# Artificial Intelligence for Games
# 2nd Delivery - Group 12

Gonçalo Marques, 84719 Inês Vilhena, 84593 Pedro Maria, 84618

## Introduction

This delivery consisted in developing Pathfinding algorithms, as well as a Path Smoothing algorithm, within a virtual 3D environment provided by Unity 3D.

## Level 1 - Traditional A*

This algorithm was implemented by following the instructions given to us in lab 3's guide. It will search through a pool of nodes in order to find the closest path from any given node to any other node. This is done by searching through adjacent nodes to already visited nodes and using a heuristic function combined with a cost function to choose which node should be explored next.
In our case, we used an Euclidean Distance Heuristic, which calculates the distance from a given node to the goal node.

The data structures used for the Open (visitable nodes) and Close (visited nodes) lists were respectively a Node Priority Heap and a Dictionary List. Node Priority Heaps have better insertion and extraction times than a traditional array, which is very useful given that those will be the most frequent operations on the Open List. Dictionary Lists have better search times than a traditional array, so they are much more useful for Close lists where searching is the most frequent operation.

## Level 2 - Node Array A*

Alike the traditional A* algorithm, Node Array A* was implemented following Lab 4's guide. This algorithm is very similar to the aforementioned one except for the existence of a list of a Node Record structure for each node, which holds information like the node's cost and status. This change allows testing whether a node is in the Open or Close state very fast (this variant implies that we use more memory in exchange for more speed) and improve the performance of the algorithm.

## Level 3 - Goal Bounding A*

Unlike the previous implementations, the Goal Bounding A* search algorithm was not implemented following a lab guide. The group started by implementing the Dijkstra Flood fill Algorithm to pre-calculate a group of bounding boxes and use them to find the shortest path from the starting node to the goal node. The algorithm is similar to the previously described Node Array A*, except for the addition of a condition checking whether the node that is about to be processed is inside the same bounding box as the goal node. If it is not, that node is discarded and the algorithm continues searching for valid nodes.

### Dijkstra Flood fill

This Dijkstra variant will be called once for every node in the graph. For each of the nodes adjacent to the starting node, a bounding box will be calculated in order to contain all the nodes that can be reached first starting from that node. At the end of the algorithm, the bounding boxes' dimensions are saved in a file and used later by the Goal Bounding algorithm during runtime.

## Level 4 - Comparing Different Path-finding Algorithms

### Lab Comparisons

Table 1: A* with unordered Open and Close lists

| Method | Calls | Execution Time (ms) |
|---|---|---|
| Search | 1 | 82.12 |
| GetBestAndRemove | 20 | 2.58 |
| AddToOpen | 26 | 0.02 |
| SearchInOpen | 104 | 11.65 |
| RemoveFromOpen | 0 | 0 |
| Replace | 5 | $> 0.001$ |
| AddToClosed | 20 | 0.02 |
| SearchInClosed | 68 | 65.6 |
| RemoveFromClosed | 0 | 0 |

Table 2: A* with unordered list for Open and Dictionary list for Close

| Method | Calls | Execution Time (ms) |
|---|---|---|
| Search | 1 | 15.54 |
| GetBestAndRemove | 20 | 2.64 |
| AddToOpen | 23 | 0.01 |
| SearchInOpen | 88 | 10.69 |
| RemoveFromOpen | 0 | 0 |
| Replace | 3 | $> 0.001$ |
| AddToClosed | 20 | 0.03 |
| SearchInClosed | 62 | 0.17 |
| RemoveFromClosed | 0 | 0 |

Table 3: A* with Node Priority Heap for Open and Dictionary list for Close

| Method | Calls | Execution Time (ms) |
|---|---|---|
| Search | 1 | 6.26 |
| GetBestAndRemove | 20 | 0.49 |
| AddToOpen | 22 | 0.08 |
| SearchInOpen | 102 | 3.7 |
| RemoveFromOpen | 0 | 0 |
| Replace | 4 | 0.12 |
| AddToClosed | 20 | 0.02 |
| SearchInClosed | 75 | 0.16 |
| RemoveFromClosed | 0 | 0 |

Table 4: Node Record Array A*

| Method | Calls | Execution Time (ms) |
|---|---|---|
| Search | 1 | 9.22 |
| GetBestAndRemove | 13 | 0.45 |
| AddToOpen | 51 | 1.06 |
| SearchInOpen (node state check) | 0 | 0 |
| RemoveFromOpen | 0 | 0 |
| Replace | 42 | 6.89 |
| AddToClosed | 12 | $> 0.001$ |
| SearchInClosed (node state check) | 0 | 0 |
| RemoveFromClosed | 0 | 0 |

## Project Comparisons

Table 5: A* Bench marking

| Name | Value |
|---|---|
| Fill | 5563 |
| Nodes Visited | 5654 |
| Total Processing Time (ms) | 6771.45 |
| Processing Time Per Node (ms) | 1.1976 |
| Memory | 1GB |

Table 6: Node Record Array A* Bench marking

| Name | Value |
|---|---|
| Fill | 5564 |
| Nodes Visited | 5655 |
| Total Processing Time (ms) | 6554.58 |
| Processing Time Per Node (ms) | 1.1591 |
| Memory | 1GB |

Table 7: Goal Bounding A* Bench marking

| Name | Value |
|---|---|
| Fill | 15 |
| Nodes Visited | 104 |
| Total Processing Time (ms) | 104.88 |
| Processing Time Per Node (ms) | 1.0085 |
| Memory | 1.64GB |

## Conclusion

The Goal Bounding A* algorithm is by far the most efficient out of the 3, due to the fact that it can rely on much more precise pre-calculated data than the other 2 variants, which are discovering data as they run. This precise information (bounding boxes that contain the goal node) help make much better decisions when it comes to which node to process, and make the algorithm much more efficient.

## Level 5 - Path Smoothing

The implemented path smoothing algorithm uses the calculated path nodes and checks whether it is possible to go from one node to the node 2 positions ahead in a straight line for every node in the path (except for the last 2 nodes). If it is, it will remove the middle node (1 position ahead) from the path and do the same test, until it has reached the end of the path.

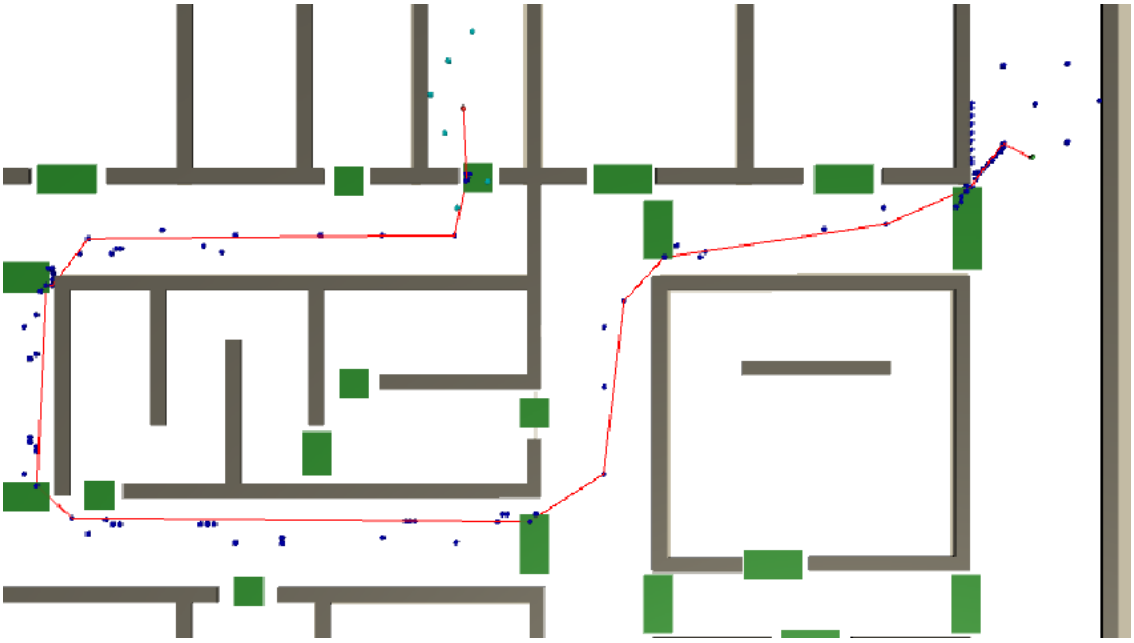

Figure 1: The Path Smoothing Algorithm visualized

## Level 6 - Optimization's

### Memory

We went through all the scripts in the project, checked whether variables were being used more than once and, if they weren't, we fed them directly where necessary. This saved a considerable amount of memory during the algorithms' execution time.

### Performance

In the Node Array A* Pathfinding algorithm, given that a consistent heuristic is used (Euclidean Distance), when processing a child node there is no need to consider closed nodes, which noticeably increases performance. Another small optimization we did was saving the float comparison value of the generated and previous g-value for each node, instead of comparing floats multiple times. This slightly increased performance when crossing long paths.