ALGO-101

Week 6 - Graph Related Problems

Novruz Amirov

ITU ACM

November 2022

Topics

Topics covered at week 6:

- Graph Related Problems
    - Dijkstra's Algorithm
    - Kruskal's Algorithm
    - Union-Find Structure
    - Prim's Algorithm

Important Graph Problems

2 of the Most Famous Problems related to Graphs

1. Shortest Path
    1.1 Dijkstra's Algorithm
2. Minimum Spanning Tree
    2.1 Kruskal's Algorithm
    2.2 Union-Find Structure
    2.3 Prim's Algorithm

Shortest Path

- **Shortest Path** between **two nodes:**

In **Graph Theory**, the Shortest Path Problem is finding a path between 2 vertices in a graph such that the sum of the weights of its constituent edges is **MINIMIZED**.

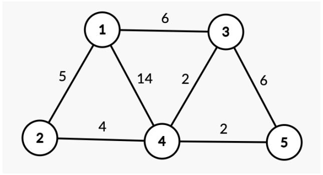


Figure: Can you find the best way from Node 1 to Node 5?

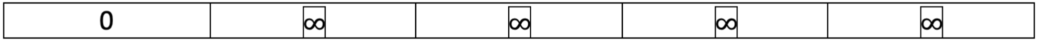
Dijkstra's Algorithm

- How the **Dijkstra's Algorithm** works:
    - The **Distance** (distance vector) to the **Starting Node** is **0** and to all **other Nodes** is **Infinite**(INTMAX).
    - All Nodes in **visited vector** are given as **False** (meaning that, in the beginning none of the nodes are visited).
    - **Starting Node** is added to the **Priority Queue** first. Priority Queue is used for choosing the Node with smallest distance (that's why it is added with minus sign. p.s. you will see it in code section).
    - The Dijkstra's Algorithm chooses **unvisited** node with **smallest** distance by comparing previous distance to that node in distance vector.
    - The algorithm continues **until there is not any unvisited node**

Dijkstra's Algorithm Example: Step by Step

STEP 1 :

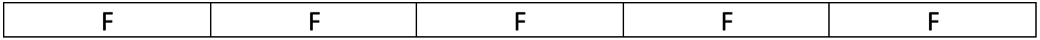
- Distance Vector:



0	∞	∞	∞	∞

Figure: The distance to each node is infinite at the beginning **(except starting node)**

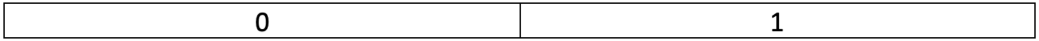
- Visited Vector:



F	F	F	F	F

Figure: Every node is marked as unvisited at the beginning

- Priority Queue:



0	1

Figure: Initially, the starting node with distance 0 in Priority Queue

Dijkstra's Algorithm Example: Step by Step

STEP 2 - 3 :

- Distance Vector:

0	5	6	14	∞

0	5	6	9	∞

- Visited Vector:

T	F	F	F	F

T	T	F	F	F

- Priority Queue:

-5	2
-6	3
-14	4

-6	3
-9	4
-14	4

Dijkstra's Algorithm Example: Step by Step

STEP 4 - 5 :

- Distance Vector:

0	5	6	8	12

0	5	6	8	10

- Visited Vector:

T	T	T	F	F

T	T	T	T	F

- Priority Queue:

-8	4
-9	4
-12	5
-14	4

-9	4
-10	5
-12	5
-14	4

Dijkstra's Algorithm Example: Step by Step

FINAL STEP:

- Distance Vector:

0	5	6	8	10

Figure: The Final distances to each node are calculated

- Visited Vector:

T	T	T	T	T

Figure: All Nodes are visited in Final Step

- Priority Queue:
  Because all Nodes are already visited, the items in priority queue will be deleted until there is not any

Sample Code

To represent the Graph using Adjacency List

```
int n, m; // n -> number of nodes, m -> number of edges
cin >> n >> m;

vector<vector<pair<int,int>>>adjacencyList(n+1,vector<pair<int, int>>(0));

for(int i = 0; i < m; i++){
     int a, b, w;
     cin >> a >> b >> w;
     adjacencyList[a].push_back(make_pair(b, w));
     adjacencyList[b].push_back(make_pair(a, w));
}
```

Sample Code (Starting Dijkstra's Algorithm)

```cpp
vector<int> distance(n + 1, INT_MAX); // representing INFINITY with INTMAX
vector<bool> visited(n + 1, false); // to know if the node is visited
distance[startingNode] = 0; // the distance to starting node is 0

priority_queue<pair<int, int>> priorityQueue; // prioity queue contains
// pair of distance from starting node and the node number

priorityQueue.push({0, startingNode}); // pushing the node 1 with the
    distance of 0.

while(!priorityQueue.empty()){
    int a = priorityQueue.top().second; // represent starting node
    priorityQueue.pop();

    if(visited[a]) // if this node is visited, continue.
        continue;

    visited[a] = true; // otherwise make this one visited true.
```

Sample Code

To find the Shortest Distance to every node from Starting Node and display it

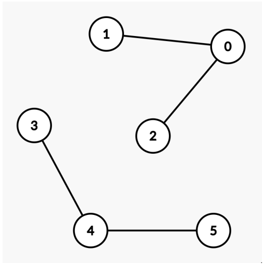
```
for(auto connection : adjacencyList[a]){
    int b = connection.first;
    int w = connection.second; // w->weight of the connection of a and b.

    // if the distance is smaller than the one found before, replace it
    if(distance[a] + w < distance[b]){
        distance[b] = distance[a] + w;
        // we are pushing with minus sign to find shortest path.
        priorityQueue.push({-distance[b], b});
    }
}

for(int i = 1; i < distance.size(); i++){
    cout << distance[i] << " ";
}
```

Spanning Trees

Spanning Tree:

- Consists **all nodes** in Graph.
- **Path** between **any** two nodes.
- Connected and **Acyclic** (Like Trees)

Asyclic Graph – A Graph having no Cycles, Tree – A Connected Acyclic Graph, Forest – Disconnected Acyclic Graph



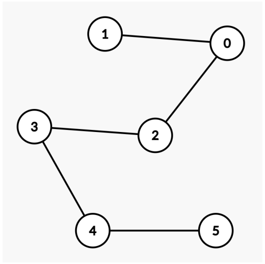


Figure: Acyclic Forest and Acyclic Tree

Minimum Spanning Tree

A **Minimum Spanning Tree** is a Spanning Tree whose Total Weight is **as small as possible**.

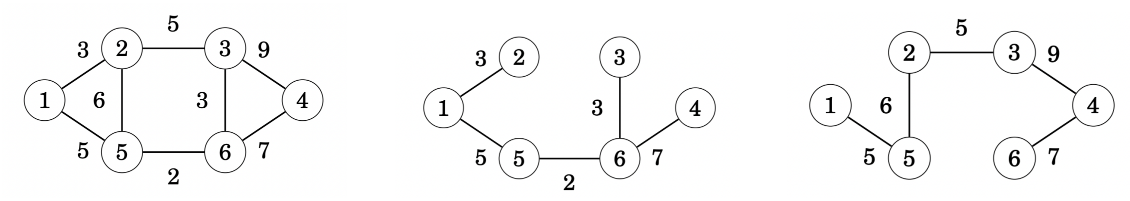


Figure: **Standard non-asyclic Graph** and **Minimum Spanning Tree** and **Maximum Spanning Tree**

Note that the Minimum and Maximum Spanning Tree of a Graph may not be **unique**.
Therefore, more than one solution **can** exist.

Algorithms to find the Minimum Spanning Tree

Kruskal's Algorithm:

- Initially, only **node** exists. There is **not** any edge.
- Edges are ordered by their **weights.**
- Starting from **smallest** edge, each edge is added, if it does not create any **cycle.**

Cycle – if the **already visited** node is visited again, while traversing Graph (such as **using DFS**), then this graph contains cycle.

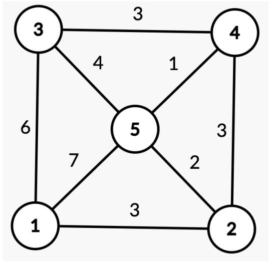


Figure: Example Non-Asyclic Graph

Steps of Kruskal's Algorithm (1)

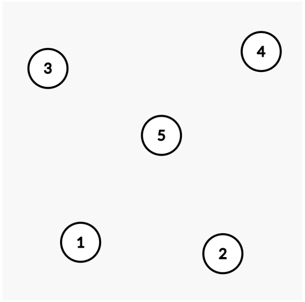
The First Step of the Algorithm is to **sort the edges in increasing (ascending) order** of their weights. The result is the following list:

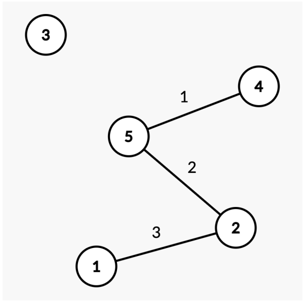
edge	weight
4 – 5	1
2 – 5	2
1 – 2	3
2 – 4	3
3 – 4	3
3 – 5	4
1 – 3	6
1 – 5	7

Table: Edges in an Increasing Order

Steps of Kruskal's Algorithm (2)

After this, the Algorithm goes through the List and adds each edge to the tree if it joins two separate components.
Initially there is no edge, then the edges according to list in **increasing order** are added, until it makes **cyclic** Graph.





Steps of Kruskal's Algorithm (3)

The edge **2 – 4** with will create a **cycle** on Graph, therefore this edge **is not added** to the Graph. After the edge 3 – 4 added to the Graph, now the Graph is connected. It means any other edge that will be added to the Graph will create a cycle. Therefore **stop** when the Graph is connected.

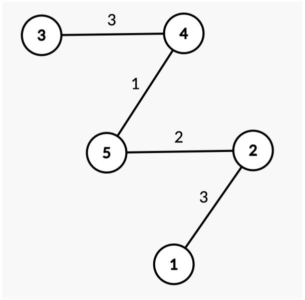


Figure: The weight of The Minimum Spanning Tree is $3 + 2 + 1 + 3 = 9$

Implementation of Kruskal

When implementing Kruskal's Algorithm, it is better to use the Adjacency List representation of the Graph. The first phase of the Algorithm sorts the edges in the list in **O(mlogm)** time. After this, the second phase of the algorithm builds the minimum spanning tree as follows:

```
// loop through each node in Adjacency List
for (...) {
if (!same(a,b)) unite(a,b);
}
```

2 **Functions** are needed to implement:
same(int, int) – check if two nodes are connected or not.
unite(int, int) – if two nodes are seperate, this methods is used for connecting two nodes.

How to Implement these Methods efficiently?

The First Idea comes to mind is to **traverse through Graph** and to see if we can move from one Node to another Node. Nevertheless, it is not a good idea, because using this method the program **will be slow**. Because we need to traverse through Graph for each time we add Edge. The Time Complexity of this will be **O(M(N+M))**.

However, we will solve it using **Union Find Structure** that implements both methods in Time Complexity of **O(logN)**. Because we will call it for each edge, the final Time Complexity of Kruskal's Algorithm will be O**(MlogN)**.

Union Find Structure (1)

The **Union-Find Structure** can be implemented using **Arrays**.

```
// initially each node are seperate set:
for (int i = 1; i <= n; i++) link[i] = i;

// because each node are seperate set, each set size is 1.
for (int i = 1; i <= n; i++) size[i] = 1;


// to find the parent of the set in which node x is.
int find(int x) {
    while (x != link[x]) x = link[x];
    return x;
}

// if the parents are same, then 2 node are in same group.
bool same(int a, int b) {
    return find(a) == find(b);
}
```

Union Find Structure (2)

To connect two separated nodes:

```cpp
// to connect two disconnected nodes.
void unite(int a, int b) {
    a = find(a); // to find the parent of the set in which node A is
    b = find(b); // to find the parent of the set in which node B is

    // add the smaller size set to larger one.
    if (size[a] < size[b]) swap(a,b);
        size[a] += size[b];

    // to make the node A parent of node B.
    link[b] = a;
}
```

Both **unite()** and **same()** function's complexity is **O(logN)**.

Kruskal's Algorithm using Union-Find structure (1)

Inserting connections into edges vector, and sorting it:

```
int main()
{
    int n, m;
    cin >> n >> m;

    // All edges of a Graph by order of {weight, startingNode, endingNode}
    vector<vector<int>> edges;
    for (int i = 0; i < m; i++)
    {
        int a, b, w;
        cin >> a >> b >> w;
        edges.push_back({w, a, b});
    }

    // Sorting the edges(increasing order) according to the weights.
    sort(edges.begin(), edges.end());
```

Kruskal's Algorithm using Union-Find structure (2)

Calculating MST Cost by checking each edge from less weighted to most weighted.

```
    int costs = 0;          // initially 0
    for (auto u : edges) // looping from less weighted to most weighted
    {
        int w = u[0]; // weight
        int a = u[1]; // starting node
        int b = u[2]; // ending node
        int x = find(a); // parent of starting node
        int y = find(b); // parent of ending node

        if (x == y) // if parents are same, they are already connected
            continue;
        unite(x, y); // if not connected, connect them
        costs += w; // and incremenet costs by w
    }
    cout << costs << endl;
    return 0;
}
```

Prim's Algorithm

Alternative method for finding **MST**(Minimum Spanning Tree).

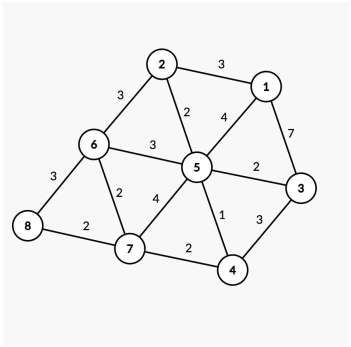
- Initially, adds an **arbitrary** node to the Tree.
- Then, always chooses **minimum-weighted edge** that is **not create a cycle**.
- Finally, all nodes have been added, and MST has been found

Prim's Algorithm looks like **Dijkstra's Algorithm**. Nevertheless, the key difference is that in Dijkstra's Algorithm the Minimum Weighted Edge from **Starting Node** is added. That is **not** the case in Prim's Algorithm. The Minimum weighted edge **which adds new Node** to the Tree will be added instead.

Implementation is also similar to Dijkstra's Algorithm, the **priority queue** is mostly used as a **data structure** to implement Prim's Algorithm. The Time Complexity of the Prim's Algorithm is same as Dijkstra's **O(N + MlogN)**.

Example (Prim's Algorithm)

We are given a Graph to find Minimum Spanning Tree. Initially there is no Edge.



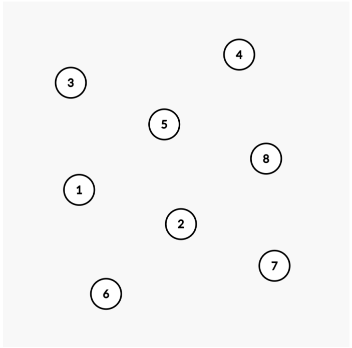
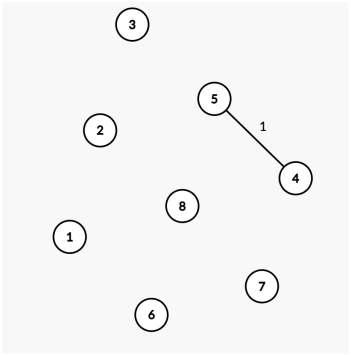
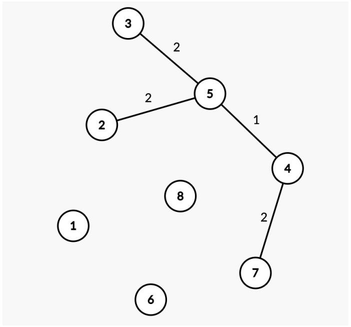


Figure: We can start from any node, let's say to start from node 5.

Example (Prim's Algorithm)

The minimum weighted edge from node 5 is node 4 with weight of 1. Then there are 3 edges with same weight (2) which goes to the new node, so let's add them to Tree.





Example (Prim's Algorithm)

We will continue until there is not any disconnected node. Like in Kruskal's Algorithm we will not add edges which will create cycle in Graph.

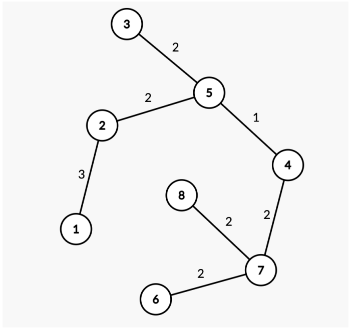


Figure: The weight of the Minimum Spanning Tree is $1 + 2 + 2 + 2 + 2 + 2 + 3 = 14$

Sample Code for Prim's Algorithm (1)

Representing Graph using Adjacency List:

```cpp
int main(){

    int n, m, startingNode;
    cin >> n >> m;
    vector<vector<pair<int, int>>> adjacencyList(n + 1, vector<pair<int,
   int>>(0));

    for (int i = 0; i < m; i++)
    {
        int a, b, w;
        cin >> a >> b >> w;
        adjacencyList[a].push_back(make_pair(b, w));
        adjacencyList[b].push_back(make_pair(a, w));
    }

    // startingNode can be arbitararily chosen
    cout << "starting point: "; cin >> startingNode;
```

Sample Code for Prim's Algorithm (2)

```cpp
// distance and visited vector are same as Dijkstra
vector<int> distance(n + 1, INT_MAX);
vector<bool> visited(n + 1, false);

int cost = 0; // storing the cost of MST
priority_queue<pair<int, int>> pq;

pq.push({0, startingNode});
distance[startingNode] = 0;

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

    if (visited[a])
        continue;
    visited[a] = true;
```

Sample Code for Prim's Algorithm (3)

```
        for (auto u : adjacencyList[a]){
            int b = u.first;
            int w = u.second;
            // unvisited and less weighted node
            if (!visited[b] && distance[b] > w)
            {
            // w is the second distance way to node, substitute previous one.
                if (distance[b] != INT_MAX)
                    count -= distance[b];
                distance[b] = w;
                pq.push({-distance[b], b});
                cost += w; // add w to count
            }
        }
    }
        cout << cost << endl;
        return 0;
    }
```

References

- A. Laaksonen – Competitive Programmer's Handbook (3 July, 2018)

Reviewers:

- Fatih Baskin
- Murat Biberoglu