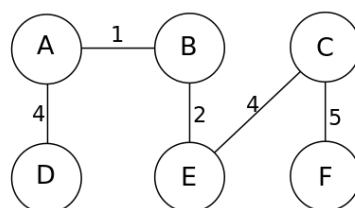
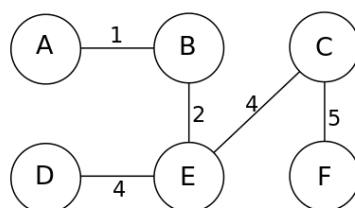
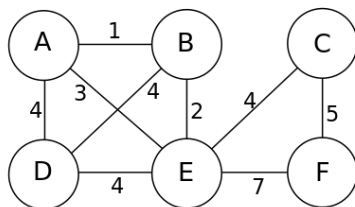


CS325

Chih Hsuan Huang

Problem1

(a) Find a graph that has multiple minimum spanning trees.



**Introduction:** Multiple Minimum Spanning. In some graphs, there may be more than one minimum spanning tree, and the trees all have the same total edge weight, but their structures may be different. In other words, if there is more than one set of edges in a weighted connected graph, these sets can form trees containing all vertices, and the sum of the edge weights of these trees is the smallest, then these trees are called multiple minimum spanning tree.

**Explain:** In the graph on the left, there are different connections connecting all the vertices, and the total weight of these connections is the same( $4+1+2+4+5=16$ ).

(From Wikipedia)

(b) Prove that any graph with distinct edge weights has a unique minimum spanning tree.

\*Assume  $G$  is a connected, weighted graph with all distinct edge weights and

Suppose  $G$  has two different minimum spanning trees  $T_1$  and  $T_2$

1. Graph  $G(V, E, W)$ , MST  $T_1$ , MST  $T_2$

\*There is at least one edge  $e$  in  $T_1$  but not in  $T_2$

2. If  $T_1 \neq T_2$ :

For each edge  $e$  in  $T_1$ :

If  $e$  is not in  $T_2$ :

\*Choose an edge  $e'$  from  $T_2$  such that adding  $e'$  to  $T_1$  would form a cycle. In this cycle, all edges except for  $e'$  are from  $T_1$

3. PutEdgeToMST( $T_2, e$ )

**Cycle C = FindCycle(T2, e)**

\*Since all edge weights of G are different, edge weights in the cycle are also different. Therefore, in this cycle, there must be an edge with the largest weight.

**4. maxWeightEdge = FindMaxWeightEdge(C)**

\*If  $e'$  is not the edge with the highest weight in this cycle, then removing the edge with the highest weight in the cycle and adding  $e'$  will result in a spanning tree that is lighter than T1 (Contradiction)

**5. If maxWeightEdge  $\neq e$ :**

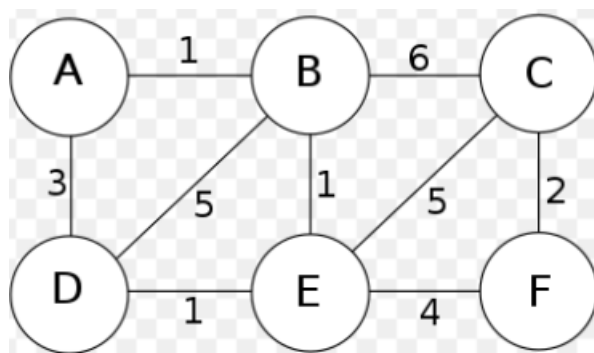
**RemoveEdgeFromMST(T1, maxWeightEdge)**

\*If  $e'$  is the edge with the largest weight in the ring, then removing  $e'$  and retaining the other edges in the ring, we will get a lighter spanning tree than T2 (Contradiction)

**6. RemoveEdgeFromMST(T2, e)**

Since the existence of T1 and T2 leads to a contradiction, the hypothesis is not valid. Therefore, the graph G cannot have two different minimum spanning trees. In other words, G must have a unique minimum spanning tree.

(c) Find a graph with non-distinct edge weights that has a unique minimum spanning tree (can you generalize (b)?).



(From Wikipedia)

We can check the minimum edge between each pair of nodes. For each pair of nodes, such as AB, BC, AD, BE, etc., we can see that the minimum weight edge between them is unique. For example, the smallest edge between nodes A and D is AD with weight 3, while the smallest edge between nodes B and E is BE with weight 1. All these edges form a minimum spanning tree, and because these edges are uniquely determined, the minimum spanning tree is also unique.

**Problem 2.**

A number maze is an  $n \times n$  grid of positive integers. A token starts in the upper left corner; your goal is to move the token to the lower-right corner. On each turn, you are allowed to move the token up, down, left, or right; the distance you may move the token is determined by the number on its current square. For example, if the token is on a square labeled 3, then you may move the token three steps up, three

steps down, three steps left, or three steps right. However, you are never allowed to move the token off the edge of the board. Describe and analyze an efficient algorithm that either returns the minimum number of moves required to solve a given number maze, or correctly reports that the maze has no solution. For example, given the number maze in the figure below, your algorithm should return the integer 8.

Time complexity: In the worst case, we may need to visit each square in the maze once. Since each square is visited at most once, the time complexity is  $O(n^2)$

\*Use the breadth-first search (BFS) algorithm

Function to solve number maze:

n = number of rows (or columns) of the maze

Create a queue Q

Create a two-dimensional array visited, initialized to False

Q enqueue (0, 0, 0) # (starting row, starting column, number of moves)

visited[0][0] = True

When Q is not empty:

(row, col, moves) = Q dequeue

# If the end point is reached, return the number of moves

If row == n-1 and col == n-1:

Return moves

# Check movement in four directions

For each direction (dr, dc) in [(-1, 0), (1, 0), (0, -1), (0, 1)]:

step = maze[row][col]

new\_row = row + dr \* step

new\_col = col + dc \* step

# Check if the new location is valid

If new\_row is at [0, n-1] and new\_col is at [0, n-1] and has not been accessed:

Q joins the team (new\_row, new\_col, moves + 1)

visited[new\_row][new\_col] = True

# If the end point cannot be reached, return no solution

Return -1