Theory Assignment 3

Chenting Zhang

December 1, 2023

1 Problem 1

The vertices for the graph is $1, 2, 3, 4, 5, 6, 7, 8, 9, 10$. And because of the rules of the knights, I firstly displayed the chessboard and then loop through all the possible directions

$$[(2, 1), (-2, 1), (-2, -1), (2, -1), (1, 2), (1, -2), (-1, 2), (-1, -2)]$$

for all valid vertices. And then I stored all in-bound edges in a adjacency list which in code representation is a dictionary with vertices as keys and its edges as values. The chessboard, adjacency list as well as the adjacency matrices are displayed in figure 1, figure 2. By using the networkx package in python, the graph image is drawn in figure 3. The code is provided at the end of the report.

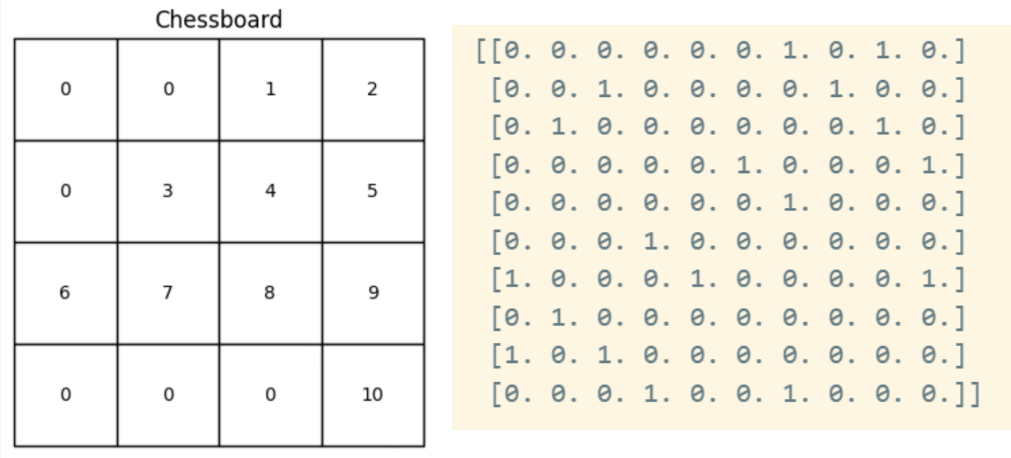

Figure 1: chessboard and adjacency matrix

```
{1: [9, 7], 2: [8, 3], 3: [9, 2], 4: [10, 6], 5: [7], 6: [4], 7: [1, 10, 5], 8: [2], 9: [1, 3], 10: [4, 7]}
```

Figure 2: adjacency list

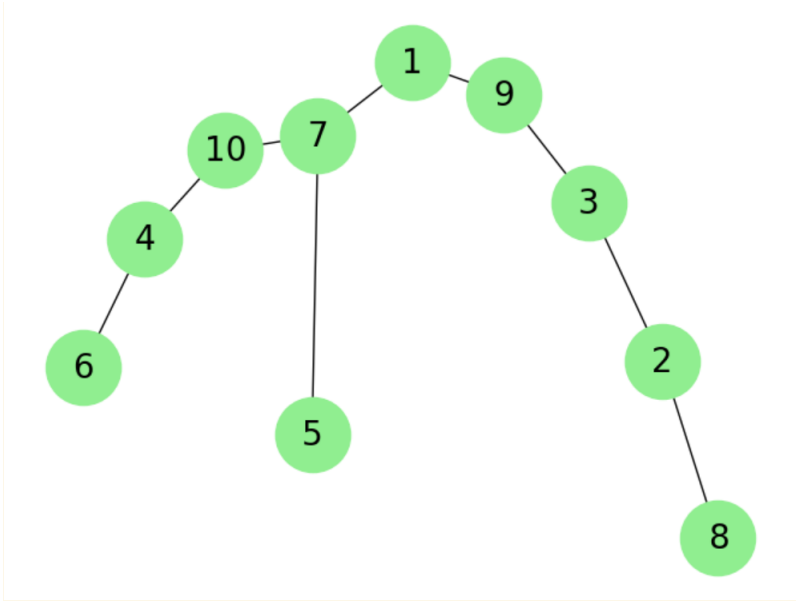

Figure 3: graph representation

2 Problem 2

1.

I would use DFS to find a path to a dashed node. Because the nature of DFS is to go into the depth of the graph instead of traversing all nodes in the next layer. If the goal is to find a solution quickly rather than necessarily finding the optimal solution, Depth-First Search (DFS) is generally the better choice compared to Breadth-First Search (BFS).

2.

Consider this graph $G = (V, E)$ with $V = v1, ...vn$. Let $n = |V|$ and $m = |E|$. In this scenerio, for DFS, the algorithm only goes through the first path, depicted in figure 4 left. The time complexity is only dependent of the length of the graph, which is $O(m+n) = O((L+1)+L) = O(L)$. For BFS, the algorithm traverse through all the layers above the last layer and stops whenever it find the first node in the last layer, depicted in figure 4 right. The time complexity is dependent of both the number of vertices in the graph and the length of the graph, which is $O(m+n) = O(K \times (L-1)+2+L \times K+1) = O(k \times L)$.

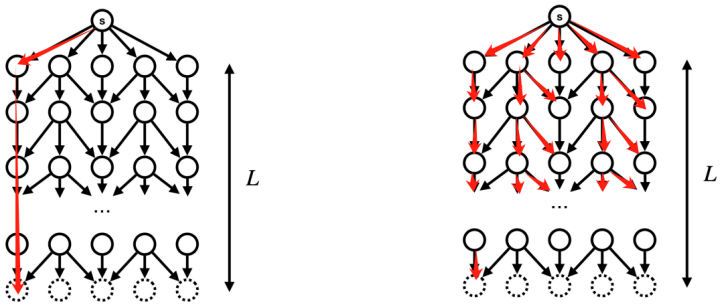

Figure 4: DFS vs BFS

3 Problem 3

Firstly, I construct code implementation of this weighted directed graph. Then because in Dijkstra's algorithm, the priority queue is needed for data structure, so I choose to use heap.py provided by

python's library to do this. In the heap, I stored the vertex and the distance as a pair, and I also used a dictionary to store this because heap is mostly used to pop the minimum value while dictionary is more easily for indexing. I also write a function called decreaseKey to update the distance of the vertex. Then I implemented the pseudo code for initialization and iteration process. The whole process is illustrated in figure 5. The expanded vertex from iteration 1 to 8 is $v2, v1, v3, v8, v5, v7, v6, v4$. The shortest distance from $v2$ to $v4$ is 66, the path is $v2 \rightarrow v3 \rightarrow v5 \rightarrow v6 \rightarrow v4$.

```
iteration 1
[[0, 'v2'], [inf, 'v1'], [inf, 'v3'], [inf, 'v4'], [inf, 'v5'], [inf, 'v6'], [inf, 'v7'], [inf, 'v8']]
expanded vertex v2
vertex v1 is modified from inf to 2
heap [[inf, 'v4'], [inf, 'v3'], [inf, 'v8'], [inf, 'v5'], [inf, 'v6'], [inf, 'v7'], [2, 'v1']]
vertex v3 is modified from inf to 3
heap [[inf, 'v4'], [inf, 'v8'], [inf, 'v5'], [inf, 'v6'], [inf, 'v7'], [2, 'v1'], [3, 'v3']]
iteration 2
[[inf, 'v4'], [inf, 'v8'], [inf, 'v5'], [inf, 'v6'], [inf, 'v7'], [2, 'v1'], [3, 'v3']]
expanded vertex v1
vertex v8 is modified from inf to 4
heap [[3, 'v3'], [inf, 'v6'], [inf, 'v4'], [inf, 'v7'], [inf, 'v5'], [4, 'v8']]
vertex v5 is modified from inf to 7
heap [[3, 'v3'], [inf, 'v6'], [inf, 'v4'], [inf, 'v7'], [4, 'v8'], [7, 'v5']]
iteration 3
[[3, 'v3'], [inf, 'v6'], [inf, 'v4'], [inf, 'v7'], [4, 'v8'], [7, 'v5']]
expanded vertex v3
vertex v5 is modified from 7 to 5
heap [[4, 'v8'], [inf, 'v4'], [inf, 'v7'], [inf, 'v6'], [5, 'v5']]
iteration 4
[[4, 'v8'], [inf, 'v4'], [inf, 'v7'], [inf, 'v6'], [5, 'v5']]
expanded vertex v8
vertex v4 is modified from inf to 104
heap [[5, 'v5'], [inf, 'v7'], [inf, 'v6'], [104, 'v4']]
vertex v7 is modified from inf to 7
heap [[5, 'v5'], [inf, 'v6'], [104, 'v4'], [7, 'v7']]
vertex v6 is modified from inf to 94
heap [[5, 'v5'], [104, 'v4'], [7, 'v7'], [94, 'v6']]
iteration 5
[[5, 'v5'], [104, 'v4'], [7, 'v7'], [94, 'v6']]
expanded vertex v5
vertex v6 is modified from 94 to 65
heap [[7, 'v7'], [104, 'v4'], [65, 'v6']]
iteration 6
[[7, 'v7'], [104, 'v4'], [65, 'v6']]
expanded vertex v7
iteration 7
[[65, 'v6'], [104, 'v4']]
expanded vertex v6
vertex v4 is modified from 104 to 66
heap [[66, 'v4']]
iteration 8
[[66, 'v4']]
expanded vertex v4
shortest path from v2 to v4 66
```

Figure 5: Iteration process

The figure of the Dijkstra's algorithm running result is illustrated in figure 6.

Dijksras algorithm

Iteration	expanded vertex	v1	v2	v3	v4	v5	v6	v7	v8
0	-	+∞	0	+∞	+∞	+∞	+∞	+∞	+∞
1	v2	2	0	3	+∞	+∞	+∞	+∞	+∞
2	v1	2	0	3	+∞	7	+∞	+∞	4
3	v3	2	0	3	+∞	5	+∞	+∞	4
4	v8	2	0	3	104	5	94	7	4
5	v5	2	0	3	104	5	65	7	4
6	v7	2	0	3	104	5	65	7	4
7	v6	2	0	3	66	5	65	7	4
8	v4	2	0	3	66	5	65	7	4

Figure 6: Dijkstra's algorithm distance

4 Problem 4

The procedure is similar to the InOrderTreeWalk in the binary tree. However, the left children and the right children in this modified InOrderTreeWalk is a list of nodes instead of a single node. And it should firstly traverse all the left nodes in order and then traverse the right nodes. The code is provided at the end of the report. And the printed result is illustrated in figure 7.

```
v5 v6 v1 v7 v8 v9 v10 v2 v11 v12 v0 v13 v14 v3 v15 v16 v17 v18 v4 v19 v20
```

Figure 7: InOrderTreeWalk

4.1 Code

4.1.1 Code for Problem1

```
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sn

# display chessboard
n = 4
chessboard = [[0] * n for _ in range(n)]

chessboard[0][2] = 1
chessboard[0][3] = 2
chessboard[1][1] = 3
chessboard[1][2] = 4
chessboard[1][3] = 5
chessboard[2][0] = 6
chessboard[2][1] = 7
chessboard[2][2] = 8
chessboard[2][3] = 9
chessboard[3][3] = 10

def draw_chessboard(chessboard):
    chessboard = np.array(chessboard)
    rows, cols    = chessboard.shape

    # Create figure of the size of the maze
    fig = plt.figure(1, figsize=(cols, rows))

    # Remove the axis ticks and add title
    ax = plt.gca()
    ax.set_title('Chessboard')
    ax.set_xticks([])
    ax.set_yticks([])

    # Create figure of the size of the maze
    fig = plt.figure(1, figsize=(cols, rows))

    # Create a table to color
    grid = plt.table(cellText=chessboard,
                           cellColours=None,
                           cellLoc='center',
                           loc=(0, 0),
                           edges='closed')
    # Modify the hight and width of the cells in the table
    tc = grid.properties()['children']
    for cell in tc:
        cell.set_height(1.0/rows)
        cell.set_width(1.0/cols)

draw_chessboard(chessboard)
print(chessboard)

# Compute the edges
valid_indices = []
for row_idx in range(n):
    for col_idx in range(n):
        if chessboard[row_idx][col_idx] == 0:
            continue
        else:
            valid_indices.append((row_idx, col_idx))
```

```
def out_of_bound(chessboard, pos,n):
    x,y = pos
    return True if x < 0 or x >= n or y < 0 or y >= n or chessboard[x][y]
        == 0 else False


Adjacency_list = {} # dict{1:[7,9], 2:[3,8], ...}
directions = [(2,1),(-2,1),(-2,-1),(2,-1),(1,2),(1,-2),(-1,2),(-1,-2)]
pos_move = tuple()

for row_idx, col_idx in valid_indices:
    Adjacency_list[chessboard[row_idx][col_idx]] = []
    for dir in directions:
        pos_move = (row_idx + dir[0], col_idx + dir[1])
        if out_of_bound(chessboard, pos_move, n):
            continue
        else:
            Adjacency_list[chessboard[row_idx][col_idx]].append(
               chessboard[pos_move[0]][pos_move[1]])

print(Adjacency_list)


Adjacency_matrix = np.zeros((10,10))
for key in Adjacency_list.keys():
    for value in Adjacency_list[key]:
        Adjacency_matrix[key - 1][value - 1] = 1

print(Adjacency_matrix)


# create the graph
G = nx.Graph(Adjacency_list)
nx.draw(G, pos = None, ax = None, with_labels = True,font_size = 20,
   node_size = 2000, node_color = 'lightgreen')
```

4.1.2 Code for Problem3

```
# Construct the graph
G = nx.DiGraph()

# Adding nodes
G.add_node('v1')
G.add_node('v2')
G.add_node('v3')
G.add_node('v4')
G.add_node('v5')
G.add_node('v6')
G.add_node('v7')
G.add_node('v8')


# Adding weighted edges
G.add_edge('v1', 'v2', weight=6)
G.add_edge('v1', 'v8', weight=2)
G.add_edge('v1', 'v5', weight=5)

G.add_edge('v2', 'v1', weight=2)
G.add_edge('v2', 'v3', weight=3)

G.add_edge('v3', 'v5', weight=2)
```

```
G.add_edge('v5', 'v1', weight=5)
G.add_edge('v5', 'v6', weight=60)
G.add_edge('v5', 'v8', weight=3)

G.add_edge('v6', 'v4', weight=1)

G.add_edge('v8', 'v4', weight=100)
G.add_edge('v8', 'v7', weight=3)
G.add_edge('v8', 'v1', weight=5)
G.add_edge('v8', 'v6', weight=90)

# Q.decreaseKey(v, dv)
def decreaseKey(v, dv, hq):
    for pair in hq:
        if pair[1] == v:
            # delete this pair and modify
            hq.remove(pair)
            pair[0] = dv
            # add new pair
            hq.append(pair)
    # print(hq)
    return hq

import heapq
table = []
table.append(['Iteration', 'expanded' + '\n' + 'vertex', 'v1', 'v2', 'v3'
    , 'v4', 'v5', 'v6', 'v7', 'v8'])
Q = []
D = dict()
heapq.heapify(Q)
start_node = 'v2'
end_node = 'v4'
nodes = G.nodes()

# initialization
iteration = 0
for v in G.nodes():
    if v == start_node:
        heapq.heappush(Q, [0, v])
        D[v] = 0
    else:
        heapq.heappush(Q, [float('inf'), v])
        D[v] = float('inf')
table.append([iteration, '-'] + [D[node] if D[node] != float('inf') else
    '+   ' for node in G.nodes()])


while len(Q) != 0:
    iteration += 1
    print(iteration)
    print(Q)
    heapq.heapify(Q)
    d, u = heapq.heappop(Q)
    print(u,'expanded')
    expanded_vertex = u
    out_neighbors = list(G.successors(u))
    for v in out_neighbors:
        dv = d + G[u][v]['weight']
        if dv < D[v]:
            print(v, dv,'modified')
            Q = decreaseKey(v, dv, Q)
            print(Q)
            D[v] = dv
```

```
    table.append([iteration, expanded_vertex] + [D[node] if D[node] !=
        float('inf') else '+   ' for node in G.nodes()])
print('shortest path from v2 to v4', D['v4'])


# display
def draw_table(table):
    table = np.array(table)
    rows, cols = table.shape

    # Create figure of the size of the maze
    fig = plt.figure(1, figsize=(cols, rows))

    # Remove the axis ticks and add title
    ax = plt.gca()
    ax.set_title('Dijksras algorithm')
    ax.set_xticks([])
    ax.set_yticks([])

    # Create figure of the size of the maze
    fig = plt.figure(1, figsize=(cols, rows))

    # Create a table to color
    grid = plt.table(cellText=table,
                             cellColours=None,
                             cellLoc='center',
                             loc=(0, 0),
                             edges='closed')
    # Modify the hight and width of the cells in the table
    tc = grid.properties()['children']
    for cell in tc:
        cell.set_height(1.0/rows)
        cell.set_width(1.0/cols)

    grid.auto_set_font_size(False)
    grid.set_fontsize(10)
    # Hide axes
    ax.axis('off')
    plt.show()

draw_table(table)
```

4.1.3 Code for Problem4

```
class Tree:
def __init__(self, node):
    self.node = node
    self.leftchildren = None
    self.rightchildren = None

node = dict()
node['v0'] = Tree(20)
node['v0'].leftchildren = ['v1', 'v2']
node['v0'].rightchildren = ['v3', 'v4']

node['v1'] = Tree(5)
node['v1'].leftchildren = ['v5', 'v6']
node['v1'].rightchildren = ['v7', 'v8']

node['v2'] = Tree(16)
node['v2'].leftchildren = ['v9', 'v10']
node['v2'].rightchildren = ['v11', 'v12']
```

```
node['v3'] = Tree(30)
node['v3'].leftchildren = ['v13', 'v14']
node['v3'].rightchildren = ['v15', 'v16']

node['v4'] = Tree(80)
node['v4'].leftchildren = ['v17', 'v18']
node['v4'].rightchildren = ['v19', 'v20']


node['v5'] = Tree(1)
node['v6'] = Tree(2)
node['v7'] = Tree(8)
node['v8'] = Tree(9)
node['v9'] = Tree(10)
node['v10'] = Tree(13)
node['v11'] = Tree(17)
node['v12'] = Tree(19)
node['v13'] = Tree(23)
node['v14'] = Tree(25)
node['v15'] = Tree(40)
node['v16'] = Tree(45)
node['v17'] = Tree(61)
node['v18'] = Tree(65)
node['v19'] = Tree(100)
node['v20'] = Tree(101)



print(node['v2'].rightchildren)
print(node['v5'].rightchildren)

def InOrderTreeWalk(node, target='v0'):
    if node[target].leftchildren is not None:
        for left_child in node[target].leftchildren:
            InOrderTreeWalk(node, left_child)

    print(target, end=' ')
    if node[target].rightchildren is not None:
        for right_child in node[target].rightchildren:
            InOrderTreeWalk(node, right_child)



InOrderTreeWalk(node)
```