CSCI 406 - Project 4

Mehmet Yilmaz
May 2020

1.1 Problem Modeling:

a. Explain how you modeled the problem as a graph (typically, a few sentences)

The Jumping Jim's Encore problem was modeled as a graph in the following way. This graph would be a directed graph where each edge is not weighted. First, each node corresponded to a cell id. A cell id is basically a cell index. For example, the example input is a 8 x 8 matrix, 64 cells total, therefore it would contain 64 individual labels ranging between 1-64. Each edge represents a path Jim can move towards both horizontally and/or vertically based on the value in the cell Jim is standing on. For example if Jim is on a cell at (x, y) and that cell has the value 4, then this graph would have edges towards cells (x+4, y), (x, y+4), (x-4, y), and (x, y-4) if those cell coordinates are within the boundaries. The graph would be directed to prevent any instances where Jim goes back to a cell that he can't reach. With this design, we can create the graph that represents all the possible cells Jim can move vertically or horizontally.

To account for all the diagonal cell movements, we would generate "another" graph. This graph would be like the previous graph but each node would have a negative label, -1 to -64. This will distinguish horizontally/vertically cell movements with diagonal cell movements. For example if Jim is on a cell at (x, y) and that cell has the value 4, then this graph would have an edge towards cells (x+4, y+4), (x-4, y-4), (x+4, y-4), and (x-4, y+4) if those cell coordinates are within the boundaries. With this design, we can create the graph that represents all the possible cells Jim can move diagonally.

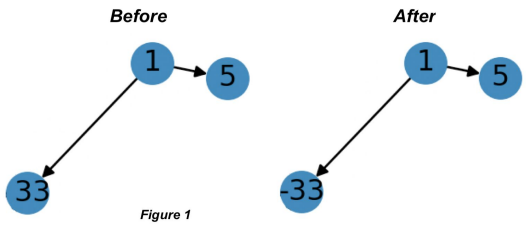


Figure 1

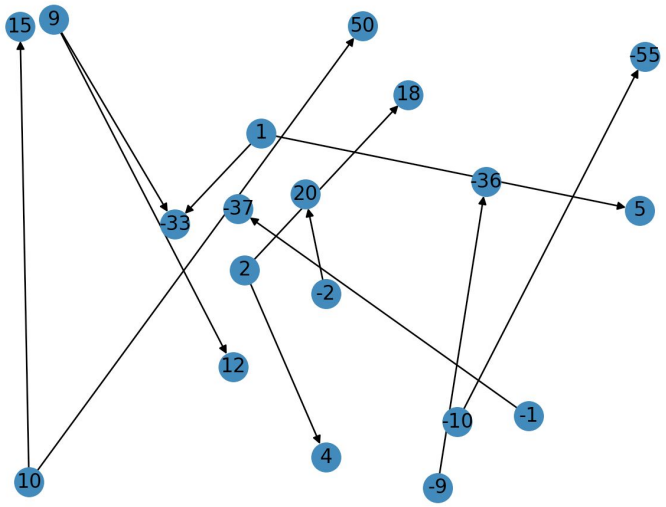
With these two graphs, we need some way to connect them. Before explaining the details, it's good to notice that for the first graph, positive nodes only point to other positive nodes. In the second graph, its negative nodes only point to other negative nodes. Based on the problem state, if Jim lands on a "Red" cell then he would switch from horizontal/vertical movement to diagonal and vice versa. To do this, we just make "Red" nodes only point to "inverted" nodes and any other nodes that point to a "Red" node would point to that "Red" node index but inverted. To see this, we can look at cell (1, 1) as an example. Node 33, cell (5, 1) is a "Red" node. Originally node 1 would point to node 33 but with this change towards "Red" cells, node 1 would point to node -33 instead. To see this example more clearly, you can view Figure 1.

By creating these two graphs, one for vertical/horizontal moves and another one for dionalical movement. Then by connecting those two graphs using rules applied towards "Red" nodes, we can mobel the Jumping Jim's Encore problem as a graph.

b. *Draw enough of the resulting graph to convince us that you have modeled the graph correctly (you don't have to draw the whole graph).*

• I included another image of the matrix to show what cell index corresponds to which cell value in the given 8x8 matrix input.

1) A position of the resulting graph (to see it more clearly):



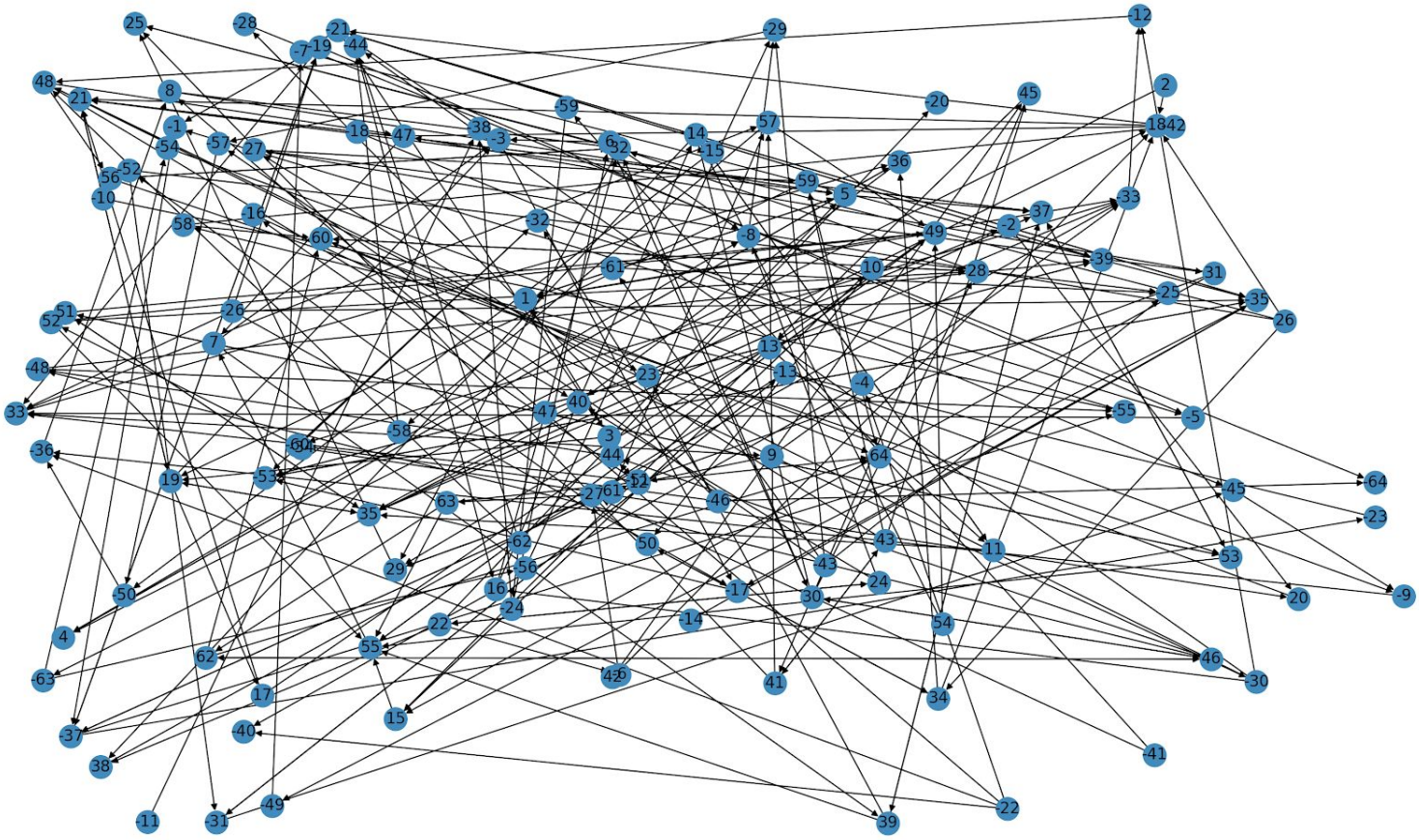
(8x8) input matrix → index matrix

4	2	-2	4	4	-3	4	-3
3	5	3	4	2	3	5	-2
4	3	2	-5	2	2	5	2
7	1	4	4	4	2	2	3
-3	2	2	4	2	5	2	5
2	-3	2	4	4	2	5	-1
6	2	2	-3	2	5	6	3
1	-2	5	4	4	2	-1	0

↓

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

2) The complete resulting graph (not as clear but shows the whole graph):



c. ***Identify the graph algorithm needed to solve the problem (one sentence).***
Due to each edge not being weighted, both Depth First Search (DFS) and Breadth First Search (BFS) would be great choices. For this project, I will pick BFS.

d. ***Convince us that this algorithm will actually solve the problem. This means that it will find a path if one exists (typically, a few sentences, depending on your approach).***
Our goal is to find a path from cell (1,1) [node 1] to cell (8,8) [node 64] from the graph generated in part a, which can be seen illustrated in part b1. To do this, we can use BFS because of the following reasons. BFS starts traversal at the root node (node 1) and can go through each node in numerical order. With this, BFS will find a path from node 1 to node 64. So if a path exists, BFS will find a path from node 1 to node 64.

1.2 Code Submission:

To see the code, please view all the content after sections 1.4. The code is located below all the other sections and the code is written in Python3 and uses the following modules to function: networkx and matplotlib. Please refer to "Code (Extension of Section 1.2)".

1.3 Results:

• Input (input.txt):

```
8 8
4 2 -2 4 4 -3 4 -3
3 5 3 4 2 3 5 -2
4 3 2 -5 2 2 5 2
7 1 4 4 4 2 2 3
-3 2 2 4 2 5 2 5
2 -3 2 4 4 2 5 -1
6 2 2 -3 2 5 6 3
1 -2 5 4 4 2 -1 0
```

• Output:

(1, 1) (5, 1) (8, 4) (4, 8) (1, 5) (5, 1) (2, 1) (2, 4) (6, 4) (6, 8) (7, 7) (1, 1) (5, 5) (3, 7) (8, 2) (8, 4) (8, 8)

1.4 Extra Credit:

I believe for Project 4, I deserve extra credit due to the following reasons:

1) For this project I used an algorithmic library. The library I used is called NetworkX and the documentations can be found here: https://networkx.github.io/documentation/stable/index.html
2) I also augmented my algorithm to visualize the overall graph for this project. The graphs generated can be seen in section 1.1 part b1 and b2. The full graph can be seen in section 1.1 part b2.
3) For this project, I did experimental comparisons between algorithms. I originally just used BFS to solve this graph problem, then I wanted to compare the resulting path of BFS to the paths generated by other algorithms. So I applied DFS and Dijkstra Search using the NetworkX library and compared the generated paths of all the algorithms. The results were a bit surprising because all of them output the same path.

go to the next page to see the code

Code (Extension of Section 1.2):

```
import matplotlib.pyplot as plt # using matplotlib for displaying graph
import networkx as nx # using NetworkX to create, manage, and search though graph

def readFile(fileName): # reads text file lines into list
    x = list()
    with open(fileName) as f:
      for line in f:
        x.append(line.replace('\n', '').split())
    return x

def createValueMatrix(x): # creates matrix of cell values
    y = []
    for i in range(len(x)):
        point = [] ; pz = [] ; start = len(x[0])*i
        for p in range(len(x[i])):
            point.append(int(x[i][p]))
        y.append(point)
    return y

def createIndexMatrix(row, col): # creates matrix of cell indexes (1-row*col)
    z = [] ; q = 0 ; point = []
    for i in range(row*col):
        q = q + 1
        point.append(q)
        if(len(point) % col == 0):
            z.append(point)
            point = []
    return z

def readData(fileName): # main function creating matrices, row, and col values
    x = readFile(fileName) # read text file into list
    y = createValueMatrix(x) # covert list of lines into a matrix of values

    row = y[0][0] # get matrix row
    col = y[0][1] # get matrix col
    y.pop(0)

    z = createIndexMatrix(row, col) # creates matrix of cell indexes (1-row*col)

    return y, z, row, col
```

```
def makePath(e, s, t): # convert NetworkX's output into a single list
    ans = []
    ans.insert(0, t)
    while(t != s):
        for i in range(len(e)):
            if(e[i][1] == t):
                ans.insert(0, e[i][0])
                t = e[i][0]
    return ans

def finalAnswer(n, answer): # convert single list output into the required format
    for i in range(len(n)):
        value = abs(n[i])
        for x1 in range(row):
            for y1 in range(col):
                if(z[x1][y1] == value):
                    answer = answer + "(" + str(x1+1) + ", " + str(y1+1) + ") "
    return answer

y, z, row, col = readData("input.txt") # read and create variables for desired file

G = nx.DiGraph() # set NetworkX graph object

# generate over all graph
for i in range(row):
    for p in range(col):
        point = z[i][p] # correct cell index
        v = abs(y[i][p]) # value in the cell
        sv = 1 # used to invert node value if that node is a Red

        # check every vertical and horizontal axis, then adds that cell(s) as an edge
        if(i+v >= 0 and i+v < row):
            if(y[i+v][p] < 0): sv = -1
            G.add_edge(point, z[i+v][p]*sv)
        sv = 1
        if(i-v >= 0 and i-v < row):
            if(y[i-v][p] < 0): sv = -1
            G.add_edge(point, z[i-v][p]*sv)
        sv = 1
        if(p+v >= 0 and p+v < col):
            if(y[i][p+v] < 0): sv = -1
            G.add_edge(point, z[i][p+v]*sv)
        sv = 1
```

```
        if(p-v >= 0 and p-v < col):
            if(y[i][p-v] < 0): sv = -1
            G.add_edge(point, z[i][p-v]*sv)

        # check every diagonal axis, then adds that cell(s) as an edge
        sv = 1
        point = z[i][p]*-1
        v = abs(y[i][p])
        if(i+v >= 0 and i+v < row and p+v >= 0 and p+v < col):
            if(y[i+v][p+v] < 0): sv = -1
            G.add_edge(point, z[i+v][p+v]*-1*sv)
        sv = 1
        if(i-v >= 0 and i-v < row and p-v >= 0 and p-v < col):
            if(y[i-v][p-v] < 0): sv = -1
            G.add_edge(point, z[i-v][p-v]*-1*sv)
        sv = 1
        if(i+v >= 0 and i+v < row and p-v >= 0 and p-v < col):
            if(y[i+v][p-v] < 0): sv = -1
            G.add_edge(point, z[i+v][p-v]*-1*sv)
        sv = 1
        if(i-v >= 0 and i-v < row and p+v >= 0 and p+v < col):
            if(y[i-v][p+v] < 0): sv = -1
            G.add_edge(point, z[i-v][p+v]*-1*sv)

nx.draw(G, with_labels=True, pos=nx.random_layout(G)) # draws graph

# checks to make sure a path does exists from 1 to row*col
if(nx.has_path(G, source=1, target=row*col)):

    # prints path and edge order using NetworkX's dijkstra search function
    n = list(nx.dijkstra_path(G, source=1, target=row*col))
    print("\n" + "Dijkstra SearchPath: " + str(finalAnswer(n, "")))
    print("Dijkstra Search Edges: " + str(n) + "\n")

    # prints path and edge order using NetworkX's breadth first search function
    n = list(nx.bfs_edges(G, source=1))
    n = makePath(n, 1, row*col)
    print("Breadth First Search Path: " + str(finalAnswer(n, "")))
    print("Breadth First Search Edges: " + str(n) + "\n")

    #prints path and edge order using NetworkX's depth first search function
    n = list(nx.dfs_edges(G, source=1))
    n = makePath(n, 1, row*col)
```

```
    print("Depth First Search Path: " + str(finalAnswer(n, "")))
    print("Depth First Search Edges: " + str(n) + "\n")

else:
    print("No Possible Path From 1-" + str(row*col))

plt.show()
```