COMP S265F Design and Analysis of Algorithms
Lab 11: Minimum Spanning Tree: Kruskal's Algorithm

In this lab, we will apply the Kruskal's algorithm to solve a LeetCode problem "1489. Find Critical and Pseudo-Critical Edges in Minimum Spanning Tree": https://leetcode.com/problems/find-critical-and-pseudo-critical-edges-in-minimum-spanning-tree/

1. The problem

Given a weighted undirected connected graph with n vertices numbered from 0 to $n-1$, and an array edges where `edges[`i`] = [a`$_i$`, b`$_i$`, weight`$_i$`]` represents a bidirectional and weighted edge between nodes a_i and b_i. A minimum spanning tree (MST) is a subset of the graph's edges that connects all vertices without cycles and with the minimum possible total edge weight. *Note that a graph may have more than one MST.*

Find all the *critical* and *pseudo-critical* edges in the given graph's minimum spanning tree (MST):

- A *critical edge* is an edge whose deletion from the graph would cause the MST weight to increase.
- A *pseudo-critical edge* is an edge which can appear in some MSTs but not all.

Note that you can return the indices of the edges in any order.

```
class Solution:
    def findCriticalAndPseudoCriticalEdges(self, n: int, edges: List[List[int]]) ->
        List[List[int]]:
```

The problem has given the following examples and constraints:

- **Example 1.**
  Input: `n = 5, edges = [[0,1,1],[1,2,1],[2,3,2],[0,3,2],[0,4,3],[3,4,3],[1,4,6]]`
  Output: `[[0,1],[2,3,4,5]]`
- **Example 2.**
  Input: `n = 4, edges = [[0,1,1],[1,2,1],[2,3,1],[0,3,1]]`
  Output: `[[],[0,1,2,3]]`

Constraints:

- `2 <= n <= 100`
- `1 <= edges.length <= min(200, n * (n - 1) / 2)`
- `edges[`i`].length == 3`
- `0 <= a`$_i$` < b`$_i$` < n`
- `1 <= weight`$_i$` <= 1000`
- All pairs `(a`$_i$`, b`$_i$`)` are distinct.

2. Implementing the Kruskal's algorithm

A necessary component to solve the problem is an implementation of the Kruskal's algorithm. In the given program `mst.py`, the function `mst` takes a list of edges sorted in ascending order of weights, and returns the total weight of the MST found by the Kruskal's algorithm. The edge is in the format `[`a`, `b`, weight]`, which indicates an undirected edge (a, b) with a weight of `weight`,

To this end, we can use the data structure for the *Union-Find Disjoint Sets*:

- The function `initArrays(n)` initializes the three lists `T, next, size` for disjoint sets with `n` vertices.
- The function `findSet(x)` returns the set name of vertex `x`.
- The function `union(a, b)` merges the two disjoint sets containing vertices `a` and `b`, respectively. The name of the merged set is the name of the set containing `a`.

- The function `unionBySize(a, b)` merges the two disjoint sets containing vertices `a` and `b`, respectively, by changing the name of the smaller set. It calls the function `union`.

mst.py

```
1  class Solution:
2      def initArrays(self, n):
3          self.T = list(range(n))
4          self.next = list(range(n))
5          self.size = [1 for i in range(n)]
6
7      def findSet(self, x):
8          return self.T[x]
9
10     # update the name of the set containing b
11     def union(self, a, b):
12         i = b
13         while True:
14             self.T[i] = self.T[a]
15             i = self.next[i]
16             if i == b:
17                 break
18         self.next[a], self.next[b] = self.next[b], self.next[a]
19
20     def unionBySize(self, a, h):
21         if self.size[a] > self.size[h]:
22             self.union(a, h)
23             self.size[a] += self.size[h]
24         else:
25             self.union(h, a)
26             self.size[h] += self.size[a]
27
28     def mst(self, edges):
29         # return the total weight of the minimum spanning tree of edges
30         # to be completed by you
31
32 if __name__ == '__main__':
33     s = Solution()
34     n = 5
35     # edge is in the format [a, b, weight]
36     edges = [[0,1,1],[1,2,1],[2,3,2],[0,3,2],[0,4,3],[3,4,3],[1,4,6]]
37
38     # Sort according to weight
39     edges.sort(key = lambda x: x[-1])
40     s.initArrays(n)
41     print(s.mst(edges)) # answer = 7
```

Task 1. Given the inputs `n` and `edges` in Example 1, we first sort the edges of `edges` in ascending order of weights (line 39). Then, we initialize the *Union-Find Disjoint Sets* (line 40). Now, complete the function `mst` such that it returns a total weight 7 of the MST in Example 1.

3. Problem formulation

We can use the Kruskal's algorithm to find the total weight W of the MST of the graph. Then, we can classify each edge e as critcal, pseudo-critical, or none of them, as follows:

1. Check whether an MST can be obtained if edge e must be included. We can modify Kruskal's algorithm by first adding edge e to the solution. If the total weight of the new "MST" equals W, then edge e may be critical or pseudo-critical. Otherwise, we can conclude that edge e must not be any of these types.

2. To check whether edge e is critical or not, we check whether an MST can be obtained if edge e is omitted. If the total weight of the new "MST" equals W, then edge e is not necessary and is therefore pseudo-critical. Otherwise, edge e is critical.

Since the function `findCriticalAndPseudoCriticalEdges` needs to return the edges' indices in `edges`, we use `enumerate(edges)` with list comprehension to include the index `ind` to each edge such that after sorting the edges by weight, the indices will not be lost.

```python
# edge is in the format [a, b, weight]
edges = [[0,1,1],[1,2,1],[2,3,2],[0,3,2],[0,4,3],[3,4,3],[1,4,6]]

# edge is in the format [0-based index, a, b, weight]
sorted_edges = [[ind]+edge for ind, edge in enumerate(edges)]

# Sort according to weight
sorted_edges.sort(key = lambda x: x[-1])
```

Task 2. Complete the `findCriticalAndPseudoCriticalEdges` in the given `lab11.py`.

lab11.py

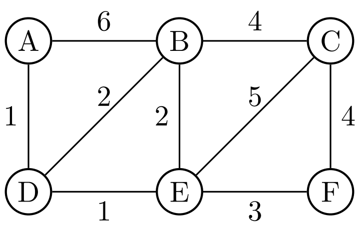
```python
from typing import List

class Solution:
    def initArrays(self, n):
        self.T = list(range(n))
        self.next = list(range(n))
        self.size = [1 for i in range(n)]

    def findSet(self, x):
        return self.T[x]

    # update the name of the set containing b
    def union(self, a, b):
        i = b
        while True:
            self.T[i] = self.T[a]
            i = self.next[i]
            if i == b:
                break
        self.next[a], self.next[b] = self.next[b], self.next[a]

    def unionBySize(self, a, h):
        if self.size[a] > self.size[h]:
            self.union(a, h)
            self.size[a] += self.size[h]
        else:
            self.union(h, a)
            self.size[h] += self.size[a]

    def mst(self, edges):
        mst_weight = 0
        for ind, a, b, w in edges: # ind has been added
            if self.findSet(a) != self.findSet(b):
                mst_weight += w
                self.unionBySize(a, b)
        return mst_weight

    def findCriticalAndPseudoCriticalEdges(self, n: int, edges: List[List[int]]) ->
        List[List[int]]:
```

```python
        # edge is in the format [index, a, b, weight]
        sorted_edges = [[ind]+edge for ind, edge in enumerate(edges)]
        # Sort according to weight
        sorted_edges.sort(key = lambda x: x[-1])

        self.initArrays(n)
        mst_weight = self.mst(sorted_edges)

        c = [] # critical edge indices
        pc = [] # pseudo-critical edge indices
        for i, edge in enumerate(sorted_edges):
            ind, a, b, w = edge # current edge
            new_edges = sorted_edges[:i] + sorted_edges[i+1:] # omit current edge

            # Step 1: Check whether MST can be obtained if current edge must be included
            # The new "MST" has total weight = new_weight
            # to be completed by you

            if new_weight != mst_weight: # the current edge is not useful for MST
                continue

            # Step 2: Check whether MST can be obtained if current edge is omitted
            # The new "MST" has total weight = new_weight
            # to be completed by you

            if new_weight != mst_weight: # the current edge is critical
                c.append(ind)
            else: # the current edge is pseudo-critical
                pc.append(ind)
        return c, pc

if __name__ == '__main__':
    s = Solution()
    n = 5
    # edge is in the format [a, b, weight]
    edges = [[0,1,1],[1,2,1],[2,3,2],[0,3,2],[0,4,3],[3,4,3],[1,4,6]]
    print(s.findCriticalAndPseudoCriticalEdges(n, edges))
```

4. Exercises

Question 1. Given a undirected connected graph with n vertices, state the number of edges in its minimum spanning tree.

Question 2. Given the following weighted undirected graph G:



Use the Kruskal's algorithm to find a minimum spanning tree (MST) of G. Show in each step, the edge considered by the Kruskal's algorithm and whether the edge is included in the resultant MST, and then draw the resultant MST.

Question 3. A *cut* $(S, V - S)$ of an undirected graph $G = (V, E)$ is a partition of V. We say that an edge $(u, v) \in E$ crosses the cut $(S, V - S)$ if one of its endpoints is in S and the other endpoint is in $V - S$. An edge is a *light edge* crossing a cut if its weight is the minimum of any edge crossing the cut. Note that there can be more than one light edge crossing a cut in the case of ties.

Show that if an edge (u, v) is contained in some minimum spanning tree, then it is a light edge crossing some cut of the graph G.