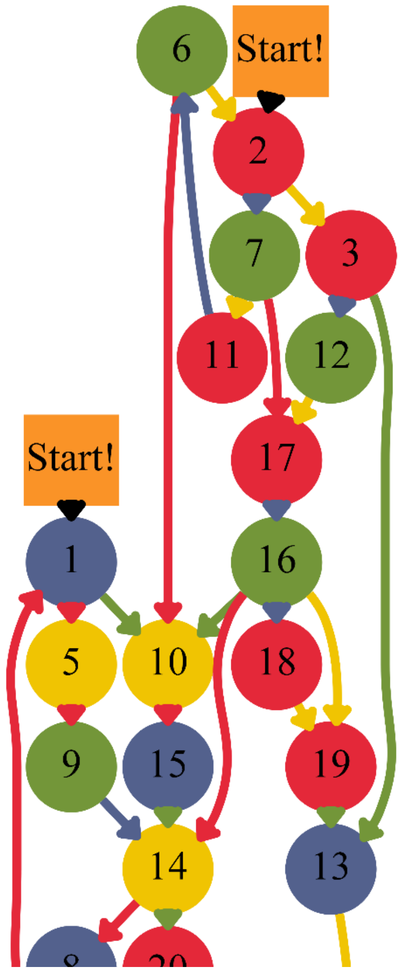
Jonathan Woolf

Maze Problem:

Problem Modeling:

The graph was loaded from the input file as a map/dictionary where each node value was mapped to a list containing a tuple of the adjacent node and the color connecting them.



The graph algorithm needed is a DFS.

DFS can be used to solve this algorithm with a slight modification of the original algorithm. Firstly, the algorithm will not be marking on the node the colors, parents, and times, but isntead on the board state. The board state is the position of Rocket and Lucky, so if Rocket was on 10 and Lucky was on 7 the board state would be (10,7). This is because the algorithm could visit the same node twice but the postion of Rocky and Lucky may be different. Secondly, the algorithm will not need to use a list of all board states as some states are unachievable, such as a state where Rocket is on node 2, which would be impossible. Instead, all graph states can be achieved from the starting state of (1,2), so there will be one primary call of the recursive function and all calls stem from that. In this way, the recursive DFS will move down each path, climbing up the recursive tree when the algorithm gets stuck or when it encounters a loop.

Jonathan Woolf

Code Submission:

Dictionary for color hex values:

```
colors = {
    'B': '#53618D',
    'G': '#739639',
    'R': '#E42839',
    'Y': '#F0C402',
    'O': '#FA9327',
    '0': '#000000',
    'W': '#FFFFFF'
}
```

Node class:

```
class Node:
    def __init__(self, num=-1, color=''):
        self.num = num
        self.color = color
        self.adj = set()
        self.start = False

    def add_adj(self, node, color):
        self.adj.add((node, color))
```

Loading file into Map/Dictionary:

```
def readFile():
    nodes = {}
    fin = open(sys.argv[1])

    n, m = fin.readline().split()

    i = 1
    for color in fin.readline().split():
        nodes[i] = Node(i, color)
        i += 1
    nodes[i] = Node(i, '0')

    x, y = fin.readline().split()
    nodes[int(x)].start = True
    nodes[int(y)].start = True

    for line in fin:
        try:
            i, j, color = line.split()
        except ValueError:
            break

        nodes[int(i)].add_adj(int(j), color)

    return int(x), int(y), nodes
```

Jonathan Woolf

Drawing the graph using graphviz:

```
def createImage(nodes):
    graph = graphviz.Digraph(comment='Graph!', format='png')
    starts = []
    for node in nodes:
        if nodes[node].start:
            starts.append(node)

        graph.node(str(nodes[node].num), style='filled',
color=colors[nodes[node].color], fontsize='128', width='4',
                   height='4')
        for edge in nodes[node].adj:
            graph.edge(str(nodes[node].num), str(nodes[edge[0]].num),
penwidth='28', color=colors[edge[1]],
                       arrowhead='open', arrowsize='4')

    for start in starts:
        graph.node('s' + str(nodes[start].num), label='Start!', shape='rect',
color=colors['O'], style='filled',
                   fontsize='128', width='4',
                   height='4')
        graph.edge('s' + str(nodes[start].num), str(nodes[start].num),
penwidth='28', color=colors['O'],
                   arrowhead='open', arrowsize='4')

    graph.render('graph')
```

DFS Class (contains next three functions):

```
class DFS:
    def __init__(self, s1, s2, nodes):
        self.s1 = s1
        self.s2 = s2
        self.nodes = nodes

        self.c = None
        self.p = None
        self.d = None
        self.time = None
        self.end = None
```

DFS initialize and call recursive:

```
def dfs(self):
    self.c = [["WHITE" for i in range(len(nodes))] for i in
range(len(nodes))]
    self.p = [[[] for i in range(len(nodes))] for i in range(len(nodes))]
    self.d = [[[] for i in range(len(nodes))] for i in range(len(nodes))]
    self.time = 0

    self.dfs_visit(s1, s2)
    return self.d, self.p, self.end
```

Jonathan Woolf

DFS recursive function:

```
def dfs_visit(self, l1, l2):
    if l1 == len(nodes) or l2 == len(nodes):
        self.end = [l1, l2]

    self.c[l1 - 1][l2 - 1] = "GREY"
    self.time += 1
    self.d[l1 - 1][l2 - 1].append(self.time)
    for adj in self.nodes[l1].adj:
        if adj[1] == self.nodes[l2].color and self.c[adj[0] - 1][l2 - 1] ==
"WHITE":
            self.p[adj[0] - 1][l2 - 1] = [l1, l2]
            self.dfs_visit(adj[0], l2)
    for adj in self.nodes[l2].adj:
        if adj[1] == self.nodes[l1].color and self.c[l1 - 1][adj[0] - 1] ==
"WHITE":
            self.p[l1 - 1][adj[0] - 1] = [l1, l2]
            self.dfs_visit(l1, adj[0])
    self.c[l1 - 1][l2 - 1] = "BLACK"
    self.time += 1
    self.d[l1 - 1][l2 - 1].append(self.time)
```

DFS print solution:

```
def printSoln(self):
    soln = []
    end = self.end
    while end != [self.s1, self.s2]:
        parent = self.p[end[0] - 1][end[1] - 1]
        if parent[0] == end[0]:
            soln.insert(0, "L " + str(end[1]))
        else:
            soln.insert(0, "R " + str(end[0]))
        end = parent

    for line in soln:
        print(line)
```

Actual use:

```
if __name__ == '__main__':
    s1, s2, nodes = readFile()
    createImage(nodes)

    dFS = DFS(s1, s2, nodes)
    dFS.dfs()
    dFS.printSoln()
```

Jonathan Woolf

Results:

```
L 7
R 10
L 11
R 15
L 6
R 14
L 2
R 8
L 7
R 4
L 11
R 1
L 6
R 10
L 2
R 15
L 7
R 14
L 11
R 8
L 6
R 4
L 2
L 3
R 1
L 12
R 10
L 17
R 15
L 16
R 14
R 20
L 14
R 21
L 8
R 22
L 4
R 23
L 1
R 27
L 10
R 26
L 15
R 25
L 14
L 20
R 28
```

Extra Credit:

I don’t know if this would count, but my algorithm does use the graphviz visual graphing library to produce the graph image, which can be seen above.