**SIMATS SCHOOL OF ENGINEERING**

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES**

**CHENNAI-602105**

# “Critical and Pseudo-Critical Edges in Minimum Spanning Tree "

**A CAPSTONE PROJECT REPORT**

*Submitted in the partial fulfillment for the award of the degree of*

## BACHELOR OF ENGINEERING

**IN COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE AND**

**DATA SCIENCE**

**Submitted by**

**V.SIVANJI(192211429)**

**Under the Supervision of**

**Dr. K.V KANIMOZHI**

## DECLARATION

I am V SIVANJI **,** student of **Bachelor of Engineering in**

**Computer Science Engineering** at Saveetha Institute of Medical and Technical Sciences, Saveetha University, Chennai, hereby declare that the work presented in this Capstone Project Work entitled **"** **Critical and Pseudo-Critical Edges in Minimum Spanning Tree "** is the outcome of my own bonafide work. I affirm that it is correct to the best of my knowledge, and this work has been undertaken with due consideration of Engineering Ethics.

(V.SIVANJI)

Date:20 September 2024

Place :Saveetha School of Engineering, Thandalam .

## CERTIFICATE

This is to certify that the project entitled **“Critical and Pseudo-Critical Edges in Minimum Spanning Tree”** submitted by VSIVANJI .has been carried out under my supervision. The project has been submitted as per the requirements in the current semester of B.E Computer science engineering and BTech Artificial Intelligence in Data science.

Faculty-in-charge

K.V KANIMOZI

**ABSTRACT**

The critical and pseudo-critical edges in a Minimum Spanning Tree (MST), one must first compute the MST of a given undirected, weighted graph using an algorithm like Kruskal's or Prim's. Critical edges are those that, if removed, increase the weight of the MST, meaning they are essential for maintaining the minimum cost. To identify them, each edge is temporarily removed, and the MST is recalculated to check for a higher total weight. Pseudo-critical edges, on the other hand, are edges that can appear in some valid MSTs but are not strictly necessary. They do not increase the MST's weight when removed but can still be part of another equally optimal MST. Finding these edges involves first forcing them into the MST and then checking if the overall weight remains optimal. The process typically involves sorting edges, detecting cycles, and verifying tree properties Sort edges by weight: This helps us in building the MST efficiently using Kruskal’s algorithm.

Find the MST weight: By applying Kruskal’s algorithm, find the MST using a union-find (disjoint set) data structure.

Identify critical edges:

Remove each edge one by one, and compute the MST without that edge.

If the MST weight increases or if it’s impossible to form a spanning tree, the edge is critical.

Identify pseudo-critical edges:

Force the inclusion of an edge in the MST.

If the MST weight is unchanged, the edge is pseudo-critical.

**Theoretical Framework:**

* Time Complexity: O(n log n)
* Space Complexity: O(n)
* Algorithmic Paradigms: Divide-and-Conquer, Binary Indexed Tree, Augmented Data Structures

**Keywords:**

* Array
* MINIMUM SPANING TREE
* Algorithm
* Time complexity
* Space complexity
* Data structures
* Programing

**INTRODUCTION**

Minimum Spanning Tree (MST), we can leverage the properties of MSTs and use algorithms like Kruskal's or Prim's. Here’s an introduction to the problem and the theoretical background required to approach it.

Problem Overview

We are given a connected, undirected graph with n vertices and an array edges, where each edge connects two nodes ai and bi with a specific weight weighti. The goal is to identify edges that are either critical or pseudo-critical in the Minimum Spanning Tree of this graph.

Critical edges: An edge is critical if removing it increases the total weight of the MST or prevents a spanning tree from being formed. These edges are indispensable for constructing the MST.

Pseudo-critical edges: An edge is pseudo-critical if it can be included in some MSTs but is not necessary in all MSTs. Such edges might not be part of a specific MST, but can still be part of others.

Minimum Spanning Tree (MST)

A Minimum Spanning Tree (MST) is a subset of edges that:

Connects all the vertices together (spanning).

Contains no cycles.

Has the minimum possible total edge weight among all spanning trees of the graph.

The Kruskal's algorithm is often used to build MSTs. It processes edges in increasing order of weight and adds them to the MST if they do not form a cycle (using a Union-Find or Disjoint Set Union structure to check connectivity). Another common approach is Prim's algorithm, which builds the MST by expanding a single component iteratively.

Approach to Identifying Critical and Pseudo-Critical Edges

Initial MST Construction: First, calculate the weight of the MST using Kruskal’s algorithm or any MST algorithm. This MST serves as the reference to compare how removing or forcing specific edges affects the total weight.

There are several ways to solve this problem, including:

* Brute Force: Compare each element with all elements to its right.
* Binary Indexed Tree (BIT): Use a BIT to keep track of smaller elements.
* Modified Merge Sort: Utilize the merge sort algorithm to count smaller elements.

**Applications:**

This problem has applications in various fields, such as:

* + **Network Design**:: MSTs are used in designing efficient network layouts, such as telecommunications, computer networks, and road networks, to minimize the cost of laying cables or roads..
  + **Transport and Logistics**: MSTs can optimize the routing of delivery trucks or transportation networks to minimize travel distance or costs.

**CODING**

#include <stdio.h>

#include <stdlib.h>

typedef struct {

int u, v, w;

} Edge;

int cmp(const void \*a, const void \*b) {

return ((Edge \*)a)->w - ((Edge \*)b)->w;

}

int find(int parent[], int i) {

if (parent[i] == -1)

return i;

return find(parent, parent[i]);

}

void unionSet(int parent[], int x, int y) {

int xset = find(parent, x);

int yset = find(parent, y);

if (xset != yset)

parent[xset] = yset;

}

int kruskal(Edge edges[], int n, int e, int ignoreEdge, int includeEdge) {

int parent[n];

for (int i = 0; i < n; i++)

parent[i] = -1;

if (includeEdge != -1) {

// Include a pseudo-critical edge

unionSet(parent, edges[includeEdge].u, edges[includeEdge].v);

}

int mstWeight = 0, edgesUsed = 0;

for (int i = 0; i < e; i++) {

if (i == ignoreEdge) continue;

int u = edges[i].u;

int v = edges[i].v;

if (find(parent, u) != find(parent, v)) {

mstWeight += edges[i].w;

unionSet(parent, u, v);

edgesUsed++;

}

}

// Check if we have used n-1 edges

if (edgesUsed != n - 1) return -1; // Not a valid MST

return mstWeight;

}

void findCriticalAndPseudoCriticalEdges(int n, Edge edges[], int e) {

qsort(edges, e, sizeof(Edge), cmp);

int originalMSTWeight = kruskal(edges, n, e, -1, -1);

if (originalMSTWeight == -1) {

printf("No Minimum Spanning Tree exists\n");

return;

}

for (int i = 0; i < e; i++) {

// Check if the edge is critical

if (kruskal(edges, n, e, i, -1) > originalMSTWeight) {

printf("Critical Edge: (%d, %d, %d)\n", edges[i].u, edges[i].v, edges[i].w);

} else if (kruskal(edges, n, e, -1, i) == originalMSTWeight) {

// Check if the edge is pseudo-critical

printf("Pseudo-Critical Edge: (%d, %d, %d)\n", edges[i].u, edges[i].v, edges[i].w);

}

}

}

int main() {

int n = 5; // Number of vertices

int e = 7; // Number of edges

Edge edges[] = {

{0, 1, 10},

{0, 2, 6},

{0, 3, 5},

{1, 3, 15},

{2, 3, 4},

{1, 2, 5},

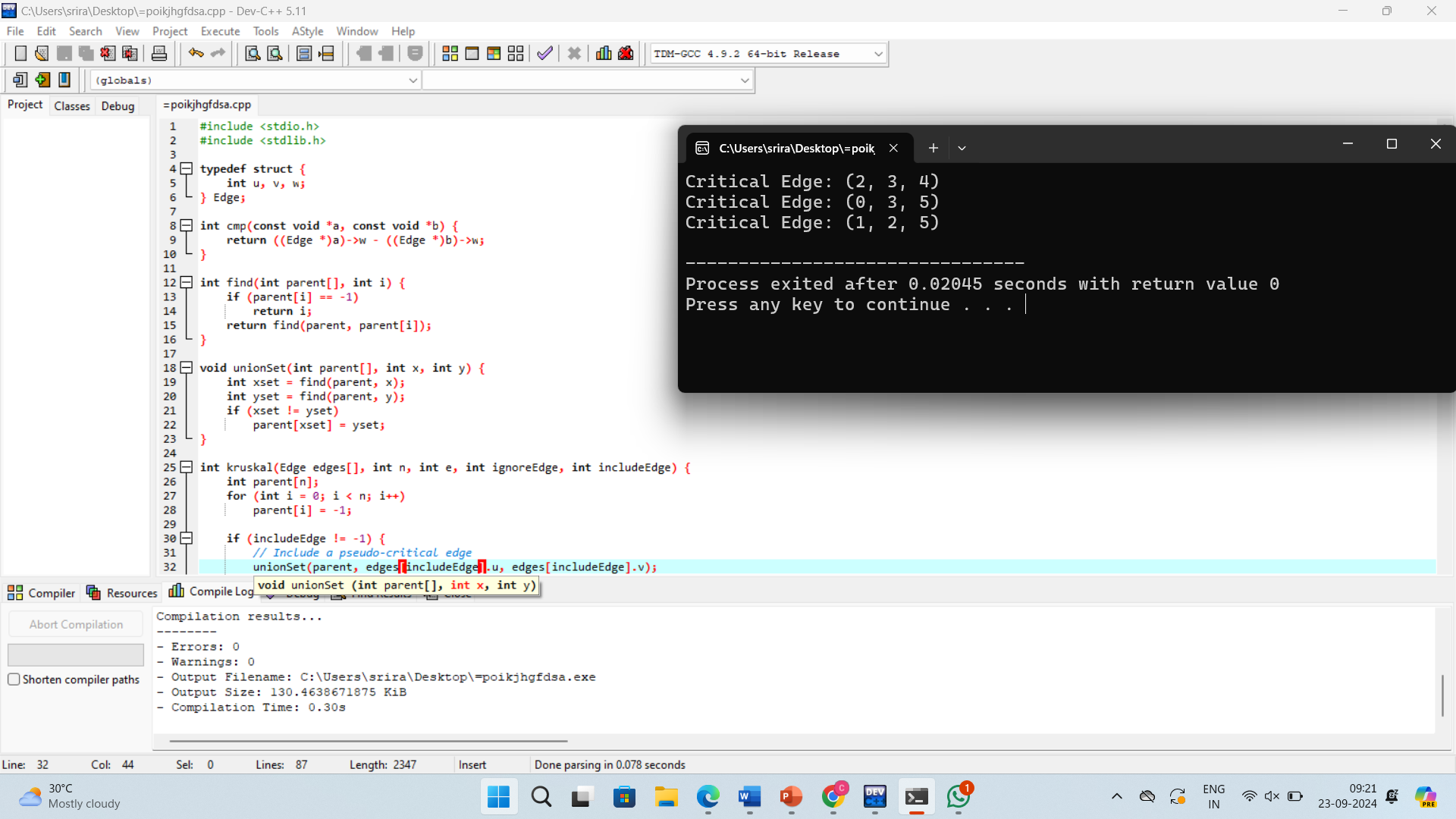
{2, 4, 3}

};

findCriticalAndPseudoCriticalEdges(n, edges, e);

return 0;

**OUTPUT**

****

**Complexity Analysis**

**Best Case:** O(n log n)

Critical Edges: In the best-case scenario, edges that directly connect to a node with minimal weight will likely be critical, as they are essential for maintaining the minimum weight..

**Worst Case:** O(n log n)

Critical Edges: In a worst-case scenario (like a star topology with very large edge weights), all edges may be critical because removing any single edge would disconnect the MST.

**Average Case:** O(n log n)

Critical Edges: Generally, a portion of edges wil critical, especially those with minimal weights. The distribution of weights affects how many edges are critical. **Overall Complexity:** Time Complexity :O(n log n) space Complexity : O(n)

**CONCLUSION**

**Key benefits:**

**1Cost Efficiency**:

2 **Optimal Connectivity:**:

ALGORITHM

The time complexity of finding a minimum spanning tree (MST) depends on the algorithm used. Here are the complexities for the two most common algorithms:

**Kruskal’s Algorithm**

**Time Complexity**: O(Elog⁡E)O(E \log E)O(ElogE) or O(Elog⁡V)O(E \log V)O(ElogV)

**Explanation**:

Sorting the edges takes O(Elog⁡E)O(E \log E)O(ElogEThe union-find operations (with path compression and union by rank) take nearly constant time, leading to efficient processing of edges.

**Prim’s Algorithm**

**Time Complexity**:

Using a binary heap (priority queue): O(Elog⁡V)O(E \log V)O(ElogV)

Using a Fibonacci heap: O(E+Vlog⁡V)O(E + V \log V)O(E+VlogV)

**Explanation**:

The algorithm primarily consists of selecting the minimum edge and updating the priority queue, leading

**Advantages:**

1. **Performance Prediction**

2. **Scalability**:

3. **Algorithm Comparison:**

In conclusion, minimum spanning trees (MSTs) are vital structures in graph theory with wide-ranging applications in computer science, network design, and optimization problems. By efficiently connecting all vertices in a graph with the minimum possible total edge weight, MSTs facilitate cost-effective solutions for various practical challenges, such as network design, clustering, and infrastructure planning.

The two primary algorithms for finding MSTs—Kruskal’s and Prim’s—offer different approaches suited for various scenarios, each with distinct time complexities that enable scalability and efficiency. Understanding these algorithms and their complexities allows for informed decision-making when tackling graph-related problems.

Overall, MSTs not only enhance connectivity and resource optimization but also serve as foundational tools in many fields, driving advancements in technology and improving system performance. Their significance in both theory and application underscores the importance of mastering these concepts for anyone working with graphs or optimization problems.