University of South Australia

INFS 2042 Data Structures Advanced

Assignment 1 – Pathfinding

UniSA STEM
The University of South Australia
2024

Originally written by James Baumeister
Modified by Brandon Matthews, Daniel Ablett, and Gun Lee

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 map, performing tasks with non-player characters and defending against attacks from enemies. The type of map you want to use will feature different elevations, visualized as a height map where blue is a low elevation and red is a high elevation. As this is a strategy game, moving between different elevations has the added difficulty of being more costly for characters.

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 in eight directions: 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 elevations 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 position on 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 elevation. 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 between elevations. The Dijkstra algorithm takes into account the weight and so more accurately provides 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. 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 may write your own testing methods, too.

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.

2 Project Setup

The assignment has been provided as an eclipse project in a zip file. You just need to import the project into an existing workspace by selecting Project → Import from the menu. Then choose "General/Existing Projects into Workspace" then press "Next". Select "Select archive file" then press "Browse" to select the project zip file provided for the assignment. Make sure the Project is checked under the Project list then hit "Finish" to import the selected project.

If setting up the project yourself, make sure that your Java JDK has been set, as well as the two jar files (junit-4.12.jar and hamcrest-core-1.3.jar) provided in the zip are also added under 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.

Even when importing the project as is, you will still need to setup the JavaFX library yourself as it differs based on your computer setup. (1) Download JavaFX 21.0.2 (LTS) from here: https://gluonhq.com/products/javafx/. You will need to select correct operating system and architecture according to your computer setup for downloading the JavaFX library SDK. Download and extract the correct version for your system. (2) Next, you need to set it up as a user library named "JavaFX". To do this, create a new User Library under Preferences -> Java -> Build Path -> User Libraries, press "New" button, then enter the name "JavaFX" for the user library. Once the "JavaFX" user library is created on the list, select it, press "Add External JARs" button, then select and add all the jar files under the "lib" folder found inside the JavaFX library folder you downloaded and extracted. Press "Apply and Close" button. (3) Now you have to make sure this user library is added to the project. Right click on the assignment project imported, then select "Properties" from the popup menu, then select "Java Build Path" from the list, "Libraries" tab, select "Modulepath" from the list, and make sure "JavaFX" is listed. If not, press "Add Library" button, then add by selecting the "User Library" then "JavaFX". Once it is all set up properly, your project should not show any error.

You can find further details on setting up JavaFX for various IDEs from here:
https://openjfx.io/openjfx-docs/#IDE-Eclipse

There is one more step left to run the project. (1) First, try to run the project as "Java Application" by right clicking on the project then selecting "Run As/Java Application" then selecting the "GraphGUI" from the list. This will run the project but nothing will show up on the screen. (2) Next right click on the project and select "Run As/Run Configurations", select "Java Application/GraphGUI" from the list, select the "Arguments" tab, then uncheck the "Use the -XstartOnFirstThread argument when launching with SWT" option, and in the VM arguments box, enter the below arguments then press "Apply".

On unix based operating systems including MacOS, use the following arguments:
--module-path /path/to/javafx-sdk-20/lib --add-modules javafx.controls,javafx.fxml

On Windows operating system, use:
--module-path "\path\to\javafx-sdk-20\lib" --add-modules javafx.controls,javafx.fxml

Now when you run project as Java Application, you will see a GUI window appear as shown in Figure 1.

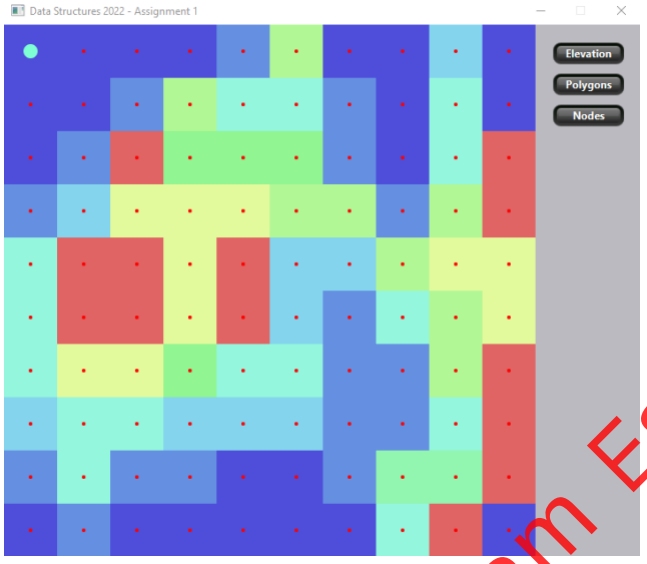

Figure 1: The GUI before completing the connect Nodes method.

GUI is helpful for understanding how your graph searching algorithm implementations are functioning. Not much will happen at this stage, but you will see an animation played as you complete the assignment tasks are completed (such as shown in Figure 2). You are now ready to start working on the assignment tasks described in the following sections.

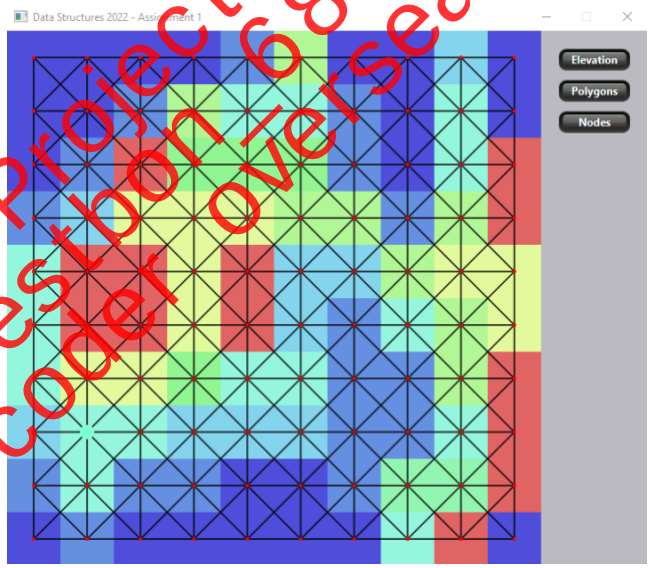

Figure 2: The GUI after completing the connect Nodes method.

3 Part A

In this section you will complete some methods that are necessary for building and utilizing the graph structure that you will build in section 4, **Part B**.

3.1 Position

The Position 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 Position object has two class variables:

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

The Position class has the following method that you need to implement:

```
public double distance(Position v2)
```

This method should calculate the Euclidean distance between two points. The method should be called on one Position object, and passed the second Position object v2 as the parameter. The distance should be returned as a double. The algorithm for calculating the Euclidean distance is as follows:

$$d(p,q) = \sqrt{(qx - px)^2 + (qy - py)^2}$$

3.2 PositionTest

PositionTest will assign marks as shown in Table 2. You can run the test by running the project as JUnit Test, instead of Java Application. As mentioned in the introduction, these are just provided as a reference for you to test your own solution. Actual marking will vary based on the quality and solidity of your code.

Table 2: PositionTest mark allocation

Test	Marks
distance	5
Total:	**5**

3.3 Edge

The Edge class represents a connection from one node to another. An Edge object has three class variables:

```
private Node fromNode; // source node
private Node toNode; // destination node
private double weight; // weight
```

The Edge class has the following method that you need to implement:

```
void calculateWeight()
```

The weight of an Edge should be calculated using this method upon creation of the Edge. The weight is calculated as the Euclidean distance between the two nodes multiplied by the absolute difference between the two nodes' elevations, plus a factor 0.01. This can be represented mathematically as follows:

$$w(e) = d(p1, p2) \times (0.01 + |e1 - e2|)$$

Where $e1$ is the elevation of the *from* node, and $e2$ is the elevation of the *to* node. d is a function that calculates the Euclidean distance between two 2D points, $p1$ and $p2$, that are positions of the two nodes.

3.4 EdgeTest

EdgeTest will assign marks as shown in Table 1.

Table 1: Edge Test mark allocation

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

4 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. As you go through these steps it will be useful the use the GUI to visualise how your algorithms are working. Using the GUI for testing is further explained in section 5.

4.1 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. The graph representation used in this assignment is neither adjacency matrix nor adjacency list, but rather a node having a list of edges that connects to another node. Details are provided below.

4.1.1 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 connect Nodes, the GUI should display as shown in Figure 1. Once all of the edges have been correctly created, the GUI will display as shown

in Figure 2. 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 square.
- In the provided nodes[], for every nodes[i] such that

$$i \mod 10 = 0$$

  nodes[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. This order should be followed in search algorithms.

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

4.1.2 Edge getEdge(Node source, Node destination)

This method takes two Node objects as arguments. It should search the 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.

4.1.3 double calculateTotalWeight(Node[] vertices)

This method should calculate the total cost (i.e. sum of weights) of travelling from the first node (Node[0]) to the last node (Node[length-1]). The total value should be returned. If the starting and target nodes are the same, the method should return 0. If there is no valid edge between nodes, it must return Double.POSITIVE_INFINITY.

4.1.4 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.

4.1.5 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.

4.1.6 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[1]. Your code will be inspected to ensure that an alternative algorithm has not been used.

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

4.2 GraphTest

GraphTest will assign marks as shown in Table 4.

Table 4: GraphTest mark allocation

Test	Marks
connectNodes	10
getEdge	5
calculateTotalWeight	10
breadthFirstSearch	15
depthFirstSearch	15
dijkstrasSearch	20
Total:	**75**

[1] Closely follow the example in Week 5 slides. Subtle differences in algorithms could impact your performance against the tests.

5 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 1). The buttons labelled 'Elevation' and 'Polygons' are essentially toggles for displaying an outline of the node squares (shown in Figure 3). 'Elevation' is activated by default. The button labelled 'Nodes' controls whether or not the red node circles and black edge lines are shown—click to toggle between the two. The light 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 = 99;

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

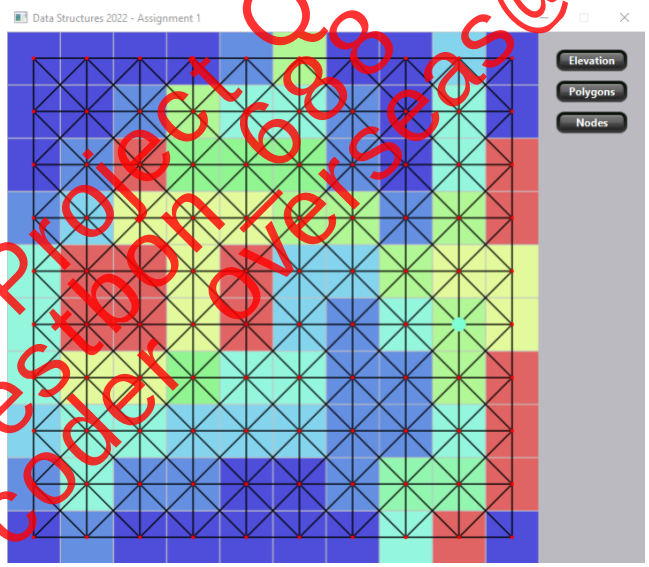

Figure 3: The GUI when the 'Polygons' button has been selected.

6. Submission Details

All assignments must be submitted via the learnonline submission system through the course webpage. Create a single zip file with the name **emailID.zip** (replace **emailID** with your own!).

When unzipped it should contain a functional eclipse project. You may work on the project in another IDE however you must ensure it works as an eclipse project as it will be marked based on eclipse.

Late Submissions and Extensions

Late submissions will be penalised by scaling the marks by 70% unless with pre-approved extension. Application for extension should be lodged through the course website and it requires a document proving acceptable reasons for extension (e.g., medical certificate, or a letter from your work supervisor). Please check the course outline available on the course website for further details on late submission and extensions.

Academic Misconduct

Students must be aware of the academic misconduct guidelines available from the University of South Australia website. Deliberate academic misconduct such as plagiarism is subject to penalties. Information about Academic integrity can be found in Section 9 of the Assessment policies and procedures manual at:
https://i.unisa.edu.au/policies-and-procedures/codes/assessment-policies/

All of the assignments are compared using special tools designed to look for similarities between Java programs. The plagiarism checking programs do not just compare the actual Java code, but instead perform comparisons on the code after it has been compiled. Performing cosmetic changes such as reformatting code, renaming variables, or reordering code may fool a human under casual inspection but will still be detected by the plagiarism checker as being similar.

Any assignments found to be in violation of the university's rules on academic misconduct will become subject of Academic Integrity investigation which will decide the penalty. Furthermore, you may also fail the course and/or receive an official note in your academic transcript.

The best way to avoid being penalised for plagiarism is to not cheat on your assignment. Do not share your code with anyone, do not let anyone else do your assignment for you, and do not leave your computer unattended or share your password with anyone. If you are working with friends it is ok to discuss the assignment and possible ways of solving it, but you should not share your code. Sharing code with others is still considered academic misconduct. The golden rule for working on your assignments is never show another student the material you intend on handing up.

7. Assessment Criteria

Test	Marks
Code Quality, Commenting and Compilation	*10*
Neatly formatted code with sufficient comments explaining the operation of the code.	5
Successful compilation with no fatal errors and no runtime errors during testing.	5
Position	*5*
distance	5
Edge	*10*
constructor	5
calculateWeight	5
Graph	*75*
connectNodes	10
getEdge	5
calculateTotalWeight	10
breadthFirstSearch	15
depthFirstSearch	15
dijkstrasSearch	20
Total:	**100**