Chapter 10

MFCS 5: Graph Theory

5 Graph Theory

Informally, a graph is a bunch of dots and lines where the lines connect some pairs of dots. An example is shown in Figure 5.1. The dots are called *nodes* (or *vertices*) and the lines are called *edges*.

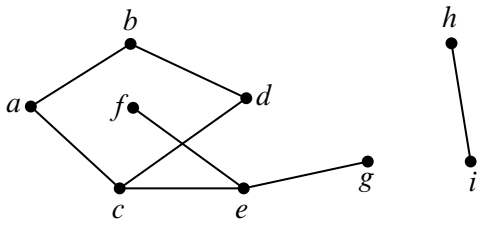

Figure 5.1 An example of a graph with 9 nodes and 8 edges.

Graphs are ubiquitous in computer science because they provide a handy way to represent a relationship between pairs of objects. The objects represent items of interest such as programs, people, cities, or web pages, and we place an edge between a pair of nodes if they are related in a certain way. For example, an edge between a pair of people might indicate that they like (or, in alternate scenarios, that they don't like) each other. An edge between a pair of courses might indicate that one needs to be taken before the other.

In this chapter, we will focus our attention on simple graphs where the relationship denoted by an edge is symmetric. Afterward, in Chapter 6, we consider the situation where the edge denotes a one-way relationship, for example, where one web page points to the other.[1]

5.1 Definitions

5.1.1 Simple Graphs

Definition 5.1.1. A *simple graph* G consists of a nonempty set V, called the *vertices* (aka *nodes*[2]) of G, and a set E of two-element subsets of V. The members of E are called the *edges* of G, and we write $G = (V, E)$.

[1] Two Stanford students analyzed such a graph to become multibillionaires. So, pay attention to graph theory, and who knows what might happen!

[2] We will use the terms vertex and node interchangeably.

The vertices correspond to the dots in Figure 5.1, and the edges correspond to the lines. The graph in Figure 5.1 is expressed mathematically as $G = (V, E)$, where:

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

Note that $\{a, b\}$ and $\{b, a\}$ are different descriptions of the same edge, since sets are unordered. In this case, the graph $G = (V, E)$ has 9 nodes and 8 edges.

Definition 5.1.2. Two vertices in a simple graph are said to be *adjacent* if they are joined by an edge, and an edge is said to be *incident* to the vertices it joins. The number of edges incident to a vertex v is called the *degree* of the vertex and is denoted by $\deg(v)$; equivalently, the degree of a vertex is equals the number of vertices adjacent to it.

For example, in the simple graph shown in Figure 5.1, vertex a is adjacent to b and b is adjacent to d, and the edge $\{a, c\}$ is incident to vertices a and c. Vertex h has degree 1, d has degree 2, and $\deg(e) = 3$. It is possible for a vertex to have degree 0, in which case it is not adjacent to any other vertices. A simple graph does not need to have any edges at all —in which case, the degree of every vertex is zero and $|E| = 0$[3] —but it does need to have at least one vertex, that is, $|V| \geq 1$.

Note that simple graphs do *not* have any *self-loops* (that is, an edge of the form $\{a, a\}$) since an edge is defined to be a set of *two* vertices. In addition, there is at most one edge between any pair of vertices in a simple graph. In other words, a simple graph does not contain *multiedges* or *multiple edges*. That is because E is a set. Lastly, and most importantly, simple graphs do not contain *directed edges* (that is, edges of the form (a, b) instead of $\{a, b\}$).

There's no harm in relaxing these conditions, and some authors do, but we don't need self-loops, multiple edges between the same two vertices, or graphs with no vertices, and it's simpler not to have them around. We will consider graphs with directed edges (called *directed graphs* or *digraphs*) at length in Chapter 6. Since we'll only be considering simple graphs in this chapter, we'll just call them "graphs" from now on.

5.1.2 Some Common Graphs

Some graphs come up so frequently that they have names. The *complete graph* on n vertices, denoted K_n, has an edge between every two vertices, for a total of $n(n-1)/2$ edges. For example, K_5 is shown in Figure 5.2.

The *empty graph* has no edges at all. For example, the empty graph with 5 nodes is shown in Figure 5.3.

[3] The *cardinality*, $|E|$, of the set E is the number of elements in E.

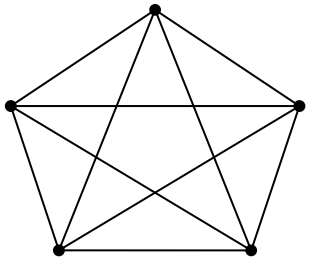

Figure 5.2 The complete graph on 5 nodes, K_5.

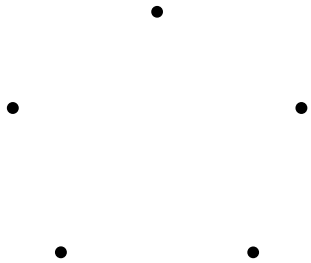

Figure 5.3 The empty graph with 5 nodes.

The n-node graph containing $n - 1$ edges in sequence is known as the *line graph* L_n. More formally, $L_n = (V, E)$ where

$$V = \{v_1, v_2, \ldots, v_n\}$$

and

$$E = \{\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{n-1}, v_n\}\}$$

For example, L_5 is displayed in Figure 5.4.

If we add the edge $\{v_n, v_1\}$ to the line graph L_n, we get the graph C_n consisting of a simple cycle. For example, C_5 is illustrated in Figure 5.5.

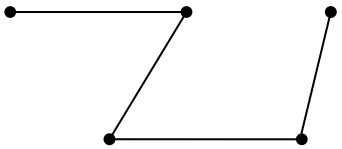

Figure 5.4 The 5-node line graph L_5.

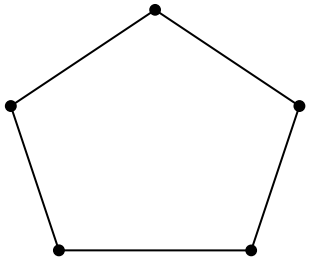

Figure 5.5 The 5-node cycle graph C_5.

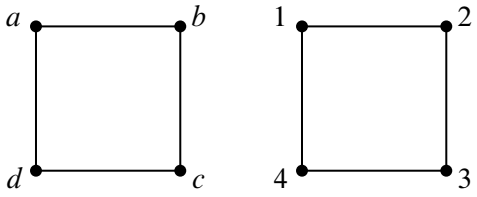

Figure 5.6 Two graphs that are isomorphic to C_4.

5.1.3 Isomorphism

Two graphs that look the same might actually be different in a formal sense. For example, the two graphs in Figure 5.6 are both simple cycles with 4 vertices, but one graph has vertex set $\{a, b, c, d\}$ while the other has vertex set $\{1, 2, 3, 4\}$. Strictly speaking, these graphs are different mathematical objects, but this is a frustrating distinction since the graphs *look the same*!

Fortunately, we can neatly capture the idea of "looks the same" through the notion of graph isomorphism.

Definition 5.1.3. If $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are two graphs, then we say that G_1 is *isomorphic* to G_2 iff there exists a *bijection*[4] $f : V_1 \to V_2$ such that for every pair of vertices $u, v \in V_1$:

$$\{u, v\} \in E_1 \quad \text{iff} \quad \{f(u), f(v)\} \in E_2.$$

The function f is called an *isomorphism* between G_1 and G_2.

In other words, two graphs are isomorphic if they are the same up to a relabeling of their vertices. For example, here is an isomorphism between vertices in the two

[4]A bijection $f : V_1 \to V_2$ is a function that associates every node in V_1 with a unique node in V_2 and vice-versa. We will study bijections more deeply in Part III.

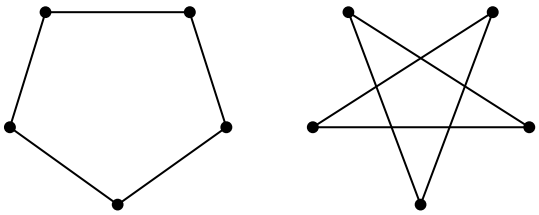

Figure 5.7 Two ways of drawing C_5.

graphs shown in Figure 5.6:

a corresponds to 1	b corresponds to 2
d corresponds to 4	c corresponds to 3.

You can check that there is an edge between two vertices in the graph on the left if and only if there is an edge between the two corresponding vertices in the graph on the right.

Two isomorphic graphs may be drawn very differently. For example, we have shown two different ways of drawing C_5 in Figure 5.7.

Isomorphism preserves the connection properties of a graph, abstracting out what the vertices are called, what they are made out of, or where they appear in a drawing of the graph. More precisely, a property of a graph is said to be *preserved under isomorphism* if whenever G has that property, every graph isomorphic to G also has that property. For example, isomorphic graphs must have the same number of vertices. What's more, if f is a graph isomorphism that maps a vertex, v, of one graph to the vertex, $f(v)$, of an isomorphic graph, then by definition of isomorphism, every vertex adjacent to v in the first graph will be mapped by f to a vertex adjacent to $f(v)$ in the isomorphic graph. This means that v and $f(v)$ will have the same degree. So if one graph has a vertex of degree 4 and another does not, then they can't be isomorphic. In fact, they can't be isomorphic if the number of degree 4 vertices in each of the graphs is not the same.

Looking for preserved properties can make it easy to determine that two graphs are not isomorphic, or to actually find an isomorphism between them if there is one. In practice, it's frequently easy to decide whether two graphs are isomorphic. However, no one has yet found a *general* procedure for determining whether two graphs are isomorphic that is *guaranteed* to run in polynomial time[5] in $|V|$.

Having such a procedure would be useful. For example, it would make it easy to search for a particular molecule in a database given the molecular bonds. On

[5] *I.e.*, in an amount of time that is upper-bounded by $|V|^c$ where c is a fixed number independent of $|V|$.

the other hand, knowing there is no such efficient procedure would also be valuable: secure protocols for encryption and remote authentication can be built on the hypothesis that graph isomorphism is computationally exhausting.

5.1.4 Subgraphs

Definition 5.1.4. A graph $G_1 = (V_1, E_1)$ is said to be a *subgraph* of a graph $G_2 = (V_2, E_2)$ if $V_1 \subseteq V_2$ and $E_1 \subseteq E_2$.

For example, the empty graph on n nodes is a subgraph of L_n, L_n is a subgraph of C_n, and C_n is a subgraph of K_n. Also, the graph $G = (V, E)$ where

$$V = \{g, h, i\} \quad \text{and} \quad E = \{\{h, i\}\}$$

is a subgraph of the graph in Figure 5.1. On the other hand, any graph containing an edge $\{g, h\}$ would not be a subgraph of the graph in Figure 5.1 because the graph in Figure 5.1 does not contain this edge.

Note that since a subgraph is itself a graph, the endpoints of any edge in a subgraph must also be in the subgraph. In other words if $G' = (V', E')$ is a subgraph of some graph G, and $\{v_i, v_j\} \in E'$, then it must be the case that $v_i \in V'$ and $v_j \in V'$.

5.1.5 Weighted Graphs

Sometimes, we will use edges to denote a connection between a pair of nodes where the connection has a *capacity* or *weight*. For example, we might be interested in the capacity of an Internet fiber between a pair of computers, the resistance of a wire between a pair of terminals, the tension of a spring connecting a pair of devices in a dynamical system, the tension of a bond between a pair of atoms in a molecule, or the distance of a highway between a pair of cities.

In such cases, it is useful to represent the system with an *edge-weighted* graph (aka a *weighted graph*). A weighted graph is the same as a simple graph except that we associate a real number (that is, the weight) with each edge in the graph. Mathematically speaking, a weighted graph consists of a graph $G = (V, E)$ and a weight function $w : E \to \mathbb{R}$. For example, Figure 5.8 shows a weighted graph where the weight of edge $\{a, b\}$ is 5.

5.1.6 Adjacency Matrices

There are many ways to represent a graph. We have already seen two ways: you can draw it, as in Figure 5.8 for example, or you can represent it with sets —as in $G = (V, E)$. Another common representation is with an adjacency matrix.

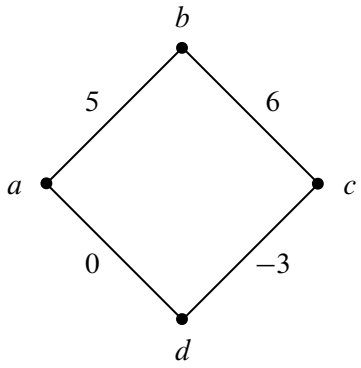

Figure 5.8 A 4-node weighted graph where the edge $\{a, b\}$ has weight 5.

$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 5 & 0 & 0 \\ 5 & 0 & 6 & 0 \\ 0 & 6 & 0 & -3 \\ 0 & 0 & -3 & 0 \end{pmatrix}$$

(a) (b)

Figure 5.9 Examples of adjacency matrices. (a) shows the adjacency matrix for the graph in Figure 5.6(a) and (b) shows the adjacency matrix for the weighted graph in Figure 5.8. In each case, we set $v_1 = a$, $v_2 = b$, $v_3 = c$, and $v_4 = d$ to construct the matrix.

Definition 5.1.5. Given an n-node graph $G = (V, E)$ where $V = \{v_1, v_2, \ldots, v_n\}$, the *adjacency matrix* for G is the $n \times n$ matrix $A_G = \{a_{ij}\}$ where

$$a_{ij} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \in E \\ 0 & \text{otherwise.} \end{cases}$$

If G is a weighted graph with edge weights given by $w : E \to \mathbb{R}$, then the adjacency matrix for G is $A_G = \{a_{ij}\}$ where

$$a_{ij} = \begin{cases} w(\{v_i, v_j\}) & \text{if } \{v_i, v_j\} \in E \\ 0 & \text{otherwise.} \end{cases}$$

For example, Figure 5.9 displays the adjacency matrices for the graphs shown in Figures 5.6(a) and 5.8 where $v_1 = a$, $v_2 = b$, $v_3 = c$, and $v_4 = d$.

One way to avoid an accidental build-up error is to use a "shrink down, grow back" process in the inductive step; that is, start with a size $n+1$ graph, remove a vertex (or edge), apply the inductive hypothesis $P(n)$ to the smaller graph, and then add back the vertex (or edge) and argue that $P(n+1)$ holds. Let's see what would have happened if we'd tried to prove the claim above by this method:

Revised inductive step: We must show that $P(n)$ implies $P(n+1)$ for all $n \geq 1$. Consider an $(n+1)$-vertex graph G in which every vertex has degree at least 1. Remove an arbitrary vertex v, leaving an n-vertex graph G' in which every vertex has degree... uh oh!

The reduced graph G' might contain a vertex of degree 0, making the inductive hypothesis $P(n)$ inapplicable! We are stuck—and properly so, since the claim is false!

Always use shrink-down, grow-back arguments and you'll never fall into this trap.

5.6 Around and Around We Go

5.6.1 Cycles and Closed Walks

Definition 5.6.1. A *closed walk*[15] in a graph G is a sequence of vertices

$$v_0, v_1, \ldots, v_k$$

and edges

$$\{v_0, v_1\}, \{v_1, v_2\}, \ldots, \{v_{k-1}, v_k\}$$

where v_0 is the same node as v_k and $\{v_i, v_{i+1}\}$ is an edge of G for all i where $0 \leq i < k$. The *length* of the closed walk is k. A closed walk is said to be a *cycle* if $k \geq 3$ and $v_0, v_1, \ldots, v_{k-1}$ are all different.

For example, b, c, d, e, c, b is a closed walk of length 5 in the graph shown in Figure 5.18. It is not a cycle since it contains node c twice. On the other hand, c, d, e, c is a cycle of length 3 in this graph since every node appears just once.

There are many ways to represent the same closed walk or cycle. For example, b, c, d, e, c, b is the same as c, d, e, c, b, c (just starting at node c instead of node b) and the same as b, c, e, d, c, b (just reversing the direction).

[15] Some texts use the word *cycle* for our definition of closed walk and *simple cycle* for our definition of cycle.

Cycles are similar to paths, except that the last node is the first node and the notion of first and last does not matter. Indeed, there are many possible vertex orders that can be used to describe cycles and closed walks, whereas walks and paths have a prescribed beginning, end, and ordering.

5.6.2 Odd Cycles and 2-Colorability

We have already seen that determining the chromatic number of a graph is a challenging problem. There is a special case where this problem is very easy; namely, the case where every cycle in the graph has even length. In this case, the graph is 2-colorable! Of course, this is optimal if the graph has any edges at all. More generally, we will prove

Theorem 5.6.2. *The following properties of a graph are equivalent (that is, if the graph has any one of the properties, then it has all of the properties):*

1. *The graph is bipartite.*
2. *The graph is 2-colorable.*
3. *The graph does not contain any cycles with odd length.*
4. *The graph does not contain any closed walks with odd length.*

Proof. We will show that property 1 IMPLIES property 2, property 2 IMPLIES property 3, property 3 IMPLIES property 4, and property 4 IMPLIES property 1. This will show that all four properties are equivalent by repeated application of Rule 2.1.2 in Section 2.1.2.

1 IMPLIES 2 Assume that $G = (V, E)$ is a bipartite graph. Then V can be partitioned into two sets L and R so that no edge connects a pair of nodes in L nor a pair of nodes in R. Hence, we can use one color for all the nodes in L and a second color for all the nodes in R. Hence $\chi(G) = 2$.

2 IMPLIES 3 Let $G = (V, E)$ be a 2-colorable graph and

$$C ::= v_0, v_1, \ldots, v_k$$

be any cycle in G. Consider any 2-coloring for the nodes of G. Since $\{v_i, v_{i+1}\} \in E$, v_i and v_{i+1} must be differently colored for $0 \leq i < k$. Hence v_0, v_2, v_4, ..., have one color and v_1, v_3, v_5, ..., have the other color. Since C is a cycle, v_k is the same node as v_0, which means they must have the same color, and so k must be an even number. This means that C has even length.

3 IMPLIES 4 The proof is by contradiction. Assume for the purposes of contradiction that G is a graph that does not contain any cycles with odd length (that is, G satisfies Property 3) but that G *does* contain a closed walk with odd length (that is, G does not satisfy Property 4).

Let

$$w ::= v_0, v_1, v_2, \ldots, v_k$$

be the *shortest* closed walk with odd length in G. Since G has no odd-length cycles, w cannot be a cycle. Hence $v_i = v_j$ for some $0 \leq i < j < k$. This means that w is the union of two closed walks:

$$v_0, v_1, \ldots, v_i, v_{j+1}, v_{j+2}, \ldots, v_k$$

and

$$v_i, v_{i+1}, \ldots, v_j.$$

Since w has odd length, one of these two closed walks must also have odd length and be shorter than w. This contradicts the minimality of w. Hence 3 IMPLIES 4.

4 IMPLIES 1 Once again, the proof is by contradiction. Assume for the purposes of contradictin that G is a graph without any closed walks with odd length (that is, G satisfies Property 4) but that G is *not* bipartite (that is, G does not satisfy Property 1).

Since G is not bipartite, it must contain a connected component $G' = (V', E')$ that is not bipartite. Let v be some node in V'. For every node $u \in V'$, define

$$\begin{aligned}\text{dist}(u) ::= &\text{ the length of the shortest path from } u \text{ to } v \text{ in } G'.\\ &\text{If } u = v, \text{ the distance is zero.}\end{aligned}$$

Partition V' into sets L and R so that

$$\begin{aligned}L &= \{\, u \mid \text{dist}(u) \text{ is even} \,\},\\ R &= \{\, u \mid \text{dist}(u) \text{ is odd} \,\}.\end{aligned}$$

Since G' is not bipartite, there must be a pair of adjacent nodes u_1 and u_2 that are both in L or both in R. Let e denote the edge incident to u_1 and u_2.

Let P_i denote a shortest path in G' from u_i to v for $i = 1, 2$. Because u_1 and u_2 are both in L or both in R, it must be the case that P_1 and P_2 both have even length or they both have odd length. In either case, the union of P_1, P_2, and e forms a closed walk with odd length, which is a contradiction. Hence 4 IMPLIES 1. ■

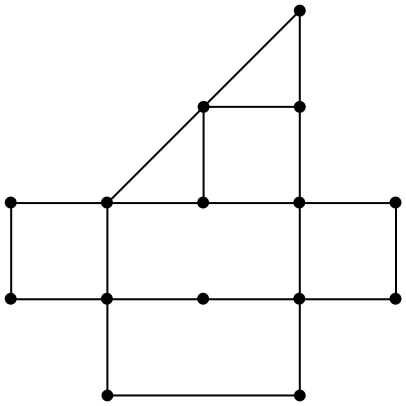

Figure 5.23 A possible floor plan for a museum. Can you find a walk that traverses every edge exactly once?

Theorem 5.6.2 turns out to be useful since bipartite graphs come up fairly often in practice. We'll see examples when we talk about planar graphs in Section 5.8 and when we talk about packet routing in communication networks in Chapter 6.

5.6.3 Euler Tours

Can you walk every hallway in the Museum of Fine Arts *exactly once*? If we represent hallways and intersections with edges and vertices, then this reduces to a question about graphs. For example, could you visit every hallway exactly once in a museum with the floor plan in Figure 5.23?

The entire field of graph theory began when Euler asked whether the seven bridges of Königsberg could all be traversed exactly once—essentially the same question we asked about the Museum of Fine Arts. In his honor, an *Euler walk* is a defined to be a walk that traverses every edge in a graph exactly once. Similarly, an *Euler tour* is an Euler walk that starts and finishes at the same vertex. Graphs with Euler tours and Euler walks both have simple characterizations.

Theorem 5.6.3. *A connected graph has an Euler tour if and only if every vertex has even degree.*

Proof. We first show that if a graph has an Euler tour, then every vertex has even degree. Assume that a graph $G = (V, E)$ has an Euler tour $v_0, v_1, \ldots, v_k$ where $v_k = v_0$. Since every edge is traversed once in the tour, $k = |E|$ and the degree of a node u in G is the number of times that node appears in the sequence $v_0, v_1, \ldots, v_{k-1}$ times two. We multiply by two since if $u = v_i$ for some i where $0 < i < k$, then both $\{v_{i-1}, v_i\}$ and $\{v_i, v_{i+1}\}$ are edges incident to u in G. If $u = v_0 = v_k$,

then both $\{v_{k-1}, v_k\}$ and $\{v_0, v_1\}$ are edges incident to u in G. Hence, the degree of every node is even.

We next show that if the degree of every node is even in a graph $G = (V, E)$, then there is an Euler tour. Let

$$W ::= v_0, v_1, \ldots, v_k$$

be the longest walk in G that traverses *no edge more than once*[16]. W must traverse every edge incident to v_k; otherwise the walk could be extended and W would not be the longest walk that traverses all edges at most once. Moreover, it must be that $v_k = v_0$ and that W is a closed walk, since otherwise v_k would have odd degree in W (and hence in G), which is not possible by assumption.

We conclude the argument with a proof by contradiction. Suppose that W is not an Euler tour. Because G is a connected graph, we can find an edge not in W but incident to some vertex in W. Call this edge $\{u, v_i\}$. But then we can construct a walk W' that is longer than W but that still uses no edge more than once:

$$W' ::= u, v_i, v_{i+1}, \ldots, v_k, v_1, v_2, \ldots, v_i.$$

This contradicts the definition of W, so W must be an Euler tour after all. ■

It is not difficult to extend Theorem 5.6.3 to prove that a connected graph G has an Euler walk if and only if precisely 0 or 2 nodes in G have odd degree. Hence, we can conclude that the graph shown in Figure 5.23 has an Euler walk but not an Euler tour since the graph has precisely two nodes with odd degree.

Although the proof of Theorem 5.6.3 does not explicitly define a method for finding an Euler tour when one exists, it is not hard to modify the proof to produce such a method. The idea is to grow a tour by continually splicing in closed walks until all the edges are consumed.

5.6.4 Hamiltonian Cycles

Hamiltonian cycles are the unruly cousins of Euler tours.

Definition 5.6.4. A *Hamiltonian cycle* in a graph G is a cycle that visits every *node* in G exactly once. Similarly, a *Hamiltonian* path is a path in G that visits every node exactly once.

[16] Did you notice that we are using a variation of the Well Ordering Principle here when we implicitly assume that a longest walk exists? This is ok since the length of a walk where no edge is used more than once is at most $|E|$.

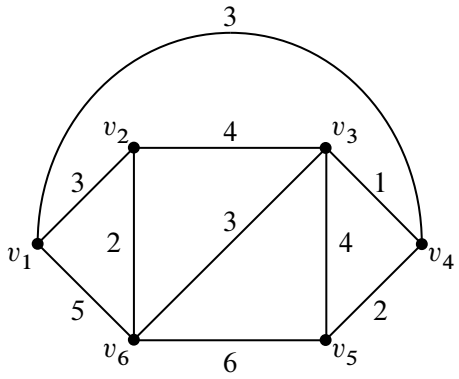

Figure 5.24 A weighted graph. Can you find a cycle with weight 15 that visits every node exactly once?

Although Hamiltonian cycles sound similar to Euler tours—one visits every node once while the other visits every edge once—finding a Hamiltonian cycle can be a lot harder than finding an Euler tour. The same is true for Hamiltonian paths. This is because no one has discovered a simple characterization of all graphs with a Hamiltonian cycle. In fact, determining whether a graph has a Hamiltonian cycle is the same category of problem as the SAT problem of Section 1.5 and the coloring problem in Section 5.3; you get a million dollars for finding an efficient way to determine when a graph has a Hamiltonian cycle—or proving that no procedure works efficiently on all graphs.

5.6.5 The Traveling Salesperson Problem

As if the problem of finding a Hamiltonian cycle is not hard enough, when the graph is weighted, we often want to find a Hamiltonian cycle that has least possible weight. This is a very famous optimization problem known as the Traveling Salesperson Problem.

Definition 5.6.5. Given a weighted graph G, the *weight* of a cycle in G is defined as the sum of the weights of the edges in the cycle.

For example, consider the graph shown in Figure 5.24 and suppose that you would like to visit every node once and finish at the node where you started. Can you find way to do this by traversing a cycle with weight 15?

Needless to say, if you can figure out a fast procedure that finds the optimal cycle for the traveling salesperson, let us know so that we can win a million dollars.

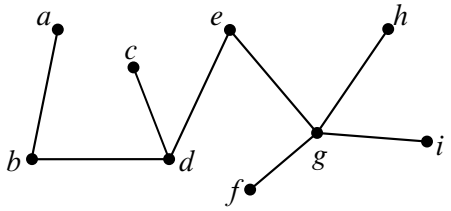

Figure 5.25 A 9-node tree.

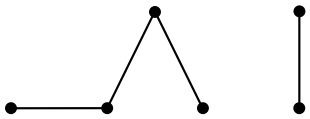

Figure 5.26 A 6-node forest consisting of 2 component trees. Note that this 6-node graph is not itself a tree since it is not connected.

5.7 Trees

As we have just seen, finding good cycles in a graph can be trickier than you might first think. But what if a graph has no cycles at all? Sounds pretty dull. But graphs without cycles (called *acyclic graphs*) are probably the most important graphs of all when it comes to computer science.

5.7.1 Definitions

Definition 5.7.1. A connected acyclic graph is called a *tree*.

For example, Figure 5.25 shows an example of a 9-node tree.

The graph shown in Figure 5.26 is not a tree since it is not connected, but it is a forest. That's because, of course, it consists of a collection of trees.

Definition 5.7.2. If every connected component of a graph G is a tree, then G is a *forest*.

One of the first things you will notice about trees is that they tend to have a lot of nodes with degree one. Such nodes are called *leaves*.

Definition 5.7.3. A *leaf* is a node with degree 1 in a tree (or forest).

For example, the tree in Figure 5.25 has 5 leaves and the forest in Figure 5.26 has 4 leaves.

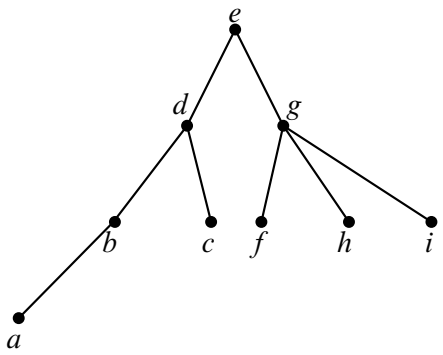

Figure 5.27 The tree from Figure 5.25 redrawn in a leveled fashion, with node E as the root.

Trees are a fundamental data structure in computer science. For example, information is often stored in tree-like data structures and the execution of many recursive programs can be modeled as the traversal of a tree. In such cases, it is often useful to draw the tree in a leveled fashion where the node in the top level is identified as the *root*, and where every edge joins a *parent* to a *child*. For example, we have redrawn the tree from Figure 5.25 in this fashion in Figure 5.27. In this example, node d is a child of node e and a parent of nodes b and c.

In the special case of *ordered binary trees*, every node is the parent of at most 2 children and the children are labeled as being a left-child or a right-child.

5.7.2 Properties

Trees have many unique properties. We have listed some of them in the following theorem.

Theorem 5.7.4. *Every tree has the following properties:*

1. *Any connected subgraph is a tree.*
2. *There is a unique simple path between every pair of vertices.*
3. *Adding an edge between nonadjacent nodes in a tree creates a graph with a cycle.*
4. *Removing any edge disconnects the graph.*
5. *If the tree has at least two vertices, then it has at least two leaves.*
6. *The number of vertices in a tree is one larger than the number of edges.*

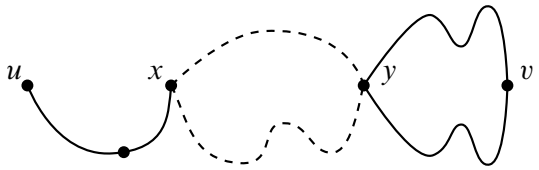

Figure 5.28 If there are two paths between u and v, the graph must contain a cycle.

Proof. 1. A cycle in a subgraph is also a cycle in the whole graph, so any subgraph of an acyclic graph must also be acyclic. If the subgraph is also connected, then by definition, it is a tree.

2. Since a tree is connected, there is at least one path between every pair of vertices. Suppose for the purposes of contradiction, that there are two different paths between some pair of vertices u and v. Beginning at u, let x be the first vertex where the paths diverge, and let y be the next vertex they share. (For example, see Figure 5.28.) Then there are two paths from x to y with no common edges, which defines a cycle. This is a contradiction, since trees are acyclic. Therefore, there is exactly one path between every pair of vertices.

3. An additional edge $\{u, v\}$ together with the unique path between u and v forms a cycle.

4. Suppose that we remove edge $\{u, v\}$. Since the tree contained a unique path between u and v, that path must have been $\{u, v\}$. Therefore, when that edge is removed, no path remains, and so the graph is not connected.

5. Let $v_1, \ldots, v_m$ be the sequence of vertices on a longest path in the tree. Then $m \geq 2$, since a tree with two vertices must contain at least one edge. There cannot be an edge $\{v_1, v_i\}$ for $2 < i \leq m$; otherwise, vertices $v_1, \ldots, v_i$ would from a cycle. Furthermore, there cannot be an edge $\{u, v_1\}$ where u is not on the path; otherwise, we could make the path longer. Therefore, the only edge incident to v_1 is $\{v_1, v_2\}$, which means that v_1 is a leaf. By a symmetric argument, v_m is a second leaf.

6. We use induction on the proposition $P(n) ::=$ there are $n - 1$ edges in any n-vertex tree.

   Base Case ($n = 1$): $P(1)$ is true since a tree with 1 node has 0 edges and $1 - 1 = 0$.

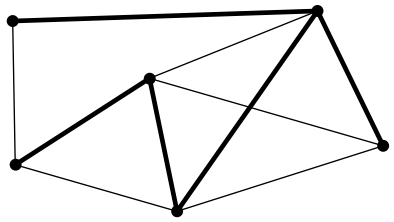

Figure 5.29 A graph where the edges of a spanning tree have been thickened.

Inductive step: Now suppose that $P(n)$ is true and consider an $(n+1)$-vertex tree, T. Let v be a leaf of the tree. You can verify that deleting a vertex of degree 1 (and its incident edge) from any connected graph leaves a connected subgraph. So by part 1 of Theorem 5.7.4, deleting v and its incident edge gives a smaller tree, and this smaller tree has $n-1$ edges by induction. If we re-attach the vertex v and its incident edge, then we find that T has $n = (n+1) - 1$ edges. Hence, $P(n+1)$ is true, and the induction proof is complete. ■

Various subsets of properties in Theorem 5.7.4 provide alternative characterizations of trees, though we won't prove this. For example, a *connected* graph with a number of vertices one larger than the number of edges is necessarily a tree. Also, a graph with unique paths between every pair of vertices is necessarily a tree.

5.7.3 Spanning Trees

Trees are everywhere. In fact, every connected graph contains a subgraph that is a tree with the same vertices as the graph. This is a called a *spanning tree* for the graph. For example, Figure 5.29 is a connected graph with a spanning tree highlighted.

Theorem 5.7.5. *Every connected graph contains a spanning tree.*

Proof. By contradiction. Assume there is some connected graph G that has no spanning tree and let T be a connected subgraph of G, with the same vertices as G, and with the smallest number of edges possible for such a subgraph. By the assumption, T is not a spanning tree and so it contains some cycle:

$$\{v_0, v_1\}, \{v_1, v_2\}, \ldots, \{v_k, v_0\}$$

Suppose that we remove the last edge, $\{v_k, v_0\}$. If a pair of vertices x and y was joined by a path not containing $\{v_k, v_0\}$, then they remain joined by that path. On the other hand, if x and y were joined by a path containing $\{v_k, v_0\}$, then they

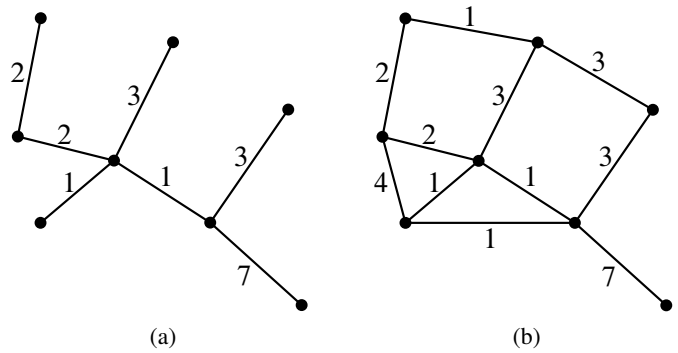

Figure 5.30 A spanning tree (a) with weight 19 for a graph (b).

remain joined by a walk containing the remainder of the cycle. By Lemma 5.4.2, they must also then be joined by a path. So all the vertices of G are still connected after we remove an edge from T. This is a contradiction, since T was defined to be a minimum size connected subgraph with all the vertices of G. So the theorem must be true. ■

5.7.4 Minimum Weight Spanning Trees

Spanning trees are interesting because they connect all the nodes of a graph using the smallest possible number of edges. For example the spanning tree for the 6-node graph shown in Figure 5.29 has 5 edges.

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

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

In any case, we typically represent the cost to connect pairs of nodes with a weighted edge, where the weight of the edge is its cost. The weight of a spanning tree is then just the sum of the weights of the edges in the tree. For example, the weight of the spanning tree shown in Figure 5.30 is 19.

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

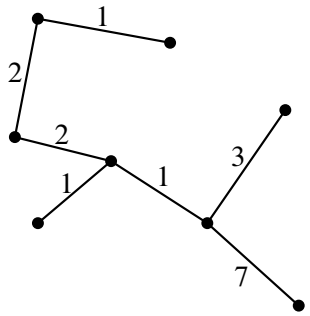

Figure 5.31 An MST with weight 17 for the graph in Figure 5.30(b).

Definition 5.7.6. The *min-weight spanning tree* (MST) of an edge-weighted graph G is the spanning tree of G with the smallest possible sum of edge weights.

Is the spanning tree shown in Figure 5.30(a) an MST of the weighted graph shown in Figure 5.30(b)? Actually, it is not, since the tree shown in Figure 5.31 is also a spanning tree of the graph shown in Figure 5.30(b), and this spanning tree has weight 17.

What about the tree shown in Figure 5.31? Is it an MST? It seems to be, but how do we prove it? In general, how do we find an MST? We could, of course, enumerate all trees, but this could take forever for very large graphs.

Here are two possible algorithms:

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

Algorithm 2. *Grow a subgraph one edge at a time by adding the minimum-weight edge possible to the subgraph, making sure that you have an acyclic subgraph at each step.*

For example, in the weighted graph we have been considering, we might run Algorithm 1 as follows. We would start by choosing one of the weight 1 edges, since this is the smallest weight in the graph. Suppose we chose the weight 1 edge on the bottom of the triangle of weight 1 edges in our graph. This edge is incident to two weight 1 edges, a weight 4 edge, a weight 7 edge, and a weight 3 edge. We would then choose the incident edge of minimum weight. In this case, one of the two weight 1 edges. At this point, we cannot choose the third weight 1 edge since this would form a cycle, but we can continue by choosing a weight 2 edge. We might end up with the spanning tree shown in Figure 5.32, which has weight 17, the smallest we've seen so far.

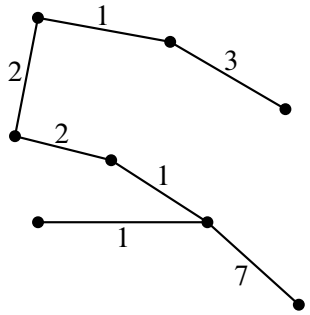

Figure 5.32 A spanning tree found by Algorithm 1.

Now suppose we instead ran Algorithm 2 on our graph. We might again choose the weight 1 edge on the bottom of the triangle of weight 1 edges in our graph. Now, instead of choosing one of the weight 1 edges it touches, we might choose the weight 1 edge on the top of the graph. Note that this edge still has minimum weight, and does not cause us to form a cycle, so Algorithm 2 can choose it. We would then choose one of the remaining weight 1 edges. Note that neither causes us to form a cycle. Continuing the algorithm, we may end up with the same spanning tree in Figure 5.32, though this need not always be the case.

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

These are examples of greedy approaches to optimization. Sometimes it works and sometimes it doesn't. The good news is that it works to find the MST. In fact, both variations work. It's a little easier to prove it for Algorithm 2, so we'll do that one here.

Theorem 5.7.7. *For any connected, weighted graph G, Algorithm 2 produces an MST for G.*

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

The key to doing all of this is to show that the algorithm never gets stuck or goes in a bad direction by adding an edge that will keep us from ultimately producing an MST. The natural way to prove this is to show that the set of edges selected at any point is contained in some MST—that is, we can always get to where we need to be. We'll state this as a lemma.

Lemma 5.7.8. *For any $m \geq 0$, let S consist of the first m edges selected by Algorithm 2. Then there exists some MST $T = (V, E)$ for G such that $S \subseteq E$, that is, the set of edges that we are growing is always contained in some MST.*

We'll prove this momentarily, but first let's see why it helps to prove the theorem. Assume the lemma is true. Then how do we know Algorithm 2 can always find an edge to add without creating a cycle? Well, as long as there are fewer than $n-1$ edges picked, there exists some edge in $E - S$ and so there is an edge that we can add to S without forming a cycle. Next, how do we know that we get an MST at the end? Well, once $m = n - 1$, we know that S is an MST.

Ok, so the theorem is an easy corollary of the lemma. To prove the lemma, we'll use induction on the number of edges chosen by the algorithm so far. This is very typical in proving that an algorithm preserves some kind of invariant condition—induct on the number of steps taken, that is, the number of edges added.

Our inductive hypothesis $P(m)$ is the following: for any G and any set S of m edges initially selected by Algorithm 2, there exists an MST $T = (V, E)$ of G such that $S \subseteq E$.

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

For the inductive step, we assume $P(m)$ holds and show that it implies $P(m+1)$. Let e denote the $(m+1)$st edge selected by Algorithm 2, and let S denote the first m edges selected by Algorithm 2. Let $T^* = (V^*, E^*)$ be the MST such that $S \subseteq E^*$, which exists by the inductive hypothesis. There are now two cases:

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

Case 2: $e \notin E^*$, as illustrated in Figure 5.33. Now we need to find a different MST that contains S and e.

What happens when we add e to T^*? Since T^* is a tree, we get a cycle. (Here we used part 3 of Theorem 5.7.4.) Moreover, the cycle cannot only contains edges in S, since e was chosen so that together with the edges in S, it does not form a cycle. This implies that $\{e\} \cup T^*$ contains a cycle that contains an edge e' of $E^* - S$. For example, such an e' is shown in Figure 5.33.

Note that the weight of e is at most that of e'. This is because Algorithm 2 picks the minimum weight edge that does not make a cycle with S. Since $e' \in T^*$, e' cannot make a cycle with S and if the weight of e were greater than the weight of e', Algorithm 2 would not have selected e ahead of e'.

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

Claim 5.7.9. *T^{**} is an MST.*

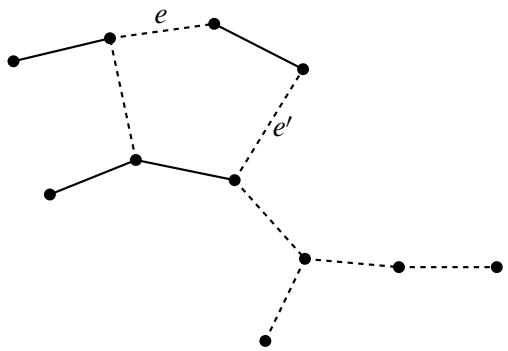

Figure 5.33 The graph formed by adding e to T^*. Edges of S are denoted with solid lines and edges of $E^* - S$ are denoted with dashed lines.

Proof of claim. We first show that T^{**} is a spanning tree. T^{**} is acyclic because it was produced by removing an edge from the only cycle in $T^* \cup \{e\}$. T^{**} is connected since the edge we deleted from $T^* \cup \{e\}$ was on a cycle. Since T^{**} contains all the nodes of G, it must be a spanning tree for G.

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

Since $S \cup \{e\} \subseteq E^{**}$, $P(m+1)$ holds. Thus, Algorithm 2 must eventually produce an MST. This will happens when it adds $n-1$ edges to the subgraph it builds. ■

So now we know for sure that the MST for our example graph has weight 17 since it was produced by Algorithm 2. And we have a fast algorithm for finding a minimum-weight spanning tree for any graph.

5.8 Planar Graphs

5.8.1 Drawing Graphs in the Plane

Suppose there are three dog houses and three human houses, as shown in Figure 5.34. Can you find a route from each dog house to each human house such that no route crosses any other route?

A *quadrapus* is a little-known animal similar to an octopus, but with four arms. Suppose there are five quadrapi resting on the sea floor, as shown in Figure 5.35.