AD*-Cut: A Search-Tree Cutting Anytime Dynamic A* Algorithm

Maciej Przybylski
Institute of Automatic Control and Robotics
Warsaw University of Technology
ul. św. A. Boboli 8
02-525 Warsaw, Poland
maciej.przybylski@mchtr.pw.edu.pl

Abstract

This paper presents a new anytime incremental search algorithm, AD*-Cut. AD*-Cut is based on two algorithms, namely, Anytime Repairing A* (ARA*) and the novel incremental search algorithm, D* Extra Lite. D* Extra Lite reinitializes (cuts) entire search-tree branches that have been affected by changes in an environment, and D* Extra Lite appears to be quicker than the reinitialization during the search utilized by the popular incremental search algorithm, D* Lite. The search-tree branch cutting is a simple and robust technique that can be easily applied to ARA*. Consequently, AD*-Cut extends D* Extra Lite in the same manner, as the state-of-the-art Anytime D* (AD*) algorithm extends D* Lite. The benchmark results suggest that AD*-Cut is quicker and achieves shorter paths than AD* when used for path planning on 3D state-lattices (a 2D position with rotation).

Optimal motion planning and re-planning in a changeable environment related to the appearance and disappearance of obstacles is a common problem in robotics. Incremental heuristic algorithms, such as the state-of-the-art D* Lite (Koenig and Likhachev 2005b), help in this context. As they can reuse knowledge from previous searches, substantially less computation time is needed for re-planning.

In complex environments, where computation time is more important than optimality, anytime planning can be used. Anytime search algorithms aim to find any sub-optimal solution as quickly as possible and to improve it incrementally in the remaining time.

In addition, combining anytime and incremental search algorithms helps solve complex problems in a changeable environment. For example, Anytime D* (AD*) (Likhachev et al. 2005), that combines Anytime Repairing A* (ARA*) (Likhachev, Gordon, and Thrun 2004) and D* Lite algorithms, was used for autonomous car navigation (Likhachev and Ferguson 2009).

Recently, a new incremental search algorithm, D* Extra Lite (Przybylski and Putz 2017), has been developed and has outperformed D* Lite in most benchmark tests. Opposite to D* Lite that performs node-by-node reinitialization during a search, D* Extra Lite instantly reinitializes whole search-tree branches that have been affected by changes in an environment. Thus, D* Extra Lite does not make superfluous operations on an open-list except for those that are necessary to repair a frontier-gap created by branch cutting.

The present study proposes a new algorithm, AD*-Cut, that combines a search-tree cutting utilized by D* Extra Lite with the ARA* algorithm, resulting in a new anytime incremental search algorithm.

The paper is organized as follows. The related work section discusses the present state of knowledge in this area. Next, the main idea behind the search-tree branch cutting technique is outlined. Afterward, the AD*-Cut is presented and discussed. Lastly, the results of the tests are presented and discussed.

Related Work

The existing incremental search algorithms utilize three main approaches that aim to reuse information from previous searches: *a search-tree repairing and reuse* (Stentz 1995; Podsedkowski et al. 2001; Trovato and Dorst 2002; Koenig, Likhachev, and Furcy 2004; Koenig and Likhachev 2005b; Sun and Koenig 2007; Gochev, Safonova, and Likhachev 2014; Przybylski and Putz 2017), *heuristic improving* (Koenig and Likhachev 2005a; Sun, Koenig, and Yeoh 2008), and *heuristic improving combined with a reuse of previously found paths* (Hernández, Asín, and Baier 2015). As AD* and AD*-Cut algorithms are based on search-tree reuse, only this approach is discussed further.

Focussed D* (Stentz 1995) and D* Lite (Koenig and Likhachev 2005b) algorithms reinitialize nodes affected by edge-cost changes during the search, i.e., before the new node-cost is set, each node with an underestimated cost is reset and pushed to an open-list. Consequently, some nodes can be visited twice.

A different approach reinitializes the entire affected portion of the search-space by cutting search-tree branches. This approach is found in the work of (Podsedkowski et al. 2001) and in the Differential A* algorithm proposed by (Trovato and Dorst 2002). The idea of a reinitialization by search-tree branch cutting has been recently improved and used in D* Extra Lite (Przybylski and Putz 2017) that outperformed D* Lite in most of 2D 8-connected grid path planning problems.

Another category of algorithms based on search-tree reuse includes algorithms that restore a search-tree to some earlier stage, specifically, the stage when the search-tree did not cover states affected by changes in the environment, e.g., Fringe-Saving A* (Sun and Koenig 2007) and Tree-Restoring A* (Gochev, Safonova, and Likhachev 2013). This approach is similar to branch cutting, as it reinitializes the entire affected area; however, it typically prunes much larger parts of a search-space. A significant benefit of tree-restoring algorithms is that they do not require the computation of affected edges, because computing them can be more expensive than running searching algorithm from scratch.

As already mentioned, incremental search algorithms can be combined with anytime search methods. AD* (Likhachev et al. 2005) is the most recognizable contribution currently to address the problem of anytime incremental search. Other incremental search algorithms are Anytime Truncated D* (Aine and Likhachev 2013) and the most recent Anytime Tree-Restoring A* (ATRA*) (Gochev, Safonova, and Likhachev 2014) that extends Tree-Restoring A*. All three algorithms and the AD*-Cut algorithm presented in this work are based on the Anytime Repairing A* (ARA*) algorithm (Likhachev, Gordon, and Thrun 2004). Concerning presented findings, search-tree branch cutting has not been used yet for anytime incremental search; hence, the AD*-Cut algorithm presented in this paper is a novel approach.

Search-Tree Branch Cutting

Throughout the paper, the following notations are used. S represents a set of all feasible states of a robot. $Succ(s) \subset S$ represents a set of all states achievable from state s. $Pred(s) \subset S$ represents a set of all states from which state s can be achieved (used in a backward-search). The transition between two states is possible by execution of an applicable action $a_{s,s'} \in A$ from an action set A. For each action, a cost function is defined: $cost(a_{s,s'}) \equiv cost(s, s') : A \to \mathbb{R}^+$. Although within a single search episode action-cost is constant, it may change as observations are made between search episodes. A state-space can be represented as a directed graph; therefore, this paper uses the terms *state*, *node*, *action*, and *edge* interchangeably. Typically, graph-searching algorithms hold additional information for each node, such as $parent(s)$, which points to a node from which node s has been expanded (necessary to recover a path), and $g(s)$, which is a value that represents the cost from the starting node to s (or the cost from the goal-node to s in the case of a backward-search).

If any change is observed to affect the explored search-space, particularly, an edge-cost $e(s_1, s_2)$ has changed, a part of the visited search-space (a branch of the search-tree) has become inconsistent and must be re-explored. The inconsistent part of a search-tree can be defined as a branch of a search-tree that contains nodes supported by an edge $e(s_1, s_2)$. A node s_2 is supported by an edge $e(s_1, s_2)$ if the node s_1 is a parent of node s_2; furthermore, if a node s_2 is a parent of node s_3 and s_2 is supported by $e(s_1, s_2)$, then s_3 must also be supported by $e(s_1, s_2)$.

The g values of nodes that belong to an inconsistent search-tree branch are either too high or too low. In the situation in which the cost of an edge $e(s_1, s_2)$ decreases, as g values in the inconsistent part of the search-tree are higher

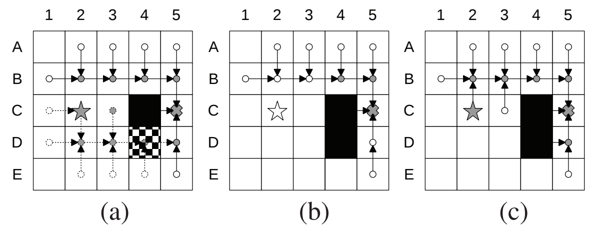

Figure 1: The robot (star), following the move from $C1$ to $C2$, observes cell $D4$ to be occupied (a). The entire branch supported by the edge $e(s_{C4}, s_{C5})$ must then be cut, and nodes that neighbor the cut branch are re-opened (b). A new optimal solution emerges that is on a different branch from the initial branch (c). White inner shape — open nodes, gray inner shape — closed nodes, arrows — parent node pointers, cross — goal node, black squares — obstacles, dashed line — affected edges.

than should be (nodes are over-consistent), it is sufficient to reopen the s_1 node and continue the search. In this situation, the affected branch of the search-tree cannot shrink (it can grow or stay unchanged).

If the cost of an edge $e(s_1, s_2)$ increases, all nodes in the branch of the search-tree supported by this edge become under-consistent, meaning its g values are lower than they should be. As the condition $g(s_2) > g(s_1) + cost(s_1, s_2)$ is not fulfilled, simple reopening of s_1 does not lead the algorithm to re-establish consistency. Therefore, before the algorithm begins a new search, the algorithm must make such nodes over-consistent by setting their g values to infinity or by marking them as unvisited. If the cost of the $e(s_1, s_2)$ edge increases, the affected search-tree branch may shrink or even disappear. Thus, the parent nodes of nodes that belong to the affected area may change radically, as shown in Fig. 1. After the branch is cut, there is a gap in the frontier fringe to be repaired. Therefore, all nodes neighboring with a cut node need to be reopened.

AD*-Cut

AD*-Cut (Alg. 1) is designed to perform a time-limited anytime search followed by a map update and reinitialization of the affected nodes (MAIN in Alg. 1). AD*-Cut combines the search-tree cutting technique used by D* Extra Lite with anytime repairing used by ARA*. procedures KEY, SOLUTIONFOUND, SEARCHSTEP, SEARCH and REEVALUATEOPEN (lines 1–47 in Alg. 1) to a large extent correspond to instructions of ARA* (cf. procedures *fvalue*, *ImprovePath* and *Main* — Likhachev, Gordon, and Thrun 2004). Furthermore, procedures REINITIALIZE, CUTBRANCH, and CUTBRANCHES (lines 48–89 in Alg. 1) correspond to procedures of D* Extra Lite (cf. Przybylski and Putz 2017). The remainder of this section explains the overall operation of the algorithm with a discussion of modifications specific to AD*-Cut.

The algorithm performs a backward-search. An anytime search loop (function SEARCH in Alg. 1) runs searches multiple times starting with $\epsilon = \epsilon_{init}$ and decreasing it by ϵ_{step}

Algorithm 1 AD*-Cut. Required parameters: $\epsilon_{init}, \epsilon_{step}$.

```
 1: function CalculateKey($s$)
 2:     return $g(s) + \epsilon \cdot h(s_{start}, s)$
 3: function SolutionFound()
 4:     $k_{start}$ = CalculateKey($s_{start}$)
 5:     $k_{top}$ = CalculateKey(topOpen())
 6:     return $visited(s_{start})$ AND $k_{start} \leq k_{top}$
 7: function Initialize()
 8:     $\epsilon = \epsilon_{init}$
 9:     $visited(s_{goal}) = true$
10:     $parent(s_{goal}) = NULL$
11:     $g(s_{goal}) = 0$
12:     pushOpen($s_{goal}$, CalculateKey($s_{goal}$))
13: function SearchStep()
14:     $s$ =topOpen()
15:     popOpen()
16:     $closed(s) = true$
17:     for all $s' \in Pred(s)$ do
18:         if NOT $visited(s')$ OR $g(s') > cost(s', s) + g(s)$ then
19:             $parent(s') = s$
20:             $g(s') = cost(s', s) + g(s)$
21:             if NOT $visited(s')$ then
22:                 $visited(s') = true$
23:             if $closed(s')$ AND $\epsilon > 1$ then
24:                 if NOT $inconsistent(s')$ then
25:                     $inconsistent(s') = true$
26:                     push($s'$, $INCONS$)
27:             else
28:                 pushOpen($s'$, CalculateKey($s'$))
29: function Search()
30:     $found = false$
31:     while open-list is not empty do
32:         if SolutionFound() then
33:             $found = true$
34:             if $\epsilon = 1$ then
35:                 return $found$
36:             $\epsilon = \epsilon - \epsilon_{step}$
37:             reevaluateOpen()
38:         if $found$ AND time elapsed then
39:             return $found$
40:         SearchStep()
41:     return $found$
42: function reevaluateOpen()
43:     $TO\_OPEN = INCONS \cup OPEN \cup SEEDS$
44:     $INCONS = OPEN = SEEDS = \emptyset$
45:     for all $s \in TO\_OPEN$ do
46:         if $visited(s)$ AND NOT $open(s)$ then
47:             pushOpen($s$, CalculateKey($s$))
48: function reinitialize()
49:     if any edge cost changed then
50:         cutBranches()
51:     if NOT $visited(s_{start})$ then $\epsilon = \epsilon_{init}$
52:     reevaluateOpen()
53: function cutBranch($s$)
54:     $visited(s) = false$
55:     $inconsistent(s) = false$
56:     $parent(s) = NULL$
57:     removeOpen($s$)
58:     for all $s' \in Succ(s)$ do
59:         if $visited(s')$ AND NOT $parent(s') = s$ then
60:             $SEEDS = SEEDS \cup s'$
61:     for all $s' \in Pred(s)$ do
62:         if $visited(s')$ AND $parent(s') = s$ then
63:             cutBranch($s'$)
64: function cutBranches()
65:     $reopen\_start = false$
66:     for all directed edges $(u, v)$ with changed cost do
67:         if $visited(u)$ AND $visited(v)$ then
68:             $c_{old} = cost(u, v)$
69:             update edge cost $cost(u, v)$
70:             if $c_{old} > cost(u, v)$ then
71:                 if $g(s_{start}) > g(v) + cost(u, v) + \epsilon \cdot h(s_{start}, u)$ then
72:                     $reopen\_start = true$
73:                 $SEEDS = SEEDS \cup v$
74:             else if $c_{old} < cost(u, v)$ then
75:                 if $parent(u) = v$ then
76:                     cutBranch($u$)
77:     if $reopen\_start = true$ AND $visited(s_{start})$ then
78:         $SEEDS = SEEDS \cup s_{start}$
79: function main()
80:     mapUpdate()
81:     initialize()
82:     while $s_{start} \neq s_{goal}$ do
83:         if NOT search() then
84:             return goal is not reachable
85:         $s_{start}$ =actionSelection($s_{start}$)
86:         mapUpdate()
87:         reinitialize()
88: function actionSelection($s_{start}$)
89:     return $\text{argmin}_{s' \in Succ(s_{start})}(cost(s_{start}, s') + g(s'))$
```

in subsequent searches, which is merely ARA*. The ϵ value is the factor by which the heuristic is inflated, making the algorithm greedier and possibly quicker (line 2, Alg. 1). Moreover, the algorithm does not allow for states reopening; instead, such states are placed in the list of inconsistent nodes (lines 23–26, Alg. 1), additionally speeding up the algorithm. After a solution with a given ϵ is found, ϵ is decreased (line 36, Alg. 1), and all nodes from $OPEN$ and $INCONS$ lists are re-opened with new keys (lines 37 and 42–47, Alg. 1). The search loop runs until the optimal solution is found (then $\epsilon = 1$) or granted time elapses but not before the first solution is found (line 38, Alg. 1). To this point, the only modification concerning ARA* is that each node holds an additional flag $visited(s)$ that is maintained to recognize nodes cut in the reinitialization (lines 21–22 and 46, Alg. 1).

In the reinitialization, search-tree branch cutting is executed if the cost of any visited edge has changed (lines 49–50, Alg. 1). For each edge with changed cost, the CutBranches procedure does one of two possible operations. If the cost of the $e(u, v)$ edge has decreased, the v node is added to the list of seeds to be reopened later (lines 70, 73 in Alg. 1). In the case of the edge-cost decreasing, there may be a shorter path. Therefore, to preserve optimality, the start node should be reopened. However, not in every case of edge-cost decrease does the start node need to be reopened. Assuming that $h(s_{start}, u)$ is admissible, for decreased $e(u, v)$ edge cost, the start node s_{start} requires reopening only if $g(s_{start}) > g(v) + cost(u, v) + \epsilon \cdot h(s_{start}, u)$ (lines 71–72 and 77–78, Alg. 1).

If the cost of the $e(u, v)$ edge has increased and node v is the parent of node u, the branch is cut starting from u (lines 74–76 in Alg. 1). The cutting operation marks node unvisited, resets its parent, and removes it from either $OPEN$

or $INCONS$, whichever it is placed (lines 54–57, Alg. 1). The CUTBRANCH() procedure is the recursive procedure that traverses throughout the branch, i.e., a next node to cut s' must be such a predecessor of a current node s that the s is the parent of s' (lines 61–63 in Alg. 1). Each successor node s', such that $s \neq parent(s')$ is placed in the list of seeds (lines 58–60 in Alg. 1). Although seeds are simply nodes to reopen, they cannot be merely pushed to the open-list as they might be cut later.

After the branches are cut, the start state can be off the search-tree. In such a case, the inflation factor ϵ is set to the initial value (line 51 in Alg. 1).

Following the CUTBRANCHES() procedure, the REINITIALIZE() procedure pushes to the open-list only these nodes from the $SEEDS$ list that remain visited and are not already open (lines 52 and 43–47 in Alg. 1). This operation repairs the frontier-gap made by branch cutting.

Discussion of the Algorithm

Regarding the reinitialization, AD*-Cut holds similar theoretical properties to D* Extra Lite on which it is based. The properties of D* Extra Lite confronted with D* Lite (on which AD* is based) have been discussed in (Przybylski and Putz 2017), and this discussion also applies to AD*-Cut and AD*. In a case of obstacles disappearance, both algorithms have similar complexity as they simply reopen overconsistent nodes. The discrepancies reveal in a case of obstacles appearance. In contrast to AD* and AD*-Cut, ATRA* reverts a search tree to some previous state in both cases. While AD*-Cut cuts branches precisely, ATRA* reverts entire search-tree. Therefore, typically, the search tree reinitialized by ATRA* is smaller than the search tree reinitialized by AD*-Cut.

The worst-case scenario for the discussed algorithms is when a large part of a search tree is affected by detected changes, for example, a dead end encountered close to the goal (close to the search tree root). In such a case, while AD*-Cut performs better if a long detour is necessary, AD* and ATRA* may outperform AD*-Cut if the detour is short. In general, as ATRA* does not require computation of states nor actions affected by map changes, the superiority of ATRA* is more pronounced with a growing number of changed map cells and a length of motion primitives (state-lattice arcs).

Experiments

The experiments compared AD*-Cut with AD* (Likhachev et al. 2005). Both algorithms used the same implementation of a heap and domain-specific functions such as successors and predecessors generation, action cost, heuristic (the Euclidean distance), and affected states computation. The implementation of AD* and domain-specific functions were obtained from the SBPL library (version 1.3.0)[1]. These tests were run on a 2.90-GHz machine with 8-GB RAM running 64-Bit Linux.

The benchmark problems and map sets (Fig. 2) *wc3*, *rooms*, *random10* (10% of map cells are randomly situated obstacles), and *mazes16* (mazes with the corridor width of 16 map cells) were obtained from the benchmark prepared by Sturtevant (2012). The map sets provide a variety of environmental changes that thoroughly test the algorithms. In addition, in the *rooms* and *random10* map set, observed obstacles do not affect the path significantly, while in the *mazes16* set, which includes many dead ends, even a minor change in the environment may introduce a long detour.

For each map set, 100 problems with distinct start-goal pairs were solved (across the set of problems, different maps were used). As suggested by Sturtevant, search results should be grouped concerning the problem length. Therefore, problems of similar length (ranging from 400 to 440 map-cells) were used.

The search-space was an (x,y,yaw)-state-lattice (a 2D position with rotation) with a total size of 512x512x16 states (16 possible orientations) and seven applicable actions (motion primitives) per state. The longest motion primitive was 8 map-cells long. The simulated robot was 10 map-cells wide and long, with the following exception for tests on the *rooms* and *random10* map-sets, in which the robot had a size of a single map cell because of the narrow passages of a single-cell width.

Each separate problem was solved based on MAIN function (lines 79–87 in Alg. 1); hence, a full navigation from the start to the goal state was performed. After each step of the robot (i.e., after an application of the next action), the map update, reinitialization, and searching were performed. The SEARCH function does multiple searches, decreasing ϵ each time ($\epsilon_{init} = 5$ and $\epsilon_{step} = 0.2$) until the allocated time elapses (1s). The algorithms could exceed the 1 second time limit when they were searching for the first solution.

To ensure comparable conditions for both algorithms, a robot moved along a pre-computed optimal path that is commonly assumed (Gochev, Safonova, and Likhachev 2014). (This suppresses ACTIONSELECTION function from line 85 in Alg. 1). The MAPUPDATE function simulates 360 degrees rangefinder working with a resolution of 0.33 degree and an observation range of 100 map cells.

During a main function run, the following parameters are logged: reinitialization time (excluding map update time), time until the first solution, overall search time, loop time (total time spent in a single iteration of the main loop, including map updates), search steps count, number of reinitialized under-consistent nodes, average ϵ, and average path cost.

The algorithms were tested in two settings: planning on partially-known maps (*random10* set with 5% of cells randomly shifted) in which obstacles may appear or disappear, and planning with a freespace assumption (*wc3*, *rooms*, and *mazes16* map sets) in which obstacles were only added.

The results presented in Table 1 were calculated as average values per single main loop after solving 100 problems for each map-set. As expected, compared with AD*, AD*-Cut spends more time on a reinitialization but less time on searching. On average, in most cases, AD*-Cut computed the first solution and improved paths and accomplished each main loop iteration quicker than AD*.

As AD*-Cut had shorter search times, it could perform

[1]The SBPL library http://sbpl.net/

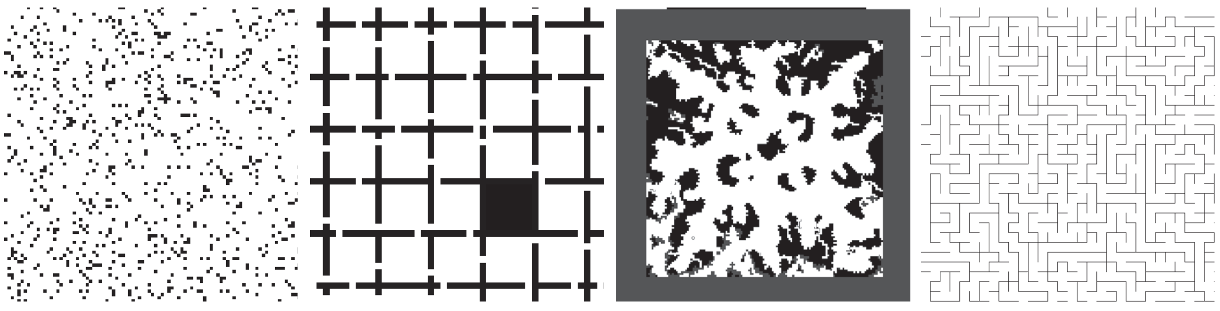

Figure 2: Sample maps used from the map sets used in tests: a) a portion of a random map, b) a portion of a rooms map, c) wc3 (World of Warcraft 3), d) maze.

Table 1: The experimental results for planning in a partially-known map and with a freespace assumption; Ratio denotes AD* to AD*-Cut ratio.

Algorithm	Reinit. time [ms]		First Solution time [ms]		Search time [ms]		Loop time [ms]		#Search Steps	#Under-con. nodes reinit.	Achieved ϵ	Path cost [s]
	Avg.	Max	Avg.	Max	Avg.	Max	Avg.	Max				
random 10 with 5% of randomly shifted cells (partially-known map)												
AD*-Cut	99	1549	23	631	185	1036	298	1826	28965	19116	1.14	21.20
AD*	12	34	22	165	353	1017	366	1026	18100	636	1.15	21.26
Ratio	0.12	0.02	0.96	0.26	**1.91**	0.98	**1.23**	0.56	0.62	0.03	**1.01**	1
rooms (planning with freespace assumption)												
AD*-Cut	101	1879	31	1006	316	1347	420	2281	31971	14724	1.28	11.41
AD*	3	19	67	887	582	1227	586	1237	24794	2855	1.23	11.59
Ratio	0.03	0.01	**2.16**	0.88	**1.84**	0.91	**1.4**	0.54	0.78	0.19	0.96	**1.02**
wc3 (planning with freespace assumption)												
AD*-Cut	141	1525	56	1290	278	1592	532	2389	43456	21200	1.27	11.01
AD*	46	126	133	1667	614	1812	664	1893	38975	4392	1.57	12.34
Ratio	0.33	0.08	**2.38**	**1.29**	**2.21**	**1.14**	**1.25**	0.79	0.9	0.21	**1.24**	**1.12**
mazes 16 (planning with freespace assumption)												
AD*-Cut	119	3523	36	1190	264	1429	411	4016	37845	29062	1.15	12.28
AD*	24	109	238	6077	651	6128	676	6196	36081	11521	1.52	12.43
Ratio	0.2	0.03	**6.61**	**5.11**	**2.47**	**4.29**	**1.64**	**1.54**	0.95	0.4	**1.32**	**1.01**

more searches with improved heuristic inflation factor (indicated by the lower average ϵ), which is also reflected by the higher search steps number. Consequently, average path cost returned by AD*-Cut was lower than that of AD*. However, the path cost ratios do not reveal the same scale of improvement as the achieved inflation factor ratio. This is due to a fact that an inflation factor is the upper-bound and, typically, the sub-optimality of the computed paths is lower.

The possible advantage of AD* over AD*-Cut is seen when the maximum times are analyzed. AD*-Cut is more sensitive to situations in which a large part of the search-tree needs to be cut without substantial change in a cost of a re-planned path that can be observed for the *random10*, *rooms*, and *wc3* map sets. In contrast, for harder problems such as problems from the *mazes16* map set, the maximum loop time of AD* is higher than that of AD*-Cut. This hardiness of problems is also reflected by the number of under-consistent nodes reinitialized by the algorithms. In easier problems (*random10*, *rooms*, and *wc3* map sets) AD* reinitialized several times less nodes than AD*-Cut, while in harder problems, AD* reinitialized 2.5 times less nodes.

Conclusions and Future Work

Search-tree branch cutting is a simple reinitialization technique utilized by the recent incremental search D* Extra Lite algorithm (Przybylski and Putz 2017). This paper shows that this technique can be easily used in combination with the ARA* algorithm (Likhachev, Gordon, and Thrun 2004), resulting in AD*-Cut, a new anytime incremental search algorithm. The results of planning on (x,y,yaw)-state-lattices suggest that AD*-Cut is quicker and achieves shorter paths than AD* (Likhachev et al. 2005).

Future studies will compare AD*-Cut with Anytime Truncated D* (Aine and Likhachev 2013) and Anytime Tree-Restoring A* (Gochev, Safonova, and Likhachev 2014), and to address the problem of time-consuming reinitialization. A practical evaluation of AD*-Cut on a real robot is also planned.

References

Aine, S., and Likhachev, M. 2013. Anytime Truncated D*: Anytime replanning with truncation. In *Proceedings of the Sixth Annual Symposium on Combinatorial Search*, 2–10. Palo Alto, Calif.: AAAI Press.

Gochev, K.; Safonova, A.; and Likhachev, M. 2013. Incremental planning with adaptive dimensionality. In *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling*, ICAPS'13, 82–90. Palo Alto, Calif.: AAAI Press.

Gochev, K.; Safonova, A.; and Likhachev, M. 2014. Anytime tree-restoring weighted A* graph search. In *Proceedings of the Seventh Annual Symposium on Combinatorial Search*, 80–88. Palo Alto, Calif.: AAAI Press.

Hernández, C.; Asín, R.; and Baier, J. A. 2015. Reusing previously found A* paths for fast goal-directed navigation in dynamic terrain. In *Twenty-Ninth AAAI Conference on Artificial Intelligence, Austin, Texas, USA*, AAAI'15, 1158–1164.

Koenig, S., and Likhachev, M. 2005a. Adaptive A*. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems, Utrecht, Netherlands*, AAMAS '05, 1311–1312.

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

Koenig, S.; Likhachev, M.; and Furcy, D. 2004. Lifelong planning A*. *Artificial Intelligence* 155(1):93–146.

Likhachev, M., and Ferguson, D. 2009. Planning long dynamically–feasible maneuvers for autonomous vehicles. *The International Journal of Robotics Research* 28(8):933–945.

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 Fifteenth International Conference on Automated Planning and Scheduling, Monterey, California, USA*, ICAPS'05, 262–271. Palo Alto, Calif.: AAAI Press.

Likhachev, M.; Gordon, G. J.; and Thrun, S. 2004. ARA*: Anytime A* with provable bounds on sub-optimality. In *Advances in Neural Information Processing Systems*, 767–774.

Podsedkowski, L.; Nowakowski, J.; Idzikowski, M.; and Vizvary, I. 2001. A new solution for path planning in partially known or unknown environment for nonholonomic mobile robots. *Robotics and Autonomous Systems* 34(2):145–152.

Przybylski, M., and Putz, B. 2017. D* Extra Lite: a Dynamic A* with search-tree cutting and frontier-gap repairing. *International Journal of Applied Mathematics and Computer Science (AMCS)* 27(2):273–290.

Stentz, A. 1995. The Focussed D* algorithm for real-time replanning. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2, Montreal, Quebec, Canada*, IJCAI'95, 1652–1659.

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

Sun, X., and Koenig, S. 2007. The Fringe-Saving A* search algorithm—a feasibility study. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence, Hyderabad, India*, IJCAI'07, 2391–2397.

Sun, X.; Koenig, S.; and Yeoh, W. 2008. Generalized Adaptive A*. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1, Estoril, Portugal*, AAMAS '08, 469–476.

Trovato, K. I., and Dorst, L. 2002. Differential A*. *IEEE Transaction on Knowledge and Data Engineering* 14(6):1218–1229.