Navigation Simulation in Machine Learning with Unity Draft

Joseph Guitian & John DiGabriele

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 (not fully completed) 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.

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 (under development), 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 has been selected for this purpose due to its success in similar learning scenarios. 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 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.

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. 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.

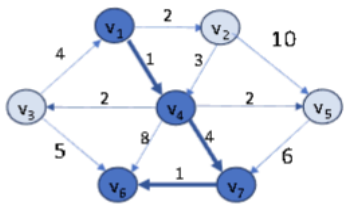

Figure 1

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).

AI model (in progress)

Results and Discussion

References (not formatted, see GitHub for now)

Appendix A: Control Model 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);
        }
    }
}
```