EECE7205 – Homework 4

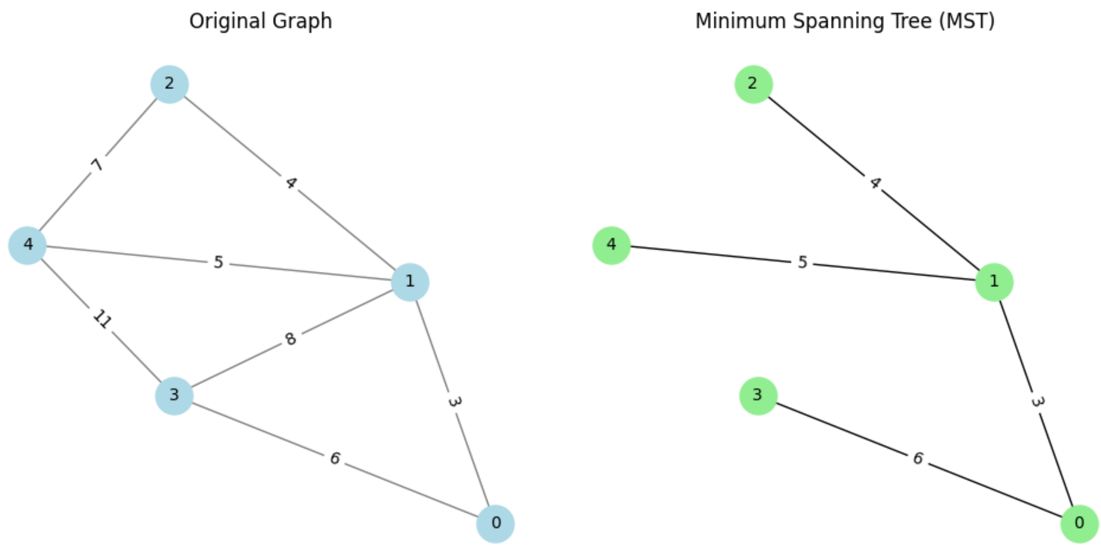
Question 1.

Version 1. Prim's Algorithm using adjacency matrix and unsorted array priority queue

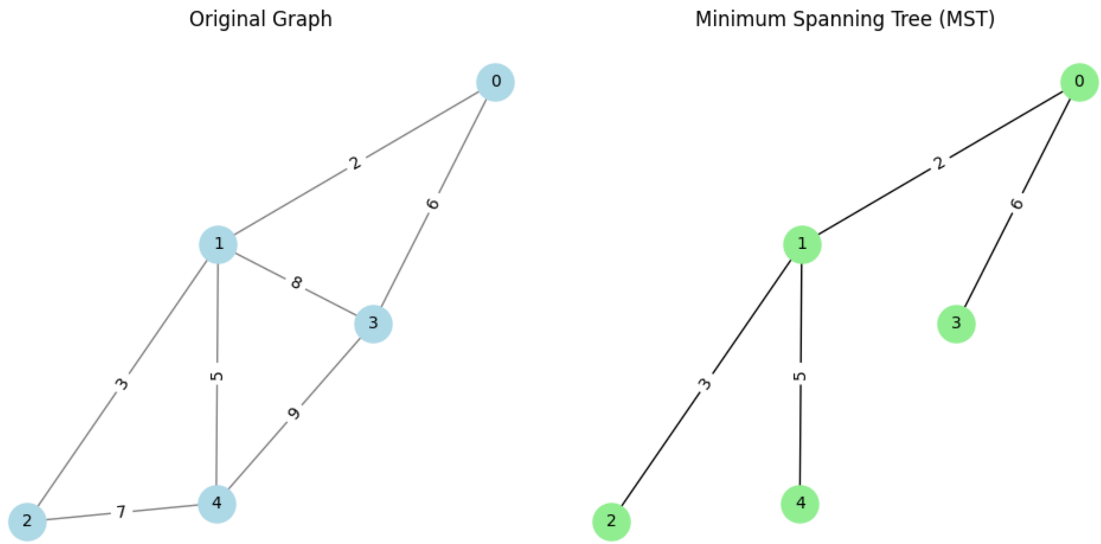
```cpp
#include <iostream>
#include <vector>
#include <climits>
using namespace std;

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;
            }
        }
    }

    cout << "Edge \tWeight\n";
    for (int i = 1; i < MST.size(); ++i) {
        if (MST[i] != -1)
            cout << MST[i] << " - " << i << "\t" << G[i][MST[i]] << "\n";
    }
}

int main() {
    int V = 5;
    vector<vector<int>> G = {
        { 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 }
    };
```

```
    prim(G, V);
    return 0;
}
```

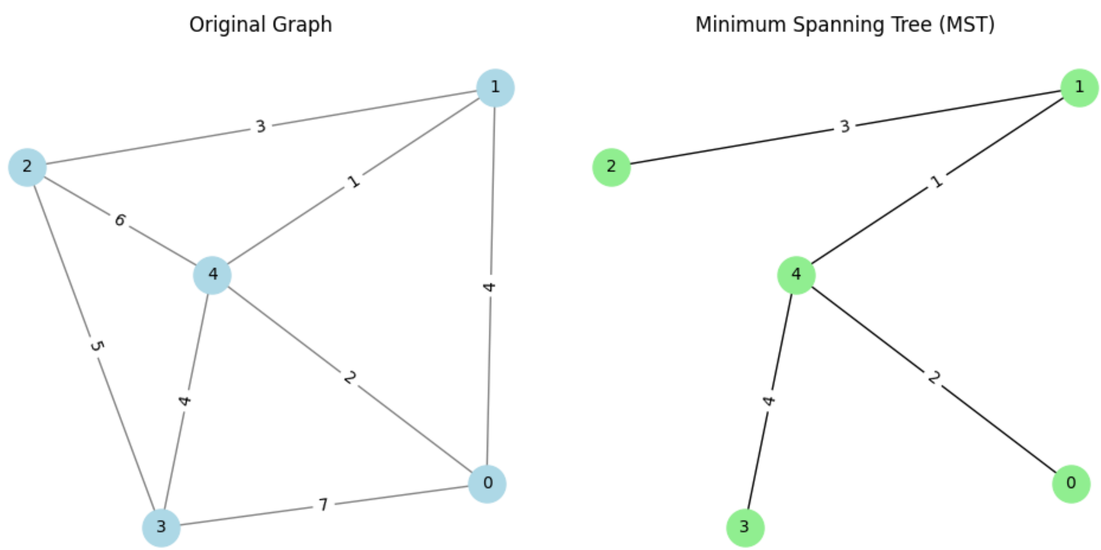
```
Graph:
    {0, 3, 0, 6, 0}
    {3, 0, 4, 8, 5}
    {0, 4, 0, 0, 7}
    {6, 8, 0, 0, 11}
    {0, 5, 7, 11, 0}

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




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




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

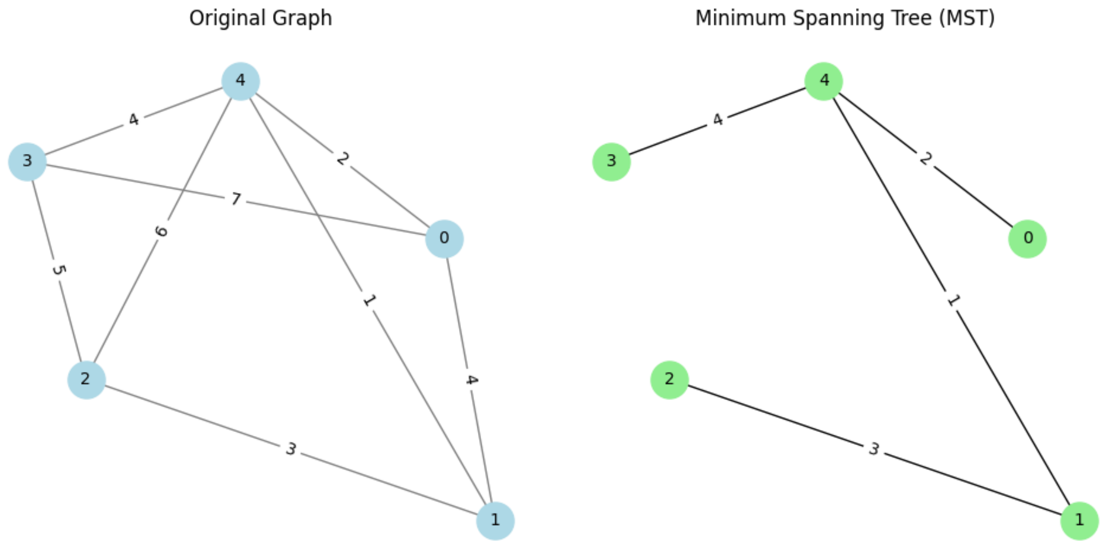


Version 2. Prim's Algorithm using adjacency list and heap priority queue

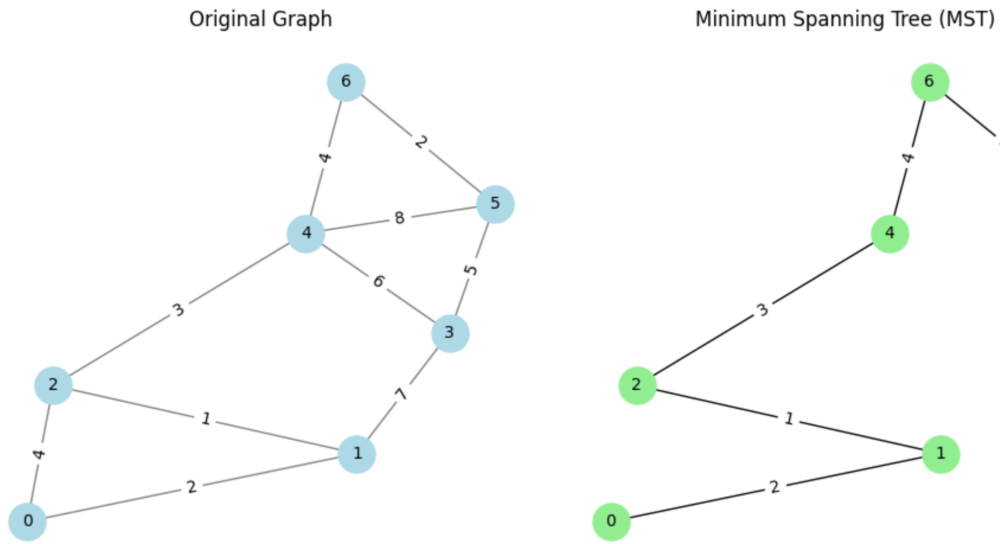
```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
using namespace std;

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});
            }
        }
    }

    cout << "Edge \tWeight\n";
    for (int i = 1; i < V; ++i) {
        if (MST[i] != -1) {
            cout << MST[i] << " - " << i << "\t" << key[i] << "\n";
        }
    }
}

int main() {
    int V = 5;
    vector<vector<pair<int, int>>> G(V);
    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}};

    prim(G, V);
    return 0;
}
```

```
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:
    Edge  Weight
    4 - 1   1
```

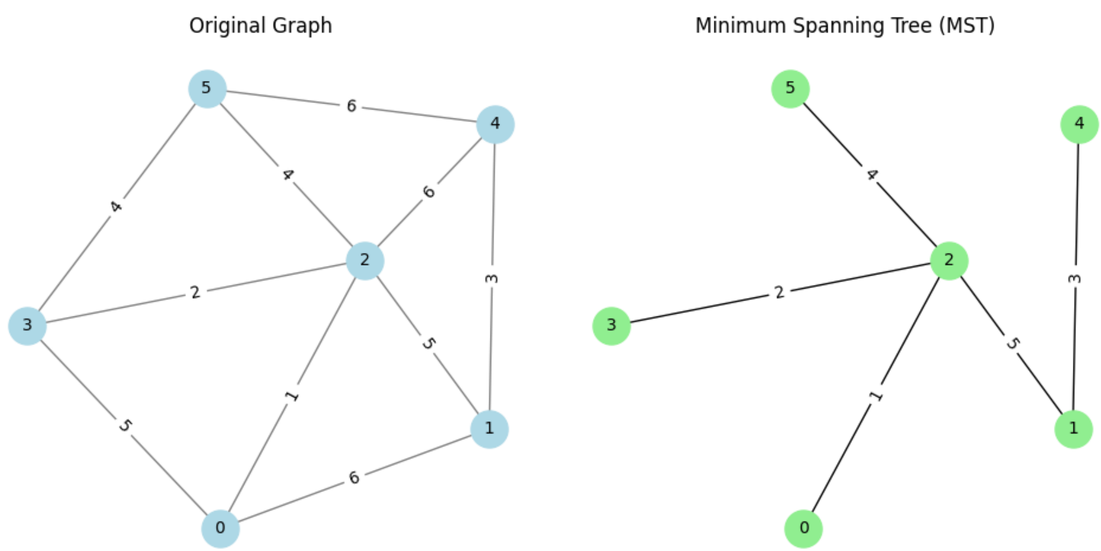
```
    1 - 2   3
    4 - 3   4
    0 - 4   2
```




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




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



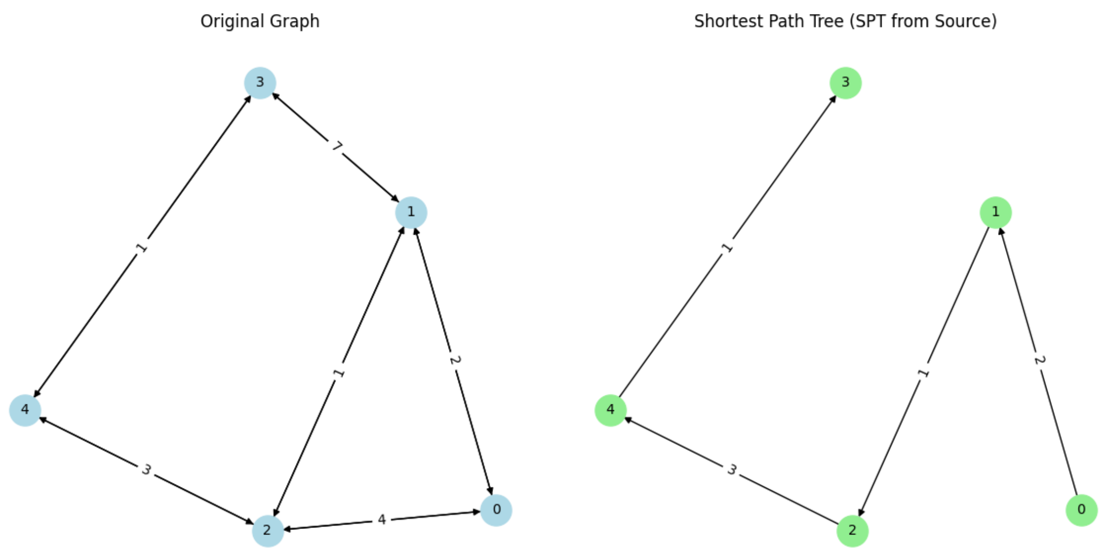
Question 2.

Dijkstra's Algorithm

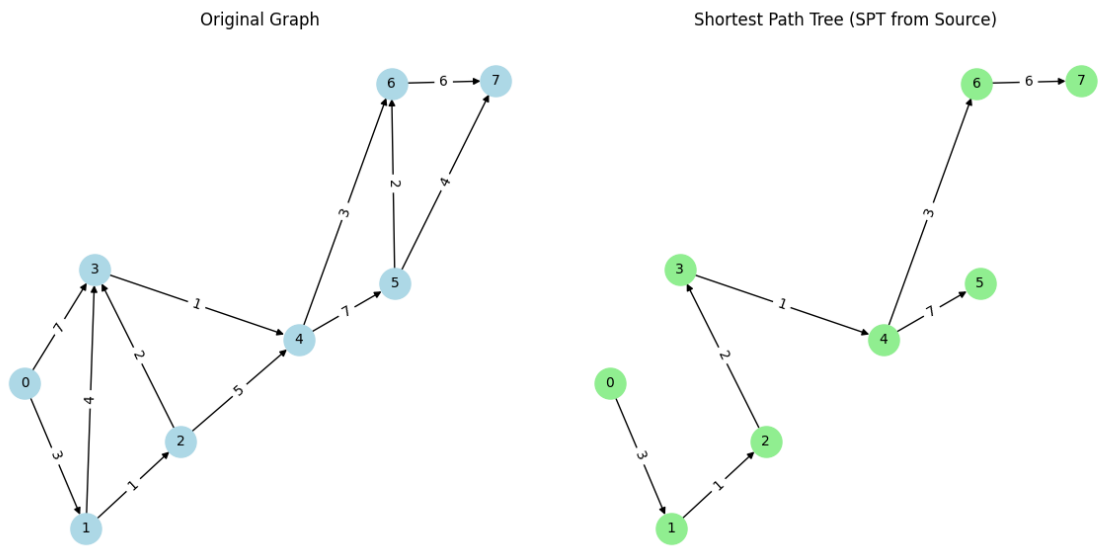
```cpp
#include <iostream>
#include <vector>
#include <tuple>
#include <map>
#include <climits>
#include <utility>
using namespace std;

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);
    vector<pair<int, int>> Q;

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

    while (!Q.empty()) {
        int min_idx = 0;
        for (int i = 1; i < Q.size(); i++) {
            if (Q[i].first < Q[min_idx].first) {
                min_idx = i;
            }
        }

        int dist = Q[min_idx].first;
        int u = Q[min_idx].second;
        Q.erase(Q.begin() + min_idx);

        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_back({d[v], v});
                }
            }
        }
    }
    return d;
}
int main() {
    int V = 8;
    map<int, vector<pair<int, int>>> G;
    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] = {};

    int src = 0;
    vector<int> distances = dijkstra(src, V, G);

    cout << "Vertex\tDistance from Source\n";
    for (int i = 0; i < distances.size(); i++) {
        cout << i << "\t" << distances[i] << "\n";
    }
    return 0;
```

```
}
```

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




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

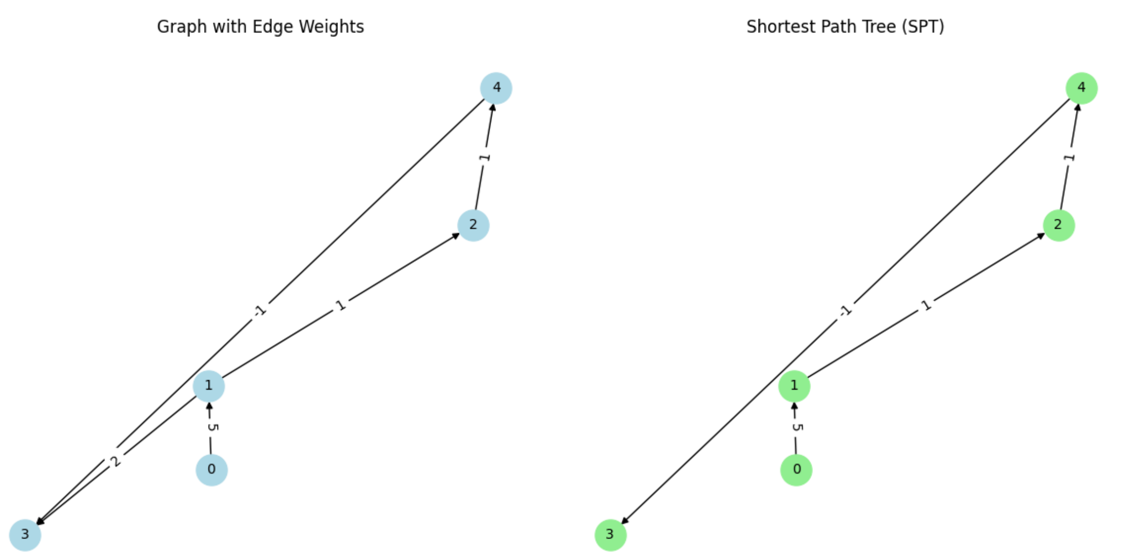


Bellman-Ford Algorithm

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

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;
        }
    }

    for (int u = 0; u < V; ++u) {
        if (G.find(u) != G.end()) {
            for (auto& e : G[u]) {
                int v = e.first;
                int w = e.second;
                if (d[u] != INT_MAX && d[u] + w < d[v]) {
                    cout << "Graph contains a negative weight cycle." << endl;
                    return false;
                }
            }
        }
    }

    cout << "Vertex\tDistance from Source" << endl;
    for (int i = 0; i < V; ++i) {
```

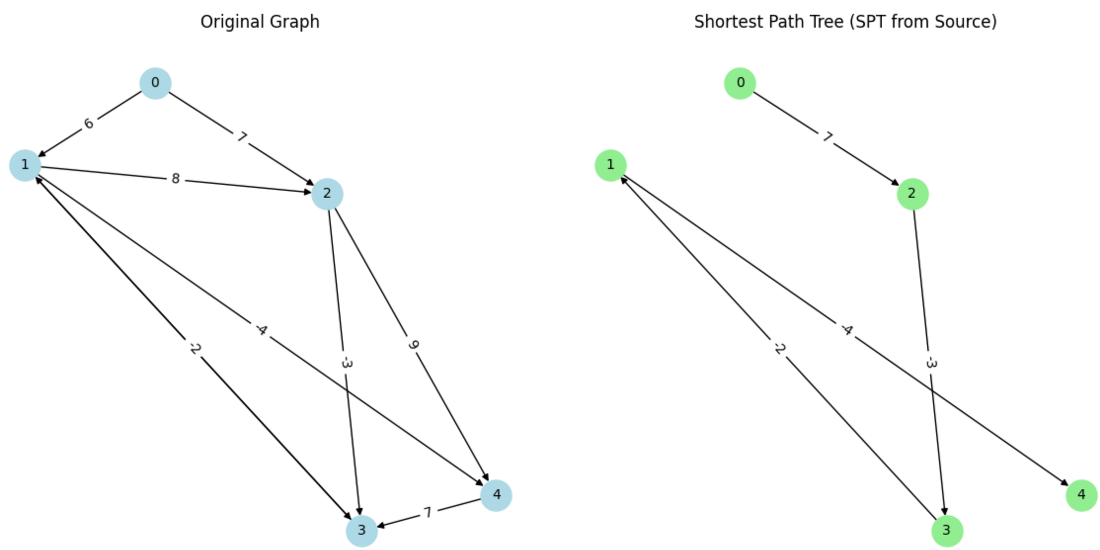
```cpp
        if (d[i] == INT_MAX) {
            cout << i << "\tINF" << endl;
        } else {
            cout << i << "\t" << d[i] << endl;
        }
    }
    return true;
}
int main() {
    int V = 5;
    map<int, vector<pair<int, int>>> G;
    G[0] = {{1, 5}};
    G[1] = {{2, 1}, {3, 2}};
    G[2] = {{4, 1}};
    G[3] = {};
    G[4] = {{3, -1}};

    int src = 0;
    bellmanFord(src, V, G);
    return 0;
}
```

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

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




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



Johnson's Algorithm

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

vector<int> bf_dist;
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;
        }
    }

    for (int u = 0; u < V; ++u) {
```

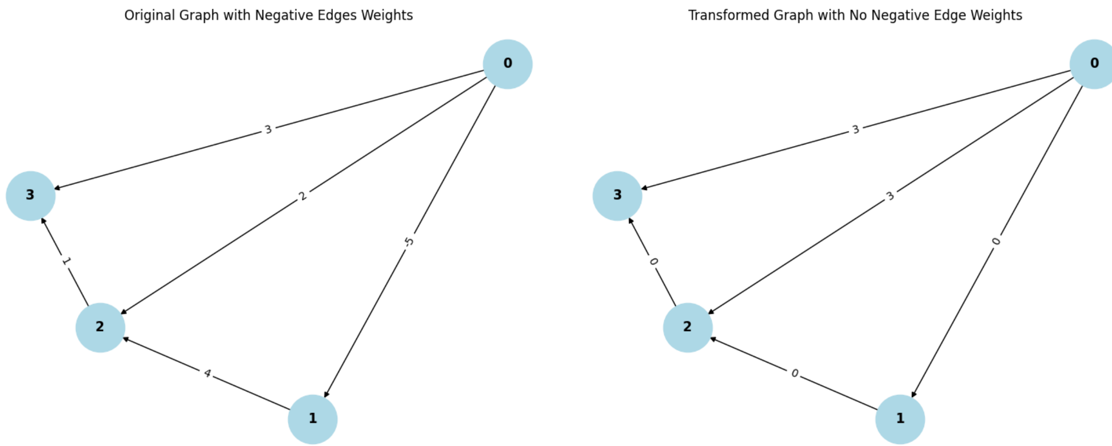
```
        if (G.find(u) != G.end()) {
            for (auto& e : G[u]) {
                int v = e.first;
                int w = e.second;
                if (d[u] != INT_MAX && d[u] + w < d[v]) {
                    cout << "Graph contains a negative weight cycle." << endl;
                    return false;
                }
            }
        }
    }
    bf_dist = d;
    return true;
}

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);
    vector<pair<int, int>> Q;

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

    while (!Q.empty()) {
        int min_idx = 0;
        for (int i = 1; i < Q.size(); i++) {
            if (Q[i].first < Q[min_idx].first) {
                min_idx = i;
            }
        }

        int dist = Q[min_idx].first;
        int u = Q[min_idx].second;
        Q.erase(Q.begin() + min_idx);

        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_back({d[v], v});
                }
            }
        }
    }
    return d;
}

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;

    cout << left << setw(10) << "Edge" << setw(20) << "Original Weight" << setw(20) << "New Weight" << endl;
    for (int u = 0; u < V; ++u) {
        for (auto& [v, w] : G[u]) {
            int new_w = w + h[u] - h[v];
```

```cpp
                new_G[u].emplace_back(v, new_w);
                cout << setw(20) << "(" + to_string(u) + " -> " + to_string(v) + ")"
                     << setw(15) << w
                     << setw(20) << new_w << endl;
            }
        }
        cout << endl;

        vector<vector<int>> all_pairs_distances(V, vector<int>(V, INT_MAX));
        for (int u = 0; u < V; ++u) {
            vector<int> distances = dijkstra(u, V, new_G);

            for (int v = 0; v < V; ++v) {
                if (distances[v] < INT_MAX) {
                    all_pairs_distances[u][v] = distances[v] + h[v] - h[u];
                }
            }
        }

        cout << "All Pairs Shortest Paths in Original Graph G:" << endl;
        cout << setw(6) << " " << " ";
        for (int i = 0; i < V; ++i) {
            cout << setw(8) << ("V" + to_string(i));
        }
        cout << endl;
        for (int u = 0; u < V; ++u) {
            cout << setw(6) << ("V" + to_string(u)) << " ";
            for (int v = 0; v < V; ++v) {
                if (all_pairs_distances[u][v] == INT_MAX)
                    cout << setw(8) << "inf";
                else
                    cout << setw(8) << all_pairs_distances[u][v];
            }
            cout << endl;
        }
}

int main() {
    int V = 5;
    map<int, vector<pair<int, int>>> G;
    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}};

    johnson(V, G);
    return 0;
}
```

```
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 (After Conversion from Reweighted Graph G'):

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




```
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 (After Conversion from Reweighted Graph G'):

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