Omer Seyfeddin Koc
EECE 7205: Fundamentals of Computer Engineering
November 18, 2023

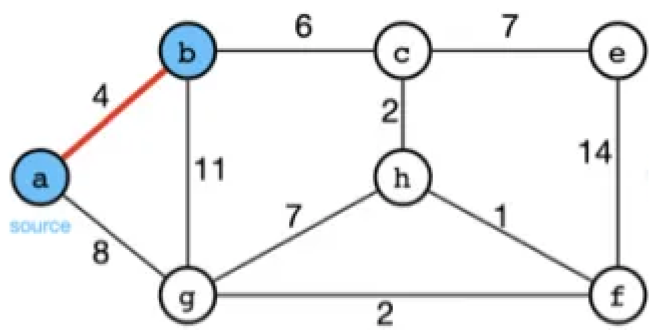
Homework 3

Submission instruction:

- Submit one single pdf file for this homework including both coding problems and analysis problems.
- For coding problems, copy and paste your codes. Report your results.
- For analysis problems, either type or hand-write and scan.

Question 1. (3 pt.) MST: Write codes for Prim's algorithm.

1) **Version 1:** Use adjacency matrix to present graph and use unsorted array for priority queue Q.
Solution: Prim's method constitutes a greedy algorithm employed for identifying the minimum spanning tree for a weighted undirected graph. This method is analogous to the algorithm commonly referred to as 'shortest path'. For this question, I used the following graph as an example.



For the graph depicted in the example, the adjacency matrix representation is as follows:

$$\begin{bmatrix} 0 & 4 & 0 & 0 & 8 & 0 & 0 \\ 4 & 0 & 6 & 0 & 11 & 0 & 0 \\ 0 & 6 & 0 & 7 & 0 & 2 & 0 \\ 0 & 0 & 7 & 0 & 0 & 0 & 14 \\ 8 & 11 & 0 & 0 & 0 & 7 & 2 \\ 0 & 0 & 2 & 0 & 7 & 0 & 1 \\ 0 & 0 & 0 & 14 & 2 & 1 & 0 \end{bmatrix}$$

The C++ program for the relevant question is as follows:

```cpp
// Required libraries
#include <iostream>
#include <vector>
#include <climits>

using namespace std;

// Structure representing a vertex in the graph
struct Node {
    int key;      // Minimum weight to connect to the MST
    int parent;   // The vertex's parent in the MST
    bool inMST;   // Flag to check if the vertex is in the MST
};

```

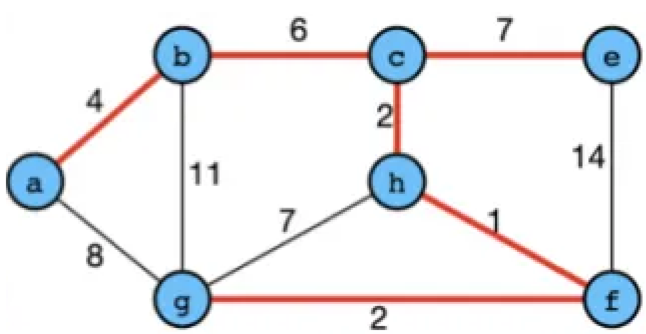
```cpp
// Function to extract the minimum key vertex from the set of vertices not yet included in MST
int extractMin(vector<Node>& nodes) {
    int minKey = INT_MAX;
    int minIndex = -1;

    // Loop through all vertices to find the minimum key vertex
    for (int i = 0; i < nodes.size(); ++i) {
        // If vertex is not in MST and its key is less than current minimum, update minimum
        if (!nodes[i].inMST && nodes[i].key < minKey) {
            minKey = nodes[i].key;
            minIndex = i;
        }
    }

    return minIndex;
}

// Function to construct and print the MST for a graph represented using adjacency matrix
void primMST(vector<vector<int>>& graph) {
    int numVertices = graph.size();
    vector<Node> nodes(numVertices);

    // Initialize all keys as INFINITE and MST as false
    for (int i = 0; i < numVertices; ++i) {
        nodes[i].key = INT_MAX;
        nodes[i].parent = -1;
        nodes[i].inMST = false;
    }

    // Start with the first vertex by setting its key to 0
    nodes[0].key = 0;

    // The MST will have numVertices vertices
    for (int count = 0; count < numVertices - 1; ++count) {
        // Pick the minimum key vertex from the set of vertices not yet included in MST
        int u = extractMin(nodes);
        nodes[u].inMST = true;

        // Update key and parent index of the adjacent vertices of the picked vertex
        for (int v = 0; v < numVertices; ++v) {
            // Update the key only if graph[u][v] is smaller than the key of v
            if (graph[u][v] && !nodes[v].inMST && graph[u][v] < nodes[v].key) {
                nodes[v].parent = u;
                nodes[v].key = graph[u][v];
            }
        }
    }

    // Print the constructed MST
    cout << "Edge   Weight" << endl;
    for (int i = 1; i < numVertices; ++i) {
        cout << nodes[i].parent << " - " << i << "    " << graph[i][nodes[i].parent] << endl;
    }
}
```

```
int main() {
    // Example graph represented as an adjacency matrix
    vector<vector<int>> graph = {
        {0, 4, 0, 0, 8, 0, 0},
        {4, 0, 6, 0, 11, 0, 0},
        {0, 6, 0, 7, 0, 2, 0},
        {0, 0, 7, 0, 0, 0, 14},
        {8, 11, 0, 0, 0, 7, 2},
        {0, 0, 2, 0, 7, 0, 1},
        {0, 0, 0, 14, 2, 1, 0}
    };

    // Run Prim's MST algorithm
    primMST(graph);

    return 0;
}
```

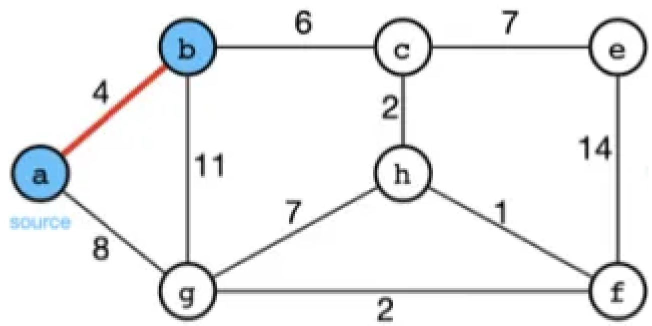
In my answer, this code implements Prim's algorithm to find the Minimum Spanning Tree (MST) of a weighted, undirected graph represented as an adjacency matrix. The algorithm maintains a set of vertices not yet included in the MST and repeatedly selects the vertex with the smallest key value. For each selected vertex, it updates the key values of its adjacent vertices and marks the vertex as part of the MST. Finally, the algorithm prints the edges and weights of the constructed MST, displaying the connection between each vertex and its parent in the MST. The output of this code is as follows.

Edge	Weight
0 - 1	4
1 - 2	6
2 - 3	7
6 - 4	2
2 - 5	2
5 - 6	1

In the "Edge" column, the starting and ending vertices of the edges to be used are specified. In the "Weight" column, the weight value of that edge is indicated. This output represents the edges included in the MST along with their weights. The format a - b indicates an edge from vertex a to vertex b with the given weight. For instance, 0 - 1 4 means there is an edge between vertex 0 and vertex 1 with a weight of 4. By combining all these edges, we can find the Minimum Spanning Tree. The output of my code, which constitutes the Minimum Spanning Tree formed by the edges, is as follows:



2) Version 2: Use adjacency lists to present graph and use heap for priority queue Q.
Solution: Prim's MST algorithm is used to construct a Minimum Spanning Tree in a weighted graph. In this question, I will use the same example as in the previous question, but this time I will represent my graph using an adjacency list. For this question, I used again the following graph as an example.



The graph in the example is represented as an adjacency list, where each vertex has a list of Edge structures. Each Edge contains the destination vertex and the weight of the edge. The representation of this graph as an adjacency list is as follows:

- {(1, 4), (4, 8)},
- {(0, 4), (2, 6), (4, 11)},
- {(1, 6), (3, 7), (5, 2)},
- {(2, 7), (6, 14)},
- {(0, 8), (1, 11), (5, 7), (6, 2)},
- {(2, 2), (4, 7), (6, 1)},
- {(3, 14), (4, 2), (5, 1)}

The C++ program for the relevant question is as follows:

```cpp
// Required libraries
#include <iostream>
#include <vector>
#include <queue>
#include <climits>

using namespace std;

// Structure to represent a weighted edge in the graph
struct Edge {
    int to;     // Destination vertex of the edge
    int weight; // Weight of the edge
};

// Overload the operator to compare edges based on weight
// This will make the minimum weight edge to be on top of the heap
bool operator>(const Edge &a, const Edge &b) {
    return a.weight > b.weight;
}

// Function to construct and print the MST using adjacency list and min heap
void primMSTUsingHeap(vector<vector<Edge>>& graph) {
    int numVertices = graph.size();
    vector<bool> inMST(numVertices, false);
    vector<int> keys(numVertices, INT_MAX);
    vector<int> parent(numVertices, -1);

    // Min heap to store vertices based on the minimum 'key' value
```

```
    priority_queue<Edge, vector<Edge>, greater<Edge>> minHeap;

    // Start with the first vertex
    keys[0] = 0;
    minHeap.push({0, 0}); // Insert source vertex with weight 0

    while (!minHeap.empty()) {
        // Extract the vertex with minimum key value
        int u = minHeap.top().to;
        minHeap.pop();

        if (inMST[u]) continue; // Skip if vertex is already included in MST

        inMST[u] = true; // Include vertex in MST

        // Iterate over all the adjacent vertices of u
        for (auto &edge : graph[u]) {
            int v = edge.to;
            int weight = edge.weight;

            // If v is not in MST and weight of (u,v) is smaller than current key of v
            if (!inMST[v] && keys[v] > weight) {
                // Update key of v
                keys[v] = weight;
                parent[v] = u;
                minHeap.push({v, keys[v]});
            }
        }
    }

    // Print the edges of MST
    cout << "Edge   Weight\n";
    for (int i = 1; i < numVertices; ++i) {
        cout << parent[i] << " - " << i << "    " << keys[i] << "\n";
    }
}

int main() {
    // Example graph represented as an adjacency list
    vector<vector<Edge>> graph = {
        // Adjacency list for each vertex
        {{1, 4}, {4, 8}},        // Edges from vertex 0
        {{0, 4}, {2, 6}, {4, 11}}, // Edges from vertex 1
        {{1, 6}, {3, 7}, {5, 2}},  // Edges from vertex 2
        {{2, 7}, {6, 14}},       // Edges from vertex 3
        {{0, 8}, {1, 11}, {5, 7}, {6, 2}}, // Edges from vertex 4
        {{2, 2}, {4, 7}, {6, 1}}, // Edges from vertex 5
        {{3, 14}, {4, 2}, {5, 1}}  // Edges from vertex 6
    };

    primMSTUsingHeap(graph);

    return 0;
}
```

The provided C++ code implements Prim's algorithm. The graph is represented using an adjacency list, where each vertex stores a list of its edges and corresponding weights. The priority queue is managed using a min heap, implemented with **priority_queue**, to efficiently determine the next vertex to be included in the MST with the minimum edge weight. The output of the program for the provided graph is as follows:

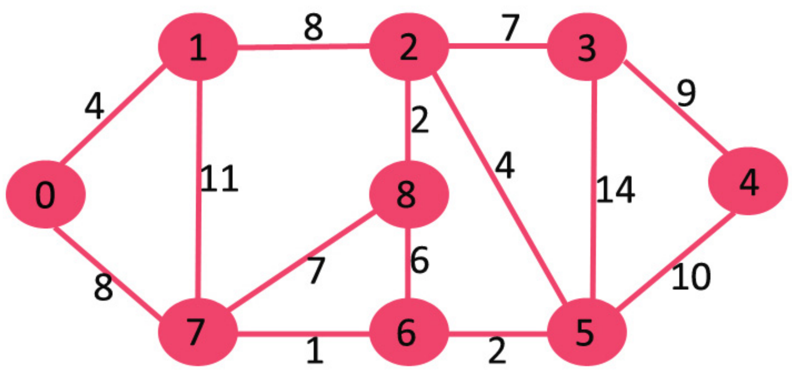
Edge	Weight
0 - 1	4
1 - 2	6
2 - 3	7
6 - 4	2
2 - 5	2
5 - 6	1

This output represents the edges included in the MST along with their weights. The implementation correctly identifies the Minimum Spanning Tree for the given graph. The use of a min heap for the priority queue significantly enhances the efficiency of the algorithm. The resulting MST connects all vertices with the minimum total edge weight, demonstrating the effectiveness of Prim's algorithm in finding an optimal spanning tree in a weighted graph.

Question 2. (7 pt.) Johnsen's Algorithm: Write codes for Johnsen's algorithm using unsorted array for priority Q. This algorithm involves Dijkstra's and Bellman-Ford algorithms, so you need to write and test codes for those two algorithms first.

Solution:

Djikstra's Algortihm: I will first test my Dijkstra algorithm implementation. For this purpose, I will check my Dijkstra algorithm using an example. The example graph is as follows:

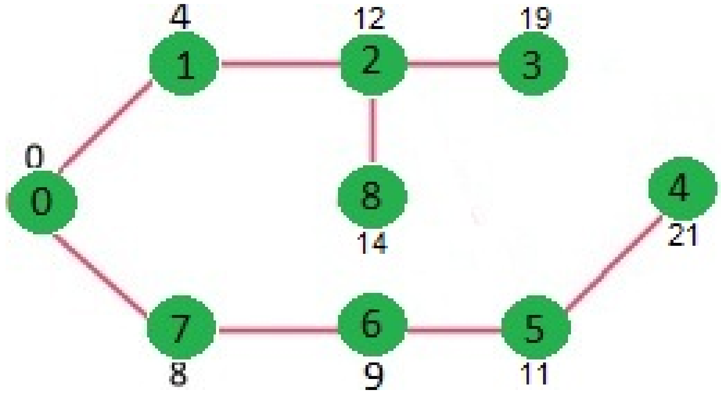


The Djikstra's Algorithm C++ code for this example is as follows:

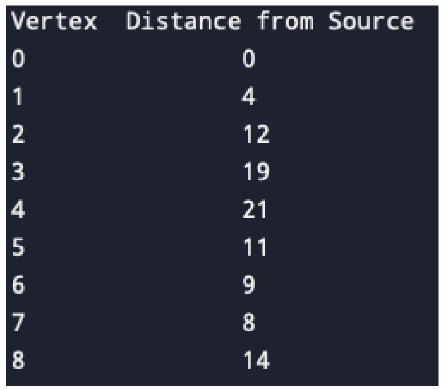
```
#include <iostream>
#include <vector>
#include <queue>
#include <limits>

using namespace std;

// Function to implement Dijkstra's algorithm
void dijkstra(int src, const vector<vector<pair<int, int>>>& adjList, int V, vector<int
    >& dist) {
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    dist.assign(V, numeric_limits<int>::max());
    dist[src] = 0;
    pq.push({0, src});

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

```

        for (auto& p : adjList[u]) {
            int v = p.first;
            int weight = p.second;

            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.push({dist[v], v});
            }
        }
    }
}

int main() {
    int V = 9; // Number of vertices
    vector<vector<pair<int, int>>> adjList(V);

    // Adding edges to the graph for example
    adjList[0].push_back({1, 4});
    adjList[0].push_back({7, 8});
    adjList[1].push_back({2, 8});
    adjList[1].push_back({7, 11});
    adjList[2].push_back({1, 8});
    adjList[2].push_back({3, 7});
    adjList[2].push_back({5, 4});
    adjList[2].push_back({8, 2});
    adjList[3].push_back({2, 7});
    adjList[3].push_back({4, 9});
    adjList[3].push_back({5, 14});
    adjList[4].push_back({3, 9});
    adjList[4].push_back({5, 10});
    adjList[5].push_back({2, 4});
    adjList[5].push_back({3, 14});
    adjList[5].push_back({4, 10});
    adjList[5].push_back({6, 2});
    adjList[6].push_back({5, 2});
    adjList[6].push_back({7, 1});
    adjList[6].push_back({8, 6});
    adjList[7].push_back({0, 8});
    adjList[7].push_back({1, 11});
    adjList[7].push_back({6, 1});
    adjList[7].push_back({8, 7});
    adjList[8].push_back({2, 2});
    adjList[8].push_back({6, 6});
    adjList[8].push_back({7, 7});

    // Vector to store distances
    vector<int> dist(V);

    // Run Dijkstra's algorithm from vertex 0
    dijkstra(0, adjList, V, dist);

    // Output the results
    cout << "Vertex\tDistance from Source" << endl;
    for (int i = 0; i < V; ++i) {
        cout << i << "\t\t\t\t" << dist[i] << endl;
    }

    return 0;
}
```

The graphical solution of this example is as follows:



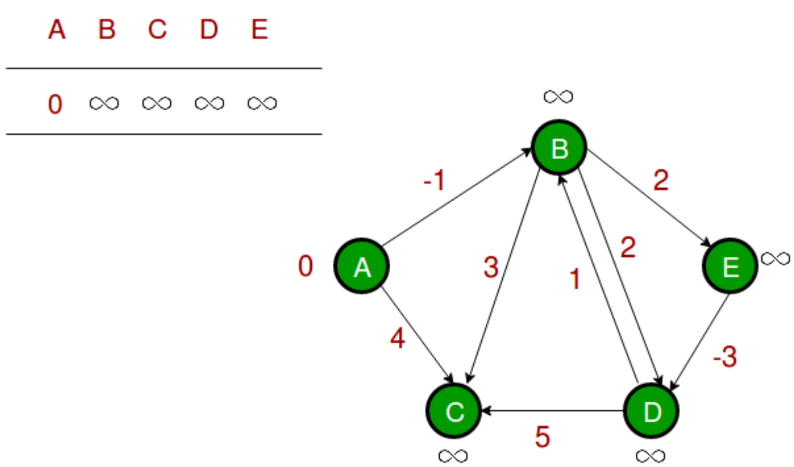
The output of my code is as follows:



Vertex	Distance from Source
0	0
1	4
2	12
3	19
4	21
5	11
6	9
7	8
8	14

As can be seen, the output of my algorithm matches the solution of the example. This indicates that the test of the Dijkstra algorithm has been successfully passed.

Bellman-Ford Algortihm: Secondly, I will test the Bellman-Ford algorithm. For this, I will use a graph that includes negative edges as an example. After applying the Bellman-Ford algorithm to this graph, which contains edges with negative weights, I will compare the results. The example graph is as follows:



The Bellman-Ford's Algortihm C++ code for this example is as follows:

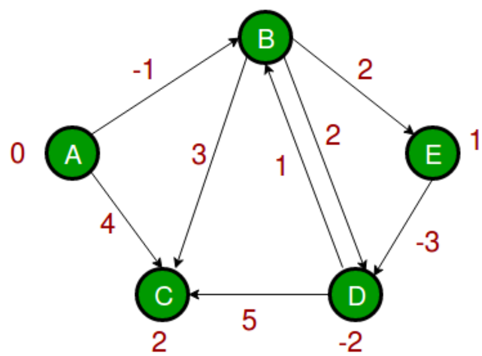
```
#include <iostream>
#include <vector>
#include <limits>

using namespace std;

// Assume we have an Edge structure defined as follows:
```

```
struct Edge {
    int src, dest, weight;
};

class Graph {
public:
    int V; // Number of vertices
    vector<Edge> edges; // All edges in the graph

    // Function to add an edge to the graph
    void addEdge(int src, int dest, int weight) {
        Edge edge = {src, dest, weight};
        edges.push_back(edge);
    }

    // Function to implement Bellman-Ford algorithm
    bool bellmanFord(int src, vector<int>& dist) {
        dist.assign(V, numeric_limits<int>::max());
        dist[src] = 0;

        for (int i = 0; i < V - 1; i++) {
            for (auto& edge : edges) {
                int u = edge.src;
                int v = edge.dest;
                int weight = edge.weight;
                if (dist[u] != numeric_limits<int>::max() && dist[u] + weight < dist[v
    ]) {
                    dist[v] = dist[u] + weight;
                }
            }
        }

        // Check for negative weight cycles
        for (auto& edge : edges) {
            int u = edge.src;
            int v = edge.dest;
            int weight = edge.weight;
            if (dist[u] != numeric_limits<int>::max() && dist[u] + weight < dist[v]) {
                return false; // Graph contains a negative weight cycle
            }
        }
        return true; // No negative weight cycles found
    }
};

int main() {
    Graph g;
    g.V = 5; // Example number of vertices

    // Adding edges according to the example graph
    g.addEdge(0, 1, -1); // A to B
    g.addEdge(0, 2, 4);  // A to C
    g.addEdge(1, 2, 3);  // B to C
    g.addEdge(1, 3, 2);  // B to D
    g.addEdge(1, 4, 2);  // B to E
    g.addEdge(3, 2, 5);  // D to C
    g.addEdge(3, 1, 1);  // D to B
    g.addEdge(4, 3, -3); // E to D


```

```
    vector<int> dist(g.V);
    if (g.bellmanFord(0, dist)) { // Run Bellman-Ford from vertex 0
        cout << "Vertex\tDistance from Source" << endl;
        for (int i = 0; i < g.V; i++) {
            cout << i << "\t\t\t\t" << dist[i] << endl;
        }
    } else {
        cout << "Graph contains a negative weight cycle" << endl;
    }

    return 0;
}
```

The graphical solution of this example is as follows:

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1

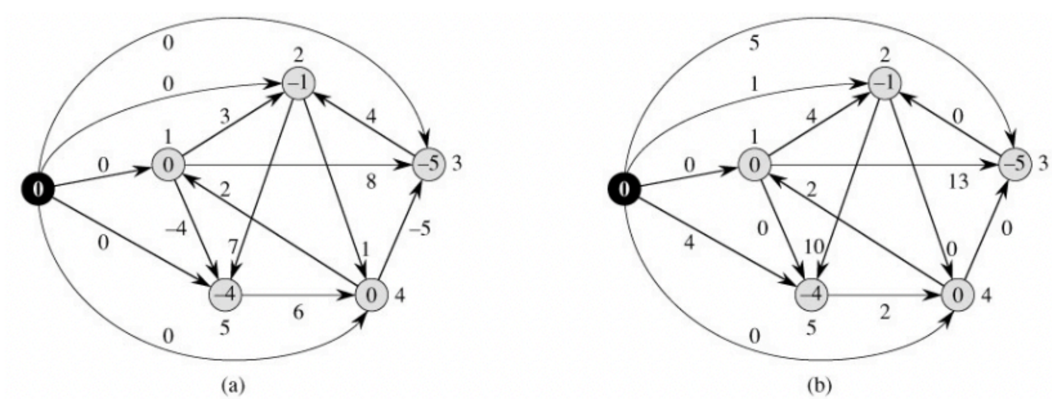


The output of my code is as follows:

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

As can be seen, the output of my algorithm matches the solution of the example. This indicates that the test of the Bellman-Ford Algortihm has been successfully passed.

Johnsen's Algorithm: Implementing Johnson's algorithm in code is a complex task, as it involves the integration of both Dijkstra's and Bellman-Ford algorithms (as provided above). Johnson's algorithm is used for finding the shortest paths between all pairs of vertices in a weighted, directed graph. It first uses Bellman-Ford to reweight the edges to remove any negative weights and then applies Dijkstra's algorithm to find the shortest paths. The example graph is as follows:



The Johnsen's Algorithm C++ code for this example is as follows:

```cpp
#include <iostream>
#include <vector>
#include <limits>
#include <queue>
#include <utility>
#include <stack>
#include <iomanip>  // For setw and left, used in formatting the output

using namespace std;

// Structure to represent an edge in a graph. It contains a source, a destination, and a weight
struct Edge {
    int src, dest, weight;
};

// Structure to represent a graph.
struct Graph {
    int V, E;  // Number of vertices (V) and edges (E)
    vector<Edge> edges;  // List of all edges in the graph
    vector<vector<pair<int, int>>> adjList;  // Adjacency list for the graph

    // Function to add an edge to the graph
    void addEdge(int u, int v, int w) {
        edges.push_back({u, v, w});
    }

    // Bellman-Ford algorithm to find the shortest path from a single source to all other vertices
    bool bellmanFord(int src, vector<int>& dist) {
        dist.assign(V, numeric_limits<int>::max());  // Initialize distances as infinite
        dist[src] = 0;  // Distance from the source to itself is zero

        // Relaxation of all edges V-1 times
        for (int i = 0; i < V - 1; i++) {
            for (auto& edge : edges) {
                int u = edge.src;
                int v = edge.dest;
                int weight = edge.weight;
                if (dist[u] != numeric_limits<int>::max() && dist[u] + weight < dist[v]) {
                    dist[v] = dist[u] + weight;
                }
            }
        }

        // Check for negative weight cycles
        for (auto& edge : edges) {
            int u = edge.src;
            int v = edge.dest;
            int weight = edge.weight;
            if (dist[u] != numeric_limits<int>::max() && dist[u] + weight < dist[v]) {
                return false;  // Graph contains a negative weight cycle
            }
        }

        return true;
```

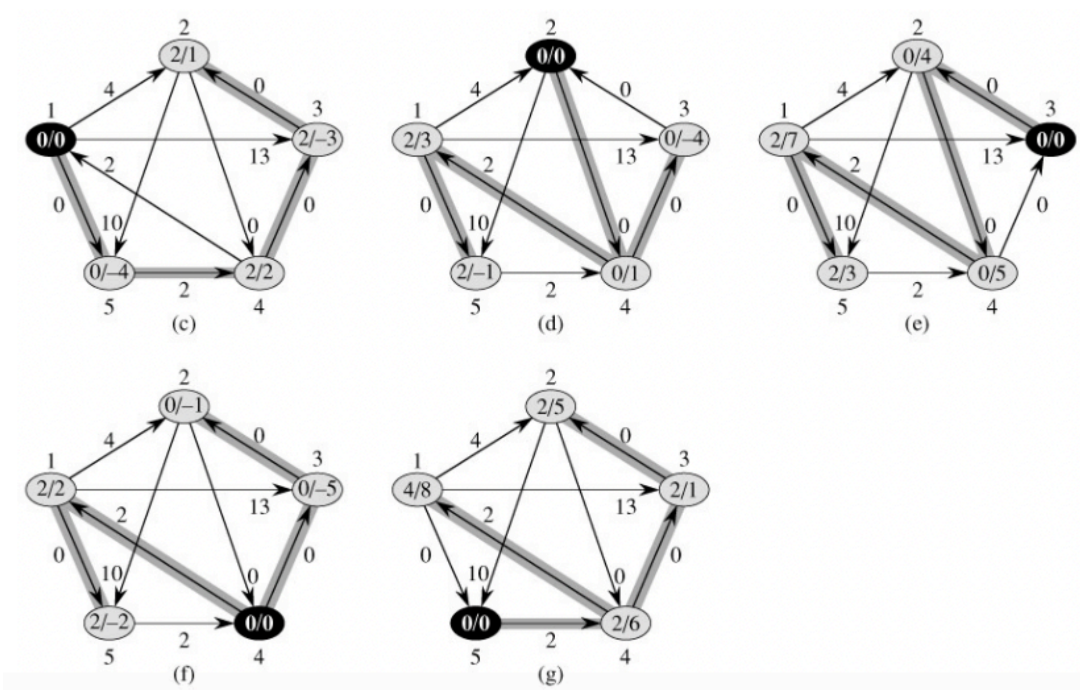
```cpp
    }

    // Function to create an adjacency list from the edge list for efficient traversal
    void createAdjList() {
        adjList.resize(V);
        for (auto& edge : edges) {
            adjList[edge.src].push_back({edge.dest, edge.weight});
        }
    }

    // Dijkstra's algorithm to find the shortest path from a single source to all other nodes
    void dijkstra(int src, vector<int>& dist, vector<int>& prev) {
        priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
        dist.assign(V, numeric_limits<int>::max());
        prev.assign(V, -1);
        dist[src] = 0;
        pq.push({0, src});

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

            for (auto& p : adjList[u]) {
                int v = p.first;
                int weight = p.second;

                if (dist[u] + weight < dist[v]) {
                    dist[v] = dist[u] + weight;
                    prev[v] = u;
                    pq.push({dist[v], v});
                }
            }
        }
    }

    // Function to calculate the total weight of a path from source to destination
    int calculatePathWeight(int src, int dest, const vector<int>& prev) {
        int totalWeight = 0;
        int current = dest;

        while (current != src && current != -1) {
            int u = prev[current];
            // Find the weight of the edge from u to current
            for (auto &p : adjList[u]) {
                if (p.first == current) {
                    totalWeight += p.second;
                    break;
                }
            }
            current = u;
        }

        return totalWeight;
    }
};

int main() {
    Graph g;
```

```cpp
    g.V = 5;  // Set number of vertices
    g.E = 9;  // Set number of edges

    // Adding edges to the graph
    g.addEdge(0, 4, -4);
    g.addEdge(0, 1, 3);
    g.addEdge(0, 2, 8);
    g.addEdge(1, 4, 7);
    g.addEdge(1, 3, 1);
    g.addEdge(2, 1, 4);
    g.addEdge(3, 2, -5);
    g.addEdge(3, 0, 2);
    g.addEdge(4, 3, 6);

    vector<int> h(g.V); // Vector to store distances from the source vertex
    vector<vector<int>> distanceMatrix(g.V, vector<int>(g.V, numeric_limits<int>::max())); // Matrix to store shortest path distances

    // Run Bellman-Ford algorithm to check for negative weight cycles and prepare for Dijkstra
    if (g.bellmanFord(0, h)) {
        g.createAdjList(); // Create adjacency list for efficient traversal

        // Run Dijkstra's algorithm for each vertex as the source
        for (int src = 0; src < g.V; src++) {
            vector<int> dist, prev;
            g.dijkstra(src, dist, prev);

            // Calculate the shortest path distances for each destination
            for (int dest = 0; dest < g.V; dest++) {
                if (dist[dest] != numeric_limits<int>::max()) {
                    distanceMatrix[src][dest] = g.calculatePathWeight(src, dest, prev);
                }
            }
        }

        // Print the shortest path distances matrix
        cout << "\nShortest Path Distances Matrix:\n";
        const int cellWidth = 5;  // Cell width for formatting output
        for (int i = 0; i < g.V; i++) {
            for (int j = 0; j < g.V; j++) {
                cout << setw(cellWidth) << left;  // Set cell width and alignment
                if (distanceMatrix[i][j] == numeric_limits<int>::max()) {
                    cout << "INF";
                } else {
                    cout << distanceMatrix[i][j];
                }
            }
            cout << endl;
        }
    } else {
        cout << "Graph contains negative weight cycle" << endl;
        return 1;
    }

    return 0;
}
```

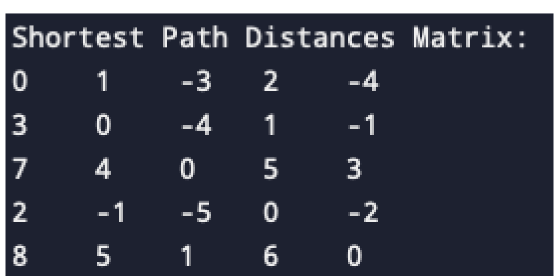
This code implements the Bellman-Ford and Dijkstra's algorithms to find the shortest paths in a weighted directed graph. Here's a brief overview:

- **Graph and Edge Structures:** The graph is represented by a Graph structure, containing the number of vertices (V), a list of edges (edges), and an adjacency list (adjList). Edge structures represent the graph's edges with source (src), destination (dest), and weight.
- **Adding Edges:** The addEdge function adds new edges to the graph, populating the edges list.
- **Bellman-Ford Algorithm:** Used to check for negative weight cycles in the graph and calculate the shortest paths from a single source to all vertices. It returns false if a negative cycle is detected.
- **Creating an Adjacency List:** The createAdjList function constructs the graph's adjacency list for efficient traversal.
- **Dijkstra's Algorithm:** Finds the shortest paths from a given source vertex to all other vertices in the graph.
- **Calculating Path Weights:** The calculatePathWeight function computes the total weight of the shortest path between two vertices.
- **Main Function:** The graph is initialized with vertices and edges. Bellman-Ford is run to ensure there are no negative cycles. Then, Dijkstra's algorithm is executed for each vertex, and the results are stored in a distance matrix. Finally, this matrix is printed, showing the shortest path weights between all vertex pairs.
- In summary, the code efficiently finds and displays the shortest path weights in a weighted directed graph, even with negative edge weights.

The graphical solution of this example is as follows:



The output of my code is as follows:



Shortest Path Distances Matrix:

0	1	-3	2	-4
3	0	-4	1	-1
7	4	0	5	3
2	-1	-5	0	-2
8	5	1	6	0

This output provides the shortest distance values for each source. The values in this matrix correspond to the values obtained in the graphical solution above. As can be seen, the output of my algorithm matches the solution of the example. This demonstrates that the Johnson Algorithm is working successfully.