Teacher's Guide for *Spanning and weighted spanning trees: a different kind of optimization*

by sarah-marie belcastro

1 The Math.

Let's talk about spanning trees. No, actually, first let's talk about *graph theory*, the area of mathematics within which the topic of spanning trees lies.

1.1 Graph Theory Background.

Informally, a *graph* is a collection of vertices (that look like dots) and edges (that look like curves), where each edge joins two vertices. Formally, A *graph* is a pair $G = (V, E)$, where V is a set of dots and E is a set of pairs of vertices. Here are a few examples of graphs, in Figure 1:

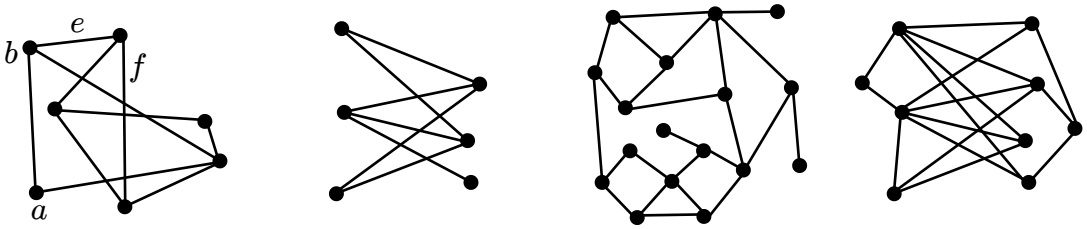

Figure 1: Examples of graphs.

Note that the word *vertex* is singular; its plural is *vertices*. Two vertices that are joined by an edge are called *adjacent*. For example, the vertices labeled a and b in the leftmost graph of Figure 1 are adjacent. Two edges that meet at a vertex are called *incident*. For example, the edges labeled e and f in the leftmost graph of Figure 1 are incident.

A *subgraph* is a graph that is contained within another graph. For example, in Figure 1 the second graph is a subgraph of the fourth graph. You can see this at left in Figure 2 where the subgraph in question is emphasized.

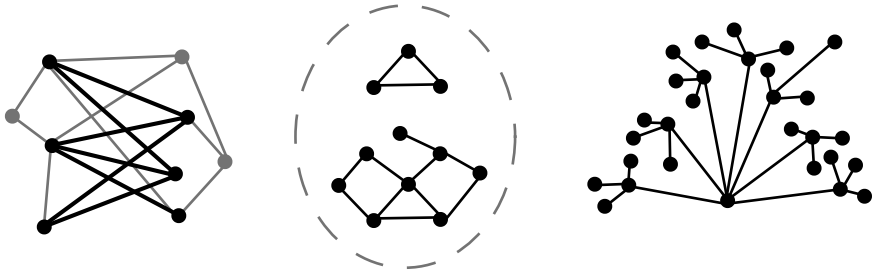

Figure 2: More examples of graphs.

In a *connected* graph, there is a way to get from any vertex to any other vertex without leaving the graph. The second graph of Figure 2 is *not* connected.

A *cycle* is a sequence that alternates between vertices and edges, and whose only repetition is the first/last vertex. There are cycles in all the graphs of Figure 1, and a cycle is shown by itself as the top part of the second graph of Figure 2.

A *tree* is a graph that is connected and has no cycles. One is shown to the right in Figure 2.

A *forest* is a bunch of trees.

An *algorithm* is a finite list of unambiguous instructions. Basically an algorithm tells you how to accomplish some task.

Pertinent facts about trees:

1. If a tree has n vertices, then it has $n - 1$ edges. This can be proved most easily by induction on the number of vertices; the simplest way is to remove a vertex and produce two smaller trees on which the inductive hypothesis holds.

2. Trees are the connected graphs with the least number of edges. This can be proved most easily by contradiction (plus the previous pertinent fact).

1.2 Spanning Trees.

A *spanning tree* is a tree that contains all the vertices of a given graph. Basically, it is the largest tree contained in a graph. The top of Figure 3 shows two graphs, and the bottom of Figure 3 shows two spanning trees for each top graph. There are generally lots of spanning trees

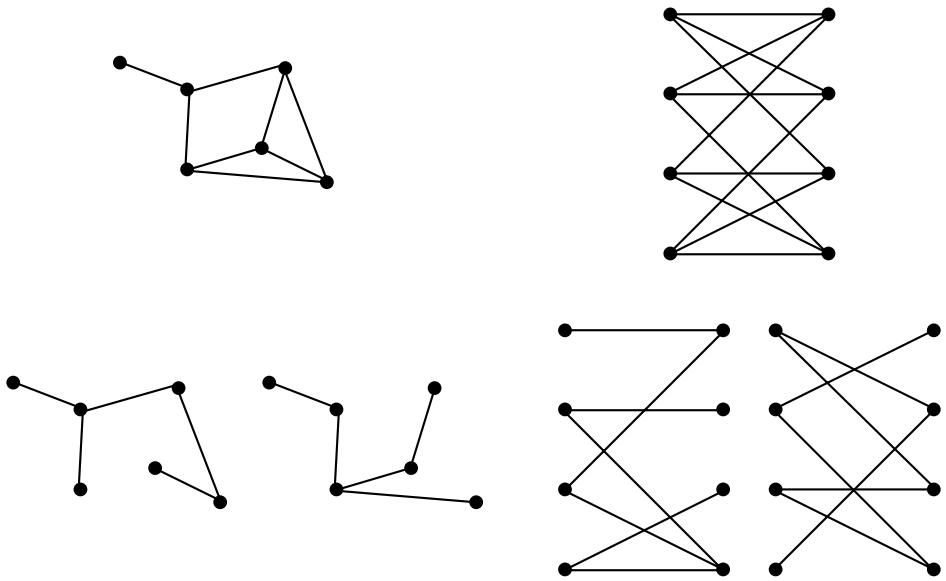

Figure 3: Two spanning trees for each of two graphs.

for a particular graph. We care about them because they are used all the time in computer science. For example, doing a search on a graph requires a spanning tree. So, it is important to have algorithms for finding spanning trees.

Here are a couple of super-informal algorithms for finding a spanning tree in a graph:

1. (Start Big) Find a cycle; remove an edge from that cycle. Repeat until the graph has no more cycles.

2. (Start Small) Hide all the edges. Replace them one at a time, always choosing to replace an edge that is incident to some other replaced edge and does not complete any cycle. Continue until all vertices are included in the resulting tree.

There are lots of different levels of formality one can use when discussing algorithms. Here's the first algorithm, but with notation introduced.
Finding a spanning tree: Start Big

1. Begin with a connected graph G.

2. Consider a duplicate of G and name it H.

3. Pick an edge, any edge, of H and call it e.

4. If $H \setminus e$ is connected, remove e and rename $H \setminus e$ as H; otherwise, mark e as necessary (and don't consider it again).

5. Pick any edge of H not marked as necessary and call it e. If there are no unmarked edges left, rename H as T and be done.

6. Go to step 4.

This is still not all that formal—for example, how exactly do you tell a computer to pick an edge of a graph? A graph must be stored as some sort of computer-readable structure in order for a computer to perform any operations on it. That structure usually includes an ordering of the edges of a graph. Additionally, testing $H \setminus e$ for connectedness requires a separate algorithm of its own, but that also depends on computer-science details.

Proving that these algorithms work is fairly straightforward.

1. (Start Big) We begin with a connected graph and at each stage retain connectedness; the result has no cycles, as otherwise there would be an edge not marked as necessary. Thus, the algorithm produces a tree. Additionally, the tree must be spanning because we began with all vertices and took no action that would remove one from the tree (because at each stage the resulting graph is connected).

2. (Start Small) We need to be sure that the resulting graph T is connected, has no cycles, and includes all the vertices of G. Because at each stage we only consider edges incident to the growing tree H, each intermediate graph is connected; and, because at each stage we only add edges that retain tree-ness, no cycles are created. Thus, T is a tree. Suppose, however, that there is a vertex v of G not included in T. It must be incident only to edges e_i marked as superfluous (as otherwise it would have been added to T at some stage), and therefore $T \cup e_i$ is not a tree for any i. This means that $T \cup e_i$ must contain a cycle (as e_i is incident to T, we know $T \cup e_i$ is connected), and therefore $v \in T$.

1.3 Weighted Spanning Trees.

Weights are labels on the edges and/or vertices of a graph that often denote costs or distances or energies.

The *total weight* of a spanning tree is the sum of the weights on its edges.

A *minimum-weight* spanning tree is one that has the lowest possible total weight. Figure 4 shows a weighted graph at left, and two weighted spanning trees at right; the rightmost tree is a minimum-weight spanning tree.

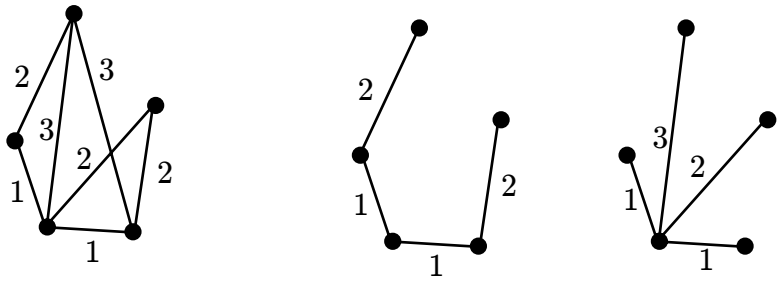

Figure 4: A weighted graph (left) with spanning trees of total weight 6 (middle) and 7 (right).

A weighted graph can represent any number of practical situations. It could be a network of oil wells with distances-by-road between them marked. It could be a network of computers with edges where two computers actually can be connected (e.g. no steel wall between them) and needed cable lengths marked. It could be towns with costs for various routes that use toll plazas marked. The possibilities are close to endless.

Here's where optimization comes in: finding a minimum-weight spanning tree also finds the cheapest ways to lay pipe for oil or install cable or travel around an area with tolls. (This is not the same as finding shortest distances between points or to travel an area—those are different problems with different solutions.)

Here are the most commonly-used algorithms for finding minimum-weight spanning trees.

Kruskal's algorithm for finding a minimum-weight spanning tree:

1. Begin with a connected edge-weighted graph G. Order the edges in increasing order of weight as $e_1, \ldots, e_n$.
2. Let e_1 along with its vertices be called H and set $j = 2$.
3. If $H \cup e_j$ has no cycles, then rename $H \cup e_j$ as H; otherwise, do nothing.
4. If $|E(H)| = |V(G)| - 1$, output H as the desired tree; otherwise, do nothing.
5. Increment j by 1 and go to step 3.

It's straightforward to show that this algorithm produces a spanning tree, but technically challenging to prove that the tree produced has minimum weight. The structure of this proof is to proceed by contradiction; assume there is some spanning tree with smaller total weight, and find the first place in the algorithm where the algorithm-tree and lighter-tree diverge. Use this to produce a contradiction.

Prim's algorithm for finding a minimum-weight spanning tree:

1. Let G be a connected edge-weighted graph. Find the edges of least weight in G and pick one of them. Name it ℓ.

2. Let ℓ along with its vertices be called P.

3. Look at all the edges of G that have exactly one vertex in P. Among these, pick one of least weight and name it e. If $P \cup e$ has no cycles, then add e and its other vertex to P, and rename $P \cup e$ as P. Otherwise, mark e as *bad* so you don't look at it again.

4. If $V(P) = V(G)$, then we're done and output P. Otherwise, go to step 3.

Again, it's straightforward to show that this algorithm produces a spanning tree. As with Kruskal's algorithm, proving that Prim's algorithm produces a minimum-weight spanning tree is also challenging and also proceeds by contradiction. Here we consider a minimum-weight spanning tree with the largest number of edges in common with the tree produced by Prim's algorithm, and consider the first edge Prim produces that differs from the minimum-weight tree. Use this (and a lot of technical manipulation) to produce a contradiction.

One might also come up with an analogue of the Start Big algorithm, but it turns out that this would be inefficient in practice.

1.4 There's More.

So far we've just discussed what spanning and weighted spanning trees are and algorithms for finding them. Directions for further investigation include the full proofs that these algorithms actually produce spanning and minimum-weight spanning trees, and how the algorithms might be coded or implemented effectively and efficiently.

There is more to study in the area of trees, both in theory and in applications. There are more graph-theoretic approaches to optimization; for example, shortest-path algorithms are used in finding driving routes in Google (and other) Maps applications. A related area is that of network flows.

And of course there is the study of graph theory as a whole!

There are tons of resources out there for learning more about these topics, both on the web (just google and you'll find some) and in book form. My favorite starting resource is (unsurprisingly) a book I wrote called *Discrete Mathematics with Ducks*, and it contains pointers to more resources.

2 The Activity/Presentation.

2.1 The Materials at Your Disposal.

Available to you are

- this document!
- a one-minute intro video
- presentation slides

- a definition half-sheet
- two worksheets
- a supplemental worksheet

Presumably you won't use all of these, or at least not all at once.

2.2 Potential Lesson Plans.

2.2.1 What I Do.

Generally I start with a synopsis to the students of what will happen (basically the following description, but abbreviated).

I give a short introduction to graph theory, live at the board rather than with projected slides. This is just a bunch of definitions and examples of the terms defined, and while I prepare a list of definitions ahead of time, I usually generate the examples on the fly. Then I break the students into groups—usually I let them self-select, because I don't know students in one-shot enrichment situations well—and hand out copies of the worksheet. I ask the students to work together on the first worksheet. (You may want to copy the two worksheets so that they are front and back of a single sheet.) The definitions and examples stay on the board so students can refer to them.

While the students are working, I circulate among the groups. I listen to what students are saying and read what they are writing. Sometimes students will ask questions; sometimes I ask them to tell me about their discussions. If I hear or read something that sounds incomplete or incorrect, I ask the students to think about it more, with questions such as "What about (this piece of the definition)?" or "Does that take care of (this aspect of an argument)? Tell me more."

After most groups have made significant progress, and/or if most groups seem to be stuck in the same place, I call the students back together for a large-group discussion. I ask students to present the results their groups have developed, and give feedback about correctness and how much of the question(s) have been answered. Depending on how much they have done, I will either release them to continue working, or (more likely) set them loose on the second worksheet—and then Lather, Rinse, Repeat until we're nearly out of time. If there are groups that are very ahead of the game, I give them the supplemental worksheet to think about while others finish the second worksheet. Possibly all groups will be given the supplemental worksheet!

At the end, I try to bring some closure to the activity with one last large-group summary discussion, preferably including an introduction to the supplemental worksheet. The last thing to say is that there's a lot more to learn about this material and graph theory in general, and students can/should look for discrete math classes in college (or self-study using my textbook *Discrete Mathematics with Ducks*).

2.2.2 What You Might Do.

Here are some variants that may work well for you:

Start by showing the one-minute intro video in order to get the students' attention.

Present the definitions and examples using projected slides. This can be done quickly because students generally pick up on elementary graph theory quickly. Or you can ask students to come to the board and contribute additional examples of each term defined (a different student for each definition to maximize participation). This will take more time, but students will also internalize the definitions and concepts more deeply; still, students will make less progress on the worksheets by the end of the class.

You could choose to have students work very briefly on constructing proofs (or skip the proofs entirely) in order to make time for the supplemental worksheet on weighted spanning trees. Or you could devote another partial class period in order to give the students time to address this material.

In general, there is a tradeoff between student work on explaining and proving and progress through the worksheets, or one might say between deep exploration/understanding of the material and overview of content. The more time students spend on trying to prove statements, the fewer questions they will address in total.

2.3 How Things Might Go.

Students are likely to utter non-words such as "vertice" (and less likely to say "vertexes"). They should be corrected promptly (and repeatedly).

If students aren't familiar with induction, then they will produce many unconvincing arguments for why a tree with n vertices has $n-1$ edges. Also, if they know induction they're likely to try to add another vertex onto an existing tree, which loses generality; it's not clear that every $(n+1)$-vertex tree comes from the tree they started with, and there are lots of ways to build up a tree. Try to keep them from spinning their wheels too long on this.

Sometimes students think that they are supposed to be working on weighted spanning trees even when no weights are present. Be aware of this possibility.