Unit GT

Basic Concepts in Graph Theory

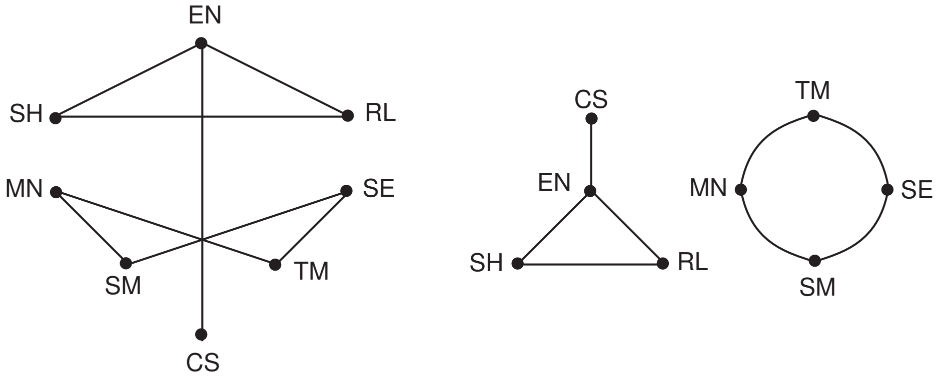
Section 1: What is a Graph?

There are various types of graphs, each with its own definition. Unfortunately, some people apply the term "graph" rather loosely, so you can't be sure what type of graph they're talking about unless you ask them. After you have finished this chapter, we expect you to use the terminology carefully, not loosely. To motivate the various definitions, we'll begin with some examples.

Example 1 (A computer network) Computers are often linked with one another so that they can interchange information. Given a collection of computers, we would like to describe this linkage in fairly clean terms so that we can answer questions such as "How can we send a message from computer A to computer B using the fewest possible intermediate computers?"

We could do this by making a list that consists of pairs of computers that are connected. Note that these pairs are unordered since, if computer C can communicate with computer D, then the reverse is also true. (There are sometimes exceptions to this, but they are rare and we will assume that our collection of computers does not have such an exception.) Also, note that we have implicitly assumed that the computers are distinguished from each other: It is insufficient to say that "A PC is connected to a Mac." We must specify which PC and which Mac. Thus, each computer has a unique identifying label of some sort.

For people who like pictures rather than lists, we can put dots on a piece of paper, one for each computer. We label each dot with a computer's identifying label and draw a curve connecting two dots if and only if the corresponding computers are connected. Note that the shape of the curve does not matter (it could be a straight line or something more complicated) because we are only interested in whether two computers are connected or not. Below are two such pictures of the same graph. Each computer has been labeled by the initials of its owner.



Computers (vertices) are indicated by dots (•) with labels. The connections (edges) are indicated by lines. When lines cross, they should be thought of as cables that lie on top of each other — not as cables that are joined. □

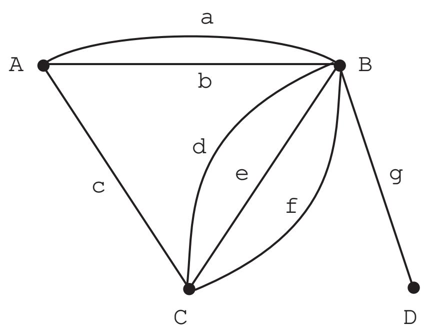
The notation $\mathcal{P}_k(V)$ stands for the set of all k-element subsets of the set V. Based on the previous example we have

Definition 1 (Simple graph) *A simple graph G is a pair $G = (V, E)$ where*

- *V is a finite set, called the vertices of G, and*
- *E is a subset of $\mathcal{P}_2(V)$ (i.e., a set E of two-element subsets of V), called the edges of G.*

In our example, the vertices are the computers and a pair of computers is in E if and only if they are connected.

Example 2 (Routes between cities) Imagine four cities named, with characteristic mathematical charm, A, B, C and D. Between these cities there are various routes of travel, denoted by a, b, c, d, e, f and g. Here is picture of this situation:



Looking at this picture, we see that there are three routes between cities B and C. These routes are named d, e and f. Our picture is intended to give us only information about the interconnections between cities. It leaves out many aspects of the situation that might be of interest to a traveler. For example, the nature of these routes (rough road, freeway, rail, etc.) is not portrayed. Furthermore, unlike a typical map, no claim is made that the picture represents in any way the distances between the cities or their geographical placement relative to each other. The object shown in this picture is called a *graph.*

Following our previous example, one is tempted to list the pairs of cities that are connected; in other words, to extract a simple graph from the information. Unfortunately, this does not describe the problem adequately because there can be more than one route connecting a pair of cities; e.g., d, e and f connecting cities B and C in the figure. How can we deal with this? Here is a precise definition of a graph of the type required to handle this type of problem. □

Definition 2 (Graph) *A graph is a triple $G = (V, E, \phi)$ where*

- *V is a finite set, called the vertices of G,*
- *E is a finite set, called the edges of G, and*
- *ϕ is a function with domain E and codomain $\mathcal{P}_2(V)$.*

In the pictorial representation of the cities graph, $G = (V, E, \phi)$ where

$$V = \{A, B, C, D\}, \quad E = \{a, b, c, d, e, f, g\}$$

and

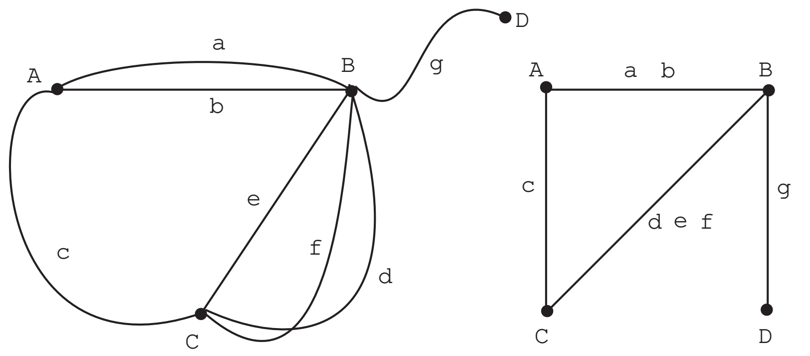
$$\phi = \begin{pmatrix} a & b & c & d & e & f & g \\ \{A,B\} & \{A,B\} & \{A,C\} & \{B,C\} & \{B,C\} & \{B,C\} & \{B,D\} \end{pmatrix}.$$

Definition 2 tells us that to specify a graph G it is necessary to specify the sets V and E and the function ϕ. We have just specified V and ϕ in *set theoretic* terms. The picture of the cities graph specifies the same V and ϕ in pictorial terms. The set V is represented clearly by dots ($\bullet$), each of which has a city name adjacent to it. Similarly, the set E is also represented clearly. The function ϕ is determined from the picture by comparing the name attached to a route with the two cities connected by that route. Thus, the route name d is attached to the route with endpoints B and C. This means that $\phi(d) = \{B, C\}$.

Note that, since part of the definition of a function includes its codomain and domain, ϕ determines $\mathcal{P}_2(V)$ and E. Also, V can be determined from $\mathcal{P}_2(V)$. Consequently, we could have said that a graph is a function ϕ whose domain is a finite set and whose codomain is $\mathcal{P}_2(V)$ for some finite set V. Instead, we choose to specify V and E explicitly because the vertices and edges play a fundamental role in thinking about a graph G.

The function ϕ is sometimes called the *incidence function* of the graph. The two elements of $\phi(x) = \{u, v\}$, for any $x \in E$, are called the vertices of the edge x, and we say u and v are *joined* by x. We also say that u and v are *adjacent vertices* and that u is *adjacent to* v or , equivalently, v is adjacent to u. For any $v \in V$, if v is a vertex of an edge x then we say x is *incident* on v. Likewise, we say v is a member of x, v is on x, or v is in x. Of course, v is a member of x actually means v is a member of $\phi(x)$.

Here are two additional pictures of the same cities graph given above:



The drawings look very different but exactly the same set V and function ϕ are specified in each case. It is *very important* that you understand exactly what information is needed to completely specify the graph. When thinking in terms of cities and routes between them, you naturally want the pictorial representation of the cities to represent their geographical positioning also. If the pictorial representation does this, that's fine, but it is not a part of the information required to define a graph. Geographical location is extra information. The geometrical positioning of the vertices A, B, C and D is very different, in the first of the two pictorial representations above, than it was in our original representation of the cities. However, in each of these cases, the vertices on a given edge are the same and hence the graphs specified are the same. In the second of the two pictures above, a different method of specifying the graph is given. There, ϕ^{-1}, the inverse of ϕ, is given. For example, $\phi^{-1}(\{C,B\})$ is shown to be $\{d,e,f\}$. Knowing ϕ^{-1} determines ϕ and hence determines G since the vertices A, B, C and D are also specified.

Example 3 (Loops) A *loop* is an edge that connects a vertex to itself. Graphs and simple graphs as defined in Definitions 1 and 2 cannot have loops. Why? Suppose $e \in E$ is a loop in a graph that connects $v \in V$ to itself. Then $\phi(e) = \{v,v\} = \{v\}$ because repeated elements in the description of a set count only once — they're the same element. Since $\{v\} \notin \mathcal{P}_2(V)$, the range of ϕ, we cannot have $\phi(e) = \{v,v\}$. In other words, we cannot have a loop.

Thus, if we want to allow loops, we will have to change our definitions. For a graph, we expand the codomain of ϕ to be $\mathcal{P}_2(V) \cup \mathcal{P}_1(V)$. For a simple graph we need to change the set of allowed edges to include loops. This can be done by saying that E is a subset of $\mathcal{P}_2(V) \cup \mathcal{P}_1(V)$ instead of a subset of just $\mathcal{P}_2(V)$. For example, if $V = \{1,2,3\}$ and $E = \{\{1,2\},\{2\},\{2,3\}\}$, this simple graph has a loop at vertex 2 and vertex 2 is connected by edges to the other two vertices. When we want to allow loops, we speak of a graph with loops or a simple graph with loops.

Examples of graphs with loops appear in the exercises. □

We have two definitions, Definition 1 (simple graph) and Definition 2 (graph). How are they related? Let $G = (V,E)$ be a simple graph. Define $\phi: E \to E$ to be the identity map; i.e., $\phi(e) = e$ for all $e \in E$. The graph $G' = (V,E,\phi)$ is essentially the same as G. There is one subtle difference in the pictures: The edges of G are unlabeled but each edge of G' is labeled by a set consisting of the two vertices at its ends. But this extra information is contained already in the specification of G. Thus, simple graphs are a special case of graphs.

Definition 3 (Degrees of vertices) *Let $G = (V,E,\phi)$ be a graph and $v \in V$ a vertex. Define the degree of v, $d(v)$ to be the number of $e \in E$ such that $v \in \phi(e)$; i.e., e is incident on v.*
Suppose $|V| = n$. Let $d_1, d_2, \ldots, d_n$, where $d_1 \le d_2 \le \cdots \le d_n$ be the sequence of degrees of the vertices of G, sorted by size. We refer to this sequence as the degree sequence of the graph G.

In the graph for routes between cities, $d(A) = 3$, $d(B) = 6$, $d(C) = 4$, and $d(D) = 1$. The degree sequence is 1,3,4,6.

Sometimes we are interested only in the "structure" or "form" of a graph and not in the names (labels) of the vertices and edges. In this case we are interested in what is called an unlabeled graph. A picture of an unlabeled graph can be obtained from a picture of a graph by erasing all of the names on the vertices and edges. This concept is simple enough, but is difficult to use mathematically because the idea of a picture is not very precise.

The concept of an *equivalence relation* on a set is an important concept in mathematics and computer science. We'll explore it here and will use it to develop an intuitive understanding of unlabeled graphs. Later we will use it to define connected components and biconnected components. Equivalence relations are discussed in more detail in *A Short Course in Discrete Mathematics*, the text for the course that precedes this course.

Definition 4 (Equivalence relation) *An equivalence relation on a set S is a partition of S. We say that $s, t \in S$ are equivalent if and only if they belong to the same block (called an equivalence class in this context) of the partition. If the symbol $\sim$ denotes the equivalence relation, then we write $s \sim t$ to indicate that s and t are equivalent.*

Example 4 (Equivalence relations) Let S be any set and let all the blocks of the partition have one element. Two elements of S are equivalent if and only if they are the same. This rather trivial equivalence relation is, of course, denoted by "=".

Now let the set be the integers $\mathbb{Z}$. Let's try to define an equivalence relation by saying that n and k are equivalent if and only if they differ by a multiple of 24. Is this an equivalence relation? If it is we should be able to find the blocks of the partition. There are 24 of them, which we could number $0, \ldots, 23$. Block j consists of all integers which equal j plus a multiple of 24; that is, they have a remainder of j when divided by 24. Since two numbers belong to the same block if and only if they both have the same remainder when divided by 24, it follows that they belong to the same block if and only if their difference gives a remainder of 0 when divided by 24, which is the same as saying their difference is a multiple of 24. Thus this partition does indeed give the desired equivalence relation.

Now let the set be $\mathbb{Z} \times \mathbb{Z}^*$, where $\mathbb{Z}^*$ is the set of all integers except 0. Write $(a, b) \sim (c, d)$ if and only if $ad = bc$. With a moment's reflection, you should see that this is a way to check if the two fractions a/b and c/d are equal. We can label each equivalence class with the fraction a/b that it represents. In an axiomatic development of the rationals from the integers, one defines a rational number to be just such an equivalence class and proves that it is possible to add, subtract, multiply and divide equivalence classes.

Suppose we consider all functions $S = \underline{m}^{\underline{n}}$. We can define a partition of S in a number of different ways. For example, we could partition the functions f into blocks where the sum of the integers in the Image(f) is constant, where the max of the integers in Image(f) is constant, or where the "type vector" of the function, namely, the number of 1's, 2's, etc. in Image(f), is constant. Each of these defines a partition of S. $\square$

In the next theorem we provide necessary and sufficient conditions for an equivalence relation. Verifying the conditions is a useful way to prove that some particular situation is an equivalence relation. Recall that a *binary relation* on a set S is a subset R of $S \times S$.

Theorem 1 (Equivalence Relations) *Let S be a set and suppose that we have a binary relation $R \subseteq S \times S$. We write $s \sim t$ whenever $(s, t) \in R$. This is an equivalence relation if and only if the following three conditions hold.*

(i) (Reflexive) For all $s \in S$ we have $s \sim s$.

(ii) (Symmetric) For all $s, t \in S$ such that $s \sim t$ we have $t \sim s$.

(iii) (Transitive) For all $r, s, t \in S$ such that $r \sim s$ and $s \sim t$ we have $r \sim t$.

Proof: We first prove that an equivalence relation satisfies (i)–(iii). Suppose that $\sim$ is an equivalence relation. Since s belongs to whatever block it is in, we have $s \sim s$. Since $s \sim t$ means that s and t belong to the same block, we have $s \sim t$ if and only if we have $t \sim s$. Now suppose that $r \sim s \sim t$. Then r and s are in the same block and s and t are in the same block. Thus r and t are in the same block and so $r \sim t$.

We now suppose that (i)–(iii) hold and prove that we have an equivalence relation. What would the blocks of the partition be? Everything equivalent to a given element should be in the same block. Thus, for each $s \in S$ let $B(s)$ be the set of all $t \in S$ such that $s \sim t$. We must show that the set of these sets form a partition of S.

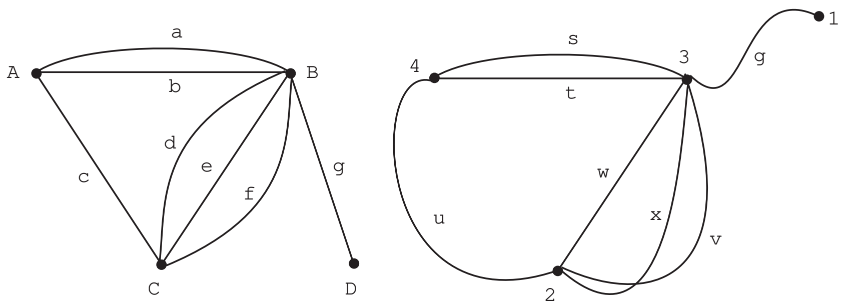
In order to have a partition of S, we must have

(a) the $B(s)$ are nonempty and every $t \in S$ is in some $B(s)$ and

(b) for every $p, q \in S$, $B(p)$ and $B(q)$ are either equal or disjoint.

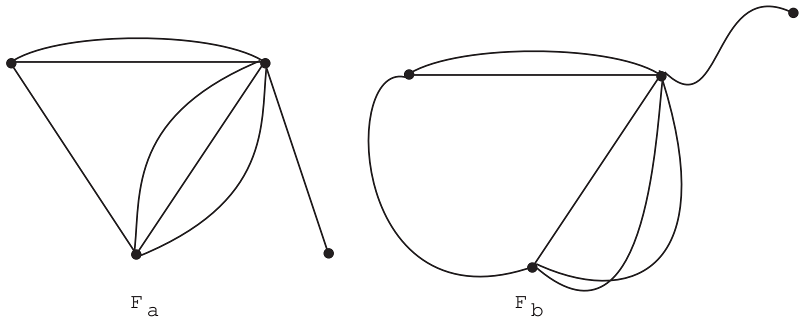
Since $\sim$ is reflexive, $s \in B(s)$, proving (a). Suppose $x \in B(p) \cap B(q)$ and $y \in B(p)$. We have, $p \sim x$, $q \sim x$ and $p \sim y$. Thus $q \sim x \sim p \sim y$ and so $y \in B(q)$, proving that $B(p) \subseteq B(q)$. Similarly $B(q) \subseteq B(p)$ and so $B(p) = B(q)$. This proves (b). $\square$

Example 5 (Equivalent forms) Consider the following two graphs, represented by pictures:



G_a G_b

Now let's remove all symbols representing edges and vertices. What we have left are two "forms" on which the graphs were drawn. You can think of drawing a picture of a graph as a two step process: (1) draw the form; (2) add the labels. One student referred to these forms as "ghosts of departed graphs." Note that form F_a and form F_b have a certain eerie similarity (appropriate for ghosts).



F_a F_b

If you use your imagination a bit you can see that form F_b can be transformed into form F_a by sliding vertices around and bending, stretching, and contracting edges as needed. The edges need not be detached from their vertices in the process and edges and vertices, while being moved, can pass through each other like shadows. Let's refer to the sliding, bending, stretching, and contracting process as "morphing" form F_a into F_b. Morphing is easily seen to define an equivalence relation $\sim$ on the set of all forms. Check out reflexive, symmetric, and transitive, for the morphing relation $\sim$. By Theorem 1, the morphing equivalence relation partitions the set of all forms of graphs into blocks or equivalence classes. This is a good example where it is easier to think of the relation $\sim$ than to think globally of the partition of the forms.

Now suppose we have any two graphs, $G_a = (V_a, E_a, \phi_a)$ and $G_b = (V_b, E_b, \phi_b)$. Think of these graphs not as pictures, but as specified in terms of sets and functions. Now choose forms F_a and F_b for G_a and G_b respectively, and draw their pictures. We leave it to your intuition to accept the fact that either $F_a \sim F_b$, no matter what you choose for F_a and F_b, or $F_a \not\sim F_b$ no matter what your choice is for the forms F_a and F_b. If $F_a \sim F_b$ we say that G_a and G_b are *isomorphic graphs* and write $G_a \approx G_b$. The fact that $\sim$ is an equivalence relation forces $\approx$ to be an equivalence relation also. In particular, two graphs G_a and G_b are isomorphic if and only if you can choose any form F_a for drawing G_a and use that same form for G_b. $\square$

In general, deciding whether or not two graphs are isomorphic can be very difficult business. You can imagine how hard it would be to look at the forms of two graphs with thousands of vertices and edges and deciding whether or not those forms are equivalent. There are no general computer programs that do the task of deciding isomorphism well. For graphs known to have special features, isomorphism of graphs can sometimes be decided efficiently. In general, if someone presents you with two graphs and asks you if they are isomorphic, your best answer is "no." You will be right most of the time.

---

*Random Graphs

We now look briefly at a subject called *random graphs*. They often arise in the analysis of graphical algorithms and of systems which can be described graphically (such as the web). There are two basic ways to describe random graphs. One is to let the probability space be the set of all graphs with, for example, n vertices and q edges and use the uniform distribution. The other, which is often easier to study is described in the following definition. It is the one we study here.

***Definition 5 (Random graph model)** *Let $\mathcal{G}(n,p)$ be the probability space obtained by letting the elementary events be the set of all n-vertex simple graphs with $V = \underline{n}$. If $G \in \mathcal{G}(n,p)$ has m edges, the $P(G) = p^m q^{N-m}$ where $q = 1-p$ and $N = \binom{n}{2}$.*

We need to show that $\mathcal{G}(n,p)$ is a probability space. There is a nice way to see this by reinterpreting P. List the $N = \binom{n}{2}$ vertices $\mathcal{P}_2(V)$ in lex order. Let the sample space be $U = \times^N\{\text{choose}, \text{reject}\}$ with $P(a_1, \ldots, a_N) = P^*(a_1) \times \cdots \times P^*(a_N)$ where $P^*(\text{choose}) = p$

and $P^*(\text{reject}) = 1 - p$. We've met this before in Unit Fn and seen that it is a probability space. To see that it is, note that $P \geq 0$ and

$$\begin{aligned}\sum_{a_1,\ldots,a_N} P(a_1,\ldots,a_N) &= \sum_{a_1,\ldots,a_N} P^*(a_1)\times\cdots\times P^*(a_N)\\ &= \Big(\sum_{a_1} P^*(a_1)\Big)\times\cdots\times\Big(\sum_{a_N} P^*(a_N)\Big)\\ &= (p+(1-p))\times\cdots\times(p+(1-p)) = 1\times\cdots\times 1 = 1.\end{aligned}$$

Why is this the same as the definition? Think of the chosen pairs as the edges of a graph chosen randomly from $\mathcal{G}(n,p)$. If G has m edges, then its probability should be $p^m(1-p)^{N-m}$ according to the definition. On the other hand, since G has m edges, exactly m of $a_1,\ldots,a_N$ equal "choose" and so, in the new space, $P(a_1,\ldots,a_N) = p^m(1-p)^{N-m}$ also. We say that we are choosing the edges of the random graph independently.

***Example 6 (The number of edges in random graph)** Let X be a random variable that counts the number of edges in a random graph in $\mathcal{G}(n,p)$. What are the expected value and variance of X? In $U = \times^N\{\text{choose},\text{reject}\}$, let

$$X_i(a_1,\ldots,a_N) = \begin{cases} 1 & \text{if } a_i = \text{choose},\\ 0 & \text{if } a_i = \text{reject}.\end{cases}$$

You should be able to see that $X = X_1+\cdots+X_N$ and that the X_i are independent random variables with $P(X_i = 1) = p$. This is just the binomial distribution (Unit Fn). We showed that the mean is Np and the variance is Npq, where $N = \binom{n}{2}$ and $q = 1-p$. $\square$

***Example 7 (Triangles in random graphs)** How often can we find 3 vertices $\{u,v,w\}$ in a random graph so that $\{u,v\}$, $\{u,w\}$, and $\{v,w\}$ are all edges of the graph? In other words, how often can we find a "triangle"? How can we do this?

First, we need a sample space. It will be the random graph space introduced in Definition 5. Since we want to count something (triangles), we need a random variable. Let X be a random variable whose value is the number of triples of vertices such that the three possible edges connecting them are present in the random graph. In other words, X is defined for each graph, G, and its value, $X(G)$, is the number of triangles in the graph G. We want to compute $E(X)$. It would also be nice to compute $\text{Var}(X)$ since that gives us some idea of how much X tends to vary from graph to graph — large $\text{Var}(X)$ means there tends to be a lot of variation in the number of triangles from graph to graph and small $\text{Var}(X)$ means there tends to be little variation.

Let $X_{u,v,w}$ be a random variable which is 1 if the triangle with vertices $\{u,v,w\}$ is present and 0 otherwise. Then X is the sum of $X_{u,v,w}$ over all $\{u,v,w\} \in \mathcal{P}_3(V)$. Since expectation is linear, $E(X)$ is the sum of $E(X_{u,v,w})$ over all $\{u,v,w\} \in \mathcal{P}_3(V)$. Clearly $E(X_{u,v,w})$ does not depend on the particular triple. Since there are $\binom{n}{3}$ possibilities for $\{u,v,w\}$, $E(X) = \binom{n}{3}E(X_{1,2,3})$.

We want to compute $E(X_{1,2,3})$. It is given by

$$E(X_{1,2,3}) \;=\; 0P(X_{1,2,3}=0) + 1P(X_{1,2,3}=1) \;=\; P(X_{1,2,3}=1).$$

The only way $X_{1,2,3} = 1$ can happen is for the edges $\{1,2\}$, $\{1,3\}$, and $\{2,3\}$ to all be present in the graph. (We don't care about any of the other possible edges.) Since each of these events has probability p and the events are independent we have $P(X_{1,2,3} = 1) = p^3$. Thus $E(X_{1,2,3}) = p^3$ and so $E(X) = \binom{n}{3}p^3$. In other words, on average we see about $\binom{n}{3}p^3$ triangles. For example, if $p = 1/2$ all graphs are equally likely (You should show this.) and so the average number of triangles over all graphs with n vertices is $\binom{n}{3}/8$. When $n = 5$, this average is 1.25. Can you verify this by looking at all the 5-vertex graphs? How much work is involved?

What happens when n is very large? Then $\binom{n}{3} = \frac{n(n-1)(n-2)}{6}$ "behaves like" $n^3/6$. ("Behaves like" means that, as n goes to infinity, the limit of the ratio $\binom{n}{3}/(n^3/6)$ is 1.) Thus the expected number of triangles behaves like $(np)^3/6$.

What about the variance? We'll work it out in the next example. For now, we'll simply tell you that it behaves like $n^4p^3(1-p^2)/2$. What does this tell us for large n? The standard deviation behaves like $n^2p^{3/2}\sqrt{(1-p^2)/2}$. A more general version of the central limit theorem than we have discussed tells us the number of triangles tends to have a normal distribution with $\mu = (np)^3/6$ and $\sigma = n^2p^{3/2}\sqrt{(1-p^2)/2}$. If p is constant, σ will grow like a constant times n^2, which is much smaller than μ for large n. Thus the number of triangles in a random graph is almost surely close to $(np)^3/6$. $\square$

***Example 8 (Variance for triangles in random graphs)** This is a continuation of the previous example. Since the various $X_{u,v,w}$ may not be independent, this is harder. Since $\mathrm{Var}(X) = E(X^2) - E(X)^2$, we will compute $E(X^2)$. Since X is a sum of terms of the form $X_{r,s,t}$, X^2 is a sum of terms of the form $X_{u,v,w}X_{a,b,c}$. Using linearity of expectation, we need to compute $E(X_{u,v,w}X_{a,b,c})$ for each possibility and add them up.

Now for the tricky part: This expectation depends on how many vertices $\{u,v,w\}$ and $\{a,b,c\}$ have in common.

- If $\{u,v,w\} = \{a,b,c\}$, then $X_{u,v,w}X_{a,b,c} = X_{u,v,w}$ and its expectation is p^3 by the previous example.
- If $\{u,v,w\}$ and $\{a,b,c\}$ have two vertices in common, then the two triangles have only 5 edges total because they have a common edge. Note that $X_{u,v,w}X_{a,b,c}$ is 1 if all five edges are present and is 0 otherwise. Reasoning as in the previous example, the expectation is p^5.
- If $\{u,v,w\}$ and $\{a,b,c\}$ have less than two vertices in common, we are concerned about six edges and obtain p^6 for the expectation.

To add up the results in the previous paragraph, we need to know how often each occurs in

$$X^2 = \left(\sum_{\{u,v,w\}\in\mathcal{P}_3(V)} X_{u,v,w}\right)\left(\sum_{\{a,b,c\}\in\mathcal{P}_3(V)} X_{a,b,c}\right) = \sum_{\substack{\{u,v,w\}\in\mathcal{P}_3(V)\\ \{a,b,c\}\in\mathcal{P}_3(V)}} X_{u,v,w}X_{a,b,c}\,.$$

- When $\{u,v,w\} = \{a,b,c\}$, we are only free to choose $\{u,v,w\}$ and this can be done in $\binom{n}{3}$ ways so we have $\binom{n}{3}p^3$ contributed to $E(X^2)$.

- Suppose $\{u,v,w\}$ and $\{a,b,c\}$ have two vertices in common. How many ways can this happen? We can first choose $\{u,v,w\}$. Then choose two of u, v, w to be in $\{a,b,c\}$ and then choose the third vertex in $\{a,b,c\}$ to be different from u, v, and w. This can be done in
$$\binom{n}{3} \times \binom{3}{2} \times (n-3) = 3(n-3)\binom{n}{3} = 12\binom{n}{4}$$
ways. Multiplying this by p^5 gives its contribution to $E(X^2)$.

- The remaining case, one vertex or no vertices in common, can be done in a similar fashion. Alternatively, we can simply subtract the above counts from all possible ways of choosing $\{u,v,w\}$ and $\{a,b,c\}$. This gives us
$$\binom{n}{3} \times \binom{n}{3} - \binom{n}{3} - 12\binom{n}{4}$$
for the third case. Multiplying this by p^6 gives its contribution to $E(X^2)$.

  Since $E(X)^2 = \binom{n}{3}^2 p^6$, we have that
$$\mathrm{Var}(X) = E(X)^2 - E(X^2) = \binom{n}{3}(p^3 - p^6) + 12\binom{n}{4}(p^3 - p^5),$$

after a bit of algebra using the results in the preceding paragraph. Whew! $\square$

The previous material would be more difficult if we had used the model for random graphs that was suggested before Definition 5. Why is this? The model we are using lets us ignore possible edges that we don't care about. The other model does not because we must be sure that the total number of edges is correct.

Exercises for Section 1

1.1. We are interested in the number of simple graphs with $V = \underline{n}$.

(a) Prove that there are $2^{\binom{n}{2}}$ (2 to the power $\binom{n}{2}$) such simple graphs.

(b) How many of them have exactly q edges?

1.2. Let (V,E,ϕ) be a graph and let $d(v)$ be the degree of the vertex $v \in V$. Prove that $\sum_{v\in V} d(v) = 2|E|$, an even number. Conclude that the number of vertices v for which $d(v)$ is odd is even.

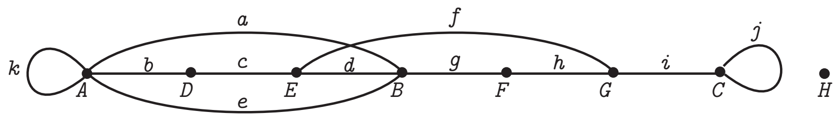
1.3. Let $Q = (V,E,\phi)$ be a graph where

$$V = \{A,B,C,D,E,F,G,H\}, \quad E = \{a,b,c,d,e,f,g,h,i,j,k\}$$

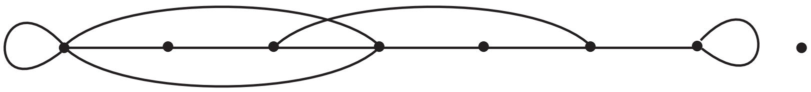
and

$$\phi = \begin{pmatrix} a & b & c & d & e & f & g & h & i & j & k \\ A & A & D & E & A & E & B & F & G & C & A \\ B & D & E & B & B & G & F & G & C & C & A \end{pmatrix}.$$

In this representation of ϕ, the first row specifies the edges and the two vertices below each edge specify the vertices incident on that edge. Here is a pictorial representation $P(Q)$ of this graph.

$P(Q)$:


Note that $\phi(k) = \{A, A\} = \{A\}$. Such an edge is called a *loop.* (See Example 3.) Adding a loop to a vertex increases its degree by two. The vertex H, which does not belong to $\phi(x)$ for any edge x (i.e., has no edge incident upon it), is called an *isolated* vertex. The degree of an isolated vertex is zero. Edges, such as a and e of Q, with the property that $\phi(a) = \phi(e)$ are called *parallel* edges. If all edge and vertex labels are removed from $P(Q)$ then we get the following picture $P'(Q)$:

$P'(Q)$:


The picture $P'(Q)$ represents the “form” of the graph just described and is sometimes referred to as a pictorial representation of the “unlabeled” graph associated with Q. (See Example 5.) For each of the following graphs R, where $R = (V, E, \phi)$, $V = \{A, B, C, D, E, F, G, H\}$, draw a pictorial representation of R by starting with $P'(Q)$ removing and/or adding as few edges as possible and then labeling the resulting picture with the edges and vertices of R. A graph R which require no additions or removals of edges is said to be “of the same form as” or “isomorphic to” the graph Q (Example 5).

(a) Let

$$E = \{a, b, c, d, e, f, g, h, i, j, k\}$$

be the set of edges of R and

$$\phi = \begin{pmatrix} a & b & c & d & e & f & g & h & i & j & k \\ C & C & F & A & H & E & E & A & D & A & A \\ C & G & G & H & H & H & F & H & G & D & F \end{pmatrix}.$$

(b) Let

$$E = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

be the set of edges of R and

$$\phi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ A & E & E & E & F & G & H & B & C & D & E \\ G & H & E & F & G & H & B & C & D & D & H \end{pmatrix}.$$

1.4. Let $Q = (V, E, \phi)$ be the graph where

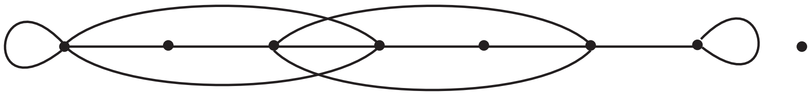
$$V = \{A, B, C, D, E, F, G, H\}, \quad E = \{a, b, c, d, e, f, g, h, i, j, k, l\}$$

and

$$\phi = \begin{pmatrix} a & b & c & d & e & f & g & h & i & j & k & l \\ A & A & D & E & A & E & B & F & G & C & A & E \\ B & D & E & B & B & G & F & G & C & C & A & G \end{pmatrix}.$$

(a) What is the degree sequence of Q?

Consider the following unlabeled pictorial representation of Q

$P'(Q)$: 

(a) Create a pictorial representation of Q by labeling $P'(Q)$ with the edges and vertices of Q.

(b) A necessary condition that a pictorial representation of a graph R can be created by labeling $P'(Q)$ with the vertices and edges of R is that the degree sequence of R be $(0, 2, 2, 3, 4, 4, 4, 5)$. True of false? Explain.

(c) A sufficient condition that a pictorial representation of a graph R can be created by labeling $P'(Q)$ with the vertices and edges of R is that the degree sequence of R be $(0, 2, 2, 3, 4, 4, 4, 5)$. True or false? Explain.

1.5. In each of the following problems information about the degree sequence of a graph is given. In each case, decide if a graph satisfying the specified conditions exists or not. Give reasons in each case.

(a) A graph Q with degree sequence $(1, 1, 2, 3, 3, 5)$?

(b) A graph Q with degree sequence $(1, 2, 2, 3, 3, 5)$, loops and parallel edges allowed?

(c) A graph Q with degree sequence $(1, 2, 2, 3, 3, 5)$, no loops but parallel edges allowed?

(d) A graph Q with degree sequence $(1, 2, 2, 3, 3, 5)$, no loops or parallel edges allowed?

(e) A simple graph Q with degree sequence $(3, 3, 3, 3)$?

(f) A graph Q with degree sequence $(3, 3, 3, 3)$, no loops or parallel edges allowed?

(g) A graph Q with degree sequence $(3, 3, 3, 5)$, no loops or parallel edges allowed?

(h) A graph Q with degree sequence $(4, 4, 4, 4, 4)$, no loops or parallel edges allowed?

(i) A graph Q with degree sequence $(4, 4, 4, 4, 6)$, no loops or parallel edges allowed?

1.6. Divide the following graphs into isomorphism equivalence classes and justify your answer; i.e., explain why you have the classes that you do. In all cases $V = \underline{4}$.

(a) $\phi = \begin{pmatrix} a & b & c & d & e & f \\ \{1,2\} & \{1,2\} & \{2,3\} & \{3,4\} & \{1,4\} & \{2,4\} \end{pmatrix}$

(b) $\phi = \begin{pmatrix} A & B & C & D & E & F \\ \{1,2\} & \{1,4\} & \{1,4\} & \{1,2\} & \{2,3\} & \{3,4\} \end{pmatrix}$

(c) $\phi = \begin{pmatrix} u & v & w & x & y & z \\ \{2,3\} & \{1,3\} & \{3,4\} & \{1,4\} & \{1,2\} & \{1,2\} \end{pmatrix}$

(d) $\phi = \begin{pmatrix} P & Q & R & S & T & U \\ \{3,4\} & \{2,4\} & \{1,3\} & \{3,4\} & \{1,2\} & \{1,2\} \end{pmatrix}$

***1.7.** In Example 7, suppose that p is a function of n, say $p = p(n)$.

(a) Show that the expected number of triangles is behaves like 1 for large n if $p(n) = 6^{1/3}/n$.

(b) Suppose the expected number of triangles behaves like 1. How does the expected number of edges behave?

***1.8.** Instead of looking for triangles as in Example 7, let's look for quadrilaterals having both diagonals. In other words, we'll look for sets of four vertices such that all of the $\binom{4}{2} = 6$ possible edges between them are present.

(a) Show that the expected number of such quadrilaterals is $\binom{n}{4}p^6$.

(b) Suppose n is large and p is a function of n so that we expect to see 1 quadrilateral on average. About how many edges do we expect to see?

(c) Generalize this problem from sets of 4 vertices to sets of k vertices.

***1.9.** Show that the variance of X, the number of triangles in a random graph as computed in Example 8 satisfies

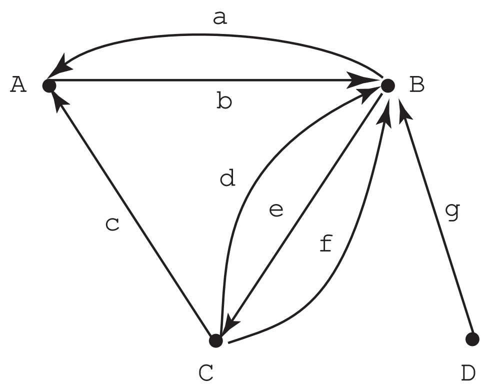
$$\mathrm{Var}(X) = \binom{n}{3}p^3\Big((1-p^3) + 3(n-3)(1-p^2)\Big) < 3n\binom{n}{3}p^3(1-p^2)\,.$$

Hint: $1 - p^3 < 1 - p^2 < 1$.

Section 2: Digraphs, Paths, and Subgraphs

In this section we introduce the notion of a directed graph and give precise definitions of some very important special substructures of both graphs and directed graphs.

Example 9 (Flow of commodities) Look again at Example 2. Imagine now that the symbols a, b, c, d, e, f and g, instead of standing for route names, stand for commodities (applesauce, bread, computers, etc.) that are produced in one town and shipped to another town. In order to get a picture of the flow of commodities, we need to know the directions in which they are shipped. This information is provided by picture below:



In set-theoretic terms, the information needed to construct the above picture can be specified by giving a pair $D = (V, E, \phi)$ where ϕ is a function. The domain of the function ϕ is $E = \{a, b, c, d, e, f, g\}$ and the codomain is $V \times V$. Specifically,

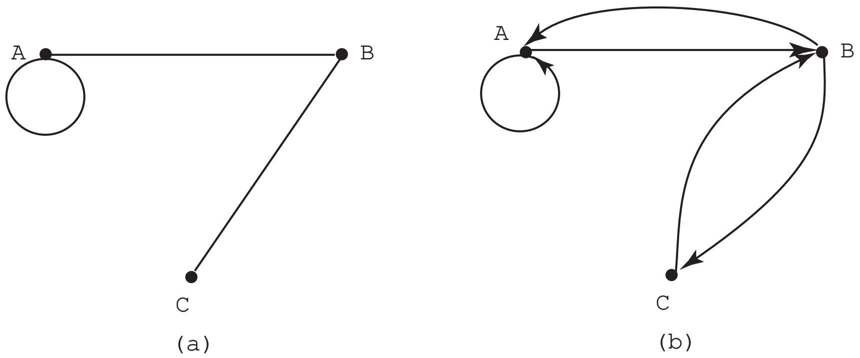
$$\phi = \begin{pmatrix} a & b & c & d & e & f & g \\ (B,A) & (A,B) & (C,A) & (C,B) & (B,C) & (C,B) & (D,B) \end{pmatrix}.$$

The structure specified by this information is an example of a *directed graph*, which we now define. □

Definition 6 (Directed graph) *A directed graph (or digraph) is a triple $D = (V, E, \phi)$ where V and E are finite sets and ϕ is a function with domain E and codomain $V \times V$. We call E the set of edges of the digraph D and call V the set of vertices of D.*

Just as with graphs, we can define a notion of a *simple digraph.* A simple digraph is a pair $D = (V, E)$, where V is a set, the vertex set, and $E \subseteq V \times V$ is the edge set. Just as with simple graphs and graphs, simple digraphs are a special case of digraphs in which ϕ is the identity function on E; that is, $\phi(e) = e$ for all $e \in E$.

There is a correspondence between simple graphs and simple digraphs that is fairly common in applications of graph theory. To interpret simple graphs in terms of simple digraphs, it is best to consider simple graphs with loops (see Example 3 and Exercises for Section 1). Thus consider $G = (V, E)$ where $E \subseteq \mathcal{P}_2(V) \cup \mathcal{P}_1(V)$. We can identify $\{u, v\} \in \mathcal{P}_2(V) \cup \mathcal{P}_1(V)$ with $(u, v) \in V \times V$ and with $(v, u) \in V \times V$. In the case were we have a loop, $u = v$, then we identify $\{u\}$ with (u, u). Here is a picture of a simple graph and its corresponding digraph:

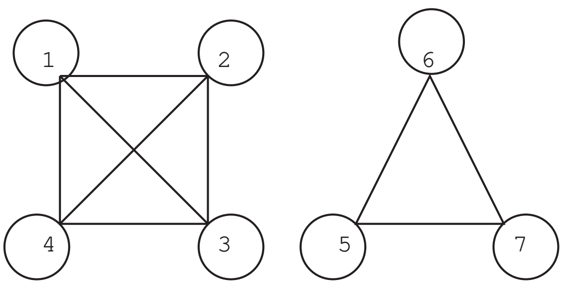


Each, edge that is not a loop in the simple graph is replaced by two edges "in opposite directions" in the corresponding simple digraph. A *loop* is replaced by a *directed loop* (e.g., $\{A\}$ is replaced by (A, A)).

Simple digraphs appear in mathematics under another important guise: *binary relations.* A binary relation on a set V is simply a subset of $V \times V$. Often the name of the relation and the subset are the same. Thus we speak of the binary relation $E \subseteq V \times V$. If you have absorbed all the terminology, you should be able to see immediately that (V, E) is a simple digraph and that any simple digraph (V', E') corresponds to a binary relation $E' \subseteq V' \times V'$.

Recall that a binary relation R is called *symmetric* if $(u, v) \in R$ implies $(v, u) \in R$. Thus simple graphs with loops correspond to symmetric binary relations.

An equivalence relation on a set S is a particular type of binary relation $R \subseteq S \times S$. For an equivalence relation, we have $(x, y) \in R$ if and only if x and y are equivalent (i.e., belong to the same equivalence class or block). Note that this is a symmetric relationship, so we may regard the associated simple digraph as a simple graph. Which simple graphs (with loops allowed) correspond to equivalence relations? As an example, take $S = \underline{7}$ and take the equivalence class partition to be $\{\{1, 2, 3, 4\}, \{5, 6, 7\}\}$. Since everything in each block is related to everything else, there are $\binom{4}{2} = 6$ non-loops and $\binom{4}{1} = 4$ loops associated with the block $\{1, 2, 3, 4\}$ for a total of ten edges. With the block $\{5, 6, 7\}$ there are three loops and three non-loops for a total of six edges Here is the graph of this equivalence relation:



A *complete simple graph G=(V,E) with loops* is a graph with every possible edge. That is, $E = \mathcal{P}_2(V) \cup \mathcal{P}_1(V)$. In the above graph, each block of the equivalence relation is replaced by the complete simple graph with loops on that block. This is the general rule.

A basic method for studying graphs and digraphs is to study substructures of these objects and their properties. One of the most important of these substructures is called a *path.*

Definition 7 (Path, trail, walk and vertex sequence) *Let $G = (V, E, \phi)$ be a graph.*

Let $e_1, e_2, \ldots, e_{n-1}$ be a sequence of elements of E (edges of G) for which there is a sequence $a_1, a_2, \ldots, a_n$ of distinct elements of V (vertices of G) such that $\phi(e_i) = \{a_i, a_{i+1}\}$ for $i = 1, 2, \ldots, n-1$. The sequence of edges $e_1, e_2, \ldots, e_{n-1}$ is called a path in G. The sequence of vertices $a_1, a_2, \ldots, a_n$ is called the vertex sequence of the path. (Note that since the vertices are distinct, so are the edges.)

If we require that $e_1, \ldots, e_{n-1}$ be distinct, but not that $a_1, \ldots, a_n$ be distinct, the sequence of edges is called a trail.

If we do not even require that the edges be distinct, it is called a walk.

If $G = (V, E, \phi)$ is a directed graph, then $\phi(e_i) = \{a_i, a_{i+1}\}$ is replaced by $\phi(e_i) = (a_i, a_{i+1})$ in the above definition to obtain a directed path, trail, and walk respectively.

Note that the definition of a path requires that it not intersect itself (i.e., have repeated vertices), while a trail may intersect itself. Although a trail may intersect itself, it may not have repeated edges, but a walk may. If $P = (e_1, \ldots, e_{n-1})$ is a path in $G = (V, E, \phi)$ with vertex sequence $a_1, \ldots, a_n$ then we say that P is a *path from a_1 to a_n*. Similarly for a trail or a walk.

In the graph of Example 2, the sequence c, d, g is a path with vertex sequence A, C, B, D. If the graph is of the form $G = (V, E)$ with $E \subseteq \mathcal{P}_2(V)$, then the vertex sequence alone specifies the sequence of edges and hence the path. Thus, Example 1, the vertex sequence MN, SM, SE, TM specifies the path {MN, SM}, {SM, SE}, {SE, TM}. Similarly for digraphs. Consider the graph of Example 9. The edge sequence $P = (g, e, c)$ is a directed path with vertex sequence (D, B, C, A). The edge sequence $P = (g, e, c, b, a)$ is a directed trail, but not a directed path. The edge sequence $P = (d, e, d)$ is a directed walk, but not a directed trail.

Note that every path is a trail and every trail is a walk, but not conversely. However, we can show that, if there is a walk between two vertices, then there is a path. This rather obvious result can be useful in proving theorems, so we state it as a theorem.

Theorem 2 (Walk implies path) *Suppose $u \neq v$ are vertices in the graph $G = (V, E, \phi)$. The following are equivalent:*

(a) There is a walk from u to v.

(b) There is a trail from u to v.

(c) There is a path from u to v.

Furthermore, given a walk from u to v, there is a path from u to v all of whose edges are in the walk.

Proof: Since every path is a trail, (c) implies (b). Since every trail is a walk, (b) implies (a). Thus it suffices to prove that (a) implies (c). Let $e_1, e_2, \ldots, e_k$ be a walk from u to v. We use induction on n, the number of repeated vertices in a walk. If the walk has no repeated vertices, it is a path. This starts the induction at $n = 0$. Suppose $n > 0$. Let r be a repeated vertex. Suppose it first appears in edge e_i and last appears in edge e_j.

If $r = u$, then $e_j, \ldots, e_k$ is a walk from u to v in which r is not a repeated vertex. If $r = v$, then $e_1, \ldots, e_i$ is a walk from u to v in which r is not a repeated vertex. Otherwise, $e_1, \ldots, e_i, e_j, \ldots, e_k$ is a walk from u to v in which r is not a repeated vertex. Hence there are less than n repeated vertices in this walk from u to v and so there is a path by induction. Since we constructed the path by removing edges from the walk, the last statement in the theorem follows. $\square$

Note that the theorem and proof are valid if graph is replaced by digraph and walk, trail, and path are replaced by directed walk, trail, and path.

Another basic notion is that of a subgraph of $G = (V, E, \phi)$, which we will soon define. First we need some terminology about functions. By a *restriction* ϕ' of ϕ to $E' \subseteq E$, we mean the function ϕ' with domain E' and satisfying $\phi'(x) = \phi(x)$ for all $x \in E'$. (When forming a restriction, we may change the codomain. Of course, the new codomain must contain $\text{Image}(\phi') = \phi(E)$. In the following definition, the codomain of ϕ' must be $\mathcal{P}_2(V')$ since G' is required to be a graph.)

Definition 8 (Subgraph) *Let $G = (V, E, \phi)$ be a graph. A graph $G' = (V', E', \phi')$ is a subgraph of G if $V' \subseteq V$, $E' \subseteq E$, and ϕ' is the restriction of ϕ to E'.*

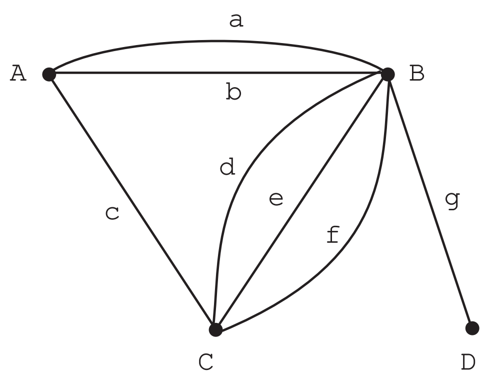
As we have noted, the fact that G' is itself a graph means that $\phi(x) \in \mathcal{P}_2(V')$ for each $x \in E'$ and, in fact, the codomain of ϕ' must be $\mathcal{P}_2(V')$. If G is a graph with loops, the codomain of ϕ' must be $\mathcal{P}_2(V') \cup \mathcal{P}_1(V')$. This definition works equally well if G is a digraph. In that case, the codomain of ϕ' must be $V \times V$.

Example 10 (Subgraph — key information) For the graph $G = (V, E, \phi)$ below, let $G' = (V', E', \phi')$ be defined by $V' = \{A, B, C\}$, $E' = \{a, b, c, f\}$, and by ϕ' being the restriction of ϕ to E' with codomain $\mathcal{P}_2(V')$. Notice that ϕ' is determined completely from knowing V', E' and ϕ. Thus, to specify a subgraph G', the key information is V' and E'.

As another example from the same graph, we let $V' = V$ and $E' = \{a, b, c, f\}$. In this case, the vertex D is not a member of any edge of the subgraph. Such a vertex is called an *isolated vertex* of G'. (See also Exercises for Section 1.)

One way of specifying a subgraph is to give a set of edges $E' \subseteq E$ and take V' to be the set of all vertices on some edge of E'. In other words , V' is the union of the sets $\phi(x)$ over all $x \in E'$. Such a subgraph is called the *subgraph induced by the edge set* E' or the *edge induced subgraph* of E'. The first subgraph of this example is the subgraph induced by $E' = \{a, b, c, f\}$.

Likewise, given a set $V' \subseteq V$, we can take E' to be the set of all edges $x \in E$ such that $\phi(x) \subseteq V'$. The resulting subgraph is called the *subgraph induced by* V' or the *vertex induced subgraph* of V'. Referring to the picture again, the edges of the subgraph induced by $V' = \{C, B\}$, are $E' = \{d, e, f\}$.



Look again at the above graph. In particular, consider the path c, a with vertex sequence C, A, B. Notice that the edge d has $\phi(d) = \{C, B\}$. The subgraph $G' = (V', E', \phi')$, where $V' = \{C, A, B\}$ and $E' = \{c, a, d\}$ is called a *cycle* of G. In general, whenever there is a path in G, say $e_1, \ldots, e_{n-1}$ with vertex sequence $a_1, \ldots, a_n$, and an edge x with $\phi(x) = \{a_1, a_n\}$, then the subgraph induced by the edges $e_1, \ldots, e_{n-1}, x$ is called a *cycle* of G. Parallel edges like a and b in the preceding figure induce a cycle. A loop also induces a cycle. $\square$

The formal definition of a cycle is:

Definition 9 (Circuit and Cycle) *Let $G = (V, E, \phi)$ be a graph and let $e_1, \ldots, e_n$ be a trail with vertex sequence $a_1, \ldots, a_n, a_1$. (It returns to its starting point.) The subgraph G' of G induced by the set of edges $\{e_1, \ldots, e_n\}$ is called a* circuit *of G. The* length *of the circuit is n.*

- *If the only repeated vertices on the trail are a_1 (the start and end), then the circuit is called a* simple circuit *or* cycle.
- *If "trail" is replaced by directed trail, we obtain a* directed circuit *and a* directed cycle.

In our definitions, a path is a *sequence* of edges but a cycle is a *subgraph* of G. In actual practice, people often think of a cycle as a path, except that it starts and ends at the same vertex. This sloppiness rarely causes trouble, but can lead to problems in formal proofs. Cycles are closely related to the existence of multiple paths between vertices:

Theorem 3 (Cycles and multiple paths) *Two vertices $u \neq v$ are on a cycle of G if and only if there are at least two paths from u to v that have no vertices in common except the endpoints u and v.*

Proof: Suppose u and v are on a cycle. Follow the cycle from u to v to obtain one path. Then follow the cycle from v to u to obtain another. Since a cycle has no repeated vertices, the only vertices that lie in both paths are u and v. On the other hand, a path from u to v followed by a path from v to u is a cycle if the paths have no vertices in common other than u and v. $\square$

One important feature of a graph is whether or not any pair of vertices can be connected by a path. You can probably imagine, without much difficulty, applications of graph theory where this sort of "connectivity" is important. Not the least of such examples would be communication networks. Here is a formal definition of *connected* graphs.

Definition 10 (Connected graph) *Let $G = (V, E, \phi)$ be a graph. If for any two distinct elements u and v of V there is a path P from u to v then G is a connected graph. If $|V| = 1$, then G is connected.*

We make two observations about the definition.

- Because of Theorem 2, we can replace "path" in the definition by "walk" or "trail" if we wish. (This observation is used in the next example.)
- The last sentence in the definition is not really needed. To see this, suppose $|V| = 1$. Now G is connected if, for any two *distinct* elements u and v of V there is a path from u to v. This is trivially satisfied since we cannot find two distinct elements in the one element set V.

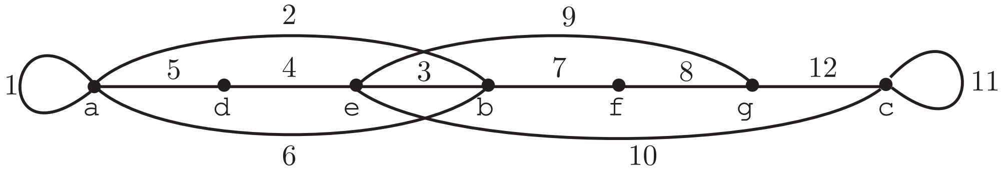
The graph of Example 1 has two distinct "pieces." It is not a connected graph. There is, for example, no path from $u = TM$ to $v = CS$. Note that one piece of this graph consists of the vertex induced subgraph of the vertex set $\{CS, EN, SH, RL\}$ and the other piece consists of the vertex induced subgraph of $\{TM, SE, MN, SM\}$. These pieces are called *connected components* of the graph. This is the case in general for a graph $G = (V, E, \phi)$: The vertex set is partitioned into subsets $V_1, V_2, \ldots, V_m$ such that if u and v are in the same subset then there is a path from u to v and if they are in different subsets there is no such path. The subgraphs $G_1 = (V_1, E_1, \phi_1), \ldots, G_m = (V_m, E_m, \phi_m)$ induced by the sets $V_1, \ldots, V_m$ are called the *connected components* of G. Every edge of G appears in one of the connected components. To see this, suppose that $\{u, v\}$ is an edge and note that the edge is a path from u to v and so u and v are in the same induced subgraph, G_i. By the definition of induced subgraph, $\{u, v\}$ is in G_i.

Example 11 (Connected components as an equivalence relation) You may have noticed that the "definition" that we have given of connected components is a bit sloppy: We need to know that the partitioning into such subsets can actually occur. To see that this is not trivially obvious, define two integers to be "connected" if they have a common factor. Thus 2 and 6 are connected and 3 and 6 are connected, but 2 and 3 are not connected and so we cannot partition the set $V = \{2, 3, 6\}$ into "connected components". We must use some property of the definition of graphs and paths to show that the partitioning of vertices is possible. One way to do this is to construct an equivalence relation.

For $u, v \in V$, write $u \sim v$ if and only if either $u = v$ or there is a walk from u to v. It is clear that $\sim$ is reflexive and symmetric. We now prove that it is transitive. Let $u \sim v \sim w$. The walk from u to v followed by the walk from v to w is a walk from u to w. This completes the proof that $u \sim v$ is an equivalence relation. The relation partitions V into subsets $V_1, \ldots, V_m$. By Theorem 2, the vertex induced subgraphs of the V_i satisfy Definition 10. □

When talking about connectivity, graphs and digraphs are different. In a digraph, the fact that there is a directed walk from u to v does not, in general, imply that there is a directed walk from v to u. Thus, the "directed walk relation", unlike the "walk relation" is not symmetric. This complicates the theory of connectivity for digraphs.

Example 12 (Eulerian graphs) We are going to describe a process for constructing a graph $G = (V, E, \phi)$ (with loops allowed). Start with $V = \{v_1\}$ consisting of a single vertex and with $E = \emptyset$. Add an edge e_1, with $\phi(e_1) = \{v_1, v_2\}$, to E. If $v_1 = v_2$, we have a graph with one vertex and one edge (a loop), else we have a graph with two vertices and one edge. Keep track of the vertices and edges in the order added. Here (v_1, v_2) is the sequence of vertices in the order added and (e_1) is the sequence of edges in order added. Suppose we continue this process to construct a sequence of vertices (not necessarily distinct) and sequence of *distinct* edges. At the point where k distinct edges have been added, if v is the last vertex added, then we add a new edge e_{k+1}, different from all previous edges, with $\phi(e_{k+1}) = \{v, v'\}$ where either v' is a vertex already added or a new vertex. Here is a picture of this process carried out with the edges numbered in the order added



where the vertex sequence is

$$S = (a, a, b, e, d, a, b, f, g, e, c, c, g).$$

Such a graph is called a graph with an *Eulerian trail*. The edges, in the order added, are the Eulerian trail and S is the vertex sequence of the trail

By construction, if G is a graph with an *Eulerian trail*, then there is a trail in G that includes every edge in G. If there is a circuit in G that includes every edge of G then G is called an *Eulerian circuit graph* or graph with an *Eulerian circuit*. Thinking about the above example, if a graph has an Eulerian trail but no Eulerian circuit, then all vertices of the graph have even degree except the start vertex (a in our example with degree 5) and end vertex (g in our example with degree 3). If a graph has an Eulerian circuit then all vertices have even degree. The converses in each case are also true (but take a little work to show): If G is a connected graph in which every vertex has even degree then G has an Eulerian circuit. If G is a connected graph with all vertices but two of even degree, then G has an Eulerian trail joining the two vertices of odd degree. $\square$

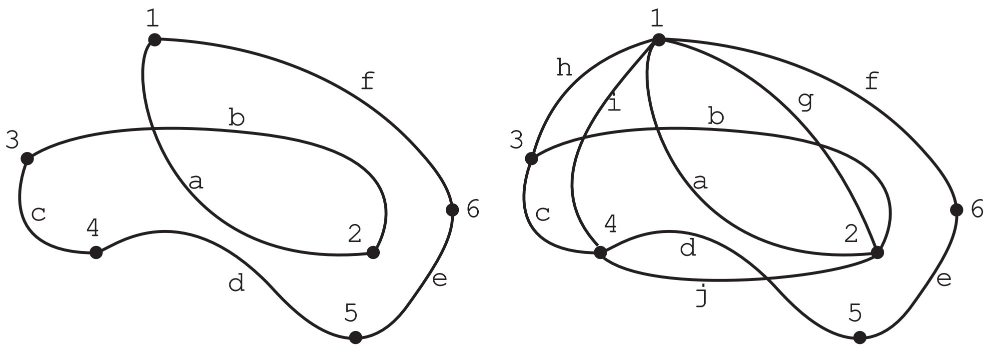
Here is a precise definition of Eulerian trail and circuit.

Definition 11 (Eulerian trail, circuit) *Let $G = (V, E, \phi)$ be a connected graph. If there is a trail with edge sequence $(e_1, e_2, \ldots, e_k)$ in G which uses each edge in E, then $(e_1, e_2, \ldots, e_k)$ is called an Eulerian trail. If there is a circuit $C = (V', E', \phi')$ in G with $E' = E$, then C is called an Eulerian circuit.*

The ideas of a directed Eulerian circuit and directed Eulerian trail for directed graphs are defined in exactly the same manner.

An Eulerian circuit in a graph contains every edge of that graph. What about a cycle that contains every vertex but not necessarily every edge? Our next example discusses that issue.

Example 13 (Hamiltonian cycle) Start with a graph $G' = (V, E', \phi')$ that is a cycle and then add additional edges, without adding any new vertices, to obtain a graph $G = (V, E, \phi)$. As an example, consider



where the first graph $G' = (V, E', \phi')$ is the cycle induced by the edges $\{a, b, c, d, e, f\}$. The second graph $G = (V, E, \phi)$ is obtained from G' by adding edges g, h, i and j. A graph that can be constructed from such a two-step process is called a *Hamiltonian graph.* The cycle G' is called a *Hamiltonian cycle for G.*

Definition 12 (Hamiltonian cycle, Hamiltonian graph) *A cycle in a graph $G = (V, E, \phi)$ is a Hamiltonian cycle for G if every element of V is a vertex of the cycle. A graph $G = (V, E, \phi)$ is Hamiltonian if it has a subgraph that is a Hamiltonian cycle for G.*

Notice that an Eulerian circuit uses every edge exactly once and a Hamiltonian cycle uses every vertex exactly once. We gave a very simple characterization of when a graph has an Eulerian circuit (in terms of degrees of vertices). There is no simple characterization of when a graph has a Hamiltonian cycle. On the contrary, the issue of whether or not a graph has a Hamiltonian cycle is notoriously difficult to resolve in general.

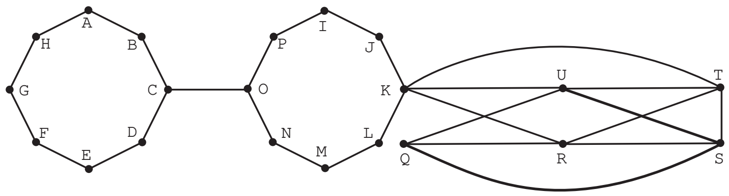
As we already mentioned, connectivity issues in digraphs are much more difficult than in graphs. A digraph is *strongly connected* if, for every two vertices v and w there is a directed path from v to w. From any digraph D, we can construct a simple graph $S(D)$ on the same set of vertices by letting $\{v, w\}$ be an edge of $S(D)$ if and only if at least one of (u, v) and (v, u) is an edge of D. You should be able to show that if D is strongly connected then $S(D)$ is connected. The converse is false. As an example, take $D = (V, E)$ to be the simple digraph where $V = \{1, 2\}$ and $E = \{(1, 2)\}$. There is no directed path from 2 to 1, but clearly $S(D) = (V, \{\{1, 2\}\})$ is connected.

Other issues for digraphs analogous to those for graphs work out pretty well, but are more technical. An example is the notion of degree for vertices. For any subset U of the vertices V of a directed graph $D = (V, E)$, define $d_{\text{in}}(U)$ to be the number of edges of e of D with $\phi(e)$ of the form (w, u) where $u \in U$ and $w \notin U$. Define $d_{\text{out}}(U)$ similarly. If $U = \{v\}$ consists of just one vertex, $d_{\text{in}}(U)$ is usually written simply as $d_{\text{in}}(v)$ rather than the more technically correct $d_{\text{in}}(\{v\})$. Similarly, we write $d_{\text{out}}(v)$. You should compute $d_{\text{in}}(v)$ and $d_{\text{out}}(v)$ for the vertices v of the graph of Example 9. You should be able to show that $\sum d_{\text{in}}(v) = \sum d_{\text{out}}(v) = |E|$, where the sums range over all $v \in V$. See the Exercises for Section 1 for the idea.

Example 14 (Bicomponents of graphs) Let $G = (V, E, \phi)$ be a graph. For $e, f \in E$ write $e \sim f$ if either $e = f$ or there is a cycle of G that contains both e and f. We claim that this is an equivalence relation. The reflexive and symmetric parts are easy. Suppose that $e \sim f \sim g$. If $e = g$, then $e \sim g$, so suppose that $e \neq g$. Let $\phi(e) = \{v_1, v_2\}$. Let $C(e, f)$ be the cycle containing e and f and $C(f, g)$ the cycle containing f and g. In $C(e, f)$ there is a path P_1 from v_1 to v_2 that does not contain e. Let x and $y \neq x$ be the first and last vertices on P_1 that lie on the cycle containing f and g. We know that there must be such points because the edge f is on P_1. Let P_2 be the path in $C(e, f)$ from y to x containing e. In $C(f, g)$ there is a path P_3 from x to y containing g. We claim that P_2 followed by P_3 defines a cycle containing e and g.

Some examples may help. Consider a graph that consists of two disjoint cycles that are joined by an edge. There are three bicomponents — each cycle and the edge joining them. Now consider three cycles that are disjoint except for one vertex that belongs to all three of them. Again there are three bicomponents — each of the cycles.

Since $\sim$ is an equivalence relation on the edges of G, it partitions them. If the partition has only one block, then we say that G is a *biconnected graph*. If E' is a block in the partition, the subgraph of G induced by E' is called a *bicomponent* of G. Note that the bicomponents of G are not necessarily disjoint: Bicomponents may have vertices in common (but *never* edges). There are four bicomponents in the following graph. Two are the cycles, one is the edge $\{C, O\}$, and the fourth consists of all of the rest of the edges.



□

Exercises for Section 2

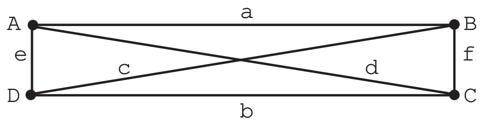
2.1. A graph $G = (V, E)$ is called *bipartite* if V can be partitioned into two sets C and S such that each edge has one vertex in C and one vertex in S. As a specific example, let C be the set of courses at the university and S the set of students. Let $V = C \cup S$ and let $\{s, c\} \in E$ if and only if student s is enrolled in course c.

(a) Prove that $G = (V, E)$ is a simple graph.

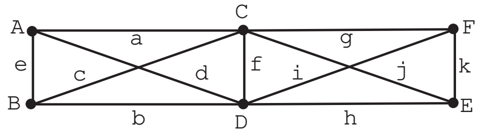
(b) Prove that every cycle of G has an even number of edges.

2.2. In each of the following graphs, find the longest trail (most edges) and longest circuit. If the graph has an Eulerian circuit or trail, say so.

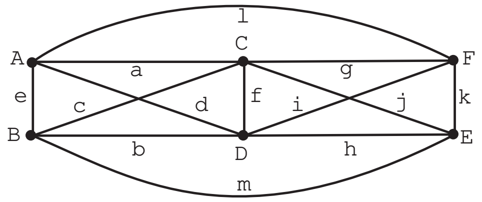
(a)



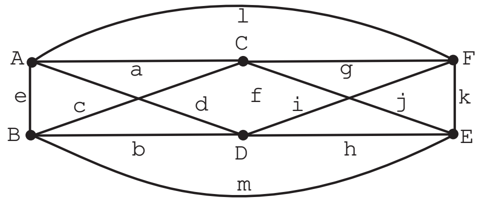
(b)



(c)

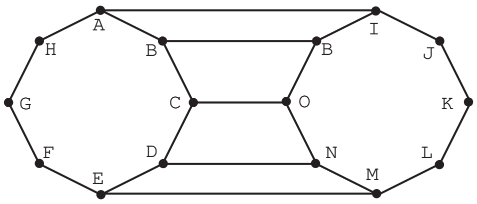


(d)

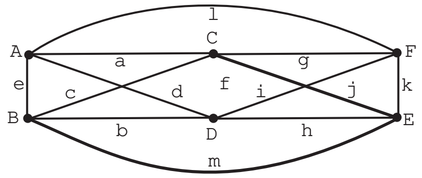


2.3. For each of the following graphs $G = (V, E, \phi)$, find a cycle in G of maximum length. State whether or not the graph is Hamiltonian.

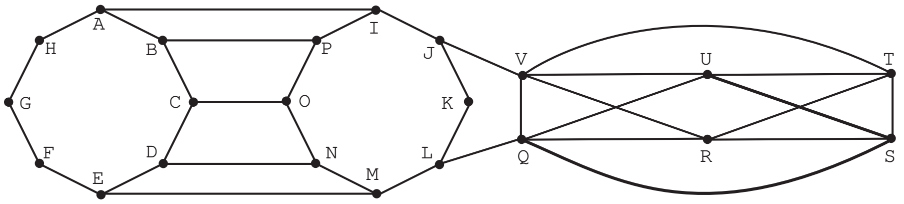
(a)



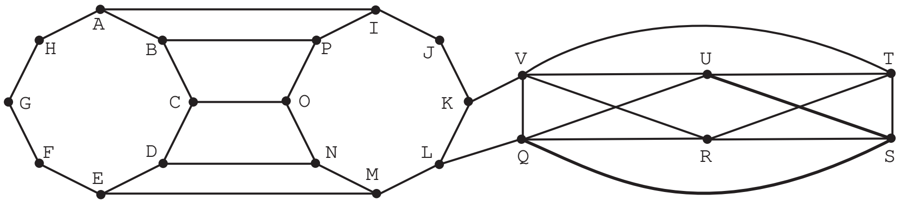
(b)



(c)



(d)



2.4. We are interested in the number of simple digraphs with $V = \underline{n}$

(a) Find the number of them.

(b) Find the number of them with no loops.

(c) In both cases, find the number of them with exactly q edges.

2.5. An *oriented simple graph* is a simple graph which has been converted to a digraph by assigning an orientation to each edge. The orientation of $\{u, v\}$ can be thought of as a mapping of it to either (u, v) or (v, u).

(a) Give an example of a simple digraph that has no loops but is not an oriented simple graph

(b) Find the number of oriented simple digraphs.

(c) Find the number of them with exactly q edges.

2.6. A binary relation R on S is an *order relation* if it is reflexive, antisymmetric, and transitive. R is *antisymmetric* if for all $(x, y) \in R$ with $x \neq y$, $(y, x) \notin R$. Given an order relation R, the covering relation H of R consists of all $(x, z) \in R$, $x \neq z$, such that there is no y, distinct from both x and z, such that $(x, y) \in R$ and $(y, z) \in R$. A pictorial representation of the covering relation as a directed graph is called a "Hasse diagram" of H.

(a) Show that the divides relation on

$$S = \{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16\}$$

is an order relation. By definition, (x, y) is in the divides relation on S is x is a factor of y. Thus, $(4, 12)$ is in the divides relation. $x|y$ is the standard notation for x is a factor of y.

(b) Find and draw a picture of the directed graph of the covering relation of the divides relation.
Hint: You must find all pairs $(x, z) \in S \times S$ such that $x|z$ but there does not exist any y, $x < y < z$, such that $x|y$ and $y|z$.

Section 3: Trees

Trees play an important role in a variety of algorithms. We have used decision trees to enhance our understanding of recursion. In this section, we define trees precisely and look at some of their properties.

Definition 13 (Tree) *If G is a connected graph without any cycles then G is called a tree. (If $|V| = 1$, then G is connected and hence is a tree.) A tree is also called a free tree.*

The graph of Example 2 is connected but is not a tree. It has many cycles, including $(\{A, B, C\}, \{a, e, c\})$. The subgraph of this graph induced by the edges $\{a, e, g\}$ is a tree. If

G is a tree, then ϕ is an injection since if $e_1 \neq e_2$ and $\phi(e_1) = \phi(e_2)$, then $\{e_1, e_2\}$ induces a cycle. In other words, any graph with parallel edges is not as tree. Likewise, a loop is a cycle, so a tree has no loops. Thus, we can think of a tree as a simple graph when we are not interested in names of the edges.

Since the notion of a tree is so important, it will be useful to have some equivalent definitions of a tree. We state them as a theorem

Theorem 4 (Alternative definitions of a tree) *If G is a connected graph, the following are equivalent.*

(a) G is a tree.

(b) G has no cycles.

(c) For every pair of vertices $u \neq v$ in G, there is exactly one path from u to v.

(d) Removing any edge from G gives a graph which is not connected.

(e) The number of vertices of G is one more than the number of edges of G.

Proof: We are given that G is connected, thus, by the definition of a tree, (a) and (b) are equivalent.

Theorem 3 can be used to prove that (b) implies (c). We leave that as an exercise (show **not** (c) implies **not** (b)).

If $\{u, v\}$ is an edge, it follows from (c) that the edge is the only path from u to v and so removing it disconnects the graph. Hence (c) implies (d).

We leave it as an exercise to prove that (d) implies (b) (show **not** (b) implies **not** (d)).

Thus far, we have shown (a) and (b) are equivalent, and we have shown that (b) implies (c) implies (d) implies (b), so (a), (b), (c), and (d) are all equivalent. All that remains is to include (e) in this equivalence class of statements. Do do this, all we have to do is show that (e) implies any of the equivalent statements (a), (b), (c), and (d) and, conversely, some one of (a), (b), (c), and (d) implies (e). We shall show that (b) implies (e) and that (e) implies (a).

We first show that (b) implies (e). We will use induction on the number of vertices of G. If G has one vertex, it has no edges and (e) is satisfied. Otherwise, we claim that G has a vertex u of degree 1; that is, it lies on only one edge $\{u, w\}$. We prove this claim shortly. Remove u and $\{u, v\}$ to obtain a graph H with one less edge and one less vertex. Since G is connected and has no cycles, the same is true of H. By the induction hypothesis, H has one less edge than vertex. Since we got from G to H by removing one vertex and one edge, G must also have one less edge than vertex. By induction, the proof is done. It remains to prove the existence of u. Suppose no such u exists; that is, suppose that each vertex lies on at least two edges. We will derive a contradiction. Start at any vertex v_1 of G leave v_1 by some edge e_1 to reach another vertex v_2. Leave v_2 by some edge e_2 different from the edge used to reach v_2. Continue with this process. Since each vertex lies on at least two edges, the process never stops. Hence we eventually repeat a vertex, say

$$v_1, e_1, v_2, \ldots, v_k, e_k, \ldots, v_n, e_n, v_{n+1} = v_k.$$

The edges $e_k, \ldots, e_n$ form a cycle, which is a contradiction.

Having shown that (b) implies (e), we now show that (e) implies (a). We use the contrapositive and show that **not** (a) implies **not** (e). Thus we assume G is not a tree. Hence, by (d) we can remove an edge from G to get a new graph which is still connected. If this is not a tree, repeat the process and keep doing so until we reach a tree T. For a tree T, we trivially satisfy (a) which implies (b) and (b) implies (e). Thus, the number of vertices is now one more than the number of edges in the graph T. Since, in going from G to T, we removed edges from G but did not remove vertices, G must have at least as many edges as vertices. This shows **not** (a) implies **not** (e) and completes the proof. $\square$

Definition 14 (Forest) *A forest is a graph all of whose connected components are trees. In particular, a forest with one component is a tree. (Connected components were defined following Definition 10.)*

Example 15 (A relation for forests) Suppose a forest has v vertices, e edges and c (connected) components. What values are possible for the triple of numbers (v, e, c)? It might seem at first that almost anything is possible, but this is not so. In fact $v - c = e$ because of Theorem 4(e). Why? Let the forest consist of trees $T_1, \ldots, T_c$ and let the triples for T_i be (v_i, e_i, c_i). Since a tree is connected, $c_i = 1$. By the theorem, $e_i = v_i - 1$. Since $v = v_1 + \cdots + v_c$ and $e = e_1 + \ldots + e_c$ we have

$$e \;=\; (v_1 - 1) + (v_2 - 1) + \cdots + (v_c - 1) \;=\; (v_1 + \cdots + v_c) - c \;=\; v - c.$$

Suppose a forest has $e = 12$ and $v = 15$. We know immediately that it must be made up of three trees because $c = v - e = 15 - 12$.

Suppose we know that a graph $G = (V, E, \phi)$ has $v = 15$ and $c = 3$, what is the fewest edges it could have? For each component of G, we can remove edges one by one until we cannot remove any more without breaking the component into two components. At this point, we are left with each component a tree. Thus we are left with a forest of $c = 3$ trees that still has $v = 15$ vertices. By our relation $v - c = e$, this forest has 12 edges. Since we may have removed edges from the original graph to get to this forest, the original graph has at least 12 edges.

What is the maximum number of edges that a graph $G = (V, E, \phi)$ with $v = 15$ and $c = 3$ could have? Since we allow multiple edges, a graph could have an arbitrarily large number of edges for a fixed v and c — if e is an edge with $\phi(e) = \{u, v\}$, add in as many edges e_i with $\phi(e_i) = \{u, v\}$ as you wish. Hence we will have to insist that G be a simple graph.

What is the maximum number of edges that a simple graph G with $v = 15$ and $c = 3$ could have? This is a bit trickier. Let's start with a graph where c is not specified. The edges in a simple graph are a subset of $\mathcal{P}_2(V)$ and since $\mathcal{P}_2(V)$ has $\binom{v}{2}$ elements, a simple graph with v vertices has at most $\binom{v}{2}$ edges.

Now let's return to the case when we know there must be three components in our simple graph. Suppose the number of vertices in the components are v_1, v_2 and v_3. Since there are no edges between components, we can look at each component by itself. Using the result in the previous paragraph for each component, the maximum number of possible edges is $\binom{v_1}{2}+\binom{v_2}{2}+\binom{v_3}{2}$. We don't know v_1, v_2, v_3. All we know is that they are strictly positive integers that sum to v. It turns out that the maximum occurs when one of v_i is as large as possible and the others equal 1, but the proof is beyond this course. Thus the answer is $\binom{v-2}{2}$, which in our case is $\binom{13}{2}=78$. In general, if there were c components, $c-1$ components would have one vertex each and the remaining component would have $v-(c-1)=v+1-c$ vertices. Hence there can be no more than $\binom{v+1-c}{2}$ edges.

Reviewing what we've done, we see:

- There is no graph $G=(V,E,\phi)$ with $v-c>e$.
- If $v-c=e$, the graph is a forest of c trees and any such forest will do as an example.
- If $v-c<e$, there are many examples, none of which are forests.
- If $v-c<e$ and we have a *simple* graph, then we must have $e \le \binom{v+1-c}{2}$. □

Recall that decision trees, as we have used them, have some special properties. First, they have a starting point. Second, the edges (decisions) out of each vertex are ordered. We now formalize these concepts.

Definition 15 (Rooted graph) *A pair (G,v), consisting of a graph $G=(V,E,\phi)$ and a specified vertex v, is called a rooted graph with root v.*

Definition 16 (Parent, child, sibling and leaf) *Let (T,r) be a rooted tree. If w is any vertex other than r, let $r=v_0,v_1,\ldots,v_k,v_{k+1}=w$, be the list of vertices on the unique path from r to w. We call v_k the parent of w and call w a child of v_k. Parents and children are also called fathers and sons. Vertices with the same parent are siblings. A vertex with no children is a leaf. All other vertices are internal vertices of the tree.*

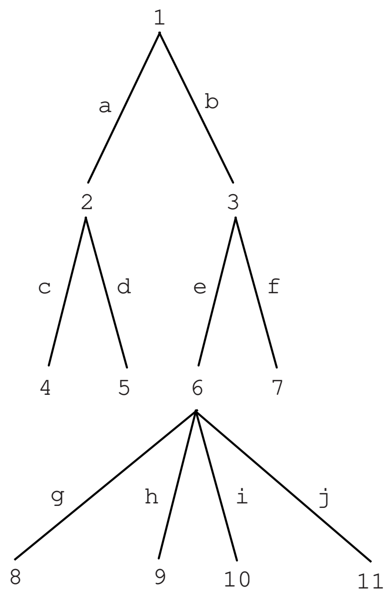
Definition 17 (Rooted plane tree) *Let (T,r) be a rooted tree. For each vertex, order the children of the vertex. The result is a rooted plane tree, which we abbreviate to RP-tree. RP-trees are also called ordered trees. An RP-tree is also called, in certain contexts, a decision tree, and, when there is no chance of misunderstanding, simply a tree.*

Since almost all trees in computer science are rooted and plane, computer scientists usually call a rooted plane tree simply a tree. It's important to know what people mean!

Example 16 (A rooted plane tree) Below is a picture of a rooted plane tree $T=(V,E,\phi)$. In this case $V=\underline{11}$ and $E=\{a,\ldots,j\}$. There are no parallel edges or loops, as required by the definition of a RP-tree. The root is $r=1$. For each vertex, there is a unique path from the root to that vertex. Since ϕ is an injection, once ϕ has been defined (as it is in the picture), that unique path can be specified by the vertex sequence alone. Thus, the path from the root to 6 is $(1,3,6)$. The path from the root to 9 is $(1,3,6,9)$. Sometimes computer scientists refer to the path from the root to a vertex v as the "stack" of v.

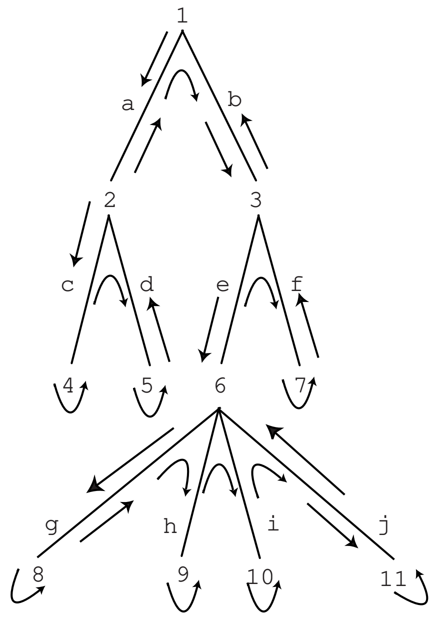
In the tree below, the vertex 6 is the parent of the vertex 9. The vertices 8, 9, 10, and 11 are the children of 6 and, they are siblings of each other. The leaves of the tree are 4, 5, 7, 8, 9, 10, and 11. All other vertices (including the root) are internal vertices of the tree.

Remember, an RP-tree is a tree with added properties. Therefore, it must satisfy (a) through (e) of Theorem 4. In particular, T has no cycles. Also, there is a unique path between any two vertices (e.g., the path from 5 to 8 is $(5, 2, 1, 3, 6, 8)$). Removing any edge gives a graph which is not connected (e.g., removing j disconnects T into a tree with 10 vertices and a tree with 1 vertex; removing e disconnects T into a tree with 6 vertices and one with 5 vertices). Finally, the number of edges (10) is one less than the number of vertices.



□

Example 17 (Traversing a rooted plane tree) Just as in the case of decision trees, one can define the notion of *depth first* traversals of a RP-tree.



Imagine going around ("traversing") the above RP-tree following arrows. Start at the root, 1, go down edge a to vertex 2, etc. Here is the sequence of vertices as encountered in this process: 1, 2, 4, 2, 5, 2, 1, 3, 6, 8, 6, 9, 6, 10, 6, 11, 6, 3, 7, 3, 1. This sequence of vertices is called the *depth first vertex sequence*, DFV(T), of the RP-tree T. The number of times each vertex appears in DFV(T) is one plus the number of children of that vertex. For edges, the corresponding sequence is a, c, c, d, d, a, b, e, g, g, h, h, i, i, j, j, e, f, f, b. This sequence is the *depth first edge sequence*, DFE(T), of the tree. Every edge appears exactly twice in DFE(T). If the vertices of the RP-tree are read left to right, top to bottom, we obtain the sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11. This is called the *breadth first vertex sequence*, BFV(T). Similarly, the *breadth first edge sequence*, BFE(T), is a, b, c, d, e, f, g, h, i, j.

The sequences BFV(T) and BFE(T) are linear orderings of the vertices and edges of the RP-tree T (i.e., each vertex or edge appears exactly once in the sequence). We also associate linear orderings with DFV(T) called the *preorder sequence of vertices* of T, PREV(T), and the *postorder sequence of vertices* of T, POSV(T).

$\mathrm{PREV}(T) = 1, 2, 4, 5, 3, 6, 8, 9, 10, 11, 7$ is the sequence of *first* occurrences of the vertices of T in DFV(T).

$\mathrm{POSV}(T) = 4, 5, 2, 8, 9, 10, 11, 6, 7, 3, 1$ is the sequence of *last* occurrences of the vertices of T in DFV(T).

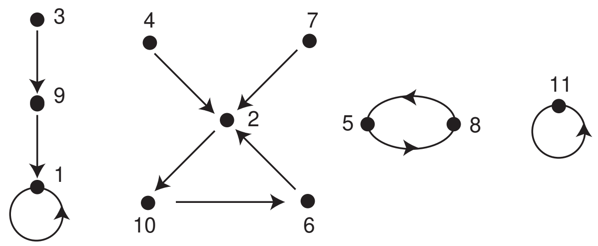
Notice that the order in which the leaves of T appear, 4, 5, 8, 9, 10, 11, is the same in both PREV(T) and POSV(T). Can you see why this is always true for any tree? $\square$

***Example 18 (The number of labeled trees)** How many n-vertex labeled trees are there? In other words, count the number of trees with vertex set $V = \underline{n}$. The answer has been obtained in a variety of ways. We will do it by establishing a correspondence between trees and functions by using digraphs.

Suppose f is a function from V to V. We can represent this as a simple digraph (V, E) where the edges are $\{(v, f(v)) \mid v \in V\}$. The function

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 1 & 10 & 9 & 2 & 8 & 2 & 2 & 5 & 1 & 6 & 11 \end{pmatrix}$$

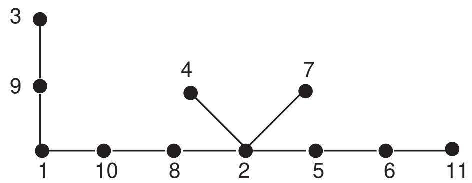
corresponds to the directed graph



Such graphs are called *functional digraphs*. You should be able to convince yourself that a functional digraph consists of cycles (including loops) with each vertex on a cycle being the root of a tree of noncyclic edges. The edges of the trees are directed toward the roots. In the previous figure,

- 1 is the root of the tree with vertex set $\{1, 3, 9\}$,
- 2 is the root of the tree with vertex set $\{2, 4, 7\}$,
- 5 is the root of the tree with vertex set $\{5\}$,
- 6 is the root of the tree with vertex set $\{6\}$,
- 8 is the root of the tree with vertex set $\{8\}$,
- 10 is the root of the tree with vertex set $\{10\}$ and
- 11 is the root of the tree with vertex set $\{11\}$.

In a tree, there is a unique path from the vertex 1 to the vertex n. Remove all the edges on the path and list the vertices on the path, excluding 1 and n, in the order they are encountered. Interpret this list as a permutation in 1 line form. Draw the functional digraph for the cycle form, adding the cycles (1) and (n). Add the trees that are attached to each of the cycle vertices, directing their edges toward the cycle vertices. Consider the following figure.



The one line form is $10, 8, 2, 5, 6$. In two line form it is $\begin{pmatrix} 2 & 5 & 6 & 8 & 10 \\ 10 & 8 & 2 & 5 & 6 \end{pmatrix}$. Thus the cycle form is (2,10,6)(5,8). When we add the two cycles (1) and (11) to this, draw the directed graph, and attach the directed trees, we obtain the functional digraph pictured earlier.

We leave it to you to convince yourself that this gives us a one-to-one correspondence between trees with $V = \underline{n}$ and functions $f : \underline{n} \to \underline{n}$ with $f(1) = 1$ and $f(n) = n$. In creating such a function, there are n choices for each of $f(2), \ldots, f(n-1)$. Thus there are n^{n-2} such functions and hence n^{n-2} trees. $\square$

Spanning Trees

Trees are not only important objects of study per se, but are important as special subgraphs of general graphs. A *spanning tree* is one such subgraph. For notational simplicity, we shall restrict ourselves to simple graphs, $G = (V, E)$, in the following discussion. The ideas we discuss extend easily to graphs $G = (V, E, \phi)$, even allowing loops.

Definition 18 (Spanning tree) *A spanning tree of a (simple) graph $G = (V, E)$ is a subgraph $T = (V, E')$ which is a tree and has the same set of vertices as G.*

Example 19 (Connected graphs and spanning trees) Since a tree is connected, a graph with a spanning tree must be connected. On the other hand, it is not hard to see that every connected graph has a spanning tree. Any simple graph $G = (V, E)$ has a subgraph that is a tree, $T' = (V', E')$. Take $V' = \{v\}$ to be one vertex and E' empty. Suppose that $T' = (V', E')$ is the largest such "subtree." If T' is not a spanning tree then there is a vertex w of G that is not a vertex of T'. If G is connected, choose a vertex u in T' and a path $w = x_1, x_2, \ldots, e_k = u$ from w to u. Let j, $1 < j \le k$, be the first integer such that x_j is a vertex of T'. Then adding the edge $\{x_{j-1}, x_j\}$ and the vertex x_{j-1} to T' creates a subtree T of G that is larger than T', a contradiction of the maximality of T'. We have, in fact, shown that a graph is connected if and only if every maximal subtree is a spanning tree. Thus we have: A graph is connected if and only if it has a spanning tree. It follows that, if we had an algorithm that was guaranteed to find a spanning tree whenever such a tree exists, then this algorithm could be used to decide if a graph is connected. □

Example 20 (Minimum spanning trees) Suppose we wish to install "lines" to link various sites together. A site may be a computer installation, a town, or a factory. A line may be a digital communication channel, a rail line or, a shipping route for supplies. We'll assume that

(a) a line operates in both directions;

(b) it must be possible to get from any site to any other site using lines;

(c) each possible line has a cost (rental rate, construction cost, or shipping cost) independent of each other line's cost;

(d) we want to choose lines to minimize the total cost.

We can think of the sites as vertices V in a (simple) graph, the possible lines as edges E and the costs as a function λ from the edges to the positive real numbers. Because of (a) and (b), the lines $E' \subseteq E$ we actually choose will be such that $T = (V, E')$ is connected. Because of (d), T will be a spanning tree since, if it had more edges, we could delete some, but if we delete any from a tree it will not be connected by Theorem 4. □

We now formalize these ideas in a definition:

Definition 19 (Weights in a graph) *Let $G = (V, E)$ be a simple graph and let λ be a function from E to the positive real numbers. We call $\lambda(e)$ the weight of the edge e. If $H = (V', E')$ is a subgraph of G, then $\lambda(H)$, the weight of H, is the sum of $\lambda(e')$ over all $e' \in E'$.*

A minimum weight spanning tree for a connected graph G is a spanning tree such that $\lambda(T) \le \lambda(T')$ whenever T' is another spanning tree.

How can we find a minimum weight spanning tree T? One approach is to construct T by adding an edge at a time in a greedy way. Since we want to minimize the weight, "greedy" means keeping the weight of each edge we add as low as possible. Here's such an algorithm.

Theorem 5 (Minimum weight spanning tree: Prim's algorithm) *Let $G = (V, E)$ be a simple graph with edge weights given by λ. If the algorithm stops with $V' \ne V$, G has no spanning tree; otherwise, (V, E') is a minimum weight spanning tree for G.*

1. **Start:** *Let* $E' = \emptyset$ *and let* $V' = \{v_0\}$ *where* v_0 *is any vertex in* V*.*
2. **Possible Edges:** *Let* $F \subseteq E$ *be those edges* $f = \{x, y\}$ *with one vertex in* V' *and one vertex not in* V'*. If* $F = \emptyset$*, stop.*
3. **Choose Edge Greedily:** *Let* $f = \{x, y\}$ *be such that* $\lambda(f)$ *is a minimum over all* $f \in F$*. Replace* V' *with* $V' \cup \{x, y\}$ *and* E' *with* $E' \cup \{f\}$*. Go to Step 2.*

Proof: We begin with the first part; i.e, if the algorithm stops with $V' \neq V$, then G has no spanning tree. The argument is similar to that used in Example 19. Suppose that $V' \neq V$ and that there is a spanning tree. We will prove that the algorithm does not stop at V'. Choose $u \in V - V'$ and $v \in V'$. Since G is connected, there must be a path from u to v. Each vertex on the path is either in V' or not. Since $u \notin V'$ and $v \in V'$, there must be an edge f on the path with one end in V' and one end not in V'. But then $f \in F$ and so the algorithm does not stop at V'.

We now prove that, if G has a spanning tree, then (V, E') is a minimum weight spanning tree. One way to do this is by induction: We will prove that at each step there is a minimum weight spanning tree of G that contains E'.

The starting case for the induction is the first step in the algorithm; i.e., $E' = \emptyset$. Since G has a spanning tree, it must have a minimum weight spanning tree. The edges of this tree obviously contain the empty set, which is what E' equals at the start.

We now carry out the inductive step of the proof. Let V' and E' be the values going into Step 3 and let $f = \{x, y\}$ be the edge chosen there. By the induction hypothesis, there is a minimum weight spanning tree T of G that contains the edges E'. If it also contains the edge f, we are done. Suppose it does not contain f. We will prove that we can replace an edge in the minimum weight tree with f and still achieve minimum weight.

Since T contains all the vertices of G, it contains x and y and, also, some path P from x to y. Suppose $x \in V'$ and $y \notin V'$, this path must contain an edge $e = \{u, v\}$ with $u \in V'$ and $v \notin V'$. We now prove that removing e from T and then adding f to T will still give a minimum spanning tree.

By the definition of F in Step 2, $e \in F$ and so, by the definition of f, $\lambda(e) \geq \lambda(f)$. Thus the weight of the tree does not increase. If we show that the result is still a tree, this will complete the proof.

The path P together with the edge f forms a cycle in G. Removing e from P and adding f still allows us to reach every vertex in P and so the altered tree is still connected. It is also still a tree because it contains no cycles — adding f created only one cycle and removing e destroyed it. This completes the proof that the algorithm is correct. $\square$

The algorithm for finding a minimum weight spanning tree that we have just proved is sometimes referred to as *Prim's Algorithm.* A variation on this algorithm, proved in a similar manner, is called *Kruskal's algorithm.* In Kruskal's algorithm, step 2 of Prim's algorithm is changed to

$2'$. **Possible Edges:** Let $F \subseteq E$ be those edges $f = \{x, y\}$ where x and y do not belong to the same component of (V, E'). If $F = \emptyset$, stop.

Intuitively, $f \notin F$ if f forms a cycle with any collection of edges from E'. Otherwise, $f \in F$. This extra freedom is sometimes convenient. Our next example gives much less freedom in choosing new edges to add to the spanning tree, but produces a type of spanning tree that is useful in many algorithms applicable to computer science.

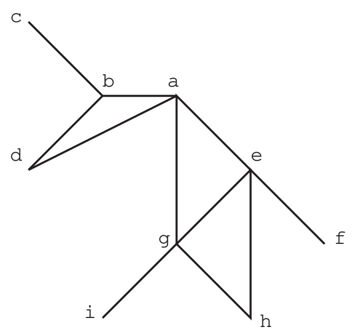
Example 21 (Algorithm for lineal or depth-first spanning trees) We start with a rooted simple graph $G = (V, E)$ with v_0 as root. The algorithmic process constructs a spanning tree rooted at v_0. It follows the same general form as Theorem 5. The weights, if there, are ignored.

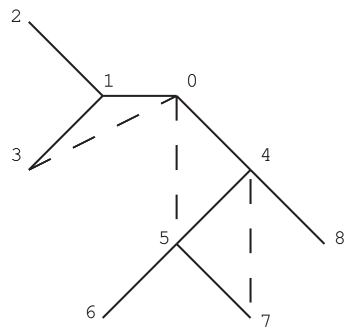
1. **Start:** Let $E' = \emptyset$ and let $V' = \{v_0\}$ where v_0 is the root of G. Let $T' = (V', E')$ be the starting subtree, rooted at v_0.

2. **Possible New Edge:** Let v be the last vertex added to V' where $T' = (V', E')$ is the subtree thus far constructed, with root v_0. Let x be the first vertex on the unique path from v to v_0 for which there is an edge $f = \{x, y\}$ with $x \in V'$ and $y \notin V'$. If there is no such x, stop.

3. **Add Edge :** Replace V' with $V' \cup \{y\}$ and E' with $E' \cup \{f\}$ to obtain $T' = (V', E')$ as the new subtree thus far constructed, with root v_0. (Note: y is now the last vertex added to V'.) Go to Step 2.

Here is an example. We are going to find a lineal spanning tree for the graph below, root a. The result is shown on the right where the original vertices have been replaced by the order in which they have been added to the "tree thus far constructed" in the algorithm.





When there is a choice, we choose the left or upward vertex. For example, at the start, when b, d, e and g are all allowed, we choose b. When vertex 2 was added, the path to the root was $(2, 1, 0)$. We went along this path towards the root and found that at 1, a new edge to 3 could be added. Now the path to the root became $(3, 1, 0)$ and we had to go all of the way to 0 to add a new edge (the edge to 4). You should go through the rest of the algorithm. Although there are some choices, the basic rule of step 2 of the algorithm must always be followed.

There are two extremely important properties that this algorithm has

1. When the rooted spanning tree T for G has been constructed, there may be edges of G not in the spanning tree. In the above picture, there are three such edges, indicated by dashed lines. If $\{x, y\}$ is such an edge, then either x lies on the path from y to the root or the other way around. For example, the edge $\{4, 7\}$ in the example has 4 on the path from 7 to the root 0. This is the "lineal" property from which the spanning trees of this class get their name.

2. If, when the rooted spanning tree T has been constructed, the vertices of T are labeled in the order added by the algorithm **AND** the children of each vertex of T are ordered by the same numbering, then an RP-tree is the result. For this RP tree, the numbers on the vertices correspond to preorder, PREV(T), of vertices on this tree (starting with the root having value 0). Check this out for the above example.

We will not prove that the algorithm we have presented has properties 1 and 2. We leave it to you to study the example, construct other examples, and come to an intuitive understanding of these properties. $\square$

Property 1 in the preceding example is the basis for the formal definition of a lineal spanning tree:

Definition 20 (Lineal or depth-first spanning tree) *Let x and y be two vertices in a rooted tree with root r. If x is on the path connecting r to y, we say that y is a descendant of x. (In particular, all vertices are descendants of r.) If one of u and v is a descendant of the other, we say that $\{u, v\}$ is a lineal pair. A lineal spanning tree or depth-first spanning tree of a connected graph $G = (V, E)$ is a rooted spanning tree of G such that each edge $\{u, v\}$ of G is a lineal pair.*

In our example, vertices $\{6, 7\}$ are not a lineal pair relative to the rooted tree constructed. But $\{4, 7\}$, which is an edge of G, is a lineal pair. Trivially, the vertices of any edge of the tree T form a lineal pair.

We close this section by proving a theorem using lineal spanning trees. We don't "overexplain" this theorem to encourage you to think about the properties of lineal spanning trees that make the proof much simpler than what we might have come up with without lineal spanning trees. Recall that a graph $G = (V, E)$ is called *bipartite* if V can be partitioned into two sets C and S such that each edge has one vertex in C and one vertex in S (Exercises for Section 2).

Theorem 6 (Bipartite and cycle lengths) *Let $G = (V, E)$ be a simple graph. G is bipartite if and only if every cycle has even length.*

Proof: If G has a cycle of odd length, label each vertex with the block of some proposed bipartite partition $\{C, S\}$. For example, if , (x_1, x_2, x_3) are the vertices, in some order, of a cycle of length three, then the block labels (start with C) would be (C, S, C) This would mean that the edge $\{x_1, x_3\}$ would have both vertices in block C. This violates the definition of a bipartite graph. Since this problem happens for any cycle of odd length, a bipartite graph can never contain a cycle of odd length.

To prove the converse, we must show that if every cycle of G has even length, then G is bipartite. Suppose every cycle of G has even length. Choose a vertex v_0 as root of G and construct a lineal spanning tree T for G with root v_0. Label the root v_0 of T with C, all vertices of T of distance 1 from v_0 with S, all of distance 2 from v_0 with C, etc. Put vertices labeled C into block C of a partition $\{C, S\}$ of V, put all other vertices into block S. If $f = \{x, y\}$ is an edge of T then x and y are in different blocks of the partition $\{C, S\}$

by construction. If $f = \{x, y\}$ is an edge of G not in T then the two facts (1) T is lineal and (2) every cycle has even length, imply that x and y are in different blocks of the partition $\{C, S\}$. This completes the proof. $\square$

Exercises for Section 3

3.1. In this exercise, we study how counting edges and vertices in a graph can establish that cycles exist. For parts (a) and (b), let $G = (V, E, \phi)$ be a graph with loops allowed.

(a) Using induction on n, prove:
If $n \geq 0$, G is connected and G has v vertices and $v + n$ edges, then G has at least $n + 1$ cycles.

(b) Prove that, if G has v vertices, e edges and c components, then G has at least $c + e - v$ cycles.
Hint: Use (a) for each component.

(c) Show that (a) is best possible, even for simple graphs. In other words, for each n construct a simple graph that has n more edges than vertices but has only $n + 1$ cycles.

3.2. Let $T = (V, E)$ be a tree and let $d(v)$ be the degree of a vertex

(a) Prove that $\sum_{v \in V}(2 - d(v)) = 2$.

(b) Prove that, if T has a vertex of degree $m \geq 2$, then it has at least m vertices of degree 1.

(c) Give an example for all $m \geq 2$ of a tree with a vertex of degree m and only m leaves.

3.3. Give an example of a graph that satisfies the specified condition or show that no such graph exists.

(a) A tree with six vertices and six edges

(b) A tree with three or more vertices, two vertices of degree one and all the other vertices with degree three or more.

(c) A disconnected graph with 10 vertices and 8 edges.

(d) A disconnected graph with 12 vertices and 11 edges and no cycle.

(e) A tree with 6 vertices and the sum of the degrees of all vertices 12.

(f) A connected graph with 6 edges, 4 vertices, and exactly 2 cycles.

(g) A graph with 6 vertices, 6 edges and no cycles.

3.4. The *height* of a rooted tree is the maximum height of any leaf. The length of the unique path from a leaf of the tree to the root is, by definition, the height of that leaf. A rooted tree in which each non-leaf vertex has at most two children is called a *binary tree*. If each non-leaf vertex has exactly two children, the tree is called a *full binary tree*.

(a) If a binary tree has l leaves and height h prove that $l \leq 2^h$. (Taking logarithms gives $\log_2(l) \leq h$.)

(b) A binary tree has l leaves. What can you say about the maximum value of h?

(c) Given a full binary tree with l leaves, what is the maximum height h?

(d) Given a full binary tree with l leaves, what is the minimum height h?

(e) Given a binary tree of l leaves, what is the minimum height h?

3.5. In each of the following cases, state whether or not such a tree is possible.

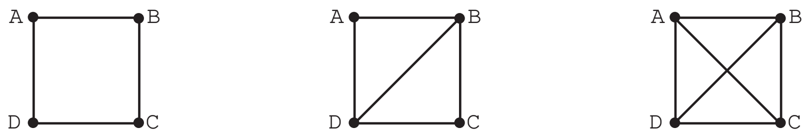
(a) A binary tree with 35 leaves and height 100.

(b) A full binary tree with 21 leaves and height 21.

(c) A binary tree with 33 leaves and height 5.

(d) A rooted tree of height 5 where every internal vertex has 3 children and there are 365 vertices.

3.6. For each of the following graphs:



(1) (2) (3)

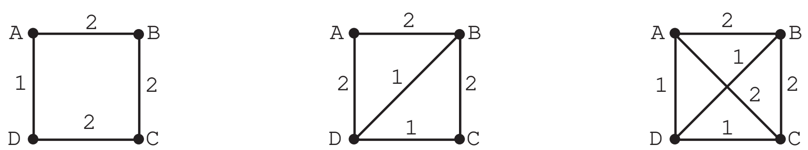
(a) Find all spanning trees.

(b) Find all spanning trees up to isomorphism.

(c) Find all depth-first spanning trees rooted at A.

(d) Find all depth-first spanning trees rooted at B.

3.7. For each of the following graphs:



(1) (2) (3)

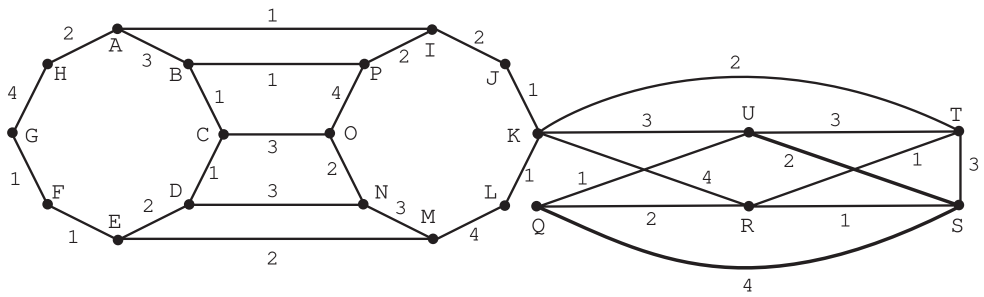
(a) Find all minimum spanning trees.

(b) Find all minimum spanning trees up to isomorphism.

(c) Among all depth-first spanning trees rooted at A, find those of minimum weight.

(d) Among all depth-first spanning trees rooted at B, find those of minimum weight.

3.8. In the following graph, the edges are weighted either 1, 2, 3, or 4.



Referring to Theorem 5 and the discussion following of Kruskal's algorithm:

(a) Find a minimum spanning tree using Prim's algorithm

(b) Find a minimum spanning tree using Kruskal's algorithm.

(c) Find a depth-first spanning tree rooted at K.

Section 4: Rates of Growth and Analysis of Algorithms

Suppose we have an algorithm and someone asks us "How good is it?" To answer that question, we need to know what they mean. They might mean "Is it correct?" or "Is it understandable?" or "Is it easy to program?" We won't deal with any of these.

They also might mean "How fast is it?" or "How much space does it need?" These two questions can be studied by similar methods, so we'll just focus on speed. Even now, the question is not precise enough. Does the person mean "How fast is it on this particular problem and this particular machine using this particular code and this particular compiler?" We could answer this simply by running the program! Unfortunately, that doesn't tell us what would happen with other machines or with other problems that the algorithm is designed to handle.

We would like to answer a question such as "How fast is Algorithm 1 for finding a spanning tree?" in such a way that we can compare that answer to "How fast is Algorithm 2 for finding a spanning tree?" and obtain something that is not machine or problem dependent. At first, this may sound like an impossible goal. To some extent it is; however, quite a bit can be said.

How do we achieve machine independence? We think in terms of simple machine operations such as multiplication, fetching from memory and so on. If one algorithm uses fewer of these than another, it should be faster. Those of you familiar with computer instruction timing will object that different basic machine operations take different amounts of time. That's true, but the times are not wildly different. Thus, if one algorithm uses *a lot fewer* operations than another, it should be faster. It should be clear from this that we can be a bit sloppy about what we call an operation; for example, we might call something like $x = a + b$ one operation. On the other hand, we can't be so sloppy that we call $x = a_1 + \cdots + a_n$ one operation if n is something that can be arbitrarily large.

Example 22 (Finding the maximum) Let's look at how long it takes to find the maximum of a list of n integers where we know nothing about the order they are in or how big the integers are. Let $a_1, \ldots, a_n$ be the list of integers. Here's our algorithm for finding the maximum.

$max = a_1$
For $i = 2, \ldots, n$
  If $a_i > max$, **then** $max = a_i$.
  End for
Return max

Being sloppy, we could say that the entire comparison and replacement in the "**If**" takes an operation and so does the stepping of the index i. Since this is done $n-1$ times, we get $2n-2$ operations. There are some setup and return operations, say s, giving a total of $2n-2+s$ operations. Since all this is rather sloppy all we can really say is that for large n and actual code on an actual machine, the procedure will take about Cn "ticks" of the machine's clock. Since we can't determine C by our methods, it will be helpful to have a notation that ignores it. We use $\Theta(f(n))$ to designate any function that behaves like a constant times $f(n)$ for arbitrarily large n. Thus we would say that the "**If**" takes time $\Theta(n)$ and the setup and return takes time $\Theta(1)$. Thus the total time is $\Theta(n) + \Theta(1)$. Since n is much bigger than 1 for large n, the total time is $\Theta(n)$. □

We need to define Θ more precisely and list its most important properties. We will also find it useful to define O, read "big oh."

Definition 21 (Notation for Θ and O) *Let f, g and h be functions from the positive integers to the nonnegative real numbers. We say that $g(n)$ is $\Theta(f(n))$ if there exist positive constants A and B such that $Af(n) \le g(n) \le Bf(n)$ for all sufficiently large n. In this case we say that f and g grow at the same rate. We say that $h(n)$ is $\mathrm{O}(f(n))$ if there exists a positive constant B such that $h(n) \le Bf(n)$ for all sufficiently large n. In this case we say that h grows no faster than f or, equivalently, that f grows at least as fast as h.*

The phrase "$\mathcal{S}(n)$ is true for all sufficiently large n" means that there is some integer N such that $\mathcal{S}(n)$ is true whenever $n \ge N$. Saying that something is $\Theta(f(n))$ gives an idea of *how big it is* for large values of n. Saying that something is $\mathrm{O}(f(n))$ gives an idea of *an upper bound on how big it is* for all large values of n. (We said "idea of" because we don't know what the constants A and B are.)

Theorem 7 (Some properties of Θ and O) *We have*

(a) *If $g(n)$ is $\Theta(f(n))$, then $g(n)$ is $\mathrm{O}(f(n))$.*

(b) $f(n)$ *is* $\Theta(f(n))$ *and* $f(n)$ *is* $\mathrm{O}(f(n))$.

(c) If $g(n)$ *is* $\Theta(f(n))$ *and* C *and* D *are positive constants, then* $Cg(n)$ *is* $\Theta(Df(n))$. *If* $g(n)$ *is* $\mathrm{O}(f(n))$ *and* C *and* D *are positive constants, then* $Cg(n)$ *is* $\mathrm{O}(Df(n))$.

(d) If $g(n)$ *is* $\Theta(f(n))$, *then* $f(n)$ *is* $\Theta(g(n))$.

(e) If $g(n)$ *is* $\Theta(f(n))$ *and* $f(n)$ *is* $\Theta(h(n))$, *then* $g(n)$ *is* $\Theta(h(n))$. *If* $g(n)$ *is* $\mathrm{O}(f(n))$ *and* $f(n)$ *is* $\mathrm{O}(h(n))$, *then* $g(n)$ *is* $\mathrm{O}(h(n))$.

(f) If $g_1(n)$ *is* $\Theta(f_1(n))$, $g_2(n)$ *is* $\Theta(f_2(n))$, *then* $g_1(n)+g_2(n)$ *is* $\Theta(\max(f_1(n), f_2(n)))$. *If* $g_1(n)$ *is* $\mathrm{O}(f_1(n))$, $g_2(n)$ *is* $\mathrm{O}(f_2(n))$, *then* $g_1(n)+g_2(n)$ *is* $\mathrm{O}(\max(f_1(n), f_2(n)))$.

Note that as a consequence of properties (b), (d) and (e) above, the statement "$g(n)$ is $\Theta(f(n))$" defines an equivalence relation on the set of functions from the positive integers to the nonnegative reals. As with any equivalence relation, we can think of it globally as partition into equivalence classes or locally as a relation between pairs of elements in the set on which the equivalence relation is defined. In the former sense "$g(n)$ is $\Theta(f(n))$" means that "$g(n)$ belongs to the equivalence class $\Theta(f(n))$ associated with f." In the latter sense, "$g(n)$ is $\Theta(f(n))$" means $g \sim_\Theta f$ where $\sim_\Theta$ is an equivalence relation called "is Θ."

Proof: Most of the proofs are left as an exercise. We'll do (e) for Θ. We are given that there are constants A_i and B_i such that

$$A_1 f(n) \le g(n) \le B_1 f(n)$$

and

$$A_2 h(n) \le f(n) \le B_2 h(n)$$

for all sufficiently large n. It follows that

$$A_1 A_2 h(n) \le A_1 f(n) \le g(n) \le B_1 f(n) \le B_1 B_2 h(n)$$

for all sufficiently large n. With $A = A_1A_2$ and $B = B_1B_2$, it follows that $g(n)$ is $\Theta(h(n))$. $\square$

Example 23 (Additional observations on Θ and O) In this example, we have collected some additional information about our notation.

Functions which are not always positive. Our definitions of Θ and O are only for functions whose values are nonnegative. The definitions can be extended to arbitrary functions by using absolute values; e.g., $A|f(n)| \le |g(n)| \le B|f(n)|$ means $g(n) = \Theta(f(n))$. All the results in the theorem still hold except (f) for Θ. This observation is most often applied to the case where the function f is "eventually" nonnegative ($\exists M$ such that $\forall\, n > M, f(n) \ge 0$). This is the case, for example with any polynomial in n with positive coefficient for the highest power of n.

Taking limits. When comparing two well-behaved functions $f(n)$ and $g(n)$, limits can be helpful:

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = C > 0 \quad \text{implies} \quad g(n) \quad \text{is} \quad \Theta(f(n))$$

and

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = C \geq 0 \quad \text{implies} \quad g(n) \quad \text{is} \quad \mathrm{O}(f(n)).$$

We assume here that the function f is never zero past some integer N so that the ratio is defined. The constants A and B of the definition can, in the first case, be taken to be $C - \epsilon$ and $C + \epsilon$, where ϵ is any positive number ($\epsilon = 1$ is a simple choice). In the second case, take B to be $C + \epsilon$. If, in the first case, $C = 1$, then f and g are said to be *asymptotic* or asymptotically equal. This is written $f \sim g$. If, in the second case, $C = 0$, then g is said to be *little oh* of f (written $g = \mathrm{o}(f)$). We will not use the "asymptotic" and "little oh" concepts.

Polynomials. In particular, you can take any polynomial as $f(n)$, say $f(n) = a_k n^k + \cdots + a_0$, and any other polynomial as $g(n)$, say $g(n) = b_k n^k + \cdots + b_0$. For f and g to be eventually positive we must have both a_k and b_k positive. If that is so, then $g(n)$ is $\Theta(f(n))$. Note in particular that we must have $g(n)$ is $\Theta(n^k)$.

Logarithms. Two questions that arise concerning logarithms are (a) "What base should I use?" and (b) "How fast do they grow?"

The base does not matter because $\log_a x = (\log_a b)(\log_b x)$ and constant factors like $\log_a b$ are ignored in $\Theta(\ )$ and $\mathrm{O}(\ )$.

It is known from calculus that $\log n \to \infty$ as $n \to \infty$ and that $\lim_{n\to\infty} (\log n)/n^\epsilon = 0$ for every $\epsilon > 0$. Thus logarithms grow, but they grow slower than powers of n. For example, $n \log n$ is $\mathrm{O}(n^{3/2})$ but $n^{3/2}$ is not $\mathrm{O}(n \log n)$.

A proof. How do we prove

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = C > 0 \qquad \text{implies} \qquad g(n) \text{ is } \Theta(f(n))?$$

By definition, the limit statement means that for any $\epsilon > 0$ there exists N such that for all $n > N$, $|\frac{g(n)}{f(n)} - C| < \epsilon$. If $\epsilon \geq C$, replace it with a smaller ϵ. From $|\frac{g(n)}{f(n)} - C| < \epsilon$, for all $n > N$,

$$C - \epsilon < \frac{g(n)}{f(n)} < C + \epsilon, \quad \text{or} \quad (C - \epsilon) f(n) < \frac{g(n)}{f(n)} < (C + \epsilon) f(n).$$

Take $A = (C - \epsilon)$ and $B = (C + \epsilon)$ in the definition of Θ. □

Example 24 (Using Θ) To illustrate these ideas, we'll consider three algorithms for evaluating a polynomial $p(x)$ of degree n at some point r; i.e., computing $p_0 + p_1 r + \cdots + p_n r^n$. We are interested in how fast they are when n is large. Here are the procedures. You should convince yourself that they work.

```
Poly1(n, p, r)
    S = p_0
    For i = 1, ..., n      S = S + p_i * Pow(r, i).
    Return S
End
```

Pow(r, i)
  $P = 1$
  For $j = 1, \ldots, n \quad P = P * r$.
  Return P
End

Poly2(n, p, r)
  $S = p_0$
  $P = 1$
  For $i = 1, \ldots, n$
    $P = P * r$.
    $S = S + p_i * P$
  End for
  Return S
End

Poly3(n, p, r)
  $S = p_n$
  For $i = n, \ldots, 2, 1 \quad S = S * r + p_{i-1}$
  Return S
End

Let T_n(Name) be the time required for the procedure Name. Let's analyze Poly1. The "**For**" loop in Pow is executed i times and so takes Ci operations for some constant C. The setup and return in Pow takes some constant number of operations D. Thus $T_n(\text{Pow}) = Ci + D$ operations. As a result, the ith iteration of the "**For**" loop in Poly1 takes $Ci + E$ operations for some constants C and $E > D$. Adding this over $i = 1, 2, \ldots, n$, we see that the total time spent in the "**For**" loop is $\Theta(n^2)$ since $\sum_{i=1}^{n} i = n(n+1)/2$. (This requires using some of the properties of Θ. You should write out the details.) Since the rest of Poly1 takes $\Theta(1)$ time, T_n(Poly1) is $\Theta(n^2)$.

The amount of time spent in the "For" loop of Poly2 is constant and the loop is executed n times. It follows that T_n(Poly2) is $\Theta(n)$. The same analysis applies to Poly3.

What can we conclude from this about the comparative speed of the algorithms? By the definition of Θ, there are positive reals A and B so that $An^2 \le T_n(\text{Poly1})$ and $T_n(\text{Poly2}) \le Bn$ for sufficiently large n. Thus $T_n(\text{Poly2})/T_n(\text{Poly1}) \le B/An$. As n gets larger, Poly2 looks better and better compared to Poly1.

Unfortunately, the crudeness of Θ does not allow us to make any distinction between Poly2 and Poly3. What we can say is that T_n(Poly2) is $\Theta(T_n(\text{Poly3}))$; i.e., T_n(Poly2) and T_n(Poly3) grow at the same rate. A more refined estimate can be obtained by counting the actual number of operations involved. □

So far we have talked about how long an algorithm takes to run as if this were a simple, clear concept. In the next example we'll see that there's an important point that we've ignored.

***Example 25 (What is average running time?)** Let's consider the problem of (a) deciding whether or not a simple graph can be properly colored with four colors and, (b) if a proper coloring exists, producing one. A *proper coloring* of a simple graph

$G = (V, E)$ is a function $\lambda: V \to C$, the set of "colors," such that, if $\{u, v\}$ is an edge, then $\lambda(u) \neq \lambda(v)$. We may as well assume that $V = \underline{n}$ and that the colors are c_1, c_2, c_3 and c_4.

Here's a simple algorithm to determine a λ by using backtracking to go lexicographically through possible colorings $\lambda(1), \lambda(2), \ldots, \lambda(n)$.

1. **Initialize:** Set $v = 1$ and $\lambda(1) = c_1$.
2. **Advance in decision tree:** If $v = n$, stop with λ determined; otherwise, set $v = v + 1$ and $\lambda(v) = c_1$.
3. **Test:** If $\lambda(i) \neq \lambda(v)$ for all $i < v$ for which $\{i, v\} \in E$, go to Step 2.
4. **Select next decision:** Let j be such that $\lambda(v) = c_j$. If $j < 4$, set $\lambda(v) = c_{j+1}$ and go to Step 3.
5. **Backtrack:** If $v = 1$, stop with coloring impossible; otherwise, set $v = v - 1$ and go to Step 4.

How fast is this algorithm? Obviously it will depend on the graph. Here are two extreme cases:

- Suppose the subgraph induced by the first five vertices is the complete graph K_5 (i.e., all of the ten possible edges are present). The algorithm stops after trying to color the first five vertices and discovering that there is no proper coloring. Thus the running time does not depend on n and so is in $\Theta(1)$.
- Suppose that the first $n-5$ vertices have no edges and that the last five vertices induce K_5. The algorithm tries all possible assignments of colors to the first $n - 5$ vertices and, for each of them, discovers that it cannot properly color the last five because they form K_5. Thus the algorithm makes between 4^{n-5} and 4^n assignments of colors and so its running time is $\Theta(4^n)$ — a much faster growing time than $\Theta(1)$.

What should we do about studying the running time of such an algorithm? It's reasonable to talk about the *average* time the algorithm takes if we expect to give it lots of graphs to look at. Most n vertex graphs will have many sets of five vertices that induce K_5. (We won't prove this.) As a result, the algorithm has running time in $\Theta(1)$ for most graphs. In fact, it can be proved that the average number of assignments of the form $\lambda(v) = c_k$ that are made is $\Theta(1)$ and so the average running time is $\Theta(1)$. This means that the average running time of the algorithm is bounded for all n, which is quite good!

Now suppose you give this algorithm to a friend, telling her that the average running time is bounded. She thanks you profusely for such a wonderful algorithm and puts it to work coloring randomly generated "planar" graphs. These are a special class of graphs whose pictures can be drawn in the plane without edges crossing each other. (All trees are planar, but K_5 is not planar.) By a famous theorem called the *Four Color Theorem*, every planar graph can be properly colored with four colors, so the algorithm will find the coloring. To do so it must make assignments of the form $\lambda(v) = c_k$ for each vertex v. Thus it must make at least n assignments. (Actually it will almost surely make *many, many* more.) Your friend soon comes back to you complaining that your algorithm takes a long time to run. What went wrong?

You were averaging over all simple graphs with n vertices. Your friend was averaging over all simple planar graphs with n vertices. The average running times are *very* different! There is a lesson here:

You must be VERY clear what you are averaging over.

Because situations like this do occur in real life, computer scientists are careful to specify what kind of running time they are talking about; either the average of the running time over some reasonable, clearly specified set of problems or the worst (longest) running time over all possibilities. □

You should be able to see that saying something is $\Theta(\ )$ leaves a lot out because we have no idea of the constants that are omitted. How can we compare two algorithms? Here are two rules of thumb.

- If one algorithm is $\Theta(f(n))$ and the other is $\Theta(g(n))$, the algorithm with the slower growing function (f or g) is probably the better choice.
- If both algorithms are $\Theta(f(n)$, the algorithm with the simpler data structures is probably better.

These rules are far from foolproof, but they provide some guidance.

---

*Polynomial Time Algorithms

Computer scientists talk about "polynomial time algorithms." What does this mean? Suppose that the algorithm can handle arbitrarily large problems and that it takes $\Theta(n)$ seconds on a problem of "size" n. Then we call it a linear time algorithm. More generally, if there is a (possibly quite large) integer k such that the worst case running time on a problem of "size" n is $O(n^k)$, then we say the algorithm is polynomial time.

You may have noticed the quotes around size and wondered why. It is necessary to specify what we mean by the size of a problem. Size is often interpreted as the number of bits required to specify the problem in binary form. You may object that this is imprecise since a problem can be specified in many ways. This is true; however, the number of bits in one "reasonable" representation doesn't differ too much from the number of bits in another. We won't pursue this further.

If the worst case time for an algorithm is polynomial, theoretical computer scientists think of this as a good algorithm. (This is because polynomials grow relatively slowly; for example, exponential functions grow much faster.) The problem that the algorithm solves is called *tractable*.

Do there exist *intractable problems*; i.e., problems for which no polynomial time algorithm can ever be found? Yes, but we won't study them here. More interesting is the fact that there are a large number of practical problems for which

- no polynomial time algorithm is known and
- no one has been able prove that the problems are intractable.

We'll discuss this a bit.

Consider the following problems.

- **Coloring problem:** For any $c > 2$, devise an algorithm whose input can be any simple graph and whose output answers the question "Can the graph be properly colored in c colors?"

- **Traveling salesman problem:** For any B, devise an algorithm whose input can be any $n > 0$ and any real valued edge labeling, $\lambda\colon \mathcal{P}_2(\underline{n}) \to \mathbb{R}$, for K_n, the complete graph on n vertices. The algorithm must answer the question "Is there a cycle through all n vertices with cost B or less?" (The cost of a cycle is the sum of $\lambda(e)$ over all e in the cycle.)
- **Clique problem:** Given a simple graph $G = (V, E)$ and an integer s, is there a subset $S \subseteq V$, $|S| = s$, whose induced subgraph is the complete graph on S (i.e., a subgraph of G with vertex set S and with $\binom{s}{2}$ edges)?

No one knows if these problems are tractable, but it is known that, if one is tractable, then they all are. There are hundreds more problems that people are interested in which belong to this particular list in which all or none are tractable. These problems are called *NP-complete problems* Many people regard deciding if the NP-complete problems are tractable to be the foremost open problem in theoretical computer science.

The NP-complete problems have an interesting property which we now discuss. If the algorithm says "yes," then there must be a specific example that shows why this is so (an assignment of colors, a cycle, an automaton). There is no requirement that the algorithm actually produce such an example. Suppose we somehow obtain a coloring, a cycle or an automaton which is claimed to be such an example. Part of the definition of NP-complete requires that we be able to check the claim in polynomial time. Thus we can check a purported example quickly but, so far as is known, it may take a long time to determine if such an example exists. In other words, I can check your guesses quickly but I don't know how to tell you quickly if any examples exist.

There are problems like the NP-complete problems where no one knows how to do any checking in polynomial time. For example, modify the traveling salesman problem to ask for the minimum cost cycle. No one knows how to verify in polynomial time that a given cycle is actually the minimum cost cycle. If the modified traveling salesman problem is tractable, so is the one we presented above: You need only find the minimum cost cycle and compare its cost to B. Such problems are called *NP-hard* because they are at least as hard as NP-complete problems. A problem which is tractable if the NP-complete problems are tractable is called *NP-easy.*

Some problems are both NP-easy and NP-hard but may not be NP-complete. Why is this? NP-complete problems must ask a "yes/no" type of question and it must be possible to check a specific example in polynomial time as noted in the previous paragraph. We discuss an example.

***Example 26 (Chromatic number)** The *chromatic number* $\chi(G)$ of a graph G is the least number of colors needed to properly color G. The problem of deciding whether a graph can be properly colored with c colors is NP-complete. The problem of determining $\chi(G)$ is NP-hard. If we know $\chi(G)$, then we can determine if c colors are enough by checking if $c \geq \chi(G)$.

The problem of determining $\chi(G)$ is also NP-easy. You can color G with c colors if and only if $c \geq \chi(G)$. We know that $0 \leq \chi(G) \leq n$ for a graph with n vertices. Ask if c colors suffice for $c = 0, 1, 2, \ldots$. The least c for which the answer is "yes" is $\chi(G)$. Thus the worst case time for finding $\chi(G)$ is at most n times the worst case time for the NP-complete problem. Hence one time is O of a polynomial in n if and only if the other is. $\square$

What can we do if we cannot find a good algorithm for a problem? There are three main types of partial algorithms:

1. **Almost good:** It is polynomial time for all but a very small subset of possible problems. (If we are interested in all graphs, our coloring algorithm in Example 25 is almost good for any fixed c.)
2. **Almost correct:** It is polynomial time but in some rare cases does not find the correct answer. (If we are interested in all graphs and a fixed c, automatically reporting that a large graph can't be colored with c colors is almost correct — but it is rather useless.) In some situations, a fast almost correct algorithm can be useful.
3. **Close:** It is a polynomial time algorithm for a minimization problem and comes close to the true minimum. (There are useful close algorithms for approximating the minimum cycle in the Traveling Salesman Problem.)

Some of the algorithms make use of random number generators in interesting ways. Unfortunately, further discussion of these problems is beyond the scope of this text.

---

*A Theorem for Recursive Algorithms

Some algorithms, such as merge sorting, call themselves. This is known as a *recursive algorithm* or a *divide and conquer algorithm*.

When we try estimate the running time of such algorithms, we obtain a recursion. In Section 2 of Unit DT, we examined the problem of solving recursions. We saw that finding exact solutions to recursions is difficult. The recursions that we obtain for algorithms are not covered by the methods in that section. Furthermore, the recursions are often not known exactly because we may only be able to obtain an estimate of the form $\Theta(\ )$ for some of the work. The next example illustrates this problem.

***Example 27 (Sorting by recursive merging)** Given a list L of n items, we wish to sort it. Here is the merge sorting algorithm from Section 3 of Unit DT.

```
Sort(L)
    If length is 1, return L
    Else
        Split L into two lists L1 and L2
        S1 = Sort(L1)
        S2 = Sort(L2)
        S = Merge(L1, L2)
        Return S
    End if
End
```

We need to be more specific about how the lists are split. Let m be $n/2$ rounded down, let L1 be the first m items in L and let L2 be the last $n-m$ items in L.

One way to measure the running time of Sort(L) is to count the number of comparisons that are required. Let this number be $T(n)$. We would like to know how fast $T(n)$ grows as a function of n so we can tell how good the algorithm is. For example, is $T(n) = \Theta(n)$? is $T(n) = \Theta(n^2)$? or does it behave differently?

We now start work on this problem. Since the sorting algorithm is recursive (calls itself), we will end up with a recursion. This is a general principle for recursive algorithms. You should see why after the next two paragraphs.

All comparisons are done in Merge(L1,L2). It can be shown that the number of comparisons in Merge is between m and $n-1$. We take that fact as given.

Three lines of code are important:

S1 = Sort(L1)	a recursive call, so it gives us $T(m)$;
S2 = Sort(L2)	a recursive call, so it gives us $T(n-m)$;
S = Merge(L1, L2)	where the comparisons are, so it gives us a_n with $m \le a_n \le n = 1$.

We obtain $T(n) = T(m) + T(n-m) + a_n$ where all we know about a_n is that it is between m and $n-1$. What can we do?

Not only is this a type of recursion we haven't seen before, we don't even know the recursion fully since all we have is upper and lower bounds for a_n. The next theorem solves this problem for us. $\square$

The following theorem provides an approximate solution to an important class of approximate recursions that arise in divide and conquer algorithms. We'll apply it to merge sorting. In the theorem

- $T(n)$ is the running time for a problem of size n.
- If the algorithm calls itself at w places in the code, then the problem is divided into w smaller problems of the same kind and $s_1(n), \ldots, s_w(n)$ are the sizes of the smaller problems.
- The constant c measures how much smaller each of these problems is.
- The time needed for the rest of the code is a_n.

***Theorem 8 (Master Theorem for Recursions*)** *Suppose that there are*

(i) numbers N, b, $w \ge 1$ and $0 < c < 1$ that do not depend on n

(ii) a sequence $a_1, a_2, \ldots,$

(iii) functions $s_1, s_2, \ldots, s_w$, and T

such that

(a) $T(n) > 0$ for all $n > N$ and $a_n \ge 0$ for all $n > N$;

* This is not the most general version of the theorem; however, this version is easier to understand and is usually sufficient. For a more general statement and a proof, see any thorough text on the analysis of algorithms.

(b) $T(n) = a_n + T(s_1(n)) + T(s_2(n)) + \cdots + T(s_w(n))$ *for all* $n > N$*;*

(c) a_n *is* $\Theta(n^b)$ *(If* $a_n = 0$ *for all large* n*, set* $b = -\infty$*.);*

(d) $|s_i(n) - cn|$ *is* $O(1)$ *for* $i = 1, 2, \ldots, w$*.*

Let $d = -\log(w)/\log(c)$*. Then*

$$T(n) \text{ is } \begin{cases} \Theta(n^d) & \text{if } b < d, \\ \Theta(n^d \log n) & \text{if } b = d, \\ \Theta(n^b) & \text{if } b > d. \end{cases}$$

Note that $b = 0$ corresponds to a_n being in $\Theta(1)$ since $n^0 = 1$. In other words, a_n is bounded by nonzero constants for all large n: $0 < C_1 \le a_n \le C_2$.

Let's apply the theorem to our recursion for merge sorting:

$$T(n) = a_n + T(s_1(n)) + T(s_2(n))$$

where

$$s_1(n) = \lfloor n/2 \rfloor, \quad s_2(n) = \lfloor n - n/2 \rfloor \quad \text{and} \quad s_1(n) \le a_n \le n - 1.$$

Note that $s_1(n)$ and $s_2(n)$ differ from $n/2$ by at most $1/2$ and that $a_n = \Theta(n)$. Thus we can apply the theorem with $w = 2$, $b = 1$ and $c = 1/2$. We have

$$d = -\log(2)/\log(1/2) = \log(2)/\log(2) = 1.$$

Since $b = d = 1$, we conclude that $T(n)$ is $\Theta(n \log n)$.

How do we use the theorem on divide and conquer algorithms? First, we must find a parameter n that measures the size of the problem; for example, the length of a list to be sorted, the degree of polynomials that we want to multiply, the number of vertices in a graph that we want to study. Then use the interpretation of the various parameters that was given just before the theorem.

Our final example is more difficult because the algorithm that we study is more complicated. It was believed for some time that the quickest way to multiply polynomials was the "obvious" way that is taught when polynomials are first studied. That is not true. The next example contains an algorithm for faster multiplication of polynomials. There are also faster algorithms for multiplying matrices.

***Example 28 (Recursive multiplication of polynomials)** Suppose we want to multiply two polynomials of degree at most n, say

$$P(x) = p_0 + p_1 x + \cdots + p_n x^n \quad \text{and} \quad Q(x) = q_0 + q_1 x + \cdots + q_n x^n.$$

The natural way to do this is to use the distributive law to generate $(n+1)^2$ products p_0q_0, p_0q_1x, $p_0q_2x^2, \ldots,$ $p_nq_nx^{2n}$ and then collect the terms that have the same powers of x. This involves $(n+1)^2$ multiplications of coefficients and, it can be shown, n^2 additions of coefficients. Thus, the amount of work is $\Theta(n^2)$. Unless we expect $P(x)$ or $Q(x)$ to have some coefficients that are zero, this seems to be best we can do. Not so! We now present and analyze a faster recursive algorithm.

The algorithm depends on the following identity which you should verify by checking the algebra.

Identity: If $P_L(x)$, $P_H(x)$, $Q_L(x)$ and $Q_H(x)$ are polynomials, then

$$\big(P_L(x)+P_H(x)x^m\big)\big(Q_L(x)+Q_H(x)x^m\big) = A(x)+\big(C(x)-A(x)-B(x)\big)x^m + B(x)x^{2m}$$

where

$$A(x) = P_L(x)Q_L(x), \quad B(x) = P_H(x)Q_H(x),$$

and

$$C(x) = \big(P_L(x)+P_H(x)\big)\big(Q_L(x)+Q_H(x)\big)$$

We can think of this identity as telling us how to multiply two polynomials $P(x)$ and $Q(x)$ by splitting them into lower degree terms ($P_L(x)$ and $Q_L(x)$) and higher degree terms ($P_H(x)x^m$ and $Q_H(x)x^m$):

$$P(x) = D(P_L(x)+P_H(x)x^m \quad \text{and} \quad Q(x) = Q_L(x)+Q_H(x)x^m.$$

The identity requires three polynomial multiplications to compute $A(x)$, $B(x)$ and $C(x)$. This leads naturally to two questions:

- Haven't things gotten worse — three polynomial multiplications instead of just one? No. The three multiplications involve polynomials of much lower degrees. We will see that this leads to a gain in speed.
- How should we do these three polynomial multiplications? Apply the identity to each of them. In other words, design a recursive algorithm. We do that now.

Here is the algorithm for multiplying two polynomials $P(x) = p_0 + p_1x\cdots + p_nx^n$ and $Q(x) = q_0 + q_1x + \cdots + q_nx^n$ of degree at most n.

MULT$(P(x), Q(x), n)$
  If (n=0) Return p_0q_0
  Else
    Let $m = n/2$ rounded up.
    $P_L(x) = p_0 + p_1x + \cdots p_{m-1}x^{m-1}$
    $P_H(x) = p_m + p_{m+1}x + \cdots p_nx^{n-m}$
    $Q_L(x) = q_0 + p_1x + \cdots q_{m-1}x^{m-1}$
    $Q_H(x) = q_m + q_{m+1}x + \cdots q_nx^{n-m}$
    $A(x) = \text{MULT}\big(P_L(x),\, Q_L(x),\, m-1\big)$
    $B(x) = \text{MULT}\big(P_H(x),\, Q_H(x),\, n-m\big)$
    $C(x) = \text{MULT}\big(P_L(x)+P_H(x),\, Q_L(x)+Q_H(x),\, n-m\big)$
    $D(x) = A(x)+\big(C(x)-A(x)-B(x)\big)x^m + B(x)x^{2m}$
    Return $D(x)$
  End if
End

As is commonly done, we imagine a polynomial stored as a vector of coefficients. The amount of work required is then the number of times we have two multiply or add two coefficients. For simplicity, we just count multiplications. Let that number be $T(n)$. You should be able to see that $T(0) = 1$ and

$$T(n) = T(m-1) + T(n-m) + T(n-m) \quad \text{for} \quad n > 0.$$

We can write this as

$$T(n) = T(m-1) + T(n-m) + T(n-m) + a_n, \quad a_0 = 1 \text{ and } a_n = 0 \text{ for } n > 0.$$

Note that, since both $m-1$ and $n-m$ differ from $n/2$ by at most 1, $w = 3$ and $c = 1/2$. Also $b = -\infty$.

We have $d = \log 3/\log 2 > b$. Thus $T(n)$ is $\Theta(n^{\log 3/\log 2})$. Since $\log 3/log 2$ is about 1.6 which is less than 2, this is less work than the straightforward method when n is large enough. (Recall that the work there was in $\Theta(n^2)$.) $\square$

Exercises for Section 4

4.1. We have three algorithms for solving a problem for graphs. Suppose algorithm A takes n^2 milliseconds to run on a graph with n vertices, algorithm B takes $100n$ milliseconds and algorithm C takes $100(2^{n/10} - 1)$ milliseconds.

(a) Compute the running times for the three algorithms with n = 5, 10, 30, 100 and 300. Which algorithm is fastest in each case? slowest?

(b) Which algorithm is fastest for all very large values of n? Which is slowest?

4.2. Let $p(x)$ be a polynomial of degree k with positive leading coefficient and suppose that $a > 1$. Prove the following.

(a) $\Theta(p(n))$ is $\Theta(n^k)$.

(b) $O(p(n))$ is $O(n^k)$.

(c) $\lim_{n\to\infty} p(n)/a^n = 0$. (Also, what does this say about the speed of a polynomial time algorithm versus one which takes exponential time?)

(d) Unless $p(x) = p_1 x^k + p_2$ for some p_1 and p_2, there is no C such that $a^{p(n)}$ is $\Theta(a^{Cn^k})$.

4.3. In each case, prove that $g(n)$ is $\Theta(f(n))$ using the definition of "g is $\Theta(f)$". (See Definition 21.)

(a) $g(n) = n^3 + 5n^2 + 10$, $f(n) = 20n^3$.

(b) $g(n) = n^2 + 5n^2 + 10$, $f(n) = 200n^2$

4.4. In each case, show that the given series has the indicated property.

(a) $\sum_{i=1}^{n} i^2$ is $\Theta(n^3)$.

(b) $\sum_{i=1}^{n} i^3$ is $\Theta(n^4)$.

(c) $\sum_{i=1}^{n} i^{1/2}$ is $\Theta(n^{3/2})$.

4.5. Show each of the following

(a) $\sum_{i=1}^{n} i^{-1}$ is $\Theta(\log_b(n))$ for any base $b > 1$.

(b) $\log_b(n!)$ is $O(n \log_b(n))$ for any base $b > 1$.

(c) $n!$ is $\Theta((n/e)^{n+1/2})$.

***4.6.** The following algorithm multiplies two $n \times n$ matrices A and B and puts the answer in C. Let $T(n)$ be the running time of the algorithm Find a simple function $f(n)$ so that is $\Theta(f(n))$.

```
MATRIXMULT(n,A,B,C)
    For i=1,...,n
        For j=1,...,n
            C(i,j)=0
            For k=1,...,n
                C(i,j) = C(i,j) + A(i,k)*B(k,j)
            End for
        End for
    End for
End
```

***4.7.** The following algorithm computes x^n for n a positive integer, where x is a complicated object (e.g., a large matrix). MULT(x, y) is a procedure that multiplies two such objects. Let $T(n)$ be the number of times MULT is called. Find a simple function $f(n)$ so that $T(n)$ is $\Theta(f(n))$.

```
POW(x, n)
    If (n=1) Return x
    Else
        Let q be n/2 rounded down and r = n − 2q.
        y = MULT(x, x)
        z = POW(y, q)
        If (r=0) Return z
        Else
            w = MULT(x, z)
            Return w
        End if
    End if
End
```

Multiple Choice Questions for Review

Some of the following questions assume that you have done the exercises.

1. Indicate which, if any, of the following five graphs $G = (V, E, \phi)$, $|V| = 5$, is not isomorphic to any of the other four.

(a) $\phi = \begin{pmatrix} A & B & C & D & E & F \\ \{1,3\} & \{2,4\} & \{1,2\} & \{2,3\} & \{3,5\} & \{4,5\} \end{pmatrix}$

(b) $\phi = \begin{pmatrix} f & b & c & d & e & a \\ \{1,2\} & \{1,2\} & \{2,3\} & \{3,4\} & \{3,4\} & \{4,5\} \end{pmatrix}$

(c) $\phi = \begin{pmatrix} b & f & e & d & c & a \\ \{4,5\} & \{1,3\} & \{1,3\} & \{2,3\} & \{2,4\} & \{4,5\} \end{pmatrix}$

(d) $\phi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ \{1,2\} & \{2,3\} & \{2,3\} & \{3,4\} & \{4,5\} & \{4,5\} \end{pmatrix}$

(e) $\phi = \begin{pmatrix} b & a & e & d & c & f \\ \{4,5\} & \{1,3\} & \{1,3\} & \{2,3\} & \{2,5\} & \{4,5\} \end{pmatrix}$

2. Indicate which, if any, of the following five graphs $G = (V, E, \phi)$, $|V| = 5$, is not connected.

(a) $\phi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ \{1,2\} & \{1,2\} & \{2,3\} & \{3,4\} & \{1,5\} & \{1,5\} \end{pmatrix}$

(b) $\phi = \begin{pmatrix} b & a & e & d & c & f \\ \{4,5\} & \{1,3\} & \{1,3\} & \{2,3\} & \{2,5\} & \{4,5\} \end{pmatrix}$

(c) $\phi = \begin{pmatrix} b & f & e & d & c & a \\ \{4,5\} & \{1,3\} & \{1,3\} & \{2,3\} & \{2,4\} & \{4,5\} \end{pmatrix}$

(d) $\phi = \begin{pmatrix} a & b & c & d & e & f \\ \{1,2\} & \{2,3\} & \{1,2\} & \{2,3\} & \{3,4\} & \{1,5\} \end{pmatrix}$

(e) $\phi = \begin{pmatrix} a & b & c & d & e & f \\ \{1,2\} & \{2,3\} & \{1,2\} & \{1,3\} & \{2,3\} & \{4,5\} \end{pmatrix}$

3. Indicate which, if any, of the following five graphs $G = (V, E, \phi)$, $|V| = 5$, have an Eulerian circuit.

(a) $\phi = \begin{pmatrix} F & B & C & D & E & A \\ \{1,2\} & \{1,2\} & \{2,3\} & \{3,4\} & \{4,5\} & \{4,5\} \end{pmatrix}$

(b) $\phi = \begin{pmatrix} b & f & e & d & c & a \\ \{4,5\} & \{1,3\} & \{1,3\} & \{2,3\} & \{2,4\} & \{4,5\} \end{pmatrix}$

(c) $\phi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ \{1,2\} & \{1,2\} & \{2,3\} & \{3,4\} & \{4,5\} & \{4,5\} \end{pmatrix}$

(d) $\phi = \begin{pmatrix} b & a & e & d & c & f \\ \{4,5\} & \{1,3\} & \{1,3\} & \{2,3\} & \{2,5\} & \{4,5\} \end{pmatrix}$

(e) $\phi = \begin{pmatrix} a & b & c & d & e & f \\ \{1,3\} & \{3,4\} & \{1,2\} & \{2,3\} & \{3,5\} & \{4,5\} \end{pmatrix}$

4. A graph with $V = \{1, 2, 3, 4\}$ is described by $\phi = \begin{pmatrix} a & b & c & d & e & f \\ \{1,2\} & \{1,2\} & \{1,4\} & \{2,3\} & \{3,4\} & \{3,4\} \end{pmatrix}$. How many Hamiltonian cycles does it have?

(a) 1 (b) 2 (c) 4 (d) 16 (e) 32

5. A graph with $V = \{1,2,3,4\}$ is described by $\phi = \begin{pmatrix} a & b & c & d & e & f \\ \{1,2\} & \{1,2\} & \{1,4\} & \{2,3\} & \{3,4\} & \{3,4\} \end{pmatrix}$. It has weights on its edges given by $\lambda = \begin{pmatrix} a & b & c & d & e & f \\ 3 & 2 & 1 & 2 & 4 & 2 \end{pmatrix}$. How many minimum spanning trees does it have?

(a) 2 (b) 3 (c) 4 (d) 5 (e) 6

6. Define an RP-tree by the parent-child adjacency lists as follows:

(i) Root B: J, H, K; (ii) H: P, Q, R; (iii) Q: S, T; (iv) K: L, M, N.

The postorder vertex sequence of this tree is

(a) J, P, S, T, Q, R, H, L, M, N, K, B.

(b) P, S, T, J, Q, R, H, L, M, N, K, B.

(c) P, S, T, Q, R, H, L, M, N, K, J, B.

(d) P, S, T, Q, R, J, H, L, M, N, K, B.

(e) S, T, Q, J, P, R, H, L, M, N, K, B.

7. Define an RP-tree by the parent-child adjacency lists as follows:

(i) Root B: J, H, K; (ii) J: P, Q, R; (iii) Q: S, T; (iv) K: L, M, N.

The preorder vertex sequence of this tree is

(a) B, J, H, K, P, Q, R, L, M, N, S, T.

(b) B, J, P, Q, S, T, R, H, K, L, M, N.

(c) B, J, P, Q, S, T, R, H, L, M, N, K.

(d) B, J, Q, P, S, T, R, H, L, M, N, K.

(e) B, J, Q, S, T, P, R, H, K, L, M, N.

8. For which of the following does there exist a graph $G = (V, E, \phi)$ satisfying the specified conditions?

(a) A tree with 9 vertices and the sum of the degrees of all the vertices 18.

(b) A graph with 5 components 12 vertices and 7 edges.

(c) A graph with 5 components 30 vertices and 24 edges.

(d) A graph with 9 vertices, 9 edges, and no cycles.

(e) A connected graph with 12 edges 5 vertices and fewer than 8 cycles.

9. For which of the following does there exist a simple graph $G = (V, E)$ satisfying the specified conditions?

(a) It has 3 components 20 vertices and 16 edges.

(b) It has 6 vertices, 11 edges, and more than one component.

(c) It is connected and has 10 edges 5 vertices and fewer than 6 cycles.

(d) It has 7 vertices, 10 edges, and more than two components.

(e) It has 8 vertices, 8 edges, and no cycles.

10. For which of the following does there exist a tree satisfying the specified constraints?

(a) A binary tree with 65 leaves and height 6.

(b) A binary tree with 33 leaves and height 5.

(c) A full binary tree with height 5 and 64 total vertices.

(d) A full binary tree with 23 leaves and height 23.

(e) A rooted tree of height 3, every vertex has at most 3 children. There are 40 total vertices.

11. For which of the following does there exist a tree satisfying the specified constraints?

(a) A full binary tree with 31 leaves, each leaf of height 5.

(b) A rooted tree of height 3 where every vertex has at most 3 children and there are 41 total vertices.

(c) A full binary tree with 11 vertices and height 6.

(d) A binary tree with 2 leaves and height 100.

(e) A full binary tree with 20 vertices.

12. The number of simple digraphs with $|V| = 3$ is

(a) 2^9 (b) 2^8 (c) 2^7 (d) 2^6 (e) 2^5

13. The number of simple digraphs with $|V| = 3$ and exactly 3 edges is

(a) 92 (b) 88 (c) 80 (d) 84 (e) 76

14. The number of oriented simple graphs with $|V| = 3$ is

(a) 27 (b) 24 (c) 21 (d) 18 (e) 15

15. The number of oriented simple graphs with $|V| = 4$ and 2 edges is

(a) 40 (b) 50 (c) 60 (d) 70 (e) 80

16. In each case the depth-first sequence of an ordered rooted spanning tree for a graph G is given. Also given are the non-tree edges of G. Which of these spanning trees is a depth-first spanning tree?

(a) 123242151 and $\{3,4\}$, $\{1,4\}$

(b) 123242151 and $\{4,5\}$, $\{1,3\}$

(c) 123245421 and $\{2,5\}$, $\{1,4\}$

(d) 123245421 and $\{3,4\}$, $\{1,4\}$

(e) 123245421 and $\{3,5\}$, $\{1,4\}$

Basic Concepts in Graph Theory

17. $\sum_{i=1}^{n} i^{-1/2}$ is

(a) $\Theta((\ln(n))^{1/2})$ (b) $\Theta(\ln(n))$ (c) $\Theta(n^{1/2})$ (d) $\Theta(n^{3/2})$ (e) $\Theta(n^2)$

18. Compute the total number of bicomponents in all of the following three simple graphs, $G = (V, E)$ with $|V| = 5$. For each graph the edge sets are as follows:

$$E = \big\{\{1,2\},\ \{2,3\},\ \{3,4\},\ \{4,5\},\ \{1,3\},\ \{1,5\},\ \{3,5\}\big\}$$

$$E = \big\{\{1,2\},\ \{2,3\},\ \{3,4\},\ \{4,5\},\ \{1,3\}\big\}$$

$$E = \big\{\{1,2\},\ \{2,3\},\ \{4,5\},\ \{1,3\}\big\}$$

(a) 4 (b) 5 (c) 6 (d) 7 (e) 8

19. Let $b > 1$. Then $\log_b((n^2)!)$ is

(a) $\Theta(\log_b(n!))$

(b) $\Theta(\log_b(2\,n!))$

(c) $\Theta(n \log_b(n))$

(d) $\Theta(n^2 \log_b(n))$

(e) $\Theta(n \log_b(n^2))$

20. What is the total number of additions and multiplications in the following code?

```
s := 0
for i := 1 to n
    s:= s + i
    for j:= 1 to i
        s := s + j*i
    next j
next i
s := s+10
```

(a) n (b) n^2 (c) $n^2 + 2n$ (d) $n(n+1)$ (e) $(n+1)^2$

Answers: **1** (a), **2** (e), **3** (e), **4** (c), **5** (b), **6** (a), **7** (b), **8** (b), **9** (d), **10** (e), **11** (d), **12** (a), **13** (d), **14** (a), **15** (c), **16** (c), **17** (c), **18** (c), **19** (d), **20** (e).