Team Members: Elise Hoang, Vanessa Tang

Project 3: Report

In the third project, the task assigned was to write a program that automatically generates and solves mazes. Each time the program is run, it should generate and print a randomized maze and its respective solutions derived from depth and breadth-first search. In order to generate a random maze, we needed to create a random number generator, generate the maze grid, print it, solve using BFS and DFS, and label the steps in order to reach the maze's exit. The proposed solution would be to first implement a simple maze generation algorithm that uses depth first search, in order to create a "perfect" maze that is solvable.

In the Cell class of our project, we created a Cell object that would represent each room of the maze. We had boolean variables for north, east, south, west walls, and to show whether the cell had been visited yet. We had integer variables for the column, row, and label of the cell. We also had a Cell parent to show the parent of the Cell for our DFS and BFS. The methods in this class were all getters and setters for the above variables.

In our Maze class, the first step would be to generate and print out a random maze that our search algorithms will then proceed to solve. To generate the maze, we first convert a grid of rooms into an adjacency matrix, then label all the cells in the grid numerically. Then, we initialize an adjacency matrix of the maze's row and column size. When finding all neighbors of the current cell with all walls intact, we needed to look at the next cell or its respective neighbors north, south, east, and west since cells have four directions. After implementing the rest of the simple maze generation algorithm from the given pseudocode that uses the DFS algorithm, we can confirm that all of the cells have been visited.

To find a path through the maze with DFS, our approach involved a stack of Cells and the adjacency matrix we got from generating the maze. With the cell at column and row 0 as currentCell, we first add that to the stack. Then, within our while loop that runs until the exit cell is reached, we set a Cell as cellStack.pop. The neighbors of the cell are then generated with our helper method: getNeighbors(). In getNeighbors, the neighboring cells are added if they are within the bounds of the matrix and if it has not been visited by our DFS search yet. Any one of those unvisited vertices is pushed onto the stack. The new cell's visitedDFS is then turned to true and becomes the next currentCell. The process continues until either the Stack is empty or if the cell visited happens to be the ending cell.

In order to find a path from the starting to finishing room using BFS, our search algorithm iterates through the entire maze to find the correct path solution. To keep track of which cells to visit, we start by visiting our start node and adding it to the queue. Then, we visit the adjacent unvisited neighbors and add them to the queue and continue this process until the cell queue becomes empty or until the exit cell is reached. For the first maze output, when a cell gets visited, we will note and label numerically the order in which they are visited. At the same time, we keep track of the unvisited rooms by noting the neighbors that were not visited by looking at the respective adjacent cells. This allows us to generate the second maze output that returns the shortest solution path after identifying the room and wall openings.

In our Print class, we utilize BufferedWriter to model the mazes that are kept in our cell matrices. We have a method to print it for DFS and another one to print a maze for BFS. The first maze that is printed is the empty random maze that was generated by our Maze class. The second Maze that is printed is either the DFS or BFS path. We use if statements to see if the label on a cell is not -1. If it is not -1, we print out the cell with its label. The third mazes that

are printed are the mazes with the shortest path labeled with "#" signs. This utilizes a stack to print it in the correct order since we start from the exit cell. We get the parent of each cell until the starting cell is reached. The last lines printed are the length of the shortest path, the number of visited cells, and the [row, column] of the shortest path.

In order to test that our mazes were being solved correctly, we tested it for when the size is equal to 4, 5, and 6. We generated and set those mazes as the target files and compared the output files we got to those files. We also compared the DFS and BFS maze solution to each other as the shortest path between the two algorithms should be the same. Everything matched correctly.

With our RunTime class, we were able to get the running times of the solutions. To generate a maze of size 4, the average was 771139 nanoseconds. For size 5, it was 98505 nanoseconds. For size 6, it was 129907 nanoseconds. On average, the running time of our DFS algorithm was faster than BFS which makes sense as BFS explores more neighbors than DFS. This was also reflected in the number of visited cells for our solutions.

Maze size	4	5	6
DFS average	596348	56780	59596
BFS average	61675	66521	62813