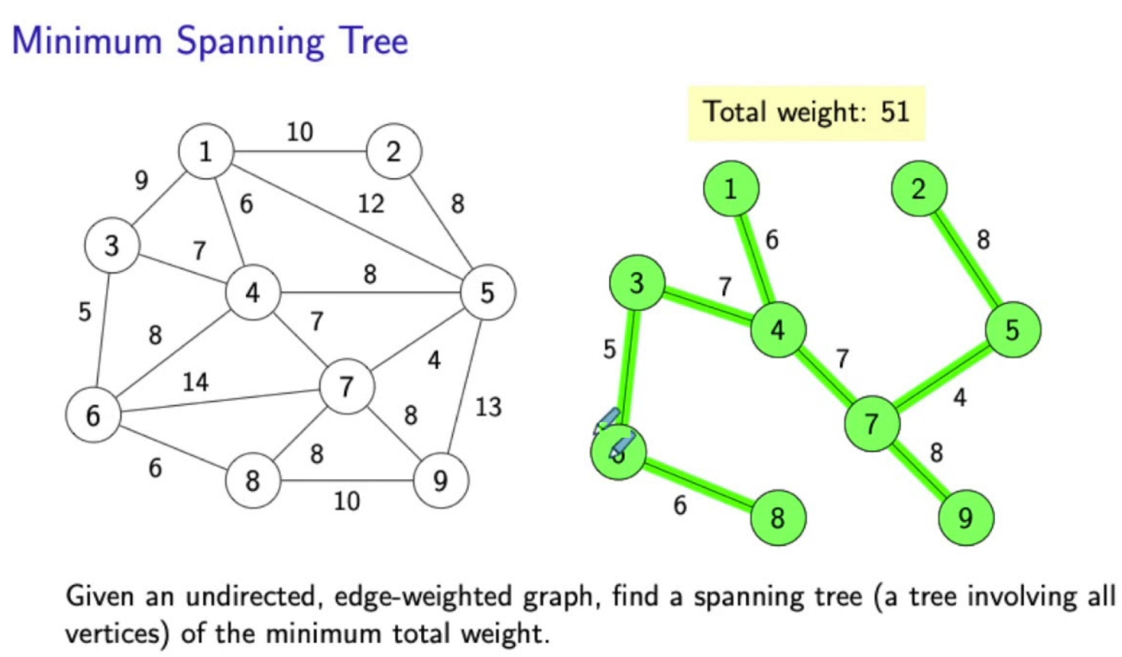
Lecture 18: Minimum Spanning Tree

Date: 10/18/2023

Problem Statement

Minimum Spanning Tree **(MST)** is a subset of the edges of a connected, edge-weighted graph that connects all the vertices together without any cycles and with the minimum possible total edge weight.

⇒ It is a way of finding the most economical way to connect a set of vertices.



reference: https://www.youtube.com/watch?v=r4jf5d4_7S4

Application

application	vertex	edge
circuit	component	wire
airline	airport	flight route
power distribution	power plant	transmission lines
image analysis	feature	proximity relationship

Typical MST applications

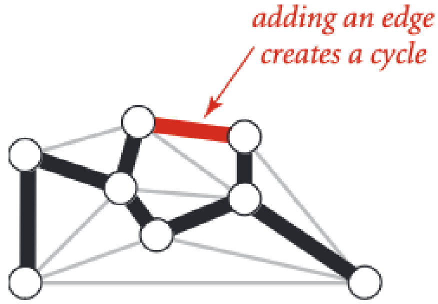
reference: Algorithms, 4e. (page 604)

Assumption

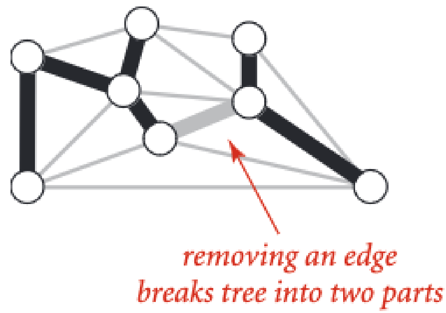
⇒ **The graph is connected.** (if the graph is not connected, then we can't create a tree which can span the whole graph.)
⇒ **The edge weights are not necessarily distances.** (to eliminate any geometrical intuitions.)
⇒ **The edge weights may be zero or negative.**
⇒ **The edge weights are unique.** (to make the proof easier)

Properties / Analysis

⇒ Adding an edge that connects two vertices in a tree creates a cycle.

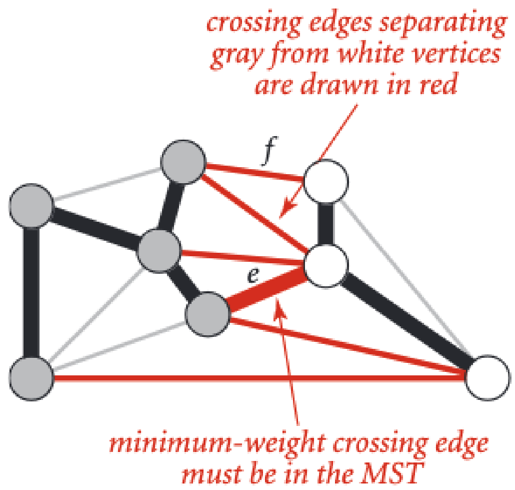


⇒ Removing an edge from a tree breaks it into two subtrees



⇒ Proposition: Given any cut in an edge-weighted graph, the crossing edge of minimum weight is in the MST of the graph.

Proof: Let e be the crossing edge of minimum weight and let T be the MST. The proof is by contradiction: Suppose that T does not contain e. Now consider the graph formed by adding e to T. This graph has a cycle that contains e, and that cycle must contain at least one other crossing edge, say, f, which has higher weight than e (since e is minimal and all edge weights are different). We can get a spanning tree of strictly lower weight by deleting f and adding e, contradicting the assumed minimality of T.



Cut property

(**Note:** **<u>Cut</u>** of a graph is partition of its vertices into two nonempty disjoint sets. A crossing edge of a cut is an edge that connects a vertex in one set with a vertex in another.)

Algorithms

Brute Force

The naive algorithm can create all the spanning trees and calculate the cost of those and pick the lowest one.

Pseudocode

```
function find_mst(graph, source){
    for tree in all_possible_trees for root as source{
        cost = calculate_cost(tree)
        if cost < min_cost{
            save_this_tree
        }
    }
}
```

Analysis

⇒ First we need to find out how many times the for loop is running for, which is the number of possible trees there are. That will be n^{n-2} (resource: https://www.geeksforgeeks.org/total-number-spanning-trees-graph)

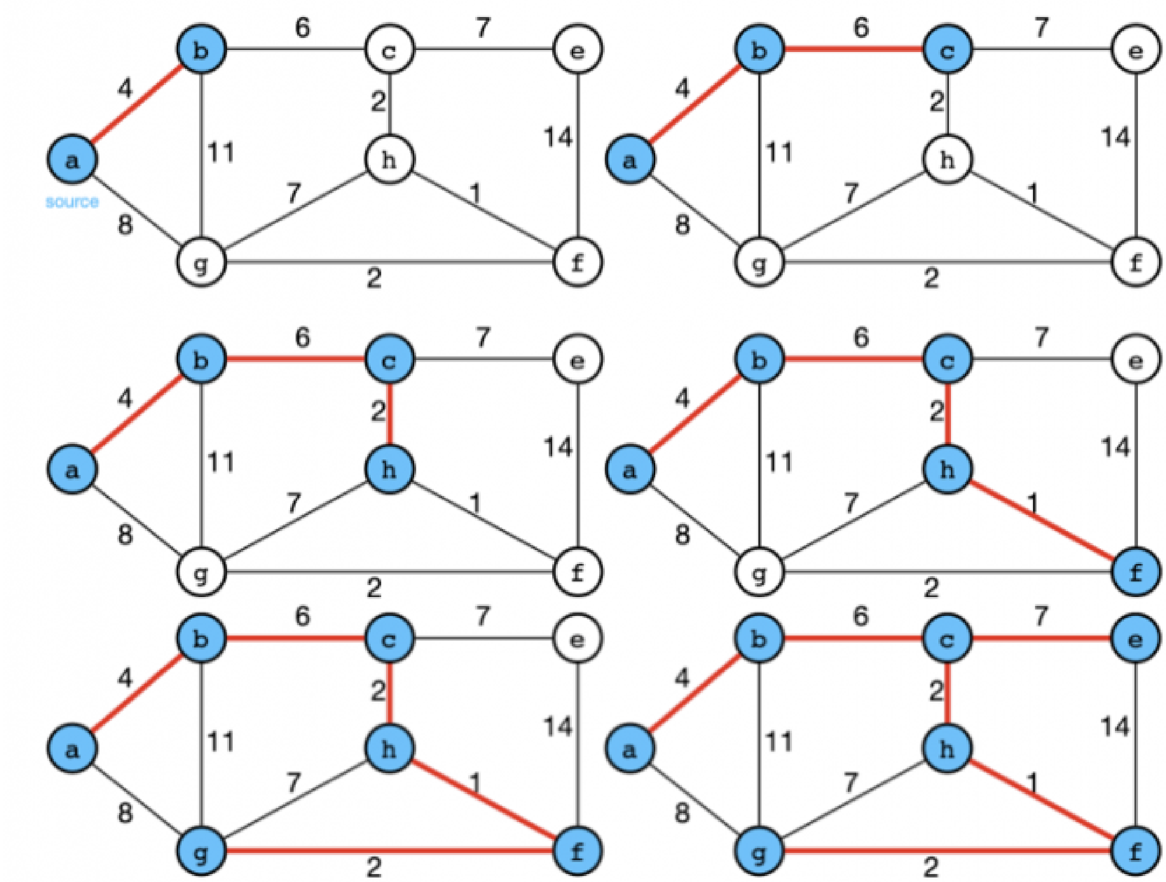
⇒ So our algorithm's time complexity will be roughly: $\Omega(n^n)$

Prim's

⇒ Idea: Based on the idea of cut property, we can make a greedy algorithm.
Apply the cut property to accept an edge into MST, and repeat this until we are done with generating MST.

Pseudocode

```
function find_mst(graph, source){
    current_cut = {source}
    mst = {}
    while not done{
        min_edge = find_min_crossing_edge(curr_cut)
        if min_edge is invalid: remove edge
        else: mst.add(min_edge)
    }
}
```



reference: https://medium.com/analytics-vidhya/minimum-spanning-tree-prim-3f32445ce854

Implementation

⇒ The one brute-force way we can find min_crossing_edge from current_cut is to find all the crossing edges and choose the minimum one. The time complexity of that way will be **O(E)**, as at some time, we might have all the edges as crossing edges.

⇒ Instead of that, we can use a Priority Queue to store all the edges and then find_min operation will just be **log(size of queue) = log(E)**

```
void find_mst(Graph g, Node* source){
      add_to_curr_cut(source);
      while(!pqEdges.empty()){
            edge_type min_edge = pqEdges.top(); pqEdges.pop();

            if(visited[min_edge.first] && visited[min_edge.second])
```

```
            continue;

            if(!visited[min_edge.first])
                  add_to_curr_cut(min_edge.first);
            if(!visited[min_edge.second])
                  add_to_curr_cut(min_edge.second);
      }
}
```

Analysis

⇒ we are finding the minimum edge from the priority queue inside the while loop. Which means time complexity will be number of times the while loop is running times log(E)

⇒ In the worst case, while loop might run for E times, as in each iteration we are removing one edge. So it might take E steps to empty the queue.

- Time Complexity: E*log(E)
- Space Complexity: E

Kruskal's [optional]