Report

by Tuan Huynh

Modeling the problem

1. The problem input gives you a verbose list of possible openings at each location in the maze. A location can be represented 3-Dimensionally using positive integers that represents the level, row, and column of that location in the maze. The location coordinates will represent the vertices of the graph. Each possible opening (north, east, south, west, up, down) will represent an edge between one location vertex to another. Since the edges themselves are only significant in identifying a connection between locations, they should be unlabeled and unweighted. Since we are going to be doing a lot of traversal, it makes sense to pick an adjacency list data structure over an adjacency matrix because calling neighbors is much faster with an adjacency list. The graph should contain vertices with the following properties:

Typedef: Location

```
1.    level, row, column
2.    neighbors = List {Locations...}
3.    DecodeNeighbors(Code)
```

Input: Code: 6-digit binary string representing all possible openings detailed in problem input

Algorithm: DecodeNeighbors

```
1.    if Code[1] = 1, add Location(level, row - 1, column) to neighbors
2.    if Code[2] = 1, add Location(level, row, column + 1) to neighbors
3.    if Code[3] = 1, add Location(level, row + 1, column) to neighbors
4.    if Code[4] = 1, add Location(level, row, column - 1) to neighbors
5.    if Code[5] = 1, add Location(level + 1, row, column) to neighbors
6.    if Code[6] = 1, add Location(level - 1, row - 1, column) to neighbors
```

2. To solve this problem, I would use a depth-first search algorithm to traverse from one location to another through an opening edge until I find the end location. I make sure the I don't traverse any vertex that I have already visited in order to prevent repeating states. Once I find the path, I break out of the recursion and return the path. I will also backtrack if there are no undiscovered openings from my current location. Some auxiliary data will include a 3D matrix that will hold the code describing the possible openings at a given location as well as a matrix of flags that marks will location has been visited

Input: start, end, l, r, c: start and end locations, dimensions of maze

Algorithm: Maze

```
1.    adjMap = Array[l][r][c]
2.    read in file and populate the adjMap with code
3.    solved = false
4.    return FindPath(start, end, )
```

Input: current, end, path: current and end location, current path as a string

Algorithm: FindPath

```
1.    mark current as visited
2.    if current = end
3.        solved = true
4.        return path
5.    output = path
6.    current.DecodeNext(adjMap[current.l][current.r][current.c])
7.    for each undiscovered loc in current.Neighbor
8.        output = Findpath(loc, end, path + edge)
9.        if solved = true
10.           return output
11.   return output
```