The University of Newcastle, Australia

COMP2230/COMP6230 Algorithms
Assignment, Total Mark: 100

Due: 11:59pm on Sunday**, 25 September 2022**

Submission: via the Assignment link in Canvas

Assignment Overview

This assignment can be done individually or in pairs and it involves performing two or four tasks, respectively. If you plan to do in pairs, then send an email by 11 September 11:59pm to the course coordinator (mahakim.newton@newcastle.edu.au) providing the two IDs and names with cc to the other member of the pair. If you do not send any email, it will be assumed that you will do the assignment individually. The assignment submission point will be configured after knowing whether you will do the assignment individually or in pair.

1. Write a program to **generate a random two-dimensional maze** of a requested size (rows and columns).
2. Write a program to implement the **Breadth First Search (BFS) technique to solve the generated maze.**
3. **Pairs only:** Write a program to implement the **Depth First Search (DFS) technique to solve the maze**.
4. **Pairs only: 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.

Maze Definition

Your aim is to create a simple maze that will have exactly one path from any point in the maze to any other point (as an example, see Figure 1). Therefore, your maze will not have **inaccessible areas** (cells without any missing walls around them), **open areas** (cells without walls), and **no circular** paths (loops).

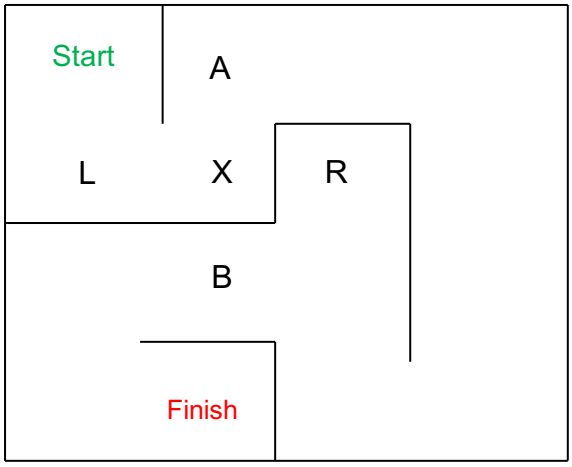

Figure 1 – sample maze

We can represent a maze as a two-dimensional array of cells. For each cell, we could have 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 within a closed area. Figure 1 shows a maze having 16 cells with dimensions 4×4. The starting and the finishing cells are marked. Notice that the top-left cell (Start) has walls above, left and right, but no wall below it, so we would be able to move from this cell to the cell L below in a valid move.

As we see from the above description, we can move from one cell to at most to the four neighbouring cells in four directions. To represent all possible combinations of moves, we would need 2^4=16 values. However, we can simply 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 closed, 1 for right only open, 2 for below only open, and 3 for both open. So, the top-left cell (Start) in Figure 1 can be represented by the value 2, since we can only move to the cell below, and not to the right. Moreover, the cell marked with an X in Figure 1 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 left cell L, we can see the information of L. Similarly, to find the information that we can move to cell X from the above cell A, we can see the information of A.

Generating a Random Maze

There are many techniques to generate mazes. However, 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.

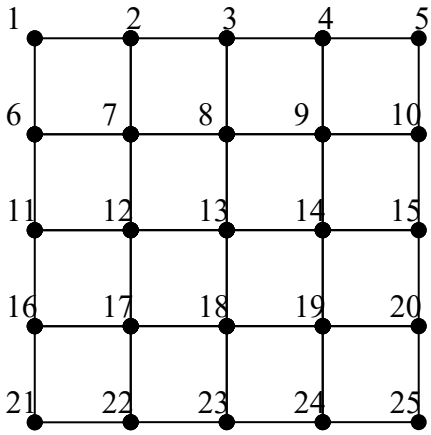

Figure 2 - sample 5x5 grid

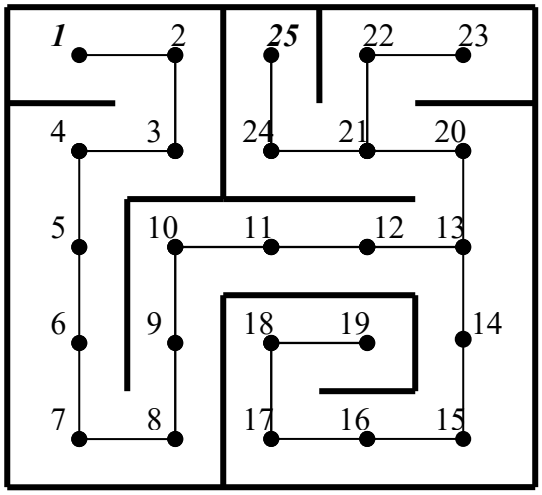

Figure 3 - sample maze generation

To generate an $n \times m$ maze, we create a grid graph of size $n \times m$. The nodes in the grid represent the cells in the maze and can be indexed by 1, …, $n \times m$ in a row-major wise. Figure 2 shows a 5×5 grid with the cells indexed by 1, …, 25. Nevertheless, for the random walk technique, we mark a random node (node 1 in Figure 2) 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 cyclic 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 3 shows a sample random walk and the resulting maze for the grid in Figure 2. However, in Figure 3, the node labels indicate the order in which the nodes were visited in the random walk. The node that is the last one 'visited' in the random walk is marked as the finishing node (labelled 25 in Figure 3 and is actually the cell 3 in Figure 2).

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. Your program will give error messages for invalid inputs. The program should then randomly generate the maze for the given size, assuming a randomly selected node as the starting node and the last visited node by the random walk as the finishing node.

Once the maze is generated, your program should save the maze to a file in the following format.

`n,m:start_node:finish_node:cell_openness_list`

where

- `start_node` and `finish_node` are not equal and are between 1 and $n \times m$
- `cell_openness_list` values are as per the following codes and are listed in the row-major order
  - 0: Both closed
  - 1: Right only open
  - 2: Down only open
  - 3: both open

Example file format for the sample maze shown in Figure 3 would be:

```
5,5:1:3:1223030112231122230210110
```

For small values (suppose up to 10) of *n* and *m*, your program will also display the sample maze on the screen using text characters `--` (two minus) or | (one bar) for each horizontal or vertical wall and two spaces for each cell. You can play with the numbers of characters and the character symbols to make the maze look better. The start and the finish nodes are also marked in the display. The display will help you quickly check whether your maze generation is correct. Figure 4 shows a sample display for the maze shown in Figure 3.

```
----------------
|S    |F |      |
|--|  |  |  |--|
|     |         |
|  |--|--|--|  |
|  |            |
|  |  |--|--|  |
|  |  |     |  |
|  |  |  |--|  |
|     |         |
----------------
```

Figure 4 Sample Maze

```
----------------
|S  * |F |      |
|--|  |  |  |--|
| * * |*  *  * |
|  |--|--|--|  |
| *|*  *  *  * |
|  |  |--|--|  |
| *|* |     |  |
|  |  |  |--|  |
| * * |         |
----------------
```

Figure 5 Sample Solution

Solving a Maze with BFS

Your second program `MazeSolverBFS` will input a maze from a file given as a command line argument in the format discussed above and solve it using a BFS algorithm. When solving a maze, your program needs to keep track of the order in which it visits the cells, and the numbers of steps taken to go from start to finish. It will also keep track of the time taken to solve the maze.

The program should output the followings on the screen:

- **The solution to the maze (path from start to finish)**. For example, looking at the path in Figure 6, the cell ordering is (1,2,7,6,11,16,21,22,17,12,13,14,15,10,9,8,3). **No node is repeated in the solution.**
- **The number of steps in the solution.** (For example, the number of steps is 16 for the path in Figure 6). 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 Figures 7, 8, and 9, for some reason, the search might take a wrong path and then come back to take another path.
- **The time taken to solve the maze, in milliseconds.**
- **Display the solution on the screen for small values (suppose up to 10) of *n* and *m*.** The display should be using text characters like the one in Figure 5 for the solution in Figure 6. You can make it nicer.

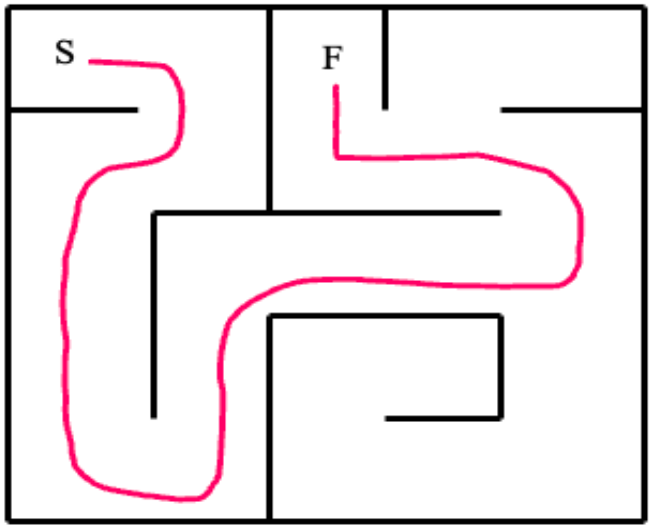

Figure 6 – solution, no repetition

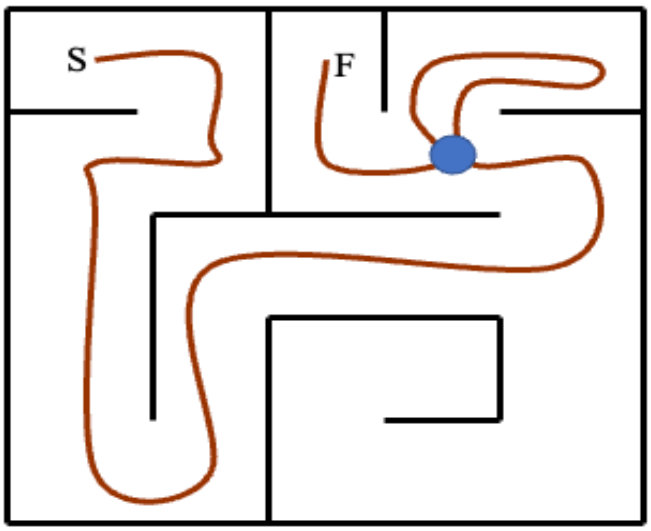

Figure 7 -solution, node repetition

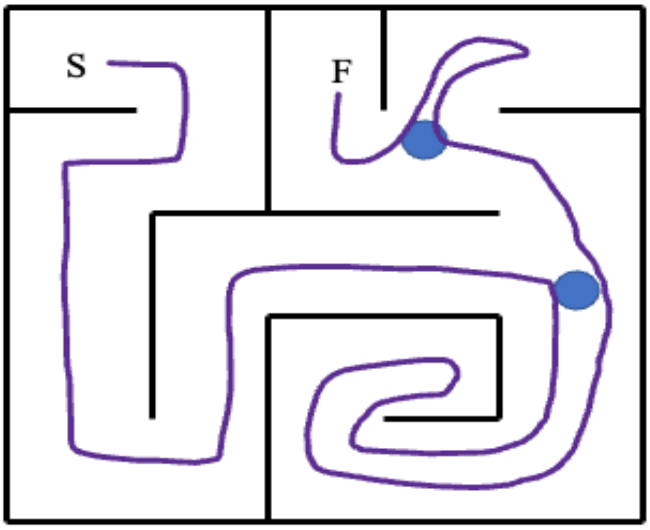

Figure 8 – solution, node repetition

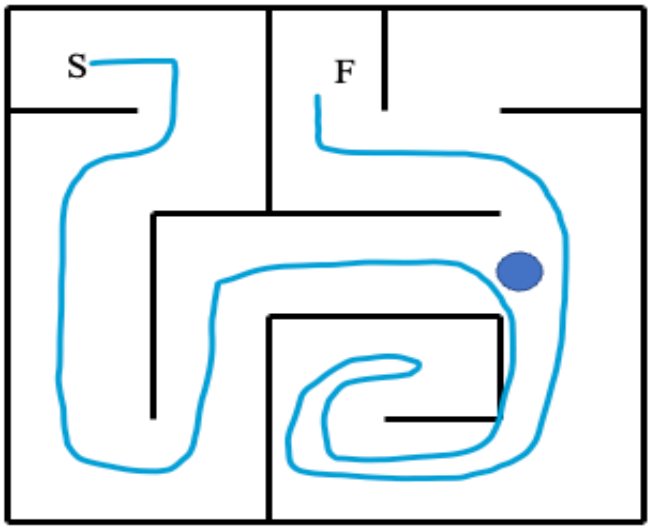

Figure 9 – solution, node repetition

Sample Output of MazeSolverBFS (or MazeSolverDFS for Pairs Only)

```
(1,2,7,6,11,16,21,22,17,12,13,14,15,10,9,8,3)
16
20    (if the search takes a wrong direction as in Figure 7)
99
Sample solution as shown in Figure 5, when m,n <= 10
```

Solving a Maze with DFS (Pairs Only)

If you are doing the assignment in pairs, your third program `MazeSolverDFS` will input a maze in the format discussed above and solve it using a DFS algorithm. During search, for the current node, you might have several nodes that you could visit next. In that case, you can use a pre-decided preference order of directions. Alternatively, you can dynamically prefer the direction of the finishing node from the current node. For example, from the nodes shown with blue dots in Figure 7, 8, and 9, one could avoid backtracking if the choices could have been preferred based on which direction the finishing node is from those nodes. The direction could be decided by considering the differences (both signs and magnitudes) in the rows and the columns of the nodes. If you cannot work out the dynamic preference,

implement the pre-decided preference order. Nevertheless, your program should produce the same 5 types of output as discussed for the BFS program before.

Comparing the Two Techniques (Pairs Only)

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 20x20, 20x50, 100x100.

For each maze you should also include `maze.dat` file containing the description of the maze in the format `n,m:start_node:finish_node:cell_openness_list`.

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. 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?

Sample table

Maze Size	BFS Solution Steps	DFS Solution Steps	DFS Actual Steps	BFS Actual Steps	DFS Time	BFS Time
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						

Submission

You must use **Java** to do your programming. Your submission should contain the code with:

- classes named correctly as specified in each part
- Use command line arguments for input e.g.
  - `java MazeGenerator 5 6 maze.dat`
  - `java MazeSolverBFS maze.dat`
  - `java MazeSolverDFS maze.dat (for Pair Only)`

  Note the programs should handle any filename or extension
- a readme file containing instructions on how to run your program
- for pairs assignment, a text file with all test mazes used in your analysis
- for pairs assignment, a document containing the table and your analysis of your test data
- a filled in **Assessment Item Coversheet**. (We cannot mark your submission if there is no coversheet).

You should zip all files and submit the assignment via the **Assignment** link in **Canvas.**

Assessment Criteria

The assessment criteria will be as follows:

Individual Assignment:

1. 40 marks for random maze generation.
    - 20 marks for correctly implementing the random walk technique
    - 10 marks for ensuring no inaccessible areas, open areas, and circular paths
    - 5 marks for correctly outputting to the maze files
    - 5 marks for displaying the maze on the screen
2. 35 marks for implementing the BFS solver.
    - 5 marks for correctly inputting from the maze files
    - 25 marks for solving the maze correctly with no node or cell repetition
    - 5 marks for displaying the solved maze on the screen
3. 10 marks for using efficient data structures
4. 10 marks for code organization and efficiency
5. 5 marks for useful commenting in the code

Pair-wise Assignment:

1. 20 marks for random maze generation.
    - 10 marks for correctly implementing the random walk technique
    - 4 marks for ensuring no inaccessible areas, open areas, and circular paths
    - 3 marks for correctly outputting to the maze files
    - 3 marks for displaying the maze on the screen
2. 20 marks for implementing the BFS solver
    - 3 marks for correctly inputting from the maze files
    - 14 marks for solving the maze correctly with no node or cell repetition
    - 3 marks for displaying the solved maze on the screen
3. 20 marks for implementing the DFS solver.
    - 6 marks for dynamic preference of choices
    - 14 marks for solving the maze correctly with no node or cell repetition
4. 15 marks for comparing the two techniques
    - 5 marks for generating the test mazes
    - 5 marks for comparison data
    - 5 marks for analysis/discussion
5. 10 marks for using efficient data structures
6. 10 marks for code organization and efficiency
7. 5 marks for useful commenting in the code