Lab 8 - Path Finding

COMP396-Game Programming 2

Purpose: To implement Path Finding via:

- BFS Algorithm (Breadth First Search)
- DFS Algorithm (Depth First Search)
- Dijstra Algorithm
- A* Algorithm

Contents

1 Introduction

Let's implement the following:

- A Graph data structure. We need to select one of the ways to represent the graphs:
    - Set (List) of Vertices: Vertices can be represented via:

        - `ints`,
        - `chars` (for very small graphs)
        - `strings`,
        - a (templatized) `structure`,
        - a (templatized) `class`,
        - ...
    - Set (List) of Edges: Edges can be represented via:
        - pairs of vertices,
        - pairs of `ids` of vertices,
        - Adjacency Matrix
        - Linked List of `ids` of vertices neighbour to a vertex
        - a (templatized) `structure`,
        - a (templatized) `class`,
        - ...
- A few simple algorithms in graphs
    - find all descendants of a given vertex
    - find all parents (antescendants) of a given vertex
    - find all neighbors (descendants and parents) of a given vertex
    - find a spanning tree (Prim's algorithm)
    - find an Hamilton Circuit (The TSP - The Traveling Salesman Problem)
    - find a path from a vertex to another vertex (BFS, DFS, Dijkstra, A*)
    - find a shortest path from a vertex to another vertex (Dijkstra, A*)
    - find a shortest path from a vertex to every other vertex (Dijkstra)
    - find a shortest path from every vertex to every other vertex (
- Implementing these algorithms:
    - BFS Algorithm (Breadth First Search)
    - DFS Algorithm (Depth First Search)
    - Dijstra Algorithm
    - A* Algorithm

2 Basic Path Finding behavior

Definitions:

- A **digraph** is a **graph** with *directed* edges (hence the *di-* prefix). **Graph** naming is usually reserved for *un-directed* graphs.
- A **directed edge** in a [di]graph is a pair of vertices, where the first is referred to as the **start** vertex and second as the **end** vertex. `Graphs` can always be

converted into `di-graphs` by converting each *un-directed* edge `(A,B)` into two directed edges `(A,B)` and `(B,A)`.

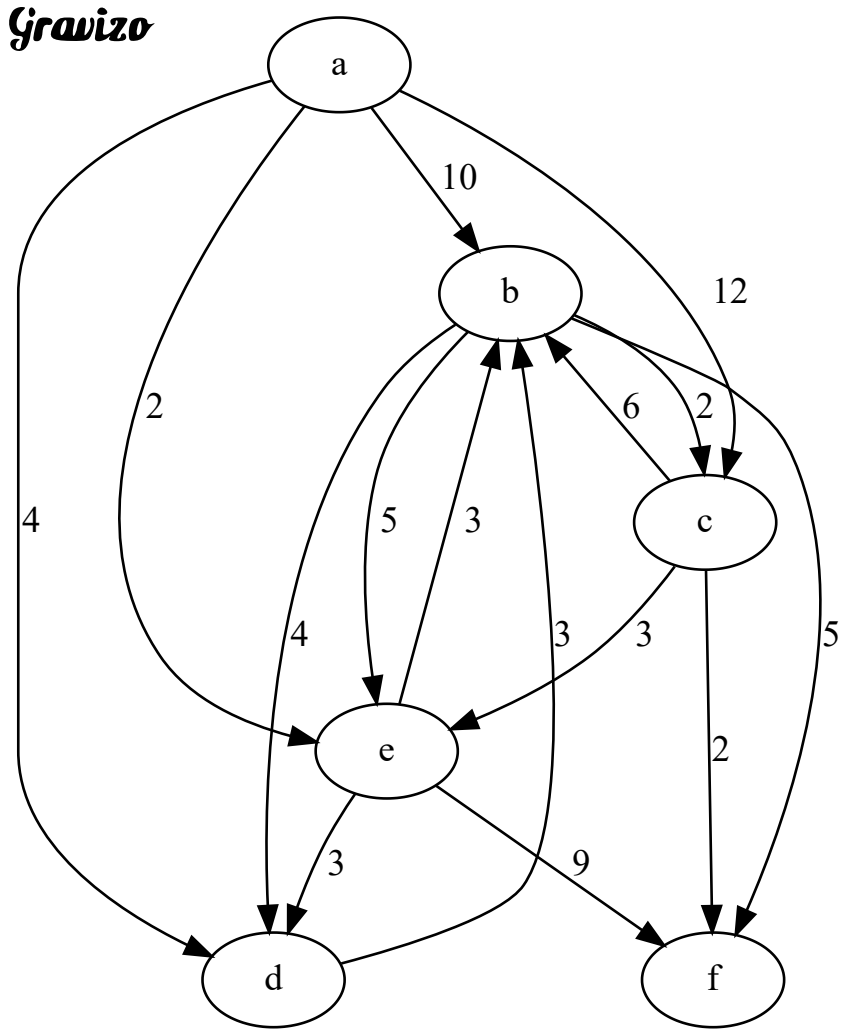
- A **path** in a [di]graph is a sequence of edges of the graph whereby the ending vertex of an edge is the starting point of the next edge.
- A path **connecting** vertices **A** and **B** is a path with *starting* vertex of the first edge the vertex **A** and the ending vertex of the last edge the vertex **B**.

2.1 Demo Graph

2.1.1 Graph Drawing Tools

- FSMs are also examples of graphs, so the tools to draw FSMs can be used to draw graphs.
- Additional specialized tools to draw graphs on the web:
  - g.gravizo.com (markdeep makes use of the above tool)
  - graphviz.org (other web tools make use of this behind the scenes, like https://sketchviz.com/).

Here is an example of a simple digraph:



Weighted Digraph Example for Dijkstra Algorithm

2.2 Demo Scene

- Open the last project
- Create a folder under Scenes folder named **Lab8_PathFinding_{YourInitials}**
- Set up a demo scene, named **Lab8_PathFinding_{YourInitials}**, with the following game objects:

Game Object	Type	Parent	P(x,y,z)	R(x,y,z)	S(x,y,z)	Color/Texture
Floor	Plane	Root	(0,0,0)	(0,0,0)	(10,1,10)	Beige
Start	Empty	Root	(−9,0,0)	(0,0,0)	(1,1,1)	-
End	Empty	Root	(9,0,0)	(0,0,0)	(1,1,1)	-
Obstacles	Empty	Root	(0,0,0)	(0,0,0)	(1,1,1)	-
Obstacle1-9	Cube	Root	Random	Random	Random	Gray/Wall

> Make sure we put markers in the waypoints so we can see them in scene view. You may download a suitable wall texture provided you put a proper reference in you submission.

- *Take Snapshot*

Let's write a **Graph.cs** script with a simple implementation of a graph data structure.

2.3 Graph.cs Script

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class Graph
{
    Dictionary<char, dictionary<char,="" int="">> vertices = new Dictionary<char,
dictionary<char,="" int="">>();
    public void add_vertex(char vertex, Dictionary<char, int=""> edges)
    {
        vertices[vertex] = edges;
    }
    public List<char> shortest_path_via_Dijkstra(char start, char finish)
    {
        //initialize
        List<char> path = new List<char>();
        var distances=new Dictionary<char, int="">();
        var previous = new Dictionary<char, char="">();
        var Pending = new List<char>();
```

```
        //step 0
        foreach (var v in vertices)
        {
            distances[v.Key] = int.MaxValue;
            previous[v.Key] = '\0';
            Pending.Add(v.Key);
        }
        distances[start] = 0;
        //main loop
        while (Pending.Count > 0)
        {
            Pending.Sort((x,y)=> distances[x].CompareTo(distances[y]));
            var u = Pending[0];
            // TODO
            // ...
            // ...
        }
        return path;
    }
}
```

- *Take Snapshot*

2.4 Create TestDijkstraAlgorithm.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class TestingDijkstraAlgorithm : MonoBehaviour
{
    // Use this for initialization
    void Start()
    {
        Graph g = new Graph();
        g.add_vertex('A', new Dictionary<char, int="">() { { 'B', 10 }, { 'C', 12 },
{ 'D', 4 }, { 'E', 2 } });
        g.add_vertex('B', new Dictionary<char, int="">() { { 'C', 2 }, { 'D', 4 }, {
'E', 5 }, { 'F', 5 } });
        g.add_vertex('C', new Dictionary<char, int="">() { { 'B', 6 }, { 'F', 2 }
});
        g.add_vertex('D', new Dictionary<char, int="">() { { 'B', 3 }, { 'E', 3 }
});
        g.add_vertex('E', new Dictionary<char, int="">() { { 'B', 3 }, { 'D', 3 }, {
```

```
'F', 9 } });
        g.add_vertex('F', new Dictionary<char, int="">());

        print("g:" + g);
        //List<char> shortest_path = g.shortest_path('A', 'F');
    }
}
```

-Take Snapshot

2.5 BFS - Breadth First Search

The pseudo code for BFS looks like:

```
procedure BFS(G, root, goal) is
  let Q be a queue
  label root as explored
  Q.enqueue(root)
  while Q is not empty do
    v := Q.dequeue()
    if v is the goal then
      return v
    for all edges from v to w in G.adjacentEdges(v) do
      if w is not labeled as explored then
        label w as explored
        w.parent := v
        Q.enqueue(w)
```

2.6 DFS - Depth First Search

The pseudo code for DFS looks like:

```
procedure DFS(G, v) is
  label v as discovered
  for w in G.adjacentEdges(v) do
    if vertex w is not labeled as discovered then
      recursively call DFS(G, w)
```

2.7 Dijkstra Search

The pseudo code for Diskstra Algorithm looks like:

```
function dijkstra(graph, start, isGoal)
  queue ← new PriorityQueue()
  queue.insert(start, 0)
  distances[v] ← inf (∀ v ∈ graph | v <> start)
  parents[v] ← null (∀ v ∈ graph)
  distances[start] ← 0
  while not queue.empty() do
    v ← queue.top()
    if isGoal(v) then
      return (v, parents)
    else
      for e in graph.adjacencyList[v] do
        u ← e.dest
        if distances[u] > distances[v] + e.weight then
          distances[u] ← e.weight + distances[v]
          parents[u] ← v
          queue.update(u, distances[u])
  return (null, parents)
```

This pseudo-code uses a `priorityqueue` data structure.

The priorityqueue implementation can be simulated (for our example case) with a regular list whereby after each `add/update` operation a `sort` is performed; for industrial use, any implementation of a priority queue will do. Tasks:

- Task 0: Use Graphviz to draw the graph given. Save as .png file (Done in Class)
- Task 1: Create Graph.cs (with partial implementation of Dijkstra Algorithm - the **shortest_path_via_Dijkstra** method) (Done in Class)
- Task 2: Create TestDijkstraAlgorithm.cs (Graph g=...) (Done in class) *Take Snapshot*
- Task 3: Implement DFS as a method of the Graph class. Test it. *Take Snapshot*
- Task 4 : Implement the rest of Dijkstra Algorithm (the **shortest_path_via_Dijkstra**1

methosd) - Task 0': Use Graphviz to draw the graph drawn in whiteboard. Save as .png file (Done in Class) - Task 2': Update the TestDijkstraAlgorithm.cs to build the above graph (Graph g2=...) *Take Snapshot*

3 Summary

In this chapter, we:

- set up two scenes
- studied how to build agents with
  - **path-following** behavior, and
  - **obstacle avoidance** behavior.
- Worked with the following Unity3D features:
  - **layers** and **layer masks**
  - **Raycasts**, and
  - **SphereCasts**
- To effectuate the movement needed we used **Steering** (application of Newton's Second Law)

In the next lab, we'll study how to implement a pathfinding algorithm, called A*, to determine the optimal path before moving, while still avoiding static obstacles.

Challenge(s): In path following section was decided this challenge:

- An alternative solution (strategy) is to program it so that our object turns around and goes back the way it came. Take this as a challenge.