**Noname manuscript No.**
(will be inserted by the editor)

# Reusing Cost-Minimal Paths for Goal-Directed Navigation in Partially Known Terrains

**Carlos Hernández · Tansel Uras · Sven Koenig · Jorge A. Baier · Xiaoxun Sun · Pedro Meseguer**

Received: date / Accepted: date

**Abstract** Situated agents frequently need to solve search problems in partially known terrains in which the costs of the arcs of the search graphs can increase (but not decrease) when the agents observe new information. An example of such search problems is goal-directed navigation with the freespace assumption in partially known terrains, where agents repeatedly follow cost-minimal paths from their current locations to given goal locations. Incremental heuristic search is an approach for solving the resulting sequences of similar search problems potentially faster than with classical heuristic search, by reusing information from previous searches to speed up its current search. There are two classes of incremental heuristic search

Carlos Hernández
Departamento de Ingeniería Informática
Universidad Católica de la Ssma. Concepción
Concepción, Chile
E-mail: chernan@ucsc.cl

Tansel Uras
Sven Koenig
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781, USA
E-mail: {turas,skoenig}@usc.edu

Jorge A. Baier
Departamento de Ciencia de la Computación
Pontificia Universidad Católica de Chile
Santiago, Chile
E-mail: jabaier@ing.puc.cl

Xiaoxun Sun
Google
1600 Amphitheatre Parkway
Mountain View, CA 94043, USA
E-mail: xiaoxunsun@google.com

Pedro Meseguer
IIIA - CSIC
Campus Universitat Autonòma de Barcelona
08193 Bellaterra, Spain
E-mail: pedro@iiia.csic.es

algorithms, namely those that make the $h$-values of the current search more informed (such as Adaptive A*) and those that reuse parts of the A* search trees of previous searches during the current search (such as D* Lite). In this article, we introduce Path-Adaptive A* and its generalization Tree-Adaptive A*. Both incremental heuristic search algorithms terminate their searches before they expand the goal state, namely when they expand a state that is on a provably cost-minimal path to the goal. Path-Adaptive A* stores a single cost-minimal path to the goal state (the reusable path), while Tree-Adaptive A* stores a set of cost-minimal paths to the goal state (the reusable tree), and is thus potentially more efficient than Path-Adaptive A* since it uses information from all previous searches and not just the last one. Tree-Adaptive A* is the first incremental heuristic search algorithm that combines the principles of both classes of incremental heuristic search algorithms. We demonstrate experimentally that both Path-Adaptive A* and Tree-Adaptive A* can be faster than Adaptive A* and D* Lite, two state-of-the-art incremental heuristic search algorithms for goal-directed navigation with the freespace assumption.

## 1 Introduction

Situated agents, such as robots and game characters, have to be able to navigate from their current location to a given destination [3]. However, they may not know a map of the terrain initially, and their sensors may only be able to observe the immediate neighborhood of their current location. One approach to navigation, popular robotics [17], uses the freespace assumption: The agents plan a cost-minimal path from their current location to their destination under the freespace assumption, which is the assumption that the terrain is traversable except for the blocked cells that they have already observed. When they observe additional blocked cells as they follow the planned path, they add them to their map. If one or more blocked cells are on their path between their current location and their destination, they replan a cost-minimal path from their current location to their destination and then repeat the process until they either reach their destination or can no longer find a path to their destination (in which case it is unreachable).

In this paper we study path planning with the freespace assumption for goal-directed navigation in unknown, static terrain. More generally, we study the problem of repeatedly following a cost-minimal path from the current location to a given destination where the arc costs can increase but not decrease, as in the case of discovering a new blocked cell in the environment. Path planning with the freespace assumption interleaves path planning with movements and thus requires repeated searches. These searches need to be fast since agents typically 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 with heuristic search algorithms such as A* [6], search can be time consuming if the terrain is large or if many agents perform simultaneous searches. Incremental heuristic search algorithms use information from the current and previous searches to solve similar future search problems potentially faster than classical heuristic search algorithms, that solve all search problems from scratch [16]. They have been

used to speed up A* searches in the context of both symbolic planning [12] and path planning [16].

There are two classes of incremental heuristic search algorithms:

- **Class 1:** Incremental heuristic search algorithms of Class 1 make the $h$-values of the current A* search more informed, which can speed up future A* searches by making them more focused. Examples include Adaptive A* (AA*) [15], Generalized Adaptive A* [25] and Multi-Target Adaptive A* [21].
- **Class 2:** Incremental heuristic search algorithms of Class 2 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. Examples include D* [23] and D* Lite [14]. D* Lite is as fast as D* but much simpler. Both of them are typically faster for path planning with the freespace assumption than versions of AA* [15]. Versions of them have been used as part of path planners in a wide range of fielded robotics systems [18,5,19], including the winning DARPA Urban Challenge entry Boss and the Mars rovers Opportunity and Spirit.

Our key observation is that the suffix of the current cost-minimal path without arc cost changes remains a cost-minimal path when the agent observes arc cost increases on the path. Path-Adaptive A* (Path-AA*) [9] reuses the suffix of the cost-minimal path of the current forward A* search (= reusable path) to terminate its next forward A* search earlier than a regular forward A* search. Tree-Adaptive A* (Tree-AA*) [10] generalizes Path-AA* by reusing suffixes of the cost-minimal paths of the current and all previous forward A* searches (= reusable tree) to terminate the next forward A* search even earlier. Thus, Tree-AA* combines incremental heuristic search algorithms of Classes 1 and 2 in a novel way since the reusable tree of Tree-AA* is similar to the search tree of incremental heuristic search algorithms from Class 2, such as D* Lite. However, Tree-AA* changes the reusable tree via forward (rather than backward) A* searches, which is a novel way of maintaining the search tree. We demonstrate experimentally that both Path-AA* and Tree-AA* can be faster than AA* and D* Lite, the two state-of-the-art incremental heuristic search algorithms for path planning with the freespace assumption.

Some of the contributions of this paper have been published in conference papers, namely [9] and [10]. This article includes new material that has not been presented before. In particular:

- We present lazy versions of Path-AA* and Tree-AA*.
- We include proofs for termination and optimality of Path-AA* and Tree-AA* (Theorems 1 and 2).
- We perform a systematic experimental analysis of the performance of the algorithms as a function of the difficulty of the search problems.
- We extend previously published experimental results for unknown terrain by including new results for a large number of game maps, office maps and maps with randomly blocked cells.
- We extend previously published experimental results by including a large number of results for partially known terrain.
- Finally, we perform an experimental evaluation of different priority queue implementations.

The rest of the paper is organized as follows. In Section 2, we present the notation used in the paper. In Section 3, we explain basic concepts of heuristic

search. In Section 4, we present basic concepts of incremental heuristic search. In Section 5, we describe Path-AA*, including its pseudocode and its properties. In Section 6, we describe Tree-AA*. In Section 7, we present a detailed experimental analysis. In Section 8, we present some variants of Path-AA* and Tree-AA*.

## 2 Background

Our agents move on a directed graph $G = (S, A)$, where $S$ is a finite set of states, and $A \subseteq S \times S$ is a finite set of arcs. There are two distinguished states: $s_{current} \in S$ is the current state of the agent, and $s_{goal} \in S$ is the goal state. The set $Succ(s) := \{t \in S \mid (s,t) \in A\}$ is the set of successor states of state $s$. An arc $a = (s,t) \in A$ represents that the agent can move from state $s$ to state $t$. $c(s,t)$ is the cost of following the arc from state $s$ to state $t$, where $c(s,t) > 0$. A path from state $s$ to state $t$ is a sequence of arcs $(s_0,t_0),\ldots,(s_n,t_n)$, such that $s_0 = s$, $t_n = t$, and $t_i = s_{i+1}$ for all $0 \leq i < n$. The cost of a path is the sum of the costs of the arcs in the path. A cost-minimal path from state $s$ to state $t$ is a path with the minimum cost among all the paths from state $s$ to state $t$. $d(s,t)$ is the cost of a cost-minimal path from $s \in S$ to $t \in S$. We call $d(s, s_{goal})$ the *goal cost* of state $s$.

A* is the basis of all incremental heuristic search algorithms discussed in this article. A* utilizes a heuristic function to help guide the search. A heuristic function $h : S \mapsto \mathbb{R}_{\geq 0}$ estimates the goal cost of states (that is, $d(s, s_{goal})$). $h$ is said to be *consistent* iff, for any states $s \in S$ and $t \in S$, $h(s) \leq d(s,t) + h(t)$ and $h(s_{goal}) = 0$. $h$ is said to be *admissible* iff, for any state $s \in S$, $0 \leq h(s) \leq d(s, s_{goal})$. A consistent heuristic function is also admissible [22]. A heuristic function $h$ is said to *weakly dominate* another heuristic function $h'$ iff, for any state $s \in S$, $h(s) \geq h'(s)$. We assume that there is a user-provided consistent heuristic function, denoted by $H$.

Our description of A* follows [26]. For each state $s$, A* maintains the following values throughout the search: The parent, denoted by $parent(s)$, keeps track of the state that comes before state $s$ on the path from state $s_{current}$ to state $s$ that has been found so far. It is initially undefined for all states. The $g$-value, denoted by $g(s)$, keeps track of the cost of the path from state $s_{current}$ to state $s$ that has been found so far. It is initially 0 for state $s_{current}$ and infinity for all other states. Any state with a finite $g$-value is said to be *generated*. The $h$-value, denoted by $h(s)$, is the estimated goal cost of state $s$. It is initially $H(s)$ for all states $s$ and remains unchanged throughout the search. The $f$-value, denoted by $f(s)$, is derived from the $g$- and $h$-values of state $s$ and satisfies the following equality: $f(s) = g(s) + h(s)$.

A* maintains a priority queue, called the OPEN list, where states are ordered with respect to their $f$-values, in increasing order. It initially contains only state $s_{current}$. A* repeatedly removes a state $s$ with the smallest $f$-value from the OPEN list and expands it by performing the following procedure for each successor state $t$ of state $s$ with $g(s) + d(s,t) < g(t)$: $g(t)$ is set to $g(s) + d(s,t)$ and $parent(t)$ is set to state $s$. If state $t$ is not in the OPEN list, then it is inserted into the OPEN list, otherwise, it is reordered in the OPEN list with its updated $f$-value. A* terminates when its OPEN list is empty or it is about to expand state $s_{goal}$.

The properties of A* are explained in more detail in [22]. We use the following properties of A* that uses a consistent heuristic function:

- **Property 1:** Every expanded state $s$ satisfies the following conditions: (a) $g(s) = d(s_{current}, s)$. (b) One can identify a cost-minimal path from state $s_{current}$ to state $s$ in reverse by following the *parent*-pointers from state $s$ to state $s_{current}$.
- **Property 2:** The sequence of $f$-values of the expanded states is monotonically non-decreasing.
- **Property 3:** If a heuristic function $h$ weakly dominates another heuristic function $h'$, an A* search using $h$ expands no more states than an A* search using $h'$ (modulo tie-breaking).
- **Property 4:** An A* search always terminates, either proving that there is no path from state $s_{current}$ to state $s_{goal}$ or returning a cost-minimal path from state $s_{current}$ to state $s_{goal}$ that can be identified in reverse by following the parent pointers from state $s_{goal}$ to state $s_{current}$.
- **Property 5:** Each state is expanded at most once.

## 3 Incremental Heuristic Search

To find a cost-minimal path from state $s_{current}$ to state $s_{goal}$ in unknown terrain, an agent could use repeated forward A* searches (from state $s_{current}$ to state $s_{goal}$) with the freespace assumption or repeated backward A* searches (from state $s_{goal}$ to state $s_{current}$) with the freespace assumption. These repeated A* searches need to be fast since agents typically have to move smoothly and without delay. Incremental heuristic search algorithms use information from previous searches to solve similar future search problems potentially faster than classical heuristic search algorithms, which solve all search problems from scratch. We provide a brief introduction to AA* in the following since both Path-AA* and Tree-AA* build on it. All three incremental heuristic search algorithms apply to path planning with the freespace assumption and use forward A* searches to find cost-minimal paths from state $s_{current}$ to state $s_{goal}$.

AA* [13] performs repeated A* searches and after each search, updates the $h$-values of all states $s$ expanded by the search, by assigning $h(s) := f(s_{goal}) - g(s)$. This results in a more informed heuristic function, therefore speeding up future A* searches by making them more focused. For the updated heuristic function to remain consistent, state $s_{goal}$ has to remain unchanged, although state $s_{current}$ can change. The $h$-values remain consistent even if the arc costs increase (but not decrease), which makes AA* applicable to path planning with the freespace assumption. AA* is based on the following "update principle" that was first described in [11] in the context of hierarchical A* search: If the $h$-values of all states $s$ expanded by an A* search that uses a consistent heuristic function are updated by assigning $h(s) := f(s_{goal}) - g(s)$, then the resulting $h$-values are again consistent and weakly dominate the original $h$-values. Thus, an A* search with the resulting $h$-values finds cost-minimal paths and expands no more states than an identical A* search with the original $h$-values (and likely many fewer states).

The properties of AA* are explained in more detail in [15,8]. We use the following property of AA* in addition to the properties of A* listed earlier:

- **Property 6:** After each update, the $h$-value of any state $s$ on the cost-minimal path from state $s_{current}$ to state $s_{goal}$ is set to its goal cost (since $h(s) = f(s_{goal}) - g(s) = d(s_{current}, s_{goal}) - d(s_{current}, s) = d(s, s_{goal})$).

```
01 procedure InitializeState(s)
02 if (generated(s) = 0)
03    g(s) := ∞;
04    h(s) := H(s);
05 else if (generated(s) ≠ counter)
06    if (g(s) + h(s) < pathcost(generated(s)))
07       h(s) := pathcost(generated(s)) − g(s);
08    g(s) := ∞;
09 generated(s) := counter;

10 procedure ComputePath()
11 while (OPEN ≠ ∅)
12    delete a state s with the smallest f-value g(s) + h(s) from OPEN;
13    if (s = s_goal)
14       pathcost(counter) := g(s) + h(s);
15       return true; /* path from s_current to s_goal found */
16    for all t ∈ Succ(s)
17       InitializeState(t);
18       if (g(t) > g(s) + c(s, t))
19          g(t) := g(s) + c(s, t);
20          parent(t) := s;
21          if (t ∈ OPEN)
22             delete t from OPEN;
23          insert t into OPEN with f-value g(t) + h(t);
24 return false; /* no path from s_current to s_goal found */

25 procedure Main()
26 counter := 1;
27 for all s ∈ S
28    generated(s) := 0;
29 while (s_current ≠ s_goal)
30    InitializeState(s_current);
31    g(s_current) := 0;
32    OPEN := ∅;
33    insert s_current into OPEN with f-value g(s_current) + h(s_current);
34    if (ComputePath() = false)
35       return false; /* failure: s_goal is unreachable */
36    path := generate a path from s_current to s_goal by following the parent-pointers in reverse
   from s_goal to s_current;
37    blocked := false;
38    while (s_current ≠ s_goal and ¬blocked)
39       s_current := next state on path;
40       update all increased arc costs (if any);
41       for all increased arc costs c(s, t)
42          if (path contains (s, t))
43             blocked :=true;
44    counter := counter + 1;
45 return true; /* success: s_goal has been reached */
```

**Fig. 1** Adaptive A* (AA*).

There are two versions of AA*, which differ in how they update the $h$-values: The *eager* version of AA* updates the $h$-values of all expanded states right after a forward A* search ends and before a new forward A* search begins. The *lazy* version of AA* [25], on the other hand, updates the $h$-value of a state only when it is needed during a future A* search [15]. It does this by remembering the expanded states and the cost of the cost-minimal path found in each search, and by using them to update the $h$-value of a state when it becomes necessary. The lazy version is faster in general, despite the additional book-keeping, since it does not update the $h$-values of states that become irrelevant for future searches.

Figure 1 shows the pseudo code of the lazy version of AA*. It maintains the following variables for its forward A* searches: *counter* is the number of the current forward A* search. *pathcost*($i$) is the cost of the path found during the $i$-th forward A* search. *generated*($s$) is the number of the last forward A* search that generated

state $s$. AA* uses these values to initialize the $g$-value of state $s$ to infinity and the $h$-value of state $s$ to $H(s)$ as needed (to avoid having to initialize them unnecessarily) and to update the $h$-value of the state as needed (procedure InitializeState). *OPEN* is the OPEN list of the current forward A* search. *parent*($s$) is the parent of state $s$ in the search tree of the last A* search that generated state $s$. $g(s)$ is the $g$-value of state $s$ at the end of the last forward A* search that generated state $s$. $h(s)$ is the $h$-value of state $s$, which may or may not be up-to-date.

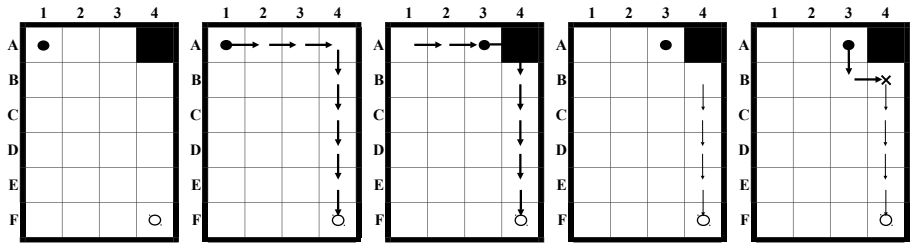
AA* performs a forward A* search from state $s_{current}$ to state $s_{goal}$ (Line 34, procedure ComputePath) until it is about to expand state $s_{goal}$ (Line 13). It then remembers the cost of the path found by the forward A* search to eventually be able to set the $h$-value of every expanded state $s$ to the cost of the path found by the forward A* search (which is the same as the $f$-value of state $s_{goal}$) minus the $g$-value of state $t$ (Line 14). It then generates the cost-minimal path from state $s_{current}$ to state $s_{goal}$ by repeatedly following the *parent*-pointers in reverse from state $s_{goal}$ to state $s_{current}$ (Line 36). The agent then repeatedly moves from state $s_{current}$ to the next state along the cost-minimal path, observing its environment after each movement. If an arc cost increases on the remaining path, the agent performs another forward A* search and then repeats the process until it either reaches state $s_{goal}$ or can no longer find a path from state $s_{current}$ to state $s_{goal}$.

D* Lite is an incremental heuristic search algorithm that solves the same path-planning problems with the freespace assumption for goal-directed navigation in unknown terrain as Path-AA* and Tree-AA*, except that it is able to handle not only arc-cost increases but also arc-cost decreases. It does so by using a version of A* that searches from the goal state to the current state of the agent. Instead of performing such an A* search from scratch every time an arc cost changes, D* Lite basically repeatedly transforms the search tree of the previous A* search to the search tree of the current A* search, which is faster than an A* search from scratch in case the two search trees are similar (but can also be slower in case they are not).

## 4 Path-Adaptive A* (Path-AA*)

AA* performs forward A* searches until it is about to expand state $s_{goal}$. Path-AA* uses AA* unchanged, except that its forward A* searches terminate earlier than the ones of AA*. It extends the path-caching strategy first described in [11] in the context of hierarchical A* search and is based on the following "termination principle": If the $h$-values of all states on a cost-minimal path from some state to state $s_{goal}$ (= reusable path) are the same as their respective goal costs, then each forward A* search of AA* can terminate when it is about to expand a state on the reusable path.

Figure 2(a-e) shows a goal-directed navigation problem in unknown terrain to illustrate the operations of Path-AA*. The terrain is discretized into cells that are either blocked or unblocked, a common practice in the context of real-time computer games [1]. The cells of the grid correspond to the states. A cell is black iff it is blocked and this fact is known to the agent. The agent can move from a cell to each neighboring unblocked cell in the four main compass directions with an arc cost of one. The agent thus operates on an undirected four-neighbor grid. The agent assumes that all cells are unblocked except for the blocked cells that it




(a) Actual environment (b) Search 1 (c) Blocked cell observed (d) Reusable path (e) Search 2

**Fig. 2** Operation of Path-AA* on Four-Neighbor Grids

has already observed. It plans a cost-minimal path from its current cell to the goal cell. It always observes whether the neighboring cells in the four main compass directions are blocked as it follows the planned path, adds newly observed blocked cells to its map, and increases the arc costs from cells to newly observed blocked cells to infinity (making them untraversable).

The agent is initially at cell A1, is trying to reach cell F4, and is unaware that cell A4 is blocked (a). The first forward A* search of Path-AA* finds a path from cell A1 to cell F4 (b), and the $h$-values of the expanded states are updated. The agent follows the path until it observes a blocked cell on the path (c). Path-AA* discards the prefix of the path up to and including the newly observed blocked cell and keeps the rest as the reusable path (d). The next forward A* search of Path-AA* expands only two cells, A3 and B3, and terminates before expanding cell B4 since cell B4 lies on the reusable path (e).

Path-AA* needs to support two operations, namely removing a prefix of the reusable path and adding a prefix to the reusable path.

- **Adding a Prefix to the Reusable Path:** When a forward A* search of Path-AA* terminates because it is about to expand a state $s$ on the reusable path, Path-AA* adds the path from state $s_{current}$ to state $s$ to the reusable path as a prefix, because the (cost-minimal) path from state $s_{current}$ to state $s$ and the (cost-minimal) path from state $s$ to state $s_{goal}$ from the reusable path form a cost-minimal path from state $s_{current}$ to state $s_{goal}$. This is so because AA* finds cost-minimal paths, and the $h$-values of all states on the reusable path are the same as their goal costs since AA* has updated them this way.
- **Removing a Prefix of the Reusable Path:** There are two cases where Path-AA* removes a prefix of the reusable path: (1) Whenever the cost of an arc $(s, t)$ on the path increases (for instance, due to a newly observed blocked cell), Path-AA* removes the prefix of the reusable path up to and including state $s$. It does this because the states contained in the prefix of the path up to state $s$ because the paths from the states in the prefix to state $s_{goal}$ might no longer be cost-minimal. (2) When a forward A* search of Path-AA* terminates because it is about to expand a state $t$ on the reusable path, Path-AA* removes the prefix of the path up to, but not including, state $t$ because it maintains only a single path. Tree-AA* (described later) does not remove the prefix of a path when a new path is found, but has to use additional methods to ensure that

all maintained paths (which form a tree) contain only states that have known cost-minimal paths to state $s_{goal}$.

### 4.1 Implementation of the Reusable Path

Path-AA* implements these operations efficiently by maintaining a pointer *pathstart*, which points to the state that begins the reusable path, and, for each state $s$, a pointer $nextstate(s)$. If state $s$ is in the reusable path and is not equal to state $s_{goal}$, then $nextstate(s)$ points to the state that follows state $s$ on the reusable path. Otherwise, $nextstate(s)$ is NULL.

### 4.2 Implementation of the Operations

Removing a prefix from and adding a prefix to the reusable path is implemented as follows:

- **Removing a Prefix of the Reusable Path:** Whenever the cost of an arc $(s, t)$ on the reusable path increases or the forward A* search of Path-AA* terminates because it is about to expand a state $t$ on the reusable path, Path-AA* removes the prefix of the reusable path up to, but not including, state $t$. It performs this operation by starting at state *pathstart* and following the *nextstate*-pointers until it reaches state $t$, setting $nextstate(s) := NULL$ for each visited state $s$ (not including state $t$) and setting $pathstart := t$.
- **Adding a Prefix to the Reusable Path:** Whenever a forward A* search of Path-AA* terminates because it is about to expand a state $t$ on the reusable path, then Path-AA* adds the path from state $s_{current}$ to state $t$ found by the forward A* search to the reusable path as a prefix by setting $nextstate(s)$ to the next state on the cost-minimal path from state $s_{current}$ to state $t$ for each state $s$ on the path from state $s_{current}$ to state $t$ (not including state $t$) and setting $pathstart := s_{current}$.

### 4.3 Pseudo Code

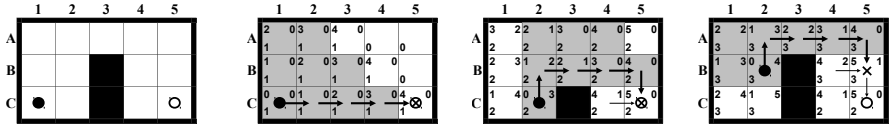
Figure 3 shows the pseudo code of the version of Path-AA* that extends the lazy version of AA*. We use this version of Path-AA* since it ran faster in our experiments than the version of Path-AA* that extends the eager version of AA*.

Path-AA* performs a forward A* search from state $s_{current}$ to state $s_{goal}$ (Line 49, function ComputePath) until it is about to expand state $s_{goal}$ or a state $s$ on the reusable path (Line 25, termination principle). It then remembers the cost of the path found during the forward A* search (Line 26, update principle) to eventually be able to set the $h$-value of every expanded state $t$ to the cost of the path found during the forward A* search minus the $g$-value of state $t$. It then shortens the reusable path to the cost-minimal path from state $s$ to state $s_{goal}$ (Line 27, procedure CleanPath) and adds the cost-minimal path from state $s_{current}$ to state $s$ as a prefix to the reusable path (Line 28, procedure MakePath). The agent then repeatedly moves from state $s_{current}$ to state $nextstate(s_{current})$ along the

```
01 procedure InitializeState(s)
02 if (generated(s) = 0)
03    g(s) := ∞;
04    h(s) := H(s);
05    nextstate(s) := NULL;
06 else if (generated(s) ≠ counter)
07    if (g(s) + h(s) < pathcost(generated(s)))
08       h(s) := pathcost(generated(s)) − g(s);
09    g(s) := ∞;
10 generated(s) := counter;

11 procedure MakePath(s)
12 while (s ≠ s_current)
13    s_aux := s;
14    s := parent(s);
15    nextstate(s) := s_aux;
16 pathstart := s_current;

17 procedure CleanPath(s)
18 while (pathstart ≠ s)
19    s_aux := nextstate(pathstart);
20    nextstate(pathstart) := NULL;
21    pathstart := s_aux;

22 procedure ComputePath()
23 while (OPEN ≠ ∅)
24    delete a state s with the smallest f-value g(s) + h(s) from OPEN;
25    if (s = s_goal or nextstate(s) ≠ NULL)
26       pathcost(counter) := g(s) + h(s);
27       CleanPath(s);
28       MakePath(s);
29       return true; /* path from s_current to s_goal found */
30    for all t ∈ Succ(s)
31       InitializeState(t);
32       if (g(t) > g(s) + c(s, t))
33          g(t) := g(s) + c(s, t);
34          parent(t) := s;
35          if (t ∈ OPEN)
36             delete t from OPEN;
37          insert t into OPEN with f-value g(t) + h(t);
38 return false; /* no path from s_current to s_goal found */

39 procedure Main()
40 counter := 1;
41 pathstart := s_goal;
42 for all s ∈ S
43    generated(s) := 0;
44 while (s_current ≠ s_goal)
45    InitializeState(s_current);
46    g(s_current) := 0;
47    OPEN := ∅;
48    insert s_current into OPEN with f-value g(s_current) + h(s_current);
49    if (ComputePath() = false)
50       return false; /* failure: s_goal is unreachable */
51    while (nextstate(s_current) ≠ NULL)
52       s_current := nextstate(s_current);
53       update all increased arc costs (if any);
54       for all increased arc costs c(s, t)
55          if (nextstate(s) = t)
56             CleanPath(t);
57    counter := counter + 1;
58 return true; /* success: s_goal has been reached */
```

**Fig. 3** Path Adaptive A* (Path-AA*).

cost-minimal path (Line 52). Whenever arc costs increase, it shortens the reusable path (Line 56, procedure CleanPath). If state $s_{current}$ is no longer on the reusable path, the agent performs another forward A* search and then repeats the process



(a) Actual environment (b) Search 1 (c) Search 2 (d) Search 3

**Fig. 4** Example Trace of Path-AA*

until it either reaches state $s_{goal}$ or can no longer find a path from state $s_{current}$ to state $s_{goal}$.

### 4.4 Example Trace

Figure 4(a-d) shows a goal-directed navigation problem in unknown terrain, similar to the one described in Figure 2. The user-provided $H$-values are all zero. The *generated*-value of a cell is shown in its lower left corner. The $h$- and $g$-values of a cell are shown in its upper right and upper left corners, respectively, iff it has been generated by a forward A* search. A cell is gray iff it was expanded by the current forward A* search. The parent of a cell is a thick arrow iff it was added to the reusable path by the current forward A* search and a thin arrow iff it was added by a previous forward A* search. The solid circle marks the current cell of the agent, and the hollow circle marks the goal cell. An 'X' marks the cell on the reusable path that the current forward A* search was about to expand before it terminated.

Figure 4(a) shows the actual environment, and Figure 4(b) shows the initial situation as the agent perceives it. The first forward A* search of Path-AA* from cell C1 to cell C5 expands cells C1, C2, B1, C3, B2, A1, C4, B3 and A2 in this order and terminates when it is about to expand goal cell C5. Figure 4(b) shows the situation after the first forward A* search terminates. It finds the path C1-C2-C3-C4-C5, makes it the reusable path and remembers the cost of the path by setting $pathcost(1) := 4$ to be able to update the $h$-values of expanded states when necessary. The agent then follows the reusable path from cell C1 to cell C2, where it observes that cell C3 is blocked. Path-AA* removes path C1-C2-C3-C4 from the reusable path, leaving path C4-C5 as the new reusable path.

The second forward A* search of Path-AA* from cell C2 to cell C5 expands cells C2, B2, B3, B4, A3, A2, B5, A4 in this order, updating the $h$-values of all generated states, and terminates when it is about to expand goal cell C5. It expands fewer cells than a forward A* search with the user-provided zero $H$-values would (which additionally expands cells C1, B1, A1 and C4), illustrating the speed-up achieved with the update principle. Figure 4(c) shows the situation after the second forward A* search terminates. Path-AA* removes C4-C5 from the reusable path, makes C2-B2-B3-B4-B5-C5 the new reusable path and sets $pathcost(2) := 5$. The agent then moves from cell C2 to cell B2, where it observes that cell B3 is blocked. Path-AA* removes path C2-B2-B3-B4 from the reusable path, leaving path B4-B5-C5 as the new reusable path.

The third forward A* search of Path-AA* from cell B2 to cell C5 expands cells B2, B1, A1, A2, A3, A4 and A5 in this order, updating the $h$-values of all generated states, and terminates when it is about to expand cell B5, which lies

on the reusable path. It terminates earlier than a regular forward A* search with the same $h$-values would (which additionally expands cell B5 and terminates only when it is about to expand goal cell C5), illustrating the speed-up achieved with the early termination principle. Figure 4(d) shows the situation after the third forward A* search terminates. Path-AA* removes path B4-B5 from the reusable path and makes B2-A2-A3-A4-A5-B5-C5 the new reusable path and sets $pathcost(3) := 6$.

### 4.5 Theoretical Results

In this section, we show that any search of Path-AA* finds a cost-minimal path if one exists and returns false otherwise. It should be noted that this does not guarantee that the agent follows a cost-minimal path from its starting position to its goal position in the actual environment. It guarantees only that the agent always follows a cost-minimal path from its current position to its goal position, with respect to the observed arc costs, under the freespace assumption.

Throughout the proofs, we refer to the $i$th execution of lines 45-50 as the $i$th search. We use the superscript $i$ to denote the value of a variable when the $i$th search terminates. We say that a state $s$ is generated during the $i$th search iff $generated^i(s) = counter^i$. Some of the following statements also hold for Tree-AA*, with small modifications. Text inside brackets is used to help translate these statements to Tree-AA* and should be ignored for Path-AA*.

**Definition 1** We recursively define $x^i(s)$ as follows:

- $x^0(s) = H(s)$;
- $x^i(s) = x^{i-1}(s)$ if state $s$ is not generated during the $i$th search; and
- $x^i(s) = max(x^{i-1}(s), pathcost(i) - g^i(s))$ if state $s$ is generated during the $i$th search.

**Definition 2** For any $i \geq 0$, the *heuristic statement* $\mathcal{H}^i$ is defined to hold at a given point during the execution iff, at that point, $x^i$ is a consistent heuristic function, given the arc costs at that point.

**Definition 3** For any $i \geq 0$, the *path statement* $\mathcal{P}^i$ is defined to hold at a given point during the execution iff, at that point,

- the *nextstate*-pointers form a path starting at *pathstart* and ending at state $s_{goal}$, called the reusable path; and
- from any state $s$ with $nextstate(s) \neq NULL$, one can use the *nextstate*-pointers to follow a path from state $s$ to state $s_{goal}$ with cost $x^i(s)$.

**Lemma 1** *For any state $s$ and any $i > 0$, $x^i(s) \geq x^{i-1}(s)$.*

*Proof* If state $s$ is not generated during the $i$th search, $x^i(s) = x^{i-1}(s)$. Otherwise, $x^i(s) = max(x^{i-1}(s), pathcost(i) - g^i(s)) \geq x^{i-1}(s)$. The inequality holds in both cases. □

**Lemma 2** *For any $i > 0$, whenever procedure InitializeState is called for any state $s$ during the $i$th search, the $h$-value of state $s$ is $x^{i-1}(s)$ when the call terminates. When the $i$th search terminates, $h^i(s) = x^{i-1}(s)$ for all states $s$ generated during the search.*

*Proof* When procedure InitializeState is called for state $s$ during the $i$th search, $counter = i$ and $generated(s) \in \{0, \ldots, i\}$. The former property holds because $counter$ is initially set to 1 (Line 40) [Line 57] and incremented after each search (Line 57) [Line 74]. The latter property holds because $generated(s)$ is initially set to 0 (Lines 42-43) [Lines 59-60] and only modified by setting it to $counter$ (Line 10) [Line 11]. If procedure InitializeState is called for a state $s$ more than once during the $i$th search, $generated(s)$ is set to $i$ (Line 10) [Line 11] during the first call and the subsequent calls have no effect. Therefore, the only calls to procedure InitializeState during the $i$th search that might affect $h(s)$ are those where $generated(s) < i$. Also note that a state $s$ is generated during the $i$th search iff procedure InitializeState is called for state $s$ during the $i$th search.

We prove the lemma by strong induction on the number of searches. During the first search, whenever procedure InitializeState is called for a state $s$ for the first time, $generated(s) = 0$ and, therefore, $h(s)$ is set to $H(s) = x^0(s)$ (Lines 2,4) [Lines 2,4] and $generated(s)$ is set to 1 (Line 10) [Line 11]. The subsequent calls to procedure InitializeState for state $s$ do not change $h(s)$ because $generated(s) = 1$ and $counter = 1$ (Lines 2,6) [Lines 2,7]. Therefore, the lemma holds for the base case $i = 1$.

For the induction step, we show that the lemma holds for the $i$th search, assuming that it holds for searches $1, \ldots, i-1$. Assume that procedure InitializeState is called for a state $s$ during the $i$th search and $generated(s) = j < i$. This means that state $s$ has not been generated during searches $j+1, \ldots, i-1$, which implies that $x^j(s) = \cdots = x^{i-1}(s)$ (Definition 1). It also means that it is the first time procedure InitializeState is called for state $s$ during the $i$th search because, otherwise, $generated(s) = i$ (Line 10) [Line 11]. We distinguish two cases: If $j = 0$, then $h(s)$ is set to $H(s) = x^j(s)$. Otherwise, $i > j = generated(s) > 0$, meaning that state $s$ was generated during the $j$th search and this is the first time that state $s$ is being generated since then. In this case, when state $s$ is generated, its $h$-value is set to $\max(h(s), pathcost(j) - g(s))$ (Lines 6-8) [Lines 7-9]. Since the $g$-value of a state cannot be modified during a search without first generating the state (Lines 31,33) [Lines 48,50] and since state $s$ has not been generated since the $j$th search, it must hold that $g(s) = g^j(s)$. Also, according to our strong induction assumption, $h(s) = x^{j-1}$. Therefore, the $h$-value of state $s$ is set to $\max(x^{j-1}(s), pathcost(j) - g^j(s)) = x^j(s) = x^{i-1}(s)$ (Definition 1), proving the induction statement. □

**Lemma 3** *For any $i > 0$, assume that $\mathcal{H}^{i-1}$ and $\mathcal{P}^{i-1}$ hold in the beginning of the $i$th search. Then, if a path does not exist between state $s_{current}$ and state $s_{goal}$, then the $i$th search returns false. Otherwise,*

- *the $i$th search finds a cost-minimal path between state $s_{current}$ and state $s_{goal}$;*
- *the cost of this path is $g^i(s) + h^i(s)$ and it contains state $s$, where state $s$ is the last state considered for expansion on Line 24 [Line 42];*
- *the cost of this path is stored in $pathcost(i)$; and*
- *one can follow this path by following the nextstate-pointers from state $s_{current}$ to state $s_{goal}$.*

*Proof* The $i$th search of Path-AA* is basically an A* search that uses $x^{i-1}$ as its heuristic function (Lemma 2), with one major difference: the $i$th search of Path-AA* terminates already if a state $s$ with $nextstate(s) \neq NULL$ $[h(s) \leq H_{max}(id(s))]$

is about to be expanded (Line 25) [Line 43]. According to the lemma's assumption, $x^{i-1}$ is a consistent heuristic function (the properties of an A* search using a consistent heuristic function are listed in Section 2 as Properties 1-4).

We first assume that $nextstate(s) \neq NULL$ $[h(s) \leq H_{max}(id(s))]$ is never satisfied for any state $s$ considered for expansion on Line 24 [Line 42] during the $i$th search. Then, the $i$th search is simply an A* search that is guaranteed to terminate, either proving that there is no path from state $s_{current}$ to state $s_{goal}$ or finding a cost-minimal path that can be identified by following the *parent*-pointers from state $s_{goal}$ to state $s_{current}$ (Properties 1 and 4). In the former case, we are done. In the latter case, Line 26 [Line 44] sets $pathcost(i) = g^i(s_{goal}) + h^i(s_{goal})$ and Line 28 [Line 45] calls procedure MakePath [AddPath], which follows the *parent*-pointers from state $s_{goal}$ to state $s_{current}$ to identify a cost-minimal path from state $s_{current}$ to state $s_{goal}$ in reverse (Property 4) and adds *nextstate*-pointers so that one can follow this path from state $s_{current}$ to state $s_{goal}$.

We now assume that $nextstate(s) \neq NULL$ $[h(s) \leq H_{max}(id(s))]$ for a state $s$ considered for expansion on Line 24 [Line 42] during the $i$th search. According to the assumption of the lemma, one can use the *nextstate*-pointers to follow a path from state $s$ to state $s_{goal}$ with cost $x^{i-1}(s) = h(s)$. Also, one can use the *parent*-pointers to follow a path from state $s_{current}$ to state $s$ in reverse, with cost $g(s)$ (Property 1). Combining the two statements, we get a path from state $s_{current}$ through state $s$ to state $s_{goal}$, whose cost is $g(s) + x^{i-1}(s) = g(s) + h(s)$. This must be a cost-minimal path because otherwise there would be a lower-cost path and the search would have found it earlier because the sequence of $f$-values of states expanded by an A* search with a consistent heuristic function is monotonically non-decreasing (Property 2). Since the search terminates after the path is found, the $g$- and $h$-values of states are not changed. Therefore, the cost of this cost-minimal path is $g^i(s) + h^i(s) = g(s) + h(s)$ and is stored as $pathcost(i)$ (Line 26) [Line 44]. Procedure CleanPath [RemovePaths] does not change the *nextstate*-pointers of any state $t$ that comes after state $s$ on the reusable path, and procedure MakePath [AddPath] adds *nextstate*-pointers from state $s_{current}$ to state $s$ so that one can follow the *nextstate*-pointers from state $s_{current}$ to state $s_{goal}$. □

**Lemma 4** *For any $i > 0$, assume that $\mathcal{H}^{i-1}$ and $\mathcal{P}^{i-1}$ hold in the beginning of the $i$th search. If the $i$th search finds a path, then the following holds: If a state $s$ is expanded during the $i$th search, then $x^i(s) = pathcost(i) - g^i(s)$. Otherwise, $x^i(s) = x^{i-1}(s)$.*

*Proof* If a state $s$ is generated during the $i$th search, then $x^i(s) = max(x^{i-1}(s), pathcost(i) - g^i(s))$ (Definition 1). The cost of the path found is equal to $pathcost(i) = g^i(t) + h^i(t)$, where state $t$ is the last state considered for expansion on Line 24 [Line 42] (Lemma 3). If state $s$ is expanded during the $i$th search, that is, if state $s$ is chosen for expansion earlier than state $t$, then it must be that $g^i(s) + h^i(s) \leq g^i(t) + h^i(t) = pathcost(i)$ because the sequence of $f$-values of the expanded states is monotonically non-decreasing (Property 2) and a state cannot be expanded twice (Property 5). Replacing $h^i(s)$ with $x^{i-1}(s)$ (Lemma 2), we get $g^i(s) + x^{i-1}(s) \leq pathcost(i)$ and thus $x^{i-1}(s) \leq pathcost(i) - g^i(s)$. Therefore, $x^i(s) = max(x^{i-1}(s), pathcost(i) - g^i(s)) = pathcost(i) - g^i(s)$. On the other hand, if state $s$ is generated but not expanded during the $i$th search, then $g^i(s) + h^i(s) \geq g^i(t) + h^i(t) = pathcost(i)$ because otherwise state $s$ would have been chosen for expansion before state $t$, which implies $x^{i-1}(s) \geq pathcost(i) - g^i(s)$. Therefore,

$x^i(s) = max(x^{i-1}(s), pathcost(i) - g^i(s)) = x^{i-1}(s)$. If state $s$ has not been generated (and therefore not been expanded) during the $i$th search, then $x^i(s) = x^{i-1}(s)$ (Definition 1). □

**Lemma 5** *For any $i > 0$, assume that $\mathcal{H}^{i-1}$ and $\mathcal{P}^{i-1}$ hold in the beginning of the $i$th search. If the $i$th search finds a path, then $\mathcal{H}^i$ holds in the beginning of the $(i+1)$st search.*

*Proof* First, we show that $x^i(s_{goal}) = 0$. According to the lemma's assumption that $\mathcal{H}^{i-1}$ holds, $x^{i-1}(s_{goal}) = 0$. Since considering $s_{goal}$ for expansion is one of the termination conditions of the search (Line 25) [Line 43], $s_{goal}$ is never expanded during a search. Therefore, $x^i(s_{goal}) = x^{i-1}(s_{goal}) = 0$ (Lemma 4).

Second, we show that, if the $i$th search finds a path, then the triangle inequality $x^i(s) \leq d(s,t) + x^i(t)$ holds for any two states $s$ and $t$ such that $(s,t) \in A$, given the arc costs at the end of the $i$th search (which are the same as the arc costs in the beginning of the $i$th search). We distinguish three cases:

- State $s$ is not expanded: Since we assume that $x^{i-1}$ is consistent in the beginning of the $i$th search, $x^{i-1}(s) \leq d(s,t) + x^{i-1}(t)$. According to Lemma 1, $x^{i-1}(t) \leq x^i(t)$ and therefore $x^{i-1}(s) \leq d(s,t) + x^i(t)$. Since state $s$ is not expanded, we can replace $x^{i-1}(s)$ with $x^i(s)$ (Lemma 4) to get $x^i(s) \leq d(s,t) + x^i(t)$.
- State $s$ is expanded before state $t$ is expanded, or state $s$ is expanded but not state $t$: Expanding state $s$ generates state $t$ if it has not been generated during the current search (Lines 30-31) [Lines 47-48]. After state $s$ is expanded, $g(t) \leq g(s) + d(s,t)$ (Lines 30-33) [Lines 47-50]. During a search, the $g$-value of a state can never increase after it is initialized (Lines 32-33) [Lines 49-50] and therefore $g^i(t) \leq g(t) \leq g(s) + d(s,t)$. Also, the $g$-value of an expanded state remains unchanged until the end of the search (Property 1). Thus, $g(s) = g^i(s)$. Furthermore, since state $s$ is expanded, $g^i(s) = pathcost(i) - x^i(s)$ (Lemma 4). Replacing $g(s)$ with $pathcost(i) - x^i(s)$ in the above inequality, we get $g^i(t) \leq pathcost(i) - x^i(s) + d(s,t)$. We distinguish two cases after state $s$ is expanded:
  - If state $t$ is also expanded before the search terminates, then we can replace $g^i(t)$ with $pathcost(i) - x^i(t)$ (Lemma 4) to get $pathcost(i) - x^i(t) \leq pathcost(i) - x^i(s) + d(s,t)$ and consequently $x^i(s) \leq d(s,t) + x^i(t)$.
  - If the search terminates before state $t$ is expanded, then $x^i(t) = x^{i-1}(t)$ (Lemma 4). From Definition 1, by replacing $x^{i-1}(t)$ with $x^i(t)$, we get $x^i(t) = max(x^i(t), pathcost(i) - g^i(t))$. Thus, $pathcost(i) - g^i(t) \leq x^i(t)$ and, consequently, $pathcost(i) - x^i(t) \leq g^i(t)$. Combined with $g^i(t) \leq pathcost(i) - x^i(s) + d(s,t)$, we get $pathcost(i) - x^i(t) \leq pathcost(i) - x^i(s) + d(s,t)$ and consequently $x^i(s) \leq d(s,t) + x^i(t)$.
- State $s$ is expanded after state $t$: Since the search uses a consistent heuristic function, $g^i(t) + h^i(t) \leq g^i(s) + h^i(s)$ (Property 2). Replacing $h^i(s)$ and $h^i(t)$ with $x^{i-1}(s)$ and $x^{i-1}(t)$, respectively, (Lemma 2) and rearranging the terms, we get $g^i(t) - g^i(s) \leq x^{i-1}(s) - x^{i-1}(t)$. Since we assume that $x^{i-1}$ is consistent, $x^{i-1}(s) - x^{i-1}(t) \leq d(s,t)$. Combined with the above inequality, we get $g^i(t) - g^i(s) \leq d(s,t)$. Since both state $s$ and state $t$ are expanded, we can replace $g^i(s)$ with $pathcost(i) - x^i(s)$ and $g^i(t)$ with $pathcost(i) - x^i(t)$ (Lemma 4) and get $pathcost(i) - x^i(t) - pathcost(i) + x^i(s) \leq d(s,t)$ and, consequently, $x^i(s) \leq d(s,t) + x^i(t)$.

We have shown that the triangle inequality holds at the end of the $i$th search. Lines 51-57 [Line 68-75] can only increase but never decrease the arc costs. Therefore, $x^i$ remains consistent and $\mathcal{H}^i$ holds at the beginning of the $(i+1)$st search. □

**Lemma 6** *For any $i > 0$, assume that $\mathcal{H}^{i-1}$ and $\mathcal{P}^{i-1}$ hold in the beginning of the $i$th search. If the $i$th search finds a path, then $\mathcal{P}^i$ holds in the beginning of the $(i+1)$st search.*

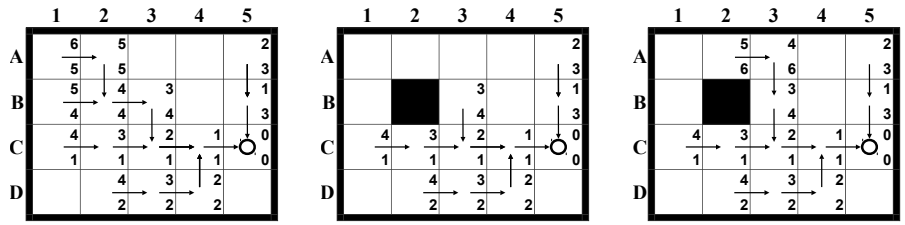
*Proof* Let state $t$ be the state that satisfied the termination condition on Line 25, that is, $t = s_{goal}$ or $nextstate(t) \neq NULL$. The call to procedure CleanPath removes the prefix of the previous path up to $t$ and the call to procedure MakePath adds *nextstate*-pointers so that there is a cost-minimal path from state $s_{current}$ to state $s_{goal}$ with cost $pathcost(i)$ (Lemma 3) and any state $s$ with $nextstate(s) \neq NULL$ is on this path. Procedure MakePath also sets *pathstart* to $s_{current}$ (Line 16). We now show that $\mathcal{P}^i$ holds at the end of the $i$th search by showing that, from any state $s$ with $nextstate(s) \neq NULL$, one can follow a path from state $s$ to state $s_{goal}$ with cost $x^i(s)$ by following the *nextstate*-pointers. We distinguish two cases:

- If state $s$ is expanded during the $i$th search, then $g^i(s)$ is the cost of a cost-minimal path from state $s_{current}$ to state $s$ (Property 1). Since $s$ is on the cost-minimal path from state $s_{current}$ to state $s_{goal}$ with cost $pathcost(i)$, and since the cost of the path from state $s_{current}$ to state $s$ is $g^i(s)$, the cost of the path from state $s$ to state $s_{goal}$ is $pathcost(i) - g^i(s)$. This is equal to $x^i(s)$ (Lemma 4).
- If state $s$ has not been expanded during the $i$th search, then it cannot be the *parent* of any other state in the $i$th search. Therefore, state $s$ cannot precede state $t$ on the path after the $i$th search (since the prefix of the path up to state $t$ is constructed by following the *parent*-pointers from state $t$ to state $s_{current}$). Since we assume that $\mathcal{P}^{i-1}$ holds in the beginning of the $i$th search, we can follow the *nextstate*-pointers from state $s$ to state $s_{goal}$ with cost $x^{i-1}(s)$ in the beginning of the $i$th search. Since the suffix of the path from state $t$ to state $s_{goal}$ and the arc costs remain the same during the $i$th search, we can follow the *nextstate*-pointers from state $s$ to state $s_{goal}$ with cost $x^{i-1}(s)$ at the end of the $i$th search. Since state $s$ has not been expanded during the $i$th search, $x^{i-1}(s) = x^i(s)$ (Lemma 4).

We have shown that $\mathcal{P}^i$ holds at the end of the $i$th search. During the execution of Lines 51-57, some arc costs might increase. If the cost of an arc on the path increases, the prefix of the path up to and including the arc with the increased cost is discarded, by calling procedure CleanPath. Therefore, $\mathcal{P}^i$ holds in the beginning of the $(i+1)$st search. □

**Lemma 7** *For any $i > 0$, $\mathcal{H}^{i-1}$ and $\mathcal{P}^{i-1}$ hold in the beginning of the $i$th search.*

*Proof* The proof is by induction on the number of searches of Path-AA*. In the beginning of the first search, $\mathcal{H}^0$ holds because $x^0(s) = H(s)$ for all states $s$ (Definition 1) and we assume that $H$ is a consistent heuristic function. $\mathcal{P}^0$ also holds because the reusable path is empty (since $nextstate(s) = NULL$ for all states $s$), and $pathstart = s_{goal}$. To prove the inductive step, we assume that $\mathcal{H}^{i-1}$ and $\mathcal{P}^{i-1}$ hold in the beginning of the $i$th search. If no path is found, then Path-AA* terminates. Otherwise, $\mathcal{H}^i$ and $\mathcal{P}^i$ hold in the beginning of the $(i+1)$st search (Lemmata 5 and 6). □




(a)

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

(b)

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

(c)

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

**Fig. 5** Illustration of the Reusable Trees

**Theorem 1** *Any search of Path-AA* finds a cost-minimal path between state* $s_{current}$ *and state* $s_{goal}$ *if one exists and returns false otherwise.*

*Proof* The proof follows from Lemmata 7 and 3. □

## 5 Tree-Adaptive A* (Tree-AA*)

Path-AA* reuses only a suffix of the cost-minimal path of the current forward A* search (= reusable path) to terminate its next forward A* search before it is about to expand the goal state. In complex terrain, the next forward A* search is unlikely to expand a state on that path far away from state $s_{goal}$ and thus unlikely to terminate much earlier than a regular forward A* search. Tree-AA* [10] generalizes Path-AA* by reusing suffixes of the cost-minimal paths of the current and all previous forward A* searches (= reusable tree) to terminate its next forward A* search even earlier. The reusable tree of Tree-AA* is similar to the search tree of D* Lite since D* Lite performs backward A* searches to guarantee that the root of the search tree does not change. The reusable trees of Tree-AA* and D* Lite are thus both rooted in state $s_{goal}$.

Tree-AA* maintains cost-minimal paths from several states to state $s_{goal}$ organized in form of the reusable tree rooted in state $s_{goal}$. If the $h$-values of all states in the reusable tree are the same as their respective goal costs, then each forward A* search of AA* can terminate when it is about to expand a state in the reusable tree, 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 a forward A* search of Tree-AA* terminates because it is about to expand a state $s$ in the reusable tree, then Tree-AA* adds the path from state $s_{current}$ to state $s$ to the reusable tree. It does this because the (cost-minimal) path from state $s_{current}$ to state $s$ and the (cost-minimal) path from state $s$ to state $s_{goal}$ along the branch of the reusable tree form a cost-minimal path from state $s_{current}$ to state $s_{goal}$ (since AA* finds

cost-minimal paths) and the $h$-values of all states on the path are the same as their goal costs (since AA* has updated them this way).
- **Removing Paths from the Reusable Tree:** When arc costs in the reusable tree increase, then Tree-AA* re-uses the largest prefix of the reusable tree that does not contain increased arc 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 cost-minimal paths from some state to state $s_{goal}$ and the $h$-values of all states in the resulting tree are still the same as their goal costs. In particular, when arc cost $c(s,t)$ in the reusable tree increases, then Tree-AA* finds the largest prefix of the reusable tree by removing the subtree rooted in state $s$ from the reusable tree.

### 5.1 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 $p = s_0 \dots s_n$ 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 a forward A* search was about to expand when it terminated. We say that the states $s_0 \dots s_{n-1}$ belong to path $p$. Every path in the reusable tree is identified with a unique integer that corresponds to the number of the forward 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 cost-minimal path from some state to state $s_{goal}$. The $h$-values of all states on the path are the same as 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 same as state $s_{goal}$ or not in the reusable tree.
- $nextstate(s)$ is the parent of state $s$ in the reusable tree if state $s$ is a state different from state $s_{goal}$ in the reusable tree.
- $H_{max}(p)$ is the largest $h$-value of any state on path $p = s_0 \dots s_n$, that is, $H_{max}(p) = h(s_0)$. $H_{max}(0) = -1$, as explained below.
- $H_{min}(p)$ is the smallest $h$-value of any state on path $p = s_0 \dots s_n$, that is, $H_{min}(p) = h(s_n)$.
- $Paths(p)$ is the set of all paths in the reusable tree that connect to one of the states $s_0 \dots s_{n-1}$ that belong to path $p = s_0 \dots s_n$. These paths "feed into" path $p$.

In the worst case, Tree-AA* needs to store a tree that contains the cost-minimal paths from every state in the state space to the goal state. This tree can be embedded into the data structure for the grid world and requires memory that is at most linear in the size of the graph.

Figure 5(a) shows a fictitious example of a reusable tree. Values are shown only for cells that are in the reusable tree. The *id*-value of a cell is shown in its bottom right corner. The $h$-value of a cell is shown in its upper right corner. The parent of a cell is pointed to by an arrow. All arrows thus form the reusable tree, that consists of five paths. Path 1 is path C1-C2-C3-C4-C5, path 2 is path D2-D3-D4-C4, path 3 is path A5-B5-C5, path 4 is path B1-B2-B3-C3, and path 5 is path A1-A2-B2. The table shows the values of the variables of all paths.

### 5.2 Implementation of the Operations

State $s_{goal}$ is always in the reusable tree. Tree-AA* could check whether $id(s) > 0$ when it needs to check whether a state different from state $s_{goal}$ is in the reusable tree. However, this would require $id(s)$ to be set to zero when state $s$ is removed 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 state different from state $s_{goal}$ is in the reusable tree. (State $s_{goal}$ fails this test.) Tree-AA* can now remove a path $p$ from the reusable tree by setting $H_{max}(p)$ to $H_{min}(p)$ without having to set $id(s)$ to zero for all states $s$ that belong to path $p$. Thus, $id(s) = 0$ is not necessarily true for states $s$ not in the reusable tree. There are two subtleties here. States $s$ different from state $s_{goal}$ that have not yet been part of the reusable tree correctly fail the test since $H_{max}(id(s)) = H_{max}(0) = -1$. States $s$ different from state $s_{goal}$ that were part of the reusable 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 $p = s_0 \dots s_n$ to the reusable tree as follows. It equates $p$ with the number of the current forward A* search (as given by the variable *counter*) to identify the path with a unique integer. It inserts $p$ into the set $Paths(id(s_n))$ if state $s_n$ is different from state $s_{goal}$ since path $p$ feeds into the path that state $s_n$ belongs to (Lines 13-14). (The line numbers refer to the pseudo code of Tree-AA* in Figure 6.) It sets $H_{min}(p)$ to $h(s_n)$ (Line 15) and $H_{max}(p)$ to $pathcost(counter)$ (Line 16) since the $h$-values are strictly monotonically decreasing along the path. It sets $Paths(p)$ to the empty set (Line 17) since no paths feed into path $p$ yet. It sets $id(s)$ to $p$ and $nextstate(s)$ to the successor of state $s$ on path $p$ for all states $s_0 \dots s_{n-1}$ (Lines 18-22) since the states $s_0 \dots s_{n-1}$ belong to path $p$. The runtime of adding a path to the reusable tree is thus basically proportional to the number of states on the path. Note that, instead of explicitly setting $H_{min}(p)$ to $h(s_0)$, it sets $H_{min}(p)$ to $pathcost(counter)$ (Line 16). This is because, due to the lazy updates of $h$-values, $h(s_0)$ is not immediately updated and, since $s_0 = s_{current}$ (that is, $g(s_0) = 0$), its updated value is $pathcost(counter)$.
- **Removing Paths from the Reusable Tree:** When arc cost $c(s, t)$ increases, then Tree-AA* removes paths from the reusable tree as follows. If $nextstate(s) = t$ then the arc might be in the reusable tree (Lines 72-73), namely on path $p := id(s)$ (Line 24). In this case, Tree-AA* sets $H_{max}(p)$ to $h(t)$ (if it was larger) to shorten path $p$ (Lines 25-27). Note that, with lazy $h$-value updates, $h(t)$ might not have been updated yet, therefore its current $h$-value needs to be computed (Line 25). Tree-AA* then removes all paths $p' \in Paths(p)$ with $H_{max}(p) < H_{min}(p')$ from the set $Paths(p)$ and schedules them for removal from the reusable tree (Lines 28-32). For each path $p$ scheduled for removal with $H_{max}(p) > H_{min}(p)$, it sets $H_{max}(p)$ to $H_{min}(p)$, removes all paths $p' \in Paths(p)$ from the set $Paths(p)$ and schedules them recursively for removal from the reusable tree (Lines 33-39). The runtime of removing paths from the reusable tree when one arc cost increases is thus basically proportional to the number of paths in the reusable tree, which is bounded by the number of forward A* searches performed so far.

Figure 5(b) continues the fictitious example from Figure 5(a) by showing the reusable tree after Tree-AA* removed paths from the reusable tree after cell B2 became blocked. Tree-AA* shortened path 4 to path B3-C3 and removed path 5. Figure 5(c) shows the reusable tree after Tree-AA* added path A2-A3-B3 to the reusable tree.

### 5.3 Pseudo Code

Figure 6 shows the pseudo code for the version of Tree-AA* that extends the lazy version of AA*. Tree-AA* performs a forward A* search from state $s_{current}$ to state $s_{goal}$ (Line 66, function ComputePath) until it is about to expand state $s_{goal}$ or a state $s$ on the reusable tree (Line 43, termination principle). It then remembers the cost of the path found during the forward A* search to eventually be able to set the $h$-value of every expanded state $t$ to the cost of the path found during the forward A* search minus the $g$-value of state $t$ (Line 44, update principle), and adds the cost-minimal path from state $s_{current}$ to state $s$ to the reusable tree (Line 45, procedure AddPath). The agent then repeatedly moves from state $s_{current}$ to state $nextstate(s_{current})$ along the cost-minimal path (Line 69). Whenever arc costs increase, it removes paths from the reusable tree (Line 73, procedure RemovePaths). If state $s_{current}$ is no longer in the reusable tree, it performs another forward A* search and then repeats the process until the agent either reaches state $s_{goal}$ or can no longer find a path to state $s_{goal}$.

### 5.4 Example Trace

Figure 7(a-e) shows the goal-directed navigation problem in unknown terrain from Figure 4. Cell C1 is the start cell, and cell C5 is the goal cell. The user-provided $H$-values are all zero. The annotation is similar to Figure 4, with the addition of the $id$-values, which are shown in the bottom right corner of a cell iff it has been generated by a forward A* search.

Figure 7(a) shows the actual environment, and Figure 7(b) shows the initial situation as the agent perceives it. Figure 7(c) shows the situation after the first forward A* search of Tree-AA* from cell C1 to cell C5 terminates when it is about to expand cell C5 and returns path C1-C2-C3-C4-C5. Tree-AA* adds the path to the reusable tree and remembers the cost of the path by setting $pathcost(1) := 4$, to be able to update the $h$-values of expanded states when necessary. The agent then follows the branch of the reusable tree from cell C1 to cell C2, where it observes that cell C3 is blocked. Tree-AA* removes path C1-C2-C3-C4 from the reusable tree. Figure 7(d) shows the situation after the second forward A* search of Tree-AA* from cell C2 to cell C5 terminates when it is about to expand cell C5 and returns path C2-B2-B3-B4-B5-C5. It expands fewer cells than a forward A* search with the user-provided zero $H$-values would (which additionally expands cells C1, B1 and A1), illustrating the speed-up achieved with the update principle. Tree-AA* adds the path to the reusable tree and remembers the cost of the path by setting $pathcost(2) := 5$, to be able to update the $h$-values of expanded states when necessary. The agent then moves from cell C2 to cell B2, where it observes that cell B3 is blocked. Tree-AA* removes path C2-B2-B3-B4 from the reusable

tree. Figure 7(e) shows the situation after the third forward A* search of Tree-AA* from cell B2 to cell C5 terminates when it is about to expand cell B5 and returns path B2-A2-A3-A4-A5-B5. It terminates earlier than a regular forward A* search with the same $h$-values would (which additionally expands cell B5 and terminates only when it is about to expand goal cell C5), illustrating the speed-up achieved with the termination principle.

### 5.5 Comparison of Path-Adaptive A* and Tree-Adaptive A*

Figures 4(a-e) and 7(a-e) showed a goal-directed navigation problem in unknown terrain where there was little difference between Path-AA* and Tree-AA*. Figures 8 and 9 show a goal-directed navigation problem in unknown terrain that illustrates their difference. Cell D1 is the start cell and cell A7 is the goal cell. Figures 8(a) and 9(a) show that the first forward A* searches of Path-AA* and Tree-AA* produce the same result. The agent then moves from start cell D1 to cell D2, where it observes that cell D3 is blocked. When sensing the blocked cell, Path-AA* removes path D1-D2-D3-D4 from the reusable path, and Tree-AA* removes the same path from the reusable tree. Figure 8(b) shows that the second forward A* search of Path-AA* terminates when it is about to expand cell C7, and Path-AA* removes path D4-D5-D6-D7-C7 from the reusable path and adds path D2-C2-C3-C4-C5-C6-C7 to the reusable path. Figure 9(b) shows that the second forward A* search of Tree-AA* also terminates when it is about to expand cell C7. Tree-AA* adds path D2-C2-C3-C4-C5-C6-C7 to the reusable tree, but it does not remove path D4-D5-D6-D7-C7 from the reusable tree. Thus, Tree-AA* removes fewer cells, which might allow its future forward A* searches to terminate earlier. The agent then moves from cell D2 to cell C2, where it observes that cell C3 is blocked. Path-AA* and Tree-AA* perform their third forward A* searches. The agent then moves from cell C2 to cell B2, where it observes that cell B3 is blocked. Path-AA* and Tree-AA* perform their fourth forward A* searches. The agent then moves from cell B2 to cell A2, where it observes that cell A3 is blocked. Figure 8(c) shows that the fifth forward A* search of Path-AA* terminates when it is about to expand goal cell A7. Figure 9(c) shows that the fifth forward A* search of Tree-AA* terminates already when it is about to expand cell D7, illustrating the speed-up resulting from reusing the paths from the current and all previous forward A* searches. The agent then moves from cell A2 to cell E4, where it observes that E5 is blocked. Figure 8(d) shows that the sixth forward A* search of Path-AA* terminates when it is about to expand cell D7. Figure 9(d) shows that the sixth forward A* search of Tree-AA* terminates already when it is about to expand cell D4, again illustrating the speed-up resulting from reusing the paths from the current and all previous forward A* searches.

### 5.6 Theoretical Results

The theoretical results for Tree-AA* are similar to the theoretical results for Path-AA*. Definitions 1-2 and Lemmata 1-5 also apply to Tree-AA*, with the following modifications:

- All line numbers that refer to lines in Figure 3 are replaced with the line numbers in brackets, which refer to lines in Figure 6.
- All occurrences of $\mathcal{P}^i$ are replaced with $\mathcal{T}^i$ (defined below).
- All occurrences of Path-AA* are replaced with Tree-AA*.
- The $i$th search refers to the $i$th execution of lines 62-67 in Figure 6.

**Definition 4** $TREE^i$ is the set of states $s$ with $x^i(s) \leq H_{max}(id(s))$ at a given point during execution. All states $s \in TREE^i$ with $id(s) = j$ form the *$j$th branch* of the tree. The *tree statement* $\mathcal{T}^i$ is defined to hold at a given point during execution iff, at that point,

- for all $j$ such that at least one state $s \in TREE^i$ satisfies $id(s) = j$, the *nextstate*-pointers of the states in the $j$th branch of the tree form a path starting from a state $t$ and ending at a state $u$, where $x^i(t) = H_{max}(j)$, $x^i(u) = H_{min}(j)$ and, if $u \neq s_{goal}$, $j \in Paths(id(u))$; and
- from any state $s \in TREE^i$, one can use the *nextstate*-pointers to follow a path from state $s$ to state $s_{goal}$ with cost $x^i(s)$, visiting only states $s \in TREE^i \cup \{s_{goal}\}$.

**Lemma 8** *For any $i > 0$, assume that $\mathcal{H}^{i-1}$ and $\mathcal{T}^{i-1}$ hold in the beginning of the $i$th search. If the $i$th search finds a path, then $\mathcal{T}^i$ holds in the beginning of the $(i+1)$st search.*

*Proof* First, we show that $\mathcal{T}^{i-1}$ holds directly before the condition on Line 43 is satisfied. This is the case because we assume that $\mathcal{T}^{i-1}$ holds in the beginning of the $i$th search and the $i$th search only modifies the $g$- and $h$-values of states before the termination condition on Line 43 is satisfied, which do not affect $\mathcal{T}^{i-1}$. Note that the changes to *nextstate*-pointers and *id*-values on Lines 5 and 6 do not affect $\mathcal{T}^{i-1}$ because they are executed for states that are being generated for the first time, meaning they are not in $TREE^i$, and setting $id(s) = 0$ does not insert state $s$ to the tree because $H_{max}(0) = -1$ (Line 58).

Second, we show that $\mathcal{T}^i$ holds directly before the condition on Line 43 is satisfied, by showing that (1) for any state $s \in TREE^{i-1}$, $x^i(s) = x^{i-1}(s)$; and (2) $TREE^i = TREE^{i-1}$. This is a sufficient argument since $\mathcal{T}^{i-1}$ holds before the condition on Line 43 is satisfied and the only difference between $\mathcal{T}^i$ and $\mathcal{T}^{i-1}$ is that $\mathcal{T}^i$ uses $x^i$-values and $TREE^i$ whereas $\mathcal{T}^{i-1}$ uses $x^{i-1}$-values and $TREE^{i-1}$ (Definition 4). To prove that (1) holds, let $s$ be a state with $x^{i-1}(s) \leq H_{max}(id(s))$. We show that $s$ can never be expanded during the search and, therefore, $x^i(s) = x^{i-1}(s)$ (Lemma 4). For $s$ to be expanded, it has to be in *OPEN* (Line 42 or Line 62) and, therefore, has to be generated first (Line 48). When $s$ is generated, its $h$-value is set to $x^{i-1}(s)$ (Lemma 2) and, thus, $h(s) \leq H_{max}(id(s))$. Throughout the search, before the condition on Line 43 is satisfied, neither $h(s)$, $id(s)$, nor $H_{max}(id(s))$ are modified. Therefore, even if $s$ is chosen for expansion on Line 42, it cannot be expanded because it would satisfy the termination condition on Line 43. To prove that (2) holds, we show that $TREE^{i-1} \setminus TREE^i = \emptyset$, and $TREE^i \setminus TREE^{i-1} = \emptyset$. $TREE^{i-1} \setminus TREE^i = \emptyset$ follows from (1) since it implies that, if $x^{i-1}(s) \leq H_{max}(id(s))$ (and thus $s \in TREE^{i-1}$), then $x^i(s) \leq H_{max}(id(s))$ (and thus $s \in TREE^i$). $TREE^i \setminus TREE^{i-1} = \emptyset$ follows from $x^i(s) \geq x^{i-1}(s)$, for any state $s$ (Lemma 1) since it implies that, if $x^i(s) \leq H_{max}(id(s))$ (and thus $s \in TREE^i$), then $x^{i-1}(s) \leq H_{max}(id(s))$ (and thus $s \in TREE^{i-1}$).

Third, we show that $\mathcal{T}^i$ holds directly after the $i$th search, by showing that procedure AddPath preserves $\mathcal{T}^i$. Let state $s$ be the state that satisfies the condition

on Line 43, that is, $s = s_{goal}$ or $h^i(s) \leq H_{max}(id(s))$. Procedure AddPath follows the *parent*-pointers from $s$ to $s_{current}$ and sets $id(t) = i$ for all states $t$ visited, not including $s$. Let $B$ denote the set of these states. Any state $t \in B$ must have been expanded by the $i$th search because, otherwise, they cannot be reached by following the *parent*-pointers. This means that they cannot be in $TREE^i$ before the condition on Line 43 is satisfied because, otherwise, the $i$th search would stop before expanding them. Therefore, procedure AddPath preserves $\mathcal{T}^i$ for any of the existing states in $TREE^i$. When procedure AddPath sets $H_{max}(i) = pathcost(i)$ (Line 16), all states in $B$ are added to $TREE^i$. This is because, for any state $t \in B$, it holds that $x^i(t) = pathcost(i) - g^i(t)$ since they have been expanded during the $i$th search (Lemma 4). Consequently, $x^i(t) = pathcost(i) - g^i(t) \leq pathcost(i) = H_{max}(i)$. We show that all states $t \in B$ satisfy $\mathcal{T}^i$ to show that $\mathcal{T}^i$ holds after the $i$th search. Procedure AddPath adds *nextstate*-pointers for every state $t \in B$, forming a path from state $s_{current}$ to state $s$, so that one can use these *nextstate*-pointers to follow a path from any state $t \in B$ to state $s$ and, eventually, to state $s_{goal}$ (since $\mathcal{T}^i$ holds for $s$). Since all the states in $B$ are added to the tree, one visits only states in $TREE^i \cup \{s_{goal}\}$ when using *nextstate*-pointers to follow a path from any state in $B$ to state $s_{goal}$. For any $t \in B$, $g^i(t)$ is the cost of a cost-minimal path from state $s_{current}$ to state $t$, since $t$ has been expanded during the $i$th search (Property 1). Then, the cost of the remaining path from state $t$ to state $s_{goal}$ is $pathcost(i) - g^i(s)$, which is equal to $x^i(s)$, as shown above. Procedure AddPath also does the following, to preserve $\mathcal{T}^i$. It adds $i$ into $Paths(id(s))$ if $s \neq s_{goal}$ (Lines 13-14) and it sets $H_{min}(i) = h(s)$ (Line 15), which is equal to $x^{i-1}(s)$ (Lemma 2), which, in turn, is equal to $x^i(s)$, since the $i$th search terminates before expanding state $s$ (Lemma 4).

Fourth, we show that $\mathcal{T}^i$ holds in the beginning of the $(i+1)$st search, that is, Lines 68-74 preserve $\mathcal{T}^i$. While the agent is moving, it might observe increased arc costs (Line 70), which might violate $\mathcal{T}^i$ because for some states $t \in TREE^i$, one will no longer be able to use the *nextstate*-pointers to follow a path from state $t$ to state $s_{goal}$ with cost $x^i(s)$, due to the increased arc cost. We now show that calling procedure RemovePath removes these states from the tree to preserve $\mathcal{T}^i$.

Let $(s, t)$ be the arc with the increased cost and let $u \in TREE^i$ be a state that uses this arc when one is using the *nextstate*-pointers to follow a path from state $u$ to state $s_{goal}$ (therefore, $u \neq t$). Let $\pi = (v_0 = s, v_1, \ldots, v_{n-1} = s, v_n = t)$ be the prefix of this path up to state $t$. Since $\mathcal{T}^i$ holds before the cost of $(s, t)$ increases, the following observations hold:

- **Observation 1:** The *id*-values along $\pi$ are contiguous, that is, if $id(v_j) = id(v_k)$, then $id(v_m) = id(v_j)$ for all $m = j, \ldots, k$. This is because all states sharing the same *id*-value form a path.
- **Observation 2:** $x^i(v_j) > x^i(v_{j+1})$. This is because the arc-costs are always positive and $x^i(v_j)$ and $x^i(v_{j+1})$ are cost of the paths from state $v_j$ an state $v_{j+1}$, respectively, to state $s_{goal}$.

Let the ordered sequence $I = (id_0 = id(u), \ldots, id_m = id(s))$ be the sequence of *id*-values of states as they appear on $\pi$, not including state $t$, where contiguous segments with the same *id*-value are collapsed into a single *id*-value. Since $\mathcal{T}^i$ holds before the cost of $(s, t)$ increases, the following observations hold:

- **Observation 3:** $id_j \in Paths(id_{j+1})$. If $s_{goal}$ appears on $\pi$, it has to appear at the end, as state $t$. Since $t$ is excluded from $I$, the second statement of $\mathcal{T}^i$ applies.

– **Observation 4:** $H_{max}(id_j) > H_{min}(id_j)$. For $id_j$ to be in $I$, there must be a state in $\pi$ with $id$-value $id_j$ and, therefore, the second statement of $\mathcal{T}^i$ applies.

We show that after procedure RemovePaths terminates, $u \notin TREE^i$ and prove the statement of the lemma. Procedure RemovePaths sets $H_{max}(id(s))$ to $x^i(t)$ (Lines 24-27). This is because Line 25 uses the same formula as Lines 8-9, which would set $h^{i+1}(t)$ to this value if state $t$ is generated during the $(i+1)$st search. Therefore, Line 25 sets $H_{max}(id(s))$ to $h^{i+1}(t)$ and, consequently, to $x^i(t)$, since $h^{i+1}(t) = x^i(t)$ (Lemma 2). We distinguish two cases. If $id(u) = id(s)$ or $id(u) = id(t)$ (which implies $id(u) = id(s)$, due to Observation 1), then, $x^i(u) > H_{max}(id(s)) = x^i(t)$ (Observation 2) and, therefore $u \notin TREE^i$. For the second case, where $id(u) \neq id(s)$, we first show that Lines 29-32 add $id_{m-1}$ to $QUEUE$. Since $id_{m-1} \in Paths(id_m)$ (Observation 3), we only need to show $H_{max}(v_m) < H_{min}(v_{m-1})$. As shown above, $H_{max}(v_m) = H_{max}(id(s)) = x^i(t)$, which is strictly smaller than $x^i(s)$ (Observation 2). Due to $I$'s construction, it holds that, $H_{min}(v_{m-1}) \geq x^i(s)$ and, consequently, $H_{max}(v_m) < H_{min}(v_{m-1})$ (Definition 4). Therefore, Lines 29-32 add $id_{m-1}$ to $QUEUE$. When an $id_j$ is added to $QUEUE$, it is eventually selected on Line 34, since procedure RemovePaths does not terminate until $QUEUE$ is empty (Line 33). When $id_j$ is selected on Line 34, $id_{j-1}$ is added to $QUEUE$ (Observations 3 and 4). Since $id_{m-1}$ is initially added to $QUEUE$, we can recursively apply this argument to conclude that $id_0 = id(u)$ is eventually selected on Line 34. Line 36 sets $H_{max}(id(u))$ to $H_{min}(id(u))$, thereby removing $u$ from $TREE^i$, since $x^i(u) > H_{min}(id(u))$ (Definition 4). □

**Lemma 9** *For any $i > 0$, $\mathcal{H}^{i-1}$ and $\mathcal{T}^{i-1}$ hold in the beginning of the $i$th search.*

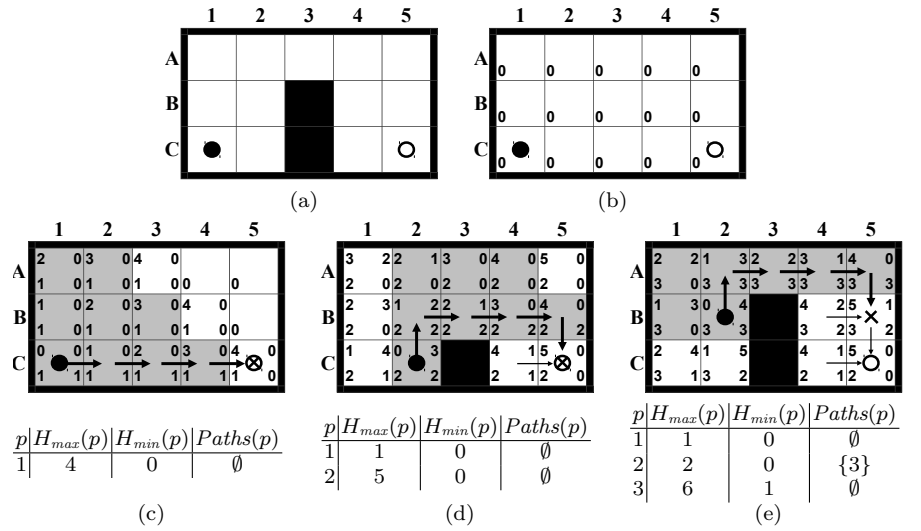
*Proof* The proof is by induction on the number of searches of Tree-AA*. In the beginning of the first search, $\mathcal{H}^0$ holds because $x^0(s) = H(s)$ for all states $s$ (Definition 1) and we assume that $H$ is a consistent heuristic function. $\mathcal{T}^0$ also holds because $TREE^0 = \emptyset$ (since the $id$-values are uninitialized). To prove the inductive step, we assume that $\mathcal{H}^{i-1}$ and $\mathcal{T}^{i-1}$ hold in th beginning of the $i$th search. If no path is found, then Tree-AA* terminates. Otherwise, $\mathcal{H}^i$ and $\mathcal{T}^i$ hold in the beginning of the $(i+1)$st search (Lemmata 5 and 8). □

**Theorem 2** *Any search of Tree-AA* finds a cost-minimal path between state $s_{current}$ and state $s_{goal}$ if one exists and returns false otherwise.*

*Proof* The proof follows from Lemmata 9 and 3. □

```
01 procedure InitializeState(s)
02 if (generated(s) = 0)
03    g(s) := ∞;
04    h(s) := H(s);
05    nextstate(s) := NULL;
06    id(s) := 0;
07 else if (generated(s) ≠ counter)
08    if (g(s) + h(s) < pathcost(generated(s)))
09       h(s) := pathcost(generated(s)) − g(s);
10    g(s) := ∞;
11 generated(s) := counter;

12 procedure AddPath(s)
13 if (s ≠ s_goal)
14    insert counter into Paths(id(s));
15 H_min(counter) := h(s);
16 H_max(counter) := pathcost(counter);
17 Paths(counter) := ∅;
18 while (s ≠ s_current)
19    s_aux := s;
20    s := parent(s);
21    id(s) := counter;
22    nextstate(s) := s_aux;

23 procedure RemovePaths(s)
24 p := id(s);
25 h_aux := max(h(nextstate(s)), pathcost(generated(nextstate(s))) − g(nextstate(s)));
26 if (H_max(p) > h_aux)
27    H_max(p) := h_aux;
28 QUEUE := ∅;
29 for all p' ∈ Paths(p)
30    if (H_max(p) < H_min(p'))
31       add p' to the end of QUEUE;
32       delete p' from Paths(p);
33 while (QUEUE ≠ ∅)
34    delete p from the head of QUEUE;
35    if (H_max(p) > H_min(p))
36       H_max(p) := H_min(p);
37       for all p' ∈ Paths(p)
38          add p' to the end of QUEUE;
39          delete p' from Paths(p);

40 function ComputePath()
41 while (OPEN ≠ ∅)
42    delete state s with the smallest f-value g(s) + h(s) value from OPEN;
43    if (s = s_goal or h(s) ≤ H_max(id(s))) /* s is in reusable tree */
44       pathcost(counter) := g(s) + h(s);
45       AddPath(s);
46       return true; /* path from s_current to s_goal found */
47    for all t ∈ Succ(s)
48       InitializeState(t);
49       if (g(t) > g(s) + c(s, t))
50          g(t) := g(s) + c(s, t);
51          parent(t) := s;
52          if (t ∈ OPEN)
53             delete t from OPEN;
54          insert t into OPEN with f-value g(t) + h(t);
55 return false; /* no path from s_current to s_goal found */

56 function Main()
57 counter := 1;
58 H_max(0) := −1;
59 for all s ∈ S
60    generated(s) := 0;
61 while (s_current ≠ s_goal)
62    InitializeState(s_current);
63    g(s_current) := 0;
64    OPEN := ∅;
65    insert s_current into OPEN with f-value g(s_current) + h(s_current);
66    if (ComputePath() = false)
67       return false; /* failure: s_goal is unreachable */
68    while (pathcost(generated(s_current)) − g(s_current) ≤ H_max(id(s_current))) /* s_current is different from s_goal and in reusable
   tree */
69       s_current := nextstate(s_current);
70       update all increased arc costs (if any);
71       for all increased arc costs c(s, t)
72          if (nextstate(s) = t)
73             RemovePaths(s);
74    counter := counter + 1;
75 return true; /* success: s_goal has been reached */
```

**Fig. 6** Tree Adaptive A* (Tree-AA*).




| $p$ | $H_{max}(p)$ | $H_{min}(p)$ | $Paths(p)$ |
|---|---|---|---|
| 1 | 4 | 0 | $\emptyset$ |

| $p$ | $H_{max}(p)$ | $H_{min}(p)$ | $Paths(p)$ |
|---|---|---|---|
| 1 | 1 | 0 | $\emptyset$ |
| 2 | 5 | 0 | $\emptyset$ |

| $p$ | $H_{max}(p)$ | $H_{min}(p)$ | $Paths(p)$ |
|---|---|---|---|
| 1 | 1 | 0 | $\emptyset$ |
| 2 | 2 | 0 | $\{3\}$ |
| 3 | 6 | 1 | $\emptyset$ |

**Fig. 7** Example Trace of Tree-AA*

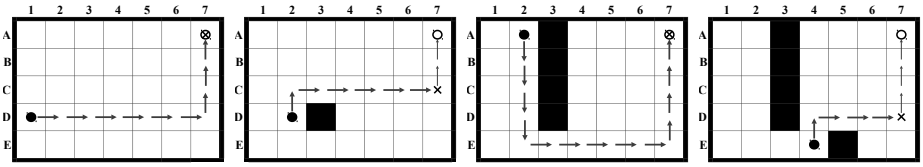



**Fig. 8** Comparison of Path-AA* and Tree-AA* - Path-AA*

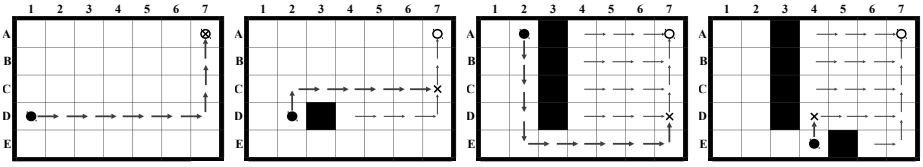



**Fig. 9** Comparison of Path-AA* and Tree-AA* - Tree-AA*

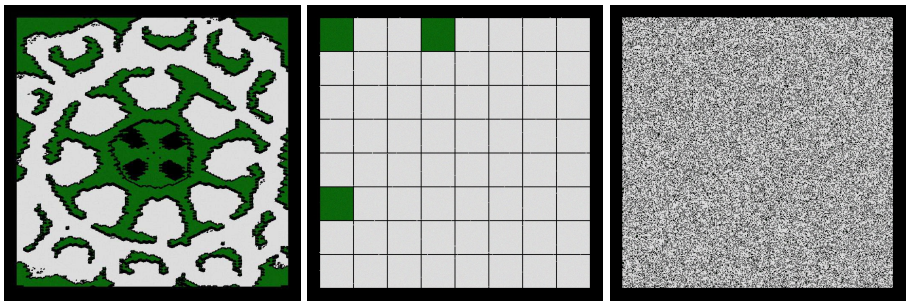
## 6 Experimental Evaluation

We compare Path-AA* and Tree-AA* to A*, AA*, and D* Lite in unknown terrain and partially known terrain. Our implementation of A* starts a new A* search only when arc costs on the path from $s_{current}$ to $s_{goal}$ have increased, rather than whenever arc costs have increased, different from [14], which explains the difference in experimental results compared to [14]. 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 the following exceptions: D* Lite breaks ties towards smaller $g$-values (because this strategy typically runs faster than a version of D* Lite that breaks ties in the opposite direction [7]). All experiments were run on a Linux PC with a Pentium CoreQuad 2.33 GHz CPU and 8 GB RAM.

We use game maps, office maps, and maps with randomly blocked cells for unknown terrain. Motivated by the benchmarks used in [27], we use populated game maps and populated office maps for partially known terrain. We evaluate the algorithms on both 4-neighbor and 8-neighbor versions of these maps, for a total of ten different settings. For 4-neighbor maps, the agent always observes the blockage status of its four neighboring cells and can then move to any one of the unblocked neighboring cells with cost one. The user-given $h$-values are the Manhattan distances. For 8-neighbor maps, the agent always observes 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. Three sample maps with size $512 \times 512$ are shown in Figure 10. The details of the maps are as follows:

- **Game Maps:** We use 342 game maps with sizes varying from $22 \times 28$ to $1260 \times 1104$.[1] For each map, we generate 300 solvable problem instances, for a total of 102,600 problem instances.
- **Office Maps:** We use 40 office maps of size $512 \times 512$, with varying room sizes (namely, 10 maps each with rooms of sizes $8 \times 8$, $16 \times 16$, $32 \times 32$, $64 \times 64$). In each map 80% of the doors between adjacent rooms are opened. For each map, we generate 300 solvable problem instances, for a total of 12,000 problem instances.
- **Maps with Randomly Blocked Cells:** We use 12,000 maps of size $1024 \times 1024$ with randomly placed blocked cells. We generate 3000 maps each with 10%, 20%, 30%, and 40% blocked cells placed randomly. For each map, we generate a single solvable problem instance, for a total of 12,000 problem instances.
- **Populated Game Maps:** We use the 342 game maps with 12% of blocked cells placed randomly in the unblocked cells of the maps. The agent knows a priori the blockage status of the cells in the game map except for the randomly blocked cells. For each map, we generate 300 solvable problem instances, for a total of 102,600 problem instances.
- **Populated Office Maps:** We use the 40 office maps with 12% of blocked cells placed randomly in the unblocked cells of the maps. The agent knows a priori the blockage status of the cells in the game map except for the randomly

[1] All game and office maps can be found in Nathan Sturtevant's repository [24].



(a) Game Map (b) Office Map (c) Map with Randomly Blocked Cells

**Fig. 10** Maps

blocked cells. For each map, we generate 300 solvable problem instances, for a total of 12,000 problem instances.

## 6.1 Experimental Evaluation of Heuristic Search Algorithms

Typically, results in the literature are averaged over all problem instances (for example, [14] [9] [10]). However, the relative performance of different heuristic search algorithms depends on the problem difficulty [7]. Therefore, for each setting we divide the problem instances into ten problem bins, which contain problem instances of different difficulty. We measure the difficulty of a problem instance by the time it takes for A* to solve it. Each bin contains the same number of problem instances. The hardest problem instance in bin $n$ is easier than the easiest problem instance in bin $n + 1$.

### *6.1.1 Results for unknown terrain*

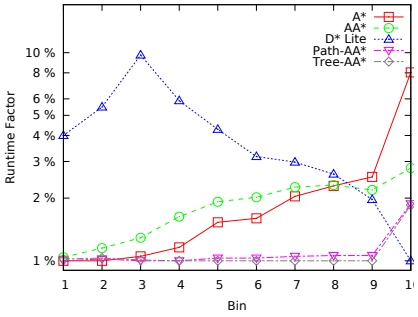
Our results are shown in Figures 11-13, where we compare the performance of the heuristic search algorithms in game maps, office maps and maps with randomly blocked cells, respectively. Each figure has six subfigures: (a) and (b) show the average performance of A* over each bin on 4-neighbor and 8-neighbor versions of the maps, respectively. (c) and (d) show the *runtime factor* of the algorithms on 4-neighbor and 8-neighbor versions of the maps, respectively, where the runtime factor is calculated as follows: For each bin, we pick the lowest average runtime among all heuristic search algorithms and treat it as the baseline runtime of that bin. Then, the runtime factor of a heuristic search algorithm for that bin is simply the ratio of its average runtime and the baseline runtime of that bin. Therefore, smaller runtime factors imply better performance, and a runtime factor of 1 means that the heuristic search algorithm is the best performing one for that bin. (e) and (f) show, for each bin and each heuristic search algorithm, the percentage of instances that were solved faster than A*. We also provide Tables 1-6 in the appendix, to complement the information shown in the figures. They report the average runtime, solution cost, number of expansions and number of heap percolations for each heuristic search algorithm over each bin in each setting.

| Bin | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| **A* RunTime** | 0.01 | 0.03 | 0.08 | 0.29 | 0.94 | 2.51 | 6.02 | 14.34 | 42.25 | 662.20 |
| **A* Cost** | 17.9 | 49.6 | 110.9 | 158.7 | 240.9 | 367.1 | 548.7 | 846.1 | 1,414.8 | 2,882.6 |
| **A* Searches** | 1.5 | 3.4 | 8.5 | 27.7 | 61.1 | 113.4 | 190.8 | 310.2 | 528.9 | 1,124.9 |

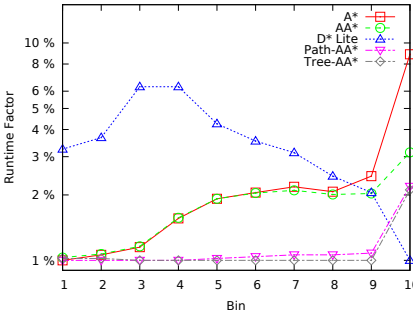
(a) Bin information, 4-neighbor

| Bin | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| **A* RunTime** | 0.01 | 0.03 | 0.08 | 0.24 | 0.78 | 2.06 | 4.93 | 12.04 | 36.21 | 724.30 |
| **A* Cost** | 14.4 | 39.9 | 94.9 | 138.5 | 188.5 | 285.5 | 425.5 | 657.9 | 1,103.7 | 2,330.1 |
| **A* Searches** | 1.4 | 2.7 | 5.9 | 20.3 | 49.8 | 93.5 | 159.9 | 269.0 | 468.1 | 1,042.1 |

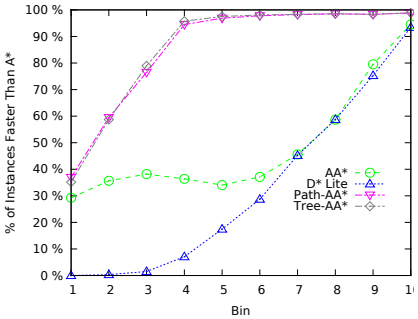
(b) Bin information, 8-neighbor



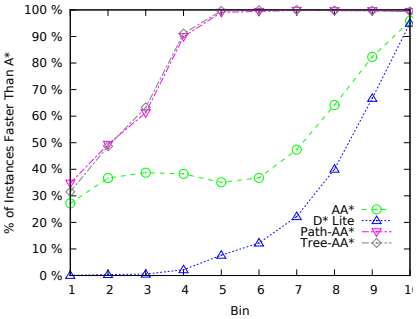

(c) Runtime factor, 4-neighbor




(d) Runtime factor, 8-neighbor




(e) Percentage of instances faster than A*, 4-neighbor




(f) Percentage of instances faster than A*, 8-neighbor

**Fig. 11** Results on Game Maps

In the experimental results, we notice the following relationships:

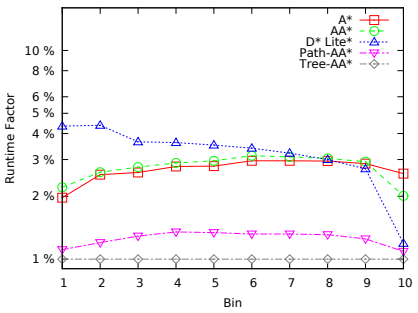
- **Tree-AA* vs. rest:** The runtime factors show that Tree-AA* is the fastest algorithm on average in the three types of maps. Tree-AA* is slower than A* for a larger number of problem instances only for easy problem instances (bins 1-4) on game maps, otherwise it is faster than A* for more than 90 percent of problem instances. Tree-AA* can be slower then D* Lite only for very difficult problem instances (bin 10).
- **Path-AA* vs. rest:** The runtime factors show that Path-AA* is the second-fastest algorithm on average in the three types of maps. It is almost as fast as Tree-AA* on game maps but slower on the other maps. The percentage of

| Bin | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| **A* RunTime** | 0.60 | 1.91 | 3.49 | 5.60 | 8.07 | 10.94 | 14.67 | 19.72 | 28.32 | 79.02 |
| **A* Cost** | 179.2 | 335.8 | 457.7 | 575.3 | 684.5 | 793.8 | 900.8 | 1,034.4 | 1,207.1 | 1,622.6 |
| **A* Searches** | 56.3 | 119.8 | 170.6 | 216.8 | 261.6 | 309.2 | 357.5 | 420.6 | 499.8 | 706.6 |

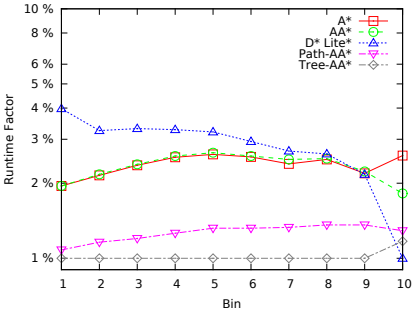
(a) Bin information, 4-neighbor

| Bin | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| **A* RunTime** | 0.64 | 2.00 | 3.63 | 5.72 | 8.09 | 10.85 | 14.46 | 19.23 | 27.50 | 110.86 |
| **A* Cost** | 151.8 | 290.9 | 395.6 | 502.1 | 592.2 | 681.2 | 789.2 | 909.5 | 1,062.6 | 1,489.8 |
| **A* Searches** | 52.6 | 115.9 | 162.0 | 210.4 | 252.8 | 300.0 | 352.6 | 412.7 | 494.2 | 717.8 |

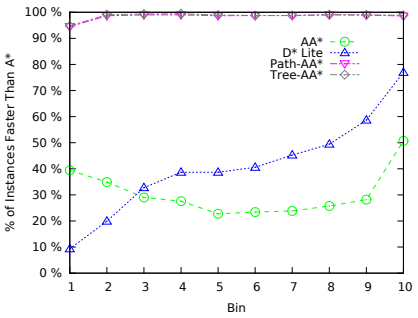
(b) Bin information, 8-neighbor



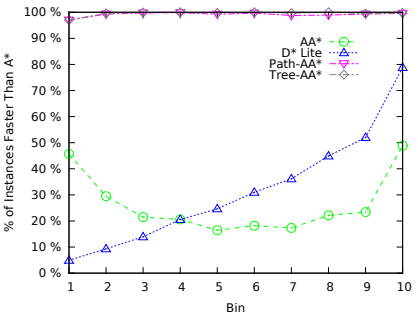

(c) Runtime factor, 4-neighbor




(d) Runtime factor, 8-neighbor




(e) Percentage of instances faster than A*, 4-neighbor




(f) Percentage of instances faster than A*, 8-neighbor

**Fig. 12** Results on Office Maps

problem instances where Path-AA* is slower than A* is similar to the one for Tree-AA*.

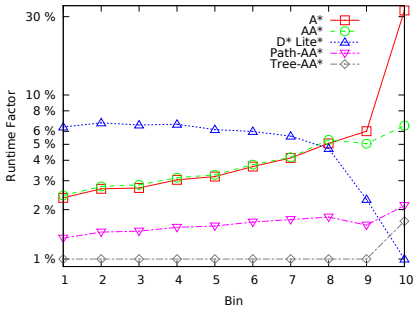
– **D* Lite vs. rest:** D* Lite performs well only on difficult problem instances: Its runtime factor decreases and the number of problem instances for which it is faster than A* increases as the difficulty of the problem instances increases (in some cases with the exception of very easy problem instances). Previously published evaluations of incremental heuristic search algorithms often average over problem instances of different difficulties. Our evaluation shows that D* Lite is often slower than alternative incremental heuristic search algorithms despite its small average runtime. For instance, D* Lite has the smallest average

| Bin | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| **A* RunTime** | 0.76 | 2.64 | 5.16 | 8.65 | 13.89 | 22.03 | 37.02 | 76.97 | 326.63 | 10,249.31 |
| **A* Cost** | 250.2 | 491.2 | 680.4 | 859.0 | 1,056.3 | 1,312.2 | 1,709.2 | 2,797.5 | 9,133.0 | 25,096.3 |
| **A* Searches** | 37.5 | 77.5 | 113.3 | 152.0 | 203.7 | 274.9 | 391.7 | 712.1 | 2,421.7 | 6,509.4 |

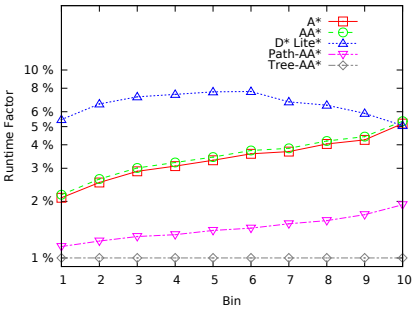
(a) Bin information, 4-neighbor

| Bin | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| **A* RunTime** | 0.41 | 1.29 | 2.37 | 3.70 | 5.39 | 7.53 | 10.52 | 14.70 | 21.91 | 42.81 |
| **A* Cost** | 159.7 | 305.6 | 411.2 | 504.2 | 594.2 | 680.2 | 756.0 | 840.2 | 964.1 | 1,178.2 |
| **A* Searches** | 22.6 | 44.8 | 63.1 | 80.1 | 100.3 | 120.9 | 149.5 | 185.6 | 235.6 | 332.6 |

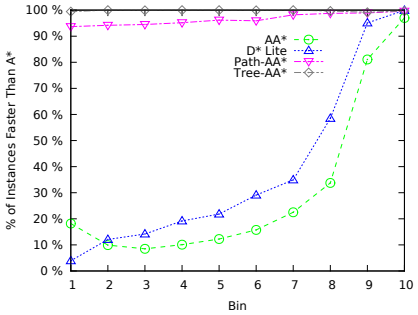
(b) Bin information, 8-neighbor



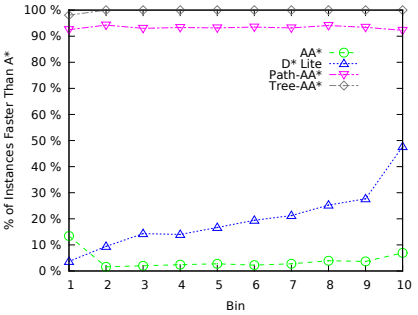

(c) Runtime factor, 4-neighbor




(d) Runtime factor, 8-neighbor




(e) Percentage of instances faster than A*, 4-neighbor




(f) Percentage of instances faster than A*, 8-neighbor

**Fig. 13** Results on Maps with Randomly Blocked Cells

runtime in Tables 1 and 2 in the appendix (see the Total column), yet it is the fastest algorithm only in bin 10.

To see why this happens we can use the same argument presented in a previous publication in which Forward A* is compared with D* Lite [7]. That argument applies here too because Tree-AA* is based on (Forward) A*. D* Lite is based on Backward A* and has the advantage over Backward A* that it typically expands fewer cells than Backward A* after the first search since it reuses information from previous searches. However, D* Lite, like Backward A*, typically expands more cells during the first search than Forward A* during the first search. D* Lite also expands cells more slowly than Forward or Backward A* due to the update of the rhs-values. This means that the first search of D*

Lite typically runs more slowly than the first search of Forward A*, an effect that becomes more pronounced the further apart the start and goal cells are. If the number of subsequent searches needed for the agent to reach the goal cell is not large then typically D* Lite runs more slowly than Forward A*. Since Tree-AA* runs even faster than Forward A* for goal-direct navigation in partially known terrain, the advantage of Forward A*over Tree-AA* can even be larger than the advantage of Forward A* over D* Lite reported in [7].

- **A* vs. AA*:** The runtime factors show that A* and AA* are often equally fast on average. A* can be slightly faster than AA* on easy problem instances and is noticeably slower than AA* only on very difficult problem instances. Even though A* is only slightly faster than AA* on easy problem instances, it is faster for a larger number of problem instances in this case on game and office maps.

#### *6.1.2 Results for partially known terrain*

Our results are shown in Figures 14–15, where we compare the performance of the algorithms in populated game maps and populated office maps, respectively. Each figure has six subfigures. The content of the subfigures was described in the previous section. We also provide Tables 7–9 in the appendix, complement the information shown in the figures. They report the average runtime, solution cost, number of expansions and number of heap percolations for each algorithm over each bin in each setting.

The experimental results are similar to those for unknown terrain, except that Path-AA* is now slightly faster on average than Tree-AA* for very easy problem instances and AA* is now faster than A*.

### 6.2 Experimental Evaluations of Priority Queue Implementations

Priority queues are commonly implemented with binary heaps or buckets [4]. The amount of memory needed to store them depends in both cases on the number of elements in the priority queue. However, in case of buckets, it also depends on the number of possible priorities. This is because a bucket implementation reserves at least one pointer for each priority, namely to the bucket containing the elements with that priority. Thus, it is only reasonable to implement priority queues with buckets when the number of possible priorities is not large, even though buckets tend to result in much smaller runtimes than binary heaps [20].

We compare A*, D* Lite and Tree-AA* with priority queues implemented with either binary heaps or buckets. For the implementation of D* Lite with buckets, we use a single key, as proposed in [20]. The key, which in the original D* Lite implementation has two components, can be implemented with integers whose size are at most about twice the size of $f$-values in A*. For the implementation of A* and Tree-AA* with buckets, we use the $f$-values as the key.

We use 8-neighbor grid maps. The user-given $h$-values are the Octile distances. The details of the maps are as follows:

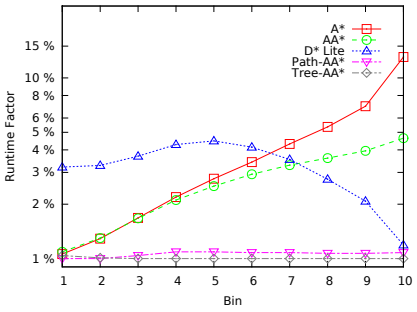
- **Maps with Randomly Blocked Cells:** We use 600 maps of size $512 \times 512$ with randomly placed blocked cells. We generate 150 maps each with 10%,

| Bin | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| **A* RunTime** | 0.01 | 0.04 | 0.13 | 0.39 | 1.10 | 2.91 | 7.66 | 21.07 | 68.61 | 671.09 |
| **A* Cost** | 17.7 | 47.1 | 93.0 | 161.8 | 252.5 | 374.5 | 541.1 | 797.9 | 1,257.9 | 2,601.4 |
| **A* Searches** | 2.5 | 6.2 | 11.7 | 20.0 | 31.3 | 46.5 | 67.5 | 99.3 | 156.2 | 321.0 |

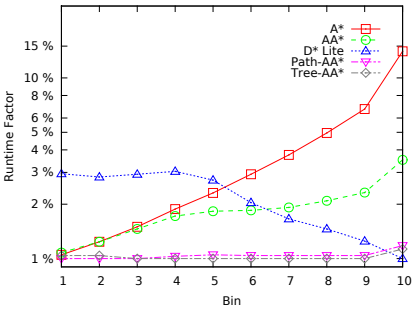
(a) Bin information, 4-neighbor

| Bin | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| **A* RunTime** | 0.01 | 0.04 | 0.12 | 0.34 | 1.06 | 3.43 | 10.47 | 31.57 | 105.95 | 966.96 |
| **A* Cost** | 13.8 | 36.6 | 74.3 | 126.1 | 182.9 | 238.8 | 328.4 | 476.8 | 750.3 | 1,643.8 |
| **A* Searches** | 2.0 | 4.3 | 8.1 | 13.4 | 19.4 | 25.8 | 35.7 | 52.3 | 82.5 | 181.3 |

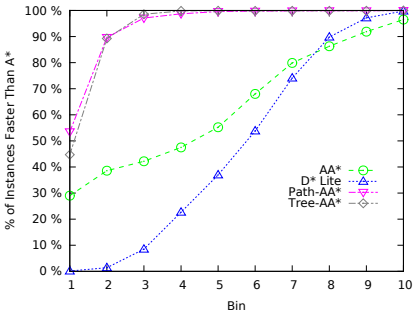
(b) Bin information, 8-neighbor



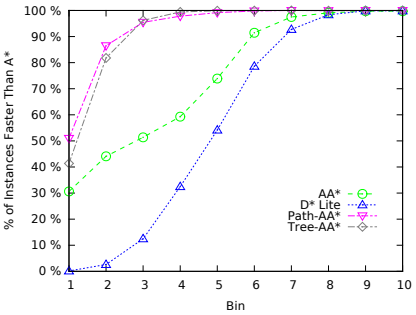

(c) Runtime factor, 4-neighbor




(d) Runtime factor, 8-neighbor




(e) Percentage of instances faster than A*, 4-neighbor




(f) Percentage of instances faster than A*, 8-neighbor

**Fig. 14** Results on Populated Game Maps

20%, 30%, and 40% blocked cells placed randomly. For each map, we generate a single solvable problem instance, for a total of 600 problem instances.

– **Original Baldur's Gate II Maps:** We use the AR0202SR, AR0307SR, AR0400SR and AR0602SR maps with sizes 208 × 244, 267 × 320, 256 × 240 and 299 × 308, respectively.[2] For each map, we generate 150 solvable problem instances, for a total of 600 problem instances.

For each setting, we divide the problem instances into six problem bins, which contain problem instances of different difficulty. We measure the difficulty of a

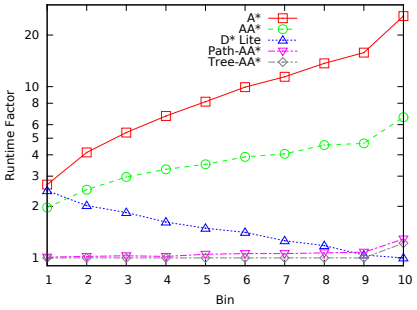
[2] All game maps can be found in Nathan Sturtevant's repository at http://www.movingai.com/benchmarks/.

| Bin | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| **A* RunTime** | 0.91 | 6.37 | 18.43 | 40.71 | 76.01 | 132.86 | 227.00 | 382.38 | 666.80 | 1,841.14 |
| **A* Cost** | 168.5 | 437.6 | 713.9 | 1,000.1 | 1,296.6 | 1,626.9 | 2,096.0 | 2,584.6 | 3,428.4 | 5,248.8 |
| **A* Searches** | 18.9 | 47.5 | 76.3 | 105.2 | 135.7 | 170.7 | 215.8 | 265.8 | 347.8 | 526.9 |

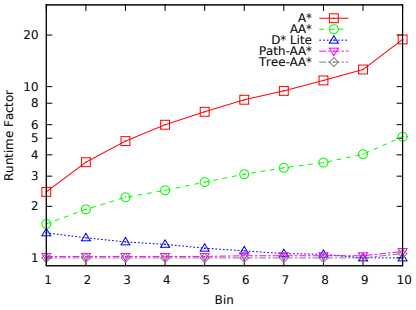
(a) Bin information, 4-neighbor

| Bin | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| **A* RunTime** | 1.13 | 5.79 | 14.46 | 28.57 | 49.13 | 78.73 | 118.91 | 183.63 | 295.74 | 779.28 |
| **A* Cost** | 113.3 | 221.2 | 300.3 | 368.8 | 434.0 | 502.8 | 571.2 | 653.1 | 752.4 | 998.2 |
| **A* Searches** | 11.5 | 22.1 | 29.6 | 36.6 | 42.9 | 49.7 | 56.3 | 64.7 | 74.4 | 98.3 |

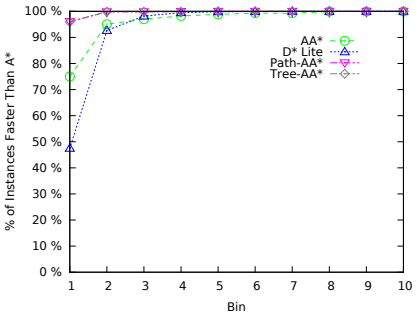
(b) Bin information, 8-neighbor



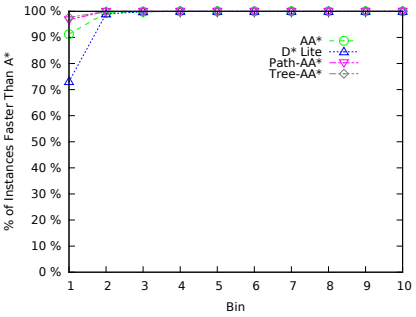

(c) Runtime factor, 4-neighbor




(d) Runtime factor, 8-neighbor




(e) Percentage of instances faster than A*, 4-neighbor

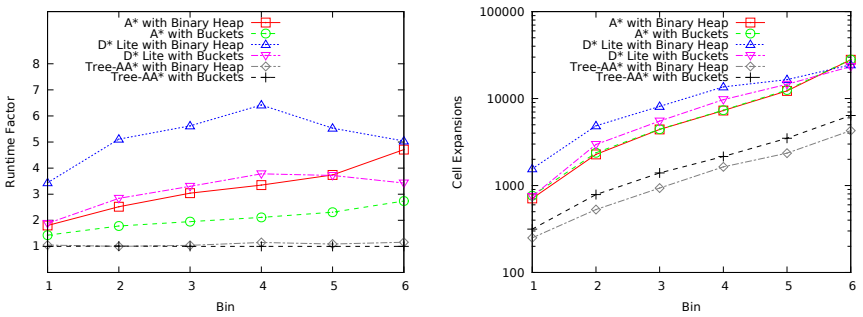



(f) Percentage of instances faster than A*, 8-neighbor

**Fig. 15** Results on Populated Office Maps

problem instance by the time it takes A* with a binary heap to solve it. Each bin contains 100 problem instances. As before, the hardest problem instance in bin $n$ is easier than the easiest problem instance in bin $n + 1$.

Figures 16 and 17 show the runtime factors and numbers of expansions per bin for the three algorithms with the two priority queue implementations. In order to compare the implementations of the priority queue, we report the runtime factor and the number of cell expansions. One key observation is that the bucket implementations are faster than the priority queue implementations. Indeed, our results show that expansions are on average 1.52 times faster with buckets than binary heaps.




(a) Runtime factor (b) Expansions

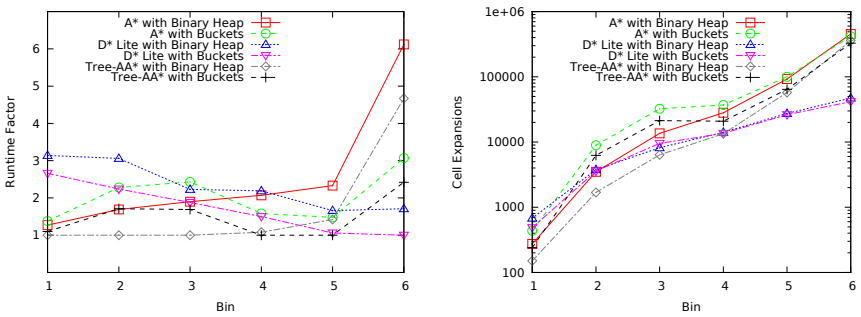
**Fig. 16** Buckets versus Priority Queues in Maps with Randomly Blocked Cells

Figure 16 presents the results obtained in maps with randomly blocked cells. In this kind of maps, we observe the following relationships:

- **Best:** The runtime factors show that Tree-AA* is the fastest algorithm on average in maps with randomly blocked cells.
- **Tree-AA*:** Tree-AA* with a Binary Heap is about as fast as Tree-AA* with Buckets on average but performs slightly fewer expansions. The differences in the numbers of expansions for all algorithms are only due to tie breaking when they choose a cell with the smallest $f$-value in the priority queue, which depends on implementation issues. In particular, A* with Buckets is not guaranteed to break ties towards larger $g$-values, which might contribute to it performing slightly more expansions than A* with a Binary Heap.
- **D* Lite:** D* Lite with Buckets is faster than D* Lite with a Binary Heap on average and performs fewer expansions.
- **A* with Buckets:** A* with Buckets is faster than A* with a Binary Heap on average but performs about the same number of expansions.

Figure 17 presents the results obtained in the original Baldur's Gate II maps. In this kind of maps we can notice the following relationships:

- **Best:** The runtime factors show that Tree-AA* with a Binary Heap is the fastest algorithm on average in Baldur's Gate II maps for easy problem instances (bins 1-3), Tree-AA* with Buckets is the fastest algorithm for medium-to-difficult problem instances (bins 4-5), and D* Lite with Buckets is the fastest algorithms for difficult problem instances (bin 6).
- **Tree-AA*:** Tree-AA* with a Binary Heap is faster than Tree-AA* with Buckets on average and performs fewer expansions for easy problem instances (bins 1-3), while Tree-AA* with Buckets is faster than Tree-AA* with a Binary Heap but performs about the same number of expansions for difficult problem instances (bins 4-6).
- **D* Lite:** D* Lite with Buckets is faster than D* Lite with a Binary Heap on average but performs about the same number of expansions.
- **A*:** A* with a Binary Heap is faster than A* with Buckets on average and performs fewer expansions for easy problem instances (bins 1-3), while A* with Buckets is faster than A* with a Binary Heap but performs about the same number of expansions for difficult problem instances (bins 4-6).




(a) Runtime factor (b) Expansions

**Fig. 17** Buckets versus Priority Queues in Baldur's Gate II Maps

## 7 Variants of Path-AA* and Tree-AA*

Our previous publications presented and evaluated variants of Path-AA* and Tree-AA*. In [9] we described a variant of Path-AA* that uses an improved tie-breaking rule to select a state from the OPEN list to guide the forward A* search towards a state on the reusable path. In [10], on the other hand, we described Tree-AA*-Back, a variant of Tree-AA* whose first A* search runs backward to build a large reusable tree.

Both variants outperform their predecessors in some but not all situations. For this reason, we describe them briefly in the following but do not provide experimental results for them.

### 7.1 Path-AA* with Tie-Breaking Optimization

How the A* searches performed by Path-AA* break ties among states with the same $f$-value when deciding which state to expand next determines how many states they expand and thus how fast they are. The objective of Path-AA* with tie-breaking optimization is to guide its A* searches so that a state on the reusable path is expanded as quickly as possible. The first A* search of Path-AA* breaks ties in favor of larger $g$-values, which is known to be a good tie-breaking strategy for A*. The following A* searches break ties in favor of states so that the estimated minimum cost from them to a state on the reusable path, as given by the user-provided $H$-values, is small. Path-AA* uses a greedy approximation for this purpose. It maintains two pointers $p$ and $p'$ and, before running an A* search, sets $p := r$ and $p' := \mathit{nextstate}(p)$, where $r$ is the first state on the reusable path. During the A* search, whenever it adds a state $s$ to the OPEN list, it computes the estimated minimum cost $\min(H(s,p), H(s,p'))$ and, as long as $H(s,p) > H(s,p')$, repeatedly advances pointers $p$ and $p'$ by assigning $p := p'$ and $p' := \mathit{nextstate}(p)$. When the reusable path consists of only state $s_{goal}$, this behaviour is similar to breaking ties in favor of larger $g$-values.

Path-AA* with tie-breaking optimization typically runs faster than Path-AA* that breaks ties in favor of larger $g$-values on four-neighbor grids but not necessarily on other graphs. More details, including experimental results, can be found in [9].

### 7.2 Tree-AA*-Back

An A* search finds cost-minimal paths from the start state of the search to all expanded states. Thus, if the first A* search of Tree-AA* is a backward search and the $h$-values of the expanded states are suitably updated, 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 (forward) A* search of (regular) Tree-AA* is degenerate since it contains only the expanded states on the cost-minimal path from state $s_{current}$ to state $s_{goal}$, while the reusable tree after the first A* search of Tree-AA*-Back is likely non-degenerate since it contains all expanded states, which might allow the future forward A* searches to terminate earlier.

Tree-AA*-Back typically spends more time searching than Tree-AA* before the agent starts to move, but its subsequent searches are much faster than those of Tree-AA*. Thus, Tree-AA*-Back should be used instead of Tree-AA* if the first search can be run offline. More details can be found in [10].

## 8 Conclusions

In this article, we introduced two new incremental heuristic search algorithms, called Path-AA* and Tree-AA*, for path planning with the freespace assumption. So far, there were two classes of incremental heuristic search algorithms: Incremental search algorithms of the first class (such as AA*) made the $h$-values of the current A* search more informed, while incremental search algorithms of the second class (such as D* Lite) changed the search tree of the current A* search to the search tree of the next A* search. Path-AA* reuses the suffix of the cost-minimal path of the current forward A* search (= reusable path) to terminate its next forward A* search before it is about to expand the goal state. Tree-AA* generalizes Path-AA* by reusing suffixes of the cost-minimal paths of the current and all previous forward A* searches (= reusable tree) to terminate the next forward A* search even earlier. Overall, Tree-AA* is the first incremental heuristic search algorithm to combine the principles of both classes of incremental heuristic search algorithms. We demonstrated experimentally that both Path-AA* and Tree-AA* can be faster than AA* and D* Lite, the state-of-the-art incremental heuristic search algorithms for path planning with the freespace assumption.

As future work, we consider extending the applicability of Tree-AA*. One possibility is to integrate Tree-AA* with Generalized Adaptive A* to deal with decreasing arc costs. This would enable Tree-AA* to work in dynamic terrain, in which already observed blocked cells may actually disappear. We also consider applying ideas from Tree-AA* to create incremental versions of sampling based methods, such as RRT* and LQR-RRT*, that work in partially known terrains.

## Acknowledgments

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 un-

der contract/grant number 11080063, NSF under contract/grant number 115-1319966, 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, companies or the U.S. government. We thank Nathan Sturtevant for making his maps available at *movingai.com*.

## Appendix

| | Runtime per Bin | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 0.011 | 0.029 | 0.079 | 0.288 | 0.944 | 2.514 | 6.016 | 14.342 | 42.253 | 662.198 | 72.867 |
| **AA*** | 0.011 | 0.033 | 0.098 | 0.402 | 1.186 | 3.182 | 6.661 | 14.514 | 36.652 | 229.741 | 29.248 |
| **D* Lite** | 0.042 | 0.160 | 0.737 | 1.453 | 2.646 | 4.987 | 8.781 | 16.413 | 32.854 | 82.347 | 15.042 |
| **PATH-AA*** | 0.011 | 0.030 | 0.076 | 0.248 | 0.633 | 1.624 | 3.104 | 6.648 | 17.711 | 155.010 | 18.509 |
| **TREE-AA*** | 0.011 | 0.030 | 0.076 | 0.246 | 0.619 | 1.570 | 2.952 | 6.257 | 16.690 | 152.388 | 18.084 |
| | **Solution Cost per Bin** | | | | | | | | | | |
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 17.9 | 49.6 | 110.9 | 158.7 | 240.9 | 367.1 | 548.7 | 846.1 | 1,414.8 | 2,882.6 | 663.7 |
| **AA*** | 18.0 | 49.9 | 111.8 | 162.4 | 247.6 | 377.2 | 557.2 | 855.2 | 1,432.9 | 2,878.5 | 669.1 |
| **D* Lite** | 18.6 | 53.4 | 127.7 | 192.7 | 291.1 | 433.1 | 615.1 | 918.3 | 1,439.4 | 2,660.9 | 675.0 |
| **PATH-AA*** | 18.0 | 49.9 | 111.8 | 162.4 | 247.6 | 377.2 | 557.3 | 855.1 | 1,432.3 | 2,879.6 | 669.1 |
| **TREE-AA*** | 18.0 | 49.9 | 111.8 | 162.5 | 248.0 | 378.2 | 559.7 | 861.0 | 1,447.4 | 2,909.0 | 674.6 |
| | **Number of Expansions per Bin** | | | | | | | | | | |
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 23 | 89 | 338 | 1,695 | 5,852 | 15,829 | 38,431 | 92,859 | 283,670 | 4,099,019 | 453,780 |
| **AA*** | 24 | 116 | 459 | 2,418 | 7,205 | 19,063 | 39,495 | 84,818 | 215,142 | 1,323,466 | 169,221 |
| **D* Lite** | 99 | 579 | 3,100 | 5,632 | 9,567 | 17,140 | 28,404 | 47,879 | 80,703 | 151,010 | 34,411 |
| **PATH-AA*** | 21 | 91 | 316 | 1,431 | 3,815 | 9,775 | 18,783 | 40,206 | 112,394 | 948,102 | 113,493 |
| **TREE-AA*** | 21 | 91 | 313 | 1,406 | 3,702 | 9,457 | 17,971 | 38,124 | 106,943 | 929,146 | 110,717 |
| | **Number of Percolations per Bin** | | | | | | | | | | |
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 200 | 933 | 3,929 | 20,661 | 77,739 | 220,609 | 543,275 | 1,311,032 | 3,777,619 | 43,667,232 | 4,962,323 |
| **AA*** | 207 | 1,282 | 5,458 | 30,477 | 98,146 | 276,880 | 598,133 | 1,333,471 | 3,410,011 | 20,594,032 | 2,634,810 |
| **D* Lite** | 451 | 3,369 | 21,846 | 46,816 | 89,449 | 171,191 | 303,741 | 551,791 | 1,010,013 | 2,193,905 | 439,257 |
| **PATH-AA*** | 186 | 1,021 | 3,738 | 16,989 | 47,888 | 130,484 | 255,684 | 559,483 | 1,533,617 | 13,461,263 | 1,601,035 |
| **TREE-AA*** | 186 | 1,018 | 3,707 | 16,673 | 46,307 | 125,691 | 242,659 | 523,825 | 1,435,266 | 13,109,310 | 1,550,464 |

**Table 1** 4-Connected Game Maps

| | Runtime per Bin | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 0.011 | 0.031 | 0.077 | 0.237 | 0.775 | 2.057 | 4.926 | 12.042 | 36.208 | 724.301 | 78.066 |
| **AA*** | 0.012 | 0.031 | 0.077 | 0.239 | 0.776 | 2.047 | 4.758 | 11.665 | 30.135 | 256.492 | 30.623 |
| **D* Lite** | 0.037 | 0.108 | 0.419 | 0.957 | 1.717 | 3.554 | 7.087 | 14.152 | 30.456 | 81.656 | 14.014 |
| **PATH-AA*** | 0.011 | 0.029 | 0.067 | 0.152 | 0.413 | 1.045 | 2.393 | 6.177 | 16.001 | 178.855 | 20.514 |
| **TREE-AA*** | 0.012 | 0.030 | 0.067 | 0.152 | 0.404 | 1.005 | 2.263 | 5.804 | 14.861 | 172.278 | 19.688 |
| | **Solution Cost per Bin** | | | | | | | | | | |
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 14.4 | 39.9 | 94.9 | 138.5 | 188.5 | 285.5 | 425.5 | 657.9 | 1,103.7 | 2,330.1 | 527.9 |
| **AA*** | 14.4 | 39.9 | 94.9 | 138.5 | 188.5 | 285.5 | 425.4 | 659.5 | 1,106.1 | 2,340.6 | 529.3 |
| **D* Lite** | 14.4 | 40.0 | 95.5 | 141.1 | 195.6 | 297.8 | 444.4 | 684.9 | 1,125.8 | 2,267.5 | 530.7 |
| **PATH-AA*** | 14.4 | 39.9 | 94.9 | 138.5 | 188.5 | 285.5 | 425.5 | 659.5 | 1,106.1 | 2,342.3 | 529.5 |
| **TREE-AA*** | 14.4 | 39.9 | 94.9 | 138.5 | 188.8 | 286.2 | 427.0 | 662.5 | 1,112.8 | 2,356.6 | 532.2 |
| | **Number of Expansions per Bin** | | | | | | | | | | |
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 16 | 57 | 187 | 895 | 3,376 | 9,235 | 22,533 | 56,650 | 174,111 | 3,028,684 | 329,574 |
| **AA*** | 16 | 56 | 184 | 864 | 3,205 | 8,640 | 20,154 | 49,455 | 126,531 | 1,050,524 | 125,963 |
| **D* Lite** | 50 | 265 | 1,486 | 3,146 | 4,671 | 8,705 | 15,620 | 28,053 | 54,731 | 121,504 | 23,823 |
| **PATH-AA*** | 14 | 44 | 123 | 409 | 1,363 | 3,686 | 8,732 | 23,924 | 65,734 | 764,551 | 86,858 |
| **TREE-AA*** | 14 | 44 | 122 | 400 | 1,309 | 3,492 | 8,174 | 22,487 | 61,525 | 737,390 | 83,496 |
| | **Number of Percolations per Bin** | | | | | | | | | | |
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 176 | 715 | 2,583 | 12,078 | 47,688 | 136,318 | 337,866 | 820,768 | 2,366,352 | 30,190,719 | 3,391,526 |
| **AA*** | 177 | 714 | 2,577 | 12,063 | 47,537 | 136,072 | 330,990 | 823,379 | 2,131,698 | 16,619,241 | 2,010,445 |
| **D* Lite** | 283 | 1,590 | 9,777 | 26,028 | 51,889 | 113,629 | 232,638 | 467,230 | 989,540 | 2,535,303 | 442,790 |
| **PATH-AA*** | 163 | 589 | 1,853 | 5,881 | 20,137 | 57,527 | 139,475 | 372,919 | 995,163 | 11,093,042 | 1,268,675 |
| **TREE-AA*** | 163 | 587 | 1,840 | 5,752 | 19,252 | 54,036 | 128,728 | 343,810 | 903,858 | 10,561,640 | 1,201,967 |

**Table 2** 8-Connected Game Maps

| | Runtime per Bin | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 0.601 | 1.915 | 3.493 | 5.603 | 8.073 | 10.935 | 14.667 | 19.719 | 28.317 | 79.023 | 17.235 |
| **AA*** | 0.677 | 1.978 | 3.695 | 5.818 | 8.576 | 11.535 | 15.319 | 20.350 | 28.900 | 61.958 | 15.880 |
| **D* Lite** | 1.322 | 3.304 | 4.886 | 7.304 | 10.191 | 12.581 | 15.942 | 20.057 | 26.690 | 36.642 | 13.892 |
| **PATH-AA*** | 0.340 | 0.905 | 1.733 | 2.719 | 3.877 | 4.857 | 6.531 | 8.757 | 12.344 | 33.517 | 7.558 |
| **TREE-AA*** | 0.305 | 0.757 | 1.340 | 2.023 | 2.885 | 3.705 | 4.964 | 6.676 | 9.833 | 30.731 | 6.322 |
| | **Solution Cost per Bin** | | | | | | | | | | |
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 179.2 | 335.8 | 457.7 | 575.3 | 684.5 | 793.8 | 900.8 | 1,034.4 | 1,207.1 | 1,622.6 | 779.1 |
| **AA*** | 184.1 | 337.2 | 459.9 | 575.6 | 692.7 | 799.7 | 904.9 | 1,034.0 | 1,209.0 | 1,594.4 | 779.1 |
| **D* Lite** | 231.9 | 416.0 | 538.6 | 674.8 | 806.8 | 904.5 | 1,012.7 | 1,143.5 | 1,312.7 | 1,613.8 | 865.5 |
| **PATH-AA*** | 184.1 | 337.3 | 460.0 | 575.7 | 692.8 | 800.0 | 905.1 | 1,034.2 | 1,208.8 | 1,595.2 | 779.3 |
| **TREE-AA*** | 184.8 | 339.5 | 463.9 | 581.3 | 699.3 | 806.6 | 913.5 | 1,043.8 | 1,217.8 | 1,601.0 | 785.1 |
| | **Number of Expansions per Bin** | | | | | | | | | | |
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 3,855 | 12,300 | 22,207 | 35,005 | 49,702 | 66,866 | 89,150 | 120,312 | 172,765 | 528,678 | 110,084 |
| **AA*** | 4,232 | 12,132 | 22,369 | 34,465 | 50,245 | 66,994 | 88,336 | 117,301 | 165,809 | 380,796 | 94,268 |
| **D* Lite** | 3,830 | 9,738 | 14,361 | 21,549 | 29,773 | 37,332 | 47,869 | 60,559 | 80,615 | 104,438 | 41,006 |
| **PATH-AA*** | 2,072 | 5,530 | 10,555 | 16,203 | 23,049 | 29,086 | 38,878 | 52,765 | 75,188 | 224,741 | 47,807 |
| **TREE-AA*** | 1,831 | 4,622 | 8,244 | 12,259 | 17,582 | 22,765 | 30,456 | 41,658 | 62,137 | 209,391 | 41,094 |
| | **Number of Percolations per Bin** | | | | | | | | | | |
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 45,317 | 163,532 | 316,032 | 528,957 | 777,464 | 1,061,984 | 1,436,051 | 1,949,635 | 2,783,090 | 6,815,935 | 1,587,800 |
| **AA*** | 51,827 | 165,880 | 327,522 | 534,989 | 801,860 | 1,089,854 | 1,462,386 | 1,967,873 | 2,792,935 | 5,907,432 | 1,510,256 |
| **D* Lite** | 47,087 | 123,596 | 184,984 | 280,032 | 394,414 | 485,909 | 617,694 | 783,207 | 1,049,973 | 1,469,790 | 543,669 |
| **PATH-AA*** | 22,201 | 67,250 | 140,041 | 231,225 | 335,136 | 423,840 | 581,014 | 796,812 | 1,122,178 | 3,038,103 | 675,780 |
| **TREE-AA*** | 19,061 | 53,906 | 103,152 | 164,309 | 239,717 | 311,990 | 427,390 | 589,999 | 876,149 | 2,749,724 | 553,540 |

**Table 3** 4-Connected Office Maps

| | Runtime per Bin | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 0.635 | 2.002 | 3.627 | 5.716 | 8.093 | 10.852 | 14.456 | 19.230 | 27.495 | 110.858 | 20.297 |
| **AA*** | 0.635 | 2.027 | 3.666 | 5.786 | 8.229 | 10.973 | 15.072 | 19.367 | 28.009 | 78.251 | 17.202 |
| **D* Lite** | 1.299 | 3.029 | 5.105 | 7.383 | 9.975 | 12.545 | 16.286 | 20.244 | 27.291 | 43.115 | 14.627 |
| **PATH-AA*** | 0.352 | 1.085 | 1.849 | 2.833 | 4.100 | 5.635 | 8.052 | 10.533 | 17.058 | 55.353 | 10.685 |
| **TREE-AA*** | 0.326 | 0.933 | 1.540 | 2.248 | 3.106 | 4.264 | 6.061 | 7.727 | 12.573 | 50.404 | 8.918 |
| | **Solution Cost per Bin** | | | | | | | | | | |
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 151.8 | 290.9 | 395.6 | 502.1 | 592.2 | 681.2 | 789.2 | 909.5 | 1,062.6 | 1,489.8 | 686.5 |
| **AA*** | 151.3 | 291.3 | 395.0 | 501.9 | 592.1 | 680.5 | 790.0 | 909.7 | 1,063.9 | 1,495.3 | 687.1 |
| **D* Lite** | 167.3 | 301.5 | 411.2 | 516.1 | 608.2 | 686.0 | 795.3 | 899.5 | 1,046.8 | 1,389.2 | 682.1 |
| **PATH-AA*** | 151.3 | 291.3 | 395.1 | 502.0 | 592.2 | 680.6 | 790.1 | 909.7 | 1,064.2 | 1,495.7 | 687.2 |
| **TREE-AA*** | 152.3 | 295.2 | 400.4 | 510.0 | 602.6 | 692.8 | 804.0 | 925.5 | 1,081.4 | 1,504.0 | 696.8 |
| | **Number of Expansions per Bin** | | | | | | | | | | |
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 2,917 | 9,478 | 17,439 | 27,250 | 38,071 | 50,763 | 67,783 | 89,838 | 127,416 | 530,584 | 96,154 |
| **AA*** | 2,718 | 8,974 | 16,477 | 25,854 | 36,469 | 48,195 | 66,315 | 84,399 | 120,396 | 354,802 | 76,460 |
| **D* Lite** | 2,569 | 5,855 | 9,461 | 13,550 | 17,806 | 22,562 | 28,329 | 34,872 | 44,099 | 64,405 | 24,351 |
| **PATH-AA*** | 1,293 | 4,262 | 7,422 | 11,304 | 16,332 | 22,473 | 32,557 | 42,208 | 69,038 | 253,875 | 46,076 |
| **TREE-AA*** | 1,152 | 3,544 | 6,031 | 8,799 | 12,222 | 16,877 | 24,678 | 31,425 | 52,448 | 235,739 | 39,292 |
| | **Number of Percolations per Bin** | | | | | | | | | | |
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 36,105 | 129,476 | 251,611 | 413,836 | 605,292 | 813,679 | 1,102,416 | 1,465,886 | 2,074,241 | 6,025,821 | 1,291,836 |
| **AA*** | 36,792 | 132,957 | 257,835 | 422,949 | 620,546 | 832,589 | 1,159,079 | 1,500,740 | 2,161,982 | 5,572,596 | 1,269,806 |
| **D* Lite** | 41,570 | 102,909 | 177,753 | 257,528 | 350,383 | 438,445 | 569,351 | 714,171 | 956,365 | 1,489,043 | 509,752 |
| **PATH-AA*** | 16,380 | 60,167 | 110,439 | 178,057 | 271,672 | 380,832 | 554,161 | 734,595 | 1,209,150 | 3,695,285 | 721,074 |
| **TREE-AA*** | 14,133 | 47,471 | 84,218 | 128,304 | 186,324 | 262,367 | 382,327 | 493,488 | 826,599 | 3,281,577 | 570,681 |

**Table 4** 8-Connected Office Maps

| | Runtime per Bin | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 0.763 | 2.643 | 5.162 | 8.651 | 13.889 | 22.033 | 37.020 | 76.973 | 326.628 | 10,249.313 | 1,074.307 |
| **AA*** | 0.790 | 2.729 | 5.409 | 8.903 | 14.283 | 22.621 | 37.428 | 80.693 | 274.249 | 2,035.345 | 248.245 |
| **D* Lite** | 2.063 | 6.660 | 12.503 | 18.833 | 26.877 | 36.085 | 50.369 | 71.521 | 125.714 | 312.085 | 66.271 |
| **PATH-AA*** | 0.433 | 1.436 | 2.816 | 4.435 | 6.932 | 10.064 | 15.599 | 27.210 | 87.493 | 664.685 | 82.110 |
| **TREE-AA*** | 0.324 | 0.984 | 1.904 | 2.845 | 4.387 | 5.995 | 8.994 | 15.202 | 54.260 | 533.014 | 62.791 |
| | **Solution Cost per Bin** | | | | | | | | | | |
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 250.2 | 491.2 | 680.4 | 859.0 | 1,056.3 | 1,312.2 | 1,709.2 | 2,797.5 | 9,133.0 | 25,096.3 | 4,338.5 |
| **AA*** | 250.5 | 491.3 | 683.6 | 859.3 | 1,057.3 | 1,323.4 | 1,717.7 | 2,879.8 | 9,551.5 | 25,010.0 | 4,382.4 |
| **D* Lite** | 255.6 | 509.5 | 705.8 | 885.4 | 1,104.2 | 1,354.2 | 1,797.9 | 3,042.8 | 9,924.1 | 24,431.7 | 4,401.1 |
| **PATH-AA*** | 250.5 | 491.3 | 683.6 | 859.4 | 1,057.3 | 1,323.4 | 1,718.6 | 2,872.8 | 9,559.3 | 25,029.2 | 4,384.5 |
| **TREE-AA*** | 253.9 | 500.5 | 698.9 | 879.9 | 1,085.8 | 1,364.4 | 1,771.2 | 2,973.4 | 9,911.6 | 25,838.8 | 4,527.8 |
| | **Number of Expansions per Bin** | | | | | | | | | | |
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 3,935 | 13,451 | 25,853 | 41,992 | 64,936 | 98,232 | 159,731 | 329,521 | 1,858,786 | 52,859,768 | 5,545,621 |
| **AA*** | 3,940 | 13,415 | 26,264 | 41,656 | 64,301 | 97,077 | 153,326 | 321,261 | 1,301,320 | 9,595,401 | 1,161,796 |
| **D* Lite** | 9,255 | 29,867 | 55,106 | 81,628 | 113,957 | 148,559 | 198,621 | 249,332 | 269,047 | 541,372 | 169,674 |
| **PATH-AA*** | 1,997 | 6,651 | 13,075 | 20,068 | 30,372 | 43,458 | 66,460 | 116,955 | 469,098 | 3,641,298 | 440,943 |
| **TREE-AA*** | 1,459 | 4,531 | 8,845 | 12,955 | 19,548 | 26,290 | 39,248 | 69,236 | 310,720 | 2,957,763 | 345,059 |
| | **Number of Percolations per Bin** | | | | | | | | | | |
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 63494 | 243485 | 493291 | 831869 | 1309026 | 1997499 | 3202061 | 5993388 | 25235537 | 252429668 | 29179932 |
| **AA*** | 63801 | 244195 | 500997 | 832959 | 1310986 | 2004093 | 3184321 | 6320118 | 23286942 | 134450131 | 17219854 |
| **D* Lite** | 62399 | 218043 | 424828 | 655991 | 948423 | 1291682 | 1826646 | 2603700 | 4654730 | 11325565 | 2401201 |
| **PATH-AA*** | 31177 | 118737 | 245671 | 397082 | 614610 | 883599 | 1356217 | 2186347 | 7214688 | 54116856 | 6716498 |
| **TREE-AA*** | 22351 | 79728 | 164674 | 253158 | 391514 | 522122 | 779746 | 1207103 | 4021852 | 41374837 | 4881709 |

**Table 5** 4-Connected Maps with Randomly Blocked Cells

| | Runtime per Bin | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 0.410 | 1.291 | 2.369 | 3.702 | 5.387 | 7.532 | 10.515 | 14.700 | 21.905 | 42.814 | 11.063 |
| **AA*** | 0.425 | 1.344 | 2.463 | 3.861 | 5.604 | 7.840 | 10.936 | 15.254 | 22.774 | 44.105 | 11.460 |
| **D* Lite** | 1.068 | 3.376 | 5.896 | 8.906 | 12.445 | 16.170 | 19.350 | 23.662 | 30.309 | 41.804 | 16.299 |
| **PATH-AA*** | 0.226 | 0.631 | 1.064 | 1.599 | 2.276 | 3.026 | 4.349 | 5.750 | 8.779 | 15.840 | 4.354 |
| **TREE-AA*** | 0.196 | 0.512 | 0.819 | 1.200 | 1.627 | 2.103 | 2.856 | 3.639 | 5.153 | 8.254 | 2.636 |
| | **Solution Cost per Bin** | | | | | | | | | | |
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 159.7 | 305.6 | 411.2 | 504.2 | 594.2 | 680.2 | 756.0 | 840.2 | 964.1 | 1,178.2 | 639.4 |
| **AA*** | 159.7 | 305.6 | 411.2 | 504.2 | 594.2 | 680.1 | 756.0 | 840.2 | 964.1 | 1,178.1 | 639.3 |
| **D* Lite** | 159.9 | 306.0 | 411.5 | 504.8 | 594.9 | 680.8 | 756.8 | 841.5 | 965.9 | 1,180.2 | 640.2 |
| **PATH-AA*** | 159.7 | 305.6 | 411.2 | 504.2 | 594.2 | 680.1 | 756.0 | 840.2 | 964.1 | 1,178.1 | 639.3 |
| **TREE-AA*** | 161.4 | 309.9 | 418.0 | 513.5 | 606.5 | 695.5 | 777.5 | 867.5 | 1,002.6 | 1,235.9 | 658.8 |
| | **Number of Expansions per Bin** | | | | | | | | | | |
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 1,543 | 5,141 | 9,531 | 14,875 | 21,510 | 29,906 | 40,469 | 55,864 | 80,648 | 135,318 | 39,480 |
| **AA*** | 1,542 | 5,139 | 9,527 | 14,869 | 21,502 | 29,897 | 40,456 | 55,845 | 80,621 | 135,269 | 39,467 |
| **D* Lite** | 3,677 | 11,657 | 19,892 | 29,849 | 40,818 | 52,204 | 61,092 | 72,761 | 89,824 | 114,104 | 49,588 |
| **PATH-AA*** | 587 | 1,820 | 3,153 | 4,796 | 6,860 | 9,101 | 13,102 | 17,348 | 26,372 | 46,452 | 12,959 |
| **TREE-AA*** | 458 | 1,337 | 2,200 | 3,282 | 4,462 | 5,779 | 7,852 | 10,005 | 14,132 | 22,035 | 7,154 |
| | **Number of Percolations per Bin** | | | | | | | | | | |
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 25,956 | 95,862 | 188,468 | 305,219 | 454,021 | 646,036 | 892,232 | 1,244,779 | 1,844,273 | 3,226,926 | 892,377 |
| **AA*** | 25,954 | 95,855 | 188,449 | 305,193 | 453,962 | 645,946 | 892,139 | 1,244,676 | 1,844,042 | 3,226,541 | 892,276 |
| **D* Lite** | 28,239 | 95,190 | 170,380 | 262,584 | 370,778 | 485,790 | 586,993 | 725,316 | 933,219 | 1,282,699 | 494,119 |
| **PATH-AA*** | 10,712 | 37,414 | 68,701 | 108,596 | 160,198 | 218,209 | 321,149 | 429,553 | 670,959 | 1,228,713 | 325,420 |
| **TREE-AA*** | 8,289 | 27,339 | 47,562 | 74,004 | 103,575 | 137,599 | 190,749 | 245,410 | 355,240 | 574,056 | 176,382 |

**Table 6** 8-Connected Maps with Randomly Blocked Cells

| | Runtime per Bin | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 0.012 | 0.040 | 0.126 | 0.390 | 1.098 | 2.914 | 7.657 | 21.072 | 68.613 | 671.087 | 77.301 |
| **AA*** | 0.013 | 0.040 | 0.125 | 0.376 | 1.000 | 2.496 | 5.834 | 14.173 | 38.865 | 238.809 | 30.173 |
| **D* Lite** | 0.037 | 0.101 | 0.276 | 0.761 | 1.776 | 3.527 | 6.283 | 10.816 | 20.449 | 61.191 | 10.522 |
| **PATH-AA*** | 0.012 | 0.031 | 0.078 | 0.193 | 0.431 | 0.921 | 1.909 | 4.207 | 10.581 | 55.630 | 7.399 |
| **TREE-AA*** | 0.012 | 0.031 | 0.075 | 0.178 | 0.396 | 0.852 | 1.776 | 3.934 | 9.850 | 51.434 | 6.854 |
| | **Solution Cost per Bin** | | | | | | | | | | |
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 17.7 | 47.1 | 93.0 | 161.8 | 252.5 | 374.5 | 541.1 | 797.9 | 1,257.9 | 2,601.4 | 614.5 |
| **AA*** | 17.7 | 47.1 | 93.1 | 162.0 | 253.0 | 376.4 | 544.8 | 802.9 | 1,264.7 | 2,605.9 | 616.8 |
| **D* Lite** | 17.7 | 47.3 | 94.9 | 168.3 | 266.6 | 397.9 | 570.4 | 828.1 | 1,262.7 | 2,539.1 | 619.3 |
| **PATH-AA*** | 17.7 | 47.1 | 93.1 | 162.1 | 253.2 | 376.9 | 545.9 | 804.6 | 1,267.9 | 2,615.2 | 618.4 |
| **TREE-AA*** | 17.7 | 47.2 | 93.6 | 163.0 | 254.7 | 378.6 | 547.7 | 807.1 | 1,269.9 | 2,620.7 | 620.0 |
| | **Number of Expansions per Bin** | | | | | | | | | | |
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 35 | 182 | 709 | 2,425 | 7,266 | 20,186 | 54,287 | 147,984 | 452,588 | 3,601,122 | 428,678 |
| **AA*** | 35 | 178 | 666 | 2,168 | 6,034 | 15,561 | 37,024 | 89,531 | 236,174 | 1,264,755 | 165,213 |
| **D* Lite** | 74 | 373 | 1,308 | 3,822 | 8,742 | 16,450 | 27,275 | 43,370 | 74,380 | 174,169 | 34,996 |
| **PATH-AA*** | 27 | 113 | 358 | 989 | 2,362 | 5,376 | 11,702 | 26,412 | 65,571 | 315,382 | 42,829 |
| **TREE-AA*** | 27 | 107 | 324 | 877 | 2,121 | 4,884 | 10,716 | 24,356 | 60,565 | 292,232 | 39,621 |
| | **Number of Percolations per Bin** | | | | | | | | | | |
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 240 | 1,623 | 7,440 | 27,807 | 85,724 | 238,790 | 643,081 | 1,778,727 | 5,580,359 | 48,052,104 | 5,641,589 |
| **AA*** | 240 | 1,617 | 7,297 | 26,612 | 78,384 | 208,268 | 508,490 | 1,270,601 | 3,502,277 | 20,554,408 | 2,615,819 |
| **D* Lite** | 331 | 1,924 | 7,578 | 24,876 | 63,462 | 131,778 | 237,607 | 407,306 | 748,726 | 1,972,875 | 359,646 |
| **PATH-AA*** | 183 | 957 | 3,568 | 10,822 | 26,335 | 59,048 | 127,878 | 295,619 | 767,108 | 4,174,390 | 546,591 |
| **TREE-AA*** | 179 | 900 | 3,167 | 9,305 | 22,946 | 52,113 | 114,082 | 266,454 | 693,684 | 3,801,342 | 496,417 |

**Table 7** 4-Connected Populated Game Maps

| | Runtime per Bin | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 0.013 | 0.039 | 0.116 | 0.343 | 1.057 | 3.427 | 10.470 | 31.566 | 105.953 | 966.963 | 111.995 |
| **AA*** | 0.013 | 0.039 | 0.112 | 0.312 | 0.840 | 2.168 | 5.367 | 13.276 | 36.455 | 242.394 | 30.098 |
| **D* Lite** | 0.036 | 0.089 | 0.225 | 0.553 | 1.247 | 2.385 | 4.625 | 9.272 | 19.655 | 68.855 | 10.694 |
| **PATH-AA*** | 0.012 | 0.032 | 0.077 | 0.188 | 0.480 | 1.221 | 2.916 | 6.634 | 16.344 | 80.995 | 10.890 |
| **TREE-AA*** | 0.013 | 0.033 | 0.077 | 0.182 | 0.458 | 1.169 | 2.792 | 6.358 | 15.732 | 78.058 | 10.487 |
| | **Solution Cost per Bin** | | | | | | | | | | |
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 13.8 | 36.6 | 74.3 | 126.1 | 182.9 | 238.8 | 328.4 | 476.8 | 750.3 | 1,643.8 | 387.2 |
| **AA*** | 13.8 | 36.6 | 74.3 | 126.2 | 183.3 | 239.5 | 329.5 | 479.2 | 752.3 | 1,646.9 | 388.1 |
| **D* Lite** | 13.8 | 36.6 | 74.4 | 126.4 | 183.7 | 240.3 | 330.2 | 481.4 | 755.6 | 1,650.5 | 389.3 |
| **PATH-AA*** | 13.8 | 36.6 | 74.4 | 126.3 | 183.4 | 239.8 | 330.1 | 480.2 | 754.2 | 1,651.5 | 389.0 |
| **TREE-AA*** | 13.8 | 36.6 | 74.5 | 126.6 | 183.8 | 240.2 | 330.6 | 481.0 | 755.7 | 1,654.0 | 389.7 |
| | **Number of Expansions per Bin** | | | | | | | | | | |
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 21 | 106 | 422 | 1,493 | 5,216 | 18,111 | 54,359 | 150,228 | 448,846 | 3,354,444 | 403,325 |
| **AA*** | 21 | 102 | 386 | 1,243 | 3,657 | 10,058 | 25,098 | 60,638 | 157,517 | 909,790 | 116,851 |
| **D* Lite** | 44 | 210 | 755 | 2,077 | 4,632 | 8,354 | 15,220 | 28,430 | 55,294 | 160,042 | 27,506 |
| **PATH-AA*** | 17 | 64 | 204 | 616 | 1,842 | 5,174 | 12,631 | 28,221 | 67,309 | 306,196 | 42,227 |
| **TREE-AA*** | 17 | 63 | 195 | 576 | 1,726 | 4,900 | 11,968 | 26,794 | 64,150 | 292,713 | 40,310 |
| | **Number of Percolations per Bin** | | | | | | | | | | |
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 193 | 1,094 | 4,846 | 17,771 | 58,732 | 189,011 | 564,543 | 1,610,761 | 4,952,935 | 40,216,891 | 4,761,678 |
| **AA*** | 193 | 1,083 | 4,695 | 16,436 | 48,571 | 128,783 | 326,170 | 823,617 | 2,240,914 | 14,553,767 | 1,814,423 |
| **D* Lite** | 253 | 1,308 | 4,997 | 14,892 | 36,152 | 70,976 | 138,579 | 277,128 | 575,635 | 1,888,728 | 300,865 |
| **PATH-AA*** | 153 | 661 | 2,306 | 7,252 | 20,765 | 55,150 | 136,535 | 317,018 | 784,670 | 3,992,633 | 531,714 |
| **TREE-AA*** | 152 | 645 | 2,175 | 6,609 | 18,924 | 51,321 | 127,536 | 296,930 | 738,360 | 3,777,095 | 501,975 |

**Table 8** 8-Connected Populated Game Maps

| | Runtime per Bin | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 0.910 | 6.375 | 18.430 | 40.707 | 76.009 | 132.862 | 227.003 | 382.376 | 666.798 | 1,841.137 | 339.261 |
| **AA*** | 0.669 | 3.861 | 10.133 | 19.864 | 32.775 | 52.092 | 80.555 | 127.567 | 196.855 | 472.805 | 99.717 |
| **D* Lite** | 0.838 | 3.119 | 6.278 | 9.758 | 13.866 | 18.923 | 24.991 | 33.059 | 43.968 | 71.327 | 22.613 |
| **PATH-AA*** | 0.344 | 1.580 | 3.519 | 6.179 | 9.764 | 14.131 | 21.080 | 30.019 | 45.395 | 92.104 | 22.412 |
| **TREE-AA*** | 0.339 | 1.542 | 3.416 | 6.041 | 9.302 | 13.374 | 19.878 | 27.979 | 42.215 | 87.056 | 21.114 |
| | **Solution Cost per Bin** | | | | | | | | | | |
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 168.5 | 437.6 | 713.9 | 1,000.1 | 1,296.6 | 1,626.9 | 2,096.0 | 2,584.6 | 3,428.4 | 5,248.8 | 1,860.1 |
| **AA*** | 169.9 | 442.3 | 726.5 | 1,014.0 | 1,316.8 | 1,648.7 | 2,113.3 | 2,617.9 | 3,434.2 | 5,261.2 | 1,874.5 |
| **D* Lite** | 174.9 | 450.2 | 746.1 | 1,023.1 | 1,330.7 | 1,681.2 | 2,138.1 | 2,628.7 | 3,431.1 | 5,216.6 | 1,882.1 |
| **PATH-AA*** | 169.7 | 442.0 | 726.5 | 1,009.3 | 1,311.1 | 1,639.1 | 2,114.2 | 2,606.6 | 3,446.3 | 5,250.2 | 1,871.5 |
| **TREE-AA*** | 169.7 | 442.7 | 727.5 | 1,013.5 | 1,314.1 | 1,642.0 | 2,120.8 | 2,607.9 | 3,442.9 | 5,250.3 | 1,873.2 |
| | **Number of Expansions per Bin** | | | | | | | | | | |
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 6948 | 50037 | 139986 | 294988 | 525307 | 872560 | 1422853 | 2308672 | 3886558 | 10203502 | 1971141 |
| **AA*** | 4362 | 26232 | 67900 | 130315 | 212584 | 329831 | 499591 | 773028 | 1171521 | 2693512 | 590888 |
| **D* Lite** | 3947 | 14232 | 26999 | 40500 | 55567 | 74267 | 95403 | 123134 | 159170 | 250181 | 84340 |
| **PATH-AA*** | 2134 | 10488 | 23231 | 40214 | 62588 | 88882 | 130182 | 182915 | 272986 | 543895 | 135751 |
| **TREE-AA*** | 2050 | 10027 | 22121 | 38630 | 58692 | 83017 | 121200 | 169058 | 251813 | 506972 | 126358 |
| | **Number of Percolations per Bin** | | | | | | | | | | |
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 59252 | 460168 | 1353105 | 2973497 | 5412802 | 9352671 | 15556845 | 25747174 | 43805972 | 113963825 | 21868531 |
| **AA*** | 44886 | 298391 | 828918 | 1664567 | 2780466 | 4464045 | 6892518 | 10934273 | 16798521 | 40137634 | 8484422 |
| **D* Lite** | 28546 | 119280 | 249755 | 395538 | 566742 | 782676 | 1043685 | 1382456 | 1847771 | 3009417 | 942587 |
| **PATH-AA*** | 17372 | 93085 | 218880 | 396975 | 640087 | 943774 | 1418545 | 2041357 | 3096637 | 6390377 | 1525709 |
| **TREE-AA*** | 16533 | 87797 | 205297 | 376421 | 588639 | 863719 | 1293736 | 1846028 | 2795626 | 5850865 | 1392466 |

**Table 9** 4-Connected Populated Office Maps

| | Runtime per Bin | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 1.129 | 5.789 | 14.462 | 28.573 | 49.128 | 78.728 | 118.907 | 183.631 | 295.741 | 779.278 | 155.537 |
| **AA*** | 0.732 | 3.060 | 6.812 | 11.819 | 19.116 | 28.949 | 42.287 | 60.885 | 94.630 | 210.878 | 47.917 |
| **D* Lite** | 0.650 | 2.088 | 3.719 | 5.720 | 7.828 | 10.349 | 13.310 | 17.746 | 23.503 | 41.322 | 12.624 |
| **PATH-AA*** | 0.473 | 1.630 | 3.068 | 4.873 | 7.048 | 9.630 | 12.909 | 17.329 | 24.135 | 45.090 | 12.618 |
| **TREE-AA*** | 0.465 | 1.598 | 3.008 | 4.773 | 6.891 | 9.392 | 12.578 | 16.897 | 23.503 | 43.984 | 12.309 |
| | **Solution Cost per Bin** | | | | | | | | | | |
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 113.3 | 221.2 | 300.3 | 368.8 | 434.0 | 502.8 | 571.2 | 653.1 | 752.4 | 998.2 | 491.5 |
| **AA*** | 113.8 | 222.0 | 301.4 | 371.1 | 435.3 | 505.7 | 573.7 | 655.3 | 755.6 | 1,002.9 | 493.7 |
| **D* Lite** | 113.9 | 222.7 | 302.6 | 371.5 | 436.6 | 506.1 | 574.9 | 656.2 | 757.4 | 1,002.9 | 494.5 |
| **PATH-AA*** | 113.6 | 221.9 | 301.2 | 370.9 | 435.0 | 505.3 | 573.2 | 655.0 | 755.1 | 1,003.1 | 493.4 |
| **TREE-AA*** | 113.7 | 222.0 | 301.3 | 371.0 | 435.3 | 505.6 | 573.5 | 655.4 | 755.8 | 1,004.7 | 493.8 |
| | **Number of Expansions per Bin** | | | | | | | | | | |
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 5886 | 30461 | 71454 | 130732 | 209856 | 321133 | 468922 | 698695 | 1090918 | 2807657 | 583572 |
| **AA*** | 3252 | 14152 | 31375 | 53548 | 84080 | 124355 | 176950 | 248348 | 375721 | 829393 | 194117 |
| **D* Lite** | 2265 | 7185 | 12324 | 18567 | 24964 | 32129 | 40614 | 52572 | 67877 | 113923 | 37242 |
| **PATH-AA*** | 1868 | 6618 | 12203 | 18958 | 26777 | 35920 | 47487 | 62455 | 85684 | 161090 | 45906 |
| **TREE-AA*** | 1843 | 6538 | 12050 | 18673 | 26375 | 35243 | 46580 | 61285 | 83895 | 158069 | 45055 |
| | **Number of Percolations per Bin** | | | | | | | | | | |
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **A*** | 48547 | 266528 | 656069 | 1236903 | 2043644 | 3208795 | 4753293 | 7223663 | 11394722 | 28603024 | 5943519 |
| **AA*** | 33541 | 162668 | 390096 | 696462 | 1138495 | 1741384 | 2549732 | 3671635 | 5710136 | 13043263 | 2913741 |
| **D* Lite** | 17539 | 62182 | 112969 | 174287 | 240845 | 317575 | 407272 | 537404 | 703976 | 1184851 | 375890 |
| **PATH-AA*** | 15709 | 60295 | 118361 | 191524 | 280816 | 389720 | 528173 | 712176 | 997210 | 1882891 | 517688 |
| **TREE-AA*** | 15435 | 59379 | 116490 | 187948 | 275510 | 380701 | 515690 | 695773 | 971104 | 1837782 | 505581 |

**Table 10** 8-Connected Populated Office Maps

## References


1. Bjornsson, Y., Enzenberger, M., Holte, R., Schaeffer, J., Yap, P.: Comparison of different grid abstractions for pathfinding on maps. In: Proceedings of the International Joint Conference on Artificial Intelligence, pp. 1511–1512 (2003)
2. Bulitko, V., Bjornsson, Y., Lustrek, M., Schaeffer, J., Sigmundarson, S.: Dynamic control in path-planning with real-time heuristic search. In: Proceedings of the International Conference on Automated Planning and Scheduling, pp. 49–56 (2007)
3. Choset, H., Thrun, S., Kavraki, L., Burgard, W., Lynch, K.: Principles of Robot Motion: Theory, Algorithms, and Implementations. MIT Press (2005)
4. Edelkamp, S., Schrödl, S.: Heuristic Search - Theory and Applications. Academic Press (2012)
5. Ferguson, D., Stentz, A.: Using interpolation to improve path planning: The Field D* algorithm. Journal of Field Robotics **23**(2), 79–101 (2006)
6. Hart, P., Nilsson, N., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. on Systems Sci. and Cybernetics **2**, 100–107 (1968)
7. Hernandez, C., Baier, J., Uras, T., Koenig, S.: Position Paper: Incremental Search Algorithms Considered Poorly Understood. In: Proceedings of the Symposium on Combinatorial Search, pp. 159–161 (2012)
8. Hernández, C., Baier, J.A.: Avoiding and escaping depressions in real-time heuristic search. Journal of Artificial Intelligence Research **43**, 523–570 (2012)
9. Hernández, C., Meseguer, P., Sun, X., Koenig, S.: Path-Adaptive A* for incremental heuristic search in unknown terrain [short paper]. In: Proceedings of the International Conference on Automated Planning and Scheduling, pp. 358–361 (2009)
10. Hernandez, C., Sun, X., Koenig, S., Meseguer, P.: Tree Adaptive A*. In: Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 123–130 (2011)
11. Holte, R., M.Perez, Zimmer, R., MacDonald, A.: Hierarchical A*: Searching abstraction hierarchies efficiently. In: Proc. of AAAI, pp. 530–535 (1996)
12. Koenig, S., Furcy, D., Bauer, C.: Heuristic search-based replanning. In: Proceedings of the International Conference on Artificial Intelligence Planning Systems, pp. 294–301 (2002)
13. Koenig, S., Likhachev, M.: Adaptive A*. In: Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 1311–1312 (2005)
14. Koenig, S., Likhachev, M.: Fast replanning for navigation in unknown terrain. Transactions on Robotics **21**, 354–363 (2005)
15. Koenig, S., Likhachev, M.: A new principle for incremental heuristic search: Theoretical results. In: Proceedings of the International Conference on Autonomous Planning and Scheduling, pp. 410–413 (2006)
16. Koenig, S., Likhachev, M., Liu, Y., Furcy, D.: Incremental heuristic search in artificial intelligence. Artificial Intelligence Magazine **25**(2), 99–112 (2004)
17. Koenig, S., Tovey, C., Smirnov, Y.: Performance bounds for planning in unknown terrain. Artificial Intelligence **147**(1-2), 253–279 (2003)
18. Likhachev, M., Ferguson, D., Gordon, G., Stentz, A., Thrun, S.: Anytime Dynamic A*: An anytime, replanning algorithm. In: Proceedings of the International Conference on Automated Planning and Scheduling, pp. 262–271 (2005)
19. Likhachev, M., Ferguson, D., Gordon, G., Stentz, A., Thrun, S.: Anytime search in dynamic graphs. Artificial Intelligence **172**(14), 1613–1643 (2008)
20. Likhachev, M., Koenig, S.: Incremental heuristic search in games: The quest for speed. In: Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, pp. 118–120 (2006)
21. Matsuta, K., Kobayashi, H., Shinohara, A.: Multi-target Adaptive A*. In: Proc. of the Int'l Conference on Autonomous Agents and Multiagent Systems, pp. 1065–1072 (2010)
22. Pearl, J.: Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley (1984)
23. Stentz, A.: The Focussed D* algorithm for real-time replanning. In: Proc. of Int'l Joint Conference in Artificial Intelligence, pp. 1652–1659 (1995)
24. Sturtevant, N.: Benchmarks for grid-based pathfinding. Transactions on Computational Intelligence and AI in Games **4**(2), 144–148 (2012)
25. Sun, X., Koenig, S., Yeoh, W.: Generalized Adaptive A*. In: Proc. of the Int'l Joint Conference on Autonomous Agents and Multiagent Systems, pp. 469–476 (2008)

26. Sun, X., Yeoh, W., Koenig, S.: Efficient incremental search for moving target search. In: Proc. of the Int'l Joint Conference on Artificial Intelligence, pp. 615–620 (2009)
27. Yap, P.K.Y., Burch, N., Holte, R.C., Schaeffer, J.: Any-Angle Path Planning for Computer Games. In: Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, pp. 201–207 (2011)