Tree Adaptive A*[*]

Carlos Hernández
Departamento de Ingeniería Informática
Universidad Católica de la Ssma. Concepción
Caupolican 491, Concepción, Chile
chernan@ucsc.cl

Xiaoxun Sun Sven Koenig
Computer Science Department
University of Southern California
Los Angeles, CA 90089, USA
{xiaoxuns,skoenig}@usc.edu

Pedro Meseguer
Institut d'Investigació en Intel.ligéncia Artificial
IIIA-CSIC
Campus UAB
08193 Bellaterra, Spain
pedro@iiia.csic.es

ABSTRACT

Incremental heuristic search algorithms can solve sequences of similar search problems potentially faster than heuristic search algorithms that solve each search problem from scratch. So far, there existed incremental heuristic search algorithms (such as Adaptive A*) that make the h-values of the current A* search more informed, which can speed up future A* searches, and incremental heuristic search algorithms (such as D* Lite) that change the search tree of the current A* search to the search tree of the next A* search, which can be faster than constructing it from scratch. In this paper, we present Tree Adaptive A*, which applies to goal-directed navigation in unknown terrain and builds on Adaptive A* but combines both classes of incremental heuristic search algorithms in a novel way. We demonstrate experimentally that it can run faster than Adaptive A*, Path Adaptive A* and D* Lite, the top incremental heuristic search algorithms in the context of goal-directed navigation in unknown grids.

Categories and Subject Descriptors

I.2.8 [**Problem Solving, Control Methods, and Search**]: [Graph and tree search strategies]

General Terms

Algorithms, Experimentation

Keywords

Agent Reasoning::Planning (single and multi-agent), Robot Reasoning::Planning, Path Planning

[*]This material is based upon work supported by NSF (while Sven Koenig was serving at NSF). It is also based upon work supported by Fondecyt-Chile under contract/grant number 11080063, ARL/ARO under contract/grant number W911NF-08-1-0468, ONR in form of a MURI under contract/grant number N00014-09-1-1031, DOT under contract/grant number DTFH61-11-C-00010 and the Spanish Ministry of Science and Innovation under grant number TIN2009-13591-C02-02. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations, agencies or the U.S. government.

Cite as: Tree Adaptive A*, Carlos Hernández, Xiaoxun Sun, Sven Koenig and Pedro Meseguer, *Proc. of 10th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2011)*, Tumer, Yolum, Sonenberg and Stone (eds.), May, 2–6, 2011, Taipei, Taiwan, pp. XXX-XXX.

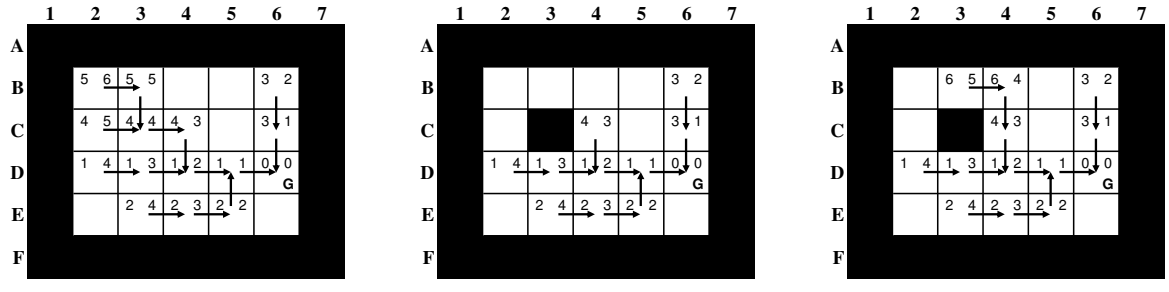
1. INTRODUCTION

Agents, such as robots and game characters, have to be able to navigate from their current location to a given destination [4]. However, they might not know a map of the terrain initially, and their sensors can typically sense the terrain only near their current location. They can use path planning with the freespace assumption to navigate from their current location to their destination, which is a popular approach in robotics [13]: The agents plan a minimum-cost path from their current location to their destination under the assumption that the terrain is traversable except for the obstacles that they have already sensed. As they move along the planned path, they sense additional obstacles and add them to their map. When they detect obstacles on the path, they replan a minimum-cost path from their current location to their destination and repeat the process until they reach their destination or can no longer find a path to their destination (in which case the destination is unreachable).

Path planning with the freespace assumption thus interleaves path planning with movement and requires repeated searches. These searches need to be fast since agents have to move smoothly and without delay. For example, the computer game company Bioware imposes a time limit of 1-3ms on each search [2]. However, even A* searches [6] can be time consuming if the terrain is large or many agents perform simultaneous searches. Incremental heuristic search algorithms use information from the current and previous searches to solve future similar search problems potentially faster than heuristic search algorithms that solve each search problem from scratch [12]. They have been used to speed up A* searches in the context of both symbolic planning [9] and path planning [12]. There are two classes of incremental heuristic search algorithms:

- Incremental heuristic search algorithms of the first class make the h-values of the current search more informed, which can speed up future searches by making them more focused. Examples include Adaptive A* [11], Generalized Adaptive A* [19] and Multi-target Adaptive A* [16].

- Incremental heuristic search algorithms of the second class change the search tree of the current search to the search tree of the next search, which can be faster than constructing it from scratch. Examples include D* [18] and D* Lite [10], which can speed up A* searches by more than one order of magnitude [10] and are typically faster than Adaptive A* and Generalized Adaptive A* in the context of goal-directed navigation in unknown terrain [11]. Versions of them have been used as part of path planners in



(a)

x	$H_{max}(x)$	$H_{min}(x)$	$Paths(x)$
1	4	0	{2,4}
2	4	1	∅
3	2	0	∅
4	5	2	{5}
5	6	4	∅

(b)

x	$H_{max}(x)$	$H_{min}(x)$	$Paths(x)$
1	4	0	{2,4}
2	4	1	∅
3	2	0	∅
4	3	2	∅
5	4	4	∅

(c)

x	$H_{max}(x)$	$H_{min}(x)$	$Paths(x)$
1	4	0	{2,4}
2	4	1	∅
3	2	0	∅
4	3	2	{6}
5	4	4	∅
6	5	3	∅

Figure 1: Illustration of Reusable Trees

a wide range of fielded robotics systems [14, 5, 15], including the winning DARPA Urban Challenge entry from Carnegie Mellon University.

We present Tree Adaptive A* (Tree-AA*) in this paper, an incremental heuristic search algorithm that applies to path-planning with the freespace assumption for goal-directed navigation in unknown terrain or, more generally, repeatedly following a minimum-cost path from the current location to a destination where the movement costs can increase (but not decrease). Tree-AA* generalizes Path Adaptive A* (Path-AA*) [7]. Path-AA* applies to path planning with the freespace assumption and generalizes Adaptive A* to reuse a suffix of the minimum-cost path of the current A* search (= reusable path) to allow the next A* search to terminate earlier. Tree-AA* also applies to path planning with the freespace assumption but generalizes Adaptive A* to reuse suffixes of the minimum-cost paths of the current and all previous A* searches (= reusable tree). Thus, Tree-AA* combines incremental heuristic search algorithms of the two above classes in a novel way. The reusable tree of Tree-AA* is similar to the search tree of incremental heuristic search algorithms (such as D* Lite) that change the search tree of the current A* search to the search tree of the next A* search since they perform backward A* searches to guarantee that the root of the search tree does not change. However, Tree-AA* changes the reusable tree via forward A* searches, which is a novel way of maintaining the search tree. We demonstrate experimentally that it can run faster than Path-AA*, Adaptive A* and D* Lite, the top incremental heuristic search algorithms in the context of goal-directed navigation in unknown grids.

2. NOTATION

We use the following notation: S is the finite set of states, which correspond to the locations. $s_{start} \in S$ is the current state of the agent, which corresponds to its current location. $s_{goal} \in S$ is the goal state, which corresponds to its destination. $Succ(s) \subseteq S$ is the set of successor states of state $s \in S$. $c(s, s') > 0$ is the cost of moving from state $s \in S$ to its successor state $s' \in Succ(s)$. The goal cost of a state is the cost of a minimum-cost path from the state to the goal state. The h-value $h(s)$ (= heuristic) of state $s \in S$ is a consistent approximation of the *goal cost* of the state, that is, one that satisfies the triangle inequality [17].

3. BACKGROUND

We provide a brief introduction to Adaptive A* and Path-AA* since Tree-AA* uses their principles. Both incremental heuristic search algorithms apply to path planning with the freespace assumption and use A* searches to find a minimum-cost path from the current state of the agent to the goal state. They perform A* searches from the current state of the agent to the goal state (= forward search), which is the most efficient search direction for Adaptive A* [10] and the only possible search direction for Path-AA*. As the agent follows the planned path, it senses additional obstacles, which increase the costs of moving from some states to their successor states (often to infinity). When one or more edges with increased costs are on the planned path between the current state of the agent and the goal state, Path-AA* replans a minimum-cost path from the current state of the agent to the goal state and then repeats the process until it reaches the goal state or it can no longer find a path to the goal state.

3.1 Adaptive A*

Adaptive A* [11] is based on the following "update principle," which was first described in [8] in the context of hierarchical A* search: If the h-value of every state expanded by an A* search with consistent h-values is set to the f-value of the goal state minus the g-value of the state, then the resulting h-values are again consistent and weakly dominate the original h-values. Thus, an A* search with the resulting h-values expands no more states than an A* search with the original h-values (and the same tie-breaking strategy). The goal state has to remain unchanged from A* search to A* search but the start state can change and some movement costs can increase (but not decrease). Thus, Adaptive A* can be used for path planning with the freespace assumption, which typically makes the A* searches more focused and thus speeds them up. The properties of Adaptive A* are explained in more detail in [11]. We make extensive use of the property that Adaptive A* sets the h-values of all states on the minimum-cost path to their goal costs.

3.2 Path Adaptive A* (Path-AA*)

Path Adaptive A* (Path-AA*) [7] is based on the following "termination principle" and extends the "path-caching strategy," which was first described in [8] in the context of hierarchical A* search: If one knows a minimum-cost path

from some state to the goal state (= reusable path) and the h-values of all states on the reusable path are equal to their goal costs, then a forward A* can terminate when it is about to expand a state on the reusable path (including the goal state). Thus, a Path-AA* search can terminate earlier than a regular A* search, that terminates only when it is about to expand the goal state. The (minimum-cost) path from the current state of the agent to the state on the reusable path and the (minimum-cost) path from the state on the reusable path to the goal state along the reusable path then form a minimum-cost path from the current state of the agent to the goal state. The properties of Path-AA* are explained in more detail in [7].

4. TREE ADAPTIVE A* (TREE-AA*)

Path-AA* was the first search algorithm to combine incremental heuristic search algorithms of the two above classes in a novel way. It is often faster than D* Lite, an alternative state-of-the-art incremental heuristic search algorithm [7], but has an important limitation: Path-AA* reuses only one path for the next A* search, namely a suffix of the minimum-cost path of the current A* search (= reusable path). In complex terrain, including terrain with large obstacles, the next A* search is unlikely to expand a state on that path far away from the goal state and thus unlikely to terminate much earlier than a regular A* search. We introduce Tree Adaptive A* (Tree-AA*) to address this limitation. Tree-AA* generalizes Path-AA* to reuse suffixes of the minimum-cost paths of the current and all previous A* searches (= reusable tree). It maintains minimum-cost paths from several states to the goal state organized in form of a tree rooted in the goal state. If one knows minimum-cost paths from several states to the goal state (= reusable tree) and the h-values of all states in the reusable tree are equal to their goal costs, then a forward A* search can terminate when it is about to expand a state in the reusable tree (including the goal state), for the same reasons as in the context of Path-AA*. Tree-AA* needs to support two operations, namely adding a path to the reusable tree and removing paths from the reusable tree:

- **Adding a Path to the Reusable Tree:** When an A* search of Tree-AA* terminates because it is about to expand a state in the reusable tree (including the goal state), then Tree-AA* adds the path from the current state of the agent to the state in the reusable tree to the reusable tree. It does this because the (minimum-cost) path from the current state of the agent to the state in the reusable tree and the (minimum-cost) path from the state in the reusable tree to the goal state along the branch of the reusable tree form a minimum-cost path from the current state of the agent to the goal state (since Adaptive A* finds minimum-cost paths) and the h-values of all states on the path are equal to their goal costs (since Adaptive A* updates the h-values this way).
- **Removing Paths from the Reusable Tree:** When the costs of edges in the reusable tree increase, then Tree-AA* uses the largest prefix of the reusable tree that does not contain edges with increased costs. (By prefix of a tree we mean the top part of the tree that includes its root.) It does this because all branches of the resulting tree are minimum-cost paths from some state to the goal state and the h-values of all states in the resulting tree are still equal to their goal costs. When the cost of an edge from state s to state s' increases, then Tree-AA* finds the largest prefix of the reusable tree by removing both the edge and the subtree rooted in state s from the reusable tree.

4.1 Implementation of the Reusable Tree

Tree-AA* implements the above two operations efficiently by maintaining two variables for every state and three variables for every path $x = s_0 \dots s_n$ in the reusable tree, where s_0 is the state at the start of the path and s_n is the state at the end of the path that an A* search was about to expand when it terminated. We say that the states $s_0 \dots s_{n-1}$ belong to path x. Every path in the reusable tree is identified with a unique integer that corresponds to the number of the A* search after which it was added to the reusable tree (starting with one). Every path in the reusable tree is the prefix of a minimum-cost path from some state to the goal state. The h-values of all states on the path are equal to their goal costs and thus are strictly monotonically decreasing along the path. The variables are as follows:

- $Id(s)$ is the path in the reusable tree which state s belongs to. These values are initialized to zero, which means that state s is either the goal state or not in the reusable tree.
- $Reusabletree(s)$ is the parent of state s in the reusable tree if state s is a non-goal state in the reusable tree.
- $H_{max}(x)$ is the largest h-value of any state $s_0 \dots s_n$, that is, $H_{max}(x) = h(s_0)$. $H_{max}(0) = -1$, as explained below.
- $H_{min}(x)$ is the smallest h-value of any state $s_0 \dots s_n$, that is, $H_{min}(x) = h(s_n)$.
- $Paths(x)$ is the set of all paths in the reusable tree that connect to one of the states $s_0 \dots s_{n-1}$. These paths "feed into" path x.

Figure 1(a) shows a fictitious example of a reusable tree. The terrain is discretized into cells that are either blocked or unblocked, a common practice in the context of real-time computer games [1]. We assume for simplicity that the agent can move in the four main compass directions with cost one and thus operates on undirected four-neighbor grids. Values are shown only for cells that are in the reusable tree. A cell is black if it is blocked and this fact is known to the agent. The Id-value of a cell is shown in its upper left corner. The h-value of a cell is shown in its upper right corner. The Reusabletree-pointer of a cell is shown as an arrow. The arrows thus show the reusable tree, that consists of five paths. Path 1 is D2 D3 D4 D5 D6, path 2 is E3 E4 E5 D5, path 3 is B6 C6 D6, path 4 is C2 C3 C4 D4, and path 5 is B2 B3 C3. The table shows the variables of all paths.

4.2 Implementation of the Operations

The goal state is always in the reusable tree. Tree-AA* could check whether $Id(s) > 0$ when it needs to check whether a non-goal state s is in the reusable tree. However, this would require it to set $Id(s)$ to zero when it removes a non-goal state from the reusable tree, which is expensive since Tree-AA* often needs to remove whole paths from the reusable tree. Thus, Tree-AA* checks whether $h(s) \leq H_{max}(Id(s))$ when it needs to check whether a non-goal state s is in the reusable tree. (The goal state fails this test.)

Tree-AA* can now remove a path x from the reusable tree by setting $H_{max}(x)$ to $H_{min}(x)$ without having to set $Id(s)$ to zero for all states s that belong to path x. Thus, $Id(s) = 0$ is not necessarily true for states s not in the reusable tree. There are two subtleties here. Non-goal states s that have not yet been part of the reusable tree correctly fail the test since $H_{max}(Id(s)) = H_{max}(0) = -1$. Non-goal states s that were part of the reusable search tree but have subsequently been removed correctly fail the test since they have an h-value larger than $H_{max}(Id(s))$. Tree-AA* adds a path to the reusable tree and removes paths from the reusable tree as follows:

- **Adding a Path to the Reusable Tree:** Tree-AA* adds a path $x = s_0 \dots s_n$ to the reusable tree as follows. It equates x with the number of the current A* search (as given by the variable *Counter*) to identify the path with a unique integer. It inserts x into the set $Paths(Id(s_n))$ if s_n is a non-goal state since path x feeds into the path that state s_n belongs to [Lines 9-10]. (The line numbers refer to the pseudo code of Tree-AA* in Figure 2). It sets $H_{min}(x)$ to $h(s_n)$ [Line 11] and $H_{max}(x)$ to $h(s_0)$ [Line 12] since the h-values are strictly monotonically decreasing along the path. It sets $Paths(x)$ to the empty set [Line 13] since no paths feed into path x yet. It sets $Id(s)$ to x and $Reusabletree(s)$ to the successor of state s on path x for all states $s_0 \dots s_{n-1}$ [Lines 14-18] since the states $s_0 \dots s_{n-1}$ belong to path x. The runtime of adding a path to the reusable tree is thus basically proportional to the number of states on the path.

- **Removing Paths from the Reusable Tree:** When the cost of an edge from state s to state s' increases, then Tree-AA* removes paths from the reusable tree as follows. If $Reusabletree(s) = s'$ then the edge might be in the reusable tree [Lines 69-70], namely on path $x := Id(s)$ [Line 20]. In this case, Tree-AA* sets $H_{max}(x)$ to $h(s')$ (if it was larger) to shorten path x [Lines 21-22]. It also removes all paths $x' \in Paths(x)$ with $H_{max}(x) \leq H_{min}(x')$ from the set $Paths(x)$ and schedules them for removal from the reusable tree [Lines 24-27]. For each path x scheduled for removal with $H_{max}(x) > H_{min}(x)$, it sets $H_{max}(x)$ to $H_{min}(x)$, removes all paths $x' \in Paths(x)$ from the set $Paths(x)$ and schedules them recursively for removal [Lines 28-34]. The runtime of removing paths from the reusable tree when the cost of one edge increases is thus basically proportional to the number of paths in the reusable tree, which is bounded by the number of A* searches performed so far.

Figure 1(b) continues the fictitious example from Figure 1(a) by showing the reusable tree after Tree-AA* removed paths from the reusable tree after C3 became blocked. Tree-AA* shortened path 4 to C4 D4 and removed path 5. Figure 1(c) shows the reusable tree after Tree-AA* added B3 B4 C4 D4 to the reusable tree after an A* search.

4.3 Pseudocode

We now put all of our insights together. The pseudo code of Tree-AA* in Figure 3 proceeds as follows:[1] It sets $Id(s)$ to

```
01 procedure InitializeState(s)
02 if (Generated(s) = 0)
03    g(s) := ∞;
04    h(s) := H(s);
05 else if (Generated(s) ≠ Counter)
06    g(s) := ∞;
07 Generated(s) := Counter;

08 procedure AddPath(s)
09 if (s ≠ s_goal)
10    insert Counter into Paths(Id(s));
11 H_min(Counter) := h(s);
12 H_max(Counter) := h(s_start);
13 Paths(Counter) := ∅;
14 while (s ≠ s_start)
15    s_aux := s;
16    s := Searchtree(s);
17    Id(s) := Counter;
18    Reusabletree(s) := s_aux;

19 procedure RemovePaths(s)
20 x := Id(s);
21 if (H_max(x) > h(Reusabletree(s)))
22    H_max(x) := h(Reusabletree(s));
23 QUEUE := ∅;
24 for all x' ∈ Paths(x)
25    if (H_max(x) < H_min(x'))
26       add x' to the end of QUEUE;
27       remove x' from Paths(x);
28 while QUEUE ≠ ∅
29    remove x from the head of QUEUE;
30    if (H_max(x) > H_min(x))
31       H_max(x) := H_min(x);
32       for all x' ∈ Paths(x)
33          add x' to the end of QUEUE;
34          remove x' from Paths(x);

35 function ComputePath()
36 while (OPEN ≠ ∅)
37    remove state s with the smallest g(s) + h(s) value from OPEN;
38    if (s = s_goal OR h(s) ≤ H_max(Id(s)))
         /* s is in reusable tree */
39       for all s' ∈ CLOSED
40          h(s') := g(s) + h(s) − g(s');
41       AddPath(s);
42       return true;
43    insert s into CLOSED;
44    for all s' ∈ Succ(s)
45       InitializeState(s');
46       if (g(s') > g(s) + c(s, s'))
47          g(s') := g(s) + c(s, s');
48          Searchtree(s') := s;
49          if (s' ∈ OPEN)
50             remove s' from OPEN;
51          insert s' into OPEN with value g(s') + h(s');
52 return false;

53 function Main()
54 Counter := 1;
55 H_max(0) := −1;
56 for all s ∈ S
57    Generated(s) := Id(s) := 0;
58    Reusabletree(s) := NULL;
59 while (s_start ≠ s_goal)
60    InitializeState(s_start);
61    g(s_start) := 0;
62    OPEN := CLOSED := ∅;
63    insert s_start into OPEN with value g(s_start) + h(s_start);
64    if (ComputePath() = false)
65       return false; /* failure: the goal state is unreachable */
66    while (h(s_start) ≤ H_max(Id(s_start)))
         /* s_start is non-goal state in reusable tree */
67       s_start := Reusabletree(s_start);
68       for all increased costs c(s, s')
69          if (Reusabletree(s) = s')
70             RemovePaths(s);
71    Counter := Counter + 1;
72 return true; /* success: the goal state has been reached */
```

Figure 2: Tree-AA*

[1]Tree-AA* maintains the following variables for its regular A* searches: *Counter* is the number of the current A* search. *OPEN* is the open list of the current A* search. *CLOSED* is the closed list of the current A* search. *Generated*(*s*) is the number of the last A* search that generated state *s*. A* uses these values to initialize the g-values and h-values of states as needed during an A* search [procedure InitializeState] to avoid having to initialize them for all states before every A* search. *Searchtree*(*s*) is the parent of state *s* in the search tree of the *Generated*(*s*)th A* search. $g(s)$ is the g-value of state *s* during the *Generated*(*s*)th A* search. $h(s)$ is the current h-value of state *s*.

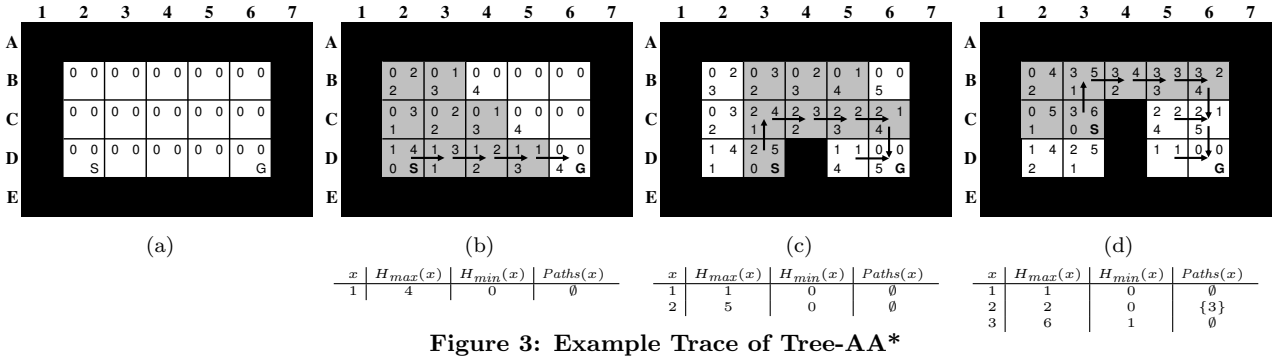


Figure 3: Example Trace of Tree-AA*

zero [Line 57], *Reusabletree*(s) to *NULL* [Line 58] and $h(s)$ (when needed for the first time during an A* search) to the user-given consistent h-value $H(s)$ [Line 4] for all states s. It performs a forward A* search [function ComputePath] until it is about to expand a state s in the reusable tree (including the goal state) [Line 38, termination principle]. It sets the h-value of every expanded state s' to the f-value of state s (which is the same as the f-value of the goal state) minus the g-value of state s' [Lines 39-40, update principle]. It then adds the (minimum-cost) path from the current state of the agent to state s to the reusable tree [procedure AddPath], as described above [Line 41]. It then follows the minimum-cost path from the current state of the agent to the goal state along the branch of the reusable tree [Lines 66-70]. Whenever edge costs increase, it removes paths from the reusable tree [procedure RemovePaths], as described above [Lines 68-70]. If the current state of the agent is no longer in the reusable tree, it performs another forward A* search and then repeats the process until the agent either reaches the goal state or it can no longer find a path to the goal state. The correctness proof of Tree-AA* is basically the same as that of Path-AA* and thus not given here.

4.4 Example Trace of Tree-AA*

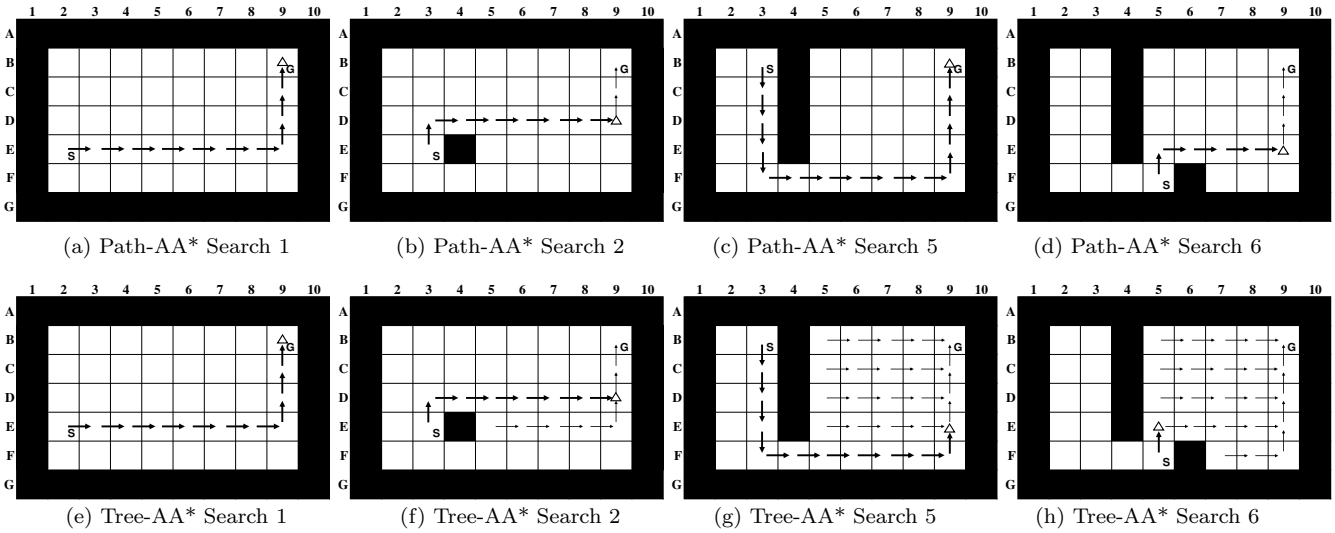
Figure 3(a-d) shows the beginning of a trace of Tree-AA*. The agent always senses the blockage status of its four neighboring cells and can then move to any one of the unblocked neighboring cells with cost one. Its task is to move from start cell S to goal cell G. It assumes that all cells are unblocked except for the blocked cells that it has already sensed. It plans a minimum-cost path from its current cell to the goal cell. As it follows the planned path, it senses additional blocked cells and adds them to its map. When it detects blocked cells on its path, it replans a minimum-cost path from its current cell to the goal cell until it reaches the goal cell or can no longer find a path to the goal cell. The user-given h-values are all zero. A cell is black if it is blocked and this fact is known to the agent. The Id-value of a cell is shown in its upper left corner. The h-value of a cell (after it was updated using the update principle) is shown in its upper right corner. The g-value of a cell is shown in its lower left corner if it was generated during the current A* search. A cell is shaded if it was expanded during the current A* search. The Reusabletree-pointer of a cell is shown as an arrow if it belongs to the reusable tree.

Figure 3(a) shows the initial situation with start cell D2. Figure 3(b) shows that the first A* search of Tree-AA* from D2 to D6 terminates when it is about to expand D6 and returns the D2 D3 D4 D5 D6. Tree-AA* adds the path to the reusable tree and updates the h-values of the expanded states using the update principle. The agent then follows the branch of the reusable tree from D2 to D3, where it senses that D4 is blocked. Tree-AA* removes D2 D3 D4 D5 from the reusable tree. Figure 3(c) shows that the second A* search of Tree-AA* from D3 to D6 terminates when it is about to expand D6 and returns D3 C3 C4 C5 C6 D6. It expands fewer cells than an A* search with the user-given zero h-values (which also expands B2, C2 and D2), illustrating the speed up achieved with the update principle. Tree-AA* adds the path to the reusable tree and updates the h-values. The agent then moves from D3 to C3, where it senses that C4 is blocked. Tree-AA* removes D3 C3 C4 C5 from the reusable tree. Figure 3(d) shows that the third A* search of Tree-AA* from C3 to D6 terminates when it is about to expand C6 and returns C3 B3 B4 B5 B6 C6. It terminates earlier than a regular A* search with the same h-values (which also expands C6 and terminates only when it is about to expand D6), illustrating the speed up achieved with the termination principle.

4.5 Comparison of Path-AA* and Tree-AA*

Figure 4 shows an example that illustrates the difference between Path-AA* (top) and Tree-AA* (bottom) on the same navigation problem. A cell is black if it is blocked and this fact is known to the agent. A Reusabletree-pointer is shown as a thick arrow if it was added to the reusable path (or reusable tree) in the current A* search and as a thin arrow if it was added in a previous A* search. A triangle marks the cell that the current A* search was about to expand before it terminated.

Figures 4(a) and 4(e) show that the first A* searches of Path-AA* and Tree-AA* produce the same result. The agent then moves from E2 to E3, where it senses that E4 is blocked. Path-AA* removes E2 E3 E4 E5 E6 E7 E8 E9 D9 from the reusable path. Figure 4(b) shows that the second A* search of Path-AA* terminates when it is about to expand D9, and Path-AA* adds E3 D3 D4 D5 D6 D7 D8 D9 (shown as thick arrows) to the reusable path. On the other hand, Tree-AA* removes E2 E3 E4 E5 from the reusable tree. Figure 4(f) shows that the second A* search of Tree-AA* also terminates when it is about to expand D9, but Tree-AA* adds E3 D3 D4 D5 D6 D7 D8 D9 (shown as thick arrows) to the reusable tree. Thus, Tree-AA* removes fewer cells, which might allow its future A* searches to terminate earlier. The agent then moves from E3 to D3, where it senses that D4 is blocked. Path-AA* and Tree-AA* perform their third A* searches. The agent then moves from



(a) Path-AA* Search 1 (b) Path-AA* Search 2 (c) Path-AA* Search 5 (d) Path-AA* Search 6

(e) Tree-AA* Search 1 (f) Tree-AA* Search 2 (g) Tree-AA* Search 5 (h) Tree-AA* Search 6

Figure 4: Comparison of Path-AA* (top) and Tree-AA* (bottom)

D3 to C3, where it senses that C4 is blocked. Path-AA* and Tree-AA* perform their fourth A* searches. We omit the details of the third and fourth A* searches due to space constraints. The agent then moves from C3 to B3, where it senses that B4 is blocked. Figure 4(c) shows that the fifth A* search of Path-AA* terminates when it is about to expand B9. Figure 4(g) shows that the fifth A* search of Tree-AA* terminates already when it is about to expand E9, illustrating the speed up resulting from reusing the paths from all previous A* searches. The agent then moves from B3 to F5, where it senses that F6 is blocked. Figure 4(d) shows that the sixth A* search of Path-AA* terminates when it is about to expand E9. Figure 4(h) shows that the fifth A* search of Tree-AA* terminates already when it is about to expand F5, again illustrating the speed up resulting from reusing the paths from all previous A* searches.

5. TREE-AA*-BACK

An A* search with consistent h-values guarantees that the g-value of every expanded state is equal to the cost of a minimum-cost path from the start state to the expanded state [17]. Thus, if the first A* search of Tree-AA* searches from the goal state to the current state of the agent (= backward search), then the resulting search tree restricted to the expanded states is a reusable tree. All subsequent A* searches of Tree-AA* must be forward searches. We refer to the resulting version of Tree-AA* as Tree-AA*-Back. The reusable tree after the first A* search of (standard) Tree-AA* contains only the expanded states on the minimum-cost path from the current state of the agent to the goal state, while the reusable tree after the first A* search of Tree-AA*-Back contains all expanded states, which might allow future A* searches to terminate earlier.

6. EXPERIMENTAL EVALUATION

We compare Tree-AA* to the top incremental heuristic search algorithms in the context of goal-directed navigation in unknown grids, namely Adaptive A* (that uses forward searches, which is the most efficient search direction for Adaptive A*), Path-AA* (that uses forward searches, which is the only possible search direction for Path-AA*)

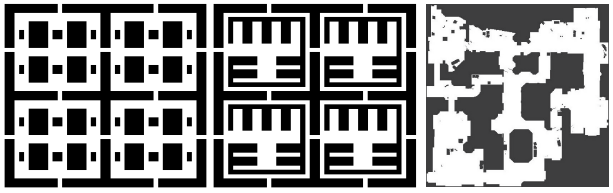


Figure 5: Maps

and D* Lite (that uses backward searches, which is the only possible search direction for D* Lite). Tree-AA* uses forward searches but we also implement Tree-AA*-Back from Section 5, whose first A* search is a backward search. For fairness, all search algorithms use binary heaps as priority queues and break ties among states with the same f-values in favor of states with larger g-values (which is known to be a good tie-breaking strategy), with one exception: During the first search, Tree-AA*-Back (1) breaks ties among states with the same f-values in favor of states with larger g-values but Tree-AA*-Back (2) breaks ties in favor of states with smaller g-values. During the remaining A* searches, both versions of Tree-AA*-Back break ties in favor of states with larger g-values.

6.1 Experimental Setup

We used four-neighbor grids as examples since they result in integer-valued g-values, h-values and f-values. We use eight-neighbor grids in the experiments since they are often preferred in practice, for example in video games [3, 2]. The agent always senses the blockage status of its eight neighboring cells and can then move to any one of the unblocked neighboring cells with cost one for horizontal or vertical movements and cost $\sqrt{2}$ for diagonal movements. The user-given h-values are the octile distances [3].

We use two indoor office maps of size $1,000 \times 1,000$ cells, where the size of each room is 20×20 cells. Figure 5 (left and center) shows areas of 2×2 rooms in office maps 1 and 2. We also use a computer game map of size $3,000 \times 3,000$ cells adapted from Counter-Strike (courtesy of Vadim Bulitko from the University of Alberta). Figure 5 (right) shows the game map. We average our experimental results

	Office Map 1								
	All A* Searches					First A* Search		Remaining A* Searches	
	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)
Adaptive A*	1,333	652	473	0.127	82.8	900	0.48	472	0.126
D* Lite	1,520	474	562	0.177	83.9	25,836	9.25	508	0.158
Path-AA*	1,333	652	114	0.043	28.0	900	0.48	113	0.042
Tree-AA*	1,333	652	13	0.005	3.3	900	0.45	11	0.004
Tree-AA*-Back (1)	1,337	615	55	0.015	9.2	25,836	5.84	13	0.006
Tree-AA*-Back (2)	1,334	652	212	0.059	38.5	134,669	36.42	6	0.003
	Office Map 2								
Adaptive A*	6,606	3,924	512	0.130	445.8	900	0.47	512	0.130
D* Lite	5,821	2,429	285	0.092	223.5	48,013	17.14	265	0.085
Path-AA*	6,637	3,094	88	0.033	102.1	900	0.47	88	0.033
Tree-AA*	6,637	3,094	19	0.006	18.6	900	0.48	19	0.006
Tree-AA*-Back (1)	6,413	2,903	36	0.008	23.2	48,013	10.77	19	0.004
Tree-AA*-Back (2)	6,635	3,088	60	0.016	49.4	132,795	36.11	17	0.004
	Game Map								
Adaptive A*	4,842	2,524	3,424	1.094	2,761.3	2,671	1.62	3,424	1.094
D* Lite	5,723	2,491	1,547	0.790	1,967.9	5,821	3.64	1,545	0.789
Path-AA*	4,841	2,520	1,442	0.525	1,323.0	2,671	1.65	1,442	0.525
Tree-AA*	4,841	2,519	1,353	0.437	1,100.8	2,671	1.57	1,353	0.437
Tree-AA*-Back (1)	4,955	2,378	1,320	0.407	967.8	5,821	2.29	1,322	0.406
Tree-AA*-Back (2)	4,841	2,519	1,610	0.466	1,173.9	1,203,590	392.12	1,132	0.310

(a) = moves per test case; (b) = A* searches per test case; (c) = cell expansions per A* search; (d) = runtime per A* search; (e) = runtime per test case; (f) = cells expansions of the ↓rst A* search; (g) = runtime of the ↓rst A* search; (h) = cell expansions per A* search (excluding the ↓rst A* search); (i) = runtime per A* search (excluding the ↓rst A* search).

Table 1: Experimental Results

over 500 test cases with a reachable goal cell for each map. For each test case in the office maps, we ensure that the start and goal cells are far apart by independently choosing the x-coordinate of the start cell randomly between 1 and 100 and the x-coordinate of the goal cell randomly between 901 and 1,000. We independently choose the y-coordinates of the start and goal cells randomly between 1 and 1,000. Similarly, for each test case in the game map, we independently choose the x-coordinate of the start cell randomly between 1 and 300 and the x-coordinate of the goal cell randomly between 2,701 and 3,000. We independently choose the y-coordinates of the start and goal cells randomly between 1 and 3,000.

6.2 Experimental Results

We report two measures of the difficulty of goal-directed navigation problems in unknown grids, namely (a) the number of moves of the agent per test case and (b) the number of A* searches per test case until the agent reaches the goal cell. These measures vary slightly among the compared search algorithms since they can determine different minimum-cost paths, in which case the agents that follow the paths might sense different blocked cells, which can make their trajectories diverge. We report three measures of the efficiency of the search algorithms, namely (c) the number of expanded cells per A* search, (d) the runtime per A* search in milliseconds and (e) the runtime per test case in milliseconds on a Linux PC with a Pentium CoreQuad 2.33 GHz CPU and 8 GB RAM. Since the number of A* searches is (approximately) the same for all search algorithms, their runtimes per test case are largely proportional to their runtimes per A* search. Therefore, the main measure of the efficiency of the search algorithms is their runtime per A* search. In order to gain more insight into the behavior of the search algorithms, we divide each test case into two parts, namely the first A* search and the remaining A* searches other than the first one. For the first A* search, we report (f) the number of expanded cells and (g) the runtime in milliseconds. For the remaining A* searches other than the first one, we report (h) the number of expanded cells per A* search and (i) the runtime per A* search in milliseconds. Table 1 shows the following relationships:

First, Tree-AA* has a smaller runtime per A* search than Adaptive A* for all maps because it has a smaller number of cell expansions per A* search due to the speed up achieved with the termination principle. For example, Tree-AA* expands only about 2.7, 3.7 and 39.5 percent of the cells per A* search that Adaptive A* expands in office maps 1 and 2 and the game map, respectively. It thus runs by factors of 25.4, 21.7 and 2.5 faster per A* search.

Second, Tree-AA* has a smaller runtime per A* search than D* Lite for all maps, which is due to two reasons. First, Tree-AA* has a smaller number of cell expansions per A* search perhaps due to the speed up achieved with the update and termination principles. For example, Tree-AA* expands only about 2.3, 6.7 and 87.5 percent of the cells per A* search that D* Lite expands in office maps 1 and 2 and the game map, respectively. Second, Tree-AA* has a smaller number of heap percolates per A* search due to both the smaller number of cell expansions per A* search (resulting in fewer heap operations) and a smaller number of cells in the open list during each A* search (resulting in fewer heap percolates per heap operation). The smaller number of cells in the open list is due to each A* search of Tree-AA* starting with an empty open list rather than the open list at the end of the previous A* search. Tree-AA* thus runs by factors of 35.4, 15.3 and 1.8 faster per A* search.[2]

Third, Tree-AA* has a smaller runtime per A* search than Path-AA* for all maps because it has a smaller number of cell expansions per A* search due to the speed up achieved with a reusable tree rather than a reusable path. For example, Tree-AA* expands only about 11.4, 21.6 and 93.8 percent of the cells per A* search that Path-AA* expands in office maps 1 and 2 and the game map, respectively. It thus runs by factors of 8.6, 5.5 and 1.2 faster per A* search.

Fourth, both versions of Tree-AA*-Back have a larger runtime for the first A* search than (standard) Tree-AA* for all maps but tend to have a smaller runtime per A* search for the remaining A* searches, which is due to the following reasons: The first A* search of Tree-AA* is a forward search, while the first A* search of both versions of Tree-AA*-Back

[2]To understand better in which situations Tree-AA* has an advantage over D* Lite, we also perform experiments on maps of size 500 × 500 cells and independently block 20, 30, 40, 50 and 60 percent of randomly chosen cells, respectively. We average our experimental results over 500 test cases with a reachable goal cell for each map. For each test case, we choose the x-coordinate of the start cell randomly between 1 and 50 and the x-coordinate of the goal cell randomly between 451 and 500. We independently choose the y-coordinates of the start and goal cells randomly between 1 and 500. Tree-AA* expands only about 11.2, 12.5, 14.9, 17.4 and 87.0 percent of the cells per A* search that D* Lite expands, respectively.

is a backward search. Backward A* searches expand more cells than forward A* searches [10]. Thus, the runtime of the first A* search of both versions of Tree-AA*-Back is larger than the one of Tree-AA*. The first A* search of Tree-AA* yields a reusable path, while the first A* search of both versions of Tree-AA*-Back yields a reusable tree. During the first search, Tree-AA*-Back (1) breaks ties among states with the same f-values in favor of states with larger g-values but Tree-AA*-Back (2) breaks ties in favor of states with smaller g-values. Thus, the runtime of the first A* search and the size of the reusable tree of Tree-AA*-Back (2) are larger than the ones of Tree-AA*-Back (1). The larger the reusable tree, the more it speeds up the first few remaining A* searches of Tree-AA*-Back due to the termination principle until the reusable trees of Tree-AA*-Back and Tree-AA* are about equally large. For example, Tree-AA*-Back (1) and, given in parentheses, Tree-AA*-Back (2) expand about 423.1 (1630.8), 189.5 (315.8) and 97.6 (119.0) percent of the cells per A* search that Tree-AA* expands in office maps 1 and 2 and the game map, respectively. They thus run by factors of 3.0 (11.8), 1.3 (2.7), and 0.9 (1.1) more slowly per A* search. However, Tree-AA*-Back (1) and (2) expand about 118.2 (54.5), 100.0 (89.5) and 97.7 (83.7) percent of the cells per A* search that Tree-AA* expands for the remaining A* searches other than the first one. They thus run by factors of 0.7 (1.3), 1.5 (1.5) and 1.1 (1.4) faster per A* search for the remaining A* searches. Therefore, Tree-AA*-Back (1) and (2) tend to run faster than Tree-AA* for applications where the first A* search can be performed offline before the goal-directed navigation problem in unknown terrain starts and its runtime thus does not matter, as is often the case in robotics.

7. CONCLUSIONS

In this paper, we introduced a new incremental heuristic search algorithm called Tree Adaptive A*. So far, there existed incremental heuristic search algorithms (such as Adaptive A*) that make the h-values of the current A* search more informed and incremental heuristic search algorithms (such as D* Lite) that change the search tree of the current A* search to the search tree of the next A* search. Tree Adaptive A* uses the update principle of Adaptive A* to make the h-values of the current A* search more informed. It also uses the termination principle of Path Adaptive A* to terminate A* searches earlier than regular A* searches but generalizes it to reuse suffixes of the minimum-cost paths of the current and all previous A* searches (= reusable tree). Overall, Tree Adaptive A* is the first incremental heuristic search algorithm to combine the principles of both classes of incremental heuristic search algorithms and can run faster than Adaptive A*, Path Adaptive A* and D* Lite, the top incremental heuristic search algorithms in the context of goal-directed navigation in unknown grids.

8. REFERENCES

[1] M. Bjornsson, M. Enzenberger, R. Holte, J. Schaeffer, and P. Yap. Comparison of different abstractions for pathfinding on maps. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1511–1512, 2003.

[2] V. Bulitko, Y. Bjornsson, M. Luvstrek, J. Schaeffer, and S. Sigmundarson. Dynamic control in path-planning with real-time heuristic search. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pages 49–56, 2007.

[3] V. Bulitko and G. Lee. Learning in real-time search: A unifying framework. *Journal of Artificial Intelligence Research*, 25:119–157, 2006.

[4] H. Choset, S. Thrun, L. Kavraki, W. Burgard, and K. Lynch. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press, 2005.

[5] D. Ferguson and A. Stentz. Using interpolation to improve path planning: The Field D* algorithm. *Journal of Field Robotics*, 23(2):79–101, 2006.

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

[7] C. Hernández, P. Meseguer, X. Sun, and S. Koenig. Path-Adaptive A* for incremental heuristic search in unknown terrain [short paper]. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pages 358–361, 2009.

[8] R. Holte, M. Perez, R. Zimmer, and A. MacDonald. Hierarchical A*: Searching abstraction hierarchies efficiently. In *Proceedings of the National Conference on Artificial Intelligence*, pages 530–535, 1996.

[9] S. Koenig, D. Furcy, and C. Bauer. Heuristic search-based replanning. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems*, pages 294–301, 2002.

[10] S. Koenig and M. Likhachev. Fast replanning for navigation in unknown terrain. *Transaction on Robotics*, 21(3):354–363, 2005.

[11] S. Koenig and M. Likhachev. A new principle for incremental heuristic search: Theoretical results. In *Proceedings of the International Conference on Autonomous Planning and Scheduling*, pages 410–413, 2006.

[12] S. Koenig, M. Likhachev, Y. Liu, and D. Furcy. Incremental heuristic search in artificial intelligence. *Artificial Intelligence Magazine*, 25(2):99–112, 2004.

[13] S. Koenig, C. Tovey, and Y. Smirnov. Performance bounds for planning in unknown terrain. *Artificial Intelligence*, 147(1-2):253–279, 2003.

[14] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun. Anytime Dynamic A*: An anytime, replanning algorithm. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pages 262–271, 2005.

[15] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun. Anytime search in dynamic graphs. *Artificial Intelligence*, 172(14):1613–1643, 2008.

[16] K. Matsuta, H. Kobayashi, and A. Shinohara. Multi-target Adaptive A*. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems*, pages 1065–1072, 2010.

[17] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1985.

[18] A. Stentz. The Focussed D* algorithm for real-time replanning. In *Proceedings of International Joint Conference in Artificial Intelligence*, pages 1652–1659, 1995.

[19] X. Sun, S. Koenig, and W. Yeoh. Generalized Adaptive A*. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 469–476, 2008.