Pathfinding Algorithms versus AI Pathfinding in Unity

Joseph Guitian, John DiGabriele
Dept. of Electrical and Computer Engineering, Stevens Institute of Technology, Hoboken, USA
Email: {jdigabri,jguitian}@stevens.edu

Abstract –

This research paper explores the development and analysis of an AI navigation simulation using Unity. It is designed to educate people about artificial intelligence pathfinding capabilities. The project is structured as two primary components: the control model and the AI learning model.
The control model utilizes prefab GameObjects in Unity to establish a graph-based terrain, where nodes are represented as critical points that are neighbors, or edges with one another. Each neighbor is associated with a cost to travel to the node. A player model in this terrain is written with a script, representing Dijkstra's Algorithm which demonstrates the process of navigating the terrain to determine the least costly path to get to the end point. The player script includes a starting point and an end point which are represented by the Nodes in the engine. This model is meant to serve as a baseline to compare with the AI model, to ensure that it is properly implemented.The AI model will replicate the control model's environment, including the same nodes and terrain as the control, but this time the player will be employed with reinforcement learning techniques like Q-learning to teach the AI to autonomously navigate a graphical path by learning Dijkstra's Algorithm. This paper aims to provide insight towards the effectiveness of simulation-based learning environments for AI in more complex environments within gaming engines like Unity.

1. Introduction

Navigating complex environments efficiently is a fundamental aspect in things like gaming or robotic pathfinding. This research introduces an interesting approach to simulate and teach pathinging to artificial intelligence within the Unity Environment, a popular game development platform known for its simulation capabilities. The project is divided into two-distinct phases: the control model and the AI learning model.

The control model serves as a foundation setup where the virtual terrain is constructed using prefab GameObjects arranged to form a graph that can be navigated through. Each node on the graph represents critical points within the environment, and the connections between each node are weighted using predefined costs to traverse each path. This setup is used to simulate realistic pathfinding scenarios, which provides a visual way to understand the movement and decision making process involved with navigating a terrain.

The control model's implementation utilizes a player model equipped with a script that uses Dijkstra's Algorithm, a well established method for finding the shortest path in a graph. This script effectively guides the player model from one node to another along the path while also maintaining the minimal cost to travel to the end point. This demonstrates the practicality of theoretical pathfinding algorithms in a controlled setting.

The next phase of the project aims to construct an AI model that operates within the same scope of the control model. The goal for this model is to learn to navigate using pathfinding algorithms through direct interaction with the environment. Reinforcement learning, specifically the Q-learning algorithm (3) has been selected for this purpose due to its success in similar learning scenarios (2). Q-learning will allow the AI to optimize its pathfinding strategy by updating its policy based on a reward system, which will be dictated by the costs of traversing different paths on the graph.

The AI model resulted in being less efficient than the control model but showed promise that it could eventually improve and mirror the control.The ultimate objective of this research is to not only demonstrate the capabilities of AI in pathfinding environments, but to also compare the process with the control model's efficiency. This comparison is anticipated to yield insight to the practical applications and limitations of using Unity as a visual tool for both AI and Algorithmic education.

2. System architectures

Our research project employs the Unity game engine to simulate and compare two distinct models of pathfinding: the traditional control model using Dijkstra's Algorithm and a more advanced AI learning model that utilizes Q-learning. This architecture section provides an in-depth look at how both models are designed, integrated, and function within Unity, highlighting their interactions and the innovative approach to AI education in pathfinding.

The control model is built within a Unity environment that includes a main camera, directional light, and terrain which acts as the physical space where pathfinding occurs. The terrain is structured as a graph, with each node represented by a GameObject in Unity. These nodes are connected by edges, each bearing a cost that simulates the effort or resource expenditure necessary for traversal. This setup provides a visual and interactive representation of Dijkstra's Algorithm in action. A player model, scripted in C#, navigates this graph, calculating the least costly path from a designated start to an endpoint. This not only serves to validate the algorithm's practical application but also offers a baseline performance metric against which the AI model can be measured.

Parallel to the control model, the AI learning model duplicates the graphical setup but incorporates elements crucial for implementing Q-learning. Here, the same terrain and nodes are used, but the player, governed by AI, learns to navigate based on rewards derived from the costs associated with moving between nodes. The integration of machine learning is facilitated through Unity's scripting capabilities(1), allowing for real-time learning and adaptation by the AI. This model aims to demonstrate the potential of reinforcement learning in optimizing pathfinding strategies, with the AI continuously updating its decisions based on accumulated experience within the simulated environment.

3. The Control Model in Unity for AI Navigation Simulation

The control model in our research serves as a baseline for simulating pathfinding behavior and as a benchmark for the evaluation of AI model's performance. By constructing the controlled environment, this part of the project (currently) represents game based terrain using graphical representations which provides a clear and measurable standard where the Ai's learning efficiency can be assessed.

The terrain in the control model (right now) was designed to be simple enough to prove that Dijkstra's Algorithm is utilized properly and to perform easy calculations. Eventually it will be more complex once the AI model continues training. Inside of a Unity Scene, there are eleven GameObjects that simulate this environment.

The Main Camera, Directional Light, and ground are required for the simulation to run smoothly and to allow us to visualize the environment easier than it would be without them (elaborate later). There are also 7 Node GameObjects scattered across the ground in the same orientation as Figure 1. This orientation represents a graphical representation of Dijkstra's Algorithm taught in CPE 593: Applied Data Structures and Algorithms at Stevens Institute of Technology. This orientation was chosen to guarantee that the implementation was correct and easy enough to visualize and understand. The Nodes are red spherical objects on the terrain. This allows the nodes to be visible to us while the player moves to those points even though they do not necessarily need to be visible. Each node is equipped with both a neighbor and a cost. The neighbor (or edge) displays where the player is allowed to travel to next after reaching the current node. Each time a player travels to another neighbor, there is a cost involved which could represent any kind of constraint like time or money that the Player can spend less or more, depending on where it needs to travel and how it gets there.(Appendix A). Lastly, the player GameObject represents the player traversing the terrain.The player is (currently) a white, capsule shaped object mapped with a pathfinding script that directly uses Dijkstra's Algorithm (5). The player will travel to its starting node and then work its way to its end node where it will stop. The algorithm is designed to optimize this travel for the player to get to its destination while spending the least amount of cost.

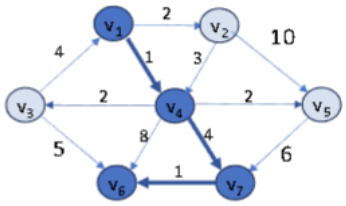

Figure 1

Dijkstra's Algorithm, a cornerstone of graph theory, is employed in the control model to facilitate pathfinding. This algorithm is very well suited for our simulation due to its effectiveness of finding the shortest path with the least amount of cost between nodes in a weighted graph. Unity allows any GameObject to be equipped with a script, where this algorithm will be used within the scene by the player (Appendix B).

$d[v] = min(d[v], d[u] + weight(u, v)$
d[v] represents the shortest distance from the source to vertex 'v'
d[u] is the shortest distance from the source to vertex 'u'
weight(u,v) is the cost of the edge from 'u' to 'v'

4. AI model

The goal of the AI model in this project is to see if it is feasible to teach an AI model Dijkstra's Algorithm in a visual representation. Our approach with this model was to use Q learning so the model gets rewarded for minimizing the costs it takes to traverse the terrain. If the AI model is to autonomously navigate environments that have predetermined costs, then Q learning would be a compelling choice due to its capacity to handle problems with rewards. Q learning also is a model free learning algorithm, meaning it does not need to have a model of the environment and can learn solely based from interactions with the current environment, or any environment that it is given. This is why it is ideal for Unity based simulations where the outcomes of every action are not predetermined but can be learned.

Q learning works by learning an action-value function that provides the expected utility of taking a given action in a state and following a more fixed policy as a result of the previous actions. Q learning's core is based on a Q value rule which improves how the agent in Unity decides to navigate over time(4). The rule is expressed with this equation

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma max_{a'} Q(s', a') - Q(s, a)]$$

Where ***s*** and ***s'*** are the current and next states, ***a*** is the current action taken in s, ***r*** is the reward received after the action, ***α*** is the learning rate, and ***γ*** is the discount factor (3).

The learning rate controls how much new information overrides old information, meaning if the agent should decide to rely on older information or newer information. For this particular project, we decided since this is a new model and has not been trained previously, it might be better for it to continue to look toward newer information rather than old information. The discount factor determines how important future rewards are. In this case, the AI model could either look for direct routes with lower costs or look ahead at a certain rate where the long term costs might be less than the current costs. The formula as a whole adjusts the Q value to attempt to get the highest possible Q value from the next state, which guides the agent to learn better long term behaviors over time.

Our reasoning for using Q learning this way was because we hoped that rewarding the model to discover the shortest and least costly paths would be very effective in teaching it Dijkstra's Algorithm, since that algorithm is based on the same thing. The AI is expected to develop a strategy that mirrors this algorithm and would be able to apply it in other scenarios.
Given the environment's setup in Unity, where nodes and edges have costs associated with them, Q learning would be able to adapt and discover the optimal routing strategy over time. Each iteration of the learning process references the AI making a move and receiving a reward or penalty based on the cost of the move.
The reward system we decided to use was based on the negative value of the costs for each Node, meaning the AI's goal would be to be as close to 0 as possible. The negative reinforcement approach works in this case where there are numerous ways to take a path and making the wrong decision penalizes the AI to incentivise it to be punished less the next time.

The AI model's implementation involves utilizing the same nodes, paths and costs as the Dijkstra's algorithm model did, but changing the script to utilize Q learning instead of programming the initial algorithm inside of the player model. The learning algorithm is then left to interact with the terrain, and receives tuning if the model begins to learn the "incorrect" way. Each action the model takes will be derived from the Q value it discovers after creating an initial path. This model specifically used 6 iterations of the initial script to make sure that the Q learning method was effective enough to learn Dijkstra's algorithm.

The base code involving the AI script was also written in C#, since Unity supports it directly in its own engine. The node and edge setup were key to ensure that the model would recognize where to go regarding the Node prefabs described above. The Q value initialization sets up Q values for each node to node transition to 0, so it can later store the expected utility of future paths. The continuous learning loop is an infinite loop that runs the learning and pathing processes until the scene in Unity is terminated (7). The learn to navigate function helps the model select the next node based on either exploration or exploitation, and calculates a reward based on the choice it made. The reward system uses the negative costs so it learns to use as little cost as possible while making these decisions. Then the Q value would be updated based on the decision. The choose next node function decides the next node to move to based on both the value of the exploration rate and the choose best node function, which selects the node set with the highest Q value from the current node. Once a path is decided, the followPath function physically moves the agent through the path for visual representation. Lastly the reset environment function resets the player and terrain to allow for multiple instances to continuously run and learn without any need for developer input. Debugging and logs were developed so we could output and visualize how the AI was improving and if the AI was meeting our expectations (appendix C).

5. Results and Discussion

Iteration 1:
The first iteration of the AI model implemented the typical Q learning algorithm and also included an exploration rate which determines how often the model should be trying a random path to discover new paths even if it believes it has found the optimal path. This iteration was set with a learning rate of 0.8, a discount factor of 0.9 and an exploration rate of 0.2. The goal of this iteration was to test and get initial results using these rates and to find out where improvements could be made. Sadly this iteration did not perform well at all, only discovering the optimal path 36 times over 1644 instances, meaning something was not working properly in the script. This caused us to review our exploration rate, since we noticed that our first iteration had a lot of looping between nodes and large paths that were definitely not optimal (Iteration 1.txt).

Iteration 2:
As a result of iteration one, we implemented an exploration decay function that would lower the exploration rate at a factor of 0.995 or at a rate of .5% each time the model reset (Appendix C.2). This would reduce variability over time and hopefully allow the model to only use the best iterations it discovers later after a certain period of time. This implementation did not see much improvement, showing 42 instances of the optimal path from Dijkstra's algorithm out of 1630 (2.6% of the time).

Iteration 3:
`Since this did not work either, we decided to go with increasing the learning rate to 1 and decreasing the exploration rate to a constant 0.05 instead of initially being at 0.2. We thought that maybe there was an issue with our exploration rate's implementation and thought it would benefit the AI to always prioritize new data versus old data. This hypothesis was also incorrect, since the model could not handle constantly throwing out old data and replacing it as fast as it was. This demonstrates how the model did not have time to use prior knowledge of previous rewards and decided to do the least punishing action which would be looping through the least costly nodes without considering the end point. Looking closer into the loop, we noticed that it somehow thought that looping between nodes 1, 2, 3 and 4 continuously was a good thing, which we figured was the discount factor affecting the model to search for less punishing, immediate paths rather than looking for long term paths.

Iteration 4:
We determined that the looping was caused by two factors in our model. The first factor was the exploration rate, which was initially set too low from the beginning. The exploration rate being too low and also decreasing overtime led to the model never getting the chance to move to new paths, especially with a high learning rate. This proves that the model was throwing out too much data, and more randomly picking different paths to take.There was also a mistake with the way we set up our exploration rate, which we have still yet to truly pinpoint. However, increasing the exploration rate to 0.5 and changing the decay constant to 1.0 would prevent the rate from decreasing over time and also led to significant improvements with the model. This iteration used the optimal path 11.2% of the time over 1811 resets. Although this was a significant improvement from 2.5%, this was still not enough to show that this model shows promise in mirroring Dijkstra's algorithm.

Iteration 5:
Noticing the significant improvement, we decided to raise the exploration rate again to 0.9. We also decided to put the learning rate at 0.9 and adjust the discount factor from 0.9 to 0.75. Running this model continued to show significant improvement, using the optimal path 13.7% of the time with 1678 resets. However, we continued to notice in the debugging logs that a significant amount of paths taken were still paths that were outrageously long, meaning the model still decided to mitigate punishment immediately by looping between lower cost nodes. Lowering the discount factor helped, but lowering too much would have caused the model to never learn using more costly nodes, so we felt like .75 was the correct number, but something else had to be done to discourage the model from going to previously discovered nodes. We first tried to reward the model a lot by adding 100 to the reward pool if the model reached Node 6, but eventually the model learned that it was always going to reach Node 6 and would always get that reward. So we had to find a way to punish the model more when it looped too many times.

Iteration 6 (Final Iteration)
The final iteration implemented a discouragement system by adding a hash set of the nodes the model has already visited during that instance, and keeping track of them. Whenever the model revisited a node, a 500 point reduction in the rewards would be significant enough for the model to definitely steer clear (Appendix C.3). This worked extremely well, and was so promising that we decided to train it on that model for 10,000 instances instead of staying around 1600. Over 9,999 instances, with this last model, the AI visited the optimal path 1745 times or 17.5% (Figure 2, sixth value) of the time. Initially that does not seem like much, but looking closer at the data, the model's accuracy significantly increased during the last 1000 times it went through the path. The model traversed the optimal path 27.4% of the time (Figure 3, first value), which shows that if given enough time and tries, it potentially could end up reaching the optimal path even more frequently.

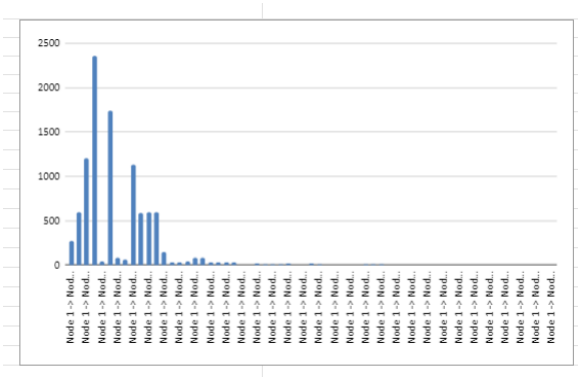

Figure 2. Frequencies of each path taken over the entire span

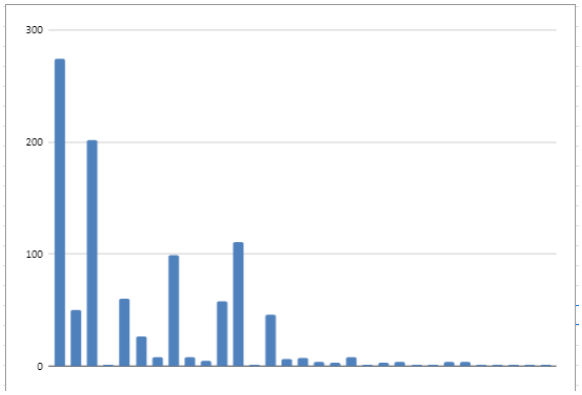

Figure 3. Frequencies of each path taken after the last 1000 instances

6. Conclusion

Overall, this research paper presented an in depth exploration and analysis of two different navigation models within a Unity based simulation environment. By using both traditional pathfinding algorithms and more modern reinforcement learning techniques, we aimed to discover the capabilities and limitations of AI in complex pathfinding tasks. This provided us with a foundational understanding of how traditional algorithms like Dijkstra's Algorithm.

The control model, using Dijkstra's Algorithm, served effectively as a benchmark. It demonstrated the practicality of theoretical algorithms in a simulated environment. This model showcased how predefined pathfinding logic could effectively navigate a graph based terrain in real world scenarios and in games where the least costly paths are optimal.

In contrast, the AI learning model powered by Q learning introduced dynamic learning capabilities where the AI was not following a straight rule set but was adapting based on its interaction with the environment. Over the course of several iterations, the model displayed a significant learning curve, illustrating both the challenges and potentials of employing reinforcement learning in pathfinding scenarios. The use of Q learning assisted the AI to attempt to optimize its decisions based on rewards tailored to the costs of traveling like Dijkstra's Algorithm.

Notably, the AI models performance evolved through precise adjustments to the learning parameters such as the exploration rate and the learning rate. These adjustments were crucial in mitigating issues like path looping and inefficient exploration strategies. Adding the penalty system at the end for revisiting nodes significantly pushed the model to explore more efficient paths and to avoid redundancy, which helped it more closely align with Dijkstra's Algorithm. In terms of time complexity, Dijkstra's Algorithm ended up being O(n^2) due to our lack of a priority queue implementation. Our Q learning model is O(n), where n represents each instance or each time the player moves through the path. This complexity depends on how many times you train the model, meaning since we train the model thousands of times it ends up multiplying by a constant that represents the amount of times the model ran through the paths.

The comparative analysis between the control and AI models were based on two findings in this project. The first finding was based on adaptability. The models ability to adapt its strategy over time reinforces the potential of reinforcement learning to go up against varied and dynamic problems that traditional algorithms may not solve. The second finding was Efficiency. While the AI model did not perform as efficiently as the control, its improvement shows promise for future optimizations and applications.

In terms of the future, some ways to improve upon this project could be to increase the amount of iterations in the final model, and also run multiple iterations of the same model to see how consistent our findings were. Secondly, improving our Q value calculations, making it more complex based on more focused factors that are tailored to the model's behavior could show significant improvement to the efficiency of the model. Once the model reaches a certain accuracy on this terrain, we can eventually use dynamic environments and see how the model performs with a greater volume of nodes and more complex edges between the nodes. We would also experiment with different learning techniques like deep learning or multi-agent systems (6).

7. References

S. Liao, "Navigation-AI-Agent," GitHub repository, 2023. [Online]. Available: https://github.com/soliao/Navigation-AI-Agent. [Accessed: April, 2024].

GeeksforGeeks, "A* Search Algorithm," 2023. [Online]. Available: https://www.geeksforgeeks.org/a-search-algorithm/. [Accessed: April, 2024].

Unity Technologies, "Unity AI: Reinforcement Learning with Q-Learning," Unity Blog, 2023. [Online]. Available: https://blog.unity.com/engine-platform/unity-ai-reinforcement-learning-with-q-learning [Accessed:April, 2024].

Unity Technologies, "Q-GridWorld," GitHub repository, 2023. [Online]. Available: https://github.com/Unity-Technologies/Q-GridWorld [Accessed:April, 2024].

Adilakshmi Siripurapu,Ravi Shankar Nowpada, K. Srinivasa Rao, “Improving Dijkstra’s algorithm for Estimating Project Characteristics and Critical Path,” pdf, RT&A, 2022 [Accessed: April, 2024].

Yusef Savid, Reza Mahmoudi, Rytis Maskeliunas, Robertas Damaševi˘cius, “Simulated Autonomous Driving Using Reinforcement Learning: A Comparative Study on Unity’s ML-Agents Framework“, pdf, MDPI, 2023 [Accessed: April 2024].

Olli-Pekka Juhola, “Creating Self-Learning AI using UnityMachine Learning“, pdf, jamk.fi, 2019 [Accessed April 2024].

Appendix A: Node Script (C#)

```
using System.Collections.Generic;
using UnityEngine;

public class Node : MonoBehaviour
{
   [System.Serializable]
   public struct Neighbor
   {
      public Node node;
      public float cost;
   }
```

```
    public List<Neighbor> neighbors = new List<Neighbor>();

    public void AddNeighbor(Node neighbor, float cost)
    {
        neighbors.Add(new Neighbor { node = neighbor, cost = cost });
    }
}
```

Appendix B: Control Model Pathfinding Script (C#)

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Pathfinding : MonoBehaviour
{
    public Node startNode;
    public Node endNode;
    public GameObject player;
    private List<Node> path = new List<Node>();
    public float speed = 5.0f; // Speed at which the player will move

    void Start()
    {
        StartCoroutine(FollowPath());
    }

    IEnumerator FollowPath()
    {
        var unvisited = new List<Node>(FindObjectsOfType<Node>());
        var dist = new Dictionary<Node, float>();
        var previous = new Dictionary<Node, Node>();

        foreach (Node node in unvisited)
        {
            dist[node] = float.MaxValue;
            previous[node] = null;
        }

        dist[startNode] = 0;

        while (unvisited.Count > 0)
        {
            Node current = null;
            float minDistance = float.MaxValue;

            foreach (Node n in unvisited)
            {
                if (dist[n] < minDistance)
                {
```

```
                minDistance = dist[n];
                current = n;
            }
        }

        if (current == null) break;

        unvisited.Remove(current);

        foreach (Node.Neighbor neighbor in current.neighbors)
        {
            Node neighborNode = neighbor.node;
            float alt = dist[current] + neighbor.cost;
            if (alt < dist[neighborNode])
            {
                dist[neighborNode] = alt;
                previous[neighborNode] = current;
            }
        }
    }

    // Build the path from the end node back to the start node
    Node step = endNode;

    while (step != null && step != previous[step])
    {
        path.Insert(0, step);
        step = previous[step];
    }
    path.Insert(0, startNode); // Add the start node at the beginning of the path

    // Move the player along the path
    foreach (Node node in path)
    {
        // Move towards each node one step at a time
        Vector3 startPosition = player.transform.position;
        Vector3 endPosition = node.transform.position;
        float journeyLength = Vector3.Distance(startPosition, endPosition);
        float startTime = Time.time;

        while (player.transform.position != endPosition)
        {
            float distCovered = (Time.time - startTime) * speed;
            float fractionOfJourney = distCovered / journeyLength;
            player.transform.position = Vector3.Lerp(startPosition, endPosition, fractionOfJourney);
            yield return null; // Wait until next frame before continuing execution
        }
        Debug.Log("Arrived at " + node.name);
    }
  }
}
```

Appendix C AI model Scripts

C.1 Iteration 1 Script

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.Text; // Needed for StringBuilder

public class AIPathfindingQ : MonoBehaviour
{
    public Node startNode;
    public Node endNode;
    public GameObject player;
    private Dictionary<(Node, Node), float> QValues = new Dictionary<(Node, Node), float>();
    public float learningRate = 0.8f;
    public float discountFactor = 0.9f;
    public float explorationRate = 0.2f;
    private List<Node> path = new List<Node>();
    public float speed = 5.0f;
    private int resetCounter = 0; // Counter to track the number of resets
    void Start()
    {
        InitializeQValues();
        StartCoroutine(ContinuousLearning());
    }

    void InitializeQValues()
    {
        Node[] allNodes = FindObjectsOfType<Node>();
        foreach (Node node in allNodes)
        {
            foreach (Node.Neighbor neighbor in node.neighbors)
            {
                if (!QValues.ContainsKey((node, neighbor.node)))
                {
                    QValues[(node, neighbor.node)] = 0f; // Initialize all Q-values to zero
                }
            }
        }
    }



    IEnumerator ContinuousLearning()
    {
        while (true) // Loop indefinitely
        {
            yield return StartCoroutine(LearnToNavigate());
            BuildAndFollowPath();
            LogPath(); // Log the path taken
            ResetEnvironment(); // Reset the environment for the next run
            UpdateExplorationRate(); // Decrease exploration rate after each run

        }
    }
```

```
IEnumerator LearnToNavigate()
{
    Node currentNode = startNode;
    while (currentNode != endNode)
    {
        Node nextNode = ChooseNextNode(currentNode);
        float reward = -currentNode.neighbors.Find(n => n.node == nextNode).cost; // Negative cost as reward

        float oldQValue = QValues[(currentNode, nextNode)];
        float maxQ = float.MinValue;

        foreach (var neighbor in nextNode.neighbors)
        {
            float qValue = QValues[(nextNode, neighbor.node)];
            if (qValue > maxQ)
            {
                maxQ = qValue;
            }
        }

        float newQValue = oldQValue + learningRate * (reward + discountFactor * maxQ - oldQValue);
        QValues[(currentNode, nextNode)] = newQValue;

        currentNode = nextNode;

        yield return null;
    }
}

Node ChooseNextNode(Node currentNode)
{
    if (Random.value < explorationRate)
    {
        return currentNode.neighbors[Random.Range(0, currentNode.neighbors.Count)].node;
    }
    else
    {
        return GetBestNextNode(currentNode);
    }
}

Node GetBestNextNode(Node currentNode)
{
    float maxQ = float.MinValue;
    Node bestNode = null;
    foreach (var neighbor in currentNode.neighbors)
    {
        float qValue = QValues[(currentNode, neighbor.node)];
        if (qValue > maxQ)
        {
            maxQ = qValue;
            bestNode = neighbor.node;
        }
    }
```

```
        return bestNode ?? currentNode.neighbors[Random.Range(0, currentNode.neighbors.Count)].node; // Fallback
to random if no best node
    }

    void ResetEnvironment()
    {
        player.transform.position = startNode.transform.position; // Reset player position to the start node
        resetCounter++; // Increment the counter each time the environment is reset
        Debug.Log("Environment has been reset " + resetCounter + " times.");
    }

    void BuildAndFollowPath()
    {
        Node currentNode = startNode;
        path.Clear();
        while (currentNode != endNode)
        {
            path.Add(currentNode);
            currentNode = ChooseNextNode(currentNode);
        }
        path.Add(endNode);
        StartCoroutine(FollowPath());
    }

    IEnumerator FollowPath()
    {
        foreach (Node node in path)
        {
            Vector3 startPosition = player.transform.position;
            Vector3 endPosition = node.transform.position;
            while (Vector3.Distance(player.transform.position, endPosition) > 0.01f)
            {
                player.transform.position = Vector3.MoveTowards(player.transform.position, endPosition, speed *
Time.deltaTime);
                yield return null;
            }
        }
        Debug.Log("Arrived at end.");
    }

    void LogPath()
    {
        StringBuilder pathLog = new StringBuilder("Current Path: ");
        foreach (Node node in path)
        {
            pathLog.Append(node.name + " -> ");
        }
        pathLog.Append("End");
        Debug.Log(pathLog.ToString());
    }
}
```

C.2 Iteration 2 (Exploration Rate Decay)

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.Text; // Needed for StringBuilder

public class AIPathfindingQ : MonoBehaviour
{
    public Node startNode;
    public Node endNode;
    public GameObject player;
    private Dictionary<(Node, Node), float> QValues = new Dictionary<(Node, Node), float>();
    public float learningRate = 0.8f;
    public float discountFactor = 0.9f;
    public float explorationRate = 0.2f;
    private List<Node> path = new List<Node>();
    public float speed = 5.0f;
    private int resetCounter = 0; // Counter to track the number of resets

    public float explorationDecay = 0.995f;

    public float minExplorationRate = 0.05f; // Minimum exploration rate

    void Start()
    {
        InitializeQValues();
        StartCoroutine(ContinuousLearning());
    }

    void InitializeQValues()
    {
        Node[] allNodes = FindObjectsOfType<Node>();
        foreach (Node node in allNodes)
        {
            foreach (Node.Neighbor neighbor in node.neighbors)
            {
                if (!QValues.ContainsKey((node, neighbor.node)))
                {
                    QValues[(node, neighbor.node)] = 0f; // Initialize all Q-values to zero
                }
            }
        }
    }

    void UpdateExplorationRate()
{
    explorationRate *= explorationDecay;
    explorationRate = Mathf.Max(explorationRate, minExplorationRate);
}

    IEnumerator ContinuousLearning()
    {
        while (true) // Loop indefinitely
```

```
    {
        yield return StartCoroutine(LearnToNavigate());
        BuildAndFollowPath();
        LogPath(); // Log the path taken
        ResetEnvironment(); // Reset the environment for the next run
        UpdateExplorationRate(); // Decrease exploration rate after each run

    }
}

IEnumerator LearnToNavigate()
{
    Node currentNode = startNode;
    while (currentNode != endNode)
    {
        Node nextNode = ChooseNextNode(currentNode);
        float reward = -currentNode.neighbors.Find(n => n.node == nextNode).cost; // Negative cost as reward

        float oldQValue = QValues[(currentNode, nextNode)];
        float maxQ = float.MinValue;

        foreach (var neighbor in nextNode.neighbors)
        {
            float qValue = QValues[(nextNode, neighbor.node)];
            if (qValue > maxQ)
            {
                maxQ = qValue;
            }
        }

        float newQValue = oldQValue + learningRate * (reward + discountFactor * maxQ - oldQValue);
        QValues[(currentNode, nextNode)] = newQValue;

        currentNode = nextNode;

        yield return null;
    }
}

Node ChooseNextNode(Node currentNode)
{
    if (Random.value < explorationRate)
    {
        return currentNode.neighbors[Random.Range(0, currentNode.neighbors.Count)].node;
    }
    else
    {
        return GetBestNextNode(currentNode);
    }
}

Node GetBestNextNode(Node currentNode)
{
    float maxQ = float.MinValue;
    Node bestNode = null;
    foreach (var neighbor in currentNode.neighbors)
```

```
        {
            float qValue = QValues[(currentNode, neighbor.node)];
            if (qValue > maxQ)
            {
                maxQ = qValue;
                bestNode = neighbor.node;
            }
        }
        return bestNode ?? currentNode.neighbors[Random.Range(0, currentNode.neighbors.Count)].node; // Fallback
to random if no best node
    }

    void ResetEnvironment()
    {
        player.transform.position = startNode.transform.position; // Reset player position to the start node
        resetCounter++; // Increment the counter each time the environment is reset
        Debug.Log("Environment has been reset " + resetCounter + " times.");
    }

    void BuildAndFollowPath()
    {
        Node currentNode = startNode;
        path.Clear();
        while (currentNode != endNode)
        {
            path.Add(currentNode);
            currentNode = ChooseNextNode(currentNode);
        }
        path.Add(endNode);
        StartCoroutine(FollowPath());
    }

    IEnumerator FollowPath()
    {
        foreach (Node node in path)
        {
            Vector3 startPosition = player.transform.position;
            Vector3 endPosition = node.transform.position;
            while (Vector3.Distance(player.transform.position, endPosition) > 0.01f)
            {
                player.transform.position = Vector3.MoveTowards(player.transform.position, endPosition, speed *
Time.deltaTime);
                yield return null;
            }
        }
        Debug.Log("Arrived at end.");
    }

    void LogPath()
    {
        StringBuilder pathLog = new StringBuilder("Current Path: ");
        foreach (Node node in path)
        {
            pathLog.Append(node.name + " -> ");
        }
        pathLog.Append("End");
```

```
        Debug.Log(pathLog.ToString());
    }
}
```

C.3 Final Iteration (Penalty for repeating Nodes)

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.Text; // Needed for StringBuilder

public class AIPathfindingQ : MonoBehaviour
{
    public Node startNode;
    public Node endNode;
    public GameObject player;
    private Dictionary<(Node, Node), float> QValues = new Dictionary<(Node, Node), float>();
    public float learningRate = 0.8f;
    public float discountFactor = 0.9f;
    public float explorationRate = 0.2f;
    private List<Node> path = new List<Node>();
    public float speed = 5.0f;
    private int resetCounter = 0; // Counter to track the number of resets

    public float explorationDecay = 0.995f;

    public float minExplorationRate = 0.05f; // Minimum exploration rate

    void Start()
    {
        InitializeQValues();
        StartCoroutine(ContinuousLearning());
    }

    void InitializeQValues()
    {
        Node[] allNodes = FindObjectsOfType<Node>();
        foreach (Node node in allNodes)
        {
            foreach (Node.Neighbor neighbor in node.neighbors)
            {
                if (!QValues.ContainsKey((node, neighbor.node)))
                {
                    QValues[(node, neighbor.node)] = 0f; // Initialize all Q-values to zero
                }
            }
        }
    }

    void UpdateExplorationRate()
{
```

```
    explorationRate *= explorationDecay;
    explorationRate = Mathf.Max(explorationRate, minExplorationRate);
}

    IEnumerator ContinuousLearning()
    {
        while (true) // Loop indefinitely
        {
            yield return StartCoroutine(LearnToNavigate());
            BuildAndFollowPath();
            LogPath(); // Log the path taken
            ResetEnvironment(); // Reset the environment for the next run
            UpdateExplorationRate(); // Decrease exploration rate after each run

        }
    }

    IEnumerator LearnToNavigate()
{
    Node currentNode = startNode;
    HashSet<Node> visitedNodes = new HashSet<Node>(); // Track visited nodes
    visitedNodes.Add(currentNode); // Start node is visited at the beginning

    while (currentNode != endNode)
    {
        Node nextNode = ChooseNextNode(currentNode);
        float reward = -currentNode.neighbors.Find(n => n.node == nextNode).cost; // Negative cost as reward

        if (visitedNodes.Contains(nextNode))
        {
            reward -= 500; // Apply a penalty for revisiting a node
        }
        else
        {
            visitedNodes.Add(nextNode); // Add to visited if it's a new node
        }

        float oldQValue = QValues[(currentNode, nextNode)];
        float maxQ = float.MinValue;

        foreach (var neighbor in nextNode.neighbors)
        {
            float qValue = QValues[(nextNode, neighbor.node)];
            if (qValue > maxQ)
            {
                maxQ = qValue;
            }
        }

        float newQValue = oldQValue + learningRate * (reward + discountFactor * maxQ - oldQValue);
        QValues[(currentNode, nextNode)] = newQValue;

        currentNode = nextNode;

        yield return null;
    }
```

```
}

    Node ChooseNextNode(Node currentNode)
    {
        if (Random.value < explorationRate)
        {
            return currentNode.neighbors[Random.Range(0, currentNode.neighbors.Count)].node;
        }
        else
        {
            return GetBestNextNode(currentNode);
        }
    }

    Node GetBestNextNode(Node currentNode)
    {
        float maxQ = float.MinValue;
        Node bestNode = null;
        foreach (var neighbor in currentNode.neighbors)
        {
            float qValue = QValues[(currentNode, neighbor.node)];
            if (qValue > maxQ)
            {
                maxQ = qValue;
                bestNode = neighbor.node;
            }
        }
        return bestNode ?? currentNode.neighbors[Random.Range(0, currentNode.neighbors.Count)].node; // Fallback
to random if no best node
    }

    void ResetEnvironment()
    {
        player.transform.position = startNode.transform.position; // Reset player position to the start node
        resetCounter++; // Increment the counter each time the environment is reset
        Debug.Log("Environment has been reset " + resetCounter + " times.");
    }

    void BuildAndFollowPath()
    {
        Node currentNode = startNode;
        path.Clear();
        while (currentNode != endNode)
        {
            path.Add(currentNode);
            currentNode = ChooseNextNode(currentNode);
        }
        path.Add(endNode);
        StartCoroutine(FollowPath());
    }

    IEnumerator FollowPath()
    {
        foreach (Node node in path)
        {
```

```
            Vector3 startPosition = player.transform.position;
            Vector3 endPosition = node.transform.position;
            while (Vector3.Distance(player.transform.position, endPosition) > 0.01f)
            {
                player.transform.position = Vector3.MoveTowards(player.transform.position, endPosition, speed *
Time.deltaTime);
                yield return null;
            }
        }
        Debug.Log("Arrived at end.");
    }

    void LogPath()
    {
        StringBuilder pathLog = new StringBuilder("Current Path: ");
        foreach (Node node in path)
        {
            pathLog.Append(node.name + " -> ");
        }
        pathLog.Append("End");
        Debug.Log(pathLog.ToString());
    }
}
```