Green University of Bangladesh
Department of Computer Science and Engineering (CSE)
Faculty of Sciences and Engineering
Semester: (Spring, Year: 2023), B.Sc. in CSE (Day)
Lab Report: 03
Course Title: Algorithm Lab
Course Code: CSE 206 Section: DA

Student Details

Name	ID
Pankaj Mahanta	213902002

Lab Date : 08- 03 - 2023
Submission Date : 03 – 25 - 2023
Course Teacher's Name : Md. Sultanul Islam Ovi

Lab Report Status

Marks: …… **Signature: ……**
Comments: …… **Date: ……**

Task-1: Suppose you are given a graph where each edge represents the path cost and each vertex has also a cost which represents that, if you select a path using this node, the cost will be added with the path cost. Implement this graph using Dijkstra's algorithm.

Here are the step-by-step explanations of the code:

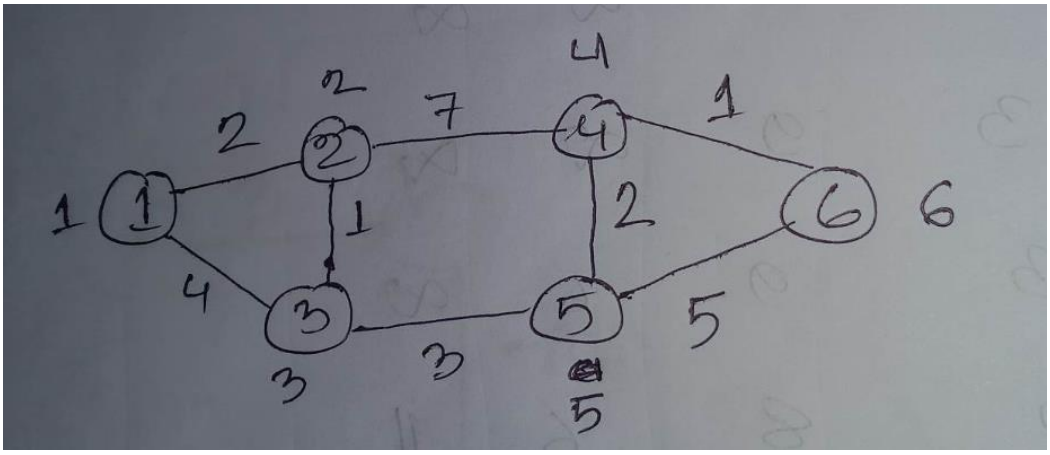
After reading the input, the program constructs an adjacency list of the graph. Then it initializes two vectors, 'dist' and 'visited', where dist stores the shortest distance from the source node to each node in the graph, and visited is used to keep track of the visited nodes. The dist vector is initialized to a large value (1e9) to represent infinity.

Then the Dijkstra's algorithm is applied, where the priority queue 'pq' stores the vertices in increasing order of their distance from the source. Initially, the distance of the source node is set to the cost of selecting that node, and the source node is pushed into the priority queue.

In each iteration, the program extracts the vertex with the minimum distance from the priority queue and marks it as visited. It then updates the distances of the neighboring nodes if a shorter path is found. The cost of selecting the node is also added to the path cost, and if this new distance is smaller than the previous distance of the node, the new distance is updated and the node is pushed into the priority queue.

Finally, the program outputs the shortest distance from each node to the source.

Graph:



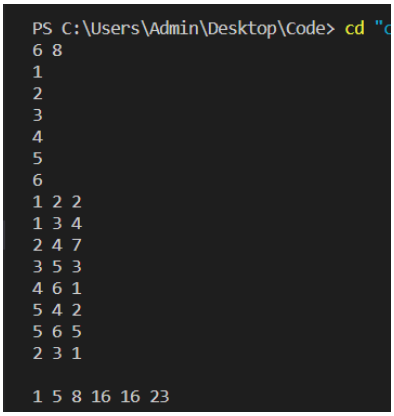
Code:

C_C++ > C++ > ShortestPath > Node_Cost.c++ > main()

```cpp
#include <bits/stdc++.h>
using namespace std;
const int INF = 1e9;
/*
@Pankaj Mahanto @ ID: 213902002 Section:PC-213-DA
*/
int main() {
    int n, m;
    cin >> n >> m;

    vector<vector<pair<int, int>>> adj(n); // adjacency list
    vector<int> node_cost(n); // cost of selecting each node

    for (int i = 0; i < n; i++) {
        cin >> node_cost[i];
    }
    for (int i = 0; i < m; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        u--; v--;
        adj[u].push_back({v, w});
        adj[v].push_back({u, w});
    }
    // Dijkstra's algorithm
    vector<int> dist(n, INF);
    vector<bool> visited(n, false);
    dist[0] = node_cost[0];
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    pq.push({dist[0], 0});

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

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

        for (auto e : adj[u]) {
            int v = e.first;
            int w = e.second;

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

```cpp
            if (dist[u] + w + node_cost[v] < dist[v]) {
                dist[v] = dist[u] + w + node_cost[v];
                pq.push({dist[v], v});
            }
        }
    }

    // output the shortest distance from node 1 to node n
    cout<<endl;
    for (int i = 0; i <n; i++){
        cout << dist[i] << " ";
    }
    cout << endl;
    cout << dist[n-1] << endl;

    return 0;
}
```

Output:



Task-2: Implement Dijkstra's algorithm for a single destination shortest path problem.

Here are the step-by-step explanations of the code:

The main function starts by reading the number of nodes (n) and edges (m) of the graph from the input.

It declares a vector of pair of integers, g, of size (n+1) where each pair represents an edge of the graph from the first element to the second element with weight given by the third element.

The next step is to read the edges of the graph from the input and populate the g vector with the given edges. Since the graph is directed, we insert two edges, one from u to v for each input edge (u, v, w) and change the direction using g[v].push_back({u, w}).

After reading the edges, we read the starting node s from the input and initialize the distance vector dist such that dist[s] is 0 and all other elements are INF.

We use a set of pairs, sh, to store the nodes in increasing order of their distances from the starting node s. The pair contains the distance and the node number.

We insert the pair (0, s) in the set sh and enter a while loop that continues until sh is not empty.

In each iteration of the while loop, we extract the node x with the smallest distance from the set sh and relax all its neighbors. We use the set sh to get the node with the smallest distance because it provides efficient deletion and insertion operations.

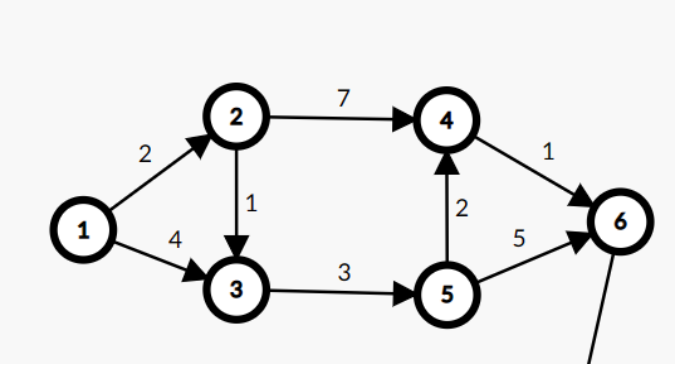
In relaxing the neighbors of node x, we compare the current distance of the neighbor i with the distance of x plus the weight of the edge (x, i). If the current distance is greater, we update the distance of i and insert the pair (dist[i], i) in the set sh.

After the while loop completes, the dist vector contains the SDSP or the single destination shortest path from the starting node s to all other nodes in the graph.

The code outputs the dist vector as the final result.

Note that the code also has comments that provide additional explanation and insights.

Graph:



Code:

C_C++ > C++ > ShortestPath > Single_Destination.c++ > main()

```cpp
// dijkstra
#include <bits/stdc++.h>
/*
@Pankaj Mahanto @ ID: 213902002 Section:PC-213-DA
*/
using namespace std;
const int INF = 1e9;
int main()
{
    int n, m;
    cin >> n >> m;
    vector<int> dist(n + 1, INF);
    vector<vector<pair<int, int>>> g(n + 1); // index 0 theke suru nah korle eta korte hoy

    for (int i = 0; i < m; i++)
    {
       int u, v, w;
        cin >> u >> v >> w;
        // undirected graph
       //  g[u].push_back({v, w});
         g[v].push_back({u, w});// single destination shortest path??
         // je graph use korbo tar direction change kore dibo tahole seta single destination shortest path
         // hoye jabe
    }
    // start node
    int s;
    cin >> s;
    dist[s] = 0;
    set<pair<int, int>> sh;
    sh.insert({0, s}); // (dist[s],s)
    while (!sh.empty())
    {
        auto x = *(sh.begin());
        sh.erase(sh.begin());
        for (auto i : g[x.second])
        {

            if (dist[i.first] > dist[x.second] + i.second)
            {
                sh.erase({dist[i.first], i.first});
                dist[i.first] = dist[x.second] + i.second;
                sh.insert({dist[i.first], i.first});
            }
        }
    }
    // single pair shortest path
//  cout<<dist[4];
    //single destination shortest path
   cout << " end here\n";
   for (int i = 1; i <= n; i++)
   {
        cout << dist[i] << "  ";
   }

    return 0;
```

Output:

```
PS C:\Users\Admin\Desktop\Code\C_C++\C+
if ($?) { .\Single_Destination }
6 8
1 2 2
1 3 4
2 4 7
3 5 3
4 6 1
5 4 2
5 6 5
2 3 1
6

9  7  6  1  3  0
PS C:\Users\Admin\Desktop\Code\C_C++\C+
```

Task-03: Implement Dijkstra's algorithm for a single pair shortest path problem.

Here are the step-by-step explanations of the code:

The main function starts by reading the number of nodes (n) and edges (m) of the graph from the input.

It declares a vector of pair of integers, g, of size (n+1) where each pair represents an edge of the graph from the first element to the second element with weight given by the third element.

The next step is to read the edges of the graph from the input and populate the g vector with the given edges. Since the graph is undirected, we insert two edges, one from u to v and another from v to u, for each input edge (u, v, w).

After reading the edges, we read the starting node s from the input and initialize the distance vector dist such that dist[s] is 0 and all other elements are INF.

We use a set of pairs, sh, to store the nodes in increasing order of their distances from the starting node s. The pair contains the distance and the node number.

We insert the pair (0, s) in the set sh and enter a while loop that continues until sh is not empty.

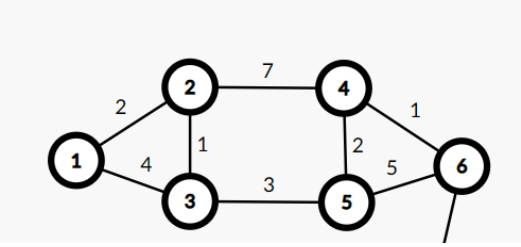
In each iteration of the while loop, we extract the node x with the smallest distance from the set sh and relax all its neighbors. We use the set sh to get the node with the smallest distance because it provides efficient deletion and insertion operations.

In relaxing the neighbors of node x, we compare the current distance of the neighbor i with the distance of x plus the weight of the edge (x, i). If the current distance is greater, we update the distance of i and insert the pair (dist[i], i) in the set sh.

After the while loop completes, the dist vector contains the SSSP or the single destination shortest path from the starting node s to all other nodes in the graph.

The code outputs the dist vector as the final result.

Graph:

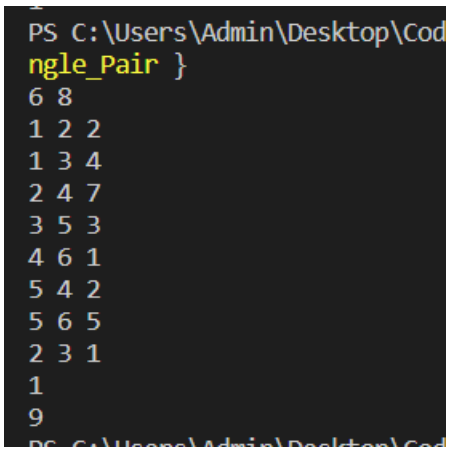


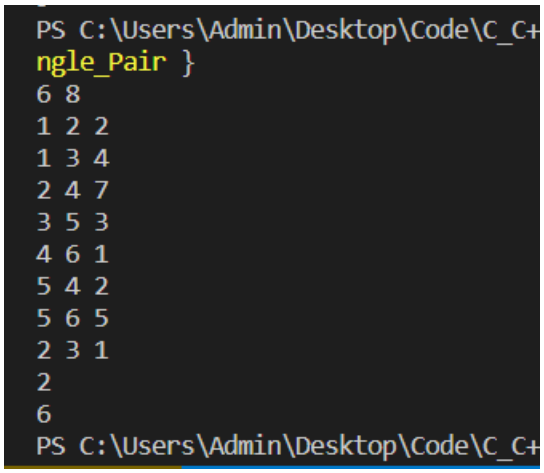
Code:

C_C++ > C++ > ShortestPath > Single_Pair.c++ > ...

```cpp
#include <bits/stdc++.h>
/*
@Pankaj Mahanto @ ID: 213902002 Section:PC-213-DA
*/
using namespace std;
const int INF = 1e9;
int main()
{
    int n, m;
    cin >> n >> m;
    vector<int> dist(n + 1, INF);
    vector<vector<pair<int, int>>> g(n + 1); // index 0 theke suru nah korle eta korte hoy
    for (int i = 0; i < m; i++)
    {
        int u, v, w;
        cin >> u >> v >> w;
        // undirected graph
        g[u].push_back({v, w});
        g[v].push_back({u, w});// single pair shortest path same single destination path ar moto
        // just poriboton ta holo output print korar somoy starting theke kon node porjon to jabo just
        // sei distances ta asbe
    }
    // start node
    int s;
    cin >> s;
    dist[s] = 0;
    set<pair<int, int>> sh;
    sh.insert({0, s}); // (dist[s],s)
    while (!sh.empty())
    {
        auto x = *(sh.begin());
        sh.erase(sh.begin());
        for (auto i : g[x.second])
        {
            if (dist[i.first] > dist[x.second] + i.second)
            {
                sh.erase({dist[i.first], i.first});
                dist[i.first] = dist[x.second] + i.second;
                sh.insert({dist[i.first], i.first});
            }
        }
    }
    // single pair shortest path
   cout<<dist[4];
    return 0;
}
```

Output:





Task-4: Implement Dijkstra's algorithm for an unweighted directed graph.

Here are the step-by-step explanations of the code:

This code implements Dijkstra's algorithm for finding the shortest path in a unweighted graph. The algorithm finds the shortest path from a given source node to all other nodes in the graph.

The code first takes input for the number of nodes (n) and the number of edges (m) in the graph. It then creates a vector of size n+1 to store the minimum distances from the source node to all other nodes in the graph. The initial values are set to INF (infinity) for all nodes except the source node, which is set to 0.

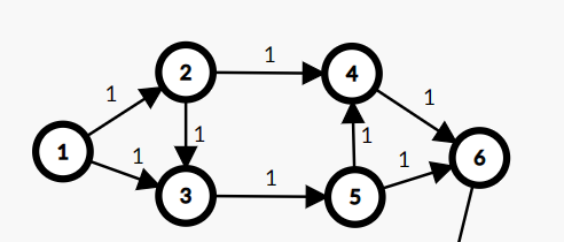
Next, the code creates a vector of vectors to represent the adjacency list of the graph. Each element of the outer vector represents a node in the graph, and each inner vector represents the edges that start from that node. Each edge is represented by a pair of (destination node, weight),Here weight is 1 because unweighted graph means every edges weight is 1. The code takes input for each edge and adds it to the adjacency list. Note that the graph is assumed to be directed, so the code adds edges in one direction.

The code then takes input for the source node and creates a set to store the nodes that are not yet processed. The set is initialized with a pair of (0, source node), representing the distance from the source node to itself.

The main loop of the algorithm runs while the set is not empty. In each iteration, the code removes the node with the smallest distance from the set, updates the distances of its neighbors if necessary, and adds them back to the set.

Finally, the code prints the minimum distances from the source node to all other nodes in the graph.

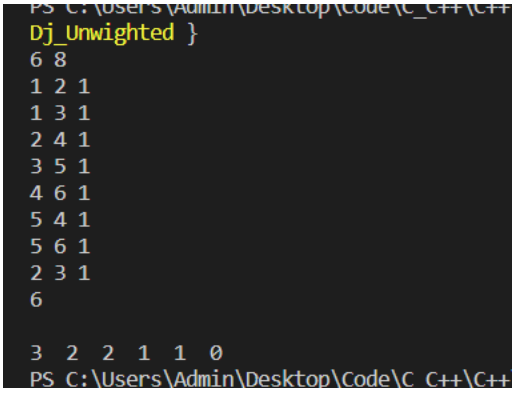
Graph:



Code:

```cpp
// dijkstra
#include <bits/stdc++.h>
/*
@Pankaj Mahanto @ ID: 213902002 Section:PC-213-DA
*/
using namespace std;
const int INF = 1e9;
int main()
{
    int n, m;
    cin >> n >> m;
    vector<int> dist(n + 1, INF);
    vector<vector<pair<int, int>>> g(n + 1); // index 0 theke suru nah korle eta korte hoy
    for (int i = 0; i < m; i++)
    {
        int u, v, w;
        cin >> u >> v >> w;
        // undirected graph
       // g[u].push_back({v, w});
        g[v].push_back({u, w}); // SSSP unweighted graph ar jonno eta use kora hoy
                                // Are sobar weight 1 dhora hoyeche unweighted graph bolar karone
                                // hoye jabe
    }
    // start node
    int s;
    cin >> s;
    dist[s] = 0;
    set<pair<int, int>> sh;
    sh.insert({0, s}); // (dist[s],s)
    while (!sh.empty())
    {
        auto x = *(sh.begin());
        sh.erase(sh.begin());
        for (auto i : g[x.second])
        {
            if (dist[i.first] > dist[x.second] + i.second)
            {
                sh.erase({dist[i.first], i.first});
                dist[i.first] = dist[x.second] + i.second;
                sh.insert({dist[i.first], i.first});
            }
        }
    }
    // single pair shortest path
   // cout << dist[4];
    // single source shortest path with unwighted graph
   // cout << " end here\n";
   cout<<endl;
    for (int i = 1; i <= n; i++)
    {
        cout << dist[i] << "  ";
    }
    return 0;
}
```

Output:



Task-5: Implement Johnson's algorithm for All-pairs shortest paths. And explain how it is better than the Floyd Warshall Algorithm.

The algorithm proceeds in two steps:

It adds a new vertex with zero-weight edges to all other vertices in the graph, such that the resulting graph becomes non-negative.

It applies Dijkstra's algorithm to each vertex in the graph to obtain the shortest paths between all pairs of vertices.

Now let's go through the code step by step:

The code defines a constant INF to represent infinity, a vector of vectors adj to store the adjacency list of the graph, and a vector h to store the heuristic values for each vertex.

The function dijkstra is defined to implement Dijkstra's algorithm for a single source vertex s. It uses a priority queue to store vertices in non-decreasing order of distance from s, and initializes a vector d to store the shortest distances from s to all other vertices. The algorithm processes each vertex in the priority queue, updating its distance if a shorter path is found through one of its neighbors.

The function johnson implements the Johnson's algorithm by adding a new vertex n to the graph, connecting it to all other vertices with zero-weight edges, and computing the heuristic values h for each vertex using Dijkstra's algorithm starting from n. Then, it modifies the weights of the edges in the graph using the heuristic

values, and applies Dijkstra's algorithm to each vertex to obtain the shortest paths between all pairs of vertices.

In the main function, the code reads the number of vertices n and the number of edges m from the input, initializes the adjacency list adj, and reads the edge information from the input.

It calls the Johnson function to compute the shortest paths between all pairs of vertices, and prints the resulting distances to the standard output.

Finally, the main function returns 0, indicating successful execution of the program.

Johnson's algorithm is a more efficient way of computing all-pairs shortest paths than the Floyd Warshall Algorithm for sparse graphs. Here are some reasons why Johnson's algorithm can be better than the Floyd Warshall Algorithm:

Running time: The Floyd Warshall Algorithm has a time complexity of O(N^3) where N is the number of nodes in the graph. On the other hand, Johnson's algorithm takes O(NM log M + N^2 log N) time where M is the number of edges in the graph. In sparse graphs, where the number of edges is significantly less than N^2, Johnson's algorithm can be much faster than Floyd Warshall Algorithm.

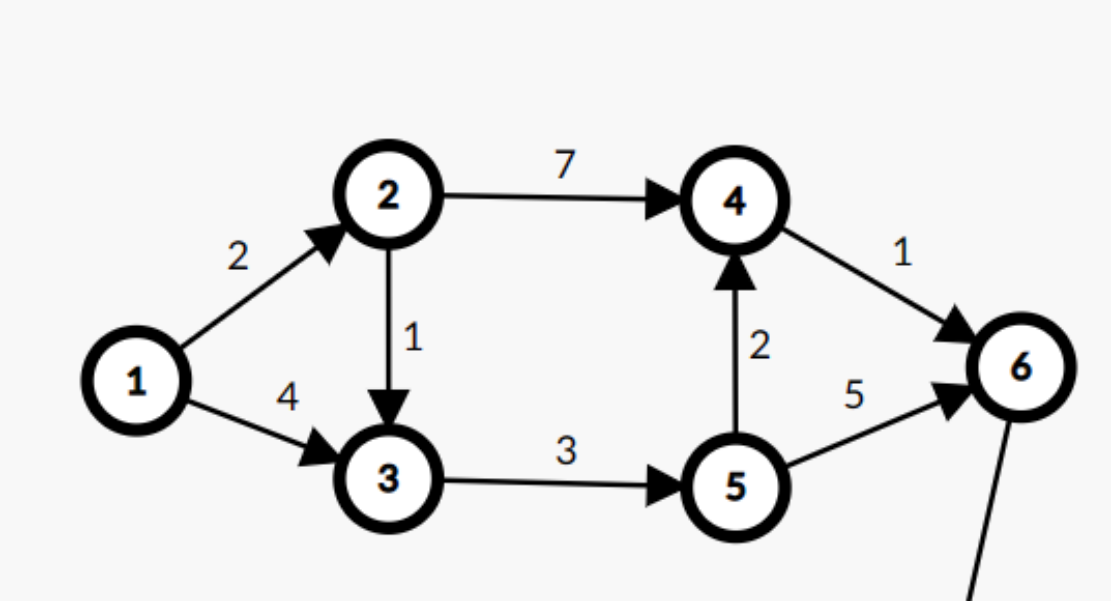
Memory requirement: The Floyd Warshall Algorithm requires a 2D matrix of size N x N to store the shortest paths between all pairs of nodes. In contrast, Johnson's algorithm requires a much smaller 1D array of size N to store the distances from the source node to all other nodes.

Handling negative edges: Floyd Warshall Algorithm does not work with graphs that have negative edge weights, while Johnson's algorithm handles negative edge weights by first modifying the edge weights using Bellman-Ford Algorithm.

Scalability: The Floyd Warshall Algorithm has a high computational complexity and can be slow to compute shortest paths for large graphs. Johnson's algorithm is more scalable than the Floyd Warshall Algorithm for large graphs.

Overall, Johnson's algorithm can be a better choice for computing all-pairs shortest paths than the Floyd Warshall Algorithm for sparse graphs, especially when negative edge weights are present. However, for dense graphs, Floyd Warshall Algorithm may be a more efficient choice.

Graph:



Code:

C_C++ > C++ > ShortestPath > Johnson.c++ > h

```cpp
#include <bits/stdc++.h>
using namespace std;
/*
@Pankaj Mahanto @ ID: 213902002 Section:PC-213-DA
*/
const int INF = 1e9;
vector<vector<pair<int, int>>> adj;
vector<int> h;
void dijkstra(int s, vector<int>& d) {
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    d.assign(adj.size(), INF);
    d[s] = 0;
    pq.emplace(0, s);
    while (!pq.empty()) {
        int u = pq.top().second;
        int dist = pq.top().first;
        pq.pop();
        if (dist > d[u]) continue;
        for (auto [v, w] : adj[u]) {
            if (d[v] > d[u] + w) {
                d[v] = d[u] + w;
                pq.emplace(d[v], v);
            }
        }
    }
}
vector<vector<int>> johnson() {
    int n = adj.size();
    vector<vector<int>> dist(n, vector<int>(n, INF));
    vector<pair<int, int>> edges;
    for (int u = 0; u < n; ++u) {
        for (auto [v, w] : adj[u]) {
            edges.emplace_back(u, v);
        }
    }
    edges.emplace_back(n, 0);
    h.assign(n + 1, INF);
    h[n] = 0;
    dijkstra(n, h);
    for (auto& [u, v] : edges) {
        u = u == n ? n - 1 : u;
        v = v == n ? n - 1 : v;
```

```
        v = v == n ? n - 1 : v;
    }
    for (int u = 0; u < n; ++u) {
        vector<int> d;
        dijkstra(u, d);
        for (int v = 0; v < n; ++v) {
            if (d[v] != INF) {
                dist[u][v] = d[v] - h[u] + h[v];
            }
        }
    }
    return dist;
}
int main() {
    int n, m;
    cin >> n >> m;
    adj.resize(n);
    for (int i = 0; i < m; ++i) {
        int u, v, w;
        cin >> u >> v >> w;
        adj[u].emplace_back(v, w);
    }
    auto dist = johnson();
    for (int u = 0; u < n; ++u) {
        for (int v = 0; v < n; ++v) {
            if (dist[u][v] == INF) {
                cout << "INF ";
            } else {
                cout << dist[u][v] << " ";
            }
        }
        cout << endl;
    }
    return 0;
}
/*
```

Output:

```
6 8
1 2 2
1 3 4
2 4 7
3 5 3
4 6 1
5 4 2
5 6 5
2 3 1
0 INF INF INF INF INF
INF 0 2 3 8 6
INF INF 0 1 6 4
INF INF INF 0 5 3
INF INF INF INF 0 INF
INF INF INF INF 2 0
PS C:\Users\Admin\Desktop\Code>
```