Curtis Davis
COT 4400 – Fall 2019
Project 3 Report

1. *What type of graph would you use to model the problem input (detailed in the Section 3.1), and how would you construct this graph? (I.e., what do the vertices, edges, etc., correspond to?) Be specific here; we discussed a number of different types of graphs in class.*

A: For this graph, you I would want to use an undirected and unweighted graph. Unweighted because we are not concerned with the distance for this matter, because it each move requires the same distance, 1 move.

This graph will be represented as an adjacency list using the Boost library for C++. At first, the input takes in the size of the graph and runs loops based on that and future input to construct vertices of 3D coordinates of sorts to represent the cells of the maze. The vertices are also kept with Boolean values which tell the program which way the spider is allowed to go from its current cell. These Boolean values are checked, and edges are added to create the pathway for the spider. The pathway edges run either North, East, South, West, Up, or Down depending on the Boolean values.

To elaborate, the vertices correspond to the cells of the 3D maze the spider is put into, while the edges represent how the spider can move to and from cells. Walls aren't represented in the graph, except as an absence of edges. Since this isn't a directed maze, the spider will be able to move freely, not having to worry about not being able to traverse a certain way.

2. *What algorithm will you use to solve the problem? Be sure to describe not just the general algorithm you will use, but how you will identify the sequence of moves the spider must take in order to reach the goal.*

A: We will traverse the graph with DFS after building an undirected and unweighted graph. Boost has the DFS function built in, but it's important to understand how it works in its implementation. In this program, a 'DFS Visitor' starts at a given start vertex from the input file and performs its recursive DFS discovery from there. Depth_first_search is called using a dfs_visitor which records predecessors on a tree edge.

The graph is already modeled to ensure that it can move according to the walls and pathways, so this algorithm of DFS is just iterating through all connected vertices and marking each one's predecessor.

As stated, the visitor explores the graph, vertices' predecessors are recorded when the visitor crosses a tree edge. This is implemented so we can see the path DFS took. Once DFS hits a dead end, it backtracks until it comes across another path it hasn't took, and recursively does the same 'predecessor recording' method for those branches as it had before. This DFS method finds how all the cells are connected for this traversal path.

After the algorithm finishes, we can then traverse the path backwards in a way starting at the goal cell. This allows us to see the reversed path the traversal found, from the goal cell back to the starting cell, by looking at the predecessor of each cell and doing the same for each new predecessor until our new predecessor is our starting cell. This path calculates the directions of the traversal and puts it into a vector. Afterwards, my program iterates backwards over said 'reversed vector' and prints out the correctly oriented path from the starting cell to the goal cell.

Thus, the Spider Maze is solved.