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
   - 2.1 Union-Find Structure
   - 2.1 Prim's Algorithm

Shortest Path

Finding a **Shortest Path** between two nodes of a Graph is an important problem that has many practical applications. But what is the **Shortest Path of a Graph?**
In **Graph Theory**, the Shortest Path Problem is the problem of finding a path between two vertices in a graph such that the sum of the weights of its constituent edges is **MINIMIZED**.

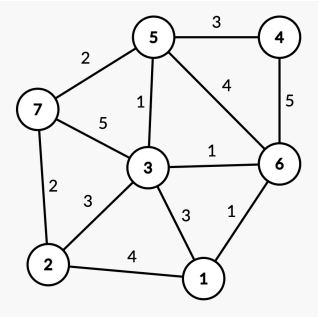


Figure: Can you find the best way from node 1 to node 7?

Algorithm to Find The Shortest Path

There are plenty of algorithms to find the Shortest Path between any 2 nodes. Nevertheless, the widely used one is **Dijkstra's algorithm**.

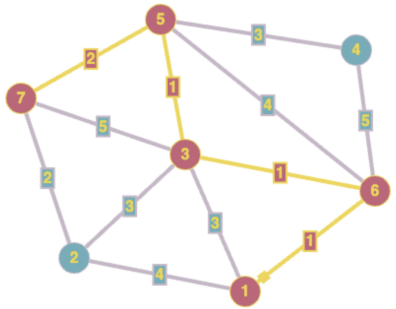


Figure: The Weight of the Shortest Path is 5: 1 – 6 – 3 – 5 – 7

Dijkstra's Algorithm

How the **Dijkstra's Algorithm** works:

Initially the **distance** to the **starting node** is 0 and the distance to all other nodes is **infinite**. At each step, Dijkstra's Algorithm selects a node that **has not been processed yet** and whose distance is **as small as possible**. The first such node is node 1 with distance 0. When a node is selected, the Algorithm goes through all edges that start at the node and reduces the distances using them:

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)

```
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 Staring 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

A Spanning Tree of a Graph consists of all nodes of the graph and some of the edges of the Graph so that there is a Path between any two nodes. Like Trees in general, Spanning Trees are connected and **acyclic**. Usually there are several ways to construct a spanning tree.
Acyclic Graph – a Graph having no Graph Cycles. A **connected** Acyclic Graph is known as a **Tree**, and a possibly **disconnected** Acyclic Graph is known as a **Forest**.

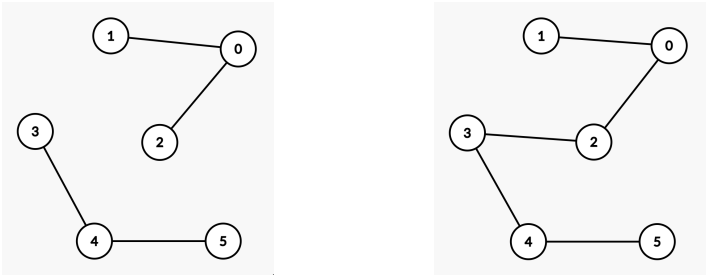


Figure: Acyclic Forest and Acyclic Tree

Minimum Spanning Tree

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

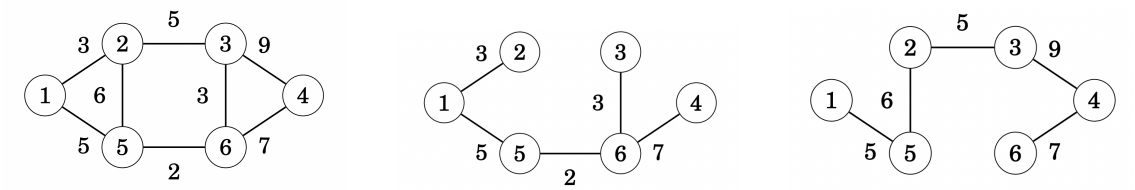


Figure: **Standard non-asyclic Graph** and **Minimum Spanning Tree** and **Maximum Spanning Tree**

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)

The first Algorithm to find Minimum Spanning Tree is known as **Kruskal's Algorithm**.
In Kruskal's Algorithm, the initial Spanning Tree only contains the nodes of the graph and **does not contain any edges**. Then the Algorithm goes through the edges **ordered by their weights**, and always adds an edge to the tree if **it does not create a cycle**.

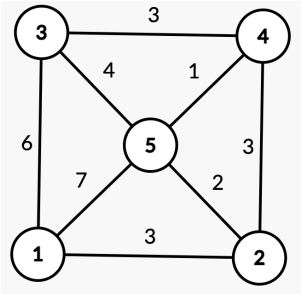


Figure: Example Non-Asyclic Graph

Steps of Kruskal's Algorithm (1)

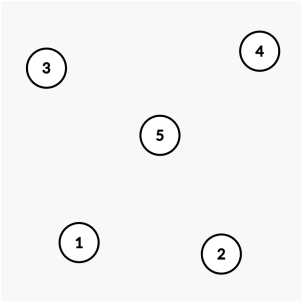
The First Step of the Algorithm is to **sort the edges in increasing order** of their weights. The result is the following list:

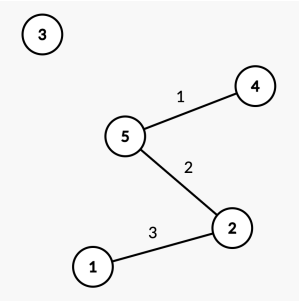
edge	weight
$4-5$	1
$2-5$	2
$1-2$	3
$2-4$	3
$3-4$	3
$3-5$	4
$1-3$	6
$1-5$	7

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.

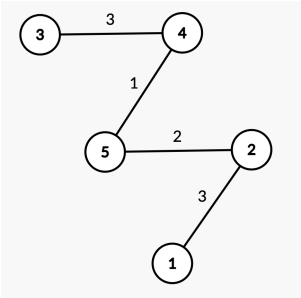


Figure: The weight of The Minimum Spanning Tree is $3 + 2 + 1 + 3 = 9$

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:

```
1 // loop through each node in Adjacency List
2 for (...) {
3 if (!same(a,b)) unite(a,b);
4 }
```

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

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

```
1  // initially each node are seperate set:
2  for (int i = 1; i <= n; i++) link[i] = i;
3
4  // because each node are seperate set, each set size is 1.
5  for (int i = 1; i <= n; i++) size[i] = 1;
6
7
8  // to find the parent of the set in which node x is.
9  int find(int x) {
10     while (x != link[x]) x = link[x];
11     return x;
12 }
13
14 // if the parents are same, then 2 node are in same group.
15 bool same(int a, int b) {
16     return find(a) == find(b);
17 }
```

Union Find Structure

To connect two separated nodes:

```
1 // to connect two disconnected nodes.
2 void unite(int a, int b) {
3     a = find(a); // to find the parent of the set in which node A is
4     b = find(b); // to find the parent of the set in which node B is
5
6     // add the smaller size set to larger one.
7     if (size[a] < size[b]) swap(a,b);
8         size[a] += size[b];
9
10    // to make the node A parent of node B.
11    link[b] = a;
12 }
```

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

Prim's Algorithm

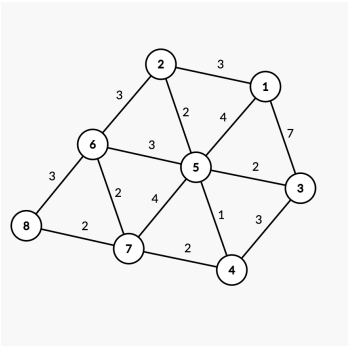
Prim's Algorithm is an alternative method for finding a Minimum Spanning Tree. The Algorithm first adds an **arbitrary** node to the Tree. After this, the Algorithm always chooses a **minimum-weight** edge that adds a new node to the Tree. Finally, all nodes have been added to the Tree and a Minimum Spanning Tree 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, because the **priority queue** is mostly used as a **data structure** to implement Prim's Algorithm. The Time Complexity of the Prim's Algorithm is **O(N + MlogM)**.

Example (Prim's Algorithm)

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



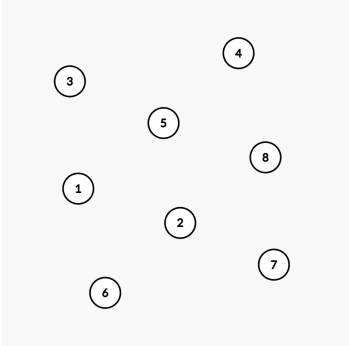
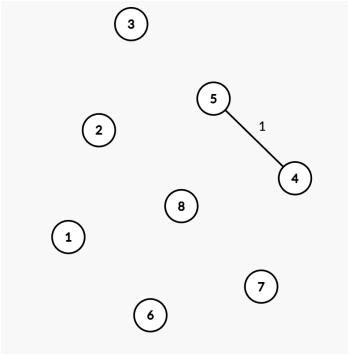
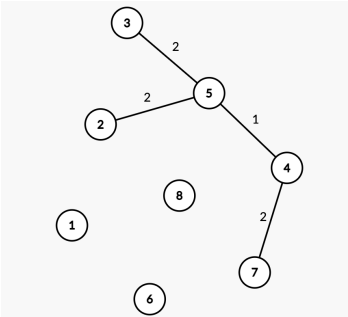


Figure: We can start from any node, let's say to start from node 5.

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.

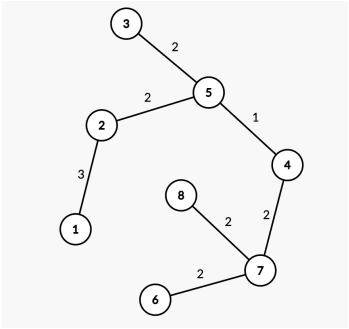


Figure: The weight of the Minimum Spanning Tree is $1 + 2 + 2 + 2 + 2 + 2 + 3 = 14$

The End

Thanks for your Attendance.

SEE YOU NEXT WEEK!