Benjamin Garcia and Sean Quach

Professor Potika

CS 146 / Section 2

11/24/2019

Project 3 Report

**Introduction**

For Project 3, there were three main goals to accomplish: generate a maze, model a maze, and solve a maze. To generate a maze, a 2D matrix of cells had to be made and then randomly had walls torn down to create a perfect maze. The maze had to be modelled by giving each cell its respective neighbors. Finally, the maze was solved using either Breadth First Search or Depth First Search.

**Background/Review**

For the project, a perfect maze must be generated so that there are two openings, one for the entrance and one for the exit. Mazes must be randomly generated so that different solutions can be found for different mazes. For the solutions, they must be solved using Breadth First Search or Depth First Search.

**Project Specification**

For the implementation of the maze design, our team decided to use classes of Cells and Mazes. Cells contained variables to retrieve the coordinates, neighbors, and walls. In addition, other variables, such as the color, starting time, finishing time, and distance, had to be added to the Cell class. Mazes were made up of Cells and contained a 2D matrix of these Cells. When the cells are initialized, the neighbors and the walls of the cells were also initialized. The maze was generated by starting at (0,0) and knocking down random walls of randomly selected neighbors that had all their walls still intact. This was done until the end of the maze was reached.

**Problem Analysis and Solution Design**

The first problem to solve was how to generate the perfect maze. When checking which neighbor to randomly select, we had to make sure that the cell exists by checking that it was not null. In addition, we had to make sure that there were openings for the entrance and exit by making the walls false for the entrance and exit. For the Breadth First Search, one of the problems was how to get the shortest path and show it through ASCII. The solution to this was to check the parents of the cells when the end was reached and add the parent if it was not a dead end. After checking through all parents and getting to the start, the list of cells traversed was parsed and shown through “#” in the solution. The distance was found by the number of cells traversed in the maze.