiments are introduced. In the first one, relatively a small number of changes are observed in the grid world, and delayed D* performs well because there are just a few decreasing changes that cause overconsistent vertices. This partially explains why, in Figure 22, Delayed D* performs better than IID* Lite when the *sensor-radius* is small.

In the second kind of experiment, Delayed D* is compared with D* Lite in completely unknown environments and Delayed D* showed a slight performance improvement. In this proposal, due to the way *sensor-radius* is used and the assumption that all initialized maps are blank, all our experiments simulate completely unknown environments. In the rock-and-garden benchmarks, there are no decreasing changes so the performance of delayed D* is not as good as the other algorithms. This is why Delayed D* suffers a lot in our experiments.

Experiments of the third kind are run in partially known environments, where there are many vertices whose priorities are higher than $rhs(v_c)$. In this kind of experiment, the possibility is high that the propagation of overconsistent vertices can generate a new least cost path. Delayed D* performs much better than D* Lite in this kind of environment.

By observing Figures 22 and 23 we can observe that IID* Lite not only performs better but also its heap percolation increases more slowly than other the other algorithms as *sensor-radius* increases. Hence we can conclude that in random rock-and-garden benchmarks, IID* Lite is best when *sensor-radius* is large, which is probably a realistic assumption given advancing technology. Figures 22 and 23 also hint at the kind of environment for which Delayed D* can outperform D* Lite.

The right side graphs of Figures 22 and 23 show a flat curve at 1 for D* Lite because every time an inconsistency is observed, exactly one re-calculation must be performed. Delayed D*'s curve is always above 1 because at least one re-calculation must be performed to propagate overconsistent vertices for every round, and in order to guarantee availability of the found least cost path some extra re-calculations to propagate underconsistent vertices may be needed [9]. The curve for IID* Lite stays much below 1 since re-calculations may be skipped when alternative paths are found. It should be pointed out that the numbers plotted in the figure include calls to **MiniCompute**() which runs much faster than **ComputeShortsetPath**(). It should also be pointed out that IID* Lite may have a ratio greater than 1 in some environments as will be illustrated below.

Results shown in Figures 24 and 25 come from experiments where $size =$

300, and *sensor-radius* is 10 and 30 respectively. The left graph of both figures shows the number of heap operations as a function of *percent* given the other parameters are fixed. The right graph of both figures shows the ratio of the number of re-calculations to the number of changes observed. From these figures it can be concluded that ID* Lite typically performs better than D* Lite, and in rock-and-garden benchmarks IID* Lite is least sensitive algorithm to the number of edge cost changes.

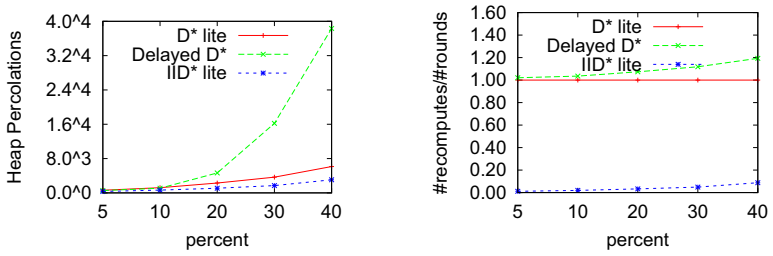


Figure 24: size=300 and sensor-radius=10

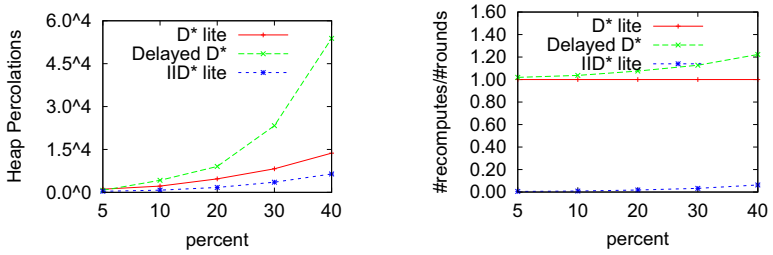


Figure 25: size=300 and sensor-radius=30

Results shown in Figure 24 show that Delayed D* performs best when *sensor-radius*=10. Considering the above discussion, it is easy to understand that when *sensor-radius* increases, there are more underconsistent changes, causing ID* Lite's performance to degrade. But why does the performance of Delayed D* suffer when *sensor-radius*=5. The reason has been discussed in [9] where it is introduced as one possible source of inefficiency. In Delayed D*, after propagating overconsistent vertices, underconsistent vertices may be needed to be inserted into OPEN to re-calculate to establish the availability of found least cost paths: if underconsistent vertices exist on the path found the path is unavailable, and hence such underconsistent vertices are

inserted into OPEN and their updated and propagated. If the underconsistent vertices are recomputed in several times, then this the performance of Delayed D* Lite suffers. When the *sensor-radius* is 5, the underconsistent vertices locate close to each other and there is an increased chance that the aforementioned bad scenario for Delayed D* will happen. But because *sensor-radius* is small, there are relatively few changes to observe and the difference in performance between Delayed D* and D* Lite may not be great. If the number of changes is certain, Delayed D* will perform better than D* Lite when *sensor-radius* is large.

Results presented in Figure 26 show the number of heap operations as a function of *size* when *sensor-radius*$= 0.1 * size$ and $percent = 30$. IID* Lite performs the best among the three algorithms

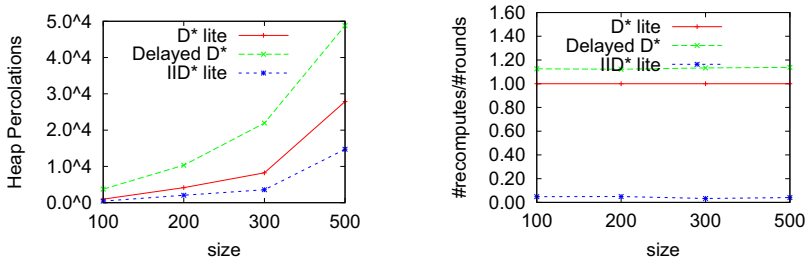


Figure 26: sensor-radius=0.1*size and percent=30

Next the performance of D* Lite, Delayed D* and IID* Lite on parking-lot benchmarks is discussed. 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. Because parking-lot-benchmark can be classified as having partially known environments, and there are many changes in every round, as discussed above, Delayed D* will be handicapped in these benchmarks. Since showing the performance results of delay D* would force a change of scale of the plots, they are not presented in the figures referred to below.

For experiments whose result are shown in Figures 27 and 28 $size = 300$, and $percent = 10$ and $percent = 30$, respectively. The left graph of both figures shows the number of heap operations as a function of *sensor-radius* given the other parameters are fixed. The results show that IID*

Lite outperforms D* Lite by nearly an order of magnitude. The results of Figures 27 and 28 show that in IID* Lite heap percolation increases more slowly that for D* Lite as *sensor-radius* increases. This result is similar to that for the rock-and-garden benchmarks. The right graphs of Figure 28 show that IID* Lite's ratio is greater than 1 when *sensor-radius*$\geq$ 20. That means, on the average, IID* Lite re-calculates more than once when changes are observed. The re-calculations tabulated in those figures include calls to **MiniCompute**().

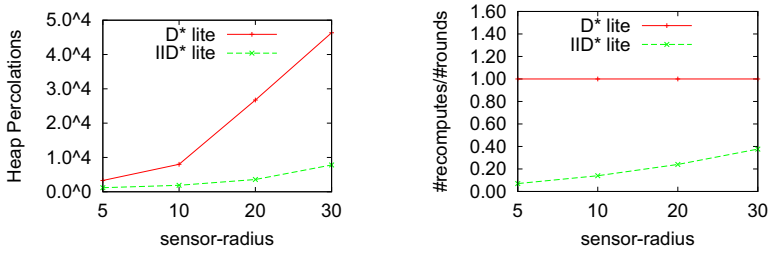


Figure 27: size=300 and percent=10

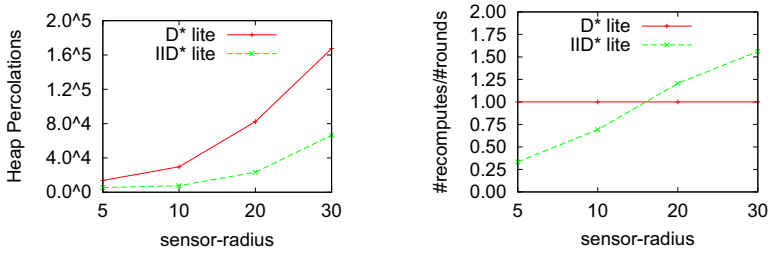


Figure 28: size=300 and percent=30

In Figures 29 and 30 *size* = 300 and *sensor-radius* is 10 and 30, respectively. The left graph of both figures shows the number of heap operations as a function of *percent* given the other parameters are fixed. The right graph of both figures shows the ratio of the number of re-calculations to the number of changes observed. The results show that IID* Lite performs better than D* Lite on the average. From Figures 29 and 30 we conclude that in IID* Lite heap Percolation increases more slowly than in D* Lite as *percent* increases. That is, in the parking-lot benchmarks, IID* Lite is less sensitive to edge cost changes than is D* Lite. We also conclude that IID*

Lite has a better ability to handle intensely changing environments than D* Lite. When $percent \geq 40$, because the agent is blocked so often, there is no big performance difference that is observed among all the algorithms. Actually, under such environments, D* Lite may perform better than Delayed D* and IID* Lite. In that case, one can apply ID* Lite instead of the other algorithms.

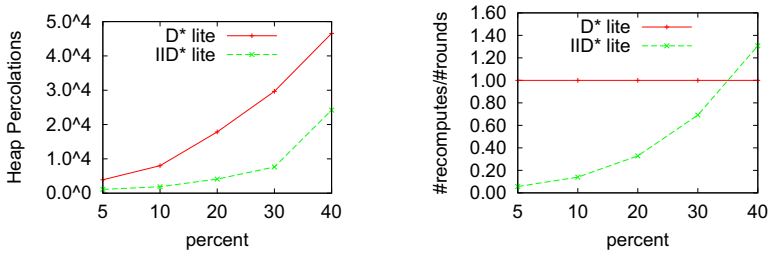


Figure 29: size=300 and sensor-radius=10

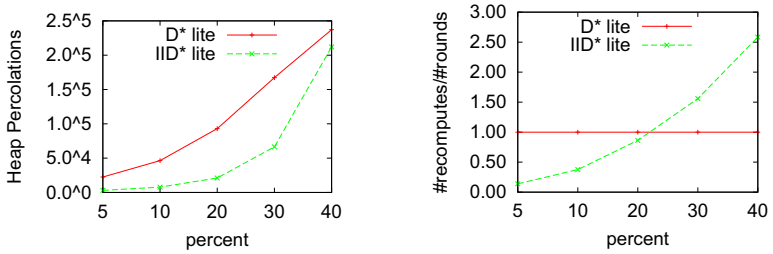


Figure 30: size=300 and sensor-radius=30

Figure 31 shows the number of heap operations as a function of *size* where *sensor-radius*$= 0.1 * size$ and $percent = 30$. IID* Lite performs better than D* Lite from $size = 100$ to $size = 500$.

Since IID* Lite needs to calculate an alternative which D* Lite does not, therefore although IID* uses less heap operations than D* and its alternative calculation cost is little, it is more assuring to do experiments and see if heap operation cost analysis coincide with operation time they spend. In Figure 32 and Figure 33 we show operation time comparison of IID* and D* respectively under the same setting used for experiments of Figure 26 and Figure 31 both of which respectively show comparison of heap operation

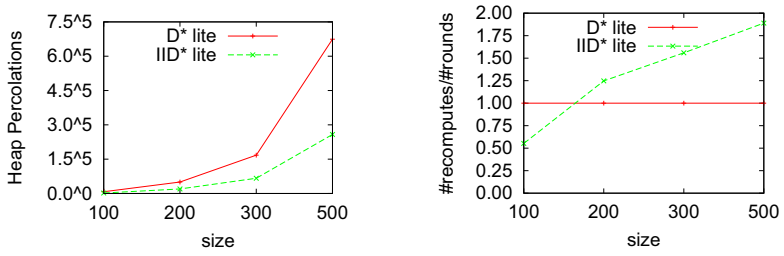


Figure 31: sensor-radius=0.1*size and percent=30

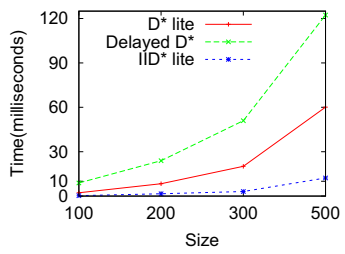


Figure 32: rock-and-garden

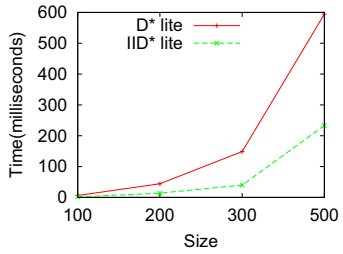


Figure 33: parking-lot

cost of IID* and D* Lite. Compare the graph in Figure 32 with the one in Figure 26 we can see the IID*'s curve of operation time has the same trend as its curve of heap operation cost, which shows heap operation cost contribute the main complexity(also time) as we predicted. Comparison between graphs in Figure 33 and Figure 31 say the same thing. By statistics the percentage of calculating alternatives is less than 1% of the total time needed. The results prove correctness of our analysis. There is an interesting phenomenon: when the priority queue is relatively small, the heap operation can be executed more efficiently. As IID* Lite delays propagation of unnecessary changes, its priority queue is smaller than that D* Lite's. So having the above time comparison, sometimes IID* Lite speeds up greater and saves more time than heap operations.

By experiments and analysis we conclude that although IID* Lite can not theoretically guarantee always performing better than D* Lite, IID* Lite practically performs better than D* Lite under various random benchmarks. Specifically IID* Lite gets up to 8 times speeding up compared with D* Lite. Allying with results analyzed for ID* Lite in Section 3.1, IID* Lite also works much better than ID* Lite.

3.3 Summary

Two improvements to the D* family of algorithms have been proposed. One improvement attempts to find an alternative least cost path before re-calculating values when the current least cost path becomes inconsistent. The second improvement is a systematic way to consider edge cost changes in order when re-calculation is necessary so that a least cost path may be found before all changes are propagated. In particular, values of overconsistent vertices are propagated with a priority based on g and rhs values and underconsistent vertices are only inserted into OPEN when looking for a consistent alternative to the current least cost path. Both improvements have been implemented under the name IID* Lite. The potential for performance improvement over other algorithms in the D* family is made apparent from the results of experiments on a variety of benchmarks including rock-and-garden and dynamic navigation families of terrain. Experiments show IID* Lite can get up to 8 times speedup over D* Lite. Experiments also show that IID* Lite is less sensitive than D* Lite to the number of changes observed in a round. The importance of this is expected to increase with advancing technology that will support finer grain simulations and allow robots to "see" further. Finally, experiments show that in terrains where the agent can be blocked often it is better to use only the first improvement since all vertices reachable from v_g have to be expanded anyway.

4 Anytime Dynamic Navigation Algorithms

Dynamic navigation algorithms play a crucial role in the reliability and functionality of autonomous vehicles. These algorithms help an agent find a near optimal path to a goal in changing environments. An optimal path can be found when resources are sufficient and some algorithms are able to do this well [14, 15, 31, 32, 40, 41, 43]. But, in environments where there are insufficient resources an agent cannot efficiently compute an optimal path and a sub-optimal path must be accepted. Chapters 1 to 3 were concerned with finding optimal and sub-optimal paths given as much time as needed for the computations. This chapter is concerned with environments where time is more critical than optimality and where a sub-optimal solution is the best that one can hope to obtain.

Time-limited search algorithms, called anytime planning algorithms, have been developed for time-critical environments. The basic idea is, first, find and store some path as soon as possible, then progressively replace the stored path with a better one, if found during search, until the time available for search expires [10,11,45]. At that point, the stored path becomes the agent's path. As examples [10, 19, 29] define and demonstrate an algorithm named Weighted A* which uses "inflated" heuristics (described below) to expand fewer vertices than the normal A* algorithm does.

In A*, the vertices in OPEN are sorted by their values $f = g + h$. In A*, if h is assumed to be admissive and if $f = g + \epsilon \cdot h$, then the returned path can be guaranteed to be ϵ sub-optimal, i.e. $g(v_g) \leq \epsilon \cdot g^*(v_g)$ [5]. This strategy is called inflated heuristics and it is used to control ϵ sub-optimality. The Weighted A* algorithm is the A* algorithm using inflated heuristics. In [10] the Anytime Weighted A* algorithm is presented and shown to provide a general method for transforming heuristic search algorithms to anytime algorithms. The Anytime Weighted A* algorithm is an anytime planning algorithm continuously improves the current path, starting from a sub-optimal path that is quickly calculated, until an optimal path is obtained.

The Anytime Weighted A* algorithm does the same initialization as the A* algorithm and the Weighted A* and the heuristic function h used is admissive. Its difference from the A* algorithm is that p is used to record the current returned path which may be improved later; ERROR is used to estimate how far away the cost of the current solution is from that of the optimal path; ϵ is the parameter used to inflate h; in a vertex $v \in OPEN$, the stored values related to it are $\langle g(v), f'(v) \rangle$ instead of $\langle g(v), f(v) \rangle$, where

$f'(v) = g(v) + \epsilon \cdot h(v)$: that is, the vertices of the priority queue OPEN in the Anytime Weighted A* algorithm are sorted by f' values which are different from the f values used in the A* algorithm. The Anytime Weighted A* algorithm uses inflated heuristic value f' to sort vertices, and the normal f values are recorded to prune the search space [11].

In a changing environment, searching for a path between a fixed pair of vertices with limited available time is mitigated by the Anytime Repairing A* algorithm (ARA*) [21], a member of the family of incremental anytime algorithms. ARA* runs the Weighted A* every time a change in path optimality is observed to find a new (sub-optimal) path. The performance of ARA* mainly depends on how much previously calculated information it uses to avoid replicating computation. This is in accordance with ideas taken from [16, 17]. In Weighted A* when h is admissive, if every vertex is allowed to be expanded only once, then the returned path is still ϵ sub-optimal. Due to this property, every time ARA* needs to recalculate, it updates a vertex at most one time. ARA* starts with a large value for the so-called inflated parameter ϵ and then reduces ϵ on each succeeding round until either $\epsilon = 1$ or the time available expires. ARA* performs in a manner similar to that of Anytime Weighted A* and has the ability to control the sub-optimality parameter ϵ.

The D* Lite algorithm can be regarded as a dynamic version of the Lifelong A* algorithm [16, 17]. As is the case for the D* algorithm [31, 32], D* lite searches backward from v_g to v_s. This is likely the critical point for the success of D* and its variants 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 a vertex v *locally consistent* if $rhs(v) = g(v)$, *locally overconsistent* if $rhs(v) < g(v)$, and *locally underconsistent* if $rhs(v) > g(v)$. A vertex that satisfies either of the latter two cases is said to be *inconsistent*. A "best" path can be found if and only if, after expansion of v_s, all vertices on the current path are locally consistent and can be computed by following the maximum-g-decrease-value vertices one by one from the goal v_g. If some change that is made in the previous round causes a vertex v to become inconsistent, then D* Lite updates $g(v)$ to make v locally consistent by setting

$g(v) = rhs(v)$ (Because D* Lite only propagates inconsistent vertices to update some vertices (g, rhs) values instead of updating all vertices values, D* Lite can perform much better than other navigation algorithms).

The Anytime D* algorithm [20] is intended 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-sub-optimality until $\epsilon = 1$ or time runs out. Thus, Anytime D* will try to give a relatively good and available path quickly. Moreover, if time allows, it tries to improve the returned path incrementally as is the case for Anytime A*.

The work in [40, 41] improves the efficiency of the D* Lite algorithm by avoiding unnecessary calculations even further. Note that if the original optimal path is still available and cannot be improved from observed changes, the path will be chosen without re-calculating. A new algorithm named IID* Lite, proposed in 3.2, maintains a threshold number to control the propagation of inconsistent vertices. Only when the weight of a path is less than or equal to the threshold number are changes associated with inconsistent vertices on the path propagated. The threshold number is increased until an optimal path is found or all inconsistent vertices have been updated.

Two criteria are used to judge the performance of dynamic anytime algorithms. One is the time an algorithm uses to compute the first qualified sub-optimal path in the course of recalculation, and the second is the sub-optimality of the final path an algorithm outputs after recalculation. Similar criteria are mentioned in [33]. An anytime algorithm is designed to find the first sub-optimal path as soon as possible, and then to iteratively improve this path until time runs out. In Section 4.1, a new anytime dynamic navigation algorithm IAD* [42] is proposed and compared with AD* using the first criterion. It is shown that IAD* gains up to an order of magnitude speed up over AD* in experiments. In Section 4.2 another dynamic anytime DAWA* is proposed and compared with IAD* and AD*. Since, as will be shown, DAWA* has the same performance as IAD* with respect to the first criterion, the algorithms are compared using only the second criterion.

4.1 Improved Anytime D* Algorithm(IAD*)

In this section a description of the Improved Anytime D* (IAD*) algorithm is given. When changes are observed, IAD* will tries to update inconsistent vertices to get a new sub-optimal path in a manner similar to that of Anytime D* (AD*). For every update it is required to return a new sub-optimal path whose weight is no greater than $\epsilon' \cdot g^*$ where ϵ' is a preset value that controls sub-optimality and g^* is the weight of current optimal path. The sub-optimality parameter, denoted here as ϵ, is different from ϵ' as described below.

Every time an update is needed ϵ is reset to be a relatively large value denoted ϵ'. Doing so results in the discovery of a path, although sub-optimal, as soon as possible [10,19,29]. In [35], it is recommended that in Weighted A* ϵ can be set to any value greater than ϵ'. Experiments show that this results in the first path being returned fastest for that algorithm. This technique can be easily combined with any Weighted A* variant. In order to speed up returning the first path, AD* allows a vertex to be expanded at most once. The reason for why this condition would work is given in [20]. However, in some benchmarks this condition may delay propagation of some critical vertices and slow down the speed [10]. Experimental results are presented and analyzed in Section 4.1.2.

All anytime variants try to improve the current path until time runs out or an optimal path is found. In AD* each update decreases the value of ϵ, starting from ϵ', until it equals 1, which means the returned path is optimal, or time runs out. When ϵ is decreased, all inconsistent vertices are inserted into a priority queue to be updated. Compared with AD*, IAD* does no update the first time changes are observed: this is just the same strategy that is used in IID* Lite 3.2. Instead, IAD* tries to find a consistent and ϵ sub-optimal path which is not affected by the changes. If such a path exists, IAD* simply returns the path; if such a path does not exist, IAD* will update inconsistent vertices part by part until an ϵ sub-optimal found. The method used to choose inconsistent vertices to be updated is similar with IID* Lite, and is discussed in Section 4.1.1.

4.1.1 Pseudo Code of IAD*

The pseudo code of IAD* is listed in Figure 34. **Initialize()** initializes ϵ, the priority queues OPEN, CLOSED, and INCONS, values for g, rhs, and *type*

values of vertices. The initial value of ϵ' is relatively large in order to make sure a path can be returned quickly. Vertex v_g and its key are inserted into priority queue OPEN.

In Function **key**(s), underconsistent vertices have their key-values updated to $g(s) + h(s)$. This guarantees that increased changes can be propagated. Function **UpdateVertex**(s) updates a vertex in the same way as Anytime D* which uses INCONS to store some of the inconsistent vertices and makes sure that each vertex is expanded at most once in an execution of **ComputeOrImprovePath**(). After doing this, a solution satisfying ϵ sub-optimality is returned [20].

Functions **ComputeorImprovePath**(), **MiniCompute**() and **GetBackVertex(v)** are the same as in IID* Lite. Function **GetAlternativePath**(v_c) returns TRUE if and only if there is a path from v_c to v_g. If it returns TRUE, it means that type values on vertices have been changed so that an ϵ sub-optimal path from v_c to v_g can be traversed by visiting neighboring vertices with positive type values until v_g is reached. At line 05 of **GetAlternativePath**(v_c) a successor y of r with $rhs(y) + c(r, y) \leq rhs(r)$ is chosen instead of a child of r: this action is different from IID* Lite. The reason for the change is that in this algorithm only a sub-optimal path is required and this results in getting a path faster.

Different from IID* Lite, IAD* may not return an optimal path but can guarantee an ϵ sub-optimal path. Moreover, it is worth noting that if the returned path contains overconsistent vertices, then it is better than an ϵ sub-optimal path. If no path is returned by function **GetAlternativePath**(v_c), every vertex $c \in C$ will be updated by function **UpdateVertex**. Note that c is underconsistent and that this is the only place in the code where underconsistent vertices are placed into OPEN. This is because only increased changes will cause underconsistent vertices and increased changes are only processed here. Decreased changes will always be inserted before function **GetAlternativePath**(v_c) is called.

ProcessChanges acts in a way that is similar to the way it does in IID* Lite. The difference only lies at lines 07 and 17 where $\epsilon * h(v_c, u) + rhs(u) < t$ instead of $h(v_c, u) + rhs(u) < t$ is used to test whether an overconsistent vertex should be put into OPEN for propagation. Functions **Main**() and **MoveAgent**() are the same as AD*.

This subsection concludes with a statement of correctness of IAD*. In the next subsection experimental results of IAD* will be presented and compared with results for AD*.

Procedure Initialize()
01. OPEN = CLOSED = INCONS = catch=$\emptyset$;
02. for all $v \in V$, $rhs(v) = g(v) = \infty$; $\mathbf{type}(v) = -1$;
03. $rhs(v_g) = g(v_g) = \mathbf{type}(v_g) = 0$; $\epsilon = \epsilon'$
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($[s, \mathbf{key}(s)]$);$\mathbf{type}(v) = 0$;
08. else
09. insert s into INCONS;

Procedure ComputeOrImprovePath():
01. while (OPEN.TopKey() $<$ $\mathbf{key}(v_s)$ 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() $<$ $\mathbf{key}(v_c)$)
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 sub-optimal 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. if significant edge cost changes were observed
08. increase ϵ or replan from scratch;
09. else if $(\epsilon > 1)$
10. decrease ϵ;
11. CLOSED $= \emptyset$;
12. **ProcessChanges**();
13. publish current ϵ sub-optimal solution;
14. if $(\epsilon == 1)$
15. wait for changes in edge costs;

Procedure MoveAgent():
01. while $(v_s \neq v_g)$
02. wait until a plan is available;
03. Set $\mathbf{type}(v_c) = 0$;
04. $v_c = u$ where u is a successor of v_c and $\mathbf{type}(u) > 0$;
05. Move the agent to v_c;

Figure 34: Main functions of IAD*

Theorem 4.1 *The path between v_c and v_g that is returned by the Improved Anytime D* algorithm has a cost no greater than $\epsilon' * g'(v_c)$ where $g'(v_c)$ is the cost of optimal path between v_c and v_g.*

Proof: It follows from the correctness of IID* Lite algorithm and Anytime D* algorithm.

4.1.2 Experiments and Analysis

In this section, the performance of IAD* is compared experimentally with that of AD* on random grid-world terrains. In each experiment the terrain is a square, 8-direction grid world of $size^2$ vertices. Vertices v_s and v_g are chosen randomly from the terrain. Initially $percent\% * size^2$ of the vertices are randomly selected as blocked points where percent is 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. Consistent heuristic function similar with Manhattan distance is used. Before navigation, the traveling agent has a initial map, in which an obstacle may be wrongly considered to be blank with fifty percent possibility.

To compare anytime dynamic navigation algorithms, the sub-optimality parameter ϵ is controlled so that solutions returned by the algorithms are ϵ sub-optimal. How many operations are needed to compute the first sub-optimal path in the course of recalculation is considered as a criterion of anytime algorithms. The less time that is used, the better the algorithm is. Experiments are run on IAD* with AD* for the same value of ϵ using two kinds of random benchmarks. The first set of results are on random rock-and-garden benchmarks where an initially set blockage remains for the entire experiment. The second set of results are on a collection of benchmarks, called parking-lot benchmarks, that model navigation through changing terrains. In that case a blockage may move to its neighborhoods randomly in the course of navigation.

Form Figure 35 to 41, IAD* and AD* are compared on rock-and-garden benchmarks. Figures 35 and 36, respectively, display the results of $size = 300$ and $size = 500$ with $percent = 10$ and $sensor - radius = 0.1 * size$. In each figure only heap operations in recalculations are compared and the number of heap operations is a function of sub-optimality Epsilon(ϵ). Also the times consumed by recalculations are compared. Compared with AD*, the time consumed by IAD* includes the extra time that is needed to calculate the alternative path.

Figures 35 and 36 show IAD* gains an order of magnitude speed up over AD*. Thus, the time it takes IAD* to calculate alternative paths is so short that the time it takes to recalculate is barely affected. Correlating the number of heap operations with the actual time consumed shows that heap operations dominate the complexity of navigation algorithms as expected.

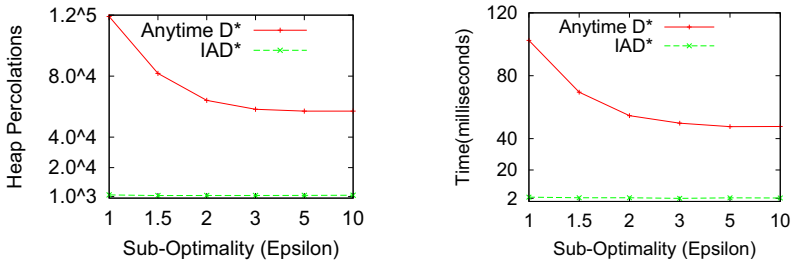


Figure 35: size=300, percent=10, sensor-radius=30(rock-and-garden)

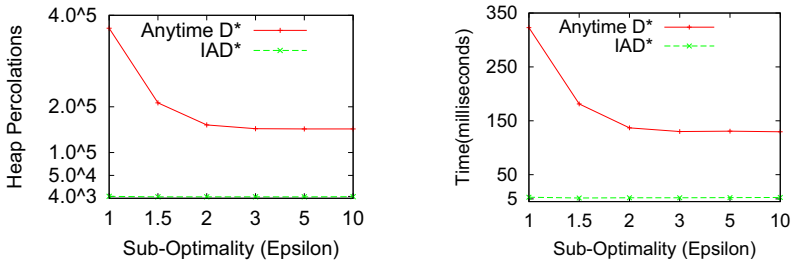


Figure 36: size=500, percent=10, sensor-radius=50(rock-and-garden)

The speedup of IAD* over AD* is even greater than that of IID* Lite over D* Lite 3.2. There are two main reasons for this. The first is, because it is sufficient to seek sub-optimal paths, there are more alternatives to choose from. In order to find an alternative path for P, IID* Lite can only choose a path whose cost is the same as $|P| = \Omega$: but this is not required by IAD* and IAD* can choose any path of less than or equal to $\epsilon * \Omega$. The second reason is that, when changes are observed, IAD* can avoid recalculation by choosing a particular alternative path and therefore can skip reordering the priority queue OPEN, whereas AD* must do the reordering when recalculation is needed. In [20] it is claimed that the method of [31] to avoid reordering of

the priority queue can be used in AD*. Unfortunately, since the heuristic used in AD* is not consistent, the proposed modification causes additional reinsertions of key values of vertices into the priority queue resulting in more heap operations. This added cost is verified by experiments so in this thesis the reordering is not avoided.

Figures 37 and 38 show the performance of AD* to IAD* with $percent = 20$ to demonstrate extent to which IAD* can perform well in dramatically changing environments. From the figures, it can be seen that IAD* still achieves up to an order of magnitude speed up over AD*. In addition, IAD* shows a lower number of heap operations than IAD* which largely accounts for the speeding up. Notice also that the smaller the sub-optimality is, the better IAD* performs relative to AD*.

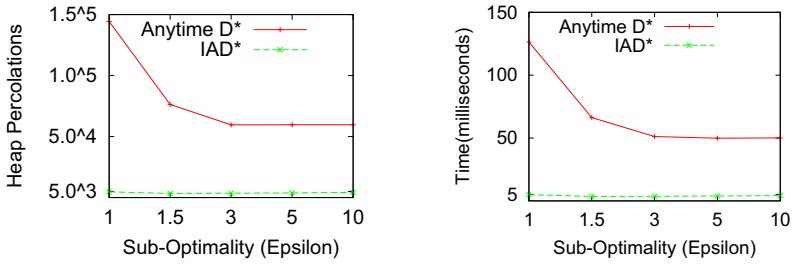


Figure 37: size=300, percent=20, sensor-radius=30(rock-and-garden)

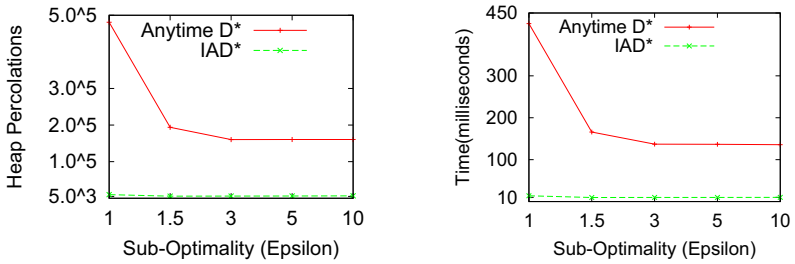


Figure 38: size=500, percent=20, sensor-radius=50(rock-and-garden)

Figure 39 shows performance comparisons for various values of *sensor-radius* with $size = 300, percent = 3$ and $\epsilon = 3$. Both AD* and IAD*

are shown to be affected negligibly by the change of sensor-radius. This is different from what has been observed for D* Lite and IID* Lite and is due to the relaxation to sub-optimality.

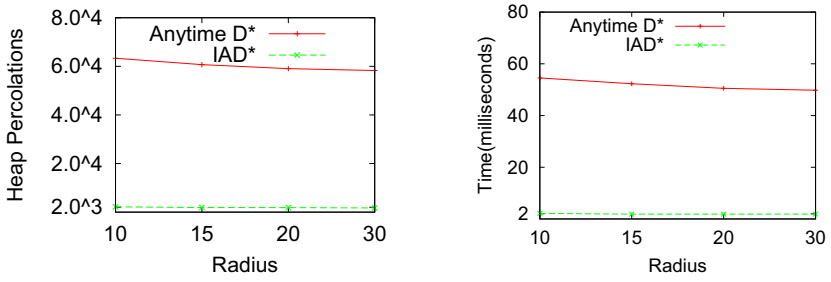


Figure 39: size=300, percent=10, ϵ=3(rock-and-garden)

Figure 40 shows performance comparisons for various values of *percent* with $size = 300, sensor - radius = 30$ and $\epsilon = 3$. Observe that increasing percent causes the number of heap operations and time of both algorithms to increase but slowly due to relaxation to sub-optimality.

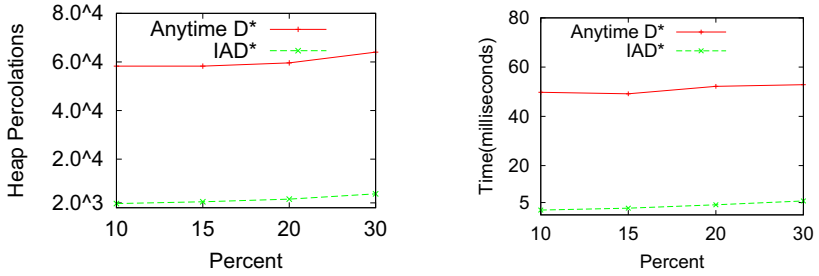


Figure 40: size=300, sensor-radius=30, ϵ=30(rock-and-garden)

Figure 41 shows performance comparisons for various values of *size* with $percent = 10, sensor - radius = 30$ and $\epsilon = 3$. Observe that IAD* appears to be more scalable than AD*.

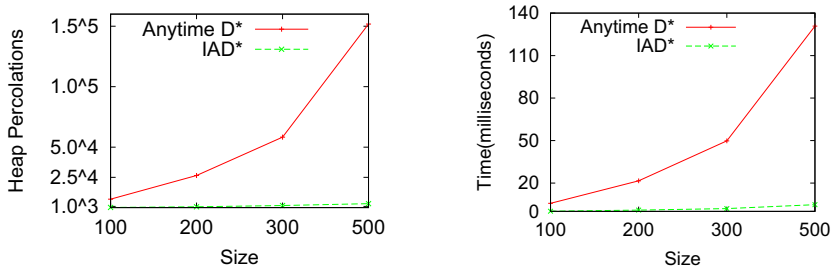


Figure 41: percent=10, sensor-radius=30, ϵ=3(rock-and-garden)

The set of parking-lot benchmarks is intended to model agent navigation in the presence of terrain changes. A number of tokens equal to a given fixed percentage of vertices are initially created and distributed in the grid with at most one token covering any vertex. As an agent moves from one vertex to another through the grid, tokens may move to adjacent vertices as well. Tokens are never removed from the grid and rules for moving tokens are fixed for every round: a token to some adjacent vertex 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, an order in which tokens are considered for being moved guarantees that no two tokens will move to the same vertex. Whenever a vertex is covered by a token in the simulation it is considered blocked and any edge cost to this vertex equals ∞. A vertex with no token on it is considered unblocked and an incident edge cost to such a vertex is a finite, positive number.

Figures 42 to 45 compare the performance of IAD* and AD* with parameter values that match the performance experiments of Figures 35 to 38. Observe that IAD* also is up to an order of magnitude faster than AD* although the performance difference between the two is smaller than for the rock-and-garden benchmarks. The reason is that for parking-lot benchmarks, IAD* needs to call **MiniCompute**() to update overconsistent vertices, and hence needs to reorder priority queue more times. This mirrors the comparison between IID* Lite with D* Lite in 3.2. Since time plots are similar to heap percolation plots, only heap percolation plots are shown below.

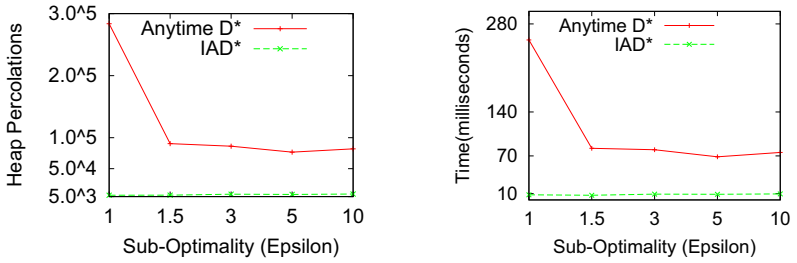


Figure 42: size=300, percent=10, sensor-radius=30(Parking-lot)

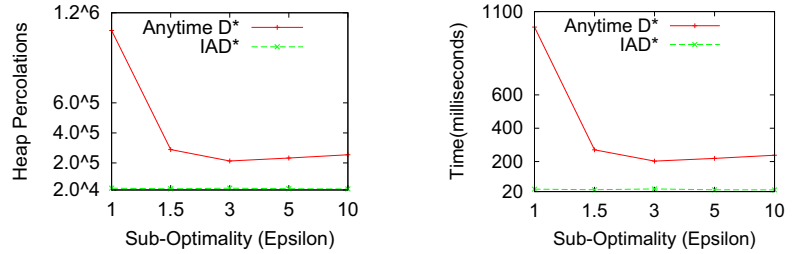


Figure 43: size=500, percent=10, sensor-radius=50(Parking-lot)

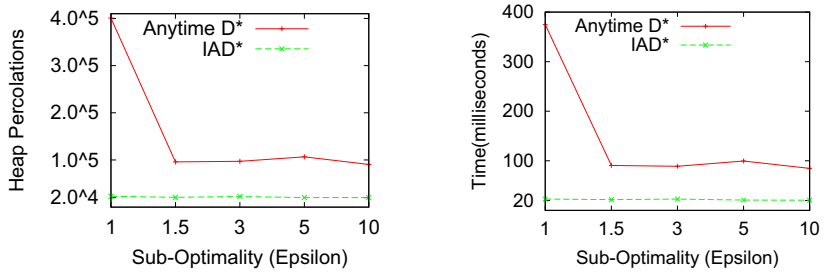


Figure 44: size=300, percent=20, sensor-radius=30(Parking-lot)

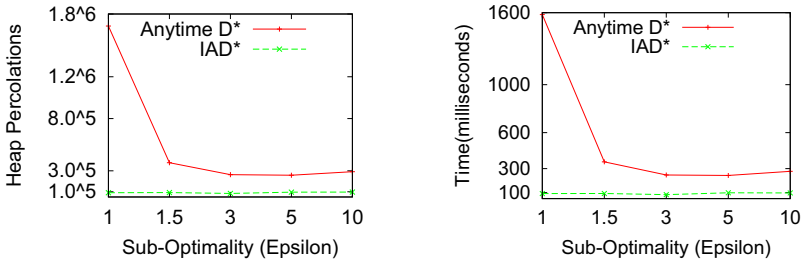


Figure 45: size=500, percent=20, sensor-radius=50(Parking-lot)

Figure 46 compares performance for various values of *sensor-radius* with $size = 300, percent = 3$ and $\epsilon = 3$. Observe that heap operations and time increase faster than they do with the rock-and-garden benchmarks. The reason is that with the parking-lot benchmarks the blockages are kept moving and therefore more changes are observed from a larger sensor-radius. This results in more updates.

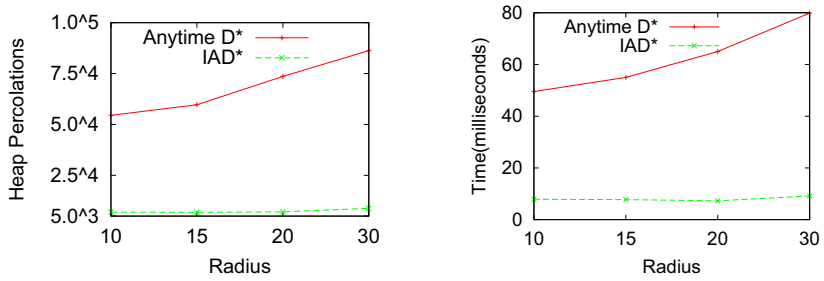


Figure 46: size=300, percent=10, ϵ=3(Parking-lot)

Figure 47 compares performance for various values of *percent* with *percent* with $size = 300, sensor - radius = 30$ and $\epsilon = 3$. Observe that the number of heap operations and time of IAD* increases faster than AD* as *percent* is increased. The reason is that as *percent* is increased, more updates are needed in IAD*, which affects its performance. Hence, in parking-lot environments, IAD* performs best relAtive to AD* when the changes in terrain over time are relatively slight.

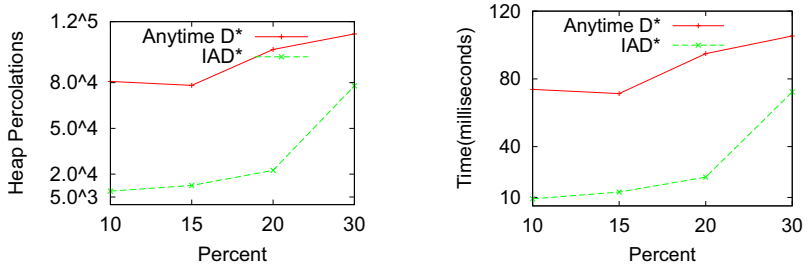


Figure 47: size=300, sensor-radius=30, ϵ=3(Parking-lot)

From the figure, it can also be observed that, even with $percent = 30$, IAD* is still about twice as fast as AD*. The difference between rock-and-garden and parking-lot benchmarks in this comparison, also comes from the difference of whether moving of observed blockages is allowed.

Finally, Figure 41 compares performance for various values of *size* with $percent = 10, sensor - radius = 30$ and $\epsilon = 3$. The results show that IAD* is scalable in parking-lot benchmarks as it was for rock-and-garden benchmarks.

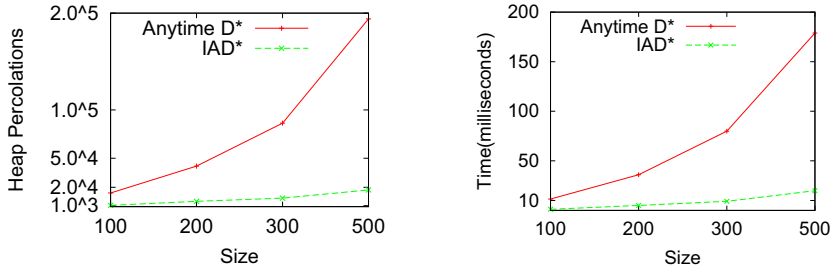


Figure 48: percent=10, sensor-radius=30, ϵ=3(Parking-lot)

4.1.3 Conclusion and Next Step of Work

In this section, a new dynamic anytime algorithm IAD* was introduced and analyzed experimentally. IAD* improves upon AD* by using a strategy similar to that used in IID* Lite. That is, whenever possible, IAD* tries to find an alternative path instead of recalculating immediately to find one as is done in AD*. Moreover, when an alternative path is not available, in order to avoid a full recalculation, IAD* will try to propagate changes incrementally with the help of a changing threshold until a new sub-optimal path is found. Experimental results show that under various random benchmarks IAD* can typically achieve an order of magnitude speed up over AD* when time is measured by the first criterion of comparing anytime algorithms, namely time to find the first qualified sub-optimal path. The comparison is based both on heap operations and time consumed on different terrains with the same preset sub-optimality ϵ.

As introduced in the beginning of Section 4, the other criterion for comparing dynamic anytime algorithms is the sub-optimality of the final path returned in every round of recalculation. Figure 35 to 38 and Figure 42 to 45

show that IAD* also does well on this metric relative to AD*. From these figures it can be seen that when ϵ is greater than 1.5 with other parameters unchanged, IAD* has roughly the same performance advantage over AD* and when ϵ is less than 1.5 IAD*'s advantage is much greater. So, when ϵ is set to a small number, IAD* can still be guaranteed to return a high-cost sub-optimal path with a high probability within acceptable time bounds. Furthermore, when changes are observed, IAD* can find the first sub-optimal path in less time; after that, IAD* can continue to improve the current path until time runs out. In other words, with ϵ less than 1.5 IAD* can guarantee a desired sub-optimal path in a shorter time with a higher possibility, which means more time can be used to improve the path that is returned first. Hence, it can be expected that IAD* can return the first sub-optimal path faster than AD* for various random benchmarks, and IAD* has greater potential to return high-cost sub-optimal paths within specified time constraints.

The next section considers how IAD* and AD* behave in each round with respect to the sub-optimality of their final returned paths when resources are limited: for example, when an upper bound on allowed heap operations is enforced. In addition, failure-ratios, representing the percentage of times an algorithm can find a sub-optimal solution within resource limits, are studied. Since Anytime Weighted A* (AWA*) [10] is claimed to have better performance than other algorithms in achieving higher-cost sub-optimal paths on some benchmarks, a new dynamic variant of AWA*, called DAWA*, is introduced as a combination of algorithms IAD* with AWA*. The new algorithm is compared against IAD* and AD*.

4.2 Dynamic Anytime Weight A* Algorithm

As stated, there are two criteria for comparing anytime algorithms: the time used to compute the first qualified sub-optimal path when updating, and the sub-optimality of the final path that is output after recalculation. A good anytime algorithm tries to find the first sub-optimal path as soon as possible, and then iteratively improves this path until time runs out. In Section 4.1.1, IAD* was proposed and compared with AD* on the first criterion. Experimental results show that IAD* is an order of magnitude faster than AD*. In this section, the same comparison is considered but when resources are limited: the limited resource in this case is heap operations since time complexity in navigation algorithms is directly related to heap operations.

As discussed in Section 4.1, IAD* has greater potential for returning high-cost sub-optimal paths. But it should be mentioned that in each round of recalculation, with the sub-optimality parameter ϵ decreased, both IAD* and AD* need to reorder the priority queue OPEN and perform extra calculations. Doing so hinders finding higher-cost sub-optimality paths when there is a bound on the number of heap operations. So, if the above extra calculations can be avoided, the chance of achieving a better sub-optimal path is increased.

In this section, IAD* with AWA* are combined to get a new algorithm called Dynamic Anytime Weight A* (DAWA*). DAWA* uses IAD*'s dynamic skeleton and is merged with AWA* in each round of recalculation. In [10] it is claimed that for some benchmarks Anytime Weighted A* (AWA*) achieves a higher-cost sub-optimal path than other algorithms including ARA* [21] the dynamic version of which is AD*. ARA* limits the updating times associated with expanding each vertex and therefore has the ability to control provable bounds of sub-optimality that may be used to speed up search. However, this strategy may defer propagations of some "good" vertices which would lead to a qualified path. Indeed, the discussion in [10] argues that the strategy used in ARA* may increase or decrease search efficiency and the result is totally depended on the benchmarks. Since IAD* has inherited ARA*'s strategy of recalculation in a round from AD*, it can be expected that both DAWA* and IAD* perform differently relative to each other on different benchmark environments.

When changes are observed, dynamic anytime algorithms (DAA) need to recalculate a new path from v_c to v_g. Because time available to do so is limited, the requirement of finding an optimal path is relaxed in favor of

finding a reasonable sub-optimal path. Since some path *must* be returned, these algorithms use a two step strategy: first, they quickly find some feasible, although generally not optimal, path and then for the second step they use the remaining available time to improve upon the path of the first step and return a path that is no further from optimal than that of the first step.

A real-numbered parameter ϵ, which is used in the calculation of f, facilitates both steps. As the value of the parameter decreases, a feasible computed path has a cost that is closer to the optimal but the time needed to compute it increases. Hence, one natural approach to finding a better path is to decrease the ϵ by some predetermined amount, then recalculate for a new path. If a replacement path is found, it is guaranteed to be ϵ sub-optimal. However, the price to pay for this is the reordering overhead. Because vertices that are in the priority queue are sorted by $f = g + \epsilon * h$, if the ϵ is changed the priority queue must be reordered. Since reordering the priority queue is work done *before* calculating a new path, it is possible that time will run out while calculating the new path and the work done to reorder the priority queue will be wasted.

Suppose a change in ϵ necessitates x priority queue operations to reorder it and suppose y priority queue operations are needed to compute an improved path. For convenience, ϵ is decreased to be $\epsilon' < \epsilon$. State available time in terms of an allowed number of priority queue operations and denote that number by *operation-limit*. Then, the following outcomes are possible on an iteration:

1. *operation-limit* $<= x$: last computed path is returned, the *operation-limit* heap operations are wasted.

2. $x <$ *operation-limit* $< x + y$: last computed ϵ sub-optimal path is returned, work done is *operation-limit* $- x$;

3. *operation-limit* $>= x+y$: a new ϵ' sub-optimal path is returned, work done y heap operations. *operation-limit* $- x - y$ heap operations are still available which can be used to further improve current solution.

In this thesis an alternative to the above strategy is used. The new strategy [10] does not wait for priority queue reordering to complete before commencing to compute a replacement path. "The approach we adopt uses weighted heuristic search to find an approximate solution quickly, and then continues the weighted search to find improved solutions as well as to improve

a bound on the suboptimality of the current solution" [10]. In this strategy, ϵ is not changed, hence no reordering of priority queue is needed. That is, as long as the parameter $operation - limit$ is non-zero, priority queue values are updated. when we updating current vertices, and the updated vertices are reinserted and so reordered. So we can say that, this method is just keep searching exactly the same before and after finding a sub-optimal solution.

The benefit in this case is to eliminate the waste of point 1. above. However, unlike the approach described above, ϵ remains unchanged. Of course, paths returned by the proposed approach must be ϵ sub-optimal. But the paths returned by the approach above are ϵ' sub-optimal where $\epsilon' < \epsilon$ so there is no guarantee that the proposed approach returns paths that are no worse than those returned by the above approach. However, empirical results show that it is often the case that a better solution is returned by the proposed approach.

DAWA* algorithm works the same as IAD* on finding the first solution. But after the first solution found, they act differently: IAD* uses the first strategy introduced above which is inherited from AD*; DAWA* algorithm uses the second strategy coming from AWA* algorithm. At the conclusion of this section DAWA* and IAD* will be compared experimentally using several different navigation benchmarks.

4.2.1 Pseudo Code of DAWA* Algorithm

This section begins with the motivation for and an overview of a new algorithm called Dynamic Anytime Weight A* (DAWA*). Pseudo code for the main functions of DAWA* is shown in Figure 49. Then the performance of DAWA* is compared with that of IAD* and again compared with the performance of AD* on random benchmarks.

IAD* inherits from AD* the control ability of sub-optimality. Both algorithms update an inconsistent vertex which is specially stored in INCONS at most one time in each round of recalculation and the solution returned by both is guaranteed to be ϵ sub-optimal [20]. By contrast, AWA* may update a vertex many times until time runs out or OPEN is empty which means an optimal path has been found [10]. Remarkably, despite the disparity between both algorithms, IAD* and AWA* can be combined to create the new dynamic algorithm DAWA*.

DAWA* works as follows. When changes are observed, DAWA* performs the same action as IAD* to return the first path. During further iterative

improvements to this path DAWA* performs the action of AWA*. Time is saved with this strategy as discussed in the beginning of this Section.

Procedure UpdateVertex(s):
01. if s has not been visited
02. $\quad g(s) = \infty$;
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. $\quad$ OPEN.insert($[s, \mathbf{key}(s)]$);$\mathbf{type}(v) = 0$;

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

Figure 49: Main functions of DAWA* Algorithm

Most of the functions of DAWA* are the same as those of IAD*, hence only the modified functions are shown in Figure 49. A description of these function follows referring to Figure 49. In IAD* INCONS is used to indicate the updated vertices in a round of recalculation. As the updated times of vertices in any round of recalculation are not needed in AWA*, these are not needed in DAWA* as well. In Function **UpdateVertex** at lines 05-06, if the vertex updated is inconsistent, it is reinserted into OPEN directly. In Function **Main** from lines 12 to 16, whenever time allows the current path is iteratively replaced by a higher-cost path; the sub-optimality parameter ϵ

is not manually set but is calculated at line 14 according to [10]. When ϵ equals 1 an optimal path is found and the algorithm stops calculating and waits for new edge cost changes.

Theorem 4.2 *In the Dynamic Anytime Weight A*(DAWA*) algorithm, when there is no time limitation, every round of recalculation always terminates and the final solution returned is optimal.*

Proof: Follows from the correctness of IID* Lite and AWA*.

4.2.2 Experiments and Analysis

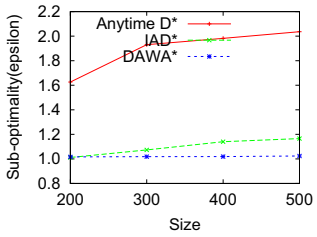
In this section, the performance of DAWA* is experimentally compared with that of AD* and IAD* on random grid world terrains. The same benchmarks as Section 4.1.2 are used. In each experiment the terrain is a square, 8-direction grid world of $size^2$ vertices. Vertices v_s and v_g are chosen randomly from the terrain. Initially $percent\%*size^2$ of the vertices are randomly selected as blocked points, where *percent* is a controlled parameter. The parameter $sensor-radius$ is used to set the maximum distance the agent can "see" from the current position. Consistent heuristic function similar with Manhattan distance is used. To clearly picture the comparison of the optimizing courses of the above different anytime algorithms, we prefer the first path in calculation is a low sub-optimal solution. For this goal, we choose not to use the best heuristic function and multiply the returned Manhattan distance heuristic value with a parameter 0.5. Before navigation, the agent has an initial map in which an obstacle may be wrongly judged as blank with fifty percent probability.

As mentioned above, the resource being limited is the number of heap operations. This is reasonable since heap operations consume most of the time in navigation algorithms. The second criterion, namely the sub-optimality of the path that is finally returned, is compared. Whenever changes are observed, ϵ is set to the parameter $epsilon-limit$ the default value of which is 3. Then, as long as there are still heap operations available, ϵ is decreased to $max(1, \epsilon-1)$. The smaller the returned ϵ, the better the algorithm performs and when (if) ϵ equals 1, the path returned is optimal. If no qualified path is found before the limit of available number of heap operations is reached, the algorithm fails. Hence a statistical analysis of failure percentage is also presented below. But, for the sake of obtaining meaningful results, a first

qualified path will always be returned even if the number of heap operations is exceeded (but then no further action is taken by the algorithm to improve upon the path).

The first results compare DAWA* against AD* and IAD* on the rock-and-garden benchmarks. Figure 50 shows the final value of ϵ reached, within the fixed operation-limit bound on the number of heap operations as a function of terrain *size* fixed and $sensor - radius$ equal to 30. As size increases, the average total cost of the returned path is greater, and therefore the priority queue OPEN gets larger. A larger priority queue requires more heap operations and that reduces the amount of resource available for iterating toward optimal paths. Thus, as seen in the figure, ϵ rises as size increases. But it rises least for DAWA*.

The table in Figure 50 shows the failure-ratios of these algorithms. Here a failure is counted when the algorithm cannot return a qualified sub-optimal solution within the specified limited number of heap operations. Observe that DAWA* also has the lowest failure ratio of the three algorithms. In addition, AD*'s failure-ratio increases faster with size than the others. The reason that DAWA*'s failure rate is consistently low is that whenever heap operations are allowed, OPEN is updated and thus becomes smaller. Thus, the failure-ratio is not so much affected by the cost of a returned path as by the size of the grid. Moreover, whenever there are still heap operations allowed, DAWA* can utilize them without wasting heap operations on reordering priority queue.

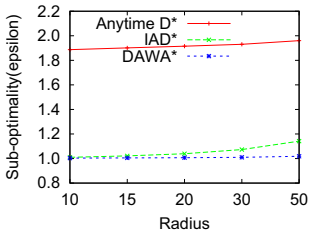


Failure-ratios with different sizes

	200	300	400	500
AD*	0.27025	0.43230	0.63234	0.75925
IAD*	0.01642	0.02153	0.01478	0.01543
DAWA*	0.00817	0.00975	0.01058	0.01415

Figure 50: operation-limit=500, percent=10, sensor-radius=30

Figure 51 shows the results of experiments where $sensor-radius$ is varied and other parameters remain fixed. When $sensor-radius$ is increased, more changes are observed in every round which leads to an increase in the number of recalculations for updating. As shown in the figure, the final ϵ increases and so does the failure-ratio.

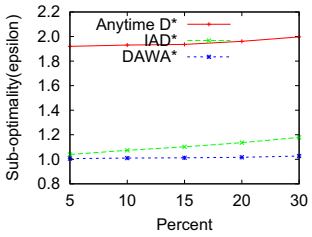


Failure-ratios with different radiuses

	10	15	20	30	50
AD*	0.37808	0.41586	0.43782	0.43230	0.41406
IAD*	0.01022	0.01492	0.01764	0.02153	0.01975
DAWA*	0.01129	0.01170	0.01194	0.00975	0.00646

Figure 51: size=300, operation-limit=500, percent=10

Figure 52 shows the results of experiments where percent is varied and other parameters remain fixed. The results are similar to those in Figure 51. The results suggests IAD* and DAWA* act similarly, but on both sub-optimality and failure-ratio, DAWA* is better.



Failure-ratios with different percentages

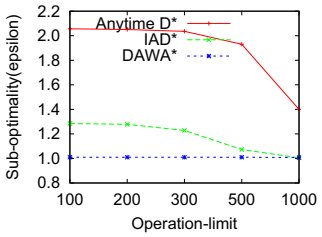
	5	10	15	20	30
AD*	0.34118	0.43230	0.50655	0.55216	0.60509
IAD*	0.00903	0.02153	0.03327	0.04574	0.08112
DAWA*	0.00545	0.00975	0.01380	0.01945	0.04184

Figure 52: size=300, operation-limit=500, sensor-radius=30

Figures 53 to 56 show the results of experiments where operation-limits is varied and all other parameters remain fixed. Of course, all algorithms perform better when their operation-limits are loosened. Specifically, when the limit is loosened, IAD* performs as well as DAWA*. But, when the limit is strict, DAWA* performs better than the other two algorithms.

As said previously, in order to enable comparison of the second criterion on different anytime algorithms under the imposition of resource limits, a first qualified path will always be returned even if the number of heap operations is exceeded (but then no further action is taken by the algorithm to improve upon the path). This case means failure in finding a solution earns extra calculations. The relation between failure-ratio and sub-optimality is not stable. More calculation may help lead to better sub-optimality. However, it is possible that the sub-optimality of the final solution generated by the extra calculations is worse than the average sub-optimality.

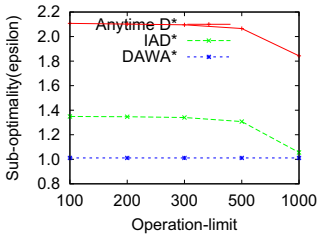
Figures 53 and Figure 54 compare returned path costs of IAD*, AD*, and DAWA* when loosening the heap operation limit. The two figures only differ in the size of terrain under which experiments are run. Both figures show IAD* outperforming AD* in path cost and failure rate and DAWA* similarly outperforming the other two.



Failure-ratios with different operation-limits

	100	200	300	500	1000
AD*	0.94849	0.88943	0.74251	0.43230	0.21764
IAD*	0.00984	0.00980	0.01128	0.02153	0.00498
DAWA*	0.00984	0.00980	0.00980	0.00975	0.00411

Figure 53: size=300, percent=10, sensor-radius=30



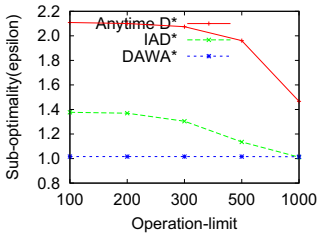
Failure-ratios with different operation-limits

	100	200	300	500	1000
AD*	0.94994	0.92084	0.86744	0.71449	0.38496
IAD*	0.00928	0.00928	0.01030	0.01121	0.02938
DAWA*	0.00945	0.00939	0.00937	0.00937	0.00925

Figure 54: size=500, percent=10, sensor-radius=50

Consider the results for IAD*. In the failure table of Figure 53, when the heap operation limit is increased from 100 to 300, sub-optimality and failure ratio of IAD* does not change much. When the heap operation limit is increased from 300 to 500, sub-optimality improves while failure ratio gets worse. When the operation limit is increased from 500 to 1000, both sub-optimality and failure ratio improve which means, in this interval, that IAD* works better overall. This non-monotonic improvement defies our intuition that increasing the operation limit should lead to better and smaller sub-optimality and failure-ratio. The observed phenomenon can be explained as follows. In the course of IAD*'s execution, each time ϵ becomes smaller, some heap operations are required to reorder the priority queue and recalculate to improve ϵ. Figure 54 shows experimental results when more changes occur due to an increase in terrain size. In Figures 53 and 54, since AD* also needs to reorder the priority queue whenever recalculation is needed as IAD* does, the improvement of AD* is also non-monotonic. However, since DAWA* improves its path with a more incremental strategy as discussed at the beginning of this Section, DAWA* has a much better ability to utilize every allowed heap operation and hence its results improve monotonically with increasing operation limit.

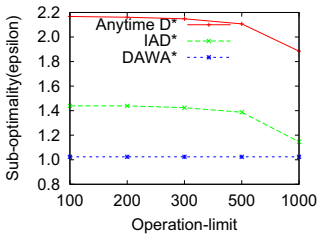
Figures 55 and 56 display similar phenomenon as in Figures 53 and 54. In these results percent is increased to 20.



Failure-ratios with different operation-limits

	100	200	300	500	1000
AD*	0.94245	0.86249	0.75220	0.55216	0.47795
IAD*	0.02006	0.01998	0.02363	0.04574	0.02355
DAWA*	0.01993	0.01989	0.01976	0.01945	0.01649

Figure 55: size=300, percent=20, sensor-radius=30

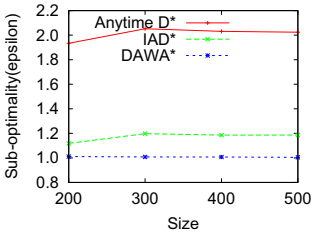


Failure-ratios with different operation-limits

	100	200	300	500	1000
AD*	0.94195	0.89426	0.84478	0.73129	0.53048
IAD*	0.02020	0.02017	0.02136	0.02495	0.07037
DAWA*	0.02022	0.02020	0.02014	0.02012	0.01985

Figure 56: size=500, percent=20, sensor-radius=50

Next the performance of AD*, IAD* and DAWA* are compared on parking-lot benchmarks. Since in parking-lot benchmarks all blocks can move randomly, more changes are typically observed than for rock-and-garden benchmarks. Therefore all algorithms perform worse on these benchmarks and generate greater sub-optimality and failure ratios. Compared with Figure 50, the sub-optimality graph in Figure 57, is more steady. Observe that the failure ratios of DAWA* and IAD*, but especially of DAWA*, improve with increasing size. The reason is probably because they call function mini-compute often and that tends to reduce the size of priority queue OPEN and therefore saves heap operations.

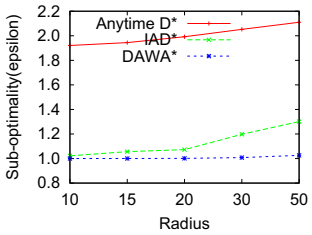


Failure-ratios with different sizes

	200	300	400	500
AD*	0.84584	0.86319	0.91814	0.93975
IAD*	0.15893	0.20370	0.20134	0.18694
DAWA*	0.02556	0.02217	0.02118	0.01740

Figure 57: operation-limit=500, percent=10, sensor-radius=30

Figure 58 shows the results of experiments of the three algorithms running on benchmarks with variable sensor-radius and all other parameters fixed. As

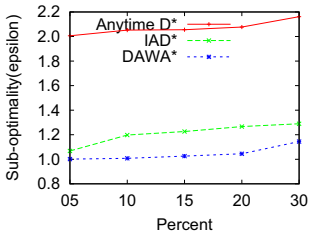


Failure-ratios with different radiuses

	10	15	20	30	50
AD*	0.80437	0.86843	0.87941	0.86319	0.87998
IAD*	0.04945	0.09061	0.10022	0.20370	0.26081
DAWA*	0.00223	0.00267	0.00630	0.02217	0.05092

Figure 58: size=300, operation-limit=500, percent=10

a property of parking-lot benchmarks, changes increase quickly with increasing $sensor - radius$. Hence sub-optimality and failure ratio both get worse quickly. Since changes that can be observed increase faster with increasing percentage than with increasing sensor-radius, the results in Figure 59 get worse faster.

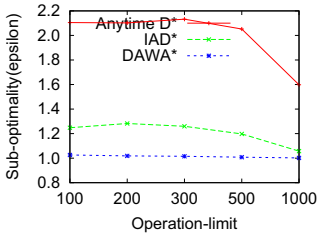


Failure-ratios with different percentages

	5	10	15	20	30
AD*	0.82650	0.86319	0.88474	0.91120	0.98197
IAD*	0.06180	0.20370	0.33987	0.47737	0.86444
DAWA*	0.00530	0.02217	0.07001	0.13457	0.59148

Figure 59: size=300, operation-limit=500, sensor-radius=30

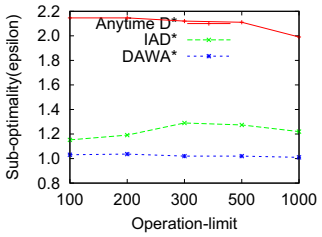
Figures 60 to 63 show the behavior of the algorithms with variable heap operation-limit but with other parameters fixed. In contrast to rock-and-garden benchmarks, all algorithms suffer from many changes in each round. Again non-monotonic relations are observed among the operation-limit, sub-optimality and failure ratios, especially for the IAD* algorithm where it may be said that IAD* appears to be more unpredictable.



Failure-ratios with different operation-limits

	100	200	300	500	1000
AD*	0.98498	0.95876	0.93886	0.86319	0.86165
IAD*	0.21792	0.18969	0.18532	0.20370	0.08261
DAWA*	0.08271	0.06224	0.03887	0.02217	0.00423

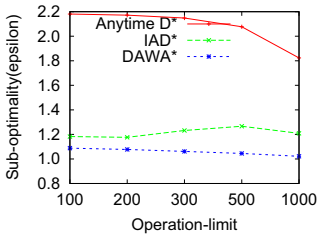
Figure 60: size=300, percent=10, sensor-radius=30(Parking-lot)



Failure-ratios with different operation-limits

	100	200	300	500	1000
AD*	0.98602	0.97837	0.97316	0.93094	0.88842
IAD*	0.33291	0.32511	0.27630	0.28178	0.30748
DAWA*	0.09540	0.10018	0.06219	0.05327	0.02645

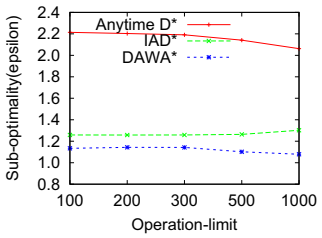
Figure 61: size=500, percent=10, sensor-radius=50(Parking-lot)



Failure-ratios with different operation-limits

	100	200	300	500	1000
AD*	0.99130	0.97285	0.94657	0.91120	0.85644
IAD*	0.45975	0.45836	0.42951	0.47737	0.38599
DAWA*	0.27607	0.21523	0.18681	0.13457	0.05813

Figure 62: size=300, percent=20, sensor-radius=30(Parking-lot)



Failure-ratios with different operation-limits

	100	200	300	500	1000
AD*	0.99606	0.98054	0.97482	0.95486	0.92107
IAD*	0.57145	0.55654	0.54457	0.55320	0.56300
DAWA*	0.38906	0.33167	0.32451	0.23199	0.19693

Figure 63: size=500, percent=20, sensor-radius=50(Parking-lot)

4.2.3 Conclusion and Next Step of Work

In this Section, we get a new algorithm DAWA* algorithm by combining IAD* with AWA* algorithm. When the available heap operations being limited, DAWA* performs much better than IAD* and AD* algorithms: not only higher-cost sub-optimal solution found, but also lower failure-ratios. In this Section, all experiments are done with preset heap operation limit. But in some environments, the limit of heap operation limit is changed and unpredicted. So in the future, we will do experiments to compare all the dynamic anytime algorithms under such experiments. Because DAWA* can utilize heap operations more incrementally, we expect DAWA* works the best among those algorithms.

4.3 Conclusion

A new Dynamic Anytime Algorithm IAD* was introduced. IAD* improves the well-known AD* by using a path-search strategy that is similar to that used by IID* Lite. Whenever possible, IAD* tries to find an alternative path instead of recalculating immediately as is done by AD*. If an alternative is not available, in order to avoid a full recalculation, IAD* tries to propagate changes incrementally with the help of an increasing threshold until a new, but sub-optimal, path is found.

A second Dynamic Anytime Algorithm DAWA* was introduced to handle cases where resource limits, particularly limits on heap operations, are imposed. DAWA* combines the advantages of AWA* and IAD*. DAWA* can be regarded to be a descendent of IAD* and AWA* as it uses the same path-search strategy as IAD* on finding the first sub-optimal solution, and the same path-search strategy as AWA* otherwise.

To show the performance of these new algorithms, experiments were run and results compared for different random benchmarks on two criteria that were introduced at the beginning of this section. For the first criterion, from the results in Section 4.1, it was observed that IAD* can find the first qualified sub-optimal path in less time than AD*. As a descendant of IAD*, DAWA* performs like IAD* on this criterion. Section 4.2 presents the comparison on the second criterion. As expected, DAWA* performs best and returns the smallest sub-optimality as well as the lowest failure-ratio. IAD* performs worse than DAWA* but better than AD* algorithm. The non-monotonic performance of IAD* was observed and the reason identified as due to that IAD* needs to use extra heap operations on reordering priority queue when recalculations needed.

From above it can be concluded that IAD* and DAWA* perform better than AD* on various random benchmarks. IAD* inherits from AD*, the ability to control sub-optimality. But DAWA* gains better performance by utilizing heap operations more incrementally and sacrificing the control of sub-optimality. Finally, the merging of techniques to arrive at the new algorithms is general and may be applied to other algorithms. Indeed, it is claimed that techniques introduced in [2, 34, 36] can be combined with our algorithms. In future work such combinations will be tried.

5 Conclusion

With the development and growing demand of artificial intelligence techniques, reliable and efficient dynamic navigation algorithms have become an indispensable tool in many important applications, for instance, robot navigation, GPS system, and AI design in modern computer games. The D* Lite algorithm is a most commonly used dynamic navigation algorithm and many variants are derived based on it to fit different environments. The key to the success of these algorithms is the use of incremental methods. Incremental algorithms have been researched for decades and applied in numerous areas. Generally speaking, using information produced by previous rounds to speed up the current round is an art, not a science. Naturally, research is focused on how to use such information better.

In Section 3, new methods to utilize previous information to improve D* Lite was introduced and two improvements to the D* family of algorithms were proposed. One improvement attempts to find an alternative least cost path before recalculating values when the current least cost path becomes inconsistent. The second improvement applies a general and systematic view on considering edge cost changes to find a least cost path before all changes are propagated when recalculation is necessary. The first improvement lead to a new algorithm named ID* Lite. Combination of the above two improvements lead to another new algorithm called IID* Lite. Both algorithms can guarantee an optimal solution in every round if one exists [40,41,43]. Analytically, ID* Lite produces optimal solutions at least as efficiently as D* Lite. Experimental results show both algorithms outperform other algorithms in the D* family.

Experiments were performed on a variety of benchmarks including rock-and-garden and dynamic navigation families of terrain. The results show IID* Lite can get up to 8 times speedup over D* Lite and ID* Lite 2 times speedup over D* Lite. It was also shown that IID* Lite is less sensitive than D* Lite to the number of changes observed in a round. The importance of this is expected to increase with advancing technology that will support finer grain simulations and allow robots to "see" further. Since the proposed improvements put no extra constraints on D* Lite, it is conjectured that they can be used to improve many variants of D* Lite as well. For example, in Section 4.1 it was applied to Anytime D* which is the first anytime dynamic navigation algorithm.

The strategy used in IID* Lite to improve upon Anytime D* inspired

a new algorithm, called Improved Anytime D* (IAD*) [42], which was presented in Section 4. Two criteria to compare the performance of dynamic anytime algorithms were stated. The first is the time an algorithm uses to compute the first qualified sub-optimal path in the course of recalculation, and the second is the sub-optimality of the final path that is output after recalculation. From experimental results, comparing against the well-known AD* algorithm, around an order of magnitude speed up is achieved by IAD* for various random benchmarks with respect to the first criterion. IAD* is designed to support the control of sub-optimality in a manner similar to AD*. This property allowed IAD* to be combined with AWA* to create a new algorithm, called DAWA*, which uses time resource incrementally by sacrificing the control of sub-optimality. As expected, DAWA* has the best performance of all algorithms tested with respect to the second criterion. Due to the modular design of IAD* and DAWA* and the properties stated above, the techniques that are the foundation for IAD* and DAWA* can be merged with other anytime algorithm variants [2,34,36] to create even more variants which may show even greater performance gains.

Incremental algorithms have been applied to numerous problems other than dynamic navigation. A property that makes these algorithms useful is that they can change strategies for reusing information in the face of changing domain knowledge. The incremental algorithms presented in this thesis do not depend on any domain knowledge so they can be merged with any other incremental algorithm that might be tailored to a specific navigation application or even other applications which are best solved with incremental algorithms, for example, when there is more than one optimal solution.

All algorithms introduced in this paper are intended to find a lowest cost path that an agent can take from a source vertex to a goal vertex in a dynamically changing terrain that is modeled as a weighted directed graph, usually a grid. At any vertex in the graph the agent is presented with a view of a relative small neighborhood of vertices. Information about vertices in the neighborhood is available to the agent but no information is available from outside the neighborhood. Exploring the entire graph to get complete information is out of the question, especially since the state of the vertices can change with time. This raises the important question of how best to plan a path from source to goal: the objective is to reach the goal covering a path of lowest or near lowest cost, using the least or at least acceptable computational resources (such as heap operations).

This problem is related to other problems that are special forms of dy-

namic navigation problems. For example, a variant of the traveling salesman problem is to find a low cost Hamiltonian cycle where the route map, modeled as an undirected weighted graph, over which a salesman is to travel is revealed in pieces, as the salesman reaches previously unvisited cities. Then, vertices become blocked when the corresponding cities are reached. A real-world embodiment of this problem might be to help an automated lawnmower navigate optimally through a park.

Another obvious variant is positioning in the temporary absence of GPS and WiFi signals, such as in a tunnel or other metal frame structure, as long as a map exists locally. In fact, such a map could be improved as the person or vehicle moves through points where GPS is available. So-called indoor positioning has vast market demand. For example, it helps disabled people to plan and navigate routes conveniently. The incremental technology presented in this thesis coupled with advances in ad-hoc networks could be quite useful in developing accurate maps for navigation systems - the data of many "agents" with a very limited view can be incrementally combined to develop a complete or at least fairly complete, accurate map that can be shared by all.

6 References

[1] S. koenig's homepage. Avaliable at `http://idm-lab.org/project-a.html`.

[2] S. Aine, P. P. Chakrabarti, and R. Kumar. *AWA* - A Window Constrained Anytime Heuristic Search Algorithm. Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 2250–2255, 2007.

[3] B. Bonet and H. Geffner. *Planning as heuristic search. Artificial Intelligence*, 129(1-2):5–33, 2001.

[4] P. P. Chakrabarti, S. Ghosh, and S. C. DeSarkar. *Admissibility of AO* when heuristics overestimate. Artificial Intelligence*, 34:97–113, 1988.

[5] H.W. Davis, A. Bramanti-Gregor, and J. Wang. *The advantages of using depth and breadth components in heuristic search. Methodologies for Intelligent Systems*, 3:19–28, 1988.

[6] T. L. Dean and M. Boddy. *An analysis of time-dependent planning. Proceedings of the National Conference on Artificial Intelligence*, 1988.

[7] R. Dechter and J. Pearl. *Generalized best-first search strategies and the optimality af A*. Journal of the Association for Computing Machinery*, 32(3):505C536, 1985.

[8] E. W. Dijkstra. *A note on two problems in connexion with graphs. Numerische Mathematik*, 1:269–271, 1959.

[9] D. Ferguson and A. Stentz. *The Delayed D* Algorithm for Efficient Path Replanning. IEEE International Conference on Robotics and Automation*, pages 968–975, 2005.

[10] E. A. Hansen and R. Zhou. *Anytime heuristic search. Journal of Artificial Intelligence Research*, 28:267–297, 2007.

[11] L. Harris. *The heuristic search under conditions of error. Artificial Intelligence*, 5(3):217–234, 1974.

[12] P. E. Hart, N. J. Nilsson, and B. Raphael. *A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems, Science, and Cybernetics*, 4(2):100–107, 1968.

[13] T. Ikeda and T. Imai. *Fast A* algorithms for multiple sequence alignment. Proceedings of Genome Informatics Workshop 94*, pages 90–99, 1994.

[14] S. Koenig and M. Likhachev. *D*lite. Eighteenth national conference on Artificial intelligence*, pages 476–483, 2002.

[15] S. Koenig and M. Likhachev. *Improved Fast Replanning for Robot Navigation in Unknown Terrain. IEEE International Conference on Robotics and Automation*, pages 968–975, 2002.

[16] S. Koenig and M. Likhachev. *Incremental A*. Advances in Neural Information Processing Systems*, pages 1539–1546, 2002.

[17] S. Koenig, M. Likhachev, and D. Furcy. *Lifelong Planning A*. Artificial Intelligence Journal*, 155(1-2):93–146, 2004.

[18] A. Koll and H. Kaindl. *A new approach to dynamic weighting. Proceedings of European Conference on Artificial Intelligence*, pages 16–17, 1992.

[19] R. Korf. *Linear-space best-first search. Artificial Intelligence*, 62(1):41–78, 1993.

[20] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun. *Anytime Dynamic A*: An Anytime, Replanning Algorithm. Proceedings of the International Conference on Automated Planning and Scheduling*, 2005.

[21] M. Likhachev, G. Gordon, and S. Thrun. *ARA*: anytime A* with provable bounds on sub-optimality. Advances in Neural Information Processing Systems*, 2003.

[22] M. Likhachev, G. Gordon, and S. Thrun. *ARA*: formal analysis*, 2003. Technical Report CMU-CS-03-148.

[23] D. Long and M. Fox. *The 3rd international planning competition: Results and analysis. Journal of Artificial Intelligence Research*, 20:1–59, 2003.

[24] G. Ayorkor Mills-Tettey, A. Stentz, and M. Bernardine Dias. *DD* Lite: Efficient Incremental Search with State Dominance. Proceedings of The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference*, pages 1032–1038, 2006.

[25] J. Pearl. Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley, 1984.

[26] J. Pearl and J. Kim. *Studies in semi-admissible heuristics. IEEE Transactions on Pattern Analysis and Machine Intelligence*, 4(4):392–399, 1982.

[27] J. Pearl and J. Kim. *Real-time heuristic search. Artificial Intelligence*, 42(2-3):197–221, 1990.

[28] I. Pohl. *First results on the effect of error in heuristic search. Machine Intelligence*, 5:219–236, 1970.

[29] I. Pohl. *Heuristic search viewed as path finding in a graph. Artificial Intelligence*, 1(3):193–204, 1970.

[30] I. Pohl. *The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem-solving. Proceedings of International Joint Conference on Artificial Intelligence*, pages 12–17, 1973.

[31] A. Stentz. *The Focussed D* Algorithm for Real-Time Replanning. Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1652–1659, 1995.

[32] A. Stentz. *Optimal and Efficient Path Planning for Partially-Known Environments. The Kluwer International Series in Engineering and Computer Science*, 388:203–220, 1997.

[33] J. T. Thayer. *Heuristic Search under Quality and Time Bounds. Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, pages 2854–2855, 2011.

[34] J. T. Thayer and W. Ruml. *Bounded Suboptimal Search: A Direct Approach Using Inadmissible Estimates. Proceedings of the 22nd International Joint Conference on Artificial Intelligence.*

[35] J. T. Thayer and W. Ruml. *Faster than weighted A*: An Optimal Approach to Bounded Suboptimal Search. Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling*, 2008.

[36] J. T. Thayer and W. Ruml. *Anytime Heuristic Search: Frameworks and Algorithms. Proceedings of the Third Annual Symposium on Combinatorial Search*, 2010.

[37] Jiuguang Wang. D* wiki, 2008.
Available at url: http://en.wikipedia.org/wiki/D*.

[38] Y. Xu and W. Yue. *A Generalized Framework for BDD-based Replanning A* Search. Proceedings of the 2009 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing*, pages 133–139, 2009.

[39] Y. Xu, W. Yue, and K. Su. *The BDD-Based Dynamic A* Algorithm for Real-Time Replanning. Proceedings of Frontiers in Algorithmics*, pages 271–282, 2009.

[40] W. Yue and J. Franco. *Avoiding Unnecessary Calculations in Robot Navigation. Proceedings of World Congress on Engineering and Computer Science 2009*, 2:718–723, 2009.

[41] W. Yue and J. Franco. *A New Way to Reduce Computing in Navigation Algorithm. Journal of Engineering Letters*, 18(4), 2010.

[42] W. Yue, J. Franco, W. Cao, and Q. Han. *A New Anytime Dynamic Navigation Algorithm. Proceedings of the World Congress on Engineering and Computer Science*, I:17–22, 2012.

[43] W. Yue, J. Franco, W. Cao, and H. Yue. *ID* Lite: improved D* Lite algorithm. Proceedings of 26th Symposium On Applied Computing*, pages 1364–1369, 2011.

[44] W. Yue, Y. Xu, and K. Su. *BDDRPA*: An Efficient BDD-Based Incremental Heuristic Search Algorithm for Replanning. AI 2006: Advances in Artificial Intelligence*, 4304:627–636, 2006.

[45] R. Zhou and E. Hansen. *Multiple sequence alignment using Anytime A*. Proceedings of Conference on Articial Intelligence*, pages 975–976, 2002.

[46] S. Zilberstein. *Using anytime algorithms in intelligent systems. AI Magazine*, 17(3):73–83, 1996.

[47] S. Zilberstein and S. Russell. *Approximate reasoning using anytime algorithms. Imprecise and Approximate Computation*, 1995.