Anytime Truncated D* : Anytime Replanning with Truncation

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

Abstract

Incremental heuristic searches reuse their previous search efforts to speed up the current search. Anytime search algorithms iteratively tune the solutions based on available search time. Anytime D* (AD*) is an incremental anytime search algorithm that combines these two approaches. AD* uses an inflated heuristic to produce bounded suboptimal solutions and improves the solution by iteratively decreasing the inflation factor. If the environment changes, AD* recomputes a new solution by propagating the new costs. Recently, a different approach to speed up replanning (TLPA*/TD* Lite) was proposed that relies on selective truncation of cost propagations instead of heuristic inflation. In this work, we present an algorithm called Anytime Truncated D* (ATD*) that combines heuristic inflation with truncation in an anytime fashion. We develop truncation rules that can work with an inflated heuristic without violating the completeness/suboptimality guarantees, and show how these rules can be applied in conjunction with heuristic inflation to iteratively refine the replanning solutions with minimal reexpansions. We explain ATD*, discuss its analytical properties and present experimental results for 2D and 3D (x, y, heading) path planning demonstrating its efficacy for anytime replanning.

Introduction

Planning for systems operating in the real world involves dealing with two major challenges, namely, uncertainty and complexity. The real world is an inherently uncertain and dynamic place; accurate models for planning are difficult to obtain and quickly become out of date, and the planner needs to modify its solution when such a change is perceived. Incremental search algorithms such as LPA* (Koenig, Likhachev, and Furcy 2004), D* Lite (Likhachev and Koenig 2005), Field D* (Ferguson and Stentz 2006) attempt to efficiently cope with such dynamic environments. These algorithms reuse the information from previous search iterations to generate the optimal solution for the current iteration and can converge faster when compared to planning from scratch. For complex planning problems, it is often desirable to obtain a trade-off between the solution quality and the runtime. Anytime search algorithms (such as AWA* (Zhou and Hansen 2002), ARA* (Likhachev, Gordon, and Thrun 2004), and etc) are very useful for such systems, as they usually provide an initial, possibly highly-suboptimal solution very quickly and then iteratively improve this solution depending on the deliberation time in hand.

Anytime D* (AD*) (Likhachev et al. 2008) is an anytime replanning algorithm that combines the benefits of incremental search (D* Lite) and anytime search (ARA*). AD* uses an inflated heuristic to produce a fast bounded suboptimal solution, and then continually improves the solution by searching with decreasing inflation factor. If the environment changes, AD* corrects its solution in an incremental manner by propagating the cost changes. AD* is widely used in real life robotics systems. For example, it was used in the DARPA Urban Challenge winner vehicle (2007).

While AD* has been a successful integration of incremental and anytime approaches, it suffers from two problems. Firstly, as AD* uses heuristic inflation to speed up planning, its efficacy is very much dependent on the heuristic accuracy. Also, while AD* works exceedingly well for high inflation factors, its convergence time increases considerably when searching for close-to-optimal solutions. Secondly, in AD*, heuristic inflation is only used for states for which the path has improved from the previously computed values (overconsistent states) whereas the states for which the path has degraded (underconsistent states) use uninflated heuristics. This can result in accumulation of the underconsistent states at the top of the queue, resulting in performance deterioration (Likhachev et al. 2008).

Recently, a new method called *truncation* was proposed for improving the replanning runtime while maintaining suboptimality guarantees (Aine and Likhachev 2013b). The basic idea of truncation is to use a target suboptimality bound to restrict the replanning cost propagations, when such propagations are not necessary to guarantee the chosen bound. The truncation based algorithms (TLPA*/TD* Lite) have been shown to be very effective when searching for close-to-optimal solutions. Also, this method is especially useful for handling underconsistent states, as it can efficiently limit underconsistent state expansions by truncating an underconsistent state when a *good enough* (depending

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.

on the target bound) path to it has been discovered.

In this work, we explore the possibility of combining heuristic inflation (AD*) with truncation (TD* Lite) to develop an anytime replanning algorithm. Combination of AD* with TD* Lite leads to an exciting proposition, as these techniques approach the replanning problem from different directions and are shown to be efficient for different parts of the anytime incremental spectrum. For example, AD* speeds up planning with inflated heuristics while TD* Lite speeds up replanning with selective truncation, AD* works very well for high suboptimality bounds whereas TD* Lite is more effective for close-to-optimal bounds, AD* may suffer from accumulation of underconsistent states while TD* Lite can efficiently truncate such states.

Unfortunately, these approaches can not be combined directly, as the truncation rules used for TD* Lite only work with consistent heuristics. To rectify this, we design *new* truncation rules that follow the same principle used in TD* Lite but can work with inflated heuristics without violating the completeness/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 speed up *planning* and *replanning*, offering greater efficacy and flexibility to solve complex dynamic planning problems under limited time. Also, ATD* ensures minimal reexpansion of states between anytime/incremental iterations.

We present the theoretical properties of ATD* demonstrating its completeness and suboptimal termination. We also show that ATD* retains the expansion efficiency of AD*. We experimentally evaluate ATD* for two domains, 2D and 3D (x, y, heading) path planning, comparing it with the state-of-the-art anytime incremental algorithm (AD*) and a widely used anytime search algorithm in robotics (ARA*).

Related Work

The incremental heuristic search algorithms found in the 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. 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.

Another group of incremental searches (Sun, Yeoh, and Koenig 2010; Sun et al. 2012) focus on solving moving target search problems in static environments, i.e., these algorithms incrementally replan paths with changes in start/goal configurations but do not accommodate changes in the edge costs.

Over the years, a large number of anytime search algorithms have been proposed in the AI literature. In general, these algorithms use a depth-first bias to guide the search toward a quick (possibly suboptimal) termination and iteratively relax this bias to improve the solution quality. A majority of such algorithms (such as AWA* (Zhou and Hansen 2002), ARA* (Likhachev, Gordon, and Thrun 2004), RWA* (Richter, Thayer, and Ruml 2010)) are based on 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. Other anytime approaches include searches that restrict the set of states that can be expanded (BeamStack search (Zhou and Hansen 2005), Anytime Window A* (Aine, Chakrabarti, and Kumar 2007)), and searches that use different cost/distance estimates to guide the search and best first heuristic estimates to provide bounds (AEES (Thayer, Benton, and Helmert 2012)).

As discussed earlier, Anytime D* (AD*) (Likhachev et al. 2008) combines the anytime approach of ARA* (Likhachev, Gordon, and Thrun 2004) with the incremental replanning of D* Lite (Likhachev and Koenig 2005). TLPA* (and TD* Lite), on the other hand, is a bounded suboptimal replanning algorithm that relies on efficient truncation of the cost propagations. Both of these algorithms are based on LPA*, and thus belong to the first category of incremental search.

Background

```
1  procedure key(s)
2  return [min(g(s), v(s)) + h(s_start, 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 InitState(s);
7  if (s ≠ s_goal)
8   bp(s) = argmin_{(s'' ∈ Succ(s))} v(s'') + c(s, s'');
9   g(s) = v(bp(s)) + c(s, bp(s));
10 if (g(s) ≠ v(s))  insert/update s in OPEN with key(s) as priority;
11 else if s ∈ OPEN  remove s from OPEN;
12 procedure ComputePath()
13 while OPEN.Minkey() < key(s_start) OR v(s_start) < g(s_start)
14  s = OPEN.Top(); remove s from OPEN;
15  if (v(s) > g(s))
16   v(s) = g(s); for each s' in Pred(s)  UpdateState(s');
17  else
18   v(s) = ∞; for each s' in Pred(s)∪s  UpdateState(s');
19 procedure Main()
20 InitState(s_start); InitState(s_goal); g(s_goal) = 0;
21 OPEN = ∅; insert s_goal into OPEN with key(s_goal) as priority;
22 forever
23  ComputePath();
24  Scan graph for changes in edge costs;
25  If any edge costs changed
26   for each directed edges (u, v) with changed edge costs
27    update the edge cost c(u, v);  UpdateState(u);
```

Figure 1: LPA*/D* Lite (searching backwards)

D* Lite

In this section, we briefly describe three algorithms, D* Lite (Likhachev and Koenig 2005), AD* (Likhachev et al. 2008), and TD* Lite (Aine and Likhachev 2013b), that form the backbone of the work presented in this paper.

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. $c^*(s, s')$ denotes optimal path cost between s to s' and $g^*(s)$ denotes $c^*(s, s_{goal})$. All the algorithms presented in this work search backward, i.e., from s_{goal} (root) to s_{start} (leaf).

D* Lite 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[1]. It maintains two kinds of estimates of the cost of a path from s to s_{goal} for each state s: $g(s)$ and $v(s)$. $v(s)$ holds the cost of the best path found from s to s_{goal} during its last expansion while $g(s)$ is computed from the v-values of its successors (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 successor of s (if computed). D* Lite always satisfies the following relationships: $bp(s_{goal}) = \textbf{null}, g(s_{goal}) = 0$ and $\forall s \in S - \{s_{goal}\}$, $bp(s) = \text{argmin}_{(s' \in Succ(s))} v(s') + c(s, s'), g(s) = v(bp(s)) + c(s, bp(s'))$ (**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 $v(s) < g(s)$). D* Lite uses a consistent heuristic $h(s_{start}, s)$ 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 $key_1(s) = min(g(s), v(s)) + h(s_{start}, s)$ 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 D* Lite is shown in Figure 1. The algorithm starts by initializing the states and inserting s_{goal} into $OPEN$. 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_{start}. If a state s is overconsistent, ComputePath makes it consistent by setting $v(s) = g(s)$ (line 16) and propagates this information to its predecessors by updating their g-, v- and bp- values according to **Invariant 1**. This may make some $s' \in Pred(s)$ 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 18) and propagates the underconsistency information to its predecessors (again ensuring **Invariant 1**), and selectively puts s and its predecessors 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 ∞, $\forall s \in S$. 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, D* Lite updates the g- and bp- values of the affected states by calling the UpdateState function 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). D* Lite then calls ComputePath again to fix these inconsistencies until a new optimal path is computed.

```
1  procedure key(s)
2  if v(s) ≥ g(s)
3   return [g(s) + ϵ * h(s_start, s); g(s))];
4  else
5   return [v(s) + h(s_start, s); v(s))];
6  procedure UpdateState(s)
7  if s was never visited InitState(s);
8  if (s ≠ s_goal)
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))
12  if (s ∉ CLOSED)  insert/update s in OPEN with key(s) as priority;
13  else if (s ∉ INCONS)  insert s in INCONS;
14 else
15  if (s ∈ OPEN) remove s from OPEN;
16  else if (s ∈ INCONS) remove s from INCONS;
17 procedure ComputePath()
18 while OPEN.Minkey() < key(s_start) OR v(s_start) < g(s_start)
19  s = OPEN.Top();  remove s from OPEN;
20  if (v(s) > g(s))
21   v(s) = g(s); for each s' in Pred(s)  UpdateState(s');
22  else
23   v(s) = ∞; for each s' in Pred(s)∪s  UpdateState(s');
24 procedure Main()
25 InitState(s_start); InitState(s_goal); g(s_goal) = 0; ϵ = ϵ0;
26 OPEN = ∅; insert s_goal into OPEN with key(s_goal) as priority;
27 ComputePath(); Publish solution;
28 forever
29  if changes in edge costs are detected
30   for each directed edges (u, v) with changed edge costs
31    update the edge cost c(u, v); UpdateState (u);
32  if ϵ > 1.0  decrease ϵ;
33  move states from INCONS into OPEN;
34  update the priorities ∀s ∈ OPEN according to key(s);
35  CLOSED = ∅;
36  ComputePath(); Publish solution;
37  if (ϵ = 1.0)  wait for changes in edge costs;
```

Figure 2: Anytime D*

Anytime D*

Anytime D* (AD*) combines ARA* (Likhachev, Gordon, and Thrun 2004) with D* Lite. It 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 locally affected inconsistent states into $OPEN$ and recomputes a path of bounded suboptimality by propagating the cost changes.

The pseudocode of a basic version of AD* is shown in Figure 2. The Main function first sets the inflation factor to a chosen value (ϵ_0) and generates an initial suboptimal plan. Then, unless changes in edge costs are detected, the Main

[1] All the algorithms described in this paper are capable of handling the movement of the agent along the path computed, by updating the s_{start} dynamically, as discussed in (Likhachev and Koenig 2005).

function decreases the ϵ bound and improves the quality of its solution until it is guaranteed to be optimal (lines 32-36, Figure 2). This part of AD* is an exact match with ARA*.

When changes in edge costs are detected, there is a chance that the current solution will no longer be ϵ-suboptimal. AD* updates the costs of affected states following **Invariant 1** and puts the inconsistent states in $OPEN$ (**Invariant 2**). The cost change information is then propagated by expanding states in $OPEN$, until there is no state in $OPEN$ with a key value less than that of s_{start} and s_{start} itself is not underconsistent (similar to D* Lite).

The handling of keys (for inconsistent states) in AD* is different than D* Lite. AD* uses inflated heuristic values for the overconsistent states (line 3, Figure 2) to provide a depth-first bias to the search. However, for the underconsistent states, it uses uninflated heuristic values (line 5, Figure 2), in order to guarantee that the underconsistent states correctly propagate their new costs to the affected neighbors. By incorporating this dual mechanism, AD* can handle changes in the edge costs as well as changes to the inflation factor.

```
1  procedure StorePath(s)
2  path(s) = ∅;
3  while (s ≠ s_goal) AND (bp(s) ∉ TRUNCATED)
4   insert bp(s) in path(s); s = bp(s);
5  procedure ObtainPathfromTruncated(s)
6  if path(s) = null return;
7  else append path(s) to Path;
8  ObtainPathfromTruncated(end state of path(s))
9  procedure ObtainPath
10  s = s_start; insert s in Path;
11  while (s ≠ s_goal)
12   if bp(s) ∈ TRUNCATED
13    ObtainPathfromTruncated (s); return;
14   insert bp(s) in Path; s = bp(s);
15  procedure ComputeGpi(s)
16  visited = ∅; cost = 0; s' = s;
17  while (s' ≠ s_goal)
18   if (s' ∈ visited) OR (bp(s') = null)
19    cost = ∞; break;
20   if (s' ∈ TRUNCATED)
21    cost = cost + g^π(s'); break;
22   insert s' in visited; cost = cost + c(s', bp(s')); s = bp(s');
23  g^π(s') = cost;
```

Figure 3: Auxiliary routines for TD* Lite

The suboptimality guarantee of AD* is derived from the fact that whenever an overconsistent state (say s) is selected for expansion, a) $g(s) \leq \epsilon * g^*(s)$, and b) the path constructed by greedily following the bp- pointers from s to s_{goal}, is guaranteed to have cost $\leq g(s)$. Thus, when s_{start} has the minimum key in $OPEN$ and $v(s_{start}) \geq g(s_{start})$, the path from s_{start} to s_{goal} has cost $\leq \epsilon * g^*(s_{start})$.

Truncated D* Lite

While AD* speeds up *planning* by using inflated heuristics and performs an unrestricted cost propagation for replanning, TD* Lite adopts a completely orthogonal approach. It uses a consistent heuristic to do the *planning* but speeds up *replanning* by selectively truncating the cost propagations. TD* Lite only propagates the cost changes when it is essential to ensure the suboptimality bound and reuses the previous search values for all other states.

```
1  procedure key(s)
2  return [min(g(s), v(s)) + h(s_start, s); min(g(s), v(s))];
3  procedure InitState(s)
4  v(s) = g(s) = g^π(s) = ∞; bp(s) = null;
5  procedure UpdateState(s)
6  if s was never visited InitState(s);
7  if (s ≠ s_goal)
8   bp(s) = argmin_{(s''∈Succ(s))} v(s'') + c(s, s'');
9   g(s) = v(bp(s)) + c(s, bp(s));
10  if (g(s) ≠ v(s))
11   if (s ∉ TRUNCATED) insert/update s in OPEN with priority key(s);
12  else if s ∈ OPEN  remove s from OPEN;
13  procedure ComputePath(ϵ)
14  while OPEN.Minkey() < key(s_start) OR v(s_start) < g(s_start)
15   s = OPEN.Top();
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); for each s' in Pred(s)  UpdateState (s');
21   else
22    ComputeGpi(s);
23    if (g^π(s) + h(s_start, s) ≤ ϵ * (v(s) + h(s_start, s)))
24     StorePath(s); insert s in TRUNCATED;
25    else
26     v(s) = ∞;  for each s' in Pred(s)∪s  UpdateState(s');
27  procedure Main(ϵ)
28  InitState(s_start); InitState(s_goal); g(s_goal) = 0;
29  OPEN = TRUNCATED = ∅;
30  insert s_goal into OPEN with key(s_goal) as priority;
31  forever
32   ComputePath(ϵ);
33   Scan graph for changes in edge costs;
34   If any edge costs changed
35    CHANGED = TRUNCATED; TRUNCATED = ∅;
36    for each directed edges (u, v) with changed edge costs
37     update the edge cost c(u, v); insert u in CHANGED;
38    for each v ∈ CHANGED  UpdateState(v);
```

Figure 4: TD* Lite

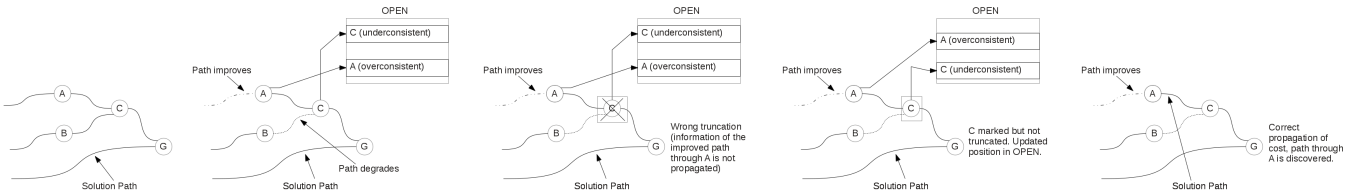
In addition to g- and v- values, TD* Lite computes another goal distance estimate for each state s, called $g^\pi(s)$. $g^\pi(s)$ denotes cost of the path from s to s_{goal} computed by following the current bp- pointers.

TD* Lite uses these g^π- values in two ways, i) for the underconsistent states, g^π values are used to decide whether a state has already discovered a path through it that satisfies the chosen suboptimality bound (ϵ) and if so, then that state is truncated (removed from $OPEN$ without expansion), and ii) before each expansion, $g^\pi(s_{start})$ is computed to decide whether the current path from s_{start} to s_{goal} can be improved by more than the ϵ-factor by continuing the search, and if not, then the search iteration is terminated. Formally, the truncation rules for TD* Lite are described by the following statements.

Rule 1. *An underconsistent state s having $key(s) \leq key(u), \forall u \in OPEN$ is truncated if $g^\pi(s) + h(s_{start}, s) \leq \epsilon * (v(s) + h(s_{start}, s))$.*

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

A basic version of the TD* Lite algorithm is presented in Figures 3 and 4. The Main function computes ϵ-optimal



(a) Original Graph (b) After cost changes (c) Wrong truncation (d) Two step policy (e) Correct cost propagation

Figure 5: Truncation with inflated heuristics

solutions by repeatedly calling ComputePath. After each ComputePath invocation, the states in $TRUNCATED$ are moved to $CHANGED$ (line 35, Figure 4). The states that are affected by the cost changes are also put in $CHANGED$ (line 37, Figure 4). For all these states the g- and bp- values are recomputed following **Invariant 1** and the resulting inconsistent states are put back to $OPEN$ ensuring **Invariant 2**. As the key computation remains exactly the same as D* Lite, **Invariant 3** is always maintained.

The ComputePath function uses the g^{π}- values to apply the truncation rules. Before each expansion, $g^{\pi}(s_{start})$ (line 16, Figure 4) is computed to check whether Rule 2 can be applied. If the check at line 17, Figure 4 is satisfied, TD* Lite terminates with solution cost $= g^{\pi}(s_{start})$. 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 22, Figure 4) to check whether Rule 1 can be applied. If the check at line 23, Figure 4 is satisfied, ComputePath truncates s (puts s into $TRUNCATED$) after storing the current path. Apart from the application of truncation rules, the expansion of states is similar to D* Lite, the only difference being that a truncated state is never reinserted into $OPEN$ during the current iteration (line 11, Figure 4).

The suboptimality guarantee of TD* Lite is derived from the fact that whenever a state s has $key(s) \leq OPEN.Minkey()$, a) its $min(g(s), v(s)) \leq g^*(s)$, and b) if $v(s) \geq g(s)$ or $s \in TRUNCATED$, the path computed by the ObtainPath routine has cost $\leq \epsilon * min(g(s), v(s)) + (\epsilon - 1) * h(s_{start}, s)$. As $h(s_{start}, s_{start}) = 0$, when ComputePath exits, the path cost from the s_{start} to s_{goal} returned by the ObtainPath routine is $\leq \epsilon * g^*(s_{start})$.

Anytime Truncated D*

In this section, we formally describe the Anytime Truncated D* (ATD*) algorithm and discuss its properties. We start by explaining new truncation rules in comparison to the rules used in TD* Lite.

Truncation Rules with Inflated Heuristics

As discussed earlier, AD* and TD* Lite use completely orthogonal approaches to obtain bounded suboptimal solutions. For 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, whereas for TD* Lite, the estimates are always a lower bound on g^* while the actual path costs lie within the chosen bound of this estimate. Thus, it may seem that we can combine them seamlessly using two suboptimality bounds (say ϵ_1 and ϵ_2). If the path estimates are within ϵ_1 bound of the optimal cost and the actual path costs are within ϵ_2 bound of the path estimates, then we can guarantee that the final path cost lies within $\epsilon_1 * \epsilon_2$ of the optimal cost.

However, this approach only works for the truncation of the overconsistent states (states for which $v(s) > g(s)$) and not for the truncation of the underconsistent states. This is due the fact that in AD*, heuristic values for the 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 $g(s_1) \leq \epsilon_1 * g^*(s_1)$, but when an underconsistent state (s_2) is selected for expansion, there is no guarantee that $v(s_2) \leq \epsilon_1 * g^*(s_2)$.

In Figure 5, we present an example of this phenomenon. After the first iteration, a) the path to C through B degrades and C becomes underconsistent, and b) the path to A improves making A overconsistent (Figure 5b). In $OPEN$, C lies above A, as A uses an inflated heuristic and C does not, although there is a better path to C through A. Now, if C is truncated (say by Rule 1), information about the improved path through A will not be propagated to G. This may cause a bound violation $v(C) > \epsilon_1 * g^*(C)$ (Figure 5c) [2].

Therefore, we can not the guarantee the suboptimality bound $\epsilon_1 * \epsilon_2$, if we combine the TD* Lite truncation rules with AD*, due the differential handling of keys in AD*. On the other hand, if we inflate the heuristic functions for both overconsistent and underconsistent states, then AD* is no longer guaranteed to be either complete or bounded suboptimal (Likhachev and Koenig 2005). To overcome this problem, we propose a two step method for truncating underconsistent states in ATD*. In the following, we describe this method by highlighting the difference between the *new* truncation rules and the corresponding TD* Lite rules.

Truncation Rule 1: As noted earlier, Rule 1 is applicable for the underconsistent states only. TD* Lite truncates the cost propagation for an underconsistent state s (selected for expansion), if $g^{\pi}(s) + h(s_{start}, s) \leq \epsilon * (v(s) + h(s_{start}, s))$. In ATD*, when an underconsistent state s is selected for expansion for the first time (in a ComputePath iteration), we compute its g^{π}- value and check whether $g^{\pi}(s) + h(s_{start}, s) \leq \epsilon_2 * (v(s) + h(s_{start}, s))$ similar to TD* Lite. However, we do not truncate s immediately if this check is true. Instead, we mark s as a state that can be potentially

[2] This does not violate the suboptimality guarantee of AD*, as AD* forces an underconsistent state (s) to become overconsistent making $v(s) = \infty$, and if s is later expanded as an overconsistent state, $g(s) \leq \epsilon_1 * g^*(s)$ is guaranteed.

truncated (we set a variable $mark(s) = true$), postpone its cost propagation, and update its position in $OPEN$ by altering its key value from $key(s) = [v(s)+h(s_{start}, s); v(s)]$ to $key(s) = [v(s) + \epsilon_1 * h(s_{start}, s); v(s)]$ (**Step 1**). If an underconsistent state s marked earlier is selected for expansion again (i.e., 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 (depending on ϵ_2), on other hand, we cover for the fact that at this point $v(s)$ may be $\geq \epsilon_1 * g^*(s)$. The updated $key(s)$ guarantees that if later s is selected for expansion as underconsistent state then $v(s) \leq \epsilon_1 * g^*(s)$ (otherwise $v(s) > g(s)$), and thus can be truncated without violating the bounds.

This behavior is depicted in Figures 5d and 5e. After C is marked for truncation, its position in $OPEN$ is updated using an inflated key (Figure 5d). As there is as better path to C through A, expansion of A will pass this information by updating $g(C)$, so that $g(C) \leq \epsilon_1 * g^*(C)$. Now, if $v(C) > \epsilon_1 * g^*(C)$, this will convert C into an overconsistent state ($v(C) > g(C)$) ensuring that this information will propagate to G (Figure 5e). Below, we formally state Truncation Rule 1 for ATD*.

ATD* Rule 1. *An underconsistent state s having $key(s) \leq key(u), \forall u \in OPEN$ and $mark(s) = false$ is marked for truncation if $g^\pi(s) + h(s_{start}, s) \leq \epsilon_2 * (v(s) + h(s_{start}, s))$, and its key value is changed to $key(s) = [v(s) + \epsilon_1 * h(s_{start}, s); v(s)]$. An underconsistent state s having $key(s) \leq key(u), \forall u \in OPEN$ and $mark(s) = true$ is truncated.*

Truncation Rule 2: Rule 2 (for TD* Lite) is applicable for both underconsistent and overconsistent states. In ATD*, we apply this rule in unchanged manner for an overconsistent state. However, for an underconsistent state, we apply Rule 2 only when it has earlier been marked as a state that can be potentially truncated (i.e., it has been selected for expansion with the modified key), as otherwise the bounds can be violated. For ATD*, Rule 2 is formulated in the following statement:

ATD* Rule 2. *A state s having $key(s) \leq key(u), \forall u \in OPEN$ is truncated if $\epsilon_2 * key_1(s) \geq g^\pi(s_{goal})$ and if either $v(s) > g(s)$ or $mark(s) = true$. Also, if any state s is truncated using Rule 2, all states $s' \in OPEN$ are truncated as $\forall s' \in OPEN,\ key_1(s') \geq key_1(s)$.*

ATD* Algorithm

The pseudocode for ATD* is included in Figures 6 and 7. The auxiliary routines used in ATD* (ComputeGpi, ObtainPath, and others) are the same as described in Figure 3. The Main function for ATD* starts with initializing the variables and the suboptimality bounds (lines 2-4, Figure 7). At the start, relatively high values are chosen for both ϵ_1 and ϵ_2, so the first search can converge quickly. The suboptimality bounds are then iteratively reduced (line 24, Figure 7) to search for better quality solutions (as in AD*).

```
1  procedure key(s)
2  if v(s) ≥ g(s)  return [g(s) + ε1 * h(s_start, s); g(s))];
3  else
4   if (mark(s) = true)  return [v(s) + ε1 * h(s_start, s); v(s))];
5   else  return [v(s) + h(s_start, s); v(s))];
6  procedure InitState(s)
7  v(s) = g(s) = g^π(s) = ∞; bp(s) = null; mark(s) = false;
8  procedure UpdateSetMemberShip(s)
9  if (g(s) ≠ v(s))
10  if (s ∉ TRUNCATED)
11   if (s ∉ CLOSED)
12    insert/update s in OPEN with key(s) as priority;
13   else if (s ∉ INCONS)   insert s in INCONS;
14  else
15   if (s ∈ OPEN) remove s from OPEN;
16   else if (s ∈ INCONS) remove s from INCONS;
17  procedure ComputePath()
18  while OPEN.Minkey() < key(s_start) OR v(s_start) < g(s_start)
19   s = OPEN.Top();
20   if (v(s) > g(s))
21    if mark(s) = true
22     mark(s) = false; remove s from MARKED;
23    ComputeGpi(s_start);
24    if (g^π(s_start) ≤ ε2 * (g(s) + h(s_start, s)))  return;
25    else
26     remove s from OPEN;
27     v(s) = g(s);
28     insert s in CLOSED;
29     for each s' in Pred(s)
30      if s' was never visited InitState(s');
31      if g(s') > g(s) + c(s', s)
32       g(s') = g(s) + c(s', s); bp(s') = s;
33       UpdateSetMembership(s');
34   else
35    if mark(s) = true
36     ComputeGpi(s_start);
37     if (g^π(s_start) ≤ ε2 * (v(s) + h(s_start, s)))  return;
38     else
39      remove s from MARKED; mark(s) = false;
40      insert s in TRUNCATED;
41    else
42     ComputeGpi(s);
43     if (g^π(s) + h(s_start, s) ≤ ε2 * (v(s) + h(s_start, s)))
44      StorePath(s); mark(s) = true; insert s in MARKED;
45      UpdateSetMembership(s);
46     else
47      v(s) = ∞;  UpdateSetMembership(s);
48      for each s' in Pred(s)
49       if s' was never visited InitState(s');
50       if bp(s') = s
51        bp(s') = argmin_{(s'' ∈ Succ(s'))} v(s'') + c(s', s'');
52        g(s') = v(bp(s')) + c(s', bp(s'));
53        UpdateSetMembership(s');
```

Figure 6: ComputePath routine for ATD*

After each call of ComputePath, the states in the lists $MARKED$, $TRUNCATED$ and $INCONS$ need to be processed in an efficient manner to ensure minimal reexpansions. If the edge costs do not change and only the suboptimality bounds change before a ComputePath call (check at line 8, Figure 7 returns false), we can reuse the stored paths for truncated states if they still satisfy the bound conditions (lines 27 and 32, Figure 7). These states (that satisfy the bound) are therefore put back to $INCONS$ with $mark(s) = true$. For others, the stored paths are discarded and they are put back to $INCONS$ with $mark(s) = false$. All the states in $INCONS$ are then merged with $OPEN$. If the edge costs of the graph change before a Com-

putePath call (line 8, Figure 7), the states in $MARKED$ and $TRUNCATED$ need to be reevaluated as their old estimates may no longer remain correct. Therefore, after each cost change, the states in $TRUNCATED$ are put to $CHANGED$ and the states in $MARKED$ are discarded (as $MARKED \subset OPEN$). The inconsistent states in $CHANGED$ are put back to $OPEN$ after their costs are updated, maintaining **Invariant 1** and **Invariant 2** (lines 18-22, Figure 7).

```
1  procedure Main()
2  InitState(s_start); InitState(s_goal); g(s_goal) = 0;
3  OPEN = CLOSED = TRUNCATED = INCONS = MARKED = ∅;
4  InitSuboptimalityBounds();
5  insert s_goal into OPEN with key(s_goal) as priority;
6  ComputePath(); ObtainPath and publish solution;
7  forever
8   if changes in edge costs are detected
9    CHANGED = ∅;
10   for each state s ∈ TRUNCATED
11    remove s from TRUNCATED; mark(s) = false;
12    insert s in CHANGED;
13   for each state s ∈ MARKED
14    remove s from MARKED; mark(s) = false;
15   TRUNCATED = MARKED = ∅;
16   for each directed edges (u, v) with changed edge costs
17    update the edge cost c(u, v); insert u in CHANGED;
18   for each v ∈ CHANGED
19    if (v ≠ s_goal) AND (v was visited before)
20     bp(v) = argmin_{(s'∈Succ(v))} v(s') + c(v, s');
21     g(v) = v(bp(v)) + c(v, bp(v));
22     UpdateSetMembership(v);
23   if ε1 > 1.0 OR ε2 > 1.0
24    UpdateSuboptimalityBounds();
25    if MARKED ≠ ∅
26     for each s ∈ MARKED
27      if (g^π(s) + h(s_start, s) ≥ ε2 * (v(s) + h(s_start, s)))
28       remove s from MARKED; mark(s) = false;
29    if TRUNCATED ≠ ∅
30     for each s ∈ TRUNCATED
31      remove s from TRUNCATED; insert s in INCONS;
32      if (g^π(s) + h(s_start, s) ≤ ε2 * (v(s) + h(s_start, s)))
33       insert s in MARKED; mark(s) = true;
34      else
35       mark(s) = false;
36    move states from INCONS into OPEN;
37    update the priorities ∀s ∈ OPEN according to key(s);
38    CLOSED = TRUNCATED = ∅;
39    ComputePath(); ObtainPath and publish solution;
40   if ε1 = 1.0 AND ε2 = 1.0
41    wait for changes in edge costs;
```

Figure 7: Main function for ATD*

The ComputePath function uses the g^π- values to apply the ATD* truncation rules. For an overconsistent state s selected for expansion, $g^\pi(s_{start})$ is computed to check whether Rule 2 can be applied. If the check at line 24, Figure 6 is satisfied, ATD* terminates with solution cost $= g^\pi(s_{start})$. If an underconsistent state s is selected for expansion for the first time ($mark(s) = false$), $g^\pi(s)$ is computed (line 42, Figure 6) to check whether Rule 1 can be applied. If the check at line 43, Figure 6 is satisfied, ComputePath sets $mark(s) = true$ and updates is position in $OPEN$ using the new key value (as computed in line 4, Figure 6). If a marked state s is selected for expansion with $v(s) < g(s)$ (underconsistent), ComputePath either exits (if Rule 2 can be applied) or it truncates s (lines 35-40, Figure 6). On the other hand, if a marked state s is selected for expansion with $v(s) > g(s)$, it is removed from $MARKED$ and $mark(s)$ is set to $false$ before s is processed as a regular overconsistent state (line 22, Figure 6). If ComputePath terminates at line 24 or line 37 (Figure 6), a finite cost path from the s_{start} to s_{goal} having cost $\leq \epsilon_1 * \epsilon_2 * g^*(s_{start})$ can be computed by calling the ObtainPath routine, otherwise no such path exists.

Theoretical Properties

In (Aine and Likhachev 2013a), we prove a number of properties of Anytime Truncated D*. 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 $(c^*(s_{start}, s) < \infty \bigwedge v(s) \geq g(s) \bigwedge key(s) \leq key(u), \forall u \in OPEN)$, $g(s) \leq \epsilon_1 * g^*(s)$ and $g^\pi(s) + h(s_{start}, s) \leq \epsilon_2 * (g(s) + h(s_{start}, s))$.*
2. *For any state s with $(c^*(s_{start}, s) < \infty \bigwedge v(s) < g(s) \bigwedge key(s) \leq key(u), \forall u \in OPEN)$ and $s \in TRUNCATED$, $v(s) \leq \epsilon_1 * g^*(s)$ and $g^\pi(s) + h(s_{start}, s) \leq \epsilon_2 * (v(s) + h(s_{start}, s))$.*
3. *The cost of the path from s_{start} to s_{goal} obtained using the ObtainPath routine is no larger than $\epsilon_1 * \epsilon_2 * g^*(s_{start})$.*

Theorem 1 states the bounded suboptimality of ATD* (bound = $\epsilon_1 * \epsilon_2$). The suboptimality guarantee stems from the fact that using the two step truncation approach, ATD* ensures that whenever a state is expanded in an overconsistent manner or truncated, we have a) the minimum of the g- and v- value remains within ϵ_1 bound on the optimal path cost, 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 ϵ_2 factor. Theorem 2 shows that ATD* retains the efficiency properties of AD*.

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

Experimental Results

We evaluated ATD* comparing it to ARA* (Likhachev, Gordon, and Thrun 2004), AD* (Likhachev et al. 2008) and TD* Lite (Aine and Likhachev 2013b) for 2D and 3D path planning domains. 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 5000×5000 16-connected grids with 10% of the cells blocked. We used Euclidean distances as the heuristics. We performed two types of experiments, for the first experiment (known terrain) the map was given as input to the robot. We randomly changed the traversability of 1% of cells from blocked to unblocked and an equal number of cells from unblocked to blocked after 10 moves made by the robot, forcing it to replan. This procedure was iterated until the robot reached the goal. For the second experiment, the robot started with an empty map (all cells are traversable) and dynamically updated the traversability of

the cells sensing a 100×100 grid around its current position. If the traversability information changed, it replanned.

In Table 1 we include the speedup results for AD*, TD* Lite and ATD* over ARA* (i.e., the total planning time taken by ARA* divided by the total planning time taken by a given given), when searching for an ϵ-bounded solution. As ATD* uses two suboptimality bounds, we distributed the original bound so that $\epsilon = \epsilon_1 * \epsilon_2$. We set $\epsilon_2 = min(1.10, \sqrt{\epsilon})$ and $\epsilon_1 = \epsilon/\epsilon_2$.

Suboptimality	Known Environment			Unknown Environment		
Bound	AD*	TD* Lite	ATD*	AD*	TD* Lite	ATD*
5.0	0.81	0.46	0.92	3.13	0.28	3.41
2.0	0.86	0.52	0.91	1.58	0.45	1.55
1.5	1.21	0.82	1.12	1.45	0.78	2.32
1.1	1.47	1.15	1.51	5.54	4.66	5.34
1.05	1.66	3.58	3.20	10.63	13.26	13.07
1.01	2.51	10.28	9.03	17.08	24.62	23.14

Table 1: Average speedup vs ARA*, for AD*, TD* Lite, ATD* with different ϵ choices.

The results show that for high ϵ values both AD* and ARA* performs better than TD* Lite, as TD* Lite does not use an inflated heuristic. On the other hand, for low ϵ values, TD* Lite performs much better than AD*/ARA*, as it can reuse the previously generated paths more effectively. ATD* shows better consistency compared to both AD* and TD* Lite. When searching with high ϵ values, its performance is close to AD*, while it can retain the efficacy of TD* Lite when searching for close-to-optimal solutions. For unknown terrains, incremental searches generally perform better (than ARA*), as the cost changes happen close to the agent, and thus large parts of the search tree can be reused.

Environment	ARA*		AD*		ATD*	
	ϵ	Cost	ϵ	Cost	ϵ	Cost
Known	1.178	1.051	1.082	1.008	1.021	1.003
Unknown	1.058	1.008	1.025	1.006	1.016	1.006

Table 2: Comparison between ARA*, AD* and ATD* in anytime mode. Every planner was given 0.5 seconds to plan per episode.

We also ran the planners in an anytime mode where each planner was given 0.5 seconds per planning episode. All planners started with $\epsilon = 5.0$, which was reduced by 0.2 after each iteration. When the environment changed, if the last satisfied $\epsilon \leq 2.0$, we set $\epsilon = 2.0$ and replanned, otherwise the last ϵ value was retained. For ATD*, ϵ_2 was set to 1.1 at the start of each replanning episode and then iteratively decreased in the following way: if $1.2 < \epsilon \leq 2.0$, we set $\epsilon_2 = 1.05$; if $1.0 < \epsilon \leq 1.2$, we set $\epsilon_2 = 1.01$; otherwise $\epsilon_2 = 1.0$. In Table 2, we present the average ϵ-bounds satisfied and average path costs (over optimal path costs obtained using A*) for both type of environments. The results show the potential of ATD* in producing better bounds as well as better quality paths, when run in anytime mode, as it can effectively use both inflation and truncation.

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 θ (heading). The search objective is to plan smooth paths for non-holonomic robots, i.e., the paths must satisfy the minimum turning radius constraints. The actions used to get successors for states are a set of motion primitives (short kinematically feasible motion sequences) (Likhachev and Ferguson 2009). Heuristics were computed by running a 16-connected 2D Dijkstra search. For 3D planning, we performed similar experiments as described earlier (for 2D), but with maps of size 1000×1000. For the unknown environments, the sensor range was set to 2-times the length of the largest motion primitive.

Suboptimality	Known Environment			Unknown Environment		
Bound	AD*	TD* Lite	ATD*	AD*	TD* Lite	ATD*
5.0	0.53	0.09	2.15	1.18	0.14	2.33
2.0	0.87	0.18	3.73	2.48	0.57	9.81
1.5	1.09	1.30	8.02	1.84	2.11	9.20
1.1	1.35	2.50	3.75	2.39	2.40	11.34
1.05	2.71	6.66	7.62	4.37	5.95	7.19
1.01	1.41	4.48	3.84	3.83	7.37	5.76

Table 3: Average speedup vs ARA*, for AD*, TD* Lite, ATD* with different ϵ choices.

In Table 3 we include the results comparing ATD* with ARA*, AD* and TD* Lite when searching for a fixed ϵ-bound, and in Table 4 we present the results for the anytime runs where each planner was given 2 seconds to plan per episode. Overall, the results show the same trend as found for 2D. However, for this domain, ATD* shows even more improvement over AD*/ARA*/TD* Lite in most of the cases, as the environments are more complex and thus provide greater opportunities to combine truncation with heuristic inflation.

Environment	ARA*		AD*		ATD*	
	ϵ	Cost	ϵ	Cost	ϵ	Cost
Known	1.631	1.147	1.492	1.066	1.121	1.048
Unknown	1.558	1.091	1.269	1.037	1.056	1.020

Table 4: Comparison between ARA*, AD* and ATD* in anytime mode. Every planner was given 2.0 seconds to plan per episode.

Overall, the results show that incremental search is useful when planning a path requires substantial effort (if searching for close-to-optimal solutions or if the search space has large local minima), otherwise (when planning is relatively easy), the overhead of replanning can become prohibitive, making ARA* a better choice. Among incremental planners, AD* is better in quickly finding suboptimal paths whereas TD* Lite is better in reusing complete/partial paths from earlier plans. ATD* can simultaneously benefit from both these strategies and thus can outperform both AD* and TD* Lite.

Conclusions

We have presented Anytime Truncated D*, an anytime incremental search algorithm that combines heuristic inflation (for planning) with truncation (for replanning). Experimental results on 2D and 3D path planning domains demonstrate that ATD* provides more flexibility and efficacy over AD* (current state-of-the-art), and thus can be a valuable tool when planning for complex dynamic systems.

References

Aine, S., and Likhachev, M. 2013a. Anytime Truncated D* : The Proofs. Technical Report TR-13-08, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, USA.

Aine, S., and Likhachev, M. 2013b. Truncated Incremental Search : Faster Replanning by Exploiting Suboptimality. In *To appear in AAAI*. AAAI Press.

Aine, S.; Chakrabarti, P. P.; and Kumar, R. 2007. AWA* - A window constrained anytime heuristic search algorithm. In Veloso (2007), 2250–2255.

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

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.

Likhachev, M.; Gordon, G. J.; and Thrun, S. 2004. ARA*: Anytime A* with provable bounds on sub-optimality. In *Advances in Neural Information Processing Systems 16*. Cambridge, MA: MIT Press.

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

Richter, S.; Thayer, J. T.; and Ruml, W. 2010. The joy of forgetting: Faster anytime search via restarting. In Brafman, R. I.; Geffner, H.; Hoffmann, J.; and Kautz, H. A., eds., *ICAPS*, 137–144. AAAI.

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 (2007), 2391–2397.

Sun, X.; Yeoh, W.; Uras, T.; and Koenig, S. 2012. Incremental ARA*: An Incremental Anytime Search Algorithm for Moving-Target Search. In McCluskey, L.; Williams, B.; Silva, J. R.; and Bonet, B., eds., *ICAPS*. AAAI.

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.

Sun, X.; Yeoh, W.; and Koenig, S. 2010. Generalized Fringe-Retrieving A*: faster moving target search on state lattices. In van der Hoek, W.; Kaminka, G. A.; Lespérance, Y.; Luck, M.; and Sen, S., eds., *AAMAS*, 1081–1088. IFAAMAS.

Thayer, J. T.; Benton, J.; and Helmert, M. 2012. Better parameter-free anytime search by minimizing time between solutions. In Borrajo, D.; Felner, A.; Korf, R. E.; Likhachev, M.; López, C. L.; Ruml, W.; and Sturtevant, N. R., eds., *SOCS*. AAAI Press.

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

Veloso, M. M., ed. 2007. *Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI 2007, Hyderabad, India, January 6-12, 2007*. Morgan Kaufmann.

Zhou, R., and Hansen, E. A. 2002. Multiple sequence alignment using anytime a*. In *Proceedings of 18th National Conference on Artificial Intelligence AAAI'2002*, 975–976.

Zhou, R., and Hansen, E. A. 2005. Beam-stack search: Integrating backtracking with beam search. In *Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS-05)*, 90–98.