Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence

# Truncated Incremental Search: Faster Replanning by Exploiting Suboptimality

**Sandip Aine and Maxim Likhachev**
Robotics Institute, Carnegie Mellon University
Pittsburgh, PA, US

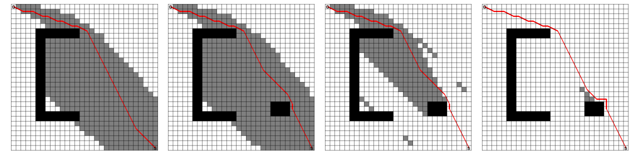
## Abstract

Incremental heuristic searches try to reuse their previous search efforts whenever these are available. As a result, they can often solve a sequence of similar planning problems much faster than planning from scratch. State-of-the-art incremental heuristic searches such as LPA*, D* and D* Lite all work by propagating cost changes to all the states on the search tree whose $g$-values (the costs of computed paths from the start) are no longer optimal. While such a complete propagation of cost changes is required to ensure optimality, the propagations can be stopped much earlier if we are looking for solutions within a given suboptimality bound. We present a framework called Truncated Incremental Search that builds on this observation, and uses a target suboptimality bound to efficiently restrict the cost propagations. Using this framework, we develop two algorithms, Truncated LPA* (TLPA*) and Truncated D* Lite (TD* Lite). We discuss their analytical properties and present experimental results for 2D and 3D (x, y, heading) path planning that show significant improvement in runtime over existing incremental heuristic searches when searching for close-to-optimal solutions. In addition, unlike typical incremental searches, Truncated Incremental Search is much less dependent on the proximity of the cost changes to the goal of the search due to the early termination of the cost change propagation.

## Introduction

Incremental search refers to a class of heuristic search algorithms for continual planning that reuses information from previous searches to speed up the current search. In many applications, search is used repeatedly to find solutions to a series of similar problems with the environment changing dynamically over time. Incremental search can be a useful strategy for such applications to obtain solutions faster than solving each problem independently.

Lifelong planning A* (LPA*) (Koenig, Likhachev, and Furcy 2004) is an incremental version of A* that solves a sequence of similar search problems efficiently by using the $g$-values from previous searches. Its first iteration is the same as that of A*, but the subsequent searches are potentially faster as it reuses parts of the previous search tree that are identical to the new search tree. The rest of the tree is rebuilt by propagating the new costs. This approach can reduce the search time if large parts of the search trees are identical. For example, if the problems change only slightly and the changes are close to the goal, LPA* can converge faster than A*. In Figures 1b and 1c, we present a simple example depicting how LPA* can replan much faster than A* for repeated planning. LPA* has been used as a backbone for several incremental algorithms, such as D* Lite (Likhachev and Koenig 2005), Field D* (Ferguson and Stentz 2006), Anytime D* (Likhachev et al. 2008) which are widely used in Robotics and have been used on the Mars Rover, DARPA Urban Challenge winner, and other real life projects.



(a) 1st A*, LPA* and TLPA* (b) 2nd A* (c) 2nd LPA* (d) 2nd TLPA*

Figure 1: A simple path planning example for a $30\times30$ grid showing the difference between A*, LPA* and TLPA*. The first search (1a) is identical for all the algorithms (expanded states are shown shaded). After the first search, a new obstacle is introduced. A* does a complete new search (1b). LPA* partially reuses the earlier search tree and is more efficient than A* (1c). However, it still expands a considerable number of states. TLPA* ($\epsilon = 1.1$) quickly finds a way around the new obstacle and recomputes a bounded path with much fewer expansions (1d). The solution by TLPA* is guaranteed to be within 10% suboptimality.

As LPA* recomputes the optimal solution every time the environment changes, it needs to propagate the cost changes to all the states of the search tree whose $g$-values have changed. This means that even for a small change in costs, large part of the search tree may need to be regenerated, especially if the changes occur close to the root of the search tree, thus increasing the replanning runtime.

Our work is based on the observation that we can avoid most of this work when searching for solutions within a

Copyright © 2013, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

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

given suboptimality bound, as opposed to optimal solutions. In particular, we can prune those cost propagations that do not impact the solution quality beyond the given suboptimality bound. To exploit this, we develop Truncated LPA* (TLPA*), an incremental search algorithm that speeds up replanning by using a suboptimality bound ($\epsilon$) to limit reexpansions. 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. As a result, it can substantially improve the replanning runtime, and still guarantee solution qualities within the chosen suboptimality bound. Also, because TLPA* terminates the cost propagations based on the bound, its performance is less dependent on whether the changes occur close to the goal state or not. Figures 1c and 1d show how TLPA* can use a suboptimality bound to significantly reduce the state expansions when compared to LPA*. While LPA* reexpands a large portion of the search tree (Figure 1c), TLPA* ($\epsilon = 1.1$) converges much faster by truncating the reexpansions of states that are not necessary to satisfy the $\epsilon$-bound (Figure 1d).

We present the theoretical properties of TLPA* demonstrating its correctness and showing that it retains the expansion efficiency of LPA*. We experimentally evaluate TLPA* for two domains, 2D and 3D (x, y, heading) path planning, comparing it with state-of-the-art optimal and bounded suboptimal incremental search algorithms.

Additionally, we demonstrate how the truncation rules can be integrated with the D* Lite algorithm, resulting in Truncated D* Lite (TD* Lite), a bounded suboptimal algorithm for navigation in dynamic graphs, and present experimental results comparing it with D* lite (and others) for 2D and 3D (x, y, heading) navigation.

## Related Work

The incremental heuristic search algorithms found in AI literature can be classified in three main categories. The first class (LPA* (Koenig, Likhachev, and Furcy 2004), D* (Stentz 1995), D* Lite (Likhachev and Koenig 2005)) 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. This approach is also used for uninformed searches (DynamicSWSF-FP (Ramalingam and Reps 1996)), to find shortest paths for a series of similar problems. The second class (Fringe Saving A* (Sun and Koenig 2007), Differential A* (Trovato and Dorst 2002)) restarts A* at the point where the current search deviates from the previous run, while reusing the earlier search queue up to that point. The third class (Adaptive A* (Koenig and Likhachev 2005), Generalized Adaptive A* (Sun, Koenig, and Yeoh 2008)) updates the $h$-values from the previous searches to make them more informed over iterations. Our algorithms (TLPA* and TD* Lite) belong to the first category, as they are based on LPA*.

Graph search based local approaches have also been used for plan adaptation/repair in dynamic environments (Gerevini and Serina 2000; Gerevini, Saetti, and Serina 2003). However, these algorithms generally focus on improving different metrics (such as plan stability (Fox et al. 2006)), and typically they do not provide any guarantees on the quality of solutions.

There is also a number of heuristic search algorithms that search for bounded suboptimal solutions. Majority of them follow the Weighted A* (WA* (Pohl 1970)) approach, where the heuristic is inflated by a constant factor ($> 1.0$) to give the search a depth first flavor. Among incremental searches, LPA* has been extended to work with such inflated heuristics, resulting in bounded suboptimal replanning algorithms such as GLPA* (Likhachev and Koenig 2005) and Anytime D* (Likhachev et al. 2008).

It should be noted that while both the inflated heuristic search algorithms (e.g., WA*, GLPA*) and Truncated Incremental Searches produce solutions within a chosen suboptimality bound, these algorithms are fundamentally different. The former speed up *planning* using inflated heuristics, whereas Truncated Incremental Searches use the suboptimality bound to accelerate *replanning* by restricting reexpansions. For this reason, Truncated Incremental Search can be easily used as an uniformed search (by setting the heuristic to zero) producing bounded suboptimal solutions for incremental shortest path problems, which is not possible with algorithms like GLPA* or Anytime D*. To our knowledge, we present the first incremental search framework that uses bounded-optimal truncation to improve replanning runtime.

1 **procedure key**($s$)
2 return $[min(g(s), v(s)) + h(s); min(g(s), v(s))]$;
3 **procedure InitState**($s$)
4 $v(s) = g(s) = \infty$; $bp(s) = \mathbf{null}$;
5 **procedure UpdateState**($s$)
6 if $s$ was never visited InitState($s$);
7 if ($s \neq s_{start}$)
8 $bp(s) = \text{argmin}_{(s'' \in Pred(v))} v(s'') + c(s'', s)$;
9 $g(s) = v(bp(s)) + c(bp(s), s)$;
10 if ($g(s) \neq v(s)$) insert/update $s$ in $OPEN$ with $key(s)$ as priority;
11 else if $s \in OPEN$ remove $s$ from $OPEN$;
12 **procedure ComputePath**()
13 while $OPEN.Minkey() < key(s_{goal})$ OR $v(s_{goal}) < g(s_{goal})$
14 $s = OPEN.Top()$;
15 remove $s$ from $OPEN$;
16 if ($v(s) > g(s)$)
17 $v(s) = g(s)$;
18 for each $s'$ in Succ($s$) UpdateState($s'$);
19 else
20 $v(s) = \infty$;
21 for each $s'$ in Succ($s$)$\cup s$ UpdateState($s'$);
22 **procedure Main**()
23 InitState($s_{start}$); InitState($s_{goal}$);
24 $g(s_{start}) = 0$; $OPEN = \emptyset$;
25 insert $s_{start}$ into $OPEN$ with $key(s_{start})$ as priority;
26 forever
27 ComputePath;
28 Wait for changes in edge costs;
29 for each directed edges $(u, v)$ with changed edge costs
30 update the edge cost $c(u, v)$; UpdateState($v$);

Figure 2: LPA*

## LPA*

**Notations** In the following, $S$ denotes the finite set of states of the domain. $c(s, s')$ denotes the cost of the edge between $s$ and $s'$, if there is no such edge, then $c(s, s') = \infty$. $Succ(s) := \{s' \in S | c(s, s') \neq \infty\}$, denotes the set of all successors of $s$. Similarly, $Pred(s) := \{s' \in S | s \in$

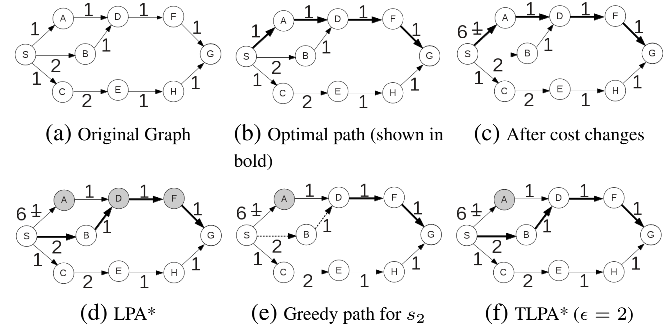
$Succ(s')\}$ denotes the set of predecessors of $s$. $g^*(s)$ denotes optimal path cost from $s_{start}$ to $s$.

LPA* repeatedly determines a minimum-cost path from a given start state to a given goal state in a graph that represents a planning problem while some of the edge costs change. It maintains two kinds of estimates of the cost of a path from $s_{start}$ for each 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 its predecessors (as stated in **Invariant 1** below) and thus is potentially better informed than $v(s)$. Additionally, it stores a backpointer $bp(s)$ for each state $s$ pointing to best predecessor of $s$ (if computed). LPA* always satisfies the following relationships: $bp(s_{start}) = \mathbf{null}, g(s_{start}) = 0$ and $\forall s \in S - \{s_{start}\},\ bp(s) = \text{argmin}_{(s' \in Pred(s))} v(s') + c(s', s), g(s) = v(bp(s)) + c(bp(s), s)$ (**Invariant 1**).

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* uses a consistent heuristic $h(s)$ and a priority queue to focus its search and to order its cost updates efficiently. The priority queue ($OPEN$) always contains the inconsistent states only (**Invariant 2**). The priority ($key(s)$) of a state $s$ is given by: $key(s) = [key_1(s), key_2(s)]$ where $key_1(s) = min(g(s), v(s)) + h(s)$ and $key_2(s) = min(g(s), v(s))$. Priorities are compared in a lexicographic order, i.e., for two states $s$ and $s'$ $key(s) \leq key(s')$, iff either $key_1(s) < key_1(s')$ or ($key_1(s) = key_1(s')$ and $key_2(s) \leq key_2(s')$ (**Invariant 3**).

The pseudo code of a basic version of LPA* is shown in Figure 2. LPA* starts by initializing the states and inserting $s_{start}$ into $OPEN$ (lines 23-25). It then calls the ComputePath function to obtain a minimum cost solution. ComputePath expands the inconsistent states from $OPEN$ in increasing order of priority in a manner that the **Invariants 1-3** are always satisfied, until it discovers a minimum cost path to $s_{goal}$. If a state $s$ is overconsistent, ComputePath makes it consistent by setting $v(s) = g(s)$ (line 17) and propagates this information to its successors by updating their $g$-, $v$- and $bp$- values according to **Invariant 1**. This may make some $s' \in Succ(s)$ inconsistent, which are then put into $OPEN$ ensuring **Invariant 2** (the UpdateState function). If $s$ is underconsistent, ComputePath forces it to become overconsistent by setting $v(s) = \infty$ (line 20) and propagates the underconsistency information to its children (again ensuring **Invariant 1**), and selectively puts $s$ and its children back to $OPEN$ maintaining **Invariant 2**. If this state ($s$) is later selected for expansion as an overconsistent state, it is made consistent as discussed before.

During the initialization, $v(s)$ is set to $\infty, \forall s \in S$. Thus, in the first iteration there are no underconsistent states, and the expansions performed are 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 by calling the UpdateState function (line 30) to maintain **Invariant 1**. This may introduce inconsistencies between $g$- and $v$- values for some states. These inconsistent states are then put into $OPEN$ to maintain **Invariant 2** (in the same UpdateState function). LPA* then calls ComputePath again to fix these inconsistencies. As before, ComputePath expands the inconsistent states in order of increasing priority, until there is no state in $OPEN$ with a key value less than that of $s_{goal}$ and $s_{goal}$ itself is not underconsistent (line 13).


(a) Original Graph (b) Optimal path (shown in bold) (c) After cost changes

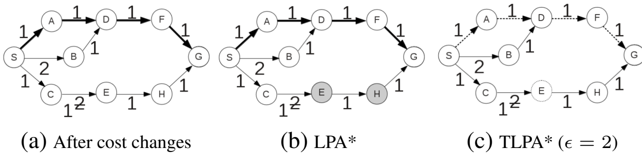
(d) LPA* (e) Greedy path for $s_2$ (f) TLPA* ($\epsilon = 2$)

Figure 3: Example of Truncation Rule 1

## Truncated LPA*

In this section, we formally present TLPA* and discuss its properties. We start by explaining the truncation rules with simple examples. First, we introduce a new term called $g^\pi(s)$, which denotes cost of the path from $s_{start}$ to $s$ computed by following the current backpointers ($bp$). If no such path exists, then $g^\pi(s) = \infty$.

**Truncation Rule 1:** Rule 1 is applicable for the underconsistent states. We explain this rule by using the example shown in Figure 3. The initial graph and the corresponding optimal path (from $S$ to $G$) are shown in Figures 3a and 3b. To simplify the example, we assume that $h(s) = 0, \forall s$. After the first iteration, cost of the edge from $S$ to $A$ changes from 1 to 6, making $A$ an underconsistent state ($g(A) = 6$ and $v(A) = 1$, after LPA* calls UpdateState($A$)). LPA* propagates this cost change to all the states that computed its $g$- values using $v(A)$ by expanding the shaded states. The new minimum-cost path is shown in Figure 3d. States $D$ and $F$ are expanded twice (once as underconsistent and once as overconsistent) resulting in 5 reexpansions. In TLPA*, we observe that when we are looking for an $\epsilon$-suboptimal solution, we can reuse the *old* $v(s)$ value for an underconsistent state $s$ (selected for expansion), as long as $g^\pi(s) + h(s) \leq \epsilon * (v(s) + h(s))$. Consider the same example (Figure 3) with $\epsilon = 2$. In the second iteration, after $A$ is expanded, the $bp$- pointer of $D$ points to $B$. Thus, when $D$ is selected for expansion, we have a path from $S$ to $D$ with cost 3 (path $S \rightarrow B \rightarrow D$, shown by dotted line in Figure 3e), i.e., $g^\pi(D) = 3$. Now, as $g^\pi(D) + h(D) = 3 + 0 \leq \epsilon * (v(D) + h(D))$ (as $v(D) + h(D) = 2$ and $\epsilon = 2$), we can truncate the cost propagation at $D$. The successors of $D$ can continue to rely on the *old* $v(D)$ (namely $v(D) = 2$) and guarantee that the path cost through them will still be within $\epsilon$-suboptimality bound. This stems from the fact that for an underconsistent state $s$ selected for expansion, $v(s) + h(s)$ is always 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, and as we already have a path that satisfies the bound on $v(s) + h(s)$, any state $s'$ that uses $v(s)$ to compute $g(s')$ will not underestimate the actual solution cost by more



(a) After cost changes (b) LPA* (c) TLPA* ($\epsilon = 2$)

Figure 4: Example of Truncation Rule 2

than the $\epsilon$ factor. In other words, even if we compute the new solution using the *old* $v(s)$, the obtained solution cost will be $\leq \epsilon*$ (cost based on the *old* $v(s)$), and thus $\leq \epsilon*$ (minimum solution cost). Therefore, for this example, we can truncate the cost propagation at $D$ and compute an 2-bounded solution, with 1 expansion compared to 5 expansions by LPA*. Below, we formally state Truncation Rule 1.

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

**Truncation Rule 2:** Rule 2 is applicable for both underconsistent and overconsistent states. We explain this rule with Figure 4 using overconsistent states. The original graph and the initial solution are same as shown in Figures 3a and 3b. After the first iteration, cost of the edge from $C$ to $E$ changes from 2 to 1. LPA* propagates this cost change until it is sure that it can not improve the minimum-cost solution. It expands $E$ and $H$ before returning the previous solution as the minimum cost solution (Figure 4b). Now, let us consider suboptimality bound $\epsilon = 2$. We observe that LPA* expands inconsistent states in an increasing order of their key values until the goal state has the lowest key value. Therefore, for any state $s$ expanded in a given iteration, its $key_1(s)$, given by $key_1(s) = min(g(s), v(s)) + h(s)$ provides a lower bound on the minimum-cost solution, i.e., $g^*(s_{goal}) \geq key_1(s)$. Generalizing this for $\epsilon$-suboptimality, we can say that if $\epsilon * key_1(s) \geq g^\pi(s_{goal})$, expansion of $s$ can not improve the current solution by more than $1/\epsilon$ factor. In Figure 4, when $E$ is selected for expansion, it has $key_1(E) = 2$ (as $g(E) = 2$, $v(E) = 3$, and $h(E) = 0$) and $g^\pi(s_{goal}) = 4$ (the corresponding path is shown in dotted line, Figure 4c). As expansion of $E$ can not produce a solution with cost less than 2 (i.e., better than $4/\epsilon$), at this point we truncate $E$. Moreover, as the key values of expanded states are monotonically non-decreasing, the same condition will ensure that expansion of any other state $s'$ in $OPEN$ can not improve the current bound. Thus, we may terminate expansions altogether for the current replanning iteration as $g^\pi(s_{goal})$ is guaranteed to be within the $\epsilon$-bound. Thus, for this example, the current iteration is terminated when $\epsilon * key_1(E) = 2 * 2 \geq 4$, providing a 2-bounded solution without any new expansions. Below, we formally state Truncation Rule 2.

**Rule 2.** *A state $s$ having $key(s) \leq key(u), \forall u \in OPEN$ is truncated if $\epsilon * key_1(s) \geq g^\pi(s_{goal})$. Also, if any state $s$ is truncated using Rule 2, all states $s' \in OPEN$ are truncated as, $\forall s' \in OPEN,\ key_1(s') \geq key_1(s)$.*

## The Algorithm

The TLPA* algorithm is presented in Figures 5 and 6. As noted earlier, TLPA* uses an extra estimate $g^\pi(s)$ for each

1 **procedure StorePath**$(s)$
2 $path(s) = \emptyset$;
3 while $(s \neq s_{start})$ AND $(bp(s) \notin TRUNCATED)$
4 insert $bp(s)$ in $path(s)$; $s = bp(s)$;
5 **procedure ObtainPathFromTruncated**$(s)$
6 if $path(s) = \textbf{null}$ return;
7 else append $path(s)$ to $Path$;
8 ObtainPathFromTruncated($end\ state$ of $path(s)$)
9 **procedure ObtainPath**
10 $s = s_{goal}$; insert $s$ in $Path$;
11 while $(s \neq s_{start})$
12 if $bp(s) \in TRUNCATED$
13 ObtainPathFromTruncated $(s)$; return;
14 insert $bp(s)$ in $Path$; $s = bp(s)$;
15 **procedure ComputeGpi**$(s)$
16 $visited = \emptyset$; $cost = 0$; $s' = s$;
17 while $(s' \neq s_{start})$
18 if $(s' \in visited)$ OR $(bp(s') = \textbf{null})$
19 $cost = \infty$; break;
20 if $(s' \in TRUNCATED)$
21 $cost = cost + g^\pi(s')$; break;
22 insert $s'$ in $visited$;
23 $cost = cost + c(bp(s'), s')$; $s' = bp(s')$;
24 $g^\pi(s) = cost$;

Figure 5: Auxiliary routines for TLPA*

state $s$, in addition to the $g$- and $v$- values. TLPA* also uses an additional list ($TRUNCATED$) to store the underconsistent states that are truncated during a particular replanning iteration.

For a state $s$, $g^\pi(s)$ is defined as the path cost returned by the ComputeGpi routine (line 15, Figure 5). The ComputeGpi function returns a finite cost if there is a path from $s_{start}$ to $s$ that can be traversed by following the current backpointers (starting at $s$), if not, it returns $\infty$. However, while following the backpointers, if ComputeGpi encounters a truncated state $s'$ ($s' \in TRUNCATED$), it adds the $g^\pi(s')$ value to the current cost and returns it, as a truncated state $s'$ always has a finite cost path from $s_{start}$ with cost $g^\pi(s')$.

The ComputePath function uses the $g^\pi$- values to apply the truncation rules. Before each expansion, $g^\pi(s_{goal})$ (line 12, Figure 6) is computed to check whether Rule 2 can be applied. If the check at line 13, Figure 6 is satisfied, TLPA* terminates with solution cost $= g^\pi(s_{goal})$. Otherwise, it continues to expand states in the increasing order of their priorities. If the state $s$ selected for expansion is underconsistent, $g^\pi(s)$ is computed (line 23, Figure 6) to check whether Rule 1 can be applied. If the check at line 24, Figure 6 is satisfied, ComputePath truncates $s$ (puts $s$ into $TRUNCATED$) after storing the current path (from $s_{start}$) to $s$ using the StorePath routine[1].

Apart from the application of truncation rules, the expansion of states is similar to LPA*, the only difference being that a truncated state is never reinserted into $OPEN$ during the current iteration (line 7, Figure 6). If ComputePath terminates at line 13 (Figure 6), a finite cost path from $s_{start}$ to $s_{goal}$ having cost $\leq \epsilon * g^*(s_{goal})$ can be computed by calling the ObtainPath routine. The ObtainPath function follows the

[1]The paths for truncated states need to be stored because the backpointers along the path can change later, making the path invalid. In general, the storage requirement for this is very low as the paths are usually shared among the truncated states.

```
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) = null;
5  procedure UpdateSetMemberShip(s)
6  if (g(s) ≠ v(s))
7    if (s ∉ TRUNCATED) insert/update s in OPEN with key(s) as priority;

8  else if (s ∈ OPEN) remove s from OPEN;
9  procedure ComputePath(ε)
10  while OPEN.Minkey() < key(s_goal) OR v(s_goal) < g(s_goal)
11   s = OPEN.Top();
12   ComputeGpi(s_goal);
13   if (g^π(s_goal) ≤ ε * (min(g(s), v(s)) + h(s))) return;
14   remove s from OPEN;
15   if (v(s) > g(s))
16    v(s) = g(s);
17    for each s' in Succ(s)
18     if s' was never visited InitState (s');
19     if g(s') > g(s) + c(s, s')
20      g(s') = g(s) + c(s, s'); bp(s') = s;
21      UpdateSetMembership(s');
22   else
23    ComputeGpi(s);
24    if (g^π(s) + h(s) ≤ ε * (v(s) + h(s)))
25     StorePath(s); insert s in TRUNCATED;
26    else
27     v(s) = ∞;  UpdateSetMembership(s);
28     for each s' in Succ(s)
29      if s' was never visited InitState (s');
30      if bp(s') = s
31       bp(s') = argmin_(s''∈Pred(s')) v(s'') + c(s'', s');
32       g(s') = v(bp(s')) + c(bp(s'), s');
33       UpdateSetMembership(s');
34  procedure Main(ε)
35  InitState(s_start); InitState(s_goal));g(s_start) = 0;
36  OPEN = TRUNCATED = ∅;
37  insert s_start into OPEN with key(s_start) as priority;
38  forever
39   ComputePath(ε); ObtainPath and publish solution;
40   CHANGED = ∅; move states from TRUNCATED to CHANGED;

41   wait for changes in edge costs;
42   for each directed edges (u, v) with changed edge costs
43    update the edge cost c(u, v); insert v in CHANGED;
44   for each v ∈ CHANGED
45    if (v ≠ s_start) AND (v was visited before)
46     bp(v) = argmin_(s'∈Pred(v)) v(s') + c(s', v);
47     g(v) = v(bp(v)) + c(bp(v), v);
48     UpdateSetMembership(v);
```

Figure 6: Truncated LPA* with suboptimality bound $\epsilon$

same *bp*- pointers as ComputeGpi and returns a path of cost $g^\pi(s_{goal})$.

The Main computes the $\epsilon$-suboptimal solutions by repeatedly calling ComputePath if there is any change in the edge costs. After each ComputePath invocation, the states in $TRUNCATED$ are moved to $CHANGED$. The states that are affected by the cost changes are also put in $CHANGED$. For all these states the $g$- and $bp$- values are recomputed following **Invariant 1** (lines 44-47, Figure 6) and the resulting inconsistent states are put back to $OPEN$ ensuring **Invariant 2** (line 48, Figure 6). As the key computation remains exactly the same as LPA*, **Invariant 3** is always maintained.

## Theoretical Properties

In (Aine and Likhachev 2013) we prove a number of properties of Truncated LPA*, here we state the most important of these theorems.

**Theorem 1.** *When the ComputePath function exits the following holds*

1. *For any state $s$ with $(c^*(s, s_{goal}) < \infty \bigwedge v(s) \geq g(s) \bigwedge key(s) \leq key(u), \forall u \in OPEN)$, $g(s) \leq g^*(s)$ and $g^\pi(s) + h(s) \leq \epsilon * (g^*(s) + h(s))$.*
2. *For any state $s$ with $(c^*(s, s_{goal}) < \infty \bigwedge v(s) < g(s) \bigwedge key(s) \leq key(u), \forall u \in OPEN)$, $v(s) \leq g^*(s)$ and if $s \in TRUNCATED$, then $g^\pi(s) + h(s) \leq \epsilon * (g^*(s) + h(s))$.*
3. *The cost of the path from $s_{start}$ to $s_{goal}$ obtained using the ObtainPath routine is no larger than $\epsilon * g^*(s_{goal})$.*

Theorem 1 states the $\epsilon$-suboptimality of TLPA*. The suboptimality guarantee stems from the facts that whenever a state is expanded as an overconsistent state or truncated, then a) the minimum of the $g$- and $v$- value remains a lower bound on the optimal path cost from $s_{start}$ to $s$, and b) the paths stored for truncated states ensure that the actual path costs are never larger than the lower bound estimate by more than the $\epsilon$ factor. In addition, Theorem 2 shows that TLPA* retains the efficiency properties of LPA*.

**Theorem 2.** *No state is expanded more than twice during the execution of the ComputePath function.*

## Optimizations

While the pseudocodes presented in Figures 5 and 6 are correct, TLPA* can be further optimized in few obvious ways. In the following, we describe the optimizations that have been incorporated in the current TLPA* implementation.

- In ComputePath, $g^\pi(s_{goal})$ is computed before each expansion. This becomes unnecessary if the current path to $s_{goal}$ has not been altered during the previous expansion. We mark the states in the current path from $s_{start}$ to $s_{goal}$, and call ComputeGpi($s_{goal}$) only if any state in this path has been updated.
- While computing the $g^\pi$ for any state $s$ other than $s_{goal}$, we terminate the computation if the cost is $> \epsilon * (v(s) + h(s))$ and set $g^\pi(s) = \infty$, as the check on line 24, Figure 6 will never be satisfied for $s$.
- The StorePath routine is not called separately from ComputeGpi as it traverses the same backpointers. The pointers are stored in the ComputeGpi routine itself. If $g^\pi(s)$ becomes more than $\epsilon * (v(s) + h(s))$ at any point, the stored pointers are discarded.
- Once a state $s$ is truncated it is not put back in $OPEN$ within ComputePath. Therefore, we do not update the $g$- and $bp$- values of a truncated state while expanding other states (additional checks at lines 17 and 28, Figure 6). The values are correctly updated in line 47, Figure 6 before they are put back in $OPEN$ for the next iteration.

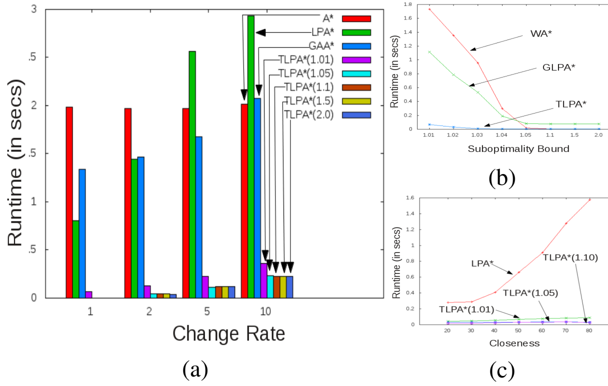



Figure 7: Experiments on 1000x1000 2D grids with random obstacles. (a) compares TLPA* ($\epsilon = 1.01 - 2.0$) with A*, LPA* and GAA*. (b) compares TLPA*, WA* and GLPA* for different $\epsilon$ with $change\ rate = 1$, and (c) compares LPA* with TLPA* ($\epsilon = 1.01, 1.05,$ and $1.10$) for different $closeness$ factors.

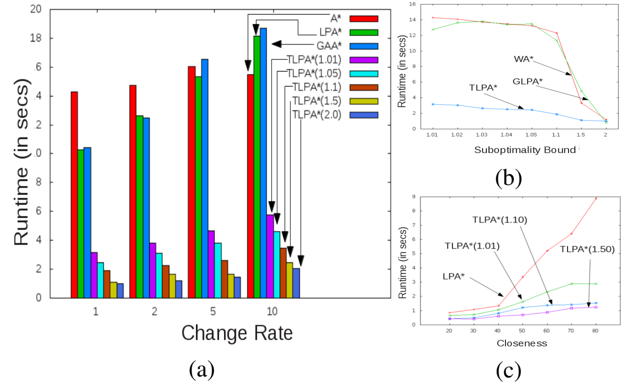



Figure 8: Experiments on 1000x1000 3D lattices. (a) compares TLPA* ($\epsilon = 1.01 - 2.0$) with A*, LPA* and GAA*. (b) compares TLPA*, WA* and GLPA* for different $\epsilon$ bounds with $change\ rate = 1$, and (c) compares LPA* with TLPA* ($\epsilon = 1.01, 1.10,$ and $1.50$) for different $closeness$ factors.

## Experimental Results

We evaluated TLPA* comparing it to various optimal and suboptimal search algorithms for 2D and 3D path planning domains. The optimal algorithms used for comparison are A* (Hart, Nilsson, and Raphael 1968), LPA* (Koenig, Likhachev, and Furcy 2004) and GAA* (Sun, Koenig, and Yeoh 2008) and the suboptimal algorithms used are WA* (Pohl 1970) (without reexpansions) and GLPA* (Likhachev and Koenig 2005). All the experiments were performed on an Intel $i7-3770$ ($3.40GHz$) PC with $16GB$ RAM.

**2D Path Planning :** For this domain, the environments were randomly generated $1000 \times 1000$ 16-connected grids with $10\%$ of the cells blocked. We used Euclidean distances as the heuristics.

For the first experiment, after the first plan, we randomly changed the traversability of $(change\ rate/2)\%$ of cells from blocked to unblocked and an equal number of cells from unblocked to blocked, and replanned. We iterated this for 100 times (for each $change\ rate$) and computed the average runtime per iteration. Figure 7a shows the average replanning runtimes for A*, LPA*, GAA* and TLPA* (with $\epsilon$ ranging from $1.01 - 2.0$) for different change rates ($1 - 10$). The plots show that TLPA* is much more efficient than the optimal incremental algorithms (as well as A*) even for very low suboptimality bounds. For example, with $\epsilon = 1.01$, TLPA* obtains around 11x speedup over LPA* and its speedup over the best optimal algorithm (LPA* is the best for $change\ rate = 1.0, 2.0$; GAA* for $5.0$; and A* for $10.0$) is in the range of $5 - 11$x. For $\epsilon = 1.05$, TLPA* is 35x faster than LPA* and about $11 - 35$x faster than the best optimal algorithm. Figure 7b shows the runtime comparisons for 3-provably suboptimal algorithms, WA*, GLPA* and TLPA* for the same $\epsilon$ ranging from $1.01 - 2.0$. The plots show that while for higher values of $\epsilon$ ($> 1.1$) all the suboptimal algorithms perform equally well, TLPA* converges much faster for lower values of $\epsilon$. For example with $\epsilon = 1.02$, TLPA* is 45x faster than WA* and 23x faster than GLPA*.

For the second experiment, we confined $80\%$ of the total cost changes within a chosen area around the start cell governed by a $closeness$ factor. For example, if $closeness = 20$, $80\%$ of the total changes were placed within an area of $0.2\times$total area of the grid, close to the start state. Figure 7c shows the results for $1000 \times 1000$ grids with $change\ rate = 1$, for $closeness$ factors ranging from 20 to 80. From the results we see that while LPA* is extremely sensitive to the position of changes ($closeness$ factor), TLPA* is relatively more consistent and the consistency improves with larger $\epsilon$ values. For example, LPA* runtime for $closeness = 80$ is 5x more than that for $closeness = 20$, for TLPA* with $\epsilon = 1.01$ the difference is about 2x, and the difference reduces further with $\epsilon \geq 1.05$.

**3D Path Planning :** For the 3D planning, we modeled the environment as a planar world and a polygonal robot with three degrees of freedom: $x$, $y$, and $\theta$ (heading). The search objective is to plan smooth paths for non-holonomic robots, i.e., the paths generated must 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 (Likhachev and Ferguson 2009) used in a lattice-type planner. We computed the consistent heuristics by running a 16-connected 2D Dijkstra search. For the testing, we ran the same experiments as done for 2D path planning.

Figure 8a shows the comparison between A*, LPA*, GAA* and TLPA* (with $\epsilon$ ranging from $1.01 - 2.0$) for different $change\ rates$ ($1-10$). Similar to the results obtained for 2D planning here also TLPA* performs noticeably better than optimal algorithms. On an average, TLPA* obtains around $4 - 11$x speedup over corresponding best algorithm for a particular $change\ rate$. Figure 8b depicts the comparison between WA*, GLPA* and TLPA* for different $\epsilon$ values with $change\ rate = 1.0$. The plots show that for $\epsilon < 2.0$, TLPA* is the best algorithm among the suboptimal ones, however, the performance gap among WA*, GLPA* and TLPA* reduces with the increase in $\epsilon$. For $\epsilon = 2.0$, GLPA* runtime becomes marginally better than TLPA*.

We include the results for the second type of experiment in Figure 8c. We see again that TLPA* is much less sensitive to $closeness$ when compared to LPA*. While LPA* runtime degrades by $11$x with $closeness$ changing from 20 to 80,

TLPA* with $\epsilon = 1.1$ degrades by 2.5x only.

```
1  procedure ComputeGpi(s)
2  visited = ∅; cost = 0; s' = s;
3  while (s' ≠ s_goal)
4    if (s' ∈ visited) OR (bp(s') = null)
5      cost = ∞; break;
6    if (s' ∈ TRUNCATED)
7      cost = cost + g^π(s'); break;
8    insert s' in visited;
9    cost = cost + c(s', bp(s')); s' = bp(s');
10 g^π(s) = cost;
```

Figure 9: ComputeGpi routine for Truncated D* Lite

# Truncated D* Lite

D* Lite (Koenig and Likhachev 2002) is an incremental heuristic search algorithm for goal directed navigation through dynamic graphs, which is widely used in robotics applications. In this section, we present Truncated D* Lite (TD* Lite), a suboptimal replanning algorithm for navigation, which combines the TLPA* truncation rules with D* Lite. We start by explaining the differences between D* Lite and LPA* and then show how to integrate them with the truncation rules.

## D* Lite vs LPA*

The D* Lite algorithm repeatedly determines shortest paths between the current position of the robot and the goal state as the edge costs of a graph change while the robot moves towards the goal state.

D* Lite is directly based on LPA*, with the following three differences,

**Search Direction:** D* Lite switches the search direction from LPA*. The LPA* version presented in Figure 2 searches from $s_{start}$ to $s_{goal}$. Thus, its $g$- and $v$- values are estimates of the start distances. D* Lite searches backwards, from $s_{goal}$ to $s_{start}$, so that the root of the search tree remains static when the robot moves. Thus, in D* Lite, the $g$- and $v$- values are estimates of the goal distances, i.e., for a state $s$, $g(s)$ denotes the distance from $s$ to $s_{goal}$. Similarly, the heuristic values in D* Lite denotes an admissible estimate for the distance between $s_{start}$ and $s$, in contrast to the forward searches (LPA*/TLPA*), where the heuristic denotes an estimate for the goal distance (distance between $s$ and $s_{goal}$). To distinguish the heuristics for D* Lite with the earlier presented algorithms, we use the notation $h(s_{start}, s)$ instead of $h(s)$.

**Movement of the Robot:** After computing the shortest path for a given graph, D* Lite moves the robot along the computed path (by following the $bp$- pointers), as long as the edge costs do not change. If the edge costs change, the shortest path is recomputed using the latest position of the robot as $s_{start}$. The search terminates when $s_{goal}$ is reached.

**Avoiding Heap Reorder:** As D* Lite moves the robot along the path computed, it needs to recalculate the priorities in $OPEN$ every time the robot notices a change in edge costs. Otherwise, the priorities will not satisfy **Invariant 3**, since they were computed with respect to the old position of the robot. However, this repeated reordering of $OPEN$ can be expensive. D* Lite avoids this reordering, by readjusting the key values in a way that the priorities remain a lower bound on LPA* priorities. The priorities of the newly generated states are altered using a variable $k_m$, which is incremented by the value $h(s_{last}, s_{start})$, after each cost change.

## The Truncated D* Lite Algorithm

```
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) = null;
5  procedure UpdateSetMemberShip(s)
6  if (g(s) ≠ v(s))
7    if (s ∉ TRUNCATED) insert/update s in OPEN with key(s) as priority;
8  else if (s ∈ OPEN) remove s from OPEN;
9  procedure ComputePath(ε)
10 while OPEN.Minkey() < key(s_start) OR v(s_start) < g(s_start)
11   s = OPEN.Top();
12   k_old = OPEN.Minkey();
13   if (k_old < key(s))
14     update s in OPEN with key(s) as priority;
15   else
16     ComputeGpi(s_start);
17     if (g^π(s_start) ≤ ε * (min(g(s), v(s)) + h(s_start, s))) return;
18     remove s from OPEN;
19     if (v(s) > g(s))
20       v(s) = g(s);
21       for each s' in Pred(s)
22         if s' was never visited InitState(s');
23         if g(s') > g(s) + c(s', s)
24           g(s') = g(s) + c(s', s); bp(s') = s;
25           UpdateSetMembership(s');
26     else
27       ComputeGpi(s);
28       if (g^π(s) + h(s) ≤ ε * (v(s) + h(s_start, s)))
29         StorePath(s); insert s in TRUNCATED;
30       else
31         v(s) = ∞;  UpdateSetMembership(s);
32         for each s' in Pred(s)
33           if s' was never visited InitState(s');
34           if bp(s') = s
35             bp(s') = argmin_(s''∈Succ(s')) v(s'') + c(s', s'');
36             g(s') = v(bp(s')) + c(s', bp(s'));
37             UpdateSetMembership(s');
38 procedure Main(ε)
39 InitState(s_start); InitState(s_goal); g(s_goal) = 0;
40 k_m = 0; OPEN = TRUNCATED = ∅; s_last = s_start;
41 insert s_goal into OPEN with key(s_goal) as priority;
42 ComputePath();
43 while (s_last ≠ s_goal)
44   s_start = bp(s_start);
45   Move to s_start;
46   Scan graph for changes in edge costs;
47   If any edge costs changed
48     k_m = k_m + h(s_last, s_start); CHANGED = ∅;
49     move states from TRUNCATED to CHANGED;
50     s_last = s_start;
51     for each directed edges (u, v) with changed edge costs
52       update the edge cost c(u, v); insert u in CHANGED;
53     for each v ∈ CHANGED
54       if (v ≠ s_goal) AND (v was visited before)
55         bp(v) = argmin_(s'∈Succ(v)) v(s') + c(v, s');
56         g(v) = v(bp(v)) + c(v, bp(v));
57         UpdateSetMembership(v);
58     ComputePath(ε);
```

Figure 10: Truncated D* Lite with suboptimality bound $\epsilon$

The truncation rules used in TLPA* are completely orthogonal to the differences between for LPA* and D* Lite, and therefore can be trivially applied to D* Lite, to obtain

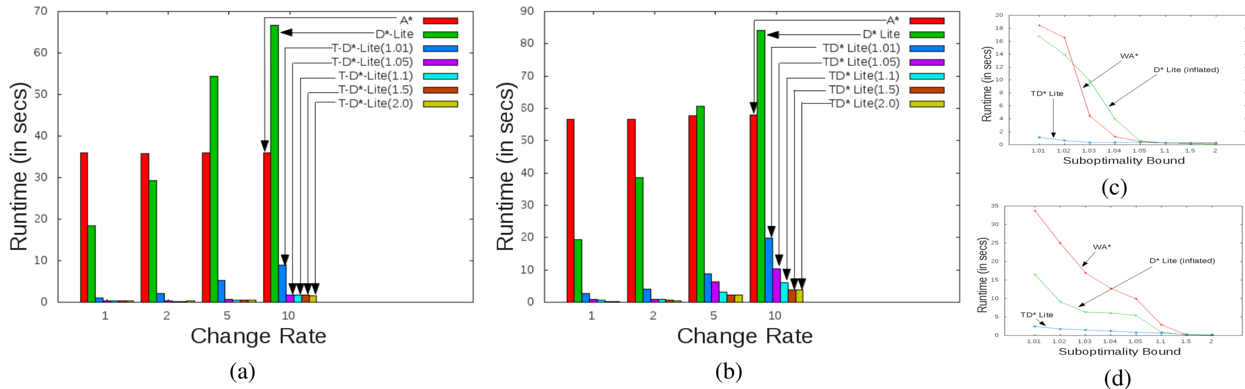



Figure 11: Experiments for navigation problems. (a) and (b) compares TD* Lite ($\epsilon = 1.01 - 2.0$) with A* and D* Lite for 1000x1000 2D grids and 1000x1000 3D lattices, respectively. (c) and (d) compares TD* Lite, WA* and D* Lite (with heuristic inflation) for different $\epsilon$ bounds with $change\ rate = 1$.

a bounded suboptimal algorithm for navigation in dynamic graphs. The Truncated D* Lite (TD* Lite) algorithm exactly does that. It inherits the D* Lite characteristics (backward search, robot movement and avoidance of heap reorder) and augments it by the application of the truncation rules.

In Figures 9 and 10, we present the pseudocode for TD* Lite. The basic differences between TLPA* and TD* Lite are the following:

- The search is performed backwards from $s_{goal}$ to $s_{start}$ (as in D* Lite), and the $g$-, $v$- , and $g^{\pi}$- values are estimates for the goal distances rather than the start distances used in TLPA*. To accommodate this, we modify the ComputeGpi routine (Figure 9), which now follows the backpointers from the current state ($s$) to $s_{goal}$ and returns the corresponding cost as $g^{\pi}(s)$. Similar alterations are done for the other auxiliary routines.
- After a path is discovered, the robot moves along the path following the *bp*- pointers (lines 44-45, Figure 10) as long as the edge costs do not change. If the edge costs change, a $\epsilon$-optimal path is recomputed by calling the ComputePath, in a manner similar to TLPA*.
- The (optional) optimization to avoid heap reorder in D* Lite requires an additional parameter ($k_m$) to alter the key values. After each cost change, $k_m$ is incremented by the heuristic distance between the current $h_{start}$ and $h_{start}$ of the earlier iteration (as stored in $h_{last}$). D* Lite also requires to update the priority of a state selected for expansion ($s$), if its actual priority (as computed by $key(s)$) is more than the priority value used. In TD* Lite, this update is performed before applying any truncation rule to ensure that the priorities remain the same as TLPA* (lines 13-14, Figure 10).
- While the application of truncation rules are exactly same as in TLPA*, the description of Rule 2 for TD* Lite needs to be modified as the $key_1(s)$ values are different from TLPA* (due to the addition of $k_m$). For TD* Lite, the Truncation Rule 2 is applied if for a state $s$ selected for expansion, satisfies $\epsilon * (min(g(s), v(s)) + h(s_{start}, s) \geq g^{\pi}(s_{start})$ (line 17, Figure 10).
- The search terminates when the robot reaches $s_{goal}$ (line 43, Figure 10).

The ComputePath routine of TD* Lite is similar to that of TLPA*, and thus, they share most of the basic properties. In particular, after each iteration of ComputePath the path (from $s_{start}$ to $s_{goal}$) computed by the ObtainPath routine has cost $\leq \epsilon * g^{*}(s_{start})$ (Theorem 1) and no state is expanded more than twice in the ComputePath function (Theorem 2).

## Experimental Results with TD* Lite

We performed similar experiments as done for TLPA*, comparing TD* Lite with optimal algorithms, namely, A* and D* Lite and bounded suboptimal algorithms, namely, WA* (without reexpansions) and D* Lite (with heuristic inflation), for 2D/3D environments. The original map is given to the robot before the first search. After a path is planned, the robot moves along the path until the environment changes. We randomly change the edge costs (depending on the change rate) after each 10 moves made by the robot. Once the costs change, the robot replans the path using the current start and goal state. This process is repeated until the goal is reached. As the number of moves required to reach the goal may vary for different suboptimal algorithms, here we report the average total planning time instead of the average time required for each planning episode (as done earlier).

The results are included in Figure 11. The plots depict a similar trend to that obtained with TLPA*, i.e., TD* Lite achieves a significant improvement in runtime over the optimal algorithms even for low suboptimality bounds. For example, with $\epsilon = 1.01$, TD* Lite obtains around 13x speedup over D* Lite for 2D and 7x speedup for 3D. Comparing bounded suboptimal algorithms, we observe that TD* Lite is much faster when searching for close-to-optimal solutions ($\epsilon = 1.01 - 1.1$), while the algorithms perform similarly for higher bounds.

## Conclusions and Future Work

We have presented Truncated Incremental Search, a replanning framework that uses a suboptimality bound to limit the propagation of cost changes. We have used this framework to develop two incremental search algorithms, TLPA* and TD* Lite. Experimental results on two path planning domains show the efficacy of this framework over state-of-the-art incremental algorithms, especially when searching for close-to-optimal solutions.

Currently, TLPA* and TD* Lite work with consistent heuristics only. In future, we would like to study whether they can be extended to work with inconsistent/inflated heuristics and how it can be used to make them anytime.

## References


Aine, S., and Likhachev, M. 2013. Truncated LPA* : The Proofs. Technical Report TR-12-32, Carnegie Mellon University, Pittsburgh, PA.

Ferguson, D., and Stentz, A. 2006. Using interpolation to improve path planning: The field D* algorithm. *J. Field Robotics* 23(2):79–101.

Fox, M.; Gerevini, A.; Long, D.; and Serina, I. 2006. Plan Stability: Replanning versus Plan Repair. In Long, D.; Smith, S. F.; Borrajo, D.; and McCluskey, L., eds., *ICAPS*, 212–221. AAAI.

Gerevini, A., and Serina, I. 2000. Fast Plan Adaptation through Planning Graphs: Local and Systematic Search Techniques. In Chien, S.; Kambhampati, S.; and Knoblock, C. A., eds., *AIPS*, 112–121. AAAI.

Gerevini, A.; Saetti, A.; and Serina, I. 2003. Planning Through Stochastic Local Search and Temporal Action Graphs in LPG. *J. Artif. Intell. Res. (JAIR)* 20:239–290.

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

Koenig, S., and Likhachev, M. 2002. D* Lite. In Dechter, R., and Sutton, R. S., eds., *AAAI/IAAI*, 476–483. AAAI Press / The MIT Press.

Koenig, S., and Likhachev, M. 2005. Adaptive A*. In Dignum, F.; Dignum, V.; Koenig, S.; Kraus, S.; Singh, M. P.; and Wooldridge, M., eds., *AAMAS*, 1311–1312. ACM.

Koenig, S.; Likhachev, M.; and Furcy, D. 2004. Lifelong Planning A*. *Artif. Intell.* 155(1-2):93–146.

Likhachev, M., and Ferguson, D. 2009. Planning Long Dynamically Feasible Maneuvers for Autonomous Vehicles. *I. J. Robotic Res.* 28(8):933–945.

Likhachev, M., and Koenig, S. 2005. A Generalized Framework for Lifelong Planning A* Search. In Biundo, S.; Myers, K. L.; and Rajan, K., eds., *ICAPS*, 99–108. AAAI.

Likhachev, M.; Ferguson, D.; Gordon, G. J.; Stentz, A.; and Thrun, S. 2008. Anytime search in dynamic graphs. *Artif. Intell.* 172(14):1613–1643.

Pohl, I. 1970. Heuristic Search Viewed as Path Finding in a Graph. *Artif. Intell.* 1(3):193–204.

Ramalingam, G., and Reps, T. W. 1996. An Incremental Algorithm for a Generalization of the Shortest-Path Problem. *J. Algorithms* 21(2):267–305.

Stentz, A. 1995. The Focussed D* Algorithm for Real-Time Replanning. In *IJCAI*, 1652–1659. Morgan Kaufmann.

Sun, X., and Koenig, S. 2007. The Fringe-Saving A* Search Algorithm - A Feasibility Study. In Veloso, M. M., ed., *IJCAI*, 2391–2397.

Sun, X.; Koenig, S.; and Yeoh, W. 2008. Generalized Adaptive A*. In Padgham, L.; Parkes, D. C.; Müller, J. P.; and Parsons, S., eds., *AAMAS (1)*, 469–476. IFAAMAS.

Trovato, K. I., and Dorst, L. 2002. Differential A*. *IEEE Transactions on Knowledge and Data Engineering* 14(6):1218–1229.