INTELIGENCIA ARTIFICIAL PARA JOGOS

GROUP 22

Miguel Amaral 78865 Tiago Vicente 79620 Francisco Santos 79719

In games we often want to find paths from one location to another. In this project, we experimented and evaluated several algorithms used, in the video game industry, to deal with the Pathfinding problem.

Algorithms Comparison

For comparing the algorithms, we used the map in Image 1. There are two points, green on (191,0,191), as the start goal and red on (-112,0,106) as the goal. All algorithms were ran with the same setup:

- The mesh has 7879 nodes (only considering NavMeshEdge as those are the only ones used by the algorithms).
- 30 nodes analysed per frame.
- Each experiment was made 5 times and an average was calculated.

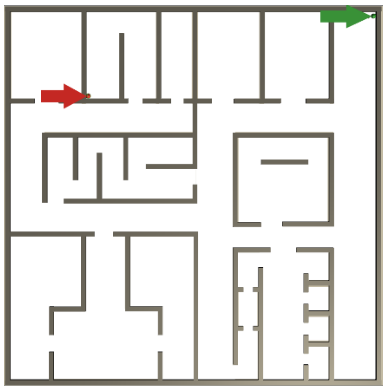


Figure 1 Map

This are the stats gathered:

A*	
Nodes Explored	6860
Fill	87.07%
Total Time (ms)	350,4
Time/ node (ms)	0.0601
Max Open Size	166

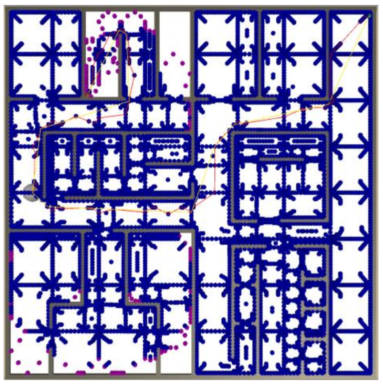


*Figure 2 A**

NodeArrayA*	
Nodes Explored	6860
Fill	87.07%
Total Time (ms)	159,34
Time/ node (ms)	0.0230
Max Open Size	166

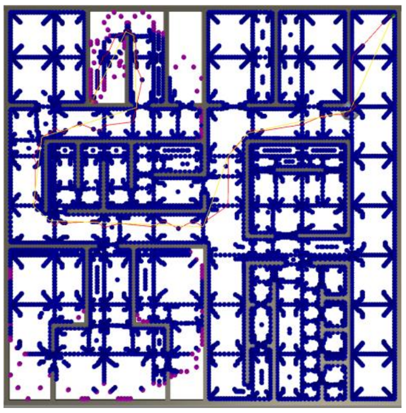


*Figure 3 NodeArrayA**

GoalBoundsA*	
Nodes Explored	142
Fill	1.80%
Total Time (ms)	3.056
Time/ node (ms)	0.0215
Max Open Size	5
Discarded Edges	744 (76%)
Visited Edges	230 (24%)
Total Edges	974

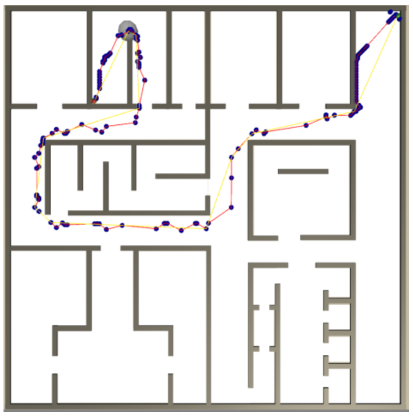


*Figure 4 GoalBoundA**

The difference between A* and NodeArrayA* is a trade-off between memory and time, therefore, ignoring the time stats, they have the same values, since it is the same algorithm running, but with different data structures. However due to the efficiency of the NodeArray we get a time decrease of 54.5%. The main advantage comes from the time it takes to access a node (O(1) in the NodeArrayA* vs O(n) in the A*).

There is a huge run time improvement between GoalBounding and the previous two algorithms (99.13% and 98.08% respectively). The gigantic decrease in time comes with a cost though, which is the pre-processing of the scene, storing it in the disk and later loading it to memory. This provides us with the foresight of knowing which directions will never lead us to the goal optimally, thus allowing the algorithm to ignore them. This results in the fill property to be much smaller (87.07% vs 1.80%) In the Figures 2, 3 and 4 we can see the fill property graphically (In blue the visited nodes and Purple the open nodes). We can realise that in GoalBound the algorithm almost only explores the nodes included the optimal path to reach the goal.

Path Smoothing

The algorithm used was the Straight-line Smoothing. It does not always return the smoothest path, but it does return a close enough and cheap smoothed path. We decided to run this algorithm 1 extra time, in order to have smoother paths, as sometimes some obstacles in the first run of the algorithm are no longer considered in the second one.

Optimizations

Offline Goal Bound Calculation

The offline goal bound table calculation is a crucial step to use this technique when the game is running. Furthermore, if the game level is changed or if a new level is created, there is the need to recalculate the table. This process takes time away from the development cycle, increasing time-to-market and making testability harder. Our optimization attenuates this problem, shortening the time to calculate and store the table.

In the computer used to evaluate the project, the original process took about 40 minutes whereas the optimized one took only about 3 minutes (92.5% improvement or 13 times faster). Additionally, the memory occupied in disk was shortened, it went from 5.5MB to 2.7MB (50% improvement or half the size). Although, disk storage nowadays is not a concern, the proportion can be interesting if the NavMesh starts to have much more nodes.

We made two optimizations to this process. First the Goal Bounds table calculation itself and secondly the storing of the table in disk.

Table Calculation

To optimize the calculation, we took advantage of the Dijkstra flooding being able to be parallelized. We launch a set of threads (number of the processor's logical cores) that execute this algorithm in some nodes, after the work is completed we feed the threads with new nodes to be processed until all nodes are calculated. We can see the difference in the following two images.

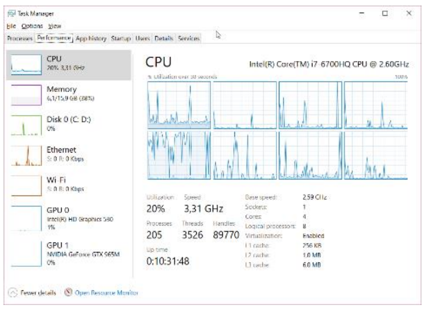


Figure 5 Non-threaded CPU usage

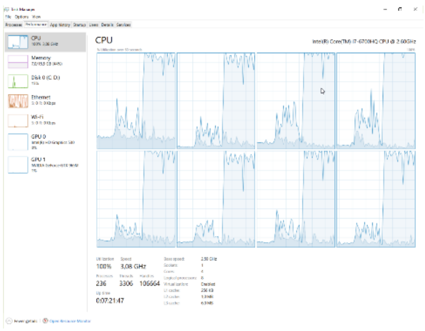


Figure 6 Threaded CPU usage

Storing the Table

Unity's serialization took a great amount of time to store the table in disk, we assumed that the reason for this to happen was because a game engine has to write small files to disk, like save game files, always giving priority to the game fluidity. Although, this approach can be great for video games, it is awful for storing something offline.

As a result, we implemented our own storing method using C# serialization mechanism. We convert the goal bounds table into objects that can be serialized by C# and store it. Vice-versa when loading.

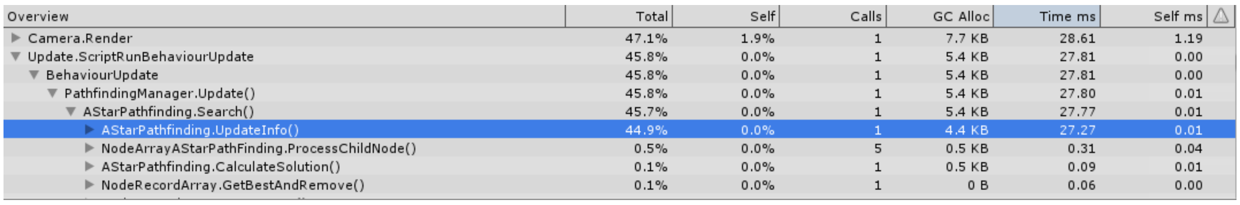
This technique has a very good performance, only taking 10 seconds to write to disk, however portability between .NET versions may be a problem, if the serialization mechanism changes. Besides, Unity no longer can interpret this file, since it is written in binary, making it impossible to edit it in the Assets menu. We do not see the last as a problem, because no one will edit manually a table with thousands of entries. Moreover, sometimes Unity freezes when opening the file.

Profiling induced optimization

Lastly, with the help of the Unity's profiler we discovered that our UpdateInfo function, that updates the information for debug purposes, was taking 98% of BehaviourUpdate's time. As such we took a deeper look and saw that we were, needlessly, traversing the entirety of the NodeRecordArray, in order to update the number of visited nodes. To fix this we simply update the integer when closing the nodes.

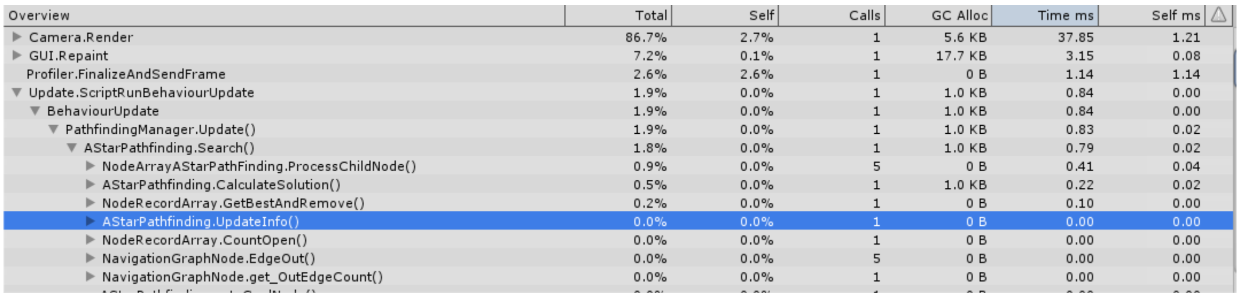
In the appendix we can see the profiling before and after this optimization (Figures 7 and 8).

Appendix



Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
▶ Camera.Render	47.1%	1.9%	1	7.7 KB	28.61	1.19
▼ Update.ScriptRunBehaviourUpdate	45.8%	0.0%	1	5.4 KB	27.81	0.00
▼ BehaviourUpdate	45.8%	0.0%	1	5.4 KB	27.81	0.00
▼ PathfindingManager.Update()	45.8%	0.0%	1	5.4 KB	27.80	0.01
▼ AStarPathfinding.Search()	45.7%	0.0%	1	5.4 KB	27.77	0.01
▶ AStarPathfinding.UpdateInfo()	44.9%	0.0%	1	4.4 KB	27.27	0.01
▶ NodeArrayAStarPathFinding.ProcessChildNode()	0.5%	0.0%	5	0.5 KB	0.31	0.04
▶ AStarPathfinding.CalculateSolution()	0.1%	0.0%	1	0.5 KB	0.09	0.01
▶ NodeRecordArray.GetBestAndRemove()	0.1%	0.0%	1	0 B	0.06	0.00

Figure 7 Profiling non-optimized



Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
▶ Camera.Render	86.7%	2.7%	1	5.6 KB	37.85	1.21
▶ GUI.Repaint	7.2%	0.1%	1	17.7 KB	3.15	0.08
Profiler.FinalizeAndSendFrame	2.6%	2.6%	1	0 B	1.14	1.14
▼ Update.ScriptRunBehaviourUpdate	1.9%	0.0%	1	1.0 KB	0.84	0.00
▼ BehaviourUpdate	1.9%	0.0%	1	1.0 KB	0.84	0.00
▼ PathfindingManager.Update()	1.9%	0.0%	1	1.0 KB	0.83	0.02
▼ AStarPathfinding.Search()	1.8%	0.0%	1	1.0 KB	0.79	0.02
▶ NodeArrayAStarPathFinding.ProcessChildNode()	0.9%	0.0%	5	0 B	0.41	0.04
▶ AStarPathfinding.CalculateSolution()	0.5%	0.0%	1	1.0 KB	0.22	0.02
▶ NodeRecordArray.GetBestAndRemove()	0.2%	0.0%	1	0 B	0.10	0.00
▶ AStarPathfinding.UpdateInfo()	0.0%	0.0%	1	0 B	0.00	0.00
▶ NodeRecordArray.CountOpen()	0.0%	0.0%	1	0 B	0.00	0.00
▶ NavigationGraphNode.EdgeOut()	0.0%	0.0%	5	0 B	0.00	0.00
▶ NavigationGraphNode.get_OutEdgeCount()	0.0%	0.0%	1	0 B	0.00	0.00

Figure 8 Profiling optimizer