

EECE7205 – Homework 4

Question 1.

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

```
1 #include <iostream>
2 #include <vector>
3 #include <climits>
4 using namespace std;
5
6 int extract_min(const vector<int>& key, const vector<bool>& selected) {
7     int min = INT_MAX;
8     int min_idx = -1;
9
10    for (int i = 0; i < key.size(); ++i) {
11        if (!selected[i] && key[i] < min) {
12            min = key[i];
13            min_idx = i;
14        }
15    }
16    return min_idx;
17 }
18
19 void prim(const vector<vector<int>>& G, int V) {
20     vector<int> key(V, INT_MAX);
21     vector<int> MST(V, -1);
22     vector<bool> selected(V, false);
23     key[0] = 0;
24
25     for (int i = 0; i < V - 1; ++i) {
26         int u = extract_min(key, selected);
27         if (u == -1) {
28             return;
29         }
30         selected[u] = true;
31
32         for (int v = 0; v < G[u].size(); ++v) {
33             if (G[u][v] > 0 && !selected[v] && G[u][v] < key[v]) {
34                 key[v] = G[u][v];
35                 MST[v] = u;
36             }
37         }
38     }
39
40     cout << "Edge \tWeight\n";
41     for (int i = 1; i < MST.size(); ++i) {
42         if (MST[i] != -1)
43             cout << MST[i] << " - " << i << "\t" << G[i][MST[i]] << "\n";
44     }
45 }
46
47 int main() {
48     int V = 5;
49     vector<vector<int>> G = {
50         { 0, 4, 0, 7, 2 },
51         { 4, 0, 3, 0, 1 },
52         { 0, 3, 0, 5, 6 },
53         { 7, 0, 5, 0, 4 },
54         { 2, 1, 6, 4, 0 }
55     };
```

```

56
57     prim(G, V);
58     return 0;
59 }

```

Graph:

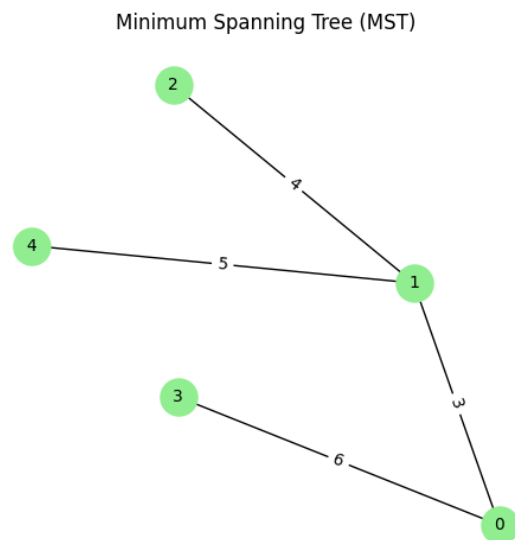
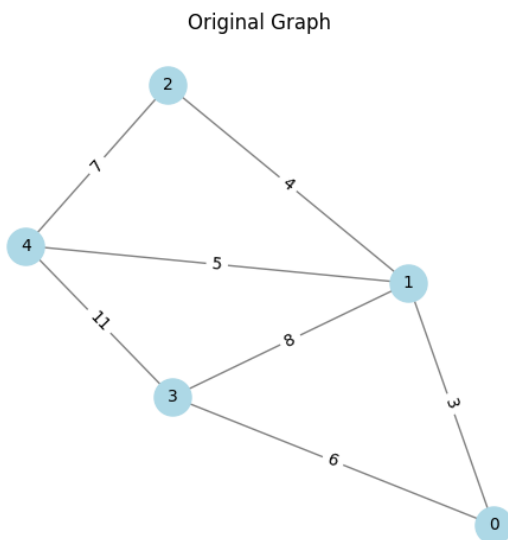
```

{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:

```

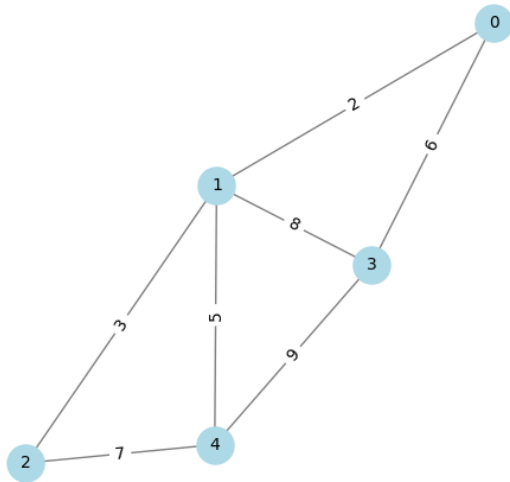
{ 0, 2, 0, 6, 0 }
{ 2, 0, 3, 8, 5 }
{ 0, 3, 0, 0, 7 }
{ 6, 8, 0, 0, 9 }
{ 0, 5, 7, 9, 0 }

```

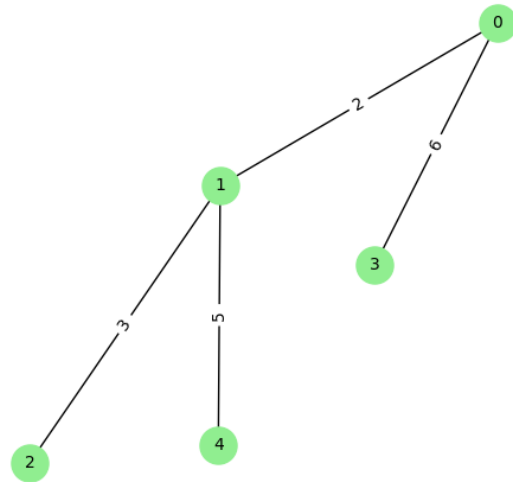
MST:

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

Original Graph



Minimum Spanning Tree (MST)



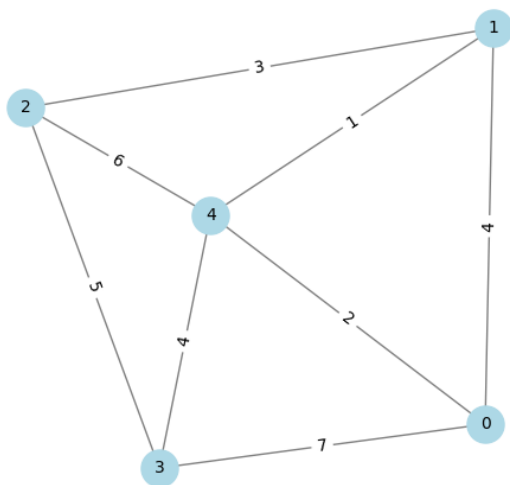
Graph:

```
{ 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 }
```

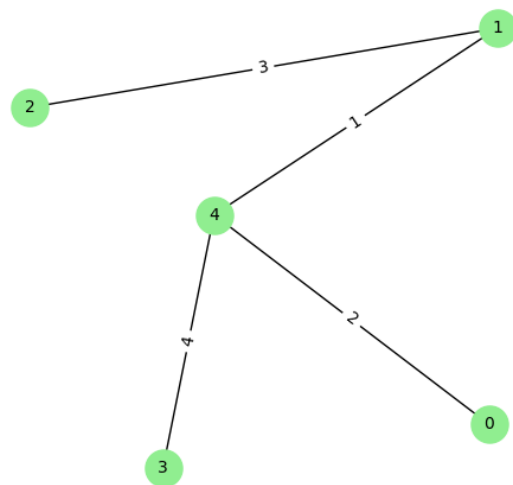
MST:

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

Original Graph



Minimum Spanning Tree (MST)



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

```

1  #include <iostream>
2  #include <vector>
3  #include <queue>
4  #include <climits>
5  using namespace std;
6
7  int extract_min(priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>>& Q) {
8      int vertex = Q.top().second;
9      Q.pop();
10     return vertex;
11 }
12
13 void prim(const vector<vector<pair<int, int>>>& G, int V) {
14     vector<int> key(V, INT_MAX);
15     vector<int> MST(V, -1);
16     vector<bool> selected(V, false);
17
18     priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> Q;
19     key[0] = 0;
20     Q.push({0, 0});
21
22     while (!Q.empty()) {
23         int u = extract_min (Q);
24
25         if (selected[u]) {
26             continue;
27         }
28         selected[u] = true;
29
30         for (const auto& [v, w] : G[u]) {
31             if (!selected[v] && w < key[v]) {
32                 key[v] = w;
33                 MST[v] = u;
34                 Q.push({key[v], v});
35             }
36         }
37     }
38
39     cout << "Edge \tWeight\n";
40     for (int i = 1; i < V; ++i) {
41         if (MST[i] != -1) {
42             cout << MST[i] << " - " << i << "\t" << key[i] << "\n";
43         }
44     }
45 }
46
47 int main() {
48     int V = 5;
49     vector<vector<pair<int, int>>> G(V);
50     G[0] = {{1, 4}, {3, 7}, {4, 2}};
51     G[1] = {{0, 4}, {2, 3}, {4, 1}};
52     G[2] = {{1, 3}, {3, 5}, {4, 6}};
53     G[3] = {{0, 7}, {2, 5}, {4, 4}};
54     G[4] = {{0, 2}, {1, 1}, {2, 6}, {3, 4}};
55
56     prim(G, V);
57     return 0;
58 }

```

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) }

```

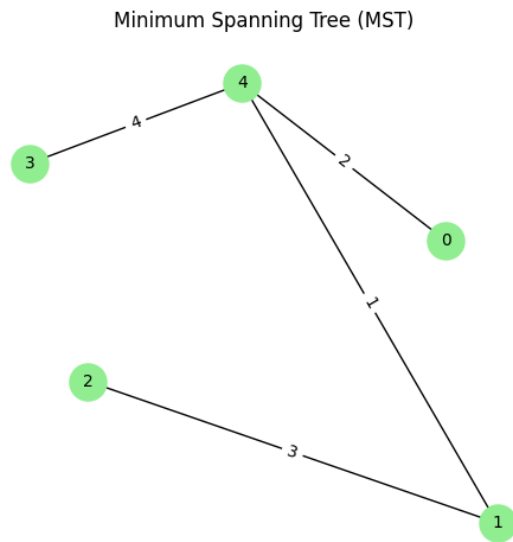
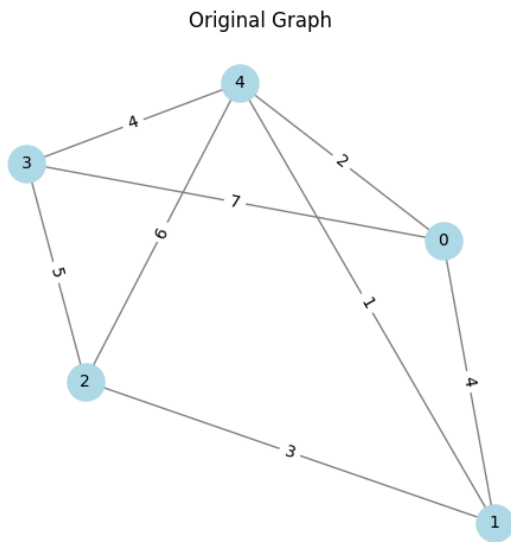
MST:

```

Edge Weight
4 - 1    1

```

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



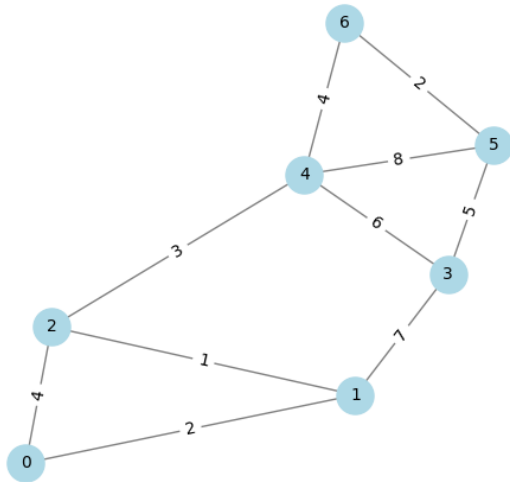
Graph:

$G(0) = \{ (1, 2), (2, 4) \}$
 $G(1) = \{ (0, 2), (2, 1), (3, 7) \}$
 $G(2) = \{ (0, 4), (1, 1), (4, 3) \}$
 $G(3) = \{ (1, 7), (4, 6), (5, 5) \}$
 $G(4) = \{ (2, 3), (3, 6), (5, 8), (6, 4) \}$
 $G(5) = \{ (3, 5), (4, 8), (6, 2) \}$
 $G(6) = \{ (4, 4), (5, 2) \}$

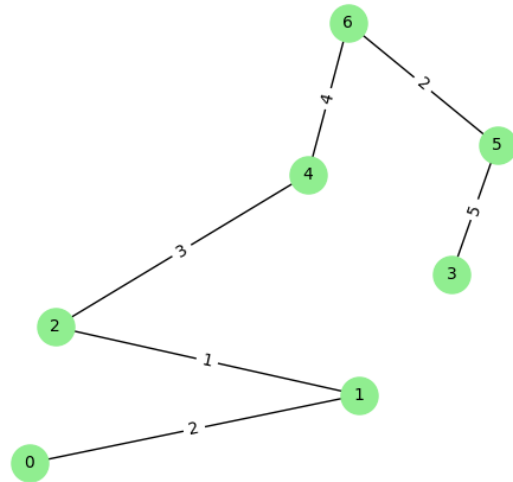
MST:

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

Original Graph



Minimum Spanning Tree (MST)



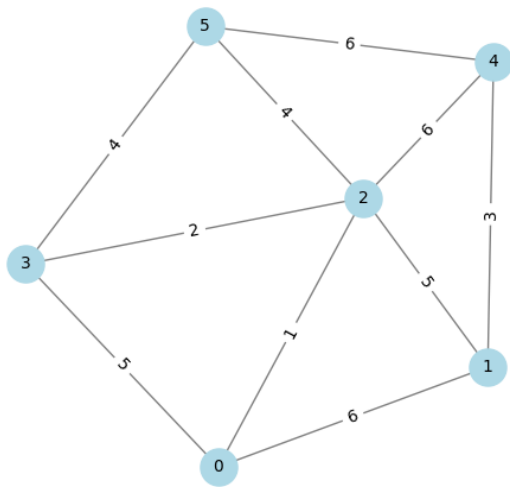
Graph:

$G(0) = \{ (1, 6), (2, 1), (3, 5) \}$
 $G(1) = \{ (0, 6), (2, 5), (4, 3) \}$
 $G(2) = \{ (0, 1), (1, 5), (3, 2), (4, 6), (5, 4) \}$
 $G(3) = \{ (0, 5), (2, 2), (5, 4) \}$
 $G(4) = \{ (1, 3), (2, 6), (5, 6) \}$
 $G(5) = \{ (2, 4), (3, 4), (4, 6) \}$

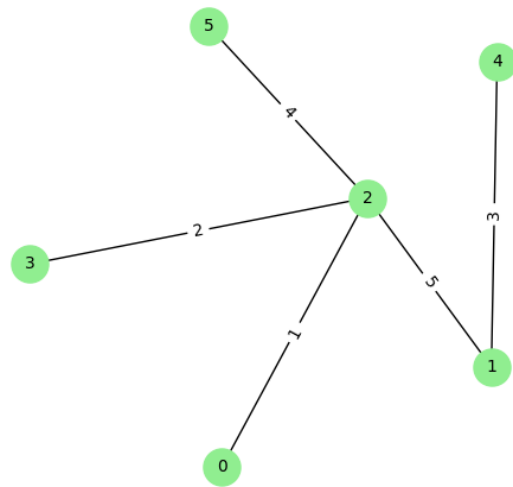
MST:

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

Original Graph



Minimum Spanning Tree (MST)



Question 2.

Dijkstra's Algorithm

```
1 #include <iostream>
2 #include <vector>
3 #include <tuple>
4 #include <map>
5 #include <climits>
6 #include <utility>
7 using namespace std;
8
9 vector<int> dijkstra(int src, int V, map<int, vector<pair<int, int>>>& G) {
10     vector<int> d(V, INT_MAX);
11     vector<bool> visited(V, false);
12     vector<pair<int, int>> Q;
13
14     d[src] = 0;
15     Q.push_back({0, src});
16
17     while (!Q.empty()) {
18         int min_idx = 0;
19         for (int i = 1; i < Q.size(); i++) {
20             if (Q[i].first < Q[min_idx].first) {
21                 min_idx = i;
22             }
23         }
24
25         int dist = Q[min_idx].first;
26         int u = Q[min_idx].second;
27         Q.erase(Q.begin() + min_idx);
28
29         if (visited[u]) {
30             continue;
31         }
32         visited[u] = true;
33
34         if (G.find(u) != G.end()) {
35             for (auto& e : G[u]) {
36                 int v = e.first;
37                 int w = e.second;
38
39                 if (!visited[v] && d[u] + w < d[v]) {
40                     d[v] = d[u] + w;
41                     Q.push_back({d[v], v});
42                 }
43             }
44         }
45     }
46     return d;
47 }
48
49 int main() {
50     int V = 8;
51     map<int, vector<pair<int, int>>> G;
52     G[0] = {{1, 3}, {3, 7}};
53     G[1] = {{2, 1}, {3, 4}};
54     G[2] = {{3, 2}, {4, 5}};
55     G[3] = {{4, 1}};
56     G[4] = {{5, 7}, {6, 3}};
57     G[5] = {{6, 2}, {7, 4}};
58     G[6] = {{7, 6}};
59     G[7] = {};
60
61     int src = 0;
62     vector<int> distances = dijkstra(src, V, G);
63
64     cout << "Vertex\tDistance from Source\n";
65     for (int i = 0; i < distances.size(); i++) {
66         cout << i << "\t" << distances[i] << "\n";
67     }
68     return 0;
69 }
```

Graph:

$G(0) = \{ (1, 2), (2, 4) \}$

$G(1) = \{ (0, 2), (2, 1), (3, 7) \}$

$G(2) = \{ (0, 4), (1, 1), (4, 3) \}$

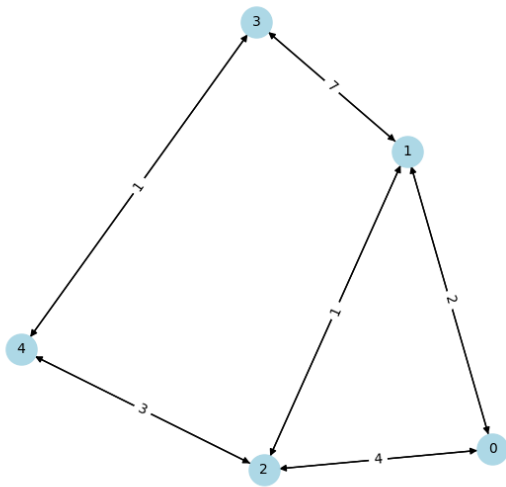
$G(3) = \{ (1, 7), (4, 1) \}$

$G(4) = \{ (2, 3), (3, 1) \}$

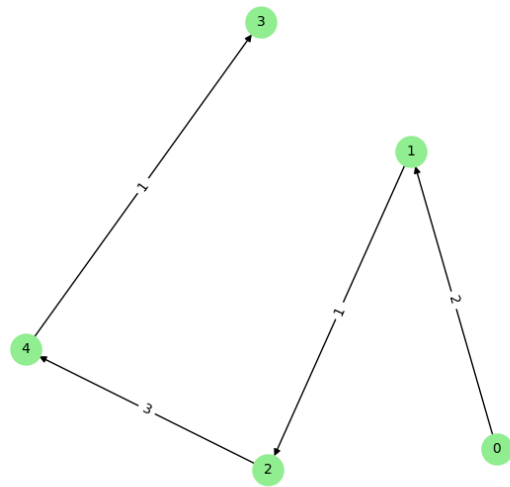
SPT:

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

Original Graph



Shortest Path Tree (SPT from Source)



Graph:

$G(0) = \{ (1, 3), (3, 7) \}$

$G(1) = \{ (2, 1), (3, 4) \}$

$G(2) = \{ (3, 2), (4, 5) \}$

$G(3) = \{ (4, 1) \}$

$G(4) = \{ (5, 7), (6, 3) \}$

$G(5) = \{ (6, 2), (7, 4) \}$

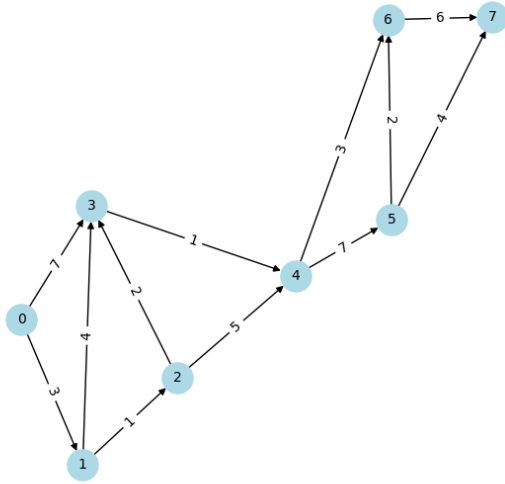
$G(6) = \{ (7, 6) \}$

$G(7) = \{ \}$

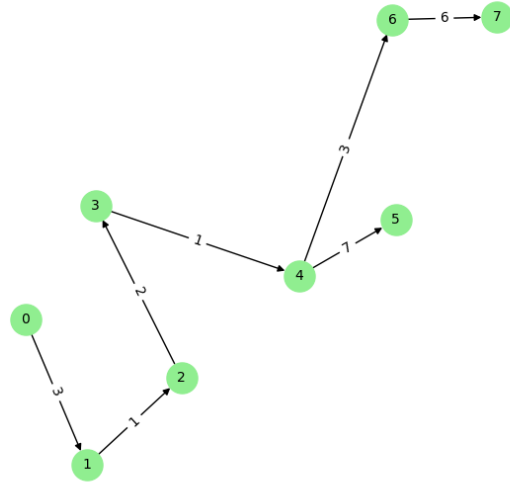
SPT:

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

Original Graph



Shortest Path Tree (SPT from Source)



Bellman-Ford Algorithm

```

1 #include <iostream>
2 #include <vector>
3 #include <climits>
4 #include <map>
5 #include <set>
6 using namespace std;
7
8 bool bellmanFord(int src, int V, map<int, vector<pair<int, int>>>& G) {
9     vector<int> d(V, INT_MAX);
10    d[src] = 0;
11
12    for (int i = 0; i < V - 1; ++i) {
13        bool relaxed = false;
14        for (int u = 0; u < V; ++u) {
15            for (auto& e : G[u]) {
16                int v = e.first;
17                int w = e.second;
18                if (d[u] != INT_MAX && d[u] + w < d[v]) {
19                    d[v] = d[u] + w;
20                    relaxed = true;
21                }
22            }
23        }
24        if (!relaxed){
25            break;
26        }
27    }
28
29    for (int u = 0; u < V; ++u) {
30        if (G.find(u) != G.end()) {
31            for (auto& e : G[u]) {
32                int v = e.first;
33                int w = e.second;
34                if (d[u] != INT_MAX && d[u] + w < d[v]) {
35                    cout << "Graph contains a negative weight cycle." << endl;
36                    return false;
37                }
38            }
39        }
40    }
41
42    cout << "Vertex\tDistance from Source" << endl;
43    for (int i = 0; i < V; ++i) {

```

```

44     if (d[i] == INT_MAX) {
45         cout << i << "\tINF" << endl;
46     } else {
47         cout << i << "\t" << d[i] << endl;
48     }
49 }
50 return true;
51 }
52 int main() {
53     int V = 5;
54     map<int, vector<pair<int, int>>> G;
55     G[0] = {{1, 5}};
56     G[1] = {{2, 1}, {3, 2}};
57     G[2] = {{4, 1}};
58     G[3] = {};
59     G[4] = {{3, -1}};
60
61     int src = 0;
62     bellmanFord(src, V, G);
63     return 0;
64 }

```

Graph:

```

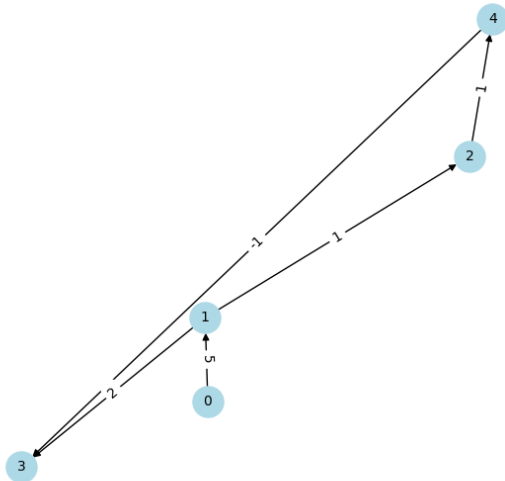
G(0) = { (1, 5) }
G(1) = { (2, 1), (3, 2) }
G(2) = { (4, 1) }
G(3) = { }
G(4) = { (3, -1) }

```

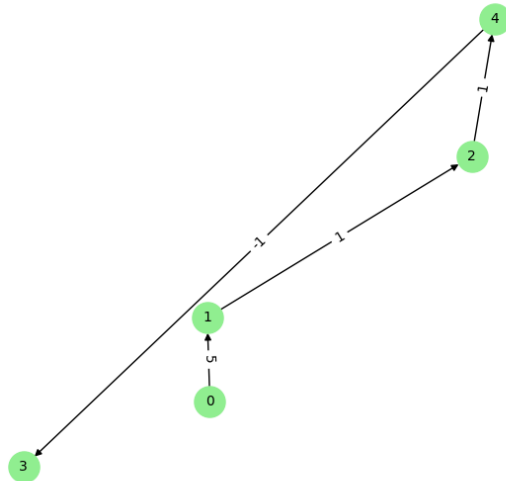
SPT:

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

Graph with Edge Weights



Shortest Path Tree (SPT)



Graph:

```

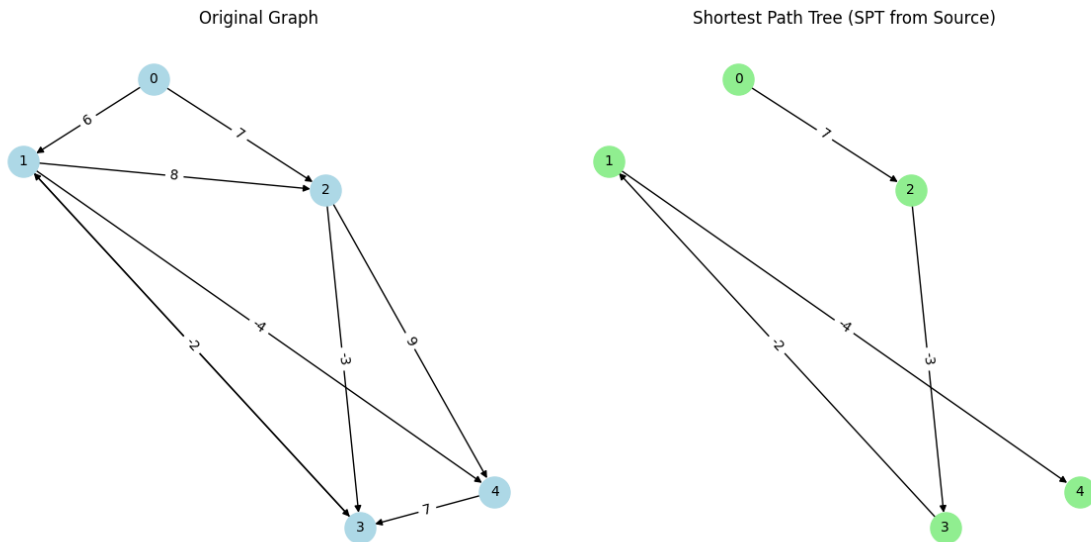
G(0) = { (1, 6), (2, 7) }
G(1) = { (2, 8), (3, 5), (4, -4) }
G(2) = { (3, -3), (4, 9) }

```

$G(3) = \{ (1, -2) \}$
 $G(4) = \{ (3, 7) \}$

SPT:

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



Johnson's Algorithm

```

1 #include <iostream>
2 #include <vector>
3 #include <climits>
4 #include <map>
5 #include <iomanip>
6 using namespace std;
7
8 vector<int> bf_dist;
9 bool bellmanFord(int src, int V, map<int, vector<pair<int, int>>>& G) {
10     vector<int> d(V, INT_MAX);
11     d[src] = 0;
12
13     for (int i = 0; i < V - 1; ++i) {
14         bool relaxed = false;
15         for (int u = 0; u < V; ++u) {
16             for (auto& e : G[u]) {
17                 int v = e.first;
18                 int w = e.second;
19                 if (d[u] != INT_MAX && d[u] + w < d[v]) {
20                     d[v] = d[u] + w;
21                     relaxed = true;
22                 }
23             }
24         }
25         if (!relaxed){
26             break;
27         }
28     }
29
30     for (int u = 0; u < V; ++u) {

```

```

31         if (G.find(u) != G.end()) {
32             for (auto& e : G[u]) {
33                 int v = e.first;
34                 int w = e.second;
35                 if (d[u] != INT_MAX && d[u] + w < d[v]) {
36                     cout << "Graph contains a negative weight cycle." << endl;
37                     return false;
38                 }
39             }
40         }
41     }
42     bf_dist = d;
43     return true;
44 }
45
46 vector<int> dijkstra(int src, int V, map<int, vector<pair<int, int>>>& G) {
47     vector<int> d(V, INT_MAX);
48     vector<bool> visited(V, false);
49     vector<pair<int, int>> Q;
50
51     d[src] = 0;
52     Q.push_back({0, src});
53
54     while (!Q.empty()) {
55         int min_idx = 0;
56         for (int i = 1; i < Q.size(); i++) {
57             if (Q[i].first < Q[min_idx].first) {
58                 min_idx = i;
59             }
60         }
61
62         int dist = Q[min_idx].first;
63         int u = Q[min_idx].second;
64         Q.erase(Q.begin() + min_idx);
65
66         if (visited[u]) {
67             continue;
68         }
69         visited[u] = true;
70
71         if (G.find(u) != G.end()) {
72             for (auto& e : G[u]) {
73                 int v = e.first;
74                 int w = e.second;
75
76                 if (!visited[v] && d[u] + w < d[v]) {
77                     d[v] = d[u] + w;
78                     Q.push_back({d[v], v});
79                 }
80             }
81         }
82     }
83     return d;
84 }
85
86 void johnson(int V, map<int, vector<pair<int, int>>>& G) {
87     for (int u = 0; u < V; ++u) {
88         G[V].emplace_back(u, 0);
89     }
90
91     if (!bellmanFord(V, V + 1, G)) {
92         return;
93     }
94
95     vector<int> h = bf_dist;
96     map<int, vector<pair<int, int>>> new_G;
97
98     cout << left << setw(10) << "Edge" << setw(20) << "Original Weight" << setw(20) << "New Weight" << endl;
99     for (int u = 0; u < V; ++u) {
100         for (auto& [v, w] : G[u]) {
101             int new_w = w + h[u] - h[v];

```

```

102         new_G[u].emplace_back(v, new_w);
103         cout << setw(20) << "(" + to_string(u) + " -> " + to_string(v) + ")"
104             << setw(15) << w
105             << setw(20) << new_w << endl;
106     }
107 }
108 cout << endl;
109
110 vector<vector<int>> all_pairs_distances(V, vector<int>(V, INT_MAX));
111 for (int u = 0; u < V; ++u) {
112     vector<int> distances = dijkstra(u, V, new_G);
113
114     for (int v = 0; v < V; ++v) {
115         if (distances[v] < INT_MAX) {
116             all_pairs_distances[u][v] = distances[v] + h[v] - h[u];
117         }
118     }
119 }
120
121 cout << "All Pairs Shortest Paths in Original Graph G:" << endl;
122 cout << setw(6) << " " << " " << " ";
123 for (int i = 0; i < V; ++i) {
124     cout << setw(8) << ("V" + to_string(i));
125 }
126 cout << endl;
127 for (int u = 0; u < V; ++u) {
128     cout << setw(6) << ("V" + to_string(u)) << " ";
129     for (int v = 0; v < V; ++v) {
130         if (all_pairs_distances[u][v] == INT_MAX)
131             cout << setw(8) << "inf";
132         else
133             cout << setw(8) << all_pairs_distances[u][v];
134     }
135     cout << endl;
136 }
137 }
138
139 int main() {
140     int V = 5;
141     map<int, vector<pair<int, int>>> G;
142     G[0] = {{1, 3}, {2, 8}};
143     G[1] = {{3, 1}, {4, -4}};
144     G[2] = {{4, 2}};
145     G[3] = {{0, 2}, {2, -5}};
146     G[4] = {{3, 6}};
147
148     johnson(V, G);
149     return 0;
150 }

```

Graph:

$G(0) = \{ (1, -5), (2, 2), (3, 3) \}$
 $G(1) = \{ (2, 4) \}$
 $G(2) = \{ (3, 1) \}$
 $G(3) = \{ \}$

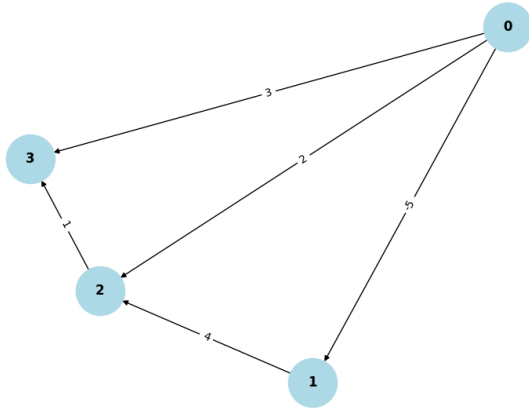
New Graph (After removing all negative weights):

$G(0) = \{ (1, 0), (2, 3), (3, 3) \}$
 $G(1) = \{ (2, 0) \}$
 $G(2) = \{ (3, 0) \}$
 $G(3) = \{ \}$

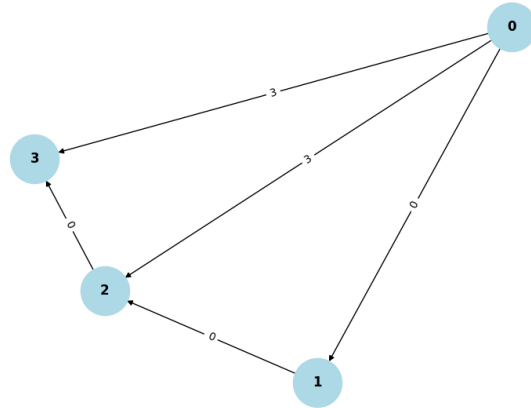
All Pairs Shortest Paths in Original Graph G (After Conversion from Reweighted Graph G'):

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

Original Graph with Negative Edges Weights



Transformed Graph with No Negative Edge Weights



Graph:

$G(0) = \{ (1, 3), (2, 8) \}$
 $G(1) = \{ (3, 1), (4, -4) \}$
 $G(2) = \{ (4, 2) \}$
 $G(3) = \{ (0, 2), (2, -5) \}$
 $G(4) = \{ (3, 6) \}$

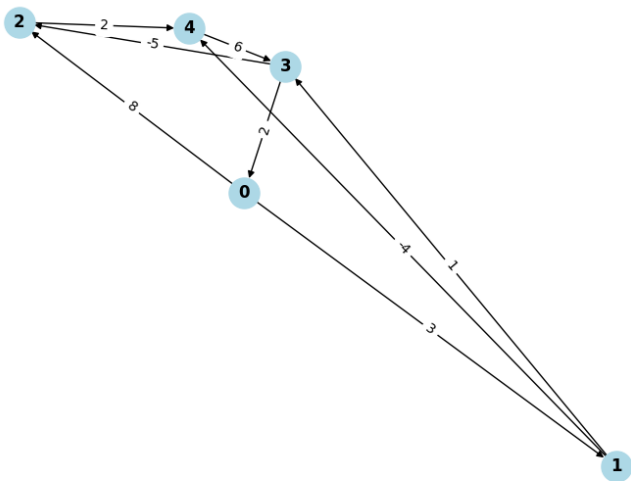
New Graph (After removing all negative weights):

$G(0) = \{ (1, 3), (2, 13) \}$
 $G(1) = \{ (3, 1), (4, 0) \}$
 $G(2) = \{ (4, 1) \}$
 $G(3) = \{ (0, 2), (2, 0) \}$
 $G(4) = \{ (3, 2) \}$

All Pairs Shortest Paths in Original Graph G (After Conversion from Reweighted Graph G'):

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

Original Graph with Negative Edges Weights



Transformed Graph with No Negative Edge Weights

