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 9 Specifications - Basic Graphs

Release date: 20-05-2024 at 06:00
Due date: 24-05-2024 at 23:59
Total marks: 200

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 you require additional data structures, you will have to implement them yourself.
- 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.

2 Plagiarism

The Department of Computer Science considers plagiarism a serious offence. Disciplinary action will be taken against students who commit plagiarism. Plagiarism includes copying someone else's work without consent, copying a friend's work (even with consent), and copying material (such as text or program code) from the Internet. Copying will not be tolerated in this course. For a formal definition of plagiarism, the student is referred to http://www.library.up.ac.za/plagiarism/index.htm (from the main page of the University of Pretoria site, follow the Library quick link, and then choose the Plagiarism option under the Services menu). **If you have any form of question regarding this, please ask one of the lecturers to avoid any misunderstanding.** Also note that the OOP principle of code reuse does not mean that you should copy and adapt code to suit your solution.

3 Outcomes

On completion of this practical, you will have gained experience with the following:

- Different implementations of graphs:
  - Using adjacency matrices
  - Using adjacency lists
- Graph operations:
  - Adding nodes and edges
  - Removing nodes and edges
  - Breadth-first and depth-first traversals
  - Unweighted shortest paths

4 Introduction

4.1 Graphs

Graphs are fundamental structures in computer science, representing networks of nodes and the connections between them. They are widely used in various domains such as social networks, transportation systems, and communication networks. This practical assignment will focus on implementing graphs using two primary representations: Adjacency List and Adjacency Matrix.

4.1.1 Adjacency List Implementation

An adjacency list is a collection of lists or arrays used to represent a graph. Each node in the graph has a list associated with it, containing all the nodes to which it is directly connected. This representation is particularly efficient for sparse graphs where the number of edges is much less than the square of the number of nodes. The dynamic structure also allows for easy addition and removal of nodes and edges.

In Java, an adjacency list can be implemented using an array or list of lists, where each list represents the neighbouring nodes of a given node.

4.1.2 Adjacency Matrix Implementation

An adjacency matrix is a 2D array used to represent a graph. The rows and columns represent the nodes, and the cell at the intersection of a row and a column indicates whether an edge exists between the corresponding nodes. This representation is more suitable for dense graphs where the number of edges is close to the square of the number of nodes. It allows for O(1) time complexity when checking the existence of an edge and is easy to implement and understand from a programmer's perspective.

In Java, an adjacency matrix can be implemented using a 2D array, where the element at position (i, j) is true if there is an edge from node i to node j, and false otherwise.

4.2 Bridge Design Pattern

In software engineering, the Bridge Design Pattern is used to decouple an abstraction from its implementation so that the two can vary independently. This pattern involves an interface which acts as a bridge, making the concrete classes independent from interface implementers. By doing so, you can

easily switch between different implementations without modifying the client code. This flexibility is particularly useful in scenarios where multiple implementations of a graph are required, allowing you to optimize for performance and memory usage based on specific needs. For more information on the bridge design pattern see `https://www.cs.up.ac.za/cs/lmarshall/TDP/Notes/_Chapter24_Bridge.pdf` by Dr Linda Marshall and Vreda Pieterse.

5 Tasks

You are tasked with using the bridge design pattern to implement two graph abstractions (`UndirectedGraph` and `DirectedGraph`) and two graph implementations (`AdjacencyList` and `AdjacencyMatrix`). You have been given skeletons for these classes as well as the `Graph` abstract class and the `GraphImplementation` interface. You have also been given a completed `LinkedList` class to aid in your implementation. Your implementation must adhere to the given class diagram (Figure 1).

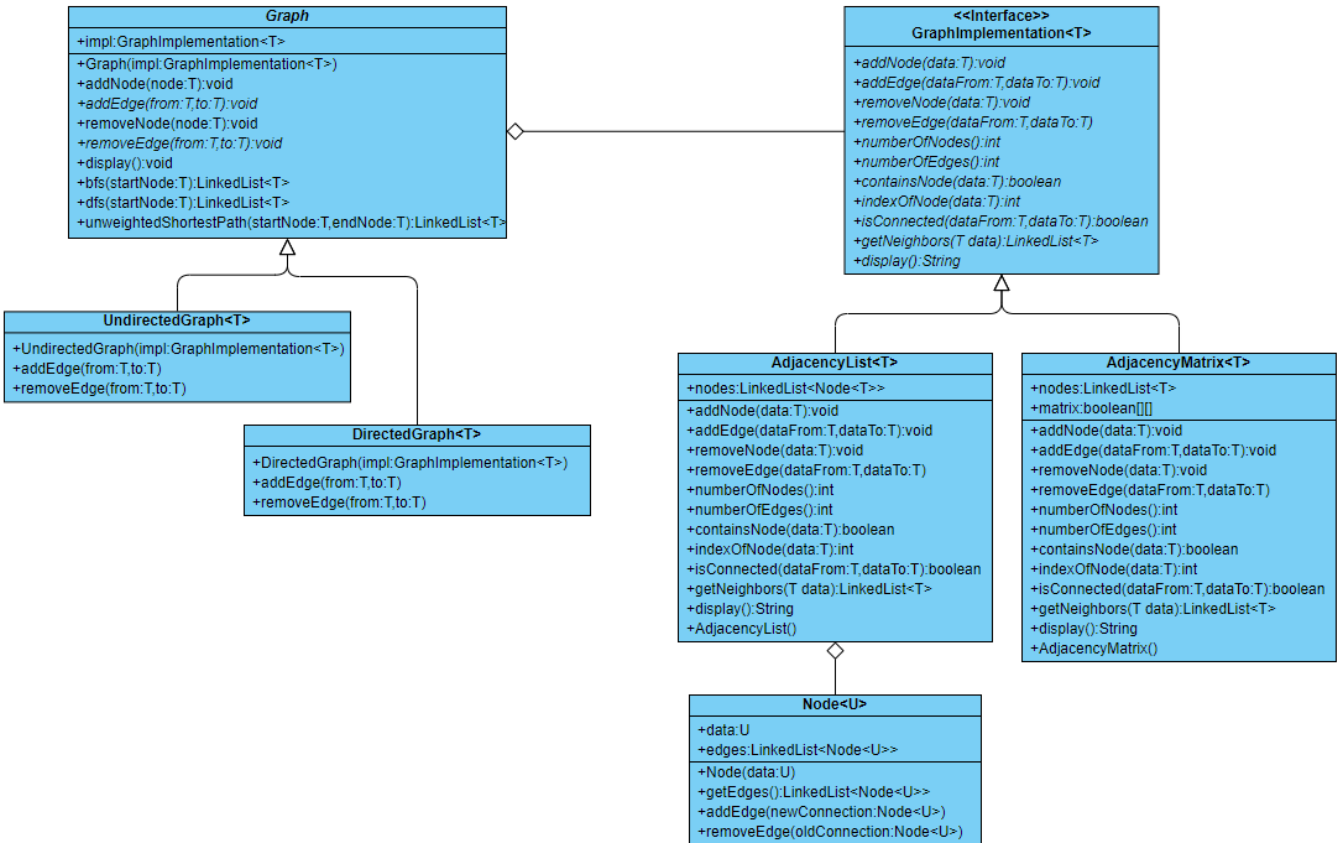

Figure 1: Graph and GraphImplementation hierarchies UML class diagram

5.1 Graph Implementations

5.1.1 GraphImplementation Interface

This interface has been provided for you. It outlines the methods that any graph implementation must provide. These methods are:

- addNode(T data):void
  - This method adds a node with the given data to the graph.
  - If a node with the given data is already present in the graph then nothing should happen (i.e. do not insert duplicates).
  - The implementation of this method is left to the class which implements the interface.

- addEdge(T dataFrom, T dataTo):void
  - This method adds an edge between the two nodes with the given data.
  - Both nodes must be present in the graph for an edge to be made between them. If a node is not in the graph then nothing should happen.
  - If an edge already exists between the two nodes then nothing should happen.
  - The implementation is left to the class which implements this interface.
- removeNode(T data):void
  - This method removes the node with the given data.
  - If a node with the given data is not present then nothing should happen. If a node is removed then the relevant edges from and to that node should also be removed.
  - The implementation is left to the class which implements this interface.
- removeEdge(T dataFrom, T dataTo):void
  - This method removes an edge between the two nodes with the given data.
  - If one or both of the nodes are not in the graph then nothing should happen.
  - If an edge does not exists between the two nodes then nothing should happen.
  - The implementation is left to the class which implements this interface.
- numberOfNodes():int
  - This method returns the number of nodes in the graph.
  - The implementation is left to the class which implements this interface.
- numberOfEdges():int
  - This method returns the number of edges in the graph.
  - For the purposes of this practical, a single edge in an undirected graph will count as two edges. This allows for cleaner parallels between undirected and directed graphs on the graph implementation side.
  - The implementation is left to the class which implements this interface.
- containsNode(T data):boolean
  - This method returns true if a node with the given data is in the graph and false otherwise.
  - The implementation is left to the class which implements this interface.
- indexOfNode(T data):int
  - This method returns the index of the node with the given data in the `nodes` list.
  - If a node with the given data is not present in the graph then this method should return -1.
  - The implementation is left to the class which implements this interface.
  - Hint: The given LinkedList will be useful when implementing this method
- nodeAtIndex(int index):T
  - This method returns the data of the node at a given index.
  - The implementation is left to the class which implements this interface.
  - Hint: The given LinkedList will be useful when implementing this method

- isConnected(T dataFrom, T dataTo): boolean
  - This method returns true if the nodes with the given data are connected by an edge.
  - This method returns false if the nodes with the given data are not connected or if one or both are not in the graph.
  - The implementation is left to the class which implements this interface.
- getNeighbors(T data):LinkedList<T>
  - This method returns a linked list containing the data of all the nodes connected to the node with the given data.
  - **NB:** The order of neighbours will differ between the implementations. This will impact the results of the traversals and can impact the result when finding the unweighted shortest path. In the implementation sections, read the instructions of the `getNeighbors()` method carefully.
  - The implementation is left to the class which implements this interface.
- display():String
  - Returns a display of the inner representation of the graph
  - The implementation is left to the class which implements this interface.
  - This method has been implemented for you to aid in debugging. It will not be tested or used in testing.

5.1.2 Adjacency List

Attributes

- nodes: LinkedList<Node<T> >
  - A linked list that stores all the nodes in the graph
  - Each node has a linked list of nodes representing its edges.
  - The `Node` class has been provided for you within the `AdjacencyList` class

Methods

- AdjacencyList():void
  - The constructor for an adjacency list graph implementation.
  - This should initialise the `nodes` variable
- addNode(T data):void
  - The method complies with the interface definition.
  - This method should create a new node with the given data and add it to the `nodes` list.
- addEdge(T dataFrom, T dataTo):void
  - The method complies with the interface definition.
  - To represent an edge the node with `dataTo` should be added to the list contained in the node with `dataFrom`.
- removeNode(T data):void

  - The method complies with the interface definition.
  - The node with the given data should be removed from the `nodes` list and the other nodes' edge lists.

- removeEdge(T dataFrom, T dataTo):void
  - The method complies with the interface definition.
  - The node with the given `dataTo` should be removed from the edges list of the node with the given `dataFrom`.

- numberOfNodes():int
  - The method complies with the interface definition.
  - This method returns the number of nodes in the graph.

- numberOfEdges():int
  - The method complies with the interface definition.
  - This method returns the number of edges in the graph.
  - For the purposes of this practical, a single edge in an undirected graph will count as two edges. This allows for cleaner parallels between undirected and directed graphs on the graph implementation side.

- containsNode(T data):boolean
  - The method complies with the interface definition.
  - This method returns true if a node with the given data is in the graph and false otherwise.

- indexOfNode(T data):int
  - The method complies with the interface definition.
  - This method returns the index of the node with the given data in the `nodes` list.
  - If a node with the given data is not present in the graph then this method should return -1.
  - Hint: The given LinkedList will be useful when implementing this method

- nodeAtIndex(int index):T
  - The method complies with the interface definition.
  - This method returns the data of the node at a given index.
  - Hint: The given LinkedList will be useful when implementing this method

- isConnected(T dataFrom, T dataTo): boolean
  - The method complies with the interface definition.
  - This method returns true if the nodes with the given data are connected by an edge.
  - This method returns false if the nodes with the given data are not connected or if one or both are not in the graph.

- getNeighbors(T data):LinkedList<T>
  - The method complies with the interface definition.
  - This method returns a linked list containing the data of all the nodes connected to the node with the given data.

- **NB:** The order of neighbours will differ between the implementations. This will impact the results of the traversals and can impact the result when finding the unweighted shortest path.
- **NB:** This implementation should return neighbours in the order the edges were added to the graph. For example, if nodes `A`, `B`, `C` and `D` are added to a graph and then edges `AC`, `AD` and `AB` are added to the graph, the neighbours of `A` are `C`, `D` and `B` in that order. This will impact the traversals and may affect the shortest paths.

- display():String
  - The method complies with the interface definition.
  - Returns a display of the inner representation of the adjacency list
  - This method has been implemented for you to aid in debugging. It will not be tested or used in testing.

5.1.3 Adjacency Matrix

Attributes

- nodes: LinkedList<T>
  - A linked list that stores all the nodes in the graph
  - Since nodes don't need to store their neighbours, nodes are the same as their data.
  - There is no `Node` class. Nodes and data are the same in this implementation.
- matrix: boolean[][]
  - A matrix of boolean that stores the connections.
  - If the value at position i,j (`matrix[i][j]`) is true then the node at position i of the `nodes` list is connected to the node at position j of the `nodes` list.
  - This matrix will need to grow as nodes are added to the graph.
  - If there are n nodes then this matrix should be $n \times n$

Methods

- AdjacencyMatrix():void
  - The constructor for an adjacency matrix graph implementation.
  - This should initialise the `nodes` variable and the matrix variable.
- addNode(T data):void
  - The method complies with the interface definition.
  - This method should create a new node with the given data and add it to the `nodes` list.
  - This method should resize the matrix appropriately.
- addEdge(T dataFrom, T dataTo):void
  - The method complies with the interface definition.
  - This method should set the relevant index of the adjacency matrix to true.
- removeNode(T data):void
  - The method complies with the interface definition.
  - The node with the given data should be removed from the `nodes` list and the other nodes' edge lists.
- removeEdge(T dataFrom, T dataTo):void
  - The method complies with the interface definition.
  - This method should set the relevant index of the adjacency matrix to false.
- numberOfNodes():int
  - The method complies with the interface definition.
  - This method returns the number of nodes in the graph.
- numberOfEdges():int
  - The method complies with the interface definition.

  - This method returns the number of edges in the graph.
  - For the purposes of this practical, a single edge in an undirected graph will count as two edges. This allows for cleaner parallels between undirected and directed graphs on the graph implementation side.

- containsNode(T data):boolean
  - The method complies with the interface definition.
  - This method returns true if a node with the given data is in the graph and false otherwise.

- indexOfNode(T data):int
  - The method complies with the interface definition.
  - This method returns the index of the node with the given data in the `nodes` list.
  - If a node with the given data is not present in the graph then this method should return -1.
  - Hint: The given LinkedList will be useful when implementing this method

- nodeAtIndex(int index):T
  - The method complies with the interface definition.
  - This method returns the data of the node at a given index.
  - Hint: The given LinkedList will be useful when implementing this method

- isConnected(T dataFrom, T dataTo): boolean
  - The method complies with the interface definition.
  - This method returns true if the nodes with the given data are connected by an edge.
  - This method returns false if the nodes with the given data are not connected or if one or both are not in the graph.

- getNeighbors(T data):LinkedList<T>
  - The method complies with the interface definition.
  - This method returns a linked list containing the data of all the nodes connected to the node with the given data.
  - **NB:** The order of neighbours will differ between the implementations. This will impact the results of the traversals and can impact the result when finding the unweighted shortest path.
  - **NB:** This implementation should return neighbours in the order the nodes were added to the graph. For example, if nodes `A`, `B`, `C` and `D` are added to a graph and then edges `AC`, `AD` and `AB` are added to the graph, the neighbours of `A` are `B`, `C` and `D` in that order. This will impact the traversals and may affect the shortest paths.

- display():String
  - The method complies with the interface definition.
  - Returns a display of the inner representation of the adjacency list
  - This method has been implemented for you to aid in debugging. It will not be tested or used in testing.

5.2 Graph Abstractions

5.2.1 Graph

This is the base graph class which will have most of the implementation to avoid repeated code. The implementations in this class may only work with the methods which the `GraphImplementation` interface exposes for the bridge pattern to work. **Attributes**

- impl: GraphImplementation<T>
  - An instance of GraphImplementation that represents the underlying graph implementation (either Adjacency List or Adjacency Matrix).
  - This allows for the graph representation to be swapped out as needed using the Bridge Design Pattern.

Methods

- Graph(GraphImplementation<T> impl)
  - The constructor for the Graph class.
  - This initializes the `impl` variable with the provided implementation.
- addNode(T node): void
  - This method adds a new node to the graph.
  - It delegates the addition of the node to the underlying implementation.
- addEdge(T from, T to): void
  - An abstract method to add an edge between two nodes.
  - This method must be implemented by subclasses.
- removeNode(T node): void
  - This method removes a node from the graph.
  - It delegates the removal of the node to the underlying implementation.
- removeEdge(T from, T to): void
  - An abstract method to remove an edge between two nodes.
  - This method must be implemented by subclasses.
- display(): void
  - Displays the current state of the graph.
  - Delegates the display functionality to the underlying implementation.
- bfs(T startNode): LinkedList<T>
  - Performs a Breadth-First Search (BFS) starting from the given node.
  - Returns a list of data items in the order they were visited.
  - Hint: The given `LinkedList` class and the `getNeigbors()` method in the implementation will be helpful here.
- dfs(T startNode): LinkedList<T>

  - Performs a Depth-First Search (DFS) starting from the given node.
  - Returns a list of data items in the order they were visited.
  - Hint: The `getNeigbors()` method in the implementation will be helpful here.

- unweightedShortestPath(T startNode, T endNode): LinkedList<T>
  - Finds the shortest path between two nodes in an unweighted graph.
  - Returns a list of data items representing the path from the start node to the end node.
  - Hint: The given `LinkedList` class as well as the `getNeigbors()`, `indexOfNode()` and `nodeAtIndex()` methods in the implementation will be helpful here.

5.2.2 UndirectedGraph

This is the implementation for an undirected graph. It implements the abstract method in the `Graph` class. The methods implemented here are:

- UndirectedGraph(GraphImplementation<T> impl)
  - A constructor which passes the `impl` parameter to the parent class
- addEdge(T from, T to): void
  - A method to add an edge between two nodes.
  - Delegates the functionality to the underlying implementation.
  - Hint: In an undirected graph, if an edge is added from A to B then there is also an edge from B to A.
- removeEdge(T from, T to): void
  - A method to remove an edge between two nodes.
  - Delegates the functionality to the underlying implementation.
  - Hint: In an undirected graph, if an edge is removed between A and B then the edge between B and A should also be removed.

5.2.3 DirectedGraph

This is the implementation for a directed graph. It implements the abstract method in the `Graph` class. The methods implemented here are:

- DirectedGraph(GraphImplementation<T> impl)
  - A constructor which passes the `impl` parameter to the parent class
- addEdge(T from, T to): void
  - A method to add an edge between two nodes.
  - Delegates the functionality to the underlying implementation.
  - Hint: In a directed graph, if an edge is added from A to B then there is not necessarily an edge from B to A.
- removeEdge(T from, T to): void
  - A method to remove an edge between two nodes.
  - Delegates the functionality to the underlying implementation.
  - Hint: In a directed graph, if the edge between A and B is removed it should not impact the edge between B and A.

6 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.cpp 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.

7 Upload checklist

The following files should be in the root of your archive

- AdjacencyList.java
- AdjacencyMatrix.java
- DirectedGraph.java
- Graph.java
- GraphImplementation.java (will be overridden)
- LinkedList.java (will be overridden)

- Main.java
- UndirectedGraph.java
- Any other .java files that your solution requires
  - If you want to use your own data structures then they may not be called `LinkedList` since the "LinkedList.java" file will be overridden
- Any .txt files you use for testing

8 Allowed libraries

- None, sorry :(

9 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!**