Minimum Spanning Trees

MST Definition

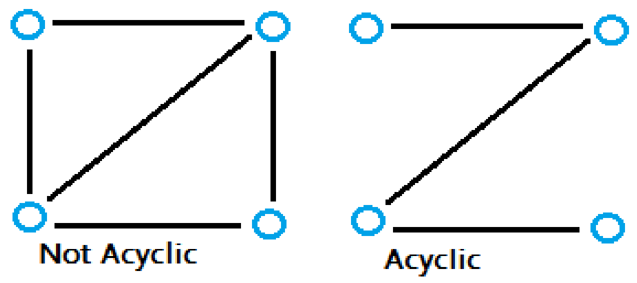
Definition 1: *An MST is a* ***minimum*** *weight* ***tree*** *that contains* ***all nodes*** *of an undirected graph.*

What do we mean by a tree in the context of graph theory?

A **tree** is an undirected graph in which any two vertices are connected by ***exactly*** *one* path. In other words, any acyclic connected graph is a tree.

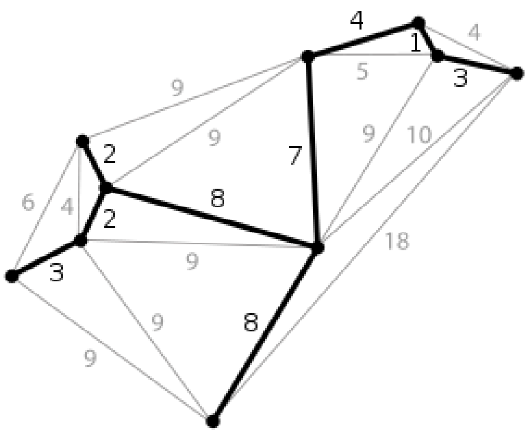
A **tree** always contains exactly one path to join any two vertices. A **spanning tree** of a graph is a tree that:

- Contains all the original graph's vertices.
- Reaches out to (spans) all vertices.
- Is acyclic. In other words, the graph doesn't have any nodes which loop back to itself.



Here's a more verbose definition from Wikipedia:

Definition 2: 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.



Note: for any connected graph of **n** nodes, the minimum spanning tree will contain **n** nodes and (**n-1**) edges.

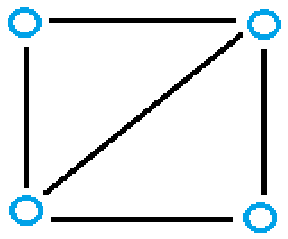
Examples of applications

Minimum spanning trees have direct applications in the design of networks, including computer networks, telecommunications networks, transportation networks, water supply networks, and electrical grids.

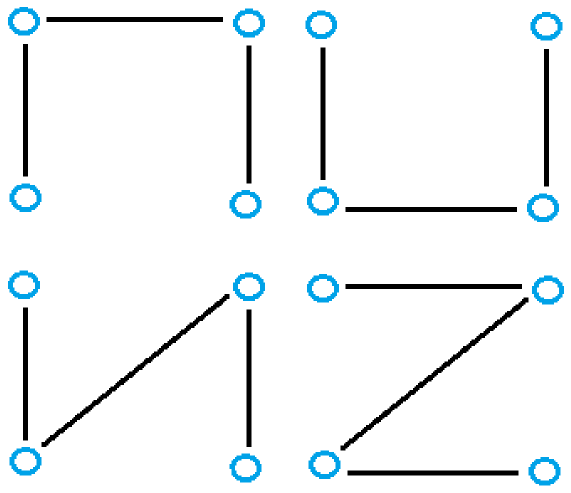
One example would be a telecommunications company laying cable to a new neighbourhood. If it is constrained to bury the cable only along certain paths (e.g. along roads), then there would be a graph representing which points could be connected by those paths. Some of those paths might be more expensive, because they are longer, or require the cable to be buried deeper; these paths would be represented by edges with larger weights. A spanning tree for that graph would be a subset of those paths that has no cycles but still connects to every house; there might be several spanning trees possible. A minimum spanning tree would be one with the lowest total cost, thus would represent the least expensive path for laying the cable.

Spanning Trees (general, not minimum)

Even the simplest of graphs can contain many spanning trees. For example, the following graph:



…has many possibilities for spanning trees, including:



While a graph can have many spanning trees, we are interested in the **minimum** spanning tree[1].

How could we go about finding this MST? Well, there are two popular algorithms: Kruskal's and Prim's. We'll look at the first one in this document.

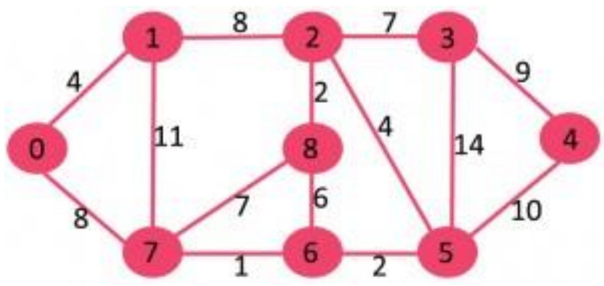
[1] That statement probably raises the question, "is there only one unique MST for any given graph?"
The answer is: It can be proven that there is only one unique MST for a graph where all edge weights are unique. If this is not the case then there exists the possibility of having more than one MST.

Kruskal's Algorithm

Much of this is taken from geeksforgeeks.org.

***1.** Sort all the edges in non-decreasing order of their weight.*
***2.** Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.*
***3.** Repeat step#2 until there are (V-1) edges in the spanning tree.*

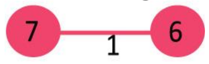
The algorithm is a Greedy Algorithm. The Greedy Choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far. Let us understand it with an example: Consider the below input graph.



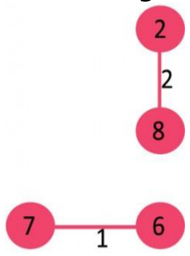
The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.

```
After sorting:

Weight    Src     Dest

1          7       6

2          8       2

2          6       5

4          0       1

4          2       5

6          8       6

7          2       3

7          7       8

8          0       7

8          1       2

9          3       4

10         5       4

11         1       7

14         3       5
```

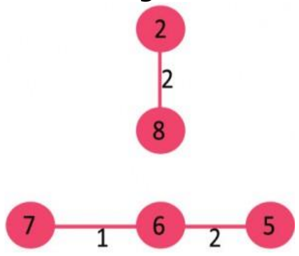
Now pick all edges one by one from sorted list of edges
1. *Pick edge 7-6:* No cycle is formed, include it.



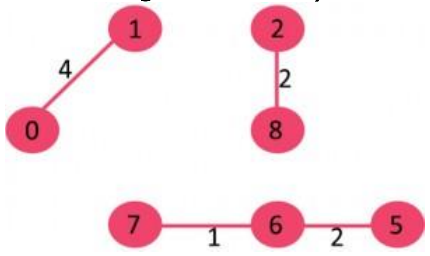
2. *Pick edge 8-2:* No cycle is formed, include it.



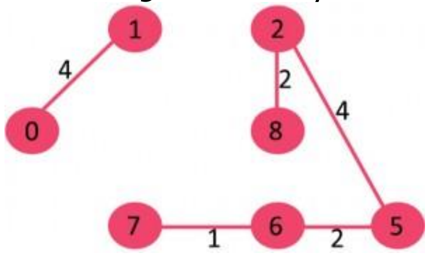
3. *Pick edge 6-5:* No cycle is formed, include it.



4. *Pick edge 0-1:* No cycle is formed, include it.

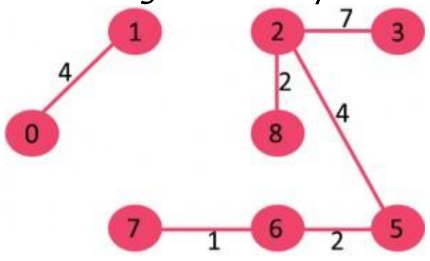


5. *Pick edge 2-5:* No cycle is formed, include it.



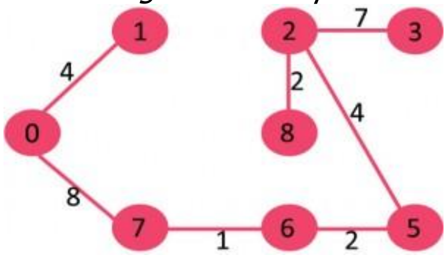
6. *Pick edge 8-6:* Since including this edge results in cycle, discard it.

7. *Pick edge 2-3:* No cycle is formed, include it.



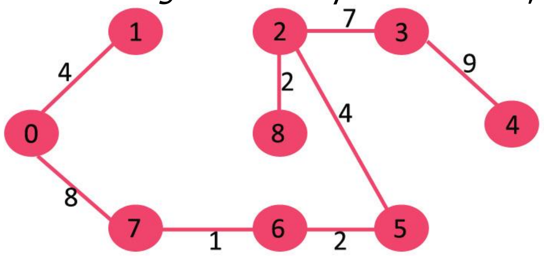
8. *Pick edge 7-8:* Since including this edge results in cycle, discard it.

9. *Pick edge 0-7:* No cycle is formed, include it.



10. *Pick edge 1-2:* Since including this edge results in cycle, discard it.

11. *Pick edge 3-4:* No cycle is formed, include it.



Since the number of edges included equals (V – 1), the algorithm stops here.

Detecting Cycles

Obviously, the ability to detect cycles is fundamental to the operation of the algorithm. How could we do this, though?

There are a number of mechanisms but the one which requires the least amount of explanation (and is the easiest to code since we already have the basic algorithm) is to use a Depth-First-Search.

Basically, we can check to see if the two nodes (associated with the edge that we are considering adding to our MST list of edges) are already connected, i.e. if there currently exists some path between those two nodes. Obviously, if this is the case, then adding this new edge will introduce a cycle.

So, we do a DFS traversal starting at the first node. As we traverse we check to see if we've visited the second node; if so, they're already connected. On the other hand, if the DFS traversal manages to finish and we haven't visited the second node, then we know they are dis-joint and we can safely add this new edge to our MST list.

Appendix A: General Graph Definitions

- **Graph** consists of
  - V = Set of **Vertices** (aka **nodes**)
  - E = Set of **Edges**
- **Edges**:
  - Edges connect 2 vertices
  - Specified by a pair

- **Weighted** Graph: each edge has a (positive) weight
- **Directed** graphs:
  - Edges have direction
  - For example: distinguish between [(A,B) and (B,A)]
  - Represent direction with arrowhead
- **Undirected** graphs:
  - Edges have no direction
  - For example: do NOT distinguish between [(A,B) and (B,A)]
  - Edge has NO arrowhead
- **Path** between two nodes
  - Path between two nodes: sequence of nodes and edges
  - Begins and ends with a node
  - Each edge connects the node preceding and following it
  - In a directed graph, a path must follow arrows

- **Connected** graph: a path exists between every pair of nodes
- **Unconnected** graph: Some pair of nodes has no path between them
- **Cycle**: path that begins and ends at same node

- **Cyclic** graph: graph that contains a cycle
- **Acyclic** graph: graph that contains NO cycles