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: 20-05-2023 at 06:30

Due Date: 07-06-2023 at 23:59

Total Marks: 300

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

After the successful completion of this assignment, you will have implemented:

- A graph-based interpretation of a maze.
- A graph complexity reduction algorithm.
- A depth-first search algorithm.
- A shortest path algorithm.
- Allowed the program to suggest the best choice based on a heuristic value.
- Your own data structure to support the graph data structure. Please see Section 7 for more information about the restrictions and implementation of your own data structure

4 Introduction

Complete the task below. Certain classes have been provided for you, alongside this specification in the Student files folder. A very basic main has been provided. **Please note this main is not extensive and you will need to expand on it**. Remember to test boundary cases. Submission instructions are given at the end of this document.

You will be implementing something similar to assignment 1, but this time utilizing graphs as the underlying data structure. You will need to traverse a maze starting from a single starting point to various locations in the maze. The maze will contain doors, keys, traps and a collection of possible **treasures/goals**.

Due to the way the graph will be constructed, a three-stage reduction algorithm will be employed to reduce the complexities of the maze, such that the programs can work with more complex and bigger mazes.

The maze is split into two parts, with doors leading between to the two parts. In order to be able to pass through a door, you first need to find a key that is able to unlock the door. For the sake of simplicity, it can be assumed that any key can unlock any door.

5 Task 1: Maze solver

Your task for this assignment would be to implement the following class diagram, as described in later sections.

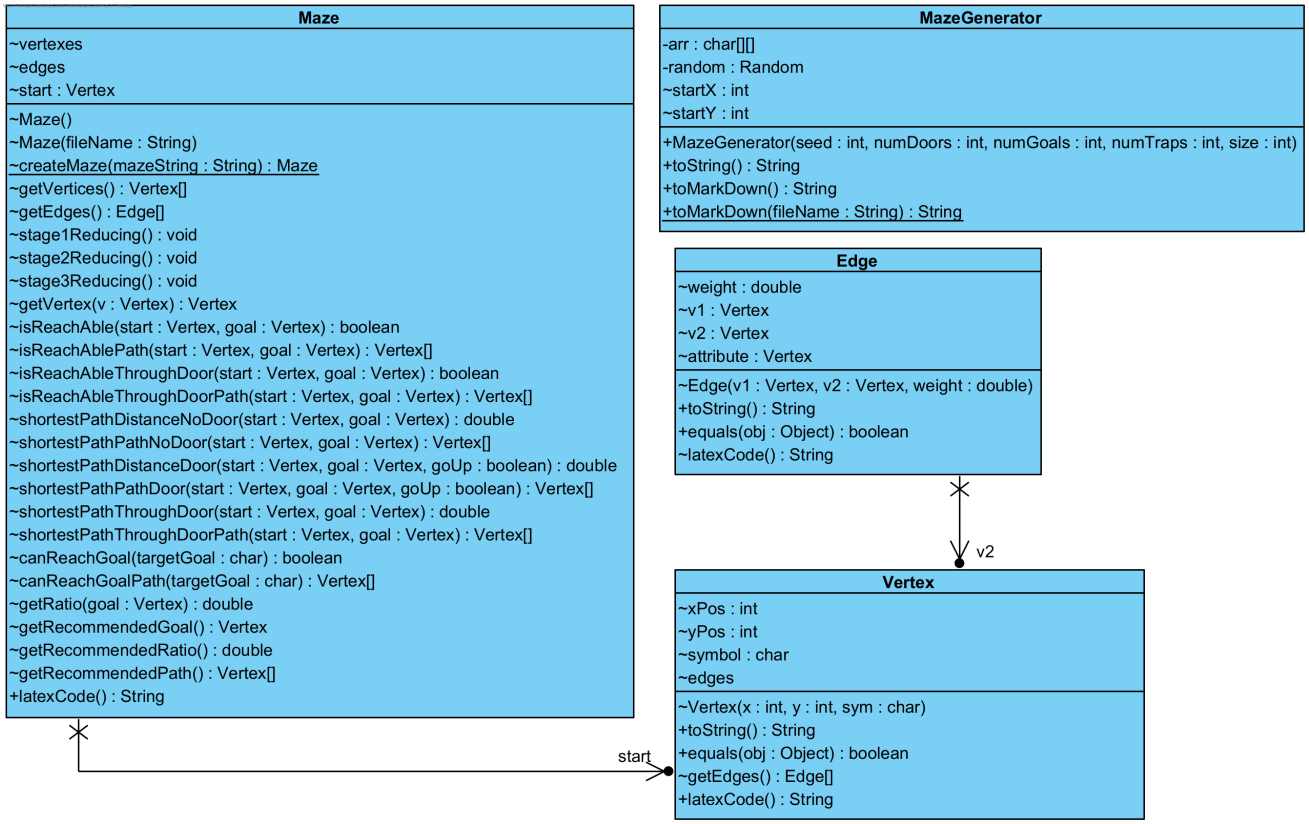


Figure 1: Class diagram

5.1 Vertex

The objects of this class represent single vertices in the graph which is connected via objects of the Edge class (See section 5.2).

5.1.1 Members

- xPos: int
  - This is the x coordinate of the vertex.

- yPos: int
  - This is the y coordinate of the vertex.
  - For example, a Vertex at position 0,0 is at the top left-hand corner of the file.
- symbol: char
  - This is the symbol of the vertex.
  - The following list represents the possible symbols:
    * Blank symbol (‘ ’): A part of the walk way in the maze.
    * D: A door.
    * K: Key for the door.
    * T: Trap in the walkway.
    * 0-9: IDs of possible goals or treasures that can be found in the maze. The importance of this number will be discussed later.
- edges
  - This is a collection of edges that is connected to the current vertex.
  - Please see Section 7 for more information about the restrictions and implementation of your own data structure.

5.1.2 Functions

- Vertex(x: int, y: int, sym: char)
  - This is the constructor for the vertex class.
  - The constructor should initialize each member variable accordingly.
- toString(): String
  - This is a provided function and should not be changed.
- equals(obj: Object): boolean

  - This is a provided function and should not be changed.
  - A vertex is seen as equal if it has the same coordinates. The symbol is irrelevant.

- getEdges(): Edge[ ]
  - This function should return an array of edges, populated from the contents of the edges member variable.
  - The array should not contain null values.
  - The order is not important.

- latexCode(): String
  - This is a provided function and should not be changed.
  - This function is used to create the latex code that will be able to graphically render the graph.

5.2 Edge

The objects of this class represent single edges in the graph which connects two vertices (See section 5.1) together.

5.2.1 Members

- weight: double
  - This is the distance between the two vertices.
  - The default distance is 1.

- v1: Vertex
  - This is the first vertex that is connected by the edge.
  - The order of vertices does not matter.

- v2: Vertex
  - This is the second vertex that is connected by the edge.
  - The order of vertices does not matter.

5.2.2 Functions

- Edge(v1: Vertex, v2: Vertex, weight: double)
  - This is the constructor for the Edge class.
  - Initialize the member variables appropriately.
  - The order of vertices does not matter.
- toString(): String
  - This is a provided function and should not be changed.
- equals(obj: Object): boolean
  - This is a provided function and should not be changed.
  - An edge is seen as equal if it has the same vertices. The order is irrelevant.
- latexCode(): String
  - This is a provided function and should not be changed.
  - This function is used to create the latex code that will be able to graphically render the graph.

5.3 Maze

The objects of this class represent a single graph, which consists of a set of vertices (See Section 5.1) which are connected with edges (See Section 5.2). The maze you will be implementing is a weighted undirected graph.

5.3.1 Members

- vertices
  - This is a collection of vertices that are contained in the graph.
  - You need to use your own data structure to store the collection of vertices.

- edges
  - This is a collection of edges that are contained in the graph.
  - *Hint: It helps if there are no duplicates in this collection.*
  - You need to use your own data structure to store the collection of edges.
- start: Vertex
  - This is the starting vertex of the maze.

5.3.2 Functions

- Maze()
  - This is the default constructor and should initialize the start vertex with null.
  - Initialize vertices and edges as you deem fit, with your own data structure.
- latexCode(): String
  - This is a provided function and should not be changed.
  - This function is used to create the latex code that will be able to graphically render the graph.
- Maze(fileName: String)
  - This constructor will build a maze from the textfile using the passed-in *fileName*.
  - It can be assumed the textfile exists.
  - If for some reason a `FileNotFound` exception was thrown catch it in the function and print out: **File not found**.
  - You should iterate through the textfile and construct the graph.
  - Each non-wall element (i.e. anything that is not a hash (#)) should be added to the graph as a vertex.

  - Each vertex is then linked via an edge to vertices that are immediately north, south, east and west of said vertex, if there are any.
  - The row number represents the y-position of the vertex and the position in the row the x-position of the vertex.
  - The vertex's symbol is represented by the character at the current vertex's coordinate in the textfile.
  - The edges should be initialized to have a default weight of 1.
  - Note there is no limit on the size of the graph, so you will need to use your own data structure for the vertices and edges.
  - Please see Section 12 for an example of a textfile to graph conversion.
  - Ensure this function works exactly as expected, as it will be used extensively during the assessment of your submission

- createMaze(mazeString: String): Maze
  - This is a **static** function that takes in a string and converts it to graph which is returned.
  - The end of each row in the string is denoted by a newline character ('\n').
  - Use the same algorithm as the Maze constructor to convert the string to a graph.
  - *Hint: Keep the concept of code re-usability and function outsourcing in mind.*
  - Ensure this function works exactly as expected, as it will be used extensively during the assessment of your submission

- getVertices(): Vertex[]
  - This function should return an array containing all of the vertices currently in the graph.
  - If there are no vertices in the graph, the function should return a new array with a size of 0.

  - *Remember null and size of 0 are two different things.*
  - Ensure the array does not contain any null values.

- getEdges(): Edge[]
  - This function should return an array containing all of the edges currently in the graph.
  - There should be no duplicates in the array.
  - *Hint: Use one of the provided functions to determine if an edge is already contained in the array.*
  - If there are no edges in the graph, the function should return a new array with a size of 0.
  - *Remember null and size of 0 are two different things.*
  - Ensure the array does not contain any null values.

- stage1Reducing(): void
  - After the initial construction of the graph, there will be a few vertices that only have exactly two edges connecting to other vertices, yet contain only a *blank symbol* (‘ ’). **Look at the example at the end of the assignment for a graphical representation of stage 1 reduction.**
  - This function aims to remove these excess vertices.
  - For each blank symbol vertex in the graph that has exactly two edges, remove the vertex and the original edges leading to this vertex and link the neighbouring vertices together via a **new** edge.
  - This new edge should have the combined weight of the two edges that were removed.
  - Reminder that **only** blank symbol vertices with exactly two edges should be removed.
  - Please see Section 12 for an example of stage 1 reduction.

  - Ensure this function works exactly as expected as it will be used extensively during the assessment of your submission
  - Please note that even after removing a vertex from the graph and relinking the edges it can still be possible for the vertex to only have two connections. This vertex will again need to be processed.

- stage2Reducing(): void
  - Now the majority of redundant vertices from the graph, the next step is to remove dead ends.
  - A dead end is defined as any vertex in the graph that has only a single edge and has a *blank symbol*(‘ ’). **Look at the example at the end of the assignment for a graphical representation of stage 2 reduction.**
  - These vertices should be simply removed from the graph.
  - Together with deadends, any isolated vertices, i.e. a vertex that has no edges connected to it, also needs to be removed.
  - Please see Section 12 for an example of stage 2 reduction.
  - Ensure this function works exactly as expected as it will be used extensively during the assessment of your submission

- stage3Reducing(): void
  - Traps are nasty things and need extra precautions.
  - As such, any edge that is connected to a vertex that is a trap (i.e. has a symbol that is a T), should have its weight doubled. **Look at the example at the end of the assignment for a graphical representation of stage 3 reduction.**
  - Please see Section 12 for an example of stage 3 reduction.
  - Ensure this function works exactly as expected as it will be used extensively during the assessment of your submission

- getVertex(v: Vertex): Vertex

  - This function should return the vertex in the graph that is equal to the passed in the vertex.
  - If there is no vertex that is equal to the passed vertex, return null.

- isReachAble(start: Vertex, goal: Vertex): boolean
  - This function needs to determine if there exists a path from the start vertex to the goal vertex.
  - This should be done in a depth-first manner.
  - Note, you are **not** allowed to traverse through a door in this function.
  - If a path does exist return true, else, return false.
  - Traverse the edges of a vertex in ascending order, based on the `toString` of the edges of the vertex.

- isReachAblePath(start: Vertex, goal: Vertex): Vertex[]
  - This function needs to determine if there exists a path from the start vertex to the goal vertex, and return the path.
  - Note, you are not allowed to traverse through a door in this function.
  - This should be done in a depth-first manner.
  - Traverse the edges of a vertex in ascending order, based on the `toString()` of the edges of the vertex.
  - You need to return an array that contains the vertices that were visited, in the order that they were visited.
  - If no path exists, return a new Vertex array of size 0.
  - Ensure the array does not contain any null values.

- isReachAbleThroughDoor(start: Vertex, goal: Vertex): boolean
  - This function needs to determine if there exists a path from the start vertex to the goal vertex through a door.
  - Note, you **must** traverse through a door in this function.

  - This should be done in a depth-first manner.
  - If you do not need to traverse through a door to reach the goal vertex, return false.
  - If a path does exist, return true.
  - Traverse the edges of a vertex in ascending order, based on the `toString`() of the edges of the vertex.
  - Reminder: **You need a key** in order to be able to traverse through a door.
  - Use Algorithm 1 for further hints on how this can be done.

- isReachAbleThroughDoorPath(start: Vertex, goal: Vertex): Vertex[]
  - This function needs to determine if there exists a path from the start vertex to the goal vertex through a door, and return the path.
  - Note you **must** traverse through a door in this function.
  - This should be done in a depth-first manner.
  - If you do not need to traverse through a door to reach the goal state, return an array of length 0.
  - In order to obtain a standardized path, the algorithm described in Algorithm 1 must be followed.
  - Traverse the edges of a vertex in ascending order, based on the `toString` of the edges of the vertex.

- shortestPathDistanceNoDoor(start: Vertex, goal: Vertex): double
  - This function needs to determine the shortest path distance from the start to the goal without traversing through a door.
  - If no path exists, return the infinite double constant.
  - If the start vertex is a door, the function also needs to return the infinite double constant.
  - If either the start or goal vertices are null, return an infinite double constant.

Algorithm 1 isReachAbleThroughDoorPath pseudocode algorithm

Require: keys := list of all the keys in the maze sorted ascendingly based on the `toString` representation of each key vertex.
Require: doors := list of all the doors in the maze sorted ascendingly based on the `toString` representation of each door vertex.

```
for k in keys do
    if k is reachable from start then
        for d in doors do
            if d is reachable from k then
                if goal is reachable from d then
                    return the path from start to k to d to goal.
                end if
            end if
        end for
    end if
end for
```

  - If the passed-in parameters are not null, it can be assumed that the vertices are in the graph.

- shortestPathPathNoDoor(start: Vertex, goal: Vertex): Vertex[]
  - This function needs to return the shortest path from the start to the goal without traversing through a door, as an array of vertices.
  - The array needs to be ordered from the starting vertex to the goal vertex.
  - Ensure the array does not contain any nulls.
  - If no path exists, return a new Vertex array of size 0.
  - If the start vertex is a door, the function also needs to return a new Vertex array of size 0.
  - If either the start or goal vertices are null, return a new Vertex array of size 0.
  - If the passed-in parameters are not null, it can be assumed that the vertices are in the graph.
  - It can be assumed that the graphs that will be used for testing your assignment, will only have a single shortest path.

- shortestPathDistanceDoor(start: Vertex, goal: Vertex, goUp: boolean): double
  - This function needs to determine the shortest path distance from the start to the goal **without traversing** through a door.
  - The difference between this function and `shortestPathDistanceNoDoor()` is that this function **starts** at a door.
  - Due to this, the *goUp* parameter is introduced.
  - If the *goUp* parameter is true, the shortest path algorithm should explore north of the door.
  - If the *goUp* parameter is false. the shortest path algorithm should explore south of the door.
  - If the start vertex is not a door, return the infinite double constant.
  - If no path exists, return the infinite double constant.
  - If either the start or goal vertices are null, return an infinite double constant.
  - If the passed-in parameters are not null, it can be assumed that the vertices are in the graph.
  - It can be assumed that the graphs that will be used for testing your assignment, will only have a single shortest path.
- shortestPathPathDoor(start: Vertex, goal: Vertex, goUp: boolean): Vertex[]
  - This function needs to return the shortest path from the start to the goal **without** traversing through a door, as an array of vertices.
  - The difference between this function and `shortestPathPathNoDoor` is that this function starts at a door.
  - Due to this, the *goUp* parameter is introduced.
  - If the *goUp* parameter is true, the shortest path algorithm should explore north of the door.

  - If the *goUp* parameter is false, the shortest path algorithm should explore south of the door.
  - The array needs to be ordered from the starting vertex to the goal vertex.
  - Ensure the array does not contain any nulls.
  - If no path exists, return a new Vertex array of size 0.
  - If the start vertex is not a door, the function also needs to return a new Vertex array of size 0.
  - If either the start or goal vertices are null, return a new Vertex array of size 0.
  - If the passed-in parameters are not null, it can be assumed that the vertices are in the graph.
  - It can be assumed that the graphs that will be used for testing your assignment, will only have a single shortest path.

- shortestPathThroughDoor(start: Vertex, goal: Vertex): double
  - This function needs to determine the shortest path distance from the start to the goal and it **must** traverse through a door.
  - Refer to Algorithm 2 for implementation assistance.
  - If either the start or goal vertices are null, return an infinite double constant.
  - If the passed-in parameters are not null, it can be assumed that the vertices are in the graph.
  - It can be assumed that the graphs that will be used for testing your assignment, will only have a single shortest path.

- shortestPathThroughDoorPath(start: Vertex, goal: Vertex): Vertex[]
  - This function needs to determine the shortest path from the start to the goal and it **must** traverse through a door.

Algorithm 2 shortestPathThroughDoor pseudocode algorithm

Require: keys := list of all the keys in the maze sorted ascendingly based on the `toString` representation of each key vertex.
Require: doors := list of all the doors in the maze sorted ascendingly based on the `toString` representation of each door vertex.

```
currentSmallestDistance = Double.POSITIVE_INFINITY
for k in keys do
    if k is reachable from start then
        for d in doors do
            if d is reachable from k then
                if goal is reachable from d then
                    cD = shortestPath(start,k) + shortestPath(k,d) + shortestPath(d,goal)
                    if cD < currentSmallestDistance then
                        currentSmallestDistance = cD
                    end if
                end if
            end if
        end for
    end if
end for
return currentSmallestDistance
```

  - The array needs to be ordered from the starting vertex to the goal vertex.
  - Refer to Algorithm 2 for implementation guidance.
  - If either the start or goal vertices are null, return a new Vertex array of size 0.
  - If the passed-in parameters are not null, it can be assumed that the vertices are in the graph.
  - It can be assumed that the graphs that will be used for testing your assignment, will only have a single shortest path.

- canReachGoal(targetGoal: char): boolean
  - This function needs to determine whether the goal vertex with the same symbol as the passed-in parameter, is reachable from the maze's start vertex.
  - In this function, you will need to first check if the goal is reachable without traversing a door, and if not, then traverse through a door.

  - If the passed in parameter does not correspond with a goal in the maze, return false.
  - *Hint: reuse some of the functions already discussed*

- canReachGoalPath(targetGoal: char): Vertex[]
  - This function needs to determine if the goal vertex with the same symbol as the passed-in parameter, is reachable from the maze's start vertex, and return a path to that goal.
  - In this function, you will need to first check if the goal is reachable without traversing a door, and if not, then traverse through a door.
  - If the passed in parameter does not correspond with a goal in the maze, return a new Vertex array of size 0.
  - *Hint: reuse some of the functions already discussed*

- getRatio(goal: Vertex): double
  - This function needs to return the "treasure ratio" for the passed in goal.
  - If the goal vertex is not in the graph, the function needs return the double infinite constant.
  - The "treasure ratio" is defined as:

$$tR = \frac{goalTreasure}{shortestDistance}$$

  - The *goalTreasure* value can be calculated as the numerical equivalent of the symbol for the passed-in goal multiplied by 100.
  - For example if the goal has a symbol of 4, then the "goal treasure" is 400.
  - To get the *shortest path distance* value, first determine if you need to traverse a door or not, and then use the appropriate function.

- getRecommendedGoal(): Vertex
  - This function needs to return the goal with the highest "treasure ratio".

  - It can be assumed that two treasure ratios on FitchFork will **never** be the same.

- getRecommendedRatio(): double
  - This function needs to return the "treasure ratio" of the recommended goal vertex.

- getRecommendedPath(): Vertex[]
  - This function needs to return the shortest path as an array of vertices, for the recommended goal vertex.
  - It can be assumed that the graphs that will be used for testing your assignment, will only have a single shortest path.

5.4 MazeGenerator

This is a provided class that has the ability to generate mazes for you. This class will be overwritten on FitchFork.

5.4.1 Functions

The class has the following functions that may be useful:

- MazeGenerator(int seed, int numDoors, int numGoals, int numTraps, int size).
  - This is the constructor for the MazeGenerator class.
  - The seed is used for the seeded randomness in the maze creation process.
  - The numDoors parameter specifies the maximum number of doors the maze can have.
  - The numGoals parameter specifies the maximum number of goals the maze can have.
  - The numTraps parameter specifies the maximum number of traps the maze can have.

- There are some finer parameter checking built-in to ensure that the passed in parameters are valid.

- toString(): String
  - This function returns a single string representation of the maze, where the end of a row is represented by a '\n' character.

- toMarkDown(): String
  - This function returns a string representation, which can be saved to a markdown (.md) file which will give a easier to visually use representation of the maze.
  - Note, you are not expected to be able to create a maze based on the markdown string or file.

6 Latex Code

To aid you with visualizing the graphs, you have been provided with a function that converts the graph to latex code and returns a string which can be used to compile a latex document with the graph drawn.

The following website can be used to render the latex: `https://latex.informatik.uni-halle.de/latex-online/latex.php`.

If the graph is too small change the following line:

```
\begin{tikzpicture}[node/.style={circle, draw, minimum
   size=1.2em},yscale=-1, xscale=1]
```

Changing the **xscale** and **yscale** will increase the size of the graph to improve the readability of the graph.

7 Own Data Structure

This assignment needs some additional data structure to accomplish a few of the implementations. You need to create your own custom data structure. You may decide which custom data structure you would like to implement from any of the data structures learnt in COS110 and COS212.

The following are some of the recommended functional requirements of your custom data structure:

- Dynamically sized
- Insert
- Remove
- Search
- Contains
- Size
- Sort based on the `toString` value of the data.

Note, this data structure will not be explicitly tested on FitchFork, **so it is your responsibility to ensure this data structure is fully functional**.

8 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 App.java file) that will be used to test an Instructor-provided solution. You may add any helper functions to the App.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 ./ App
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.

9 Upload checklist

The following files should be in the root of your archive

- Maze.java
- App.java
- Edge.java
- MazeGenerator.java
- Vertex.java
- Any other java files you used to create your own datastructue

10 Allowed libraries

- Iterator
- File
- FileNotFoundException
- Scanner
- Random

11 Submission

You need to submit your source files on the Fitch Fork website (https://ff.cs.up.ac.za/). All methods need to be implemented (or at least stubbed) before submission.

Your code should be able to be compiled with the following command:

```
make *.java
```

and run with the following command:

```
java App
```

You have 3 submissions and your best mark will be your final mark. Upload your archive to the Assignment 3 slot on the Fitch Fork website. Submit your work before the deadline. **No late submissions will be accepted!**

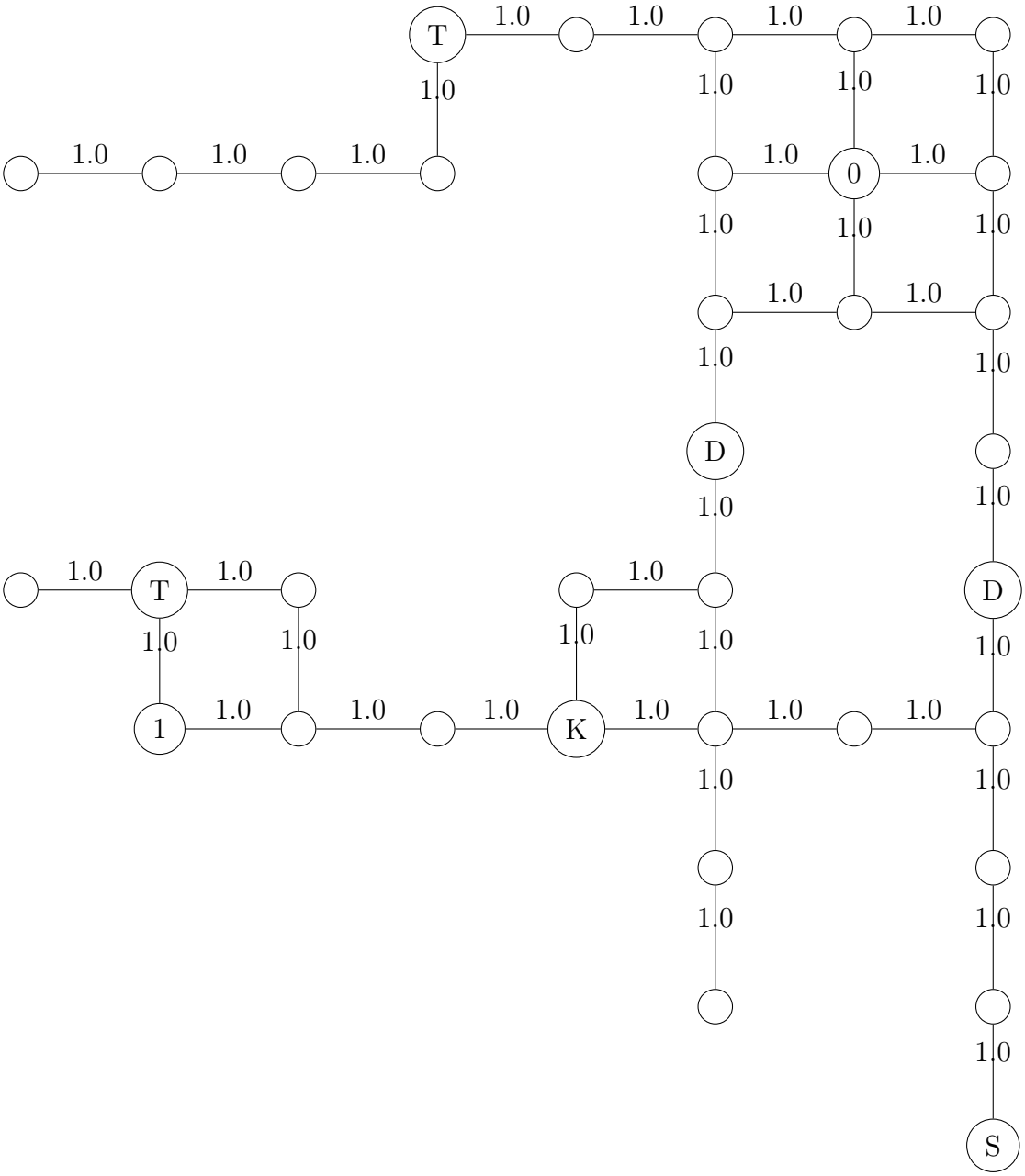
12 Example

Consider the following maze was created:

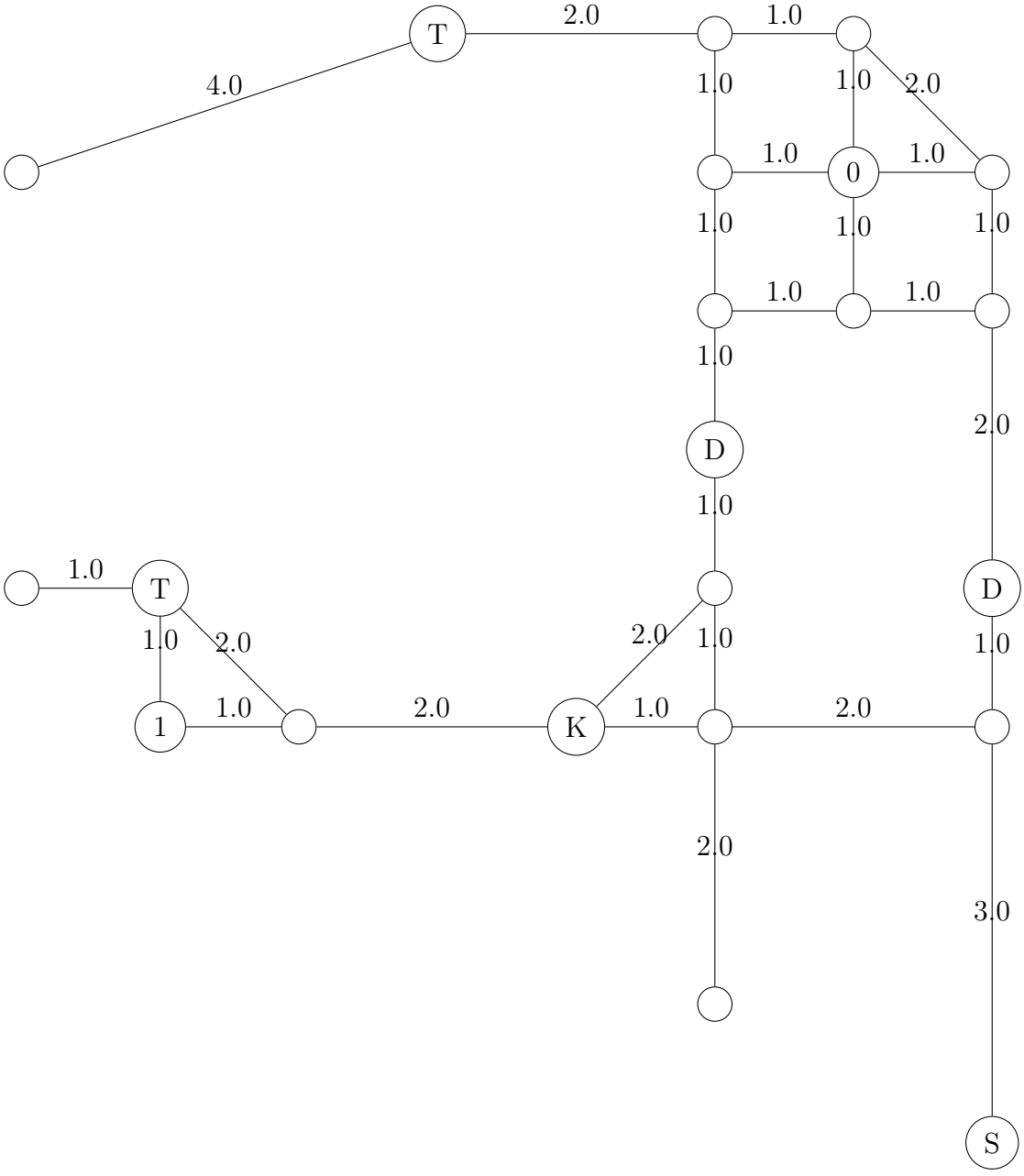
```
##########
####T     #
#     # 0 #
######    #
######D# #
# T #  #D#
##1  K    #
###### # #
###### # #
########S#
```

Note the coordinates start at (0,0) in the upper left-hand corner and goes to (9,9) in the lower right corner.

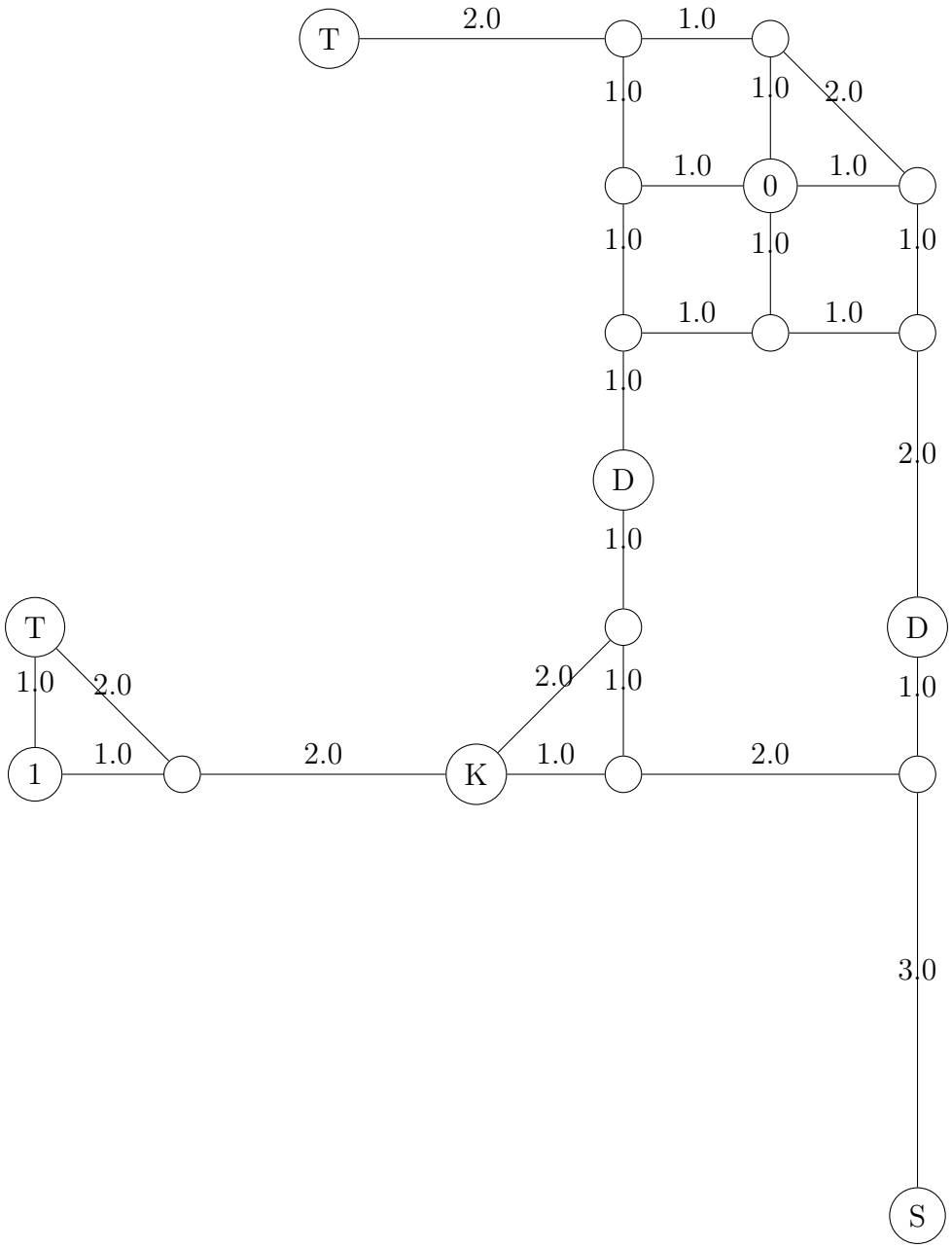
When converted to a graph in the constructor the graph is as follows:



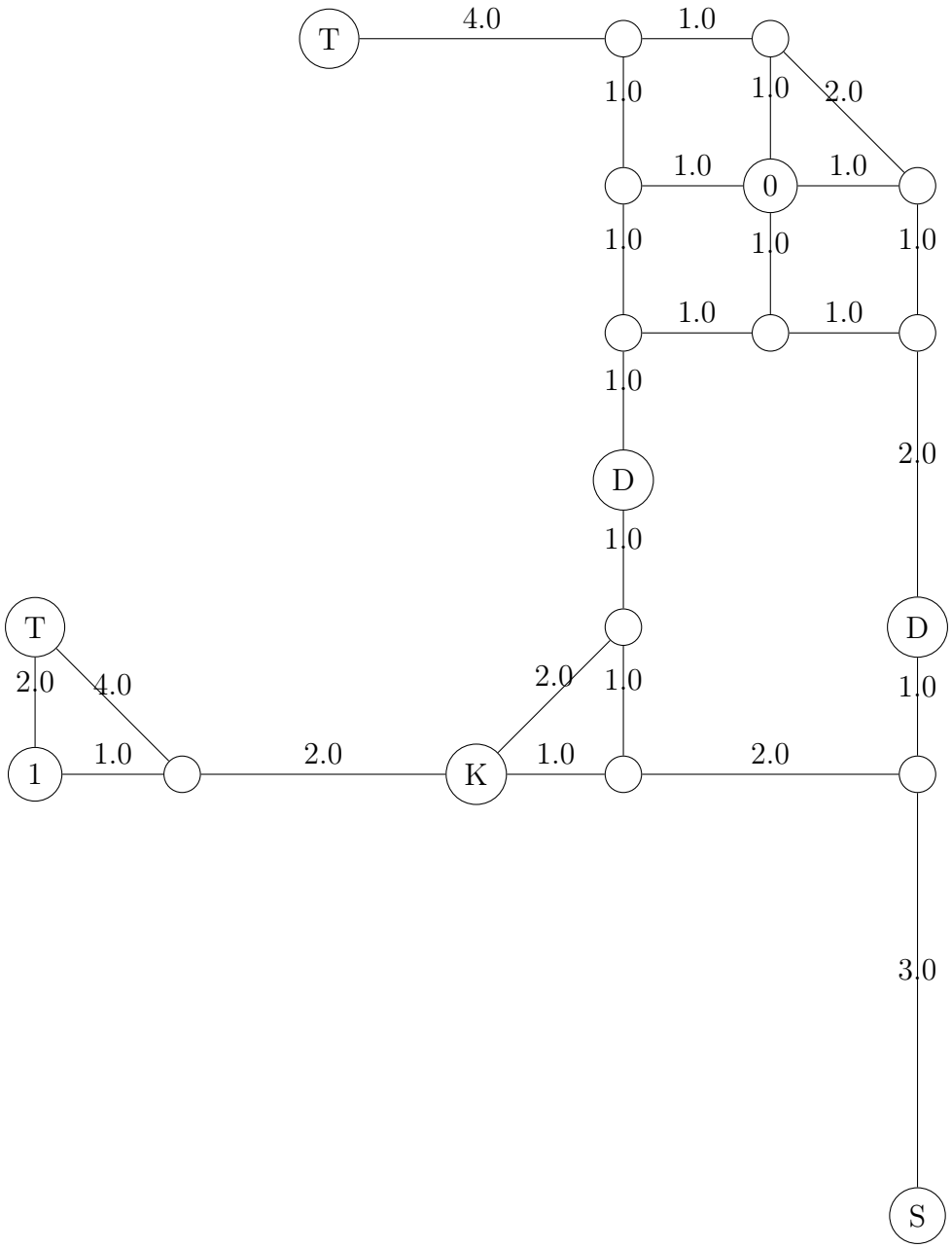
After stage1 reduction, we obtain:



After stage2 reduction, we obtain:



After stage3 reduction, we obtain:



- Is T reachable from S: false
- Is 1 reachable from S: true
- Is 1 reachable from S path: (8:9)[S] (8:6)[ ] (8:5)[D] (6:6)[ ] (5:6)[K] (3:6)[ ] (2:6)[1]
- Is T reachable from S through a door: true
- Is T reachable from S through a door: (8:9)[S] (8:6)[ ] (8:5)[D] (6:6)[ ] (5:6)[K] (3:6)[ ] (2:6)[1] (2:5)[T] (6:5)[ ] (6:4)[D] (6:3)[ ] (6:2)[ ] (6:1)[ ] (4:1)[T]
- Shortest path from S to T without traversing a door: Infinity
- Shortest path from S to 1 without traversing a door: 9
- Shortest path from S to T with traversing a door: 16
- Shortest path from S to 1 with traversing a door: Infinity
- “treasure ratio" from S to 1: 11.11111111111111

13 Change log

- 25/05/2024:
  - Added additional clarity on the type of graph.
  - Added additional clarity on stage 1 reduction.
  - Added instruction on what to do with isolated vertices for stage 2 reduction.
  - Included the provided code for the latexCode functions in the student files.
- 27/05/2024:
  - Vertex’s edges member variable description updated.
  - Added explicit instruction about the FileNotFound exception in the Maze constructor.