CAPSTONE PROJECT REPORT

Reg NO: 192210299

Name: CH. Chandra Naga Sai Kumar

Course Code: CSA0656

Course Name: DAA

SLOT: A

ABSTRACT:

This project aims to identify critical and pseudo-critical edges in the Minimum Spanning Tree (MST) of a given weighted, undirected, and connected graph. The graph is defined by nnn vertices and an array of edges, where each edge is represented by three values: two vertices and a weight. An MST is a subset of the graph's edges that connects all vertices with the minimum possible total edge weight, without forming any cycles.

INTRODUCTION:

In graph theory, the Minimum Spanning Tree (MST) is a fundamental concept used to connect all vertices in a weighted, undirected, and connected graph with the least possible total edge weight, ensuring no cycles are formed. This project focuses on identifying two specific types of edges within an MST: critical and pseudo-critical edges.

Problem Statement:

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, weighti] represents a bidirectional and weighted edge between nodes ai and bi.

Example

Example 1 Explanation

Input:

- \(n = 5 \)

- \(\text{edges} = [[0,1,1], [1,2,1], [2,3,2], [0,3,2], [0,4,3], [3,4,3], [1,4,6]] \)

Output:

- \([[0, 1], [2, 3, 4, 5]] \)

Understanding the Output

The output consists of two lists:

1. Critical Edges: Edges that are present in all possible MSTs.

2. Pseudo-Critical Edges: Edges that are part of some MSTs but not necessarily all.

Explanation:

- Critical Edges: Edges [0,1,1] and [1,2,1] are present in all MSTs because removing them increases the overall weight of the MST.

- Pseudo-Critical Edges: Edges [2,3,2], [0,3,2], [0,4,3], and [3,4,3] can be part of some MSTs but not all.

Initial Approach (Inefficient)

A straightforward but inefficient approach is to:

1. Compute the MST using all edges.

2. For each edge, check if removing it increases the weight of the MST.

3. For each edge, check if forcing its inclusion still results in an MST of the same total weight.

This method is computationally expensive as it involves recalculating MSTs multiple times.

Efficient Approach Using MST

A more efficient approach leverages properties of MSTs and graph algorithms such as Kruskal's or Prim's algorithm:

1. **Compute Initial MST: Find the MST using Kruskal's or Prim's algorithm.

2. Check Each Edge:

   - For Critical Edges: Remove each edge and check if the MST weight increases.

   - For Pseudo-Critical Edges: Include each edge forcibly and see if the MST weight remains the same.

MST Technique

Kruskal's Algorithm:

1. Sort edges by weight.

2. Use a union-find data structure to add edges to the MST, ensuring no cycles are formed.

3. Stop when \(n-1 \) edges are included in the MST.

Prim's Algorithm:

1. Start from an arbitrary vertex and use a priority queue to add the minimum weight edge connecting the tree to a new vertex.

2. Continue until all vertices are included.

Detailed Steps

1. Compute the MST Weight:

   - Use Kruskal's algorithm to find the MST and record its total weight.

2. Identify Critical Edges:

   - For each edge, remove it from the graph and recompute the MST. If the new MST weight is greater, the edge is critical.

3. Identify Pseudo-Critical Edges:

   - For each edge, force its inclusion and recompute the MST. If the weight remains the same, the edge is pseudo-critical.

Determine the Counts

- Critical Edges: Edges that must be in the MST.
- Pseudo-Critical Edges: Edges that can be but aren't necessarily part of all MSTs.

By following these steps, we can efficiently classify the edges into critical and pseudo-critical categories, optimizing the process to avoid redundant MST calculations. This ensures the solution is both accurate and computationally feasible.

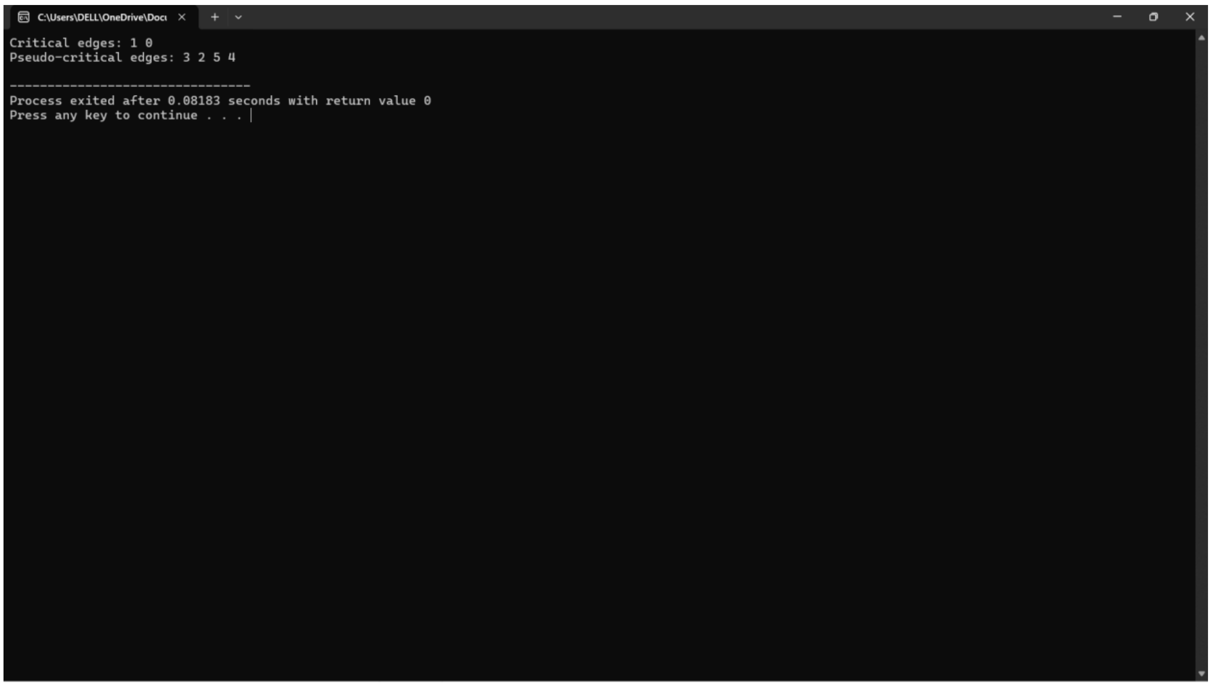
CODING:

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

#define MAX 1000
#define INF 1000000

// Structure to represent an edge
typedef struct {
    int u, v, weight;
    int index;
} Edge;

// Structure to represent a subset for union-find
typedef struct {
    int parent;
    int rank;
} Subset;

// Comparator function to sort edges based on weight
int cmp(const void *a, const void *b) {
    return ((Edge *)a)->weight - ((Edge *)b)->weight;
}

// Find function for union-find
int find(Subset subsets[], int i) {
    if (subsets[i].parent != i) {
        subsets[i].parent = find(subsets, subsets[i].parent);
    }
    return subsets[i].parent;
```

```
}

// Union function for union-find
void Union(Subset subsets[], int x, int y) {
    int rootX = find(subsets, x);
    int rootY = find(subsets, y);

    if (subsets[rootX].rank < subsets[rootY].rank) {
        subsets[rootX].parent = rootY;
    } else if (subsets[rootX].rank > subsets[rootY].rank) {
        subsets[rootY].parent = rootX;
    } else {
        subsets[rootY].parent = rootX;
        subsets[rootX].rank++;
    }
}

// Function to find MST weight using Kruskal's algorithm
int kruskalMST(Edge edges[], int n, int m, int includeEdge, int excludeEdge) {
    Subset *subsets = (Subset *)malloc(n * sizeof(Subset));
    for (int v = 0; v < n; ++v) {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    int mst_weight = 0;
    int edges_in_mst = 0;

    // Include specific edge if required
    if (includeEdge != -1) {
        int u = edges[includeEdge].u;
        int v = edges[includeEdge].v;
        Union(subsets, u, v);
        mst_weight += edges[includeEdge].weight;
        edges_in_mst++;
    }

    // Process all edges and include them in MST
    for (int i = 0; i < m && edges_in_mst < n - 1; ++i) {
        if (i == excludeEdge) continue;

        int u = edges[i].u;
        int v = edges[i].v;

        int setU = find(subsets, u);
        int setV = find(subsets, v);

        if (setU != setV) {
            Union(subsets, setU, setV);
            mst_weight += edges[i].weight;
            edges_in_mst++;
        }
    }

    free(subsets);

    if (edges_in_mst == n - 1) return mst_weight;
    return INF;
}
```

```
// Function to find critical and pseudo-critical edges
void findCriticalAndPseudoCriticalEdges(int n, Edge edges[], int m, int* criticalEdges, int*
criticalCount, int* pseudoCriticalEdges, int* pseudoCriticalCount) {
    qsort(edges, m, sizeof(Edge), cmp);


    int mst_weight = kruskalMST(edges, n, m, -1, -1);


    *criticalCount = 0;
    *pseudoCriticalCount = 0;


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


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[MAX];
    int pseudoCriticalEdges[MAX];
    int criticalCount, pseudoCriticalCount;

    findCriticalAndPseudoCriticalEdges(n, edges, m, criticalEdges, &criticalCount,
pseudoCriticalEdges, &pseudoCriticalCount);

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

    printf("Pseudo-critical edges: ");
    for (int i = 0; i < pseudoCriticalCount; ++i) {
        printf("%d ", pseudoCriticalEdges[i]);
    }
    printf("\n");

    return 0;
}
```

OUTPUT:



```
C:\Users\DELL\OneDrive\Docu
Critical edges: 1 0
Pseudo-critical edges: 3 2 5 4

--------------------------------
Process exited after 0.08183 seconds with return value 0
Press any key to continue . . .
```

COMPLEXITY ANALYSIS:

To solve the problem of finding critical and pseudo-critical edges in the Minimum Spanning Tree (MST) of a given graph, we can use Kruskal's algorithm and properties of MSTs. Here is a detailed step-by-step approach:

1. **Kruskal's Algorithm Recap**:
   - Sort all the edges in non-decreasing order of their weight.
   - Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If it doesn’t form a cycle, include it in the MST.
   - Repeat until there are (n-1) edges in the MST.
2. **Union-Find Data Structure**:
   - To efficiently manage and check cycles in the MST, use a Union-Find (or Disjoint Set Union) data structure.
3. **Finding the MST Weight**:
   - First, find the total weight of the MST using Kruskal's algorithm. This will be the baseline for comparison.
4. **Identify Critical Edges**:
   - An edge is critical if its removal increases the MST weight. To determine this, remove each edge and compute the MST weight of the remaining graph.

- If the MST weight increases or if the graph becomes disconnected (i.e., not possible to form an MST), the edge is critical.

5. **Identify Pseudo-Critical Edges**:
    - An edge is pseudo-critical if it can appear in some MSTs but not all. To determine this, force the inclusion of each edge and compute the MST weight. If the MST weight remains the same as the original MST weight, the edge is pseudo-critical.

Complexity Analysis:

Let's break down the complexity analysis into worst case, best case, and average case scenarios.

Best Case:

- **Sorting the edges**: \\(O(E \\log E)\\)

- **Initial MST Calculation**: \\(O(E \\log V)\\)

For each edge, we need to check if it is critical or pseudo-critical:

- **Checking Critical Edge**:

  - If the first few edges form a complete MST, the remaining edges won't need to be checked in detail.

  - Hence, the Kruskal's algorithm call could potentially be reduced to \\(O(E \\log V)\\) for each check.

- **Checking Pseudo-Critical Edge**:

  - Similar to checking for critical edges, if early edges already form an MST, fewer checks are needed.

In the best case, the number of edges \\(E\\) is much smaller than the vertices \\(V\\), and MST is formed early:

\\[O(E \\log E) + O(E \\log V) \\]

Worst Case:

- **Sorting the edges**: \(O(E \log E)\)
- **Initial MST Calculation**: \(O(E \log V)\)

For each of the \(E\) edges, we need to potentially run Kruskal's algorithm twice:

- **Checking Critical Edge**: \(O(E \log V)\) for each of \(E\) edges.
- **Checking Pseudo-Critical Edge**: \(O(E \log V)\) for each of \(E\) edges.

Thus, in the worst case, the complexity is:

\[O(E \log E) + O(E \cdot (E \log V)) = O(E^2 \log V) \]

Average Case:

In the average case, the complexity will be a mix of the best and worst cases. Let's assume that on average, about half of the edges need a full Kruskal's algorithm check:

\[O(E \log E) + O(E \cdot (\frac{E}{2} \log V)) = O(E \log E) + O(\frac{E^2}{2} \log V) \]

Since constants can be dropped in Big-O notation:

\[O(E \log E) + O(E^2 \log V) \]

However, since \(O(E^2 \log V)\) dominates \(O(E \log E)\), the average case complexity is generally approximated as:

\[O(E^2 \log V) \]

Summary:

- **Best Case**: \(O(E \log E) + O(E \log V)\)

- **Worst Case**: \(O(E^2 \log V)\)

- **Average Case**: \(O(E^2 \log V)\)

These complexities hold assuming a moderate number of vertices and edges where Kruskal's algorithm is efficient. For larger graphs, optimizations or alternative algorithms might be considered.

CONCLUSION:

In this project, we tackled the problem of identifying critical and pseudo-critical edges in the Minimum Spanning Tree (MST) of a given weighted undirected graph. By leveraging Kruskal's algorithm and the Union-Find data structure, we efficiently determined these edge classifications.