A* pathfinding algorithm modification for a 3D engine

Jakub Smołka[1,*], *Kamil* Miszta[2], *Maria* Skublewska-Paszkowska[1], and *Edyta* Łukasik[1]

[1]Lublin University of Technology, Electrical Engineering and Computer Science Faculty, Institute of Computer Science, Nadbystrzycka 36b, 20-618 Lublin, Poland
[2]Sii Polska, Szeligowskiego 6, 20-883 Lublin, Poland

Abstract. Frequently the quality of a path returned by a pathfinding algorithm is more important than the performance of that algorithm. This paper presents a new algorithm, based on A*, which is better suited for use in 3D game engines. The modification was evaluated by a series of comparative tests. The standard A* algorithm was used as a benchmark in the comparisons. The changes in the algorithm consist in using a different heuristic, adding vertex penalties, and post-processing of the path. A custom-built 3D graphics engine was used as the test environment. The paths generated by the new algorithm are a more natural choice for humans than the ones selected by the standard A* algorithm.

1 Introduction

Computer games are currently increasing in complexity. They need not only more computing power to store and render complex objects but also artificial intelligence. AI is needed for generating comments (for digital commentators), for creating rivals for the player, and for making the game world more realistic [1]. This last aspect is just what pathfinding algorithms are suitable for.

This paper presents a modification of the A* algorithm. Due to the fact that it is a very popular algorithm, there are many proven ways to modify it in terms of performance, reducing the number of traversed vertices and execution time. An example is the REA* algorithm described in more detail in [2]. Many improvements affect the optimisation of pathfinding, and not the path itself. This paper focuses on "imitating human behaviour" [3], that is, improving the algorithm so that the resulting route is close to the one that would be chosen by a human (that will be referred to as the natural path throughout this paper).

2 Review of A* modifications

In [4], algorithms for searching the shortest path in a grid graph are described. This paper is important not only because of the algorithm determining the shortest path by means of urban metrics, but also because of the graph structure itself. Grid graphs are commonly used for the generation of maps in graphical engines, which are usually rectangular (this significantly simplifies the rendering of such a map; more information in section 3.1). The problem is, however, the dynamic routing of the path, which does not allow for pre-determination of the route (a process important in a graphics engine).

In [5], a new heuristic for path searching algorithms (using A* as an example) is described. The work focuses on a new method for detecting vertices and dividing them into those that should be checked and those that should be omitted.

The study [6] describes the use of the A* algorithm in the path detection process for a moving robot. An important part of the work is the traversal of the vertices of the graph with respect to angles (Theta* [7] and JPS [6] algorithms). These algorithms should be used when near-real-time pathfinding is required. Real-time operation is possible at the expense of the quality of the generated route. However, this approach is not good for video games, where much higher computing power is available than in an average mobile robot.

In [8], a comparison of two popular pathfinding algorithms, *i.e.* A* and Basic Theta* is presented. These algorithms differ significantly in their implementation, which is the main reason for comparing them. Due to the fact that the Basic Theta* algorithm searches with respect to angles, the paths it generates are usually shorter. The A* algorithm, however, proved better in the case of searching through a smaller number of vertices.

In [2], two algorithms were analysed: A* and REA* (Rectangle Expansion A*). The REA* algorithm is a variation of the A* algorithm, which uses the division of the graph into smaller rectangles without blocked vertices (obstacles) [2]. REA* proved to be a faster and more efficient algorithm, selecting points on the map that optimise the length of the route while bypassing the traversal of certain vertices (which A* is incapable of). REA* works well in the case of maps with large, empty spaces and obstacles of regular shapes, while A* performs better in the case of maps with a complicated and irregular structure.

Paper [9] describes seven different path searching algorithms. A series of tests were carried out using 2D

* Corresponding author: jakub.smolka@pollub.pl

maps with obstacles, as well as with 3D maps composed of several 2D layers. The best algorithm in terms of computational complexity turned out to be Iterative Deeping A*.

The last paper [10] presents an introduction to finding the optimal path using genetic algorithms. The article focuses on various problems that should be solved in order to implement the search. Its goal is to demonstrate the applicability of genetic algorithms in pathfinding.

As can be seen, most of the articles are based on a review of existing algorithms or the creation of a new algorithm based on a popular algorithm, usually through a change of the heuristic function. These changes mainly concern the method of selecting vertices in the graph for traversal. Alternatively, there are attempts to modify the algorithm in order to make it work faster. Methods for improving path quality (reducing its curvatures for example) were, however, not found, as the referenced articles do not deal with the possibility of post-processing the path.

3 Pathfinding algorithms

There are numerous ways to move objects used in games. Game developers often create predetermined routes for given units, and thus simulate patrolling or just walking without purpose. Sometimes they give control to the player who chooses the path to be covered and include particular locations. Combinations of those solutions are also used that allow the player to determine the destination (while the route itself is determined by an algorithm).

3.1 Grid graph

Pathfinding algorithms work on graphs. Although they can employ any kind of graph, these are mainly grid graphs that are used [2]. Their use allows mapping a rectangular (usually square) area, which significantly facilitates map rendering. A grid graph is a variation of an undirected planar graph in which vertices can be assigned XY integer coordinates and are arranged in a rectangular (square) grid. The neighbours of a given vertex have coordinates that are equal or differ by one [4]. Depending on the type of the grid graph, a vertex can have up to eight neighbours.

In the remainder of the paper, all possible edges (including diagonal ones) are used; however, the paths and graphs are visualised without diagonal edges in order to improve their readability.

3.2 The A* algorithm

The review in Section 2 demonstrates that A* is one of the most popular and best-performing pathfinding algorithms. It is a best-first-search algorithm that favours the most promising paths. It exhibits the behaviour of GBFS (Greedy Breadth First Search) and Dijkstra algorithms. The creators of A* realised that traversing all possible vertices allows for finding the best possible path, but, for more complex graphs, the time needed for that is too long. On the other hand, using heuristics significantly accelerates path detection, but often gives suboptimal results, especially in the case of a large number of obstacles. For this reason, the creators of A* decided to combine these two solutions into the optimal one [11].

4 The operation of a 3D graphics engine in the context of searching for a path

This section presents the environment in which the research was carried out. It is a custom 3D engine created for educational purposes in Java, based on OpenGL libraries. The engine allows for loading Wavefront objects (.obj) and putting them programmatically into the game scene. The terrain can be generated randomly or from a heightmap (see Section 4.2.). Water, shadows and basic physics have been implemented. It is also possible to create a basic graphical user interface (GUI).

4.1 Object representation of the graph

Pathfinding algorithms are designed for a 2D space where the player moves only on the surface. The surface can be thought of as a projection of a three-dimensional terrain. The third dimension (height) is omitted when searching for paths (the graph has no vertices below or above the surface). It is important to note that, when adjacent vertices are at different heights, the resulting path is longer. Therefore, a mechanism that takes elevation differences into account should be implemented. This mechanism can be realised as penalties that are applied in the event of a significant difference in elevation. Thanks to this, the pathfinding algorithm will prefer the route with an even terrain, while avoiding larger elevations and valleys.

Considering the above facts, a vertex class is created, which consists of XZ coordinates (representing a 2D space), as well as the weight (value) which represents the Y coordinate (elevation). In addition, a vertex contains information about its priority (used for path determination) and whether it is passable. Edge weights are not needed.

The weights of individual vertices are determined on the basis of a heightmap in accordance with:

$$W_w = \frac{C_{Gpxl} \cdot W_{max}}{255} \quad (1)$$

where:
• C_{Gpxl} – pixel value (vertex elevation) in the heightmap (the heightmap is represented as a grayscale bitmap),
• W_{max} – maximum vertex height

4.2 Creating a 3D map

The heightmap is a special type of image consisting only of shades of grey [12]. The brightness of a given pixel determines how high the area will be at a specific point. By setting the minimum value corresponding to black and the maximum corresponding to white, each pixel can be mapped to the appropriate height and generate the appropriate map model. An example of mapping is shown in Figure 1.

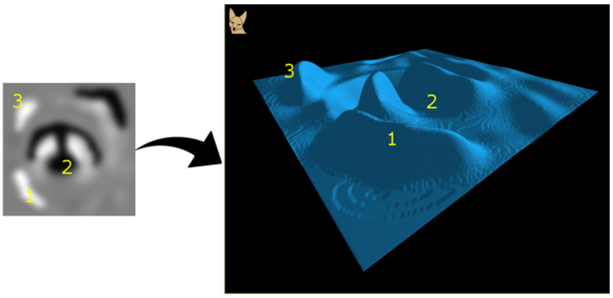


Fig. 1. Example of mapping of a grayscale image to a 3D terrain elevation.

4.3 Path representation on a 3D map

For testing purposes, the path should be displayed on a map. Thus a simple texture generator is created. By using texture blending (combining several textures into one, using a map that defines the position of textures), one can easily visualise the generated path, blocked points and the graph's granularity. An example of such visualisation is shown in Figure 2 (on the left).

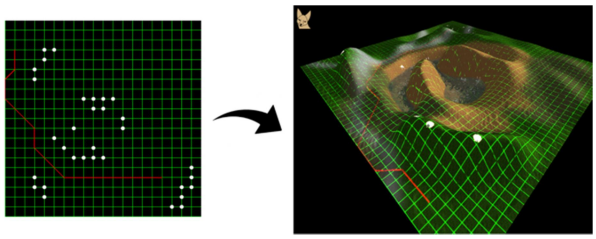


Fig. 2. Example of drawing a path on a 3D map (red line – optimal path, white dots – blocked vertices, green grid – graph's granularity).

5 Modification of the A* algorithm

In the algorithm's modification proposed in this paper referred later as A*MOD there are three significant changes, two of them related to each other.

5.1 Change of heuristics and introduction of vertex penalties

The heuristic "taxicab metric" is the preferred function of distance mainly when only moving vertically and horizontally. In the case of the default behaviour of the pathfinding algorithms the path can also be diagonal. This requires vertex weights to be higher on diagonals. In the described modification, a different heuristic is used – the Chebyshev distance (chess distance) [13]. It is given by the following formula:

$$L_D = \max(|A_X - B_X|, |A_Z - B_Z|) \qquad (2)$$

where:
• A_X, A_Z – coordinates of point A on a surface
• B_X, B_Z – coordinates of point B on a surface

Changing the metric causes the path to go along the diagonals more frequently (and look more natural to a human observer). However, it can also cause the path to zig-zag. To avoid this, vertex penalties are applied. A penalty is a multiplier included in the computer priorities of corner vertices (priority determines the order of processing vertices in the queue). The multiplier $\sqrt{2}$ is used (more details in subsection 5.3).

Sample results of using the modifications in question are shown in Figure 3.

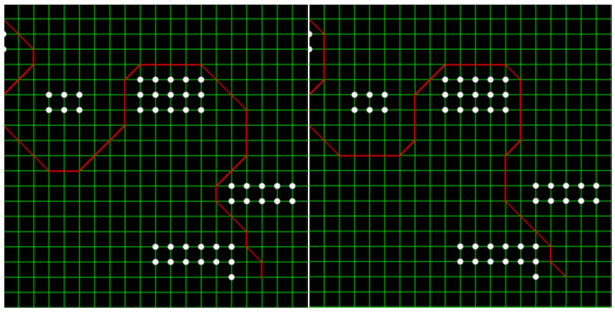


Fig. 3. Left: A* before heuristic change, Right: A* using Chebyshev metric.

5.2 Path post-processing

Another modification is the reduction of excess vertices in the path. Many vertices in the path are on the same straight line and are not important for the graphics engine (the 3D engine needs to generate smooth transitions in 3D space). Therefore, some vertices can be deleted. An example of the removal of unneeded vertices is shown in Figure 4 below.

In this case, the length of the resulting path is less than half in terms of the vertex count. However, one should remember two possible problems that can arise: (1) the vertices can be dynamically blocked – if any objects in the scene can block vertices, a post-processing step should check if the redundant vertices are not blocked, (2) The difference in height between two vertices is too large – the path being returned can be used to change the behaviour of a moving object, for example, in animation. In this case, too large a difference in the weight of the vertices may adversely affect the behaviour of the game.

These are cases that do not apply to the custom 3D engine in which the A*MOD algorithm has been implemented. It is worth having them in mind when implementing this algorithm in a different graphics engine.

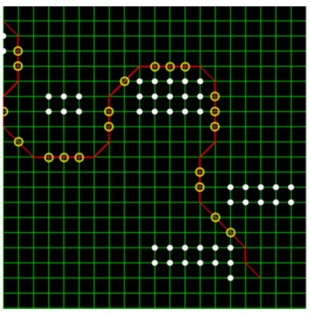


Fig. 4. Excess vertices are marked with a yellow circle.

5.3 Modified algorithm

The following list of steps describes of the proposed A*MOD algorithm:
1) Assign the start vertex to SV.
2) Assign the end vertex to EV.

3) Setup a map VM that will contain for each vertex a vertex that precedes it in the path.
4) Setup a map CM that will contain for each vertex a cost of reaching that vertex.
5) Add SV to the VPQ priority queue with priority equal to 0.
6) Add SV with the null value to VM.
7) Add SV with a value of 0 to CM.
8) As long as there are vertices in the VPQ priority queue, repeat steps 9-18.
9) Assign the first vertex from the VPQ to AV. Remove that vertex from VPQ
10) If AV is equal to EV, go to step 20.
11) Assign consecutive non-blocked neighbours of AV to NV, and repeat steps 11 - 13 for each vertex assigned to NV.
12) Set the cost of vertex NV as the cost of path from SV to NV. Save it in VC.
13) If NV is not in CM or VC<CM value for NV, execute steps 14-18.
14) Add NV with the value VC to CM.
15) If NV is a diagonal neighbour of AV, set M to $\sqrt{2}$ otherwise to 1
16) Set the priority of NV to priority of VC + heuristic(EV,NV) * M.
17) Add NV to the VPQ with the priority computed in steps 15-16.
18) Save the AV-NV path segment to VM.
19) End algorithm, return "NO_PATH"
20) Create list vertices of the route from EV to SV by reading the preceding vertices from VM. Save the result in VL.
--- POST-PROCESSING ---
21) Add the first vertex from VL to the list of PL.
22) For i equal 1 to length of (VL)-1 follow the steps 22-26.
23) Assign the last vertex from PL to P1.
24) Assign VL[i] to P2.
25) Assign VL[i+1] to P3.
26) If P2 is blocked or P1 and P3 are not on one line, add P2 to PL.
---
27) Reverse PL and return as a result.
(PATH_DETERMINED)

5.4 Tests

Fig. 5. Left: heightmap for test 1, name: uniform, grid size: 20x20, Right: heightmap for test 2, name: unified track, grid size: 20x20.

In order to analyse the differences between the operation of individual algorithms, 10 test cases were created. Both algorithms were subjected to the same tests. Heightmaps and grid graph sizes are shown in Figures 5-9. The sizes of individual graphs were assumed to be the minimal sizes possible to ensure proper operation of the algorithm.

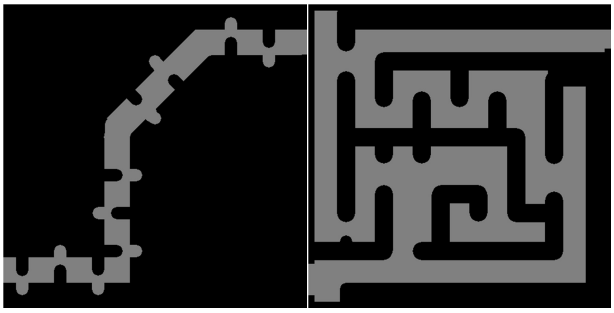


Fig. 6. Left: heightmap for test 3, name: uniform track with obstacles, grid size: 20x20, Right: heightmap for test 4, name: simple labyrinth, grid size: 20x20.

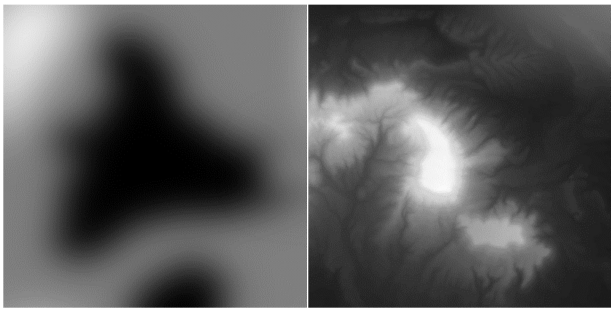


Fig. 7. Left: heightmap for test 5, name: simple heightmap, grid size: 20x20, Right: heightmap for test 6, name: complex heightmap, grid size: 40x40.

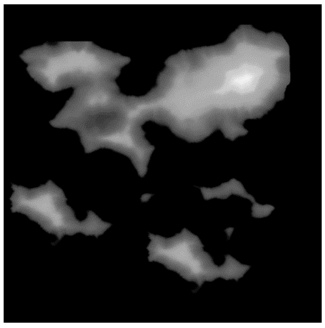


Fig. 8. Heightmap for tests 7 and 8 name: game map, grid size for test 7: 20 x 20, grid size for test 8: 40 x 40.

Fig. 9. Left: heightmap for test 9, name: complex labyrinth, grid size: 80x80, Right: heightmap for test 10, name: real photo 40x40.

Tests 1-4 check the behaviour of algorithms (A* and A*MOD) in a 2D space. The purpose of these tests is to

check whether the algorithms are able to find a path on a plane, avoiding obstacles. Tests 5-8 check the behaviour of algorithms in various (simple and complex) heightmaps. Test 9 checks the operation of the algorithms in a more complex case, when it is necessary to increase the granularity of the graph. The last test checks the behaviour when using a real photo as a heightmap instead of a specially prepared heightmap.

The results and their analysis are presented in the following subsection. The following parameters were computed for resulting paths: the lengths of paths, the lengths of the post-processed paths, the costs of paths (equation 3), the number of traversed vertices, and the average execution times (average of 1000 path searches). The path cost was determined as the sum of the differences of the successive vertices weights in the path:

$$C_P = \sum_{i=0}^{n-1}(W_{i+1} - W_i) \cdot K_W \qquad (3)$$

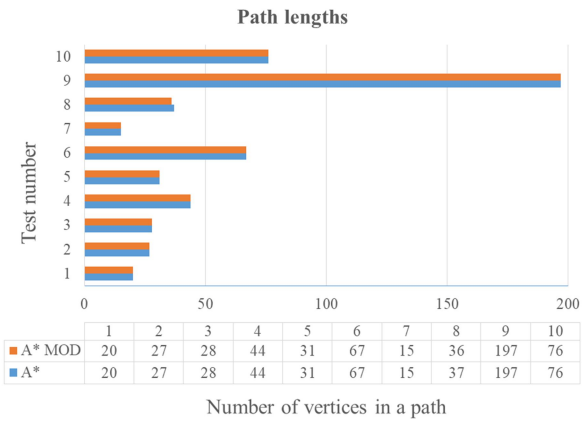
where:
• W_i – the weight of the i-th vertex in the path
• K_W – penalty multiplier $\sqrt{2}$ used if the i-th vertex in the path and its successor are diagonal (otherwise it is equal to 1)

5.5 Analysis of the results

Each of the following graphs presents one of the parameters compared in the tests.

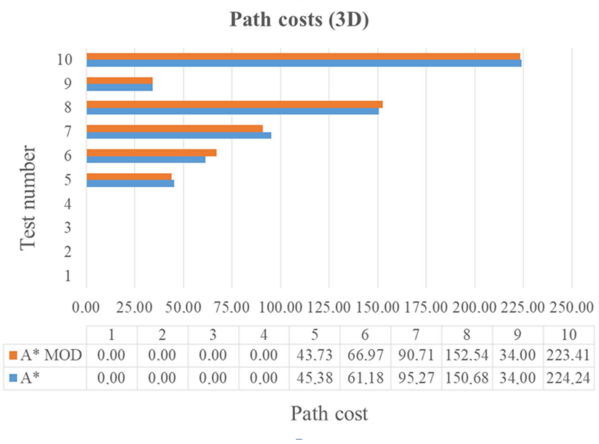
The graph in Figure 10 shows lengths of paths generated by the A* and A*MOD algorithms. The algorithms almost always return a path with the same number of vertices; however, the path created by A*MOD has less bent edges, which allows for better post-processing (as shown in the example in Figure 3).

The next graph (Figure 11) shows the path costs, which take the third dimension into account. Here too, the values are very similar to each other. The minimal differences show that a different path is generated for each algorithm. It also confirms that the introduced modification did not deteriorate the quality of the created path.



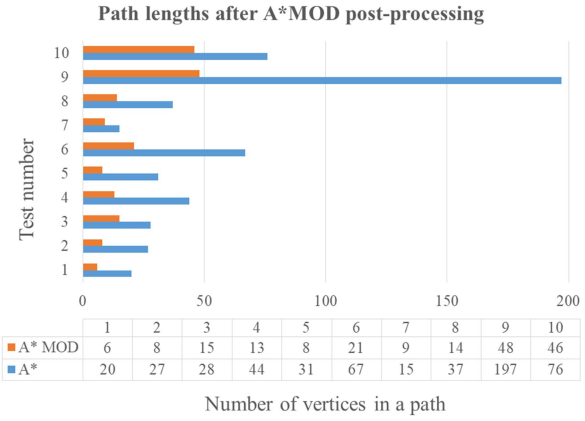
	1	2	3	4	5	6	7	8	9	10
A* MOD	20	27	28	44	31	67	15	36	197	76
A*	20	27	28	44	31	67	15	37	197	76

Fig. 10. Lengths of paths in individual tests.



	1	2	3	4	5	6	7	8	9	10
A* MOD	0.00	0.00	0.00	0.00	43.73	66.97	90.71	152.54	34.00	223.41
A*	0.00	0.00	0.00	0.00	45.38	61.18	95.27	150.68	34.00	224.24

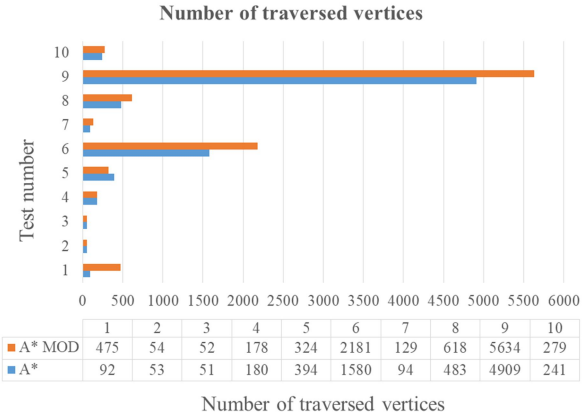
Fig. 11. Path costs in individual tests.



	1	2	3	4	5	6	7	8	9	10
A* MOD	6	8	15	13	8	21	9	14	48	46
A*	20	27	28	44	31	67	15	37	197	76

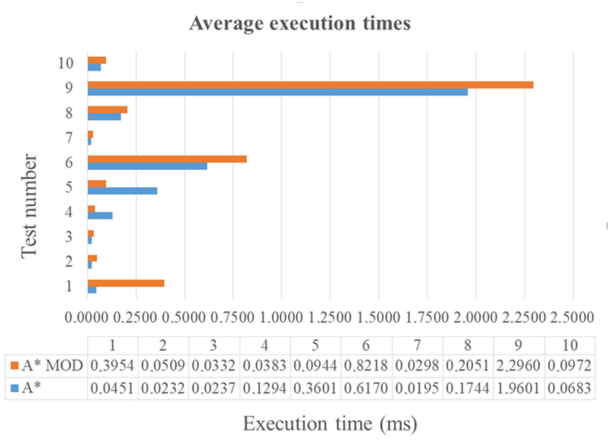
Fig. 12. Lengths of paths after post-processing in individual tests.

The graph in Figure 12 shows the improvement of the generated path after post-processing. This allowed us to shorten the path from the redundant vertices in each test. The results achieved in the tests range from 40% to over 75% of the removed vertices. Post-processing gives very good results in the case of larger graph granularity.



	1	2	3	4	5	6	7	8	9	10
A* MOD	475	54	52	178	324	2181	129	618	5634	279
A*	92	53	51	180	394	1580	94	483	4909	241

Fig. 13. Graph of the number of checked vertices in individual tests.



	1	2	3	4	5	6	7	8	9	10
A* MOD	0.3954	0.0509	0.0332	0.0383	0.0944	0.8218	0.0298	0.2051	2.2960	0.0972
A*	0.0451	0.0232	0.0237	0.1294	0.3601	0.6170	0.0195	0.1744	1.9601	0.0683

Fig. 14. Graph of the average time of operation of the algorithm in individual tests.

As the graph in Figure 13 shows, the modified algorithm must traverse a greater number of vertices in the graph to be able to generate a smoother path. In most cases, this is a small overhead ranging from 15% to 30%. In test 1, the number of vertices traversed by A*MOD algorithm is approximately five times higher than in the case of the original A* algorithm. This indicates that A*MOD has problems with "open" graphs without weights and blocked vertices that could limit the search for additional improvements to the path.

The last graph presented in Figure 14 shows how modifications affected the average execution time of the algorithms. Introduced modifications are mainly additional operations that the algorithm has to perform. In addition, post-processing extends the algorithm's execution time. Post-processing time is directly proportional to the length (in terms of vertex count) of the generated path. The additional overhead that occurs compared to the basic version of the A* algorithm is from 15% to 50%. The execution time is also dependent on the number of traversed vertices, and therefore in certain cases (such as test 4 and 5), the algorithm may execute faster.

6 Conclusions

The purpose of this article is to modify the A* algorithm to achieve a simplified, more "natural" route, *i.e.* to eliminate unnecessary bends in the path. These objectives were determined on the basis of the review in Section 2, which showed that modifications of the path-searching algorithms focus mainly on optimisation of the algorithm's execution time and not on the improvement of the path quality. Therefore, the presented modifications concern improving the path quality at the expense of the algorithm's operating time. For this purpose, the heuristic used by the algorithm was changed to the Chebyshev distance, and penalties were introduced for neighbours of the currently processed vertex. The modifications allowed for generating smoother, more natural routes without unnecessary bends. Path lengths and costs did not increase, and execution times increased by an acceptable percentage. A second introduced modification was path post-processing. Up to 75% of redundant vertices were removed in the conducted tests. A more natural shape of the route also positively affects post-processing since it allows for the removal of more vertices than would be possible with the basic version of the A* algorithm. The combination of the two modifications allowed for an improvement of generated paths in comparison to the base A* algorithm.

The proposed modifications can be used in conjunction with existing modifications of the A* algorithm, such as REA* or Jump Point Search, which focus solely on the algorithm's modification in terms of performance [2].

References

1. B. Kuźmińska-Sołśnia, T. Siwiec, *Zastosowania technologii informatycznych – teoria i praktyka*, 65-76, (2015)
2. Z. An, L. Chong, B. Wenhao, Chinese Journal of Aeronautics **29**, 5, 1385-1396, (2016)
3. S. Russell, P. Norvig, *Artificial Intelligence. A Modern Approach. Third Edition*, (Pearson Education, 2010)
4. F. Hadlock, Networks **7**, 323-334, (1977)
5. G. Mathew, *2nd International Conference on Electronics and Communication Systems (ICECS)*, 1651-1657, (2015)
6. F. Duchoň, A. Babinec, M. Kajan, P. Beňo, M. Florek, T. Fico, L. Jurišica, Procedia Engineering, **96**, 59-69, (2014)
7. P. Yap, N. Burch, R. Holte, J. Schaeffer, *AIIDE'11 Proceedings of the Seventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 201-207, (2011)
8. E. Firmansyah, S. Masruroh, F. Fahrianto, *6th International Conference on Information and Communication Technology for the Muslim World (ICT4M)*, 275-280, (2016)
9. K. Khantanapoka, K. Chinnasarn, *Eighth International Symposium on Natural Language Processing*, 184-188, (2009)
10. M. Gen, R. Cheng, D. Wang, *Proceedings of 1997 IEEE International Conference on Evolutionary Computation (ICEC '97)*, 401-406, (1997)
11. P. Hart, N. Nilsson, B. Raphael, IEEE Trans. of System Science and Cybernetics **4**, 100-107, (1968)
12. P. Singh, *OpenGL ES 3.0 Cookbook*, (Packt Publishing, 2015)
13. T. Kløve, T. Lin, S. Tsai, W. Tzeng, IEEE Transactions on Information Theory **56,** 2611-2617, **(**2010)