David Hatcher

U74842093

Project 3

1. Simple, directed, weighted, and labelled graph, store it in an adjacency list.

2. The vertices represent each square on the board and contain labels to determine whether Alice should grow or shrink and be named for their place on the board (i.e., v1,1 v1,2 ... v7,6 v7,7). The edges will represent the other vertices that this point can connect to, each edge will have a weight that is the distance between the two vertices.

3. This algorithm will go through the list of squares and assign the speed change value to the corresponding vertex in the graph, 1 if it is a grow square, -1 if it is a shrink square, and 0 if it is neither. It will then add an edge between the vertex and every vertex that Alice could possibly travel to based on the directions from the squares input. As such if the graph has north it will add an edge between that vertex and every vertex directly north, if it has south, it will add an edge between that vertex and every vertex directly south, it will do the same for east and west, with vertices to the east and west respectively. It will add do the same thing for NW, NE, SW, SE but will only add the vertices that are in a straight line in those directions. As each these edges will have a distance value you to them as well, I will be using a version of addEdge(u, v) that also accept a distance between the u and v, addEdge(u, v, d).

```
Input: r, numbers of rows of the maze
Input: c, number of columns of the maze
Input: squares, 2D array where each index contains a square, which contains the arrows(i.e., n, s, e, w, ne, se, nw, sw) it has and if Alice should grow or shrink
Algorithm: BuildMaze

adjList = Graph(r*c);   //This list will have the key of (r, c) where r represents the row of the square and c represents the column of the square and the value will be a struct, that contains the speed change value, speedChange, and it's adjacencies, with the distance to them. Thus, addEdge will take three parameters, u, v, dist
for i = 1 to r do
    for j = 1 to c do
        if squares[i, j] has grow:
            adjList(r, c).speedChange = 1;
        else if squares[i, j] has shrink:
            adjList(r, c).speedChange = -1;
        else
            adjList(r, c).speedChange = 0;
        end if
        for each arrow in list of arrows:
            if arrow is n:
                for k = i - 1 to 1 do
```

```
      adjList.AddEdge((i, j), (k, j), i-k)
    end
  else if arrow is s:
    for k = i + 1 to r do
      adjList.AddEdge((i, j), (k, j), k-i)
    end
  else if arrow is e:
    for k = j + 1 to c do
      adjList.AddEdge((i, j), (i, k), k-j)
    end
  else if arrow is w:
    for k = j - 1 to 1 do
      adjList.AddEdge((i, j), (i, k), j-k)
    end
  else if arrow is ne:
    dist = 1, x = i-1, y = j+1;
    while x != 0 and y != c do
      adjList.AddEdge((i, j), (x, y), dist)
      x--, y++, dist++;
    end while
  else if arrow is nw:
    dist = 1, x = i-1, y = j-1;
    while x != 0 and y != 0 do
      adjList.AddEdge((i, j), (x, y), dist)
      x--, y--, dist++;
    end while
  else if arrow is sw:
    dist = 1, x = i+1, y = j-1;
    while x != r and y != 0 do
      adjList.AddEdge((i, j), (x, y), dist)
      x++, y--, dist++;
    end while
  else if arrow is se:
    dist = 1, x = i+1, y = j+1;
    while x != r and y != c do
      adjList.AddEdge((i, j), (x, y), dist)
      x++, y++, dist++;
    end while
  end if
```

4. This algorithm will create a path, which will be a list of vertices that need to be taken to reach the goal square, it takes the input of a start vertex, the goal vertex, and outputs this path. To do this it will start by adjusting the speed by the speed change value its current vertex, then check if the current vertex is the goal, if it is it will return the path that it has built. After this check if the current speed is 0, if that is the case it will remove the last added vertex from the path then

return the path. Otherwise, it will loop through all the neighbors of the current vertex, for each of the neighbors that have a distance equal to the current speed and that have not already been added to the path with the same speed, this is done to avoid infinite loops that can occur when the current speed allows the algorithm to visit the same nodes repeatedly, it will then recursively call itself on with the following parameters (sr, sc) = current vertex, (gr, gc) = goal vertex, s = current speed, and Path = the current path but with the current vertex added. After each recursion it will check if the goal vertex is in the path, if it is it will return the Path, this will cascade to each of the previous recursive calls and return the final path from the starting vertex to the goal vertex.

```
Input: G, Graph from BuildMaze
Input: (sr, sc), Start location, vertex on the graph
Input: (gr, gc), Goal Location, vertex on the graph
Output: Path to reach the goal position from the start position

goalPath = list();
goalPath = DepthFirstMazeTraversal((sr, sc),(gr, gc), 1, goalPath);
return goalPath;

    DepthFirstMazeTraversal
    Input: (gr, gc), goal location
    Input: (c1, c2), current location
    Input: s, integer representing player speed
    Input: Path, a list of the vertices in the current path, with the speed at which they
were reached in the form (u, s)
    Output: The path from the initial location to the goal location

    s += (c1, c2).speedChange;
    if (gr, gc) == (c1, c2)
        return Path;
    else if s == 0:
        remove last added vertex from Path;
        return Path;
    else:
        for each vertex in N((c1, c2)) as u:
            if distance to neighbor is s and (u, s) is not in Path:
                currentPath = Path;
                currentPath.append(u, s);
                Path = DepthFirstMazeTraversal((gr, gc), u, s, currentPath);
            endif
            if (gc, gr) in Path:
                return Path;
        end for
    end if
```