Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling

Making A* Run Faster than D*-Lite for Path-Planning in Partially Known Terrain

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

Jorge A. Baier
Depto. de Ciencia de la Computación
PUC Chile
Santiago, Chile

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

Abstract

Focused D* and D*-Lite are two popular incremental heuristic search algorithm amenable to goal-directed navigation in partially known terrain. Recently it has been shown that, unlike commonly believed, a version of A* is in many cases faster than D*-Lite, posing the question of whether or not there exist other variants of A* which could outperform algorithms in the D* family on most problems. In this paper we present Multipath Adaptive A* (MPAA*), a simple, easy-to-implement modification of Adaptive A* (AA*) that reuses paths found in previous searches to speed up subsequent searches, and that almost always outperforms D*Lite. We evaluate MPAA* against D*-Lite on random maps and standard game, room, and maze maps, assuming partially known terrain. In environments comparable to indoor and outdoor navigation (room and game maps) MPAA* is 35% faster than D*Lite on average, while on random maps MPAA* is over 3 times faster than D*Lite. D*Lite is faster than MPAA* only in mazes; notwithstanding, we show that if a small percentage of obstacle cells in a maze are made traversable, MPAA* outperforms D*Lite. In addition, we prove MPAA* is optimal and that it finds a solution if one exists. We conclude that for most real-life goal-directed navigation applications MPAA* should be preferred to D*Lite.

Introduction

Focused D* (Stentz 1995), and D*Lite (Koenig and Likhachev 2005) are popular replanning algorithms amenable for path-planning in partially known terrain. D*Lite, the simplest to understand of the two, implements the same behavior as Focused D* and has a comparable performance (Koenig and Likhachev 2005). D*Lite extensions are used in deployed applications; to name a few, it has been used in the navigation algorithms for a Mars Rover (Ferguson and Stentz 2007), and in the DARPA Urban challenge competition (Likhachev et al. 2005).

The main motivation for using algorithms of the D* family over A* replanning (Hart, Nilsson, and Raphael 1968) is that re-using information from previous search speeds up subsequent searches significantly. Indeed Stentz (1995) reported orders-of-magnitude speedups over A*.

A recent evaluation by Hernández et al. (2012) has shown that a careful implementation of A* replanning, called *Forward Repeated A**, is in many cases faster than D*Lite. This naturally motivates the question of whether or not there exist versions of A* superior to D*-family algorithms. A positive answer to this question is interesting both from a practitioner's standpoint and for academic reasons, since A* is a more standard algorithm, and has been studied more extensively than the D*-family algorithms.

In this paper we present Multipath Adaptive A* (MPAA*), an extremely simple yet powerful extension of Adaptive A* (AA*) (Koenig and Likhachev 2006), which we show almost always outperforms an optimized version of D*Lite. The idea underlying MPAA* is to reuse paths found by previous A* searches when replanning is needed. As such, when a state that belongs to a previously found path is selected for expansion, MPAA* runs a quick procedure to check whether or not such a path is still optimal. In that case, the A* search is stopped early, allowing it to save significant computation compared to AA*. Like AA*, it uses a simple and efficient rule to update the heuristic values of states expanded. We prove MPAA* inherits all of AA*'s desirable properties: it is optimal and terminates if a solution exist.

We evaluate MPAA* against Repeated A*, Adaptive A*, and D*Lite over benchmarks traditionally used in the literature for evaluating these algorithms. We observe MPAA* outperforms D*Lite in almost all problems we tried. Our empirical evaluation suggests, actually, that the only navigation problems in which D*Lite is still competitive is in

problems in which there are *very few* optimal paths between the current position and the goal state. Those situations do not seem common in real-world navigation settings.

MPAA* is not the only extension of Adaptive A* that outperforms D*Lite. Tree Adaptive A* (Hernández et al. 2011), an algorithm that during runtime builds a tree of previously found paths, has also been shown to outperform D*Lite. Unlike MPAA*, the implementation of Tree AA* is more complicated, is harder to explain and requires specific data structures. We elaborate on this later in the paper.

The rest of the paper is organized as follows. We formally describe the task of goal-directed navigation in partially known terrain. Then we describe Adaptive A* in detail. We continue describing our algorithm and its properties. We then describe our experimental evaluation and related work. We finish with a summary.

Path-Planning in Partially Known Domains

Path-planning in partially known terrain is the problem of moving an agent from a position s_{start} to a position s_{goal} through the edges defined by an undirected graph $G = (V, E, c)$, where V is a set of cells, E is a set of arcs, and c is a cost function $c : E \to \mathbb{R}^+$. We assume there is a path between s_{start} and s_{goal} in G.

When the agent has partial knowledge of the terrain we make a freespace assumption (Zelinsky 1992) which is an optimistic assumption about the grid in which unknown cells are considered obstacle-free. Specifically, during execution the agent believes the search graph is $G_a = (V, E_a, c_a)$, where $E \subseteq E_a$, and $c(s,t) \geq c_a(s,t)$ for each $(s,t) \in E$. As the agent moves through the terrain it may discover that an arc in G_a is not traversable in the actual domain, in which case it updates the cost of such an arc to ∞. It may also discover that an arc cost increased, in which case it updates the cost to the correct value.

A heuristic function h for a path-planning problem (s_{start}, s_{goal}, G) is a non-negative function such that $h(s)$ estimates the cost of a path from s to s_{goal}. Function h is *admissible* if for every state s, $h(s)$ does not overestimate the cost of any path from s to s_{goal}. 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 $h(s_{goal}) = 0$. Consistency implies admissibility.

Adaptive A*

Adaptive A* (AA*) (Koenig and Likhachev 2006) is a replanning algorithm that uses A* to compute a complete solution each time one is needed. After a complete path to the goal is found, AA* updates the h-values of all states in A*'s closed list (Lines 48–49), making them more informed. Then it follows the arcs in $path[s_{start}]$, which denotes the path starting in s_{start} which follows the pointers $next(s)$ up to s_{goal}. As soon as the agent detects an increase in the cost of an arc in the path, it recomputes another path using A*.

Now we clarify some of the specifics of the pseudocode. We assume $H(s, s_{goal})$ (used in the initialization; Lines 39–42) is an admissible estimate of the cost of a path between the s and s_{goal}. Once a path is found by A* function `BuildPath` sets the $next(s)$ pointer for each state s along the path found by A*. Before every A* search, the variable $counter$ is incremented; such variable is used by procedure `InitializeState` to set $g(s)$ to infinity when appropriate. Finally, we point out that Lines 35 and 54, which set the $next$ pointer to null in different situations, are not strictly necessary in an AA* implementation, and thus removing them will not change its behavior.

Algorithm 1: Adaptive A* Pseudo-Code

```
1  procedure InitializeState(s)
2      if search(s) ≠ counter then
3          g(s) ← ∞
4      search(s) ← counter
5  procedure A*(s_init)
6      InitializeState(s_init)
7      parent(s_init) ← null
8      g(s_init) ← 0
9      Open ← ∅
10     insert s_init into Open with f-value g(s_init) + h(s_init)
11     Closed ← ∅
12     while Open ≠ ∅ do
13         remove a state s from Open with the smallest f-value g(s) + h(s)
14         if GoalCondition(s) then
15             return s
16         insert s into Closed
17         for each s' ∈ succ(s) do
18             InitializeState(s')
19             if g(s') > g(s) + c(s, s') then
20                 g(s') ← g(s) + c(s, s')
21                 parent(s') ← s
22                 if s' is in Open then
23                     set priority of s' in Open to g(s') + h(s')
24                 else
25                     insert s' into Open with priority g(s') + h(s')
26     return null
27 procedure BuildPath(s)
28     while s ≠ s_start do
29         next(parent(s)) ← s
30         s ← parent(s)
31 procedure Observe(s)
32     for each arc (t, t') in the range of visibility from s do
33         if cost of (t, t') has increased then
34             update c(t, t')
35             next(t) ← null
36 procedure main()
37     counter ← 0
38     Observe(s_start)
39     for each state s ∈ S do
40         search(s) ← 0
41         h(s) ← H(s, s_goal)
42         next(s) ← null
43     while s_start ≠ s_goal do
44         counter ← counter + 1
45         s ← A*(s_start)
46         if s = null then
47             return "goal is not reachable"
48         for each s' ∈ Closed do
49             h(s') ← g(s) + h(s) − g(s') /* heuristic update */
50         BuildPath(s)
51         while no action cost has just increased in path[s_start] do
52             t ← s_start
53             s_start ← next(s_start)
54             next(t) ← null
55             Move agent to s_start
56             Observe(s_start)
```

Algorithm 2: Adaptive A*'s Goal Condition

```
1 function GoalCondition(s)
2     return s = s_goal
```

Properties of Adaptive A* Koenig and Likhachev (2006) proved the following property of 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 48–49.*

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

Theorem 2 (Optimality) *Let G_a denote the search graph given by the algorithm's cost function c immediately after a movement is performed at Line 55. Then such a movement lies on an optimal path to s_{goal} over graph G_a.*

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.

Finally, it is simple to prove termination.

Theorem 3 (Termination) *Adaptive A* terminates.*

Finally, Forward Repeated A* (Hernández et al. 2012) is AA* without heuristic update (Lines 48–49).

Multipath Adaptive A*

One of the main characteristics of incremental heuristic search algorithms is that they exploit information gained in a search episode to speed up subsequent searches. As such, each time a search episode finishes, Adaptive A* updates the h values of many of the states of the search space, making them more informative. Indeed, the h-values of states in the path connecting s_{start} to s_{goal} are perfect after the update.

Heuristic update is however just one piece of information exploitable for later searches. An additional exploitable element is information contained in the A* tree. For example, each time an A* search returns, an optimal path—say, σ—to the goal has been found. Usually, even after discovering additional obstacles there exists a suffix of σ that still defines an optimal path to the goal.

MPAA*, exploits this fact by stopping an A* search as soon it selects a state s for expansion and the following conditions hold: (1) s belongs to a previously found path σ, and (2) the suffix of σ that starts in s is a provably optimal path from s to the s_{goal}. Checking condition (1) is actually very simple since already Adaptive A* sets the $next(s)$ in the `BuildPath` procedure, immediately after an A* finds a path to the goal. Condition (2), on the other hand, can be checked efficiently by using the fact that h remains consistent (if initially consistent).

MPAA* is obtained by simply replacing the `GoalCondition` function in Adaptive A*'s pseudocode by Algorithm 3. The reader may infer that a MPAA* implementation differs from an Adaptive A* implementation by only a handful of lines of code.

Properties of MPAA*

We prove that MPAA* has the same properties of Adaptive A* when h is initially consistent.

Algorithm 3: MPAA*'s Goal Condition

```
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
```

Algorithm	RunTime	Expansions	Percolations
Repeated A*	1.00	1.00	1.00
AA*	0.99	1.00	1.00
D*Lite	1.08	0.60	0.61
MPAA*	0.46	0.37	0.37

Table 1: Performance ratios wrt. to Repeated A* (Setting 1).

Theorem 4 (Consistency Preservation) *If h is consistent with respect to cost function c, then it remains consistent after the heuristic update of Lines 48–49.*

Proof: It is a trivial modification of the original proof provided by Koenig and Likhachev (2006). ■

We also obtain that MPAA* is optimal.

Theorem 5 (Optimality) *Let G_a denote the search graph given by the algorithm's cost function c immediately after a movement is performed at Line 55. Then such a movement lies on an optimal path to s_{goal} over graph G_a.*

Theorem 6 (Termination) *MPAA* terminates.*

Experimental Evaluation

The objective of our evaluation was to compare MPAA* against Repeated A*, AA*, Tree-AA*, and an optimized version D*Lite (Koenig and Likhachev 2005) in an application widely used in the literature. Specifically we compare the algorithms in path-planning in a priori unknown and partially known eight-neighbour grids. MPAA* was implemented in ANSI C, using a heap as a priority queue for the Open list. The heap implementation is very similar to the heap used by D*Lite. The source code for D*Lite was provided by their authors. We observed that the differences in performance between MPAA* and Tree-AA* are very consistent along all three settings and therefore we report results regarding this comparison at the end of this section.

We considered several evaluation settings. They follow.

Setting 1 We compared Repeated A*, Adaptive A*, D*Lite and MPAA* in the setting used by Koenig and Likhachev (2005): i.e., 129×129 grids with 40% obstacle ratio. Each obstacle is modeled as a cell with no incoming or outgoing edges. The initial state is $(12, 12)$ and the goal state is $(116, 116)$. The cost of cardinal and diagonal moves is 1 and the minimum of the absolute differences of the x and y coordinates of any two cells is used as heuristic. We generated 500 problems. Because Repeated A*'s results

	RunTime Ratio			Expansions Ratio			Percolations Ratio		
Dens.	400	600	800	400	600	800	400	600	800
15%	4.46	4.89	5.81	7.23	7.48	8.52	2.24	2.52	2.82
25%	3.93	4.18	4.56	5.32	5.61	6.12	2.61	2.59	2.76
35%	3.86	3.92	4.06	4.33	4.55	4.67	2.54	2.50	2.53
45%	3.98	3.86	3.93	3.46	3.53	3.71	2.64	2.36	2.42

Table 2: Performance ratios with respect to MPAA* (Setting 2).

Dens.	RoomMaps (Indoor Navigation) Runtime	Expansions	Percs.	Warcraft 3 Maps (Outdoor Navigation) Runtime	Expansions	Percs
5%	1.34	0.96	0.84	1.36	0.95	0.74
10%	1.41	0.91	0.85	1.37	1.00	0.75
15%	1.44	0.85	0.84	1.38	1.06	0.77
20%	1.43	0.78	0.82	1.46	1.17	0.84
Uknown	1.32	0.43	0.67	1.95	0.84	1.06

Table 3: Performance ratios with respect to MPAA* (Setting 3).

% Unblocks	Runtime	Expansions	Percs.
0%	0.22	0.06	0.10
1%	1.13	0.28	0.63
2%	2.11	0.64	1.33
4%	3.13	1.31	2.00
8%	3.81	2.31	2.34

Table 4: Performance ratios with respect to D*Lite (Setting 4).

were consistently worse than those of MPAA* (details below), we did not consider this algorithm for the remaining settings.

Setting 2 We compared D*Lite and MPAA* over larger random grids, for a broad range of obstacle ratios. We used random grids of size 400×400, 600×600, and 800×800 with 15%, 25%, 35%, and 45% obstacle ratio. For each grid size and obstacle ratio, we generate 500 problems, with start and goal cells chosen randomly. In this and the following settings, the cost for cardinal moves is 1, for diagonal moves is $\sqrt{2}$, and the octile distance is used as heuristic.

Setting 3 We compared D*Lite and MPAA* in partially known terrain. In this setting the robot knows the general map, but there are certain amount of unknown randomly introduced blocked cells—we used 5%, 10%, 15%, and 20% obstacle ratio. We evaluated over 40 Room Maps and 36 Warcraft III maps of size 512×512 from N. Sturtevant's repository (Sturtevant 2012). Room maps and Warcraft III maps can be regarded as good simulation scenarios for indoor and outdoor path-planning, respectively. Again, we generate 500 random problems. Also, for each map, we compare the algorithms considering an a priori unknown terrain, with no additional obstacles.

Setting 4 We compared D*Lite and MPAA* in a priori unknown Maze maps. We used 40 512×512 mazes from Sturtevant's repository. We considered a variation of the topology of these maps by removing at random 0%, 1%, 2%, 4%, and 8% of the blocked cells. For each configuration we generate 500 random problems.

For each problem we ran, we recorded the total runtime, the number of expansions and heap percolations, which is the number of array swaps required to perform heap insertions, deletions, and updates. Experiments were run on a 3.4GHz Intel Xeon machine with 8GB RAM running Linux. In all problems, the agent could observe the eight neighbors of the current cell and determine their blockage status.

Now we provide details of our results. Because AA* is consistently inferior to MPAA*, we only show performance indicators for Setting 1. On average AA* is about two times slower than MPAA*.

Table 1 (for Setting 1) shows MPAA* is about twice as fast as other algorithms, needing fewer expansions and heap percolations. Despite this, D*Lite is slightly slower; this phenomenon has been observed in previous evaluations (Hernández et al. 2012), and is due to the overhead D*Lite incurs when updating so-called rhs-values.

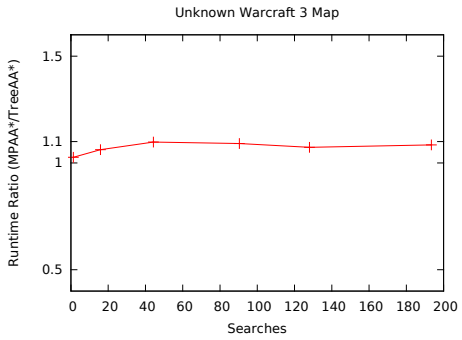

Figure 1: Tree-AA* 10% faster than MPAA* consistently.

Table 2 (for Setting 2) shows better performance indicators for MPAA* relative to D*Lite—up to 5.81 times faster in a 800×800 grid size and 15% of obstacles. Nevertheless, we observe the performance gap narrows down as the obstacles ratio increases. The opposite is observed when the size of the grid increases.

Table 3 (Setting 3) again shows that MPAA* is superior to D*Lite in terms of runtime, getting slightly better as the obstacle ratio increases in Room and Warcraft III maps. When the terrain is unknown, MPAA* is also faster than D*Lite in both indoor and outdoor scenarios.

Table 4 (Setting 4) shows D*Lite is 4.55 faster than MPAA* in mazes when no walls have been removed (0%). However, when only 1% of wall cells are removed, MPAA* begins to outperform D*Lite rather significantly. This suggests that D*Lite's performance niche seems to be problems in where there are few paths to the goal, an unusual situation in both indoor and outdoor navigation.

Finally, we compared MPAA* and Tree-AA* in all settings described above and found that the difference in performance is very consistent, with Tree-AA* being between 3–7% faster. As a reference, Figure 1 shows the time ratio (MPAA*/Tree-AA*) versus the number of searches required per problem averaged over 1,000 random problems over the map *divide-and-conquer* from the game Warcraft III. We observe that the performance difference is quite stable as the number of searches increase. Tree-AA*, unlike MPAA*, needs more memory per search since it maintains a table of paths.

Related Work

Both the idea of updating the heuristic and the idea of reusing a previously found path were first described by Holte et al. (1996) in the context of hierarchical search.

MPAA* is more closely related to Tree AA* (Hernández et al. 2011) which is also an extension of AA* that explicitly stores a tree of paths to the goal, where each branch corresponds to a path previously found by AA*. Such a tree is actively maintained during runtime and requires a specific data structure. The main advantage of MPAA* is its simplicity: it differs from AA* only in a handful of lines of code, and thus possibly simpler to extend. Moreover, it easier to

analyze in theory: e.g., optimality of MPAA* is extremely simple to prove, which is not the case for Tree AA*.

Summary and Future Work

We presented Multipath Adaptive A* (MPAA*), a simple extension of Adaptive A*. MPAA* is simple to implement, simple to analyze, and, we think, simple to extend too. We showed MPAA* outperforms D*Lite almost always for path-planning in partially known terrain in situations that are comparable to indoor and outdoor navigation (game and room maps). MPAA* outperforms D*Lite on random maps but not on maze maps. Indeed, D*Lite seems to excel only in problems in which very few optimal paths exist.

Future work considers studying whether or not state-of-the-art extensions of D* can also be outperformed by analogous extensions of MPAA*.

References

Ferguson, D., and Stentz, A. 2007. Field d*: An interpolation-based path planner and replanner. In *Robotics Research*. Springer. 239–253.

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.; Sun, X.; Koenig, S.; and Meseguer, P. 2011. Tree adaptive A*. In *Proceedings of the 10th International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS)*.

Hernández, C.; Baier, J. A.; Uras, T.; and Koenig, S. 2012. Position paper: Incremental search algorithms considered poorly understood. In *Proceedings of the 5th Symposium on Combinatorial Search (SoCS)*.

Holte, R. C.; Perez, M. B.; Zimmer, R. M.; and MacDonald, A. J. 1996. Hierarchical A*: Searching abstraction hierarchies efficiently. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI)*, 530–535.

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

Koenig, S., and Likhachev, M. 2006. A new principle for incremental heuristic search: Theoretical results. In *Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS)*, 402–405.

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.

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.

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