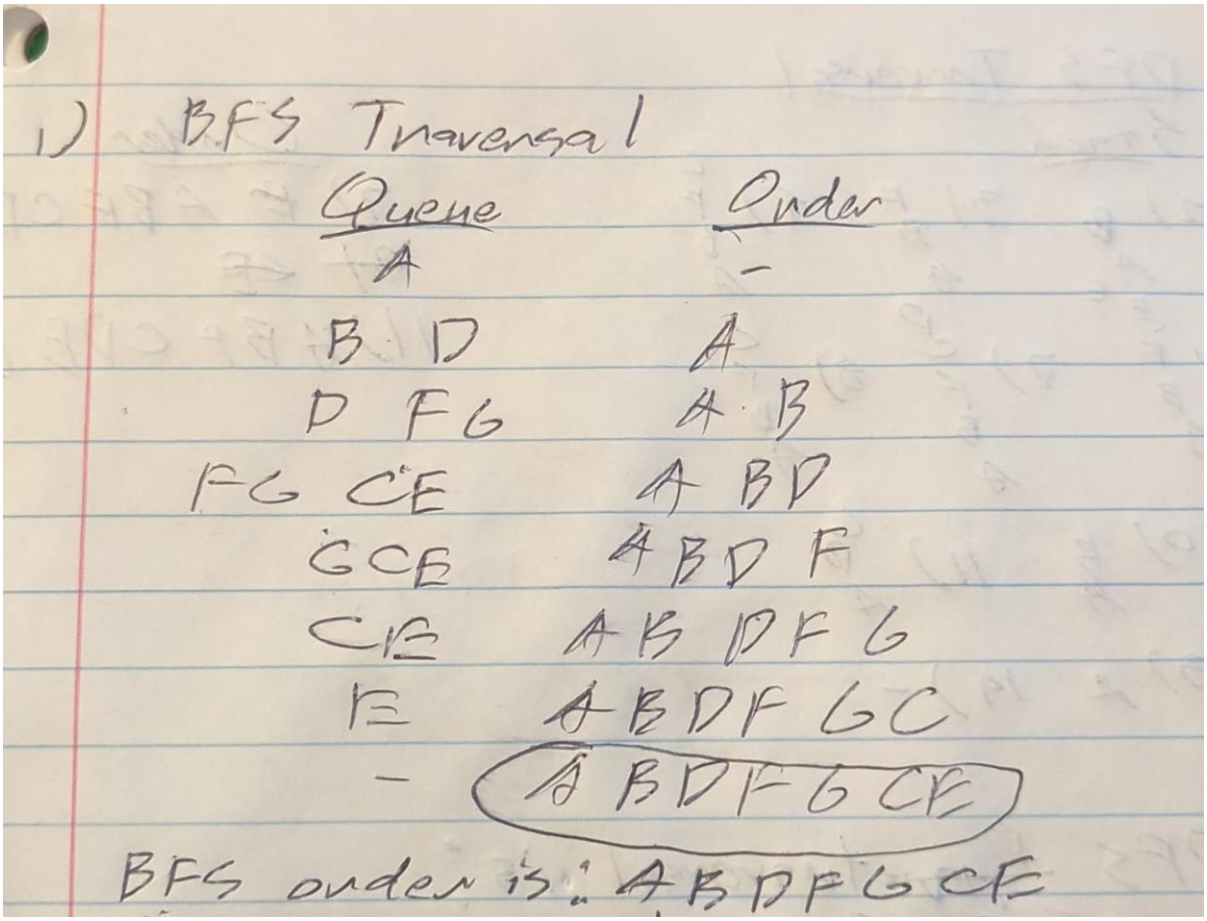
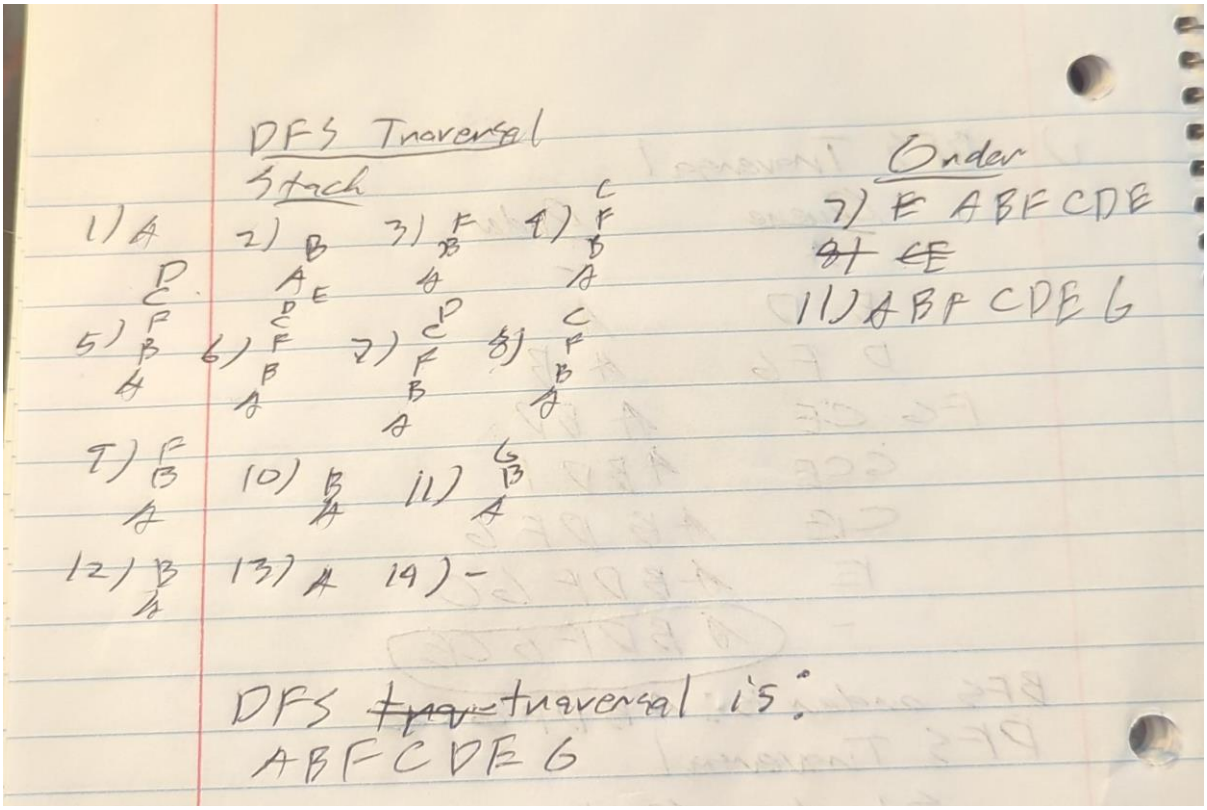
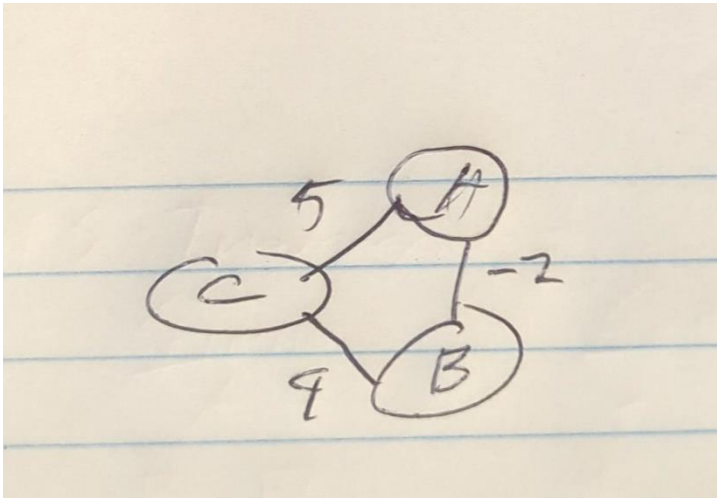
Assignment: Graph Algorithms

1. **Write BFS and DFS for a graph**: What would be BFS and DFS traversal for the below graphs. Write the nodes for BFS and DFS. Start at node A.

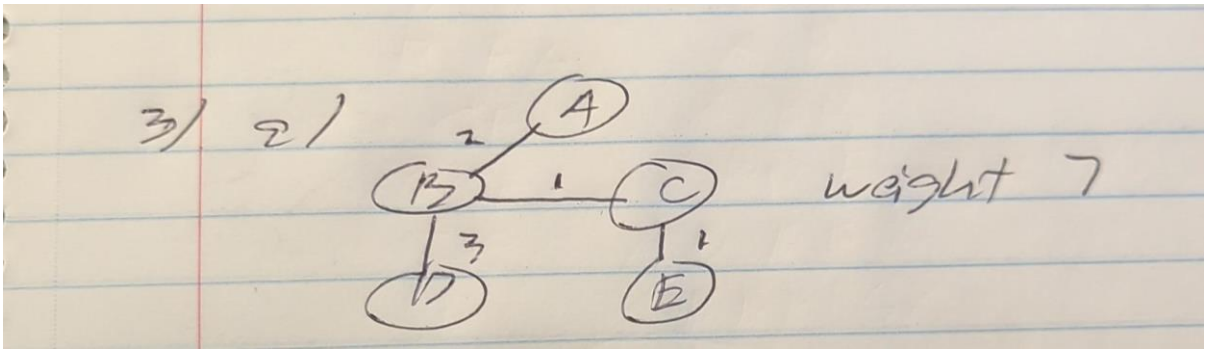
   a.

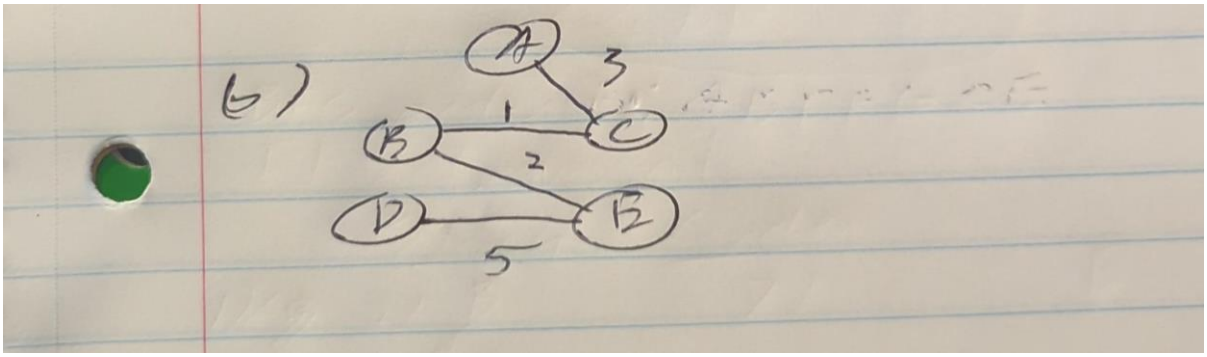


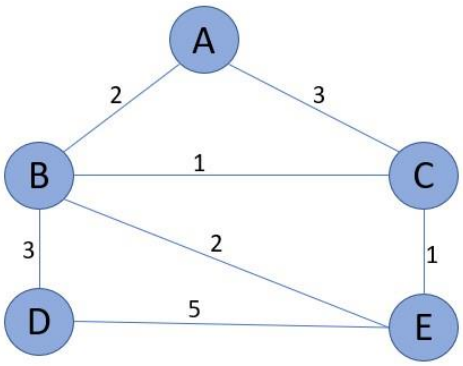


2. **Analyze Dijkstra with negative edges**: Analyze with a sample graph and show why Dijkstra does not work with negative edges. Give the sample graph and write your explanation why Dijkstra would not work in this case.
   a. 

   b. Dijkstra's algorithm will find the shortest distance from C to B as 4 instead of 3 since B is visited after S in the beginning. The shortest path is C to A to B with a distance of 3. With negatively weighted ages, distances can decrease the distance to a node. On the other hand, positively weighted edges increase the distance to a node.

3. **Draw Minimum Spanning Tree**
   a. Draw minimum spanning tree for the below graph.


   b. Draw spanning Tree that is not minimum




4. **MST implementation:**

a. Implement Prims' algorithm Name your function **Prims(G).** Include function in the file **MST.PY.** Mention in your submission the input format and output format of your program.

Input: a graph represented as an adjacency matrix

For example, the graph in the Exploration would be represented as the below (where index 0 is A, index 1 is B, etc.).

```
input = [
  [0, 8, 5, 0, 0, 0, 0],
  [8, 0, 10, 2, 18, 0, 0],
  [5, 10, 0, 3, 0, 16, 0],
  [0, 2, 3, 0, 12, 30, 14],
  [0, 18, 0, 12, 0, 0, 4],
  [0, 0, 16, 30, 0, 0, 26],
  [0, 0, 0, 14, 4, 26, 0]
]
```

Output: a list of tuples, wherein each tuple represents an edge of the MST as (v1, v2, weight)

For example, the MST of the graph in the Exploration would be represented as the below.

```
output = [(0, 2, 5), (2, 3, 3), (3, 1, 2), (3, 4, 12), (2, 5, 16), (4, 6, 4)]
```

Note: the order of edge tuples within the output does not matter; additionally, the order of vertices within each edge does not matter. For example, another valid output would be below (v1 and v2 in the first edge are flip-flopped; the last two edges in the list are flip-flopped).

```
output = [(2, 0, 5), (2, 3, 3), (3, 1, 2), (3, 4, 12), (4, 6, 4), (2, 5, 16)]
```

b. What is the difference between the Kruskal's and the Prim's algorithm?

- Kruskal's algorithm focuses on the edges of the graph. It iteratively selects the edge with the minimum wait and checks if the edge does not create a cycle. This continues until all vertices are connected or MST is completed.
- Prim's algorithm focuses on the vertices of the graph. The algorithm starts with a single vertex and adds to the graph via the minimum weight edge that

connects to the MST. It has a prior set of vertices in the MST and updates the minimum weight edge connecting the set. This continues until all vertices are connected.

5. Apply BFS/DFS/MST to solve a problem (Portfolio Project Problem):

You are given a 2-D puzzle of size MxN, that has N rows and M column (M and N can be different). Each cell in the puzzle is either empty or has a barrier. An empty cell is marked by '-' (hyphen) and the one with a barrier is marked by '#'. You are given two coordinates from the puzzle (a,b) and (x,y). You are currently located at (a,b) and want to reach (x,y). You can move only in the following directions. L: move to left cell from the current cell
R: move to right cell from the current cell
U: move to upper cell from the current cell
D: move to the lower cell from the current cell

You can move to only an empty cell and cannot move to a cell with a barrier in it. Your goal is to reach the destination cells covering the minimum number of cells as you travel from the starting cell.

Example Board:

-	-	-	-	-
-	-	#	-	-
-	-	-	-	-
#	-	#	#	-
-	#	-	-	-

Input: board, source, destination.

Puzzle: A list of lists, each list represents a row in the rectangular puzzle. Each element is either '-' for empty (passable) or '#' for obstacle (impassable). The same as in the example. Example:

```
Puzzle = [
    ['-', '-', '-', '-', '-'],
    ['-', '-', '#', '-', '-'],
    ['-', '-', '-', '-', '-'],
```

```
    ['#', '-', '#', '#', '-'],
['-', '#', '-', '-', '-']
]
```

source: A tuple representing the indices of the starting position, e.g. for the upper right corner, source=(0, 4).
destination: A tuple representing the indices of the goal position, e.g. for the lower right corner, goal=(3, 4).

Output: A list of tuples representing the indices of each position in the path. The first tuple should be the starting position, or source, and the last tuple should be the destination. If there is no valid path, None should be returned. Not an empty list, but the None object. If source and destination are same return the same cell.

Note: The order of these tuples matters, as they encode a path. Each position in the path must be empty (correspond to a '-' on the board) and adjacent to the previous position.

Example 1 (consider above puzzle)
Input: puzzle, (0,2), (2,2)
Output: [(0, 2), (0, 1), (1, 1), (2, 1), (2, 2)]

Example 2 (consider above puzzle)
Input: puzzle, (0,0), (4,4)
Output: [(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 4), (2, 4), (3, 4), (4, 4)]

Example 3: (consider above puzzle)
Input: puzzle, (0,0), (4,0)
Output: None

Example 4: (consider above puzzle)
Input: puzzle, (0,0), (0,0)
Output: [(0,0)]

a. Describe an algorithm to solve the above problem.

a To solve the problem, we can use a breadth-first search (BFS) for graph traversal. Each cell in the 2-D puzzle can be considered as a node in the graph, and edges exist between each cell and its adjacent cells (left, right, up, down). The BFS algorithm can be used to find the shortest path from the source cell to the destination cell.

1. Define the problem as a graph, where each cell in the puzzle is a node and edges exist between each cell and its adjacent cells (left, right, up, down).
2. Initialize an empty queue and a visited list. The queue will be used to keep track of cells to visit, and the visited list will record which cells have already been visited.

3. Start from the source cell and add it to the queue. Also mark this cell as visited in the visited list.
4. Start a loop that continues until the queue is empty:
    a. Dequeue a cell from the queue and check if it's the destination cell.
    b. If it is, return the path taken to reach it and stop the algorithm.
    c. If it's not the destination, consider all of its adjacent cells.
    d. For each adjacent cell, check if it's within the puzzle boundaries, not visited before, and not a barrier.
    e. If these conditions are met, add the cell to the queue, mark it as visited, and update the path with this cell.

5. If the queue becomes empty and the destination cell hasn't been found, we return None to indicate that there is no valid path from source to the destination.

6. The output from the BFS function is a list of tuples representing the indices of each position in the path from the source to the destination, or None if no valid path exists.

b. Implement your solution in a function **solve_puzzle(Board, Source, Destination).** Name your file Puzzle.py

c. What is the time complexity of your solution?

    a In the worst case, all cells need to be explored, so the total number of iterations to explore the board is proportional to the number of cells. This is O(rows * columns).

d. **(Extra Credit):** For the above puzzle in addition to the output return a set of possible directions as well in the form of a string.

For above example 1

Output: ([(0, 2), (0, 1), (1, 1), (2, 1), (2, 2)], 'LDDR')

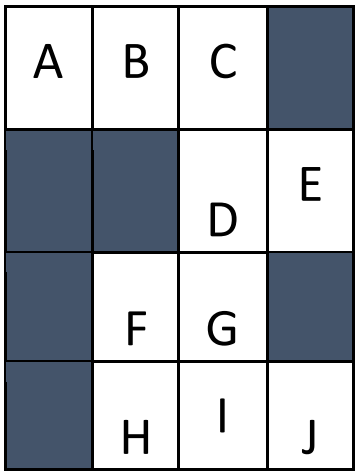
For above example 2

Output: ([(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 4), (2, 4), (3, 4), (4, 4)], 'RRRRDDDD')

For above example 4

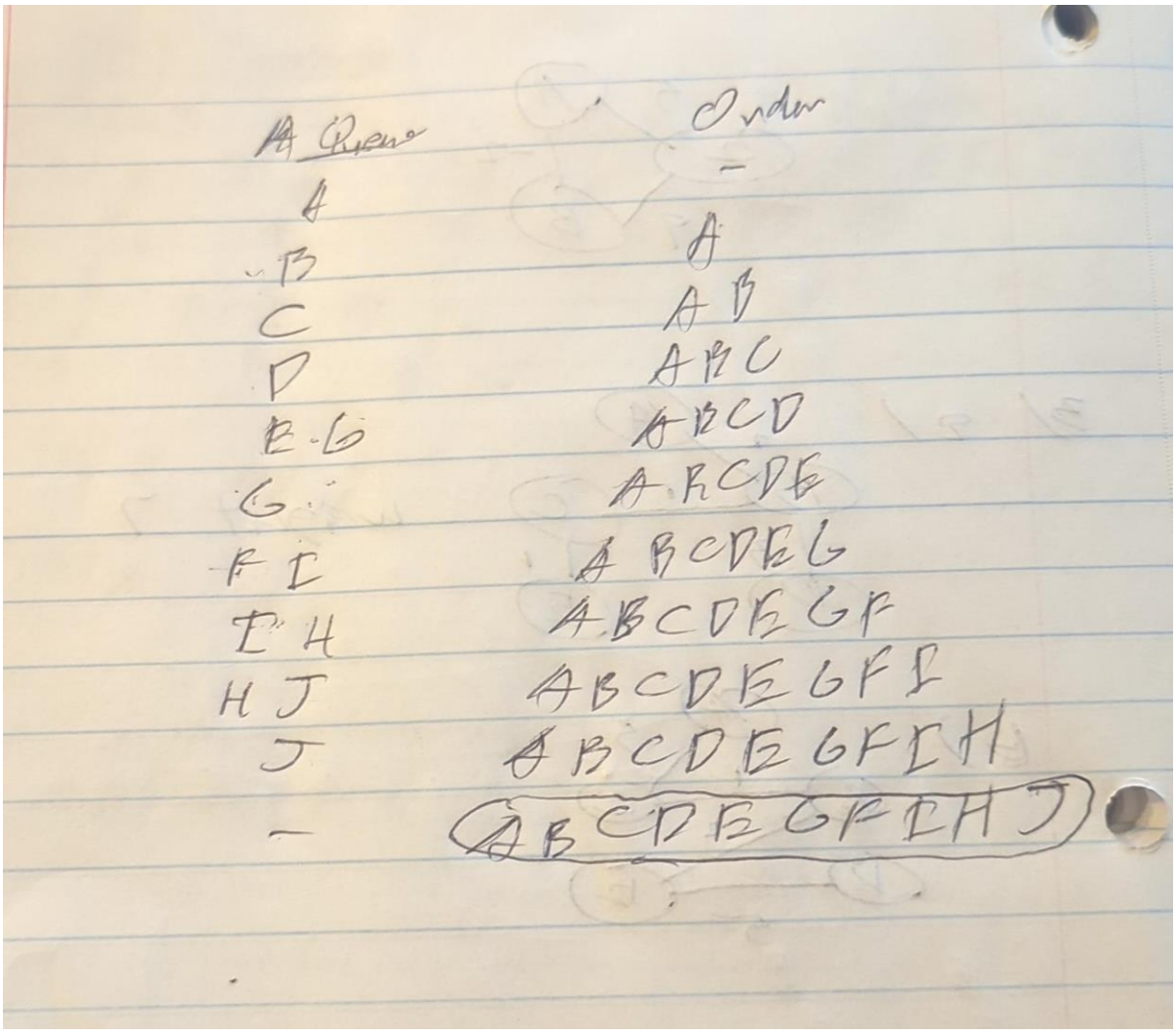
Output: ([(0, 0)], '')

6. (Extra Credit): What would be BFS and DFS traversal in below puzzle. Start at node A. Allowed movements are Up, Down, Left and Right (no diagonal).



a The BFS traversal will visit the nodes in the following order: A, B, C, D, E, G, F, I, H, J.

b The DFS traversal will visit the nodes in the following order: A, B, C, D, E, G, F, H, I, J.


A Queue
Order
A
B
C
D
E G
G
F I
E H
H J
J
A
A B
A B C
A B C D
A B C D E
A B C D E G
A B C D E G F
A B C D E G F I
A B C D E G F I H
A B C D E G F I H J

c

Stack

1) A 2) B 3) C
B A
A

4) D 5) E 6) D
C D C
B B B
A A A

7) G 8) I 9) H
D D G
B C D
A B C
F E
A

10) H
G
D
C
B
A

11) G
D
C
B
A

Order

6) ABCD F G

~~7) A B C D E G I~~

~~8) A B C D E G D H~~

~~9) A B C D E G D H F~~

8) A B C D E G F

11) A B C D E G F H I J