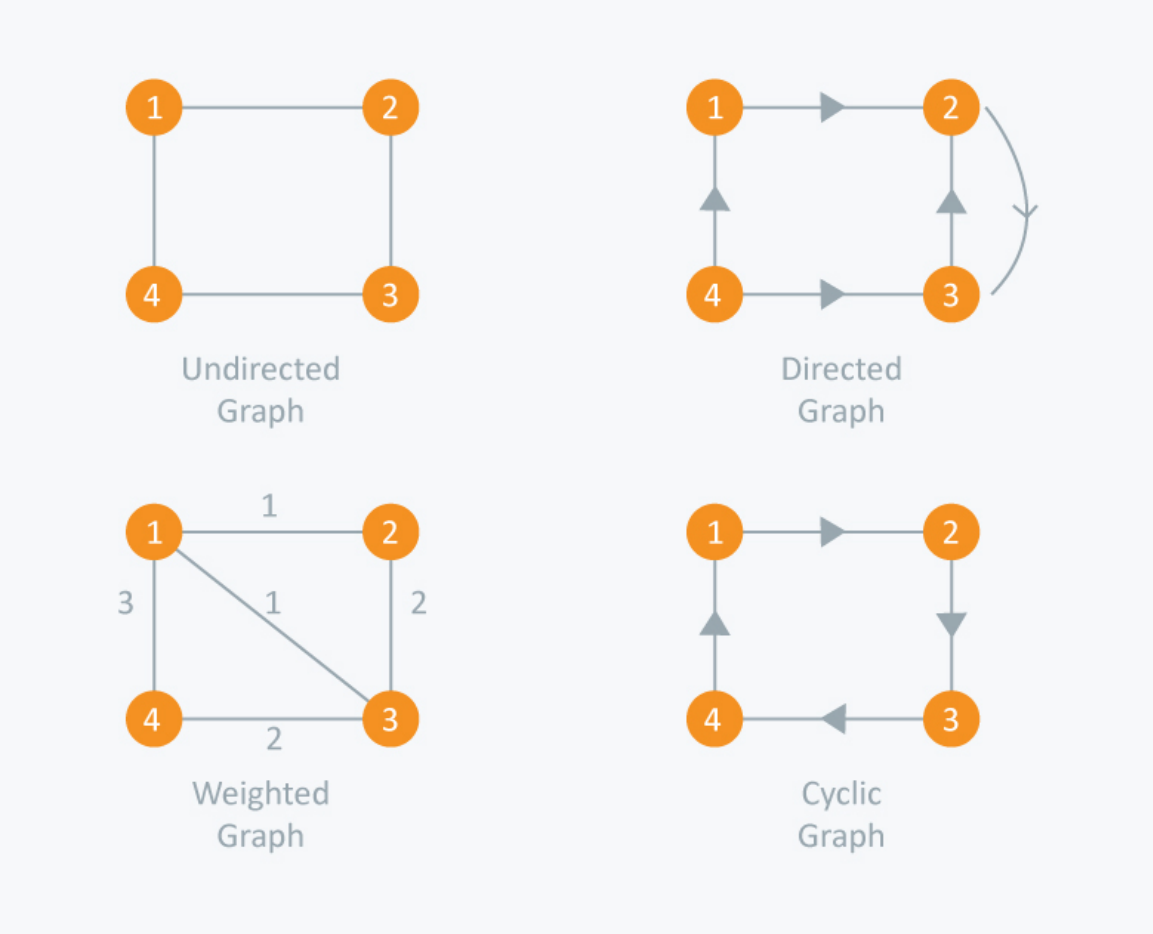
Minimum Spanning Tree

75 Minimum-spanning-tree Prims-algorithm Kruskals-algorithm CodeMonk

Graph is an ordered pair **G = (V, E)** where **V** is the set of nodes or vertices and **E** is the set of edges. Graphs are of two types:

1. **Undirected:** Undirected graph is a graph in which all the edges are bidirectional, essentially the edges don't point in a specific direction.
2. **Directed:** Directed graph is a graph in which all the edges are unidirectional.

Graph can also be weighted. A **weighted graph** is the one in which each edge is assigned an integer or weight. A graph is called **cyclic** if there is a path in the graph which starts from and ends at the same vertex.



A **tree** is an undirected graph in which any two vertices are connected by only one path. Tree is acyclic and has **N - 1** edges where **N** is the number of vertices.

What is a Spanning Tree?

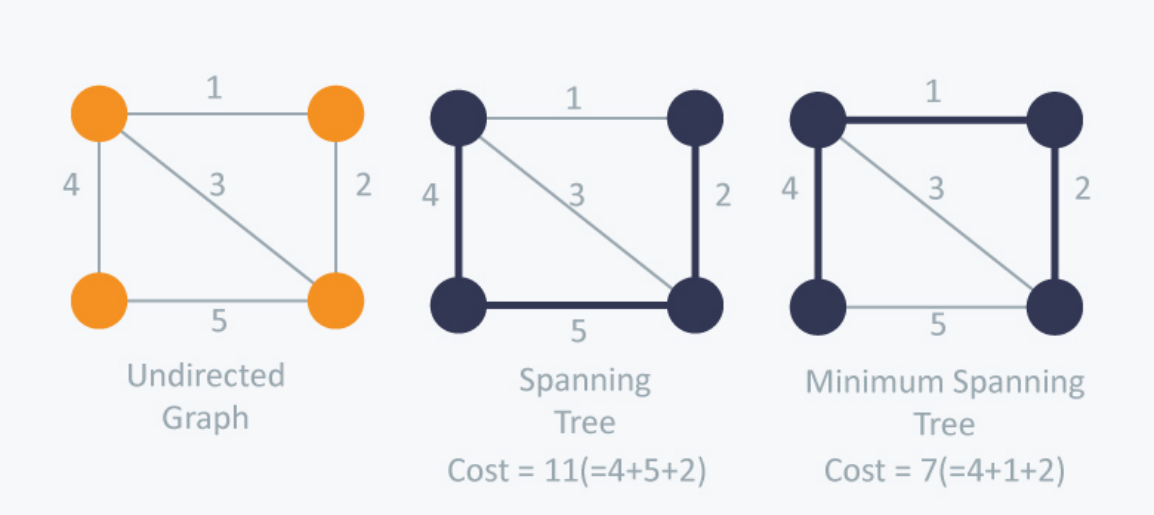
Given an undirected and connected graph G = (V, E), a spanning tree of the graph G is a tree that spans G (that is, it includes every vertex of G) and is a subgraph of G (every edge in the tree belongs to G)

What is a Minimum Spanning Tree?

The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the

spanning trees. There can be many minimum spanning trees also.
Minimum spanning tree has direct application in the design of networks. It is used in algorithms approximating the travelling salesman problem, multi-terminal minimum cut problem and minimum-cost weighted perfect matching. Other practical applications are:

1. Cluster Analysis
2. Handwriting recognition
3. Image segmentation

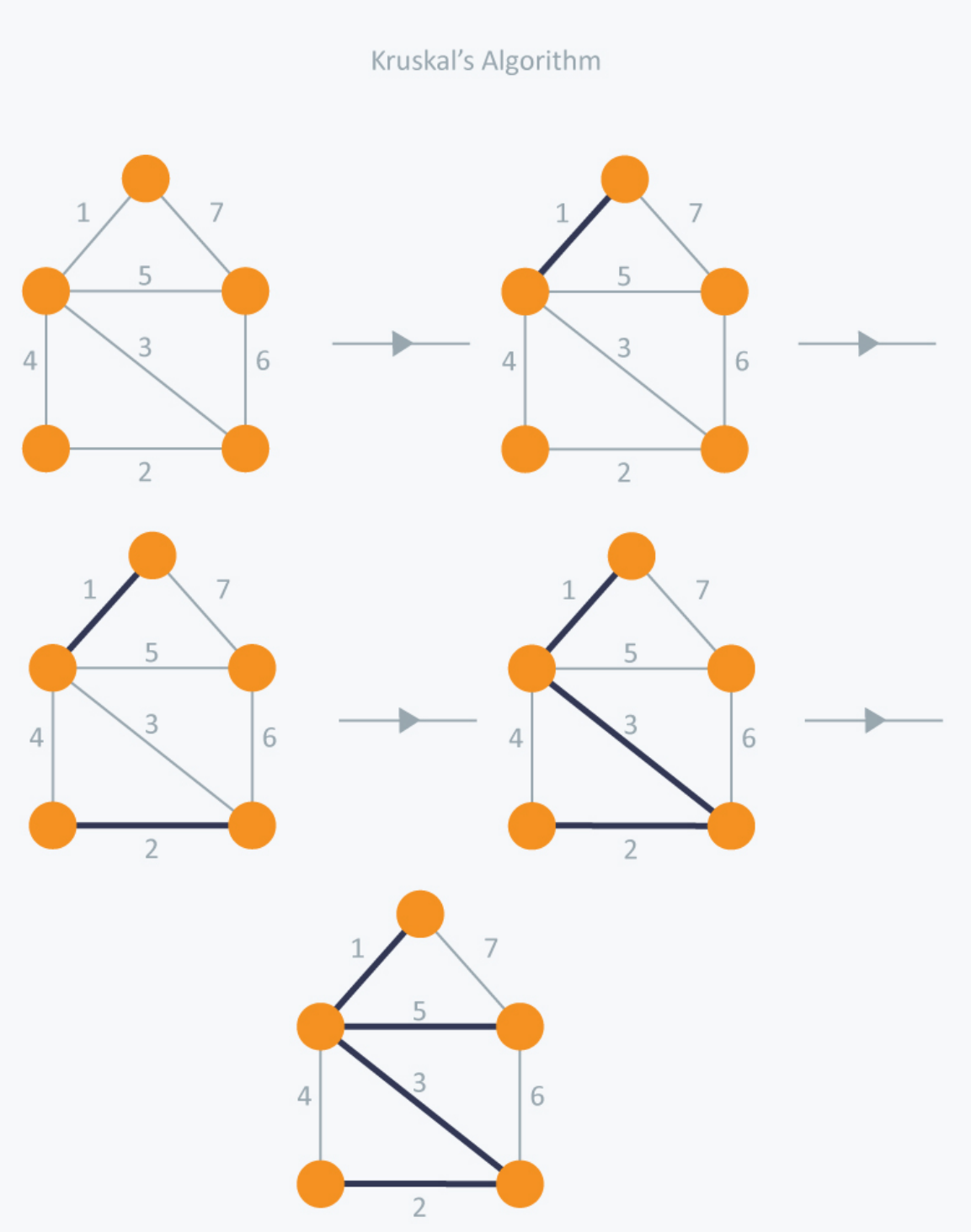


There are two famous algorithms for finding the Minimum Spanning Tree:

Kruskal's Algorithm

Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree. To do this we will simply sort the edges in the increasing order of their weight and select the edges one by one from the beginning. But sometimes, selecting edges in this order can create a cycle. To avoid cycle creation, we have to check if the selected edge is creating a cycle or not. If it is creating a cycle then simply ignore it otherwise add it to into the growing spanning tree. This can be done simply use **Disjoint Set data structure (Union Find)**. Disjoint Set data structure can tell us if two nodes are connected or not. So if the endpoints of an edge have same root , i.e., they are connected, then this edge will create a cycle, so just ignore it otherwise add it into the growing spanning tree. So at the end of the algorithm we will end up with a tree which is a minimum spanning tree.

Let's consider an example.



In Kruskal's algorithm, at each iteration we will select the edge with the lowest weight. So, we will start with the lowest weighted edge first i.e., the edges with weight 1. After that we will select the second lowest weighted edge i.e., edge with weight 2. Notice these two edges are totally disjoint. Now, the next edge will be the third lowest weighted edge i.e., edge with weight 3, which connects the two disjoint pieces of the graph. Now, we are not allowed to pick the edge with weight 4, that will create a cycle and we can't have any cycles. So we will select the fifth lowest weighted edge i.e., edge with weight 5. Now the other two edges will create cycles so we will ignore them. In the end, we end up with a minimum spanning tree with total cost 11 (= 1 + 2 + 3 + 5).

Pseudo code:

1. Sort the edges in nondecreasing order of the edge weights.
2. Select the edge with minimum weight and if cycle is formed, ignore it, otherwise add this edge into the spanning tree.
3. Repeat step 2 for all edges.

Implementation:

```cpp
#include <iostream>
#include <vector>
#include <utility>
#include <algorithm>

using namespace std;
const int MAX = 1e4 + 5;
int id[MAX], nodes, edges;
pair <long long, pair<int, int> > p[MAX];

void initialize()
{
    for(int i = 0;i < MAX;++i)
        id[i] = i;
}

int root(int x)
{
    while(id[x] != x)
    {
        id[x] = id[id[x]];
        x = id[x];
    }
    return x;
}

void union1(int x, int y)
{
    int p = root(x);
    int q = root(y);
    id[p] = id[q];
}

long long kruskal(pair<long long, pair<int, int> > p[])
{
    int x, y;
    long long cost, minimumCost = 0;
    for(int i = 0;i < edges;++i)
    {
        // Selecting edges one by one in increasing order from the beginning
        x = p[i].second.first;
        y = p[i].second.second;
        cost = p[i].first;
        // Check if the selected edge is creating a cycle or not
        if(root(x) != root(y))
        {
            minimumCost += cost;
            union1(x, y);
        }
    }
    return minimumCost;
}

int main()
{
    int x, y;
    long long weight, cost, minimumCost;
    initialize();
    cin >> nodes >> edges;
```

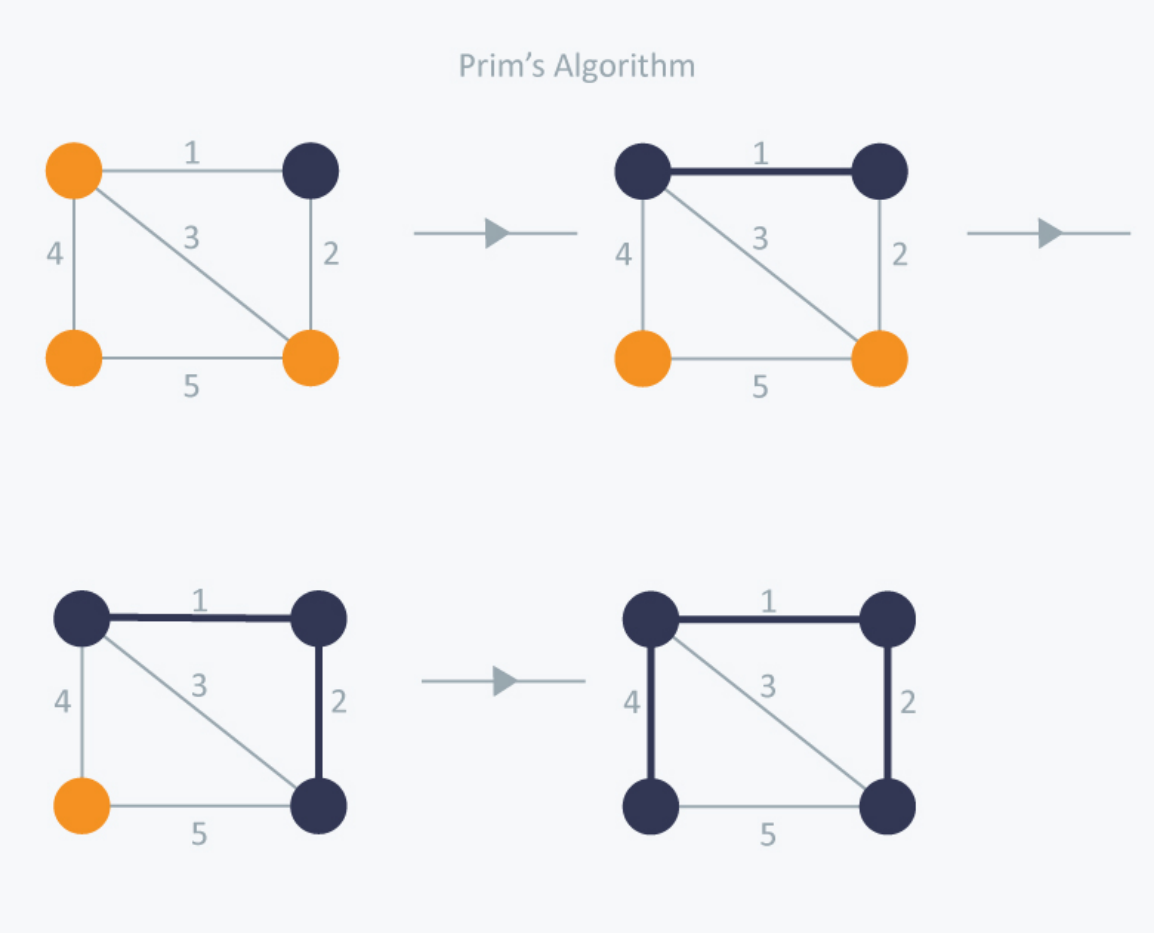
```
    for(int i = 0;i < edges;++i)
    {
        cin >> x >> y >> weight;
        p[i] = make_pair(weight, make_pair(x, y));
    }
    // Sort the edges in the ascending order
    sort(p, p + edges);
    minimumCost = kruskal(p);
    cout << minimumCost << endl;
    return 0;
}
```

Time Complexity:
In Kruskal's algorithm, most time consuming operation is sorting because the total complexity of the Disjoint-Set operations will be **O(E(log*)V)**. So total running time will be **O(ElgE)**. But since $|E| < |V|^2$, logE = O(lgV). So time complexity of Kruskal's algorithm is **O(ElgV)**.

Prim's Algorithm

Prim's Algorithm also use Greedy Approach to find the minimum spanning tree. In Prim's Algorithm we grow the spanning tree from a starting position. In contrast to the Kruskal's Algorithm, Prim's Algorithm selects one vertex at a time and adds it into the growing spanning tree. Prim's Algorithm maintains two disjoint sets of vertices. One contains vertices that are in the growing spanning tree and other that are not in the growing spanning tree. Prim's algorithm selects the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. We will simply insert the vertices, that are connected to growing spanning tree, into the Priority Queue. But just like Kruskal's Algorithm, we need to check for cycles. To do that we will mark the nodes which we have already selected and insert only those nodes in the Priority Queue that are not marked.

Let's look at an example.



In Prim's Algorithm, we will start with an arbitrary node (it doesn't matter which one) and mark it. In each

iteration we will mark a new vertex that is adjacent to the one that we have already marked. As a greedy algorithm, Prim's algorithm will select the cheapest edge and mark the vertex. So we will simply choose the edge with weight 1. In the next iteration we have three options, edges with weight 2, 3 and 4. So, we will select the edge with weight 2 and mark the vertex. Now again we have three options, edges with weight 3, 4 and 5. But we can't choose edge with weight 3 as it is creating a cycle. So we will select the edge with weight 4 and we end up with the minimum spanning tree of total cost 7 (= 1 + 2 +4).

Pseudo code:

1. Start with a random vertex and mark it.
2. Check all the vertices that are adjacent to any marked vertex and select the cheapest vertex does not create a cycle and mark it.
3. Repeat step 2 until all the vertices are marked.

Implementation:

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <functional>
#include <utility>

using namespace std;
const int MAX = 1e4 + 5;
typedef pair<long long, int> PII;
bool marked[MAX];
vector <PII> adj[MAX];

long long prim(int x)
{
    priority_queue<PII, vector<PII>, greater<PII> > Q;
    int y;
    long long minimumCost = 0;
    PII p;
    Q.push(make_pair(0, x));
    while(!Q.empty())
    {
        // Select the edge with minimum weight
        p = Q.top();
        Q.pop();
        x = p.second;
        // Checking for cycle
        if(marked[x] == true)
            continue;
        minimumCost += p.first;
        marked[x] = true;
        for(int i = 0;i < adj[x].size();++i)
        {
            y = adj[x][i].second;
            if(marked[y] == false)
                Q.push(adj[x][i]);
        }
    }
    return minimumCost;
}

int main()
{
    int nodes, edges, x, y;
    long long weight, minimumCost;
    cin >> nodes >> edges;
    for(int i = 0;i < edges;++i)
    {
        cin >> x >> y >> weight;
        adj[x].push_back(make_pair(weight, y));
        adj[y].push_back(make_pair(weight, x));
    }
    // Selecting 1 as the starting node
    minimumCost = prim(1);
    cout << minimumCost << endl;
    return 0;
}
```

Time Complexity:

The time complexity of the Prim's Algorithm is **O((V + E)logV)** because each vertex is inserted in the priority queue only once and insertion in priority queue take logarithmic time.

Solve Problems

AUTHOR

Akash Sharma
Problem Setter and Tester ...
Dehradun
7 notes

TRENDING NOTES

Sorting And Searching Algorithms - Time Complexities Cheat Sheet
written by Vipin Khushu

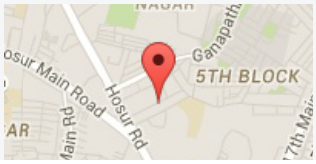
CodeMonk Dynamic Programming II
written by Tanmay Sahay

Efficient Factorials Calculation !
written by Ankur Anand

Trie, Suffix Tree, Suffix Array
written by pkacprzak

Technique to play online Bot games
written by Catalin Stefan Tiseanu

more ...



Hackathons

FAQs

Customers

Annual Report

IIIrd Floor, Salarpuria Business Center,
4th B Cross Road, 5th A Block,
Koramangala Industrial Layout,
Bangalore, Karnataka 560095, India.

contact@hackerearth.com

+91-80-4155-4695

+1-650-461-4192