Name:Jinhao Chen
Date: 04/26/2016
AlgoProject3

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.

In this project, I would like to come up with directed, finite, simple connected graph. Because each point will directed to other point, I can think the graph that a vertex direct it to other vertex, however, the other vertex also might directed it back to previous vertex. For creating the graph, the first line two numbers in input file are total row and total column, and the rest I assume those number are all vertexes. Then each vertex might have four direction can go, which is east, west, south, and north. And then check the direction valid or not, if the direction valid and value of each point will be the edge when the vertex goes to next vertex. And then in my code, I create the adjacency list to create a vector and push all the vertex neighbors to the adjacency list vector.

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.

In my project, I am using DFS to solve the problem. First I create the adjacency list to hold the vector, and then push all the neighbors to this vector. By created the adjacency list, it is clear to see how many neighbors each vertex. Because each vertex has 4 directions to go, so I assume each vertex each direction will be in a struct call vertex. And the struct vertex will hold the values, row, and column when checking the conditions statement. In addition I also create a 2D vertex for my adjacency list. After adjacency list being created, I am going to call DFS to get the result. For this project, each vertex will check south, east, north, and west in this order. Before check direction of vertex, I will assign all vertexes as visited. By this way will make DFS works perfectly. Because when goes to each adjacency list, it will check the size of this adjacency list and whether the vertex is visited or not, if not it will push this vertex and break. If the vertex is visited, it will go to other vertexes on this adjacency list. If all the vertexes in the same vector of the adjacency list, it will pop it, go back to the previous vertex. And repeat these steps again until it reach the exit point. I also test two input files, and for small one I got the same output with professor, and for 7*7 input, I get 2 different in the middle of output sequence. The reason why I will have difference output, because it might be when I check the neighbor and the order is different from professor. However, I check the output of my sequence is also getting same exit vertex.

This following is code how I run my DFS in adjacency list.

```
bool *visited;
    visited = new bool[p*q];
    for (int i = 0; i < p*q; i++)
    {
        visited[i] = false;                    // set up all vertex are not visited
    }

    stack <int> s;                             // create a stack to get path of DFS
    int start = 0;
    int exit = p*q - 1;                        // when reach each vertex, set statues as visited
    visited[start] = true;
    while (start != exit)                                          //
    {
    if (mylist.at(start).size() ==      // when the size == 0, it means that no neighbor of this vertex,
so
        {                                          // we need pop the vector and update the start
            s.pop();
            start = s.top();
        }
        for (int i = 0; i < mylist.at(start).size(); i++)        // check if all neighbor are visited or not
        {
            if (visited[mylist.at(start).at(i)] == false)        // if not visited, mark as visited and
goes to next vertex
            {
                start = mylist.at(start).at(i);
                visited[start] = true;
                s.push(start);
                break;
            }
            if (i == mylist.at(start).size() - 1){             // if all visited, pop and update start again


                if (visited[mylist.at(start).at(mylist.at(start).size() - 1)] == true)
                {
                    s.pop();
                    start = s.top();
                    break;
                }
            }
        }
    }
```