Data Structures: Assignment 3

Pathfinding

SP2, 2017

James Baumeister

May 2017

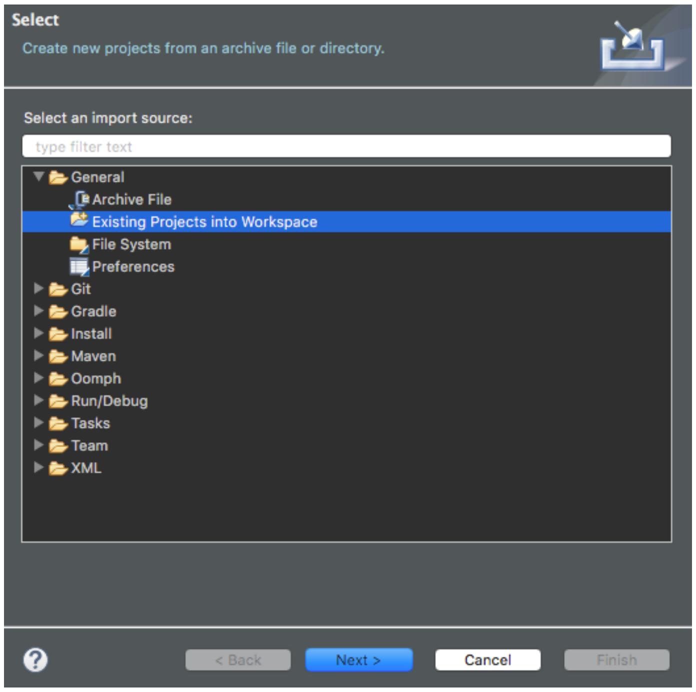
1 Introduction

As a game designer you want to develop your first 2D real-time strategy game. You envision a game where a player is in a procedurally generated terrain map, performing tasks with non-player characters and defending against attacks from enemies. The type of map you want to use will feature several biomes, ranging from deserts to lush rainforests. Each of these biomes will have different characteristics and moving through the different biomes will have different difficulty levels. As this is your first foray into game development, you want to start with a very simple map—a 10×10 square grid where the player can move north, south, east, west, north-east, south-east, north-west, south-west. With these criteria in mind, you decide that a graph is the perfect data structure in which to store all the possible biomes and their connections to each other.

In this assignment we will build an undirected graph structure. A node, or vertex, in this graph will represent a terrain biome with its position in the graph being the centre of a 1x1 square. Each node contains information about the node's position in the map, as well as its terrain features, including the biome, elevation and other locale- and weather-based characteristics. Each node can have up to eight connections, or edges, to other nodes, depending on its position in the map. These edges are what allow travel from one node to another.

This assignment consists of two parts. In part A you will complete a number of helper methods that will be useful when implementing search algorithms in the next part. In part B you will generate all the edges between each of the nodes to form them into the 10×10 grid. You will also implement a number of different search algorithms. Depth- and breadth-first searches can both find a path from one node to another, but do it in different ways and can have very different results. They also do not take into account the weight of the edge or, in other words, the difficulty of travelling over particular biomes. The Dijkstra's and A* search algorithms both take into account the weight and so more accurately provide a path that is both short and least costly, or difficult to travel.

This assignment provides two means by which you can test your code. Running `GraphGUI` will provide a graphical user interface (GUI) that visualises the graph, the terrains, the nodes and the edges. It also animates the player node, showing the path that your search method calculates. You can use this GUI to view the outcome of your algorithm implementations and troubleshoot[1]. There are also unit tests that will give you an idea of the accuracy of your implementations. The tests are not exhaustive and the mark should only be viewed as a guide. Alternative tests will be run by the markers to ensure that your answers are not hard-coded to the tests. Many exception cases are not tested, so you should write your own testing methods. It is suggested that you complete the assignment in the order outlined in the following sections. The later steps rely on the correct implementation of the earlier steps, particularly the `connectNodes` method.


Select

Create new projects from an archive file or directory.

Select an import source:

Figure 1: Importing the project through File → Import

[1]See Appendix A to learn how to control the GUI

1.1 Importing into eclipse

The assignment has been provided as an eclipse project. You just need to import the project into an existing workspace. See Figure 1 for a visual guide. Make sure that your Java JDK has been set, as well as the two jar files that you need for junit to function. This can be found in Project → Properties → Java Build Path → Libraries. The jar files have been provided within the project; there is no need to download any other version and doing so may impact the testing environment.

2 Part A

In this section you will complete some methods that are necessary for building and utilising the graph structure that you will build in section 3.

2.1 Edge

The `Edge` class represents a connection from one node to up to eight others. An `Edge` object has three class variables:

```
private Node node1;
private Node node2;
private double weight;
```

2.1.1 `void calculateWeight()`

The weight of an `Edge` should be calculated using the `calculateWeight()` method upon creation of the `Edge`. The weight is calculated as the Euclidean distance between the two nodes multiplied by the average biome weight between the two nodes. This can be represented mathematically as follows:

$$w(e) = d(p, q) \times ((b1 + b2)/2)$$

where $b1$ is the biome value of the source node, and $b2$ is the biome value of the target node. d is a function that calculates the Euclidean distance between two 2D points, p and q.

2.2 EdgeTest

`EdgeTest` will assign marks as shown in Table 1.

2.3 Vector2

The `Vector2` class represents a 2D point in space and contains an x (horizontal) and a y (vertical) coordinate. For this assignment we are only concerned with finding the distance between two 2D points. A `Vector2` object has two class variables:

Table 1: `EdgeTest` mark allocation

Test	Marks
constructor	5
calculateWeight	5
Total:	**10**

```
public double x;
public double y;
```

2.3.1 `public double distance(Vector2 v2)`

This method should calculate the Euclidean distance between two points. The method should be called on one `Vector2` object, and passed the second `Vector2` object as the parameter. The distance should be returned as a `double`. The algorithm for calculating the Euclidean distance is as follows:

$$d(p,q) = \sqrt{(q1 - p1)^2 + (q2 - p2)^2}$$

2.4 VectorTest

`VectorTest` will assign marks as shown in Table 2.

Table 2: `VectorTest` mark allocation

Test	Marks
distance	5
Total:	**5**

3 Part B

In this section you will implement a number of methods in the `Graph` class. First, you will create edges between a given set of vertices. Next, you will implement some helper methods for navigating the graph. Lastly, you will implement several graph searching algorithms.

3.1 Graph

The graph structure that you must build in this assignment forms a 10×10 grid, with all edges between the nodes being undirected. Due to the way in which our graph is built, node pairs have mirrored edges—node 1 has an edge to node 2, node 2 has an edge to node 1. The `Graph` class has no class variables.

3.1.1 `void connectNodes(Node[] nodes)`

This method connects all nodes in a given array to form a 10×10 grid-shaped graph. *This method must be successfully completed before attempting any other graph searching methods!* The provided GUI can help you visualise how well your implementation is functioning. Before completing `connectNodes`, the GUI should display as shown in Figure 2. Once all of the edges have been correctly created, the GUI will display as shown in Figure 3. Every node in the graph can have up to eight edges, depending on its position. Central nodes will use all eight to connect to all their surrounding neighbours. Think about how many neighbours corner and edge nodes have and how many edges you need to create. In order to develop an algorithm there are some simple constants that you may utilise:

- The top-left corner of the graph has the 2D coordinate (0, 0).
- The bottom-right corner of the graph has the 2D coordinate (9, 9).
- A node's position is the exact centre of a biome square.
- In the provided `Node[]`, for every node(i) such that

  $$i \mod 10 = 0$$

  node(i) is on the left edge.

It is very important to adhere to the *order* of the mappings shown in Table 3 when populating a node's edge list. Note that a node does not need a list containing eight edges if it only requires three, but the order must be maintained—for example, east before south, north before south-east.

3.1.2 `Edge getEdge(Node source, Node destination)`

This methods takes as arguments two `Node` objects. It should search each node's list of `Edge` objects for one that connects the two nodes and return the source node's edge. If there is none found, the method should return `null`.

3.1.3 `double calculateCost(Node[] vertices)`

This method should calculate the total cost of travelling from one node (`Node[0]`) to a target node (`Node[length-1]`). The total value should be returned. If the starting and target nodes are the same, the method should return 0.

Table 3: Edge list index–direction mappings

Edge list index	Direction of connected node
0	East
1	West
2	South
3	North
4	North-east
5	South-east
6	North-west
7	South-west

3.1.4 `Node[] breadthFirstSearch(Node start, Node target)`

The `breadthFirstSearch` method takes as arguments two `Node` objects—a starting node and a target node, respectively. You must implement a breadth-first search algorithm to find the shortest path from start to target and return that path as a `Node` array, ordered from start (index 0) to target (index length−1). This method should *not* take into account edge weights.

3.1.5 `Node[] depthFirstSearch(Node start, Node target)`

The `depthFirstSearch` method takes as arguments two `Node` objects—a starting node and a target node, respectively. Unlike the breadth-first search, depth-first searching will likely not find the shortest path, so you should see drastically different paths being generated. `depthFirstSearch` should return the path as a `Node` array, ordered from start (index 0) to target (index length−1). This method should *not* take into account edge weights.

3.1.6 `Node[] dijkstrasSearch(Node start, Node target)`

The method should use Dijkstra's algorithm to search the graph for the shortest path from start to target while taking into account the cost of travel (i.e. edge weight). Visualising this algorithm should show that sometimes the path may not be the most direct route. Rather, it should be the least costly. Your implementation should be a true implementation of the algorithm[2]. Your code will be inspected to ensure that an alternative algorithm has not been used.

[2]Closely follow the textbook example. Subtle differences in algorithms could impact your performance against the tests

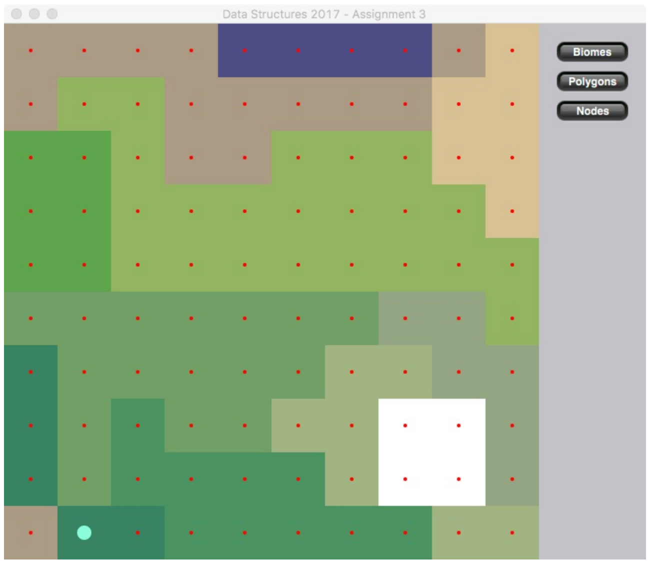


Figure 2: The GUI before completing the `connectNodes` method

`dijkstrasSearch` should return the path as a `Node` array, ordered from start (index 0) to target (index length−1).

3.1.7 `Node[] aStarSearch(Node start, Node target`

This method should use the A* algorithm, similar to Dijkstra's algorithm, to search the graph for the least costly path. Unlike Dijkstra's, which searches in all directions, the A* algorithm uses a heuristic to predict the direction of search. The heuristic you should use should be shortest distance, using the `distance` algorithm you implemented earlier. Your implementation should be a true implementation of the algorithm[3]. Your code will be inspected to ensure that an alternative algorithm has not been used. `aStarSearch` should return a `Node` array containing the path, ordered from start (index 0) to target (index length−1).

3.2 GraphTest

`GraphTest` will assign marks as shown in Table 4.

[3]This is a research task, but this website should be very helpful: `http://www.redblobgames.com/pathfinding/a-star/introduction.html`

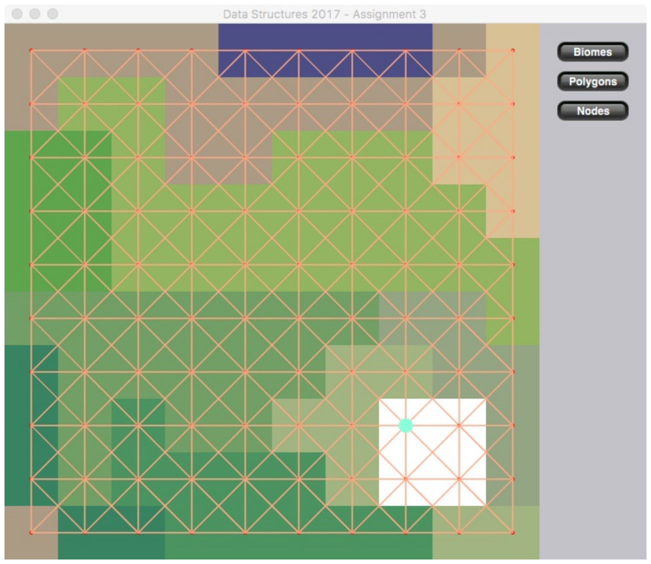


Figure 3: The GUI after completing the `connectNodes` method

A Using the GUI

A GUI has been provided to aid understanding how your graph searching algorithm implementations are functioning. The window contains a graphical representation of the graph on the left, and three buttons on the right (see Figure 3. The buttons labelled 'Biomes' and 'Polygons' are essentially toggles for displaying an outline of the node squares (shown in Figure 4. 'Biomes' is activated by default. The button labelled 'Nodes' controls whether or not the red node circles and pink edge lines are shown—click to toggle between the two. The blue player node will render at the nominated start node and its position will update if a path is provided.

As the GUI operates independently to the testing suite, there are some aspects that you must manually control in order to show the desired information. The `GraphRender` class has the following constants:

```
private final int START_NODE = 0;
private final int TARGET_NODE = 9;
private final int ANIMATION_DELAY = 500;
private final String METHOD = "breadthFirstSearch";
```

You may modify these values. As an example, if you were testing your Dijkstra's algorithm implementation and wanted to match one of the unit tests, you could change `START_NODE` to 8, `TARGET_NODE` to 0 and `METHOD` to `"dijkstrasSearch"`. `ANIMATION_DELAY` represents the delay for the blue player circle to jump along

Table 4: `GraphTest` mark allocation

Test	Marks
connectNodes	10
getEdge	5
calculateCost	5
breadthFirstSearch	20
depthFirstSearch	15
dijkstrasSearch	20
aStarSearch	10
Total:	**85**

nodes in the path, in milliseconds; increase to slow the animation, decrease to quicken.

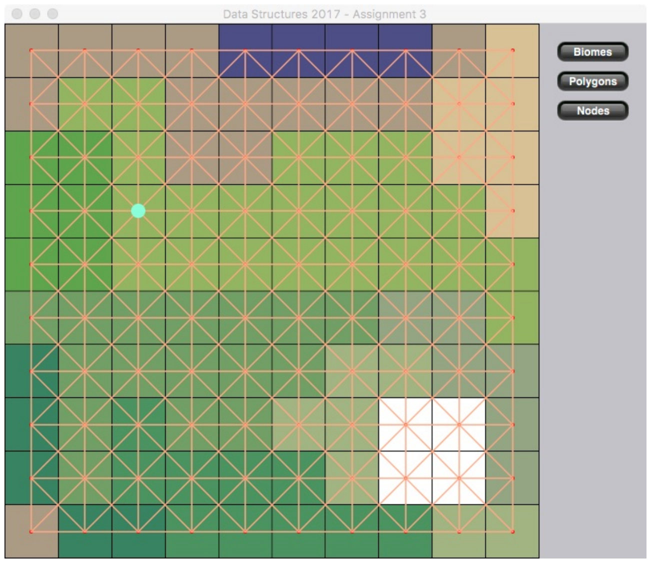


Figure 4: The GUI when the ‘Polygons’ button has been selected