Graph Theory III

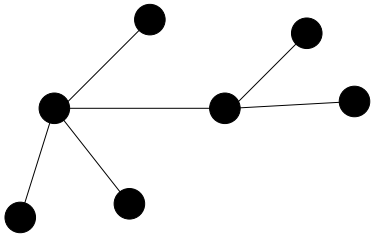
Draft: please check back in a couple of days for a modified version of these notes.

1 Trees

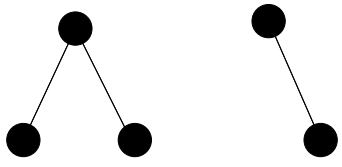
Today we'll talk about a very special class of graphs called *trees*. Trees arise in all sorts of applications and you'll see them in just about every computer science class that you'll take at MIT. There are at least half a dozen ways to define a tree, but the simplest is the following.

Definition. *A* tree *is a simple*[1]*, connected, acyclic graph.*

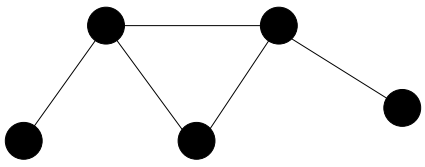
For example, a tree might look like this.



On the other hand, this is not a tree

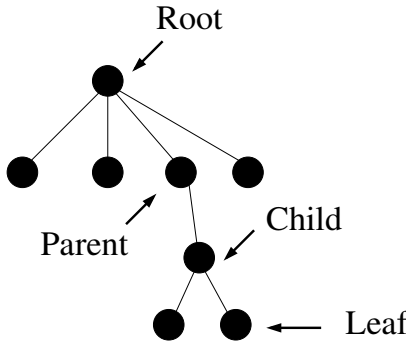


and neither is this



[1]Recall that we only consider simple graphs in this class, that is, graphs without loops or multiple edges.

Sometimes we'll draw trees in a leveled fashion, in which case we can identify the top node as the root, and every edge joints a "parent" to a "child".



The nodes at the bottom of degree 1 are called leaves.

Definition. *A* leaf *is a node in a tree with degree* 1*.*

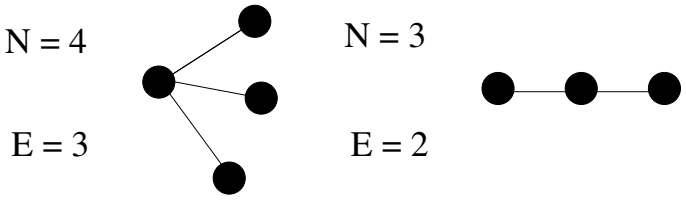
For example, in the tree above there are 5 leaves. It turns out that no matter how we draw a tree, every tree with more than 1 node always has some leaves.

Theorem 1. *Every tree with more than* 1 *node has at least one leaf.*

Proof. We prove the theorem by contradiction. Suppose it is not true. Then there exists a tree $T = (V, E)$ where every node has degree at least 2. Let $P = v_1, v_2, \ldots, v_m$ be a path of maximum length in T, i.e., there is no path with $m + 1$ or more nodes in T (recall that all nodes in a path are distinct by definition). Note that we are implicitly using the well-ordering principle here.

Since $|V| \geq 2$ and T is connected, such a P exists and we have $2 \leq m \leq n$. Since T has minimum degree at least 2 by assumption, v_m is joined to at least 2 nodes. In particular, there is a node $x \neq v_{m-1}$ such that $\{v_m, x\} \in E$. Since P has maximum length, $v_1, v_2, \ldots, v_m, x$ is not a path, which means $x = v_i$ for some $1 \leq i < m - 1$. But then $v_i, v_{i+1}, \ldots, v_{m-1}, v_m, v_i$ is a cycle in T. But T is a tree, and so by definition contains no cycle. This is a contradiction. □

As it turns out, every tree has at least 2 leaves, which you'll prove in the problem sets. There is also a close correlation between the number of nodes and number of edges in a tree. In the previous example we saw that $N = 7$ and $E = 6$. We also have the following examples.

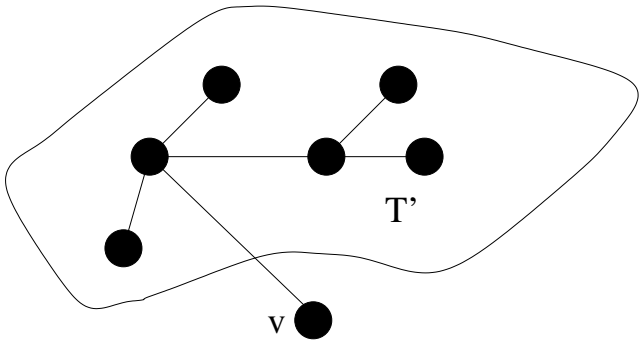


Naturally this leads us to guess,

Theorem 2. *For any tree $T = (V, E)$, $|E| = |V| - 1$.*

Proof. We prove the theorem by induction on the number of nodes N. Our inductive hypothesis $P(N)$ is that every N-node tree has exactly $N - 1$ edges. For the base case, i.e., to show $P(1)$, we just note that every 1 node graph has no edges. Now assume that $P(N)$ holds for some $N \geq 1$, and let's show that $P(N + 1)$ holds.

Consider any $(N + 1)$-node tree T. Let v be a leaf of T. We know by the previous theorem that such a leaf v exists. Let $T' = (V', E')$ be the tree formed by removing v and its incident edge from T.



We first claim that T' is in fact a tree. Since T is connected, so is T' since we have only removed a leaf. Moreover, removing vertices or edges cannot create any new cycles, so T' is also acyclic. Thus T' is a tree. By our inductive hypothesis, it follows that $|E'| = |V'| - 1$.

Now, we formed E' by removing one edge of E. Thus $|E'| = |E| - 1$. Similarly, $|V'| = |V| - 1$. Since $|E'| = |V'| - 1$, this means

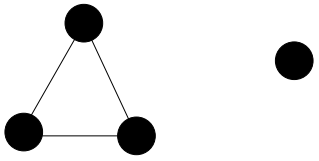
$$|E| - 1 = |V| - 1 - 1,$$

or $|E| = |V| - 1$, which shows $P(N + 1)$, and completes the proof. □

In fact, in the problems sets you will show the converse:

Theorem 3. *Any connected, N-node graph with $N - 1$ edges is a tree.*

Note that we need to assume the graph is connected, as otherwise the following graph would be a counterexample.

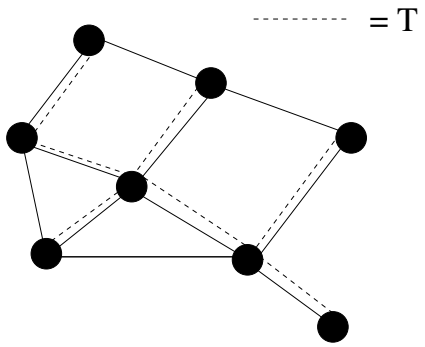


Besides this theorem, there are many other ways to characterize a tree, though we won't cover them here.

Graphs typically contain lots of trees as subgraphs. Of special interest are trees that contain every node of the graph. Such tree are called *spanning trees.*

Definition. *A tree* $T = (V, E)$ *is a* spanning tree *for a graph* $G = (V', E')$ *if* $V = V'$ *and* $E \subseteq E'$.

The following figure shows a spanning tree T inside of a graph G.

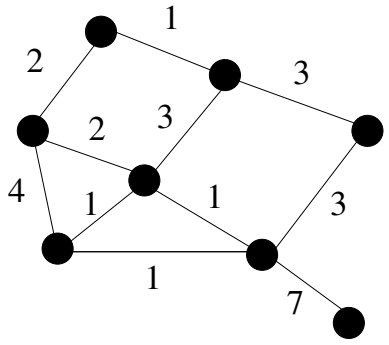


Spanning trees are interesting because they connect all the nodes of a graph using the smallest possible number of edges. For example, in the graph above there are 7 edges in the spanning tree, while there are 8 vertices in the graph. It is not hard to show that any connected graph contains a spanning tree, and often lots of them.

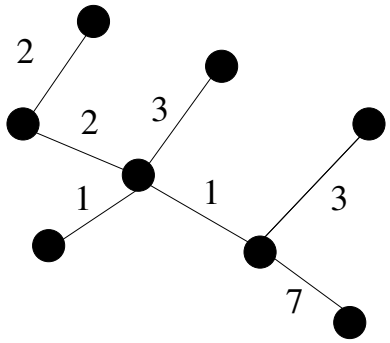
Spanning trees are very useful in practice, but in the real world, not all spanning trees are equally desirable. That's because, in practice, there are often costs associated with the edges of the graph.

For example, suppose the nodes of a graph represent buildings or towns and edges represent connections between buildings or towns. The cost to actually make a connection may vary a lot from one pair of buildings or towns to another. The cost might depend on distance or topography. For example, the cost to connect LA to NY might be much higher than that to connect NY to Boston. Or the cost of a pipe through Manhatten might be more than the cost of a pipe through a cornfield.

In any case, we typically represent the cost to connect pairs of nodes with a *weighted* edge, where the weight of the edge is its cost.



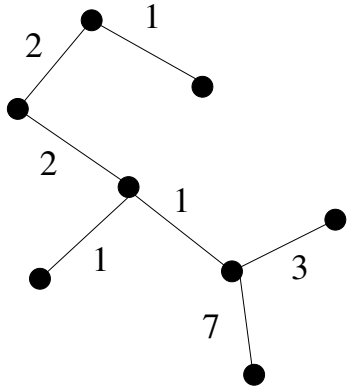
The weight of a spanning tree is then just the sum of the weights of the edges in the tree. For example, the weight of the following spanning tree is 19.



The goal, of course, is to find the spanning tree with minimum weight, called the min-weight spanning tree (MST) for short.

Definition. *The* min-weight spanning tree *(MST) of an edge-weighted graph G is the spanning tree of G with the smallest possible sum of edge weights.*

Is the spanning tree above an MST of the weighted graph we presented? Actually, it is not, since the following tree is also a spanning tree of our graph, of cost only 17.



Now is this spanning tree an MST? It seems to be, but how do we prove it? In general, how do we find an MST? We could, of course, enumerate all trees, but this could take forever for very large graphs.

Here are two possible algorithms:

ALG1:

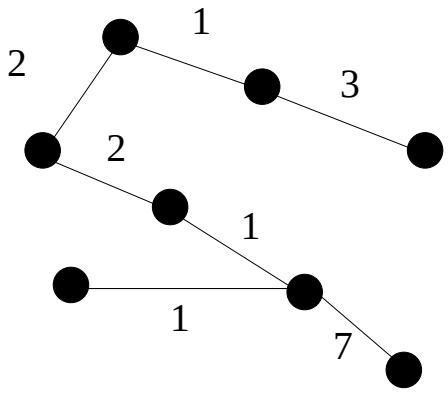
1. Grow a tree one edge at a time by adding the minimum weight edge of the graph to the tree, making sure that you have a tree at each step.

ALG2:

1. Select edges one at a time, always choosing the minimum weight edge that does not create a cycle with previously selected edges. We do not impose the constraint that the graph is a tree - our selected edges may indeed form a disconnected graph.

2. Continue until we get $N - 1$ edges, i.e., a spanning tree.

For example, in the weighted graph we have been considering, we might run ALG1 as follows. We would start by choosing one of the weight 1 edges, since this is the smallest weight in the graph. Suppose we chose the weight 1 edge on the bottom of the triangle of weight 1 edges in our graph. This edge is incident to two weight 1 edges, a weight 4 edge, a weight 7 edge, and a weight 3 edge. We would then choose the incident edge of minimum weight. In this case, one of the two weight one edges. At this point, we cannot choose the third weight 1 edge since this would form a cycle, but we can continue by choosing a weight 2 edge. We might end up with the following spanning tree, which has weight 17, the smallest we've seen so far.



Now suppose we instead ran ALG2 on our graph. We might again choose the weight 1 edge on the bottom of the triangle of weight 1 edges in our graph. Now, instead of choosing one of the weight 1 edges it touches, we might choose the weight 1 edge on the top of the graph. Note that this edge still has minimum weight, and does not cause us to form a cycle, so ALG2 can choose it. We would then choose one of the remaining weight 1 edges. Note that neither causes us to form a cycle. Continuing the algorithm, we may end up with the same spanning tree as above, though this need not always be the case.

It turns out that both algorithms work, but they might end up with different MSTs. The MST is not necessarily unique - indeed, if all edges of an N-node graph have the same weight $(= 1)$, then all spanning trees have weight $N - 1$.

These are examples of greedy approaches to optimization. Sometimes it works and sometimes it doesn't. The good news is that it works to find the MST. In fact, both variations of ALG work. It's a little easier to prove it for ALG2 so we'll do that one here.

Theorem. *For any connected, weighted graph G, ALG2 produces an MST for G.*

Proof. The proof is a bit tricky. We need to show the algorithm terminates, i.e., if we have selected $< N - 1$ edges, then we can always find an edge to add that does not create a cycle. We also need to show the algorithm creates a tree of minimum weight.

The key to doing all of this is to show that the algorithm never gets stuck or goes in a bad direction by adding an edge that will keep us from ultimately producing an MST. The

natural way to prove this is to show that the set of edges selected at any point is contain in some MST - i.e., we can always get to where we need to be. We'll state this as a Lemma.

Lemma 4. *For any $m \geq 0$, let S consist of the first m edges selected by ALG2. Then there exists some MST $T = (V, E)$ for G s.t. $S \subseteq E$, i.e., the set of edges we are growing is always contained in some MST.*

We'll prove this momentarily, but first let's see why it helps prove the theorem. Assume the lemma is true. Then how do we know ALG2 can always find an edge to add without creating a cycle? Well, as long as there are $< N - 1$ edges picked, there exists some edge in $T \setminus S$ and so the tree has no cycle. Next, how do we know that we get an MST at the end? Well, once $M = N - 1$, we know that S is a tree

Okay, so the theorem is an easy corollary of the lemma. Let's prove the lemma. Any ideas how to proceed? We'll use induction on the number of edges chosen by the algorithm so far. This is very typical in proving an algorithm preserves some kind of invariant condition - induct on the number of steps taken, i.e., the number of edges added.

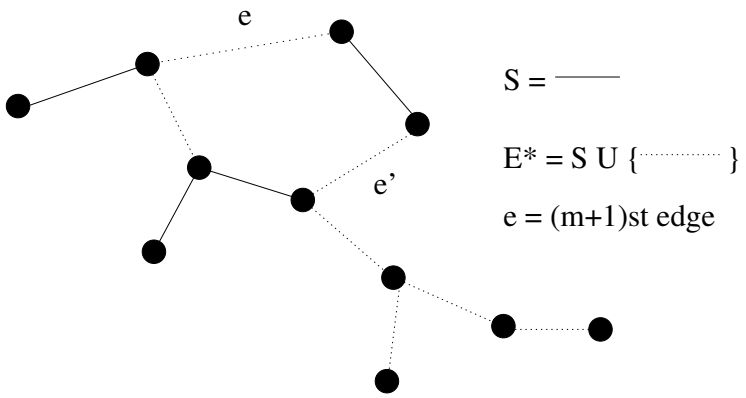
Our inductive hypothesis is the following: $P(m)$: For any G and any set S of m edges initially selected by ALG2, there exists an MST $T = (V, E)$ of G such that $S \subseteq E$.

For the base case, we need to show $P(0)$. In this case $S = \emptyset$, so $S \subseteq E$ trivially holds for any MST $T = (V, E)$.

For the inductive step, we assume $P(m)$ holds and we show that it implies $P(m + 1)$. Let e denote the $(m + 1)$st edge selected by ALG2, and let S denote the first m edges selected by ALG2. Let $T^* = (V^*, E^*)$ be the MST such that $S \subseteq E^*$, which exists by the inductive hypothesis. There are now two cases.

In the first case, $e \in E^*$, in which case $S \cup \{e\} \subseteq E^*$, and thus $P(m + 1)$ holds.

In the second case, $e \notin E^*$, as illustrated by the following diagram. Now we need to find a different MST that contains S and e.



What happens when we add e to T^*? Well since T^* is a tree, we get a cycle. Moreover, the cycle cannot only contain edges in S, since e was chosen so that together with the edges in S, it does not form a cycle. This implies that $e \cup T^*$ contains a cycle that contains an edge e' of $E^* \setminus S$. See the figure above.

Now, we claim that the weight of e is at most that of e'. Indeed, ALG2 picks the minimum weight edge that does not make a cycle with S. Note that $e' \in T^*$ so it cannot make a cycle with S.

Okay, we're almost done. Now we'll make an MST that contains $S \cup \{e\}$. Let $T^{**} = (V, E^{**})$ where $E^{**} = (E^* - \{e'\}) \cup \{e\}$, that is, we swap e and e' in T^*.

Claim. *T^{**} is an MST.*

Proof. We first show that T^{**} is a spanning tree. We have $|E^{**}| = |E^*| = N - 1$ so T^{**} has $N - 1$ edges. Moreover, T^{**} is still connected since we deleted an edge of $e \cup T^*$ to form T^{**}, and that edge was on a cycle. It follows by Theorem 2 that T^{**} is a spanning tree.

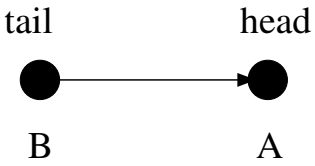
Next, let's look at the weight of T^{**}. Well, since the weight of e was at most that of e', the weight of T^{**} is at most that of T^*, and thus T^{**} is an MST. □

Since $S \cup \{e\} \subseteq E^{**}$, $P(m+1)$ holds. Thus, when ALG2 has $N - 1$ edges, it produces an MST. □

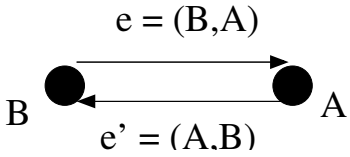
So now we know for sure that the MST for our example graph has weight 17 since it was produced by ALG2.

2 Directed Graphs

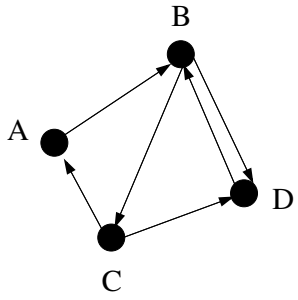
So far we have been working with graphs with undirected edges. Graphs with directed edges are also very useful in computer science. A directed edge is an edge where the endpoints are distinguished - one is the head and one is the tail. Thus its endpoints as forming an *ordered pair*.



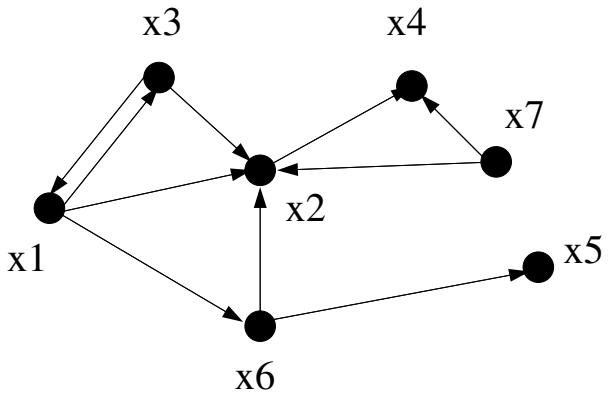
A graph with directed edges is called an *directed graph*, or *digraph*. These graphs can have directed edges in both directions:



Note that we treat e and e' as different edges in the graph above. With directed graphs, the notion of degree splits into *indegree* and *outdegree*. In the following digraph $indegree(D) = 2$ while $outdegree(D) = 1$.



Directed graphs arise in all sorts of applications. One interesting example is the digraph that represents the hyperlink structure of the World Wide Web. In this example, every page or file is represented by a node and there is a directed edge from node x to node y if the page associated with x has a link to the page associated with y. For example, in the following graph the vertices $V = \{x_1, \ldots, x_N\}$ are web pages and $x_i \to x_j$ if x_i contains a hyperlink to x_j.



The web graph is an enormous graph with many billions and probably even trillions of nodes. At first glance, this graph wouldn't seem to be very interesting. But in 1995, two students at Stanford, Larry Page and Sergey Brin realized that the structure of this graph could be very useful in building a search engine. Traditional document searching programs had been around for a long time and they worked in a fairly straighforward way. Basically, you would enter some search terms and the searching program would return all documents containing those terms. A relevance score might also be returned for each document based on the frequency or position that the search terms appeared in the document. For example, if the search term appeared in the title or appeared 100 times in a document, that document would get a higher score. (So if an author wanted a document to get a higher score for certain keywords, the would put the keywords in the title and make it appear in lots of places. You can even see this today with some bogus web sites.)

This approach works fine if you only have a few documents that match a search term. But on the web, there are billions of documents and millions of matches to a typical search.

For example, doing a search on Google for "math computer science notes" gives over 17 million hits! How do you decide which 10 or 20 to show first? (If someone pays then

maybe it's OK, but what about the people who don't??) Probably you don't want to pick the ones that have these words lots of times: math math ... math across the front of the document.

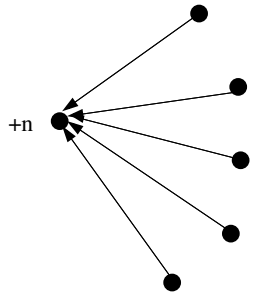
Now in this case, the 3rd ranked hit was for the Spring 2006 class notes for 6.042. Google did not know I was interested in MIT - how on earth did they pick 6.042 to be third out of 17 million? Incidentally, the number 1 was Elsevier - books for sale, and the number 2 was the reference list for the University of Paderborn.

Well back in 1995, Larry and Sergey got the idea to exploit the structure of the web graph to figure out which pages are likely to be the most important and then they would show those first in response to a search request.

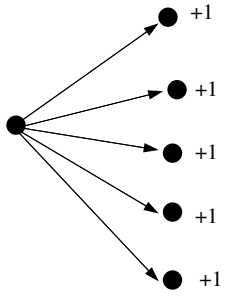
Looking at the web graph, any idea which node/page might be the best to rank 1st? Assume that all the pages match the search terms for now. Well, intuitively, we should choose x_2 since lots of other pages point to it. This leads us to their first idea.

Define the page rank of x to equal $indegree(x)$, which we denote as $PR(x)$. Then, of the pages that match the search terms, list those with the highest page rank first. The idea is to think of web pages as voting for the most important page - the more votes, the better rank.

Of course, there are some problems with this idea. Suppose you wanted to have your page get a high ranking. One thing you could do is to create lots of dummy pages with links to your page.



There is another problem - one page might have a lot of links, thereby increasing the rank of lots of other pages.

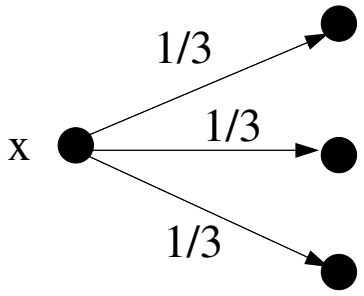


This is sort of like the expression, "vote early, vote often", and this is not good if you want to build a search engine with real money involved.

So, admittedly, their original idea was not so great. It was better than nothing, but certainly not worth billions of dollars.

But then Sergey and Larry thought some more and came up with a couple of improvements.

To solve the "vote early, vote often" problem, they decided to weight each vote so that any page gets at most one total vote. This is done by dividing the vote equally among outgoing links so if a node has outdegree d, every link gets $1/d$ of the vote. So now every edge has weight $1/outdegree(x)$ if its tail is x.



Now any page has the same impact regardless of the number of links it has.

To solve the dummy page problem making your page look more important, they decided to weight the vote of a page by its PageRank! The hope is that dummy pages will have low page rank and thus won't influence the average rank of real pages. In particular, they defined the page rank of a node x by solving the following system of linear equations: find $\{PR(x)\}$ such that

$$\forall x,\ PR(x) = \sum_{i|x_i \to x} \frac{PR(x_i)}{outdegree(x_i)}.$$

For example, in our web graph, we have

$$PR(x_4) = \frac{PR(x_7)}{2} + \frac{PR(x_2)}{1}.$$

One can think of this as x_7 sending $\frac{1}{2}$ of its page rank to x_2 and the other half to x_4. x_2 sends all of its page rank to x_4.

This provides a system of N equations in N variables where N is the number of nodes, or pages, and the variables are the page ranks. At first glance, it's not even clear that there is a solution to this system of equations – other than the solution where all the pageranks are zero, which is not very interesting. Indeed, you'll consider a variation of this scheme in the problem sets, where there is no good solution.

But Sergey and Larry were smart fellows and they set up their page rank algorithm so it would always have a meaningful solution (all PR values of 0, for instance, is not helpful). Furthermore, there will always be a meaningful solution that is non-negative

and for which the sums of the page ranks is equal to 1, and we will consider only such solutions. In fact, it turns out that the page rank computed for a page is the probability that a surfer lands on that page after a long random walk through the web starting from a random page (and clicking on links uniformly). We'll talk more about randomness later in the course, but you can intuitively see why this might be true.

The probability of being at x_4 at time T is just

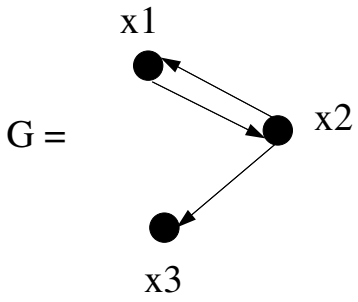
$$\Pr[\text{ at } x_7 \text{ at time } T-1]/2 + \Pr[\text{ at } x_2 \text{ at time } T-1]/1.$$

This matches the page rank formula. Cool, right?

There is still the problem of figuring out how to solve the system of equations to compute the page ranks. To do this, it is helpful to represent the web graph by its adjacency matrix.

Definition. *The* adjacency matrix *for an N-node graph $G = (V, E)$ is the $N \times N$ matrix $A = \{a_{i,j}\}$ where $a_{i,j} = 1$ if and only if $(x_i, x_j) \in E$.*

For example, the following graph

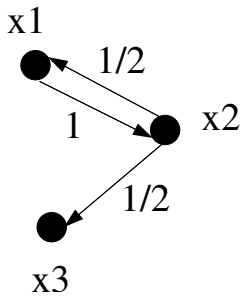


would have adjacency matrix

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

That is, x_1 points to x_2, x_2 points to x_1 and x_3, and x_3 has no outgoing edges.

For a *weighted* graph, the weighted adjacency matrix is $W = \{w_{i,j}\}$, where $w_{i,j}$ is the weight on edge $i \to j$ if $(i, j) \in E$, and is 0 otherwise. For example, the graph



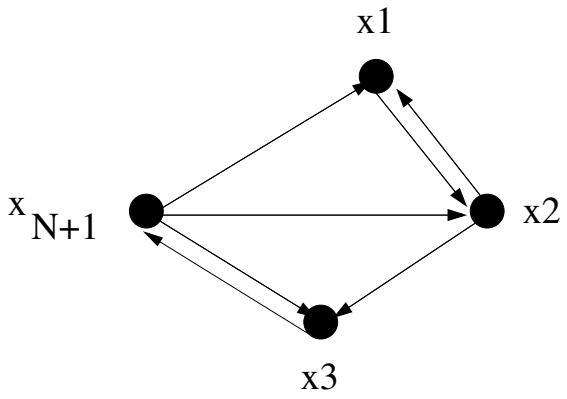
would have adjacency matrix

$$\begin{pmatrix} 0 & 1 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 0 & 0 \end{pmatrix}$$

Now for Sergey and Larry's graph, the edge weights are just $1/outdegree(x)$ for nodes x. The row sums in such a weighted graph must equal 0 or 1 since in the xth row, each non-zero entry, if any, has weight $1/outdegree(x)$. Of course some rows may have zero sum, and these correspond to pages with no hyperlinks out, i.e., nodes with outdegree 0. These nodes are called *sinks* in a digraph because once you are there, you can't get out.

Definition. *A* sink *is a node with outdegree* 0.

Of course, in the web you don't really get stuck, you hit back or you go somewhere new, and so to model this to make their page rank values work better, Sergey and Larry added a supernode to the web graph and had every sink point to it. Moreover, the supernode points to every other node in the graph.



Now you can't get stuck at any node. (Note that we could have also added reverse edges everywhere to simulate the back button.) The weighted adjacency matrix W' now is 4×4 and looks like

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 1 \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} & 0 \end{pmatrix}$$

Now all the row sums are one. This turns out to make the page rank calculation better. Sergey and Larry also did a lot of other stuff, but we don't have time to go too deeply into it here, and also most of it is closely held secret!

The nice thing about the weighted adjacency matrix is that it gives a very simple form for expressing the linear equations for page rank. If you put page ranks in vector $\vec{P}$:

$$\begin{pmatrix} PR(x_1) \\ PR(x_2) \\ \dots \\ PR(x_N) \end{pmatrix}$$

Then $W^T\vec{P} = \vec{P}$, where W^T denotes the transpose of W. Note that this follows from the fact that the equations have the form $\sum_{1\leq i\leq N} w_{i,j}PR(x_i) = PR(x_j)$ for each $1 \leq j \leq N$.

If you have taken a linear algebra or numerical analysis course, you realize that the vector of page ranks is just the principle eigenvector of the weighted adjacency matrix of the web graph! Once you've had such a course, these values are easy to compute. Of course, when you are dealing with matrices of this size, the problem gets a little more interesting. Just keeping track of billions of pages is a daunting task. That's why Google is building power plants. Indeed, Larry and Sergey named their system Google after the number 10^{100}, called googol), to reflect the fact that the web graph is so enormous.

Anyway, now you can see how 6.042 ranked third out of 17 million matches. Lots of other universities use our notes and probably have links to the 6.042 open courseware site and they are themselves legitimate, which ultimately leads 6.042 to get a high page rank in the web graph.

The moral of this story is that you too can become a billionaire if you study your graph theory!