TÉCNICO LISBOA

Artificial Intelligence in Games
Project 2 - Efficient and smoothed pathfinding

Grupo 19

1. Implemented Features

Level 1– Traditional A*: **Implemented.**
Level 2– Node Array A*: **Implemented.**
Level 3 – Goal Bounding : **Implemented but not working properly.**
Level 4 – Comparing the pathfinding algorithms: **Implemented.**
Level 5 – Path Smoothing: **Implemented.**
Level 6 – Optimizations: **Implemented 1 Optimization.**

2. Implementation & Design Decisions and Commentaries

Tradicional A*

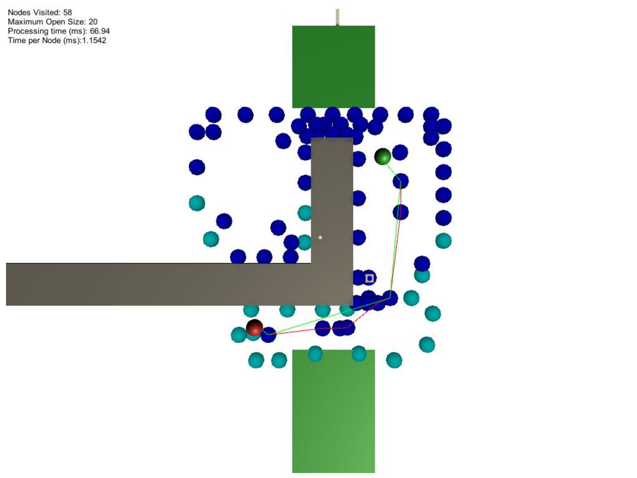
In the tradicional A* we use an Hash Table as a data structure for the closed set and a Priority Heap for the open set. For the heuristic we use the Euclidean heuristic, which is a simple yet consistent heuristic.
Therefore, in the implementation of the algorithm, when we process a ChildNode, we do not test if the node is in the closed set with a larger FValue, removing it from the closed set and adding it to the open set, in that case, since the heuristic is consistent. And we choose not to test the case when there are ties between nodes with the same FValue, since there are no significant changes in the processing time neither in the processed nodes as this is not a grid-based environment with several ties possible.

Node Array A*

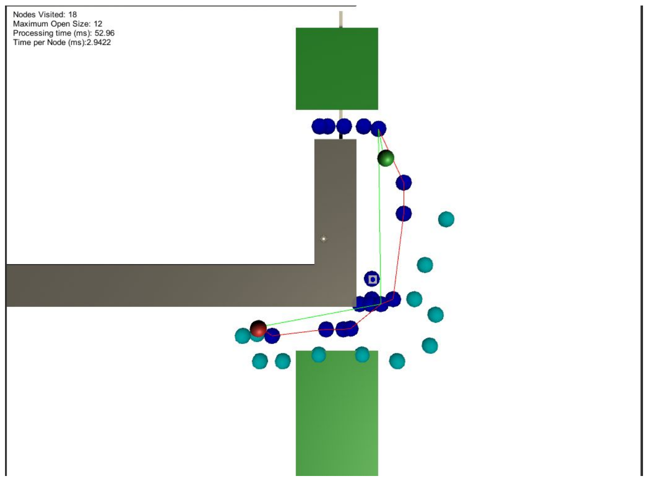
Using a node array A* pathfinding we are trading memory performance with how much time the algorithm uses to find the best path. In this type of pathfinding the structure of the node record changes in comparison with A* pathfinding, now we have categories for each of the node record (unvisited, open or closed).
One aspect in favour is that we don't need to build one structure to keep the closed list. Another one is that we use a priority heap for the open node, we only need this open data structure because we still need to order the nodes to get the best ones, but because we are using a priority heap we still get great results.

Goal Bounding

Given that the Goal Bounding pathfinding is not working properly, it is hard to discuss implementation decisions given that one of the components of the algorithm is not working, and we are not exactly sure which one. For some, particularly smaller paths, goalBounding seems to work well, as can be illustrated in a smaller navmesh example we tested the algorithm in:



```
Nodes Visited: 58
Maximum Open Size: 20
Processing time (ms): 66.94
Time per Node (ms):1.1542
```



```
Nodes Visited: 18
Maximum Open Size: 12
Processing time (ms): 52.96
Time per Node (ms):2.9422
```

On the left we can see the regular A* exploration pattern using the euclidean heuristic, and on the right we have the GoalBounding Pathfinding algorithm with the same heuristic, which shows a much narrower exploration pattern, hinting that several edges were discarded. Given the randomness of the algorithm's problem, we propose that one of two things can be happening:

- The Bounding Table is malformed, perhaps due to incorrect updates to the bounds due to a bug in the Dijkstra flooding, and edges that should not have been discarded are being so, making the algorithm stall without any further nodes to open.
- The Bounding table is well-formed, but the access that is being performed to the table does not correspond to the correct edge that should be processed, which leads to some situations where the algorithm works due to random chance, but most situations where it doesn't. Given that the index.NodeIndex entry is being used in both the generation of the table and its access, indexing alone doesn't seem to be the issue.

Path Smoothing

To smooth the path returned by the pathfinding algorithm we based ourselves on the path post processing technique for grids, Straight-line Smoothing. We apply this algorithm 1/10 times the number of nodes in the original path, to achieve a good balance between maximum nodes discarded and unnecessary calls. And on Draw, we print the smoothed path in green. The implementation itself follows what is shown in the theoretical classes.

3. Comparison of Pathfinding Algorithms and Lab 4 Table

A* Pathfinding with Euclidean distance - **Unordered List for Open and Closed** (15 nodes per frame)

Metrics	Path 1	Path 2	Path 3	Path 4	Path 5
Nodes visited	237	873	1748	1890	6794
Maximum open size	55	92	145	171	186
Total processing time (ms)	252.78	1171.17	3157.29	3570.92	32965.56
Processing time per node (ms)	1.0666	1.3416	1.8062	1.9005	4.8522

Result for path 4 (50 nodes per frame):

Unordered list for Open and Closed		
Method	Calls	Execution Time
A*Pathfinding.Search	1	309.93
GetBestAndRemove	51	7.11
AddToOpen	41	0.05
SearchInOpen	261	30.05
RemoveFromOpen	-	-
Replace	21	0.00
AddToClosed	50	0.07
SearchInClosed	261	252.45
RemoveFromClosed	-	-

A* Pathfinding with Euclidean distance - **Dictionary for Closed** (15 nodes per frame)

Metrics	Path 1	Path 2	Path 3	Path 4	Path 5
Nodes visited	237	873	1748	1890	6794
Maximum open size	55	92	145	171	186
Total processing time (ms)	252.74	1009.24	2522.42	2805.08	20776.89
Processing time per node (ms)	1.0664	1.1561	1.4430	1.4842	3.0581

Result for path 4 (50 nodes per frame):

Dictionary for Closed		
Method	Calls	Execution Time
A*Pathfinding.Search	1	52.46
GetBestAndRemove	51	0.01
AddToOpen	37	0.01
SearchInOpen	248	27.87
RemoveFromOpen	-	-
Replace	21	0.00
AddToClosed	50	0.00
SearchInClosed	248	0.06
RemoveFromClosed	-	-

We can see here how using the Dictionary for closed has a tremendous impact in performance. Instead of searching in a list for a closed node (which grows as the number of nodes visited increases), it can perform that access much faster as it is Θ(1) vs Θ(n) search.

A* Pathfinding with Euclidean distance - **Node Priority Heap for Open** (15 nodes per frame)

Metrics	Path 1	Path 2	Path 3	Path 4	Path 5
Nodes visited	237	873	1748	1890	6794
Maximum open size	55	92	145	171	186
Total processing time (ms)	257.52	1012.49	2320.89	2783.39	20094.83
Processing time per node (ms)	1.0866	1.1598	1.3277	1.4727	2.9577

Result for path 4 (50 nodes per frame):

Node Priority Heap for Open		
Method	Calls	Execution Time
A*Pathfinding.Search	1	30.82
GetBestAndRemove	51	1.59
AddToOpen	36	0.26
SearchInOpen	260	9.12
RemoveFromOpen	-	-
Replace	16	0.00
AddToClosed	36	0.00
SearchInClosed	260	0.31
RemoveFromClosed	-	-

We can see that the addition of priorities to the Open set has greatly sped up the search time in open, resulting in the most significant difference between this and the previous example.

Node Array A* Pathfinding with Euclidean distance (15 nodes per frame)

Metrics	Path 1	Path 2	Path 3	Path 4	Path 5
Nodes visited	238	874	1749	1891	6795
Maximum open size	55	92	145	171	186
Total processing time (ms)	252.74	989.27	2388.75	2705.94	19441.45
Processing time per node (ms)	1.0619	1.1319	1.3658	1.4310	2.8611

Result for path 4 (50 nodes per frame):1

Node Array A*		
Method	Calls	Execution Time
A*Pathfinding.Search	1	19.35
GetBestAndRemove	51	1.48
AddToOpen	37	0.23
SearchInOpen	248	0.02
RemoveFromOpen	-	-
Replace	21	0.76
AddToClosed	50	0.00
SearchInClosed	248	0.03
RemoveFromClosed	-	-

In terms of nodes visited, and fill. Both NodeArrayA* and A* perform similarly, as NodeArray's advantages are in access times to nodes. As such, NodeArrayA* has lower total processing time, as well as processing time per node. Although we cannot test GoalBounding and compare it in the given examples, the main advantage of GoalBounding with regards to tradicional A* is due to its guided search pattern with the use of bounding boxes, which allows the discarding of edges that won't culminate in the goal solution. This ultimately improves upon tradicional A* as long as bounding table checks are faster than the cost of exploring the additional nodes. As a result, GoalBounding will have a slightly higher computing cost per node (The table check overhead), but lower total process time.

4. Optimizations

Fast Heuristic Calculation

In the analysis of the deep profiling execution calls, we noticed that the heuristic call was taking up a large slice of the execution time. Upon inspection, the reason was due to the fact that node positions are being accessed 8 times for each heuristic calculation. As such we extended the heuristic interface to include a Fast_H method, which returns the same result in the euclidean heuristic, but uses much less computation time by preloading the positions for calculation. The following table demonstrates the results:

Node Array A* time comparison (ms)		
Method	**H**	**Fast_H**
A*Pathfinding.Search	20.82	11.26
ProcessChildNode	18.88	8.64
HeuclideanHeuristic	14.87	3.99
Total Process Time (Reported in GUI):	2011.54	1583.18

The results are very notorious with deep profiling on, but gains are visible even without it.