Efficient and Smoothed Pathfinding Report

Ana Costa 81105, André Fernandes 78076 and Rodrigo Lousada 81115

October 2018

1 Traditional A*

The first algorithm used for this project was A* algorithm. This algorithm find the least-cost path between an initial node to the goal node. To do so, it uses an distance-plus-cost heuristic function, this function is the sum of the path-cost and an admissible heuristic. In this case, the heuristic chosen was Euclidean Distance. The initial structure used was an unordered list, has it is possible to see on Figure 1, the processing time was slow. To improve, we implemented a dictionary, in this structure the key was the NavigationGraphNode and the value is a NodeRecord, for the Closed set.

2 Node Array A*

In this section we implemented the Node Array A* Algorithm, in this algorithm the node records are initially created for all nodes, from a fixed size Node Array is possible to access a given node, by is index, almost instantly. Each node has a record on which state is in: unvisited, open, closed. Its still necessary the list of Open nodes, to get the best ones. But the Closed list is no longer necessary, the algorithm only needs to check a node status instead of consulting a data structure.

3 Goal Bounding

Goal Bounding can be applied to any search space (grid, graph, NavMesh, etc.) using any search algorithm (Dijkstra, A*, JPS+, etc.). This search needs a pre-calculation of the goal bounds table, the table will store the bounds reachable by each edge of a node. Considering this information it will be used in the pruning strategy. After this offline calculation, the key point of the Algorithm is: he will only explore node/children if the goal node is on the same bounding box of the edge. So, a parent node will only expand if a children node is on his edge. Because of this a considerable number of nodes is immediately pruned decreasing the size and the speedup search.

4 Comparing the pathfinding algorithms

The three implemented algorithms were compared considering the following metrics: number of nodes visited, maximum open size, processing time (in ms) and time per node (in ms). We tested 5 different paths (pressing keys 1,2,3,4 and 5 of the keyboard) and the values are the mean of 5 tries. We can observe the results by the tables presented below:

Column1	Path 1	Path 2	Path 3	Path 4	Path5
Nodes Visited	135	346	793	852	5776
Max. Open Size	47	70	123	145	124
Processing Time (ms)	11.28	52.64	259.46	293.43	10293.24
Time per Node (ms)	0.0835	0.1521	0.3272	0.3444	1.7821

Figure 1: Performance of algorithm A*

Column1	Path 1	Path 2	Path 3	Path 4	Path5
Nodes Visited	135	346	793	852	5776
Max. Open Size	47	70	123	145	124
Processing Time (ms)	6.52	9.00	20.00	21.16	172.39
Time per Node (ms)	0.0483	0.0260	0.0252	0.0248	0.0298

Figure 2: Performance of algorithm A* using a Dictionary

Column1	Path 1	Path 2	Path 3	Path 4	Path5
Nodes Visited	135	393	831	913	5871
Max. Open Size	48	84	120	154	131
Processing Time (ms)	7.41	9.48	16.05	15.88	107.62
Time per Node (ms)	0.0549	0.0241	0.0193	0.0174	0.0183

Figure 3: Performance of algorithm Node Array A*

Column1	Path 1	Path 2	Path 3	Path 4	Path5
Nodes Visited	54	125	78	218	105
Max. Open Size	8	19	7	15	7
Processing Time (ms)	4.24	1.30	0.92	3.18	1.12
Time per Node (ms)	0.0784	0.0104	0.0118	0.0146	0.0107

Figure 4: Performance of algorithm Goal Bounding

By this, we can observe that the Goal Bounding algorithm is the fastest, since the nodes the number of nodes that are visited in this algorithms is reduced 40%, we can conclude that is also the cheapest regarding memory space used.

This algorithm outperforms the others because it only processes the child nodes that are, most certainly, in the goal path. By doing this, the number of explored nodes are less. The path where the process time is greater, using this algorithm, is the path 4 because of its complexity.

Comparing the performance of the A* algorithm using an unordered list and a dictionary, the processing time using the dictionary is reduced to more than half.

Analyzing the A* algorithm using Dictionary and Node Array A* algorithm tables, the Node Array performs better in the paths 3, 4 and 5. This can be related to the time spent per node, since with the Node Array it only needs to fetch the state of the node by its index instead of searching for it.

5 Path Smoothing

Considering the implementation of path smoothing, the main goal is to reduce the number of nodes by discarding the redundant nodes. The nodes are considered redundant if it's possible to connect two further nodes without crossing walls. This was achieved by, given a determined point, we consider three points further from the given point, or the endpoint if the index is out of range. We consider the possibility of having a collision in the middle of the path, if we don't identify any intersections, the path can be smoothed ignoring the middle point. We verify if there is no intersection by linear interpolation.

6 Optimizations

Regarding the optimization:

The first one we optimized the number of calls of the function isPointOnGraph on the function StraightLineSmoothing. On the beginning we had the folloing structure were A, B and C are calles to the function isPointOnGraph. Instead of making a comparison of the type

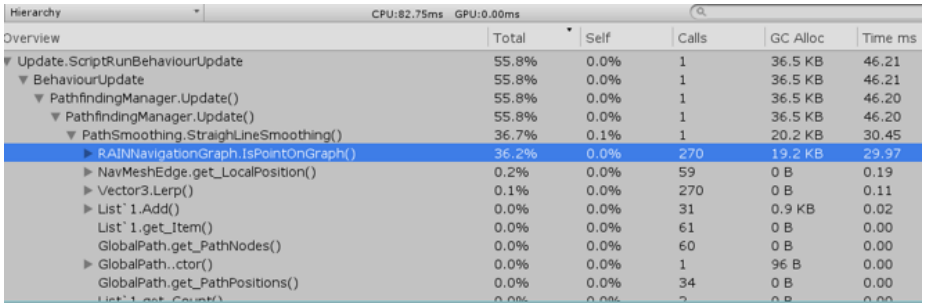
$$\neg(A \wedge B \wedge C) \tag{1}$$

we expanded the logical function to execute less comparisons.

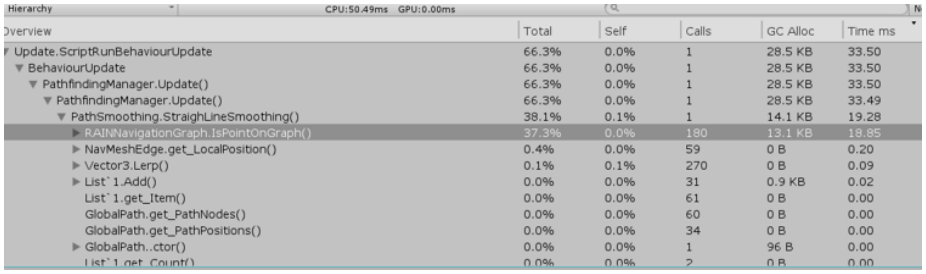
$$\neg A \vee \neg B \vee \neg C \tag{2}$$

By doing so, if A is true, it executes the condition of the comparison without verifying B nor C, it's possible to visualize on the Figure 6.

The second optimization we created a function to do a subtraction of two vectors. This optimization reduces the Time from 4ms to 2.3ms.

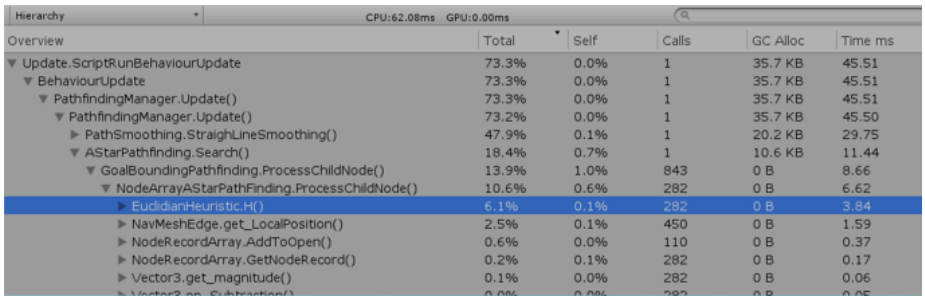


(a)

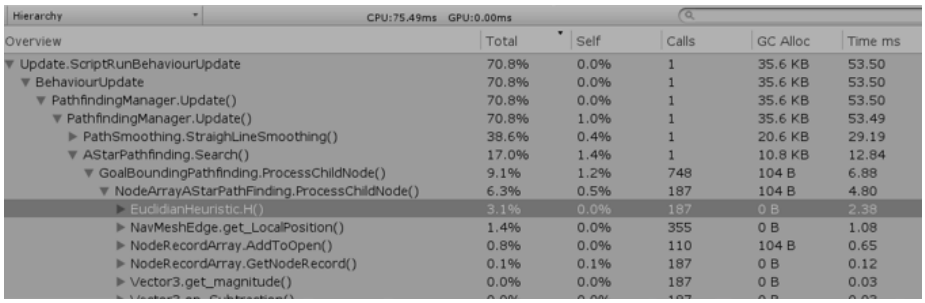


(b)

Figure 5: First Optimization (a) before the optimization and (b) after the optimization



(a)



(b)

Figure 6: Second Optimization (a) before the optimization and (b) after the optimization