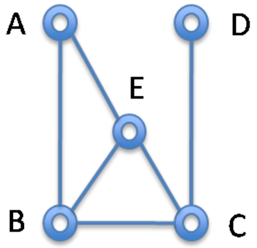
Lecture 25 Notes
Spanning Trees

15-122: Principles of Imperative Computation (Spring 2016)
Frank Pfenning

1 Introduction

The following is a simple example of a connected, undirected graph with 5 vertices (A, B, C, D, E) and 6 edges (AB, BC, CD, AE, BE, CE).



In this lecture we are particularly interested in the problem of computing a *spanning tree* for a connected graph. What is a tree here? They are a bit different than the binary search trees we considered early. One simple definition is that a *tree* is a *connected graph with no simple cycles*, where a simple cycle is a path that lets you go from a node to itself without repeating an edge. A *spanning tree* for a connected graph G is a tree containing all the vertices of G and a subset of the edges of G.

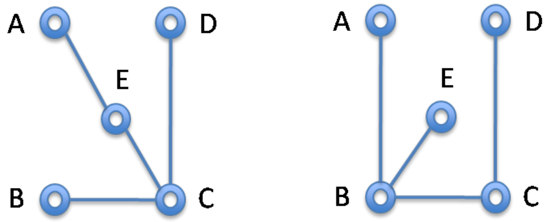
The corresponding learning goals are as follows:

Computational Thinking: We continue our introduction to graphs by defining spanning trees and minimum spanning trees for graphs with weighted edges.

Algorithms and Data Structures: We examine two ways to compute a spanning tree, and introduce Kruskal's algorithm, a classical method for calculating a minimum spanning tree.

Programming: We leave the implementation of these algorithms as exercises to the reader.

Below are two spanning trees for our original example graph — there are more.



When dealing with a new kind of data structure, it is a good strategy to try to think of as many different characterizations as we can. This is somewhat similar to the problem of coming up with good representations of the data; different ones may be appropriate for different purposes. Here are some alternative characterizations the class came up with:

1. Connected graph with no cycle (original).

2. Connected graph where no two neighbors are otherwise connected. *Neighbors* are vertices connected directly by an edge, *otherwise connected* means connected without the connecting edge.

3. Two trees connected by a single edge. This is a recursive characterization. The base case is a single node, with the empty tree (no vertices) as a possible special case.

4. A vertex connected to a tree by a single edge. The base case is again a single vertex. This is another recursive characterization.

5. A connected graph with exactly $v - 1$ edges, where v is the number of vertices.

6. A graph with exactly one path between any two distinct vertices, where a path is a sequence of distinct vertices where each is connected to the next by an edge. (For paths in a tree to be distinct, we have to disallow paths that double back on themselves).

When considering the asymptotic complexity of our various algorithms on graphs, it is often useful to categorize graphs as *dense* or *sparse*. Dense graphs have a lot of edges compared to the number of vertices. Writing $v = |V|$ for the number of vertices (which will be our notation in the rest of the lecture) we can see that the number of edges can be at most $v \times (v-1)/2$: each node could be connected to any other node ($v \times (v-1)$), but in an undirected way ($v \times (v-1)/2$). If we write e for the number of edges, we have $e \in O(v^2)$. By comparison, a tree is *sparse* because $e = v - 1 \in O(v)$.

2 Computing a Spanning Tree

There are many algorithms to compute a spanning tree for a connected graph. The first is an example of a *vertex-centric* algorithm.

1. Pick an arbitrary node and mark it as being *in the tree*.
2. Repeat until all nodes are marked as *in the tree*:

   > Pick an arbitrary node u in the tree with an edge e to a node w not in the tree. Add e to the spanning tree and mark w as in the tree.

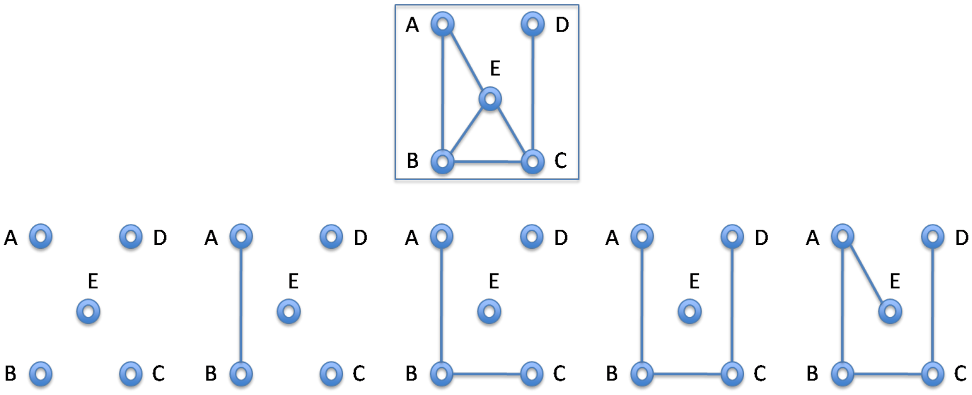
We iterate $v - 1$ times in Step 2, because there are $v - 1$ vertices that have to be added to the tree. The efficiency of the algorithm is determined by how efficiently we can find a qualifying w.

The second algorithm is *edge-centric*.

1. Start with the collection of singleton trees, each with exactly one node.
2. As long as we have more than one tree, connect two trees together with an edge in the graph.

This second algorithm also performs v steps, because it has to add $v - 1$ edges to the trees until we have a spanning tree. Its efficiency is determined by how quickly we can tell if an edge would connect two trees or would connect two nodes already in the same tree, a question we come back to in the next lecture.

Let's try this algorithm on our first graph, considering edges in the listed order: (AB, BC, CD, AE, BE, CE).



The given graph is highlighted on top. The completely disconnected graph on the left is the starting point for this algorithm. At the far right, we have computed the spanning tree, which we know because we have added $v - 1 = 4$ edges. If we tried to continue, the next edge BE could not be added because it does not connect two trees, and neither can CE. The spanning tree is complete.

3 Creating a Random Maze

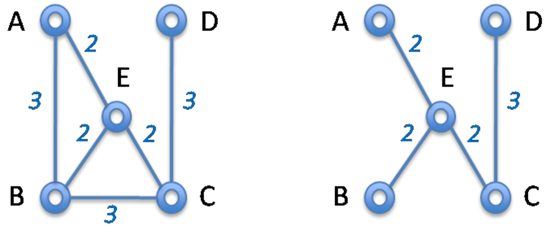
We can use the algorithm to compute a spanning tree for creating a random maze. We start with the graph where the vertices are the cells and the edges represent the neighbors we can move to in the maze. In the graph, all potential neighbors are connected. A spanning tree will be defined by a subset of the edges in which all cells in the maze are still connected by some (unique) path. Because a spanning tree connects all cells, we can arbitrarily decide on the starting point and end point after we have computed it.

How would we ensure that the maze is random? The idea is to generate a random permutation (see Exercise 1) of the edges and then consider the edges in the fixed order. Each edge is either added (if it connects two disconnected parts of the maze) or not (if the two vertices are already connected). But, of course, we need an efficient way to determine if the two vertices are already connected. We could use one of the search methods from yesterday's lecture, but as we'll see tomorrow, we can do better.

4 Minimum Weight Spanning Trees

In many applications of graphs, there is some measure associated with the edges. For example, when the vertices are locations then the edge weights could be distances. We might then be interested in not any spanning tree, but one whose total edge weight is minimal among all the possible spanning trees, a so-called *minimum weight spanning tree* (MST). An MST is not necessarily unique. For example, all the edge weights could be identical in which case any spanning tree will be minimal.

We annotate the edges in our running example with edge weights as shown on the left below. On the right is the minimum weight spanning tree, which has weight 9.



Before we develop a refinement of our edge-centric algorithm for spanning trees to take edge weights into account, we discuss a basic property it is based on.

Cycle Property. Let C be a simple cycle in graph G, and e be an edge of maximal weight in C. Then there is some MST of G that does not contain e.

How do we convince ourselves of this property? Assume we have a minimum spanning tree T, and edge e from the cycle property connects vertices u and w. If e is not in T, then, indeed, we don't need it. If e is in T, we will construct another spanning tree without e of weight less than or equal to T's weight. Removing edge e splits T into two subtrees. There must be another edge e' from C that is not in T which also connects the two subtrees. Removing e and adding e' instead yields another spanning tree, T', which does not contain e. T' has equal or lower weight to T, since e' must have weight less than or equal to e.

The cycle property is the basis for *Kruskal's algorithm*.

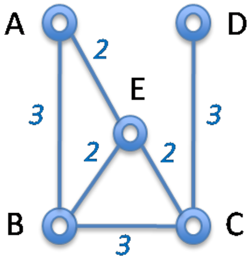
1. Sort all edges in increasing weight order.

2. Consider the edges in order. If the edge does not create a cycle, add it to the spanning tree. Otherwise discard it. Stop when $n-1$ edges have been added, because then we must have a spanning tree.

Why does this create a minimum-weight spanning tree? It is a straightforward application of the cycle property (see Exercise 2).

Sorting the edges will take $O(e \log e)$ steps with most appropriate sorting algorithms. The complexity of the second part of the algorithm depends on how efficiently we can check if adding an edge will create a cycle or not. As we will see in the next lecture, this can be $O(n \log n)$ or even more efficient if we use a so-called *union-find* data structure.

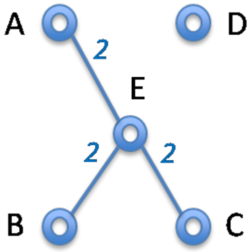
Illustrating the algorithm on our example



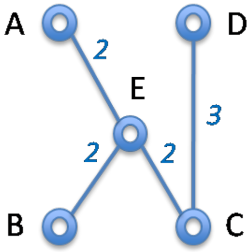
we first sort the edges. There is some ambiguity—say we obtain the following list

AE	2
BE	2
CE	2
BC	3
CD	3
AB	3

We now add the edges in order, making sure we do not create a cycle. After AE, BE, CE, we have



At this point we consider BC. However, this edge would create a cycle BCE since it connects two vertices in the same tree instead of two different trees. We therefore do not add it to the spanning tree. Next we consider CD, which does connect two trees. At this point we have a minimum spanning tree



We do *not* consider the last edge, AB, because we have already added $n - 1 = 4$ edges.

In the next lecture we will analyze the problem of incrementally adding edges to a tree in a way that allows us to quickly determine if an edge would create a cycle.

Exercises

Exercise 1. *Write a function to generate a random permutation of a given array, using a random number generator with the interface in the standard* `rand` *library. What is the asymptotic complexity of your function?*

Exercise 2. *Prove that the cycle property implies the correctness of Kruskal's algorithm.*