Amer Abdelaziz

April 16, 2018

COT 4400: Analysis of Algorithms

Professor Hendrix

Project #3: Graph Algorithms

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.

    The layout of the problem asks us to essentially be able to get from the start of the puzzle (Top left) to the bottom right. The route will consist of the red arrow at the top which points to two blue arrows, you choose one of those and move on alternatively.

    The most realistic solution for this model would be using a Depth-First Search traversal to be able to visit the neighbor of the current node and keep visiting till node reaches a dead end or reaches a solution. The construction of the graph was consisted of a two-dimensional adjacency matrix that's composed of a double pointer array of nodes. With each node containing a color, next direction to traverse, an i and j point that determine the position of the current node, and a finish bool that determines whether the Node has reached the endpoint. The construction of the graph was done by running two nested for loops and assigning a pointer row array to a column in the for loop. The vertices represent nodes in which Apollo travels, the edges represent a connection between two nodes. In respect to Apollo's path, the connected edges represent the path in which he travels. The nodes are pushed onto the stack to yield a final path to the destination from the start point to the endpoint.

2. What algorithm will you use to solve the problem? You should describe your algorithm in pseudocode. Be sure to describe not just the general algorithm you will use, but how you will identify the sequence of moves Apollo must take in order to reach the goal. Your algorithm must be correct, and it must have the minimum possible complexity.

The algorithm I used was a DFS algorithm; the DFS algorithm is constructed such that we visit a node and determine whether it's the endpoint, if it's the endpoint then our traversal is complete, else we need to visit the neighboring node and repeat the same process till we've reached the endpoint. The decisions onto which neighbor to visit next is determined by the direction of the current node. In the constructed DFS algorithm, we model the decision by using a switch statement of the current direction, and the directional values are either N, S, E, W, NE, NW, SE, or SW. After processing the direction, we modify the values of the coordinates and push the node into our stack to keep track of the nodes we've visited. A graph class was also created that contained the blueprint for a graph, which contained the x and y coordinate capacity of a graph and a n x n node graph. Apollo's moves are determined by the direction given in the input file. The sequence of moves is determined by the previous node; the previous node's direction is an indication of where Apollo should go next. The Abstract Data Type that was used in the algorithm was a stack; every move made by Apollo was pushed onto the stack. If the next move yields to a wall, then we pop the top of the stack. The algorithm does this for all possible directions Apollo could go, moreover, it checks whether colors of the nodes are also equivalent as well as if it had been visited before. The DFS algorithm

Pseudo Code:

```
Void DFS(Graph g)
{
        Stack <Node> stack;
        Stack <int> num;
        int count, i, j = 0;
        Node current = g.start, temp;
        Stack.push(current);
        Num.push(0);
        While(current != g.finish)
        {
                If stack.empty
                        Return “Maze can’t be solved”
                Current = stack.top;
                Curr = Current.getdir();
                Color = current.getcolor();
                Switch(curr)
                {
                        Case ‘N’ :
                        If i = 0 then pop both stacks and continue (Varies for dif dir)
```

```
                    i - -; (could be j for other directions)

                    count++;

                    temp = current;

        (while loop varies by direction but same logic)

                    while(temp.color == current.color || temp.isvisited && i !=1)

                    {       i--;

                            Count++;

                            If(I !=-1)

                            {

                                    Temp = current

                            }

                    }

                    If(I == -1)

                            { pop both stacks and continue}

                    Set finish to true for Node at I and j

                    Push the count and temp


            }


            Stack <Node> rev;
        While( stack is not empty)
        {
                Node cur = stack.top;
                Push cur to rev
                And pop stack
        }
        Stack<int> revnum;
        While(num is not empty)
        {
                X = num.top
                Num.pop
                Push x onto revnum
        }
        Int c = 0;
        While( rev is not empty)
        {
                If c = 0 then pop revnum, increment c and continue
                If rev size = 1 then pop rev


        }
}
```

```
}
```