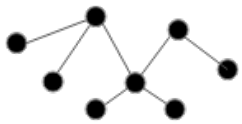
Spanning Trees

We will be working with an undirected graph G with n nodes and e edges. In the graph to the right, n is 8 and e is 7.



G is called a *tree* if, as shown, each pair of nodes is connected by a unique simple path. What's the root of the tree? It doesn't matter. Choose any node you want as the root.

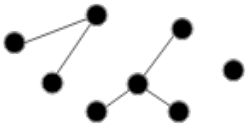
There are several different conditions for G to be a tree. Some are listed below. Study each one and verify for yourself that it defines a tree. Use the tree above to the right to help you.

1. Each pair of nodes is connected by a unique simple path.
2. G is connected and `e = n-1`.
3. G is acyclic and `e = n-1`.
4. G is connected and acyclic.
5. G is connected and removing any edge makes it unconnected.
6. G is acyclic and adding any edge results in a cycle.

Take a look at the properties in definitions 2, 3, and 4: G is connected, G is acyclic, and `e = n-1`. If any two of them is true, the graph is a tree and the third property is also true. That's neat.

Forests

A forest is an undirected graph all of whose connected components are trees. The graph at the top of the page is a forest of one tree. To the right is a forest of three trees.

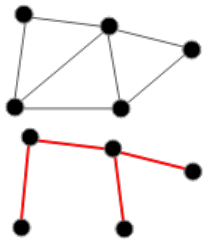


Put another way: a *forest* is a set of trees in which no two trees have a node in common. We say that the forest is the disjoint union of the trees.

Here is one more equivalent definition of a forest: A *forest* is an undirected acyclic graph. The number of trees in the graph is the number of connected components in the graph.

Spanning trees

We introduced trees because we wanted to talk about *spanning trees*. A spanning tree of a connected graph G is a tree G′ (1) whose nodes are those of G and (2) whose edges are a subset of those of G. Note: Since G′ has to be a tree, it is connected.



Let G be the graph of 5 nodes that is to the right of the previous paragraph. To the right of *this* paragraph is a spanning tree G′, with its edges colored red. You can verify that G′ is indeed a tree.

The spanning tree of a connected graph is generally not unique. In fact, the 5-node graph G above has at least 10 spanning trees —how many can you find? We encourage you to draw some of them.

Two other definitions of a spanning tree lead to algorithms to find them

We can define a spanning tree G′ of G in the following two ways.

G′ is a spanning tree of G if:	G′ is a spanning tree of G if
• V′ = V • E′ is a maximal subset of edges of E that contains no cycle.	• V′ = V • E′ is a minimal subset of edges of E that connect all nodes.

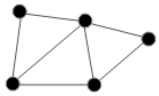
Each definition inspires an algorithm for constructing a spanning tree. Here are the main ideas.

To find a maximal set of items, start with all of them and delete one at a time, deleting as few as possible. That is a *subtractive method*. It will be used with the definition on the left.

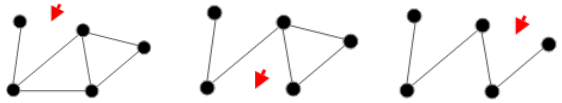
On the other hand, to find a minimal set of items, start with nothing and add items, adding as few as possible. That is an *additive method*. It will be used with the definition on the right.

1. Subtractive method. Since we want a maximal subset of edges that contains no cycle, we construct a spanning tree using the subtractive method.

Start with G′ containing all nodes and all edges of G, as shown to the right, and iteratively delete an edge of a cycle until no cycle remains. Deleting an edge of a cycle leaves the graph connected (why?). Here's the abstract algorithm:



```
// Store in G′ a spanning tree of G.
G′ = G;
while (G′ contains a cycle) {
    Find a cycle and delete one edge of the cycle;
}
```



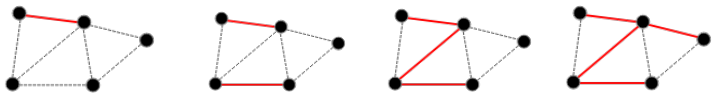
To the right, we show the three iterations of this algorithm. In each iteration, we point with a red arrow to where the deleted edge was. It is obvious that the deleted edge was part of a cycle.

How many edges must be removed in the worst case —how many iterations will this algorithm make? If G is complete, it has $n(n-1)/2 = n^2/2 - n/2$ edges. A spanning tree has $n-1$ edges. Therefore $n^2/2 - n/2 - (n-1)$ edges have to be removed. $O(n^2)$ edges have to be removed! The denser G is, the worse this algorithm looks. We don't want to use this algorithm. Let's hope there is a better way.

2. Additive method. Since we want a minimal set of edges that connect all nodes, we construct a spanning tree using the additive method.

Start with G′ containing all nodes of G and no edges. We show the initial G′ to the right. At this point, G′ consists of 5 connected components, each one being one node. The dashed edges are there only to show what edges might be added. Here's the abstract algorithm:

```
// Store in G′ a spanning tree of G;
G′ = all nodes of G and no edges;
while (G′ is not connected) {
    Add an edge that connects two
        unconnected components;
}
```



To the right, we show the four iterations of this algorithm. In each iteration, the added edge is red. After 1 iteration, there are 4 connected components, after 2 iterations, 3 connected components, and so on.

How many iterations will this additive algorithm perform? A tree with n nodes has $n-1$ edges. Therefore, exactly $n-1$ iterations will be performed. Always. That's much better than with the subtractive method! In another pdf file, we look more closely at implementing the additive method.

Undetermined specifications and nondeterministic algorithms

The spec of these algorithms says simply to "Store in G′; a spanning tree of G." The spec doesn't say which spanning tree to store in G′. The word *underdetermined* means not having enough constraints to specify a unique solution. Therefore, we say that the specification is *underdetermined*. If it were not underdetermined, it would say which spanning tree to construct.

This *underdetermined* specification is a good thing, for it allows the implementer freedom, which may be useful in developing an efficient algorithm.

The algorithms themselves are *nondeterministic*, which means that running them twice with the same input can result in different outputs. How is this possible? Consider the additive method. At each iteration, it adds an edge that connects two unconnected components. There often are several such edges, and it doesn't say which one to add. Consequently, during runtime, any choice will do.

This *nondeterminism* is a good thing because it gives the implementer freedom, which may be useful in developing an efficient algorithm.

Some programs are inherently nondeterministic, for example because they rely on timing of events inside the computer, which is not always the same. This is a topic for another discussion.