FUNDAMENTALS OF COMPUTER ENGINEERING

HOMEWORK - 4.

AKSHATA KUMBLE

①

PRIM'S ALGORITHM

Prim's algorithm finds the minimum spanning tree (MST) of a connected, weighted graph. It grows the MST one edge at a time, starting from an arbitrary vertex. At each step:

1. Choose the smallest edge that connects a vertex in the MST to a vertex outside it.
2. Add this edge and the corresponding vertex to the MST

eg:

Version 1: Using Adjacency Matrix and Unsorted array.

The graph is represented using an adjacency matrix and an unsorted array is used to find the next minimum weight edge.

main.cpp

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

// function to implement Prim's Algorithm using adjacency matrix and unsorted array
void primAdjMatrix(int n, vector<vector<int>>& graph) {
    vector<int> key(n, INT_MAX); // minimum weights
    vector<bool> inMST(n, false);
    vector<int> parent(n, -1); // store MST edges
    vector<vector<int>> mstMatrix(n, vector<int>(n, 0)); // store MST

    key[0] = 0; // start from first vertex

    for (int count = 0; count < n - 1; ++count) {
        // findng vertex with minimum key value
        int minKey = INT_MAX, u = -1;
        for (int v = 0; v < n; ++v) {
            if (!inMST[v] && key[v] < minKey) {
                minKey = key[v];
                u = v;
            }
        }

        inMST[u] = true;

        // updating key values of adjacent vertices
        for (int v = 0; v < n; ++v) {
            if (graph[u][v] && !inMST[v] && graph[u][v] < key[v]) {
                key[v] = graph[u][v];
                parent[v] = u;
            }
        }
    }

```

Find the vertex with the smallest key not in MST. Update keys of its adjacent vertices if a smaller weight is found. Then the edges and weights of the MST are printed.

```cpp
    // construct MST adjacency matrix
    for (int i = 1; i < n; ++i) {
        mstMatrix[parent[i]][i] = graph[parent[i]][i];
        mstMatrix[i][parent[i]] = graph[parent[i]][i];
    }

    // print MST adjacency matrix
    cout << "MST Adjacency Matrix:\n";
    for (const auto& row : mstMatrix) {
        for (int val : row) {
            cout << val << " ";
        }
        cout << endl;
    }

    // print MST edges
    cout << "Edges in MST:\n";
    for (int i = 1; i < n; ++i) {
        cout << parent[i] + 1 << " - " << i + 1 << " (Weight: " << graph[parent[i]][i] << ")\n";
    }
}

int main() {
    int n;
    cout << "Enter the number of vertices: ";
    cin >> n;

    vector<vector<int>> graph(n, vector<int>(n));
    cout << "Enter the adjacency matrix:\n";
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            cin >> graph[i][j];

    auto start = chrono::high_resolution_clock::now();
    primAdjMatrix(n, graph);
    auto end = chrono::high_resolution_clock::now();

    chrono::duration<double> elapsed = end - start;
    cout << "Time taken: " << elapsed.count() << " seconds\n";
    return 0;
}
```

```
Enter the number of vertices: 7
Enter the adjacency matrix:
0 28 0 0 0 10 0
28 0 16 0 0 0 14
0 16 0 12 0 0 0
0 0 12 0 22 0 18
0 0 0 22 0 25 24
10 0 0 0 25 0 0
0 14 0 18 24 0 0
MST Adjacency Matrix:
0 0 0 0 0 10 0
0 0 16 0 0 0 14
0 16 0 12 0 0 0
0 0 12 0 22 0 0
0 0 0 22 0 25 0
10 0 0 0 25 0 0
0 14 0 0 0 0 0
Edges in MST:
3 - 2 (Weight: 16)
4 - 3 (Weight: 12)
5 - 4 (Weight: 22)
6 - 5 (Weight: 25)
1 - 6 (Weight: 10)
2 - 7 (Weight: 14)
Time taken: 6.3711e-05 seconds


...Program finished with exit code 0
Press ENTER to exit console.
```

Running time analysis:

Outer loop $O(V)$ for V vertices

Inner loop $O(V)$ for finding minimum key

Total $O(V^2)$

Suitable for dense graphs

Version 2: Using Adjacency List and heap

The graph is represented using an adjacency list and a min-heap (priority queue) is used for efficient minimum weight selection.

main.cpp

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

typedef pair<int, int> pii;

// function to implement Prim's Algorithm using adjacency list and heap
void primAdjList(int n, vector<vector<pii>>& adj) {
    priority_queue<pii, vector<pii>, greater<pii>> pq; // min-heap
    vector<bool> inMST(n, false);
    vector<int> parent(n, -1); // store MST edges
    vector<int> key(n, INT_MAX); // minimum weights
    vector<vector<pii>> mstAdjList(n); // store the MST adjacency list

    key[0] = 0;
    pq.push({0, 0}); // {weight, vertex}

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

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

        for (const auto& edge : adj[u]) {
            int weight = edge.first;
            int v = edge.second;

            if (!inMST[v] && weight < key[v]) {
                key[v] = weight;
                parent[v] = u;
                pq.push({key[v], v});
            }
        }
    }

    // construct MST adjacency list
    for (int i = 1; i < n; ++i) {
        mstAdjList[parent[i]].push_back({key[i], i});
        mstAdjList[i].push_back({key[i], parent[i]});
    }
    // print MST adjacency list
    cout << "MST Adjacency List:\n";
    for (int i = 0; i < n; ++i) {
        cout << i + 1 << ": ";
        for (const auto& edge : mstAdjList[i]) {
            cout << "(" << edge.second + 1 << ", " << edge.first << ") ";
        }
        cout << endl;
    }
    // print MST edges
    cout << "Edges in MST:\n";
    for (int i = 1; i < n; ++i) {
        cout << parent[i] + 1 << " - " << i + 1 << " (Weight: " << key[i] << ")\n";
    }
}

int main() {
    int n, m;
    cout << "Enter the number of vertices and edges: ";
    cin >> n >> m;

    vector<vector<pii>> adj(n);
    cout << "Enter the edges (u v w):\n";
    for (int i = 0; i < m; ++i) {
        int u, v, w;
        cin >> u >> v >> w;
        adj[u - 1].push_back({w, v - 1});
        adj[v - 1].push_back({w, u - 1});
    }

    auto start = chrono::high_resolution_clock::now();
    primAdjList(n, adj);
    auto end = chrono::high_resolution_clock::now();

    chrono::duration<double> elapsed = end - start;
    cout << "Time taken: " << elapsed.count() << " seconds\n";
    return 0;
}
```

Use a priority queue to efficiently find the minimum edge and then update keys and parents based on the adjacency list. Finally print the edges & weights of the MST

```
Enter the number of vertices and edges: 7 9
Enter the edges (u v w):
1 2 28
2 3 16
3 4 12
4 5 22
5 6 25
6 1 10
5 7 24
4 7 18
7 2 14
MST Adjacency List:
1: (6, 10)
2: (3, 16) (7, 14)
3: (2, 16) (4, 12)
4: (3, 12) (5, 22)
5: (4, 22) (6, 25)
6: (5, 25) (1, 10)
7: (2, 14)
Edges in MST:
3 - 2 (Weight: 16)
4 - 3 (Weight: 12)
5 - 4 (Weight: 22)
6 - 5 (Weight: 25)
1 - 6 (Weight: 10)
2 - 7 (Weight: 14)
Time taken: 8.1331e-05 seconds


...Program finished with exit code 0
Press ENTER to exit console.
```

Running Time analysis.

Min-heap Operations : $O(\log V)$

For all edges : $O(E \log V)$

Total : $O((V+E) \log V)$

Suitable for sparse graphs.

② Johnsen's Algorithm

Johnsen's Algorithm is typically used for finding the shortest path in a graph with both positive and negative weights. It involves a combination of Bellman-Ford and Dijkstra's algorithm.

1. Bellman-Ford Algorithm computes a potential function h[v] for all vertices, ensuring there are no negative weight cycles. This is used to reweight the graph, making all edge weights non-negative.
2. Dijkstra's Algorithm uses the reweighted graph to compute shortest paths efficiently with a priority queue.
3. Johnsen's Algorithm combines the results to return shortest paths for the original graph

main.cpp

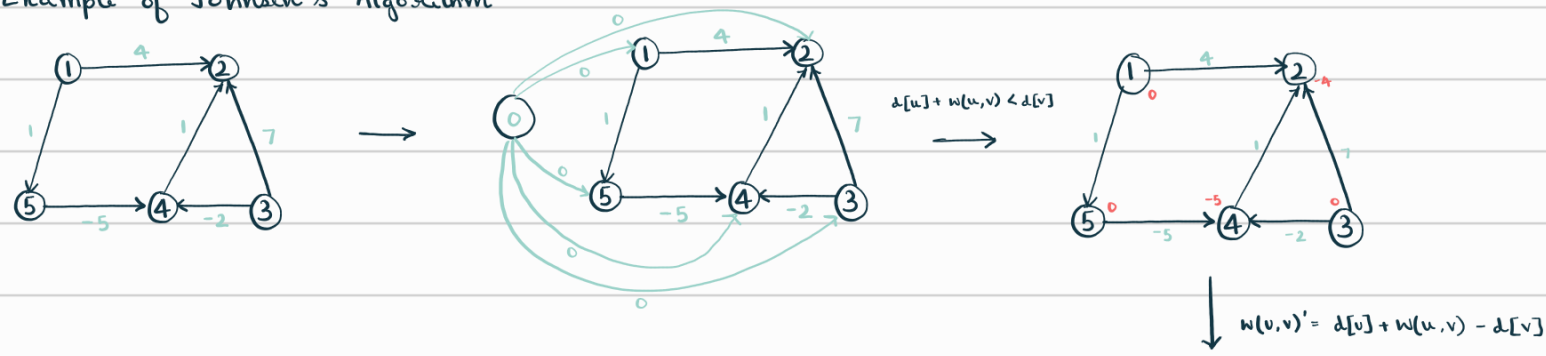
```cpp
#include <iostream>
#include <vector>
#include <limits>
#include <queue>
#include <iomanip>

using namespace std;

const int INF = numeric_limits<int>::max();

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

// Bellman-Ford algorithm
vector<int> bellmanFord(int V, int src, const vector<Edge>& edges) {
    vector<int> dist(V + 1, INF); // 1-based indexing
    dist[src] = 0;

    for (int i = 1; i <= V - 1; i++) {
        for (const auto& edge : edges) {
            if (dist[edge.src] != INF && dist[edge.src] + edge.weight < dist[edge.dest]) {
                dist[edge.dest] = dist[edge.src] + edge.weight;
            }
        }
    }

    for (const auto& edge : edges) {
        if (dist[edge.src] != INF && dist[edge.src] + edge.weight < dist[edge.dest]) {
            cout << "Negative weight cycle detected!" << endl;
            return {};
        }
    }

    return dist;
}
```

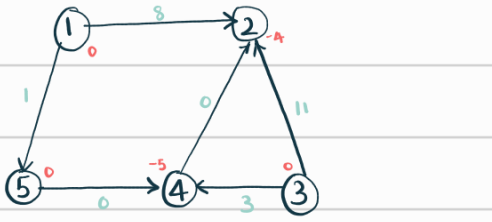
```cpp
// Dijkstra's algorithm using a priority queue (min-heap)
vector<int> dijkstra(int V, int src, const vector<vector<pair<int, int>>>& adj) {
    vector<int> dist(V + 1, INF); // 1-based indexing
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

    dist[src] = 0;
    pq.push({0, src});

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

        if (d > dist[u]) continue;

        for (const auto& [v, weight] : adj[u]) {
            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.push({dist[v], v});
            }
        }
    }

    return dist;
}
```

```cpp
// Johnson's algorithm
void johnsonsAlgorithm(int V, const vector<Edge>& edges) {
    // Step 1: Add a new vertex and connect it to every other vertex with weight 0
    vector<Edge> newEdges = edges;
    for (int i = 1; i <= V; i++) {
        newEdges.push_back({V + 1, i, 0});
    }

    vector<int> h = bellmanFord(V + 1, V + 1, newEdges);
    if (h.empty()) return;

    cout << "Reweighted graph (h values):" << endl;
    for (int i = 1; i <= V; i++) {
        cout << "h[" << i << "] = " << h[i] << endl;
    }

    // Step 2: Reweight edges based on h values and print them
    vector<vector<pair<int, int>>> adj(V + 1);
    cout << "\nReweighted edges:" << endl;
    for (const auto& edge : edges) {
        int newWeight = edge.weight + h[edge.src] - h[edge.dest];
        adj[edge.src].push_back({edge.dest, newWeight});
        cout << "Edge (" << edge.src << " -> " << edge.dest << ") : New Weight = " << newWeight << endl;
    }

    // Step 3: Run Dijkstra's algorithm for each vertex and print results
    cout << "\nShortest paths using Dijkstra's (Johnson's reweighted):\n";
    for (int i = 1; i <= V; i++) {
        vector<int> dist = dijkstra(V, i, adj);
        cout << "From vertex " << i << ":\n";
        for (int j = 1; j <= V; j++) {
            if (dist[j] == INF) {
                cout << "To vertex " << j << " = INF" << endl;
            } else {
                cout << "To vertex " << j << " = " << dist[j] << endl;
            }
        }
        cout << endl;
    }
}

int main() {
    int V, E;
    cout << "Enter number of vertices and edges: ";
    cin >> V >> E;

    vector<Edge> edges(E);
    cout << "Enter each edge (src dest weight):" << endl;
    for (int i = 0; i < E; i++) {
        cin >> edges[i].src >> edges[i].dest >> edges[i].weight;
    }

    int choice;
    cout << "\nOptions:\n1. Bellman-Ford\n2. Dijkstra's\n3. Johnson's Algorithm\nChoose: ";
    cin >> choice;

    if (choice == 1) {
        int src;
        cout << "Enter source vertex for Bellman-Ford: ";
        cin >> src;
        vector<int> dist = bellmanFord(V, src, edges);
        if (!dist.empty()) {
            cout << "Distances from source:\n";
            for (int i = 1; i <= V; i++) {
                if (dist[i] == INF) {
                    cout << "To vertex " << i << " = INF" << endl;
                } else {
                    cout << "To vertex " << i << " = " << dist[i] << endl;
                }
            }
        }
    } else if (choice == 2) {
        int src;
        cout << "Enter source vertex for Dijkstra's: ";
        cin >> src;

        vector<vector<pair<int, int>>> adj(V + 1); // 1-based indexing
        for (const auto& edge : edges) {
            adj[edge.src].push_back({edge.dest, edge.weight});
        }

        vector<int> dist = dijkstra(V, src, adj);
        cout << "Distances from source vertex " << src << ":\n";
        for (int i = 1; i <= V; i++) {
            if (dist[i] == INF) {
                cout << "To vertex " << i << " = INF" << endl;
            } else {
                cout << "To vertex " << i << " = " << dist[i] << endl;
            }
        }
    } else if (choice == 3) {
        johnsonsAlgorithm(V, edges);
    }

    return 0;
}
```

Johnson's algorithm reweights the graph using Bellman-Ford algorithm then runs Dijkstra's algorithm for each vertex and prints the shortest paths

Example of Johnson's Algorithm






```
Enter number of vertices and edges: 5 6
Enter each edge (src dest weight):
1 2 4
1 5 1
5 4 -5
3 4 -2
3 2 7
4 2 1

Options:
1. Bellman-Ford
2. Dijkstra's
3. Johnson's Algorithm
Choose: 3
Reweighted graph (h values):
h[1] = 0
h[2] = -4
h[3] = 0
h[4] = -5
h[5] = 0

Reweighted edges:
Edge (1 -> 2) : New Weight = 8
Edge (1 -> 5) : New Weight = 1
Edge (5 -> 4) : New Weight = 0
Edge (3 -> 4) : New Weight = 3
Edge (3 -> 2) : New Weight = 11
Edge (4 -> 2) : New Weight = 0
```

Distance from 0 to 1, 2, 3, 4, 5 are 0, -4, 0, -5, 0 respectively

```
Shortest paths using Dijkstra's (Johnson's reweighted):
From vertex 1:
To vertex 1 = 0
To vertex 2 = 1
To vertex 3 = INF
To vertex 4 = 1
To vertex 5 = 1

From vertex 2:
To vertex 1 = INF
To vertex 2 = 0
To vertex 3 = INF
To vertex 4 = INF
To vertex 5 = INF

From vertex 3:
To vertex 1 = INF
To vertex 2 = 3
To vertex 3 = 0
To vertex 4 = 3
To vertex 5 = INF

From vertex 4:
To vertex 1 = INF
To vertex 2 = 0
To vertex 3 = INF
To vertex 4 = 0
To vertex 5 = INF

From vertex 5:
To vertex 1 = INF
To vertex 2 = 0
To vertex 3 = INF
To vertex 4 = 0
To vertex 5 = 0
Johnson's Algorithm execution time: 130 microseconds
```

Running Time Analysis:

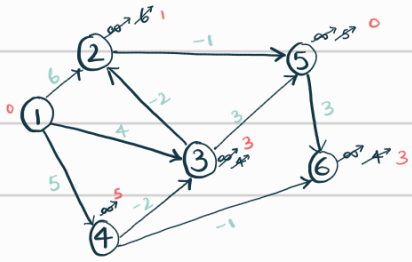
Bellman-Ford Algorithm takes $O(V \cdot E)$

Dijkstra's Algorithm runs for each vertex taking $O(V \cdot (V+E) \cdot \log V) \rightarrow O(V \log V)$

$\therefore$ Time Complexity of Johnson's Algorithm

$$O(V \cdot E + V \cdot (V+E) \log V) \rightarrow O(V^2 \log V + VE)$$

Example of Bellman-Ford Algorithm

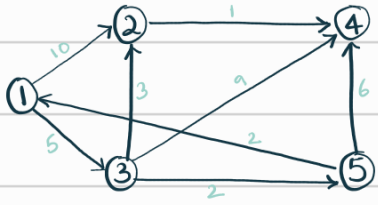



```
Enter number of vertices and edges: 6 9
Enter each edge (src dest weight):
1 2 6
1 3 4
1 4 5
3 2 -2
4 3 -2
2 5 -1
3 5 3
4 6 -1
5 6 3

Options:
1. Bellman-Ford
2. Dijkstra's
3. Johnson's Algorithm
Choose: 1
Enter source vertex for Bellman-Ford: 1
Distances from source vertex 1:
To vertex 1 = 0
To vertex 2 = 1
To vertex 3 = 3
To vertex 4 = 5
To vertex 5 = 0
To vertex 6 = 3
Bellman-Ford execution time: 2065474 microseconds
```

Bellman-Ford computes shortest distances from a single source and detects negative weight cycles.

Relaxes all edges V-1 times and each relaxation step takes O(E) ∴ Time Complexity : O(VE)

Example of Dijkstra's Algorithm



1	2	3	4	5
0	∞	∞	∞	∞
/	10	5	∞	∞
/	8	/	14	7
/	8	/	13	/
/	/	/	9	/

```
Enter number of vertices and edges: 5 7
Enter each edge (src dest weight):
1 2 10
1 3 5
2 4 1
3 2 3
3 5 2
5 4 6
3 4 9

Options:
1. Bellman-Ford
2. Dijkstra's
3. Johnson's Algorithm
Choose: 2
Enter source vertex for Dijkstra's: 1
Distances from source vertex 1:
To vertex 1 = 0
To vertex 2 = 8
To vertex 3 = 5
To vertex 4 = 9
To vertex 5 = 7
Dijkstra's execution time: 987897 microseconds
```

Dijkstra's algorithm finds shortest path from a source vertex using an unsorted priority queue. Each vertex is processed once & each edge is relaxed once using a min-heap.

∴ Time Complexity : $O((V+E)\log V)$