Anytime Tree-Restoring Weighted A* Graph Search

Kalin Gochev
University of Pennsylvania

Alla Safonova
University of Pennsylvania

Maxim Likhachev
Carnegie Mellon University

Abstract

Incremental graph search methods reuse information from previous searches in order to minimize redundant computation and to find solutions to series of similar search queries much faster than it is possible by solving each query from scratch. In this work, we present a simple, but very effective, technique for performing incremental weighted A^* graph search in an anytime fashion. On the theoretical side, we show that our anytime incremental algorithm preserves the strong theoretical guarantees provided by the weighted A^* algorithm, such as completeness and bounds on solution cost sub-optimality. We also show that our algorithm can handle a variety of changes to the underlying graph, such as both increasing and decreasing edge costs, and changes in the heuristic. On the experimental side, we demonstrate the effectiveness of our algorithm in the context of (x,y,z,yaw) navigation planning for an unmanned aerial vehicle and compare our algorithm to popular incremental and anytime graph search algorithms.

Keywords: Path Planning, Heuristic Search, Incremental Graph Search, Anytime Graph Search

Introduction

Many search algorithms exist for solving path planning problems in a graph-theoretical context. The main goal of these algorithms is to perform the search as fast as possible. A variety of techniques have been developed to speed up graph search, such as using heuristics to focus the search towards the goal, and trading off between search time and the cost of the resulting path. In this work, we discuss a different way of speeding up searches—incremental search.

Incremental search is a technique for continual planning that reuses information from previous searches to find solutions to a series of similar search problems potentially faster than it is possible by solving each search problem from scratch. In many situations, a system has to continuously adapt its plan to changes in its environment or in its knowledge of the environment. In such cases, the original plan might no longer be valid, and thus, the system needs to replan for the new situation. In these situations, solving the new search problem independently of previous search efforts (planning from scratch) can be very inefficient. This is especially true for situations when the changes of the search problem are small or very localized. For example, a robot might have to replan when it detects a previously unknown obstacle, which generally affects the graph structure and edge costs in a very localized fashion. Incremental graph search methods aim to re-use information from previous searches in order to minimize redundant computation.

In this work, we present a simple, but very effective, technique for performing incremental weighted A^* graph search in an anytime fashion. The algorithm employs a heuristic to focus the search and allows for trading off bounded path cost sub-optimality for faster search, just like weighted A^*. In addition, the algorithm re-uses information from previous search queries in order to improve planning times. Moreover, the algorithm can be used for anytime search, similarly to the Anytime Repairing A* (ARA^*) algorithm (Likhachev, Gordon, and Thrun 2003) starting the search with a large heuristic inflation factor ϵ to produce an initial solution faster, and continuously decreasing ϵ to 1 as time permits to find paths of lower sub-optimality bound. On the theoretical side, we show that our anytime incremental algorithm preserves the strong theoretical guarantees provided by the weighted A^* and ARA^* algorithms, such as completeness and bounds on solution cost sub-optimality. This work is an extension to (Gochev, Safonova, and Likhachev 2013), which allows the algorithm to handle a variety of changes to the underlying graph, such as both increasing and decreasing edge costs, and changes in the heuristic. On the experimental side, we demonstrate the effectiveness of our algorithm in the context of (x,y,z,yaw) navigation planning for an unmanned aerial vehicle and compare our algorithm to popular incremental and anytime graph search algorithms.

Related Work

Researchers have developed various methods for performing incremental heuristic searches, based on the observation that information computed during previous search queries can be used to perform the current search faster. Generally, incremental heuristic search algorithms fall into three categories.

The first class of algorithms, such as Lifelong Plan-

This work was supported by NSF Grant IIS-1018486 and DARPA Computer Science Study Group (CSSG) grant D11AP00275.

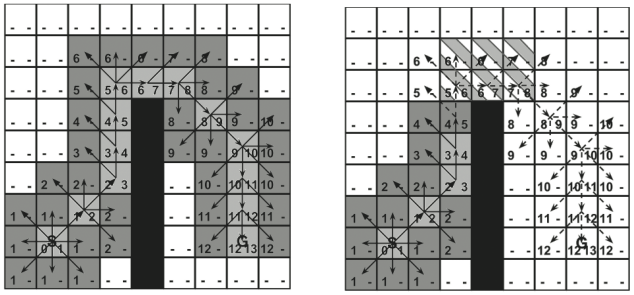
ning A^* (Koenig, Likhachev, and Furcy 2004), D^* (Stentz 1995), D^*-Lite (Koenig and Likhachev 2002), Anytime D^* (Likhachev et al. 2005), and Anytime Truncated D^* (Aine and Likhachev 2013), aim to identify and repair inconsistencies in a previously-generated search tree. These approaches are very general and don't make limiting assumptions about the structure or behavior of the underlying graph. They also demonstrate excellent performance by repairing search tree inconsistencies that are relevant to the current search task. The main drawback of these algorithms is the book-keeping overhead required, which sometimes may significantly offset the benefits of avoiding redundant computation.

The second class of algorithms, such as Fringe-Saving A^* (Sun, Yeoh, and Koenig 2009) and Differential A^* (Trovato and Dorst 2002), also try to re-use a previously-generated search tree, but rather than attempting to repair it, these approaches aim to identify the largest portion of the search tree that is unaffected by the changes (and thus is still valid), and resume searching from there. These approaches tend to be less general and to make limiting assumptions. The Fringe-Saving A^*, for example, is very similar in nature to our approach, but it only works on 2D grids with unit cost transitions between neighboring cells. It uses geometric techniques to reconstruct the search frontier based on the 2D grid structure of the graph. The assumptions made by these algorithms allow them to perform very well in scenarios that meet these assumptions. The algorithm presented in this work falls into this class of incremental heuristic search algorithms. Our approach is an extension to (Gochev, Safonova, and Likhachev 2013), which only handles increasing edge costs. The main advantage of our approach is that it is general and is able to handle arbitrary graphs and a variety of changes to the graph—increasing and decreasing edge costs, changes in the heuristic, and changes in the search goal state. We also extend the algorithm to be able to perform anytime search.

The third class of incremental heuristic search algorithms, such as Generalized Adaptive A^* (Sun, Koenig, and Yeoh 2008), aim to compute more accurate heuristic values by using information from previous searches. As the heuristic becomes more informative, search tasks are performed faster. The main challenge for such algorithms is maintaining the admissibility or consistency of the heuristic when edge costs are allowed to decrease. Path- and Tree-Adaptive A^* (Hernández et al. 2011) algorithms, for example, rely on the fact that optimal search is performed on a graph and edge costs are only allowed to increase. Our algorithm can handle changes in the heuristic, and therefore, it can theoretically be combined with such approaches that compute a better-informed heuristic, as long as the heuristic remains admissible.

Problem Definition

We are assuming that the planning problem is represented by a finite state-space S and a set of transitions $T = \{(X_i, X_j) | X_i, X_j \in S\}$. Each transition is associated with a positive cost $c(X_i, X_j)$. The state-space S and the transition set T define an edge-weighted graph $G = (S, T)$ with a vertex set S and edge set T. We will use the notation



(a) Tree-restoring A^* search showing the creation time (bottom left) and expansion time (bottom right) of each state. A dash indicates ∞.

(b) The first modified state is generated at step 5. Restoring the weighted A^* search ***state*** at step 4 produces a valid A^* search ***state***.

Figure 1: Simple 8-connected grid tree-restoring weighted A^* example (assuming a perfect heuristic for simplicity). Light gray: $CLOSED$ list (expanded states), dark gray: $OPEN$ list, striped: modified states, black: obstacles/invalid states, solid arrows: valid back-pointer tree, dashed arrows: invalid back-pointer tree.

$\pi_G(X_i, X_j)$ to denote a path from state X_i to state X_j in G, and its cost will be denoted by $c(\pi_G(X_i, X_j))$. We will use $\pi^*_G(X_i, X_j)$ to denote a least-cost path and $\pi^\epsilon_G(X_i, X_j)$ for $\epsilon \geq 1$ to denote a path of bounded cost sub-optimality: $c(\pi^\epsilon_G(X_i, X_j)) \leq \epsilon \cdot c(\pi^*_G(X_i, X_j))$. The goal of the planner is to find a least-cost path in G from a given start state X_S to a single goal state X_G. Alternatively, given a desired sub-optimality bound $\epsilon \geq 1$, the goal of the planner is to find a path $\pi^\epsilon_G(X_S, X_G)$. The costs of transitions (edges) in G are allowed to change between search queries. We call a state X modified if the cost of any of its outgoing transitions $(X, X') \in T$, change between search queries. The way to efficiently identify modified states is domain dependent, but in general one has to iterate through all the edges with changed costs and grab the source states.

Tree-Restoring Weighted A^* Search

Algorithm

The ***state*** of a weighted A^* search can be defined by the $OPEN$ list, the $CLOSED$ list, the g-values of all states, and the back-pointer tree. Note the distinction between a ***state*** of a search and a state in the graph being searched; we will use "***state***" when referring to a state of a search. The idea of our approach to incremental weighted A^* planning is to keep track of the ***state*** of the search, so that when the graph structure is modified, we can restore a valid previous search ***state*** and resume searching from there.

We call a ***state*** of a weighted A^* search valid with respect to a set of modified states, if the $OPEN$ and $CLOSED$ lists, and the back-pointer tree do not contain any of the modified states and the g-values of all states are correct with respect to the back-pointer tree.

At any one time during a weighted A^* search, each state

falls in exactly one of the following categories:

- $unseen$ - the state has not yet been encountered during the search; its g-value is infinite; the state is not in the back-pointer tree.
- $inOPEN$ - the state is currently in the $OPEN$ list; the state has been encountered (generated), but has not yet been expanded; its g-value is finite (assuming that when states with infinite g-values are encountered, they are not put in the $OPEN$ list); the state is in the back-pointer tree.
- $inCLOSED$ - the state is currently in the $CLOSED$ list; the state has been generated and expanded; its g-value is finite; the state is in the back-pointer tree.

We assume that the weighted A^* search expands each state at most once, which preserves the sub-optimality guarantees of the algorithm as proven in (Likhachev, Gordon, and Thrun 2003). The Tree-Restoring Weighted A^* algorithm (TRA^*) keeps a discrete time variable $step$ that is initialized at 1 and incremented by 1 after every state expansion. Thus, if we record the step $C(X)$ in which a state X is generated (first placed in the $OPEN$ list, $C(X) = \infty$ if state has not yet been generated) and the step $E(X)$ in which a state is expanded (placed in the $CLOSED$ list, $E(X) = \infty$ if the state has not yet been expanded), we can reconstruct the $OPEN$ and $CLOSED$ lists at the end of any step s (Fig. 1).

$$CLOSED_s = \{X | E(X) \leq s\}$$

$$OPEN_s = \{X | C(X) \leq s \text{ and } E(X) > s\}$$

Note that $C(X) < E(X)$ (i.e. a state's creation time is before the state's expansion time), and if $E(X) = E(X')$ then $X \equiv X'$ (i.e. no two states could have been expanded during the same step).

In order to be able to reconstruct the back-pointer tree and g-values for all states at the end of a previous step s, each state must store a history of its parents and g-values. Every time a better g-value g and parent X_p are found for a state X (when X_p is being expanded), a pair (X_p, g) is stored for the state X. Note that the pair stores the g-value of the state X itself, not the g-value of its parent X_p. Thus, we can compute the parent $P_s(X)$ and g-value $g_s(X)$ of a state X at the end of a previous step s by going through X's list L_X of stored parent/g-value pairs.

$$(P_s(X), g_s(X)) =$$

$$(X_p, g)_{\in L_X} | \forall (X', g')_{\in L_X} : E(X') \leq E(X_p) \leq s$$

In other words, the valid parent/g-value pair of X at step s is the pair containing the parent that was expanded last (most recently), but before or during step s. Storing the history in a list or array and searching it backwards seems to be very effective in quickly identifying the most recent valid parent and g-value.

When a set of states M get modified between search episodes by changes in the costs of some of their transitions, we identify the earliest step c_{min} in which a modified state was created: $c_{min} = min(C(X) | X \in M)$. If we then restore the search ***state*** at the end of step $c_{min} - 1$, we will end up with a valid search ***state*** with respect to the modified states, and thus, we can resume searching from there, provided the heuristic has not changed or does not need to be recomputed.

An important contribution of this work is allowing the algorithm to handle decreasing edge costs, which in turn, require the heuristic to be recomputed so that it remains admissible. In such cases, we might have to restore the search ***state*** to an even earlier step than $c_{min} - 1$ in order to ensure that correct expansion order is maintained with respect to the new heuristic values. We maintain correct expansion order by identifying all possible states that might have been expanded out-of-order relative to the current search ***state*** and the new heuristic values. An expanded state X might have been expanded out-of-order relative to the current best candidate for expansion X' from $OPEN$, if X's f-value at the time of its expansion was lower than the current f-value of X', and also, at the step when X was selected for expansion X' had been created and was in $OPEN$ (i.e. $C(X') < E(X)$). In other words, at time $E(X) - 1$ both X and X' were in $OPEN$ and X' had potentially better f-value than X, and therefore X might have been expanded incorrectly before X'. If we don't find any such states, then the current search ***state*** is valid with respect to the new heuristic and does not violate the proper expansion order. On the other hand, if we find a set of states I, that were potentially expanded out-of-order, we identify the state $X_f = \arg\min_{X \in I}(E(X))$ with the earliest expansion time and restore the search ***state*** at step $E(X_f) - 1$, right before the potentially incorrectly expanded state X_f was selected for expansion. We repeat this process of restoring previous search ***states*** until the current search ***state*** does not have any states that might have been expanded out-of-order.

We note that the TRA^* algorithm can be extended to allow for re-expansion of states by keeping multiple records of C and E values for each state for every time a state is placed on $OPEN$ and every time a state is expanded, respectively. However, such an extension will additionally increase the memory overhead of the algorithm. If re-expansions are allowed, however, maintaining correct expansion order is no longer necessary, as re-expansions of states will correctly propagate any inconsistencies in the search tree within the current search iteration.

Algorithms 1 and 2 give the pseudo code for all the important functions in the TRA^* algorithm.

Theoretical Properties

Theorem 1 *All states X with $C(X) > c$ will become $unseen$ after the function $restoreSearch(c)$ is called.*

Proof Follows trivially from definition. □

Theorem 2 *The contents of the $OPEN$ and $CLOSED$ lists after the function $restoreSearch(c)$ is called are identical to what they were at the end of step c of the algorithm.*

Proof (sketch) Let $OPEN_c$ and $CLOSED_c$ be the $OPEN$ and $CLOSED$ lists at the end of step c of the algorithm. Let $OPEN'$ and $CLOSED'$ be the $OPEN$ and $CLOSED$ lists

after the function $restoreSearch(c)$ is called. In can be easily shown that $X \in OPEN_c$ iff $X \in OPEN'$ and $X \in CLOSED_c$ iff $X \in CLOSED'$. Thus, $OPEN_c \equiv OPEN'$ and $CLOSED_c \equiv CLOSED'$. □

Theorem 3 *All states X with $C(X) \leq c$ will have correct parent pointers and corresponding g-values after $restoreSearch(c)$ is called.*

Proof (sketch) We construct a proof by contradiction. Suppose a state X has an incorrect parent pointer, i.e there exists a state $P' \in CLOSED$ such that $g(P') + cost(P', X) < g(P) + cost(P, X)$ (a better parent P' for X exists in the $CLOSED$ list). We argue that P' must have been expanded before P, and since P' provides better g-value than P, then P cannot have been recorded as a parent for X—contradiction. □

Theorem 4 *Let M be the set of all modified states after a successful incremental A^* search episode. Let $c_{min} = min(C(X)|X \in M)$. $restoreSearch(c)$ for any $c < c_{min}$ results in a search **state** that is valid with respect to the modified states M.*

Proof The result follows directly from the above theorems. □

If edge costs cannot decrease, the heuristic remains admissible between search episodes and does not need to be re-computed. However, the heuristic does need to be recomputed when edge costs decrease, in order to ensure that the current search is performed with admissible heuristic values. Changes in the heuristic values, however, affect the ordering of states in the $OPEN$ list and the order of state expansions during the search. As we only allow states to be expanded once, it is necessary to maintain correct expansion order. Thus, although by Theorem 4 $restoreSearch(c_{min} - 1)$ produces a search ***state*** that is valid with respect to the modified states, that search ***state*** is not necessarily valid with respect to the new heuristic values, as the order of expansions might be no longer correct. $heuristicChanged()$ is the function that maintains the correct expansion order when the heuristic changes. As described above, the idea of this function is to keep restoring the search to earlier search ***state*** until there are no states that could have been expanded in incorrect order. In the worst case, the change in the heuristic is such that expansion order changes from the very beginning, in which case $heuristicChanged()$ will restore the search ***state*** to the end of step 0–right after the start state was expanded, which would be equivalent to starting the search from scratch.

Theorem 5 *The function $heuristicChanged()$ terminates and at the time of its termination the search is restored to a search **state** that is valid with respect to the new heuristic values. That is, no state has been expanded out-of-order with respect to the new f-values.*

Proof Let X_0 be the state with lowest f-value in $OPEN$ in the current search ***state***. X_0 was first put in $OPEN$ at step $C(X_0)$.

Consider the set I computed in $heuristicChanged()$. As in (Likhachev, Gordon, and Thrun 2003), $v(X)$ stores the value of $g(X)$ at the time X was expanded. Therefore $v(X) + \epsilon \cdot h(X)$ represents the f-value of X at the time of its expansion $E(X)$, but also accounting for the new heuristic values. $I = \{X_i \in CLOSED | v(X_i) + \epsilon \cdot h(X_i) > f(X_0) \wedge C(X_0) < E(X_i)\}$. In other words, I contains all expanded states that had higher f-values at the time of their expansion than the current candidate for expansion X_0 and that were expanded while X_0 was in $OPEN$. As such, I contains all possible states that might have been expanded incorrectly before X_0 according to the new f-values. Note that it is possible that the current $f(X_0)$ is lower than the value of $f(X_0)$ at step $E(X_i)$, as $g(X_0)$ might have decreased as the search progressed after step $E(X_i)$. Therefore, it is possible that $f(X_i) \leq f(X_0)$ was true at step $E(X_i)$ and that $f(X_i)$ was correctly selected for expansion before X_0. Thus, states in I are not necessarily expanded incorrectly, but they are the only possible states that might have been expanded incorrectly. Let $s' = \min(E(X')|X' \in I) - 1$ as computed in $heuristicChanged()$. Restoring the search ***state*** to step s' ensures that no states have been expanded incorrectly before X_0. At the end of the while loop $I = \emptyset$, thus no states in $CLOSED$ could have been expanded incorrectly with respect to the current expansion candidate X_0.

To prove that $heuristicChanged()$ terminates, we argue that the integer s' strictly decreases through the execution of the while loop. If s' becomes 0, then $CLOSED = \emptyset$ making $I = \emptyset$. □

By Theorem 5, TRA^* algorithm maintains the same expansion order (up to tie-breaking) as non-incremental weighted A^* and thus, both algorithms have the same theoretical guarantees for completeness, termination, and upper bounds on path cost sub-optimality, assuming that an admissible heuristic is used.

Theorem 6 *TRA^* expands each state at most once per search query and never expands more states than Weighted A* from scratch (up to tie-breaking).*

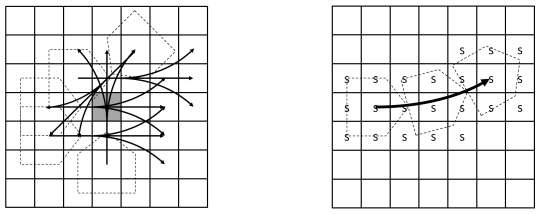
Proof It is easy to verify that each state can be expanded at most once per search query, as once a state has been expanded and put in $CLOSED$ it can never be placed in $OPEN$. The fact that TRA^* does not expand more states than performing Weighted A^* from scratch follows almost trivially from the fact that the two algorithms produce the same order of state expansions (up to tie-breaking), but TRA^* is able to resume searching from a step $s \geq 0$, thus not performing the first s expansions that Weighted A^* from scratch would have to perform. □

Anytime Tree-Restoring Weighted A*

In many situations, producing a lower-quality initial solution very quickly, and then improving the solution as time permits, is a desirable property of a planning algorithm.

By following the concept of the ARA^* search algorithm, we can extend the TRA^* algorithm to perform in an anytime fashion. ARA^* runs a series of searches with decreasing heuristic weighting factor ϵ until the allocated time runs out or an optimal solution is found for $\epsilon = 1$. It keeps track of an INCONSISTENT list of all states that have been expanded already during the current search iteration (in CLOSED), yet a better parent and lower g-value for them was found after their expansion. The states in INCONS. are moved to OPEN at the beginning of every search iteration, OPEN is re-ordered based on the new ϵ value, and the search proceeds.

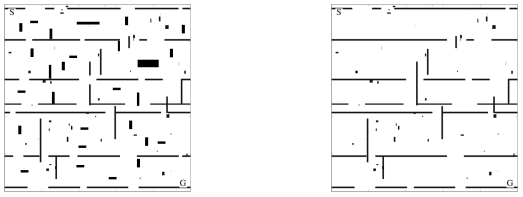
To make TRA^* an anytime algorithm similar to ARA^*, we need to be able to reconstruct the INCONS. list at a particular time step. Thus, we have to record the step at which a state X is inserted into INCONS., $I(X)$. Also, since ARA^* allows re-expansions of states between search iterations (the



(a) Computing all graph edges affected by a change in a map cell. The figure shows a sub-set of the edges (arrows) affected by a change in the shaded cell. Dashed polygons represent the robot's perimeter.

(b) $ATRA^*$ algorithm storing the expansion step s for which each cell is encountered first, done during the expansion and collision checking of each edge (arrow).

Figure 2: Computing affected graph edges from changed map cells.



(a) Example environment (top view)

(b) Initial partially-known map provided to the robot (top view)

Figure 3: Example environment and corresponding initial map. The start and goal locations are marked by S and G, respectively.

ones from INCONS. list), we also need to maintain separate creation $C_\epsilon(X)$, expansion $E_\epsilon(X)$, and inconsistent $I_\epsilon(X)$ records for each ϵ value. Thus, the memory overhead introduced by the algorithm for each state increases proportionally to the number of times it is expanded.

We can reconstruct INCONS. at a desired step s by noting that a state X is in INCONS. from the step $I_{\epsilon_1}(X)$ when the state was inserted into INCONS. for a particular ϵ_1, until it was inserted in OPEN at the beginning of the next planning iteration (for ϵ_2). Thus, $X \in$ INCONS. iff $I_{\epsilon_1}(X) \leq s < C_{\epsilon_2}(X)$.

Then, given a desired target restore step s, we can reconstruct the contents of OPEN, CLOSED, and INCONS. lists, the back-pointer tree, g-values, and the ϵ_s value of the search ***state*** at step s. For every state we drop the creation $C_\epsilon(X)$, expansion $E_\epsilon(X)$, and inconsistent $I_\epsilon(X)$ records for $\epsilon < \epsilon_s$, only maintaining the records up to the current heuristic inflation value ϵ_s.

The proposed Anytime Tree-Restoring Weighted A^* ($ATRA^*$) search algorithm preserves the theoretical properties of the ARA* algorithm, such as completeness with respect to the graph encoding the problem and bounds on solution cost sub-optimality.

Detecting Changes in the Graph

In the context of navigation planning, lattice-based graphs are often used to encode the search problem by discretizing the configuration space of the robot, and using pre-computed kinodynamically feasible transitions between states (Likhachev and Ferguson 2008). On the other hand, the map data and obstacle information is usually stored on a grid. Thus, most incremental search algorithms, such as D*, D*-Lite, and Anytime D*, need to be able to translate changes in the map grid to the actual graph edges that are affected by the changes. In other words, the algorithm needs to consider all edges in the graph that cause the robot's perimeter to pass through the changed cell (Fig. 2(a)). This procedure can be prohibitively expensive for graphs with high edge density and for large robot perimeters, often significantly diminishing or completely eliminating the benefit of using incremental graph search.

Our approach, however, does not rely on knowing all affected edges, but rather just the expansion step at which the first affected edge was encountered during the search. Thus, for each cell on the map grid, we can record the earliest expansion step for which the search encountered an edge that passes through this cell (Fig. 2(b)). This introduces a small memory overhead to the size of the map grid data stored (additional integer per cell). However, the performance overhead is negligible, as the collision-checking procedure already enumerates all map cells that an edge passes through to make sure they are obstacle-free. With this extension, when a map cell changes, we can very quickly look up the earliest expansion step for which this cell affected an edge in the graph. Taking the minimum expansion step s across all changed cells in the map and restoring the search ***state*** to step $s-1$ produces a valid search tree with respect to the modified map cells, and thus, their respective modified graph edges.

As shown in our experiments, this approach significantly reduces the time needed for TRA^* and $ATRA^*$ to compute the changes to the graph, and subsequently, the overhead of performing incremental search.

Experimental Setup

To validate the $ATRA^*$ algorithm we implemented it for 4-DoF (x,y,z,yaw) path planning for an unmanned aerial vehicle. The graph representing the problem was constructed as a lattice-based graph, similar to the approach taken in (Likhachev and Ferguson 2008), except we used constant resolution for all lattices. In lattice-based planning, each state consists of a vertex encoding a state vector and edges corresponding to feasible transitions to other states. The set of edges incident to a state are computed based on a set of pre-computed motion primitives, which are executable by the robot. The state-space was obtained by uniformly discretizing the environment into $5cm \times 5cm \times 5cm$ cells and the heading values were uniformly discretized into 16 on the interval $[0, 2\pi)$. The robot was tasked to navigate to a fixed goal location. Search was performed backwards from the goal state and the start state changed as the vehicle navigated through the environment. Whenever a path to the goal

Algorithm	Avg. Sub-optimality Bound Achieved	Compute Changes Avg. Time (s)	Repair/Restore Avg. Time (s)	% Iters finished within 1s	# Expansions per Re-plan avg	# Expansions per Re-plan std dev	Avg. Path Cost Ratio
$ATRA^*$	2.2720	0.0000	0.1615	95.95%	52029	29826	1.0 (baseline)
Anytime D^*	2.2324	0.6214	0.3240	91.01%	50052	47874	0.98
ARA^*	2.4211	0 (n/a)	0 (n/a)	93.70%	77377	16786	1.21
Anytime Truncated D^*	2.1124	0.6271	0.3952	91.67%	48853	46904	0.95
Beam-Stack Search	(n/a)	0 (n/a)	0 (n/a)	98.07%	65149	21479	1.32
$ATRA^*$	1.8185	0.0000	0.1501	99.63%	47230	20194	1.0 (baseline)
Anytime D^*	1.8176	0.4874	0.4067	96.40%	53976	39737	1.03
ARA^*	2.1600	0 (n/a)	0 (n/a)	99.20%	82041	17681	1.17
Anytime Truncated D^*	1.7802	0.4753	0.4247	97.73%	50974	38225	0.98
Beam-Stack Search	(n/a)	0 (n/a)	0 (n/a)	99.54%	69203	22405	1.24

Table 1: Simulation results on a set of 50 unknown maps (top) and 50 partially-known maps (bottom) for 4-DoF (x,y,z,yaw) path planning for an unmanned aerial vehicle performing anytime planning with time limit of 1 second.

Algorithm	Sub-optimality Bound	Compute Changes Avg. Time (s)	Repair/Restore Avg. Time (s)	Re-planning Time (s) avg	Re-planning Time (s) std dev	# Expansions per Re-plan avg	# Expansions per Re-plan std dev
$ATRA^*$	5.0	0.0000	0.0327	0.2105	0.4643	11065	21499
Anytime D^*	5.0	0.4063	0.0194	0.5973	3.7712	12799	44616
ARA^*	5.0	0 (n/a)	0 (n/a)	0.2770	0.3163	22666	21602
Anytime Truncated D^*	5.0	0.4177	0.0231	0.5031	1.6433	11533	36551
$ATRA^*$	2.0	0.0000	0.0895	0.3583	0.6435	19994	30477
Anytime D^*	2.0	0.6111	0.2109	0.3397	0.8574	21580	54726
ARA^*	2.0	0 (n/a)	0 (n/a)	0.5004	0.4017	44352	26316
Anytime Truncated D^*	2.0	0.6093	0.2242	0.3088	0.6881	18306	28487
$ATRA^*$	1.25	0.0000	0.1593	1.6718	5.9480	70116	225425
Anytime D^*	1.25	1.5722	1.5150	4.0458	11.565	213777	592017
ARA^*	1.25	0 (n/a)	0 (n/a)	5.6696	16.038	384546	904201
Anytime Truncated D^*	1.25	1.5983	1.5311	2.3184	8.1722	107634	472733
$ATRA^*$	5.0	0.0000	0.0258	0.0322	0.1260	2338	8248
Anytime D^*	5.0	0.1986	0.0207	0.1326	0.6359	5427	25114
ARA^*	5.0	0 (n/a)	0 (n/a)	0.3118	0.2125	30705	19612
Anytime Truncated D^*	5.0	0.2043	0.0313	0.1196	0.5363	5214	22756
$ATRA^*$	2.0	0.0000	0.0698	0.2635	1.0427	14178	50706
Anytime D^*	2.0	0.3367	0.1338	0.2531	1.2339	16042	85201
ARA^*	2.0	0 (n/a)	0 (n/a)	0.7860	0.8860	70229	64742
Anytime Truncated D^*	2.0	0.3274	0.1459	0.2364	1.1491	14954	76638
$ATRA^*$	1.25	0.0000	0.4295	1.5719	10.448	66014	395437
Anytime D^*	1.25	0.6864	0.9521	1.9882	8.1290	120754	467172
ARA^*	1.25	0 (n/a)	0 (n/a)	4.1886	9.5945	271330	548481
Anytime Truncated D^*	1.25	0.6593	0.9361	1.7318	6.9560	97811	422109

Table 2: Simulation results on a set of 50 unknown maps (top) and 50 partially-known maps (bottom) for 4-DoF (x,y,z,yaw) path planning for an unmanned aerial vehicle performing fixed-ϵ planning until first solution for various sub-optimality bounds.

was computed, the robot advanced by one edge along the path to a new start state; sensed any previously unknown obstacles or gaps through obstacles in its vicinity and updated its environment map; then re-planned for a new path to the goal accounting for the changes in the environment. The appearing and disappearing of obstacles in the map caused, respectively, increasing and decreasing of edge costs in the graph. This, in turn, required a set of modified states to be computed and the heuristic to be re-computed. Moreover, it was necessary re-compute the heuristic at the beginning of every re-planning iteration, as the robot moved through the environment and the start state changed. The heuristic was computed using 3D BFS search from the (x,y,z) position of the start state on an 26-connected 3D grid accounting for obstacles. The heuristic was not perfect as did not account for the orientation of the robot or its perimeter shape. Thus, some scenarios exhibited pronounced heuristic local minima. We ran the planner on 50 maps of size $25m \times 25m \times 2m$ ($500 \times 500 \times 40$ cells) (example shown in Fig. 3). For each of the environments, the planner was run on both an unknown initial map, and a partially-known initial map. An example of a partially-known initial map is shown in Fig. 3(b). The maps were generated semi-randomly to resemble floor plans. The partially-known initial maps were generated by randomly adding and removing obstacles from the true map. The start and goal states for each environment were in diagonally opposite corners of the map. We used a set of pre-computed transitions obeying minimal turning radius constraints. The vehicle was also allowed to turn in-place, but the cost of such transitions was penalized by a factor of 5. The non-holonomic transitions and the high penalty factor for turning in-place made the path planning problem very challenging. For sensing obstacles, we simulated a forward-facing tilting laser range finder with 180° horizontal and 90° vertical field of view, and a maximum sensing range of $2.0m$.

Algorithm 1 Tree-Restoring Weighted A^*

$CLOSED$: Set
$OPEN$: MinHeap
$CREATED$: Array
$step$: Integer
function INITIALIZESEARCH(X_S X_G)
  $CLOSED \leftarrow \emptyset$
  $OPEN \leftarrow \{X_S\}$
  $g(X_S) \leftarrow 0$
  $f(X_S) \leftarrow g(X_S) + \epsilon \cdot h(X_S)$
  $step \leftarrow 1$
  $C(X_S) \leftarrow 0$
  insert($CREATED$, X_S)
  $E(X_S) \leftarrow \infty$
end function
function RESUMESEARCH( )
  if needed to recompute heuristic **then**
    recompute admissible heuristic
    heuristicChanged()
  end if
  while $OPEN \neq \emptyset$ **do**
    $X \leftarrow$ extractMin($OPEN$)
    if $f(X_G) > f(X)$ **then**
      return reconstructPath()
    end if
    Expand(X)
  end while
  return no path exists
end function
function HEURISTICCHANGED
  update f-values for created states and re-order $OPEN$
  while not done **do**
    Let X_0 be the state with lowest f-value in $OPEN$
    $I \leftarrow \{X \in CLOSED | v(X) + \epsilon \cdot h(X) > f(X_0) \wedge C(X_0) < E(X)\}$
    if $I = \emptyset$ **then**
      done
    else
      $s' \leftarrow \min(E(X') | X' \in I) - 1$
      restoreSearch(s')
    end if
  end while
end function
function UPDATEPARENTS(X, s)
  $latestG \leftarrow 0$
  $latestParent \leftarrow \varnothing$
  $latestParentStep \leftarrow 0$
  for all (X_p, g_p) in stored parent/g-value pairs of X **do**
    if $E(X_p) \leq s$ **then** ▷ X_p is a valid parent for step s
      if $E(X_p) > latestParentStep$ **then** ▷ Found more recent parent
        $latestParentStep \leftarrow E(X_p)$
        $latestParent \leftarrow X_p$
        $latestG \leftarrow g_p$
      end if
    else ▷ X_p is not a valid parent for step s
      Remove (X_p, g_p) from stored parent/g-value pairs
    end if
  end for
  return $(latestParent, latestG)$
end function

Algorithm 2 Tree-Restoring Weighted A^*

function RESTORESEARCH(s)
    ▷ restores the search state to just after the expansion at step s
  $OPEN \leftarrow \emptyset$
  $CLOSED \leftarrow \emptyset$
  $CREATED' \leftarrow \emptyset$
  if $s \leq 0$ **then**
    initializeSearch(X_S, X_G)
    return
  end if
  for all $X \in CREATED$ **do**
    if $E(X) \leq s$ **then** ▷ state created and expanded
      $(X_p, g) \leftarrow$ updateParents(X, s)
      $g(X) \leftarrow g$
      $parent(X) \leftarrow X_p$
      insert($CLOSED$, X)
      insert($CREATED'$, X)
    else if $C(X) \leq s$ **then** ▷ state created, not expanded
      $(X_p, g) \leftarrow$ updateParents(X, s)
      $g(X) \leftarrow g$
      $v(X) \leftarrow \infty$
      $parent(X) \leftarrow X_p$
      $f(X) \leftarrow g + \epsilon \cdot h(X')$
      insertOpen(X, $f(X)$)
      $E(X) \leftarrow \infty$
      insert($CREATED'$, X)
    else ▷ state not created
      clearParents(X)
      $g(X) \leftarrow \infty$
      $v(X) \leftarrow \infty$
      $parent(X) \leftarrow \varnothing$
      $C(X) \leftarrow \infty$
      $E(X) \leftarrow \infty$
    end if
  end for
  $CREATED \leftarrow CREATED'$
  $step \leftarrow s + 1$
end function
function EXPAND(X)
  $v(X) \leftarrow g(X)$
  for all X' ∈ successors of X **do**
    if X' was not visited before **then**
      $g(X') = \infty$
    end if
    $g' \leftarrow g(X) + cost(X, X')$
    if $g' \leq g(X')$ **then**
      $g(X') \leftarrow g'$
      storeParent(X',(X, g'),$step$)
      $f(X') \leftarrow g' + \epsilon \cdot h(X')$
      if $X' \notin CLOSED$ **then**
        if $X' \notin OPEN$ **then**
          insertOPEN(X', $f(X')$)
          $C(X') \leftarrow step$ ▷ record state put in OPEN
          insert($CREATED$, X')
        else
          updateOPEN(X', $f(X')$)
        end if
      end if
    end if
  end for
  $E(X) \leftarrow step$ ▷ record state expanded
  insert($CLOSED$, X)
  $step \leftarrow step + 1$
end function

We ran our planner in anytime mode with an initial sub-optimality bound of $\epsilon = 5.0$ with 1 second allowed for planning. In cases when no plan was found within the time limit, the planner was allowed to continue planning for an additional 1 second for up to 10 times until a solution is found. We also ran our planner in fixed-ϵ mode, planning until the first solution satisfying the specified sub-optimality bound is found.

Results

We compared the $ATRA^*$ algorithm to other incremental and anytime graph search algorithms—Anytime D^* (Koenig and Likhachev 2002), ARA^* (Likhachev, Gordon, and Thrun 2003), Anytime Truncated D^* (Aine and Likhachev 2013), and Beam-Stack Search (Zhou and Hansen 2005). The non-incremental algorithms ARA^* and Beam-Stack Search performed planning from scratch at each iteration. In order to replicate the planning conditions across all planners for fair performance comparison, the vehicle followed a predefined path through the environment regardless of the paths produced by the planners. Thus, each of the planners performed the same number of re-planning iterations with identical map information. All planners used the same heuristic, recomputed for every re-planning iteration. The time reported as "Compute Changes" is the time each incremental algorithm required to translate changes in the map grid into relevant changes to the graph (computing modified edges for Anytime D^* and Anytime Truncated D^*, and computing the target restore step for $ATRA^*$). The time reported as "Repair/Restore" is the time each incremental algorithm took to update its search ***state*** with the new edge costs and heuristic values, so that a new search iteration can be started.

The results we observed for anytime planning for each of the planners for unknown and partially-known maps are summarized in Table 1. As seen from the results, $ATRA^*$ is able to detect graph changes and restore the search tree significantly faster than Anytime D^* and Anytime Truncated D^*, while achieving nearly identical sub-optimality bounds and path costs, on average, on both unknown and partially-known maps. Beam Stack Search was able to meet the 1-second deadline in the highest number of iterations at the expense of about 20-30% higher solution cost and no theoretical sub-optimality bound guarantees.

The results we observed for planning until the first solution satisfying a fixed sub-optimality bound for each of the planners for unknown and partially-known maps are summarized in Table 2. Beam Stack Search was not included in these experiments as it does not provide theoretical sub-optimality bounds on intermediate solutions. The results illustrate the benefit of using incremental graph search as the desired sub-optimality bound decreases. Overall, $ATRA^*$ performed significantly better than the rest of the planners by both reducing the overhead of performing incremental search and reducing the number of expansions (and thus re-planning time) required for each iteration.

We observed that $ATRA^*$ is able to outperform planning from scratch most significantly on the difficult planning scenarios (ones exhibiting heuristic local minima, or ones with low sub-optimality bound), as it is able to avoid re-expanding a large number of states between iterations in such cases. The most significant performance gain of $ATRA^*$ over the two D^*-based algorithms, apart from the reduced overhead in computing changes in the graph, were in scenarios when increasing edge costs cause a large number of expansions of under-consistent states (significantly more expensive than regular over-consistent expansions) in the D^*-based algorithms. On the other hand, our approach suffers most in situations where the search ***state*** needs to be restored to a very early step , in which cases the overhead of performing repeated tree restoring eliminates the benefits of avoiding relatively few re-expansions.

Conclusion

In this work we have presented a novel algorithm for performing anytime incremental weighted A^* search. The approach is general and can handle a variety of changes to the graph, such as increasing and decreasing edge costs, changes in the heuristic values, and changing search goals. We have shown that the algorithm preserves the theoretical guarantees of weighted A^* and ARA^* algorithms, on which it is based, such as completeness, termination, and bounds on path cost sub-optimality. Some of the advantages of the Anytime Tree-Restoring Weighted A^* algorithm over D^*-Lite-based algorithms are the relative simplicity of our approach and the reduced overhead of performing incremental search. The main drawback of our approach is the increased memory overhead required. Our experimental results suggest that our algorithm is able to outperform popular state-of-the-art incremental and anytime search algorithms such as Anytime D^*, Anytime Truncated D^*, and ARA^*, making it a viable alternative for performing incremental graph search with bounded solution cost sub-optimality.

References

Aine, S., and Likhachev, M. 2013. Anytime truncated D* : Anytime replanning with truncation. In Helmert, M., and Rger, G., eds., *SOCS*. AAAI Press.

Gochev, K.; Safonova, A.; and Likhachev, M. 2013. Incremental planning with adaptive dimensionality. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*.

Hernández, C.; Sun, X.; Koenig, S.; and Meseguer, P. 2011. Tree adaptive A*. In *The 10th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '11, 123–130. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems.

Koenig, S., and Likhachev, M. 2002. D*-lite. In *Eighteenth national conference on Artificial intelligence*, 476–483. Menlo Park, CA, USA: American Association for Artificial Intelligence.

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

Likhachev, M., and Ferguson, D. 2008. Planning long dynamically-feasible maneuvers for autonomous vehicles. In *Proceedings of Robotics: Science and Systems (RSS)*.

Likhachev, M.; Ferguson, D.; Gordon, G.; Stentz, A.; and Thrun, S. 2005. Anytime dynamic a*: An anytime, replanning algorithm. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*.

Likhachev, M.; Gordon, G.; and Thrun, S. 2003. ARA*: Anytime A* with provable bounds on sub-optimality. In *Advances in Neural Information Processing Systems (NIPS)*. Cambridge, MA: MIT Press.

Stentz, A. 1995. The focussed D* algorithm for real-time replanning. In *Proceedings of the 14th international joint conference on Artificial intelligence - Volume 2*, IJCAI'95, 1652–1659. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Sun, X.; Koenig, S.; and Yeoh, W. 2008. Generalized adaptive A*. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems - Volume 1*, AAMAS '08, 469–476. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems.

Sun, X.; Yeoh, W.; and Koenig, S. 2009. Dynamic fringe-saving A*. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '09, 891–898. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems.

Trovato, K. I., and Dorst, L. 2002. Differential A*. *IEEE Trans. on Knowl. and Data Eng.* 14(6):1218–1229.

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