Assignment 4

COS 212

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

Department of Computer Science
Deadline: 10/05/2019 at 23:00

General instructions:

- This assignment should be completed individually, no group effort is allowed.
- Be ready to upload your assignment well before the deadline, as no extension will be granted.
- You may not import any of Java's built-in data structures. Doing so will result in a mark of zero. You may only make use of native 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 0. Only the output of your program will be considered for marks, but your code may be inspected.
- **All submissions will be checked for plagiarism.**
- Read the entire assignment before you start coding.
- You will be afforded three upload opportunities.

Plagiarism:

The Department of Computer Science regards plagiarism as a serious offence. Your code will be subject to plagiarism checks and appropriate action will be taken against offending parties. You may also refer to the the Library's website at `www.library.up.ac.za/plagiarism/index.htm` for more information.

After completing this assignment:

Upon successful completion of this assignment you will have implemented a graph data structure to represent a three dimensional maze. You will also have implemented an algorithm to find the longest simple path in the maze by using the graph data structure.

Your task

Your task will be to find the longest simple path in a three dimensional maze. This will involve creating a graph data structure from the maze representation. Once the maze is converted to a graph, graph theory can be applied and the problem becomes more familiar.

This assignment is divided into a number of tasks which build on each other.

Task 1: Creating a Graph from the Maze

To assist in finding the longest path the maze consisting of a 3D character array will be converted to a graph.

Maze Representation

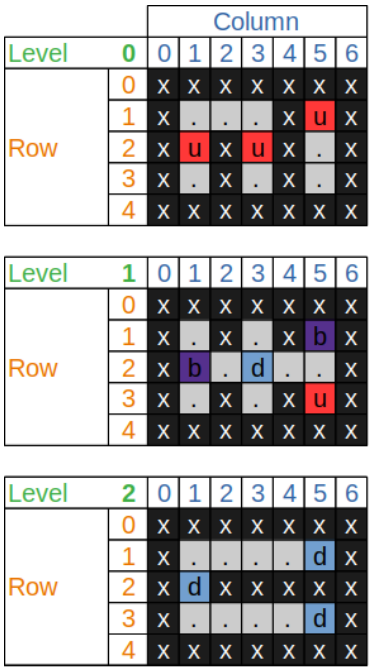

Figure 1: Representation of the 3D maze character array

The maze will be represented as a 3D character array:

```
Character[][][] maze = new Character[levels][rows][cols]
```

Each position in the 3D array will contain one of five characters to indicate different types of tiles in the maze. The first dimension will represent the *level*. The second dimension will represent the row in that level. The third dimension will represent the column in that level.

For example to access the character on the level index 0, at row index 3, column index 4 you would use `maze[0][3][4]`

Navigating the maze can be done in 6 directions, depending on the type of tile:

- Let "forward" mean `row - 1`.
- Let "backward" mean `row + 1`.
- Let "right" mean `col + 1`.

- Let "left" mean `col - 1`.
- Let "up" mean `level + 1`.
- Let "down" mean `level -1`.
- It is not possible to navigate diagonally.

The maze consists of 5 types of tiles represented by different characters:

- `'x'` - **Wall**. It is not possible to "walk" trough a wall. These should not be included in the graph.
- `'.'` (period) - **Empty tile**. These can be walked over and may form part of a path. When creating the graph consider tiles that are forward, backward, left and right of an empty tile.
- `'u'` - **Empty tile, with a ladder going up**. This tile has all the properties of an empty tile with the addition that from the `'u'` tile you can go up to the tile on the level above. It is possible to walk over a `'u'` tile without going up. A `'u'` tile will always be paired with a `'d'` or `'b'` tile on the level above.
- `'d'` - **Empty tile, with a ladder going down**. This tile has all the properties of an empty tile with the addition that from the `'d'` tile you can go down to the tile on the level below. It is possible to walk over a `'d'` tile without going down. A `'d'` tile will always be paired with a `'u'` or `'b'` tile on the level below.
- `'b'` - **Empty tile, with a ladder going up and down**. This tile has all the properties of an empty tile with the addition that from the `'d'` tile you can go up and down to the tile on the level above and the level below. It is possible to walk over a `'b'` tile without going up or own. A `'b'` tile will always be paired with a `'u'` or `'d'` or `'b'` tile on the level above and below.

Graph Representation

Coordinates

Vertices in the graph will be uniquely identified by their coordinates as in the maze. In Fig. 1 the tile at position (1, 2, 1) will be the `'b'` tile. From this tile you can go to (1, 1, 1), (1, 3, 1), (1, 2, 2) on the same level, (0, 2, 1) on the level below and (2, 2, 1) on the level above.

When creating the graph the coordinates should be stored in the vertex.

Constructing the Graph

Level 0	Column 0	1	2	3	4
Row 0	x	x	x	x	x
1	x	.	.	x	x
2	x	u	x	u	x
3	x	x	x	.	x
4	x	x	x	x	x

Level 1	Column 0	1	2	3	4
Row 0	x	x	x	x	x
1	x	.	.	.	x
2	x	d	x	b	x
3	x	.	.	.	x
4	x	x	x	x	x

Level 2	Column 0	1	2	3	4
Row 0	x	x	x	x	x
1	x	.	.	.	x
2	x	x	x	d	x
3	x	x	x	x	x
4	x	x	x	x	x

Figure 2: Visual representation of the 3D maze character array

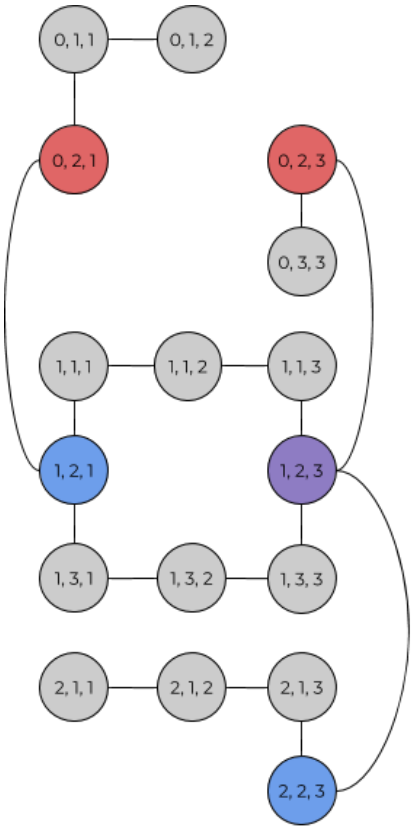

Figure 3: Visual representation of the graph created from Fig. 2

Based on the properties of the different tiles in the maze, an undirected graph has to be created. The edges will have an implicit weight of 1. Fig. 3 shows the graph representation of the maze depicted

in Fig. 1. The colours in the figure are not part of the graph and have been added for clarity. Each vertex stores the coordinates it represents in the maze.

The implementation of the graph data structure is left up to you.

Functions to Implement for Task 1

You are required to implement the following functions in `Graph.java`.

```
void createGraphFrom3DMaze(Character[][][] maze)
```

Use the information provided above to create a graph from the given maze. The input will be in the form of Fig. 2.

```
Vertex getVertex(Integer level, Integer row, Integer col)
```

Return the vertex with the given coordinates (level, row, col) If the vertex does not exist, return null. If the coordinates are out of bounds, return null.

```
Vertex[] getAllVertices()
```

Return all the vertices in the graph.

```
Vertex[] getAdjacentVertices(Vertex vertex)
```

Return the vertices adjacent to the given vertex. The vertices in the returned array can be in any order. Return an empty array if there are no adjacent vertices. If the vertex does not exist, return null. For example in Fig. 2 the vertices adjacent to the vertex at (0, 2, 1) are (0, 1, 1) and (1, 2, 1).

Task 2: Finding the Longest Simple Path in the Graph

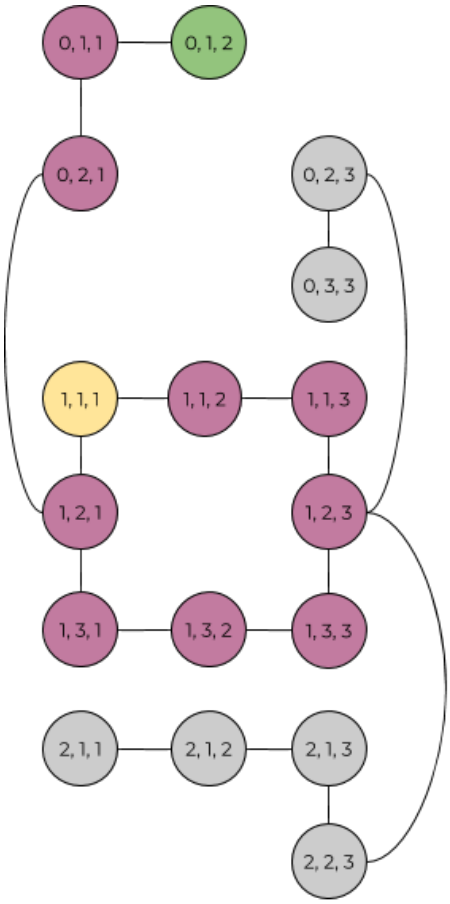

Figure 4: Longest path in the graph. Green = start vertex. Yellow = End vertex. Red = Intermediate vertices in the path

You will be required to find the longest simple path in the graph created in task 1. The path should have no repeating vertices. You can assume that the maze will only contain one longest path. For this task, the path length is the number of edges in the path. Fig. 4 depicts the longest path in the graph, which is 10 in this case. Note that the edge between (1, 1, 1) and (1, 2, 1) is not part of the path.

Functions to Implement for Task 2

You are required to implement the following functions in `Graph.java`.

```
Integer getLongestPathLength(Coordinates start, Coordinates end)
```

Return the length of the longest path from the given starting vertex coordinates, to the end vertex coordinates. The start and end vertices should be part of the path. The length can be calculated by summing the weights of edges in the path. For Task 2, each edge has an implicit weight of 1, so the path length is the number of edges in the path. If no path exists, return null.

```
Vertex[] getLongestPath(Coordinates start, Coordinates end)
```

Return an array of vertices that make up the longest path in order from start to end. The starting vertex should be the first element, and the ending vertex should be the last element, If there is no path, return an empty array. If no vertex exists at the given coordinates, return null.

Submission instructions

You must create your own makefile and submit it along with your Java code. Your code should be compiled with the following command:

```
javac *.java
```

Your makefile should also include a `run` rule which will execute your code if typed on the commandline as in `make run`.

Once you are satisfied that everything is working, you must tar all of your Java code, including any additional files which you've created, into one archive called sXXX.tar.gz, where XXX is your student/staff number. Submit your code for marking under the appropriate link (Assignment 4) before the deadline.