Lab 9 - Path Finding - Continuing with AStar algorithm

COMP396-Game Programming 2

Purpose: To **study** and **implement** Path Finding via:

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

Contents

1. Introduction

In Lab8 we implemented the following:

- A Graph data structure. We needed to select one of the ways to represent the graphs:
    - Set (List) of Vertices and Set (List) of Edges.
        - Vertices can be represented via:
            - `int`s,
            - `char`s (for very small graphs) **we used this**
            - `string`s,
            - a (templatized) `structure`,
            - a (templatized) `class`,
            - ...
        - Edges can be represented via:
            - pairs of vertices,
            - pairs of `id`s of vertices,
            - Adjacency Matrix
            - Linked List of `id`s of vertices neighbour to a vertex
            - a (templatized) `structure`,
            - a (templatized) `class`,
            - ...
- Dictionary with KeyType the NodeType and ValueType the type of the collection of a=edges that are adjacent to the correspondin vertex (**we used this method**)

> The above is enough to represent an abstract graph, and therefor to be able to implement algorithms such as **BFS** and **DFS**. However **weighted graph algorithms** (such as **Dijkstra's** and/or **ASTar** need some extra information per edge: **weights** (or lengths/costs/times etc). Additionally, some algorithms (like A*), may require us to “pay” more with some more information, with the promise to give us in “return” better efficiency (A* reqires us to “pay” by giving the **heuristic function**, that is an estimate of the distance from each node to the target.)

- A few simple algorithms in graphs

> The complexity (or lack thereof) of these algorithms depends of the graph representation we have chosen.

- 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 (Dijkstra, A* running multiple times)

- Implemented these algorithms:

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

We will implement hereby the A* Algorithm.

2. A* Algorithm for PathFinding

Definitions:

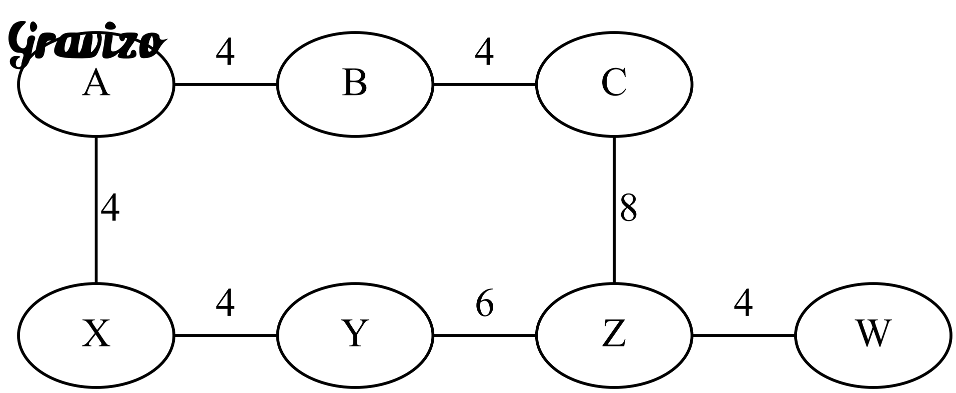
- A **digraph** is a **graph** with *directed* edges (hence the *di-* prefix). **Graph** naming is usually reserved for *un-directed* graphs.
- A **directed edge** in a [di]graph is a pair of vertices, where the first is referred to as the **start** vertex and second as the **end** vertex. `Graph`s can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

2.2. Modify the Graph.cs Script

Let's modify our **Graph.cs** script from Lab8 with:

- An enumeration for the heuristic strategy: **Euclidean** or **Manhattan**

- A consructor that accepts one of those strategies
- a method `add_vertex_for_AStar` and a skeleton of method `shortest_path_via_AStar`.
- the required methods for **AStar** like:
  - EuclideanDistance
  - ManhatanDistance
  - DictionaryDistance
  - GoalDistanceEstimate
- A skeleton for the method `shortest_path_via_AStar_algorithm`

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public enum HeuristicStrategy { EuclideanDistance, ManhatanDistance, DictionaryDistance };
public class Graph
{
  HeuristicStrategy strategy;
  public Graph(HeuristicStrategy strategy=HeuristicStrategy.EuclideanDistance)
  {
    this.strategy = strategy;
  }
  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;
  }
  Dictionary<char, Vector3> verticesData = new Dictionary<char, Vector3>();
  public void add_vertex_for_AStar_with_position(char vertex, Vector3 pos, Dictionary<char, int> edges)
  {
    vertices[vertex] = edges;
    verticesData[vertex] = pos;
  }
  public void add_vertex_for_AStar_with_heuristic(char vertex, float heuristic, Dictionary<char, int>
edges)
  {
    vertices[vertex] = edges;
    verticesData[vertex] = Vector3.zero;
    verticesData[vertex].x=heuristic;    //we use the 'x' component to save h[n]
  }
  public float EuclideanDistance(Vector3 v1, Vector3 v2)
  {
    return Vector3.Distance(v1, v2);
  }
  public float ManhatanDistance(Vector3 v1, Vector3 v2)
  {
    return Mathf.Abs(v1.x - v2.x) + Mathf.Abs(v1.y - v2.y) + Mathf.Abs(v1.z - v2.z);
  }
  public float GoalDistanceEstimate(char node, char finish)
  {
    float res = 0f;
    switch (strategy)
    {
      case HeuristicStrategy.EuclideanDistance:
        res = EuclideanDistance(verticesData[node], verticesData[finish]);
        break;
      case HeuristicStrategy.ManhatanDistance:
        res = ManhatanDistance(verticesData[node], verticesData[finish]);
        break;
      case HeuristicStrategy.DictionaryDistance:
        res = verticesData[node].x;
        break;
      default:
        break;
    }
    return res;
  }
```

```
public List<char> shortest_path_via_AStar_algo(char start, char finish)
{
  List<char> path = null;
  var previous = new Dictionary<char, char>();
  var distances = new Dictionary<char, float>(); //try to put fScore (= gScore+hScore)
  var gScore = new Dictionary<char, float>();

  var Pending = new List<char>(); //Open priority queue
  var Closed = new List<char>();  //Closed list
  //Step 0
  gScore[start] = 0;
  float hScore = GoalDistanceEstimate(start, finish);
  distances[start] = gScore[start] + hScore;
  previous[start] = '-';
  Pending.Add(start);
  //main loop
  while (Pending.Count > 0)
  {
    Pending.Sort((x,y)=> distances[x].CompareTo(distances[y]));
    var u = Pending[0];
    // TODO
    // ...
    // ...
  }
  return path;
}
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*

⚠ You may have to do some casting from `int` to `float` and vice-versa.

2.3. Create TestAStarAlgorithm.cs

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

```
using UnityEngine;
public class TestAStarAlgorithm : MonoBehaviour
{
  // Use this for initialization
  void Start()
  {
    //Testing with EuclideanDistance heuristic
    Graph g = new Graph();
    Graph g = new Graph(HeuristicStrategy.EuclideanDistance);
    g.add_vertex_for_AStar('A', new Vector3(0, 4, 0), new Dictionary<char, int>() { { 'B', 4 }, { 'X', 
20 } });
    g.add_vertex_for_AStar('B', new Vector3(4, 4, 0), new Dictionary<char, int>() { { 'A', 4 }, { 'C', 4
} });
    g.add_vertex_for_AStar('C', new Vector3(8, 4, 0), new Dictionary<char, int>() { { 'B', 4 }, { 'Z', 4
} });
    g.add_vertex_for_AStar('X', new Vector3(0, 0, 0), new Dictionary<char, int>() { { 'A', 20 }, { 'W', 
4 }, { 'Y', 4 } });
    g.add_vertex_for_AStar('Y', new Vector3(4, 0, 0), new Dictionary<char, int>() { { 'X', 4 }, { 'Z', 6
} });
    g.add_vertex_for_AStar('Z', new Vector3(8, 0, 0), new Dictionary<char, int>() { { 'C', 4 }, { 'Y', 4
} });
    g.add_vertex_for_AStar('W', new Vector3(12, 0, 0), new Dictionary<char, int>() { { 'X', 4 } });
    print("g:" + g);
   //List<char> shortest_path = g.shortest_path_via_AStar_algo('A', 'X');

    //Testing with ManhattanDistance heuristic
    Graph g2 = new Graph();
    Graph g2 = new Graph(HeuristicStrategy.ManhattanDistance);
    g2.add_vertex_for_AStar('A', new Vector3(0, 4, 0), new Dictionary<char, int>() { { 'B', 4 }, { 'X', 
20 } });
    g2.add_vertex_for_AStar('B', new Vector3(4, 4, 0), new Dictionary<char, int>() { { 'A', 4 }, { 'C', 
4 } });
    g2.add_vertex_for_AStar('C', new Vector3(8, 4, 0), new Dictionary<char, int>() { { 'B', 4 }, { 'Z', 
4 } });
    g2.add_vertex_for_AStar('X', new Vector3(0, 0, 0), new Dictionary<char, int>() { { 'A', 20 }, { 'W', 
4 }, { 'Y', 4 } });
    g2.add_vertex_for_AStar('Y', new Vector3(4, 0, 0), new Dictionary<char, int>() { { 'X', 4 }, { 'Z', 
6 } });
    g2.add_vertex_for_AStar('Z', new Vector3(8, 0, 0), new Dictionary<char, int>() { { 'C', 4 }, { 'Y', 
4 } });
    g2.add_vertex_for_AStar('W', new Vector3(12, 0, 0), new Dictionary<char, int>() { { 'X', 4 } });
    print("g2:" + g2);
   //List<char> shortest_path = g.shortest_path_via_AStar_algo('A', 'X');

    //Testing with DictionaryDistance heuristic
    Graph g3 = new Graph();
    Graph g3 = new Graph(HeuristicStrategy.DictionaryDistance);
    g3.add_vertex_for_AStar('A', new Vector3(12, 0 ,0), new Dictionary<char, int>() { { 'B', 4 }, { 'X', 
20 } });
    g3.add_vertex_for_AStar('B', new Vector3(8, 0, 0), new Dictionary<char, int>() { { 'A', 4 }, { 'C', 
4 } });
    g3.add_vertex_for_AStar('C', new Vector3(4, 0, 0), new Dictionary<char, int>() { { 'B', 4 }, { 'Z', 
4 } });
    g3.add_vertex_for_AStar('X', new Vector3(8, 0, 0), new Dictionary<char, int>() { { 'A', 20 }, { 'W', 
4 }, { 'Y', 4 } });
    g3.add_vertex_for_AStar('Y', new Vector3(4, 0, 0), new Dictionary<char, int>() { { 'X', 4 }, { 'Z', 
6 } });
    g3.add_vertex_for_AStar('Z', new Vector3(0, 0, 0), new Dictionary<char, int>() { { 'C', 4 }, { 'Y', 
4 } });
    g3.add_vertex_for_AStar('W', new Vector3(4, 0, 0), new Dictionary<char, int>() { { 'X', 4 } });
    print("g3:" + g3);
   //List<char> shortest_path = g.shortest_path_via_AStar_algo('A', 'X');

  }
}
```

-*Take Snapshot*

2.4. A* Search pseudocode

The pseudo code for A* looks like:

```
priorityqueue Open
list Closed

s.g = 0  # s is the start node
s.h = GoalDistEstimate( s )
s.f = s.g + s.h
s.parent = null
push s on Open
while Open is not empty
  pop node n from Open  # n has the lowest f
  if n is a goal node
    construct path
    return success
  for each successor n' of n
    newg = n.g + cost(n,n')
    if n' is in Open or Closed, and n'.g < = newg
      skip
    n'.parent = n
    n'.g = newg
    n'.h = GoalDistEstimate( n' )
    n'.f = n'.g + n'.h
    if n' is in Closed
       remove it from Closed
    if n' is not yet in Open
       push n' on Open
  push n onto Closed
return failure
```

This pseudo-code uses a `priorityqueue` data structure (for `Open`) and a `GoalDistEstimate` heuristic function. The latter can have these possibilities:

- Dictionary
- Euclidean Distance: $\sqrt((n'.x-n.x)^2+(n'.y-n.y)^2+(n'.z-n.z)^2)$
- Manhattan Distance: $|n'.x-n.x|+|n'.y-n.y|+|n'.z-n.z|$

Again, as in Dijkstra's Algorithm, 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.

2.5. Tasks:

- Task 1: Modify Graph.cs to accept extra information for each vertex:
  - Case 1 (**heuristic function values**): Value of $h(n)$
  - Case 2:(**3D coordinates of vertices**): Position of the Vertex $Vector3$
    - In the latter case you need to specify the **heuristic function type** (strategy):
      - Euclidean Distance, or
      - Manhattan Distance
- Task 2: Create **TestingAStarAlgorithm.cs** (Graph g3=...) $Take\ Snapshot$
- Task 3 : Implement the rest of AStar Algorithm (the **shortest_path_via_AStar** method) !!!
- Optional Tasks
  - Task 0': Use **Graphviz to draw** the graph drawn in whiteboard (if any). Save as **.png** file

- Task 2': Update the **TestingAStarAlgorithm.cs** to build the above graph (Graph g4=...) $Take\ Snapshot$
- Bonus Task: Pick a simple graph and run the algorithm manually in a piece of paper, as we do in class (any of them actually not just A* — BFS, DFS, Dijkstra). If anything I can give you graph samples. You can strat with very simple ones (3-4 nodes) to get the feeling of the algorithm. Show me your work. I may ask you to go over one or two steps. You can have the pseudocode open.. **NB: This might be very important in your job interviews. If we don't have time in class, I will happily consider this outside of the class time - any time.**

3. Summary

In this lab, **studied**, **implemented** and **tested** the:

- A* Algorithm, with the following heuristic strategies:
  - Euclidean Distance
  - Manhattan Distnce
  - Dictionary Distance

In the next lab, we'll use Unity to:

- Implement a version of AStar based an a given grid, hence no need to explicitely specify edges, as each node is supposed to be a neighbor with (hence have an edge with) any of of the 4 (or eight) adjacent nodes (N,E,S, and W, or, for 8 neighbors, N,NE, E, SE, S, SW, W, NW), which will be available via a `GetNeighbors(node)` method.
- Demonstrate Unity's Pathfinding techniques and classes such as `NavMesh`, `NavMeshAgent` etc. (`NavMesh pathfinding is implemented by a judicious use of AStar algorithm underneath).