The University of Newcastle, Australia
School of Information and Physical Sciences
COMP2230/COMP6230 Algorithms
Assignment 1 Marks 100 Weight 15%
Individual Submission via Canvas

1 Assignment Overview

This assignment is to be done by each student *individually* using the Java programming language.

1. Write a *generator program* to take as input r the number of rows and c the number of columns, generate a random two-dimensional simple maze having r rows and c columns, and give the maze as output to a file in a predefined format and also on screen. For the maze generation, use an iterative Depth First Search (DFS) algorithm along with random neighbour selection.
2. Write a *solver program* based on the iterative versions of the Breadth First Search (BFS) and the DFS techniques to solve the generated maze. The solution to a maze is a sequence of moves and the solution will be written to a file in a predefined format and on screen.
3. Write a *verifier program* to take as input a maze and a solution to the maze, to verify that the maze is valid and the solution is valid for the maze, and to display an on-screen status report.
4. Compare and contrast the BFS and DFS techniques in terms of the program running times, the numbers of cells visited to solve the mazes, and the numbers of steps in the paths from the starting to the finishing cells without any repetition of any cells on the paths.

Use the same template iterative algorithm but with a queue for BFS and a stack for DFS when you write the generator and the solver programs. Using recursive DFS algorithms will be penalised.

2 Maze Defition

A simple maze has exactly one path from any point in the maze to any other point. Figure 1 (Left) shows an example. A simple maze does not have *inaccessible cells* (without any missing walls around), *open cells* (without walls around), and *circular paths* (visiting the same node more than once).

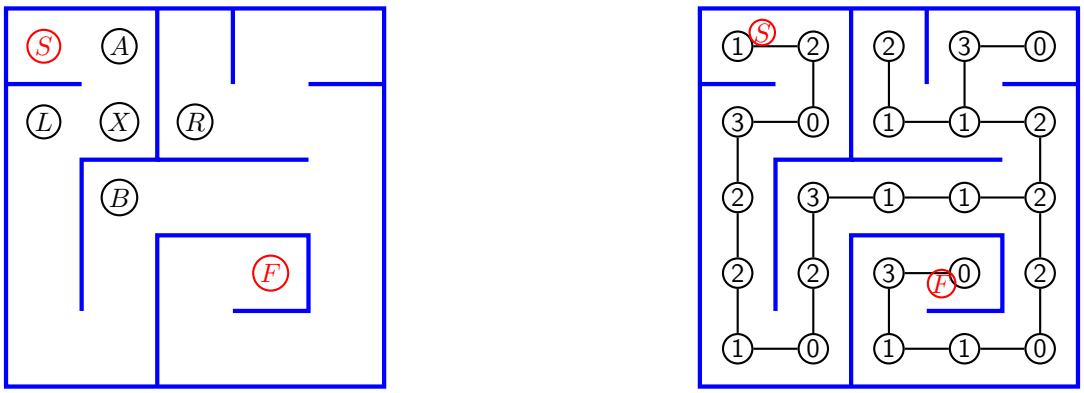

Figure 1: (Left) A sample maze with size 5×5 and (Right) its representation

A maze can be represented by a two-dimensional array of cells. A cell could have at most four walls and we know which walls exist and which ones are missing. We also know the starting and the finishing cells of the maze. The whole maze is assumed to be within a closed area. Figure 1 shows a maze having 25 cells with dimensions 5×5. The starting and the finishing cells are marked in the figure with S and F respectively. Notice that the top-left cell S has walls above, left, and below,

but no wall at the right, so we would be able to move from cell S to cell A at the right in a valid move. Actually, we can move from one cell to at most four neighbouring cells in four directions. To represent all possible combinations of moves, we would need $2^4 = 16$ values. However, we can simplify this somewhat. For each cell, we can just keep the information about whether we can move to the cell to the right or below. For connectivity with the cells to the left or above, we can just look at the information kept for those cells. This requires only 4 possible combinations:

0: for both right and below closed
1: for right only open
2: for below only open
3: for both right and below open

So, the top-left cell S in Figure 1 (Left) is represented by the value 1, since we can only move to the cell to the right, and not to the cell below. Moreover, the cell X in Figure 1 (Left) could be represented by 0, since we cannot move to the right cell R or the cell B below from cell X. To find the information that we can move to cell X from the cell A above, we can see the information of cell A. Similarly, to find the information that we can move to cell X from the left cell L, we can see the information of cell L. Figure 1 (Right) shows the representation of the simple maze.

3 Maze Generation

Mazes can be generated in many ways. We will focus on only one technique in this assignment. We will use a Random Walk technique to generate a random maze. The technique is as follows.

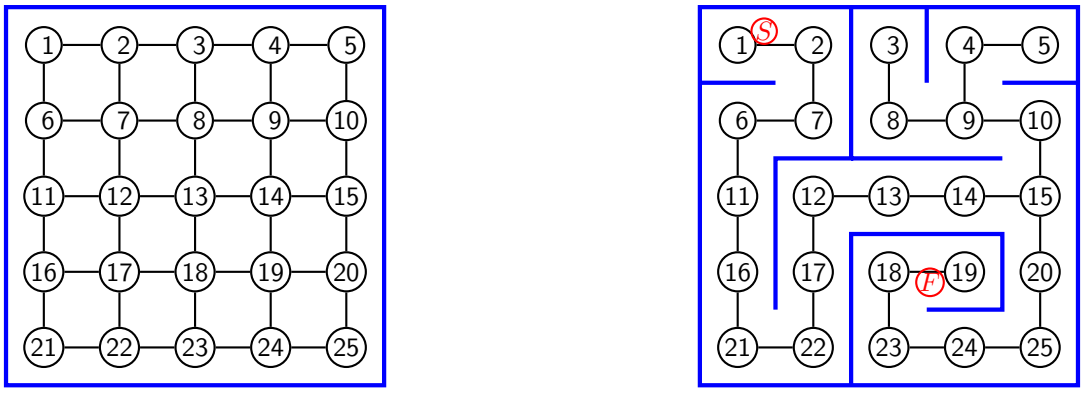

Figure 2: (Left) Sample 5×5 Grid and (Right) Sample Maze Generation

To generate an $r \times c$ maze, we create a grid graph of size $r \times c$. The nodes in the grid represent the cells in the maze and can be indexed by $1, \ldots, (r \times c)$ in a row-major wise. Figure 2 (Left) shows a 5×5 grid with the cells indexed by $1, \ldots, 25$. Nevertheless, for the random walk technique, we mark a random node e.g. node 1 in Figure 2 (Left), as the starting node, and then walk around randomly on the graph. When selecting which node to visit from the current node, we randomly select an unvisited node from the neighbouring nodes that could be accessed. Note that not visiting any previously visited node will ensure no circular path. Nevertheless, for each move that we make from one node to a neighbouring node, we have an edge, and we consider the corresponding wall between the respective two cells is missing. As part of the random walk, if no movement is made between two neighbouring cells, then we assume the wall between the respective cells exists. Figure 2 (Right) shows a sample random walk and the resulting maze for the grid in Figure 2 (Left). For a generated maze like this, we select a node that is **farthest in the number of steps** in the random walk from the starting node as the finishing node. So the maze has a starting and a finishing node.

Using command-line arguments, your `MazeGenerator` program should take the number of rows and the number of columns as input for the size of the maze and a file name for output of the generated maze. Your program will give error messages for invalid inputs or if it cannot allocate

memory for the given maze size. The program should then randomly generate the maze for the given size, assuming a randomly selected node as the starting node and the farthest node in the number of steps in the random walk from the starting node as the finishing node. Once the maze is generated, your program should save the maze to a file with the given name in the following format. Your program will also display the same information on the screen.

`row_count:col_count:starting_node:finishing_node:cell_connectivity_list`

- `row_count` and `col_count`: the numbers of rows and columns in the maze.
- `starting_node` and `finishing_node`: the starting and the finishing nodes for the maze.
- `cell_connectivity_list`: a row-major list of node connectivity values.

Using the above format, the maze in Figure 1 (Right) or Figure 2 (Right) is as below:

`5:5:1:19:1223030112231122230210110`

4 Maze Solving

Your `MazeSolver` program will input a maze file name as a command line argument. It will then read the maze from the file which will be in the format discussed above. The program will then solve the maze separately using both the BFS and the DFS algorithms. While solving a maze using each algorithm, your program needs to keep track of the order in which it visits the cells, and the numbers of steps taken to go from the starting to the finishing nodes. It will also keep track of the time taken to solve the maze. These information will help compare the two algorithms.

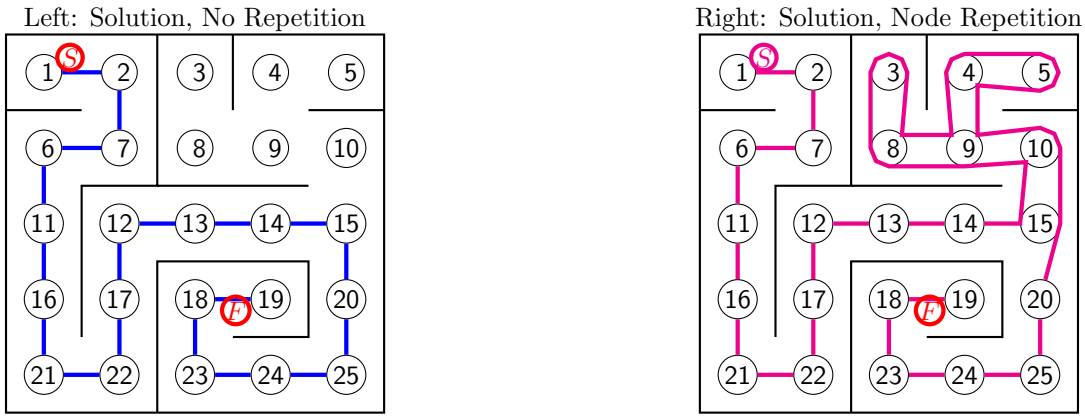

Figure 3: Maze Solutions with/without Node Repetitions

For each algorithm, the program should output the followings on the screen:

- The solution to the maze i.e. the path from the starting to the finishing cells. The solution in Figure 3 (Left) has no node repetition and its cell ordering is as below.

$$(1, 2, 7, 6, 11, 16, 21, 22, 17, 12, 13, 14, 15, 20, 25, 24, 23, 18, 19)$$

- The number of steps in the solution. For example, the number of steps is 18 (not 19) for the solution path in Figure 3 (Left). A step is a move from one cell to the next.
- The number of steps actually taken by the search. For example, as shown in Figure 3 (Right), for some reasons or even for node selection for exploration ordering, the search might take a wrong path and then come back to take another path. Alternatively, you can show the number of nodes that have been ever added to the open list or the closed list by your iterative implementation of the search algorithm. Clearly mention in the code what you show for this.

- The time in milliseconds taken to solve the maze while using an algorithm.

Sample output of the `MazeSolver` program while using either DFS or BFS algorithm.

```
BFS                                                    ▷ The algorithm used BFS or DFS
(1, 2, 7, 6, 11, 16, 21, 22, 17, 12, 13, 14, 15, 20, 25, 24, 23, 18, 19) ▷ The cell ordering in the solution
18                                                     ▷ the number of steps in the above solution
30                                                     ▷ if the search takes a wrong direction as in Figure 3 (Right)
                                                       ▷ or you might show the number of nodes ever added to the open or the closed list
99                                                     ▷ The number of milliseconds taken to solve the maze
```

The `MazeSolver` program will have another command line argument to denote the output filename. In the output file, the program will write the number of steps k followed by $k+1$ numbers for each solution but in the following format. So the output file could have multiple solutions, in fact two solutions at least: one for the BFS and the other for the DFS.

$$18 : (1, 2, 7, 6, 11, 16, 21, 22, 17, 12, 13, 14, 15, 20, 25, 24, 23, 18, 19)$$

5 Maze Verifier

Your `MazeVerifier` program will take the generated maze file name and a solution file name as command line arguements. The program will then read the maze from the maze file and verify its validity. The program will also read each solution from the solution file and verify whether the solution is a valid solution in the maze.

For maze verification, the program will display the following information on the screen.

1. The number of cells having all four walls.
2. The number of cells with none of the four walls.
3. Whether there is any circular path in the maze.
4. Whether all nodes can be visited from the starting nodes.

For a valid maze, the answers to the above four questions would be 0, 0, `No`, and `Yes` respectively. The first two queries can be responded just using loops to scan over the cells. The last two queries can be responded using a DFS visiting of all nodes in the maze.

For solution verification, the program will check whether the path from the starting cell is indeed through the valid pathways, not breaking any maze walls, not getting stuck at any deadend, and indeed reaching the finishing cell. The output will show on screen `valid` or `invalid`, and for an invalid solution, will display the partial path (sequence of nodes) up to the node travelled to.

For small values (suppose up to 10) of r and c, your `MazeVerifier` program will also display the sample maze and the solution on the screen using text characters `--` (two minus) or | (one bar) for each horizontal or vertical wall, two spaces for each cell, and `*` for the cells visited. You can play with the numbers of characters and the character symbols to make the maze look better. If possible for you, instead of the textual display, you can display the maze and the solution using graphics. The starting and the finishing nodes are also marked in the display. The display will help you quickly check whether your maze generation is correct. Figure 4 (Left) shows a sample display for the maze shown in Figure 2 (Right). Also, Figure 4 (Right) shows the sample solution in Figure 3 (Left).

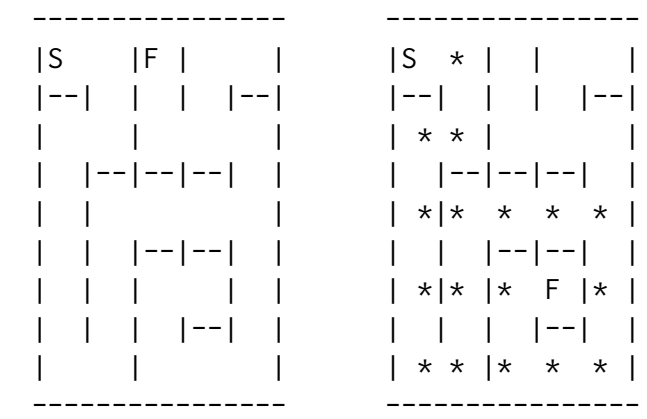

Figure 4: (Left) Sample Maze and (Right) Sample Solution

6 Solution Comparison

To compare your two techniques for solving a maze, please produce the following data. Generate 5 random mazes for each of the following three grid sizes 20×20, 20×50, and 100×100.

Generate a table that compares the numbers of steps in the solutions having no node repetition, the numbers of actual steps, and the running time, for each set of mazes, and for each method. You can use the following table format. Based on this data, write a very brief (50 words max) analysis of your results. That is, which method do you think performs better, and why?

Table 1: Sample Data Table

Maze Size	Solution Steps		ActualSteps or Nodes Generate/Evaluated		Time Millises	
Rows×Cols	BFS	DFS	BFS	DFS	BFS	DFS
20×20						
20×20						
20×20						
20×20						
20×20						
20×50						
20×50						
20×50						
20×50						
20×50						
100×100						
100×100						
100×100						
100×100						
100×100						

7 Submission Information

1. You must use Java to do your programming. Your code should contain
   - Classes named correctly as specified in each part.
   - Comments written to explain your code segments.
   - Use command line arguments for input. For example,
     ```
     java MazeGenerator 5 6 maze.dat
     java MazeSolver maze.dat solution.dat
     java MazeVerifier maze.dat solution.dat
     ```
   - Note the programs should handle any filename or extension
2. You must submit a readme file containing instructions on how to run your program.
3. You must submit text files with all test mazes used in your experiment and analysis.
4. You must submit a document containing the table and your analysis of your test data.
5. You must include a filled in Assessment Item Coversheet; otherwise, cannot mark.
6. You should zip all files and submit the assignment via the Assignment link in Canvas.

8 Assessment Criteria

- **20 Marks** for maze generation using random walk
  - **10 Marks** for correctly implementing the random walk technique
  - **5 Marks** for ensuring no inaccessible areas, open areas, and circular paths
  - **5 Marks** for correctly outputting to the maze to a file
- **30 Marks** for maze solving using BFS and DFS algorithms
  - **5 Marks** for correctly inputting from the maze files
  - **10 Marks** for implementing the generic iterative search algorithm
  - **5 Marks** for using the generic algorithm for BFS and DFS
  - **5 Marks** for solving the maze correctly with no node or cell repetition
  - **5 Marks** for correctly outputting solution information and writing to a file
- **15 Marks** for implementing the verifier program
  - **5 Marks** for maze verification
  - **5 Marks** for solution verification
  - **5 Marks** for maze and solution display
- **15 Marks** for comparing the BFS and DFS techniques
  - **5 Marks** for generating the test mazes
  - **5 Marks** for showing the comparison data
  - **5 Marks** for providing analysis/discussion
- **10 Marks** for using efficient data structures, explain your choices in comments.
- **5 Marks** for using better code organization, explain your choices in comments.
- **5 Marks** for useful commenting in the code.

END OF ASSIGNMENT