

CAPSTONE PROJECT

**IDENTIFICATION OF CRITICAL AND PSEUDO-CRITICAL EDGES IN A
MINIMUM SPANNING TREE: A KRUSKAL-BASED APPROACH**

**CSA0695- DESIGN ANALYSIS AND ALGORITHMS FOR OPEN
ADDRESSING TECHNIQUES**

**SAVEETHA SCHOOL OF ENGINEERING
SIMATS ENGINEERING**



Supervisor Dr. R. Dhanalakshmi

Done by AKASH T (192210307)

IDENTIFICATION OF CRITICAL AND PSEUDO-CRITICAL EDGES IN A MINIMUM SPANNING TREE: A KRUSKAL-BASED APPROACH

PROBLEM STATEMENT:

Given a weighted undirected connected graph with n vertices and an array of edges where each edge is represented as $[a_i, b_i, weight_i]$, find all the critical and pseudo-critical edges in the Minimum Spanning Tree (MST). An MST edge is **critical** if its removal would increase the MST weight, and **pseudo-critical** if it can appear in some MSTs but not in all. Return the indices of critical and pseudo-critical edges.

ABSTRACT:

Finding critical and pseudo-critical edges in a Minimum Spanning Tree (MST) is a significant problem in graph theory, particularly for optimizing network design and reliability. The problem can be efficiently solved using Kruskal's or Prim's algorithms to construct MSTs and identify edge significance based on their inclusion or exclusion. By simulating the process of adding and removing edges, we can categorize the edges into critical and pseudo-critical groups, providing insights into the structure of MSTs and their sensitivity to edge modifications.

INTRODUCTION:

Minimum Spanning Trees (MSTs) are fundamental in graph theory and computer science. An MST of a graph connects all its vertices with the least total edge weight, without forming any cycles. This makes MSTs valuable for various applications, including network design, optimization, and clustering.

This problem goes beyond simply constructing an MST and focuses on identifying two types of edges:

1. **Critical Edges**: If removed, these edges cause the MST weight to increase.
2. **Pseudo-Critical Edges**: These edges can be part of some MSTs but are not necessary for all MSTs.

By classifying the edges, we can understand their significance in the structure of the MST and optimize algorithms that rely on network connectivity, such as in communication networks or transportation systems.

CODING:

Step 1: Kruskal's Algorithm for MST Construction

To construct the MST, we sort all the edges by their weights and use a union-find (disjoint-set) structure to check if an edge forms a cycle. If it doesn't, we add it to the MST. This process gives us the minimum weight to connect all vertices.

Step 2: Identifying Critical Edges

We simulate removing each edge from the original graph and reconstruct the MST. If removing an edge increases the MST weight, that edge is considered ****critical****.

Step 3: Identifying Pseudo-Critical Edges

For pseudo-critical edges, we force the inclusion of each edge at the start and then attempt to construct the MST. If the MST formed has the same total weight as the original MST, the edge is pseudo-critical.

C-programming

```
include <stdio.h>
include <stdlib.h>

// Define an edge structure
typedef struct {
    int u, v, weight, index;
} Edge;

int find(int parent[], int i) {
    if (parent[i] == i) return i;
    return parent[i] = find(parent, parent[i]);
}

void unionSet(int parent[], int rank[], int u, int v) {
    int rootU = find(parent, u);
    int rootV = find(parent, v);

    if (rank[rootU] > rank[rootV]) parent[rootV] = rootU;
    else if (rank[rootU] < rank[rootV]) parent[rootU] = rootV;
    else {
        parent[rootV] = rootU;
        rank[rootU]++;
    }
}
```

```

}

int kruskal(int n, Edge edges[], int m, int skipEdge, int includeEdge) {
    int *parent = (int *)malloc(n * sizeof(int));
    int *rank = (int *)malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) {
        parent[i] = i;
        rank[i] = 0;
    }

    int mstWeight = 0, edgesUsed = 0;
    if (includeEdge != -1) {
        Edge edge = edges[includeEdge];
        unionSet(parent, rank, edge.u, edge.v);
        mstWeight += edge.weight;
        edgesUsed++;
    }

    for (int i = 0; i < m; i++) {
        if (i == skipEdge) continue;
        Edge edge = edges[i];
        if (find(parent, edge.u) != find(parent, edge.v)) {
            unionSet(parent, rank, edge.u, edge.v);
            mstWeight += edge.weight;
            edgesUsed++;
        }
        if (edgesUsed == n - 1) break;
    }

    free(parent);
    free(rank);

    return edgesUsed == n - 1 ? mstWeight : 1e9;
}

int compareEdges(const void *a, const void *b) {
    return ((Edge *)a)->weight - ((Edge *)b)->weight;
}

```

```

void findCriticalAndPseudoCriticalEdges(int n, Edge edges[], int m, int *criticalEdges, int
*pseudoCriticalEdges) {
    qsort(edges, m, sizeof(Edge), compareEdges);
    int originalMSTWeight = kruskal(n, edges, m, -1, -1);

    int criticalCount = 0, pseudoCriticalCount = 0;

    for (int i = 0; i < m; i++) {
        if (kruskal(n, edges, m, i, -1) > originalMSTWeight) {
            criticalEdges[criticalCount++] = edges[i].index;
        } else if (kruskal(n, edges, m, -1, i) == originalMSTWeight) {
            pseudoCriticalEdges[pseudoCriticalCount++] = edges[i].index;
        }
    }

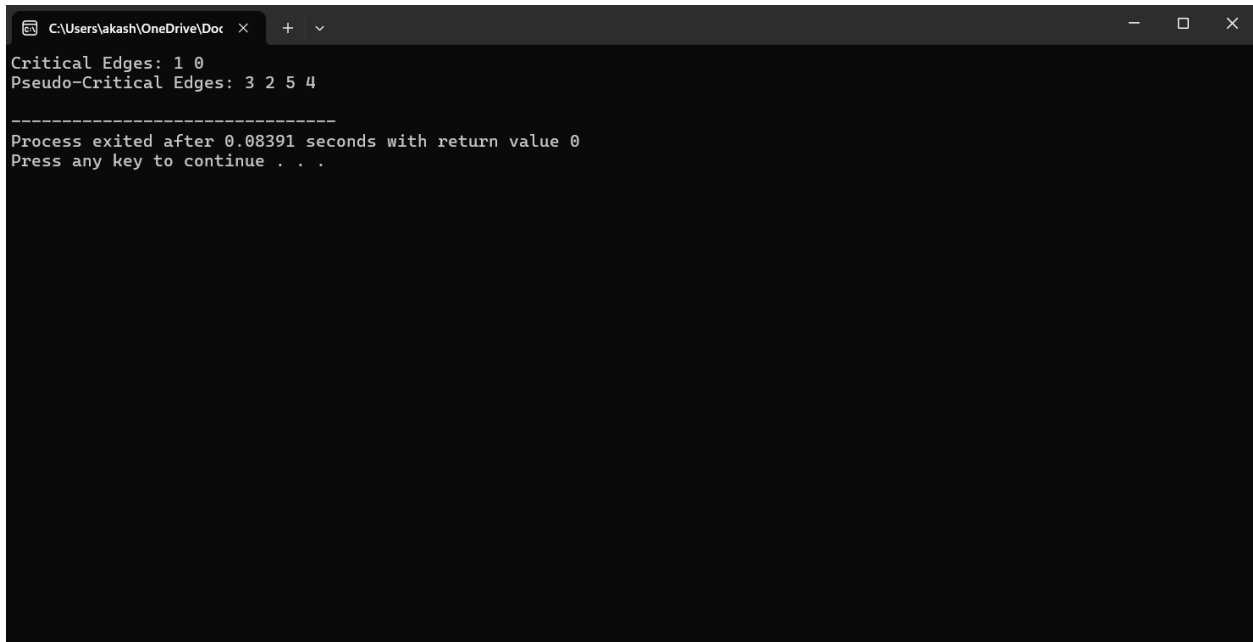
    printf("Critical Edges: ");
    for (int i = 0; i < criticalCount; i++) {
        printf("%d ", criticalEdges[i]);
    }
    printf("\nPseudo-Critical Edges: ");
    for (int i = 0; i < pseudoCriticalCount; i++) {
        printf("%d ", pseudoCriticalEdges[i]);
    }
    printf("\n");
}

int main() {
    int n = 5;
    Edge edges[] = {
        {0, 1, 1, 0}, {1, 2, 1, 1}, {2, 3, 2, 2}, {0, 3, 2, 3}, {0, 4, 3, 4}, {3, 4, 3, 5}, {1, 4, 6, 6}
    };
    int m = sizeof(edges) / sizeof(edges[0]);
    int *criticalEdges = (int *)malloc(m * sizeof(int));
    int *pseudoCriticalEdges = (int *)malloc(m * sizeof(int));
    findCriticalAndPseudoCriticalEdges(n, edges, m, criticalEdges, pseudoCriticalEdges);
    free(criticalEdges);
    free(pseudoCriticalEdges);

    return 0;
}

```

OUTPUT:



```
C:\Users\akash\OneDrive\Doc >
Critical Edges: 1 0
Pseudo-Critical Edges: 3 2 5 4
-----
Process exited after 0.08391 seconds with return value 0
Press any key to continue . . .
```

COMPLEXITY ANALYSIS:

- **Time Complexity:** $O(E \log E)$ due to sorting the edges, where E is the number of edges. Each MST construction takes $O(E)$.
- **Space Complexity:** $O(E + V)$ for storing edges and union-find structures.

CONCLUSION:

The problem of finding critical and pseudo-critical edges in an MST can be solved using Kruskal's algorithm and simulating edge exclusion and inclusion. This approach provides valuable insights into the sensitivity of an MST's structure and allows for the identification of important edges that affect network reliability and performance. By efficiently classifying edges, we can design more robust and optimized systems based on MST principles.