Question 1: Minimum Spanning Trees (Prim's Algorithm)

1.1. Version 1: Adjacency Matrix & Unsorted Array Priority Queue

```cpp
#include <iostream>
#include <vector>
#include <climits>
using namespace std;

int extract_min(const vector<int>& key, const vector<bool>& selected) {
    int min = INT_MAX;
    int min_idx = -1;
    for (int i = 0; i < key.size(); ++i) {
        if (!selected[i] && key[i] < min) {
            min = key[i];
            min_idx = i;
        }
    }
    return min_idx;
}

void prim(const vector<vector<int>>& G, int V) {
    vector<int> key(V, INT_MAX);
    vector<int> MST(V, -1);
    vector<bool> selected(V, false);
    key[0] = 0;

    for (int i = 0; i < V - 1; ++i) {
        int u = extract_min(key, selected);
        if (u == -1) return;
        selected[u] = true;

        for (int v = 0; v < G[u].size(); ++v) {
            if (G[u][v] > 0 && !selected[v] && G[u][v] < key[v]) {
                key[v] = G[u][v];
                MST[v] = u;
            }
        }
    }
    // Output formatting omitted for brevity
}
```

Listing 1: Prim's Algorithm using Adjacency Matrix

Experimental Results

Test Case 1

```
Graph:
    {0, 3, 0, 6, 0}
    {3, 0, 4, 8, 5}
    {0, 4, 0, 0, 7}
    {6, 8, 0, 0, 11}
    {0, 5, 7, 11, 0}
```

```
MST Output:
    Edge    Weight
    0 - 1     3
    1 - 2     4
    0 - 3     6
    1 - 4     5
```

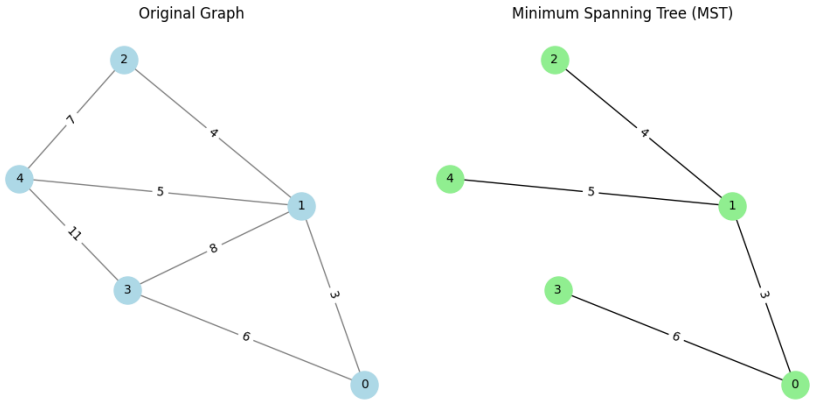

Figure 1: MST Visualization for Test Case 1

Test Case 2

```
Graph:
    { 0, 2, 0, 6, 0 }
    { 2, 0, 3, 8, 5 }
    { 0, 3, 0, 0, 7 }
    { 6, 8, 0, 0, 9 }
    { 0, 5, 7, 9, 0 }

MST Output:
    Edge    Weight
    0 - 1     2
    1 - 2     3
    0 - 3     6
    1 - 4     5
```

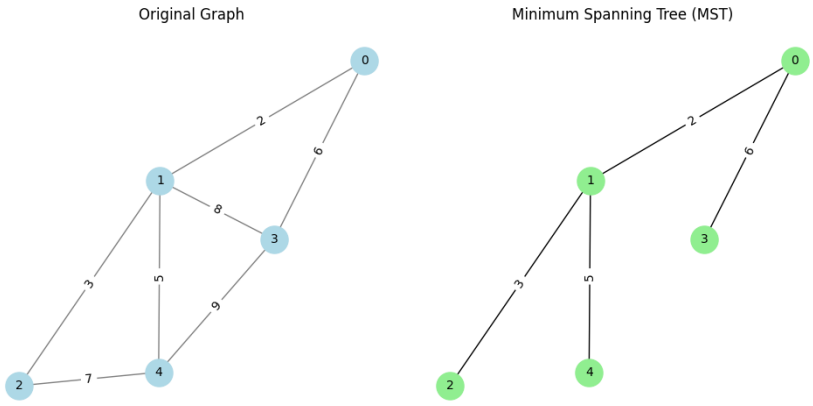

Figure 2: MST Visualization for Test Case 2

Test Case 3

```
Graph:
    { 0, 4, 0, 7, 2 }
    { 4, 0, 3, 0, 1 }
    { 0, 3, 0, 5, 6 }
    { 7, 0, 5, 0, 4 }
    { 2, 1, 6, 4, 0 }

MST Output:
    Edge    Weight
    4 - 1     1
    1 - 2     3
    4 - 3     4
    0 - 4     2
```

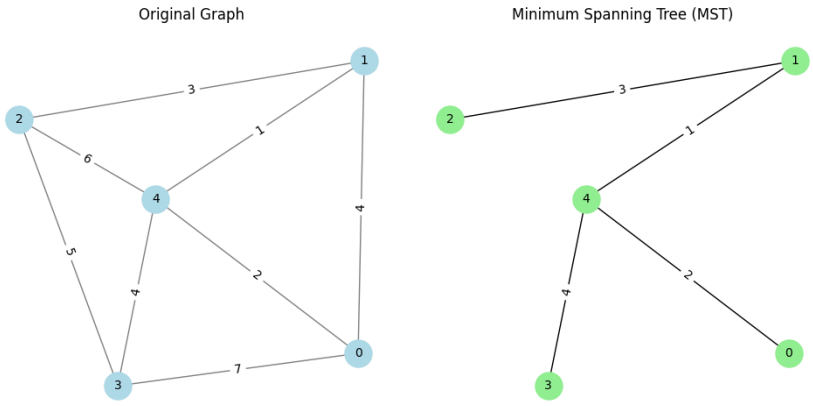

Figure 3: MST Visualization for Test Case 3

1.2. Version 2: Adjacency List & Binary Heap Priority Queue

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
using namespace std;

int extract_min(priority_queue<pair<int, int>, vector<pair<int, int>>, greater<
    pair<int, int>>>& Q) {
    int vertex = Q.top().second;
    Q.pop();
    return vertex;
}

void prim(const vector<vector<pair<int, int>>>& G, int V) {
    vector<int> key(V, INT_MAX);
    vector<int> MST(V, -1);
    vector<bool> selected(V, false);
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int
    >>> Q;

    key[0] = 0;
    Q.push({0, 0});

    while (!Q.empty()) {
        int u = extract_min(Q);
        if (selected[u]) continue;
        selected[u] = true;

        for (const auto& [v, w] : G[u]) {
            if (!selected[v] && w < key[v]) {
                key[v] = w;
                MST[v] = u;
                Q.push({key[v], v});
            }
        }
    }
    // Output formatting omitted
}
```

Listing 2: Prim's Algorithm using Adjacency List

Experimental Results

Test Case 4

```
Graph:
    G(0) = { (1, 4), (3, 7), (4, 2) }
    G(1) = { (0, 4), (2, 3), (4, 1) }
    G(2) = { (1, 3), (3, 5), (4, 6) }
    G(3) = { (0, 7), (2, 5), (4, 4) }
    G(4) = { (0, 2), (1, 1), (2, 6), (3, 4) }

MST Output:
    Edge    Weight
    4 - 1      1
```

```
    1 - 2     3
    4 - 3     4
    0 - 4     2
```

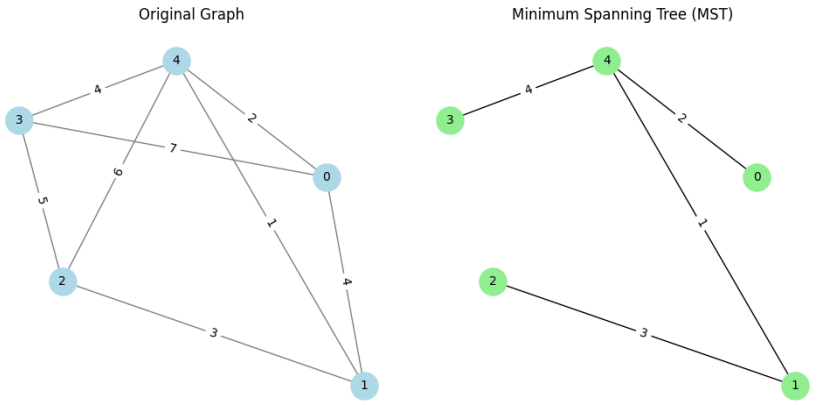

Figure 4: MST Visualization for Test Case 4

Test Case 5

```
Graph:
     G(0) = { (1, 2), (2, 4) }
     G(1) = { (0, 2), (2, 1), (3, 7) }
     G(2) = { (0, 4), (1, 1), (4, 3) }
     G(3) = { (1, 7), (4, 6), (5, 5) }
     G(4) = { (2, 3), (3, 6), (5, 8), (6, 4) }
     G(5) = { (3, 5), (4, 8), (6, 2) }
     G(6) = { (4, 4), (5, 2) }

MST Output:
    Edge    Weight
    0 - 1     2
    1 - 2     1
    5 - 3     5
    2 - 4     3
    6 - 5     2
    4 - 6     4
```

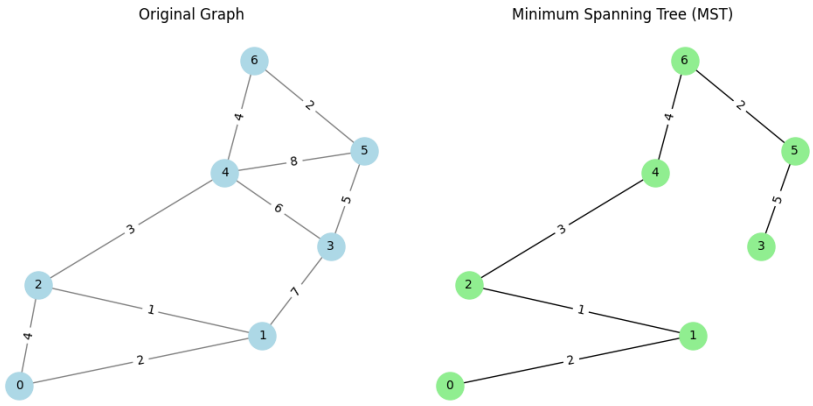

Figure 5: MST Visualization for Test Case 5

Test Case 6

```
Graph:
    G(0) = { (1, 6), (2, 1), (3, 5) }
    G(1) = { (0, 6), (2, 5), (4, 3) }
    G(2) = { (0, 1), (1, 5), (3, 2), (4, 6), (5, 4) }
    G(3) = { (0, 5), (2, 2), (5, 4) }
    G(4) = { (1, 3), (2, 6), (5, 6) }
    G(5) = { (2, 4), (3, 4), (4, 6) }

MST Output:
    Edge    Weight
    2 - 1     5
    0 - 2     1
    2 - 3     2
    1 - 4     3
    2 - 5     4
```

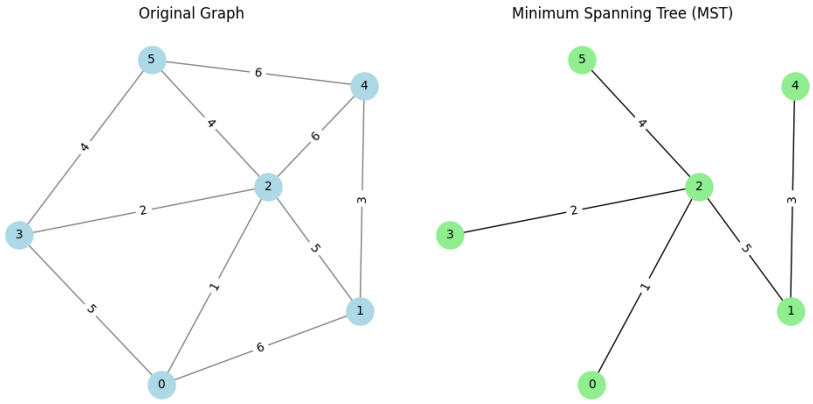

Figure 6: MST Visualization for Test Case 6

Question 2: Single-Source Shortest Paths

2.1. Dijkstra's Algorithm

```cpp
vector<int> dijkstra(int src, int V, map<int, vector<pair<int, int>>>& G) {
    vector<int> d(V, INT_MAX);
    vector<bool> visited(V, false);
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int
    >>> Q;

    d[src] = 0;
    Q.push({0, src});

    while (!Q.empty()) {
        int u = Q.top().second;
        Q.pop();

        if (visited[u]) continue;
        visited[u] = true;

        if (G.find(u) != G.end()) {
            for (auto& e : G[u]) {
                int v = e.first;
                int w = e.second;
                if (!visited[v] && d[u] + w < d[v]) {
                    d[v] = d[u] + w;
                    Q.push({d[v], v});
                }
            }
        }
    }
    return d;
}
```

Listing 3: Dijkstra's Algorithm Implementation

Experimental Results

Test Case 7

```
Graph:
    G(0) = { (1, 2), (2, 4) }
    G(1) = { (0, 2), (2, 1), (3, 7) }
    G(2) = { (0, 4), (1, 1), (4, 3) }
    G(3) = { (1, 7), (4, 1) }
    G(4) = { (2, 3), (3, 1) }

SPT Output:
    Vertex    Distance from Source Vertex 0
        0             0
        1             2
        2             3
        3             7
        4             6
```

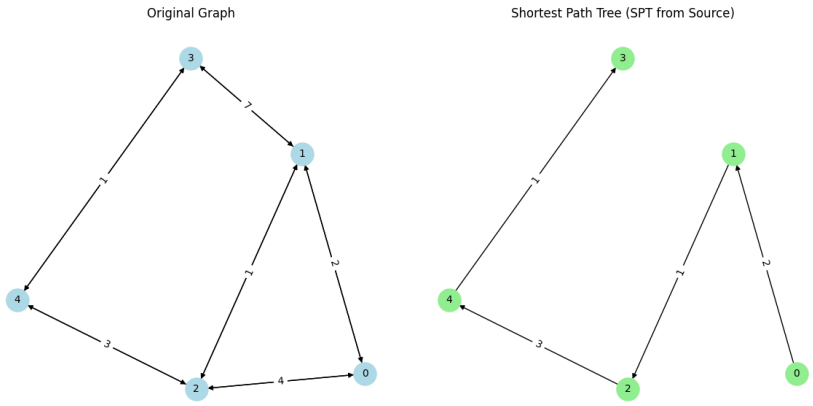

Figure 7: SPT Visualization for Test Case 7

Test Case 8

```
Graph:
    G(0) = { (1, 3), (3, 7) }
    G(1) = { (2, 1), (3, 4) }
    G(2) = { (3, 2), (4, 5) }
    G(3) = { (4, 1) }
    G(4) = { (5, 7), (6, 3) }
    G(5) = { (6, 2), (7, 4) }
    G(6) = { (7, 6) }
    G(7) = { }

SPT Output:
    Vertex    Distance from Source Vertex 0
        0             0
        1             3
        2             4
        3             6
        4             7
        5             14
        6             10
        7             16
```

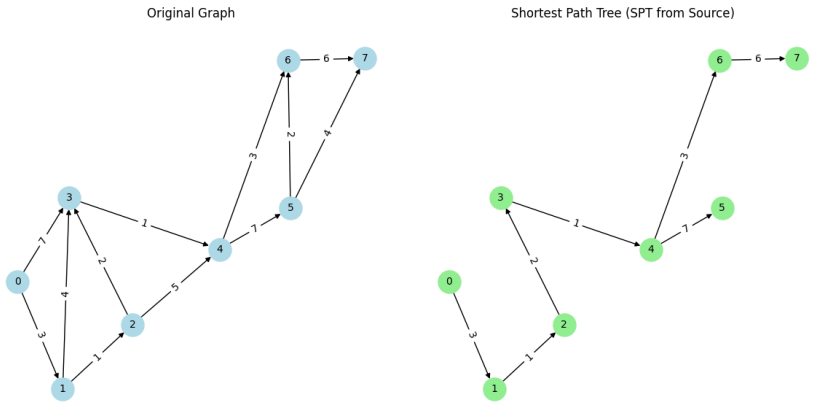

Figure 8: SPT Visualization for Test Case 8

2.2. Bellman-Ford Algorithm

```cpp
bool bellmanFord(int src, int V, map<int, vector<pair<int, int>>>& G) {
    vector<int> d(V, INT_MAX);
    d[src] = 0;

    for (int i = 0; i < V - 1; ++i) {
        bool relaxed = false;
        for (int u = 0; u < V; ++u) {
            for (auto& e : G[u]) {
                int v = e.first;
                int w = e.second;
                if (d[u] != INT_MAX && d[u] + w < d[v]) {
                    d[v] = d[u] + w;
                    relaxed = true;
                }
            }
        }
        if (!relaxed) break;
    }
    // Negative cycle check omitted for brevity
    return true;
}
```

Listing 4: Bellman-Ford Implementation

Experimental Results

Test Case 9

```
Graph:
    G(0) = { (1, 5) }
    G(1) = { (2, 1), (3, 2) }
    G(2) = { (4, 1) }
    G(3) = { }
    G(4) = { (3, -1) }
```

```
SPT Output:
    Vertex       Distance from Source
        0             0
        1             5
        2             6
        3             6
        4             7
```

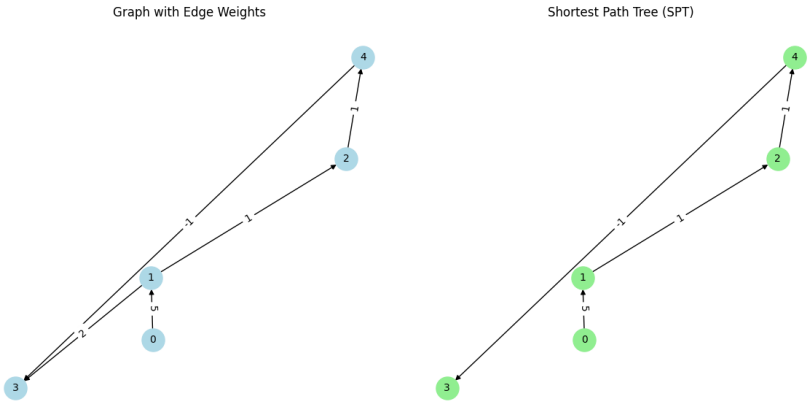

Figure 9: SPT Visualization for Test Case 9

Test Case 10

```
Graph:
    G(0) = { (1, 6), (2, 7) }
    G(1) = { (2, 8), (3, 5), (4, -4) }
    G(2) = { (3, -3), (4, 9) }
    G(3) = { (1, -2) }
    G(4) = { (3, 7) }

SPT Output:
    Vertex       Distance from Source
        0             0
        1             2
        2             7
        3             4
        4            -2
```

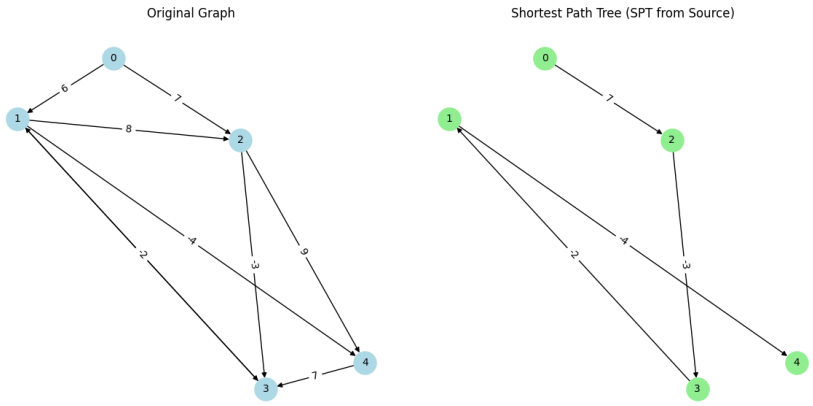

Figure 10: SPT Visualization for Test Case 10

2.3. Johnson's Algorithm

```cpp
void johnson(int V, map<int, vector<pair<int, int>>>& G) {
    for (int u = 0; u < V; ++u) {
        G[V].emplace_back(u, 0);
    }
    if (!bellmanFord(V, V + 1, G)) return;

    vector<int> h = bf_dist;
    map<int, vector<pair<int, int>>> new_G;

    for (int u = 0; u < V; ++u) {
        for (auto& [v, w] : G[u]) {
            int new_w = w + h[u] - h[v];
            new_G[u].emplace_back(v, new_w);
        }
    }
    // All-pairs calculation using Dijkstra omitted for brevity
}
```

Listing 5: Johnson's Algorithm Implementation

Experimental Results

Test Case 11

```
Graph:
    G(0) = { (1, -5), (2, 2), (3, 3) }
    G(1) = { (2, 4) }
    G(2) = { (3, 1) }
    G(3) = { }

New Graph (After removing all negative weights):
    G(0) = { (1, 0), (2, 3), (3, 3) }
    G(1) = { (2, 0) }
    G(2) = { (3, 0) }
    G(3) = { }
```

All Pairs Shortest Paths in Original Graph G:

$$\begin{bmatrix} 0 & -5 & -1 & 0 \\ \infty & 0 & 4 & 5 \\ \infty & \infty & 0 & 1 \\ \infty & \infty & \infty & 0 \end{bmatrix}$$

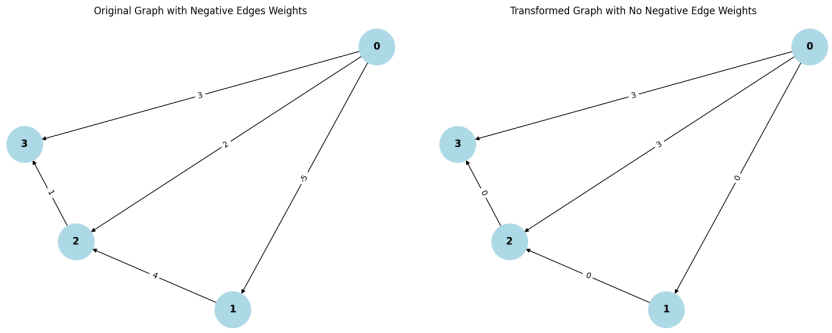

Figure 11: Johnson's Algorithm Visualization for Test Case 11

Test Case 12

```
Graph:
    G(0) = { (1, 3), (2, 8) }
    G(1) = { (3, 1), (4, -4) }
    G(2) = { (4, 2) }
    G(3) = { (0, 2), (2, -5) }
    G(4) = { (3, 6) }

New Graph (After removing all negative weights):
    G(0) = { (1, 3), (2, 13) }
    G(1) = { (3, 1), (4, 0) }
    G(2) = { (4, 1) }
    G(3) = { (0, 2), (2, 0) }
    G(4) = { (3, 2) }
```

All Pairs Shortest Paths in Original Graph G:

$$\begin{bmatrix} 0 & 3 & -1 & 4 & -1 \\ 3 & 0 & -4 & 1 & -4 \\ 10 & 13 & 0 & 8 & 2 \\ 2 & 5 & -5 & 0 & -3 \\ 8 & 11 & 1 & 6 & 0 \end{bmatrix}$$

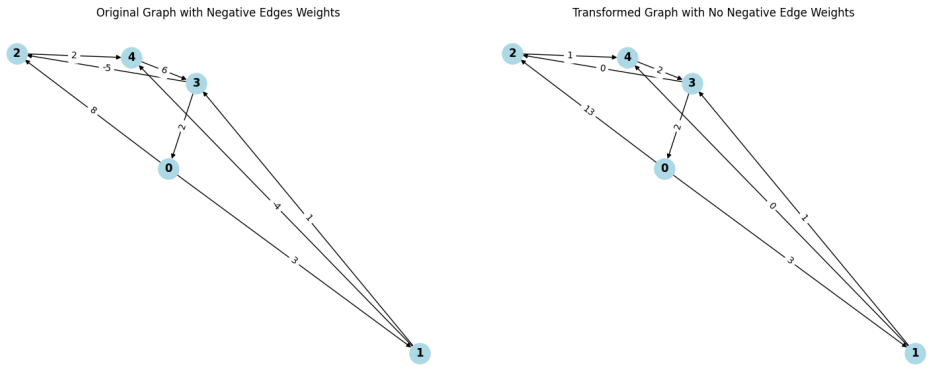

Figure 12: Johnson's Algorithm Visualization for Test Case 12