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

Assignment 3 Specifications: Graphs

Release date: 12-05-2025 at 06:00
Due date: 29-05-2025 at 23:59
Total marks: 560

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

Graphs are non-linear data structures that represent relationships between objects, consisting of nodes (or vertices) and edges.Each edge connects a pair of vertices. Edges can be directional, making the graph a directed graph, or they can be undirected, i.e. an undirected graph. Edges can contain values which may represent various metrics such as weight, distance, cost, etc. A graph that has edges that have weighted edges is called a weighted graph.

2.1 Metagraphs

A Metagraph is a specialized graph that goes beyond pairwise node connections. In a metagraph, edges can connect any number of nodes, not just pairs. The relationships between nodes are in themselves graphs. This allows for hierarchical and complex structures, going beyond traditional graphs which typically only represent relationships between individual entities. This extension allows us to form a graph of graphs structure.A visualisation is provided below to aid in your understanding of this concept. In this practical MetaGraphs contain individual regular subgraphs that are interconnected by specialized edges called MetaEdges.

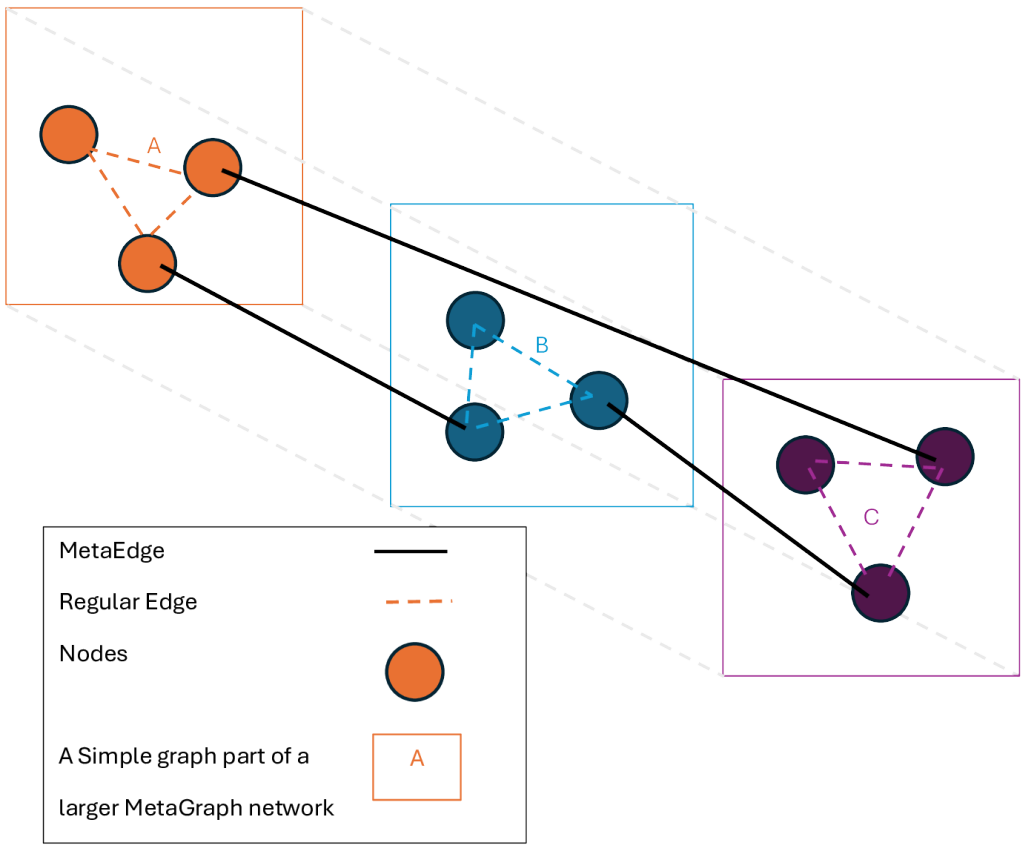

Figure 1: Metagraph structure

3 Tasks

In this practical, we will work with a **MetaGraph** structure that models interconnected cities (**MetaCities**) and their utility networks. Refer to the diagram below,

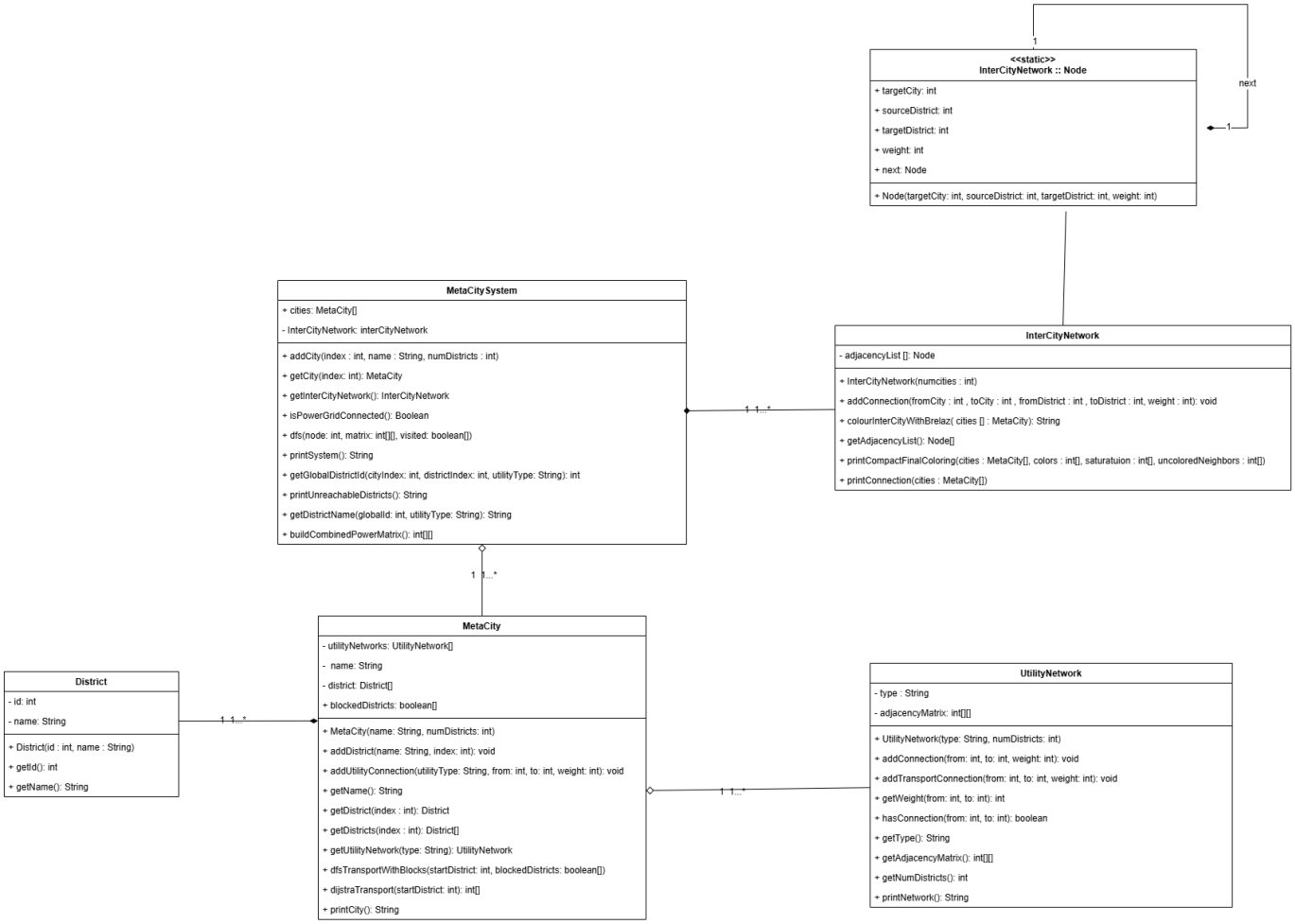

Figure 2: MetaCity Layout

MetaCity Structure

Each `MetaCity` consists of:

- **Districts**: Represented as nodes (e.g., `"Gotham:Downtown"`)
- **Utility Networks**: Three network types connecting districts:
  - `POWER` (undirected)
  - `WATER` (undirected)
  - `TRANSPORTATION` (directed)

Inter-City Connections

- `InterCityNetwork` edges link districts across different MetaCities
- Stored as weighted edges (e.g., power line capacity)

Graph Operations

We will implement and analyze:

Operation	Graph Theory Concept
Combined adjacency matrix construction	Graph representation
Combined adjacency list construction	Graph representation
Block-optimized DFS traversal	Graph traversal
Power grid connectivity check	Connected components
Unreachable districts identification	Connected components
Inter-city connection handling	Multigraph edges
Utility network differentiation	Edge type/weight classification
District ID global mapping	Vertex labeling/numbering
Transportation network pathfinding	Shortest path (Dijkstra's)
System-wide connectivity verification	Strong connectivity
System-wide color associations	Graph Coloring(Brelaz)
MetaEdge connections	Hypergraph edges

Table 1: Graph theory concepts implemented in MetaCitySystem

4 Classes

4.1 District

This class represents a district within a MetaCity, containing basic identification information.These are the individual metacity nodes.

4.1.1 Members

id

- Integer representing the unique identifier for the district
- To keep better track of a district it may correspond to its index in the city's district array

name

- String containing the human-readable name of the district
- Examples: "Downtown", "Industrial Zone", "Residential Sector"

4.1.2 Functions

District(int id, String name)

- Constructor that initializes a district with given ID and name
- Parameters must satisfy: $id \geq 0$ and name not null/empty
- otherwise throw new IllegalArgumentException("ID must be greater or equal than 0.") and throw new IllegalArgumentException("Name cannot be null or empty.")
- Take care to use these exact Exceptions.

getId()

- Returns the district's unique identifier (integer)

getName()

- Returns the district's name (String)

4.2 MetaCitySystem

The top-level class managing multiple MetaCity instances and their intercity connections via an InterCityNetwork.

4.2.1 Members

cities

- Array of MetaCity objects representing all cities in the system.

interCityNetwork

- InterCityNetwork object handling connections between cities (between different districts across cities).

4.2.2 Functions

MetaCitySystem(int numCities)

- Initializes the system with space for numCities.
- Creates an empty InterCityNetwork of the same size.

addCity(int index, String name, int numDistricts)

- Adds a new MetaCity at index with a given name and district count.
- Example: addCity(0, "Gotham", 5) creates Gotham with 5 districts.
- validation checks for valid index, non null city name, number of districts > 0
- otherwise throw new IllegalArgumentException("Index out of bounds for cities array.") and throw new IllegalArgumentException("City name cannot be null or empty.") and throw new IllegalArgumentException("Number of districts must be greater than zero.");

getCity(int index)

- Returns the MetaCity at the specified index.

getInterCityNetwork()

- Provides access to the intercity connection.

buildCombinedPowerMatrix()

- Generates a global adjacency matrix merging all cities' power networks.
- Used for system-wide connectivity checks (e.g., `isPowerGridConnected()`)

4.2.3 Matrix Structure

The combined matrix represents the entire power grid as a single graph where:

- **Nodes** are all districts from all cities
- **Edges** represent power connections (both intra-city and inter-city)
- **Weights** represent connection capacities or distances

4.2.4 Matrix Construction

The matrix is built in four logical steps:

1. **Calculate Total Districts**:
2. **Initialize Global Matrix**:
3. **Fill Intra-City Blocks**:
4. **Add Inter-City Connections**:

4.2.5 Visual Representation

For two cities with 2 districts each:

$$\begin{bmatrix} \text{City1-D1} & \text{City1-D2} & \text{City2-D1} & \text{City2-D2} \\ \hline 0 & w_{1,2} & w_{1,3} & 0 \\ w_{2,1} & 0 & 0 & w_{2,4} \\ w_{3,1} & 0 & 0 & w_{3,4} \\ 0 & w_{4,2} & w_{4,3} & 0 \end{bmatrix}$$

Where:

- $w_{1,2}$ = Intra-city connection (City1)
- $w_{1,3}$ = Inter-city connection (City1-D1 $\leftrightarrow$ City2-D1)
- 0 = No direct connection

4.2.6 Key Properties

- **Symmetric**: All power connections are bidirectional

getGlobalDistrictId(int cityIndex, int districtIndex, String utilityType)

4.2.7 Global District ID System

4.2.8 Concept

The `getGlobalDistrictId()` method creates a unified numbering system for all districts across all cities in the system. Each district receives a unique integer ID based on:

- Its position within its city
- The number of districts in preceding cities
- The utility network type (though IDs are consistent across types)

4.2.9 ID Assignment Formula

The global ID is calculated as:

$$\text{Global ID} = \left(\sum_{i=0}^{\text{cityIndex}-1} \text{numDistrictsInCity}(i) \right) + \text{districtIndex}$$

Where:

- cityIndex is the 0-based city position
- districtIndex is the 0-based district position
- numDistrictsInCity(i) is districts in city i for the given utility

4.2.10 Example

Consider this 3-city system:

City	Districts	Global IDs
Gotham (2)	Downtown, Uptown	0, 1
Metropolis (3)	Business, Residential, Industrial	2, 3, 4
Star City (1)	Tech Park	5

4.2.11 Key Properties

Property	Description
Uniqueness	Each district has exactly one global ID per utility
Contiguity	IDs are assigned sequentially without gaps
Consistency	Same district has matching IDs across utilities
Offset-Based	Depends on preceding cities' district counts

4.2.12 Usage in System

These IDs are critical for:

- Building the combined adjacency matrix
- Tracking connectivity across cities
- Mapping between local and global representations

isPowerGridConnected()

- Checks if all districts in all cities are reachable via power lines using DFS algorithm.
- Returns true if the entire grid is connected and false if otherwise.

printSystem()

- Returns a formatted string summarizing all cities and intercity links.
- Includes district lists, utility networks, and connection details.

4.3 UtilityNetwork

The UtilityNetwork class models infrastructure networks (power, water, and transportation) within a city's districts. It uses an adjacency matrix to represent connections between districts, where weights denote capacity, distance, or other relevant metrics.

4.3.1 Members

type

- String indicating the utility type ("POWER", "WATER", or "TRANSPORTATION")

adjacencyMatrix

- 2D integer array representing connections between districts
- Non-zero values indicate connection weights
- Symmetric for POWER/WATER (undirected), asymmetric for TRANSPORTATION (directed)

4.3.2 Functions

UtilityNetwork(String type, int numDistricts)

- Constructor that initializes the network with a given type and number of districts
- validation checks include checking that the utility types are either one of the prescribrd 3 only and districts >0
- otherwise throw new IllegalArgumentException("Invalid type. Must be 'POWER', 'TRANSPORTATION', or 'WATER'.") and throw new IllegalArgumentException("Number of districts must be greater than or equal 0.")
- Take care to use exact exceptions
- Creates an empty adjacency matrix (all zeros)

addConnection(int from, int to, int weight)

- Adds an undirected connection between two districts with specified weight
- Used for POWER and WATER networks
- Ensure district index is valid and weight is greater than 0 otherwise IllegalArgumentException("Invalid district indices.");

  throw new IllegalArgumentException("Weight must be greater than 0.");
- make sure to use exact exception

addTransportConnection(int from, int to, int weight)

- Adds a directed connection for TRANSPORTATION networks
- Ensure district index is valid and weight is greater than 0 otherwise IllegalArgumentException("Invalid district indices.");

  throw new IllegalArgumentException("Weight must be greater than 0.");
- make sure to use exact exception

getWeight(int from, int to)

- Returns the connection weight between two districts
- Returns 0 if no connection exists

hasConnection(int from, int to)

- Checks if a connection exists between two districts
- Returns true if weight > 0

getType()

- Returns the network type string

getAdjacencyMatrix()

- Returns the complete adjacency matrix

getNumDistricts()

- Returns the number of districts in the network

printNetwork()

- Generates a formatted string representation of the network
- Includes both matrix view and connection list
- Handles directed/undirected display appropriately
- Returns "No connections" message for empty networks
- **Do not alter this function as it will negatively affect your marks

4.4 InterCityNetwork

Manages connections between districts across different cities using an adjacency list representation, unlike the adjacency matrix used in intercity networks and utility networks,

4.4.1 Members

adjacencyList

- Array of Node objects, where each index corresponds to a city.
- Each Node contains a linked list of intercity connections.

4.4.2 Functions

InterCityNetwork(int numCities)

- Initializes an empty adjacency list for numCities.

addConnection(int fromCity, int toCity, int fromDistrict, int toDistrict, int weight)

- Adds bidirectional connections between districts of two cities.
- Example: addConnection(0, 1, 2, 3, 50) links: - City 0's District 2 <-> City 1's District 3 (weight=50).
- ** This is an adjacency list. array index = metacity id/index in metasystem
- make sure to append nodes

colorIntercityWithBrelaz(MetaCity[] cities)

- Implements the DSATUR algorithm to color cities, ensuring no adjacent cities share colors.
- **Please take note of the following colour order "Red", "Green", "Blue", "Yellow", "Purple"

- Returns a summary of assigned colors and saturation degrees.
- you are welcome to use the provided helper to print final coloring

printConnections(MetaCity[] cities)

- Generates a formatted string of all intercity links.
- Please do not edit this function as it will negatively affect your marks.

4.5 MetaCity

Represents a city composed of districts and their utility networks (power, water, transportation).

4.5.1 Members

name

- String storing the city's name (e.g., "Gotham")

districts

- Array of `District` objects
- Index corresponds to district ID

utilityNetworks

- Array of three `UtilityNetwork` objects:
  1. `POWER` (index 0)
  2. `WATER` (index 1)
  3. `TRANSPORTATION` (index 2)

blockedDistricts

- Boolean array marking inaccessible districts
- Used in `dfsTransportWithBlocks()`

4.5.2 Functions

MetaCity(String name, int numDistricts)

- Initializes city with:
  - Name
  - Empty district array of size `numDistricts`
  - Three pre-configured utility networks

addDistrict(int index, String name)

- Creates new district at specified index
- Validates:
  - Index bounds
  - Non-empty name

  - No duplicate district
  - throw below exceptions otherwise
  - IllegalArgumentException("Index out of bounds for districts array.");
    throw new IllegalArgumentException("District name cannot be null or empty.");
    throw new IllegalStateException("A district already exists at this index.");

addUtilityConnection(String type, int from, int to, int weight)

- Adds connection to specified network:
  - `TRANSPORTATION`: Directed (one-way)
  - `POWER/WATER`: Undirected (two-way)

dfsTransportWithBlocks(int start, boolean[] blocked)

- Performs DFS on transportation network
- Skips blocked districts
- Returns visitation order as string
- take note of the following string output, ensure you have newlines Visiting: XVisiting: YVisiting: Z

dijkstraTransport(int start)

- Computes shortest paths from starting district
- Returns array of travel times (minutes)
- Uses classic Dijkstra's algorithm

printDistances(int start)

- Formats Dijkstra results as:

```
Downtown : 0 minutes
Uptown : 5 minutes
Industrial Zone : Unreachable
```

getUtilityNetwork(String type)

- Returns network by type (case-sensitive)
- Example: `getUtilityNetwork("WATER")`

printCity()

- Generates comprehensive report including:
  - District list
  - All utility network matrices
  - Connection lists
  - Please do not edit this function as it will negatively affect your marks.

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 2:

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 2: 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.
- District.java
- UtilityNetwork.java
- IntercItyNetwork.java
- MetaCity.java
- MetaCitySytem.java
- Any textfiles needed by your Main

7 Allowed libraries

- java.Array.Utils

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 3 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!**