Pathfinder: Navigating Mazes

Algorithms presented in Unity: 1. Floodfill, 2. Dijkstra, 3. A* & 4. Q-Learning

Edited by:

1. Apostolou Athanasios (mpsp 2203)

2. Birmpakos Georgios (mpsp 2220)

3. Evangelou Alexandros – Ioannis (mpsp 2210)

CONTENTS

INTRODUCTION

1.1 PROBLEM STATEMENT

Finding the shortest path between two points in a maze can be a challenging task, especially for applications that require real-time decision-making. This project aims to develop a Unity app that implements four pathfinding algorithms (Floodfill, Dijkstra, A*, and Q-Learning) to efficiently navigate complex mazes and find the shortest path between any two points. The goal of the project is to provide users with a tool that can visualize the algorithms in action and allow them to input their own maze configurations. This app has the potential to be used in a wide range of applications, such as game development or robotics.

1.2 MOTIVATION

The motivation behind this project is to address the challenges of pathfinding in complex mazes and to provide a tool that can efficiently solve pathfinding problems in real-time.

For example, game developers may require pathfinding algorithms to create AI-controlled characters that can navigate through complex environments in a realistic and efficient manner. Robotics engineers may require pathfinding algorithms to develop autonomous robots that can navigate through complex indoor or outdoor environments without human intervention.

Moreover, providing a tool that can help users visualize the algorithms in action and compare their performance could improve understanding of how these algorithms work and help users choose the most suitable algorithm for their specific use case.

Also, it was a challenge for the editors to get used with Unity and C# environment.

Overall, the motivation for this project is to provide a tool that can efficiently and effectively solve pathfinding problems in complex mazes, improving the efficiency and realism of applications that require real-time decision-making.

1.3 SCOPE

The scope of this project includes several aspects related to developing a Unity app that implements pathfinding algorithms for mazes. Some aspects of the project's scope include:

1. Implementing multiple pathfinding algorithms: The project could include implementing several pathfinding algorithms, such as Floodfill, Dijkstra, A*, and Q-Learning. These algorithms could be optimized to efficiently solve pathfinding problems for mazes of different sizes and complexities.
2. Designing a user-friendly interface: The app could include a user-friendly interface that allows users to input their own maze configurations, select the pathfinding algorithm to use, and view the pathfinding results in real-time.
3. Maze generation: The project could include generating different types of mazes, ranging from simple to complex, to test the pathfinding algorithms' efficiency and accuracy.
4. Optimizing the algorithms for performance: The pathfinding algorithms could be optimized for performance to improve the app's speed and efficiency in real-time applications.
5. Providing visualizations of the algorithms in action: The app could provide visualizations of the pathfinding algorithms in action to help users understand how each algorithm works and to compare the efficiency of different algorithms.

1.4 GOALS

The goals of the project could be to develop a functional and efficient Unity app that implements multiple pathfinding algorithms for complex mazes. Some potential goals of the project could include:

1. Developing functional pathfinding algorithms: The app should accurately and efficiently solve pathfinding problems for mazes of varying complexities using the implemented algorithms.
2. Optimizing algorithms for performance: The algorithms should be optimized for performance to provide real-time solutions to pathfinding problems and to reduce the app's resource requirements.
3. Creating a user-friendly interface: The app should have a user-friendly interface that is easy to navigate and allows users to input their own maze configurations, select the pathfinding algorithm, and view the results in real-time.
4. Providing visualizations of the algorithms in action: The app should provide visualizations of the pathfinding algorithms in action to help users understand how each algorithm works and to compare the efficiency of different algorithms.
5. Testing and evaluating the app: The app should be tested and evaluated to ensure that it meets the project's requirements and goals, and to identify areas for improvement.
6. Providing visualizations of the algorithms in action: The app could provide visualizations of the pathfinding algorithms in action to help users understand how each algorithm works and to compare the efficiency of different algorithms.

BRIEF HISTORY OF THE ALGORITHMS

2.1 Floodfill Algorithm

The Flood Fill algorithm is a simple algorithm that is used to identify and fill contiguous regions in a grid. The algorithm dates to the 1960s and is widely used in computer graphics for tasks such as image segmentation and flood filling.

- Advantages: always finds the shortest path, is easy to implement, and is guaranteed to find a path if one exists.
- Disadvantages: it may not be the fastest algorithm for large or complex maps, and it can be memory-intensive if the map is too large.

2.2 Dijkstra Algorithm

Dijkstra's Algorithm is a popular shortest path algorithm named after its inventor, Dutch computer scientist Edsger W. Dijkstra. The algorithm was first described in 1959 and is used to find the shortest path between nodes in a graph. It has numerous applications in transportation, logistics, and computer networks.

- Advantages: it always finds the shortest path, and it can work well for large or complex maps with weighted edges.
- Disadvantages: it may take a long time to find the path, especially if the map has many nodes or the edges have different weights.

2.3 A* Algorithm

The A* algorithm is a popular pathfinding algorithm that was first described in 1968 by computer scientist Peter Hart and his colleagues at Stanford Research Institute. The algorithm is a combination of Dijkstra's Algorithm and a heuristic function that estimates the distance between the current node and the goal node. A* is widely used in robotics, video games, and other applications that require efficient pathfinding.

- Advantages: it finds the shortest path and is usually faster than Dijkstra's Algorithm due to the heuristic function, which helps it prioritize nodes that are more likely to be on the shortest path.

- Disadvantages: it may not always find the shortest path, especially if the heuristic function is not well-chosen or the map is very large or complex.

2.4 Q-Learning Algorithm

The Q-Learning algorithm is a type of reinforcement learning algorithm that was first described by computer scientist Christopher Watkins in 1989. The algorithm is used to train an agent to make decisions based on rewards and punishments. Q-Learning has numerous applications in robotics, game AI, and control systems.

- Advantages: it can learn optimal paths by exploration and exploitation of the environment, and it can work well for complex maps with many possible actions.
- Disadvantages: it may take a long time to converge to the optimal path, especially if the environment is complex or the state space is large. Additionally, it requires a lot of data to learn effectively.

METHODOLOGY AND IMPLEMENTAION

2.1 Floodfill Methodology and Implementation

Pathfinding.cs

```
Queue<Tile> FloodFill(Tile start, Tile goal)
{
    Dictionary<Tile, Tile> nextTileToGoal = new Dictionary<Tile,
Tile>();
    Queue<Tile> frontier = new Queue<Tile>();
    List<Tile> visited = new List<Tile>();

    frontier.Enqueue(goal);

    while(frontier.Count > 0)
    {
        Tile curTile = frontier.Dequeue();

        foreach(Tile neighbor in _mapGenerator.Neighbors(curTile))
        {
            if (visited.Contains(neighbor) == false &&
frontier.Contains(neighbor) == false)
            {
                if (neighbor._TileType != Tile.TileType.Wall)
                {
                    frontier.Enqueue(neighbor);
                    nextTileToGoal[neighbor] = curTile;
                }
            }
        }
        visited.Add(curTile);
    }

    if (visited.Contains(start) == false)
        return null;

    Queue<Tile> path = new Queue<Tile>();
    Tile curPathTile = start;
    while(curPathTile != goal)
    {
        curPathTile = nextTileToGoal[curPathTile];
        path.Enqueue(curPathTile);
    }

    return path;
}
```

The algorithm starts by initializing a dictionary nextTileToGoal and a queue frontier. The nextTileToGoal dictionary is used to keep track of the next tile to visit to get to the goal tile, and the frontier queue is used to keep track of the tiles that need to be visited.

The frontier queue is initialized with the goal tile, and then the algorithm enters a loop that continues until the frontier queue is empty. In each iteration of the loop, the algorithm dequeues a tile from the frontier queue, and then checks each of its neighbors to see if they can be added to the frontier queue.

If a neighbor has not been visited before and is not already in the frontier queue, and it is not a wall tile, it is added to the frontier queue and its parent is stored in the nextTileToGoal dictionary.

The algorithm keeps track of visited tiles in the visited list. Once the algorithm has visited all the tiles that can be reached from the goal tile, it checks if the start tile has been visited. If the start tile has not been visited, it means that there is no path from the start tile to the goal tile, and the algorithm returns null.

If the start tile has been visited, the algorithm constructs the path from the start tile to the goal tile by following the parent tiles stored in the nextTileToGoal dictionary. The path is stored in a queue path, which is then returned by the algorithm.

2.2 Dijkstra Methodology and Implementation

Pathfinding.cs

```
Queue<Tile> Dijkstra(Tile start, Tile goal)
{
    Dictionary<Tile, Tile> NextTileToGoal = new Dictionary<Tile,
Tile>();//Determines for each tile where you need to go to reach the
goal. Key=Tile, Value=Direction to Goal
    Dictionary<Tile, int> costToReachTile = new Dictionary<Tile,
int>();//Total Movement Cost to reach the tile

    PriorityQueue<Tile> frontier = new PriorityQueue<Tile>();
    frontier.Enqueue(goal, 0);
    costToReachTile[goal] = 0;

    while (frontier.Count > 0)
    {
        Tile curTile = frontier.Dequeue();
        if (curTile == start)
            break;

        foreach (Tile neighbor in _mapGenerator.Neighbors(curTile))
        {
            int newCost = costToReachTile[curTile] + neighbor._Cost;
            if (costToReachTile.ContainsKey(neighbor) == false ||
newCost < costToReachTile[neighbor])
            {
                if (neighbor._TileType != Tile.TileType.Wall)
                {
                    costToReachTile[neighbor] = newCost;
                    int priority = newCost;
                    frontier.Enqueue(neighbor, priority);
                    NextTileToGoal[neighbor] = curTile;
                    neighbor._Text =
costToReachTile[neighbor].ToString();
                }
            }
        }
    }

    //Get the Path

    //check if tile is reachable
    if (NextTileToGoal.ContainsKey(start) == false)
    {
        return null;
    }

    Queue<Tile> path = new Queue<Tile>();
    Tile pathTile = start;
    while (goal != pathTile)
    {
        pathTile = NextTileToGoal[pathTile];
        path.Enqueue(pathTile);
    }
    return path;
}
```

This algorithm implements Dijkstra's shortest path algorithm for finding the shortest path between two points on a tile-based map. The input to the algorithm is the starting tile and the goal tile. The algorithm uses a priority queue to keep track of the tiles with the lowest movement cost, and it uses two dictionaries: "NextTileToGoal" and "costToReachTile".

The "NextTileToGoal" dictionary determines for each tile where you need to go to reach the goal, and the "costToReachTile" dictionary keeps track of the total movement cost to reach the tile.

The algorithm starts by enqueueing the goal tile in the priority queue with a priority of 0 and a cost of 0. It then continues to dequeue tiles from the priority queue until it reaches the starting tile or until there are no more tiles left in the queue.

For each dequeued tile, the algorithm examines each of its neighboring tiles and calculates the movement cost to reach each neighbor tile. If the neighbor tile has not been visited yet or if the new cost to reach it is lower than the previous cost, the algorithm updates the "NextTileToGoal" and "costToReachTile" dictionaries for the tile and enqueues it in the priority queue with its new priority value.

Once the algorithm has found the shortest path to the starting tile, it backtracks through the "NextTileToGoal" dictionary to find the path and returns it as a queue of tiles. If the starting tile is unreachable, the algorithm returns null.

2.3 A* Methodology and Implementation

Pathfinding.cs

```
Queue<Tile> AStar(Tile start, Tile goal)
{
    Dictionary<Tile, Tile> NextTileToGoal = new Dictionary<Tile,
Tile>();//Determines for each tile where you need to go to reach the
goal. Key=Tile, Value=Direction to Goal
    Dictionary<Tile, int> costToReachTile = new Dictionary<Tile,
int>();//Total Movement Cost to reach the tile

    PriorityQueue<Tile> frontier = new PriorityQueue<Tile>();
    frontier.Enqueue(goal, 0);
    costToReachTile[goal] = 0;

    while (frontier.Count > 0)
    {
        Tile curTile = frontier.Dequeue();
        if (curTile == start)
            break;

        foreach (Tile neighbor in _mapGenerator.Neighbors(curTile))
        {
            int newCost = costToReachTile[curTile] + neighbor._Cost;
            if (costToReachTile.ContainsKey(neighbor) == false ||
newCost < costToReachTile[neighbor])
            {
                if (neighbor._TileType != Tile.TileType.Wall)
                {
                    costToReachTile[neighbor] = newCost;
                    int priority = newCost + Distance(neighbor,
start);
                    frontier.Enqueue(neighbor, priority);
                    NextTileToGoal[neighbor] = curTile;
                    neighbor._Text =
costToReachTile[neighbor].ToString();
                }
            }
        }
    }

    //Get the Path

    //check if tile is reachable
    if (NextTileToGoal.ContainsKey(start) == false)
    {
        return null;
    }

    Queue<Tile> path = new Queue<Tile>();
    Tile pathTile = start;
    while (goal != pathTile)
    {
        pathTile = NextTileToGoal[pathTile];
        path.Enqueue(pathTile);
    }
    return path;
}
```

This is an implementation of the A* algorithm for finding the shortest path between two points on a grid. It uses a priority queue to explore the most promising paths first, based on an estimate of the remaining cost to the goal (heuristic). In this case, the heuristic is the straight-line distance between the current tile and the goal. The algorithm also keeps track of the total cost of reaching each tile, which is the sum of the cost of moving from the start tile to the current tile and the heuristic cost to the goal. Finally, it backtracks from the goal tile to the start tile using the NextTileToGoal dictionary to construct the shortest path.

2.4 Q-Learning Methodology and Implementation

```
Queue<Tile> QLearning(Tile start, Tile goal)
{
    // Set hyperparameters
    float learningRate = 0.1f;
    float discountFactor = 0.99f;
    int maxEpisodes = 10000;
    int maxStepsPerEpisode = 100;

    // Create the Q-Table
    Dictionary<Tile, Dictionary<Tile, float>> qTable = new
Dictionary<Tile, Dictionary<Tile, float>>();
    foreach (Tile tile in _mapGenerator.GetAllTiles())
    {
        qTable[tile] = new Dictionary<Tile, float>();
        foreach (Tile action in _mapGenerator.Neighbors(tile))
        {
            qTable[tile][action] = 0f;
        }
    }

    // Run episodes
    for (int episode = 0; episode < maxEpisodes; episode++)
    {
        Tile curTile = start;
        int step = 0;
        while (curTile != goal && step < maxStepsPerEpisode)
        {
            // Choose the action with the highest Q-value
            Tile nextTile = null;
            float maxQValue = float.MinValue;
            foreach (Tile neighbor in
_mapGenerator.Neighbors(curTile))
            {
                if (qTable[curTile][neighbor] > maxQValue)
                {
                    maxQValue = qTable[curTile][neighbor];
                    nextTile = neighbor;
                }
            }

            // Update the Q-value of the current state-action pair
            float reward = 0f;
            if (nextTile == goal)
            {
                reward = 100f;
            }
            float oldQValue = qTable[curTile][nextTile];
            float newQValue = oldQValue + learningRate * (reward +
discountFactor * maxQValue - oldQValue);
            qTable[curTile][nextTile] = newQValue;

            // Move to the next state
            curTile = nextTile;
            step++;
        }
    }

    // Follow the policy determined by the Q-table to get the path
    Queue<Tile> path = new Queue<Tile>();
    path.Enqueue(start);
    Tile curPathTile = start;
    while (curPathTile != goal)
    {
        Tile nextPathTile = null;
        float maxQValue = float.MinValue;
        foreach (Tile neighbor in
_mapGenerator.Neighbors(curPathTile))
        {
            if (qTable[curPathTile][neighbor] > maxQValue)
            {
                maxQValue = qTable[curPathTile][neighbor];
                nextPathTile = neighbor;
            }
        }

        curPathTile = nextPathTile;
        path.Enqueue(curPathTile);
    }

    return path;
}
```

This code implements the Q-learning algorithm to find a path from a start tile to a goal tile in a map. The Q-learning algorithm is a reinforcement learning algorithm that learns an optimal policy by updating Q-values for state-action pairs. The Q-value represents the expected cumulative reward for taking a particular action in a particular state.

The code initializes a Q-table that stores the Q-values for each state-action pair. The Q-values are initially set to zero. Then, the code runs a specified number of episodes, where each episode consists of taking actions and updating Q-values based on the observed rewards. In each step of an episode, the code chooses the action with the highest Q-value and moves to the next state. After each action, the code updates the Q-value for the current state-action pair based on the observed reward and the maximum Q-value for the next state.

After all episodes have been run, the code follows the policy determined by the Q-table to generate a path from the start tile to the goal tile. The policy is determined by choosing the action with the highest Q-value in each state until the goal tile is reached. The resulting path is returned as a queue of tiles.

4. MANUAL – USER INTERFACE

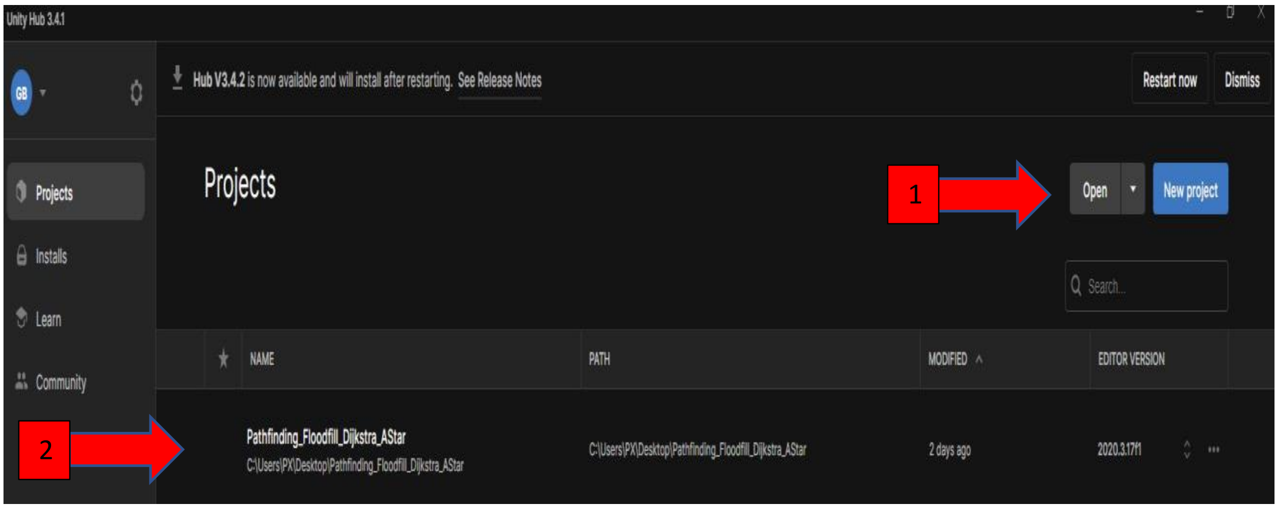
Here are the steps that the user should follow to run the Unity Application:

1. Download Unity Hub and Unity Editor

2. Open Unity Hub

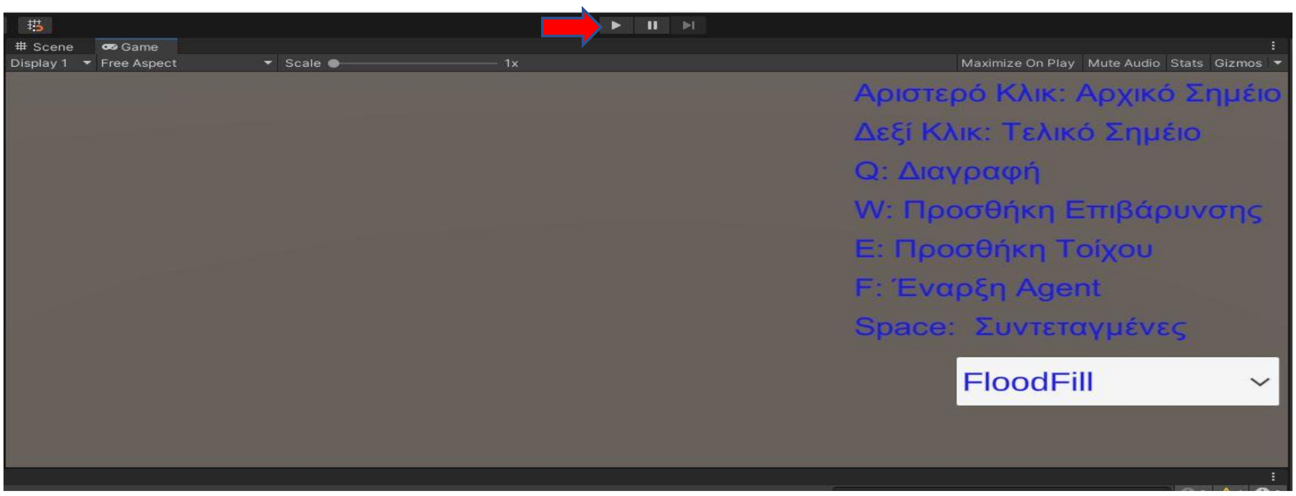
3. Click “Add project from disk” and chose “Pathfinding_Floodfill_Dijkstra_AStar” (Step 1)

4. Open Project (Step 2)



	NAME	PATH	MODIFIED	EDITOR VERSION
	Pathfinding_Floodfill_Dijkstra_AStar C:\Users\PX\Desktop\Pathfinding_Floodfill_Dijkstra_AStar	C:\Users\PX\Desktop\Pathfinding_Floodfill_Dijkstra_AStar	2 days ago	2020.3.17f1

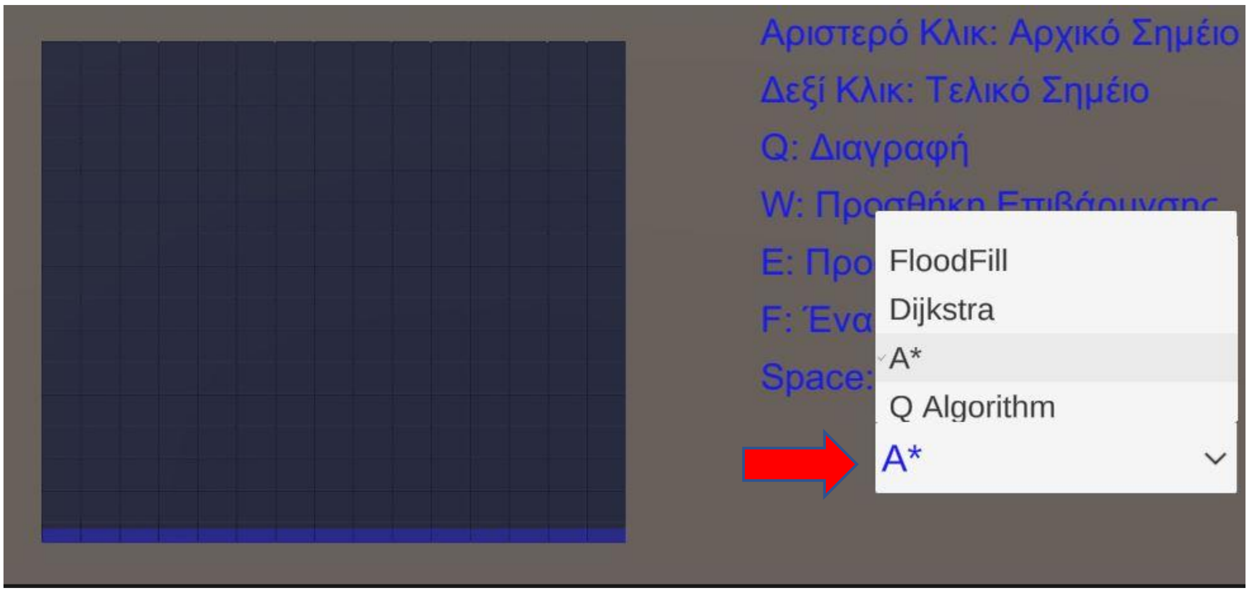
5. Click the “Play” button



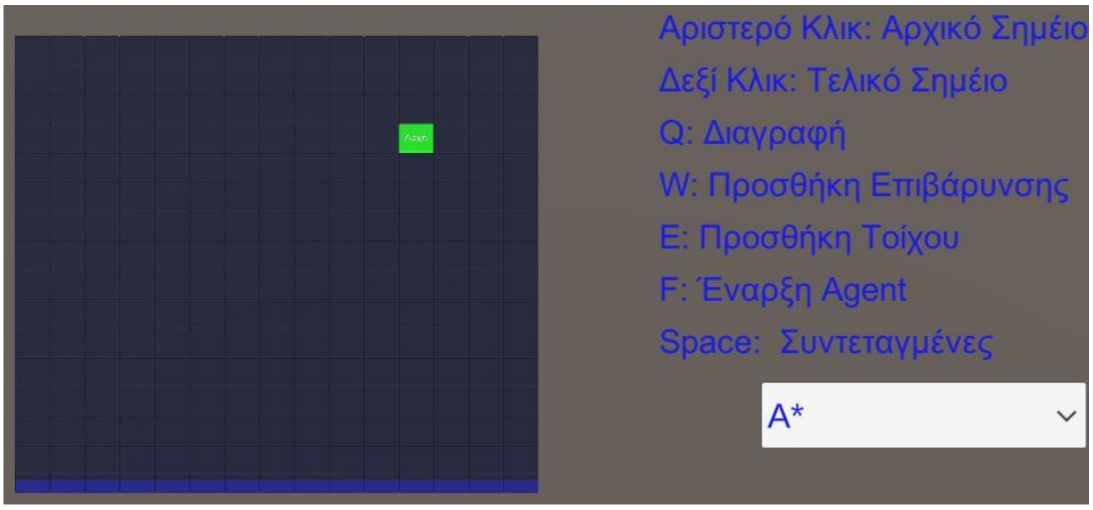
Αριστερό Κλικ: Αρχικό Σημείο
Δεξί Κλικ: Τελικό Σημείο
Q: Διαγραφή
W: Προσθήκη Επιβάρυνσης
E: Προσθήκη Τοίχου
F: Έναρξη Agent
Space: Συντεταγμένες

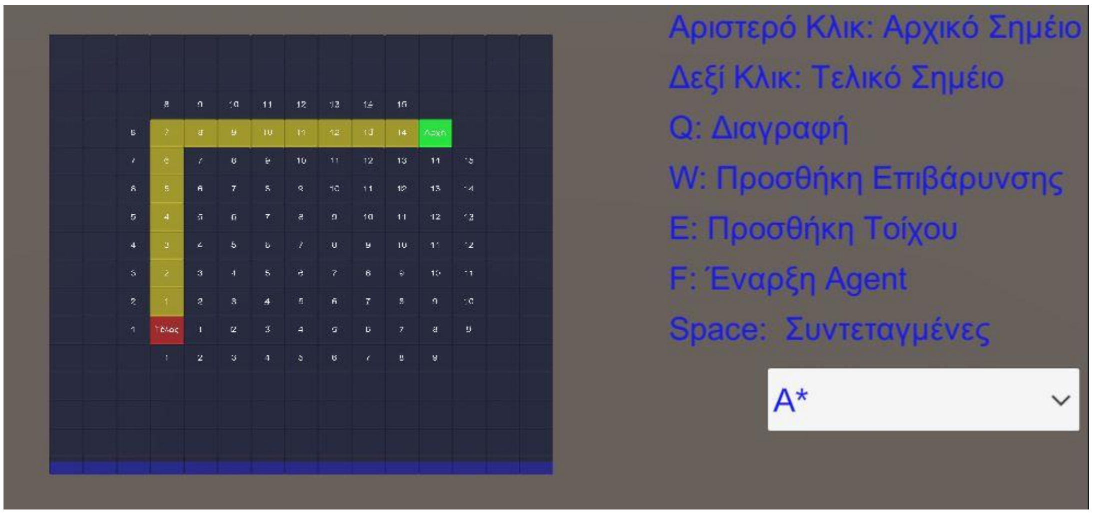
FloodFill

6. Select a Maze Navigation Algorithm from the Dropdown Menu

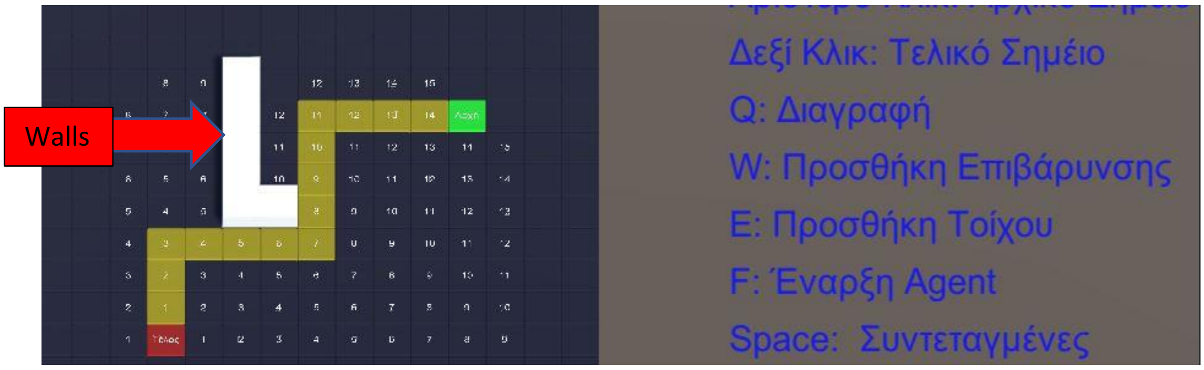

Αριστερό Κλικ: Αρχικό Σημείο
Δεξί Κλικ: Τελικό Σημείο
Q: Διαγραφή
W: Προσθήκη Επιβάρυνσης
E: Προ
F: Ένα
Space:

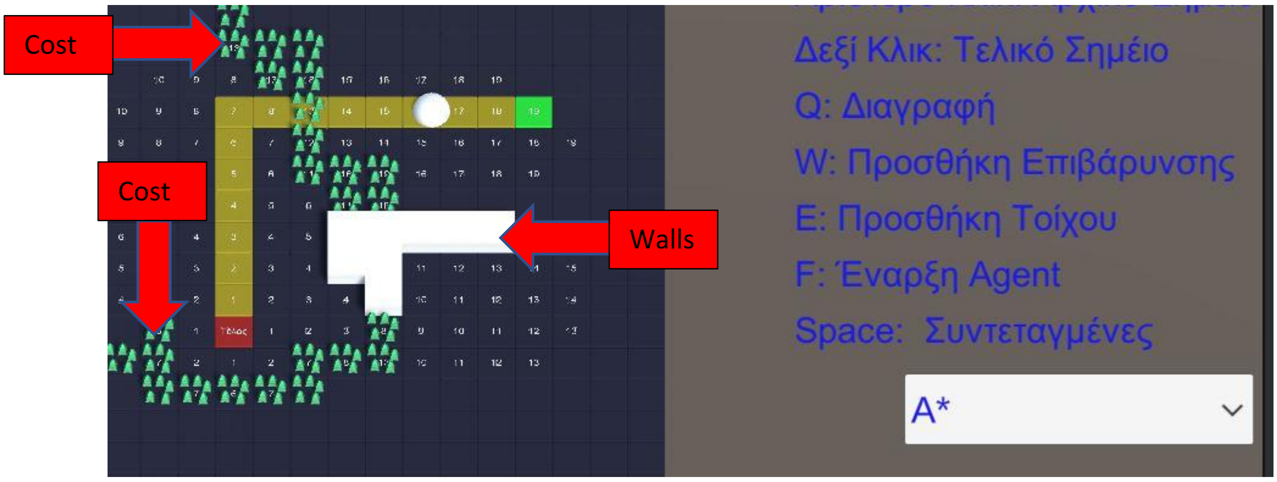
7. Set start and end point (Left Click and Right Click)


Αριστερό Κλικ: Αρχικό Σημείο
Δεξί Κλικ: Τελικό Σημείο
Q: Διαγραφή
W: Προσθήκη Επιβάρυνσης
E: Προσθήκη Τοίχου
F: Έναρξη Agent
Space: Συντεταγμένες


Αριστερό Κλικ: Αρχικό Σημείο
Δεξί Κλικ: Τελικό Σημείο
Q: Διαγραφή
W: Προσθήκη Επιβάρυνσης
E: Προσθήκη Τοίχου
F: Έναρξη Agent
Space: Συντεταγμένες

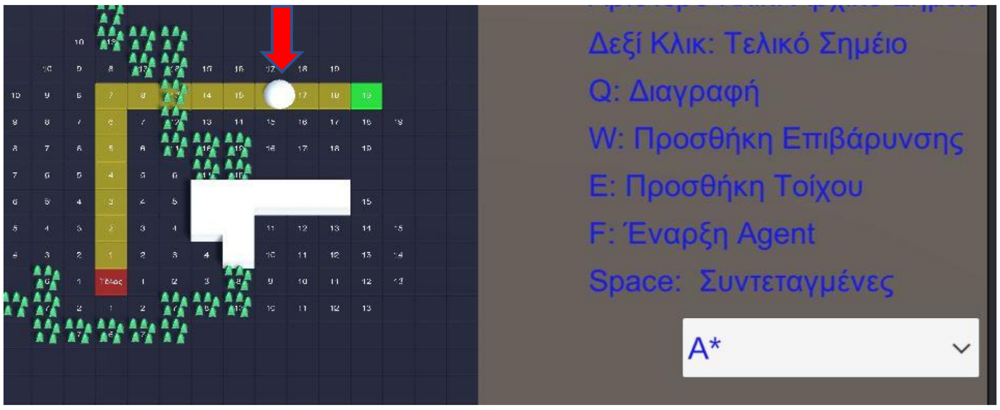
8. Set walls or costs (E and W buttons while hovering the mouse)

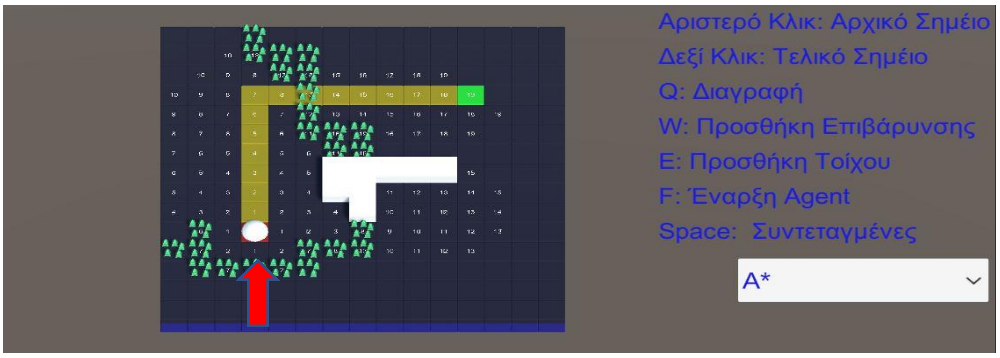




*By hitting the Q Button while hovering we can delete the walls and the costs we have added.

9. Hit the F Button so that the Agent starts moving.





5 CONCLUSION

In this documentation, we mentioned several popular algorithms used in computer science and artificial intelligence. We started with the Flood Fill algorithm, which is used for filling an area with a specific color. Then we talked about the Dijkstra algorithm, which is used to find the shortest path between two points in a graph. We also covered the A* algorithm, which is an extension of Dijkstra's algorithm that uses heuristics to search more efficiently.

After that, we moved on to the Q-learning algorithm, which is a type of reinforcement learning used in machine learning. It is used to learn the optimal policy for an agent in each environment, where the agent receives rewards or punishments based on its actions.

We discussed the implementation and methodology of each algorithm, explaining their key features, advantages, and disadvantages. It is important to note that these algorithms are just a few examples of the vast collection of algorithms used in computer science, and they have different use cases and applications depending on the problem being solved.

These algorithms were presented in the context of Unity game development using C# scripts.

In conclusion, these algorithms are essential tools for solving various problems in computer science, from pathfinding and optimization to machine learning and artificial intelligence. Understanding their principles and characteristics is crucial for any computer scientist, programmer, or engineer who wants to design efficient and effective algorithms and systems.