Michael Wong
Partner: Hoang Nguyen
Professor Potika
CS 146 Sec. 02
November 14, 2018

Project 2 Maze Report

For project 2 Maze, me and my partner were tasked with randomly generating a maze and then solving it by using Breadth First Search and Depth First Search. The way that the maze was to be generated was to create a maze with all its wall intact and randomly select walls to break down. The top left would always be the starting location and the bottom right would always be the finishing location. In between each of the four walls, was a square space that was represented by a cell. The idea to generate the maze was to start at the first cell and visit each cell in the maze until all of them were visited. The requirements were that there can't be any open areas or sections closed off and inaccessible. As you visit each cell, walls would break between them and create a single solution maze. Once the maze has been generated, run BFS and DFS on the maze to yield a solution to the maze.

In our program there are two distinct terms to it that should be clarified: maze and graph. The first part is the maze. The maze is the physical and visual representation of the maze using ASCII symbols in the form of strings. The graph is the data structure that is being used to solve the maze with BFS and DFS. Complete with vertices and edges, the graph is the data representation of the maze produced with strings. Both works together to generate, produce, and solve the mazes for this project.

This project was difficult for both of us, but eventually we were able to complete the problem presented to us with just a few inconsistencies to what was expected. One of the easier parts of the project that we completed was the creation of the starter maze. The maze was created with a simple 2D String array to represent the rows and columns of the maze. The four ASCII characters that we used were "+", "-", "|" and " ". Spaces represented cells, pluses represented corners and the "-" and "|"s represented the walls. A pattern we discovered was that odd numbered rows would have "|" walls and even numbered rows would have "-" walls. Also "|" walls only appear with even numbered columns and "+" occurs with both even rows and columns. With this fact we used a double for loop and created the default maze.

Another part that wasn't crystal clear to us from the beginning, but fairly simple to implement was the Vertex class. Like the previous project, a class to represent nodes or vertices was required in order to create the graphs to solve the maze. From the start we knew that the vertices needed some sort of identifier, so we gave it an int index starting from 0 to the total number of

cells. They incremented from left to right, top to bottom. Later, we decided to give the vertices a row and column coordinate so that we could pinpoint where in the maze the vertex laid. This will come into play once I start discussing how we broke the walls. Boolean expression were created to flag whether the vertex was visited or not and if it has broken wall(s) around it. The vertex class has two arraylist neighbors and nearby. Neighbors represents connected vertices and nearby represent unconnected vertices that can be connected if a wall was broken down.

After that it was a rough ride to the end. The immediate problem that we've faced was creating a graph. Before we dealt with linked list were they were linear and could not have more than two neighbors. With a graph however we were stump on how to assign the neighbors and nearby vertices. In the previous project there was only two directions for neighbors: previous and next. For this project though, we had to deal with potentially four neighbors and nearby vertices. We could have assigned getter and setters for each direction and situation, but we felt that that was too much and would complicate things. We've decided to use arraylists and simply add neighboring and nearby vertices to a vertex. We would then just extract the vertices with the get method from those list when running the maze generator and solving algorithms. Setting nearby vertices was done in the constructor and adding neighbors was done when walls broke. We also created an adjacency list, but never ended up using it.

The actual breaking of the walls was not complicated, but creating a way to visit every vertex and choosing the walls to break to create a single solution path was extremely strenuous. The process of breaking the wall consisted of two methods, breakWalls() and wallBreaker(). The breakWalls method does the visual wall breaking by replacing the wall with a space character. It then adds the two vertices that were separated by the wall as neighbors of each other. The way that the wall was chosen was by getting the vertex's coordinates and either plus/minus its row/column to get the wall's coordinate. After getting the coordinate, the wall was replaced with a space.

The wallBreaker method handles the random selection, prevention of cycles and enclosed areas, and makes sure every vertex has been visited with a path to get to all of them. Pseudocode was provided to implement this search and destroy algorithm to break the walls. The pseudocode made sense to me, but we were having a hard time implementing the code. The trouble lied in finding and choosing the random neighbor. Originally we didn't have an arraylist of nearby vertices that could be connected. After we added that arraylist to the Vertex class, it made it easier to obtain a random vertex nearby. The nearby vertices were adding in the order of above, below, left and right to start with. We thought it was fine until we tested it and found out the wrong walls were being broken. After switching the order around we finally got the right sequence of adding to the list. The order followed as below, right, above, and left. We then moved on to breaking the walls. Corresponding the chosen vertex with the current vertex was

troubling until we noticed a pattern regarding the indexes of vertices. Obviously, vertices to the left and right are +1 or -1 index of the current, but above and below indexes were not immediately clear. We found out that for the above or below vertices, their index was either +the number of vertices in the row or -the number of vertices in the row. With this fact, we were able to distinguish which wall needed to be broken and then managed the status of the two vertices. The statuses of the vertices was also a hurdle, but we were able to straighten it out as we moved on with the project.

Lastly the implementation of the BFS and DFS algorithms were a struggle to start. We understood the process and how the two algorithms worked, but implementing it in code was a tough task. We honestly didn't even know where to start on it or how it was suppose to look like. But after we reviewed the slides and searched around the web, we were finally able to create working dfs and bfs behavioural algorithms. Now the problem was that the solutions that they were outputting did not match our expected output. There seemed to be no consistently with the searches with some paths moving up first then down and others moving down first then up. We looked at the ordering of the vertex's neighbors and found the list to be in no particular order, so we gave it an order, similar to the nearby vertices. We sorted the list of neighbors using Collections.sort() in java that sorted the vertices in ascending order. We then ran DFS and the expected output was reached. Thus we concluded that DFS should push neighboring vertices onto the stack starting with the first neighbor whereas BFS has to add to the queue starting with the last neighbor.

The one concept that we were not entirely able to grasp was the shortest path algorithm. We understood that the shortest path can should be found by using BFS, but we obtained it through DFS. Our DFS solution was already finding the shortest path, so we piggybacked on it to generate the path with hashtags. To connect the hashtags, we created a whole new method that checks for spaces that need hashtags to create a single path. I don't think this was the intended or appropriate way of finding the shortest path, but we tried using BFS, but could not produce the path. We were setting previous vertices and tried creating an arraylist to hold the vertices that led to the shortest path, but our efforts did not produce results. So we were able to find and output the shortest path, but not with the correct method.

In conclusion, my partner and I were able to achieve the goal of the project, but somethings were not completed to their full objective. Mazes were randomly generated and solved with DFS and BFS. Our most prominent issue was finding the shortest path between two vertices using BFS. I think we worked together well and made the most of our time working together and individually on the project. I wish the best of luck to my partner, Hoang, and hope to work together again perhaps in the near future.