Assignment 12

Name: DONGWOOK LEE

Problem 12.1 *Shortest Path Algorithm* (3 points)

Idea Explanation:

Assume for our original map we have

$\delta(u,v)$ contains 5 paths. (possibly with negative weight)

and our alternative path

$p(u,v)$ contains 3 paths. (possibly with negative weight)

that

- weight $\delta(u,v) = a + b + c + d + e$
- weight $p(u,v) = f + g + h$

Now we add constant C to every Edge

- weight $\delta(u,v) = a + b + c + d + e + 5 \cdot C$
- weight $p(u,v) = f + g + h + 3 \cdot C$

Even we had $\delta(u,v) < p(u,v)$,

$\delta(u,v) + 5 \cdot C < p(u,v) + 3 \cdot C$

can possibly not hold.

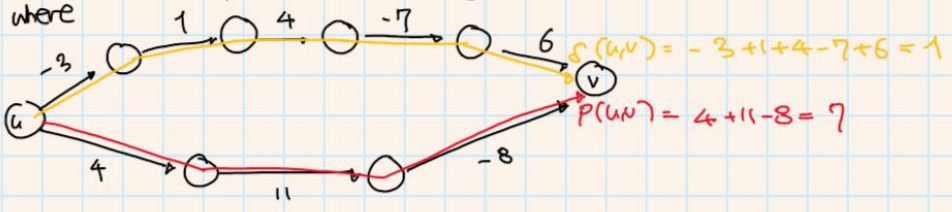
Counter Example:

Even we had $\delta(u,v) < p(u,v)$,

$\delta(u,v) + 5 \cdot C < p(u,v) + 3 \cdot C$

can possibly not hold.

All I need to prove now is giving a counter example where



$\delta(u,v) = 1 < p(u,v) = 7$, but

$\delta(u,v) + 5 \cdot 100 = 501 > p(u,v) + 3 \cdot 100 = 307$

Problem 12.2 *Optimal Meeting Point* (7 points)

Dijkstra Algorithm:

```
def find_meetup_city(adj_matrix, my_city, friend_city):
    # my_city is the starting node
    n = len(adj_matrix)
    visited = list()

    my_matrix = [[0 for col in range(3)] for row in range(n)]
    for i in range(n):
        my_matrix[i][0] = adj_matrix[i][0]
        my_matrix[i][1] = float('inf')  # shortest distance from my_city
        my_matrix[i][2] = 0

    visiting_city = my_city
    next_city = 0
    my_matrix[visiting_city][1] = 0
    while len(visited) != n:
        visited.append(visiting_city)

        min = float('inf')
        for i in range(0, n):
            if i not in visited:
                if my_matrix[i][1] > my_matrix[visiting_city][1] + adj_matrix[visiting_city][i]:
                    my_matrix[i][1] = my_matrix[visiting_city][1] + adj_matrix[visiting_city][i]
                    my_matrix[i][2] = visiting_city

                if my_matrix[i][1] < min and my_matrix[i][0] != visiting_city:
                    min = my_matrix[i][1]
                    next_city = i

        visiting_city = next_city
```

```
    print("Visited Cities in order : ", end=' ')
    for i in range(0, len(visited)):
        print(visited[i], end=' ')
    print()

    # Now I need to back-track the path
    global path
    path = list()

    current_city = friend_city
    while current_city != my_city:
        path.insert(0, current_city)
        current_city = my_matrix[current_city][2]
    path.insert(0, current_city)

    # Now I find the place in the middle, which is m
    tsum_u = 0
    tsum_v = 0

    it_one = 0
    it_sec = len(path)-1
    for i in range(0, len(path)-1):
        if tsum_u <= tsum_v:
            tsum_u += adj_matrix[path[it_one]][path[it_one+1]]
            dest_u = path[it_one+1]
            it_one += 1
        else:
            tsum_v += adj_matrix[path[it_sec-1]][path[it_sec]]
            dest_v = path[it_sec-1]
            it_sec -= 1

    print("Total time (", my_city, ", ", friend_city, ") : ", my_matrix[friend_city][1])
```

```
    print("Path (", my_city, ", ", friend_city, ") : ", end=' ')
    for i in range(0, len(path)):
        print(path[i], end=' ')
    print()

    return dest_u
```

Input 1:

```
74      my_city = 0
75      friend_city = 2
76
77      adj_matrix = [[0, 3, 6, 8, 7],
78                    [3, 0, 2, 4, 8],
79                    [6, 2, 0, 5, 5],
80                    [8, 4, 5, 0, 2],
81                    [7, 8, 5, 2, 0]]
82      print("Ideal Meet_Up City : ", find_meetup_city(adj_matrix, my_city, friend_city))
```

1) Make my own Matrix with n rows and 3 columns

Current City	Shortest Distance from my_city	Previous City
0 (Start)	INF	**None**
1	INF	**None**
2	INF	**None**
3	INF	**None**
4	INF	**None**

2) Use Dijkstra Algorithm to fill up the above matrix

Current City	Shortest Distance from my_city	Previous City
0 (Start)	INF	
1	3	0
2	5	1
3	7	1
4	7	0

3) Find the Path by back tracking (ex) 0 → 2)

Current City	Shortest Distance from my_city	Previous City
0 (Start)	INF	
1	3	0
2	5	1
3	7	1
4	7	0

4) Find the middle point by my own algorithm

m = 1

Output 1:

```
Visited Cities in order :  0 1 2 3 4
Total time ( 0 ,  2 ) :  5
Path ( 0 ,  2 ) :  0 1 2
Ideal Meet_Up City :  1
```

Input 2:

```
my_city2 = 0
friend_city2 = 3

adj_matrix2 = [[0,              7,              float('inf'),   float('inf'),   3,              10,             float('inf')],
               [7,              0,              4,              10,             2,              6,              float('inf')],
               [float('inf'),   4,              0,              2,              float('inf'),   float('inf'),   float('inf')],
               [float('inf'),   10,             2,              0,              11,             9,              4],
               [3,              2,              float('inf'),   11,             0,              float('inf'),   5],
               [10,             6,              float('inf'),   9,              float('inf'),   0,              float('inf')],
               [float('inf'),   float('inf'),   float('inf'),   4,              5,              float('inf'),   0]]
print("Ideal Meet_Up City : ", find_meetup_city(adj_matrix2, my_city2, friend_city2))
```

1) Follow the same algorithm used in Input 1

Output 2:

```
Visited Cities in order :  0 4 1 6 2 5 3
Total time ( 0 ,  3 ) :  11
Path ( 0 ,  3 ) :  0 4 1 2 3
Ideal Meet_Up City :  1
```

Problem 12.3 *Number Maze* (10 points)

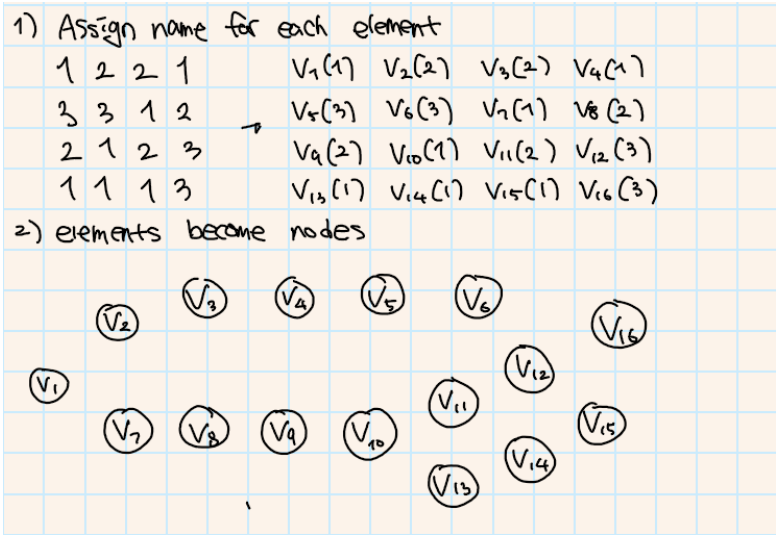
(a) (2 points)

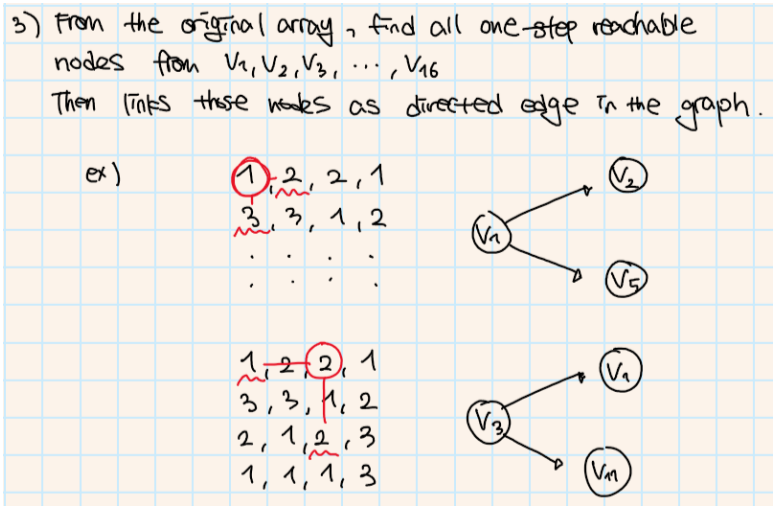
Assume we have a matrix **A** with nxn size.

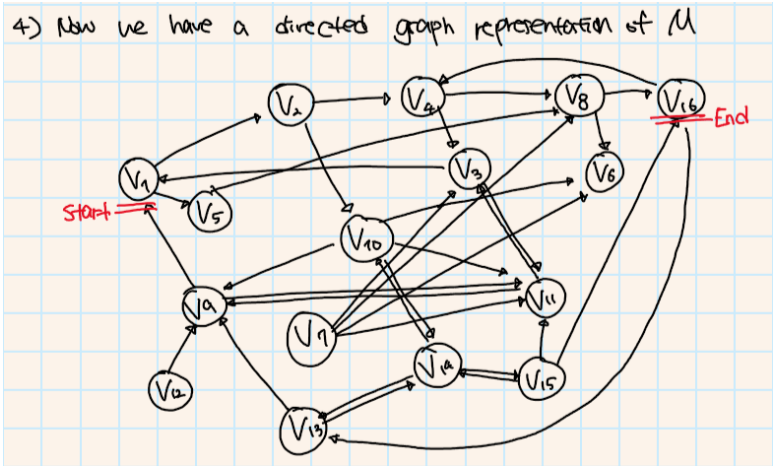
All the elements of the array become nodes of our graph,

and all the one-step reachable elements from one another are represented as directed edges.

Ex) Our matrix A = [[1, 2, 2, 1], [3, 3, 1, 2], [2, 1, 2, 3], [1, 1, 1, 3]]







(b) (4 points)

Class Declaration and Definition:

```python
import random

INF = float('inf')
```

```python
class PuzzleBoard:
    def __init__(self, n):
        self.grid_num = n
        self.number_maze = [[0 for col in range(n)]for row in range(n)]
        for i in range(n):
            for j in range(n):
                temp = random.randrange(1, n)
                self.number_maze[i][j] = temp
        self.current_row = 0
        self.current_column = 0
        self.__adj_matrix = node_edge_matrix(self.number_maze)

    def make_move(self, direction):
        # 0 --> up
        # 1 --> right
        # 2 --> down
        # 3 --> left
        factor = self.number_maze[self.current_row][self.current_column]

        if direction == 0:
            if 0 <= (self.current_row - (1 * factor)):
                self.current_row -= (1 * factor)
                return True

        elif direction == 1:
            if (self.current_column + (1 * factor)) <= self.grid_num:
                self.current_column += (1 * factor)
                return True

        elif direction == 2:
```

```python
            if (self.current_row + (1 * factor)) <= self.grid_num:
                self.current_row += (1 * factor)
                return True

        elif direction == 3:
            if 0 <= (self.current_column - (1 * factor)):
                self.current_column -= (1 * factor)
                return True

        return False

    def get_result(self):
        if self.current_row == n-1 and self.current_column == n-1:
            return True
        else:
            return False

    def __str__(self):
        temp_str = ""
        for i in range(self.grid_num):
            for j in range(self.grid_num):
                temp_str += str(self.number_maze[i][j])
                temp_str += " "
            temp_str += "₩n"

        return temp_str

    def solve(self):
        return number_maze_path(self.number_maze)
```

*'def number_maze_path(self.number_maze)' of 'def solve(self)' defined in Bonus Answer Sheet

Main Function:

```python
if __name__ == '__main__':
    grid_size = 5                                # Set grid size
    my_puzzle = PuzzleBoard(grid_size)

    n = len(my_puzzle.number_maze)
    print(my_puzzle)

    print("Starting from [0] ...₩n0: UP₩n1: RIGHT₩n2: DOWN₩n3: LEFT₩n4: STOP and SOLVE")
    while True:
        current_idx = (grid_size*my_puzzle.current_row)+my_puzzle.current_column
        # print("₩nCurrent Key =", current_idx)
        print("Current Position =", my_puzzle.current_row, my_puzzle.current_column)
        if current_idx == (n**2)-1:
            print("Puzzle Solved!")
            break
        user = int(input("Direction: "))
        if user == 4:
            print("Puzzle Solved!" if my_puzzle.get_result() else "Not Solved")
            break
        elif user < 0 or user > 4:
            print("Invalid Key")
            continue
        print("Valid Key" if my_puzzle.make_move(user) else "Invalid Key")

    print(my_puzzle.solve())
```

Sample Test:

```
2 3 3 2
1 1 2 1
3 1 1 3
3 2 1 1

Starting from [0] ...
0: UP
1: RIGHT
2: DOWN
3: LEFT
4: STOP and SOLVE
Current Position = 0 0
Direction: 1
Valid Key
Current Position = 0 2
Direction: 2
Valid Key
Current Position = 3 2
Direction: 1
Valid Key
Current Position = 3 3
Puzzle Solved!
Path ( 0 ,  15 ) :  0 2 14 15
3
```

Changed Grid Size into 4

```
4 4 1 3 4
3 4 4 1 1
2 1 1 2 2
2 1 2 4 2
2 1 3 3 3

Starting from [0] ...
0: UP
1: RIGHT
2: DOWN
3: LEFT
4: STOP and SOLVE
Current Position = 0 0
Direction: 1
Valid Key
Current Position = 0 4
Direction: 2
Valid Key
Current Position = 4 4
Puzzle Solved!
Path ( 0 ,  24 ) :  0 4 24
2
```

Changed Grid Size into 5