GRAPH THEORY

The first text on graph theory ([König]) appeared in 1936. Undoubtedly, one of the reasons for the recent interest in graph theory is its applicability in many diverse fields, including computer science, chemistry, operations research, electrical engineering, linguistics, and economics.

We begin with some basic graph terminology and examples. We then discuss some important concepts in graph theory, including paths and cycles.

Two classical graph problems, the existence of Hamiltonian cycles and the traveling salesperson problem, are then considered. A shortest-path algorithm is presented that efficiently finds the shortest path between two given points. After presenting ways of representing graphs, we study the question of when two graphs are essentially the same (i.e., when two graphs are isomorphic) and when a graph can be drawn in the plane without having any of its edges crossed. We conclude by presenting a solution based on a graph model to the Instant Insanity puzzle.

8.1 Introduction

Figure 8.1.1 shows the highway system in Wyoming that a particular person is responsible for inspecting. Specifically, this road inspector must travel all these roads and file reports on road conditions, visibility of lines on the roads, status of traffic signs, and so on. Since the road inspector lives in Greybull, He would be to start in Greybull, travel each of the roads exactly once, and return to Greybull. Is this possible?

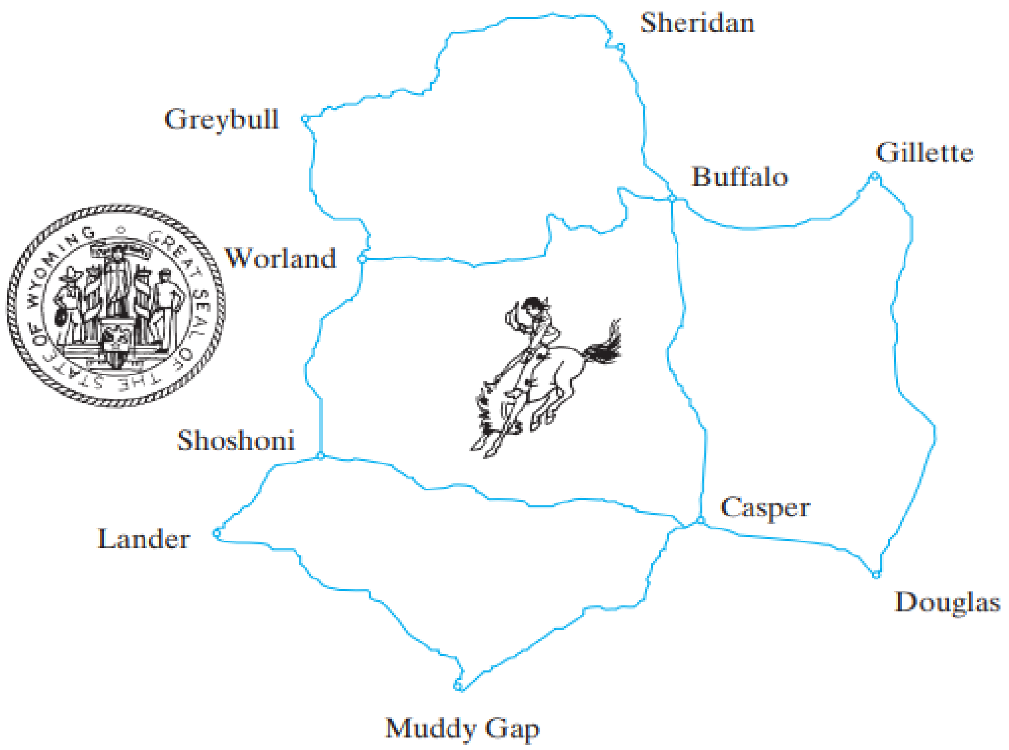


Figure 8.1.1 Part of the Wyoming highway system.

the only information of importance is which vertices are connected by which edges. For this reason, the graph of Figure 8.1.2 could just as well be drawn as in Figure 8.1.3.

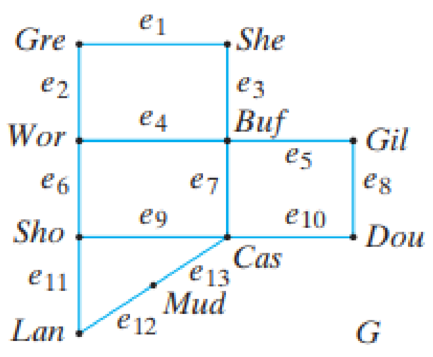


Figure 8.1.2 A graph model of the highway system shown in Figure 8.1.1.

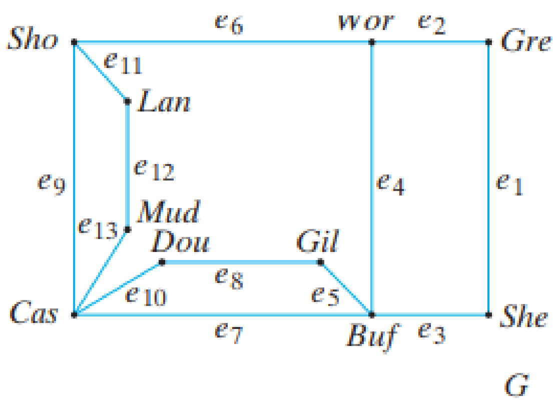


Figure 8.1.3 An alternative, but equivalent, graph model of the highway system shown in Figure 8.1.1.

The problem can be modeled as a **graph.** In fact, since graphs are drawn with dots and lines, they look like road maps. In Figure 8.1.2, we have drawn a graph *G* that models the map of Figure 8.1.1. The dots in Figure 8.1.2 are called **vertices**, and the lines that connect the vertices are called **edges**.

We have labeled each vertex with the first three letters of the city to which it corresponds. We have labeled the edges $e_1, \ldots, e_{13}$. When we draw a graph, the only information of importance is which vertices are connected by which edges. For this reason, the graph of Figure 8.1.2 could just as well be drawn as in Figure 8.1.3.

If we start at a vertex v_0, travel along an edge to vertex v_1, travel along another edge to vertex v_2, and so on, and eventually arrive at vertex v_n, we call the complete tour a **path** from v_0 to v_n.

The path that starts at *She*, then goes to *Buf*, and ends at *Gil* corresponds to a trip on the map of Figure 8.1.1 that begins in Sheridan, goes to Buffalo, and ends at Gillette. The road inspector’s problem can be rephrased for the graph model *G* in the following way: Is there a path from vertex *Gre* to vertex *Gre* that traverses every edge exactly once?

We can show that the road inspector cannot start in Greybull, travel each of the roads exactly once, and return to Greybull. To put the answer in graph terms, there is no path from vertex *Gre* to vertex *Gre* in Figure 8.1.2 that traverses every edge exactly once. To see this, suppose that there is such a path and consider vertex *Wor*. Each time we arrive at *Wor* on some edge, we must leave *Wor* on a different edge. Furthermore, every edge that touches *Wor* must be used. Thus, the edges at *Wor* occur in pairs. It follows that an even number of edges must touch *Wor*. Since three edges touch *Wor*, we have a contradiction. Therefore, there is no path from vertex *Gre* to vertex *Gre* in Figure

8.1.2 that traverses every edge exactly once. The argument applies to an arbitrary graph *G*. If *G* has a path from vertex *v* to *v* that traverses every edge exactly once, an even number of edges must touch each vertex. We discuss this problem in greater detail in Section 8.2.

Definition 8.1.1 A *graph* (or *undirected graph*) *G* consists of a set ***V*** of *vertices* (or *nodes*) and a set ***E*** of *edges* (or *arcs*) such that each edge $e \in E$ is associated with an unordered pair of vertices. If there is a unique edge ***e*** associated with the vertices *v* and *w*, we write $e = (v,w)$

or *e* = *(w, v)*. In this context, *(v,w)* denotes an edge between ***v*** and ***w*** in an undirected graph and *not* an ordered pair.

A *directed graph* (or *digraph*) *G* consists of a set ***V*** of *vertices* (or *nodes*) and a set *E* of *edges* (or *arcs*) such that each edge $e \in E$ is associated with an ordered pair of vertices. If there is a unique edge ***e*** associated with the ordered pair ***(v,w)*** of vertices, we write *e* = *(**v,w**)*, which denotes an edge from *v* to *w*. An edge ***e*** in a graph (undirected or directed) that is associated with the pair of vertices ***v*** and

w is said to be ***incident*** *on* ***v*** and ***w***, and ***v*** and ***w*** are said to be *incident on* ***e*** and to be ***adjacent vertices***.

If *G* is a graph (undirected or directed) with vertices ***V*** and edges ***E***, we write ***G=(V, E)***. Unless specified otherwise, the sets ***E*** and ***V*** are assumed to be finite, and ***V*** is assumed to be nonempty

Example 8.1.2 In Figure 8.1.2 the (undirected) graph *G* consists of the set

$$V = \{Gre, She, Wor, Buf, Gil, Sho, Cas, Dou, Lan, Mud\}$$

of vertices and the set

$$E = \{e_1, e_2, \ldots, e_{13}\}$$

of edges. Edge e_1 is associated with the unordered pair {***Gre, She***} of vertices, and edge e_{10} is associated with the unordered pair {***Cas, Dou***} of vertices. Edge e_1 is denoted (***Gre, She***) or (***She, Gre***), and edge e_{10} is denoted (***Cas, Dou***) or (***Dou, Cas***). Edge e_4 is incident on ***Wor*** and ***Buf***, and the vertices ***Wor*** and ***Buf*** are adjacent.

Example 8.1.3 A directed graph is shown in Figure 8.1.4. The directed

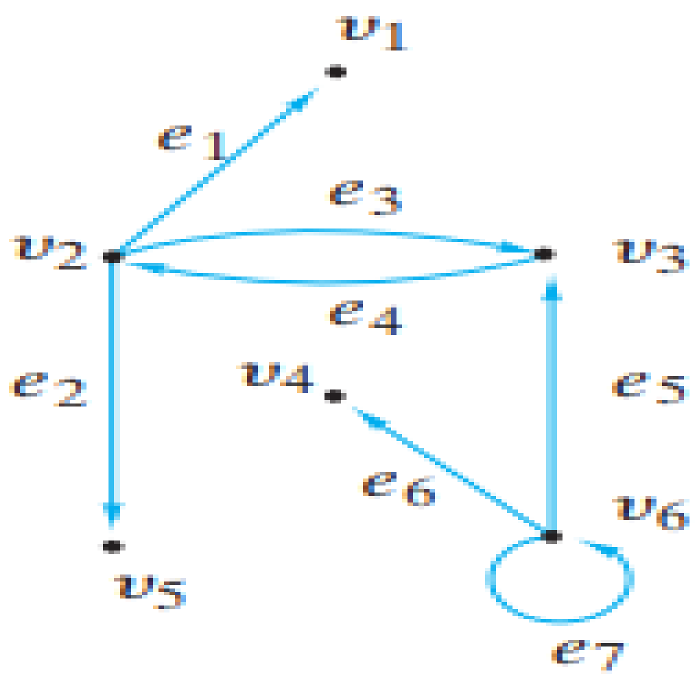


Figure 8.1.4 A directed graph.

edges are indicated by arrows. Edge $\boldsymbol{e_1}$ is associated with the ordered pair $(\boldsymbol{v_2}, \boldsymbol{v_1})$ of vertices, and edge $\boldsymbol{e_7}$ is associated with the ordered pair $(\boldsymbol{v_6}, \boldsymbol{v_6})$ of vertices. Edge $\boldsymbol{e_1}$ is denoted $(\boldsymbol{v_2}, \boldsymbol{v_1})$, and edge $\boldsymbol{e_7}$ is denoted $(\boldsymbol{v_6}, \boldsymbol{v_6})$.

Definition 8.1.1 allows distinct edges to be associated with the same pair of vertices. For example, in Figure 8.1.5, edges $\boldsymbol{e_1}$ and $\boldsymbol{e_2}$ are both associated with the vertex pair $\{\boldsymbol{v_1}, \boldsymbol{v_2}\}$. Such edges are called **parallel edges.** An edge incident on a single vertex is called a **loop.** For example, in Figure 8.1.5, edge $e_3 = (v_2, v_2)$ is a loop. A vertex, such as vertex v_4 in Figure 8.1.5, that is not incident on any edge is called an **isolated vertex.** A graph with neither loops nor parallel edges is called a **simple graph**.

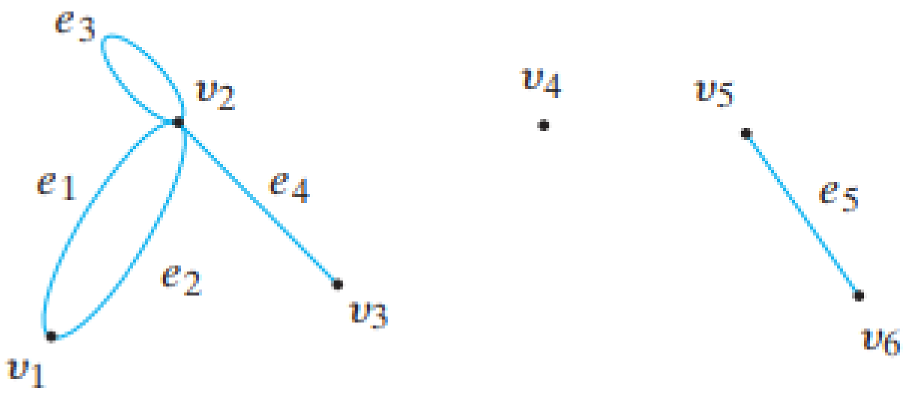


Figure 8.1.5 A graph with parallel edges and loops.

Example 8.1.4 Since the graph of Figure 8.1.2 has neither parallel edges nor loops, it is a **simple graph**.

We turn next to an example that shows how a graph model can be used to analyze a manufacturing problem.

Example 8.1.5

Frequently in manufacturing, it is necessary to bore many holes in sheets of metal (see Figure 8.1.6). Components can then be bolted to these sheets of metal. The holes can be drilled using a drill press under the control of a computer. To save time and money, the drill press should be moved as quickly as possible. Model the situation as a graph.

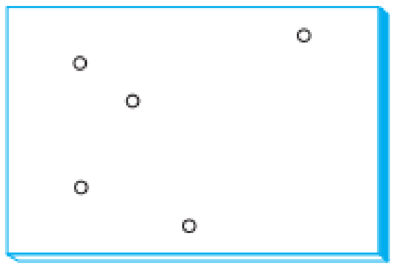


Figure 8.1.6 A sheet of metal with holes for bolts.

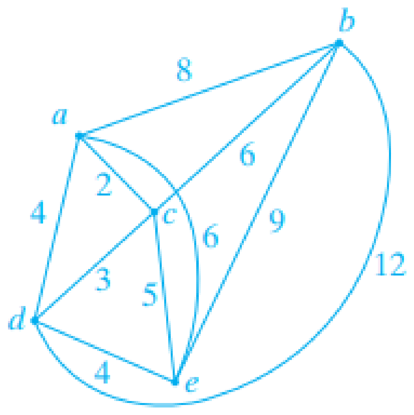


Figure 8.1.7 A graph model of sheet metal in Figure 8.1.6. The edge weight is the time to move the drill press.

***SOLUTION*:** The vertices of the graph correspond to the holes (see Figure

8.1.7). Every pair of vertices is connected by an edge. We write on each edge the time to move the drill press between the corresponding holes. A graph with numbers on the edges (such as the graph of Figure 8.1.7) is called a **weighted graph**. If edge ***e*** is labeled ***k***, we say that the **weight of edge *e*** is ***k***. For example, in Figure 8.1.7 the weight of edge ***(c, e)*** is 5. In a weighted graph, the **length of a path** is the sum of the weights of the edges in the path. For example, in Figure 8.1.7 the length of the path that starts at a, visits c, and terminates at b is 8.

In this problem, the length of a path that starts at vertex v_1 and then visits v_2, v_3, . . . , in this order, and terminates at v_n represents the time it takes the drill press to start at hole h_1 and then move to h_2, h_3, . . . , in this order, and terminate at h_n, where hole h_i corresponds to vertex v_i. A path of minimum length that visits every vertex exactly one time represents the optimal path for the drill press to follow. Suppose that in this problem the path is required to begin at vertex $\boldsymbol{a}$ and end at vertex ***e***. We can find the minimum-length path by listing all possible paths from *a* to *e* that pass through every vertex exactly one time and choose the shortest one (see Table 8.1.1). We see that the path that visits the vertices *a*, *b*, *c*, *d*, *e*, in this order, has minimum length. Of course, a different pair of starting and ending vertices might produce an even shorter path.

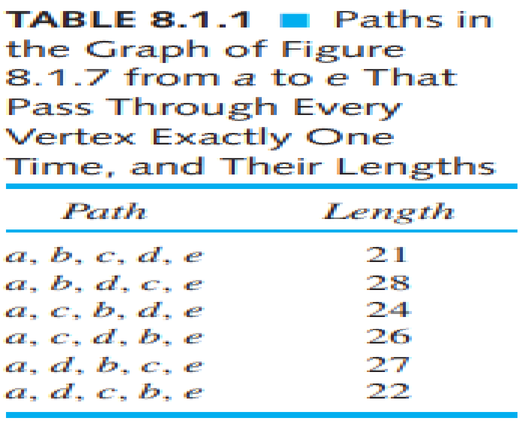


TABLE 8.1.1 ■ Paths in the Graph of Figure 8.1.7 from *a* to *e* That Pass Through Every Vertex Exactly One Time, and Their Lengths

Path	*Length*
a, *b*, *c*, *d*, *e*	21
a, *b*, *d*, *c*, *e*	28
a, *c*, *b*, *d*, *e*	24
a, *c*, *d*, *b*, *e*	26
a, *d*, *b*, *c*, *e*	27
a, *d*, *c*, *b*, *e*	22

Listing all paths from vertex *v* to vertex *w*, as we did in Example 8.1.5, is a rather time-consuming way to find a minimum-length path from *v* to *w* that visits every vertex exactly one time. Unfortunately, no one knows a method that is much more practical for arbitrary graphs. This problem is a version of the **traveling salesperson problem.** We discuss that problem in Section 8.3.

Example 8.1.6 Bacon Numbers

Actor Kevin Bacon has appeared in numerous films including *Diner* and *Apollo 13*. Actors who have appeared in a film with Bacon are said to have *Bacon number one*. For example, Ellen Barkin has Bacon number one because she appeared with Bacon in *Diner*. Actors who did not appear in a film with Bacon but who appeared in a film with an actor whose Bacon number is one are said to have *Bacon number two*. Higher Bacon numbers are defined similarly. For example, Bela Lugosi has Bacon number three. Lugosi was in *Black Friday* with Emmett Vogan, Vogan was in *With a Song in My Heart* with Robert Wagner, and Wagner was in *Wild Things* with Bacon. Develop a graph model for Bacon numbers.

***SOLUTION*:** We let vertices denote actors, and we place one edge between two distinct actors if they appeared in at least one film together. In an unweighted graph, the length of a path is the number of edges in the path. Thus an actor's Bacon number is the length of a shortest path from the vertex corresponding to that actor to the vertex corresponding to Bacon. In Section 8.4, we discuss the general problem of finding shortest paths in graphs. Unlike the situation in

Example 8.1.5, there are efficient algorithms for finding shortest paths.

Example 8.1.7 Similarity Graphs

This example deals with the problem of grouping "like" objects into classes based on properties of the objects. For example, suppose that a particular algorithm is implemented in C++ by a number of persons and we want to group "like" programs into classes based on certain properties of the programs (see Table 8.1.2). Suppose that we select as properties

1. The number of lines in the program

2. The number of return statements in the program

3. The number of functions calls in the program

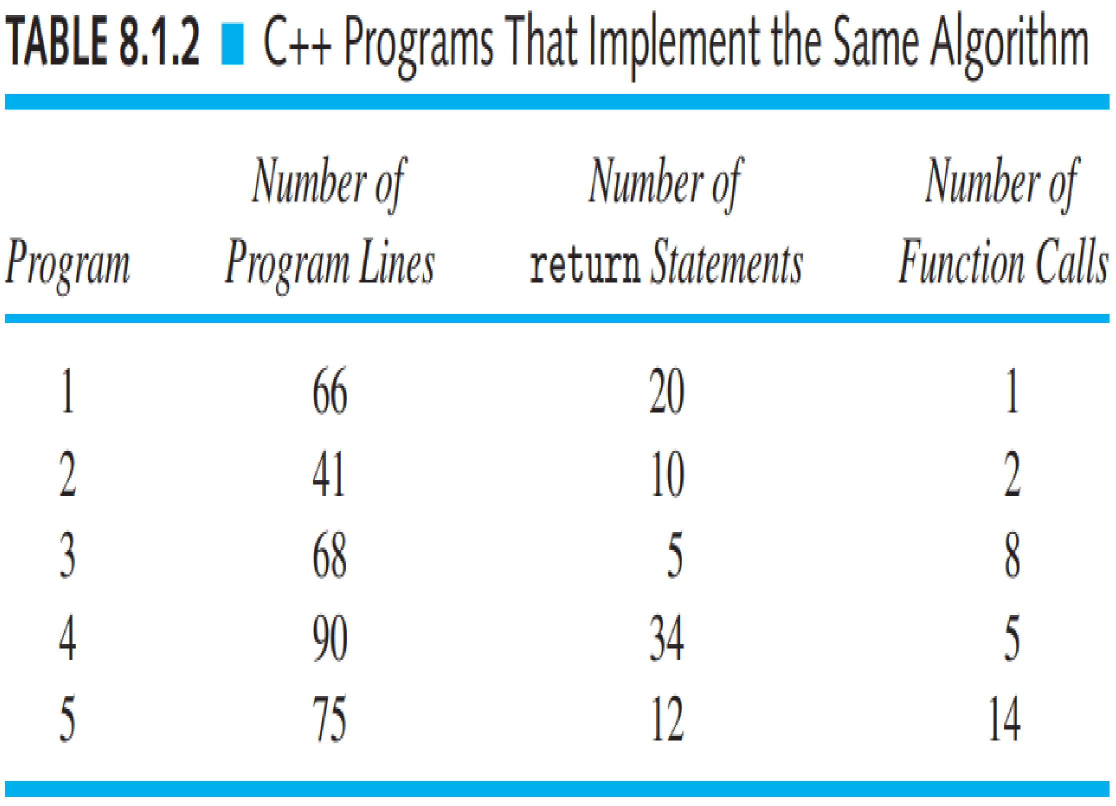


TABLE 8.1.2 ▪ C++ Programs That Implement the Same Algorithm

Program	*Number of Program Lines*	*Number of* `return` *Statements*	*Number of Function Calls*
1	66	20	1
2	41	10	2
3	68	5	8
4	90	34	5
5	75	12	14

A **similarity graph** *G* is constructed as follows. The vertices correspond to programs. A vertex is denoted (p_1, p_2, p_3), where p_i is the value of property i. We define a **dissimilarity function *s*** as follows. For each pair of vertices $v=(p_1,p_2, p_3)$ and $w = (q_1, q_2, q_3)$, we set $\boldsymbol{s(v,w)= |p_1- q_1|+|p_2- q_2| + |p_3-q_3|}$**.**

If we let v_i be the vertex corresponding to program i, we obtain

***s(v*1*, v*2*)* = 36, *s(v*1*, v*3*)* = 24, *s(v*1*, v*4*)* = 42, *s(v*1*, v*5*)* = 30, *s(v*2*, v*3*)* = 38, *s(v*2*, v*4*)* = 76, *s(v*2*, v*5*)* = 48, *s(v*3*, v*4*)* = 54, *s(v*3*, v*5*)* = 20, *s(v*4*, v*5*)* = 46.**

If *v* and *w* are vertices corresponding to two programs, *s(v,w)* is a measure of how dissimilar the programs are. A large value of *s(v,w)* indicates dissimilarity, while a small value indicates similarity. For a fixed number *S*, we insert an edge between vertices *v* and *w* if *s(v,w)*< *S*. (In general, there will be different similarity graphs for different values of *S*). We say that *v* and *w* are **in the same class** if *v*=*w* or there is a path from *v* to *w*.

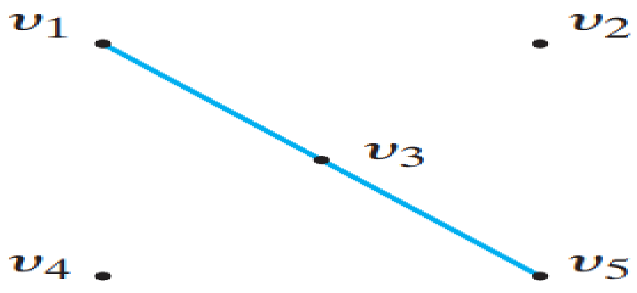


Figure 8.1.9 A similarity graph corresponding to the programs of Table 8.1.2 with $S = 25$.

In Figure 8.1.9 we show the graph corresponding to the programs of Table 8.1.2 with S = 25. In this graph, the programs are grouped into three classes: {1, 3, 5}, {2}, and {4}. In a real problem, an appropriate value for S might be selected by trial and error or the value of S might be selected automatically according to some predetermined criteria.

Example 8.1.8 The *n*-Cube (Hypercube) The traditional computer, often called a **serial computer**, executes one instruction at a time. Our definition of “algorithm” also assumes that one instruction is executed at a time. Such algorithms are called **serial algorithms.** As hard as hardware costs have declined, it has become feasible to build **parallel computers** with many processors that are capable of executing several instructions at a time. Graphs are often convenient models to describe these machines. The associated algorithms are known as **parallel algorithms.** Many problems can be solved much faster using parallel computers rather than serial computers. We discuss one model for parallel computation known as the ***n*-cube** or **hypercube.**

The n-cube has 2^n processors, $n \geq 1$, which are represented by vertices (see Figure 8.1.10) labeled 0, 1, . . . , $2^n - 1$.

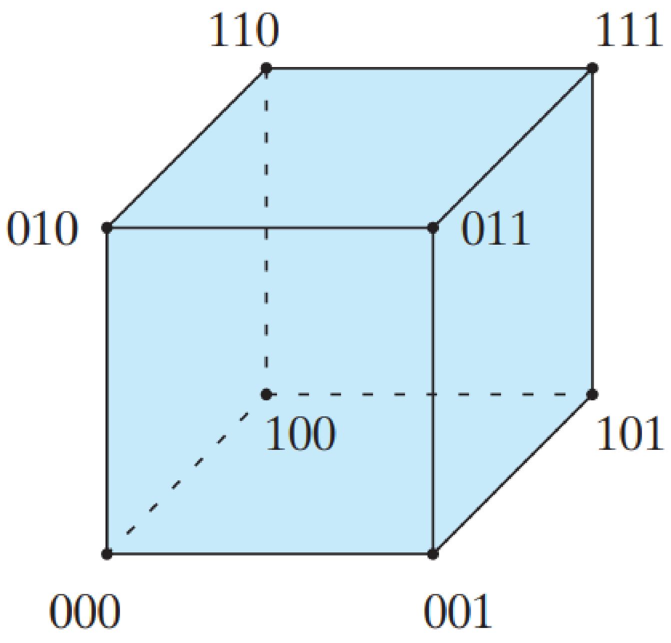


Figure 8.1.10 The 3-cube.

Each processor has its own local memory. An edge connects two vertices if the binary representation of their labels differs by exactly one bit. During one time unit, all processors in the *n*-cube may execute an instruction simultaneously and then communicate with an adjacent processor. If a processor needs to communicate with a nonadjacent processor, the first processor sends a message that includes the route to, and ultimate destination of, the recipient. It may take several time units for a processor to communicate with a nonadjacent processor.

The n-cube is an important model of computation because several such machines have been built and are running. Furthermore, several other parallel computation models can be simulated by the hypercube. The latter point is considered in more detail in Examples 8.3.5 and 8.6.3

Definition 8.1.9 The *complete graph on n vertices*, denoted K_n , is the simple graph with *n* vertices in which there is an edge between every pair of distinct vertices.

Example 8.1.10 The complete graph on four vertices, K_4, is shown in Figure 8.1.12.

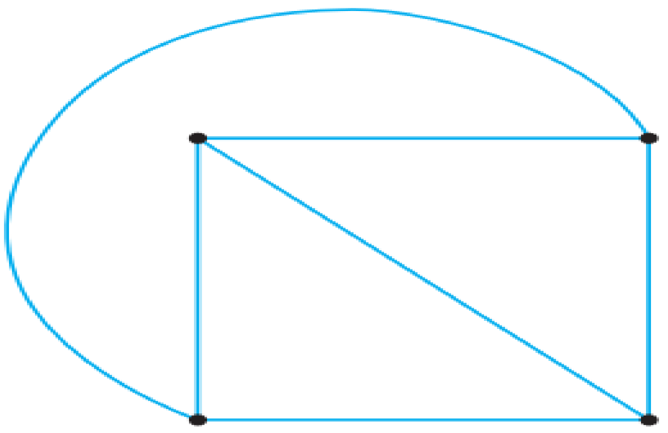


Figure 8.1.12 The complete graph K_4.

Definition 8.1.11 A graph $G = (V, E)$ is *bipartite* if there exist subsets V_1 and V_2 (either possibly empty) of V such that

$$V_1 \cap V_2 = \emptyset,\ V_1 \cup V_2 = V,$$

And each V_1 edge in E is incident on one vertex in $V1$ and one vertex in V_2.

Example 8.1.12 The graph in Figure 8.1.13 is bipartite since if we let

$V_1 = \{v_1, v_2, v_3\}$ and $V_2 = \{v_4, v_5\}$,

each edge is incident on one vertex in V_1 and one vertex in V_2.

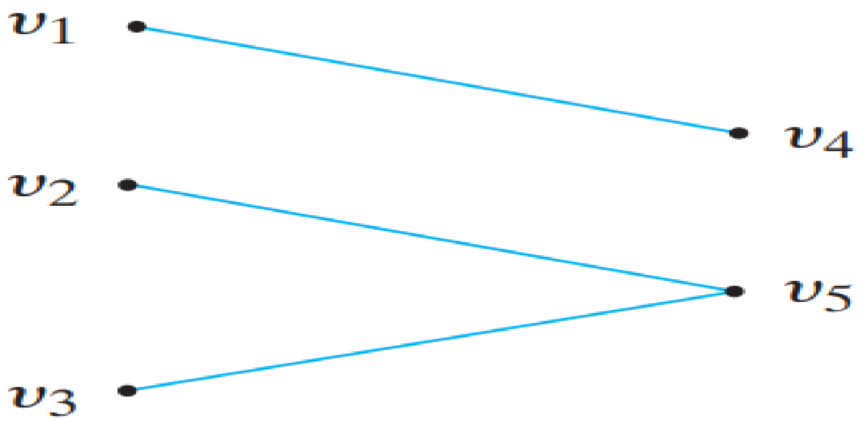


Figure 8.1.13 A bipartite graph.

Notice that Definition 8.1.11 states that if e is an edge in a bipartite graph, then e

is incident on one vertex in V_1 and one vertex in V_2. It does *not* state that if v_1 is a vertex in V_1 and v_2 is a vertex in V_2, then there is an edge between v_1 and v_2.

Example 8.1.13 The graph in Figure 8.1.14 is not bipartite. It is often easiest to prove that a graph is not bipartite by arguing by contradiction.

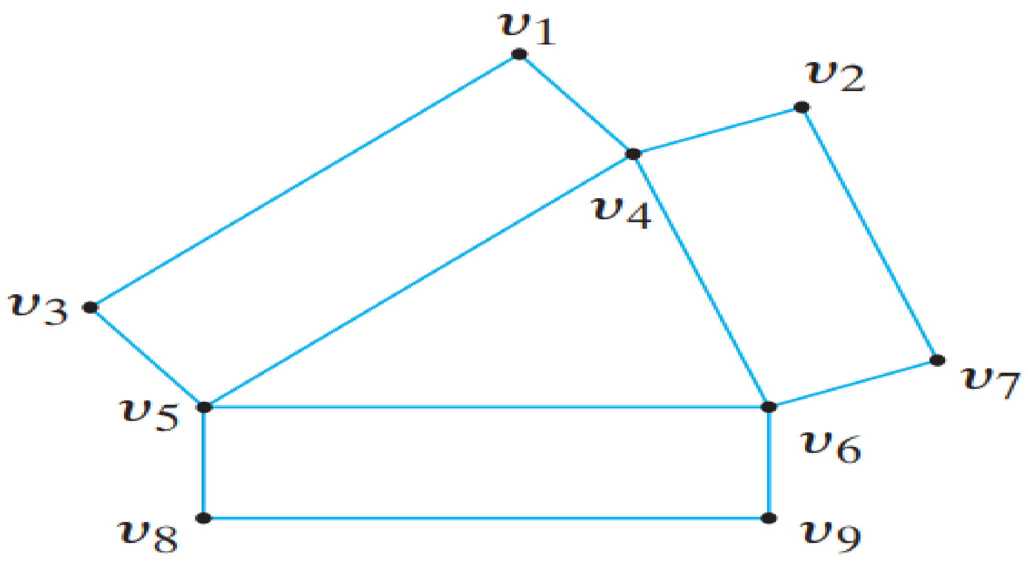


Figure 8.1.14 A graph that is not bipartite.

Suppose that the graph in Figure 8.1.14 is bipartite. Then the vertex set can be partitioned into two subsets V_1 and V_2 such that each edge is incident on one vertex in V_1 and one vertex in V_2. Now consider the vertices v_4, v_5, and v_6. Since v_4 and v_5 are adjacent, one is in V_1 and the other in V_2. We may assume that v_4 is in V_1 and that v_5 is in V_2. Since v_5 and v_6 are adjacent and v_5 is in V_2, v_6 is in V_1. Since v_4 and v_6 are adjacent and $v_4$is in V_1, v_6 is in V_2. But now v_6 is in both

V_1, and V_2, which is a contradiction since V_1 and V_2 are disjoint. Therefore, the graph in Figure 8.1.14 is not bipartite.

Example 8.1.14 The complete graph K_1 on one vertex is bipartite. We may let V_1 be the set containing the one vertex and V_2 be the empty set. Then each edge (namely none!) is incident on one vertex in V_1 and one vertex in V_2.

Definition 8.1.15 The *complete bipartite graph on m and n vertices*, denoted $K_{m,n}$ is the simple graph whose vertex set is partitioned into sets V_1 with

m vertices and V_2 with n vertices in which the edge set consists of all edges of the form (v_1, v_2) with $v_1 \in V_1$ and $v_2 \in V_2$.

Example 8.1.16 The complete bipartite graph on two and four vertices, $K_{2,4}$, is shown in Figure 8.1.15.

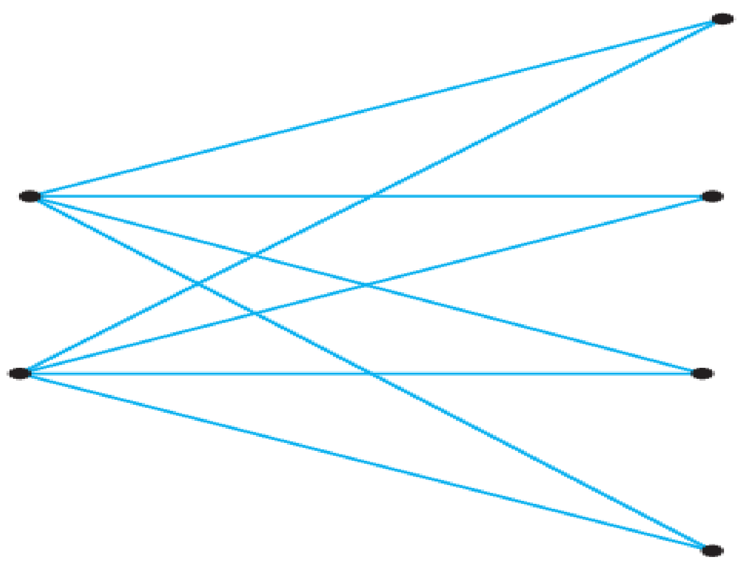


Figure 8.1.15 The complete bipartite graph $K_{2,4}$.

8.2 Paths and Cycles

If we think of the vertices in a graph as cities and the edges as roads, a path corresponds to a trip beginning at some city, passing through several cities, and terminating at some city. We begin by giving a formal definition of path.

Definition 8.2.1

Let v_0 and v_n be vertices in a graph. A ***path*** from v_0 to v_n of length n is an alternating sequence of $n + 1$ vertices and n edges beginning with vertex v_0 and ending with vertex v_n,

$$(v_0, e_1, v_1, e_2, v_2, \ldots, v_{n-1}, e_n, v_n),$$

in which edge e_i is incident on vertices v_{i-1} and v_i for *i* = 1, . . . , *n*.

The formalism in Definition 8.2.1 means: Start at vertex v_0; go along edge e_1 to v_1; go along edge e_2 to ; and so on.

Example 8.2.2

In the graph of Figure 8.2.1,

$(1, e_1, 2, e_2, 3, e_3, 4, e_4, 2)$ **(8.2.1)**

is a path of length 3 from vertex 1 to vertex 4.

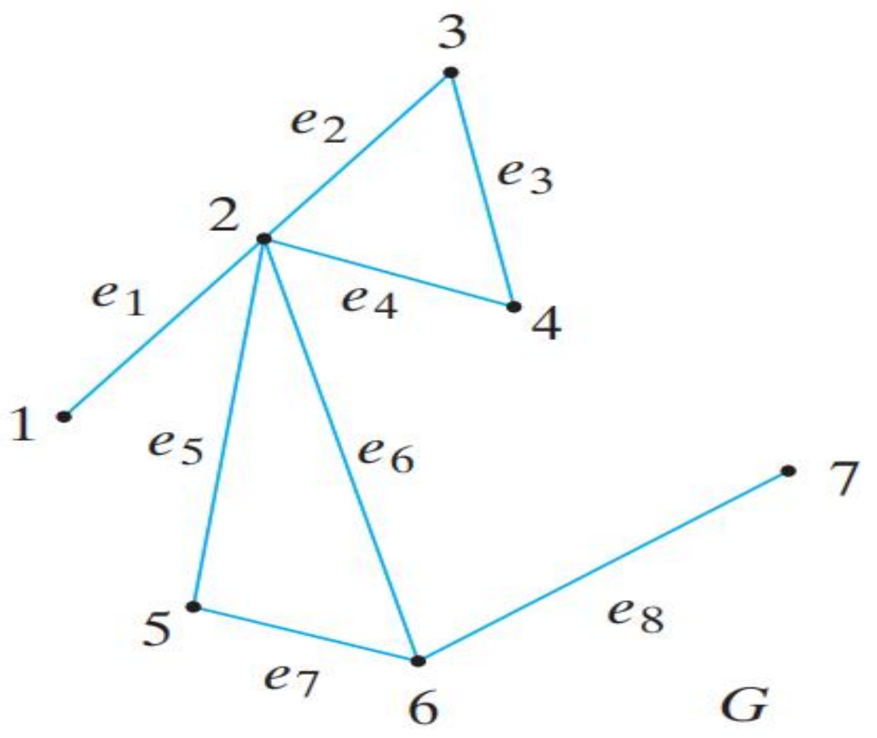


Figure 8.2.1 A connected graph with paths $(1, e_1, 2, e_2, 3, e_3, 4, e_4, 2)$ of length 4 and (6) of length 0.

Example 8.2.3 In the graph of Figure 8.2.1, the path (6) consisting solely of vertex 6 is a path of length 0 from vertex 6 to vertex 6. In the absence of parallel edges, in denoting a path we may suppress the edges. For example, the path (8.2.1) may also be written (1, 2, 3, 4).

A **connected graph** is a graph in which we can get from any vertex to any other vertex on a path. The formal definition follows.

Definition 8.2.4 A graph G is *connected* if given any vertices v and w in G, there is a path from v to w.

Example 8.2.5 The graph G of Figure 8.2.1 is connected since, given any vertices v and w in G, there is a path from v to w.

Example 8.2.6 The graph *G* of Figure 8.2.2 is not connected since, for example, there is no path from vertex v_2 to vertex v_5.

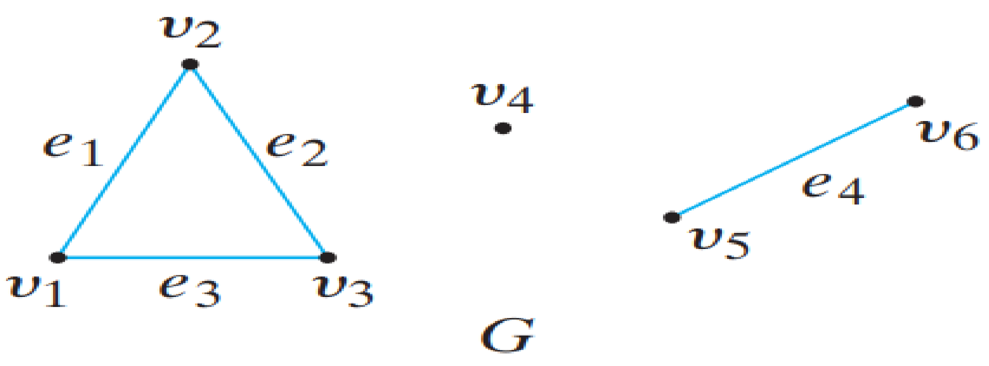


Figure 8.2.2 A graph that is not connected.

Example 8.2.7 Let *G* be the graph whose vertex set consists of the 50 states of the United States. Put an edge between states *v* and *w* if *v* and *w* share a border. For example, there is an edge between California and Oregon and between Illinois and Missouri. There is no edge between Georgia and New York, nor is there an edge between Utah and New Mexico.

(Touching does not count; the states must share a border.) The graph *G* is not connected because there is no path from Hawaii to California (or from Hawaii to any other state).

As we can see from Figures 8.2.1 and 8.2.2, a connected graph consists of one "piece," while a graph that is not connected consists of two or more “pieces.” These “pieces” are **subgraphs** of the original graph and are called **components**. We give the formal definitions beginning with subgraph.

A subgraph G' of a graph G is obtained by selecting certain edges and vertices from G subject to the restriction that if we select an edge e in G that is incident on vertices v and w, we must include v and w in G'. The restriction is to ensure that G' is actually a graph. The formal definition follows.

Definition 8.2.8 Let $G = (V, E)$ be a graph. We call (V' , E') a *subgraph* of G if

(a) ($V' \subseteq V$ and $E' \subseteq E$).

(b) For every edge $e' \in E'$, if e' is incident on v' and w', then $v', w' \in V'$.

Example 8.2.9 The graph $G' = (V', E')$ of Figure 8.2.3 is a subgraph of the graph $G = (V, E)$ of Figure 8.2.4 since $V' \subseteq V$ and $E' \subseteq E$.

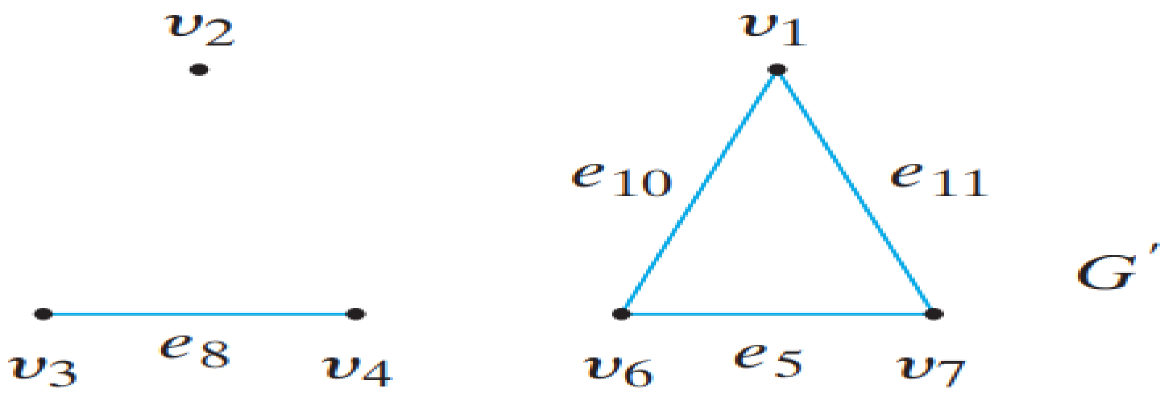


Figure 8.2.3 A subgraph of the graph of Figure 8.2.4.

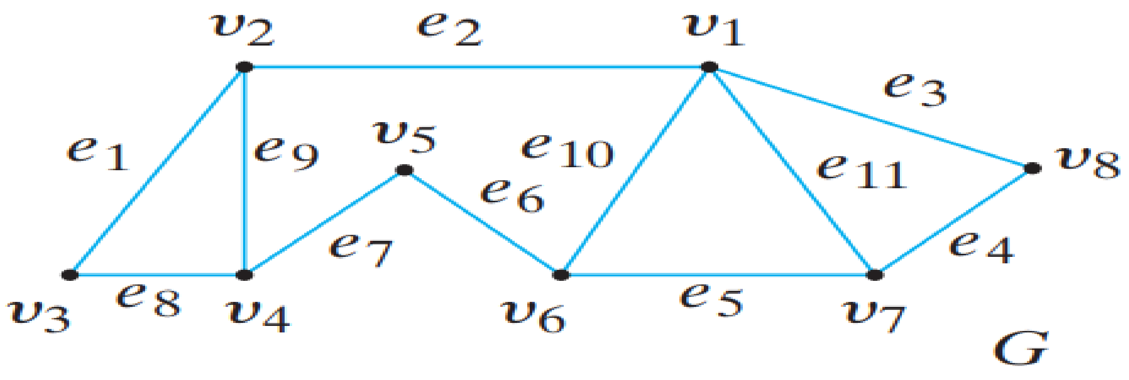


Figure 8.2.4 A graph, one of whose subgraphs is shown in Figure 8.2.3.

Example 8.2.10 Find all subgraphs of the graph G of Figure 8.2.5 having at least one vertex.

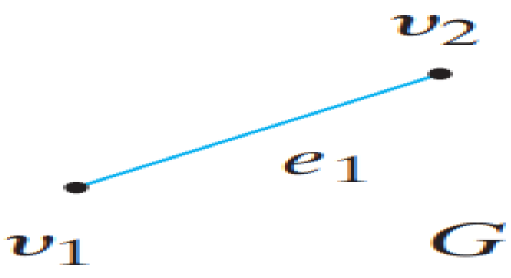


Figure 8.2.5 The graph for Example 8.2.10.

SOLUTION If we select no edges, we may select one or both vertices yielding the subgraphs G_1, G_2, and G_3 shown in Figure 8.2.6.

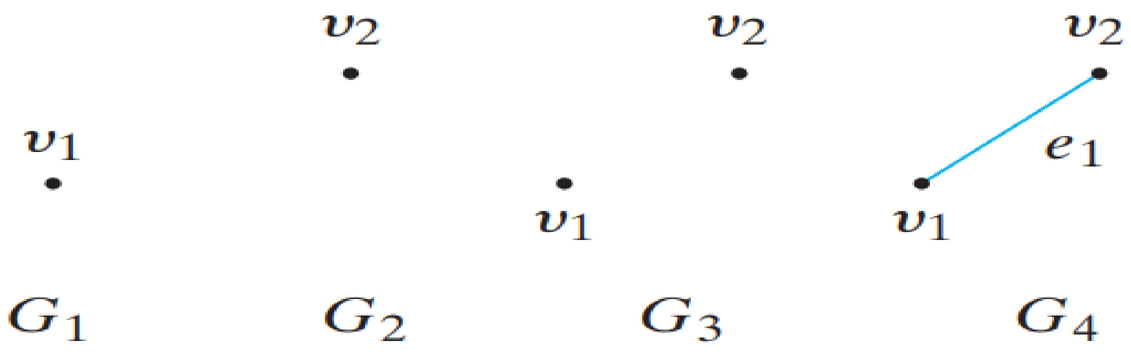


Figure 8.2.6 The four subgraphs of the graph of Figure 8.2.5.

If we select the one available edge e_1, we must select the two vertices on which e_1 is incident. In this case, we obtain the subgraph G_4 shown in Figure 8.2.6.

Thus G has the four subgraphs shown in Figure 8.2.6.

We can now define "component."

Definition 8.2.11

Let G be a graph and let v be a vertex in G. The subgraph G' of G consisting of all edges and vertices in G that are contained in some path beginning at v is called the ***component*** of G containing v.

Example 8.2.12 The graph G of Figure 8.2.1 has one component, namely itself.

Indeed, a graph is connected if and only if it has exactly one component.

Example 8.2.13 Let G be the graph of Figure 8.2.2. The component of G containing v_3 is the subgraph

$G_1 = (V_1, E_1)$, $V_1 = \{v_1, v_2, v_3\}$, $E_1 = \{e_1, e_2, e_3\}$.

The component of G containing v_4 is the subgraph

$G_2 = (V_2, E_2)$, $V_2 = \{v_4\}$, $E_2 = \emptyset$.

The component of G containing v_5 is the subgraph

$G_3 = (V_3, E_3)$, $V_3 = \{v_5, v_6\}$, $E_3 = \{e_4\}$.

Another characterization of the components of a graph $G = (V, E)$ is obtained by defining a relation R on the set of vertices V by the rule $v_1 R v_2$ if there is a path from v_1 to v_2.

It can be shown (Exercise 69) that R is an equivalence relation on V and that if $v \in V$, the set of vertices in the component containing v is the equivalence class

$$[v] = \{w \in V \mid wRv\}.$$

Notice that the definition of "path" allows repetitions of vertices or edges or both.

In the path (8.2.1), vertex 2 appears twice.

Subclasses of paths are obtained by prohibiting duplicate vertices or edges or by making the vertices v_0 and v_n of Definition 8.2.1 identical.

Definition 8.2.14 Let *v* and *w* be vertices in a graph *G*.

A ***simple path*** from *v* to *w* is a path from *v* to *w* with no repeated vertices.

A *cycle* (or *circuit*) is a path of nonzero length from *v* to *v* with no repeated edges.

A ***simple cycle*** is a cycle from *v* to *v* in which, except for the beginning and ending vertices that are both equal to *v*, there are no repeated vertices.

Example 8.2.15 For the graph of Figure 8.2.1, we have the following information.

Path	*Simple Path?*	*Cycle?*	*Simple Cycle?*
(6, 5, 2, 4, 3, 2, 1)	No	No	No
(6, 5, 2, 4)	Yes	No	No
(2, 6, 5, 2, 4, 3, 2)	No	Yes	No
(5, 6, 2, 5)	No	Yes	Yes
(7)	Yes	No	No

We next reexamine the problem introduced in Section 8.1 of finding a cycle in a graph that traverses each edge exactly one time.

Example 8.2.16

Kö nigsberg Bridge Problem The first paper in graph theory was Leonhard Euler’s in 1736. The paper presented a general theory that included a solution to what is now called the Kö nigsberg bridge problem.

Two islands lying in the Pregel River in K¨onigsberg (now Kaliningrad in

Russia) were connected to each other and the river banks by bridges, as shown in Figure 8.2.7.

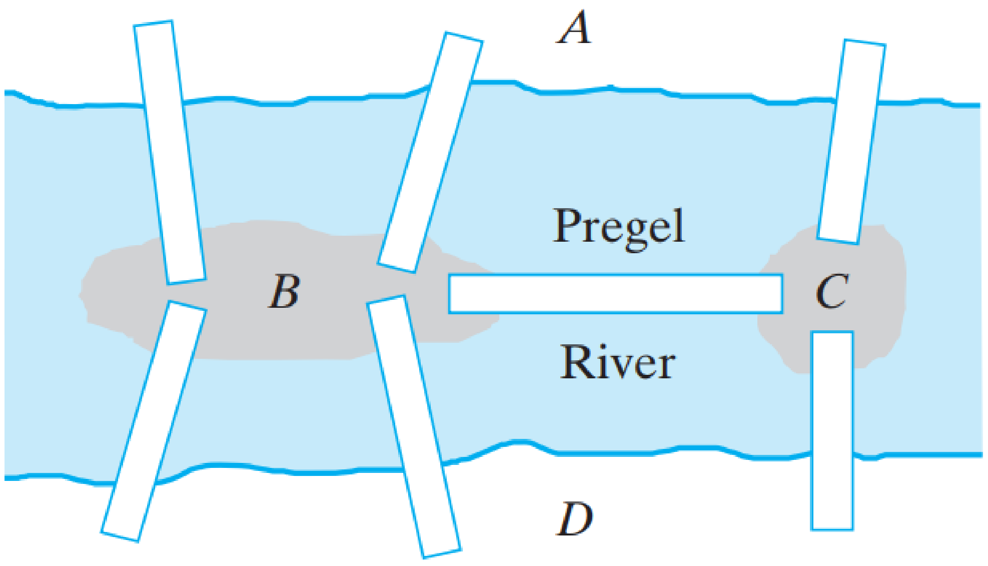


Figure 8.2.7 The bridges of Königsberg.

The problem is to start at any location– *A*, *B*, *C*, or *D*; walk over each bridge exactly once; then return to the starting location.

The bridge configuration can be modeled as a graph, as shown in Figure 8.2.8.

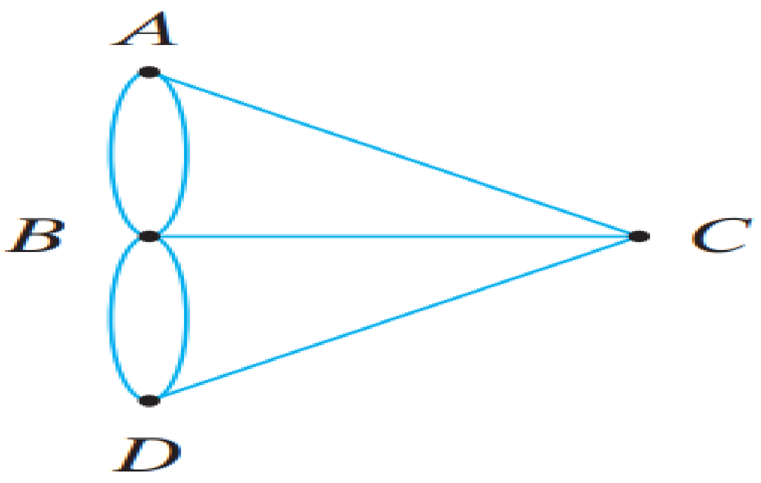


Figure 8.2.8 A graph model of the bridges of Königsberg.

The vertices represent the locations and the edges represent the bridges.

The Königsberg bridge problem is now reduced to finding a cycle in the graph of Figure 8.2.8 that includes all of the edges and all of the vertices. In honor of Euler, a cycle in a graph *G* that includes all of the edges and all of the vertices of *G* is called an **Euler cycle.**† From the discussion of Section 8.1, we see that there is no Euler cycle in the graph of Figure 8.2.8 because the number of edges incident on vertex *A* is odd. (In fact, in the graph of Figure 8.2.8, every vertex is incident on an odd number of edges.

The solution to the existence of Euler cycles is nicely stated by introducing the degree of a vertex. The **degree of a**

vertex *v*, δ(v), is the number of edges incident on *v*. (By definition, each loop on *v* contributes 2 to the degree of *v*.) In Section 8.1 we found that if a graph *G* has an Euler cycle, then every vertex in *G* has even degree. We can also prove that *G* is connected.

Theorem 8.2.17 If a graph *G* has an Euler cycle, then *G* is connected and every vertex has even degree.

Proof:

Suppose that *G* has an Euler cycle. We argued in Section 8.1 that every vertex in *G* has even degree. If *v* and *w* are vertices in G, the portion of the Euler cycle that takes us from v to w serves as a path from v to w. Therefore, G is connected. ■

The converse of Theorem 8.2.17 is also true. We give a proof by mathematical induction due to [Fowler].

Theorem 8.2.18 If G is a connected graph and every vertex has an even degree, then G has an Euler cycle.

Proof:

The proof is by induction on the number n of edges in G.

Basis Step (n =0)

Since G is connected, if G has no edges, G consists of a single vertex. A Euler cycle consists of the single vertex and no edges.

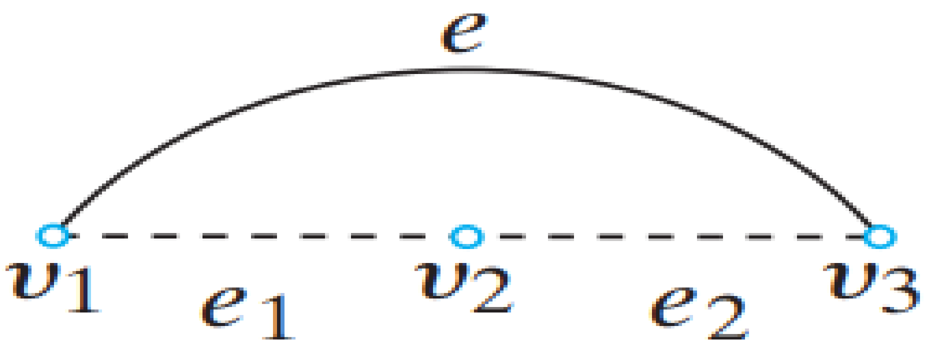
Inductive Step

Suppose that G has n edges, $n > 0$, and that any connected graph with k edges, $k< n$, in which every vertex has even degree, has an Euler cycle.

It is straightforward to verify that a connected graph with one or two vertices, each of which has even degree, has an Euler cycle (see Exercise 70);

thus we assume that the graph has at least three vertices.

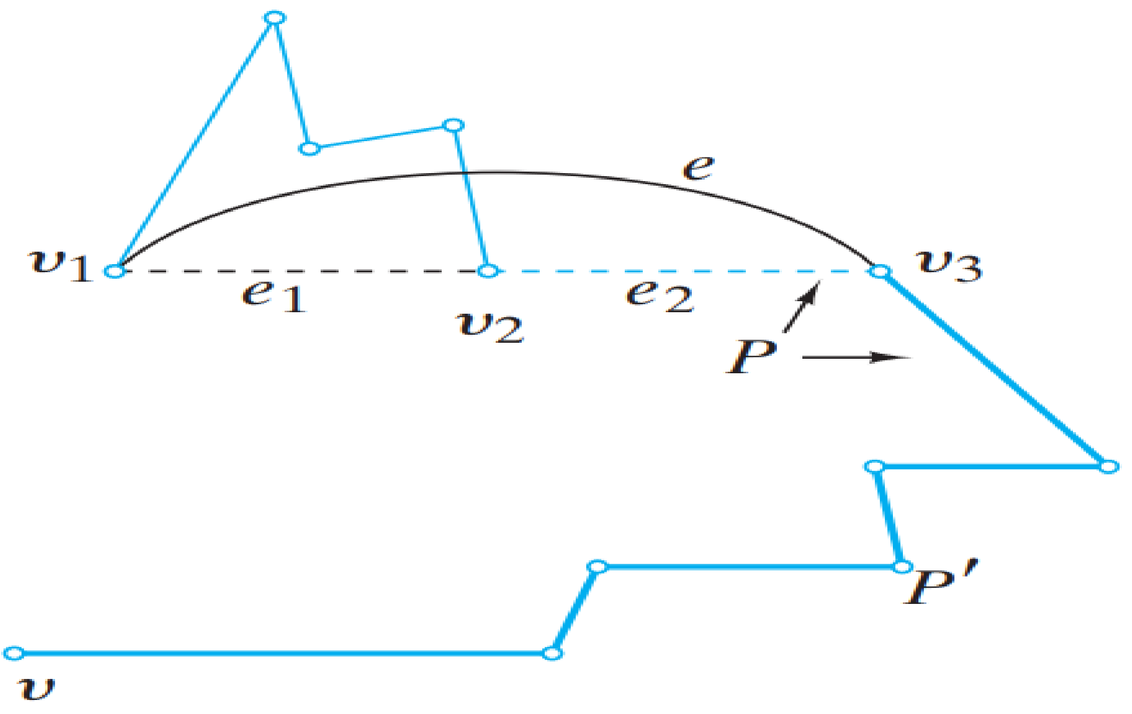
Since G is connected, there are vertices v_1, v_2, and v_3 in G with edge e_1 incident on v_1 and v_2 and edge e_2 incident on v_2 and v_3.We delete edges e_1 and e_2, but no vertices, and add an edge e incident on v_1 and v_3 to obtain the graph G' [see Figure 8.2.9(a)].



(a)

Notice that each component of the graph G' has less than n edges and that in each component of the graph G', every vertex has even degree. We show that G' has either one or two components.

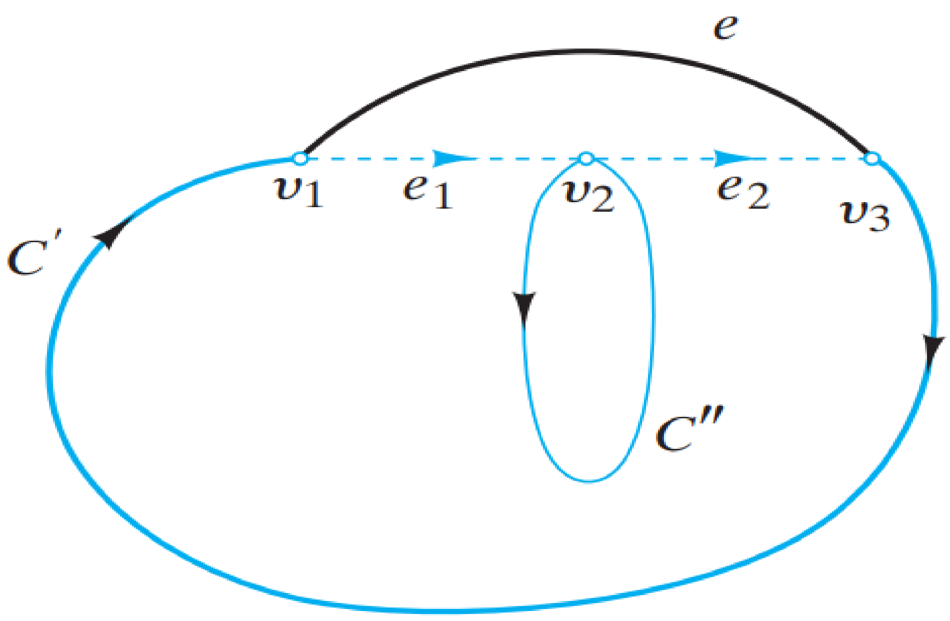
Let v be a vertex. Since G is connected, there is a path P in G from v to v_1. Let P' be the portion of the path P starting at v whose edges are also in G'. Now P' ends at either v_1, v_2, or v_3 because the only way that P could fail to be a path in G' is that P contains one of the deleted edges e_1 or e_2. If P' ends at v_1, then v is in the same component as v_1 in G'. If P' ends at v_3 [see Figure 8.2.9(b)],



(b)

then v is in the same component as v_3 in G', which is in the same component as $v_1$in G' (since edge e in G' is incident on v_1 and v_3). If P' ends at v_2, then v_2 is in the same component as v. Therefore, any vertex in G' is in the same component as either v_1 or v_2. Thus G' has one or two components.

If G' has one component, that is, if G' is connected, we may apply the inductive hypothesis to conclude that G' has an Euler cycle C'. This Euler cycle may be modified to produce a Euler cycle in *G*: We simply replace the occurrence of edge e in C' by edges e_1 and e_2. Suppose that G' has two components [see Figure 8.2.9(c)].



(c)

By the inductive hypothesis, the component containing v_1 has an Euler cycle C' and the component containing v_2 has an Euler cycle C'' beginning and ending at v_2. An Euler cycle in G is obtained by modifying C' by replacing (v_1, v_3) in C' by (v_1, v_2) followed by C'' followed by (v_2, v_3) or by replacing

$v_3, v_1)$ in C' by (v_3, v_2) followed by C'' followed by (v_2, v_1).

The Inductive Step is complete; *G* has a Euler cycle. ■

If *G* is a connected graph and every vertex has even degree and *G* has only a few edges, we can usually find a Euler cycle by inspection.

Example 8.2.19 Let *G* be the graph of Figure 8.2.10. Use Theorem 8.2.18 to verify that *G* has a Euler cycle. Find a Euler cycle for *G*.

SOLUTION We observe that G is connected and that

$\delta(v_1) = \delta(v_2) = \delta(v_3) = \delta(v_5) = 4, \delta(v_4) = 6, \delta(v_6) = \delta(v_7) = 2.$

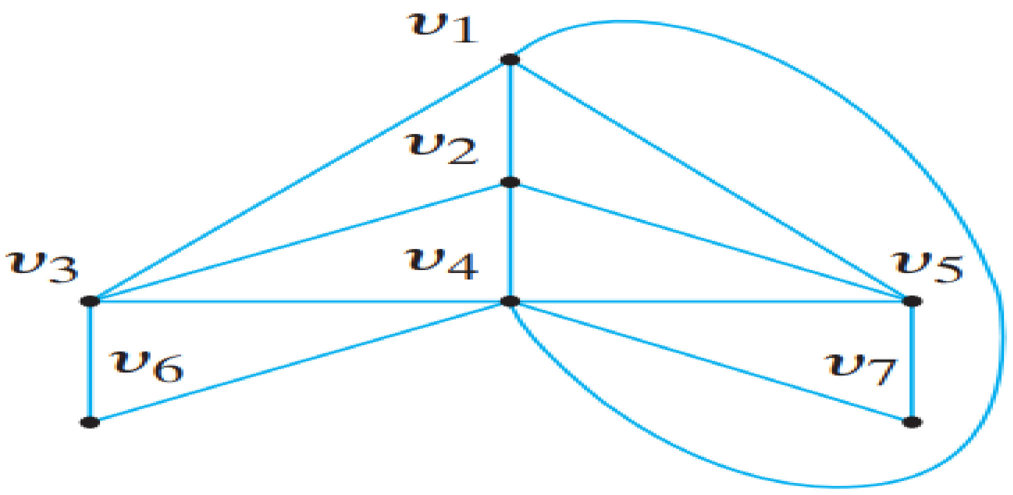


Figure 8.2.10 The graph for Example 8.2.19.

Since the degree of every vertex is even, by Theorem 8.2.18, G has an Euler cycle. By inspection, we find the Euler cycle

$(v_6, v_4, v_7, v_5, v_1, v_3, v_4, v_1, v_2, v_5, v_4, v_2, v_3, v_6)$

■

Example 8.2.20 A domino is a rectangle divided into two squares with each square numbered one of 0, 1, . . . , 6 (see Figure 8.2.11).

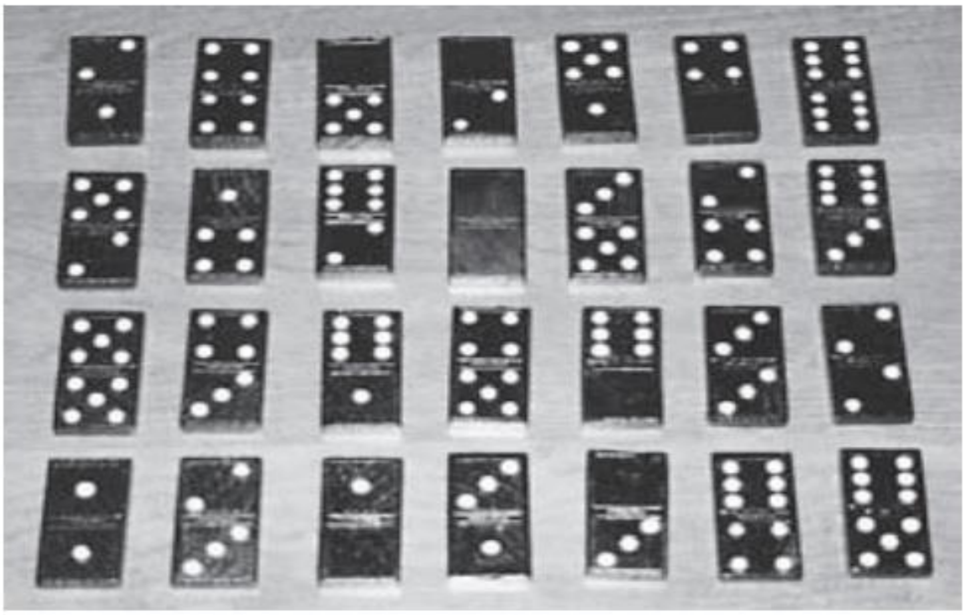


Figure 8.2.11 Dominoes. [*Photo by the author.*]

Two squares on a single domino can have the same number.

Show that the distinct dominoes can be arranged in a circle so that touching dominoes have adjacent squares with identical numbers.

SOLUTION **:** We model the situation as a graph G with seven vertices labeled 0, 1, . . . , 6.

The edges represent the dominoes: There is one edge between each distinct pair of vertices and there is one loop at each vertex. Notice that G is connected. Now the dominoes can be arranged in a circle so that touching dominoes have adjacent squares with identical numbers if and only if G contains a Euler cycle. Since the degree of each vertex is 8 (remember that a loop contributes 2 to the degree), each vertex has even degree. By Theorem 8.2.18, G has a Euler cycle. Therefore, the dominoes can be arranged in a circle so that touching dominoes have adjacent squares with identical numbers.

■

What can be said about a connected graph in which not all the vertices have even degree? The first observation (Corollary 8.2.22) is that the number of vertices of odd degree is even. This follows from the fact (Theorem 8.2.21) that the sum of all the degrees in a graph is an even number.

Theorem 8.2.21 If *G* is a graph with *m* edges and vertices $\{v_1, v_2, . . . , v_n\}$, then

$$\sum_{i=1}^{n} \delta(vi) = 2\mathrm{m}$$

In particular, the sum of the degrees of all the vertices in a graph is even.

Proof

When we sum over the degrees of all the vertices, we count each edge (v_i, v_j) twice - once when we count it as (v_i, v_j) in the degree of v_i and again when we count it as (v_j, v_i) in the degree of v_j The conclusion follows. ■

Corollary 8.2.22 In any graph, the number of vertices of odd degree is even.

Proof Let us divide the vertices into two groups: those with even degree $\boldsymbol{x_1}, \ldots,$ $\boldsymbol{x_m}$ and those with odd degree $y_1, \ldots, y_n$. Let

$$S = \delta(\boldsymbol{x_1}) + \delta(x_2) + \cdots + \delta(\boldsymbol{x_m}),$$

$$T = \delta(y_1) + \delta(y_2) + \cdot \quad \cdot \quad \cdot + \delta(y_n).$$

By Theorem 8.2.21, $S + T$ is even. Since S is the sum of even numbers, S is even. Thus T is even. But T is the sum of n odd numbers, and therefore n is even. ■

Theorem 8.2.23 A graph has a path with no repeated edges from v to w $(v \neq w)$ containing all the edges and vertices if and only if it is connected and v and w are the only vertices having odd degree.

Proof :

Suppose that a graph has a path P with no repeated edges from v to w containing all the edges and vertices. The graph is surely connected. If we add an edge from *v* to *w*, the resulting graph has an Euler cycle, namely, the path *P* together with the added edge. By Theorem 8.2.17, every vertex has even degree. Removing the added edge affects only the degrees of *v* and *w*, which are each reduced by 1. Thus in the original graph, *v* and *w* have odd degree and all other vertices have even degree.

Conversely, suppose that a connected graph *G* has exactly two vertices *v* and *w* of odd degree. Let us temporarily insert an edge e from v to w. The resulting graph G' is connected and every vertex has even degree. By Theorem 8.2.18, G' has an Euler cycle. If we delete e from this Euler cycle, we obtain a path with no repeated edges from v to w containing all the edges and vertices of G. ■

Generalizations of Theorem 8.2.23 are given as Exercises 43 and 45.

We conclude by proving a rather special result that we will use in Section 9.2.

Theorem 8.2.24 If a graph G contains a cycle from v to v, G contains a simple cycle from v to v.

Proof

Let

$C = (v_0, e_1, v_1, \ldots, e_i, v_i, e_{i+1}, \ldots, e_j, v_j, e_{j+1}, v_{j+1}, \ldots, e_n, v_n)$

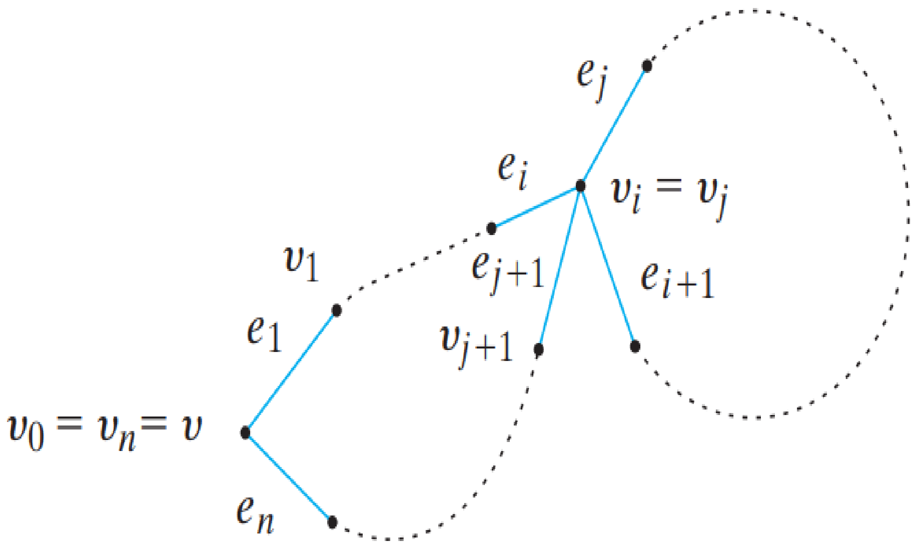


Figure 8.2.12 A cycle that either is a simple cycle or can be reduced to a simple cycle.

be a cycle from v to v where $v = v_0 = v_n$ (see Figure 8.2.12). If C is not a simple cycle, then $v_i = v_j$, for some $i < j < n$. We can replace C by the cycle

$C\_ = (v_0, e_1, v_1, \ldots, e_i, v_i, e_{j+1}, v_{j+1}, \ldots, e_n, v_n)$.

8.3 Hamiltonian Cycles and the Traveling Salesperson Problem

Sir William Rowan Hamilton marketed a puzzle in the mid-1800s in the form of a dodecahedron (see Figure 8.3.1).

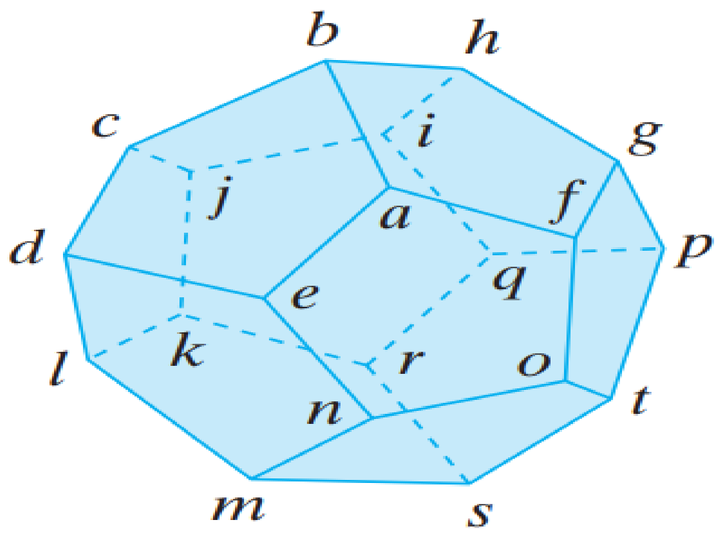


Figure 8.3.1 Hamilton's puzzle.

Each corner bore the name of a city and the problem was to start at any city, travel along the edges, visit each city exactly one time, and return to the initial city. The graph of the edges of the dodecahedron is given in Figure 8.3.2.

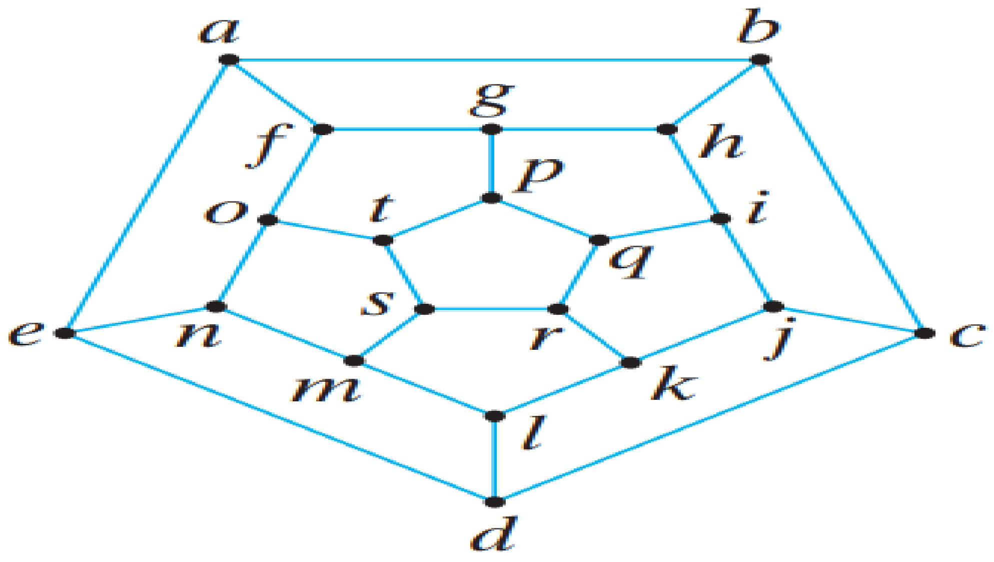


Figure 8.3.2 The graph of Hamilton's puzzle.

We can solve Hamilton's puzzle if we can find a cycle in the graph of Figure

8.3.2 that contains each vertex exactly once (except for the starting and ending vertex that appears twice). See if you can find a solution before looking at a solution given in Figure 8.3.3.

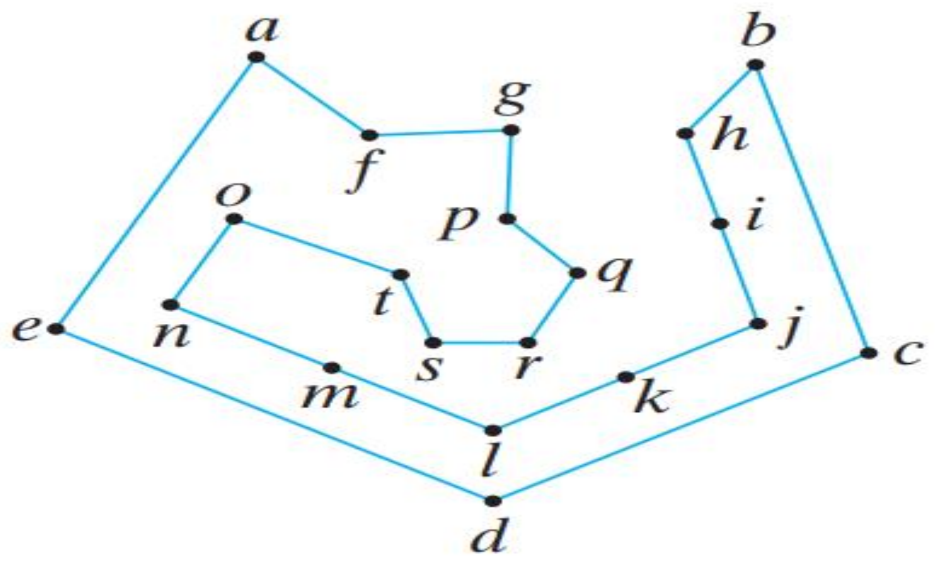


Figure 8.3.3 Visiting each vertex once in the graph of Figure 8.3.2.

In honor of Hamilton, we call a cycle in a graph *G* that contains each vertex in *G* exactly once, except for the starting and ending vertex that appears twice, a **Hamiltonian cycle.**

Example 8.3.1 The cycle *(a, b, c, d, e, f, g, a)* is a Hamiltonian cycle for the graph of Figure 8.3.4.

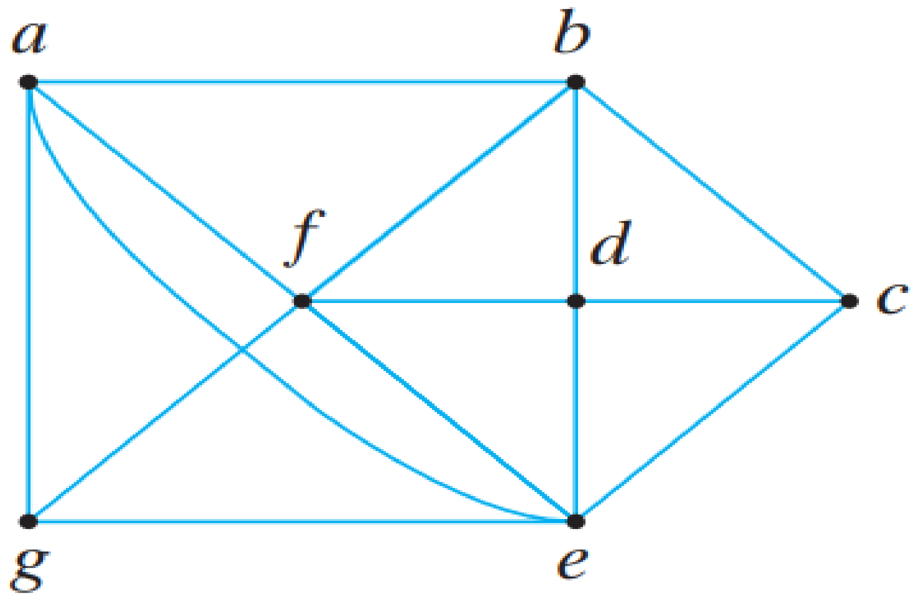


Figure 8.3.4 A graph with a Hamiltonian cycle.

The problem of finding a Hamiltonian cycle in a graph sounds similar to the problem of finding an Euler cycle in a graph. An Euler cycle visits each edge once, whereas a Hamiltonian cycle visits each vertex once; however, the problems are actually quite distinct. For example, the graph *G* of Figure 8.3.4 does not have an Euler cycle since there are vertices of odd degree, yet Example 8.3.1 showed that *G* has a Hamiltonian cycle. Furthermore, unlike the situation for Euler cycles (see Theorems 8.2.17 and 8.2.18), **no easily verified**

necessary and sufficient conditions are known for the existence of a Hamiltonian cycle in a graph.

Sometimeswing examples show that sometimes we can argue that a graph does not contain a Hamiltonian cycle.

Example 8.3.2

Show that the graph of Figure 8.3.5 does not contain a Hamiltonian cycle.

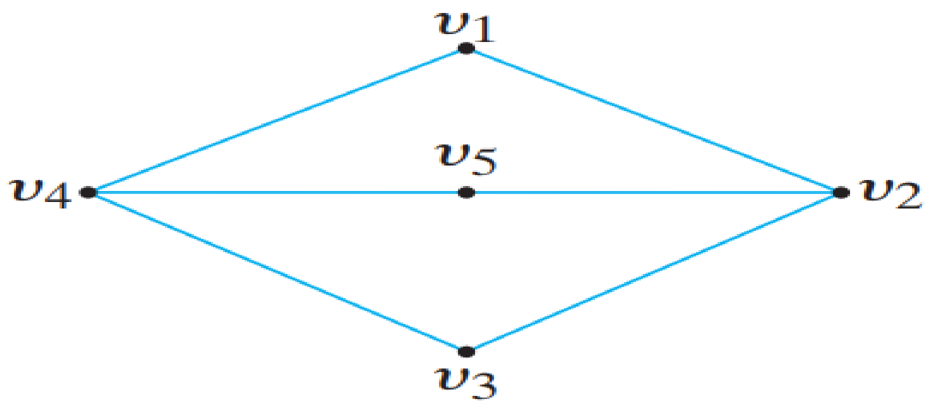


Figure 8.3.5 A graph with no Hamiltonian cycle.

SOLUTION Since there are five vertices, a Hamiltonian cycle must have five edges. Suppose that we could eliminate edges from the graph, leaving just a Hamiltonian cycle. We would have to eliminate one edge incident at v_2 and one edge incident at v_4, since each vertex in a Hamiltonian cycle has degree 2. But this leaves only four edges—not enough for a Hamiltonian cycle of length 5. Therefore, the graph of Figure 8.3.5 does not contain a Hamiltonian cycle.

■

We must be careful not to count an eliminated edge more than once when using an argument like that in Example 8.3.2 to show that a graph does not have a Hamiltonian cycle.

Notice in Example 8.3.2 (which refers to Figure 8.3.5) that if we eliminate one edge incident at v_2 and one edge incident at v_4, these edges are distinct. Therefore, we are correct in reasoning that we must eliminate two edges from the graph of Figure 8.3.5 to produce a Hamiltonian cycle.

As an example of double counting, consider the following ***faulty*** argument to show that the graph of Figure 8.3.6 has no Hamiltonian cycle.

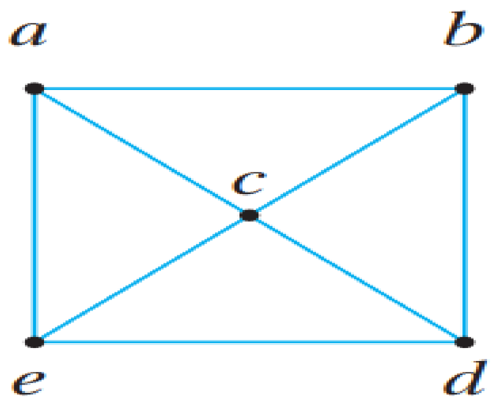


Figure 8.3.6 A graph *with* a Hamiltonian cycle.

Since there are five vertices, a Hamiltonian cycle must have five edges. Suppose that we could eliminate edges from the graph to produce a Hamiltonian cycle. We would have to eliminate two edges incident at *c* and one edge incident at each of *a*, *b*, *d*, and *e*. This leaves two Edges - not enough for a Hamiltonian cycle. Therefore, the graph of Figure 8.3.6 does not contain a Hamiltonian cycle. The **error** in this argument is that if we eliminate two edges incident at *c* (as we must do), we also eliminate edges incident at two of *a*, *b*, *d*, or *e*. We must not count the two eliminated edges incident at the two vertices again.

Notice that the graph of Figure 8.3.6 **does have a Hamiltonian cycle.** ■

Example 8.3.3

Show that the graph *G* of Figure 8.3.7 does not contain a Hamiltonian cycle.

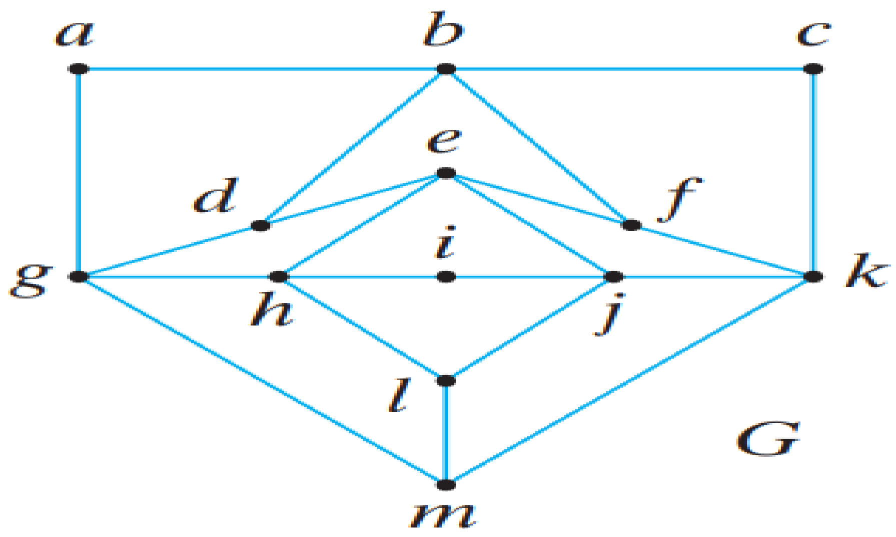


Figure 8.3.7 A graph with no Hamiltonian cycle.

SOLUTION

Suppose that *G* has a Hamiltonian cycle *H*. The edges *(a, b)*, *(a, g)*, *(b, c)*, and *(c, k)* must be in *H* since each vertex in a

Hamiltonian cycle has degree 2. Thus edges*(b, d)* and *(b, f)* are not in *H*. Therefore, edges *(g, d)*, *(d, e)*, *(e, f)*, and *(f , k)* are in *H*. The edges now known to be in *H* form a cycle *C*. Adding an additional edge to *C* will give some vertex in *H* degree greater than 2. This contradiction shows that *G* does not have a Hamiltonian cycle.

■

Traveling Salesperson Problem

The **traveling salesperson problem** is related to the problem of finding a Hamiltonian cycle in a graph. (We referred briefly to a variant of the traveling salesperson problem in Section 8.1.) The problem is given a weighted graph *G*, find a minimum length Hamiltonian cycle in *G*. If we think of the vertices in a weighted graph as cities and the edge weights as distances, the traveling salesperson problem is to find a shortest route in which the salesperson can visit each city one time, starting and ending at the same city.

Example 8.3.4

The cycle $C = (a, b, c, d, a)$ is a Hamiltonian cycle for the graph G of Figure 8.3.8.

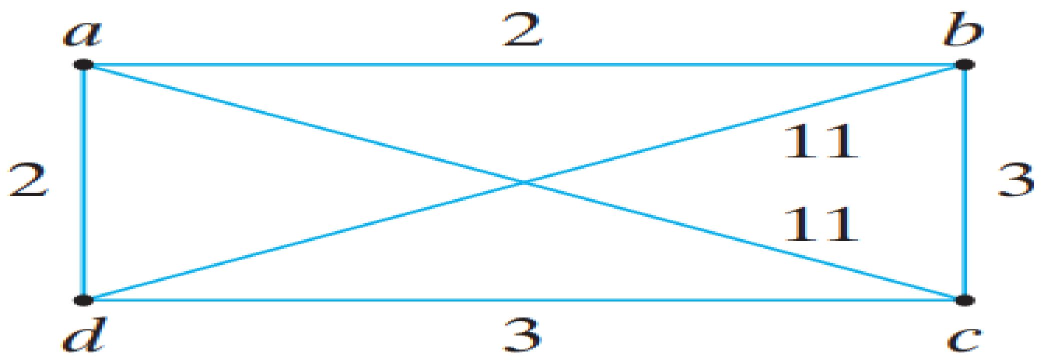


Figure 8.3.8 A graph for the traveling salesperson problem.

Replacing any of the edges in C by either of the edges labeled 11 would increase the length of C; thus, C is a minimum-length Hamiltonian cycle for G. Thus, C solves the traveling salesperson problem for G.

We next look at Hamiltonian cycles in the n-cube.

Example 8.3.5

Gray Codes and Hamiltonian Cycles in the *n*-Cube

Consider a **ring model** for parallel computation that, when represented as a graph, is a simple cycle (see Figure 8.3.9).

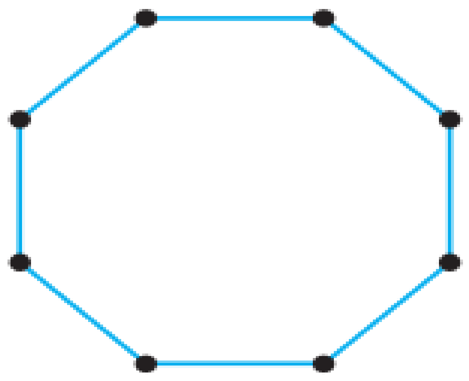


Figure 8.3.9 The ring model for parallel computation.

The vertices represent processors. An edge between processors *p* and *q* indicates that *p* and *q* can communicate directly with one another. We see that each processor can communicate directly with exactly two other processors. Nonadjacent processors communicate by sending messages.

The *n*-cube (see Example 8.1.7) is another model for parallel computation. The *n*-cube has a greater degree of connectivity among its processors. We consider the question of when an *n*-cube can simulate a ring model with 2^n processors. In graph terminology, we are asking when the *n*-cube contains a simple cycle with 2^n vertices as a subgraph or, since the *n*-cube has 2^n processors, when the *n*-cube contains a Hamiltonian cycle. [We leave to the exercises the question of when an *n*-cube can simulate a ring model with an arbitrary number of processors (see Exercise 18).]

We first observe that if the *n*-cube contains a Hamiltonian cycle, we must have $n \geq 2$ since the 1-cube has no cycles at all.

Recall (see Example 8.1.7) that we may label the vertices of the *n*-cube 0, 1, . . . ,

$2^n - 1$ in such a way that an edge connects two vertices if and only if the binary representation of their labels differs in exactly one bit. Thus the *n*-cube has a Hamiltonian cycle if and only if $n \geq 2$ and there is a sequence,

$$s_1, s_2, \ldots, s_{2^n} \qquad (8.3.1)$$

where each s_i is a string of *n* bits, satisfying:

- Every *n*-bit string appears somewhere in the sequence.
- s_i and s_{i+1} differ in exactly one bit,

$$I = 1, \ldots, 2^n - 1.$$

- s_{2^n} and s_1 differ in exactly one bit.

A sequence (8.3.1) is called a **Gray code.** When $n \geq 2$, a Gray code (8.3.1) corresponds to the Hamiltonian cycle s_1, s_2, . . . , s_{2^n}, s_1 since every vertex appears and the edges (s_i, s_{i+1}), $i = 1, . . . , 2^n - 1$, and (s_{2^n}, s_1) are distinct. When $n = 1$, the Gray code 0, 1 corresponds to the path (0, 1, 0), which is not a cycle because the edge (0, 1) is repeated. Gray codes have been extensively studied in other contexts. For example, Gray codes have been used in converting analog information to digital form (see [Deo]). We show how to construct a Gray code for each positive integer *n*, thus proving that the *n*-cube has a Hamiltonian cycle for every positive integer $n \geq 2$.

■

Theorem 8.3.6 Let G_1 denote the sequence 0, 1. We define G_1 in terms of G_{n-1} by the following rules:

(a) Let G_{n-1}^R denote the sequence G_{n-1} written in reverse.

(b) Let G'_{n-1} denote the sequence obtained by prefixing each member of G_{n-1} with 0.

(c) Let G''_{n-1} denote the sequence obtained by prefixing each member of $G^R_{n-1}$with 1.

(d) Let G_n be the sequence consisting of G'_{n-1} followed by G''_{n-1} .

Then G_n is a Gray code for every positive integer *n*.

Proof

We prove the theorem by induction on *n*.

Corollary 8.3.7

The *n*-cube has a Hamiltonian cycle for every positive integer $n \geq 2$.