Minimum Spanning Tree

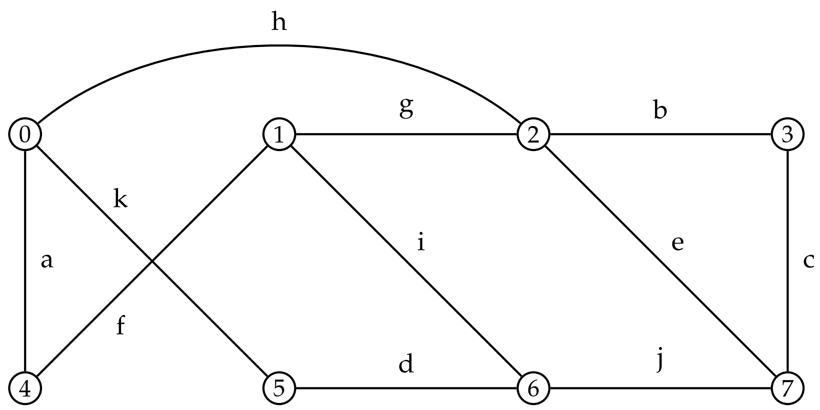
Salvador Roura

Let $G = (V, E)$ be an undirected and connected graph with n vertices and m edges. A spanning tree T of G is $T = (V, E')$, where $E' \subseteq E$, and

- T has $n - 1$ edges,
- T has no cycles,
- T is connected.

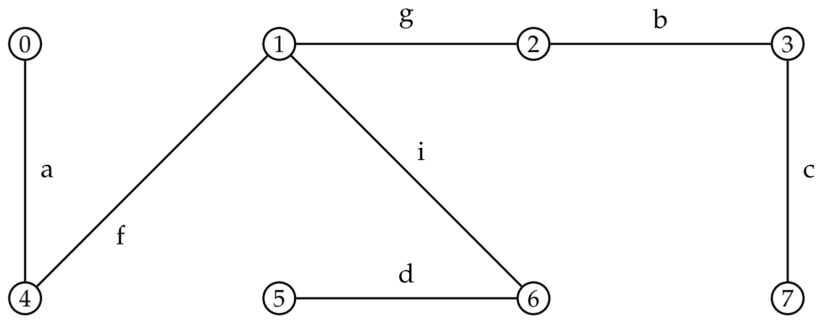
In fact, any two of the above conditions implies the third one. Informally speaking, a spanning tree of a graph is a way to choose a subset of edges such that the graph remains connected and no edge is superfluous.

For instance, consider this graph with $n = 8$ and $m = 11$.



This graph has many possible spanning trees. Perhaps the simplest way to find one of them is the following: Start with a T with the n vertices and no edges. So, initially, we have a forest with n subtrees (each one with just one vertex). Afterwards, consider each edge e one by one. If e is useful, that is, if it connects two different subtrees, we add e to T; otherwise, we discard e. Note that adding an edge joins two subtrees into one. At the end, T has exactly one subtree: a spanning tree.

The result only depends on the order of the edges. Therefore, if we consider the edges in a different order, we may end up with a different spanning tree. For instance, this is the result if we consider the edges in alphabetical order:



Observe that we start by adding the edges a, b, c and d, we discard e (2 and 7 already belong to the same subtree), next we add f and g, we discard h (0 and 2 already belong to the same subtree), and finally we add i. At this moment (after $n - 1 = 7$ edges have been added to T) we can stop, because when all the vertices are transitively connected, the rest of edges are useless.

Note that the algorithm above uses the first two properties for a spanning tree: $n - 1$ edges and no cycles. The nontrivial part is to be able to efficiently tell if the two vertices of a given edge already belong to the same subtree or not (in the later case the edge would create a cycle). Fortunately, the merge-find set (a.k.a. union-find algorithm, a.k.a. disjoint-set data structure) does the trick. There are many variants of merge-find sets. Here, we use path compression, which is simple to program and the most efficient in many practical situations.

Suppose that an edge is a pair of vertices, numbered from 0 to $n - 1$:

```
struct Edge {
  int x, y;
};
```

For convenience, let *parent* be a global *vector*<**int**> that stores the parent of each vertex, or a -1 if the vertex is the root of its subtree (and therefore it is the representative of its class). Recursion allows us to easily implement path compression:

```
// returns the representative of the class of x
int repre(int x) {
  if (parent[x] == −1) return x;
  int res = repre(parent[x]);
  parent[x] = res;
  return res;
}
```

We can even program the same function in just one line:

```
int repre(int x) {
  return (parent[x] == −1 ? x : parent[x] = repre(parent[x]));
}
```

The following procedure returns a spanning tree of a graph G (given as a vector of edges) with n vertices.

```
vector<Edge> spanning_tree(int n, const vector<Edge>& G) {
  vector<Edge> T;
  parent = vector<int>(n, −1);
  for (int i = 0; n > 1 and i < G.size(); ++i) {
    int rx = repre(G[i].x);
    int ry = repre(G[i].y);
    if (rx ≠ ry) {
      parent[ry] = rx;
      T.push_back(G[i]);
      −−n;
    }
  }
  return T;
}
```

The condition $n > 1$ is optional. It only makes the code faster in general, when a spanning tree is already found, and we do not need to consider more edges. The worst-case cost in any case is $\Theta(n+m)$—in fact slightly larger because *repre*(x) does not have constant cost (in theory).

So far, we have implicitly assumed that every edge has the same cost (say 1). But what if they could have different non-negative costs? Can we adapt the above ideas to compute a spanning tree of minimum cost?

The answer is Kruskal's algorithm. Before we consider the edges one by one, we sort them by cost in non-decreasing order. No more changes are required. It can be proven that this greedy approach provides a minimum spanning tree. If we sort the edges efficiently (for instance, using the C++ *sort* () procedure), Kruskal's algorithm has cost $\Theta(m \log m)$.