L23 : Graphs 2

1-Tut : Kruskal's Algorithm

Send Feedback

Given an undirected, connected and weighted graph G(V, E) with V number of vertices (which are numbered from 0 to V-1) and E number of edges.

Find and print the total weight of Minimum Spanning Tree (MST) using Kruskal's algorithm.

Input Format :

First line will contain T(number of test case), each test case follows as.
Line 1: Two Integers V and E (separated by space)
Next E lines : Three integers ei, ej and wi, denoting that there exists an edge between vertex ei and vertex ej with weight wi (separated by space)

Output Format :

Weight of MST for each test case in new line.

Constraints :

1 <= T <= 10
2 <= V, E <= 10^5
1 <= wt <= 10^4

Sample Input 1 :

```
1
4 4
0 1 3
0 3 5
1 2 1
2 3 8
```

Sample Output 1 :

```
9
```

```cpp
#include <iostream>
#include <algorithm>
using namespace std;
// Class that store values for each vertex
class Edge
{
public:
    int source;
    int dest;
    int weight;
};
// Comparator function used to sort edges
bool compare(Edge e1, Edge e2)
{
```

```
    // Edges will sorted in order of their weights
    return e1.weight < e2.weight;
}
// Function to find the parent of a vertex
int findParent(int v, int *parent)
{
    // Base case, when a vertex is parent of itself
    if (parent[v] == v)
    {
        return v;
    }
    // Recursively called to find the topmost parent of the vertex.
    return findParent(parent[v], parent);
}
void kruskals(Edge *input, int n, int E)
{
    // In-built sort function: Sorts the edges in
    // increasing order of their weights
    sort(input, input + E, compare);
    // Array to store final edges of MST
    Edge *output = new Edge[n - 1];
    // Parent array initialized with their indexes
    int *parent = new int[n];
    for (int i = 0; i < n; i++)
    {
        parent[i] = i;
    }
    int count = 0; // To maintain the count of number of edges in    the MST
    int i = 0;     // Index to traverse over the input array
    while (count != n - 1)
    { // As the MST contains n-1 edges.
        Edge currentEdge = input[i];
        // Figuring out the parent of each edge's vertices
        int sourceParent = findParent(currentEdge.source, parent);
        int destParent = findParent(currentEdge.dest, parent);
        // If their parents are not equal, then we added that edge to        output
        if (sourceParent != destParent)
        {
            output[count] = currentEdge;
            count++;                    // Increased the count
            parent[sourceParent] = destParent; // Updated the parent array
        }
        i++;
    }
```

```
    // Finally, printing the MST obtained.
    long long int sum = 0;
    for (int i = 0; i < n - 1; i++)
    {
        if (output[i].source < output[i].dest)
        {
            sum += (output[i].weight);
        }
        else
        {
            sum += (output[i].weight);
        }
    }

    cout << sum << endl;
}
int main()
{

    int t;
    cin >> t;
    while (t--)
    {

        int n, E;
        cin >> n;
        cin >> E;
        Edge *input = new Edge[E];
        for (int i = 0; i < E; i++)
        {
            int s, d, w;
            cin >> s >> d >> w;
            input[i].source = s;
            input[i].dest = d;
            input[i].weight = w;
        }
        kruskals(input, n, E);
    }
    return 0;
}
```

2-Tut : Prim's Algorithm

Send Feedback

Given an undirected, connected and weighted graph G(V, E) with V number of vertices (which are numbered from 0 to V-1) and E number of edges.

Find and print the total weight of Minimum Spanning Tree (MST) using Prim's algorithm.

Input Format :

First line will contain T(number of test case), each test case follows as.
Line 1: Two Integers V and E (separated by space)
Next E lines : Three integers ei, ej and wi, denoting that there exists an edge between vertex ei and vertex ej with weight wi (separated by space)

Output Format :

Weight of MST for each test case in new line.

Constraints :

1 <= T <= 10
2 <= V, E <= 10^5
1 <= wt <= 10^4

Sample Input 1 :

```
1
4 4
0 1 3
0 3 5
1 2 1
2 3 8
```

Sample Output 1 :

```
9
```

```cpp
#include <bits/stdc++.h>
using namespace std;

int findMinVertex(int *weights, bool *visited, int n)
{
    // Initialized to -1 means there is no vertex till now
    int minVertex = -1;
    for (int i = 0; i < n; i++)
    {
        // Conditions: the vertex must be unvisited and either minVertex value is -1
        //  or if minVertex has some vertex to it, then weight of currentvertex
        //  should be less than the weight of the minVertex.
        if (!visited[i] && (minVertex == -1 || weights[i] < weights[minVertex]))
        {
            minVertex = i;
        }
    }
```

```
    return minVertex;
}
void prims(int **edges, int n)
{
    int *parent = new int[n];
    int *weights = new int[n];
    bool *visited = new bool[n];
    // Initially, the visited array is assigned to false and weights array
    // to infinity.
    for (int i = 0; i < n; i++)
    {
        visited[i] = false;
        weights[i] = INT_MAX;
    }
    // Values assigned to vertex 0.(the selected starting vertex to begin with)
    parent[0] = -1;
    weights[0] = 0;
    for (int i = 0; i < n - 1; i++)
    {
        // Find min vertex
        int minVertex = findMinVertex(weights, visited, n);
        visited[minVertex] = true;
        // Explore unvisited neighbors
        for (int j = 0; j < n; j++)
        {
            if (edges[minVertex][j] != 0 && !visited[j])
            {
                if (edges[minVertex][j] < weights[j])
                {
                    // updating weight array and parent array
                    weights[j] = edges[minVertex][j];
                    parent[j] = minVertex;
                }
            }
        }
    }
    // Final MST printed
    long long int sum = 0;
    for (int i = 0; i < n; i++)
    {
        sum += (weights[i]);
    }

    cout << sum << endl;
```

```
}
int main()
{
    int t;
    cin >> t;
    while (t--)
    {
        int n;
        int e;
        cin >> n >> e;
        int **edges = new int *[n]; // Adjacency matrix used to store the graph
        for (int i = 0; i < n; i++)
        {
            edges[i] = new int[n];
            for (int j = 0; j < n; j++)
            {
                // Initially all pairs are assigned 0 weight which
                // means that there is no edge between them
                edges[i][j] = INT_MAX;
            }
        }
        for (int i = 0; i < e; i++)
        {
            int f, s, weight;
            cin >> f >> s >> weight;

            if (edges[f][s] > weight)
            {
                edges[f][s] = weight;
            }
            if (edges[s][f] > weight)
            {
                edges[s][f] = weight;
            }
        }
        prims(edges, n);
        for (int i = 0; i < n; i++)
        {
                delete[] edges[i];
            }
            delete[] edges;
        }

        return 0; }
```

3-Tut : Dijkstra's Algorithm

Send Feedback

Given an undirected, connected and weighted graph G(V, E) with V number of vertices (which are numbered from 0 to V-1) and E number of edges.

Find and print the shortest distance from the source vertex (i.e. Vertex 0) to all other vertices (including source vertex also) using Dijkstra's Algorithm.

Print the ith vertex number and the distance from source in one line separated by space. Print different vertices in different lines.

Note : Order of vertices in output doesn't matter.

Input Format :

First line will contain T(number of test case), each test case follows as.
Line 1: Two Integers V and E (separated by space)
Next E lines : Three integers ei, ej and wi, denoting that there exists an edge between vertex ei and vertex ej with weight wi (separated by space)

Output Format :

In different lines, ith vertex number and its distance from source (separated by space)

Constraints :

1 <= T <= 10
2 <= V, E <= 10^3

Sample Input 1 :

```
1
4 4
0 1 3
0 3 5
1 2 1
2 3 8
```

Sample Output 1 :

```
0 0
1 3
2 4
3 5
```

```
#include <iostream>
#include <climits>
using namespace std;
int findMinVertex(int *distance, bool *visited, int n)
{
    int minVertex = -1;
    for (int i = 0; i < n; i++)
    {
        if (!visited[i] && (minVertex == -1 || distance[i] < distance[minVertex]))
        {
            minVertex = i;
```

```
        }
    }
    return minVertex;
}
void dijkstra(int **edges, int n)
{
    int *distance = new int[n];
    bool *visited = new bool[n];
    for (int i = 0; i < n; i++)
    {
        visited[i] = false;
        distance[i] = INT_MAX;
    }
    distance[0] = 0; // 0 is considered as the starting node.
    for (int i = 0; i < n - 1; i++)
    {
        // Find min vertex
        int minVertex = findMinVertex(distance, visited, n);
        visited[minVertex] = true;
        // Explore unvisited neighbors
        for (int j = 0; j < n; j++)
        {
            if (edges[minVertex][j] != 0 && !visited[j])
            {
                // distance of any node will be the current node's distance + the weight
                // of the edge between them
                int dist = distance[minVertex] + edges[minVertex][j];
                if (dist < distance[j])
                { // If required, then updated.
                    distance[j] = dist;
                }
            }
        }
    }
    // Final output of distance of each node with respect to 0
    for (int i = 0; i < n; i++)
    {
        cout << i << " " << distance[i] << endl;
    }
}
int main()
{
    int t;
    cin >> t;
```

```
    while (t--)
    {
        int n;
        int e;
        cin >> n >> e;
        int **edges = new int *[n];
        for (int i = 0; i < n; i++)
        {
            edges[i] = new int[n];
            for (int j = 0; j < n; j++)
            {
                edges[i][j] = 0;
            }
        }
        for (int i = 0; i < e; i++)
        {
            int f, s, weight;
            cin >> f >> s >> weight;
            edges[f][s] = weight;
            edges[s][f] = weight;
        }
        dijkstra(edges, n);
        for (int i = 0; i < n; i++)
        {
            delete[] edges[i];
        }
        delete[] edges;
    }
    return 0;
}
```

4-Tut : Bellman-Ford Algorithm

Send Feedback

you are given a weighted directed graph G with n vertices and m edges, and two specified vertex src and des. You want to find the length of shortest paths from vertex src to des. The graph may contain the edges with negative weight.

Input Format:

First line of input will contain T(number of test case), each test case follows as.
Line1: contain two space-separated integers N and M denoting the number of vertex and number of edges in graph respectively.
Line2: contain two space-separated integers src, des.
Next M line will contain three space-separated integers a, b, wt representing the edge from a to b with weight wt.

Output Format:

For each test case print the distance of des from src in new line.
Note: In case of no path is found print (10 ^ 9) in that case.

Constraints:

1 <= T <= 100
1 <= N <= 200
1 <= M <= min(800, N*(N-1))
1 <= a,b <= N
-10^5 <= wt <= 10^5

Sample Input:

1
3 6
3 1
3 1 -2
1 3 244
2 3 -2
2 1 201
3 2 220
1 2 223

Sample output:

-2

```
#include <bits/stdc++.h>
using namespace std;
struct Edge
{
    int src, dest, weight;
};
struct Graph
{
    int V, E;
    struct Edge *edge;
};
struct Graph *createGraph(int V, int E)
{
    struct Graph *graph = new Graph;
    graph->V = V;
    graph->E = E;
    graph->edge = new Edge[E];
    return graph;
}
void printAns(int dist[], int n, int src, int dest)
{
    for (int i = 1; i <= n; ++i)
        if (i == dest)
```

```
            if (dist[i] == INT_MAX)
            {
                cout << "1000000000" << endl;
            }
            else
            {
                cout << dist[i] << endl;
            }
}
void BellmanFord(struct Graph *graph, int src, int dest)
{
    int V = graph->V;
    int E = graph->E;
    int dist[V + 1];
    for (int i = 0; i <= V; i++)
        dist[i] = INT_MAX;
    dist[src] = 0;
    for (int i = 1; i <= V - 1; i++)
    {
        for (int j = 0; j < E; j++)
        {
            int u = graph->edge[j].src;
            int v = graph->edge[j].dest;
            int weight = graph->edge[j].weight;
            if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
                dist[v] = dist[u] + weight;
        }
    }
    for (int i = 0; i < E; i++)
    {
        int u = graph->edge[i].src;
        int v = graph->edge[i].dest;
        int weight = graph->edge[i].weight;
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
        {
            cout << "1000000000" << endl;
            return; // If negative cycle is detected, simply return
        }
    }
    printAns(dist, V, src, dest);
    return;
}
int main()
{
```

```
    int t;
    cin >> t;
    while (t--)
    {
        int V, E;
        int src, dest;
        cin >> V;
        cin >> E;
        cin >> src;
        cin >> dest;
        struct Graph *graph = createGraph(V, E);
        for (int i = 0; i < E; i++)
        {
            int u, v, w;
            cin >> u >> v >> w;
            graph->edge[i].src = u;
            graph->edge[i].dest = v;
            graph->edge[i].weight = w;
        }
        BellmanFord(graph, src, dest);
    }

    return 0;
}
```

5-Tut : Floyd-Warshall Algorithm

Send Feedback

You are given an undirected weighted graph G with n vertices. And Q queries, each query consists of two integers a and b and you have print the distance of shortest path between a and b.

Note: If there is no path between a and b print 10^9

Input Format:

First line of Input will contain T(number of test cases), each test case follows as.
Line1: contains two space-separated integers N and M denoting the number of vertex and edge in graph.
Next M line contain three space-separated integers a, b, c denoting the edge between a and b with weight c.
Next line will contain Q (number of queries)
Next Q lines will contain two space-separated integers a and b.

Output Format:

For each query of each test case print the answer in a newline.

Constraints:

1 <= T <= 50
1 <= N <= 100
1 <= M <= 10^4
1 <= Q <= 10^4

1 <= wt <= 10^5 (for each edge)
Note: Graph may contain multiple edges.

Sample Input:

1
3 6
3 1 4
1 3 17
2 3 2
1 3 7
3 2 11
2 3 15
3
1 1
2 2
2 3

Sample output:

0
0

2

```
#include <bits/stdc++.h>
using namespace std;
int INF = 1e9;
int main()
{

    int t;
    cin >> t;
    while (t--)
    {
        int n, m;

        cin >> n;

        cin >> m;
        int mat[n + 1][n + 1];
        for (int i = 0; i <= n; i++)
        {
            for (int j = 0; j <= n; j++)
            {
                mat[i][j] = INF;
                if (i == j)
                {
                    mat[i][j] = 0;
```

```
            }
        }
    }
    for (int i = 0; i < m; i++)
    {
        int a, b, c;
        cin >> a >> b >> c;
        if (mat[a][b] < c)
        {
            continue;
        }
        else
        {
            mat[a][b] = c;
            mat[b][a] = c;
        }
    }
    int i, j, k;
    for (k = 1; k <= n; k++)
    {
        for (i = 1; i <= n; i++)
        {
            for (j = 1; j <= n; j++)
            {
                if (mat[i][j] > (mat[i][k] + mat[k][j]) && (mat[k][j] != INF && mat[i][k] != INF))
                    mat[i][j] = mat[i][k] + mat[k][j];
            }
        }
    }
    int q;
    cin >> q;
    while (q--)
    {
        int aa;
        int bb;
        cin >> aa >> bb;

        if (aa == bb)
        {
            cout << 0 << endl;
        }
        else
        {
            cout << mat[aa][bb] << endl;
```

```
            }
        }
    }
}
```

6-Bonus : Disjoint Set Union

```
#include <bits/stdc++.h>
using namespace std;

class DSNode
{
public:
    int data;
    DSNode *parent;
    int rank;
};

class DisjointSet
{
private:
    map<int, DSNode *> hash;

    DSNode *searchInSetHelper(DSNode *node)
    {

        if (node == node->parent)
        {
            return node;
        }
        node->parent = searchInSetHelper(node->parent);
        return node->parent;
    }

public:
    void initializeSet(int data)
    {
        DSNode *node = new DSNode();
        node->data = data;
        node->parent = node;
        node->rank = 0;
        hash[data] = node;
    }
```

```

    void Union(int data1, int data2)
    {

        DSNode *node1 = hash[data1];
        DSNode *node2 = hash[data2];

        DSNode *parent1 = searchInSetHelper(node1);
        DSNode *parent2 = searchInSetHelper(node2);

        if (parent1->data == parent2->data)
        {
            return;
        }

        if (parent1->rank >= parent2->rank)
        {
            if (parent1->rank == parent2->rank)
            {
                parent1->rank = parent1->rank + 1;
            }
            parent2->parent = parent1;
        }
        else
        {
            parent1->parent = parent2;
        }
    }

    int searchInSet(int data)
    {
        return searchInSetHelper(hash[data])->data;
    }
};

int main()
{

    DisjointSet ds;

    ds.initializeSet(0);
    ds.initializeSet(1);
    ds.initializeSet(2);
    ds.initializeSet(3);
```

```
    ds.initializeSet(4);
    ds.initializeSet(5);
    ds.initializeSet(6);

    ds.Union(0, 1);
    ds.Union(1, 2);
    ds.Union(3, 4);
    ds.Union(2, 4);
    ds.Union(5, 6);
    ds.Union(4, 6);
    ds.Union(0, 1);

    cout << ds.searchInSet(0) << endl;
    cout << ds.searchInSet(1) << endl;
    cout << ds.searchInSet(2) << endl;
    cout << ds.searchInSet(3) << endl;
    cout << ds.searchInSet(4) << endl;
    cout << ds.searchInSet(5) << endl;
    cout << ds.searchInSet(6) << endl;

        return 0;
}
```