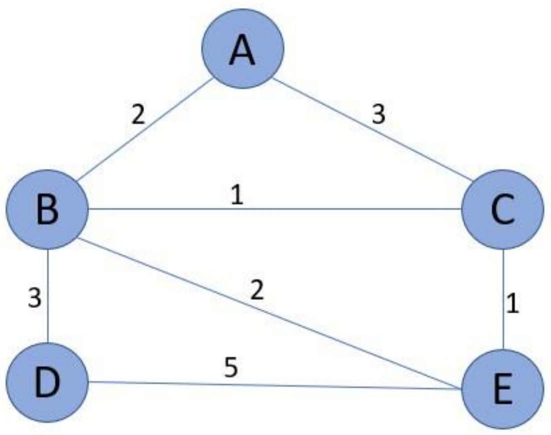
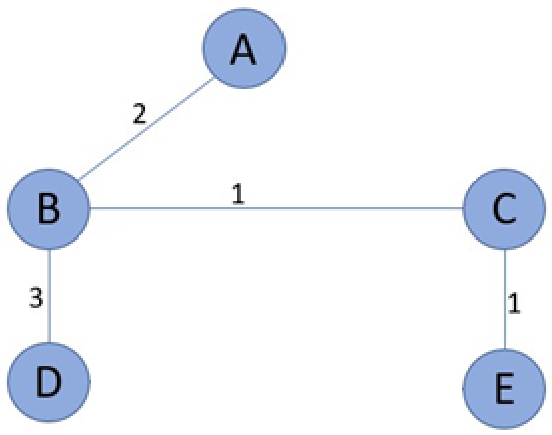
Marcos Valdez
valdemar
CS 325 Sec 400
Winter 23
03/06/2023

Homework 8: Graph Algorithms – II

1. **Draw Minimum Spanning Tree**

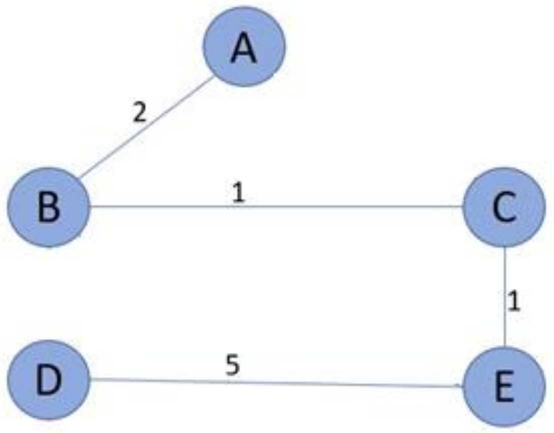


a. Draw minimum spanning tree for the above graph.



(cont. next page)

b. Draw spanning Tree that is not minimum



2. MST implementation:

a. Implement Prims' algorithm Name your function **Prims(G).** Include function in the file **MST.PY.**

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)]
```

(see MST.py on Gradescope)

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

Prim's algorithm starts with a visited region defined by an arbitrary node. At each step, the edge with the minimum weight that extends to a vertex outside the visited region is added to the MST and the vertex is added to the region. The region expands to include all visited nodes and any edges that do not extend to an unvisited vertex. This is repeated until all vertices are visited.

Kruskal's algorithm compares all edge weights and successively adds the edge with the lowest weight that does not create a cycle until no edges fit the criteria.

Differences that can be identified:

- Kruskal's algorithm will always add edges in non-descending order of weight whereas Prim's may not.
- The graph generated by Prim's algorithm will be connected at every step. The graph generated by Kruskal's may be disjoint until the algorithm is complete.
- Prim's algorithm will implicitly reject edges that create cycles because they are defined as interior to the visited region. Kruskal's must explicitly identify whether an edge creates a cycle.

3. Apply Graph traversal 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=(4, 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.

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

a. Describe an algorithm to solve the above problem.

A BFS can be used to find the shortest path. The traversable board cells are vertices and unweighted edges exist between adjacent traversable cells. The shortest paths evaluated will always be at the front of the queue and, as such, explicitly tracking the distance traversed is unnecessary.

**The below explanation of the algorithm includes the extra credit functionality.

- Create a list to store visited cells. Initialize it to a single element which is Source.
- Create an empty queue to store candidate paths. Each path will store two values: a list of cells and a direction string. Enqueue the first element containing:
    - a list containing only Source
    - an empty string
- While the queue is not empty:
    - Dequeue the first path (one of the shortest evaluated paths)
    - Extract the last element of the list of cells (the current cell) from the path
    - If the current cell is Destination, return the path.
    - For each cell adjacent to the current cell that is traversable and unvisited:
        - Add cell to visited cells
        - Make a copy of the path
        - Append the cell to the copy's list of cells
        - Append the direction to the copy's direction string
        - Enqueue the copy

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

(see Puzzle.py on Gradescope)

c. What is the time complexity of your solution?

ϴ(MN) (basic operation is performed once per cell)

d. **(Extra Credit):** For the above puzzle in addition to the output return the 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')

Implemented (see Puzzle.py on Gradescope)