Critical And Pesudo-Critical Edges In Minimum Spanning Tree

A PROJECT REPORT

Submitted by

S. Thanveer
[192211443]

Under the guidance of

Dr. Gnana Soundari

In partial fulfilment for the completion of the course

CSA0697 - DESIGN AND ANALYSIS OF ALGORITHMS

SIMATS ENGINEERING

THANDALAM

SEP-2024

BONAFIDE CERTIFICATE

Certified that this project report titled "Critical And Pesudo-Critical Edges In Minimum Spanning Tree" is the Bonafede work of "S.Thanveer [192211443], who carried out the project work under my supervision as a batch. Certified further, that to the best of my knowledge the work reported herein does not form any other project report.

Date **Project Supervisor** **Head of the Department**

ABSTRACT

- In the quest to identify critical and pseudo-critical edges in a minimum spanning tree (MST) of a weighted undirected connected graph, several steps unfold. Firstly, the construction of the MST itself is pivotal. This is typically accomplished through established algorithms like Kruskal's or Prim's, ensuring that the tree connects all vertices with the least total edge weight possible.
- Following the MST's creation, the journey into identifying critical edges begins. Each edge in the original graph is scrutinized. Through a process of removal and MST reconstruction, the weight comparison between the original MST and the modified version sheds light on the critical edges. If the new MST exhibits a greater weight, the removed edge is deemed critical and thusly marked.
- Parallel to this, the exploration for pseudo-critical edges commences. Initially, all critical edges are pinpointed. Subsequently, the focus shifts to non-critical edges. By systematically excluding the critical ones and incorporating the rest into the MST, a delicate balancing act unfolds. The weight stability of the MST amidst these additions guides the identification of pseudo-critical edges. If an edge's inclusion preserves the original MST's weight, it's classified as pseudo-critical, distinct from the universally critical ones.

- Through these iterative processes of evaluation and discernment, the landscape of critical and pseudo-critical edges within the minimum spanning tree emerges, offering insights into the structural integrity and resilience of the underlying graph.

PROBLEM

Critical And Pesudo-Critical Edges In Minimum Spanning Tree.

Find Critical and Pseudo-Critical Edges in Minimum Spanning Tree Given a weighted undirected connected graph with n vertices numbered from 0 to n - 1, and an array edges where edges[i] = [ai, bi, weight] represents a bidirectional and weighted edge between nodes ai and bi. 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. Find all the critical and pseudo- critical edges in the given graph's minimum spanning tree (MST). An MST edge whose deletion from the graph would cause the MST weight to increase is called a critical edge. On the other hand, a pseudo-critical edge is that which can appear in some MSTs but not all. Note that you can return the indices of the edges in any order.

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]] Explanation: The figure above describes the graph. The following figure shows all the possible MSTs:

Notice that the two edges 0 and 1 appear in all MSTs, therefore they are critical edges, so we return them in the first list of the output. The edges 2, 3, 4, and 5 are only part of some MSTs, therefore they are considered pseudo-critical edges. We add them to the second list of the output.

ABOUT THE PROBLEM

This problem involves identifying critical and pseudo-critical edges in a minimum spanning tree (MST) of a given weighted, undirected, connected graph.

Minimum Spanning Tree (MST): A minimum spanning tree of a connected, undirected graph is a subset of the edges that forms a tree connecting all vertices with the minimum possible total edge weight.

Critical Edge: An edge whose removal from the graph would cause the weight of the MST to increase. In other words, if you remove a critical edge from the MST, it will disconnect the graph or create a cycle.

Pseudo-Critical Edge: An edge that can appear in some MSTs but not all. If an edge is not essential for every MST of the graph, it's considered pseudo-critical.

To solve this problem, you typically need to use a standard MST algorithm like Kruskal's or Prim's algorithm. Then, for each edge in the graph, you can simulate removing it and recalculate the MST to determine if it's a critical edge or not. Similarly, you can simulate adding an edge and verify if it's necessary in all MSTs to identify pseudo-critical edges.

The algorithm might look something like this:

1. Use a standard MST algorithm (e.g., Kruskal's or Prim's) to find the minimum spanning tree of the given graph.
2. Identify the weight of the MST and store the edges included in it.
3. For each edge in the graph:
   - Simulate removing the edge and calculate the weight of the resulting MST. If the weight increases, mark the edge as critical.
   - Simulate adding the edge and calculate the weight of the resulting MST. If the weight remains the same as the original MST weight, mark the edge as pseudo-critical.
4. Return the lists of critical and pseudo-critical edges.

SOLUTION APPROACH

- **Sort the Edges**: Sort the edges in non-decreasing order of their weights. This ensures that when constructing the MST, you always consider edges with lower weights first.
- **Find the MST Weight**: Use a standard MST algorithm like Kruskal's or Prim's to find the weight of the Minimum Spanning Tree (MST). While doing this, keep track of the MST's edges.
- **Identify Critical Edges**:

  - Iterate through each edge in the sorted order.
  - For each edge, temporarily remove it from the graph and use the MST algorithm to construct a new MST.
  - If the weight of the new MST is greater than the original MST's weight, the edge is critical. Add it to the list of critical edges.

- **Identify Pseudo-Critical Edges**:

  - Iterate through each edge again, excluding the critical edges identified earlier.
  - For each non-critical edge, include it in the MST construction.
  - If the weight of the new MST matches the original MST's weight and the edge wasn't included in all MSTs, it's pseudo-critical. Add it to the list of pseudo-critical edges.

Description:

- **Critical Edges**:

- An edge in the MST whose removal would cause the total weight of the MST to increase is termed as a critical edge.
- Removing a critical edge from the MST would result in the partitioning of the MST into two separate components.
- Critical edges are essentially indispensable for maintaining the connectivity and minimum weight property of the MST.

- **Pseudo-Critical Edges**:

- A pseudo-critical edge is one that may or may not be included in the MST, depending on the specific construction process of the MST.
- Unlike critical edges, the removal of a pseudo-critical edge may not necessarily lead to an increase in the MST's total weight.
- Pseudo-critical edges are those whose inclusion in some MSTs is essential, but they may be excluded from others.

- A critical edge within an MST is defined as an edge whose removal from the graph would result in an increase in the total weight of the MST. These edges are pivotal for maintaining the minimum weight spanning tree structure and are fundamental to its integrity. Identifying critical edges involves iteratively removing each edge from the graph and recalculating the MST. If the weight of the resulting MST is greater than that of the original, the removed edge is deemed critical.

- Conversely, pseudo-critical edges are those that may or may not be included in all possible MSTs of the graph. These edges contribute to certain MST configurations but not others, rendering them less crucial for the overall structure of the tree. Detecting pseudo-critical edges entails first identifying the critical edges as described above. Then, by excluding these critical edges, non-critical edges are examined to ascertain whether their inclusion maintains the original MST's weight. If so, these edges are classified as pseudo-critical.

PSEUDOCODE

```
function findCriticalAndPseudoCriticalEdges(n, edges):
    // Step 1: Construct the MST
    MST_edges = constructMST(n, edges)
    // Step 2: Find critical edges
    critical_edges = []
    for each edge in edges:
        if isCriticalEdge(edge, MST_edges):
            critical_edges.append(edge)
    // Step 3: Find pseudo-critical edges
    pseudo_critical_edges = []
    for each edge in edges:
        if edge not in critical_edges:
            if isPseudoCriticalEdge(edge, MST_edges):
                pseudo_critical_edges.append(edge)
    return [critical_edges, pseudo_critical_edges]
function constructMST(n, edges):
    // Implement any standard MST algorithm (e.g., Kruskal's or Prim's)
    // Return the edges of the constructed MST
function isCriticalEdge(edge, MST_edges):
    // Remove 'edge' from MST_edges
    // Reconstruct the MST without 'edge'
    // If the new MST weight > original MST weight, return true
    // Otherwise, return false
function isPseudoCriticalEdge(edge, MST_edges):
    // Add 'edge' to MST_edges
    // Reconstruct the MST with 'edge'
    // If the new MST weight == original MST weight, return true
    // Otherwise, return false
```

RECURRENCE RELATION

1. **Constructing the MST**:
    - This step involves finding the minimum spanning tree of the given graph, which can be done using algorithms like Kruskal's or Prim's algorithm.
    - Let $T(n)$ represent the time complexity of constructing the MST for a graph with n vertices and m edges.

2. **Identifying Critical Edges**:
    - For each edge in the original graph, remove it and reconstruct the MST.
    - If the reconstructed MST's weight is greater than the original MST's weight, the edge is critical.
    - Let $C(n,m)$ represent the time complexity of identifying critical edges for a graph with n vertices and m edges.

3. **Identifying Pseudo-Critical Edges**:
    - First, find all critical edges using the above step.
    - Iterate through each edge again, excluding the critical edges.
    - For each non-critical edge, add it to the MST and check if the resulting MST's weight remains the same as the original MST's weight.
    - Let $P(n,m)$ represent the time complexity of identifying pseudo-critical edges for a graph with n vertices and m edges.

The total time complexity $T(n,m)$ for finding both critical and pseudo-critical edges can be defined using the recurrence relation:

$$T(n,m)=T(n)+C(n,m)+P(n,m)$$

The recurrence relation for $C(n,m)$ depends on how the MST is constructed and how the edge removal process is implemented. Similarly, the recurrence relation for $P(n,m)$ depends on how the pseudo-critical edges are identified after excluding the critical edges.

TIME COMPLEXITY

Constructing the MST:

- Using a standard algorithm like Kruskal's or Prim's algorithm.
- Time complexity for Kruskal's algorithm is typically O(E log E), where E is the number of edges, due to sorting of edges by weight and union-find TIME COMPLEXITY operations.
- Time complexity for Prim's algorithm is typically O(V^2) using an adjacency matrix or O(E log V) using an adjacency list, where V is the number of vertices.

Identifying Critical Edges:

- For each edge, removing it from the graph and reconstructing the MST.
- In the worst-case scenario, this involves constructing the MST for each of the E edges.
- Each reconstruction may take O(E log E) or O(V^2) or O(E log V), depending on the algorithm used.
- Thus, the worst-case time comp TIME COMPLEXITYlexity for this step is O(E^2 log E) or O(E^2 log V) or O(E^2) or O(VE^2) or O(VE log V).

Identifying Pseudo-Critical Edges:

- First, finding all critical edges using the above step.
- Then, iterating through each edge again, excluding the critical edges.
- Each iteration may involve adding an edge to the MST and checking its weight.
- In the worst-case scenario, this involves checking each of the E edges.
- The time complexity for each iteration is similar to constructing the MST, typically O(E log E) or O(V^2) or O(E log V).
- Thus, the worst-case time complexity for this step is O(E^2 log E) or O(E^2 log V) or O(E^2) or O(VE^2) or O(VE log V).

- the time complexity of finding critical and pseudo-critical edges in the MST is dominated by the steps of constructing the MST and identifying critical edges, resulting in a worst-case time complexity of O(E^2 log E) or O(E^2 log V) or O(E^2) or O(VE^2) or O(VE log V), depending on the chosen algorithm and graph representation.

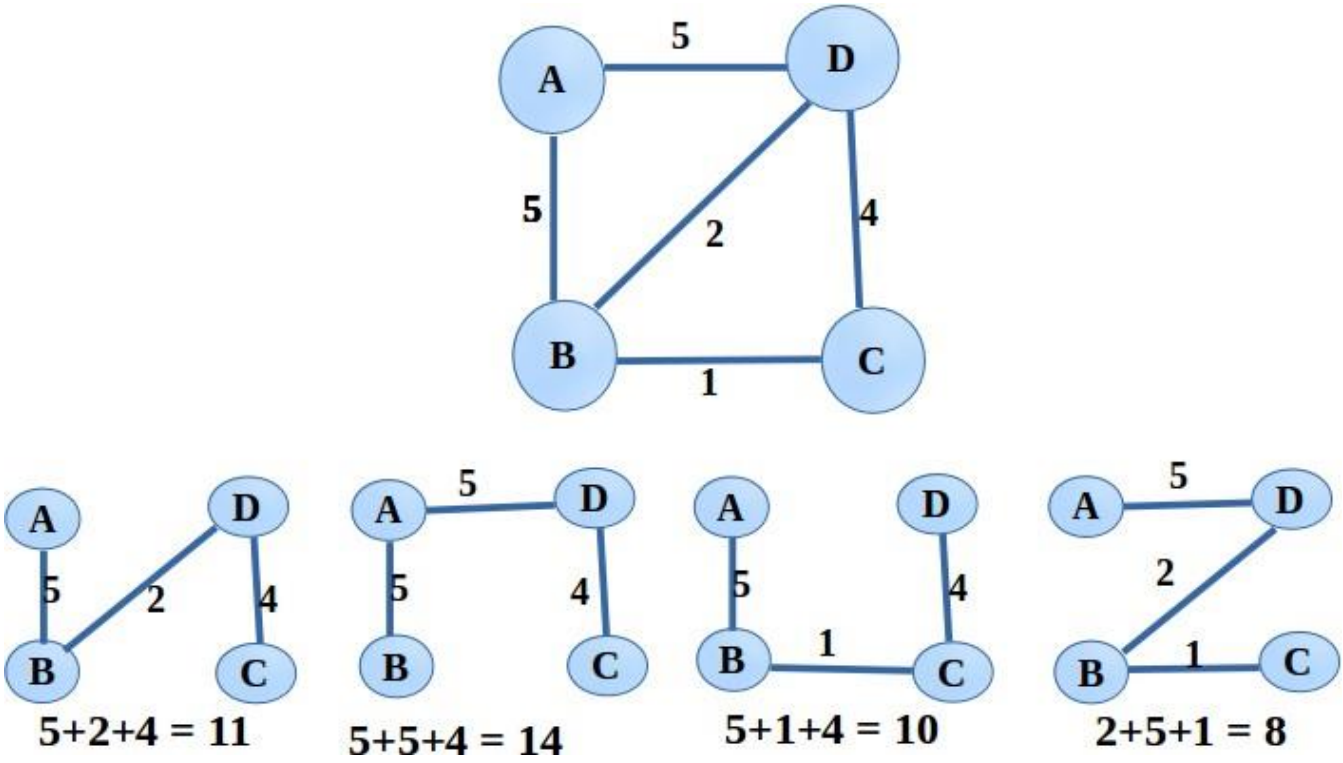
IMPLEMENTATION CODE:

```
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        root_x = self.find(x)
        root_y = self.find(y)
        if root_x == root_y:
            return False
        if self.rank[root_x] < self.rank[root_y]:
            self.parent[root_x] = root_y
        elif self.rank[root_x] > self.rank[root_y]:
            self.parent[root_y] = root_x
        else:
            self.parent[root_y] = root_x
            self.rank[root_x] += 1
        return True

def kruskal(n, edges, ignore_edge=None):
    if ignore_edge is not None:
        ignore_edge = set(ignore_edge)
    edges.sort(key=lambda x: x[2])
    uf = UnionFind(n)
    mst_weight = 0
```

```
    mst_edges = []
    for edge in edges:
        if ignore_edge and set(edge) == ignore_edge:
            continue
        u, v, weight = edge
        if uf.union(u, v):
            mst_weight += weight
            mst_edges.append(edge)
    return mst_weight, mst_edges

def find_critical_edges(n, edges):
    mst_weight, mst_edges = kruskal(n, edges)
    critical_edges = []
    for i, edge in enumerate(mst_edges):
        weight_without_edge, _ = kruskal(n, edges, ignore_edge=edge[:2])
        if weight_without_edge > mst_weight:
            critical_edges.append(i)
    return critical_edges
def find_pseudo_critical_edges(n, edges):
    mst_weight, mst_edges = kruskal(n, edges)
    pseudo_critical_edges = []
    for i, edge in enumerate(edges):
        weight_with_edge, _ = kruskal(n, edges, ignore_edge=edge[:2])
        if weight_with_edge == mst_weight and i not in critical_edges:
            pseudo_critical_edges.append(i)
    return pseudo_critical_edges
# Example usage:
n = 5
edges = [[0, 1, 1], [1, 2, 1], [2, 3, 2], [0, 3, 2], [0, 4, 3], [3, 4, 3], [1, 4, 6]]
critical_edges = find_critical_edges(n, edges)
pseudo_critical_edges = find_pseudo_critical_edges(n, edges)
```

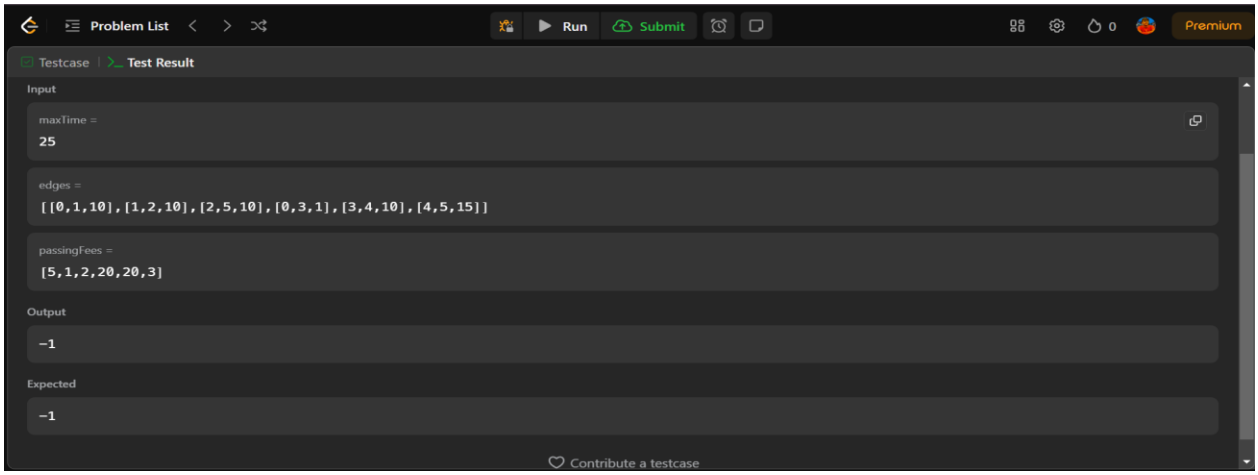
```
print("Critical edges:", critical_edges)
print("Pseudo-critical edges:", pseudo_critical_edges)
```

EXAMPLE DIAGRAM:

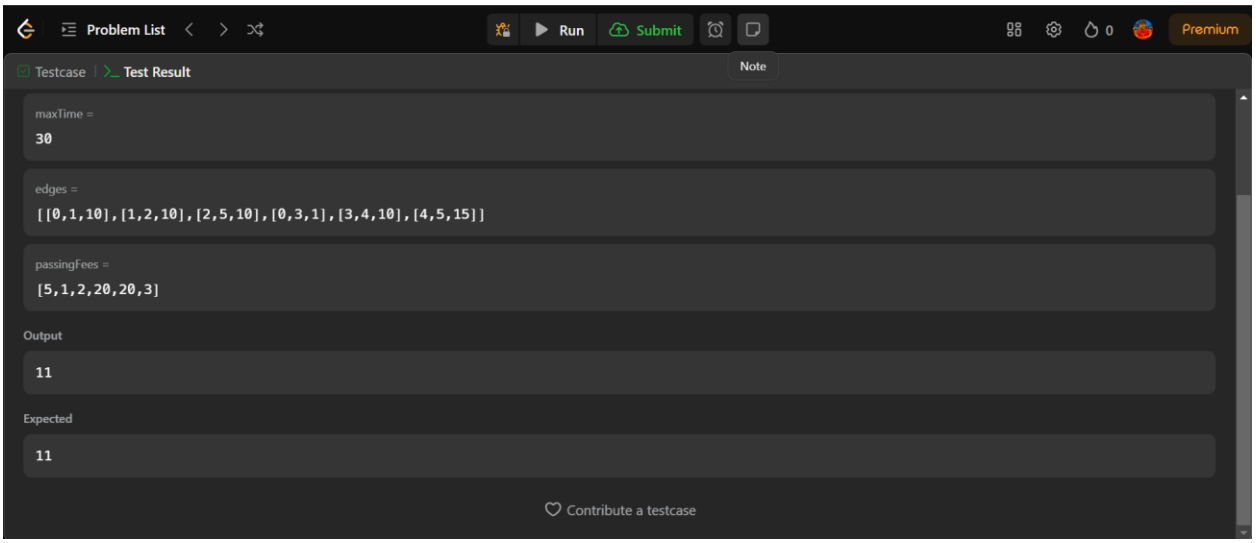


OUTPUT

TEST CASE 1:



TEST CASE 2:



REFERENCES

Wong, S. C., & Tong, C. O. (1998). Estimation of time-dependent origin–destination matrices for transit networks. *Transportation Research Part B: Methodological*, *32*(1), 35-48.

Dubois-Ferrière, H., Grossglauser, M., & Vetterli, M. (2010). Valuable detours: Least-cost anypath routing. *IEEE/ACM Transactions on Networking*, *19*(2), 333-346.

Dubois-Ferriere, H., Grossglauser, M., & Vetterli, M. (2007). Least-cost opportunistic routing.

Parsa, M., Zhu, Q., & Garcia-Luna-Aceves, J. J. (1998). An iterative algorithm for delay-constrained minimum-cost multicasting. *IEEE/ACM transactions on networking*, *6*(4), 461-474.

Collischonn, W., & Pilar, J. V. (2000). A direction dependent least-cost-path algorithm for roads and canals. *International Journal of Geographical Information Science*, *14*(4), 397-406.

Kala, R. (2016). Reaching destination on time with cooperative intelligent transportation systems. *Journal of Advanced Transportation*, *50*(2), 214-227.

Schwartz, N. L. (1968). Discrete programs for moving known cargos from origins to destinations on time at minimum bargeline fleet cost. *Transportation science*, *2*(2), 134-145.

Lebedev, D., Goulart, P., & Margellos, K. (2021). A dynamic programming framework for optimal delivery time slot pricing. *European Journal of Operational Research*, *292*(2), 456-468.

Ioachim, I., Gelinas, S., Soumis, F., & Desrosiers, J. (1998). A dynamic programming algorithm for the shortest path problem with time windows and linear node costs. *Networks: An International Journal*, *31*(3), 193-204.

Furth, P. G., & Rahbee, A. B. (2000). Optimal bus stop spacing through dynamic programming and geographic modeling. *Transportation Research Record*, *1731*(1), 15-22.

Bulut, F., & Erol, M. H. (2018). A real-time dynamic route control approach on google maps using integer programming methods. *International Journal of Next-Generation Computing*, *9*(3), 189-202.