UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science
Faculty of Engineering, Built Environment & IT
University of Pretoria

COS212 - Data structures and algorithms

Practical 10 Specifications: Directed Graphs

Release date: 19-05-2025 at 06:00
Due date: 23-05-2025 at 23:59
Total marks: 157

Contents

1 General Instructions

- ***Read the entire assignment thoroughly before you start coding.***
- This assignment should be completed individually; no group effort is allowed.
- **To prevent plagiarism, every submission will be inspected with the help of dedicated software.**
- Be ready to upload your assignment well before the deadline, as no extension will be granted.
- You may not import any of the built-in Java data structures. Doing so will result in a zero mark. You may only use native 1-dimensional arrays where applicable.
- If your code does not compile, you will be awarded a mark of zero. Only the output of your program will be considered for marks, but your code may be inspected for the presence or absence of certain prescribed features.
- If your code experiences a runtime error, you will be awarded a zero mark. Runtime errors are considered unsafe programming.
- Read the entire specification before you start coding.
- **Ensure your code compiles with Java 8**
- The usage of ChatGPT and other AI-Related software is strictly forbidden and will be considered as plagiarism.
- Note that plagiarism is considered a very serious offence. Plagiarism will not be tolerated, and disciplinary action will be taken against offending students. Please refer to the University of Pretoria's plagiarism page at `https://portal.cs.up.ac.za/files/departmental-guide/`.
- Please note that there is a late deadline which is 1 hour after the initial deadline, but you will lose 20% of your mark.

2 Overview

Directed graphs also known as digraphs are regular graphs that have edges with a direction associated with it indicating a one way connection. In this practical, you will be implementing a directed graph to model two graph theory concepts, strongly connected components(maximal sets of connected nodes in a directed graph.) and minimum spanning trees(the collection of edges required to connect all vertices in an undirected graph, with the minimum total edge weight).

3 Tasks

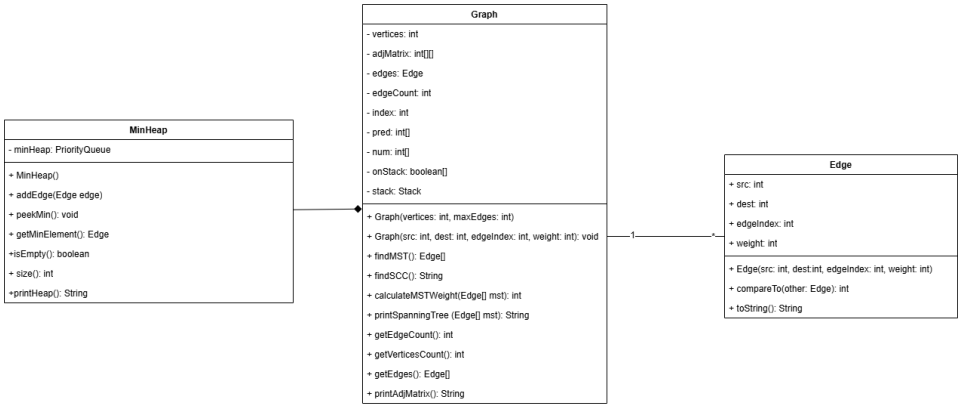

Figure 1: Graph structure

4 Classes

4.1 Graph

- Represents a directed graph using an adjacency matrix representation
- Supports weighted edges
- Implements Kruskal's algorithm for finding Minimum Spanning Trees
- Implements strongDFS() algorithm for finding Strongly Connected Components

4.1.1 Members

- `vertices`:
  - Number of vertices in the graph
  - Used to size data structures and bound iterations
- `adjMatrix`:
  - 2D integer array representing the adjacency matrix
  - Stores edge weights between vertices
- `edges`:
  - Array of Edge objects
  - Stores all edges added to the graph

- `edgeCount`:
  - Current number of edges in the graph
  - Used to track array position for new edges
- `index, num, pred, onStack`:
  - Data structures used by strongDFS algorithm
- `stack`:
  - Stack data structure used in strongDFS algorithm
  - Maintains vertices being processed in current component
- `sccCount`:
  - Count of strongly connected components found
  - Incremented when a new component is identified

4.1.2 Functions

- `addEdge`:
  - Adds a new edge to the graph
  - Updates adjacency matrix with weight information
  - throw new IllegalStateException("Max edges reached"); if edges added exceed maximum edges allowed
- `findMST`:
  - Implements Kruskal's algorithm
  - Finds minimum spanning tree of the graph
  - Uses min-heap and union-find data structures
- `find`:
  - Helper function for Union-Find data structure
- `union`:
  - Helper function for Union-Find data structure

  *** Union-Find (or Disjoint Set Union - DSU) is a data structure that efficiently manages a collection of disjoint (non-overlapping) sets. It supports two key operations: Find (to determine which set an element belongs to) and Union (to merge two sets). ***
- `findSCC`:
  - Implements strongDFS algorithm
  - Identifies all strongly connected components
  - Returns total number of compononets found and a string rep of components
  - SCC 1: A B
  - SCC 2: X Y Z
  - SCC 3: D C B

  - note there is newline character at the end of each SCC and the spaces are important

- `printAdjMatrix`:
  - Displays the adjacency matrix to console
  - Used for debugging and visualization
  - Do not alter this function
- `getEdges`:
  - Returns the array of edges in the graph
- `getVerticesCount`:
  - Returns the number of vertices in the graph
- `getEdgeCount`:
  - Returns the current number of edges in the graph
- `calculateMSTWeight`:
  - Calculates total weight of a spanning tree
  - Sums weights of all edges in the tree
- `printSpanningTree`:
  - Creates a string representation of a spanning tree
  - Formats edge information for display

4.2 Edge

- Represents a weighted edge in a graph
- Stores source and destination vertices
- Maintains weight/cost information
- Implements Comparable interface for use in sorting algorithms
- negative edge weights are allowed

4.2.1 Members

- `src`:
  - Source vertex of the edge
  - Starting point of the directed edge
- `dest`:
  - Destination vertex of the edge
  - Ending point of the directed edge
- `edgeIndex`:
  - Unique identifier for the edge
  - Used to distinguish between edges
- `weight`:
  - Weight/cost of the edge

4.2.2 Functions

- `compareTo`:
  - Compares edges based on weight
  - do not alter this function
- `toString`:
  - Creates a string representation of the edge
  - do not alter this function

4.3 MinHeap

This class has been provided for you, do not alter it.

- Implements a priority queue-based min-heap specialized for Edge objects
- Provides methods for efficient edge extraction by minimum weight
- Supports Kruskal's algorithm by efficiently sorting edges

4.3.1 Members

- `minHeap`:
  - PriorityQueue instance to handle the min-heap operations
  - Automatically maintains edges sorted by weight
  - Uses Edge's compareTo method for ordering

4.3.2 Functions

- `addEdge`:
  - Adds an edge to the min-heap
  - Edge is placed according to its weight
- `peekMin`:
  - Returns the minimum edge without removing it
- `getMinElement`:
  - Removes and returns the minimum edge
  - Used in Kruskal's algorithm to process edges in weight order
- `isEmpty`:
  - Checks if the heap is empty
- `size`:
  - Returns the number of elements in the heap
- `printHeap`:
  - Displays the contents of the heap
  - Used for debugging and visualization

5 Testing

As testing is a vital skill that all software developers need to know and be able to perform daily, approximately 10% of the assignment marks will be assigned to your testing skills. To do this, you will need to submit a testing main (inside the Main.java file) that will be used to test an Instructor-provided solution. You may add any helper functions to the Main.java file to aid your testing. In order to determine the coverage of your testing the jacoco tool. The following set of commands will be used to run jacoco:

```
javac *.java
rm -Rf cov
mkdir ./cov
java -javaagent:jacocoagent.jar=excludes=org.jacoco.*,destfile=./cov/output.exec
    -cp ./ Main
mv *.class ./cov
java -jar ./jacococli.jar report ./cov/output.exec --classfiles ./cov --html
    ./cov/report
```

This will generate output which we will use to determine your testing coverage. The following coverage ratio will be used:

$$\frac{\text{number of lines executed}}{\text{number of source code lines}}$$

and we will scale this ratio according to the size of the class.
The mark you will receive for the testing coverage task is determined using table 1:

Coverage ratio range	% of testing mark
0%-5%	0%
5%-20%	20%
20%-40%	40%
40%-60%	60%
60%-80%	80%
80%-100%	100%

Table 1: Mark assignment for testing

Note the top boundary for the Coverage ratio range is not inclusive except for 100%. Also, note that only the function stipulated in this specification will be considered to determine your mark. Remember that your main will be testing the instructor-provided code and as such it can only be assumed that the functions outlined in this specification are defined and implemented.

6 Upload checklist

The following files should be in the root of your archive

- Main.java
- Graph.java
- Edge.java
- MinHeap.java
- Any textfiles needed by your Main

7 Allowed libraries

- java.util.Stack
- java.util.PriorityQueue;

8 Submission

You need to submit your source files on the FitchFork website (https://ff.cs.up.ac.za/). All methods need to be implemented (or at least stubbed) before submission. Place the above-mentioned files in a zip named uXXXXXXXX.zip where XXXXXXXX is your student number. Your code must be able to be compiled with the Java 8 standard.

For this practical, you will have 5 upload opportunities and your best mark will be your final mark. Upload your archive to the appropriate slot on the FitchFork website well before the deadline. **No late submissions will be accepted!**