Accelerated A* Trajectory Planning: Grid-based Path Planning Comparison

David Šišlák and Přemysl Volf and Michal Pěchouček*
Agent Technology Center, FEE, Czech Technical University in Prague

Abstract

The contribution of the paper is a high performance path-planning algorithm designed to be used within a multi-agent planning framework solving a UAV collision avoidance problem. Due to the lack of benchmark examples and available algorithms for 3D+time planning, the algorithm performance has been compared in the classical domain of path planning in grids with blocked and unblocked cells. The Accelerated A* algorithm has been compared against the Theta* path planner, Rapid-Exploring Random Trees-based planners and the original A* searching in graphs providing the shortest any-angle paths. Experiments have shown that Accelerated A* finds the shortest paths in all scenarios including many randomized configurations. Experiments document that Accelerated A* is slower than Theta* and RRT-based planers in many cases, but it is faster than the original A*. In comparison to the original A*, Accelerated A* reduces memory requirements which makes it usable for large-scale worlds where the original A* is not usable.

Introduction

The paper presents the original Accelerated A* trajectory planning algorithm which has been designed for fast planning and replanning of the UAV *free-flight* operations. When performing free-flight the aircraft follow their individual plans, detect possible collisions and they repair their trajectories by peer-to-peer negotiations so that the collision is avoided. There are no predefined corridors neither flight levels, thus the planning needs to carried in 3D+time space. The state-space of possible collision avoidance maneuvers is vast and it needs to be searched very quickly.

The Accelerated A* planning algorithm (Šišlák, Volf, and Pěchouček 2009) satisfies these requirements and empirical measurements proved its fine performance. There are no widely accepted benchmark problems for 3D+time robotic planning. In order to compare some of its performance metrics with the state-of-the-art robotic algorithms, we have decided to perform scalability tests in the simpler grid-based path planning scenario.

The grid-based path planning addresses the vehicle path planning problem in a two-dimensional terrain which is discretized into grid cells that are either blocked or unblocked. The goal is to find the shortest path from the start location to the goal location that doesn't intersect any blocked cell. Both start and goal locations are at corners of cells (grid positions). To avoid unrealistic paths restricted only to grid edges using only a limited set of possible heading changes (Yap 2002), a path can connect any two nodes (any-angle path) if the line between them doesn't intersect any blocked cell, as shown in Figure 1.

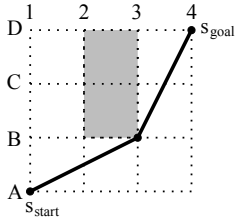

Figure 1: An any-angle path in a grid - white cells are unblocked, grey cells are blocked and the solid line is the shortest path.

Several approaches to the grid path planning problem exist in the research community. The grid is usually replaced with a graph, where nodes are placed either in centers or corners of grid cells and edges connect nodes if the respective straight-line doesn't intersect any obstacle. The most common one is eight-connected graph mapping which puts edges only between nodes from adjacent cells. Planning on such a graph is fast since the number of edges is linear in the number of grid cells. For any-angle path planing, a graph contains edges connecting all node pairs which can be connected (fully-connected). In such a graph, the number of edges is quadratic in the number of cells.

The original A* algorithm (Hart, Nilsson, and Raphael 1968) uses a heuristic to focus the search towards the goal position. Using an edge cost and a heuristic based on the Euclidean distance, the A* algorithm finds the shortest paths for used graphs. In the case of fully-connected graphs, it finds the shortest any-angle paths. But the search is slow due to high number of edges in the graph. A* running on four-connected grids in combination with post-smoothing (PS) (Botea, Müller, and Schaeffer 2004) is able to find non-shortest any-angle paths faster. There exist several incremental modifications of A* that incrementally repair paths after each change of obstacles and thus in the underlying graph: D* (Stentz 1995), incremental A* (Koenig and

*The work has been supported by the Air Force Office of Scientific Research, Air Force Material Command, USAF, under grant number FA8655-06-1-3073 and by Czech Ministry of Education grant number 6840770038.

Likhachev 2002b), D* Lite (Koenig and Likhachev 2002a), and fringe-saving A* (Sun and Koenig 2007). But all these approaches allow only limited transitions from each node using four or eight-connected graphs which results in sub-optimal paths restricting heading changes by multiples of $\Pi/4$ or $\Pi/8$. Field D* (FD*) (Ferguson and Stentz 2006) is a variant of A* that uses linear interpolation of path costs along grid edges to find any-angle paths. Theta* (Θ*) (Nash et al. 2007) allows the parent of a vertex to be any vertex and not only a direct predecessor like in A*. It connects each successor of a vertex with its parent if it is possible considering the blocked cells. Θ* using eight neighbors in a grid for generating successors of a vertex finds shorter paths in less time than FD*, as shown in (Nash et al. 2007).

The grid size is crucial for path planing – the size of one grid cell defines the minimal distance between obstacles to search for a path between them. If the number of cells in the grid is high, then the search algorithm is slow. There exist several modifications of the A* algorithm relevant to the paper. The incrementally refined A* search (Cormen et al. 2001) starts sampling initially with a sparse grid and its resolution is iteratively refined until a solution is found. Such a modification generates a path that can go around obstacles instead of going through small gaps between them. To check that no path exists between the initial and goal positions, it needs to iteratively fail several times until it fails for the most precise search. The hierarchical path-finding A* (Botea, Müller, and Schaeffer 2004) reduces the path planning complexity by planning in abstraction levels (hierarchy of sectors). It searches consecutively from the highest abstraction level towards a precise world model. Decomposition of the world definition into such abstraction levels with map clusters with identified links among them is a very complex task.

The 3D field D* algorithm (Hildum and Smith 2007) extends the FD* algorithm with an acceleration of the search by search tree running over the smallest unoccupied cells in an octant tree structure which is used for environment representation. The acceleration of planning in areas where there are no close obstacles is similar to the dynamic expansion approach presented in the paper. But it requires expensive transformation of obstacles (blocked cells) into one octant tree structure. For a large-scale world, it is required to limit the maximum depth of the tree structure. This causes an increase of the smallest cell dimension (it defines the smallest gap between any two obstacles where the algorithm can find a path).

There exist many algorithms based on a randomized search. They are very efficient and provide a solution quickly. However, they provide a different result each time and cannot guarantee any properties of the provided paths except the fact that they don't intersect any obstacle. Moreover, some of the algorithms provide complex paths with many unnecessary segments which need to be further smoothed to be executable by a vehicle. Algorithms based on the rapidly exploring random tree (RRT) (La Valle and Kuffner 2001) are very popular for a search in complex large-scale environments.

The novel Accelerated A* (AA*) algorithm is presented in the paper. AA* algorithm uses four neighbors as successors during expansion of a node whose distances from that node differ. This adaptive expansion varies depending on the distance from obstacles in that area. To provide short any-angle paths, AA* tries to amend the parent of each node to a suitable candidate from a set of already expanded nodes in order to find a shorter any-angle path. In experiments, it is shown that AA* is faster than the original A* and slower than Θ* and RRT-based planners in many cases. But the main advantage of AA* is in the fact that it finds the shortest paths in all scenarios including many randomized configurations.

```
{1}  Search(s_start, s_goal)
{2}      g(s_start) ← 0;
{3}      h(s_start) ← c(s_start, s_goal);
{4}      parent(s_start) ← false;
{5}      OPEN ← {s_start};
{6}      CLOSED ← ∅;
{7}      while OPEN ≠ ∅ do
{8}          s_c ← RemoveTheBest(OPEN);
{9}          if s_c = s_goal then return s_c;
{10}         Insert(s_c, CLOSED);
{11}         foreach s_d ∈ Candidates(s_c) do
{12}             if Contains(s_d, CLOSED) then continue;
{13}             if Intersect(s_c, s_d) then continue;
{14}             g(s_d) ← g(s_c) + c(s_c, s_d);
{15}             h(s_d) ← c(s_d, s_goal);
{16}             parent(s_d) ← s_c;
{17}             ProcessNode(s_d);
{18}         end
{19}     end
{20}     return false;
{21} end
{22} Candidates(s_c)
{23}     return NODES;
{24} end
{25} ProcessNode(s_d)
{26}     InsertOrReplaceIfBetter(s_d, OPEN);
{27} end
```

Algorithm 1: A* algorithm

A* Algorithm

Both Θ* and AA* planning algorithms are modified versions of the original A* (Hart, Nilsson, and Raphael 1968), shown in Algorithm 1. The algorithm is organized so that the `Search` function is reused for both Θ* and AA* algorithms. To focus A* search, a heuristic based on the Euclidean distance to the goal is used in all modifications. The algorithm works with three values for each vertex s: (i) $\mathrm{g}(s)$ is the length of the path from the start vertex s_{start} to s found so far, (ii) $\mathrm{h}(s)$ is the value of the heuristic for s and (iii) $\mathrm{parent}(s)$ is used to store the link to the predecessor of s which is also used to extract the final path. $\mathrm{c}(s_i, s_j)$ is the straight line Euclidean distance between s_i and s_j. The algorithm maintains two global structures: (i) $OPEN$ is a priority queue that contains vertices for expansion and (ii) $CLOSED$ contains already processed vertices and is used to ensure that each vertex is processed only once. Initially,

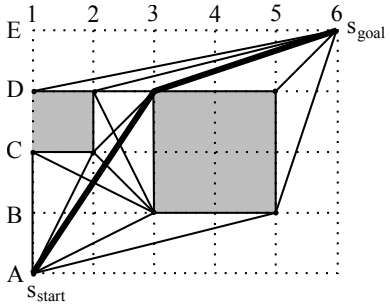

Figure 2: Visibility graph: the bold solid line is the shortest path, solid lines are edges in the visibility graph.

the algorithm initializes values for the start vertex s_{start}, $OPEN$ and $CLOSED$ structures, lines 2–6.

The main search loop is repeated until $OPEN$ is empty, lines 7–19. If $OPEN$ is empty it means that no path is found and the function `Search` returns $false$. The function `RemoveTheBest` pops one candidate s_c with the lowest value of $\mathrm{g}(s_c) + \mathrm{h}(s_c)$, line 8. s_c is tested if it matches the goal vertex s_{goal}, line 9. If the goal vertex is found, the search just found a path from the start to the goal and the algorithm returns the last vertex which is used for a path reconstruction using parent references. Otherwise, s_c is stored in $CLOSED$, line 10. Then, s_c is used for generation of all possible successors of s_c regardless of intersection with blocked cells, generated by the function `Candidates`, line 11.

Each successor candidate vertex s_d is tested whether it hasn't been processed yet (line 12) and the straight-line to this vertex from the predecessor s_c doesn't intersect any blocked cell (line 13). Then, $\mathrm{g}(s_d)$, $\mathrm{h}(s_d)$ and $\mathrm{parent}(s_d)$ values are updated, lines 14–16. Finally, s_d is passed for further processing to the function `ProcessNode`, line 17.

The problem of searching for the shortest any-angle paths in grids can be transformed into a search for the shortest paths in visibility graphs, as shown in Figure 2. Visibility graphs contain the start vertex, the goal vertex and vertices in corners of all blocked cells (Lozano-Pérez and Wesley 1979). Edges between any two vertices are defined if and only if the straight-line between them doesn't intersect any blocked cell. The A* algorithm (Algorithm 1) uses a set $NODES$ in the function `Candidates` to generate all possible successors for a node, line 23. $NODES$ is constructed in the following manner. For each blocked cell in a grid, insert all its corner vertices to $NODES$ for which there are not four blocked cells around the vertex, e.g. the vertex C4 in Figure 2 is not inserted because there are four blocked cells around it. The goal vertex is inserted into $NODES$ too. Each vertex can be in $NODES$ only once. The function `ProcessNode` is pretty simple for the original A* implementation. It just inserts the vertex s_d to $OPEN$, line 26.

Theta*

The key difference between Θ* (Nash et al. 2007) and A* is that Θ* allows the parent of a vertex to be any from its predecessors not only the direct predecessor vertex like in A*. In the paper, Θ* refers to the basic Theta*. The extended angle-propagation Theta* (AP Θ*) version removes intersection checks with angle lower and upper bound checks but each newly generated node requires a complex computation of its new angle range. In (Nash et al. 2007), it is documented that AP Θ* over-constrains the angle ranges which causes that some paths are removed even though they can be used. Moreover, AP Θ* is slower than the basic Θ* in almost all experiments due to a complex angle propagation method. Thus, AP Θ* is not used in the paper.

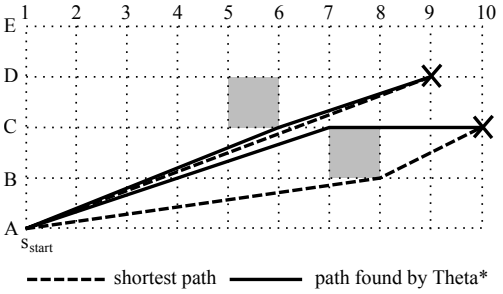

Figure 3: Sub-optimality of Θ* algorithm (Nash et al. 2007).

```
{28} Candidates(s_c)
{29}     return EightNeighbors(s_c);
{30} end
{31} ProcessNode(s_d)
{32}     s_pp ← parent(parent(s_d));
{33}     if not Intersect(s_pp, s_d) then
{34}         g(s_d) ← g(s_pp) + c(s_pp, s_d);
{35}         parent(s_d) ← s_pp;
{36}     end
{37}     InsertOrReplaceIfBetter(s_d, OPEN);
{38} end
```

Algorithm 2: Theta* algorithm

The pseudo-code of the Θ* algorithm is shown in Algorithm 2. The main search function is the same as in A*, Algorithm 1 lines 1–21. In comparison to A*, Θ* produces only the eight neighbors of a vertex as its successors, line 29. In the function `ProcessNode`, Θ* implements a path truncation which is applied to all generated vertices. The predecessor of c_d is replaced with the parent of this predecessor if the straight-line between that parent and c_d doesn't intersect any blocked cell, lines 32–36. The described one step truncation of Θ* provides shorter paths, but it is not guaranteed that such paths are the shortest ones, as shown in Figure 3.

Accelerated A*

The AA* algorithm uses only successors from four candidates within the function `Expand`, as shown in Algorithm 3. Similarly to Θ*, this accelerates the search process by reduction of the search branching factor from linear to constant in number of vertices. The reduction of the search branching factor effectively reduces the number of all generated states and the size of the $OPEN$ list as well. Thus, it reduces memory requirements and speeds up the $OPEN$ list operations.

```
{39} Candidates(s_c)
{40}     sq ← DetectMaxSquare(s_c);
{41}     return UsableSideCenters(sq);
{42} end
{43} ProcessNode(s_d)
{44}     foreach s_n ∈ EllipseMbs(CLOSED, s_start, s_d)
         do
{45}         if g(s_n) + c(s_n, s_d) < g(s_d) then
{46}             if not Intersect(s_n, s_d) then
{47}                 g(s_d) ← g(s_n) + c(s_n, s_d);
{48}                 parent(s_d) ← s_n;
{49}             end
{50}         end
{51}     end
{52}     InsertOrReplaceIfBetter(s_d, OPEN);
{53} end
```

Algorithm 3: Accelerated A* algorithm

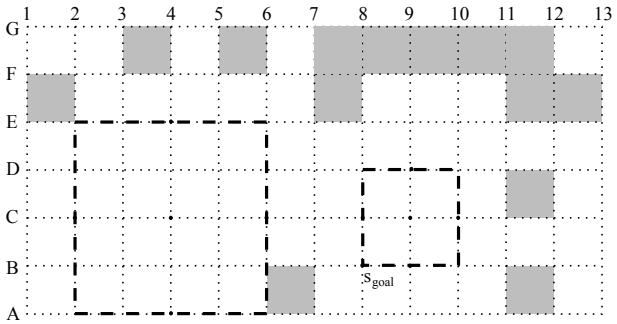

Figure 4: Maximum unblocked squares and successor vertices for C4 and C9.

There are two major differences between AA* and Θ*: (i) use of a *dynamic adaptive expansion* and (ii) the way how AA* searches for the *path truncation*. The function `Candidates` prepares four successor candidates (each in one direction) at maximum using the *maximum unblocked square*, lines 40 and 41. To find the maximum unblocked square for a given vertex (always positioned in the square center), both blocked cells and the goal vertex are considered, see two examples in Figure 4. No blocked cell can be in the square area and the goal vertex cannot lie inside the squeare perimeter. The minimal size of the square is 2x2 and grid dimensions don't restrict the square size. For example, the vertex C4 has the maximum unblocked square with size 4x4 due to blocked cells and the square defines C2, A4, C6 and E4 as successors of C4. On the other hand, the vertex C9 has the maximum unblocked square with the size 2x2 only due to the goal vertex located at the position B8. Thus, C8, B9, C10 and D9 are defined as successors of C9. Such a generation of successors guarantees that the algorithm doesn't skip any vertex from the visibility graphs where the shortest paths come from.

AA* truncates the current path to each generated successor s_d to be the shortest one taking into account the already processed vertices stored in $CLOSED$, lines 44–51. It searches for a new parent vertex in $CLOSED$ which is: (i) usable (a path between the vertex and its new parent doesn't intersect any blocked cell) and (ii) the g cost through this vertex is the minimal one. It is not necessary to traverse all vertices in $CLOSED$, because such a candidate has to be located inside the ellipse uniquely defined by the start vertex s_{start}, the current vertex s_d and the cost of the path to the current vertex $\mathtt{g}(s_d)$. s_{start} and s_d defines two foci of the ellipse, the major radius is $a = \mathtt{g}(s_d)/2$ and the minor radius is $b = (\sqrt{\mathtt{g}(s_d)^2 - \mathtt{c}(s_{start}, s_d)^2})/2$, see illustration in Figure 5. The vertex D8 has been generated from the vertex E8 because the goal vertex is in the upper right direction from the start. Using the ellipse test, the vertex B7 from $CLOSED$ is identified as the new parent for D8. The path to D8 through B7 is shorter than the original one. The extraction of all vertices from $CLOSED$ which are inside the ellipse is accelerated by using a spatial-based hash table (Cormen et al. 2001). $CLOSED$ already uses a hash table for contains tests. A path going through a vertex outside the ellipse cannot be shorter than the existing path. The boundary of the ellipse defines exactly those points where the cost of a straight line from s_{start} to the point, plus the cost of a straight line from that point to s_d, is exactly $\mathtt{g}(s_d)$, which is the cost of the current path. If you ever go from s_{start} outside the ellipse, you definitely cannot get back to s_d without incurring a cost strictly greater than $\mathtt{g}(s_d)$.

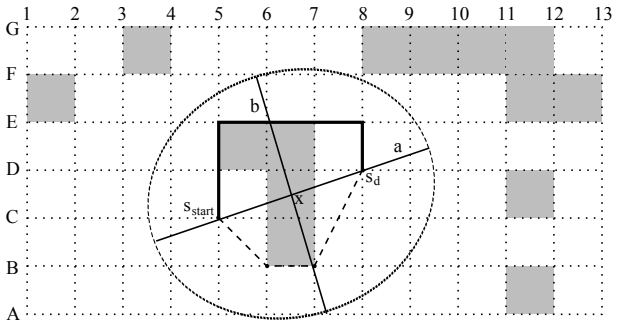

Figure 5: Path truncation in AA*: a is the major radius, b is the minor radius and x is the ellipse center.

The complexity of the function `DetectMaxSquare` is quadratic in the number of cells, but can use a bisection method (Cormen et al. 2001) to accelerate the detection process. In such a case, a bisection is used to reduce the number of tests required for finding the maximum unblocked square. The maximum unblocked square has to have the size ranging from 2x2 to NxN, where $N = 2 * \max(\varepsilon, 1)$ and ε is the minimum of the vertical and horizontal grid distances between the square center vertex and the goal vertex. AA* truncation in the function `ProcessNode` has quadratic complexity. However, AA* works with less vertices than A* and thus AA* is still faster than A*. In the case where few cells are blocked, AA* explores a grid rapidly and the ellipse test covers only a limited number of parent candidates during the truncation. On the other hand, if there are many blocked cells, the path length g to a vertex near the goal is much longer than the Euclidean distance to the start and thus the ellipse covers a major part of the grid. In such a

case, the time required to generate and process successors is almost similar to the A*. But for vertices closer to the start, the ellipse becomes smaller and it requires less time. The overall run-time of the search is still faster than for A*.

```
{54} RRT(s_start, s_goal)
{55}     τ.init(s_start);
{56}     for k=1 to K do
{57}         s_rand ← RandomVertex();
{58}         s_near ← τ.nearest(s_rand);
{59}         s_new ← τ.stopping_configuration(s_near, s_rand);
{60}         if s_new ≠ s_near then
{61}             τ.add_vertex(s_new);
{62}             τ.add_edge(s_near, s_new);
{63}         end
{64}         s_near ← τ.nearest(s_goal);
{65}         if not Intersect(s_near, s_goal) then
{66}             return τ, s_near;
{67}     end
{68}     return false;
{69} end
```

Algorithm 4: RRT algorithm

Rapidly-exploring Random Tree

In this section, the Rapidly-exploring Random Tree (RRT) path planning technique is briefly introduced. RRT techniques are very popular nowadays and they have been successfully applied to many planning problems also in robotics. RRT was chosen as a representative for random-based path planners within experiments. However, RRT is suitable for high dimensional spaces, contrarily to most of sampling-based techniques. Specifically, two version are used: (i) RRT with one exploring tree (La Valle and Kuffner 2001) and (ii) dynamic domain RRT with two exploring trees (Yershova et al. 2005). The pseudo-code of unidirectional RRT is shown in Algorithm 4. Initially, the tree τ is initialized with the start vertex, line 55. Then, RRT incrementally searches a grid for a path connecting start and goal vertices, lines 56–67. The number of iterations is limited to the constant K and if a path is not found the search fails, line 68. At each iteration, a new vertex is sampled and the extension from the nearest node in the tree towards this sample is attempted, lines 57–59. The function $\mathrm{stopping\_configuration}$ returns the last vertex s_{new} in the direction from s_{near} towards s_{rand} for which the straight-line between s_{near} and s_{new} doesn't intersect any blocked cell. If the extension succeeds (s_{new} is different from s_{near}), a new node and edge in tree is created, lines 61 and 62. Then, RRT checks if the goal can be connected to the tree not intersecting any blocked cell, lines 64–66. If such a test passes, a path is found. The constructed tree τ and s_{near} is used for the reconstruction of the path from the start to the goal.

The most complex part of the RRT algorithm is the function $\mathrm{nearest}$, line 58. In order to implement the search for an any-angle path over a grid, the function needs to search for the nearest vertex not only from the set of inserted vertices in the tree τ but to any grid position at any edge in τ. This behavior can be simplified in the following manner. Each time when a new sample is added in τ, lines 61 and 62, all intermediate vertices between s_{new} and s_{near} corresponding to a grid position are inserted too. In such an approach, $\mathrm{nearest}$ is implemented as a process of finding the nearest node in τ. This results in the computation time for the function $\mathrm{nearest}$ which is linear in the number of vertices. To speed up such a search, the widely used KD-tree structure (Cormen et al. 2001) is used.

The dynamic-domain modification provides significant speed up for RRT planning. It reduces the negative effects of large Voronoi regions causing a considerable bias towards the vertices near obstacles (Yershova et al. 2005). The dynamic-domain modification of RRT limits large Voronoi regions for vertices near obstacles. When a point is quite far from obstacles its boundary domain is the same as the RRT's sampling domain, that is the whole Voronoi region. The dynamic domain for the random vertex selection is implemented using radius value for each vertex. By default, it is set to infinity which guarantees the default domain behavior. If a generated vertex doesn't provide any growth of the tree, the domain is restricted. For all experiments, the restricted region has the radius 10. The bidirectional balanced RRT expands two trees, one from the start and the second from the goal. After each successful tree extension, both trees are swapped. A path from the start to the goal is found if the inserted vertex s_{new} can be connected also with the other tree without intersection with any obstacle.

Due to the random nature of RRTs, the post-smoothing applied to paths formed by RRT planners can considerable shorten their lengths. In the paper, the following post-smoothing (PS) (Botea, Müller, and Schaeffer 2004) is used. The path's last vertex is set as the current vertex (s_0). PS checks whether the straight-line between the current vertex (s_0) and the parent of its parent on the path (s_2) doesn't intersect any blocked cell. If so, PS removes the parent (s_1) of the current vertex (s_0) from the path and repeats the procedure by checking again whether the straight-line between the current vertex (s_0) and the parent of its parent on the path (s_3) is usable, and so on. If not, PS uses the parent vertex (s_1) as the current vertex and repeats the procedure again until the path is shortened to the start vertex or the start vertex is set as the current one.

Experiments

The presented AA* algorithm is compared to the original A*, the basic Theta* (Θ*), the rapidly-exploring random tree with post-smoothing (RRT PS) and the dynamic domain bi-directional rapidly-exploring random tree with post-smoothing (dynamic bi-RRT PS) in various grids of size 100x100, 500x500 and 1000x1000. Field D* (FD*) (Ferguson and Stentz 2006) is not included in the experiment, because it is shown in (Nash et al. 2007) that Θ* finds shorter paths in less time than FD* in similar tests. Both RRT planners were executed also without post-smoothing, but it was found that post-smoothing shortens paths with insignificant time consumption. Thus, all results are provided only for post-smoothed RRTs. All path planning methods were implemented in Java and executed on a 2.5 GHz Intel Xeon CPU.

Configuration	Shortest Paths A*	Θ*	AA*	RRT PS	dynamic bi-RRT PS
5% blocked cells	54.210 (10.084)	54.345 (0.023)	54.210 (0.079)	73.834 (0.003)	73.411 (0.0004)
10% blocked cells	53.190 (11.896)	53.428 (0.026)	53.190 (0.082)	79.558 (0.013)	75.896 (0.0015)
20% blocked cells	53.301 (18.476)	53.623 (0.036)	53.301 (0.101)	85.207 (0.032)	78.030 (0.0037)
30% blocked cells	53.206 (31.493)	53.611 (0.049)	53.206 (0.129)	85.344 (0.077)	81.566 (0.0089)

Table 1: Path lengths and run-times (in parenthesis), each averaged from 500 runs for random grids of size 100 x 100.

Configuration	Shortest Paths A*	Θ*	AA*	RRT PS	dynamic bi-RRT PS
5% blocked cells	56 (1962)	172 (275)	138 (181)	70	181
10% blocked cells	102 (3296)	216 (324)	162 (205)	323	412
20% blocked cells	198 (2597)	302 (403)	238 (294)	921	988
30% blocked cells	294 (1809)	372 (466)	324 (363)	1938	2053

Table 2: The number of vertex expansions and generated vertices (in parenthesis), each averaged from 500 runs for random grids of size 100 x 100.

Randomized grids

In the first set of experiments, all path planning algorithms were executed on grids of size 100 x 100 with randomly blocked cells. Start and goal vertices were selected randomly too. Similarly to (Nash et al. 2007), four different densities of obstacles were selected: 5%, 10%, 20% and 30% blocked cells of the whole grid. Table 1 summarizes results presenting lengths of paths and run-times of tested algorithms. Each different density of obstacles were measured 500 times and results show average values from all repetitions. Please, note that each generated planning task (grid, start and goal vertices) was executed by all algorithms. In other words, the same 500 tasks were executed by A*, Θ*, AA*, RDT PS and dynamic bi-RDT PS.

AA* finds the same shortest paths as the original A* running on visibility graph in all cases. Θ* finds longer (but still very close) paths than the shortest ones. Paths found by RRT planners are more than 36% longer than the shortest paths. In randomized grids, dynamic bi-RRT PS provides shorter paths than RRT PS. On the other hand, both RRT planners are very fast. Both Θ* and AA* are many times faster than the original A*. Θ* is about three times faster than AA* in these configurations.

Table 2 provides results about the number of expanded vertices (those which were popped from $OPEN$) and the total number of generated vertices. For RRT-based planners only the number of generated vertices is presented. AA* works with less vertices than Θ* and the original A*. Thus, AA* requires less memory. The reduction of the number of vertices for AA* is primarily given by the reduction of the search branching factor to 4 in these configurations. It is not necessary to work with all visible vertices (connected by an edge in the visibility graph) like in A* to find the shortest paths. Although the original A* algorithm processes the lowest number of vertices in all cases, it is the slowest one because it detects all visibility edges during each expansion and put all of them to $OPEN$. Dynamic bi-RRT PS visits more vertices than RRT PS in grids but the fact that the search is running simultaneously in both directions provides significant acceleration of the search.

Selected Grids

In the second set of experiments, selected obstacle configurations were used, as shown in Figure 6: *a wall* (a), *a half circle* (b), *a single gap* (c), *a double gap* (d), *a maze* (e) and *multi-obstacles* (f). Obstacle positions were motivated by path planning testing setups as presented in (LaValle 2006). The first five configurations, (a)–(e), are within grids of size 500x500 and the sixth configuration, (f), is within the grid of size 1000x1000. In all these configurations, each solid line in Figure 6 is mapped to many blocked cells which do not allow a path connection to the opposite side of the line.

Configurations (a) and (b) have obstacles positioned so that they cause deviation from paths which are strongly preferred by the used distance to goal heuristic. In the configuration (b), the path needs to go slightly in a opposite direction, away from the goal as the start vertex is positioned within that half circle obstacle. Configurations (c) and (d) are used for checking that the planning algorithm is able to find a path through a small hole in obstacles. The configuration (e) is used for verification of acceleration capabilities in the case where the heuristic is completely inefficient. Finally, the configuration (f) has many half circle obstacles between start and goal vertices and demonstrates a large-scale search problem. Each selected configuration was executed once by A*, Θ* and AA* path planning algorithms. Their behavior is deterministic and results are the same for the same configurations. Only values for RRT-based planners are averaged from 500 successive executions because each search provides a different result due to the random nature of the algorithms.

Table 3 provides a comparison of lengths of paths and required run-times for selected configurations. Similarly to random grids, AA* finds same the shortest paths as the original A* running on visibility graph for all configurations. Θ* finds slightly longer paths than the shortest ones. Both RRT planners provides much longer paths than others. For example, in the multi-obstacles configuration, average path lengths are more than 4 times longer which is caused by a very complex combination of blocked cells. In all cases, the original A* algorithm requires the longest run-time to

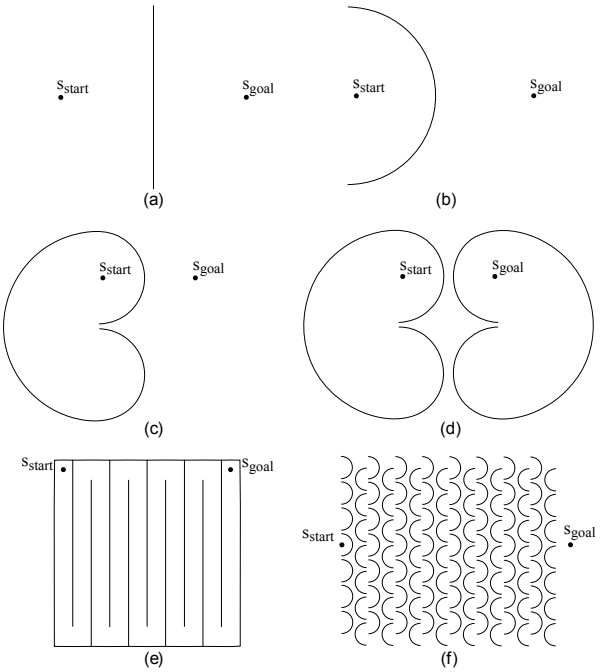

Figure 6: Selected experiment configurations: (a) a wall, (b) a half circle, (c) a single gap, (d) a double gap, (e) a maze and (f) multi-obstacles. The start vertex is denoted as s_{start} and the goal vertex is denoted as s_{goal}.

find a solution. Even though AA* algorithm has the same quadratic complexity like the original A*, AA* is much faster than A* in these configurations too. The speed up effect is gained by both the reduction of the search branching factor and the adaptive expansion. The adaptive expansion helps AA* to explore quickly parts of grids far from any obstacle and not to lose the ability to find paths through small holes like in configurations (c) and (d).

Required run-times in Table 3 correlates with the number of expanded and generated vertices in Table 4. The adaptive expansion of AA* significantly reduces the number of expanded vertices in comparison to Θ*. The huge number of vertices in $OPEN$ further slows down operations with it, especially removal of the best candidate from $OPEN$. Although $OPEN$ combines together both a *hash table* for speed up of contains' tests and a *heap* structure (Cormen et al. 2001) in measured implementations, Θ* is slower than AA* in four configurations. For example, in the large-scale configuration (f), AA* is about 12 times faster than Θ* and works with about one sixth of vertices. Beside the speed up effect, the reduction of processed vertices also decreases memory requirements during the search. These memory savings allow the AA* algorithm to find a path also in configurations where it is impossible with the original A* and also with Θ*.

RRT-based planners are very fast in configurations (a) and (b). But their paths are much longer than for others even though the post-smoothing is applied. RRTs' run-times are considerably increased in configurations where there exist only a few unblocked cells through which the path needs to pass. In the single gap case (c), the use of two searching trees still provides fast result. In the complex large-scale configuration (f), RRT PS is two times slower than AA* and dynamic bi-RRT PS has similar run-times even though KD-tree acceleration structure (Cormen et al. 2001) is used in RRT implementations. In this configuration, both RRTs work with a huge number of vertices which increases its memory requirements.

Conclusion

The novel Accelerated A* (AA*) algorithm applied to grid-based any-angle path planning problem has been presented in the paper. AA* modifies the original A* searching for the shortest any-angle paths in two key parts: (i) AA* *reduces the search branching factor* and (ii) AA* uses *an adaptive expansion*. AA* reduces the search branching factor from quadratic to constant in comparison to the original A* applied to visibility graphs providing any-angle paths. Each vertex can have four successors at maximum in AA*. This idea is not new, because a similar one has been already used in Theta* (Θ*) (Nash et al. 2007) which uses eight grid neighbors. However, AA* comes with novel *progressive truncation* applied to each generated node. In contrast to Θ*, AA* searches for a new suitable parent for each generated node in a limited set of vertices in $CLOSED$ and not only in a set of node's predecessors. Beside the reduced branching factor and the progressive truncation, AA* comes with an adaptive expansion in grids. It is based on the identification of the maximum unblocked square around a vertex selected for the expansion. Successors are then selected in the middle of each side of such a square. This modification of AA* avoids expansion of states in parts far from any obstacle and don't affect the capability to find a path through a small hole (one unblocked cell) between obstacles. The used modification doesn't require any preprocessing of grids. The idea of an accelerated space exploration is not novel but the way how the accelerated search is done is novel. It was previously used in 3D Field D* (3D FD*) (Hildum and Smith 2007) where the acceleration is given by varying size of octant tree cells. In 3D FD*, the octant tree is used as a structure for storage information about obstacles. Each time an obstacle definition is changed, the octant tree needs to be rebuilt which is a complex task.

The described AA* doesn't reduce the quadratic search complexity like Θ* but the number of processed nodes is much lower than in the original A*. Thus, AA* is also many times faster than A*. Properties of AA* have been validated on many configurations. The main AA* advantage is the fact that it provides the shortest paths in all of more than two thousand configurations used during experiments. Beside a few selected configurations, many randomized grids have been used. No grid where AA* provides different result than the original A* searching for the shortest any-angle paths was observed. Although AA* is slower than Θ* and rapid-exploring random trees (RRT) planners in randomized grids, AA* is faster than Θ in four of six selected configurations. Moreover, in the complex large-scale multi-obstacles configuration, AA* is faster than unidirectional RRT and is

Configuration	Shortest Paths A*	Θ*	AA*	RRT PS	dynamic bi-RRT PS
a wall	357.106 (2 249.5)	357.125 (0.504)	357.106 (0.881)	490.124 (0.0004)	525.953 (0.0001)
a half circle	422.154 (2 827.2)	424.876 (0.652)	422.154 (0.264)	685.465 (0.0021)	695.129 (0.0001)
a single gap	395.991 (3 985.4)	399.072 (0.736)	395.991 (0.207)	505.632 (0.1962)	737.162 (0.0006)
a double gap	485.213 (6 131.4)	490.056 (1.744)	485.213 (0.735)	581.479 (0.2949)	614.254 (0.2173)
a maze	4 121.478 (10 989.3)	4 133.491 (3.148)	4 121.478 (7.202)	4 542.958 (0.4502)	4 559.624 (0.0188)
multi-obstacles	662.550 (6 750.2)	696.697 (250.208)	662.550 (20.586)	2 971.787 (40.3726)	2 768.783 (17.7803)

Table 3: Path lengths and run-times (in parenthesis). Values for RRT PS and dynamic bi-RRT PS are averages from 500 runs.

Configuration	Shortest Paths A*	Θ*	AA*	RRT PS	dynamic bi-RRT PS
a wall	436 (876)	24 020 (25 216)	687 (715)	24	74
a half circle	605 (2 132)	25 756 (26 912)	811 (855)	48	131
a single gap	562 (1 871)	24 308 (25 524)	786 (836)	24 462	463
a double gap	1 092 (2 694)	44 380 (45 302)	1 940 (2 003)	25 266	28 228
a maze	3 324 (5 347)	145 284 (147 940)	13 769 (13 851)	46 272	81 843
multi-obstacles	17 634 (43 796)	166 091 (168 355)	28 149 (28 532)	781 558	638 117

Table 4: The number of vertex expansions and generated vertices (in parenthesis). Values for RRT PS and dynamic bi-RRT PS are averages from 500 runs.

as fast as dynamic domain bidirectional RRT.

Another strong property of AA* is the reduction of used vertices during the search. AA* works with less vertices than A* and Θ* in all configurations. In many cases, AA* uses less vertices than RRT-based planners. Such a reduction of used vertices implies reduction of memory requirements during the search. These memory savings allow to use AA* also to search in complex large-scale grids. In the future, it has to be checked whether AA* techniques are usable also for grids with non-uniform traversing costs for the movement through a cell which are widely used in robotics.

References

Botea, A.; Müller, M.; and Schaeffer, J. 2004. Near optimal hierarchical path-finding. *Journal of Game Development* 1(1):7–28.

Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; and Stein, C. 2001. *Introduction to Algorithms (2nd Edition)*. Cambridge, MA: MIT Press and McGraw-Hill.

Ferguson, D., and Stentz, A. 2006. Using Interpolation to Improve Path Planning: The Field D* Algorithm. *Journal of Field Robotics* 23(1):79–101.

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

Hildum, D., and Smith, S. F. 2007. Constructing conflict-free schedules in space and time. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS)*.

Koenig, S., and Likhachev, M. 2002a. D* lite. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*.

Koenig, S., and Likhachev, M. 2002b. Incremental A*. In *Advances in neural information processing systems*. MIIT Press.

La Valle, S. M., and Kuffner, J. J. 2001. Rapidly exploring random trees: Progress and prospects. In Donald, B. R.; Lynch, K. M.; and Rus, D., eds., *Algorithmic and Computational Robitics: New Directions*, 293–308. MA, USA: A K Peters, Wellesley.

LaValle, S. M. 2006. *Planning Algorithms*. Cambridge University Press.

Lozano-Pérez, T., and Wesley, M. A. 1979. An algorithm for planning collision-free paths among polyhedral obstacles. *Communication of the ACM* 22(10):560–570.

Nash, A.; Daniel, K.; Koenig, S.; and Felner, A. 2007. Any-angle path planning on grids. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 1177–1183.

Stentz, A. 1995. The Focussed D* Algorithm for Real-Time Replanning. In *Proceedings of the International Joint Conference on Artificial Intelligence*.

Sun, X., and Koenig, S. 2007. Fringe-Saving A* Search Algorithm: A Feasibility Study. In *Proceedings of the International Joint Conference on Arificial Intelligence (IJCAI)*, 2391–2397.

Šišlák, D.; Volf, P.; and Pěchouček, M. 2009. Accelerated a* path planning. In *The Eighth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009)*.

Yap, P. 2002. Grid-based path-finding. In *Proceedings of the Canadian Conference on Aritificial Intelligence*, 44–55.

Yershova, A.; Jaillet, L.; Siméon, T.; and LaValle, S. M. 2005. Dynamic-Domain RRTs: Efficient Exploration by Controlling the Sampling Domain. In *Proceedings IEEE International Conference on Robotics and Automation*, 3867–3872.