Artificial Intelligence 234 (2016) 49–77

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Truncated incremental search

Sandip Aine [a,*], Maxim Likhachev [b]

[a] Indraprastha Institute of Technology Delhi (IIIT Delhi), Delhi, India
[b] Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, United States

ARTICLE INFO

Article history:
Received 6 April 2014
Received in revised form 29 November 2015
Accepted 12 January 2016
Available online 18 January 2016

Keywords:
Planning
Replanning
Anytime planning
Heuristic search
Anytime search
Incremental search

ABSTRACT

Incremental heuristic search algorithms reuse their previous search efforts whenever these are available. As a result, they can often solve a sequence of similar planning problems faster than planning from scratch. State-of-the-art incremental heuristic searches (such as LPA*, D* and D* Lite) work by propagating cost changes to all the states in the search tree whose g values (the costs of computed paths from the start state) are no longer optimal. This work is based on the observation that while a complete propagation of cost changes is essential to ensure optimality, the propagations can be stopped earlier if we are looking close-to-optimal solutions instead of the optimal one. We develop a framework called Truncated Incremental Search that builds on this observation and uses a target suboptimality bound to efficiently restrict cost propagations. We present two truncation based algorithms, Truncated LPA* (TLPA*) and Truncated D* Lite (TD* Lite), for bounded suboptimal planning and navigation in dynamic graphs. We also develop an anytime replanning algorithm, Anytime Truncated D* (ATD*), that combines the inflated heuristic search with truncation, in an anytime manner. We discuss the theoretical properties of these algorithms proving their correctness and efficiency, and present experimental results on 2D and 3D (x, y, heading) path planning domains evaluating their performance. The empirical results show that the truncated incremental searches can provide significant improvement in runtime over existing incremental search algorithms, especially when searching for close-to-optimal solutions in large, dynamic graphs.

1. Introduction

Heuristic search is a fundamental problem solving technique in Artificial Intelligence (AI) [1], that models an optimization problem in terms of planning paths in state-space graphs. Search algorithms explore the state-space using domain specific information (heuristics), which guide the search towards solutions. Over the years, search algorithms have been used in many applications, including classical planning [2], learning [3,4], robotics [5–10], network design [11–13], VLSI [14–16], drug design [17,18], bio-informatics [19], with the list growing every day.

Planning for real-world applications involves two major problems, uncertainty and complexity. Real world is an inherently uncertain and dynamic place, which means accurate models are difficult to obtain and can quickly become out of date. Replanning becomes a necessity when such a change is perceived. The challenge here is to efficiently utilize the information gathered from earlier searches to facilitate the current planning. Incremental search algorithms are meant for such dynamic environments. They reuse previous search efforts to speed up the current search, and thus can often replan faster

* Corresponding author.
E-mail address: sandipaine@gmail.com (S. Aine).

http://dx.doi.org/10.1016/j.artint.2016.01.009
0004-3702/

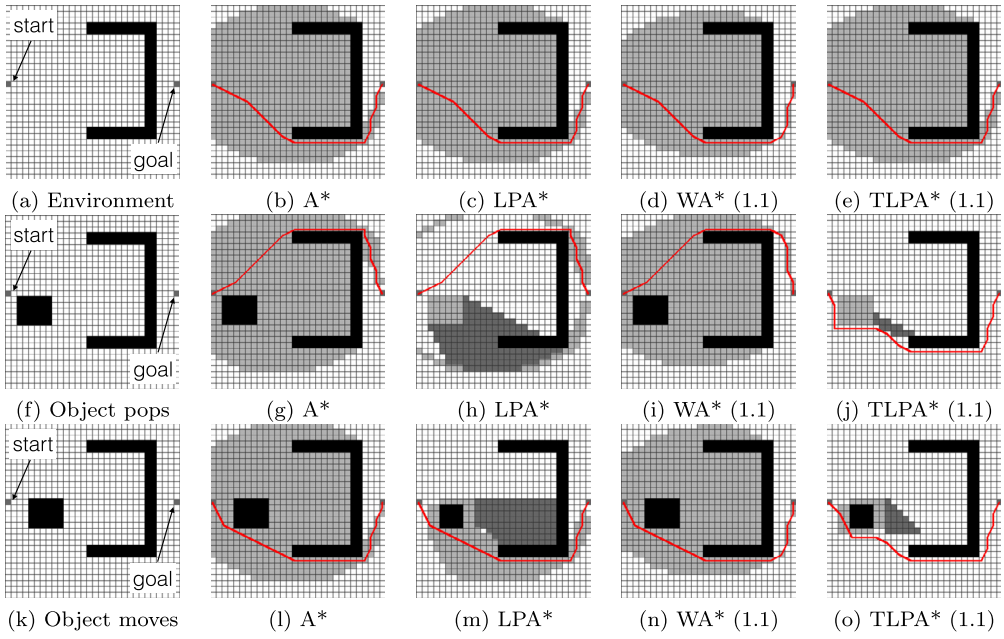


Fig. 1. A 30×30 grid path planning example showing the difference between A*, LPA*, WA* and TLPA* (WA* and TLPA* are run with suboptimality bound 1.1). In each case, we show the expanded states in grey and the path from start to goal in red (grey in print). Note that A* and WA* expand a state once only (per iteration), whereas LPA* and TLPA* can expand a state twice. To distinguish, we show the states expanded twice using dark grey and the states expanded once using light grey. The first search is identical for A*, LPA* and TLPA*, while WA* is a bit more efficient. After the first search, a new obstacle is introduced. A* and WA* recompute a *new* path from scratch. LPA* reuses the previous search tree and only rebuilds where the current tree is different from the previous one, and expands less states than both A* and WA* (1.1). However, it still expands a considerable number of states. TLPA* (1.1) quickly finds a way around the new obstacle and recomputes a bounded path with much fewer expansions. For the next iteration, the obstacle moves, blocking and unblocking some cells from the previous environment. Again, we observe that while A*, WA* (1.1) and LPA* expand a substantial number of states, TLPA* (1.1) terminates much faster, as it only propagates cost changes which are required to satisfy the cost bounds and truncates other expansions. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

than planning from scratch. For complex planning problems, it is often desirable to obtain a trade-off between the solution quality and the runtime, especially when optimal planning becomes infeasible due to resource (time/memory) constraints. Anytime search algorithms target such trade-offs, providing an initial (possibly highly suboptimal) solution quickly, and then iteratively improving the solution quality depending on the deliberation time. In this work, we focus on these two classes of search algorithms, incremental and anytime, and present *novel* algorithms for efficiently planning/replanning bounded suboptimal paths through large, dynamic graphs.

Lifelong planning A* (LPA*) [20] is an incremental version of A* (with a consistent heuristic function) which optimally solves a sequence of search problems in a dynamic environment. Its first iteration is the same as that of A*, but the subsequent searches are potentially faster as they reuse parts of the previous search tree that are identical to the current search tree. The rest of the tree is rebuilt by expanding states for which the g values (path cost from the start state) differ from the previous run (cost propagation). If large parts of the search tree remain unchanged over episodes, i.e., if the environments change only slightly and the changes are close to the goal, LPA* can converge significantly faster than A*. In Fig. 1, we present a simple path planning example in dynamic environment, showing how LPA* can outperform A*. The tree repair approach of LPA* has been used as a backbone for several incremental algorithms, such as D* Lite [21], Field D* [22], Anytime D* [23], that are widely used in practice, especially in robotics.

As LPA* recomputes the optimal solution every time the environment changes, it needs to propagate the cost changes for all the states (in the search tree) whose g values are no longer optimal. This means that even for a small change in the environment, large parts of the search tree may be regenerated, especially if the changes occur close to the root. We start with the observation that while the g values may change for a large number of states, the cost difference between the previous and the current values may not be significant for a majority of such states. LPA* treats all such states equally, as it uses a *binary* notion of change and does not account for the impact of a particular change. In contrast, if we *quantify* the possible impact of the costs changes (find out how much the path cost has changed), we may improve the replanning runtime by only re-expanding the states for which the change is significant and reusing the previous paths for the other states. For example, in Fig. 1, after the first search, an obstacle pops (Fig. 1f). This (potentially) changes the g values for 202 states, and LPA* re-expands all these states (some of them twice) before it recomputes an optimal solution. However, if we only consider states for which the (potential) change in g is more than 5% or 10%, the number of states affected reduces to 78 and 41, respectively. Similarly, when the obstacle moves (Fig. 1k), the total number of states (potentially) affected is

229, whereas the number of states for which the (potential) cost change is more than 5% is 117, and the number of states with (potential) cost change more than 10% is 67.

This observation provides an opportunity to target an effective trade-off between solution quality and replanning runtime. On one hand, we can achieve good quality solutions by propagating the cost changes for states where the changes are significant. On the other hand, we can reduce the replanning runtime by reusing the previous search values for states where the cost changes have little impact. To exploit this trade-off, we develop Truncated LPA* (TLPA*), an incremental search algorithm that speeds up replanning by using a suboptimality bound (ϵ) to limit re-expansions. TLPA* only propagates the cost changes when it is essential to ensure the suboptimality bound and reuses the previous search values for all other states (truncation). As a result, it can substantially improve the replanning runtime, and yet guarantee bounded suboptimal termination. In Fig. 1, we present an example dynamic planning scenario comparing A*, LPA*, WA* and TLPA* (the last two with suboptimality bound 1.1), highlighting the efficacy of TLPA*.

We propose two simple rules for truncation and explain how they can be integrated with LPA* to develop TLPA*. We discuss the theoretical properties of TLPA* demonstrating its correctness and efficiency, and present experimental results for two domains, 2D and 3D (x, y, heading) path planning, highlighting its efficacy over optimal and bounded suboptimal incremental searches. In addition, we discuss how the truncation rules can be applied on top of the D* Lite algorithm, resulting in Truncated D* Lite (TD* Lite), a bounded suboptimal algorithm for navigation in dynamic graphs.

We also discuss how the truncation approach can be extended to work as an anytime algorithm. Anytime D* (AD*) [23] is the current state-of-the-art algorithm for anytime replanning, which combines the benefits of incremental replanning (D* Lite [21]) and anytime planning (ARA* [5]). While AD* is both incremental and anytime, it actually speeds up the planning part (path computation) using inflated heuristics, whereas its replanning part (cost propagation) is exactly the same as LPA*/D* Lite. In contrast, truncation methods improve the replanning by restricting the cost propagations, whereas the planning part is essentially similar to A*. This indicates that if we can combine heuristic inflation with truncation (in an anytime fashion), the resulting algorithm may provide a more effective solution to the problem of anytime replanning by simultaneously improving planning and replanning.

Unfortunately, these approaches cannot be combined directly, as the truncation rules used in TLPA* (and TD* Lite) cannot work with inflated heuristics to guarantee a bounded suboptimal termination. To rectify this, we propose two *new* truncation rules, that can work on top of inflated heuristic search, without violating the suboptimality constraints. We develop Anytime Truncated D* (ATD*), an anytime replanning algorithm that uses these *new* truncation rules in conjunction with heuristic inflation, and thus can simultaneously improve both planning and replanning, offering greater efficacy and flexibility for solving complex and dynamic planning problems under limited time. We describe the theoretical properties of ATD*, and present experimental results for 2D and 3D (x, y, heading) navigation evaluating its performance.

Preliminary versions of this paper appeared in previous publications [24,25]. In this paper we provide a complete report on the truncation approach and discuss all the truncation based algorithms in greater details with suitable examples. We also include a broader analysis of the theoretical properties of truncated incremental searches and present a more thorough evaluation of their performance.

2. Related work

In this section we discuss some of the related works in the domains of incremental and anytime heuristic search.

2.1. Incremental search algorithms

The incremental heuristic search algorithms found in AI literature can be broadly classified in three main categories. The first class (LPA* [20], D* [26], D* Lite [21]) reuses the g values from the previous search during the current search to correct them when necessary, which can be interpreted as transforming the A* tree from the previous run into the A* tree for the current run. The same approach has also been used for uninformed searches (DynamicSWSF-FP [27]), to compute shortest paths in dynamic graphs. These algorithms store the previous g values of the states using an extra variable (v) and only process states for which $g \neq v$, thus reducing replanning effort. One of the major advantages of these approaches is that they generally do not make limiting assumptions about the structure or behavior of the underlying graph, and thus, can work with any type of state-space graph as well as any type of changes (edge addition/removal, cost increase/decrease).

The second class (Fringe Saving A* [28], Differential A* [29]) also reuses the previous search tree, but instead of repairing the tree, they identify the point till which the previous search is valid and resume searching from there. In particular, these algorithms restart A* at the point where the current search queue differs from the previous run. These algorithms often make limiting assumptions about the state-space or the dynamic behavior, which makes them inapplicable for general cases. For example, Fringe Saving A* [28] only works on 2D grids with unit cost transitions between neighboring cells. However, the assumptions made by these algorithms allow them to perform well in scenarios where they are applicable. Recently, an algorithm called Tree Restoring Weighted A* [30] was proposed, which uses the same philosophy of restoring the search queue, but can work with generic graphs as well as inflated heuristics.

The third class (Adaptive A* [31], Generalized Adaptive A* [32]) updates the h values from the previous searches to make them more informed over iterations. As the heuristic becomes more informed, search gets better. These approaches are simple, effective and they do not require storing the entire search tree (or the search queue) across iterations (although, some

of them also reuse previous path [33]/search tree [34] information in addition to the heuristic updates). However, as these algorithms alter the heuristic values between search iterations, they may have limitations or require additional processing. For example, the Adaptive A* [31] algorithm (also [33] and [34]) only works with cost increases and cannot handle cost decreases. Generalized Adaptive A* [32] does not have this limitation, but it requires an additional pass of heuristic updates to ensure consistency, which may have significant overhead. Another recent algorithm, Multipath Generalized Adaptive A* [35] improves upon Generalized Adaptive A* by reusing paths from previous searches.

The algorithms presented in the paper belong to the first category, as they are based on the tree repairing philosophy of LPA*/D* Lite.

Incremental search approaches have also been used in other contexts than searching in dynamic graphs. For example, incremental search algorithms (Generalized Fringe Retrieving A* [36], Moving Target D* Lite [37]) have been proposed for moving target search, where the underlying graph remains static but the search goal changes as the target moves.

Graph search based local approaches have also been used for plan adaptation/repair in dynamic environments [38,39]. These algorithms generally focus on improving different metrics (such as plan stability [40]) and typically they do not provide any guarantees on the quality of solutions.

Another closely related class of search algorithms is real-time search. Real-time search algorithms (such as RTA* and LRTA* [41], RTAA* [42], etc.) interleave planning and execution. They compute a partial plan by limiting the search horizon, execute one or more of the chosen actions, and replan from the current state. The fundamental difference between incremental and real-time search is that the incremental search algorithms compute a complete plan according to the current environment (actual or perceived) and only replan when they sense a change in the environment, whereas real-time search algorithms compute partial plans, execute and replan iteratively until it reaches the goal, in a static environment. There are also search algorithms (Real-time D* [43], RTTES [44]) that are both real-time and incremental.

2.2. Anytime search algorithms

Design of anytime search algorithms has received considerable attention in the last two decades. A number of anytime algorithms are based on the Weighted A* (WA*) [45], where the actual (consistent) heuristic values are multiplied by an inflation factor ($w > 1.0$) to give the search a depth-first flavor. Inflated heuristic searches are shown to be fast for many domains [46,5], they also provide an implicit bound on the suboptimality, namely, the factor (w) by which the heuristic is inflated [47]. Hansen et al. [48,49] proposed a simple technique for converting WA* into an anytime algorithm, Anytime Weighted A* (AWA*). Instead of terminating when the first solution is found, AWA* continues to search more, computing a sequence of improved solutions, updating the quality bounds whenever a new solution is discovered. Anytime Repairing A* (ARA*) [5] adopts a slightly different approach. It reduces the inflation factor every time a solution is found, iteratively targeting tighter bounds. It also reuses the previous search efforts maximally, and restricts the states expansions to one per iteration. ARA* is widely used for path planning in dense graphs, especially in robotics applications. Other algorithms, like RWA* [50] and ANA* [51] also exploit the WA* principle. RWA* proposes a restart based approach in contrast to the reuse based approach of ARA*, whereas ANA* is a non-parametric (i.e., it does not require input parameters) anytime search that greedily looks for solutions among states with the highest potential to improve upon the incumbent solution.

There are also several anytime searches that are not based on WA*. Most of these use some constraints to restrict the search space for quick termination, or use a different metric than the conventional f values to order the state expansions. These constraints are iteratively relaxed to improve the solution quality in an anytime manner. These algorithms include variants of depth-first search (Depth-first Branch-and-Bound search [52] and Complete Anytime Beam search [53]), variants of the beam search [54] (Beam-Stack search [55], ABULB [56,57], Anytime Pack Search [58], Anytime Column Search [59]), Anytime Window A* (AWinA*) [60], where a window of chosen depth is used to restrict the active search space, Anytime Explicit Estimation Search (AEES) [61], that uses a distance-to-go estimate (similar to the A^*_ϵ algorithm [62]) to determine the order of expansions. Unlike WA* based approaches, most of these algorithms do not provide any implicit bounds on their solutions. Although, some of them (BQAWinA* [60], AEES [61]) can adapt to target suboptimality bounds. On the other hand, most of these algorithms use less memory than the WA* approaches (some can run under any memory restrictions [53,63]), and therefore, can scale better to solve large sized problems under reasonable memory/runtime limits.

2.3. Anytime and incremental search

While there are a number of heuristic search algorithms that are either incremental or anytime, algorithms that satisfy both these properties are rare. In fact, to our knowledge Anytime D* [23], which combines ARA* with D* Lite, and Anytime Tree Repairing A* [64] which combines ARA* with Tree Repairing A*, are the only two anytime algorithms that work for general dynamic graphs. Another algorithm called Incremental ARA* (I-ARA*) [65] also uses ARA* incrementally, but it is applicable for moving target search in static graphs only and not for dynamic graphs with arbitrary cost changes.

At this point, it may be noted that while both the inflated heuristic search algorithms (ARA* [5], GLPA* [66], AD* [23], I-ARA* [65], ATRA* [64]) and truncation based searches proposed in this work produce provably guaranteed solutions within a chosen suboptimality bound, these algorithms are fundamentally different. The former speed up the search (planning) using inflated heuristics to order the state expansions, whereas truncated based algorithm use the suboptimality bound to accelerate replanning by restricting state re-expansions, with or without heuristic inflation.

3. Truncated LPA*

In this section we discuss the concept of truncation and demonstrate how it can be used to convert LPA* into a bounded suboptimal replanning algorithm. We start with brief description of LPA*, and then move on to explain the truncation rules.

Notation. In the following, S denotes the finite set of states of the domain. s_{start} denotes the start state and s_{goal} denotes the goal state. $c(s, s')$ denotes the cost of the edge between s and s', if there is no such edge, then $c(s, s') = \infty$. We assume $c(s, s') \geq 0$ and $c(s, s) = 0$, i.e., all the edge costs are non-negative and self-loops have zero cost. $\text{Succ}(s) := \{s' \in S | c(s, s') \neq \infty\}$, denotes the set of all successors of s. Similarly, $\text{Pred}(s) := \{s' \in S | s \in \text{Succ}(s')\}$ denotes the set of predecessors of s. $c^*(s, s')$ denotes the cost of the optimal path from s to s', if no such path exists, then $c^*(s, s') = \infty$. $g^*(s)$ denotes optimal path cost from s_{start} to s. We use $\pi(s)$ to denote a path from s_{start} to s. $h(s)$ denotes the heuristic value of s, which we assume to be consistent (and thus admissible), i.e., we assume $h(s_{goal}) = 0$ and $h(s) \leq h(s') + c(s, s')$, $\forall s, s'$ such that $s' \in Succ(s)$ and $s \neq s_{goal}$. We use the symbol ∞ to denote a very large number (more than any possible value of path cost). We define the comparison operator with ∞ in the following manner, $[x < \infty]$ is true for any $x \neq \infty$, else it is false.

3.1. LPA*

LPA* [20] is an incremental version of A* that uses consistent heuristics and breaks ties among states with the same f values in favor of states with smaller g values. LPA* repeatedly determines a minimum-cost path from s_{start} to s_{goal} in a state-space graph, while some of the edge costs change. It maintains two kinds of estimates of the cost of a path from s_{start} to a state s: $g(s)$ and $v(s)$. $v(s)$ holds the cost of the best path found from s_{start} to s during its last expansion, while $g(s)$ is computed from the v values of $\text{Pred}(s)$, and thus, is potentially better informed than $v(s)$. Additionally, it stores a back-pointer $bp(s)$ pointing to the best predecessor of s (if computed), for each s. A state s is called consistent if $v(s) = g(s)$, otherwise it is either overconsistent (if $v(s) > g(s)$) or underconsistent (if $v(s) < g(s)$).

LPA* works by identifying inconsistent states (in a search tree) and systematically updating their v, g, and bp values, till an optimal path is discovered. At any intermediate point, it satisfies the following invariants,

- **Invariant 1:** $bp(s_{start}) = \textbf{null}$, $g(s_{start}) = 0$ and $\forall s \in S \setminus \{s_{start}\}$, $bp(s) = \text{argmin}_{(s' \in \text{Pred}(s))} v(s') + c(s', s)$, $g(s) = v(bp(s)) + c(bp(s), s)$, i.e., all states update their g values using the best path information from $\text{Pred}(s)$, other than s_{start} for which the g value is set to 0 by definition.
- **Invariant 2:** The priority queue (OPEN) only contains the inconsistent states.
- **Invariant 3:** The priority of a state s is given by: $\text{Key}(s) = [\text{Key}_1(s), \text{Key}_2(s)]$ where $\text{Key}_1(s) = min(g(s), v(s)) + h(s)$ and $\text{Key}_2(s) = min(g(s), v(s))$. Priorities are compared in lexicographic order, i.e., for two states s and s', $\text{Key}(s) \leq \text{Key}(s')$, iff either $\text{Key}_1(s) < \text{Key}_1(s')$ or $(\text{Key}_1(s) = \text{Key}_1(s')$ and $\text{Key}_2(s) \leq \text{Key}_2(s'))$.

It may be noted that the tie-breaking part of the Key function is important for LPA*. A* can use a tie-breaking favoring states with larger g values. The same is not true for LPA*, as in case of a tie, it needs to process an underconsistent state before an overconsistent state [20]. However, the tie-breaking in LPA* (and related algorithms) can be improved by using $\text{Key}_1(s) = min(g(s), v(s)) + h(s)$; and $\text{Key}_2(s) = 0$ if $v(s) < g(s)$ (i.e. if s in underconsistent), $\text{Key}_2(s) = 1$, otherwise [66]; without violating any properties. Throughout this paper, whenever we use A*/WA*, we use tie-breaking favoring states with larger g values, and whenever we use LPA* based algorithms, we use the tie-breaking favoring underconsistent states.

The pseudocode of a basic version of LPA* is shown in Algorithm 1. LPA* starts by initializing the states and inserting s_{start} into OPEN (lines 30–32). It then calls ComputePath to obtain a minimum cost solution. ComputePath expands the inconsistent states from OPEN in increasing order of priority, maintaining **Invariants 1–3**, until it discovers a minimum cost path to s_{goal}. If a state s is overconsistent, ComputePath makes it consistent (line 21) and propagates this information to its successors according to **Invariant 1**. This may make some $s' \in \text{Succ}(s)$ inconsistent, which are then put into OPEN ensuring **Invariant 2**. If s is underconsistent, ComputePath makes it overconsistent by setting $v(s) = \infty$ (line 25) and propagates the underconsistency information to its children, and selectively puts s and its children back to OPEN (maintaining **Invariants 1–2**). Note that, during the initialization, $v(s)$ is set to ∞, $\forall s \in S$ (line 4). Thus, the first iteration of ComputePath is exactly the same as A*. After the first iteration, if one or more edge costs change, LPA* updates the g and bp values of the affected states (line 38), and inserts the inconsistent state in OPEN. It then calls ComputePath again to recompute an optimal path.

In Fig. 2, we illustrate the working of LPA* on a simple example graph. To simplify the example, we assume that $h(s) = 0, \forall s$. Fig. 2a shows the first run where each expanded state (shown in grey) has $v = g$. The initial solution path is shown by the back-pointers (dashed line). After the first search, the cost of the edge from S to A changes from 1 to 6, making A an underconsistent state, with $g(A) = 6$ and $v(A) = 1$ (Fig. 2b). LPA* expands A (Fig. 2c) to propagate this inconsistency information to D, and continues to expand of D and F (Figs. 2d and 2e) before computing the optimal solution (Fig. 2f). A total of 5 states are expanded in the second iteration (A once, D and F twice).

LPA* guarantees the following properties,

Algorithm 1 LPA*.

```
1: procedure Key(s)
2:     return [min(g(s), v(s)) + h(s); min(g(s), v(s))]
3: procedure InitState(s)
4:     v(s) = g(s) = ∞; bp(s) = null
5: procedure UpdateState(s)
6:     if s was never visited then
7:         InitState(s)
8:     if s ≠ s_start then
9:         bp(s) = argmin_{(s' ∈ Pred(s))} v(s') + c(s', s)
10:        g(s) = v(bp(s)) + c(bp(s), s)
11:        if g(s) ≠ v(s) then
12:            insert/update s in OPEN with Key(s) as priority
13:        else
14:            if s ∈ OPEN then
15:                remove s from OPEN
16: procedure ComputePath
17:    while OPEN.MinKey() < Key(s_goal) ⋁ v(s_goal) < g(s_goal) do
18:        s = OPEN.Top()
19:        remove s from OPEN
20:        if v(s) > g(s) then
21:            v(s) = g(s)
22:            for all s' ∈ Succ(s) do
23:                UpdateState(s')
24:        else
25:            v(s) = ∞
26:            UpdateState(s)
27:            for all s' ∈ Succ(s) do
28:                UpdateState(s')
29: procedure Main
30:    InitState(s_start); InitState(s_goal)
31:    g(s_start) = 0; OPEN ← ∅
32:    Insert s_start in OPEN with Key(s_start)
33:    while 1 do
34:        ComputePath
35:        Wait for changes in edge costs
36:        for all directed edges (u, v) with changed edge costs do
37:            update edge cost c(u, v)
38:            UpdateState(v)
```

- **Optimality:** When the ComputePath function exits, the cost of the path from s_{start} to s_{goal} obtained by following the back-pointers stored in every state is no larger than $g^*(s_{goal})$.
- **Efficiency:** No state is expanded more than twice during the execution of ComputePath. Moreover, a state s is expanded in ComputePath, only if either it was inconsistent initially or made inconsistent during the current execution of ComputePath.

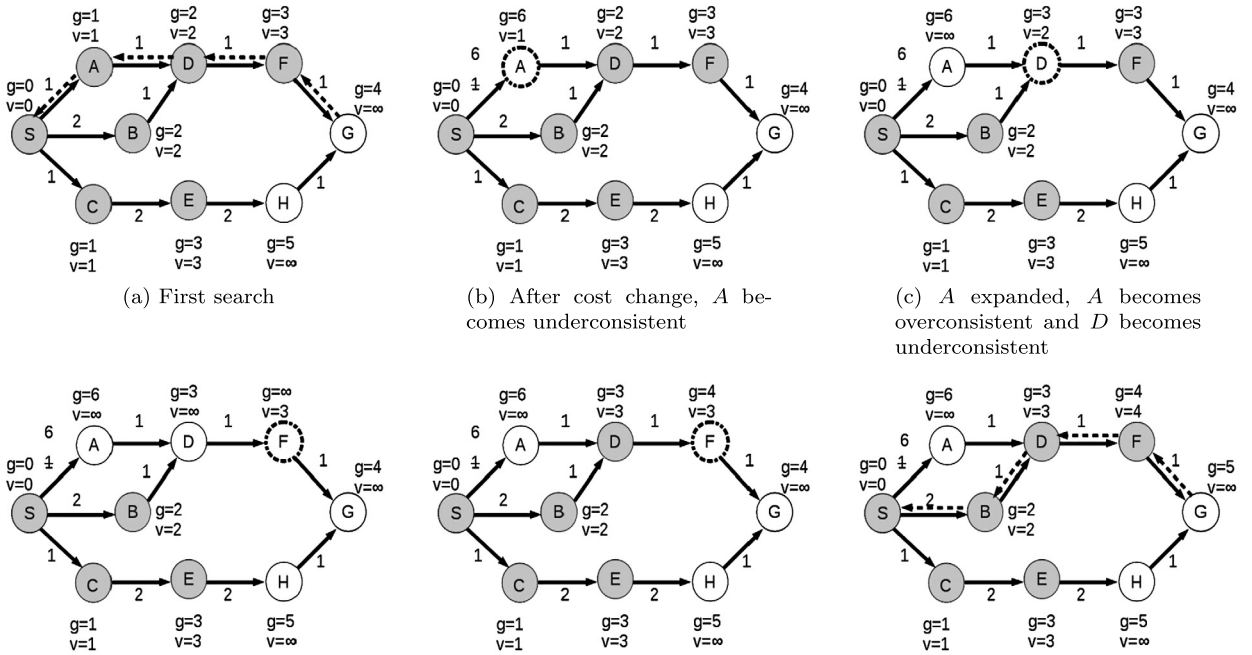
3.2. Truncation rules

As discussed in Section 3.1, when cost changes occur, LPA* recomputes an optimal solution by propagating the cost change information through expansion of states. In this section, we present two rules, that can be used to limit such re-expansions using a target suboptimality bound.

First, we introduce a new term called $g^{\pi}(s)$, to denote the cost of the path from s_{start} to s computed by following the current back-pointers (bp). If no such path exists, then $g^{\pi}(s) = \infty$. Note that, $g^{\pi}(s)$ may be different from $g(s)$, as $g(s)$ holds the path cost computed directly using v values of Pred(s) (**Invariant 1**), whereas $g^{\pi}(s)$ is computed by following the back-pointers from s to s_{start}. In the truncation rules, we use the g^{π} value for an inconsistent state s and a chosen suboptimality bound ϵ, to decide whether we need to expand state s to satisfy the bound. If the bounds can be guaranteed without expanding s, we truncate s (remove it from OPEN without expansion).

Truncation Rule 1 Rule 1 is applicable for underconsistent states. We observe that when we are looking for an ϵ bounded solution, we can reuse the *old* path cost $v(s)$ for an underconsistent state s (selected for expansion), as long as $g^{\pi}(s) + h(s) \leq \epsilon \cdot (v(s) + h(s))$. This stems from the fact that for an underconsistent state s selected for expansion, $v(s) + h(s)$ is a lower bound on the solution cost through s, as $v(s)$ holds previous shortest path cost (from s_{start}) and $h(s)$ is a consistent heuristic. If the current path to s (from s_{start}) satisfies the bound on $v(s) + h(s)$, any state s' that uses $v(s)$ to compute its $g(s')$ will never underestimate the actual solution cost by more than the ϵ factor. In other words, even if we compute the new solution using the *old* $v(s)$, the obtained solution cost will be less than or equal to $\epsilon \cdot g^*(s_{goal})$, as the *old* paths have not deteriorated beyond the chosen bound.

We explain this rule in Fig. 3 using the same example as shown in Fig. 2, however, unlike Fig. 2, here we search for a two bounded solution (i.e., $\epsilon = 2.0$). The first search (same as A*/LPA*) is shown in Fig. 3a. After the first episode, the edge cost from S to A changes from 1 to 6 (Fig. 3b). Now, when A is selected for expansion, we compute $g^{\pi}(A)$ (Fig. 3c) to check



(a) First search

(b) After cost change, A becomes underconsistent

(c) A expanded, A becomes overconsistent and D becomes underconsistent

(d) D expanded, F becomes underconsistent and D becomes overconsistent

(e) D expanded again making it consistent, F still underconsistent

(f) F expanded twice, and the new optimal solution is computed

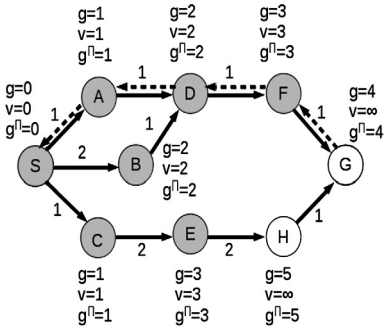
Fig. 2. An example explaining the LPA* algorithm.

whether $g^{\pi}(A) + h(A) \leq \epsilon \cdot (v(A) + h(A))$. For A, this condition is not true ($g^{\pi}(A) = 6$, $v(A) = 1$ and $h(A) = 0$). Thus, we expand A, making D underconsistent. Next, when D is selected for expansion, we find that $g^{\pi}(D) + h(D) \leq \epsilon \cdot (v(D) + h(D))$, as $g^{\pi}(D) = 3$ and $v(D) = 2$, i.e., D satisfies the truncation condition. At this point, we can truncate the cost propagation at D, as the successors of D can continue to rely on the *old* $v(D)$ ($v(D) = 2$) and yet guarantee that the path cost through them will be within the ϵ bound. In this example, the replanning iteration terminates at this point with a solution of cost 5, as there are no more inconsistent states to process. Note that, for this example, LPA* requires 5 re-expansions to recompute an optimal solution, whereas TLPA* provides a 2 bounded solution with 1 re-expansion only. Next, we formally state Truncation Rule 1.

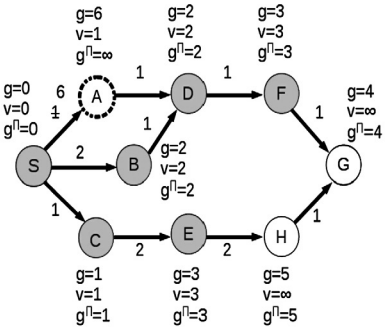
Rule 1. *An underconsistent state s having* $\textsc{Key}(s) \leq \textsc{Key}(u), \forall u \in OPEN$ *is truncated (removed from OPEN without expansion) if* $g^{\pi}(s) + h(s) \leq \epsilon \cdot (v(s) + h(s))$.

Truncation Rule 2 Rule 2 is applicable for both underconsistent and overconsistent states. The underlying principle of this rule is very simple. We observe that LPA* expands inconsistent states in an increasing order of their Key values, until the goal state has the lowest Key value in OPEN. Therefore, for any state s selected for expansion, $\textsc{Key}_1(s) = min(g(s), v(s)) + h(s)$ provides a lower bound on the minimum-cost solution, i.e., $\textsc{Key}_1(s) \leq g^*(s_{goal})$. Now, if $g^{\pi}(s_{goal}) \leq \epsilon \cdot \textsc{Key}_1(s)$, i.e., the cost of the solution obtained by following the back-pointers from s_{goal} to s_{start} is not more than $\epsilon \cdot \textsc{Key}_1(s)$, we have $g^{\pi}(s_{goal}) \leq \epsilon \cdot \textsc{Key}_1(s) \implies g^{\pi}(s_{goal}) \leq \epsilon \cdot g^*(s_{goal})$. In other words, the current solution is already within the ϵ bound of the optimal cost solution. Therefore, we can truncate s without violating the bound. Moreover, as the Key values of expanded states in LPA* are monotonically non-decreasing during the execution of ComputePath, the same condition ensures that expansion of any other state s' in OPEN cannot improve the current bound, i.e., we may terminate the current replanning iteration at this point.

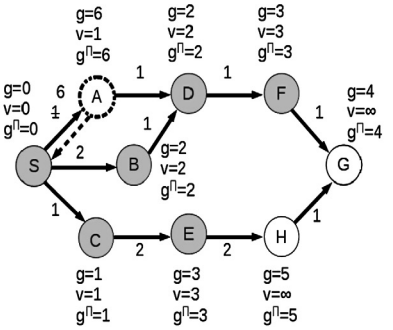
We explain this rule with Fig. 4 in comparison to LPA*. The first search is shown in Fig. 4a. After the first episode, cost of the edge from C to E changes from 2 to 1, making E overconsistent (Fig. 4b). LPA* propagates this cost change until $\textsc{Key}(s_{goal})$ becomes the minimum Key value in OPEN, i.e., it expands E and H before returning the previous solution as the minimum cost solution (Fig. 4c). Now, let us consider Truncation Rule 2 with $\epsilon = 2.0$. In Fig. 4d, when E is selected for expansion, it has $\textsc{Key}_1(E) = 2$ ($g(E) = 2$, $v(E) = 3$, and $h(E) = 0$), and we $g^{\pi}(G) = 4$ (path shown in dashed arrows, Fig. 4e). As expansion of E cannot produce a solution with cost less than 2 and $g^{\pi}(G) = 4$, at this point, we can truncate E and return the current path as a 2 bounded solution. Thus, for this example, we can guarantee 2 bounded solution without any re-expansions. Next, we formally state Truncation Rule 2.



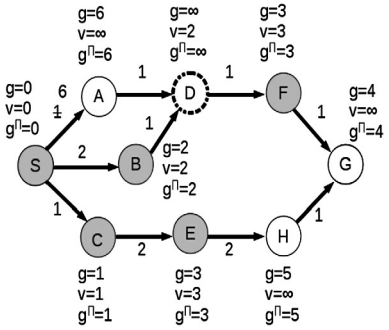
(a) First search



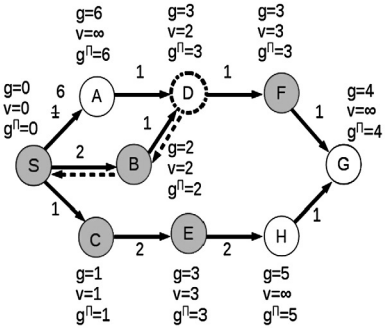
(b) After cost change, underconsistent state A



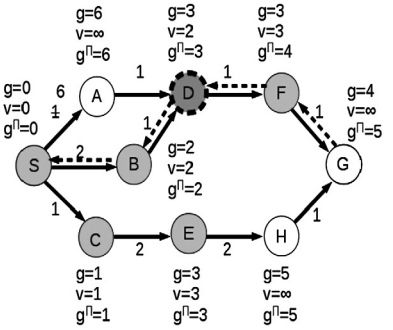
(c) $g^{\pi}(A) + h(A) > \epsilon \cdot (v(A) + h(A))$, A will be expanded



(d) A expanded, A becomes overconsistent, D becomes underconsistent



(e) $g^{\pi}(D) + h(D) \leq \epsilon \cdot (v(D) + h(D))$, D satisfies Truncation Rule 1



(f) D is truncated, the replanning iteration ends with an ϵ bounded solution

Fig. 3. Example of Truncation Rule 1 with $\epsilon = 2.0$.

Rule 2. *A state s having* $\textsc{Key}(s) \leq \textsc{Key}(u), \forall u \in OPEN$ *is truncated if* $g^{\pi}(s_{goal}) \leq \epsilon \cdot \textsc{Key}_1(s)$. *Also, if any state s is truncated using Rule 2, all states* $s' \in OPEN$ *are truncated as,* $\forall s' \in OPEN,\ \textsc{Key}_1(s') \geq \textsc{Key}_1(s)$.

3.3. Truncated LPA* algorithm

Truncated LPA* (TLPA*) incorporates the truncation rules in LPA*. We include the pseudocode for TLPA* in Algorithm 3. TLPA* uses two extra variables for each state (compared to LPA*): $g^{\pi}(s)$, as described earlier, and $\pi(s)$, which stores the current path (from s_{start}) for a truncated state s. It also uses an additional list TRUNCATED to store the underconsistent states that are truncated.

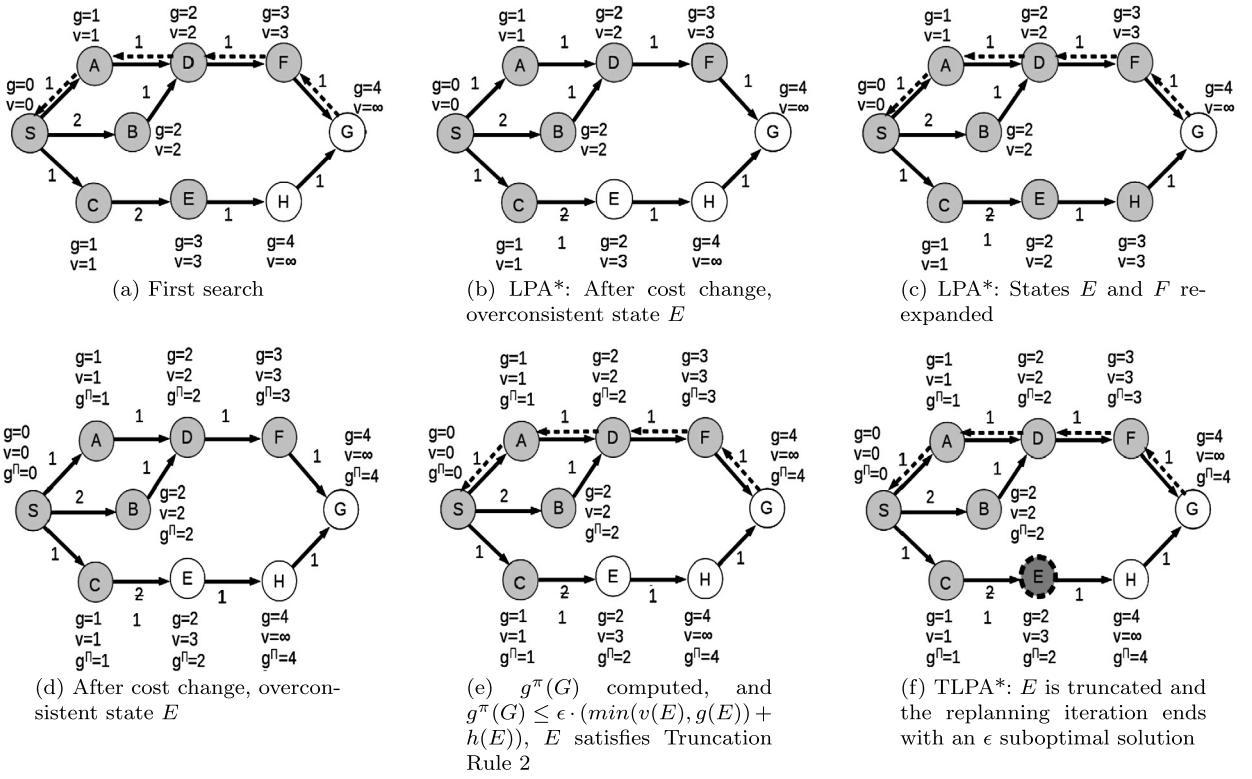
TLPA* uses two auxiliary functions, namely ComputeGpi, to populate the $g^{\pi}(s)$ and $\pi(s)$ for a given state, and ObtainPath, to populate the solution path when ComputePath terminates. These functions are described in Algorithm 2.

ComputePath in TLPA* uses the g^{π} values to apply the truncation rules. Before each expansion, $g^{\pi}(s_{goal})$ is computed to check whether Rule 2 can be applied. If the check at line 21 (Algorithm 3) is satisfied, ComputePath terminates with solution cost equal to $g^{\pi}(s_{goal})$. When an underconsistent state s is selected for expansion, its $g^{\pi}(s)$ is used to check whether Rule 1 can be applied. If the check at line 30 is satisfied, s is truncated, otherwise it expanded in the same manner as LPA*.

Apart from the application of the truncation rules the behavior of TLPA* is essentially the same as LPA*, other than the fact that before each call of ComputePath, TLPA* also updates the values for the truncated states in addition to the states for which the path cost has changed, and inserts all the inconsistent states in OPEN (lines 48–50).

While the pseudocodes in Algorithms 2 and 3 describe the behavior of TLPA*, its runtime and memory can be further optimized in a few obvious ways. Some of the optimizations we use are the following,

- In ComputePath, $g^{\pi}(s_{goal})$ is computed before each expansion. We reduce the number of calls by hashing the states in current path s_{start} to s_{goal} and only re-computing $g^{\pi}(s_{goal})$ if any of the state in this path is updated after the previous computation.
- In ComputeGpi, we do not store the entire path from s_{start} to s, if we encounter a truncated state s', we store the path up to s' and modify the ObtainPath to dynamically update the paths from truncated states.
- Furthermore, while computing the g^{π} for any state s other than s_{goal}, we terminate the computation if the cost of the path is greater than or equal to $\epsilon \cdot (v(s) + h(s))$ and set $g^{\pi}(s) = \infty, \pi(s) =$ **null**, as s will not be truncated in this case.



(a) First search

(b) LPA*: After cost change, overconsistent state E

(c) LPA*: States E and F re-expanded

(d) After cost change, overconsistent state E

(e) $g^{\pi}(G)$ computed, and $g^{\pi}(G) \leq \epsilon \cdot (min(v(E), g(E)) + h(E))$, E satisfies Truncation Rule 2

(f) TLPA*: E is truncated and the replanning iteration ends with an ϵ suboptimal solution

Fig. 4. Example of Truncation Rule 2 (with $\epsilon = 2.0$) in comparison with LPA*. LPA* runs are shown in the first row, and the second row shows the TLPA* runs.

Algorithm 2 Truncated LPA*: auxiliary routines.

```
1: procedure ComputeGpi(s)
2:     cost = 0; s' = s; π = null
3:     Visited ← ∅
4:     while s' ≠ s_start do
5:         if s' ∈ Visited ⋁ bp(s') = null then
6:             g^π(s) = ∞; π(s) = null
7:             return
8:         else
9:             if s' ∈ TRUNCATED then
10:                g^π(s) = cost + g^π(s'); π(s) = π ∪ π(s')
11:                return
12:        insert s' in Visited;
13:        cost = cost + c(bp(s'), s'); π = π ∪ bp(s'); s' = bp(s');
14:    g^π(s) = cost; π(s) ← π
15: procedure ObtainPath(s)
16:    π = s
17:    while s ≠ s_start do
18:        if bp(s) ∈ TRUNCATED then
19:            π = π ∪ π(bp(s))
20:            return π
21:        π = π ∪ bp(s); s = bp(s)
22:    return π
```

- Once a state s is truncated it is never re-inserted in OPEN during ComputePath. Therefore, we do not update the g and bp values of a truncated state while expanding other states.

3.4. Theoretical properties of TLPA*

In [67], we prove a number of properties of TLPA*. Here, we discuss some of the important results. We start with a series of Lemmas describing the properties of expanded and truncated states. Later, we will use these properties to ascertain the correctness and efficiency of TLPA*. For the ease of the proofs, we assume a trivial initialization, that sets $g(s) = v(s) = g^{\pi}(s) = \infty$ and $bp(s) = \textbf{null}$, $\forall s \in S$.

Algorithm 3 Truncated LPA*.

```
 1: procedure Key(s)
 2:     return [min(g(s), v(s)) + h(s); min(g(s), v(s))]
 3: procedure InitState(s)
 4:     v(s) = g(s) = g^π(s) = ∞; bp(s) = π(s) = null
 5: procedure UpdateState(s)
 6:     if s was never visited then
 7:         InitState(s)
 8:     if s ≠ s_start then
 9:         bp(s) = argmin_{(s'∈Pred(v))} v(s') + c(s', s)
10:         g(s) = v(bp(s)) + c(bp(s), s)
11:         if s ∉ TRUNCATED then
12:             if g(s) ≠ v(s) then
13:                 insert/update s in OPEN with Key(s) as priority
14:             else
15:                 if s ∈ OPEN then
16:                     remove s from OPEN
17: procedure ComputePath
18:     while OPEN.MinKey() < Key(s_goal) ⋁ v(s_goal) < g(s_goal) do
19:         s = OPEN.Top()
20:         ComputeGpi(s_goal)
21:         if g^π(s_goal) ≤ ϵ · (min(g(s), v(s)) + h(s)) then
22:             Terminate
23:         remove s from OPEN
24:         if v(s) > g(s) then
25:             v(s) = g(s)
26:             for all s' ∈ Succ(s) do
27:                 UpdateState(s')
28:         else
29:             ComputeGpi(s)
30:             if g^π(s) + h(s) ≤ ϵ · (v(s) + h(s)) then
31:                 insert s in TRUNCATED
32:             else
33:                 v(s) = ∞
34:                 UpdateState(s)
35:                 for all s' ∈ Succ(s) do
36:                     UpdateState(s')
37: procedure Main
38:     InitState(s_start); InitState(s_goal)
39:     g(s_start) = g^π(s_start) = 0;
40:     OPEN ← ∅; TRUNCATED ← ∅
41:     Insert s_start in OPEN with Key(s_start)
42:     while 1 do
43:         ComputePath; ObtainPath(s_goal)
44:         Wait for changes in edge costs
45:         for all directed edges (u, v) with changed edge costs do
46:             update edge cost c(u, v)
47:             UpdateState(v)
48:         for all s ∈ TRUNCATED do
49:             remove s from TRUNCATED; g^π(s) = ∞; π(s) ← null
50:             UpdateState(v)
```

Lemma 1. *In TLPA*, i) all v and g values are non-negative, ii)* $bp(s_{start}) =$ **null** *and* $g(s_{start}) = 0$*, and iii)* $\forall s \neq s_{start}$, $bp(s) = argmin_{(s' \in Pred(s))}\, v(s') + c(s', s)$ *and* $g(s) = v(bp(s)) + c(bp(s), s)$.

Proof. The Lemma holds right after the trivial initialization, as all v and g values are ∞, and bp values are set to **null**. At the start, $g(s_{start})$ is set to 0, and $bp(s_{start})$ is set to **null**. Note that, g and bp of s_{start} are never changed in UpdateState. Thus, $bp(s_{start}) =$ **null** and $g(s_{start}) = 0$, remains true throughout (ii).

Now, for any state $v(s)$ is only altered if the state is expanded (line 25 or line 33), and in both the cases the UpdateState is called $\forall s' \in Succ(s)$, which updates the g and bp values according to the Lemma statement (iii).

As, $g(s_{start})$ is 0 at the start, thus, $v(s_{start})$ is set to 0 during the first expansion. As all the costs are non-negative, any subsequent assignment of g values (for a state s) through UpdateState ensures $g(s) \geq 0$, and since $v(s)$ is either set to $g(s)$ or set to ∞, v values can never be negative (i). □

Lemma 2. *In TLPA*, i) OPEN and TRUNCATED are disjoint, ii) OPEN only contains the inconsistent states and OPEN ∪ TRUNCATED contains all the inconsistent states, and iii) before each call of* ComputePath*, OPEN contains all the inconsistent states and TRUNCATED is empty.*

Proof. All i), ii) and iii) are certainly true before the first call of ComputePath, as TRUNCATED is empty and OPEN only contains s_{start} which is inconsistent ($g(s_{start}) = 0$, $v(s_{start}) = \infty$).

Now, during ComputePath whenever a state s is removed from OPEN, the following cases may occur,

1. $v(s) > g(s)$: $v(s)$ is set to $g(s)$ (i.e., s becomes consistent), and UpdateState is called $\forall s' \in \text{Succ}(s)$ which ensures that the resulting inconsistent states are inserted in OPEN if they are not in TRUNCATED.
2. $v(s) < g(s)$, and the state is expanded: $v(s)$ is set to ∞, and UpdateState is called for $s \cup \text{Succ}(s)$ ensuring the Lemma statements.
3. $v(s) < g(s)$, and s is truncated: in this case, s is added to TRUNCATED and no other states g and v values are altered.

Thus, in all three cases the Lemma statements hold true.

When ComputePath terminates, all the states from TRUNCATED and all the states for which the cost has potentially changed are processed through UpdateState, ensuring that the inconsistent states are inserted to OPEN, and TRUNCATED is made empty. □

Lemma 3. *If a state s is truncated ($s \in$ TRUNCATED), $v(s)$ is never altered during the execution of* ComputePath.

Proof. The v value for a state is only altered when it is expanded. Now, if $s \in$ *TRUNCATED*, it cannot be selected for expansion, as the selections are made from OPEN, and OPEN and TRUNCATED are mutually disjoint (Lemma 2). □

Lemma 4. *In* ComputePath, $\forall s \in$ *TRUNCATED, there is a finite cost path from* s_{start} *to s with cost* $g^{\pi}(s)$, *stored in* $\pi(s)$.

Proof. We prove this by induction.

Consider the first state s that gets truncated. First, note that $v(s) < \infty$ (as $v(s) < g(s)$), which implies $g^{\pi}(s) < \infty$. Now, as no other states have been truncated, $g^{\pi}(s)$ is computed by adding the action costs from s_{start} to s following the *bp* pointers for each state in the path (check in line 9, Algorithm 2 is never true).

Assume that the path is given by $\pi(s_0 = s_{start}, \ldots, s_k = s)$. For any state s_i in this path, $bp(s_i) \neq$ **null** and also, the path does not contain a cycle, because in both the cases $g^{\pi}(s)$ will be set to ∞ (line 5, Algorithm 2). The cost of every edge along this path is added to $g^{\pi}(s)$ (the states are added to $\pi(s)$). Thus, when s is truncated, the path followed in ComputeGpi(s) routine is a valid one, with cost $g^{\pi}(s)$, stored in $\pi(s)$.

Now, let us assume that the Lemma statement holds for all prior truncation, and we will prove it for a current under-consistent state s' which is truncated. If the path followed by ComputeGpi(s') does not visit any truncated state then the Lemma is valid (from the earlier case). Whereas, if it encounters a state $s'' \in$ TRUNCATED, we augment $g^{\pi}(s')$ with $g^{\pi}(s'')$ and concatenate $\pi(s'')$ to $\pi(s')$. As $s'' \in$ TRUNCATED, $\pi(s'')$ is valid and has cost $g^{\pi}(s'')$ (from induction hypothesis), which means $\pi(s')$ is valid and has cost $g^{\pi}(s')$. □

Lemma 5. *In* ComputePath, *any state s selected for expansion/truncation (i.e., if* Key$(s) \leq$ Key$(u), \forall u \in$ *OPEN at line 18) satisfies,* $g(s) \leq g^*(s)$ *if* $v(s) \geq g(s)$ *and* $v(s) \leq g^*(s)$ *if* $v(s) < g(s)$.

Proof. We prove this inductively using contradiction. The statement is true when OPEN contains only s_{start} (as $g(s_{start}) = 0 = g^*(s_{start})$). Let us assume that the Lemma is true for all selections prior to the current one, when the state s is selected.

First, we consider the case where $v(s) \geq g(s)$. Let us say, $g(s) > g^*(s)$ (which implies $g^*(s) < \infty$). Consider a least-cost path from s_{start} to s, $\pi(s_0 = s_{start}, \ldots, s_k = s)$ with cost $g^*(s)$. Such a path must exist, since $g^*(s) < \infty$. Our assumption, that $g(s) > g^*(s)$ means that there exists at least one $s_i \in \pi(s_0 = s_{start}, \ldots, s_k = s)$, namely s_{k-1}, whose $v(s_i) > g^*(s_i)$. Otherwise, from Lemma 1,

$$\begin{aligned} g(s) = g(s_k) &= min_{(s' \in Pred(s))}\, v(s') + c(s', s_k) \\ &\leq v(s_{k-1}) + c(s_{k-1}, s_k) \\ &\leq g^*(s_{k-1}) + c(s_{k-1}, s_k) \\ &\leq g^*(s_k) \\ &\leq g^*(s) \end{aligned} \tag{1}$$

Let us now consider $s_i \in \pi(s_0, \ldots, s_{k-1})$ with the smallest index $i \geq 0$, such that $v(s_i) > g^*(s_i)$. We first show that $g^*(s_i) \geq g(s_i)$. It is clearly so when $i = 0$ according to Lemma 1 which says that $g(s_i) = g(s_{start}) = 0$. For $i > 0$, we use the fact that $v(s_{i-1}) \leq g^*(s_{i-1})$ from the way s_i was chosen. Now,

$$\begin{aligned} g(s_i) &= min_{(s' \in Pred(s_i))}\, v(s') + c(s', s) \\ &\leq v(s_{i-1}) + c(s_{i-1}, s_i) \\ &\leq g^*(s_{i-1}) + c(s_{i-1}, s_i) \\ &\leq g^*(s_i) \end{aligned} \tag{2}$$

We thus have $v(s_i) > g^*(s_i) \geq g(s_i)$, which also implies that $s_i \in$ OPEN (from Lemma 2). Note that, s_i cannot be in TRUNCATED as that would mean $v(s_i) \leq g^*(s_i)$, because s_i is selected prior to s for truncation (from induction hypothesis). Now, according to our assumption

$$\begin{aligned}
\textsc{Key}(s) &= [g(s) + h(s); g(s)] \\
&> [g^*(s) + h(s); g^*(s)] \\
&> [g^*(s_i) + c^*(s_i, s) + h(s); g^*(s_i) + c^*(s_i, s)] \\
&> [g^*(s_i) + h(s_i); g^*(s_i)] \text{ consistent heuristic} \\
&> [g(s_i) + h(s_i); g(s_i)] \\
&> \textsc{Key}(s_i)
\end{aligned} \tag{3}$$

Thus, we reach a contradiction that $\textsc{Key}(s) \leq \textsc{Key}(u), \forall u \in$ OPEN.

The other case ($v(s) < g(s)$), can be proved in the exact same manner, considering that $\textsc{Key}(s)$ is $[v(s) + h(s); v(s)]$. □

Lemma 6. *In* ComputePath, *any state s selected for expansion/truncation (i.e., if $\textsc{Key}(s) \leq \textsc{Key}(u), \forall u \in$ OPEN at line 18) satisfies, $min(g(s), v(s)) + h(s) \leq g^*(s_{goal})$.*

Proof. We assume that the $g^*(s_{goal}) < \infty$, as otherwise the Lemma holds trivially. We first note that, s_{goal} is never expanded in ComputePath, as whenever s_{goal} has the minimum Key in OPEN, ComputePath terminates (i.e., always $v(s_{goal}) = \infty$). Let us now assume, $g^*(s_{goal}) < min(g(s), v(s)) + h(s)$. At this point, $\textsc{Key}(s) \leq \textsc{Key}(s_{goal})$, as otherwise ComputePath will terminate before s is selected.

Now, let us consider a least cost path from s_{start} to s_{goal} given as $\pi(s_0 = s_{start}, \ldots, s_k = s_{goal})$. If all states s' in this path has $v(s') \leq g^*(s')$, then, $g(s_{goal}) \leq g^*(s_{goal})$ (from Lemma 1), and therefore $\textsc{Key}(s_{goal}) \leq [g^*(s_{goal}); g^*(s_{goal})]$ (as $h(s_{goal}) = 0$). This combined with the fact that $\textsc{Key}(s) \leq \textsc{Key}(s_{goal})$, implies $g^*(s_{goal}) \geq min(g(s), v(s)) + h(s)$.

On the other hand, if there are some state(s) $s' \in \pi(s_0 = s_{start}, \ldots, s_k = s_{goal})$, such that $v(s') > g^*(s')$, we pick the first state s_i with $v(s_i) > g^*(s_i)$. Let us examine this state s_i. If $i = 0$, we have $g(s_0) = g^*(s_0) = 0$, and $v(s_0) > g^*(s_0)$, and if $i > 0$, we have $g(s_i) \leq v(s_{i-1}) + c(s_{i-1}, s_i) \leq g^*(s_{i-1}) + c(s_{i-1}, s_i) \leq g^*(s_i)$. Thus, for s_i, if $v(s_i) > g^*(s_i)$, we have $g(s_i) \leq g^*(s_i)$, which implies $v(s_i) > g(s_i)$ and thus $s_i \in$ OPEN (same as Lemma 5). Now,

$$\begin{aligned}
\textsc{Key}(s_i) &= [g(s_i) + h(s_i); g(s_i)] \\
&\leq [g^*(s_i) + h(s_i); g^*(s_i)] \\
&\leq [g^*(s_i) + c^*(s_i, s_{goal}); g^*(s_i)] \\
&\leq [g^*(s_{goal}); g^*(s_i)]
\end{aligned} \tag{4}$$

Also,

$$\begin{aligned}
& min(g(s), v(s)) + h(s) > g^*(s_{goal}) \\
& \quad \Longrightarrow \ \textsc{Key}(s) > [g^*(s_{goal}); min(g(s), v(s)] \\
& \quad \Longrightarrow \ \textsc{Key}(s) > \textsc{Key}(s_i)
\end{aligned} \tag{5}$$

Therefore, we reach a contradiction to the statement that $\textsc{Key}(s) \leq \textsc{Key}(u), \forall u \in$ OPEN. □

Lemma 7. *In* ComputePath, *for any overconsistent state s ($v(s) \geq g(s)$) selected for expansion (i.e., if $\textsc{Key}(s) \leq \textsc{Key}(u), \forall u \in$ OPEN at line 18), there is a finite cost path from s_{start} to s with cost gc such that $gc + h(s) \leq \epsilon \cdot (g(s) + h(s))$. This path can be constructed using the* ObtainPath *routine if none of back-pointers change.*

Proof. We assume that $g(s) < \infty$, otherwise the statement holds trivially. Suppose we start following the back-pointers starting at s. We need to show that we will reach s_{start} at the cumulative cost of the transitions less than or equal to $\epsilon \cdot g(s) + (\epsilon - 1) \cdot h(s)$.

We first show that we are guaranteed not to encounter an underconsistent state or a state with $bp =$ **null**, that is not truncated. Suppose the back-pointers and the sequence of back-pointer transitions leads us through the states $\{s_0 = s, s_1, \ldots, s_i\}$ where s_i is the first state that is either underconsistent or has $bp(s_i) =$ **null** (or both). It could not have been state s, since $v(s) \geq g(s)$, from the assumptions of the theorem and $g(s) < \infty$ implies $bp(s) \neq$ **null** according to Lemma 1 (except when $s = s_{start}$, in which case the Lemma holds trivially).

We now show that s_i cannot be underconsistent and not truncated by contradiction. Since all the states after s_i are not underconsistent and have defined back-pointer values we have,

$$\begin{aligned} g(s) &= v(s_1) + c(s_1, s) \\ &\geq g(s_1) + c(s_1, s) = v(s_2) + c(s_2, s_1) + c(s_1, s) \\ &\geq \ldots \geq v(s_i) + \Sigma_{k=1..i} c(s_k, s_{k-1}) \\ &\geq v(s_i) + c^*(s_i, s) \end{aligned} \tag{6}$$

Now, considering the key values,

$$\begin{aligned} \textsc{Key}(s_i) &= [v(s_i) + h(s_i); v(s_i)] \\ &\leq [v(s_i) + c^*(s_i, s) + h(s); v(s_i)] \\ &\leq [g(s) + h(s); v(s_i)] \\ &< [g(s) + h(s); g(s)] \\ &< \textsc{Key}(s) \end{aligned} \tag{7}$$

Thus, s_i cannot be in OPEN as $\textsc{Key}(s)$ is the minimum, thus if s_i is underconsistent $s_i \in$ TRUNCATED (from Lemma 2). Also, $bp(s_i)$ cannot be equal to **null** if s_i is overconsistent. From our assumption that $g(s) < \infty$ and the fact that $g(s) \geq v(s_i) + c^*(s_i, s)$ it then follows that $g(s_i)$ is finite. As a result, from Lemma 1, we have $bp(s_i) \neq$ **null** unless $s_i = s_{start}$. On the other hand, if s_i is underconsistent we already have $s_i \in$ TRUNCATED.

We are now ready to show that the cost of the path from s_{start} to s defined by back-pointers is no larger than $\epsilon \cdot g(s) + (\epsilon - 1) \cdot h(s)$. We consider two cases,

1. There are no truncated state encountered by following the back-pointers from s to s_{start}. In this case, let us denote the path (from s_{start}) as: $\{s_0 = s_{start}, s_1, \ldots, s_k = s\}$. Since all states on this path are either consistent or overconsistent and their bp values are defined (except for s_{start}), for any i, $k \geq i > 0$, we have $g(s_i) = v(s_{i-1}) + c(s_{i-1}, s_i) \geq g(s_{i-1}) + c(s_{i-1}, s_i)$ from Lemma 1. For $i = 0$, $g(s_i) = g(s_{start}) = 0$ from the same Lemma. Thus, $g(s) = g(s_k) \geq g(s_{k-1}) + c(s_{k-1}, s_k) \geq g(s_{k-2}) + c(s_{k-2}, s_{k-1}) + c(s_{k-1}, s_k) \ldots \geq \Sigma_{j=1..k} c(s_{j-1}, s_j)$. That is, $g(s)$ is at least as large as the cost of the path from s_{start} to s as defined by the back-pointers, i.e., the path cost $gc = g(s)$.
2. If following the path a truncated state s_s is reached, we denote the path from s_s to s as, $\{s_0 = s_s, s_1, \ldots, s_k = s\}$, Since, all the states in this path are either consistent or overconsistent and their bp values are defined, for any $i, k \geq i > 0$, we have $g(s_i) = v(s_{i-1}) + c(s_{i-1}, s_i) \geq g(s_{i-1}) + c(s_{i-1}, s_i)$ from Lemma 1. Thus, $g(s) = g(s_k) \geq g(s_{k-1}) + c(s_{k-1}, s_k) \geq g(s_{k-2}) + c(s_{k-2}, s_{k-1}) + c(s_{k-1}, s_k) \ldots \geq \Sigma_{j=1..k} c(s_{j-1}, s_j) + v(s_s)$. Now, for $s_s \in$ TRUNCATED we have a stored path $\pi(s_s)$ with cost $g^\pi(s_s) \leq \epsilon \cdot v(s_s) + (\epsilon - 1) \cdot h(s_s)$ (from Lemmas 3, 4 and truncation condition). Thus, for the total path cost (gc), we have

$$\begin{aligned} gc &\leq \Sigma_{j=1..k} c(s_{j-1}, s_j) + \epsilon \cdot v(s_s) + (\epsilon - 1) \cdot h(s_s) \\ &\leq \Sigma_{j=1..k} c(s_{j-1}, s_j) + \epsilon \cdot v(s_s) + (\epsilon - 1) \cdot (h(s) + \Sigma_{j=1..k} c(s_{j-1}, s_j)) \\ &\quad \text{consistent heuristic} \\ &\leq \epsilon \cdot \Sigma_{j=1..k} c(s_{j-1}, s_j) + \epsilon \cdot v(s_s) + (\epsilon - 1) \cdot h(s) \\ &\leq \epsilon \cdot g(s) + (\epsilon - 1) \cdot h(s) \end{aligned} \tag{8}$$

So, in both the cases, $gc + h(s) \leq \epsilon \cdot (g(s) + h(s))$. Also, if none of the back-pointers change the ObtainPath will follow the exact same pointers as used above to construct the path from s_{start} to s, and thus will return a path with cost gc. □

Lemma 8. *In* ComputePath, *for any state s, if the* ComputeGpi(s) *routine returns a path of finite cost, i.e., $g^\pi(s) < \infty$, the same path can be constructed using the* ObtainPath *routine if none of the back-pointers change.*

Proof. The proof can be done using the same logic as Lemma 4. We consider two cases. First, when the ComputeGpi routine does not encounter any truncated states while traversing from s to s_{start} using the back-pointers. As no truncated states are encountered, $g^\pi(s_{goal})$ is computed by adding the action costs from s_{start} to s_{goal} following the bp pointer for each state in the path, and a finite g^π indicates there is no **null** back-pointers or cycles. In the second case, if a state $s' \in$ TRUNCATED is encountered, $g^\pi(s)$ will be incremented using $g^\pi(s')$ and $\pi(s')$ will be used to construct the remaining path. As ObtainPath follows the same back-pointers as ComputeGpi the path returned will be unchanged if none of the back-pointers change. □

Using these Lemmas, we now prove the completeness and bounded suboptimality of TLPA*.

Theorem 1. *When the* ComputePath *function exits, the path from s_{start} to s_{goal}, constructed using* ObtainPath(s_{goal}) *has cost less than or equal to $\epsilon \cdot g^*(s_{goal})$ for a chosen $\epsilon \geq 1.0$.*

Proof. We assume $g^*(s_{goal}) < \infty$, otherwise the theorem is proved trivially.

If ComputePath exits in the line 18, we have $g(s_{goal}) \leq \text{Key}(u), \forall u \in OPEN$ and $v(s_{goal}) \geq g(s_{goal})$. Now, from Lemma 5, we have $g(s_{goal}) \leq g^*(s_{goal})$ and from Lemma 7, we have that ObtainPath(s_{goal}) will return a path of cost $\leq \epsilon \cdot g(s_{goal})$ at this point (note that $h(s_{goal}) = 0$). Thus, the theorem holds.

On the other hand, if ComputePath terminates using the condition in line 21, from Lemma 6 we have, $g^{\pi}(s_{goal}) \leq \epsilon \cdot (min(g(s), v(s)) + h(s)) \leq \epsilon \cdot g^*(s_{goal})$ (as Key(s) is the minimum in OPEN), and from Lemma 8, we have that at this point ObtainPath(s_{goal}) will return a path of cost $g^{\pi}(s_{goal})$. Thus, the theorem holds. □

Next, we include two Theorems, which show that TLPA* retains the efficiency properties of LPA*. We do not discuss the proofs of these theorems as they can be proved in the exact same manner as LPA* [20] (detailed proofs can be found in [67]).

Theorem 2. *In* ComputePath*, no state is expanded more than twice.*

Theorem 3. *A state s is expanded by* ComputePath *only if either it was inconsistent initially or its v value was altered by* ComputePath *at some point during its execution.*

It may be noted that while TLPA* improves the replanning runtime by truncating states, it does not guarantee that the number of state expansions will always be less than or equal to LPA*. TLPA* ensures that the number of underconsistent expansions will not be more than LPA*. However, in a pathological case, it may happen that some overconsistent states gets expanded in TLPA*, which would not be expanded in LPA* (as they will either be made consistent or have Key value greater than $g^*(s_{goal})$, if no state is truncated).

4. Truncated D* Lite

D* Lite [21] is an incremental heuristic search algorithm for goal directed navigation through dynamic graphs that is widely used in real-life robotics applications. In this section, we present Truncated D* Lite (TD* Lite), a bounded suboptimal replanning algorithm for navigation in dynamic graphs, which combines the truncation rules with D* Lite.

4.1. D* Lite

D* Lite (Algorithm 4) repeatedly determines shortest paths between the current position of the robot and the goal state, as the edge costs of a graph change and the robot moves towards the goal state. It is directly based on LPA*, with the following three differences.

Search direction: D* Lite always searches from goal state to start state. LPA* is a forward search algorithm and thus, its g and v values are estimates of the start distances.[1] D* Lite searches backwards, from s_{goal} to s_{start}, so that the root of the search tree remains static when the robot moves. Therefore, in D* Lite, the g and v values are estimates of the goal distances, i.e., for a state s, $g(s)$ denotes the path cost from s to s_{goal}. Similarly, the heuristic values in D* Lite denotes an admissible (and consistent) estimate for the distance between s_{start} and s. We use the notation $h(s_{start}, s)$ instead of $h(s)$ to denote the heuristic for D* Lite. Also, in D* Lite, expansion of a state updates the state's predecessors and not the successors.

Movement of the robot: After computing the shortest path for a given graph, D* Lite moves the robot along the computed path, as long as there is no change in the environment (line 39). If the edge costs change, D* Lite recomputes a shortest path using the latest position of the robot as s_{start}. The navigation terminates when the robot reaches s_{goal}.

Avoiding heap reorder: As D* Lite moves the robot along the path computed, it needs to recalculate the priorities in OPEN when a change in edge costs occurs. Otherwise, **Invariant 3** (as discussed in Section 3.1, page 53) will break, since the priorities are computed with respect to the previous s_{start}. D* Lite avoids repeated reordering by adjusting the Key values following D* [26]. This is achieved in two steps. First, the priorities of the newly generated states are altered using a variable k_m (line 2), which is set to 0 at initialization, and then incremented by $h(s_{last}, s_{start})$ before each call of ComputePath. This optimization ensures that D* Lite does need not to alter the Key values of states that already in OPEN (i.e., no reordering required), and also that the Key values of newly generated states maintain the lower bound on the total path cost (note that, $h(s_{last}, s_{start}) \leq c^*(s_{last}, s_{start})$). Second, within ComputePath, when a state (s) is selected for expansion, its Key is recomputed (line 20). If the current Key value is greater than k_{old}, the state is re-inserted into OPEN with the modified Key. This ensures that a state is only expanded when it's actual priority (Key) is a lower bound of the priority used for ordering (k_{old}), otherwise the state's position is updated according to it's actual priority.

[1] Although, LPA* can also be run as a backward search [66].

Algorithm 4 D* Lite.

```
1: procedure KEY(s)
2:     return [min(g(s), v(s)) + h(s_start, s) + k_m; min(g(s), v(s))];
3: procedure INITSTATE(s)
4:     v(s) = g(s) = ∞; bp(s) = null
5: procedure UPDATESTATE(s)
6:     if s was never visited then
7:         INITSTATE(s)
8:     if s ≠ s_goal then
9:         bp(s) = argmin_{(s'∈Succ(s))} v(s') + c(s, s')
10:        g(s) = v(bp(s)) + c(s, bp(s))
11:    if g(s) ≠ v(s) then
12:        insert/update s in OPEN with KEY(s) as priority
13:    else
14:        if s ∈ OPEN then
15:            remove s from OPEN
16: procedure COMPUTEPATH
17:    while OPEN.MINKEY() < KEY(s_start) ⋁ v(s_start) < g(s_start) do
18:        k_old = OPEN.MINKEY()
19:        s = OPEN.TOP()
20:        if k_old < KEY(s) then
21:            insert s in OPEN with KEY(s) as priority
22:        else
23:            remove s from OPEN
24:            if v(s) > g(s) then
25:                v(s) = g(s)
26:                for all s' ∈ Pred(s) do
27:                    UPDATESTATE(s')
28:            else
29:                v(s) = ∞
30:                UPDATESTATE(s)
31:                for all s' ∈ Pred(s) do
32:                    UPDATESTATE(s')
33: procedure MAIN
34:    INITSTATE(s_start); INITSTATE(s_goal)
35:    g(s_goal) = 0; k_m = 0; s_last = s_start; OPEN ← ∅
36:    Insert s_goal in OPEN with KEY(s_goal)
37:    COMPUTEPATH
38:    while s_last ≠ s_start do
39:        s_start = bp(s_start); Move to s_start;
40:        Scan graph for changes in edge costs
41:        if any edge costs changed then
42:            k_m = k_m + h(s_last, s_start); s_last = s_start
43:            for all directed edges (u, v) with changed edge costs do
44:                update edge cost c(u, v)
45:                UPDATESTATE(u)
46:            COMPUTEPATH
```

4.2. Truncated D* lite algorithm

The first thing we note is that the truncation rules described in Section 3.2 are completely orthogonal to the optimizations used in D* Lite. Therefore, these rules can be directly applied to D* Lite. The TD* Lite algorithm (Algorithm 5)[2] precisely does that. It inherits all the D* Lite characteristics, namely, backward search, robot movement and avoidance of heap reorder; and augments it with the application of the truncation rules.

The only significant change from TLPA* (other than searching backward) is the lower bound correction required for the optimization to avoid heap reorder (lines 18–21). In TD* Lite, the truncation rules are applied only when a state selected for expansion satisfies the lower bound (check at line 20), if not, the state is re-inserted in OPEN after updating its KEY. This optimization is the same as used D* Lite, which ensures that we need not reorder the heap every time there is a change of cost and yet maintain the order invariants.

While the application of truncation rules in TD* Lite is exactly the same as in TLPA*, the description of Rule 2 needs to be modified as the $\text{KEY}_1(s)$ values for TD* Lite are different from the values in TLPA*, due to the addition of k_m. For TD* Lite, Truncation Rule 2 is applied if a state s selected for expansion, satisfies $g^{\pi}(s_{start}) \leq \epsilon \cdot (min(g(s), v(s)) + h(s_{start}, s))$ (line 24).

TD* Lite guarantees the same properties as TLPA*, i.e., it always returns a provably bounded suboptimal solution, and no state is expanded more than twice in COMPUTEPATH. These properties can be derived in exactly the same way as done for TLPA*.

[2] The auxiliary routines are the same as Algorithm 2, the only difference is that the search is backwards here, so s_{start} and s_{goal} should exchange positions.

Algorithm 5 TD* Lite.

```
1: procedure KEY(s)
2:     return [min(g(s), v(s)) + h(s_start, s) + k_m; min(g(s), v(s))];
3: procedure INITSTATE(s)
4:     v(s) = g(s) = g^π(s) = ∞; bp(s) = π(s) = null
5: procedure UPDATESTATE(s)
6:     if s was never visited then
7:         INITSTATE(s)
8:     if s ≠ s_goal ⋀ s ∉ TRUNCATED then
9:         bp(s) = argmin_{(s'∈Succ(s))} v(s') + c(s, s')
10:        g(s) = v(bp(s)) + c(s, bp(s))
11:        if g(s) ≠ v(s) then
12:            insert/update s in OPEN with KEY(s) as priority
13:        else
14:            if s ∈ OPEN then
15:                remove s from OPEN
16: procedure COMPUTEPATH
17:    while OPEN.MINKEY() < KEY(s_start) ⋁ v(s_start) < g(s_start) do
18:        k_old = OPEN.MINKEY()
19:        s = OPEN.TOP()
20:        if k_old < KEY(s) then
21:            insert s in OPEN with KEY(s) as priority
22:        else
23:            COMPUTEGPI(s_start)
24:            if g^π(s_start) ≤ ϵ · (min(g(s), v(s)) + h(s_start, s)) then
25:                Terminate
26:            remove s from OPEN
27:            if v(s) > g(s) then
28:                v(s) = g(s)
29:                for all s' ∈ Pred(s) do
30:                    UPDATESTATE(s')
31:            else
32:                COMPUTEGPI(s)
33:                if g^π(s) + h(s) ≤ ϵ · (v(s) + h(s_start, s)) then
34:                    insert s in TRUNCATED
35:                else
36:                    v(s) = ∞
37:                    UPDATESTATE(s)
38:                    for all s' ∈ Pred(s) do
39:                        UPDATESTATE(s')
40: procedure MAIN
41:    INITSTATE(s_start); INITSTATE(s_goal)
42:    g(s_goal) = g^π(s_goal) = 0; k_m = 0; s_last = s_start
43:    OPEN ← ∅; TRUNCATED ← ∅
44:    Insert s_goal in OPEN with KEY(s_goal)
45:    COMPUTEPATH; OBTAINPATH(s_start)
46:    while s_last ≠ s_start do
47:        s_start = bp(s_start); Move to s_start;
48:        Scan graph for changes in edge costs
49:        if any edge costs changed then
50:            k_m = k_m + h(s_last, s_start); s_last = s_start
51:            for all directed edges (u, v) with changed edge costs do
52:                update edge cost c(u, v)
53:                UPDATESTATE(u)
54:            for all s ∈ TRUNCATED do
55:                remove s from TRUNCATED; g^π(s) = ∞; π(s) ← null
56:                UPDATESTATE(v)
57:            COMPUTEPATH; OBTAINPATH(s_start)
```

5. Anytime Truncated D*

In this section, we discuss the modifications of the truncation rules required to make them work with inflated heuristics and show how these rules can be integrated with Anytime D* [23] algorithm to develop a truncation based anytime replanning algorithm. The motivation for combining AD* with truncated searches (TLPA*/TD* Lite) can be summarized from the following observations,

- While both AD* and TLPA* provides bounded suboptimal solutions for replanning, algorithmically they adopt different directions, and thus can provide complimentary benefits (AD* uses heuristic inflation, whereas TLPA* uses truncation).
- In AD*, inflated heuristics are only used for overconsistent states, while underconsistent use uninflated heuristics. At times, this results in accumulation of underconsistent states (in OPEN), causing performance deterioration [23]. Truncation is especially powerful for handling underconsistent states (Rule 1), as it can restrict cost propagations for majority of such states (if a *good enough* path to it has been discovered).

Algorithm 6 Anytime D*.

```
1: procedure KEY(s)
2:     if v(s) ≥ g(s) then
3:         return [g(s) + ε · h(s); g(s)];
4:     else
5:         return [v(s) + h(s); v(s)];
6: procedure INITSTATE(s)
7:     v(s) = g(s) = ∞; bp(s) = null
8: procedure UPDATESTATE(s)
9:     if s was never visited then
10:        INITSTATE(s)
11:    if s ≠ s_start then
12:        bp(s) = argmin_{(s'∈Pred(v))} v(s') + c(s', s)
13:        g(s) = v(bp(s)) + c(bp(s), s)
14:        if g(s) ≠ v(s) then
15:            if s ∉ CLOSED then
16:                insert/update s in OPEN with KEY(s) as priority
17:            else
18:                if s ∉ INCONS then
19:                    insert s in INCONS
20:        else
21:            if s ∈ OPEN then
22:                remove s from OPEN
23:            else
24:                if s ∈ INCONS then
25:                    remove s from INCONS
26: procedure COMPUTEPATH
27:    while OPEN.MINKEY() < KEY(s_goal) ⋁ v(s_goal) < g(s_goal) do
28:        s = OPEN.TOP()
29:        remove s from OPEN
30:        if v(s) > g(s) then
31:            v(s) = g(s)
32:            insert s in CLOSED
33:            for all s' ∈ Succ(s) do
34:                UPDATESTATE(s')
35:        else
36:            v(s) = ∞
37:            UPDATESTATE(s)
38:            for all s' ∈ Succ(s) do
39:                UPDATESTATE(s')
40: procedure MAIN
41:    INITSTATE(s_start); INITSTATE(s_goal)
42:    g(s_start) = 0; ε = ε_0;
43:    OPEN ← ∅; CLOSED ← ∅
44:    INCONS ← ∅
45:    Insert s_start in OPEN with KEY(s_start)
46:    COMPUTEPATH
47:    while 1 do
48:        if changes in edge costs are detected then
49:            for all directed edges (u, v) with changed edge costs do
50:                update edge cost c(u, v)
51:                UPDATESTATE(v)
52:        if ε > 1.0 then
53:            decrease ε
54:        move states from INCONS to OPEN
55:        update state priorities with current ε
56:        CLOSED ← ∅
57:        COMPUTEPATH
58:        if ε = 1.0 then
59:            wait for changes in edge costs
```

- Performance of inflated heuristic searches like AD* can degrade considerably in presence of local minima [68,69], due to its heuristic driven greediness. Truncation based algorithms do not rely on inflation, and thus do not suffer from this problem.

5.1. Anytime D*

Anytime D* (AD*) is anytime search algorithm for dynamic graphs that combines the ARA* [5] with LPA*/D* Lite [20,21]. Following ARA*, AD* performs a series of searches with decreasing inflation factors, to iteratively generate solutions with improved bounds. If there is a change in the environment, AD* puts the resulting inconsistent states into OPEN and recomputes a bounded suboptimal path by propagating the cost changes, following LPA*/D* Lite.

The pseudocode for a basic version of AD* is shown in Algorithm 6. The MAIN function first sets the inflation factor to a chosen ϵ_0, usually a high value, and generates an initial plan. Then, unless changes in edge costs are detected, it decreases

the ϵ bound, and improves the quality of its solution by repeatedly executing ComputePath, until an optimal solution is found. This part of AD* is an exact match with ARA* [5].

When changes in edge costs are detected, AD* updates the costs of affected states following **Invariant 1** and puts the resulting inconsistent states in OPEN (**Invariant 2**). It then calls ComputePath to find a new solution. The state expansions and cost updates in AD* are similar to LPA*/D* Lite, however, it handles the priorities in a different way. AD* uses inflated heuristic values for overconsistent (and consistent) states (line 3) to provide a depth-first bias to the search. Whereas, for underconsistent states, it uses uninflated heuristic values (line 5), in order to ensure that underconsistent states correctly propagate their new costs to the affected successors. By incorporating this dual mechanism, AD* can handle arbitrary changes in the edge costs as well as changes to the inflation factor within a single algorithmic framework.

In AD*, each call of ComputePath guarantees an ϵ suboptimal solution [23]. It also has the same efficiency properties as LPA*, i.e., no state is expanded more than twice within ComputePath and consistent states are not re-expanded.

5.2. Truncation rules with inflated heuristics

AD* and TLPA* use orthogonal approaches to compute bounded suboptimal solutions. In AD*, the path estimates are guaranteed to be within the chosen bound of g^*, while the actual path cost is guaranteed to be less than or equal to the estimate [23]. Whereas in TLPA*, the path estimates are always a lower bound on g^*, while the actual path costs lie within the chosen bound of this estimate (Lemmas 5, 7, and 8). Thus, it may seem that we can combine these two approaches seamlessly using two factors (say ϵ_1 and ϵ_2), one for heuristic inflation and another for truncation. If heuristic inflation ensures that the path estimates are always within ϵ_1 bound of the optimal path cost and the truncation rules ensure that the actual path costs are within ϵ_2 bound of the path estimates, we can guarantee that the solution cost will be within $\epsilon_1 \cdot \epsilon_2$ of the optimal solution cost.

Unfortunately, this only works for the truncation of overconsistent states, and not for underconsistent states. This is due to the fact that in AD*, heuristic values for overconsistent states are inflated, whereas an underconsistent state uses an uninflated heuristic. Therefore, when an overconsistent s_1 is selected for expansion (in AD*), we have $g(s_1) \leq \epsilon_1 \cdot g^*(s_1)$, but when an underconsistent state (s_2) is selected for expansion, there is no guarantee that $v(s_2) \leq \epsilon_1 \cdot g^*(s_2)$. Now, if we apply the truncation rule (say Truncation Rule 1) to a state s having $v(s) \geq \epsilon_1 \cdot g^*(s)$, the total path cost $C \leq \epsilon_2 \cdot (v(s) + h(s))$ does not ensure that $C \leq \epsilon_1 \cdot \epsilon_2 \cdot (g^*(s) + h(s))$. Therefore, in such a case, we cannot guarantee bounded suboptimal termination.

In Fig. 5, we present an example bound violation, when we simultaneously use heuristic inflation and truncation. Here, we use $\epsilon_1 = 2.0$ and $\epsilon_2 = 1.25$ (target bound 2.5). Fig. 5a shows the first iteration where a solution of cost 155 is obtained. After the first iteration, the cost of the edge from S to B changes from 50 to 0 and the cost of edge from A to D changes from 20 to 24, making B overconsistent and D underconsistent. Therefore, in AD*, $\text{Key}_1(B) = g(B) + \epsilon_1 \cdot h(B) = 0 + 2.0 \cdot 60 = 120$ and $\text{Key}_1(D) = v(D) + h(D) = 50 + 45 = 95$, i.e., D is selected before B for expansion (as shown in Fig. 5b). Now, when D is selected for expansion, we have $g^{\pi}(D) = 54$ (path shown using dashed arrows in Fig. 5c). As, $g^{\pi}(D) + h(D) = 54 + 45 = 99$ and $v(D) + h(D) = 95$, we have $g^{\pi}(D) + h(D) \leq 1.25 \cdot (v(D) + h(D))$, and thus, D is truncated using Rule 1 (Fig. 5c). In the next step, when B is expanded, a *better* path to D through B is discovered. However, as D is already truncated, this *new* path information is not propagated. Therefore, the second iteration ends after the expansion of B ($\text{Key}_1(G)$ is the minimum Key in OPEN), returning the old path as the solution (shown using dashed arrows, Fig. 5d). Obviously, this solution violates the bounds as the current path cost (155) is more than 2.5 times the optimal path cost, which is 60 (path shown by grey arrows in Fig. 5d).[3]

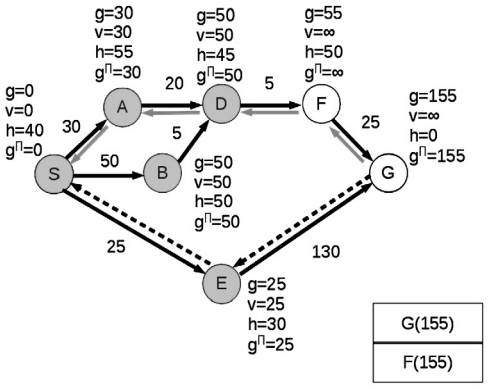
From the example, we observe that if we combine the truncation rules from Section 3.2 with AD*, we may end up violating the bounds. To overcome this problem, we propose a two-step method for truncating underconsistent states.

Truncation Rule 1 As noted in Section 3.2, Rule 1 is applicable for underconsistent states only. TLPA* truncates the cost propagation for an underconsistent state s (selected for expansion), if $g^{\pi}(s) + h(s) \leq \epsilon \cdot (v(s) + h(s))$. In ATD*, when an underconsistent state s is selected for expansion for the first time, we compute its g^{π} value and check whether $g^{\pi}(s) + h(s) \leq \epsilon_2 \cdot (v(s) + h(s))$, in the same manner as TLPA*. However, we do not truncate s immediately if this check is true. Instead, we mark s as a state that can be potentially truncated (by inserting it in a list called MARKED), postpone its cost propagation, and update its position in OPEN by altering its Key value from $\text{Key}(s) = [v(s) + h(s); v(s)]$ to $\text{Key}(s) = [v(s) + \epsilon_1 \cdot h(s); v(s)]$ (**Step 1**).

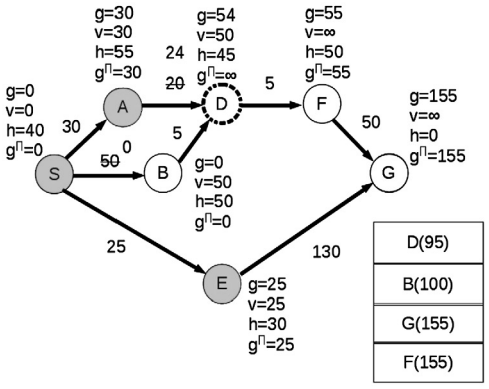
If a state $s \in$ MARKED, is selected for expansion again as an underconsistent state ($v(s) < g(s)$), i.e., it is selected for expansion with the inflated heuristic Key, we truncate s (**Step 2**).

Using this two-step policy, on one hand we ensure that we do not propagate cost changes for an underconsistent state (s) when it has already discovered a *good enough* path from s_{start}. On the other hand, we cover for the fact that at this point $v(s)$ may be more than $\epsilon_1 \cdot g^*(s)$. The updated $\text{Key}(s)$ guarantees that if s is selected again for expansion as an

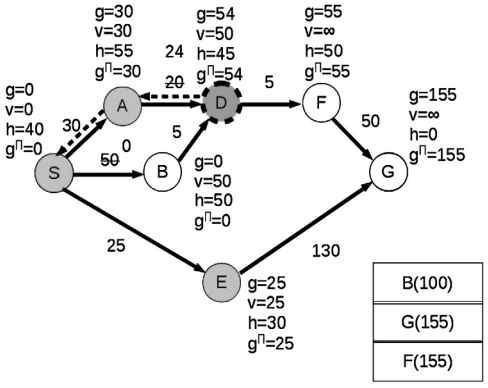
[3] It should be noted that this dual handling of the keys does not violate the suboptimality guarantee of AD*, as AD* forces an underconsistent state to become overconsistent by making $v(s) = \infty$, and if s is later expanded as an overconsistent state, $g(s) \leq \epsilon_1 \cdot g^*(s)$ is guaranteed. In the example if when $v(D)$ is set to ∞, the relative position of B and D in *OPEN* will change and the new path cost will be propagated correctly. In other words, AD* will propagate the costs correctly through expansions of D (underconsistent), B (overconsistent) and D (overconsistent), in that order.



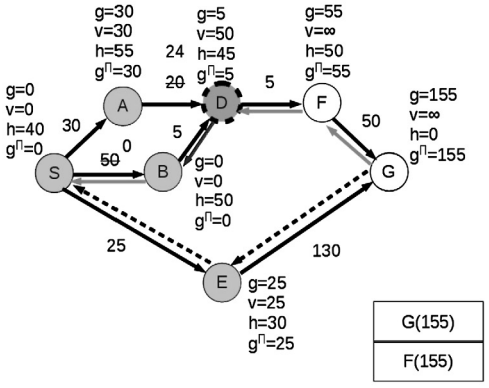
(a) First search



(b) After cost change, underconsistent state D, overconsistent state B



(c) D comes for expansion, $g^{\pi}(D) + h(D) \leq \epsilon_2 \cdot (v(D) + h(D))$, we truncate D



(d) B expanded but as D is truncated before the path information is not propagated, causing bound violation

Fig. 5. A simple example depicting how a *direct* combination of truncation and heuristic inflation may cause bound violations. We choose $\epsilon_1 = 2.0$ and $\epsilon_2 = 1.25$, i.e., the target bound is 2.5. In addition to g, v and g^{π} values, for this case, we also show the (consistent) heuristic values for each state. The boxes in the right hand show the current state of OPEN (KEY values for each state shown in brackets).

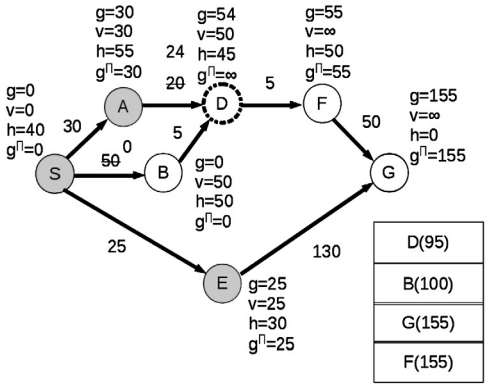
underconsistent state, we have $v(s) \leq \epsilon_1 \cdot g^*(s)$ and thus s can be truncated without violating the bounds. Otherwise, if $v(s) > \epsilon_1 \cdot g^*(s)$, s will be converted to an overconsistent state before it is selected for expansion again.

In Fig. 6, we use the earlier example (Fig. 5) to show that how such a two-step truncation policy guarantees the sub-optimality bounds. Similar to the earlier case, here also we set $\epsilon_1 = 2.0$ and $\epsilon_2 = 1.25$. After the first run, cost of the edge from S to B decreases from 50 to 0, whereas cost of the edge from S to A increases from 20 to 24 (Fig. 6a). The initial positions of B and D (in OPEN) are identical to Fig. 5b. Now, when D is selected for expansion (before B) in Fig. 6b, we compute $g^{\pi}(D)$ and note that it satisfies the truncation condition. However, unlike Fig. 5c, here we alter the KEY of D (from 95 to 140) and reinsert it in OPEN (Fig. 6b). Next, when B expands, it propagates the *new* path information to D converting it to an overconsistent state (Fig. 6c). This cost change information is then propagated to F and G (through state expansions), computing the new solution (with cost 60) guaranteed to be within the 2.5 bound (Fig. 6d). Next, we formally state Truncation Rule 1 for ATD*.

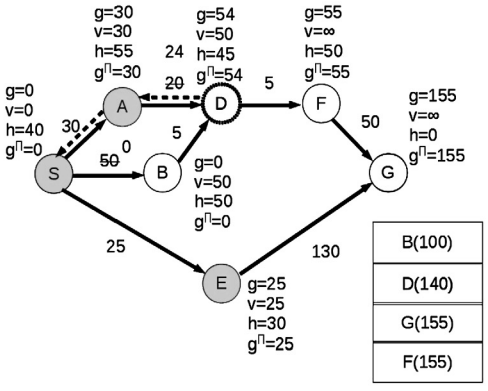
ATD* Rule 1. *An underconsistent state s having* $\text{KEY}(s) \leq \text{KEY}(u), \forall u \in \text{OPEN}$ *and* $s \notin \text{MARKED}$ *is inserted in* MARKED *if* $g^{\pi}(s) + h(s) \leq \epsilon_2 \cdot (v(s) + h(s))$*, its* KEY *is changed to* $\text{KEY}(s) = [v(s) + \epsilon_1 \cdot h(s); v(s)]$*, and its position in* OPEN *is updated using the changed* KEY*.*

An underconsistent state $s \in$ MARKED with $\text{KEY}(s) \leq \text{KEY}(u), \forall u \in \text{OPEN}$ *is truncated (removed from* OPEN *without expansion).*

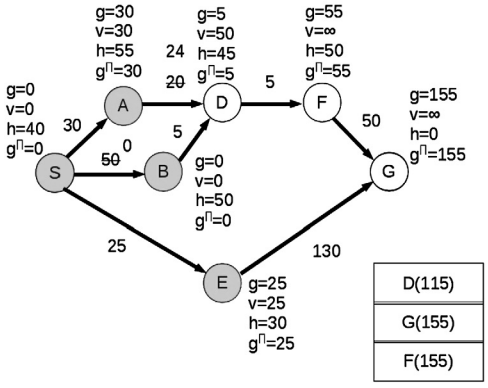
Truncation Rule 2 Truncation Rule 2 (as described in Section 3.2) is applicable for both underconsistent and overconsistent states. In ATD*, for an overconsistent state s, we apply this rule in unchanged manner, as for an overconsistent state $g(s) \leq \epsilon_1 \cdot g^*(s)$ is guaranteed. However, for an underconsistent state s, we apply Rule 2 only when it has earlier been marked ($s \in$ MARKED), i.e., it has been selected for expansion with the modified KEY ($\text{KEY}_1(s) = v(s) + \epsilon_1 \cdot h(s)$), as otherwise the bounds can be violated in a manner similar as discussed for Rule 1. For ATD*, Truncation Rule 2 is formulated in the following statement.



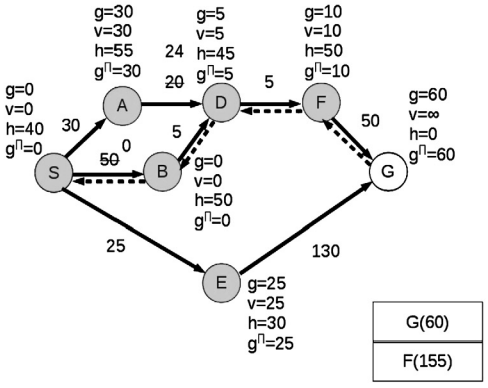
(a) After cost change, underconsistent state D, overconsistent state B



(b) D comes for expansion, $g^{\pi}(D) + h(D) \leq \epsilon_2 \cdot (v(D)+h(D))$, we mark D, and reinsert it in OPEN with altered Key



(c) B expanded, D gets converted to an overconsistent state



(d) New solution computed by expanding D and E

Fig. 6. Example of the two-step truncation policy for inflated heuristic replanning. The planning problem is the same as included Fig. 5.

ATD* Rule 2. *A state s having* $\textsc{Key}(s) \leq \textsc{Key}(u), \forall u \in$ *OPEN is truncated if* $g^{\pi}(s_{goal}) \leq \epsilon_2 \cdot \textsc{Key}_1(s)$ *and if either* $v(s) > g(s)$ *or* $s \in$ *MARKED.*

Also, if any state s is truncated using Rule 2, all states $s' \in$ *OPEN are truncated as* $\forall s' \in$ *OPEN,* $\textsc{Key}_1(s') \geq \textsc{Key}_1(s)$.

5.3. Anytime Truncated D* algorithm

Anytime Truncated D* (ATD*) augments the anytime replanning approach of AD* using the two truncation rules described in Section 5.2. It uses two constants, ϵ_1 to inflate the heuristics and ϵ_2 for truncation, and guarantees that ComputePath will return a solution within $\epsilon_1 \cdot \epsilon_2$ factor of the optimal solution cost. We include the pseudocode for ATD* in Algorithms 7 and 8.

ATD* uses an extra list MARKED to perform the two-step truncation. The auxiliary functions (ComputeGpi and ObtainPath) are the same as shown in Algorithm 2. However, the Key function and the truncation rules are different. The Key in ATD* provides different values depending on $v(s)$ and $g(s)$, for $v(s) \geq g(s)$ it is same at AD*, whereas for $v(s) < g(s)$, it uses heuristic inflation if $s \in$ MARKED, otherwise it uses an uninflated heuristic. For an underconsistent state, truncation is done in two phases (lines 43–59, Algorithm 7), whereas for an overconsistent state, it is same as TLPA*. Note that, ATD* does not re-expand an overconsistent state (i.e., a state $s \in$ CLOSED) within the ComputePath routine. If a better path to such a state s is discovered, s is put into INCONS instead. These states are put back to OPEN before the next call of ComputePath.

The Main function for ATD* repeatedly calls ComputePath either to compute a new solution (for the first time, or when the edge costs change) or to improve an incumbent solution (after updating the bounds). After each call of ComputePath, it processes the states in lists MARKED, TRUNCATED and INCONS in an efficient manner to ensure minimal re-expansions. If the edge costs do not change, it reuses the stored paths for truncated states if they still satisfy the truncation rules with the new bounds (lines 22 and 26, Algorithm 8). The states that satisfy the current bounds are inserted to both MARKED and INCONS, others are only inserted in INCONS. All the states in INCONS are then merged with OPEN. If the edge costs change,

Algorithm 7 Anytime Truncated D* (part 1).

```
1: procedure KEY(s)
2:     if v(s) ≥ g(s) then
3:         return [g(s) + ε1 · h(s); g(s)];
4:     else
5:         if s ∈ MARKED then
6:             return [v(s) + ε1 · h(s); v(s)];
7:         else
8:             return [v(s) + h(s); v(s)];
9: procedure INITSTATE(s)
10:    v(s) = g(s) = g^π(s) = ∞; bp(s) = π(s) = null;
11: procedure UPDATESTATE(s)
12:    if s was never visited then
13:        INITSTATE(s)
14:    if s ≠ s_start ⋀ s ∉ TRUNCATED then
15:        bp(s) = argmin_{(s'∈Pred(s))} v(s') + c(s', s)
16:        g(s) = v(bp(s)) + c(bp(s), s)
17:        if g(s) ≠ v(s) then
18:            if s ∉ CLOSED then
19:                insert/update s in OPEN with KEY(s) as priority
20:            else
21:                if s ∉ INCONS then
22:                    insert s in INCONS
23:        else
24:            if s ∈ OPEN then
25:                remove s from OPEN
26:            else
27:                if s ∈ INCONS then
28:                    remove s from INCONS
29: procedure COMPUTEPATH
30:    while OPEN.MINKEY() < KEY(s_goal) ⋁ v(s_goal) < g(s_goal) do
31:        s = OPEN.TOP()
32:        if v(s) > g(s) then
33:            if s ∈ MARKED then
34:                remove s from MARKED
35:            COMPUTEGPI(s_goal)
36:            if g^π(s_goal) ≤ ε2 · (g(s) + ε1 · h(s)) then
37:                Terminate
38:            remove s from OPEN
39:            v(s) = g(s); insert s in CLOSED
40:            for all s' ∈ Succ(s) do
41:                UPDATESTATE(s')
42:        else
43:            if s ∈ MARKED then
44:                COMPUTEGPI(s_goal)
45:                if g^π(s_goal) ≤ ε2 · (v(s) + ε1 · h(s)) then
46:                    Terminate
47:                else
48:                    remove s from MARKED and OPEN
49:                    insert s in TRUNCATED
50:            else
51:                COMPUTEGPI(s)
52:                if g^π(s) + h(s) ≤ ε2 · (v(s) + h(s)) then
53:                    insert s in MARKED
54:                    UPDATESTATE(s)
55:                else
56:                    v(s) = ∞
57:                    UPDATESTATE(s)
58:                    for all s' ∈ Succ(s) do
59:                        UPDATESTATE(s')
```

the states in TRUNCATED are reevaluated, as their old estimates may no longer remain correct, and resulting inconsistent states are put into OPEN following the same invariants as TLPA*.

5.4. Theoretical properties of ATD*

In this section, we discuss some important theoretical properties of ATD*. Detailed proofs of these properties (and several others) are included in [70]. In general, the properties of ATD* are similar to that of TLPA* and AD*. Therefore, most of the following results can be proved in the same way as done for TLPA* (Section 3.4). We explain the cases where the results differ.

Lemma 9. *In ATD*, all v and g values are non-negative,* $bp(s_{start}) = \textbf{null}$, $g(s_{start}) = 0$, *and* $\forall s \neq s_{start}$, $bp(s) = argmin_{(s' \in Pred(s))}\, v(s') + c(s', s)$ *and* $g(s) = v(bp(s)) + c(bp(s), s)$ *(same as Lemma 1).*

Algorithm 8 Anytime Truncated D* (part 2).

```
 1: procedure Main
 2:     InitState(s_start); InitState(s_goal)
 3:     InitBounds
 4:     OPEN ← ∅; CLOSED ← ∅
 5:     INCONS ← ∅; MARKED ← ∅
 6:     g(s_start) = 0; insert s_start in OPEN with Key(s_start)
 7:     ComputePath; ObtainPath(s_goal)
 8:     while 1 do
 9:         if changes in edge costs are detected then
10:             for all directed edges (u, v) with changed edge costs do
11:                 update edge cost c(u, v)
12:                 UpdateState(v)
13:             for all s ∈ TRUNCATED do
14:                 remove s from TRUNCATED; g^π(s) = ∞; π(s) ← null
15:                 UpdateState(v)
16:             remove states from MARKED; MARKED ← ∅
17:         UpdateBounds
18:         if ε1 = 1.0 ⋀ ε2 = 1.0 then
19:             wait for changes in edge costs
20:         else
21:             for all s ∈ MARKED do
22:                 if g^π(s) + h(s) > ε2 · (v(s) + h(s)) then
23:                     remove s from MARKED
24:             for all s ∈ TRUNCATED do
25:                 remove s from TRUNCATED; insert s in INCONS;
26:                 if g^π(s) + h(s) ≤ ε2 · (v(s) + h(s)) then
27:                     insert s in MARKED;
28:             move states from INCONS to OPEN
29:             update state priorities with current ε1
30:             CLOSED ← ∅
31:             ComputePath; ObtainPath(s_goal)
```

Lemma 10. *In ATD*, OPEN, INCONS and TRUNCATED are disjoint and OPEN ∪ INCONS ∪ TRUNCATED contains all the inconsistent states (i.e., CLOSED contains no inconsistent states). Also, before each call of* ComputePath, *OPEN contains all the inconsistent states and both TRUNCATED and INCONS are empty.*

Proof. This can be proved in the same way as Lemma 2. At the start, OPEN contains all the inconsistent states. Later, when an overconsistent state is expanded, it is made consistent, removed from OPEN and added to CLOSED. If that state is made inconsistent again (decrease in g), it is added to INCONS. On the other hand, an underconsistent state is either expanded or truncated, satisfying the Lemma statement in both the cases. When ComputePath terminates, all states from TRUNCATED are processed through UpdateState if there is a cost change, otherwise they are added to INCONS. In both the cases, the states in INCONS are moved to OPEN. □

Lemma 11. *If a state s is truncated ($s \in$ TRUNCATED), $v(s)$ is never altered during the execution of* ComputePath *(same as Lemma 3).*

Lemma 12. *In* ComputePath, *$\forall s \in$ TRUNCATED, there is a finite cost path from s_{start} to s with cost $g^{\pi}(s)$, stored in $\pi(s)$ (same as Lemma 4).*

Lemma 13. *In* ComputePath, *any state s selected for expansion/truncation (i.e., if $\text{Key}(s) \leq \text{Key}(u), \forall u \in$ OPEN at line 30, Algorithm 8) satisfies, $g(s) \leq \epsilon_1 \cdot g^*(s)$ if $v(s) \geq g(s)$ and $v(s) \leq \epsilon_1 \cdot g^*(s)$ if $v(s) < g(s)$ and $s \in$ MARKED.*

Proof. While this can be proved in the same manner as Lemma 5, we may note two significant differences. First, the g and v values are now within ϵ_1 bound of the optimal path cost (g^*). This stems from the use of inflated heuristic in ATD*. Therefore, in the proof, Equation (3) should be replaced by the following,

$$\begin{aligned}
\text{Key}(s) &= [g(s) + \epsilon_1 \cdot h(s); g(s)] \\
&> [\epsilon_1 \cdot g^*(s) + \epsilon_1 \cdot h(s); \epsilon_1 \cdot g^*(s)] \\
&> [\epsilon_1 \cdot g^*(s_i) + \epsilon_1 \cdot c^*(s_i, s) + \epsilon_1 \cdot h(s); \epsilon_1 \cdot g^*(s_i) + \epsilon_1 \cdot c^*(s_i, s)] \\
&> [\epsilon_1 \cdot g^*(s_i) + \epsilon_1 \cdot h(s_i); \epsilon_1 \cdot g^*(s_i)] \text{ consistent heuristic} \\
&> [g(s_i) + \epsilon_1 \cdot h(s_i); g(s_i)] \text{ choice of } s_i \text{ ensure } g(s_i) \leq \epsilon_1 \cdot g^*(s_i) \\
&> \text{Key}(s_i) \qquad (9)
\end{aligned}$$

More importantly, the bound on v values is only true for underconsistent states in MARKED. The states that are not in MARKED use an uninflated heuristic, i.e., their Key is given by $[v(s) + h(s); v(s)]$, which will not satisfy the inequalities

used in Equation (9). However, for a state $s \in$ MARKED, KEY(s) is modified to $[v(s) + \epsilon_1 \cdot h(s); v(s)]$ which ensures that the equation and subsequently the bound is satisfied. □

Lemma 14. *In* COMPUTEPATH*, any state s selected for expansion/truncation (i.e., if* KEY$(s) \leq$ KEY$(u), \forall u \in$ *OPEN at line 30, Algorithm 8) satisfies,* $min(g(s), v(s)) + \epsilon_1 \cdot h(s) \leq \epsilon_1 \cdot g^*(s_{goal})$ *if either* $v(s) \geq g(s)$ *or* $s \in$ *MARKED.*

Proof. Again, the proof is very similar to Lemma 6. However, note the extra qualification required for underconsistent states, due to the differential handling of KEY values. □

Lemma 15. *In* COMPUTEPATH*, for any overconsistent state s (*$v(s) \geq g(s)$*) selected for expansion (i.e., if* KEY$(s) \leq$ KEY$(u), \forall u \in$ *OPEN at line 30, Algorithm 8), there is a finite cost path from* s_{start} *to s with cost gc such that* $gc + h(s) \leq \epsilon_2 \cdot (g(s) + h(s))$*. This path can be constructed using the* OBTAINPATH *routine if none of back-pointers change.*

Proof. The proof is same as Lemma 7, other than the fact the ϵ in Lemma 7 is now replaced by ϵ_2. □

Lemma 16. *In* COMPUTEPATH*, for any state s, if the* COMPUTEGPI(s) *routine returns a path of finite cost, i.e.,* $g^{\pi}(s) < \infty$*, the same path can be constructed using the* OBTAINPATH *routine if none of the back-pointers change (same as Lemma 8).*

Now, we state the correctness property of ATD*.

Theorem 4. *When the* COMPUTEPATH *function exits, the path from* s_{start} *to* s_{goal}*, constructed using* OBTAINPATH(s_{goal}) *has cost* $\leq \epsilon_1 \cdot \epsilon_2 \cdot g^*(s_{goal})$.

Proof. We assume $g^*(s_{goal}) < \infty$, otherwise the theorem is proved trivially. The proof is very similar to Theorem 1. If COMPUTEPATH exits in the line 30, we have $g(s_{goal}) \leq$ KEY$(u), \forall u \in$ *OPEN* and $v(s_{goal}) \geq g(s_{goal})$. Now, from Lemma 13, we have $g(s_{goal}) \leq \epsilon_1 \cdot g^*(s_{goal})$ and from Lemma 15, we have that OBTAINPATH(s_{goal}) will return a path of cost $\leq \epsilon_2 \cdot g(s_{goal})$. Thus, the theorem holds.

On the other hand, if COMPUTEPATH terminates using the conditions in lines 36 or 45, from Lemma 14, we have, $g^{\pi}(s_{goal}) \leq \epsilon_2 \cdot (min(g(s), v(s)) + \epsilon_1 \cdot h(s)) \leq \epsilon_2 \cdot \epsilon_1 \cdot g^*(s_{goal})$ (as KEY(s) is the minimum in OPEN), and from Lemma 16, we have that at this point ObtainPath(s_{goal}) will return a path of cost $g^{\pi}(s_{goal})$. Thus, the theorem holds. □

The efficiency theorems (Theorems 2 and 3), are also valid for ATD*, i.e., it never expands a state more than twice in COMPUTEPATH, neither does it expand a consistent state.

6. Experimental results

We empirically evaluate the algorithms presented in this paper for two planning domains, 2D 16-connected grids for a point robot and 3D (x, y, heading) state lattices for PR2 robot base.[4] For each domain, we generate two kinds of maps. Indoor maps, that are composed of randomly placed narrow hallways and large rooms with obstacles of various shapes, and outdoor maps, that have large open spaces with random convex obstacles. In each case, we compile a test suite of 100 maps of 1000×1000 size, with the start and goal states randomly chosen on the border. Note that these environments provide different levels of difficulties in planning [69]. Outdoor planning is relatively easier, due to high solution density and absence of large local minima. In contrast, for indoor maps, the presence of large halls and narrow pathways often create large local minima, which coupled with lower solution density makes planning more challenging. For 2D grids, we use Euclidean distance as a consistent heuristic, and for 3D lattices, we compute a consistent heuristic by running a 2D Dijkstra search after inflating the objects using the robot's (PR2 base) in-radius [69].

6.1. Planning in dynamic graphs: TLPA*

In this section, we compare TLPA* with a set of optimal (A* [72], LPA* [20], GAA* [32] and MPGAA* [35]) and bounded suboptimal (WA* [45] without re-expansion and GLPA* [66]) search algorithms, for planning in dynamic graphs.

For the first experiment, after each planning episode, we randomly change the traversability of some of the cells in the map from blocked to unblocked and an equal number of cells from unblocked to blocked, and replan. The percentage of cells blocked/unblocked per iteration is input to the experiment (change rate). We iterate the replanning loop for 100 times for each change rate, and iterate the entire process for different change rates ranging from 1% to 10%.[5] In Table 1 we include

[4] For 3D lattices, the search objective is to plan smooth paths that satisfy the constraints on the minimum turning radius. The actions used to get successors for states are a set of *motion primitives*, which are short kinematically feasible motion sequences [71].

[5] While LPA*/GAA*/MPGAA*/TLPA* are incremental algorithms, A* is not. As A* is oblivious to the changes in the graph, it may replan in cases where the previous solution cannot be changed. To make things fair, we restrict the traversability changes to the cells that are in OPEN ∪ CLOSED, ensuring the necessity of replanning. Also, to make the changes more realistic, we use 5×5 blocks of cells as a unit rather than a single cell.

Table 1
Comparison of TLPA* with optimal search algorithms for replanning in dynamic environments. Legend: RT – runtime, SE – state expansions (×10000).

		Change rate	A*		LPA*		GAA*		MPGAA*		TLPA*(1.01)		TLPA*(1.05)		TLPA*(1.10)	
			RT	SE	RT	SE	RT	SE	RT	SE	RT	SE	RT	SE	RT	SE
2D Grid	Outdoor	1%	1.16	2.15	0.42	0.54	0.97	1.67	0.77	1.02	0.27	0.36	0.06	0.07	0.05	0.07
		2%	1.23	2.16	0.86	1.03	1.19	1.93	1.03	1.51	0.31	0.39	0.09	0.14	0.08	0.13
		5%	1.31	2.22	2.21	2.41	1.75	2.08	1.61	1.97	0.32	0.47	0.27	0.23	0.14	0.16
		10%	1.26	2.15	2.10	2.22	2.33	2.00	2.19	1.95	0.39	0.62	0.30	0.57	0.31	0.54
	Indoor	1%	2.43	4.39	1.02	1.32	1.62	2.76	1.18	1.47	0.36	0.52	0.16	0.21	0.14	0.19
		2%	2.45	4.39	1.83	2.29	1.91	3.21	1.60	2.14	0.42	0.67	0.17	0.24	0.16	0.23
		5%	2.52	4.45	4.55	5.12	2.55	3.93	2.32	3.61	1.09	1.08	0.21	0.30	0.17	0.27
		10%	2.57	4.45	5.39	5.51	3.20	4.29	3.16	4.17	1.49	1.19	0.34	0.45	0.23	0.36
3D Lattice	Outdoor	1%	3.05	3.95	0.97	0.95	2.00	3.25	1.52	2.46	0.23	0.28	0.19	0.17	0.19	0.17
		2%	3.07	3.95	1.50	1.39	2.54	3.40	1.93	3.02	0.24	0.33	0.25	0.33	0.25	0.33
		5%	3.10	3.95	4.18	3.25	3.94	3.87	3.58	3.71	0.77	1.00	0.49	0.50	0.27	0.38
		10%	3.24	4.03	5.59	4.57	5.55	3.99	5.41	3.97	0.93	1.04	0.66	0.62	0.31	0.41
	Indoor	1%	6.31	7.92	1.34	1.18	3.68	6.51	2.89	4.86	0.59	0.63	0.28	0.27	0.29	0.27
		2%	6.32	7.95	3.01	2.61	4.04	6.54	3.36	5.94	0.82	0.71	0.29	0.30	0.29	0.30
		5%	6.65	8.17	8.42	6.61	6.27	7.04	5.02	6.88	1.45	0.93	0.74	0.65	0.52	0.56
		10%	6.88	8.22	11.06	8.65	7.36	7.07	5.97	7.02	1.74	1.42	1.00	1.06	0.66	0.74

the results (in terms of average runtime and state expansions per planning iteration) comparing TLPA* ($\epsilon = 1.01, 1.05, 1.1$) with A*, LPA* and GAA*.

The results in Table 1 show some interesting trends. First, we note that the optimal incremental algorithms are good for low change rates (1%–2%). Beyond that, A* starts to outperform the incremental algorithms as the overhead of running incremental search starts to dominate.[6] Second, we observe that the run times do not always correlate with state expansions. The incremental algorithms have extra overheads (for example, the state expansions in LPA*/MPGAA* are costlier than A*, GAA*/MPGAA* requires an additional procedure to make heuristics consistent), and thus, if the incremental algorithms do not reduce the state expansions by a fair margin, they end up using more time than A*. Finally, we note the efficacy of TLPA*, even for the small bounds used here (1.01–1.10), it consistently outperforms all optimal algorithms by a significant margin, with improvements ranging from 1.5X to 30X.

It may be noted that in a way TLPA* delays the cost propagation (path correction) for states, in particular when it is not necessary to satisfy the bounds. This means that over the replanning iterations such states may accumulate, especially if the edge costs are increasing consistently. Therefore, it may happen that after a few iterations, there is an iteration where TLPA* needs to correct the paths for a large number of states, which of course increases the run time (of that iteration). To avoid such cases, we may periodically plan from scratch (so that such accumulations do not occur) or we may plan from scratch depending on the size of the TRUNCATED list (similar measures were suggested in [25]). In our experiments, we do not use any such modifications, i.e., we present the results as is.

In the second experiment, we compare TLPA* with 2 bounded suboptimal algorithms, WA* and GLPA*, for ϵ ranging from 1.01–5.0 with change rate 1%. The results of this experiment is included in Table 2. From the results, we observe that TLPA* performs reasonably well across different bounds (ϵ) for both indoor and outdoor environments. We also observe that the inflated heuristic searches (WA*/GLPA*) behave differently for outdoor and indoor environments. For outdoor environments, both algorithms converge much faster with higher ϵ. Whereas for indoor environments, the improvement reduces quite a bit, and at times increasing ϵ results in degradation in runtime. This is due to the fact that in indoor environments, often there are large local minima and inflated heuristic searches tend to get trapped in those minima, thereby causing degradation.[7] The impact is more severe in case of GLPA* due to its differential handling of underconsistent and overconsistent states. TLPA* does not use an inflated heuristic. Therefore, its performance remains more or less consistent for both kinds of maps. Overall, the results show that TLPA* is significantly better than both WA* and GLPA* for close-to-optimal bounds. For higher bounds, all the algorithms perform equally well for outdoor maps, however, for indoor maps, TLPA* remains a better choice as it is less sensitive toward the presence of local minima.

In the third experiment, we test the dependence of TLPA* on the location of the cost changes. As noted in [20,23], tree repairing searches (LPA*/GLPA*/AD*) are extremely sensitive to the position of the cost changes. If the cost changes occur close to the goal, these searches can reuse a large portion of the previous search tree and thus, are very efficient. Whereas if the changes occur close to root of the search tree, they need to rebuild most of the search tree, which considerably degrades their performance. To parameterize the experiments using the location of the cost changes, we confine 90% of the total cost

[6] This observation is corroborated in most of the studies on incremental search. For example in [20], the change rates used were within 2%. Similarly, in [66] and [23], both the amount and the position of cost changes were restricted, and it was discussed that if the changes are more than a certain bound, it is better to replan from scratch.

[7] Inflated heuristic searches magnify the impact of the heuristic, and thus are more prone to get trapped in local minima, where the heuristic values do not correlate well with the actual cost-to-go values [68,69].

Table 2
Comparison of TLPA* with bounded suboptimal search algorithms for replanning in dynamic environments (change rate 1%). Legend: RT – runtime, SE – state expansions (×10 000).

		Outdoor						Indoor					
		WA*		GLPA*		TLPA*		WA*		GLPA*		TLPA*	
	ϵ	RT	SE	RT	SE	RT	SE	RT	SE	RT	SE	RT	SE
2D Grid	5.00	0.08	0.20	0.26	0.22	0.05	0.07	0.69	1.74	1.88	1.65	0.09	0.14
	2.00	0.15	0.36	0.38	0.38	0.05	0.07	1.16	2.60	1.11	1.28	0.09	0.14
	1.50	0.18	0.47	0.39	0.40	0.05	0.07	1.44	3.14	0.93	1.24	0.08	0.14
	1.10	0.72	1.42	0.51	0.51	0.05	0.07	2.14	3.98	1.56	1.77	0.14	0.19
	1.05	0.90	1.75	0.40	0.39	0.06	0.07	2.25	4.03	1.19	1.48	0.16	0.21
	1.01	1.05	2.01	0.42	0.53	0.27	0.36	2.47	4.30	1.05	1.30	0.36	0.52
3D Lattice	5.00	0.26	0.51	0.49	0.66	0.20	0.17	1.53	2.96	3.69	2.71	0.30	0.27
	2.00	1.24	1.86	0.60	0.89	0.15	0.17	1.96	3.37	2.03	2.18	0.30	0.27
	1.50	1.16	1.90	0.95	0.88	0.20	0.17	3.87	5.72	1.87	2.77	0.26	0.27
	1.10	2.43	3.11	0.91	0.87	0.19	0.17	5.99	7.28	2.62	3.03	0.29	0.27
	1.05	2.94	3.47	0.96	0.84	0.19	0.17	6.31	7.45	1.51	1.31	0.28	0.27
	1.01	3.10	3.74	0.92	0.85	0.23	0.28	6.84	7.67	1.35	1.11	0.59	0.63

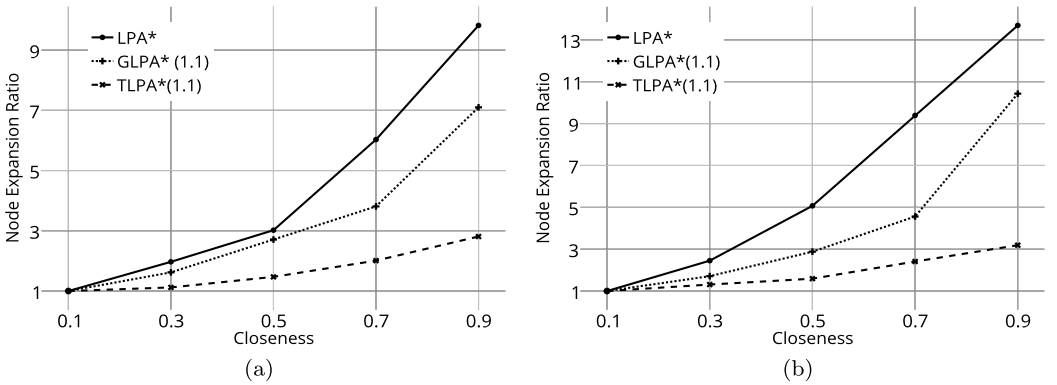


Fig. 7. State expansion ratios for LPA*, GLPA* (1.1) and TLPA* (1.1) for 2D (7a) and 3D (7b) path planning with closeness ranging from 0.1 to 0.9.

changes within a chosen area around the goal state (governed by a closeness factor). For example, if closeness is 0.1, 90% of the total cost changes will take place within 10% of the total area of the map around the goal state, the rest of the map will contain the remaining 10% of the changes. We run LPA*, GLPA* (1.1) and TLPA* (1.1) with change rate 1%, varying the closeness from 0.1 to 0.9. In Fig. 7, we include the relative performance (in terms of state expansions) for these algorithms for 2D and 3D planning, on a combined set of indoor and outdoor maps. We plot the values as a ratio between the number of states expanded for a particular run, over the number of states expanded for the run with closeness 0.1.

The figure clearly shows how sensitive LPA* is on closeness. As we increase the closeness factor, the number of state expansions increases significantly. GLPA* (1.1) is better than LPA*, as it uses an inflated heuristic which results in smaller search trees, however it still degrades quite a bit. TLPA* (1.1) is least sensitive on closeness (among these three algorithms), mainly due to two reasons. First, even if a change is close to the root, TLPA* can localize the cost propagations using truncation. More importantly, TLPA* uses both g and h to decide the truncation condition, which means it uses the total path information to arrive at a truncation decision, independent of the position of a cost change. In contrast, for LPA*/GLPA*, cost updates are done based on g values only. For example, after a cost change, LPA* will re-expand (at least once) all the states whose previous f-value is less than the optimal solution cost and whose g-value has increased, and the number of such states increases when the changes take place closer to the root of the tree. TLPA* can truncate a subset of such states if the current path estimate ($g^{\pi} + h$) is within the chosen bound of the previous total path cost ($v + h$). For states close to the root, h values should be higher, leading to more truncation. However, please note that the h value for a state is a lower bound on the remaining path cost and not the actual path cost. Thus, even for TLPA* changes close to the goal is better than changes close to the root.

6.2. Navigation in dynamic graphs (TD Lite)*

We evaluate TD* Lite comparing it with D* Lite with an inflated heuristic (so that the bounds provided by both algorithms are the same). We present results for two types of problems. One, in which the entire map is known to the robot at the start, it computes a plan and navigates towards the goal. However after every 50 steps, the map changes (i.e., some cells become blocked from unblocked and some become unblocked from blocked), so that the robot needs to replan. This procedure is

Table 3
Comparison of TD* Lite and D* Lite with inflated heuristic for navigation in known dynamic and partially known static graphs. Best results in each case is highlighted using *bold* letters. Legend: RT – runtime, SE – state expansions (×10 000).

		Known dynamic graphs								Partially known static graphs							
		Outdoor				Indoor				Outdoor				Indoor			
		D* Lite		TD* Lite		D* Lite		TD* Lite		D* Lite		TD* Lite		D* Lite		TD* Lite	
	ϵ	RT	SE	RT	SE	RT	SE	RT	SE	RT	SE	RT	SE	RT	SE	RT	SE
2D Grid	5.00	1.65	1.88	1.00	2.38	3.06	3.22	1.78	3.12	**0.40**	**0.64**	0.60	0.65	**1.47**	**2.55**	2.19	3.13
	2.00	2.71	3.43	1.16	2.38	4.28	4.68	2.01	3.12	**0.41**	**1.11**	0.49	1.12	2.49	5.10	**2.40**	**3.11**
	1.50	4.36	5.32	1.21	2.38	3.38	5.52	2.02	3.12	0.76	2.07	**0.61**	**1.76**	**1.69**	3.20	2.26	**3.16**
	1.10	5.54	8.95	1.22	2.40	3.83	7.43	2.17	3.36	**0.92**	2.41	0.96	**1.92**	2.69	4.98	**2.35**	**3.27**
	1.05	4.48	5.06	1.26	2.41	3.87	7.32	2.41	3.44	1.25	2.45	**1.01**	**2.04**	2.85	5.23	**2.34**	**3.48**
	1.01	4.86	5.95	1.34	2.53	4.01	7.60	2.42	3.51	1.97	3.43	**1.13**	**2.36**	3.18	5.68	**3.16**	**4.57**
3D Lattice	5.00	3.98	4.42	2.44	3.31	15.25	12.73	5.24	5.22	10.68	10.20	**4.56**	**4.31**	**1.98**	**3.48**	3.69	3.56
	2.00	3.13	4.00	2.34	3.31	20.71	18.52	5.20	5.22	5.54	8.49	**4.50**	**4.31**	**3.31**	5.41	3.39	**3.24**
	1.50	3.98	4.93	2.47	3.31	8.91	16.86	5.44	5.29	7.47	10.46	**4.45**	**4.31**	**3.42**	7.20	3.79	**3.77**
	1.10	6.78	7.33	2.60	3.32	10.21	11.45	5.96	5.71	5.35	7.63	**4.78**	**4.34**	7.76	10.35	**3.53**	**4.25**
	1.05	7.83	7.98	2.81	3.39	10.46	11.87	5.33	6.08	6.68	8.09	**4.94**	**4.35**	8.89	11.19	**3.44**	**4.05**
	1.01	8.87	9.17	3.35	3.71	11.81	13.34	5.29	8.11	7.54	9.00	**4.53**	**4.50**	7.92	10.33	**3.99**	**4.23**

iterated till the robot reaches its goal (known dynamic graph). And two, in which the robot only knows partially about the map (depending on its sensor range), and assumes the rest of the map to empty (free space). It plans according its current perception, and navigates towards the goal following the plan. While navigating, if it perceives any change in the graph, it replans (partially known static graph). For the first case, we use a change rate of 1% and for the second case we use a sensor range of 100 cells (the robot updates its map after 50 moves). The results for both these experiments with ϵ ranging from 1.01–5.0 are included in Table 3. Note that, here we report the total runtime and state expansions (average over 100 maps) over the entire navigation process, instead of per iteration values reported earlier.

From the results in Table 3, we see two clear trends. For known dynamic graphs, TD* Lite is a better algorithm compared to D* Lite (with inflated heuristic), especially for low bounds and harder problems. For most values of ϵ, it converges earlier, although the improvement is not as much as observed with TLPA*. This is expected, as when the robot navigates, the search space gets smaller and smaller over iterations, reducing the scope of improvement. On the other hand, for navigation in partially known graphs with free space assumption, TD* Lite does not significantly outperform D* Lite. In fact in several cases, D* Lite performs better than TD* Lite (we depict the best results for this domain using *bold* letters, as there is no clear winner). This is mainly due to the fact that with the free space assumption, the changes take place very close to the goal, making D* Lite (with inflated heuristic) very fast, as only a small fraction of tree needs to be rebuilt. This also means that for TD* Lite, the scope of truncation gets reduced, making its performance almost similar to D* Lite. Another interesting thing to note is that for navigation in partially known domains, indoor planning is not necessarily harder, as now the local minima depends on the visible part of the graph rather than the actual graph.

6.3. Anytime navigation (ATD*)

In this section, we evaluate Anytime Truncated D* (ATD*) in comparison with ARA* [5], AD* [23] and ATRA* [64], for 2D and 3D (x, y, heading) navigation. We use the same domains as used for TD* Lite, i.e., known dynamic graphs and partially known static graphs. However, here we run the algorithms in anytime mode, where each planner starts by looking for a solution with high suboptimality bound and iteratively improves the solution quality by reducing the target bound until the time limit is reached (or the optimal solution is found). For each algorithm we start with a bound of 10 and iteratively reduce the bound by 0.2 after each episode. When the time limit is reached (or the optimal solution if found), the robot starts to navigate using the current until it needs to replan due to change in graphs (for the known dynamic case) or change in perception (for the unknown static case). This process is iterated till the goal is reached. Note that, unlike ARA*/AD*/ATRA*, which use a single suboptimality bound ϵ, ATD* uses two values (ϵ_1 and ϵ_2). For ATD*, we set $\epsilon_2 = min(1.10, \sqrt{\epsilon})$ and $\epsilon_1 = \epsilon/\epsilon_2$, so that the target bound remains the same for all the algorithms. We use a time limit of 1 second for 2D planning and a time limit of 2 seconds for 3D planning.

In Table 4, we include the results of this experiment. Note that, as the robot moves towards the goal, the search space gets iteratively smaller. Which means the later episodes are generally easier to solve. For this reason, here, we do not report the average bounds over all iterations. Instead, we pick the worst 5 bounds reported over all iterations and take the average of that. The results highlight the efficacy of ATD* over other algorithms. For almost all the cases, ATD* reports a better bound and achieves a better quality solution, with the difference getting more pronounced for known dynamic graphs, especially in case of indoor navigation. The primary reason for this is the fact that while all the other algorithms rely on inflated heuristics to speedup search, ATD* can use both heuristic inflation and truncation, and thus, is more robust. Another thing to note here is that for known dynamic graphs ARA* and ATRA* both perform quite well, whereas for partially known graphs AD* and ATD* consistently produce better bounds compared to WA*/ATRA*, primarily due to the localization of

Table 4
Comparison of ARA*, AD*, ATRA* and ATD* for anytime navigation in known dynamic and partially known static graphs. Legend: ϵ – reported bound, ϵ_A – actual bound (obtain by comparing with the optimal solution cost).

		Known dynamic graphs								Partially known static graphs							
		ARA*		AD*		ATRA*		ATD*		ARA*		AD*		ATRA*		ATD*	
	Environment	ϵ	ϵ_A	ϵ	ϵ_A	ϵ	ϵ_A	ϵ	ϵ_A	ϵ	ϵ_A	ϵ	ϵ_A	ϵ	ϵ_A	ϵ	ϵ_A
2D	Outdoor	1.24	1.03	1.18	1.03	1.13	1.03	1.07	1.02	1.11	1.04	1.00	1.00	1.08	1.04	1.00	1.00
	Indoor	1.85	1.07	1.92	1.07	1.74	1.05	1.16	1.00	1.46	1.15	1.27	1.06	1.35	1.14	1.07	1.03
3D	Outdoor	1.12	1.05	1.13	1.08	1.12	1.05	1.07	1.04	1.72	1.19	1.12	1.06	1.29	1.08	1.04	1.01
	Indoor	1.77	1.14	1.63	1.19	1.55	1.14	1.19	1.08	1.64	1.13	1.24	1.10	1.41	1.13	1.09	1.05

changes, close to the goal state. Finally, we observe that all the algorithms produce relatively high quality (close-to-optimal) solutions within the chosen time limits, which shows the utility of anytime planning.

7. Conclusions

Planning for real-world applications is difficult, primarily due to the inherent uncertainties in world models. Such models often have imperfect information and thus, the models and the plans obtained using them need to be updated, when the system receives new information. Moreover, even if the model is accurate, the world may change, making the earlier plans invalid. Incremental search algorithms efficiently solve such repeated planning problems by effectively reusing the previous searches. In addition to the inherent uncertainties, often a planning problem is so complex that finding optimal solutions becomes infeasible under reasonable resource (time/memory) constraints. Anytime algorithms are suitable for such cases, as they offer the flexibility to reason about the quality-resource trade-off. Combination of these requirements (incremental and anytime) makes planning for real-world tasks a challenging area of research.

We contribute to this research by developing three new algorithms for replanning. We propose a novel method called *truncation*, that enables an incremental search to selectively re-expand states that can significantly impact the solution quality and reuse the previous values for other states, and thus, can improve the replanning runtime significantly when searching for solutions within a chosen suboptimality bound. We describe two simple rules for truncation, and show how these can be used to convert LPA* to Truncated LPA* (TLPA*), a bounded suboptimal replanning algorithm. We discuss the properties of TLPA*, proving its correctness, and experimentally demonstrate its efficacy for 2D and 3D (x, y, heading) planning. In addition, we apply the truncation rules on D* Lite, to develop Truncated D* Lite (TD* Lite), a bounded suboptimal incremental search for navigation. We also develop a novel anytime incremental search, Anytime Truncated D* (ATD*), that combines the inflated heuristic search with truncation in an anytime manner. We explain why the truncation rules used in TLPA* cannot be directly used with inflated heuristics, and propose two new truncation rules to rectify this problem. We discuss the analytical properties of ATD* and experimentally evaluate its performance in comparison with state-of-the-art anytime incremental searches.

The empirical evaluations presented in this paper suggest that the utility of incremental planning depends on the complexity of the problem. If the planning task is complex (where planning a path from scratch requires substantial effort), effective reuse of the previous planning information can significantly improve the runtime, making incremental replanning effective. In contrast, when planning is relatively easy, the overheads associated with incremental replanning can at times become prohibitive. In a way, our experiments reinforce the observations described in [73]. Interestingly, when incremental planning is effective, truncated incremental searches can significantly improve efficiency by enhancing the scope of information reuse.

All the algorithms proposed in this work are reasonably simple to implement and extend, are theoretically well-founded and (to our belief) provide significant advantage over the current state-of-the-art, especially for large, complex problems. Also, while we have presented the algorithms in this paper as heuristic search algorithms, the truncation methodology is generic enough to work seamlessly for uninformed searches (by setting the heuristic to zero) to compute fast bounded suboptimal shortest paths in dynamic graphs. Note that, this is not possible with algorithms like GLPA* or AD*, which rely on heuristic inflation to provide bounded suboptimal solutions. As such, we hope they will contribute to and motivate other researchers developing search algorithms for complex and dynamic real-world applications.

Acknowledgements

This research was sponsored by the DARPA Computer Science Study Group (CSSG) grant D11AP00275 and ONR DR-IRIS MURI grant N00014-09-1-1052.

References

[1] S.J. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, 2nd edition, Pearson Education, 2003.
[2] B. Bonet, H. Geffner, Planning as heuristic search, Artif. Intell. 129 (1–2) (2001) 5–33.
[3] C. Yuan, B. Malone, Learning optimal bayesian networks: a shortest path perspective, J. Artif. Intell. Res. 48 (2013) 23–65.

[4] X. Fan, B. Malone, C. Yuan, Finding optimal bayesian network structures with constraints learned from data, in: Proceedings of the 30th Annual Conference on Uncertainty in Artificial Intelligence, UAI-14, 2014.
[5] M. Likhachev, G.J. Gordon, S. Thrun, ARA*: Anytime A* with Provable Bounds on Sub-Optimality, Advances in Neural Information Processing Systems, vol. 16, MIT Press, Cambridge, MA, 2004.
[6] J. Butzke, K. Daniilidis, A. Kushleyev, D.D. Lee, M. Likhachev, C. Phillips, M. Phillips, The University of Pennsylvania Magic 2010 multi-robot unmanned vehicle system, J. Field Robotics 29 (5) (2012) 745–761.
[7] B. MacAllister, J. Butzke, A. Kushleyev, M. Likhachev, Path planning for non-circular micro aerial vehicles in constrained environments, in: Proceedings of the IEEE International Conference on Robotics and Automation, ICRA, 2013.
[8] B. Cohen, S. Chitta, M. Likhachev, Heuristic search-based planning for manipulation, Int. J. Robotics Res. (2013).
[9] A. Hornung, A. Dornbush, M. Likhachev, M. Bennewitz, Anytime footstep planning with suboptimality bounds, in: Proceedings of the IEEE-RAS International Conference on Humanoid Robots, HUMANOIDS, 2012.
[10] M. Zucker, N. Ratliff, M. Stole, J. Chestnutt, J.A. Bagnell, C.G. Atkeson, J. Kuffner, Optimization and learning for rough terrain legged locomotion, Int. J. Robot. Res. 30 (2) (2011) 175–191.
[11] A. Bagchi, A. Mahanti, Three approaches to heuristic search in networks, J. ACM 32 (1) (1985) 1–27.
[12] D. Stanojevic, I. Gamvros, B. Golden, S. Raghavan, Heuristic search for network design, in: H. Greenberg (Ed.), Tutorials on Emerging Methodologies and Applications in Operations Research, Springer, 2005.
[13] D. Stanojevic, B. Golden, S. Raghavan, Heuristic search for the generalized minimum spanning tree problem, INFORMS J. Comput. 17 (3) (2005) 290–304.
[14] P.S. Dasgupta, S. Sur-Kolay, B.B. Bhattacharya, VLSI floorplan generation and area optimization using AND–OR graph search, in: VLSI Design, 1995, pp. 370–375.
[15] P. Dasgupta, P.P. Chakrabarti, A. Dey, S. Ghose, W. Bibel, Solving constraint optimization problems from CLP-style specifications using heuristic search techniques, IEEE Trans. Knowl. Data Eng. 14 (2) (2002) 353–368.
[16] S. Das, P.P. Chakrabarti, P. Dasgupta, Instruction-set-extension exploration using decomposable heuristic search, in: VLSI Design, 2006, pp. 293–298.
[17] Q. Du, G.A. Arteca, P.G. Mezey, Heuristic lipophilicity potential for computer-aided rational drug design, J. Comput.-Aided Mol. Des. 11 (5) (1997) 503–515.
[18] D.R. Westhead, D.E. Clark, C.W. Murray, A comparison of heuristic search algorithms for molecular docking, J. Comput.-Aided Mol. Des. 11 (3) (1997) 209–228.
[19] E. Keedwell, A. Narayanan, Intelligent Bioinformatics: The Application of Artificial Intelligence Techniques to Bioinformatics Problems, John Wiley and Sons, June 2005.
[20] S. Koenig, M. Likhachev, D. Furcy, Lifelong planning A*, Artif. Intell. 155 (1–2) (2004) 93–146.
[21] S. Koenig, M. Likhachev, D* Lite, in: R. Dechter, R.S. Sutton (Eds.), AAAI/IAAI, AAAI Press/The MIT Press, 2002, pp. 476–483.
[22] D. Ferguson, A. Stentz, Using interpolation to improve path planning: the field D* algorithm, J. Field Robot. 23 (2) (2006) 79–101.
[23] M. Likhachev, D. Ferguson, G.J. Gordon, A. Stentz, S. Thrun, Anytime search in dynamic graphs, Artif. Intell. 172 (14) (2008) 1613–1643.
[24] S. Aine, M. Likhachev, Truncated incremental search: faster replanning by exploiting suboptimality, in: M. desJardins, M.L. Littman (Eds.), Proc. of the 27th AAAI Conference, AAAI Press, 2013.
[25] S. Aine, M. Likhachev, Anytime truncated D*: anytime replanning with truncation, in: M. Helmert, G. Röger (Eds.), SOCS, AAAI Press, 2013.
[26] A. Stentz, The focussed D* algorithm for real-time replanning, in: IJCAI, Morgan Kaufmann, 1995, pp. 1652–1659.
[27] G. Ramalingam, T.W. Reps, An incremental algorithm for a generalization of the shortest-path problem, J. Algorithms 21 (2) (1996) 267–305.
[28] X. Sun, S. Koenig, The fringe-saving A* search algorithm – a feasibility study, in: IJCAI, 2007, pp. 2391–2397.
[29] K.I. Trovato, L. Dorst, Differential A*, IEEE Trans. Knowl. Data Eng. 14 (6) (2002) 1218–1229.
[30] K. Gochev, A. Safonova, M. Likhachev, Incremental planning with adaptive dimensionality, in: D. Borrajo, S. Kambhampati, A. Oddi, S. Fratini (Eds.), ICAPS, AAAI, 2013.
[31] S. Koenig, M. Likhachev, Adaptive A*, in: F. Dignum, V. Dignum, S. Koenig, S. Kraus, M.P. Singh, M. Wooldridge (Eds.), AAMAS, ACM, 2005, pp. 1311–1312.
[32] X. Sun, S. Koenig, W. Yeoh, Generalized adaptive A*, in: L. Padgham, D.C. Parkes, J.P. Müller, S. Parsons (Eds.), AAMAS (1), IFAAMAS, 2008, pp. 469–476.
[33] C. Hernández, P. Meseguer, X. Sun, S. Koenig, Path-adaptive A* for incremental heuristic search in unknown terrain, in: ICAPS, 2009.
[34] C. Hernández, X. Sun, S. Koenig, P. Meseguer, Tree adaptive A*, in: The 10th International Conference on Autonomous Agents and Multiagent Systems, vol. 1, International Foundation for Autonomous Agents and Multiagent Systems, 2011, pp. 123–130.
[35] C. Hernández, R. Asín, J.A. Baier, Reusing previously found A* paths for fast goal-directed navigation in dynamic terrain, in: Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25–30, 2015, Austin, Texas, USA, AAAI Press, 2015, pp. 1158–1164.
[36] X. Sun, W. Yeoh, S. Koenig, Generalized fringe-retrieving A*: faster moving target search on state lattices, in: AAMAS, 2010, pp. 1081–1088.
[37] X. Sun, W. Yeoh, S. Koenig, Moving target D* lite, in: AAMAS, 2010, pp. 67–74.
[38] A. Gerevini, I. Serina, Fast plan adaptation through planning graphs: local and systematic search techniques, in: S. Chien, S. Kambhampati, C.A. Knoblock (Eds.), AIPS, AAAI, 2000, pp. 112–121.
[39] A. Gerevini, A. Saetti, I. Serina, Planning through stochastic local search and temporal action graphs in LPG, J. Artif. Intell. Res. 20 (2003) 239–290.
[40] M. Fox, A. Gerevini, D. Long, I. Serina, Plan stability: replanning versus plan repair, in: D. Long, S.F. Smith, D. Borrajo, L. McCluskey (Eds.), ICAPS, AAAI, 2006, pp. 212–221.
[41] R.E. Korf, Real-time heuristic search, Artif. Intell. 42 (2–3) (1990) 189–211.
[42] S. Koenig, M. Likhachev, Real-time adaptive A*, in: Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems, ACM, 2006, pp. 281–288.
[43] D. Bond, N.A. Widger, W. Ruml, X. Sun, Real-time search in dynamic worlds, in: Third Annual Symposium on Combinatorial Search, 2010.
[44] C. Undeger, F. Polat, RTTES: real-time search in dynamic environments, Appl. Intell. 27 (2) (2007) 113–129.
[45] I. Pohl, Heuristic search viewed as path finding in a graph, Artif. Intell. 1 (3) (1970) 193–204.
[46] R. Zhou, E.A. Hansen, Multiple sequence alignment using anytime A*, in: Proceedings of 18th National Conference on Artificial Intelligence, AAAI'2002, 2002, pp. 975–976.
[47] J. Pearl, Heuristics: Intelligent Search Strategies for Computer Problem Solving, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.
[48] E.A. Hansen, S. Zilberstein, V.A. Danilchenko, Anytime heuristic search: first results, Tech. rep. 50, Univ. of Massachusetts, 1997.
[49] E.A. Hansen, R. Zhou, Anytime heuristic search, J. Artif. Intell. Res. 28 (1) (2007) 267–297.
[50] S. Richter, J.T. Thayer, W. Ruml, The joy of forgetting: faster anytime search via restarting, in: R.I. Brafman, H. Geffner, J. Hoffmann, H.A. Kautz (Eds.), ICAPS, AAAI, 2010, pp. 137–144.
[51] R. Stern, A. Felner, J. van den Berg, R. Puzis, R. Shah, K. Goldberg, Potential-based bounded-cost search and anytime non-parametric A*, Artif. Intell. 214 (2014) 1–25.
[52] V. Kumar, Branch-and-bound search, in: Encyclopedia of Artificial Intelligence, 1992, pp. 1468–1472.
[53] W. Zhang, Complete anytime beam search, in: Proceedings of 14th National Conference of Artificial Intelligence, AAAI'98, AAAI Press, 1998, pp. 425–430.
[54] R. Bisiani, Beam search, in: Encyclopedia of Artificial Intelligence, 1987, pp. 56–58.

[55] R. Zhou, E.A. Hansen, Beam-stack search: integrating backtracking with beam search, in: Proceedings of the 15th International Conference on Automated Planning and Scheduling, ICAPS-05, Monterey, CA, 2005, pp. 90–98.
[56] D. Furcy, ITSA*: iterative tunneling search with A*, in: Proceedings of AAAI Workshop on Heuristic Search, Memory-Based Heuristics and Their Applications, 2006, pp. 21–26.
[57] D. Furcy, S. Koenig, Limited discrepancy beam search, in: Proceedings of the 19th International Joint Conference on Artificial Intelligence, IJCAI'05, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005, pp. 125–131.
[58] S.G. Vadlamudi, S. Aine, P.P. Chakrabarti, Anytime pack search, Nat. Comput. (2015) 1–20.
[59] S.G. Vadlamudi, P. Gaurav, S. Aine, P.P. Chakrabarti, Anytime column search, in: AI 2012: Advances in Artificial Intelligence – 25th Australasian Joint Conference, Proceedings, Sydney, Australia, December 4–7, 2012, Springer, 2012, pp. 254–265.
[60] S. Aine, P.P. Chakrabarti, R. Kumar, AWA* – a window constrained anytime heuristic search algorithm, in: IJCAI, 2007, pp. 2250–2255.
[61] J.T. Thayer, J. Benton, M. Helmert, Better parameter-free anytime search by minimizing time between solutions, in: SOCS, 2012.
[62] J. Pearl, J.H. Kim, Studies in semi-admissible heuristics, IEEE Trans. Pattern Anal. Mach. Intell. 4 (4) (1982) 392–399.
[63] S.G. Vadlamudi, S. Aine, P.P. Chakrabarti, MAWA* – a memory-bounded anytime heuristic-search algorithm, IEEE Trans. Syst. Man Cybern., Part B, Cybern. 41 (3) (2011) 725–735.
[64] K. Gochev, A. Safonova, M. Likhachev, Anytime tree-restoring weighted A* graph search, in: Proceedings of the Seventh Annual Symposium on Combinatorial Search, SOCS 2014, Prague, Czech Republic, 15–17 August 2014, 2014.
[65] X. Sun, W. Yeoh, T. Uras, S. Koenig, Incremental ARA*: an incremental anytime search algorithm for moving-target search, in: L. McCluskey, B. Williams, J.R. Silva, B. Bonet (Eds.), ICAPS, AAAI, 2012.
[66] M. Likhachev, S. Koenig, A generalized framework for lifelong planning A* search, in: S. Biundo, K.L. Myers, K. Rajan (Eds.), ICAPS, AAAI, 2005, pp. 99–108.
[67] S. Aine, M. Likhachev, Truncated LPA*: the proofs, Tech. rep. TR-12-32, Carnegie Mellon University, Pittsburgh, PA, 2013.
[68] C.M. Wilt, W. Ruml, When does weighted A* fail?, in: SOCS, AAAI Press, 2012.
[69] S. Aine, S. Swaminathan, V. Narayanan, V. Hwang, M. Likhachev, Multi-heuristic A*, in: Robotics: Science and Systems, 2014.
[70] S. Aine, M. Likhachev, Anytime truncated D*: the proofs, Tech. rep. TR-13-08, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, USA, 2013.
[71] M. Likhachev, D. Ferguson, Planning long dynamically feasible maneuvers for autonomous vehicles, I, Int. J. Robot. Res. 28 (8) (2009) 933–945.
[72] P.E. Hart, N.J. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE Trans. Syst. Sci. Cybern. 4 (2) (1968) 100–107.
[73] C. Hernández, J.A. Baier, T. Uras, S. Koenig, Position paper: incremental search algorithms considered poorly understood, in: SOCS, 2012.