#31: Graph Theory II

May 30, 2009

Last week, you learned the basic definition of a *graph*. But there are several ways that this basic definition can be usefully generalized in order to model various sorts of structures. This week we'll look at several of these generalizations.

1 Weighted graphs and minimum spanning trees

weighted graphs

A *weighted*, or *labelled* graph is one where each edge has some kind of *weight* or *label* associated with it. This is useful when you want to represent not only the fact that things are connected in certain ways, but that there is some sort of cost, length, capacity, or other sort of measurement associated with each connection. Let's look at an example.

*ooooo**OOOOO**ooo*

It is the year 2055. Humans have established several colonies on Mars. Figure 1 shows a graph representing the colonies (labeled a through h) and the roads between them. (There used to be a road between b and c, but as you can see it has been taken over by Martians.)

Problem 1. Is it possible to walk along every road exactly once without retracing your steps or walking along the same road twice? If not, can you do it if you leave out one road? If you leave out two roads?

Problem 2. Is it possible to walk along the roads in such a way that you visit every colony exactly once?

zoom!

The humans decide to link some of the colonies with special high-speed trains, which need to be built along existing roads. However, high-speed trains are really expensive. The number next to each road shows the cost of turning that road into a high-speed train (in zillions of zlqrts, the currency used by the colonists).

Problem 3. Suppose the colonists want to link a and f with high-speed trains. Which roads should be turned into trains to link a and f at minimum cost? What is the minimum cost?

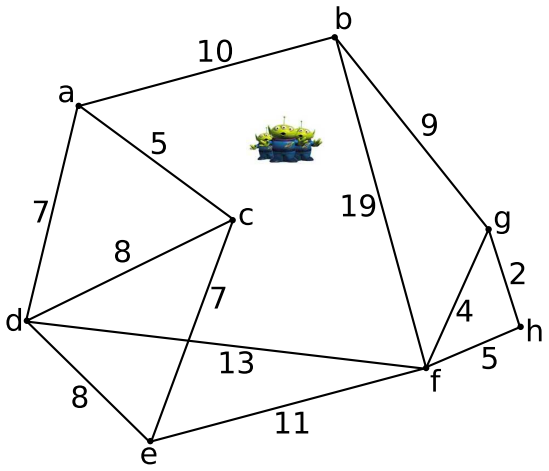

Figure 1: Mars colonies, with the cost of turning each road into a high-speed train

As it turns out, the humans decide they want to link *all* the colonies with high-speed trains; that is, it should be possible to get from any colony to any other just by riding trains. However, building a high-speed train along *every* road would cost too much money, not to mention that it would be unnecessary. For example, there would be no need to build a high-speed train from, say, a to d if there were already high-speed trains from a to c and c to d: anyone wanting to get from a to d could first take the train to c and from there to d.

minimum spanning trees

So, the problem is this: which roads should be turned into trains so that all the colonies are connected for the *least total amount of money*? Such a subset of edges in a weighted graph which connects all the vertices and has minimum total weight is called a *minimum spanning tree*.

Kruskal's algorithm

It turns out that there is a simple algorithm for finding minimum spanning trees in a weighted graph, called *Kruskal's algorithm*. It works by building up the minimal spanning tree one edge at a time:

- Look at the edges in order of weight, from smallest to largest.
- For each edge, if it would form a cycle with other edges already in the minimum spanning tree, throw it out. Otherwise, add it to the tree.

Let's see how this works, using Figure 1 as an example. First, we look at the edge with the smallest weight, which is the edge from g to h (weight

2). It doesn't make a cycle with any of the edges already in the minimum spanning tree (there *aren't* any other edges in the minimum spanning tree yet!) so we add it to the minimum spanning tree, which I've indicated in Figure 2 by coloring it red.

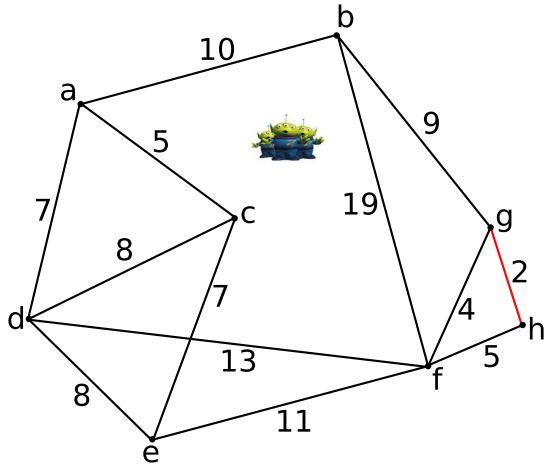

Figure 2: The first step of Kruskal's algorithm. The minimum spanning tree so far is colored red.

Now we look at the edge with the next smallest weight, which is the edge from g to f with weight 4. Again, there are no cycles in sight yet, so we add it to the minimum spanning tree (Figure 3).

The next smallest edge is the one from f to h, with weight 5. (Actually, there are two edges with weight 5; we are free to consider them in either order we like.) However, edges fg and gh are already in the minimum spanning tree, so adding fh would create the cycle fgh. We don't need to build a train from f to h if there are already trains from f to g and g to h! So we throw away edge fh (shown in Figure 4 by drawing it with a dotted line).

I'll show one more step: the next smallest edge is ac; it wouldn't form any cycles, so we add it (Figure 5).

Problem 4. Complete Kruskal's algorithm for the given graph. Which other roads should be turned into high-speed trains? What is the total (minimum) cost of connecting all the colonies with trains?

the claw has chosen!

Problem 5. It turns out that the Martians are actually friendly, and offer to help build a high-speed train from b to c. With the aliens' help, this

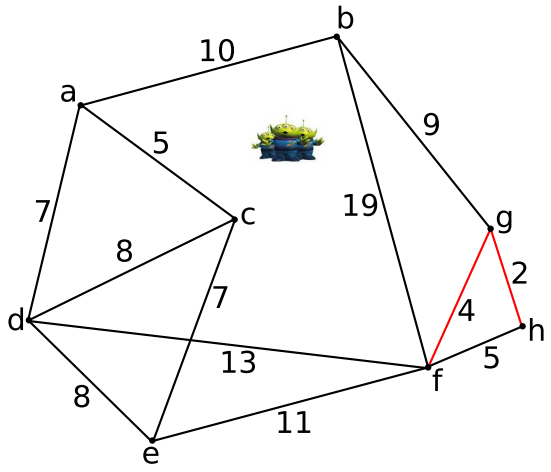

Figure 3: The second step of Kruskal's algorithm

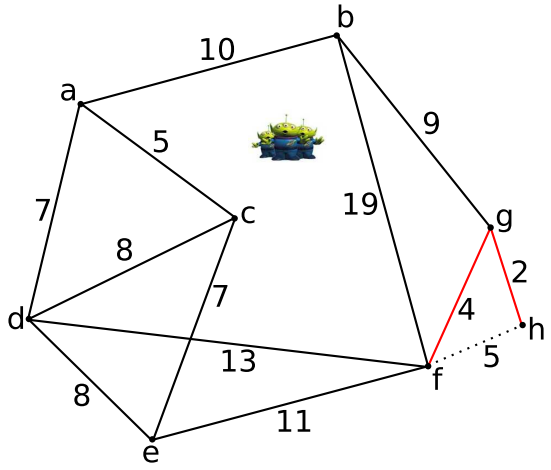

Figure 4: The third step of Kruskal's algorithm

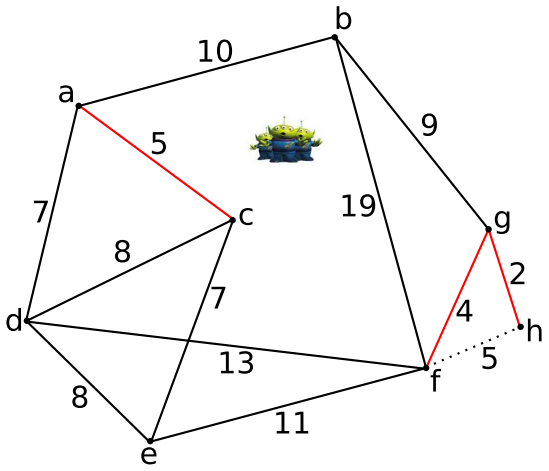

Figure 5: The fourth step of Kruskal's algorithm

would only cost one zillion zlqrts. How does the minimum spanning tree change? What is the new minimum total cost to connect all the colonies?

2 Directed graphs

directional edges

In all the graphs we've seen so far, the edges have been *undirected*—that is, they just connect two vertices, with no inherent directionality. For some applications, however, the *direction* of the edges matters. For example, suppose we are making a graph to represent the streets in downtown DC, with a vertex for each intersection. As anyone who has driven around in downtown DC knows, any map of downtown DC is useless if it doesn't show you *which roads are one-way*! So the *direction* of the connections matters.

In a *directed graph*, each edge has a *direction*, which we indicate in drawings of the graph by putting an arrowhead on one end. Note that *every* edge has a direction—if we want to indicate a two-way connection, we just draw two edges, one going in each direction.

write your own adventure

Figure 6 shows the plot outline of a choose-your-own adventure story, where 1 is the beginning of the story, and 8 and 9 are the endings.

Problem 6. Please write a story corresponding to the given plot outline. Here is an example of the beginning of such a story:

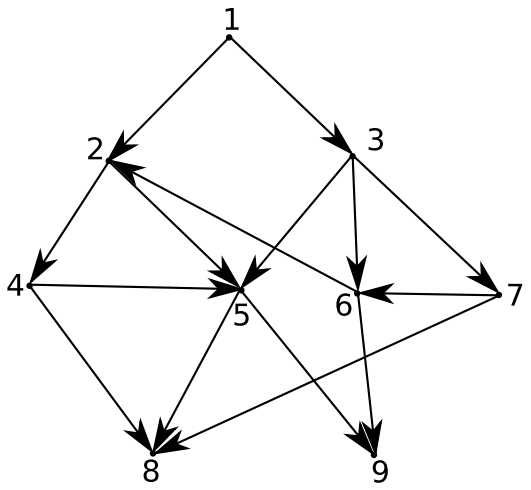

Figure 6: The plot thickens

1. It is a lazy Saturday afternoon, and you are bored. You suddenly have a great idea! If you decide to skateboard down the middle of the highway wearing a pretzel costume, go to 2. If you decide to fill your swimming pool with blue Gatorade, go to 3.

This corresponds to the vertex labeled 1 on the graph since there are edges going from 1 to 2 and 3. You can use this as #1 if you want, or you can make up your own.

DAGs

The graph in Figure 6 is called a *directed acyclic graph*, or DAG, since there are no cycles in the graph: there is no way to start at some node, follow edges in the proper directions, and end up back at the node where you started. (Obviously there *would* be cycles if the edges were undirected, but you are not allowed to go backwards along the edges.)

topological sort

Problem 7. List the vertices in order so that whenever there is an edge from vertex m to vertex n, m comes before n in the list of vertices. For example, 1, 2, 3, 6, ... would *not* be a correct order, since there is an edge from 6 to 2, so 6 has to come before 2 in the list.

(Note: such a list is called a *topological sort*, and it turns out that it is always possible to topologically sort a DAG.)

Problem 8. How many different possible stories are there? In other words, how many different paths are there from vertex 1 to either vertex 8 or vertex

9? (*Hint*: consider the vertices in order of your answer to the previous problem; for each vertex, figure out how many paths there are to it from vertex 1.)

2.1 Directed graphs and degree

in- and outdegree

In an undirected graph, the *degree* of each vertex was just the number of edges connected to it. But in a directed graph, this isn't enough: it matters whether each edge is coming *into* or going *out of* the vertex. So, in a directed graph, we talk about two different degrees: the *indegree* and the *outdegree*. The indegree of a vertex is the number of edges pointing *towards* the vertex (coming *in*) and the outdegree is the number of edges pointing *away from* the vertex (going *out*).

Problem 9. List the indegree and outdegree of each vertex in the graph in Figure 6.

Eulerian cycles in directed graphs

Problem 10. A directed graph has an Eulerian cycle if every vertex has equal indegree and outdegree—every time you come in, you have to be able to go out. (Note, the indegree and outdegree of each individual vertex have to be the same as each other; this doesn't mean they have to be equal to the indegree and outdegree of other vertices.) Draw a directed graph which does have an Eulerian cycle when considered as an undirected graph (that is, by ignoring all the arrows) but does *not* have an Eulerian cycle when considered as a directed graph.

3 Self-loops

Another way we can classify different types of graphs is by asking whether *self-loops* are allowed. A self-loop is an edge from a vertex to itself. For example, the graph in Figure 7 has two edges, one from A to B and the other from B to itself.

Figure 7: A graph with a self-loop

state transition diagrams

When might self-loops be useful? One common situation is if a graph (often a directed graph) is used to represent transitions in some sort of system; a self-loop would indicate that the system stays in the same state. For example, the directed, weighted graph in Figure 8 is a (completely made up) model of a nuclear reactor.

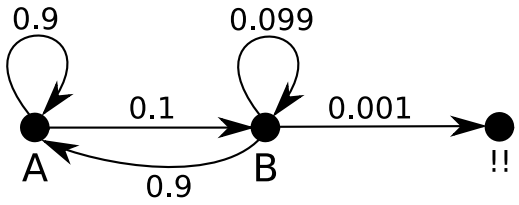

Figure 8: The state model of a nuclear reactor

State A represents normal operation; state B is a danger state in which there is excess energy in the system; and state !! represents a meltdown/explosion/general catastrophic failure. The edges represent transitions between the states, and are labeled with the probability of each transition in any given minute. For example, if the reactor is in state A, then nine out of ten times, on average, it will still be in state A after one minute (since the self-loop from A to A is labeled with 0.9); one time out of ten, on average, it will be in state B after one minute. From state B, after one minute there is a 90% chance the reactor will be back in state A, a 9.9% chance it will still be in state B, and an 0.1% chance that it will blow up.

Problem 11. If the reactor is in state A, what is the probability that it will blow up after...

(a) exactly one minute?

(b) exactly two minutes?

(c) exactly three minutes?

Hint: part (c) is a bit tricky. Start by finding all the paths of length 3 from A to !!.

The information in Figure 8 doesn't have to be represented by a graph—for example, it could be represented in a table as well—but a graph is a nice visual way to represent the information.

