<u>AI Pathfinding Project</u>

<u>Introduction</u>

In this project, we used various pathfinding algorithms to trace the path to a target. Each algorithm leaves a different colored trail behind it. The goal of this pathfinding agent is to find a target without encountering any obstacles and without inadvertently creating an obstacle.

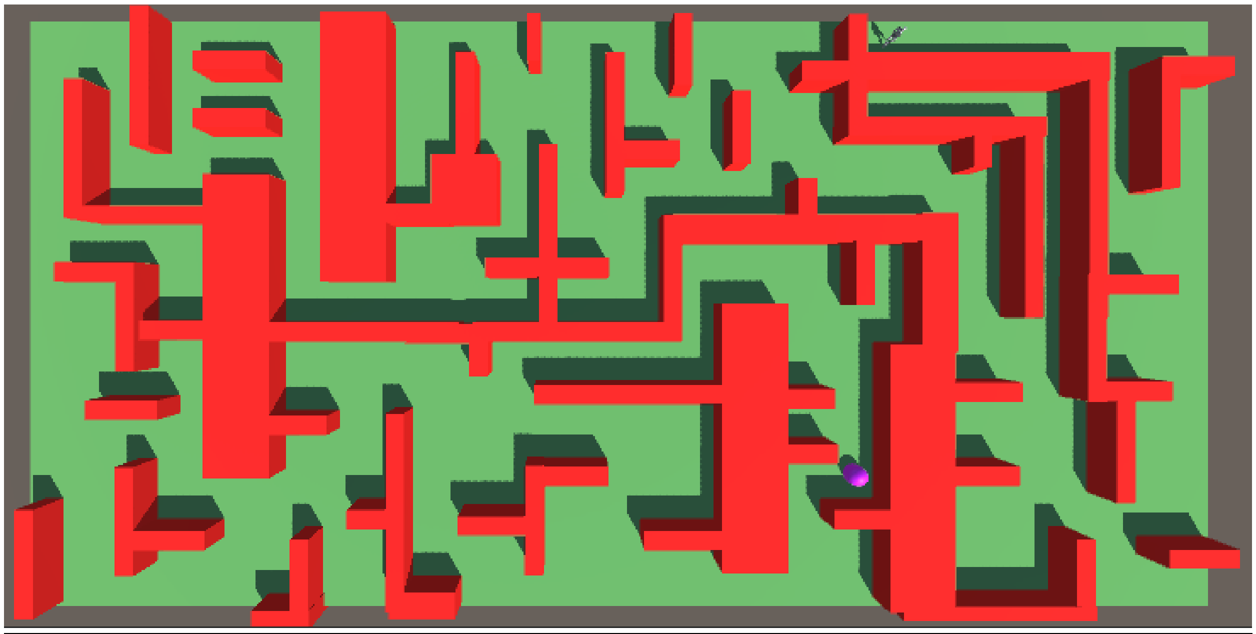
We started with a graph that contained none of these nodes.

Add a few randomly placed nodes to make it look like one with impassable obstacles, and then run all five algorithms on this graph.

The algorithm with the shortest average distance traveled and the least amount of time complexity wins.

The robot in the upper right corner of the images must locate the purple capsule, and the generated squares represent the path found by the different algorithms

<u>Description of the environment</u>



The environment is a complex maze with a seeker represented by a robot in the top right and a target represented by a purple capsule. At first glance, it appears that there are multiple paths to the target; each algorithm will select the path that he finds best. The robot can leave the green ground because the red walls are impassable. These are the environment's ground rules; let's see how the simulation turns out.

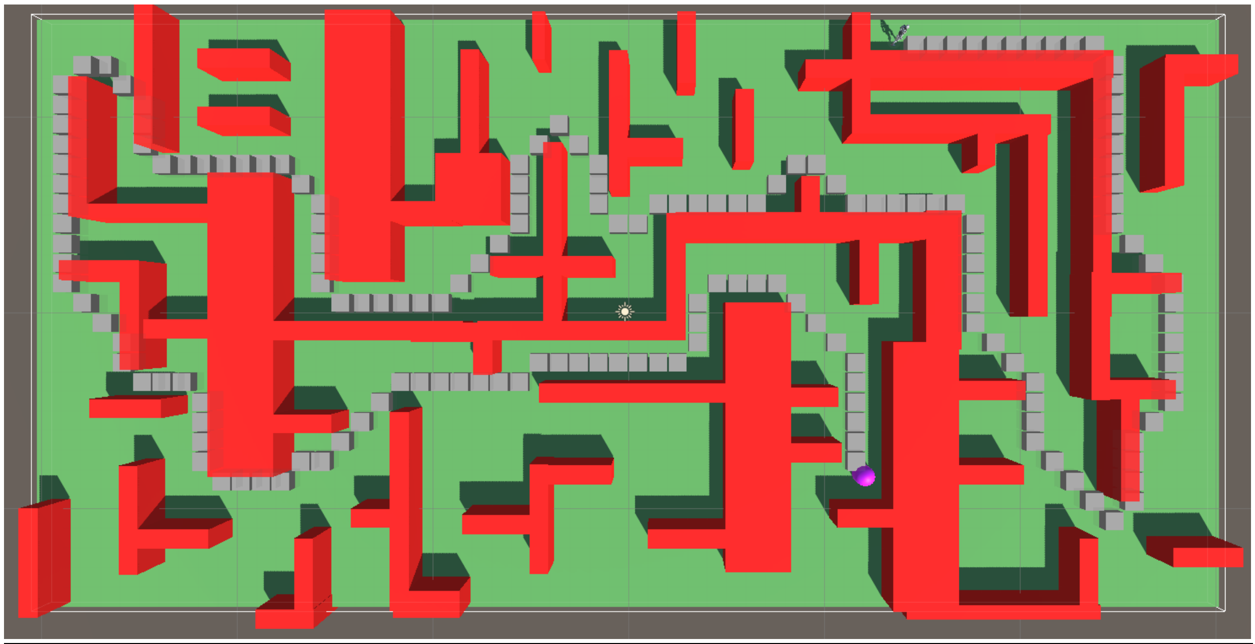
Result

Here is the result of the first two algorithms which are tuned for solving this maze:

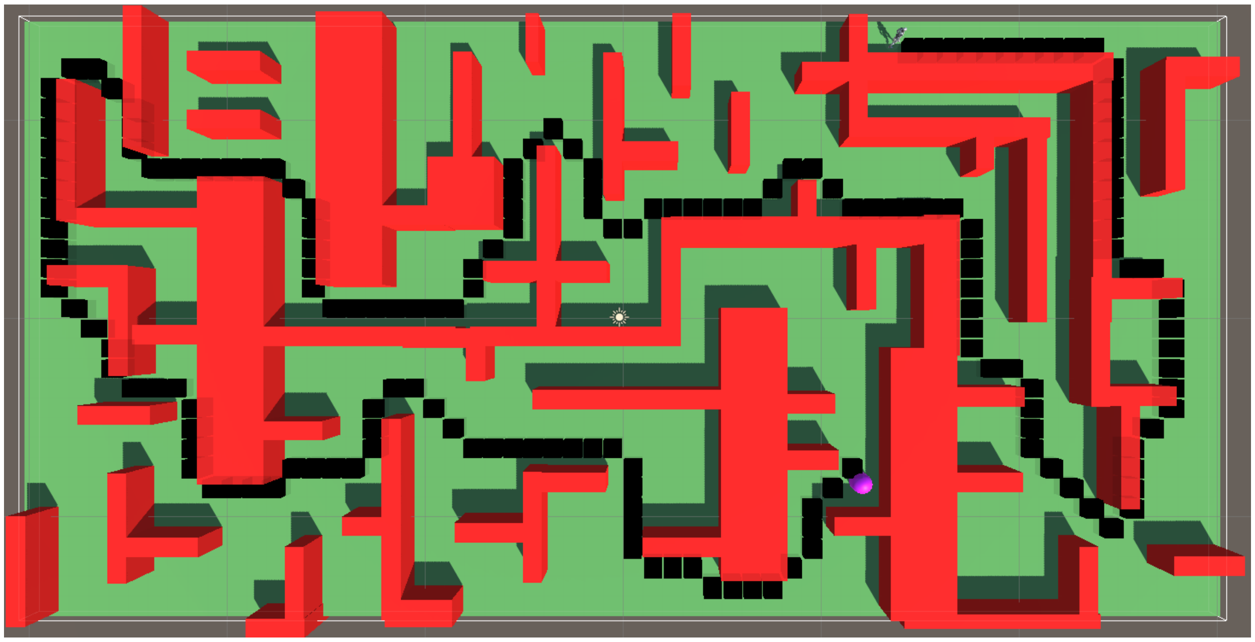
A* (Euclidian) and A*(Manhattan)

These are the same algorithms but using different heuristics to calculate the distance between the goal and the seeker: Euclidian Method and Manhattan Method.

A* Euclidian:

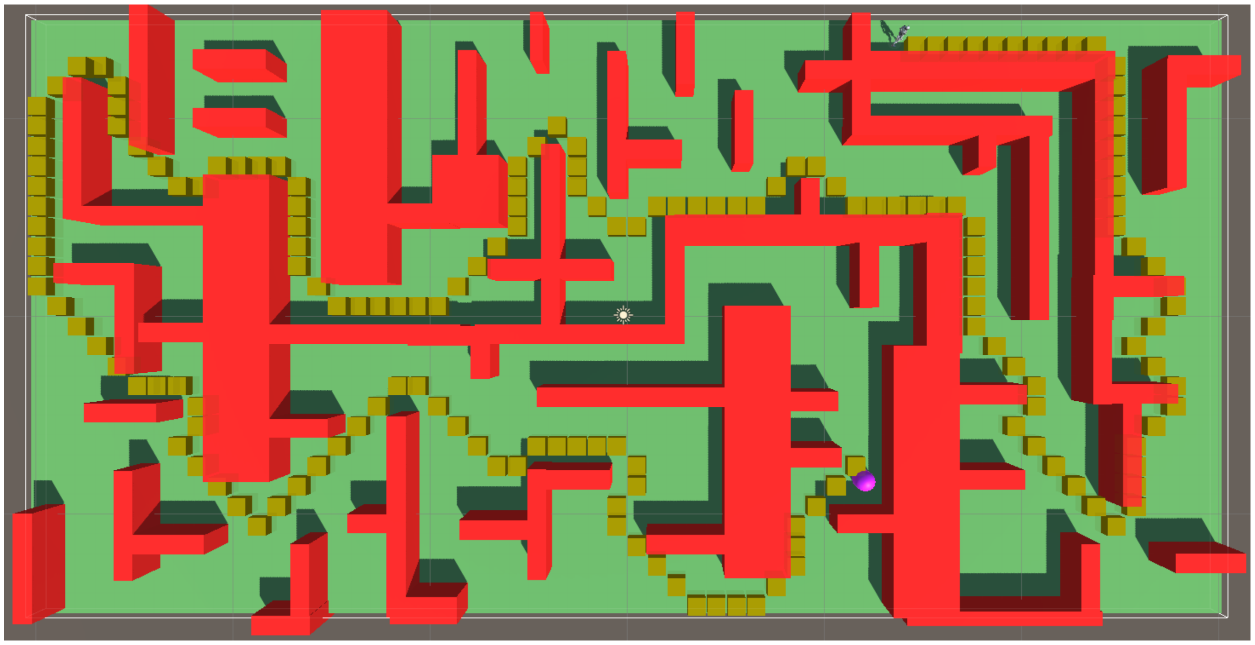


A* Manhattan:

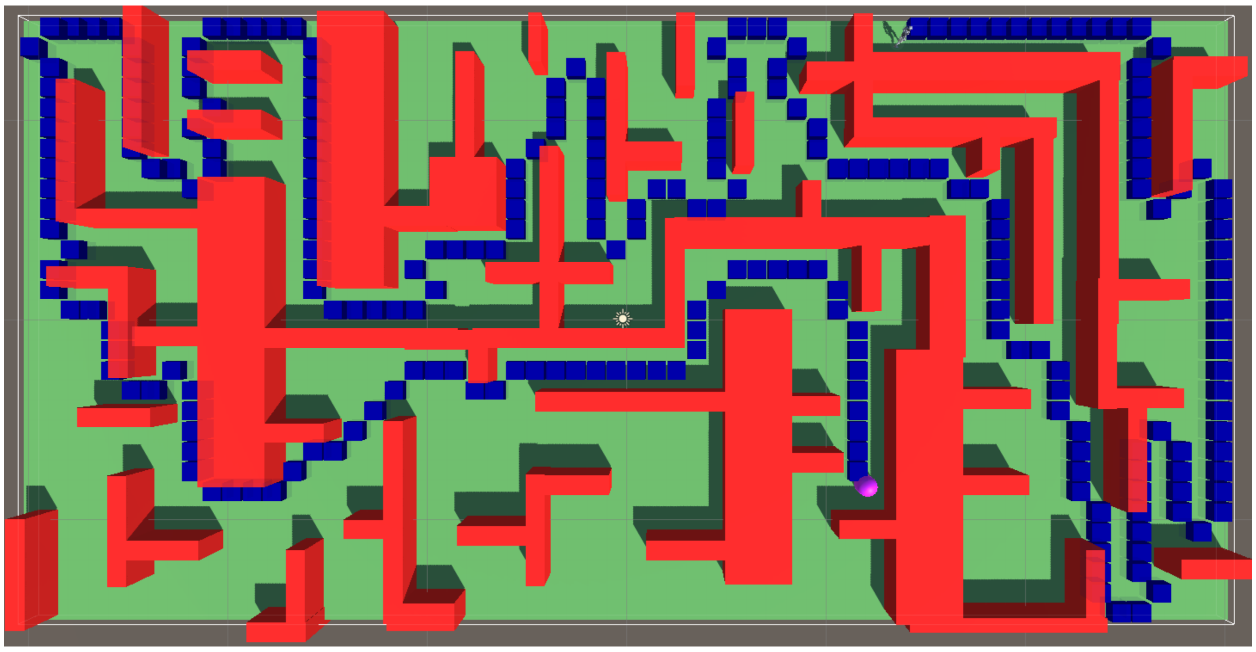


Secondly we ran simuliation using the Breath-First algorithm and then the Depth First Algorithm:

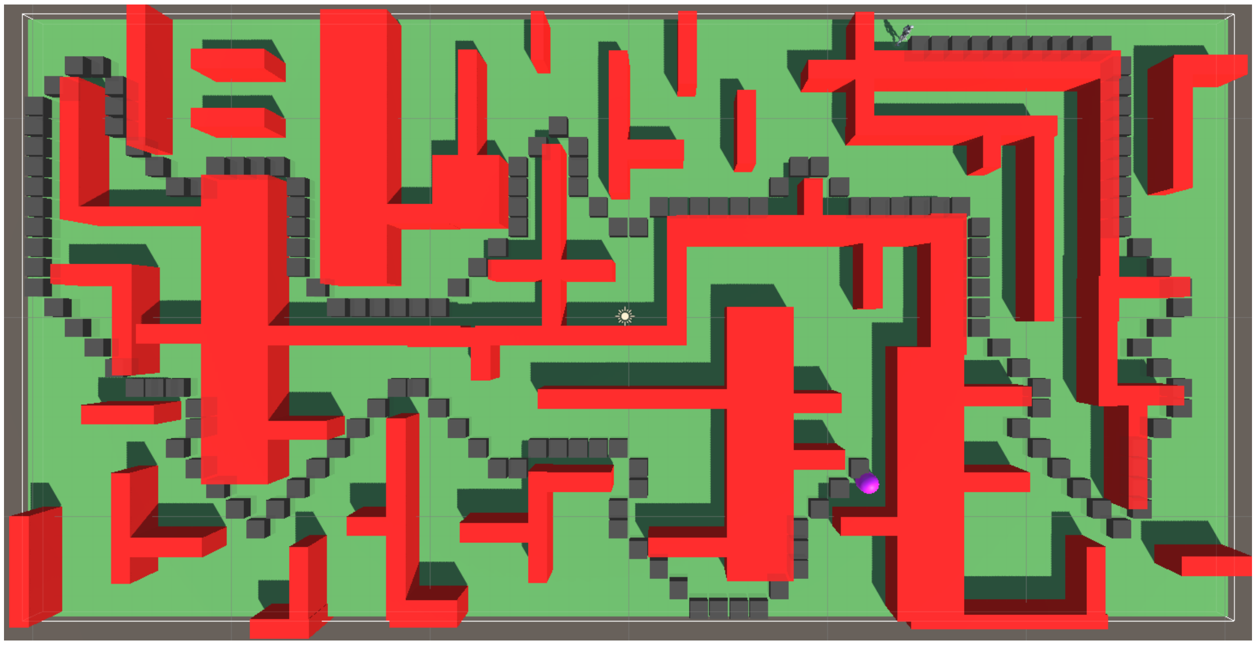
BFS:



DFS:

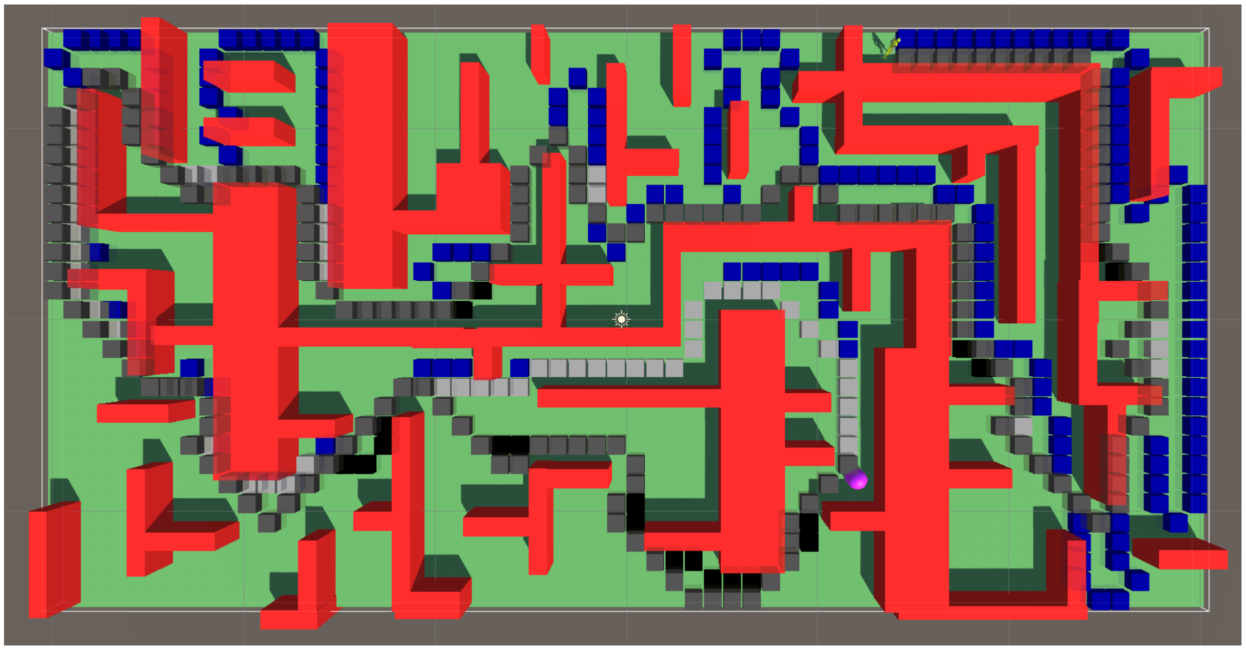


And finally, the UCS algorithm:



You can notice that the path in the UCS and the BFS are the same because all the costs in this example are 1.

If we put all the different algorithms together, the result resembles something like this:



Fringe

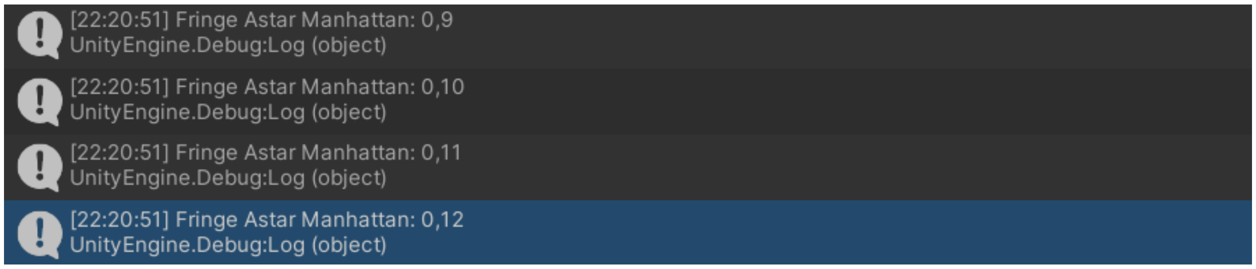
Due to the large number of elements in the stacks and queues I couldn't include the fringe in the report.

You can find the fringe in the debug section.

The debug shows the node's coordinate that is popped from the queue/Stack or the fringe in general

These are some examples:

Fringe Astar Manhattan:



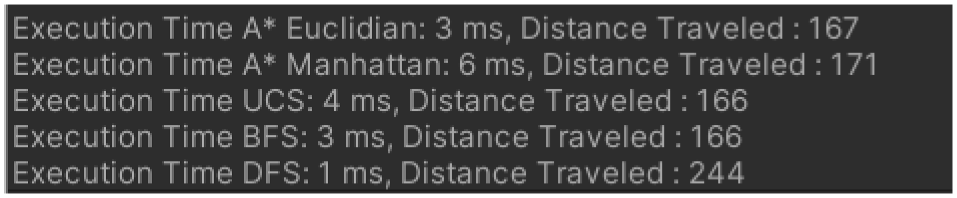
```
[22:20:51] Fringe Astar Manhattan: 0,9
UnityEngine.Debug:Log (object)
[22:20:51] Fringe Astar Manhattan: 0,10
UnityEngine.Debug:Log (object)
[22:20:51] Fringe Astar Manhattan: 0,11
UnityEngine.Debug:Log (object)
[22:20:51] Fringe Astar Manhattan: 0,12
UnityEngine.Debug:Log (object)
```

Time complexity and the Space complexity:

Here is a table that summarize the execution time and the traversal of each algorithm.

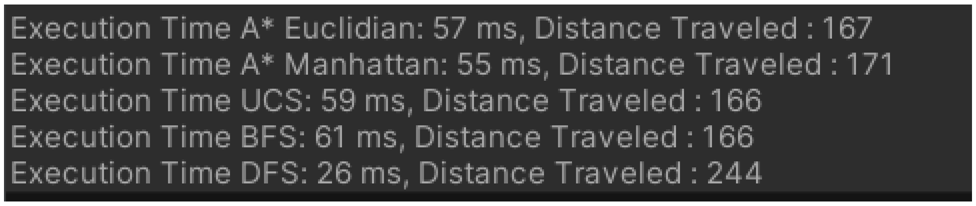
	Time Complexity	Space Complexity
A*(Manhattan)	6ms	171
A*(Euclidian)	3ms	167
UCS	4ms	166
BFS	3ms	166
DFS	1ms	244

Without fringe processing (printing fringe):



```
Execution Time A* Euclidian: 3 ms, Distance Traveled : 167
Execution Time A* Manhattan: 6 ms, Distance Traveled : 171
Execution Time UCS: 4 ms, Distance Traveled : 166
Execution Time BFS: 3 ms, Distance Traveled : 166
Execution Time DFS: 1 ms, Distance Traveled : 244
```

With fringe processing:



```
Execution Time A* Euclidian: 57 ms, Distance Traveled : 167
Execution Time A* Manhattan: 55 ms, Distance Traveled : 171
Execution Time UCS: 59 ms, Distance Traveled : 166
Execution Time BFS: 61 ms, Distance Traveled : 166
Execution Time DFS: 26 ms, Distance Traveled : 244
```

<u>Comparison:</u>

We can see that the best algorithm in time complexity is DFS because it takes only 1 ms to execute, but it is the algorithm with the greatest distance traveled (244). As a result, it isn't the most efficient algorithm.

The A* Manhattan algorithm is by far the slowest in terms of execution time, but it has a reasonable time traveled and is not the most efficient.

The BFS, UCS, and A* Euclidian are the three most efficient algorithms. However, the A* Euclidian trails the UCS and BFS by only one distance traveled.

So, the BFS algorithm is the most efficient here because, like the UCS algorithm, it has the shortest distance traveled and the second shortest execution time (3ms).

Furthermore, we can consider the BFS algorithm as the most efficient algorithm in this problem