Modeling The Problem Project Three

Joseph Cox (U00594912) 11/28/18

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.

   To solve the problem I will use an unweighted graph, as I will not calculate the distance between each vertex. I will use the vertex value to determine North movement, South movement, East movement, and West movement. From a a particular vertex to get the neighbors of each vertex as a set of pointers to another location in the array. From this I will build an adjacency list. To describe all possible destinations of traveling North, South, East, or West from a particular vertex. I will transform the 2d matrix into a 1D array and use a combination of static and and nonstatic instace variables to solve the problem A cdoe sample is provided below for the neighbors and data members:

   This is the class that I will load my vaibales into to solve the problem: North, South, East, and West are pointers that point to a location in the 1D array.

```
class Node{
    /* make everything public so that everything is accessable*/
    public int row; // the row the vertex is located
    public int col; // the column the vertex is located in
    public int value; // the value that the vertex can bounce
    public Node north; // for the adjacency list, default it will be set
to NULL
    public Node south;
    public Node east;
    public Node west;

    // pass some variables as static so that it is shared accross all
instances of the class
    public static int [][] grid;
    public static Node [] adjacentcyList; // Stats as a 1D array of n*n
values use pointers N,S,E,W to point to location for neighbors
    public static int row_size; // col_row size of the data should not
change accrross the class
    public static int col_size;
    public boolean visited; // mark a particular node as visited or not
visited for the search
    public static Stack <String> dfs_stack;// String stack to hold the
direction of travel
    public static String exitPath;
```

This is the the get neighbors function will link North, South, East, and West to the proper location in the 1D array.

```
 public void getNeighbors(){
        // North
        if(this.row - this.value>=0){
            this.north = adjacentcyList[((this.row - 
this.value)*col_size)+col];
        }
        // South
        if(this.row + value<row_size){
            this.south = 
adjacentcyList[((this.row+this.value)*row_size)+col];
        }
        // East
        if(this.col + this.value<col_size){
            this.east = adjacentcyList[((this.col+this.value)+(this.row)*
(col_size))];
        }
        // West
        if(this.col - this.value>=0){
            this.west = adjacentcyList[((this.row*col_size)+(this.col - 
this.value))];
        }
    }
```

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 Jim must take in order to reach the goal.

I will use a depth first search to locate the goal of the maze and the direction of travel as described below:

```
public void depthFirstSearch(Node vertex){
        vertex.visited = true; // Mark the vertex as visited
        // if the goal was gound stop the goal is always at the bottom 
right of the matrix
        // This is the column size and the row size - 1
        if(vertex.row == row_size-1 && vertex.col == col_size-1){
              while(!dfs_stack.isEmpty()){
                    exitPath +=dfs_stack.peek();
                    dfs_stack.pop();
            }
        }
        // check going north
        if(vertex.north!=null && vertex.north.visited==false){
            dfs_stack.push("N"); // put the direction of travel into the
stack
            vertex.depthFirstSearch(vertex.north);
        }
        // check going south
        if(vertex.south!=null && vertex.south.visited==false){
            dfs_stack.push("S");// put the direction of travel into the
stack
            vertex.depthFirstSearch(vertex.south);
        }
        // check going west
        if(vertex.west !=null && vertex.west.visited ==false){
            dfs_stack.push("W");// put the direction of travel into the
stack
            vertex.depthFirstSearch(vertex.west);
        }
        // check going east
        if(vertex.east !=null && vertex.east.visited==false){
            dfs_stack.push("E");// put the direction of travel into the
stack
            vertex.depthFirstSearch(vertex.east);
        }
        // if neither of these pop the direction from the stack we only
want the direct path
        if (!dfs_stack.isEmpty()){
            dfs_stack.pop();
        }
        return;
    }
```