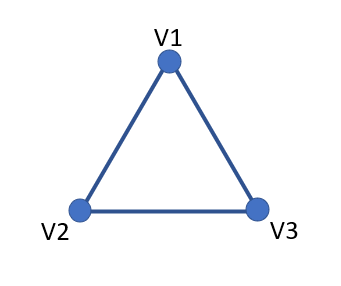
1. Find a graph that has multiple minimum spanning trees.

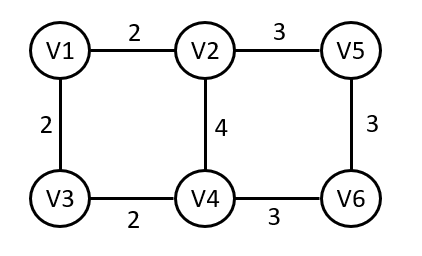
The definition of the minimum spanning tree is "connected", "acyclic", and "undirected", and it doesn't have to be the only solution. for example, we assume we have three points, V1, V2, and V3. They compose an undirected and unweighted triangle graph as below. The minimum spanning tree can be (V1->V2->V3, 2), (V1->V3->V2, 2), (V2->V1->V3, 2), (V2->V3->V1, 2), (V3->V1->V2, 2), or (V3->V2->V1, 2).

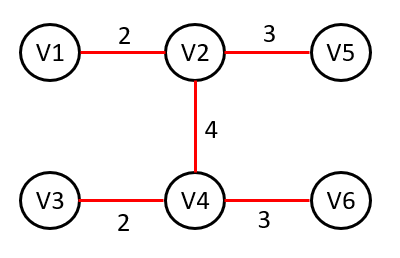
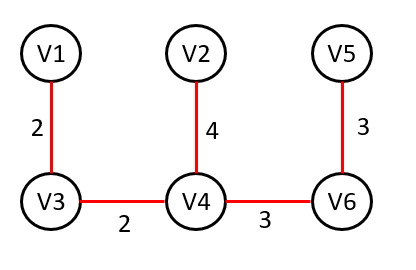
fig1

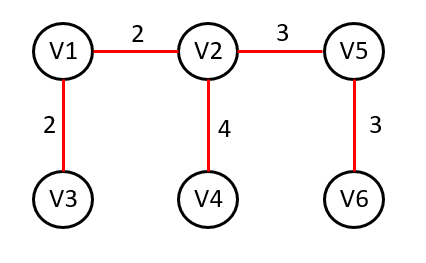
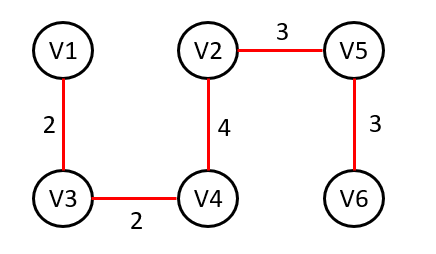
1. Prove that any graph with distinct edge weights has a unique minimum spanning tree.

It's easier to use contradiction to prove that every graph with distinct edge weights has a unique minimum spanning tree. First, we have to assume that there exist two different minimum spanning trees MST1 and MST2 for the same graph. And because MST1 and MST2 are different, there must exist at least one edge that is in MST1 but not in MST2 (or in MST2 but not in MST1), we name it edge1. Second, based on our assumption, edge1 is the edge that only exists in MST1 but not in MST2, so we can add the edge1 to MST2, However, both MST1 and MST2 are minimum spanning trees, so they have to satisfy the rule, "If a tree has n vertices, then it has n-1 edges". we have to find an edge that only exists in MST2 but not in MST1, and we name it edge2. After we add one edge1 to MST2, we have to remove edge2 to satisfy the rule, and we assume the graph has distinct edge weights. So, the sum of the weights for MST1 and MST2 has been changed, which contradicts our assumption that they have the same sum of the weights. So that we can prove that every graph with distinct edge weights has a unique minimum spanning tree.

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

For problem (b), we assume all the edges of the graph are distinct weights to prove that all the MSTs are unique. and for problem (c), there could be that edge1 and edge2 are the same value, and the proof for the problem (b) will no longer exist. So, there might exist different MSTs for the same graph. If the graph exists the situation that at least two same edges value in the cycle, there will be at least two different MSTs. But if the same edge value is not in the cycle, there will be only a unique MST. Example as below. fig2 (graph with same edges value in a cycle)

fig3 (MST 1)  fig4 (MST 2)

fig5 (MST 3)  fig6 (MST 4)

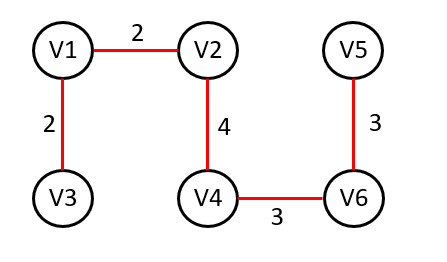
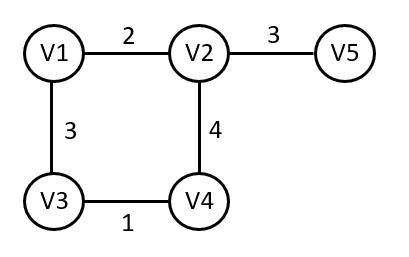
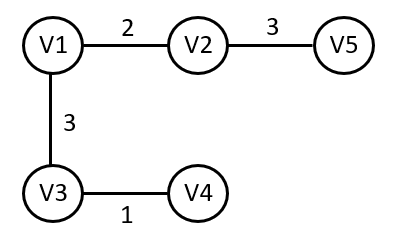
fig7 (MST 5)

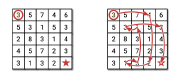
Fig3, fig4, fig5, fig6, fig7 are the possibilities MSTs for fig2. (same value happened in the cycle)

fig8 (graph with same edges value not in the cycle)

fig9 (MST)

And fig9 is the only MST in fig8.

1. A number maze is an n \* 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.



To efficiently solve the maze searching like problem (d), the suitable algorithm is breadth-first search (BFS). It's because BFS will check all the neighbor nodes before moving to the next level. Before we move to the algorithm, let's describe the concept that how to solve it with BFS briefly.

1. Initial a queue for BFS.

2. Enqueue the initial unit. (Coordinates and the distance = 0)

3. Create an array to store the track of visited elements.

4. If the queue is not empty, dequeue and check the neighbors, if the neighbors are in the boundary and not visited, enqueue it after adding 1 (the distance doesn't count itself).

5. Iterative the procedure until it reaches the goal or has searched the whole elements (If reaches the goal, return distance, if the queue is empty and hasn't reached the goal, return no solution).

Code:

from collections import deque

def min\_moves\_to\_solve\_maze(maze):

n = len(maze)

if maze[0][0] == 0 or maze[n - 1][n - 1] == 0: # Check if the start or end cell is blocked

return -1

queue = deque([(0, 0, 0)]) # (row, col, distance)

visited = set([(0, 0)])

while queue:

row, col, distance = queue.popleft()

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

return distance

moves = maze[row][col]

for dr, dc in [(0, 1), (0, -1), (1, 0), (-1, 0)]:

nr, nc = row + dr \* moves, col + dc \* moves

if 0 <= nr < n and 0 <= nc < n and (nr, nc) not in visited:

visited.add((nr, nc))

queue.append((nr, nc, distance + 1))

return -1 # No solution

# Example usage:

maze = [

[3, 5, 7, 4, 6],

[5, 3, 1, 5, 3],

[2, 8, 3, 1, 4],

[4, 5, 7, 2, 3],

[3, 1, 3, 2, 1]

]

print(min\_moves\_to\_solve\_maze(maze))