Spring 2022

CSC 4301

<u>Project 1:</u> Pathfinding project

Team members:

Name	ID
Nour Elhouda Touti	101234
Nada Bounajma(teammate)	77807

Supervised by:

Dr. Tajjeeddine Rachidi

Introduction:

This project goal is to reproduce the pathfinding project and to and experiment the different search method learned during the class session like Depth first search (DFS), breath first search (BFS), and uniformed cost search (UCS), also the search with heuristic strategy.

This project is also an opportunity to learn a new language c and to experience working with a new software which is unity which is dedicated to develop games.

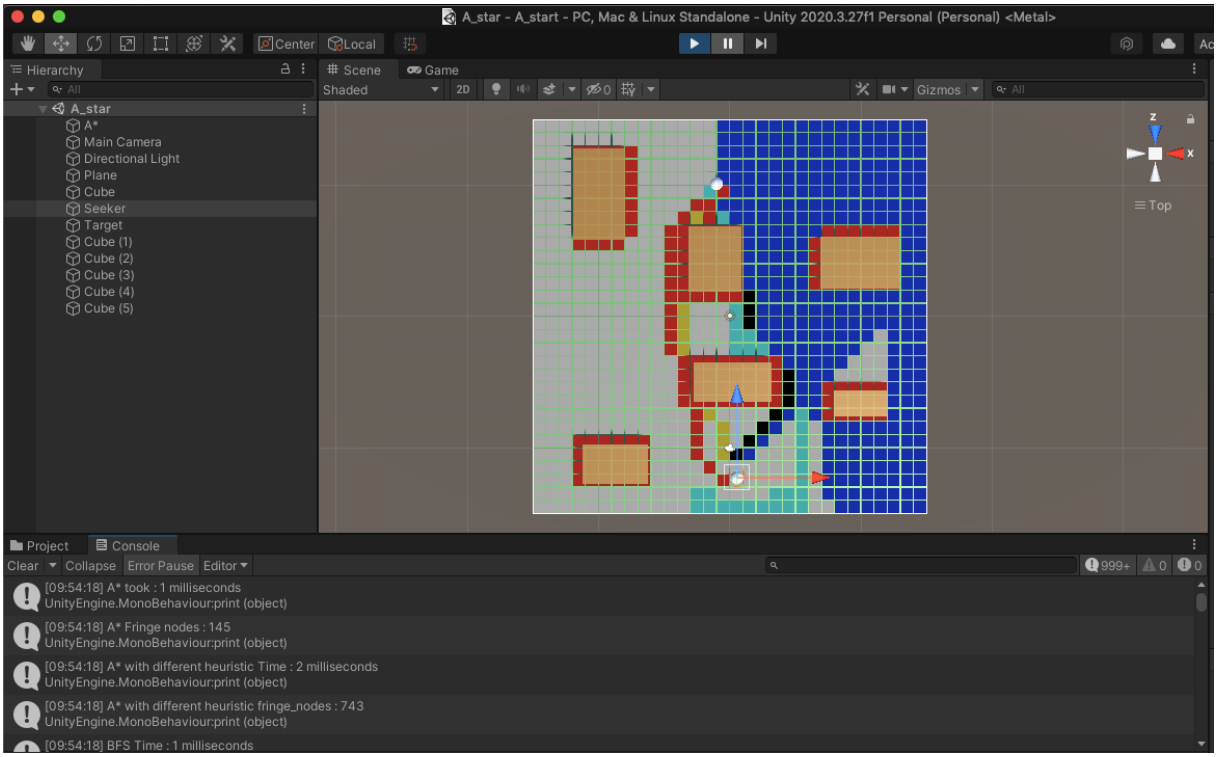
Technology used:

Unity is a cross-platform game engine developed by Unity Technologies, first announced and released in June 2005 at Apple Inc.'s Worldwide Developers Conference as a Mac OS X-exclusive game engine. The engine has since been gradually extended to support a variety of desktop, mobile, console and virtual reality platforms. It is particularly popular for iOS and Android mobile game development and used for games such as *Pokémon Go*, *Monument Valley*, *Call of Duty: Mobile*, *Beat Saber* and *Cuphead.* It is considered easy to use for beginner developers and is popular for indie game development.

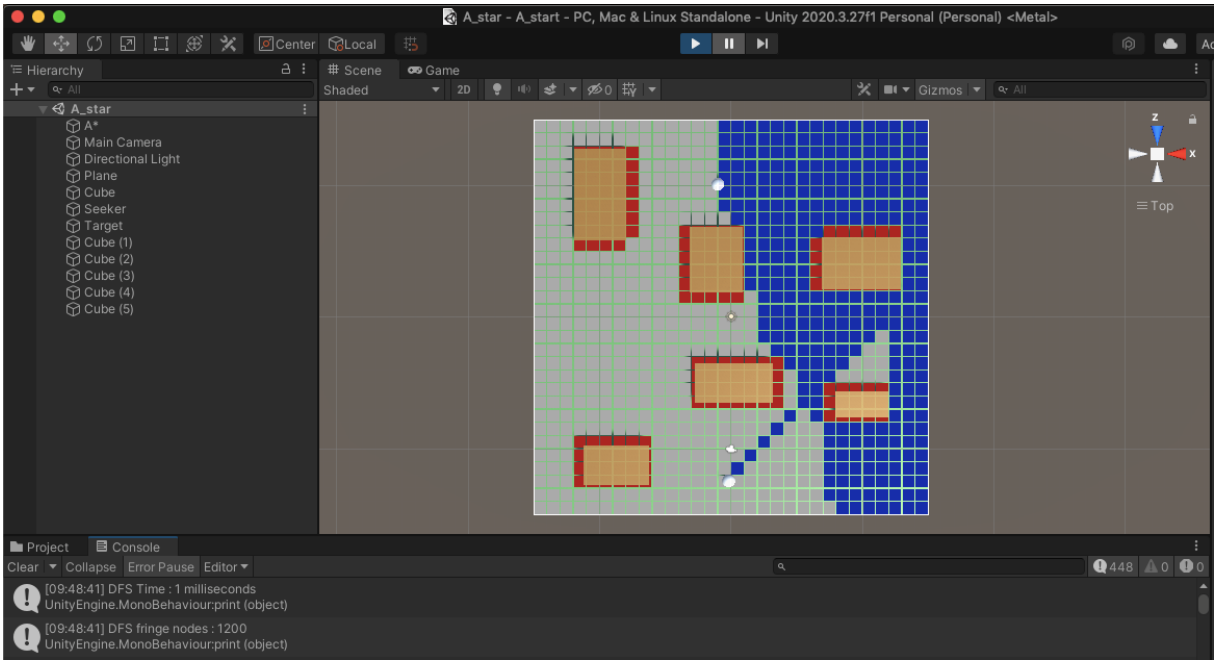
C# is a general object-oriented programming (OOP) language for networking and Web development. C# is specified as a common language infrastructure (CLI) language.

In January 1999, Dutch software engineer Anders Hejlsberg formed a team to develop C# as a complement to Microsoft's NET framework. Initially, C# was developed as C-Like Object Oriented Language. The actual name was changed to avert potential trademark issues. In January 2000, NET was released as C#. Its NET framework promotes multiple Web technologies.

Screenshots of each search method:



DFS:



The time complexity is 1 millisecond.

The size complexity: fringe nodes 1200.

Depth-first search (**DFS**) is an algorithm for traversing and searching in a tree or a graph. The algorithm starts from the root node (StartNode) and explore it as far as possible each branch before backtracking.

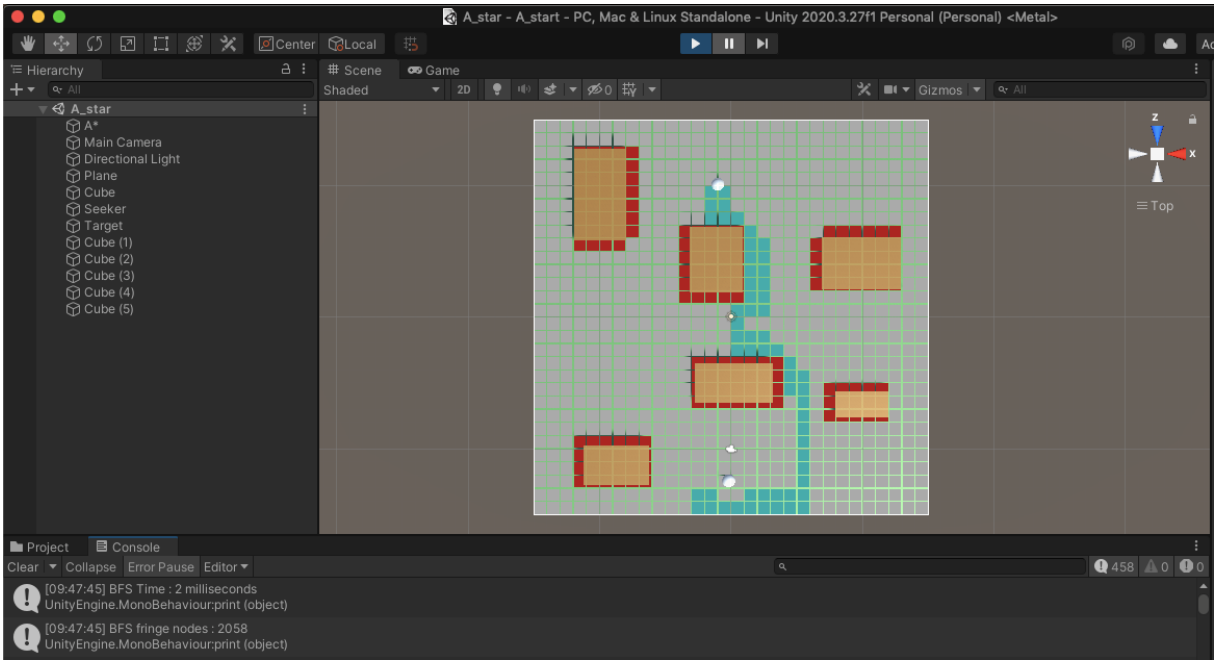
Depth-First Search algorithm use a stack as a data structure, and we also use data structure HashSet to store the visited node (the node that was expanded/explored). To search for the goal (Target node), we will make sure to visit all the nodes and each time we should compare the current node with the target node(goal), because the time we found the goal we will exit.

In DFS 1200 are the total number of the node in the fringe, it is logical to have such a number because the goal is so far from the start node so we should expand all the nodes that exist in each layer between the start node and the target node.

```
void FindPathDFS(Vector3 startPos, Vector3 targetPos){
              Node startNode = Grid.NodeFromWorldPoint(startPos);

       Node targetNode = Grid.NodeFromWorldPoint(targetPos);
              int fringe_nodes=0;
              Stack<Node> StackDFS = new Stack<Node>();
              HashSet<Node> visited = new HashSet<Node>();


              Stopwatch stopwatch  = new Stopwatch();
              stopwatch.Start();
```

```
            StackDFS.Push(startNode);
            fringe_nodes++;

            while (StackDFS.Count > 0){
                  Node current_node = StackDFS.Pop();

                  if (current_node == targetNode){
                        stopwatch.Stop();
                        print("DFS Time : "+stopwatch.ElapsedMilliseconds+"
milliseconds");
                        print("DFS fringe nodes : "+fringe_nodes);
                        RetracePath(startNode, targetNode,3);
                        return;
                  }

                  if(!visited.Contains(current_node)){
                        visited.Add(current_node);
                        foreach (Node neighbour in
Grid.GetNeighbours(current_node)){
                              if (!neighbour.walkable ||
visited.Contains(neighbour))continue;
                              neighbour.parent = current_node;
                              StackDFS.Push(neighbour);
                              fringe_nodes++;

                        }
                  }


            }
      }
```

BFS:



The time complexity is 2 milliseconds.

The size complexity: fringe nodes 2058.

Breadth-First Search algorithm is a graph traversing technique, where we choose a random initial node (StartNode) and then we traverse the graph layer-wise so that all the nodes and their children are expanded and visited until we found the goal.

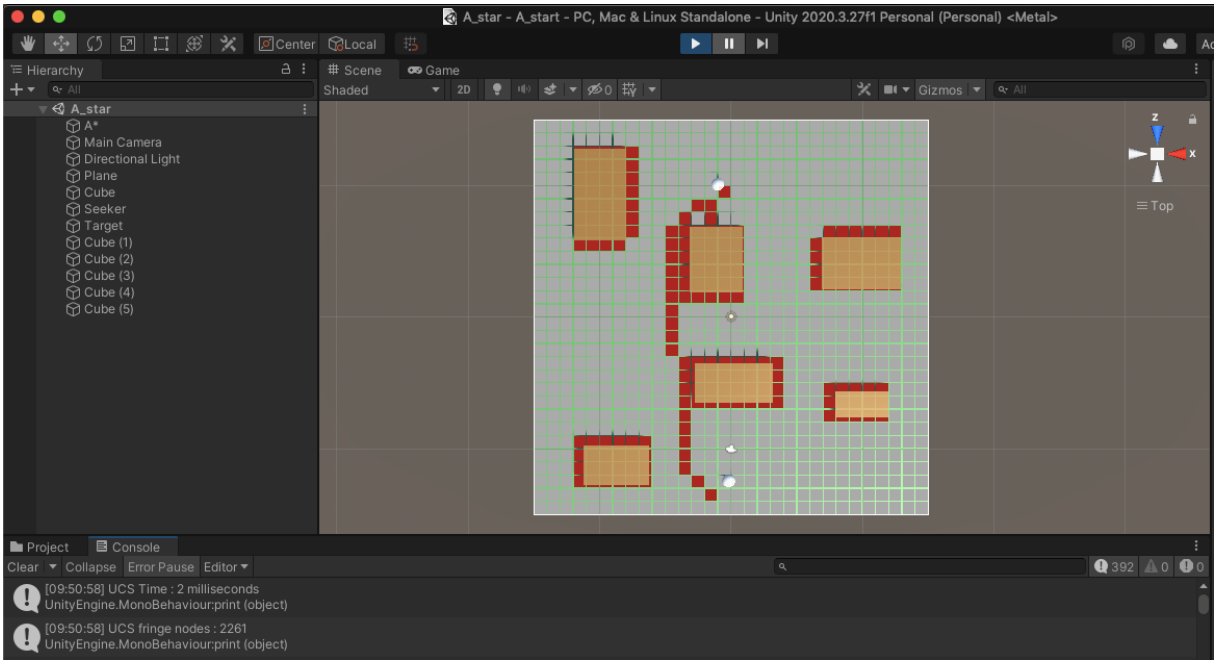
Breadth-First Search algorithm use a queue as a data structure, and we also use data structure HashSet to store the visited node (the node that was expanded/explored). To search for the goal (Target node), we will make sure to visit all the nodes and each time we should compare the current node with the target node(goal), because the time we found the goal we will exit.

In BFS 2058 are the total number of the node in the fringe, it is logical to have such a number because the goal is so far from the start node so we should expand all the nodes that exist in each layer between the start node and the target node. Also, because the big number of expanded node impact of the amount of time to reach the goal.

```
void FindPathBFS(Vector3 startPos, Vector3 targetPos){
            Node startNode = Grid.NodeFromWorldPoint(startPos);
            Node targetNode = Grid.NodeFromWorldPoint(targetPos);
            int fringe_nodes=0;
            Queue<Node> queue = new Queue<Node>();
            HashSet<Node> visited = new HashSet<Node>();

            Stopwatch stopwatch  = new Stopwatch();
            stopwatch.Start();

            queue.Enqueue(startNode); //make sure to add at the begining
            fringe_nodes++;

            while(queue.Count > 0){
                  Node current_node = queue.Dequeue();
```

```
                    if(current_node == targetNode){
                        stopwatch.Stop();
                        print("BFS Time : "+stopwatch.ElapsedMilliseconds+"
milliseconds");
                        print("BFS fringe nodes : "+fringe_nodes);
                        RetracePath(startNode, targetNode,2);
                        return;
                    }
                    if(!visited.Contains(current_node)){
                        visited.Add(current_node);
                        foreach (Node neighbour in
Grid.GetNeighbours(current_node)) {
                            if (!neighbour.walkable ||
visited.Contains(neighbour))continue;

                            neighbour.parent = current_node;
                            queue.Enqueue(neighbour);
                            fringe_nodes++;

                        }
                    }


            }
        }
```

UCS:



The time complexity is 2 milliseconds.

The size complexity: fringe nodes are 2261.

Uniform-cost search is an uninformed search algorithm that uses the lowest cumulative cost to find a path from the source to the destination. Nodes are expanded, starting from the root, according to the minimum cumulative cost.

Uniform-cost search use priority list. To find the goal, we first insert the start node (StartNode) into the priority queue, and while the queue is not empty, we remove the node with highest priority, if the node id the target node we print the total cost and then we quit. If the current node is not the target node, we enqueue all the children of the current node in the priority queue with their cumulative cost(hcost).

```
void FindPathUCS(Vector3 startPos, Vector3 targetPos){
            Node startNode = Grid.NodeFromWorldPoint(startPos);
            Node targetNode = Grid.NodeFromWorldPoint(targetPos);
            int fringe_nodes=0;
            List<Node> queue = new List<Node>();
            HashSet<Node> visited = new HashSet<Node>();
Stopwatch stopwatch  = new Stopwatch();
            stopwatch.Start();

            queue.Add(startNode);
            fringe_nodes++;

            while (queue.Count > 0)
            {
                  Node node = queue[0];
                  for (int i = 1; i < queue.Count; i++){
                        if (queue[i].gCost <= node.gCost){
```

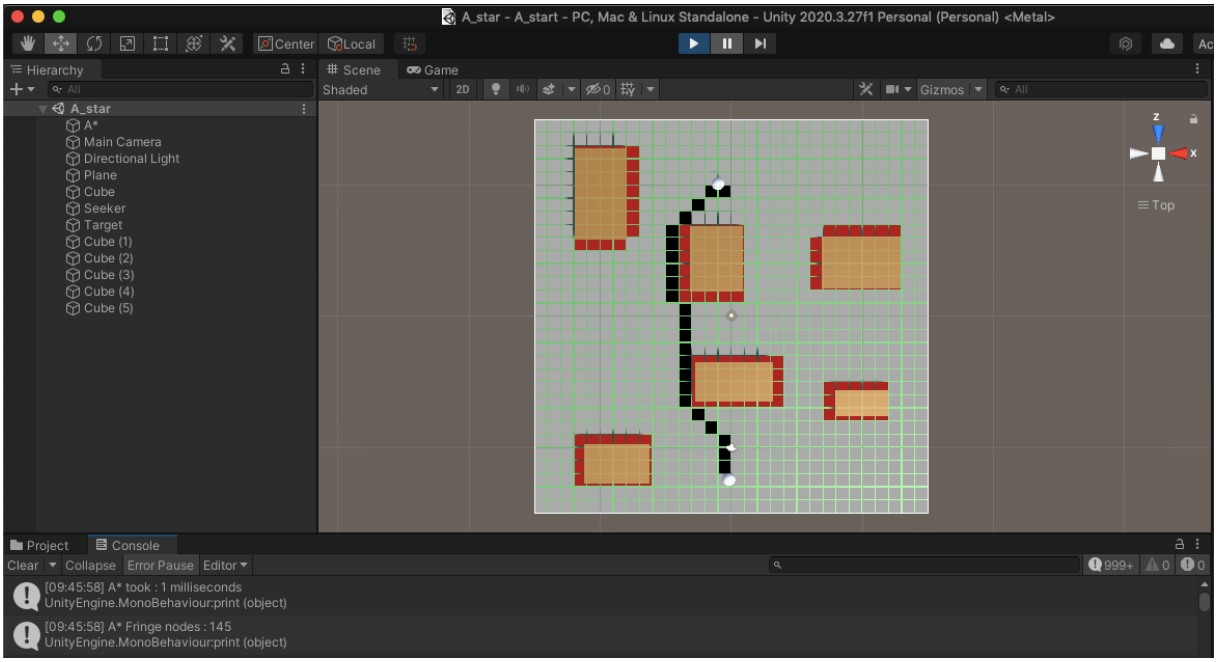
```
                                    node = queue[i];
                                }
                        }

                        queue.Remove(node);
                        visited.Add(node);

                        if (node == targetNode){
                                stopwatch.Stop();
                                print("UCS Time : "+stopwatch.ElapsedMilliseconds+"
milliseconds");
                                print("UCS fringe nodes : "+fringe_nodes);
                                RetracePath(startNode, targetNode, 4);
                                return;
                        }

                        foreach (Node neighbor in Grid.GetNeighbours(node)){
                                if (!neighbor.walkable ||
visited.Contains(neighbor))continue;

                                if (node.gCost < neighbor.gCost ||
!visited.Contains(neighbor)){
                                        neighbor.gCost = node.gCost + GetDistance(node,
neighbor, 1);
                                        neighbor.hCost = 0;
                                        neighbor.parent = node;

                                        if (!queue.Contains(neighbor))
                                        queue.Add(neighbor);
                                        fringe_nodes++;
                                }
                        }
                }
        }
```

A*:

A* Search algorithm is one of the best and popular technique used in path-finding and graph traversals. This is really a smart algorithm which separates it from the other conventional algorithms, because there is a specific condition to should the next node. A*, is the combination of greedy algorithm(**hcost**) how far the current node is from the goal(targetNode), and also UCS which mean the accumulated cost of a path from the startNode to the currentNode(**gcost**). The condition is fcost= **hcost** + **gcost.**

A* also use priority list as a data structure.

A:



The time complexity is 1 millisecond.

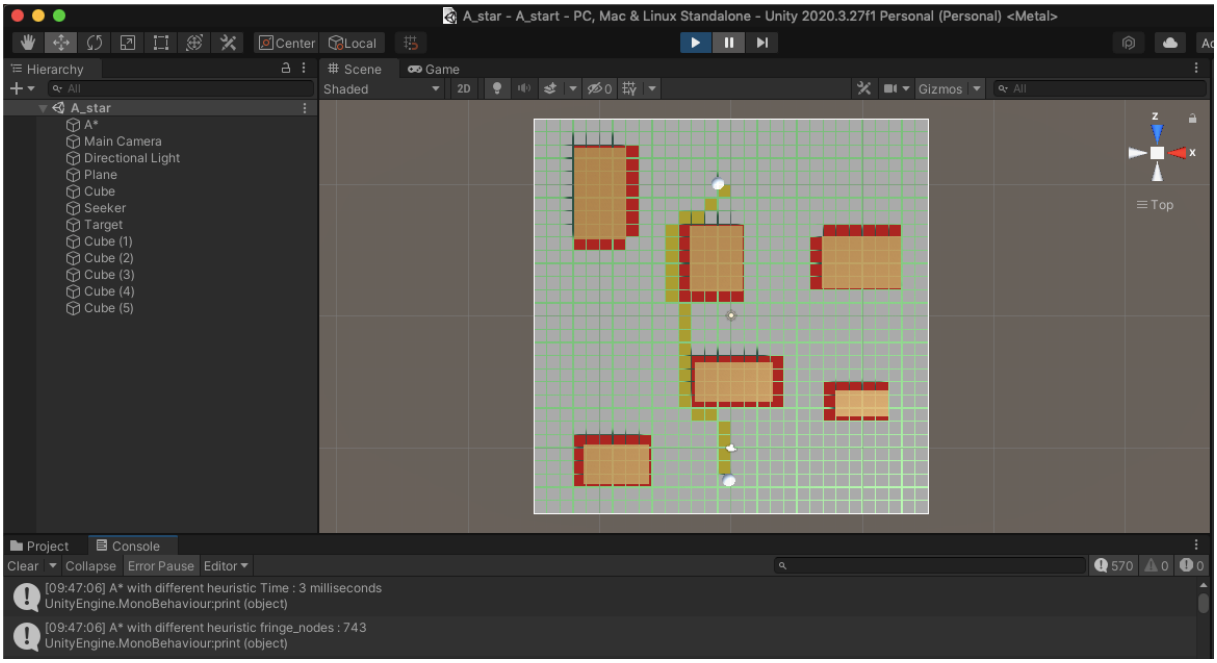
The size complexity: node fringe 145.

A select the next node carefully depending on the fcost so that we can find the shortest path with the least cost.

```
void FindPathA(Vector3 startPos, Vector3 targetPos) {
            Node startNode = Grid.NodeFromWorldPoint(startPos);
            Node targetNode = Grid.NodeFromWorldPoint(targetPos);
            int fringe_nodes = 0;
            List<Node> openSet = new List<Node>();
            HashSet<Node> closedSet = new HashSet<Node>();

            Stopwatch stopwatch  = new Stopwatch();
            stopwatch.Start();

            openSet.Add(startNode);
            fringe_nodes++;
```

```
            while (openSet.Count > 0) {
                  Node node = openSet[0];
                  for (int i = 1; i < openSet.Count; i ++) {
                        if (openSet[i].fCost < node.fCost || openSet[i].fCost ==
node.fCost) {
                              if (openSet[i].hCost < node.hCost)
                                    node = openSet[i];
                        }
                  }

                  openSet.Remove(node);
                  closedSet.Add(node);

                  if (node == targetNode) {
                        stopwatch.Stop();
                        print("A* took : "+stopwatch.ElapsedMilliseconds+"
milliseconds");
                        print("A* Fringe nodes : "+fringe_nodes);
                        RetracePath(startNode,targetNode,1);
                        return;
                  }

                  foreach (Node neighbour in Grid.GetNeighbours(node)) {
                        if (!neighbour.walkable || closedSet.Contains(neighbour))
{
                              continue;
                        }

                        int newCostToNeighbour = node.gCost + GetDistance(node,
neighbour);
                        if (newCostToNeighbour < neighbour.gCost ||
!openSet.Contains(neighbour)) {
                              neighbour.gCost = newCostToNeighbour;
                              neighbour.hCost = GetDistance(neighbour,
targetNode);
                              neighbour.parent = node;

                              if (!openSet.Contains(neighbour)){
                                    openSet.Add(neighbour);
                                    fringe_nodes++;
                              }
                        }
                  }
            }
      }
```

A2:



The time complexity is 3 milliseconds.

The size complexity: expanded node is 743.

```
void FindPathA2(Vector3 startPos, Vector3 targetPos) {
		Node startNode = Grid.NodeFromWorldPoint(startPos);
		Node targetNode = Grid.NodeFromWorldPoint(targetPos);
		int fringe_nodes = 0;
		List<Node> openSet = new List<Node>();
		HashSet<Node> closedSet = new HashSet<Node>();

		Stopwatch stopwatch  = new Stopwatch();
		stopwatch.Start();

		openSet.Add(startNode);
		fringe_nodes++;
		while (openSet.Count > 0) {
			Node node = openSet[0];
			for (int i = 1; i < openSet.Count; i ++) {
				if (openSet[i].fCost2 < node.fCost2 || openSet[i].fCost2
== node.fCost2) {
					if (openSet[i].hCost < node.hCost)
						node = openSet[i];
				}
			}

			openSet.Remove(node);
			closedSet.Add(node);

			if (node == targetNode) {
				stopwatch.Stop();
				print("A* with different heuristic Time :
"+stopwatch.ElapsedMilliseconds+" milliseconds");
				print("A* with different heuristic fringe_nodes :
"+fringe_nodes);
				RetracePath(startNode,targetNode,5);
				return;
			}
```

```
                    foreach (Node neighbour in Grid.GetNeighbours(node)) {
                            if (!neighbour.walkable || closedSet.Contains(neighbour))
{
                                    continue;
                            }

                            int newCostToNeighbour = node.gCost + GetDistance(node,
neighbour);
                            if (newCostToNeighbour < neighbour.gCost ||
!openSet.Contains(neighbour)) {
                                    neighbour.gCost = newCostToNeighbour;
                                    neighbour.hCost = GetDistance(neighbour,
targetNode);
                                    neighbour.parent = node;

                                    if (!openSet.Contains(neighbour))
                                            openSet.Add(neighbour);
                                            fringe_nodes++;
                            }
                    }
            }
    }
```

Link in GitHub: https://github.com/NourElhoudaTouti/Project1AI.git

Conclusion:

This project was so helpful to understand better the concept of each method in pathfinding. Also, having a real example to implement in the different methods of pathfinding was challenging but so important for us as a student in introduction to artificial intelligence.