Fall 2021 - MATH 1101 Discrete Structures
Lecture 11-12. Graphs and Trees

PART 1. GRAPHS AND GRAPH MODELS. BASIC TERMINOLOGY.

PART 2. REPRESENTING GRAPHS

PART 3. GRAPH ISOMORPHISM (Additional reading)

PART 4. PATHS, CONNECTIVITY, EULERIAN AND HAMILTONIAN GRAPHS

PART 5. SHORTEST PATHS IN WEIGHTED GRAPHS. TREE GRAPHS. SPANNING TREES.

PART 6. KEY TERMS AND RESULTS.

INTRODUCTION.

Graph is a discrete structure consisting of vertices and edges that connect these vertices. There are different kinds of graphs, depending on whether edges have directions, whether multiple edges can connect the same pair of vertices, and whether loops are allowed. Problems in almost every conceivable discipline can be solved using graph models. We will give examples to illustrate how graphs are used as models in a variety of areas.

We start with formal definitions and terminology.

PART 1. GRAPHS AND GRAPH MODELS. BASIC TERMINOLOGY.

Definition 1: A *graph* G = (V, E) consists of V, a nonempty set of *vertices* (or *nodes*) and E, a set of *edges*. Each edge has either one or two vertices associated with it, called its *endpoints*. An edge is said to *connect* its endpoints. ■

Definition 2: A graph in which each edge connects two different vertices and where no two edges connect the same pair of vertices is called a **simple graph**.

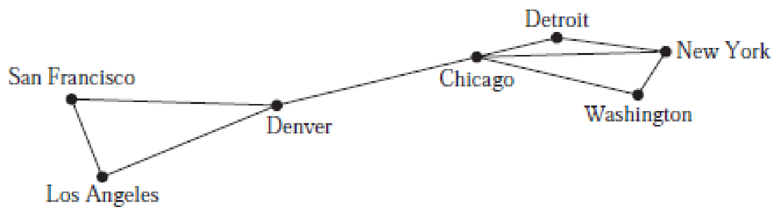


Figure A. Computer network – **simple graph**

Graphs that may have **multiple edges** connecting the same vertices are called **multigraphs** (Figure B). When there are m different edges associated to the same unordered pair of vertices {u, *v*}, we also say that {u, *v*} is an edge of multiplicity m.

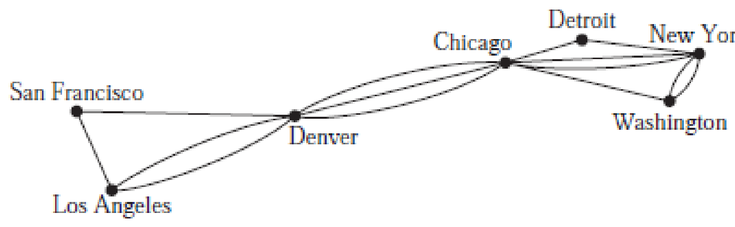


Figure B. Computer network – **multigraph**

Sometimes a communications link connects a data center with itself, perhaps a feedback loop for diagnostic purposes. Such a network is illustrated in Figure C. To model this network we need to include edges that connect a vertex to itself. Such edges are called **loops**, and sometimes we may even have more than one loop at a vertex.

Graphs that may include loops, and possibly multiple edges connecting the same pair of vertices or a vertex to itself, are sometimes called **pseudographs** (Figure C).

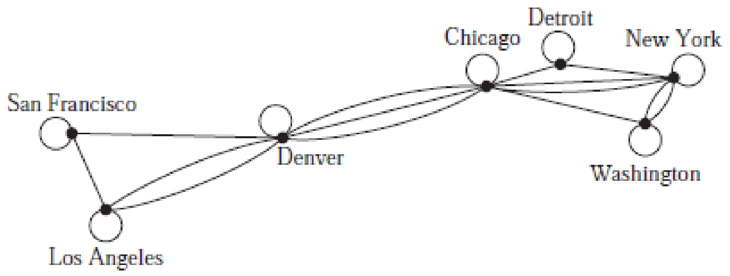


Figure C. A Computer Network with Diagnostic Links.

■

So far the graphs we have introduced are **undirected** graphs. Their edges are also said to be **undirected**. However, to construct a graph model, we may find it necessary to assign directions to the edges of a graph.

Definition 3: A *directed graph* (or *digraph*) (V, E) consists of a nonempty set of vertices V and a set of *directed edges* (or *arcs*) E. Each directed edge is associated with an ordered pair of vertices. The directed edge associated with the ordered pair (u, *v*) is said to *start* at u and *end* at *v*. ■

Definition 4: When a directed graph has no loops and has no multiple directed edges, it is called a **simple directed graph** (Figure D).

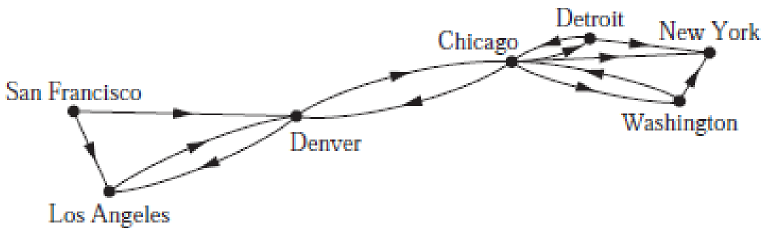


Figure D. A Communications Network with One-Way Communications Links.

Directed graph having **multiple directed edges** from a vertex to a second (possibly the same) vertex is called **directed multigraph** (Figure E)**.** When there are m directed edges, each associated to an ordered pair of vertices (u, *v*), we say that (u, *v*) is an edge of **multiplicity m.**

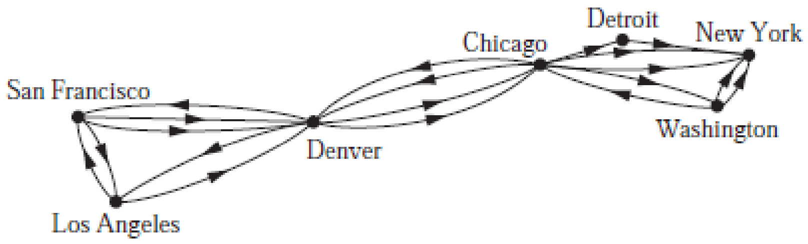


Figure E. A Computer Network with Multiple One-Way Links.

A graph with both directed and undirected edges is called **a mixed graph.** ■

The terminology for the various types of graphs is summarized in Table 1 below.

TABLE 1 Graph Terminology (based on Kenneth Rosen textbook)

Type	**Edges**	***Multiple Edges Allowed?***	***Loops Allowed?***
Simple graph	Undirected	No	No
Multigraph	Undirected	Yes	No
Pseudograph	Undirected	Yes	Yes
Simple directed graph	Directed	No	No
Directed multigraph	Directed	Yes	Yes
Mixed graph	Directed and Undirected	Yes	Yes

Because of graph theory has applications to a wide variety of disciplines, many different terminologies of graph theory have been introduced. The terminology used by mathematicians to describe graphs has been increasingly standardized, but the terminology used to discuss graphs when they are used in other disciplines is still quite varied. Although the terminology used to describe graphs may vary, three key questions can help us understand the structure of a graph:

- Are the edges of the graph undirected or directed (or both)?

- If the graph is undirected, are multiple edges present that connect the same pair of vertices?

 If the graph is directed, are multiple directed edges present?

- Are loops present?

Answering such questions helps us understand graphs.

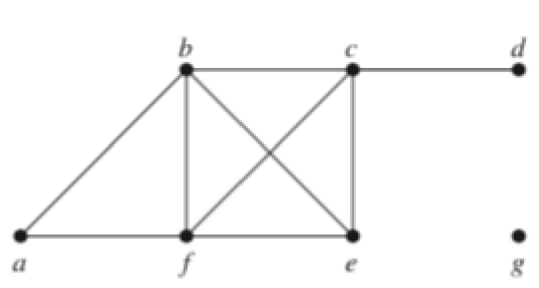
Definition 5: Two vertices u and *v* in an undirected graph G are called *adjacent* (or *neighbors*) in G if u and *v* are endpoints of an edge e of G. Such an edge e is called *incident with* the vertices u and *v* and e is said to *connect* u and *v*. ■

Definition 6: The set of all neighbors of a vertex *v* of G = (V,E), denoted by N(*v*), is called the *neighborhood* of *v*. If A is a subset of V, we denote by *N(A)* the set of all vertices in G that are adjacent to at least one vertex in A. So, $N(A)=\bigcup_{v\in A} N(v)$ ■

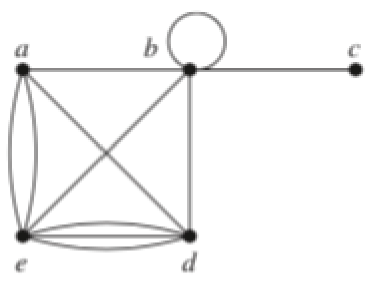
To keep track of how many edges are incident to a vertex, we make the following definition.

Definition 7: The *degree of a vertex in an undirected graph* is the number of edges incident with it, except that a loop at a vertex contributes twice to the degree of that vertex. The degree of the vertex *v* is denoted by deg(*v*). ■

EXAMPLE 1. What are the degrees and what are the neighborhoods of the vertices in the graphs G and H displayed in Figure 1?



Graph G



Graph H

FIGURE 1 The Undirected Graphs G and H .

Solution:

In G, deg(a)=2, deg(b)=deg(c)=deg(f)=4, deg(d)=1, deg(e)=3, and deg(g)=0. The neighborhoods of these vertices are N(a)={b, f}, N(b)={a, c, e, f}, N(c)={b, d, e, f}, N(d)={c}, N(e)={b, c, f}, N(f)={a, b, c, e}, and N(g)=∅.

In H, deg(a)=4, deg(b)=deg(e)=6, deg(c)=1, and deg(d)=5. The neighborhoods of these vertices are N(a)={b, d, e}, N(b)={a, b, c, d, e}, N(c)={b}, N(d)={a, b, e}, and N(e)={a, b, d}. ■

A vertex of degree zero is called **isolated**. It follows that an isolated vertex is not adjacent to any vertex. Vertex g in graph G in Example 1 is isolated. A vertex is **pendant** if and only if it has degree one. Consequently, a pendant vertex is adjacent to exactly one other vertex. Vertex d in graph G in Example 1 is pendant.

Examining the degrees of vertices in a graph model can provide useful information about the model.

What do we get when we add the degrees of all the vertices of a graph G = (V , E)? Each edge contributes two to the sum of the degrees of the vertices because **an edge is incident with exactly two (possibly equal) vertices**. This means that the sum of the degrees of the vertices is twice the number of edges. We have the result in Theorem 1, which is sometimes called the handshaking theorem (and is also often known as the handshaking lemma), because of the analogy between an edge having two endpoints and a handshake involving two hands.

Theorem 1: (The Handshaking theorem). Let G = (V , E) be an undirected graph with m edges. Then

$$2m=\sum_{v\in V} \deg(v)$$

(Note that this applies even if multiple edges and loops are present.) ■

EXAMPLE 2. How many edges are there in a graph with 10 vertices each of degree six?

Solution: Because the sum of the degrees of the vertices is $6 \cdot 10 = 60$, it follows that $2m = 60$ where m is the number of edges. Therefore, $m = 30$. ■

Theorem 1 shows that the sum of the degrees of the vertices of an undirected graph is even. This simple fact has many consequences, one of which is given as Theorem 2.

Theorem 2: An undirected graph has an even number of vertices of odd degree.

Proof. Exercise.

Terminology for graphs with directed edges reflects the fact that edges in directed graphs have directions.

Definition 8: When (u, *v*) is an edge of the graph G with directed edges, u is said to be *adjacent to v* and *v* is said to be *adjacent from u*. The vertex u is called the *initial vertex* of (u, *v*), and *v* is called the *terminal* or *end vertex* of (u, *v*). The initial vertex and terminal vertex of a loop are the same. ■

Because the edges in graphs with directed edges are ordered pairs, the definition of the degree of a vertex can be refined to reflect the number of edges with this vertex as the initial vertex and as the terminal vertex.

Definition 9: In a directed graph G with directed edges the *in-degree of a vertex v*, denoted by deg−(*v*), is the number of edges with *v* as their terminal vertex. The *out-degree of v*, denoted by deg+(*v*), is the number of edges with *v* as their initial vertex. (Note that a loop at a vertex contributes 1 to both the in-degree and the out-degree of this vertex.) . ■

EXAMPLE 3. Find the in-degree and out-degree of each vertex in the graph G in Figure 2.

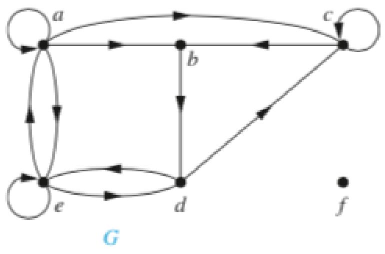


FIGURE 2 The Directed Graph G.

Solution: The in-degrees in G are deg−(a) = 2, deg−(b) = 2, deg−(c) = 3, deg−(d) = 2, deg−(e) = 3, and deg−(f) = 0. The out-degrees are deg+(a) = 4, deg+(b) = 1, deg+(c) = 2, deg+(d) = 2, deg+(e) = 3, and deg+(f) = 0. ■

Theorem 3: Let G = (V , E) be a graph with directed edges. Then

$$\sum_{v\in V} deg^{-}(v) = \sum_{v\in V} deg^{+}(v) = |E|$$

Proof. Because each edge has an initial vertex and a terminal vertex, the sum of the in-degrees and the sum of the out-degrees of all vertices in a graph with directed edges are the same. Both sums are the number of edges in the graph. ■

PART 2. REPRESENTING GRAPHS

There are many useful ways to represent graphs. As we will see throughout this Part, in working with a graph it is helpful to be able to choose its most convenient representation. In this section we will show how to represent graphs in several different ways.

Sometimes, two graphs have the same form, in the sense that there is a one-to-one correspondence between their vertex sets that preserves edges. In such a case, we say that the two graphs are **isomorphic**. Determining whether two graphs are isomorphic is an important problem of graph theory that we will study in this section.

Representing Graphs.

Representing by Adjacency Lists

One way to represent a **graph without multiple edges** is to list all the edges of this graph. Another way to represent a graph with no multiple edges is to use **adjacency lists**, which specify the vertices that are adjacent to each vertex of the graph.

EXAMPLE 4. Use adjacency lists to describe the **simple graph** given in Figure 3

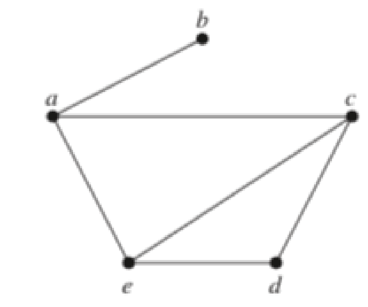


FIGURE 3 A Simple Graph.

TABLE 1 An Adjacency List for a Simple Graph.

Vertex	*Adjacent Vertices*
a	b, c, e
b	a
c	a, d, e
d	c, e
e	a, c, d

Solution: Table 1 lists those vertices adjacent to each of the vertices of the graph. ■

EXAMPLE 5. Represent the **directed graph** shown in Figure 4 by listing all the vertices that are the terminal vertices of edges starting at each vertex of the graph.

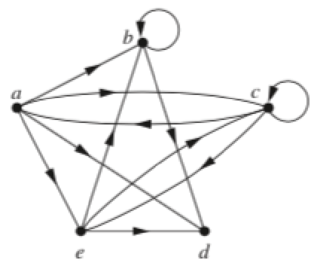


FIGURE 4 A Directed Graph.

TABLE 2 An Adjacency List for a Directed Graph**.**

Initial Vertex	*Terminal Vertices*
a	b, c, d, e
b	b, d
c	a, c, e
d	
e	b, c, d

Solution: Table 2 represents the directed graph shown in Figure 4. ■

Representing by Matrices

Carrying out graph algorithms using the representation of graphs by lists of edges, or by adjacency lists, can be cumbersome if there are many edges in the graph. To simplify computation, graphs can be represented using matrices. **Two types of matrices commonly used to represent graphs will be presented here. One is based on the adjacency of vertices, and the other is based on incidence of vertices and edges.**

Adjacency Matrices.

Definition 10: Suppose that G=(V, E) is a simple graph where |V | = n. Suppose that the vertices of G are listed arbitrarily as $v_1, v_2, \ldots, v_n$. The **adjacency matrix A** (or **A**G) of G, with respect to this listing of the vertices, is the (n×n) zero–one matrix with 1 as its (i, j)th entry when v_i and v_j are adjacent, and 0 as its (i, j)th entry when they are not adjacent. In other words, if its adjacency matrix is **A**=(a_{ij}), then

$$a_{ij} = \begin{cases} 1, & if\ (v_i,\ v_j) is\ an\ edge\ of\ G \\ 0, & otherwise \end{cases}$$ ■

EXAMPLE 6. Use an adjacency matrix to represent the graph shown in Figure 5

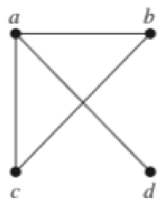


FIGURE 5. Simple Graph.

Solution: We order the vertices as a, b, c, d. The matrix representing this graph is

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

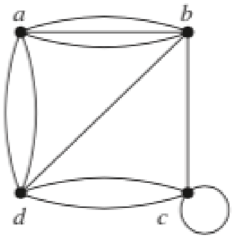
■

Note that **an adjacency matrix of a graph is based on the ordering chosen for the vertices**. Hence, there may be as many as n! different adjacency matrices for a graph with n vertices, because there are n! different orderings (=permutations) of n vertices.

The adjacency matrix of a simple graph is symmetric, that is, $a_{ij}=a_{ji}$, because both of these entries are 1 when v_i and v_j a are adjacent, and both are 0 otherwise. Furthermore, because a simple graph has no loops, each entry a_{ii}, i=1, 2, 3, ..., n, is 0.

Adjacency matrices can also be used to represent undirected graphs with loops and with multiple edges. A loop at the vertex v_i is represented by a 1 at the (i, i) th position of the adjacency matrix. When multiple edges connecting the same pair of vertices v_i and v_j, or multiple loops at the same vertex, are present, the adjacency matrix is no longer a zero–one matrix, because the (i, j)th entry of this matrix equals the number of edges that are associated to $\{v_i, v_j\}$. **All undirected graphs, including multigraphs and pseudographs, have symmetric adjacency matrices.**

EXAMPLE 7. Use an adjacency matrix to represent the pseudograph shown in Figure 6(a)



$$\begin{bmatrix} 0 & 3 & 0 & 2 \\ 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 2 \\ 2 & 1 & 2 & 0 \end{bmatrix}$$

(a) (b)

FIGURE 6. A Pseudograph 6(a) and its representation by adjacency matrix 6(b).

We use zero–one matrix to represent directed graphs. The matrix for a directed graph G = (V, E) has a 1 in its (i,j)-th position if there is an edge from v_i to v_j, where $v_1, v_2, \ldots, v_n$ is an arbitrary listing of the vertices of the directed graph. In other words, if **A** = $[a_{ij}]$ is the adjacency matrix for the directed graph with respect to this listing of the vertices, then

$$a_{ij} = \begin{cases} 1, & if\ (v_i,\ v_j) is\ an\ edge\ of\ G \\ 0, & otherwise \end{cases}$$

The adjacency matrix for a directed graph **does not have to be symmetric,** because there may not be an edge from v_j to v_i when there is an edge from v_i to v_j.

Adjacency matrices can also be used to represent directed multigraphs. Again, such matrices are not zero–one matrices when there are multiple edges in the same direction connecting two vertices. In the adjacency matrix for a directed multigraph, a_{ij} equals the number of edges that are associated to (v_i, v_j).

TRADE-OFFS BETWEEN ADJACENCY LISTS AND ADJACENCY MATRICES.

When a simple graph contains relatively few edges, that is, when it is **sparse**, it is usually preferable to use adjacency lists rather than an adjacency matrix to represent the graph. For example, if each vertex has degree not exceeding c, where c is a constant much smaller than n, then each adjacency list contains c or fewer vertices. Hence, there are no more than cn items in all these adjacency lists. On the other hand, the adjacency matrix for the graph has n^2 entries. Note, however, that the adjacency matrix of a sparse graph is a **sparse matrix**, that is, a matrix with few nonzero entries, and there are special techniques for representing, and computing with, sparse matrices.

Now suppose that a simple graph is **dense**, that is, suppose that it contains many edges, such as a graph that contains more than half of all possible edges. In this case, using an adjacency matrix to represent the graph is usually preferable over using adjacency lists. To see why, we compare the complexity of determining whether the possible edge $\{v_i, v_j\}$ is present. Using an adjacency matrix, we can determine whether this edge is present by examining the (i, j)-th entry in the matrix. This entry is 1 if the graph contains this edge and is 0 otherwise. Consequently, we need make only one comparison, namely, comparing this entry with 0, to determine whether this edge is present. On the other hand, when we use adjacency lists to represent the graph, we need to search the list of vertices adjacent to either v_i or v_j to determine whether this edge is present. This can require $\Theta(|V|)$ comparisons when many edges are present. (See next collection of lecture notes on complexity of algorithms for $\Theta(|V|)$).

Incidence Matrices.

Another common way to represent graphs is to use **incidence matrices**.

Definition 11: Let G=(V, E) be an undirected graph. Suppose that $v_1, v_2, \ldots, v_2$ are the vertices and $e_1, e_2, \ldots, e_m$ are the edges of G. Then the incidence matrix with respect to this ordering of V and E is the (n×m) matrix **M**=$[m_{ij}]$, where

$$m_{ij}=\begin{cases}1, if\ e_j\ \ is\ incident\ with\ \ v_i\\ 0, \qquad\qquad\qquad otherwise\end{cases}$$

EXAMPLE 8. Represent the graph shown in Figure 7 with an incidence matrix.

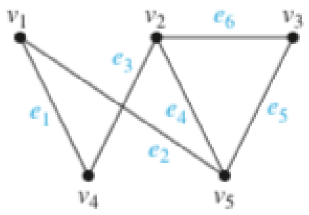


FIGURE 7. An Undirected Graph.

Solution: The incidence matrix is

$$\begin{array}{c c} & \begin{array}{cccccc} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 \end{array} \\ \begin{array}{c} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array} & \left[\begin{array}{cccccc} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \end{array}\right]. \end{array}$$

■

Incidence matrices can also be used to represent multiple edges and loops. Multiple edges are represented in the incidence matrix using columns with identical entries, because these edges are incident with the same pair of vertices. Loops are represented using a column with exactly one entry equal to 1, corresponding to the vertex that is incident with this loop.

EXAMPLE 9. Represent the graph shown in Figure 8 with an incidence matrix.

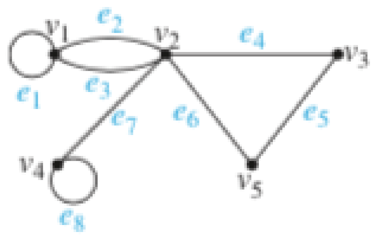


FIGURE 8. A Pseudograph.

Solution: The incidence matrix is

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array}
\begin{array}{c} \begin{array}{cccccccc} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & e_8 \end{array} \\
\begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

■

PART 3. GRAPH ISOMORPHISM (Additional reading)

We often need to know whether it is possible to draw two graphs in the same way. That is, do the graphs have the same structure when we ignore the identities of their vertices?

There is a useful terminology for graphs with the same structure.

Definition 12: The simple graphs $G_1=(V_1, E_1)$ and $G_2=(V_2, E_2)$ are *isomorphic* if the following two conditions are satisfied:

1) there exists a one-to-one and onto function *f* from V_1 to V_2

2) vertices *a* and *b* are adjacent in G_1 if and only if *f*(*a*) and *f*(*b*) are adjacent in G_2, for all *a* and *b* in V_1.

Such a function *f* is called an *isomorphism*. Two simple graphs that are not isomorphic are called *nonisomorphic*. ■

In other words, when two simple graphs are isomorphic, there is a one-to-one correspondence between vertices of the two graphs that preserves the adjacency relationship.

Theorem 4: Isomorphism of simple graphs is an equivalence relation.

Proof. Exercise. ■

EXAMPLE 10. Show that the graphs G=(V, E) and H=(W, F) in Figure 9, are isomorphic.

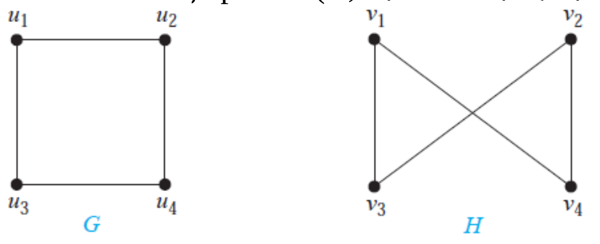


FIGURE 9 The Graphs G and H.

Solution:

The function f with $f(u_1)=v_1$, $f(u_2)=v_4$, $f(u_3)=v_3$, and $f(u_4)=v_2$ is a one-to-one correspondence between V and W. To see that this correspondence preserves adjacency, note that adjacent vertices in G are pairs of vertices (u_1, u_2), (u_1, u_3), (u_2, u_4), and (u_3, u_4), and each of the pairs [f(u_1)=v_1, f(u_2)=v_4], [f(u_1)=v_1, f(u_3)=v_3], [f(u_2)=v_4, f(u_4)=v_2] and [f(u_3)=v_3, f(u_4)=v_2] consists of two adjacent vertices in H . ■

Determining whether Two Simple Graphs are Isomorphic

It is often difficult to determine whether two simple graphs are isomorphic. There are n! possible one-to-one correspondences between the vertex sets of two simple graphs with n vertices. Testing each such correspondence to see whether it preserves adjacency and nonadjacency is impractical if n is at all large.

Sometimes it is not hard to show that two graphs are not isomorphic. In particular, we can show that two graphs are not isomorphic if we can find a property only one of the two graphs has, but that is preserved by isomorphism.

Definition 13. A property of a graph which is preserved under an isomorphism of graphs is called a **graph invariant.** ■

Clear that:

- isomorphic simple graphs must have the same number of vertices, because there is a one-to-one correspondence between the sets of vertices of the graphs;
- isomorphic simple graphs also must have the same number of edges, because the one-to-one correspondence between vertices establishes a one-to-one correspondence between edges;
- in isomorphic simple graphs the degrees of the corresponding vertices must be the same. That is, a vertex v of degree d in G must correspond to a vertex f(v) of degree d in H , because a vertex w in G is adjacent to v if and only if f(v) and f(w) are adjacent in H.

Thus,

- the number of vertices,
- the number of edges,
- the number of vertices of each degree

are all invariants under isomorphism. If any of these quantities differ in two simple graphs, these graphs cannot be isomorphic.

EXAMPLE 11. Show that the graphs displayed in Figure 10 are not isomorphic.

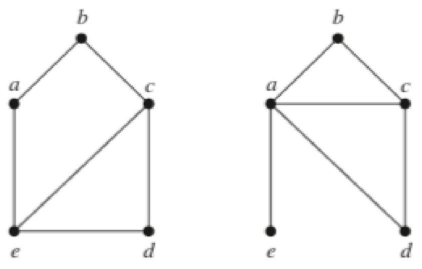


FIGURE 10 The Graphs G and H.

Solution: Both G and H have five vertices and six edges. However, H has a vertex of degree one, namely, e, whereas G has no vertices of degree one. It follows that G and H are not isomorphic. ■

However, when these invariants are the same, it does not necessarily mean that the two graphs are isomorphic. There are no useful sets of invariants currently known that can be used to determine whether simple graphs are isomorphic.

EXAMPLE 12. Determine whether the graphs shown in Figure 11 are isomorphic.

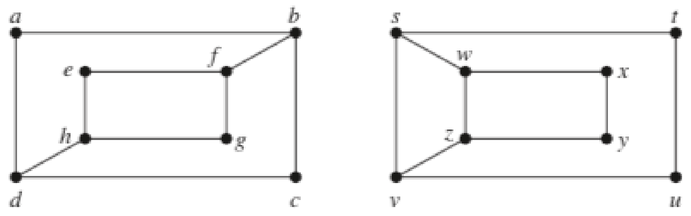


FIGURE 11. The Graphs G and H.

Solution: The graphs G and H both have 8 vertices and 10 edges. They also both have four vertices of degree two and four of degree three. Because these invariants all agree, it is still conceivable that these graphs are isomorphic.

However, G and H are not isomorphic. To see this, note that because deg(a)=2 in G, the vertex *a* must correspond to either t, u, x, or y in H, because these are the vertices of degree two in H. However, each of these four vertices in H is adjacent to another vertex of degree two in H, which is not true for a in G. ■

Another way to see that G and H are not isomorphic is to note that the subgraphs of G and H made up of **vertices of degree three and the edges connecting them** must be isomorphic if these two graphs are isomorphic (the reader should verify this). However, these subgraphs, shown in Figure 12, are not isomorphic. ■

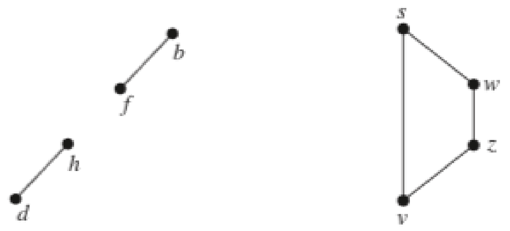


FIGURE 12. The Subgraphs of G and H made up of vertices of deg(3) and the edges connecting them.

EXAMPLE 13. Determine whether the graphs G and H displayed in Figure 13 are isomorphic.

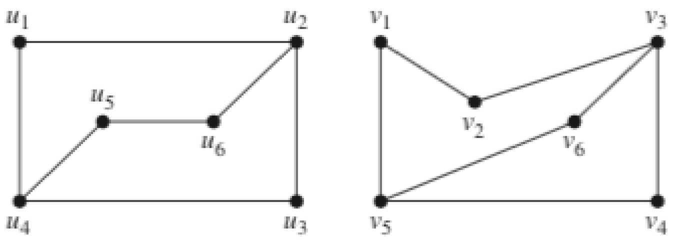


FIGURE 13 Graphs G and H.

Solution: Both G and H have six vertices and seven edges. Both have four vertices of degree two and two vertices of degree three. It is also easy to see that the subgraphs of G and H consisting of all vertices of degree two and the edges connecting them are isomorphic (as the reader should verify). Because G and H agree with respect to these invariants, it is reasonable to try to find an isomorphism f.

We now will define a function f and then determine whether it is an isomorphism. Because deg(u_1)=2 and because u_1 is not adjacent to any other vertex of degree two, the image of u_1 must be either v_4 or v_6, the only vertices of degree two in H not adjacent to a vertex of degree two. We arbitrarily set f (u_1)=v_6. (If we found that this choice did not lead to isomorphism, we would then try f(u_1)=v_4). Because u_2 is adjacent to u_1, the possible images of u_2 are v_3 and v_5. We arbitrarily set f(u_2)=v_3. Continuing in this way, using adjacency of vertices and degrees as a guide, we set f(u_3)=v_4, f(u_4)=v_5, f(u_5)=v_1, and f(u_6)=v_2.We now have a one-to-one correspondence between the vertex set of

G and the vertex set of H, namely, $f(u_1)=v_6$, $f(u_2)=v_3$, $f(u_3)=v_4$, $f(u_4)=v_5$, $f(u_5)=v_1$, $f(u_6)=v_2$.

To see whether f preserves edges, we examine the adjacency matrix of G (denoted as $\mathbf{A}_G$), and the adjacency matrix of H (denoted as $\mathbf{A}_H$) with the rows and columns labeled by the images of the corresponding vertices in G.

Because $\mathbf{A}_G=\mathbf{A}_H$, it follows that f preserves edges. We conclude that f is an isomorphism, so G and H are isomorphic. Note that if f turned out not to be an isomorphism, we would *not* have established that G and H are not isomorphic, because another correspondence of the vertices in G and H may be an isomorphism.

$$\mathbf{A}_G = \begin{array}{c|cccccc} & u_1 & u_2 & u_3 & u_4 & u_5 & u_6 \\ \hline u_1 & 0 & 1 & 0 & 1 & 0 & 0 \\ u_2 & 1 & 0 & 1 & 0 & 0 & 1 \\ u_3 & 0 & 1 & 0 & 1 & 0 & 0 \\ u_4 & 1 & 0 & 1 & 0 & 1 & 0 \\ u_5 & 0 & 0 & 0 & 1 & 0 & 1 \\ u_6 & 0 & 1 & 0 & 0 & 1 & 0 \end{array}, \qquad \mathbf{A}_H = \begin{array}{c|cccccc} & v_6 & v_3 & v_4 & v_5 & v_1 & v_2 \\ \hline v_6 & 0 & 1 & 0 & 1 & 0 & 0 \\ v_3 & 1 & 0 & 1 & 0 & 0 & 1 \\ v_4 & 0 & 1 & 0 & 1 & 0 & 0 \\ v_5 & 1 & 0 & 1 & 0 & 1 & 0 \\ v_1 & 0 & 0 & 0 & 1 & 0 & 1 \\ v_2 & 0 & 1 & 0 & 0 & 1 & 0 \end{array}. \quad \blacksquare$$

APPLICATIONS OF GRAPH ISOMORPHISMS

Graph isomorphisms, and functions that are almost graph isomorphisms, arise in applications of graph theory to chemistry and to the design of electronic circuits, and other areas including bioinformatics and computer vision.

Chemists use multigraphs, known as molecular graphs, to model chemical compounds. In these graphs, vertices represent atoms and edges represent chemical bonds between these atoms. Two structural isomers, molecules with identical molecular formulas but with atoms bonded differently, have non-isomorphic molecular graphs. When a potentially new chemical compound is synthesized, a database of molecular graphs is checked to see whether the molecular graph of the compound is the same as one already known.

Electronic circuits are modeled using graphs in which vertices represent components and edges represent connections between them. Modern integrated circuits, known as chips, are miniaturized electronic circuits, often with millions of transistors and connections between them. Because of the complexity of modern chips, automation tools are used to design them. Graph isomorphism is the basis for the verification that a particular layout of a circuit produced by an automated tool corresponds to the original schematic of the design. Graph isomorphism can also be used to determine whether a chip from one vendor includes intellectual property from a different vendor. This can be done by looking for large isomorphic subgraphs in the graphs modeling these chips.

PART 4. PATHS, CONNECTIVITY, EULERIAN AND HAMILTONIAN GRAPHS

Introduction.

Question 1. **Can we travel along the edges of a graph starting at a vertex and returning to it** by traversing **each edge** of the graph **exactly once**?

Question 2. **Can we travel along the edges of a graph starting at a vertex and returning to it** while visiting **each vertex** of the graph **exactly once**?

Although these questions seem to be similar, the first question, which asks whether a graph has an ***Euler circuit***, can be easily answered simply by examining the degrees of the vertices of the graph, while the second question, which asks whether a graph has a ***Hamilton circuit***, is quite difficult to solve for most graphs.

In the lecture we study these questions and discuss the difficulty of solving them.

We use terminology from ***SL*** (in Parenthesis - equivalent terminology from ***KR***)

Definition 14: A ***path*** in a pseudograph G consists of an alternating sequence of vertices and edges of the form $v_0, e_1, v_1, e_2, v_2, ..., e_{n-1}, v_{n-1}, e_n, v_n$ where each edge e_i contains the vertices v_{i-1} and v_i (which appear on the sides of e_i in the sequence). The number n of edges is called the *length* of the path. When there is no ambiguity, we denote a path by its sequence of vertices $(v_0, v_1, . . . , v_n)$ – it is possible when listing these vertices uniquely determines the path).

- The path is said to be ***circuit** or **closed*** **if $v_0=v_n$** and it has a length greater than zero. Otherwise, we say the path is from v_0, to v_n or *between* v_0 and v_n, or *connects* v_0 to v_n.
- A ***simple path*** is a path in which **all** vertices **are distinct** (no repeated vertices).
- A **trail** is defined to be a path in which **all** edges **are distinct** (no repeated edges) ***(trail=simple path in KR terminology)***.
- A **cycle(=simple circuit)** is a *circuit $v_1, v_2, ..., v_n$* in which **all** **vertices** **are distinct besides *$v_1=v_n$***. ■

Clear that any simple path is the trail but not vice versa (Example 14 below).

EXAMPLE 14. Consider the graph G in Figure 14 with the following sequences:

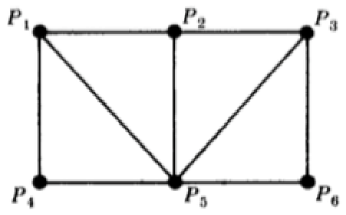


Figure 14.

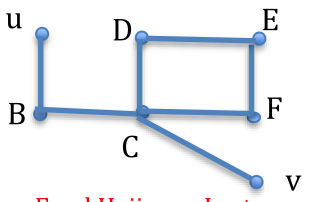
$\alpha = (P_4, P_1, P_2, P_5, P_1, P_2, P_3, P_6)$, $\beta = (P_4, P_1, P_5, P_2, P_6)$,

$\gamma = (P_4, P_1, P_5, P_2, P_3, P_5, P_6)$, $\delta = (P_4, P_1, P_5, P_3, P_6)$.

- **The path α is** a path from P_4 to P_6; but it is **not a trail** since the edge $\{P_1, P_2\}$ is used twice.
- The sequence β **is not a path** since there is no edge $\{P_2, P_6\}$.
- The sequence γ **is a trail** since no edge is used twice; but γ **is not a simple path** since the vertex P_5 is used twice.
- The sequence δ **is a simple path** from P_4 to P_6; but it is not the shortest path (with respect to length) from P_4 to P_6. The shortest path from P_4 to P_6 is the simple path (P_4, P_5, P_6) which has length 2. ■

By eliminating unnecessary edges, it is not difficult to see that any path from a vertex u to a vertex v can be replaced by a simple path from u to v. We state this result formally.

Theorem 5: There is a path from a vertex u to a vertex v if and only if there exists a simple path from u to v. ■



uBCDEFCv (path from u to v) uBCv (simple path from u to v)

Connectivity, Connected Components

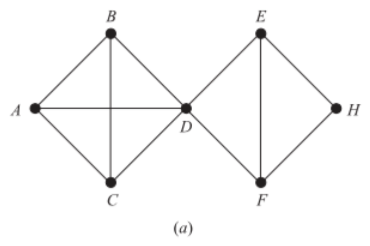
A graph G is ***connected*** if there is a path between any two of its vertices. The graph in Figure 14 is connected.

Suppose G is a graph. A connected subgraph H of G is called a ***connected component*** of G if H is not contained in any larger connected subgraph of G. It is intuitively clear that any graph G can be partitioned into its connected components.

Remark: Formally speaking, assuming any vertex u is connected to itself, the relation "u is connected to v" is an equivalence relation on the vertex set of a graph G and the equivalence classes of the relation form the connected components of G.

Distance and Diameter

Consider a connected graph G. The ***distance*** between vertices u and v in G, written d(u, v), is the length of the shortest path between u and v. The ***diameter*** of G, written diam(G), is the maximum distance between any two points in G. For example, in Figure 15 (a), d(A, F)=2 and diam(G)=3, whereas in Figure 15 (b), d(A, F)=3 and diam(G)=4.



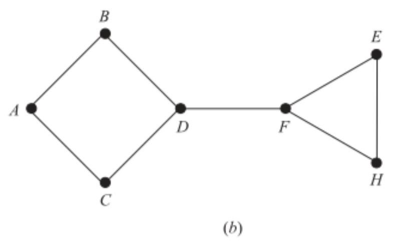


FIGURE 15 (a)   FIGURE 15 (b)

Cutpoints and Bridges

Let G be a connected graph. A vertex v in G is called a ***cutpoint*** if (G – v) is disconnected. (By definition, (G – v) is the graph obtained from G by deleting v and all edges containing v.) An edge e of G is called a ***bridge*** if G − e is disconnected. (By definition, (G – e) is the graph obtained from G by simply deleting the edge e). In Figure 15(a), the vertex D is a cutpoint and there are no bridges. In Figure 15 (b), the edge={D, F } is a bridge. (Its endpoints D and F are necessarily cutpoints.)

TRAVERSABLE AND EULERIAN GRAPHS

The eighteenth-century East Prussian town of Königsberg included two islands and seven bridges as shown in Figure 16(a). Question: **Beginning anywhere and ending anywhere, can a person walk through town crossing all seven bridges but not crossing any bridge twice?** The people of Königsberg wrote to the celebrated Swiss mathematician L. Euler about this question. Euler proved in 1736 that such a walk is impossible. He replaced the islands and the two sides of the river by points and the bridges by curves, obtaining Figure 16(b).

Observe that Figure 16 (b) is a multigraph.

Definition 15: Let G be a multigraph (or pseudograph) and P is a trail in G. P is called a ***traversable trail*** if P has the following properties:

- P includes **all vertices** of G
- P uses **each edge of G** (exactly once because P is a trail**)**.

Such a graph G is called a ***traversable graph***. ■

Corollary. Clearly a traversable multigraph (pseudograph) must be finite and connected.

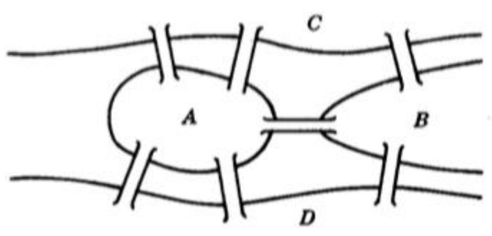


Figure 16 (a)

Konigsberg's bridges in 1736

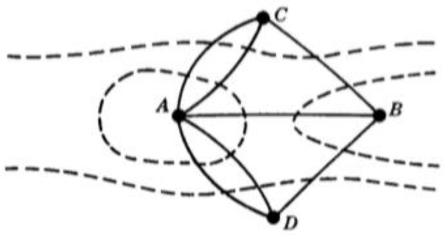
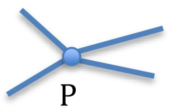


Figure 16 (b)

Multigraph Model of Königsberg.

We now show how Euler proved that the multigraph in Figure 16 (b) is not traversable and hence that the walk in Königsberg is impossible. Recall first that a vertex is even or odd according as its degree is an even or an odd number.

Assume the contrary. Suppose a multigraph is traversable and that a traversable trail does not begin or end at a vertex P. We claim that P is an even vertex. For whenever the traversable trail enters P by an edge, there must always be an edge not previously used by which the trail can leave P. Thus, the edges in the trail incident with P must appear in pairs, and so P is an even vertex. Therefore, if a vertex Q is odd, the traversable trail must begin or end at Q. Consequently, a multigraph with more than two odd vertices cannot be traversable. Observe that the multigraph corresponding to the bridge problem has four odd vertices. Thus, one cannot walk through Königsberg so that each bridge is crossed exactly once. ■



P

Euler proved the converse of the above statement, which is contained in the following theorem and corollary.

Definition 16: A graph G is called an *Eulerian* graph if there exists an ***Eulerian* circuit (= closed traversable trail**, or **closed *Eulerian* trail** or closed ***Eulerian*** path). ■

Theorem 6 (Euler): A finite connected graph is Eulerian **iff** each vertex has even degree.

Proof. Suppose G is Eulerian and T is a closed Eulerian trail (=Eulerian circuit). For any vertex v of G, the trail T enters and leaves v the same number of times without repeating any edge. Hence v has even degree.

Conversely, suppose that each vertex of G has even degree. We construct an Eulerian circuit. We begin a circuit T_1 at any edge e. We extend T_1 by adding one edge after the other. If T_1 is not closed at any step, say, T_1 begins at u but ends at $v \neq u$, then only an odd number of the edges incident on v appear in T_1; hence, since deg(v) is even, we can extend T_1 by another edge incident on v. Thus, we can continue to extend T_1 until T_1 returns to its initial vertex u, i.e., until T_1 is closed (circuit). Note than under this procedure T_1 contains an even number of the edges incident on any vertex T_1. Therefore, under this procedure so-called T_1 -degree [$=T_1$deg(v)] of any vertex T_1 is even.

If T_1 includes all the edges of G (and therefore T_1 includes all vertices because G is connected), then T_1 is our Eulerian circuit.

Suppose T_1 does not include all edges of G. Consider the graph H obtained by deleting all edges of T_1 from G. H may not be connected, but each vertex of H has even degree since T_1 contains an even number of the edges incident on any vertex T_1 (Hdeg(v)=Gdeg(v)- T_1deg(v)="even-even"). Since G is connected, there is an edge e′ of H which has an endpoint u′ in T_1. We construct a circuit T_2 in H beginning at u′ and using e′. Since all vertices in H have even degree, we can continue to extend T_2

in H until T_2 returns to u′ as pictured in Figure 17. We can clearly put T_1 and T_2 together to form a larger circuit (=closed trail) in G. We continue this process until all the edges of G are used. We finally obtain an Eulerian circuit, and so G is Eulerian. ■

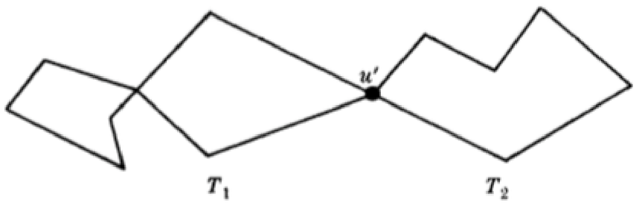


Figure 17

Corollary: Any finite connected graph with two odd vertices is traversable. A traversable trail may begin at either odd vertex and will end at the other odd vertex. ■

Hamiltonian Graphs

The above discussion of Eulerian graphs emphasized traveling edges; here we concentrate on visiting vertices.

Definition 17: A *Hamiltonian circuit* in a graph G is a **closed path** that **visits every vertex in G exactly once.** (Such a closed path must be a cycle.) If G does admit a Hamiltonian circuit, then G is called a *Hamiltonian graph*. ■

Note that an Eulerian circuit traverses every edge exactly once, but may repeat vertices, while a Hamiltonian circuit visits each vertex exactly once. Figure 18 gives an example of a graph which is Hamiltonian but not Eulerian, and vice versa.

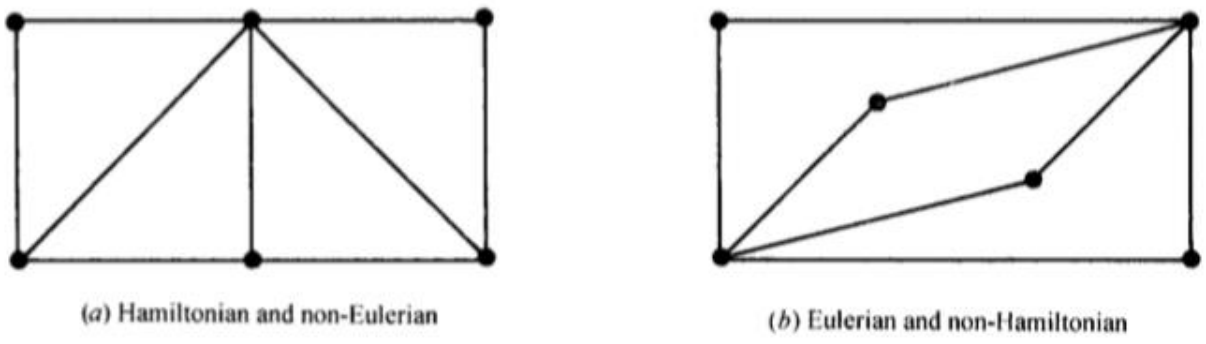

(a) Hamiltonian and non-Eulerian
(b) Eulerian and non-Hamiltonian

Figure 18

Although only connected graphs can be Hamiltonian, there is no simple criterion to tell us whether a graph is Hamiltonian as there is for Eulerian graphs. We do have the following sufficient condition which is due to G. A. Dirac.

Theorem 7: Let G be a connected graph with n vertices. Then G is Hamiltonian if $n \geq 3$ and $n \leq \deg(v)$ for each vertex v in G. ■

PART 4. SHORTEST PATHS IN WEIGHTED GRAPHS. TREE GRAPHS. SPANNING TREES.

Labeled and weighted graphs

Definition 18: A graph G is called a *labeled graph* if its edges and/or vertices are assigned data of one kind or another. In particular, G is called a *weighted graph* if each edge ***e*** of G is assigned a nonnegative number w(e) called *the weight* or *length* of ***e***. ■

Figure 19 shows a weighted graph where the weight of each edge is given in the obvious way. The *weight* (or *length*) of a path in such a weighted graph G is defined to be the sum of the weights of the edges in the path.

Important problem in graph theory is to find a *shortest path*, that is, a path of minimum length, between any two given vertices. The length of a shortest path between P and Q in Figure 19 is 14; one such path is (P, A_1, A_2, A_5, A_3, A_6, Q). You can try to find another shortest path.

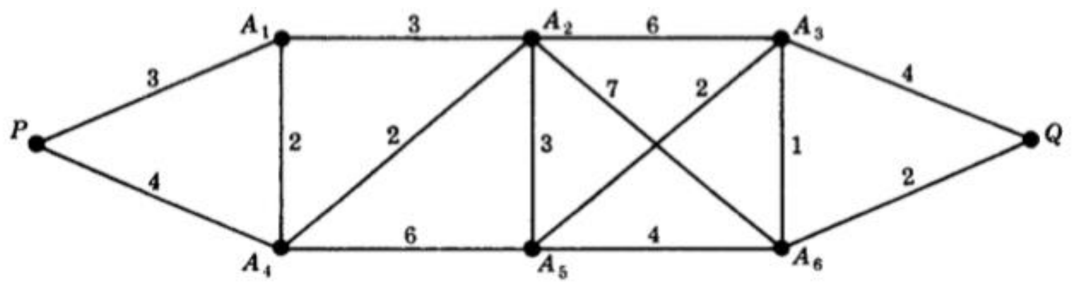


Figure 19

TREE GRAPHS

Definition 19. A graph T is called a *tree* if

- T is connected and
- T has no cycles (simple circuits).

A forest G is a graph with no cycles; hence the connected components of a forest G are trees. A graph without cycles is said to be *cycle-free*. The tree consisting of a single vertex with no edges is called the *degenerate tree*. ■

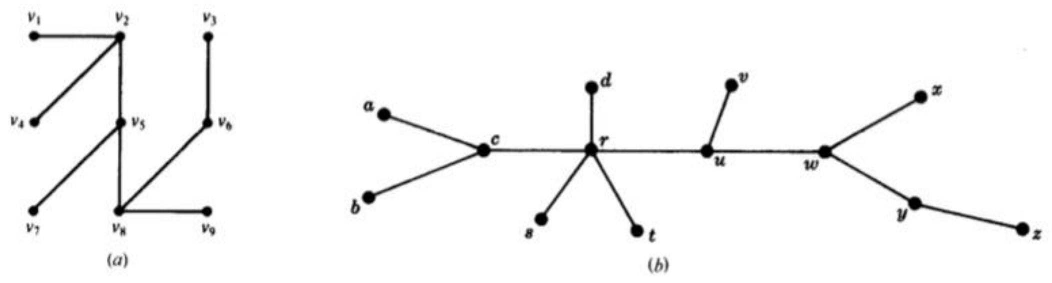
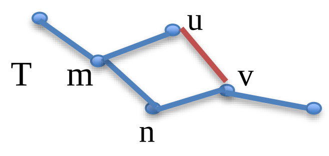


Figure 20. Trees

Consider a tree T. Clearly, there is only one simple path (no repeated vertices) between two vertices of T; otherwise, the two paths would form a cycle. Also:

. (a) Suppose that u and v are vertices in T and there is no edge{u v}in T. Since T is connected graph so there is a path from u to v. Let it will be the sequence: umnv. If we add the edge ***e***={u, v} to T then the simple path from u to v in T and ***e*** will form a cycle; hence T is no longer a tree.



. (b) On the other hand, suppose there is an edge ***e*** = {u, v} in T, and we delete ***e*** from T. Then T is no longer connected (since there cannot be a path from u to v); hence T is no longer a tree.

Lemma 1: Let G is a finite cycle-free graph with at least one edge. Show that G has at least two vertices of degree 1.

Proof. Exercise. (*Hint* : Consider a maximal simple path α, and show that its endpoints have degree 1). ■

Lemma 2: A connected graph G with n vertices must have at least n − 1 edges.

Proof. Exercise. ■

The following theorem applies when our graphs are finite.
Theorem 8: Let G be a graph with n>1 vertices. Then the following are equivalent:

(i) G is a tree, that is, connected and cycle-free.
(ii) G is a cycle-free and has n−1 edges.
(iii) G is connected and has n−1 edges.

Proof. The proof is by induction on n. The theorem is certainly true for the graph with only one vertex and hence no edges. That is, the theorem holds for n=1. We now assume that n>1 and that the theorem holds for graphs with less than n vertices.

(i) *implies* (ii).

Suppose G is a tree. Then G is cycle-free, so we only need to show that G has n−1 edges. By Lemma 1, G has a vertex of degree 1. Deleting this vertex and its edge, we obtain a tree T which has n−1 vertices. The theorem holds for T, so T has n−2 edges. Hence G has n−1 edges.

(ii) *implies* (iii)

Suppose G is cycle-free and has n−1 edges. We only need show that G is connected. Suppose G is disconnected and has k components, $T_1, ..., T_k$, which are trees since each is connected and cycle-free. Say T_i has n_i vertices. Note $n_i<n$. Hence the theorem holds for T_i , so T_i has $n_i−1$ edges. Thus,

$$n=n_1+n_2+\cdots+n_k$$

and

$$n−1=(n_1−1)+(n_2−1)+\cdots+(n_k−1)=n_1+n_2+\cdots+n_k−k=n−k$$

Hence k=1. But this contradicts the assumption that G is disconnected and has k>1 components. Hence G is connected.

(iii) *implies* (i)

Suppose G is connected and has n−1 edges. We only need to show that G is cycle-free. Assume the contrary. Let G has a cycle containing an edge ***e***. Deleting e we obtain the graph H=G−e which is also connected. But H has n vertices and n−2 edges, and this contradicts Lemma 2. Thus, G is cycle-free and hence is a tree. ■

Theorem 8 also tells us that **a finite tree T with n vertices must have (n–1) edges**. For example, the tree in Figure 20(a) has 9 vertices and 8 edges, and the tree in Figure 20(b) has 13 vertices and 12 edges.

Another equivalent definition of a tree comes from the next theorem 9.

Theorem 9: Let G be an undirected graph. The following conditions are equivalent:

- G is a tree
- There is a unique simple path between any two of its vertices.

Proof. Exercise.

Spanning Trees.

Definition 20. A subgraph T of a connected graph G is called a *spanning tree* of G if:

1). T is a tree, and

2). T includes all the vertices of G. ■

Figure 21 shows a connected graph G and spanning trees T_1, T_2, and T_3 of G.

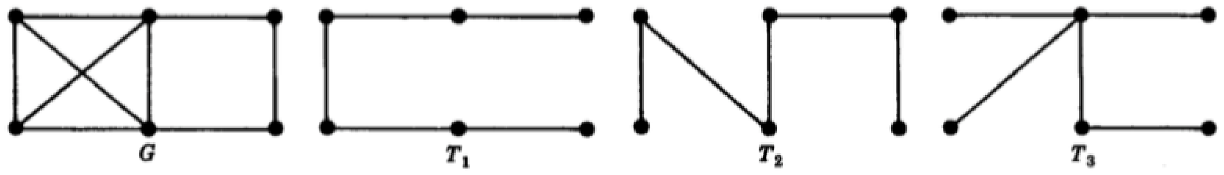


Figure 21. Connected graph G and spanning trees T_1, T_2, and T_3 of G

Minimum Spanning Trees

Suppose G is a connected **weighted** graph. That is, each edge of G is assigned a nonnegative number called the *weight* of the edge. Then any spanning tree T of G is assigned a total weight obtained by adding the weights of the edges in T. A *minimal spanning tree* of G is a spanning tree whose total weight is as small as possible.

Algorithms 1, 2 and 3 below enable us to find a minimal spanning tree T of a connected weighted graph G where G has n vertices (T must have (n–1) edges.) ■

Algorithm 1 (Kruskal):

- **Step 1.** Arrange the edges of G in the order of increasing weights
- **Step 2.** Successively add edges with minimum weight that do not form a simple circuit with those edges already chosen. Stop after n−1 edges have been selected. ■

Algorithm 2 (Prim):

- **Step 1.** Begin by choosing any edge with smallest weight
- **Step 2.** Successively add to the tree edges of minimum weight that are incident to a vertex already in the tree, never forming a simple circuit with those edges already in the tree. Stop when (n−1) edges have been added. ■

Algorithm 3 (Kruskal: reverse-delete algorithm).

- **Step 1.** Arrange the edges of G in the order of decreasing weights
- **Step 2.** Proceeding sequentially, delete each edge that does not disconnect the graph until n-1 edges remain. ■

The weight of a minimal spanning tree is unique, but the minimal spanning tree itself is not. Different minimal spanning trees can occur when two or more edges have the same weight. In such a case, the arrangement of the edges in Step 1 of Algorithms 1 or 3 is not unique and hence may result in different minimal spanning trees as illustrated in the following example.

EXAMPLE 15. Find a minimal spanning tree of the weighted graph Q in Figure 22(a). Note that Q has six vertices, so a minimal spanning tree will have five edges.

Solution.

1. **Reverse-delete algorithm.**

First we order the edges by decreasing weights, and then we successively delete edges **without disconnecting Q** until five edges remain. This yields the following data:

Edges:	BC	AF	AC	BE	CE	BF	AE	DF	BD
Weight:	8	7	7	7	6	5	4	4	3
Delete?:	Yes	Yes	Yes	No	No	Yes			

Thus, the minimal spanning tree of Q which is obtained contains the edges BE, CE, AE, DF, BD. The spanning tree has weight 24 and it is shown in Figure 22(b). ■

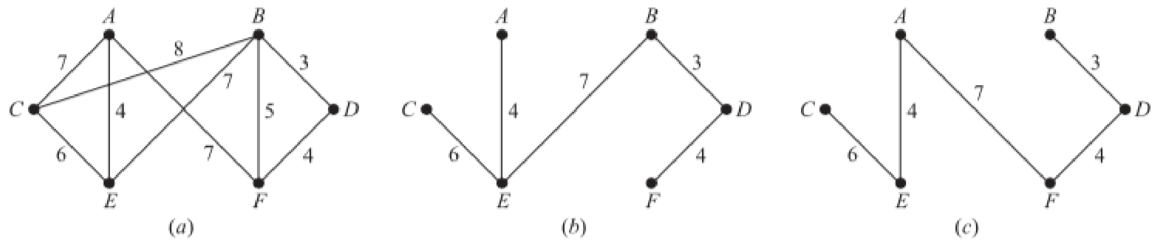


Figure 22

2. **Kruskal algorithm.**

First we order the edges by increasing weights, and then we successively add edges **without forming any cycles** until five edges are included. This yields the following data:

Edges:	BD	AE	DF	BF	CE	AC	AF	BE	BC
Weight:	3	4	4	5	6	7	7	7	8
Add?	Yes	Yes	Yes	No	Yes	No	Yes		

Thus the minimal spanning tree of Q which is obtained contains the edges BD, AE, DF, CE, AF. The spanning tree appears in Figure 22(c). Observe that this spanning tree is not the same as the one obtained using Algorithm 1 as expected it also has weight 24. ■

Remark: The above algorithms are easily executed when the graph G is relatively small. Suppose G has dozens of vertices and hundreds of edges which, say, are given by a list of pairs of vertices. Then even deciding whether G is connected is not obvious; it may require some type of depth-first search (DFS) or breadth-first search (BFS) graph algorithm. ■

3. **Algorithm 3 (Prim).**

Choice	**Edge**	**Weight**
1	BD	3
2	DF	4
3	FA (or BE)	7
4	AE	4
5	EC	6

As expected, length on minimal spanning tree is equal again 24

EXAMPLE 16. Use Prim's algorithm to find a minimum spanning tree in the graph in Figure 23.

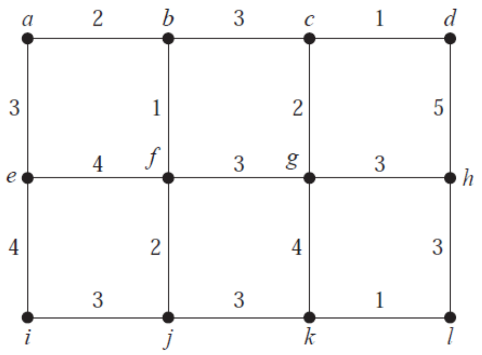
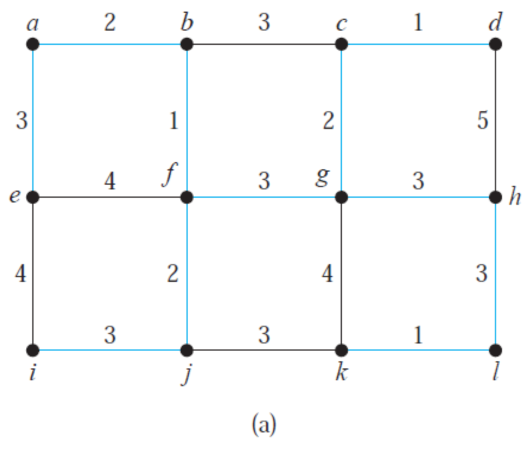


Figure 23. Weighted Graph.

Solution. Minimum spanning tree constructed using Prim's algorithm is shown in Figure 24. The successive edges chosen are displayed.



Choice	Edge	Weight
1	$\{b, f\}$	1
2	$\{a, b\}$	2
3	$\{f, j\}$	2
4	$\{a, e\}$	3
5	$\{i, j\}$	3
6	$\{f, g\}$	3
7	$\{c, g\}$	2
8	$\{c, d\}$	1
9	$\{g, h\}$	3
10	$\{h, l\}$	3
11	$\{k, l\}$	1
	Total:	24

(b)

FIGURE 24 A Minimum Spanning Tree Produced Using Prim’s Algorithm.

Successively add edges with minimum weight that do not form a simple circuit with those edges already chosen. Stop after (n-1) edges have been selected.
The edges in color in Figure 24 show a minimum spanning tree produced by Prim’s algorithm, with the choice made at each step displayed. ■

Pseudocode for Kruskal’s algorithm.

```
procedure Kruskal(G: weighted connected undirected graph with n vertices)
T := empty graph
for i := 1 to n − 1
    e := any edge in G with smallest weight that does not form a simple circuit when added to T
    T := T with e added
return T {T is a minimum spanning tree of G}
```
■

Pseudocode for Prim’s algorithm.

```
procedure Prim(G: weighted connected undirected graph with n vertices)
T := a minimum-weight edge
for i := 1 to n − 2
    e := an edge of minimum weight incident to a vertex in T and not forming a simple circuit in
         T if added to T
    T := T with e added
return T {T is a minimum spanning tree of G}
```
■

Pseudocode for Reverse-delete algorithm.

Exercise. ■

Theorem 10: Prim’s algorithm produces a minimum spanning tree.

Proof. Let G be a connected weighted graph. Suppose that the successive edges chosen by Prim’s algorithm are $e_1, e_2, ..., e_{n-1}$. Let S be the tree with $e_1, e_2, ..., e_{n-1}$ as its edges. **Our goal is to prove that S is the minimum spanning of G.**

Let S_K be the tree with $e_1, e_2, ..., e_k$ as its edges. Let T be a minimum spanning tree of G containing the edges $e_1, e_2, ..., e_k$, where *k* is the maximum integer with the property that a minimum spanning tree exists containing the first k edges chosen by Prim’s algorithm. **The theorem follows if we can show that S=T.**

Assume the contrary. Suppose that $S \neq T$, so that $k < n-1$. Consequently, T contains $e_1, e_2, \ldots, e_k$, **but not e_{k+1}.** Consider the graph made up of T together with e_{k+1}. Because this graph is connected and has n edges, too many edges to be a tree, it must contain a simple circuit. This simple circuit must contain e_{k+1} because there was no simple circuit in T. Furthermore, there must be an edge in the simple circuit that does not belong to S_{K+1} because S_{K+1} is a tree. By starting at an endpoint of e_{k+1} that is also an endpoint of one of the edges $e_1, e_2, \ldots, e_k$, and following the circuit until it reaches an edge not in S_{K+1}, we can find an edge e not in S_{K+1} that has an endpoint that is also an endpoint of one of the edges $e_1, e_2, \ldots, e_k$.
By deleting e from T and adding e_{k+1}, we obtain a tree T_1 with $n-1$ edges (it is a tree because it has no simple circuits). Note that the tree T_1 contains $e_1, e_2, \ldots, e_k, e_{k+1}$. Furthermore, because e_{k+1} was chosen by Prim's algorithm at the (k+1)st step, and e was also available at that step, the weight of e_{k+1} is less than or equal to the weight of *e*. From this observation, it follows that T_1 is also a minimum spanning tree, because the sum of the weights of its edges does not exceed the sum of the weights of the edges of T. This contradicts the choice of k as the maximum integer such that a minimum spanning tree exists containing $e_1, e_2, \ldots, e_k$. Hence, $k=n-1$, and $S=T$. It follows that Prim's algorithm produces a minimum spanning tree. ■

Theorem 11: Kruskal's algorithm produces minimum spanning trees.

Proof. Exercise. ■

It can be shown that to find a minimum spanning tree of a graph with m edges and n vertices, Kruskal's algorithm can be carried out using O(mlogm) operations and Prim's algorithm can be carried out using O(mlogn) operations. Consequently, it is preferable to use Kruskal's algorithm for graphs that are **sparse**, that is, where m is very small compared to $C(n, 2)=n(n-1)/2$, the total number of possible edges in an undirected graph with n vertices. Otherwise, there is little difference in the complexity of these two algorithms.

Shortest Path Algorithm (Dijkstra's algorithm)

We will now consider the general problem of finding the length of a shortest path between a vertices, say *a* and *z*, in an undirected connected simple weighted graph. **Dijkstra's algorithm proceeds by finding the length of a shortest path from *a* to a first vertex, the length of a shortest path from *a* to a second vertex, and so on, until the length of a shortest path from *a* to *z* is found.** As a side benefit, this algorithm is easily extended to find the length of the shortest path from *a* to all other vertices of the graph, and not just to *z*.

The algorithm relies on a series of iterations.

1. *A **distinguished set of vertices** is constructed by adding one vertex at each iteration.*
2. *A **labeling procedure** is carried out at each iteration. In this labeling procedure, a vertex **w is labeled with the length of a shortest path from a to w** that contains only vertices already in the distinguished set.*
3. *The vertex added to the distinguished set is one with a minimal label among those vertices not already in the set.*

We now give the details of Dijkstra's algorithm.

- It begins by labeling *a* with 0 and the other vertices with ∞. We use the notation $L_0(a)=0$ and $L_0(v)=\infty$ for these labels before any iterations have taken place (the subscript 0 stands for the "0th" iteration). These labels are the lengths of shortest paths from *a* to the vertices, where the paths contain only the vertex *a*. (Because no path from *a* to a vertex different from *a* exists, ∞ is the length of a shortest path between *a* and this vertex.)
- Dijkstra's algorithm proceeds by forming a distinguished set of vertices. Let S_k denote this set after k iterations of the labeling procedure. We begin with $S_0=\emptyset$. The set S_k is formed from S_{k-1}

by adding a vertex u not in S_{k-1} with the smallest label.

- Once u is added to S_k, we update the labels of all vertices not in S_k, so that **$L_k(v)$, the label of the vertex v at the kth stage, is the length of a shortest path from a to v that contains vertices only in** S_k (that is, vertices that were already in the distinguished set together with u). Note that the way we choose the vertex u to add to S_k at each step is an optimal choice at each step, making this a greedy algorithm. (We will prove shortly that this greedy algorithm always produces an optimal solution.)

Let v be a vertex not in S_k. To update the label of v, note that $L_k(v)$ is the length of a shortest path from a to v containing only vertices in S_k. The updating can be carried out efficiently when this observation is used: **A shortest path from a to v containing only elements of S_k is either a shortest path from a to v that contains only elements of S_{k-1} (that is, the distinguished vertices not including u), or it is a shortest path from a to u at the (k−1)st stage with the edge {u, v} added.** In other words,

$$L_k(a, v) = \min\{ L_{k-1}(a, v), L_{k-1}(a, u) + w(u, v)\},$$

where $w(u, v)$ is the length of the edge with u and v as endpoints.

This procedure is iterated by successively adding vertices to the distinguished set **S** until z is added. When z is added to **S**, its label is the length of a shortest path from a to z. ■

EXAMPLE 17. Use Dijkstra's algorithm to find the length of a shortest path between the vertices a and z in the weighted graph displayed in Figure 25(a) below.

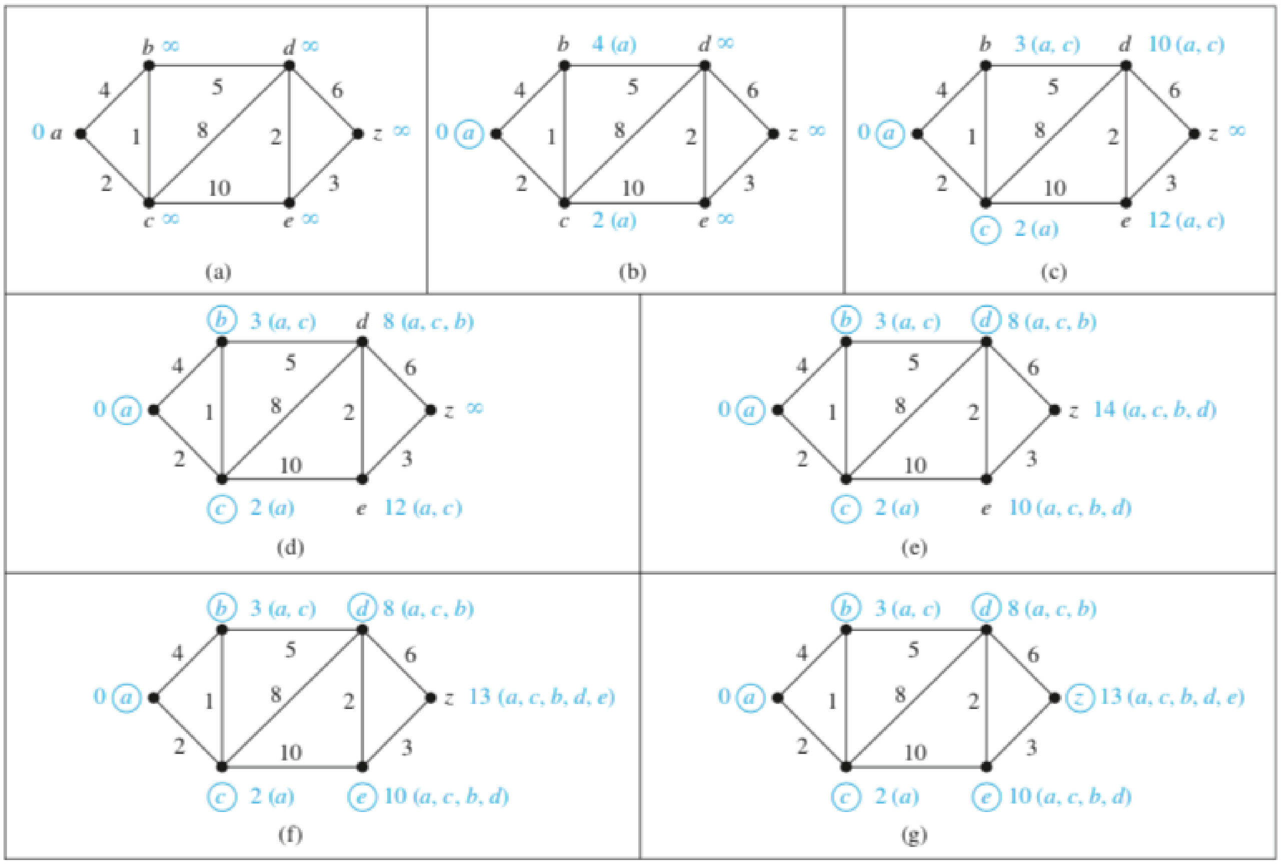


Figure 25. Using Dijkstra's Algorithm to Find a Shortest Path from a to z.

Solution: The steps used by Dijkstra's algorithm to find a shortest path between a and z are shown in Figure 25. At each iteration of the algorithm the vertices of the set S_k are circled. A shortest path from a to each vertex containing only vertices in S_k is indicated for each iteration. The algorithm terminates when z is circled. We find that a shortest path from a to z is a, c, b, d, e, z, with length 13. ■

Remark: In performing Dijkstra's algorithm it is sometimes more convenient to keep track of labels of vertices in each step using a table instead of redrawing the graph for each step.

Table. Using Dijkstra's Algorithm to Find a Shortest Path from *a* to *z* .

vertices / iteration	a	b	c	d	e	z	S - distinguished set
k=	$\boxed{L_k(a)}$	$\boxed{L_k(b)}$	$\boxed{L_k(c)}$	$\boxed{L_k(d)}$	$\boxed{L_k(e)}$	$\boxed{L_k(z)}$	S0=∅
0	0	∞	∞	∞	∞	∞	S1={a}
1		4	2	∞	∞	∞	S2={a,c}
2		3		10	12	∞	S3={a,c,b}
3				8	12	∞	S4={a,c,b,d}
4					10	14	S5={a,c,b,d,e}
5						13	S6={a,c,b,d,e,z}

ALGORITHM 1 Dijkstra's Algorithm.

```
procedure Dijkstra(G: weighted connected simple graph, with all weights positive)
{G has vertices a=$v_0$, $v_1$, ... , $v_n$ = z and lengths $w(v_i,v_j)$ where $w(v_i,v_j)=\infty$ if {$v_i,v_j$} is not an edge in G}
for i:=1 to n
    L($v_i$):=∞
L(a):= 0
S:= ∅
{the labels are now initialized so that the label of a is 0 and all other labels are ∞, and S={∅} }
while z∉S
    u:= a vertex not in S with L(u) minimal
    S:= S∪{u}
    for all vertices v not in S
        if L(u) + w(u, v) < L(v) then L(v) := L(u) + w(u, v)
        {this adds a vertex to S with minimal label and updates the labels of vertices not in S}
return L(z) {L(z) = length of a shortest path from a to z}                ■
```

Theorem 12: Dijkstra's algorithm finds the length of a shortest path between two vertices in a connected simple undirected weighted graph.

Proof. We use an inductive argument to show that Dijkstra's algorithm produces the length of a shortest path between two vertices *a* and *z* in an undirected connected weighted graph. Take as the inductive hypothesis the following assertion: At the ***kth*** iteration

(*i*) the label of every vertex *v* in S is the length of a shortest path from a to this vertex, and

(*ii*) the label of every vertex not in S is the length of a shortest path from a to this vertex that contains only (besides the vertex itself) vertices in S.

When k=0, before any iterations are carried out, S=∅, so the length of a shortest path from a to a vertex other than a is ∞. Hence, the basis case is true.

Assume that the inductive hypothesis holds for the kth iteration. Let *v* be the vertex added to S at the (k+1)st iteration, so v is a vertex not in S at the end of the kth iteration with the smallest label (in the case of ties, any vertex with smallest label may be used).

From the inductive hypothesis we see that the vertices in S before the (k+1)st iteration are labeled with the length of a shortest path from a. Also, v must be labeled with the length of a shortest path to it from a. If this were not the case, at the end of the kth iteration there would be a path of length less than $L_k(v)$ containing a vertex not in S [because $L_k(v)$ is the length of a shortest path from a to v containing only vertices in S after the kth iteration]. Let u be the first vertex not in S in such a path. There is a path with length less than $L_k(v)$ from a to u containing only vertices of S. This contradicts the choice of v. Hence, (i) holds at the end of the (k+1)st iteration.

Let u be a vertex not in S after k+1 iterations. A shortest path from a to u containing only elements of S either contains v or it does not. If it does not contain v, then by the inductive hypothesis its length is $L_k(u)$. If it does contain v, then it must be made up of a path from a to v of shortest possible length containing elements of S other than v, followed by the edge from v to u. In this case, its length would be $L_k(v)+w(v, u)$.

This shows that (*ii*) is true, because

$$L_{k+1}(u)=\min\{L_k(u), L_k(v)+w(v, u)\}. \qquad \blacksquare$$

PART 5. KEY TERMS AND RESULTS.

TERMS

undirected edge: an edge associated to a set {u, v}, where u and v are vertices
directed edge: an edge associated to an ordered pair (u, v), where u and v are vertices
multiple edges: distinct edges connecting the same vertices
multiple directed edges: distinct directed edges associated with the same ordered pair (u, v), where u and v are vertices
loop: an edge connecting a vertex with itself
undirected graph: a set of vertices and a set of undirected edges each of which is associated with a set of one or two of these vertices
simple graph: an undirected graph with no multiple edges or loops
multigraph: an undirected graph that may contain multiple edges but no loops
pseudograph: an undirected graph that may contain multiple edges and loops
directed graph: a set of vertices together with a set of directed edges each of which is associated with an ordered pair of vertices
directed multigraph: a graph with directed edges that may contain multiple directed edges
simple directed graph: a directed graph without loops or multiple directed edges
adjacent: two vertices are adjacent if there is an edge between them
incident: an edge is incident with a vertex if the vertex is an endpoint of that edge
deg *v* (degree of the vertex *v* in an undirected graph): the number of edges incident with v with loops counted twice
deg−(*v*) (the in-degree of the vertex *v* in a graph with directed edges): the number of edges with v as their terminal vertex
deg+ (*v*) (the out-degree of the vertex *v* in a graph with directed edges): the number of edges with v as their initial vertex
underlying undirected graph of a graph with directed edges: the undirected graph obtained by ignoring the directions of the edges
isolated vertex: a vertex of degree zero
pendant vertex: a vertex of degree one
regular graph: a graph where all vertices have the same degree
subgraph of a graph *G* = (V, E): a graph (W, F), where W is a subset of V and F is a subset of E
$G_1 \cup G_2$ (union of G_1 and G_2): the graph $(V_1 \cup V_2, E_1 \cup E_2)$, where $G_1=(V_1, E_1)$ and $G_2=(V_2, E_2)$
adjacency matrix: a matrix representing a graph using the adjacency of vertices
incidence matrix: a matrix representing a graph using the incidence of edges and vertices

isomorphic simple graphs: simple graphs G_1=(V_1, E_1) and G_2=(V_2, E_2) are isomorphic if there exists a 1-1 correspondence *f* from V_1 to V_2 such that $\{f(v_1), f(v_2)\} \in E_2$ if and only if $\{v_1, v_2\} \in E_1$ for all v_1 and v_2 in V_1.
invariant for graph isomorphism: a property that isomorphic graphs either both have or both do not have
path from *u* to *v* in an undirected graph: a sequence of edges e_1, e_2, ..., e_n, where e_i is associated to $\{x_i, x_{i+1}\}$ for i=0, 1, . . . , n, where x_0=u and x_{n+1}=v
path from *u* to *v* in a graph with directed edges: a sequence of edges e_1, e_2, ..., e_n, where e_i is associated to (x_i, x_{i+1}) for i=0, 1, . . . , n, where x_0=u and x_{n+1}=v
simple path: a path in which all vertices are distinct
cycle: a closed path (v_1, v_2, ..., v_n) in which all vertices are distinct besides v_1=v_n
trail: a path in which all edges are distinct; clear that each simple path is a trail but not vice versa.
circuit: a path of length $n \geq 1$ that begins and ends at the same vertex
connected graph: an undirected graph with the property that there is a path between every pair of vertices
cut vertex of *G*: a vertex v such that $G - v$ is disconnected
cut edge of *G*: an edge e such that $G - e$ is disconnected
non-separable graph: a graph without a cut vertex
connected component of a graph *G*: a maximal connected subgraph of G
Euler path: a path that contains every edge of a graph exactly once
Euler circuit: a circuit that contains every edge of a graph exactly once
Hamilton path: a path in a graph that passes through each vertex exactly once
Hamilton circuit: a circuit in a graph that passes through each vertex exactly once
weighted graph: a graph with numbers assigned to its edges
shortest-path problem: the problem of determining the path in a weighted graph such that the sum of the weights of the edges in this path is a minimum over all paths between specified vertices
traveling salesperson problem: the problem that asks for the circuit of shortest total length that visits every vertex of a weighted graph exactly once
planar graph: a graph that can be drawn in the plane with no crossings
elementary subdivision: the removal of an edge $\{u, v\}$ of an undirected graph and the addition of a new vertex w together with edges $\{u, w\}$ and $\{w, v\}$
homeomorphic: two undirected graphs are homeomorphic if they can be obtained from the same graph by a sequence of elementary subdivisions
tree: a connected and cycle-free undirected graph.
forest: an undirected graph with no simple circuits
rooted tree: a directed graph with a specified vertex, called the root, such that there is a unique path to every other vertex from this root
subtree: a subgraph of a tree that is also a tree
parent of *v* in a rooted tree: the vertex u such that (u, v) is an edge of the rooted tree
child of a vertex *v* in a rooted tree: any vertex with v as its parent
sibling of a vertex *v* in a rooted tree: a vertex with the same parent as v
ancestor of a vertex *v* in a rooted tree: any vertex on the path from the root to v
descendant of a vertex *v* in a rooted tree: any vertex that has v as an ancestor
internal vertex: a vertex that has children
leaf: a vertex with no children
level of a vertex: the length of the path from the root to this vertex
height of a tree: the largest level of the vertices of a tree
***m*-ary tree:** a tree with the property that every internal vertex has no more than m children
full *m*-ary tree: a tree with the property that every internal vertex has exactly m children
binary tree: an m-ary tree with m = 2 (each child may be designated as a left or a right child of its parent)
ordered tree: a tree in which the children of each internal vertex are linearly ordered

balanced tree: a tree in which every leaf is at level h or $h-1$, where h is the height of the tree
binary search tree: a binary tree in which the vertices are labeled with items so that a label of a vertex is greater than the labels of all vertices in the left subtree of this vertex and is less than the labels of all vertices in the right subtree of this vertex
decision tree: a rooted tree where each vertex represents a possible outcome of a decision and the leaves represent the possible solutions of a problem
spanning tree: a tree containing all vertices of a graph
minimum spanning tree: a spanning tree with smallest possible sum of weights of its edges.

RESULTS

The handshaking theorem: If G = (V , E) be an undirected graph with m edges. Then $2m=\sum_{v\in V} \deg(v)$
Dijkstra's algorithm: a procedure for finding a shortest path between two vertices in a weighted graph.
Euler's formula: $r = e - v + 2$ where r, e, and v are the number of regions of a planar representation, the number of edges, and the number of vertices, respectively, of a connected planar graph.
Tree graph.
Let G be a graph with n vertices, n>0. Satisfying any two of the following three conditions implies the third one:

- G is connected
- G is cycle-free
- G has exactly (n-1) edges

A graph G is a tree if and only if G is connected and has exactly (n-1) edges or equivalently, G is a cycle free and has exactly (n-1) edges.
A graph G is a tree if and only if there is a unique simple path between every pair of its vertices.
Prim's algorithm: a procedure for producing a minimum spanning tree in a weighted graph that successively adds edges with minimal weight among all edges incident to a vertex already in the tree so that no edge produces a simple circuit when it is added
Kruskal's algorithm: a procedure for producing a minimum spanning tree in a weighted graph that successively adds edges of least weight that are not already in the tree such that no edge produces a simple circuit when it is added