Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence

Reusing Previously Found A* Paths for Fast Goal-Directed Navigation in Dynamic Terrain

Carlos Hernández
Depto. de Ingeniería Informática
Univ. Católica de la Ssma. Concepción
Concepción, Chile

Roberto Asín
Depto. de Ingeniería Informática
Univ. Católica de la Ssma. Concepción
Concepción, Chile

Jorge A. Baier
Depto. de Ciencia de la Computación
Pontificia Universidad Católica de Chile
Santiago, Chile

Abstract

Generalized Adaptive A* (GAA*) is an incremental algorithm that replans using A* when solving goal-directed navigation problems in dynamic terrain. Immediately after each A* search, it runs an efficient procedure that updates the heuristic values of states that were just expanded by A*, making them more informed. Those updates allow GAA* to speed up subsequent A* searches. Being based on A*, it is simple to describe and communicate; however, it is outperformed by other incremental algorithms like the state-of-the-art D*Lite algorithm at goal-directed navigation. In this paper we show how GAA* can be modified to exploit more information from a previous search in addition to the updated heuristic function. Specifically, we show how GAA* can be modified to utilize the paths found by a previous A* search. Our algorithm—Multipath Generalized Adaptive A* (MPGAA*)—has the same theoretical properties of GAA* and differs from it by only a few lines of pseudocode. Arguably, MPGAA* is simpler to understand than D*Lite. We evaluate MPGAA* over various realistic dynamic terrain settings, and observed that it generally outperforms the state-of-the-art algorithm D*Lite in scenarios resembling outdoor and indoor navigation.

Introduction

Goal-directed navigation, the problem of leading an autonomous agent from an initial location to a goal location, is an important problem in AI with recognized applications in robotics. In the most general of its variants—under the so-called *dynamic terrain*—one assumes the environment (i.e., the map) may change while the agent is moving.

Incremental heuristic search (IHS) algorithms (e.g., Koenig et al. 2004) are replanning heuristic-search algorithms amenable for goal-directed navigation. Their main characteristic is that they reuse information from previous searches to speed up subsequent searches. Among the most effective algorithms for goal-directed navigation in dynamic terrain are D* (Stentz 1995), and the simpler, algorithmically equivalent version of D*, D*Lite (Koenig and Likhachev 2002). Both of them search backwards from the goal state to the current state, saving information that allows fast replanning whenever the environment changes.

Arguably, D* and D* Lite are not simple algorithms. Even though they are based on the well-known A* (Hart, Nilsson, and Raphael 1968) algorithm, they modify A* significantly, and as such are arguably harder to understand, communicate, and implement than A*. Alternative IHS algorithms directly based on A* for goal-directed navigation in dynamic terrain do exist: Generalized Adaptive A* (Sun, Koenig, and Yeoh 2008) (GAA*) is one of them. It is arguably much simpler than D* Lite since it does not modify the core A* search but unfortunately is outperformed by D* Lite at goal-directed navigation in dynamic terrain (Sun, Koenig, and Yeoh 2008).

Being an IHS algorithm, GAA* reuses information that is computed by A* in a previous search. Specifically, after an A* search is completed, it updates the heuristic function of the expanded states. This information has proven empirically to speed up subsequent searches but it is clearly insufficient to allow GAA* to outperform D* Lite.

In this paper we propose Multipath Generalized Adaptive A*, a simple but powerful extension of GAA* that exploits more information from previous A* searches than GAA* does. In particular, MPGAA* reuses paths towards the goal which were computed by previous calls to A*. As soon as a state that belonged to any of those paths is considered for expansion, MPGAA* runs a linear algorithm that allows it to verify whether or not such a path allows it to build an optimal complete path to the goal state. If the test is passed, search is stopped early, allowing it to save significant time.

We prove that MPGAA* has the same properties of GAA*; the most important being that the paths found are optimal with respect to the agent's current knowledge. In addition, our experimental evaluation shows that MPGAA* is superior to D* Lite in indoor navigation settings and usually faster than D* Lite in scenarios comparable to outdoor navigation.

Reusing the end of a previously found A* path is not a new idea; indeed, it could be traced back down to Bi-Directional Search (Pohl 1971). In IHS this idea has been used before in other algorithms, specifically in Tree Adaptive A* (Hernández et al. 2014) and Multipath Adaptive A* (Hernández, Baier, and Asín 2014). While none of these algorithms can be used in dynamic terrain settings (indeed, they can only be used in a priori partially known terrain), MPGAA* can be seen as an application of the same prin-

ciple underlying Multipath Adaptive A*. As such, the specific technical contribution of this paper is to (1) show how to exactly integrate these ideas into GAA*, and (2) an experimental evaluation showing that MPGAA* is a superior algorithm for goal-directed navigation in dynamic terrain.

The rest of the paper is organized as follows. We start off formally describing the problem of goal-directed navigation in dynamic terrain and describing GAA*. We then describe our algorithm and perform a theoretical analysis. We continue presenting our experimental evaluation in random, indoor- and outdoor-like terrain. We finish by discussing some related work and conclusions.

Path-Planning in Dynamic Terrain

A path-planning problem in dynamic terrain is a tuple $P = (G, \gamma, s_{start}, s_{goal})$, where $G = (V, E)$ is an undirected graph where V is a set of states, E is a set of arcs, s_{start}, the start state, and s_{goal}, the goal state, are both in V, and $\gamma = c_0 c_1 \ldots$ is an infinite sequence of cost functions, each of which maps each element in E to a non-negative value in $\mathbb{R} \cup \{\infty\}$. Sequence γ is used to model the fact that the terrain changes as the agent moves, and the cost value ∞ is used to represent the fact that states may, in practice, become disconnected from its neighbors in G.

An agent acting in a dynamic terrain follows a *path* over G, which is a sequence of states $s_0 s_1 \ldots s_n$ such that for all $i \in \{1, \ldots, n\}$, it holds that $(s_{i-1}, s_i) \in E$. The *cost of a path* $s_0 \ldots s_n$ over G is $\sum_{i=0}^{n-1} c_i(s_i, s_{i+1})$.[1] A path $\sigma = s_0 \ldots s_n$ is a *solution* to path-planning problem P if $s_0 = s_{start}$, $s_n = s_{goal}$, and the cost of σ is finite. Observe that even if there exists a path between s_{start} and s_{goal} over G it cannot be guaranteed that there exists a solution to P, because this depends on γ.

Solving a Path-Planning Problem

We assume that an agent solving a problem P has partial knowledge about the terrain. This is represented by the fact that, during execution, the agent knows the search graph G, but does not have access to the sequence of cost functions γ. Rather, it believes that the actual cost function is c. As the agent moves through the terrain it may observe the true cost of a subset of the arcs in E and update its cost function c accordingly. The exact subset of arcs that can be observed depends on an additional parameter, called the *visibility range*, which intuitively defines a set of arcs whose cost can be observed by the agent depending on its position. Most realistic dynamic terrain settings involve a restricted visibility range, which includes only arcs that connect states whose distance from the current state is bounded. Realistic settings also assume that successive cost functions in γ differ in the cost assigned to a few arcs only. Finally, if one assumes the terrain is *known* a priori, then $c = c_0$; otherwise if the terrain is *partially known* c is equal to c_0 for some of the arcs of the search space whereas for other arcs some standard assumption is made about its cost (in 8-connected grids, for example, it is assumed that such cost is 1 for vertical and horizontal moves and $\sqrt{2}$ for diagonal moves). This assumption is usually referred to as the freespace assumption (Zelinsky 1992).

Heuristic-search approaches to solving path-planning problems in dynamic terrain use a heuristic function h. Given a graph $G = (V, E)$ and a goal state s_{goal}, $h : V \to \mathbb{R}$ is such that $h(s)$ is a non-negative estimate of the cost of a path from s to s_{goal}. Function h is *admissible* iff for every state s, $h(s)$ does not overestimate the cost of any path from s to s_{goal}, with respect to the current cost function c. Furthermore, we say h is *consistent* if for every $(s, t) \in E$ it holds that $h(s) \leq c(s, t) + h(t)$, and $h(s_{goal}) = 0$. It is easy to prove that consistency implies admissibility. Finally, we say that $h(s)$ is *perfect* iff it is equal to the cost of an optimal path from s to s_{goal}.

Generalized Adaptive A*

A straightforward way of using the well-known A* algorithm for goal-directed navigation in dynamic terrain is as follows: (1) run a forward A* from the current state to the goal, (2) follow the path returned until the goal has been reached or until the cost function of the graph has changed, in which case go back to (1). This algorithm has been referred to as *Repeated A** (Koenig and Likhachev 2002).

Generalized Adaptive A* (GAA*) (Sun, Koenig, and Yeoh 2008) can be understood as a very simple extension of Repeated A*. It differs from it in two aspects. First, after each A* search it *updates* the h-values of all states in A*'s closed list, making them more informed. The way GAA* does this has a desirable property: if the heuristic was consistent, it remains consistent and, consequently, optimality is guaranteed in subsequent searches. The second aspect by which GAA* differs from Repeated A* is that it may update the h-values of some states of the search space when it observes that the cost of an arc has *decreased*. This update is needed to reestablish the consistency of the heuristic, which can be broken when arc costs decrease.

The pseudocode of GAA* is shown as Algorithm 1. By removing Lines 68–69, which run the heuristic updates that aim at making h more informed, and Lines 39–40, which call the procedure for reestablishing consistency, GAA* would become Repeated A*.

Now we describe further details of the pseudocode. We assume $H(s, s_{goal})$ (used in the initialization; Lines 59–62) is an admissible estimate of the cost of a path between the s and s_{goal} with respect to the initial cost function c. Once a path is found by A* function `BuildPath` sets the $next(s)$ pointer for each state s along the path found by A*. Variable $search(s)$ returns the number of the last search in which s was generated; it is equal to 0 if s has never been generated. Before every A* search, the variable $counter$ is incremented; this variable is used by procedure `InitializeState` to set $g(s)$ to infinity when s has not been generated during the current search. We point out that Lines 38 and 74, which set the $next$ pointer

[1] Note that here we assume that the cost structure of the graph can may only change with an agent movement. This is a general assumption if we allow no-ops, which can be modeled via zero-cost self-loops in G.

to null in different situations, are not strictly necessary in a GAA* implementation, and thus removing them will not change its behavior. Finally, we remark that procedure `ReestablishConsistency` is a simple modification of Dijkstra's algorithm, which upon termination guarantees that $h(s) = \min_{s' \in Succ(s)} c(s, s') + h(s')$, for each s that was ever added to the priority queue Q.

Properties of Generalized Adaptive A*

Sun, Koenig, and Yeoh (2008) proved the following property of Generalized Adaptive A*.

Theorem 1 (Consistency Preservation) *If h is consistent with respect to cost function c, then it remains consistent after the heuristic update of Lines 68–69, and it remains consistent after a call the call to observe function in Line 76.*

This results allows to prove that Generalized Adaptive A* is *optimal*, in the following sense.

Theorem 2 (Optimality) *The movement performed in Line 75 lies on an optimal path to s_{goal} over graph G with respect to (current) cost function c.*

In other words, optimality here intuitively refers to the fact that the agent always does the best it can, given its current knowledge about the terrain.

Unfortunately, given that the terrain is dynamic, it is not possible to prove that the algorithm terminates finding a solution even if every cost function in γ is such that a finite-cost path between the current state and the goal exists. Indeed, since the agent always follows an optimal path to the goal, it is not hard to see that one can design a sequence of cost functions that would make GAA* enter an infinite loop. It is simple to prove however that the algorithm produces a correct solution:

Theorem 3 (Soundness) *If at each iteration the visibility range allows observing the true cost of all arcs out of s_{start} and the algorithm terminates with $s_{start} = s_{goal}$, then the sequence of movements produced by GAA* defines a solution to the path-finding problem.*

Finally, the following property establishes that GAA*'s update rule is yields a perfect heuristic for states along the path to the goal.

Proposition 1 *Let σ denote the path found by A* after the call in Line 65. After the update in Lines 68–69, $h(s)$ is perfect for every s in σ.*

Multipath Generalized Adaptive A*

The main piece of information that GAA* reuses from a previous search to speed up the current search is the h-values that were updated previously. While this has proven experimentally to pay off, there is more information that a previous A* computes and that is not exploited by GAA*. Within it, are the paths that have been previously computed and that reach the goal optimally (under some version of the cost function c).

The main idea underlying our algorithm, Multipath Generalized Adaptive A* (MPGAA*), is to reuse such paths. MPGAA* will stop an A* search early if it selected for expansion a state belonging to a previously found path which in addition provides an optimality guarantee.

Algorithm 1: Generalized Adaptive A* Pseudo-Code

```
1  procedure InitializeState(s)
2      if search(s) ≠ counter then
3          g(s) ← ∞
4      search(s) ← counter
5  function GoalCondition(s)
6      return s = s_goal
7  procedure A*(s_init)
8      InitializeState(s_init)
9      parent(s_init) ← null
10     g(s_init) ← 0
11     Open ← ∅
12     insert s_init into Open with f-value g(s_init) + h(s_init)
13     Closed ← ∅
14     while Open ≠ ∅ do
15         remove a state s from Open with the smallest f-value g(s) + h(s)
16         if GoalCondition(s) then
17             return s
18         insert s into Closed
19         for each s' ∈ succ(s) do
20             InitializeState(s')
21             if g(s') > g(s) + c(s, s') then
22                 g(s') ← g(s) + c(s, s')
23                 parent(s') ← s
24                 if s' is in Open then
25                     set priority of s' in Open to g(s') + h(s')
26                 else
27                     insert s' into Open with priority g(s') + h(s')
28     return null
29 procedure BuildPath(s)
30     while s ≠ s_start do
31         next(parent(s)) ← s
32         s ← parent(s)
33 procedure Observe(s)
34     T ← arcs in the range of visibility from s
35     for each (t, t') in T do
36         update c(t, t')
37         if c(t, t') just increased then
38             next(t) ← null
39     if there is an arc (t, t') in T whose cost decreased then
40         ReestablishConsitency()
41 procedure InsertState(s, s', Q)
42     if h(s) > c(s, s') + h(s') then
43         h(s) ← c(s, s') + h(s')
44         if s in Q then
45             Update priority of s in Q to h(s)
46         else
47             Insert s into Q with priority h(s)
48 procedure ReestablishConsitency()
49     Q ← empty priority queue
50     for each (s, s') such that c(s, s') decreased do
51         InsertState(s, s', Q)
52     while Q is not empty do
53         Extract state s' with lowest h-value in Q
54         for each s such that s' ∈ Succ(s) do
55             InsertState(s, s', Q)
56 procedure main()
57     counter ← 0
58     Observe(s_start)
59     for each state s ∈ S do
60         search(s) ← 0
61         h(s) ← H(s, s_goal)
62         next(s) ← null
63     while s_start ≠ s_goal do
64         counter ← counter + 1
65         s ← A*(s_start)
66         if s = null then
67             return "goal is not reachable"
68         for each s' ∈ Closed do
69             h(s') ← g(s) + h(s) − g(s') // heuristic update
70         BuildPath(s)
71         repeat
72             t ← s_start
73             s_start ← next(s_start)
74             next(t) ← null
75             Move agent to s_start
76             Observe(s_start)
77         until a change in c has been observed or s_start = s_goal
```

MPGAA* replaces three of GAA*'s procedures. Their pseudocode are shown in Algorithm 2. `GoalCondition` is the new procedure that is called by A* to decide when to stop search. It will traverse the path from the argument state s to the goal, returning true iff these two conditions hold: (1) the goal is reachable from s via the $next(s)$ pointers, and (2) the heuristic on such a path is still perfect for the current cost function c.

In addition, MPGAA* modifies slightly GAA*'s procedure to reestablish consistency to keep track of those states which *support* the h-values of other states. The main objective of this is to extend those paths previously found by A* with states that will satisfy property (2) above. As paths to the goal state are extended, the chances of stopping search earlier increase. Specifically, we say that s' supports s if $h(s) = c(s, s') + h(s')$. Our version of `ReestablishConsistency`, in Line 20, will proactively set the $next(s)$ to its supporter state s' if s' may be in a path to a goal state.

Algorithm 2: MPGAA*'s Procedures

```
1  function GoalCondition(s)
2      while next(s) ≠ null and h(s) = h(next(s)) + c(s, next(s)) do
3          s ← next(s)
4      return s_goal = s
5  procedure InsertState(s, s', Q)
6      if h(s) > c(s, s') + h(s') then
7          h(s) ← c(s, s') + h(s')
8          next(s) ← null
9          support(s) ← s'
10         if s in Q then
11             Update priority of s in Q to h(s)
12         else
13             Insert s into Q with priority h(s)
14 procedure ReestablishConsitency()
15     Q ← empty priority queue
16     for each (s, s') such that c(s, s') decreased do
17         InsertState(s, s', Q)
18     while Q is not empty do
19         Extract state s' with lowest h-value in Q
20         if next(support(s')) ≠ null then next(s') ← support(s')
21         for each s such that s' ∈ Succ(s) do
22             InsertState(s, s', Q)
```

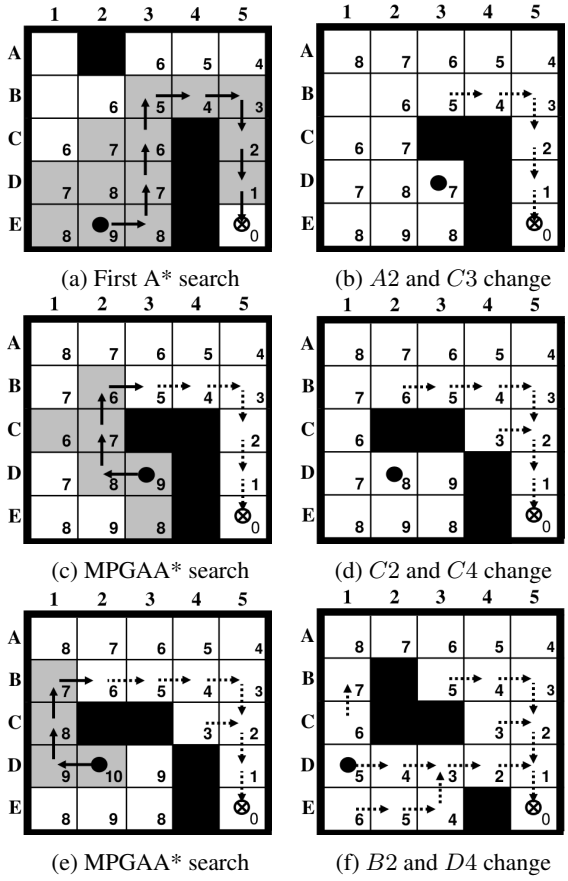
Properties of Multi-Path GAA*

Assuming h is consistent, we prove that the path returned by A* is optimal.

Theorem 4 (Optimality of MPGAA*) *If h is consistent, the movement performed in Line 75 lies on an optimal path to s_{goal} over graph G with respect to (current) cost function c.*

Proof: Assume the call to A* stops when a state $s \neq s_{goal}$ is expanded. Let σ denote the path connecting s to s_{goal} via the $next$ pointers. Since `GoalCondition` returns true $h(s) = c(\sigma)$, and thus $f(s) = c(\pi)$, where π is an actual path connecting s_{start} and s_{goal} that passes through s. Since h is consistent, we have that $f(s) \leq c^*$, where c^* is the optimal cost. We hence conclude that the cost of π is optimal. ■

We now can prove that consistency is preserved by both updates.



(a) First A* search (b) $A2$ and $C3$ change (c) MPGAA* search (d) $C2$ and $C4$ change (e) MPGAA* search (f) $B2$ and $D4$ change

Figure 1: MPGAA* example

Theorem 5 (Consistency Preservation) *If h is consistent with respect to cost function c, then it remains consistent after the heuristic update of Lines 68–69, and it remains consistent after the call to observe function in Line 76.*

Proof: Follows from Theorem 4 and reuses the proof by Sun, Koenig, and Yeoh (2008). ■

Finally, the algorithm is also sound.

Theorem 6 (Soundness) *If, at each iteration, the visibility range allows observing the true cost of all arcs out of s_{start} and the algorithm terminates with $s_{start} = s_{goal}$, then the sequence of movements produced by GAA* defines a solution to the path-finding problem.*

An Example

Figure 1 shows an example run of MPGAA*. The terrain is represented as a 4-neighbour grid. Visibility range is assumed as infinite, for simplicity. Black cells are blocked and the remaining are unblocked. The black dot (●) shows the position of the agent while the ⊗ shows the goal state. The initial heuristic is the Manhattan distance. Numbers in the cells are the current heuristic values. After each search, the agent follows the path until the goal is reached or until the terrain changes, triggering a new planning episode. Solid arrows show the path computed by A*, while dashed arrows represent $next$ pointers defining a previously found A* path.

Algorithm	Random	Room	WC III
Repeated A*	38.24	2.83	13.42
GAA*	28.83	2.72	4.72
D*Lite	8.04	1.95	1.42
MPGAA*	4.90	0.47	1.41
% runs MPGAA* faster D*Lite	70%	92%	62%

Table 1: Mean times (in ms) for all four algorithms over Random, room and Warcraft 3 maps.

Grey cells are those expanded by A* in the corresponding planning episode.

Figure 1a illustrates the result of the first A* search, followed by the heuristic update of the cells in Closed. Then we assume the agent follows the path and that right after reaching `D3`, cell `A2` becomes unblocked and cell `C3` becomes blocked. After the changes are reflected in the current cost function, `ReestablishConsitency` is run and the $next$ pointers are updated (Figure 1c). Figure 1d shows the second search, which stops early because the function `GoalCondition` returns `true` when it is called with cell `B3`. Then, the agent follows the path. When it reaches `D2`, cells `C4` and `C2` switch their blockage status. The procedure `ReestablishConsitency` is performed and the $next$ pointers are updated (Figure 1e). Figure 1f shows the third search. Again, the search stops early because the function `GoalCondition` returns `true` when called with cell `B2`. The agent moves and, when it is visiting `D1`, cell `D4` gets unblocked and `B2` blocked. The procedure `ReestablishConsitency` is performed and the $next$ pointers are updated (Figure 1f). The fourth search stops even earlier (without the need for a single cell expansion) since the function `GoalCondition` returns `true` when called from the agent's current cell.

Experimental Results

In this section we present our experimental results comparing MPGAA* against Repeated A*, GAA*, and an optimized version D*Lite (Koenig and Likhachev 2005) in an application widely used in the literature. Specifically, we compare all four algorithms in path-planning in dynamic eight-neighbour grids. We use eight-neighbor grids since they are often preferred in practice (Bulitko et al. 2011; Koenig and Likhachev 2005). The cost of orthogonal movements is 1 and the cost of diagonal moves is $\sqrt{2}$. The user-given heuristic are the octile distances. All the experiments were run on an Intel(R) i7-2600 CPU at 3.40GHz machine with 8 GB of RAM, running Linux.

MPGAA* was implemented in ANSI C, using a binary heap as a priority queue for the Open list. The heap implementation is very similar to the heap used by D*Lite. The C source code for D*Lite was provided by their authors.

For all runs, the grid changes each time the agent performs k moves. We used $2, 4, 8, 16, 32$ and 64 as values for k. The changes to the map are controlled by the *change rate* parameter cr. In each change phase $\frac{cr}{2}$ randomly chosen unblocked cells become blocked and equal number of the originally blocked cells are randomly chosen and set as unblocked. We used change rates of $1\%, 5\%, 10\%, 15\%, 20\%$ and 30%. While changes can occur in any part of the search space, the agent can only observe the cost of the arcs up to depth n from the current position. This defines its *visibility range*. For these experiments, we set n equal to k.

We considered the following evaluation settings.

Setting 1 Random maps, which are usually considered when evaluating algorithms for goal-directed navigation (e.g., Koenig and Likhachev 2002). We used a setting similar to that used by Aine and Likhachev (2013): i.e., 1000×1000 grids with 10% obstacle ratio. Each obstacle is modeled as a cell with no incoming or outgoing edges. We generated 100 instances with start and goal cells chosen randomly.

		Change rate					
	k	1%	5%	10%	15%	20%	30%
A*	2	97.03	99.29	102.26	127.20	130.04	133.35
	4	49.59	50.21	51.29	62.93	65.20	66.31
	8	25.16	25.59	26.24	32.20	32.89	33.76
	16	13.04	13.38	13.66	16.56	16.93	17.11
	32	7.12	7.28	7.39	8.86	8.81	8.90
	64	4.20	4.23	4.33	4.82	4.67	4.76
GAA*	2	73.03	71.22	71.50	90.34	93.72	94.64
	4	38.37	37.23	37.58	47.05	48.04	49.34
	8	20.54	20.10	19.89	25.13	25.65	26.16
	16	11.15	11.12	11.18	13.91	14.33	14.36
	32	6.33	6.58	6.76	8.17	8.24	8.30
	64	3.92	4.14	4.41	5.06	5.03	5.32
D*Lite	2	7.94	8.29	8.55	9.22	9.14	9.25
	4	7.68	8.02	8.14	8.75	8.81	8.81
	8	7.38	7.67	7.78	8.46	8.45	8.50
	16	7.23	7.47	7.62	8.26	8.27	8.29
	32	7.11	7.33	7.44	8.06	7.94	8.22
	64	7.03	7.15	7.35	7.98	7.92	8.06
MPGAA*	2	4.88	6.00	6.18	8.68	9.62	10.69
	4	3.56	4.84	5.44	6.98	7.49	8.35
	8	2.76	4.09	4.62	5.97	6.48	6.70
	16	2.19	3.46	3.95	4.96	5.15	5.70
	32	1.77	2.70	3.24	3.97	4.12	4.61
	64	1.58	2.14	2.74	3.29	3.46	3.92
Percentage of runs in which MPGAA* is faster than D*Lite	2	68%	62%	65%	58%	51%	49%
	4	78%	71%	68%	61%	59%	56%
	8	80%	74%	70%	64%	63%	60%
	16	81%	75%	71%	70%	66%	65%
	32	84%	78%	77%	72%	71%	69%
	64	87%	81%	77%	76%	75%	75%

Table 2: Mean times (in ms) for runs on 1000×1000 random scenarios for A*, GAA*, D*Lite and MPGAA*.

Setting 2 Room maps from N. Sturtevant's repository (Sturtevant 2012), which can be regarded as good simulation scenarios for indoor navigation. We removed 2% of the original blocked cells in order to guarantee solvable problems. Here, the agent knows the original map and a certain amount of randomly introduced blocked cells—we used a 5% obstacle ratio. The blocked cells of the original map (i.e., the walls) do not change their blockage status. We used the 40 Room maps of size 512×512. Again, we generated 100 random instances per map.

Setting 3 Warcraft III maps from N. Sturtevant's repository (Sturtevant 2012). Game maps can be regarded as good simulation scenarios for outdoor navigation. In this setting, the robot knows the original map and a certain amount of randomly introduced blocked cells—we used 5% obstacle

	k	Change rate 1%	5%	10%	15%	20%	30%
A*	2	8.26	8.18	8.18	8.15	8.20	8.21
	4	4.27	4.23	4.23	4.20	4.21	4.22
	8	2.24	2.22	2.21	2.20	2.21	2.22
	16	1.21	1.20	1.20	1.20	1.20	1.21
	32	0.70	0.70	0.70	0.70	0.70	0.70
	64	0.47	0.46	0.46	0.47	0.46	0.46
GAA*	2	7.96	7.64	7.61	7.53	7.57	7.52
	4	4.20	4.07	4.00	3.97	3.97	3.96
	8	2.24	2.20	2.18	2.16	2.15	2.15
	16	1.23	1.23	1.22	1.22	1.22	1.22
	32	0.72	0.72	0.72	0.73	0.74	0.76
	64	0.47	0.49	0.50	0.51	0.54	0.56
D*Lite	2	1.96	2.05	2.06	2.07	2.07	2.08
	4	1.89	1.96	1.99	2.01	2.01	2.03
	8	1.86	1.92	1.95	1.96	1.97	1.99
	16	1.84	1.89	1.91	1.93	1.94	1.96
	32	1.83	1.86	1.89	1.92	1.93	1.95
	64	1.85	1.87	1.89	1.91	1.92	1.93
MPGAA*	2	0.50	0.72	0.81	0.82	0.83	0.85
	4	0.37	0.53	0.59	0.63	0.64	0.65
	8	0.30	0.41	0.47	0.50	0.52	0.54
	16	0.27	0.33	0.38	0.42	0.44	0.47
	32	0.24	0.28	0.32	0.36	0.40	0.43
	64	0.24	0.28	0.32	0.33	0.37	0.41
Percentage of runs in which MPGAA* is faster than D*Lite	2	90%	86%	84%	85%	85%	84%
	4	94%	90%	90%	89%	89%	88%
	8	96%	93%	92%	92%	91%	91%
	16	97%	95%	94%	93%	93%	92%
	32	97%	96%	95%	94%	93%	93%
	64	98%	97%	95%	94%	93%	92%

Table 3: Mean times (in ms) for runs on room scenarios for A*, GAA*, D*Lite and MPGAA*.

ratio. The blocked cells of the original map do not change. We used 36 Warcraft III maps of size 512×512. We generated 100 random instances per map.

	k	Change rate 1%	5%	10%	15%	20%	30%
A*	2	39.62	40.28	40.15	39.17	39.18	39.16
	4	20.17	20.47	20.49	19.77	19.82	19.89
	8	10.40	10.46	10.51	10.20	10.23	10.29
	16	5.55	5.59	5.60	5.42	5.44	5.49
	32	3.13	3.13	3.13	3.03	3.05	3.05
	64	1.95	1.93	1.93	1.86	1.87	1.87
GAA*	2	12.26	12.06	12.04	11.78	11.82	11.79
	4	6.68	6.70	6.72	6.52	6.55	6.54
	8	3.93	3.94	3.96	3.85	3.88	3.88
	16	2.46	2.53	2.56	2.51	2.52	2.56
	32	1.75	1.81	1.84	1.83	1.85	1.89
	64	1.40	1.48	1.52	1.50	1.53	1.59
D*Lite	2	1.43	1.51	1.51	1.49	1.49	1.50
	4	1.42	1.48	1.47	1.43	1.45	1.46
	8	1.39	1.42	1.43	1.40	1.42	1.44
	16	1.39	1.40	1.42	1.38	1.40	1.42
	32	1.39	1.41	1.41	1.37	1.39	1.39
	64	1.39	1.39	1.40	1.37	1.38	1.39
MPGAA*	2	1.32	1.56	1.68	1.66	1.70	1.73
	4	1.24	1.43	1.53	1.52	1.56	1.60
	8	1.18	1.35	1.44	1.44	1.47	1.52
	16	1.15	1.29	1.38	1.38	1.42	1.48
	32	1.14	1.26	1.34	1.35	1.40	1.46
	64	1.12	1.25	1.33	1.33	1.38	1.45
Percentage of runs in which MPGAA* is faster than D*Lite	2	64%	59%	57%	56%	55%	54%
	4	66%	63%	60%	59%	58%	57%
	8	68%	63%	61%	61%	60%	58%
	16	69%	65%	63%	62%	61%	59%
	32	70%	67%	64%	63%	62%	60%
	64	70%	67%	65%	63%	62%	61%

Table 4: Mean times (in ms) for runs on warcraft3 scenarios for A*, GAA*, D*Lite and MPGAA*.

Table 1 shows average runtimes (in milliseconds) for the 3 settings over all the test cases (aggregating all k and cr values). The total number of runs were $3,600$, $144,000$ and $129,600$ for Random maps, Room maps and Warcraft III maps, respectively. The last row of the table reports the percentage of the test cases where MPGAA* is faster than D* Lite. We observe that MPGAA* is faster than D* Lite in most cases. Regarding average runtime, the results show that MPGAA* is faster than other algorithms. MPGAA* is between 6.0 and 9.5 times faster than Repeated A*, between 3.3 and 5.9 times faster than GAA* and between 1.01 and 4.1 times faster than D* Lite.

As can be seen in Tables 2, 3, and 4, for all algorithms, runtime increases as cr increases, and decreases when k increases. This is easily explained since smaller values of k involve more frequent changes and thus more replanning. This also happens for larger values of cr, since, for greater change rates, more changes occur resulting in more search effort when replanning.

In general, MPGAA* is the fastest algorithm, followed by D*Lite, GAA* and Repeated A*. D*Lite may outperform MPGAA* in certain cases. This may be more likely to happen for small values of k and for values of cr above 20%. For instance, in random maps, this happens with $k = 2$ and $cr = 30\%$, where D*Lite outperforms MPGAA* in slightly more than 50% of problems.

Memory Analysis Both the D* Lite and our A* variants use a data structure for the map which stores, for each cell, all the necessary information that is used and updated during search (i.e., heuristic value, parent pointer, etc). As such, the actual memory requirements are very similar for all algorithms, and they grow linearly on the size of the grid.

To have an idea of memory usage when it is not the case that all states are always stored a priori in memory, we recorded the number of *generated* cells by each algorithm in a 1000×1000 random map for the same visibility ranges and three change rates. Because there are multiple searches, if a cell is generated more than once, we only count it once. The number of generated cells is a good measure of memory usage since those are the cells that actually need to be stored in memory. We observed that, in all configurations, D* Lite generates more cells than MPAA*; specifically, between 2 and 3.8 more cells. As a conclusion, MPAA* also seems superior to D* Lite with respect to memory usage.

Summary and Perspectives

We presented Multipath Generalized Adaptive A* (MPGAA*), a simple but powerful extension of Generalized Adaptive A* that, in realistic goal-directed navigation settings, runs usually faster than D* Lite. We proved that MPGAA* satisfies the same properties of GAA*; most importantly, that the paths computed are optimal with respect to the agent's knowledge. Even though the idea underlying MPGAA* is the same used in a recently proposed algorithm, Multipath Adaptive A* (MPAA*), our algorithm can be used in dynamic terrain scenarios, whereas MPAA* cannot. In addition, MPGAA* extends GAA*'s consistency-preserving procedure to further extend paths that can be

reused in future search. This feature is not present in MPAA*.

Among the most interesting features of MPGAA* is that, being based on A*, it is significantly simpler to understand, communicate, and implement than D* or D* Lite. It is also arguably easier to extend. This is interesting since extensions of D*—for example Field D*, for any-angle path planning (Ferguson and Stentz 2006), and Anytime Dynamic D* (Likhachev et al. 2005)—set the state-of-the-art in other relevant areas. Our research calls for the study of whether or not extensions of MPGAA* could also provide simpler and faster search algorithms in those settings too.

It is also conceivable to extend MPGAA* to compute suboptimal paths by using weighted A* rather than A*. This has the potential of reducing search times, and seems relevant in the light of a recent version of D* Lite, Truncated D* Lite (Aine and Likhachev 2013), which also aims at reducing search times by sacrificing optimality.

References

Aine, S., and Likhachev, M. 2013. Truncated incremental search: Faster replanning by exploiting suboptimality. In *Proceedings of the 27th AAAI Conference on Artificial Intelligence (AAAI)*.

Bulitko, V.; Björnsson, Y.; Sturtevant, N.; and Lawrence, R. 2011. *Real-time Heuristic Search for Game Pathfinding*. Applied Research in Artificial Intelligence for Computer Games. Springer.

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

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

Hernández, C.; Baier, J. A.; and Asín, R. 2014. Making A* Run Faster than D*-Lite for Path-Planning in Partially Known Terrain. In *Proceedings of the 24th International Conference on Automated Planning and Scheduling (ICAPS)*.

Hernández, C.; Uras, T.; Koenig, S.; Baier, J. A.; Sun, X.; and Meseguer, P. 2014. Reusing cost-minimal paths for goal-directed navigation in partially known terrains. *Autonomous Agents and Multi-Agent Systems* 1–46.

Koenig, S., and Likhachev, M. 2002. D* lite. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI)*, 476–483.

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

Koenig, S.; Likhachev, M.; Liu, Y.; and Furcy, D. 2004. Incremental heuristic search in ai. *AI Magazine* 25(2):99.

Likhachev, M.; Ferguson, D. I.; Gordon, G. J.; Stentz, A.; and Thrun, S. 2005. Anytime dynamic A*: An anytime, replanning algorithm. In *Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS)*, 262–271.

Pohl, I. 1971. Bi-directional search. In *Machine Intelligence 6*. 127–140.

Stentz, A. 1995. The focussed D* algorithm for real-time replanning. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI)*, 1652–1659.

Sturtevant, N. 2012. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games* 4(2):144 – 148.

Sun, X.; Koenig, S.; and Yeoh, W. 2008. Generalized Adaptive A*. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS)*, 469–476.

Zelinsky, A. 1992. A mobile robot exploration algorithm. *IEEE Transactions on Robotics and Automation* 8(6):707–717.