

GRAPH DATA STRUCTURE AND ALGORITHMS - 2

Minimum (Cost) Spanning Tree

Spanning Tree

In an undirected and connected graph $G = (V, E)$, a spanning tree is a subgraph that is a tree which includes all of the vertices of G , with minimum possible number of edges. A graph may have several spanning trees. The cost of the spanning tree is the sum of the weights of all the edges in the tree.

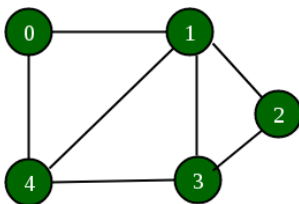
Properties of Spanning Tree:

- There may be several minimum spanning trees of the same weight having the minimum number of edges.
- If all the edge weights of a given graph are the same, then every spanning tree of that graph is minimum.
- If each edge has a distinct weight, then there will be only one, unique minimum spanning tree.
- A connected graph G can have more than one spanning trees.
- A disconnected graph can't have to span the tree, or it can't span all the vertices.
- Spanning Tree doesn't contain cycles.
- Spanning Tree has $(n-1)$ edges where n is the number of vertices.

Algorithms for Spanning Tree:

- BFS
- DFS

Sample Problem 1: Printing Spanning Tree of an Unweighted Graph



Sample Input

```

5 7
0 1
4 3
0 4
3 1
3 2
1 2
1 4

```

Sample Output

```

(1, 0) (2, 3) (3, 1) (4, 3)

```

Code: Solution with DFS

```
// Spanning Tree.cpp : Graph is always connected

#include <fstream>
#include <vector>

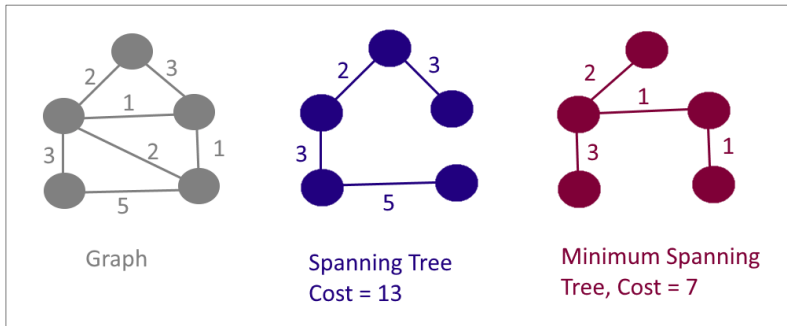
using namespace std;

ifstream cin("1.in");
ofstream cout("1.out");

int N, M;
vector<int> parent;
vector<vector<int>> adjList;
//-----
void readInput()
{
    cin >> N >> M;
    adjList.resize(N);
    for (int i = 0; i < N; i++)
    {
        int u, v;
        cin >> u >> v;
        adjList[u].push_back(v);
        adjList[v].push_back(u);
    }
}
//-----
void dfs(int u)
{
    for (int v:adjList[u])
        if (parent[v] == -1) //not visited yet
        {
            parent[v] = u; //MST edge between u and v
            dfs(v);
        }
}
//-----
void printMST()
{
    for (int i = 1; i < N; i++)
        cout << "(" << i << ", " << parent[i] << ") ";
}
//-----
int main()
{
    readInput();
    parent.resize(N, -1);
    parent[0] = 0;
    dfs(0);
    printMST();
}
```

Minimum Spanning Tree

A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted (un)directed graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. That is, it is a spanning tree whose sum of edge weights is as small as possible. Number of edges in MST: $V-1$. All MST algorithms fall under a class of algorithms called *greedy algorithms* which find the local optimum in the hopes of finding a global optimum.



Algorithms for Minimum Spanning Tree:

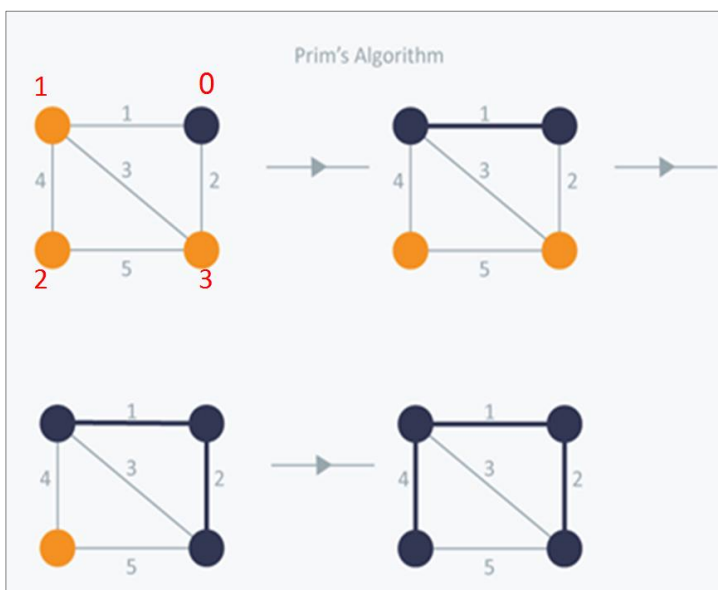
- Prim (Jarnik's) algorithm
- Kruskal's algorithm
- Bruvka (Sollin's) algorithm
- Reverse delete algorithm
- Remove heaviest edges in cycles algorithm

Prim's Algorithm

Prim's Algorithm also greedy approach to find the minimum spanning tree. In Prim's Algorithm we grow the spanning tree from a starting position. We add vertex to the growing spanning tree in Prim's. It runs in $O((V+E)\log V)$ time with **adjacency list** representation and help of a **priority queue**.

Algorithm Steps:

- Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
- Select 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. Insert the vertices that are connected to growing spanning tree, into the Priority Queue.
- Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.



Code 1: $O(N^2)$ solution with adjacency list

```
// Prim MST.cpp :  $O(N^2)$  solution with adjacency list.

#include <iostream>
#include <vector>
#include <algorithm>

#define BIGNUM 999

using namespace std;

int N = 4, M = 5;

struct Edge
{
    int u, v, cost;
};

vector<Edge> edgeList = { {0, 1, 1}, {0, 3, 2}, {2, 1, 4}, {3, 2, 5}, {1, 3, 3} };
vector<vector<pair<int,int>>> adjList;
int mstCost;
int startNode = 0;
//-----
void constructAdjList()
{
    adjList.resize(N);
    for (auto e : edgeList)
    {
        adjList[e.u].push_back({ e.v, e.cost });
        adjList[e.v].push_back({ e.u, e.cost });
    }
}
//-----
//Calculate the cost of the MST
void primMST()
{
    vector<int> distances(N, BIGNUM);
    vector<bool> visited(N, false);

    distances[startNode] = 0;
    visited[startNode] = true;

    int currentNode = startNode;

    for(int i=0; i<N-1; i++)
    {
        //update the distances from the current node.
        for (auto e : adjList[currentNode])
        {
            distances[e.first] = min(distances[e.first], e.second);
        }

        //Find the next closest unvisited node
        int nextNode;
        int d = BIGNUM;
        for (int j = 0; j < N; j++)
        {
            if (visited[j])
                continue;
            if (distances[j] < d)
            {
                d = distances[j];
                nextNode = j;
            }
        }

        //Include the new edge into the mst
        visited[nextNode] = true;
        mstCost += d;
        currentNode = nextNode;
    }
}
```

```

    }
}
//-----
int main()
{
    constructAdjList();
    primMST();
    cout<<mstCost<<endl;
}

```

Output

7

Code 2: $O(N*N)$ solution with adjacency matrix

```

// Prim MST.cpp :  $O(N*N)$  solution with adjacency matrix.

#include <iostream>
#include <vector>
#include <algorithm>

#define BIGNUM 999

using namespace std;

int N = 4, M = 5;

struct Edge
{
    int u, v, cost;
};

vector<Edge> edgeList = { {0, 1, 1}, {0, 3, 2}, {2, 1, 4}, {3, 2, 5}, {1, 3, 3} };
vector<vector<int>> adjMat;
int mstCost;
int startNode = 0;
//-----
void constructAdjList()
{
    adjMat.resize(N, vector<int>(N, BIGNUM));
    for (auto e : edgeList)
    {
        adjMat[e.u][e.v] = adjMat[e.v][e.u] = e.cost;
    }
}
//-----
void primMST()
{
    vector<int> distances(N, BIGNUM);
    vector<bool> visited(N, false);

    distances[startNode] = 0;
    int currentNode = startNode;

    for (int i = 0; i < N - 1; i++)
    {
        visited[currentNode] = true;

        //update the distances from the current node.
        for (int j=0; j<N; j++)
        {
            distances[j] = min(distances[j], adjMat[currentNode][j]);
        }

        //Find the next closest unvisited node
        int nextNode;
        int d = BIGNUM;
    }
}

```

```

        for (int j = 0; j < N; j++)
        {
            if (visited[j])
                continue;
            if (distances[j] < d)
            {
                d = distances[j];
                nextNode = j;
            }
        }

        //Include the new edge into the mst
        mstCost += d;
        currentNode = nextNode;
    }
}
//-----
int main()
{
    constructAdjList();
    primMST();
    cout << mstCost << endl;
}

```

Output

7

Code 3: $O(N\log N)$ solution with priority queue

```

// Prim MST.cpp :  $O(N*\log N)$  solution.

#include <iostream>
#include <vector>
#include <algorithm>
#include <queue>

#define BIGNUM 999

using namespace std;

int N = 4, M = 5;

struct Edge
{
    int u, v, cost;
};

vector<Edge> edgeList = { {0, 1, 1}, {0, 3, 2}, {2, 1, 4}, {3, 2, 5}, {1, 3, 3} };
vector<vector<pair<int,int>>> adjList;
int mstCost;
int startNode = 0;
//-----
void constructAdjList()
{
    adjList.resize(N);
    for (auto e : edgeList)
    {
        adjList[e.u].push_back({ e.v, e.cost });
        adjList[e.v].push_back({ e.u, e.cost });
    }
}
//-----
void primMST()
{
    vector<bool> visited(N, false);

    //Declare a min heap. first: distance, second: node id
    priority_queue<pair<int, int>, vector<pair<int,int>>, greater<pair<int,int>>> > pq;

    pq.push({ 0, startNode });
}

```

```

while(!pq.empty())
{
    pair<int, int> p = pq.top();
    pq.pop();

    int currentNode = p.second;
    if (visited[currentNode])
        continue;

    //update the total cost and visited status of the current node
    visited[currentNode] = true;
    mstCost += p.first;

    //push all unvisited neighbours of the current node into the pq
    for (auto p2 : adjList[currentNode])
    {
        if (!visited[p2.first])
            pq.push({ p2.second, p2.first });
    }
}
//-----
int main()
{
    constructAdjList();
    primMST();
    cout<<mstCost<<endl;
}

```

Output

7

Sample Problem 2: Min Cost to Repair Edges

There's an undirected connected graph with n nodes labeled $1 \dots n$. But some of the edges have been broken disconnecting the graph. Find the minimum cost to repair the edges so that all the nodes are once again accessible from each other.

Input:

- **n**: an int representing the total number of nodes.
- **edges**: a list of integer pair representing the nodes connected by an edge.
- **edgesToRepair**: a list where each element is a triplet representing the pair of nodes between which an edge is currently broken and the cost of repairing that edge, respectively (e.g. $[1, 2, 12]$ means to repair an edge between nodes 1 and 2, the cost would be 12).

Example

Input: $n = 5$, edges = $[[1, 2], [2, 3], [3, 4], [4, 5], [1, 5]]$, edgesToRepair = $[[1, 2, 12], [3, 4, 30], [1, 5, 8]]$

Output: 20

Explanation:

There are 3 connected components due to broken edges: $[1]$, $[2, 3]$ and $[4, 5]$.

We can connect these components into a single component by repairing the edges between nodes 1 and 2, and nodes 1 and 5 at a minimum cost $12 + 8 = 20$.

Sample Input

```
5 5 3
1 2
2 3
3 4
4 5
1 5
1 2 12
3 4 30
1 5 8
```

Sample Output

```
20
```

Solution Idea

Just take existing edges to have 0 cost and broken edges have their given cost. And finding minimum spanning tree cost will be the answer.

(Source: LeetCode. Amazon interview question)

Code: Solution with Prim's algorithm

// Min Cost to Repair Edges.cpp : Solution with Prim's Algorithm

```
#include <fstream>
#include <vector>
#include <queue>
#include <map>

#define BIGNUM INT_MAX

using namespace std;

ifstream cin("1.in");
ofstream cout("1.out");

int N, M, K;
vector < vector<int>> adjList;
map<pair<int, int>, int> m;
int res;

//*****
void readInput()
{
    //number of nodes, edges and edges to repair
    cin >> N >> M >> K;

    //Create an adjacency list without costs
    adjList.resize(N);
    for (int i = 0; i < M; i++)
    {
        int u, v;
        cin >> u >> v;
        u--; v--;
        adjList[u].push_back(v);
        adjList[v].push_back(u);
    }

    //map the edges with their costs
```



```

for (int i = 0; i < K; i++)
{
    int u, v, c;
    cin >> u >> v >> c;
    u--; v--;
    if (v < u)
        swap(u, v);
    m[make_pair(u, v)] = c;
}
}
//-----
//Create a MST and calculate the total cost
void go()
{
    //min heap. first: cost, second: node id.
    priority_queue< pair<int, int>, vector<pair<int,int>>, greater<pair<int,int>> > pq;
    vector<bool> visited(N, false);

    //We consider that node 0 is the starting node.
    pq.push({ 0,0 });

    while (!pq.empty())
    {
        pair<int, int> p = pq.top();
        pq.pop();

        int currentNode = p.second;
        int cost = p.first;

        if (visited[currentNode])
            continue;

        res += cost;
        visited[currentNode] = true;

        for (int x : adjList[currentNode])
        {
            if (visited[x])
                continue;

            int u = currentNode;
            int v = x;
            if (v < u)
                swap(u, v);
            //if the edge is not in the map, default cost will be 0.
            pq.push({ m[{u, v}], x });
        }
    }
}
//-----
int main()
{
    readInput();
    go();
    cout << res << endl;
}

```

Kruskal's Algorithm

Kruskal's algorithm initially places all the nodes of the original graph isolated from each other, to form a forest of single node trees, and then gradually merges these trees, combining at each iteration any two of all the trees with some edge of the original graph.

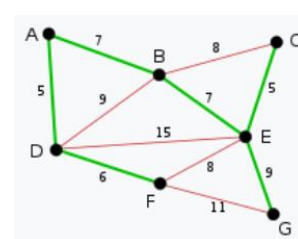
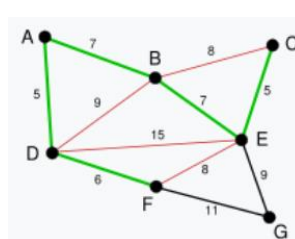
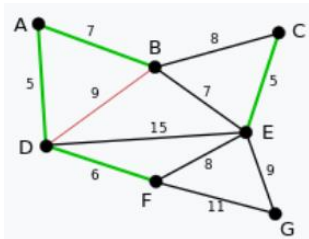
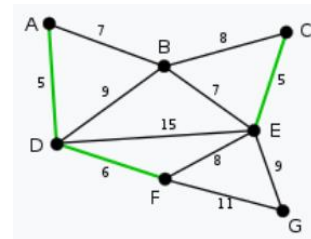
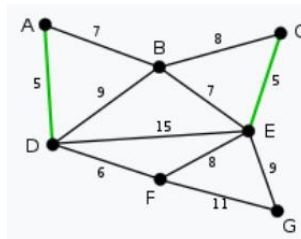
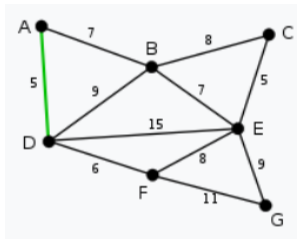
Before the execution of the algorithm, all edges are sorted by weight (in non-decreasing order). Then begins the process of unification: pick all edges from the first to the last (in sorted order), and if the ends of

the currently picked edge belong to different subtrees, these subtrees are combined, and the edge is added to the answer.

After iterating through all the edges, all the vertices will belong to the same sub-tree, and we will get the answer. Kruskal's algorithm uses **DSU (disjoint set union)** structure to unify the disjoint trees.

- create a forest F (a set of trees), where each vertex in the graph is a separate tree
- create a set S containing all the edges in the graph
- while S is nonempty
 - remove an edge with minimum weight from S
 - if that edge connects two different trees, then add it to the forest, combining two trees into a single tree
 - otherwise discard that edge

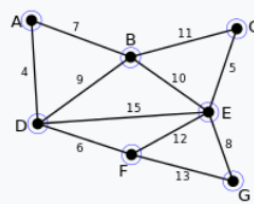
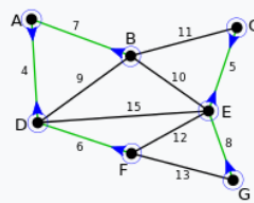
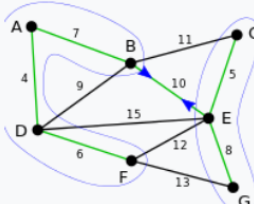
At the termination of the algorithm, the forest has only one component and forms a minimum spanning tree of the graph.



Bruvka's Algorithm

The algorithm begins by finding the minimum-weight edge incident to each vertex of the graph, and adding all of those edges to the forest. Then, it repeats a similar process of finding the minimum-weight edge from each tree constructed so far to a different tree, and adding all of those edges to the forest. Bruvka's algorithm uses **DSU (disjoint set union)** structure to unify the disjoint trees.

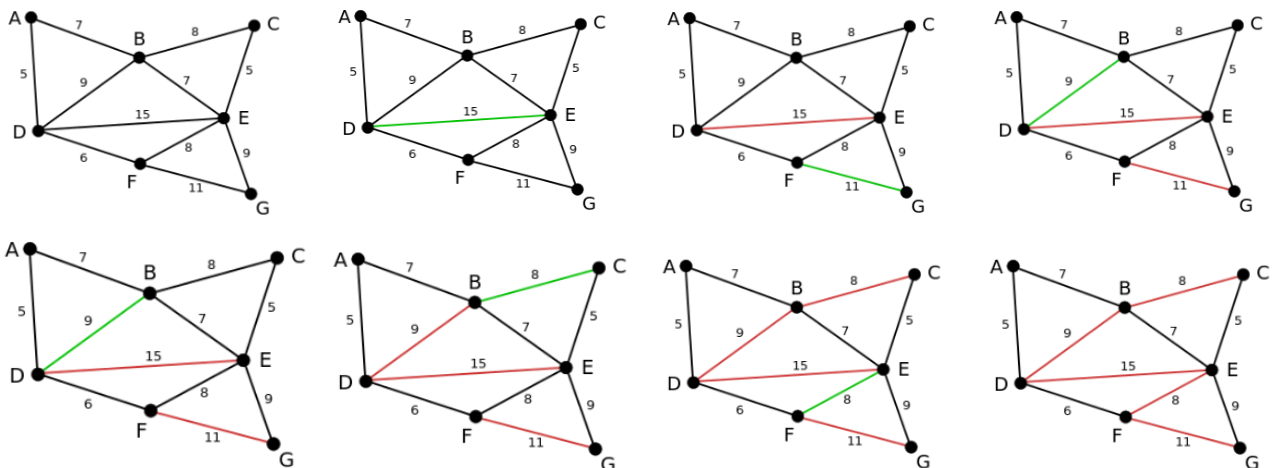
Each repetition of this process reduces the number of trees, within each connected component of the graph, to at most half of this former value, so after logarithmically many repetitions the process finishes. When it does, the set of edges it has added forms the minimum spanning forest

Image	components
	{A} {B} {C} {D} {E} {F} {G}
	{A,B,D,F} {C,E,G}
	{A,B,C,D,E,F,G}

Reverse Delete Algorithm

It is the reverse of Kruskal's algorithm, which is another greedy algorithm to find a minimum spanning tree. Kruskal's algorithm starts with an empty graph and adds edges while the Reverse-Delete algorithm starts with the original graph and deletes edges from it. The algorithm works as follows:

- Start with graph G, which contains a list of edges E.
- Go through E in decreasing order of edge weights.
- Check if deleting current edge will further disconnect graph.
- If G is not further disconnected, delete the edge.



Removing Heaviest Edged in Cycles

The algorithm does not use any sorting algorithm or priority queue or binary heaps. The algorithm is based on Depth-First-Search (DFS) and the heaviest edge of the seen cycle in DFS is deleted.

An alternative implementation is to sort the edges in descending order and test each of them if it is in a cycle.

Practice Problems

1. Implement the “Printing Spanning Tree of an Unweighted Graph” program with BFS.
2. Calculate the cost of the MST and print the edges of MST.
3. In the priority queue implementation of the Prim’s algorithm, we may have the multiple occurrence of the same node in the priority queue. Re-implement the program and do not push a node if it is not useful.
4. Enumerate all spanning trees of a graph.
5. Enumerate all minimum spanning trees of a graph.

Useful Links and References

<https://algorithms.tutorialhorizon.com/introduction-to-minimum-spanning-tree-mst/>

<https://www.javatpoint.com/minimum-spanning-tree-introduction>

<https://leetcode.com/discuss/interview-question/357310>

<https://www.hackerearth.com/practice/algorithms/graphs/minimum-spanning-tree/tutorial/>

https://cosmopedia.net/Reverse-delete_algorithm

<https://www.wikizeroo.org/index.php?q=aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnL3dpa2kvQm9yxa92a2Enc19hbGdvcmI0aG0>

<https://www.tandfonline.com/doi/abs/10.1080/00207169508804419?journalCode=gcom20>

https://cp-algorithms.com/graph/mst_kruskal.html