## 1.2 BASIC CONCEPTS

A *graph* $G$ consists of a pair $(V, E)$ where $V$ is a nonempty finite set whose elements are called *vertices* and $E$ is a set of unordered pairs of distinct elements of $V$. The elements of $E$ are called *edges* of the graph $G$. If $e = \{u, v\} \in E$, the edge $e$ is said to join $u$ and $v$. The vertices $u$ and $v$ are called the end vertices of the edge $uv$. We write $e = uv$ and we say that the vertices $u$ and $v$ are *adjacent*. We also say that the vertex $u$ and the edge $e$ are *incident* with each other. If two distinct edges $e_1$ and $e_2$ are incident with a common vertex, then they are called *adjacent edges*. A graph with $n$ vertices and $m$ edges is called a $(n, m)$ *graph*. The number of vertices in $G$ is called the *order* of $G$. The number of edges of $G$ is called the *size* of $G$.

A graph is normally represented by a diagram in which each vertex is represented by a dot and each edge is represented by a line segment joining two vertices with which the edge is incident. For example, if $G = (V, E)$ is a graph where $V = \{a, b, c, d\}$ and $E = \{ab, ac, ad\}$, then $G$ is a $(4, 3)$ graph and it is represented by the diagram given in Figure 1.1.

The definition of a graph does not allow more than one edge joining two vertices. It also does not allow any edge joining a vertex to itself. Such an edge joining a vertex to itself is called a *self-loop* or simply a *loop*. If more than one edge joining two vertices are allowed, the resulting object is called a *multigraph*. Edges joining the same pair of vertices are called *multiple edges*. If loops are also allowed, the resulting object is called a *pseudo graph*.

A graph in which any two distinct vertices are adjacent is called a *complete graph*. The complete graph on $n$ vertices is denoted by $K_n$. A graph whose edge set is empty is called a *null graph* or a *totally disconnected graph*.

A graph $G$ is called a *bipartite graph* if $V$ can be partitioned into two disjoint subsets $V_1$ and $V_2$ such that every edge of $G$ joins a vertex of $V_1$ to a vertex of $V_2$ and $(V_1, V_2)$ is called a *bipartition* of $G$. If $G$ contains every edge joining the vertices of $V_1$ to the vertices of $V_2$ then $G$ is called a *complete bipartite graph*. If $V_1$ contains $r$ vertices and $V_2$ contains $s$ vertices, then the complete bipartite graph $G$ is denoted by $K_{r,s}$. The graph given in Figure 1.1 is $K_{1,3}$.

A graph $G$ is *k-partite*, $k \geq 1$, if it is possible to partition $V(G)$ into $k$ subsets $V_1, V_2, \ldots, V_k$ (called *partite sets*) such that every element of $E(G)$ joins a vertex of $V_i$ to a vertex of $V_j, i \neq j$.

A *complete k-partite graph* $G$ is a $k$-partite graph with partite sets $V_1, V_2, \ldots, V_k$ having the additional property that if $u \in V_i$ and $v \in V_j, i \neq j$, then $uv \in E(G)$. If $|V_i| = n_i$, then this graph is denoted by $K(n_1, n_2, \ldots, n_k)$ or $K_{n_1,n_2,..,n_k}$.

The *degree* of a vertex $v_i$ in a graph $G$ is the number of edges incident with $v_i$. The degree of $v_i$ is denoted by $d_G(v_i)$ or *deg* $v_i$ or simply $d(v_i)$. A vertex $v$ of degree 0 is called an *isolated vertex*. A vertex $v$ of degree 1 is called a *pendant vertex*.

**Theorem 1.1** *The sum of the degrees of the vertices of a graph $G$ is twice the number of edges.* ■

**Corollary 1.1** *In any graph $G$ the number of vertices of odd degree is even.* ■

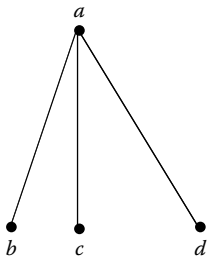


Figure 1.1 Example of a graph.

For any graph $G$, we define

$$\delta(G) = \min\{deg\ v : v \in V(G)\}$$

and

$$\Delta(G) = \max\{deg\ v : v \in V(G)\}.$$

If all the vertices of $G$ have the same degree $r$, then $\delta(G) = \Delta(G) = r$ and in this case $G$ is called a *regular graph* of degree $r$. A regular graph of degree 3 is called a *cubic graph*.

## 1.3 SUBGRAPHS AND COMPLEMENTS

A subgraph of a graph $G$ is a graph $H$ such that $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$ and the assignment of end vertices to edges in $H$ is the same as in $G$. We then write $H \subseteq G$ and say that $G$ *contains* $H$. The graph $H$ is called a proper subgraph of $G$ if either $E(H)$ is a proper subset of $E(G)$ or $V(H)$ is a proper subset of $V(G)$. Also $H$ is called a *spanning subgraph* of $G$ if $V(H) = V(G)$. If $V_1 \subset V$, then the subgraph $G_1 = (V_1, E_1)$ is called an *induced subgraph* of $G$ if $G_1$ is the maximal subgraph of $G$ with vertex set $V_1$. Thus, if $G_1$ is an induced subgraph of $G$, then two vertices in $V_1$ are adjacent in $G_1$ if and only if they are adjacent in $G$. The subgraph induced by $V_1$ is denoted by $\langle V_1 \rangle$ or $G[V_1]$. It is also called a vertex-induced subgraph of $G$.

If $E_1 \subset E$, then the subgraph of $G$ with edge set $E_1$ and having no isolated vertices is called the subgraph induced by $E_1$ and is denoted by $G[E_1]$. This is also called *edge-induced subgraph* of $G$.

Let $G = (V, E)$ be a graph. Let $v_i \in V$. The subgraph of $G$ obtained by removing the vertex $v_i$ and all the edges incident with $v_i$ is called the *subgraph obtained by the removal of the vertex* $v_i$ and is denoted by $G - v_i$. Clearly $G - v_i$ is an induced subgraph of $G$. If $G$ is connected $S \subset V$ and $G - S$ is not connected, then $S$ is called a vertex cut of $G$.

Let $e_j \in E$. Then $G - e_j = (V, E - \{e_j\})$ is called the subgraph of $G$ obtained by the removal of the edge $e_j$. Clearly $G - e_j$ is a spanning subgraph of $G$ which contains all the edges of $G$ except $e_j$.

Let $G = (V, E)$ be a graph. Let $v_i, v_j$ be two vertices which are not adjacent in $G$. Then $G + v_iv_j = (V, E \cup \{v_i, v_j\})$ is called the graph obtained by the *addition of the edge* $v_iv_j$ to $G$.

Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are said to be *isomorphic* if there exists a bijection $f : V_1 \to V_2$ such that $u, v$ are adjacent in $G_1$ if and only if $f(u), f(v)$ are adjacent in $G_2$. If $G_1$ is isomorphic to $G_2$, we write $G_1 \cong G_2$. The map $f$ is called an *isomorphism* from $G_1$ to $G_2$.

Let $f$ be an isomorphism of the graph $G_1 = (V_1, E_1)$ to the graph $G_2 = (V_2, E_2)$. Let $v \in V_1$. Then $deg\ v = deg\ f(v)$.

An isomorphism of a graph $G$ onto itself is called an *automorphism* of $G$. Let $\Gamma(G)$ denote the set of all automorphisms of $G$. Clearly the identity map $i : V \to V$ defined by $i(v) = v$ is an automorphism of $G$ so that $i \in \Gamma(G)$. Further if $\alpha$ and $\beta$ are automorphisms of $G$ then $\alpha\beta$ and $\alpha^{-1}$ are also automorphisms of $G$. Hence $\Gamma(G)$ is a group and is called the *automorphism group* of $G$.

Let $G = (V, E)$ be a graph. The *complement* $\overline{G}$ of $G$ is defined to be the graph which has $V$ as its set of vertices and two vertices are adjacent in $\overline{G}$ if and only if they are not adjacent in $G$. The graph $G$ is said to be a *self-complementary graph* if $G$ is isomorphic to $\overline{G}$.

## 1.4 CONNECTEDNESS

A *walk* of a graph $G$ is an alternating sequence of vertices and edges $v_0, x_1, v_1, x_2, v_2, \ldots, v_{n-1}, x_n, v_n$ beginning and ending with vertices such that each edge $x_i$ is incident with $v_{i-1}$ and $v_i$.

We say that the walk joins $v_0$ and $v_n$ and it is called a $v_0 - v_n$ walk. Also $v_0$ is called the *initial vertex* and $v_n$ is called the *terminal vertex* of the walk. The above walk is also denoted by $v_0, v_1, \ldots, v_n$, the edges of the walk being self evident. The number of edges in the walk is called the length of the walk.

A single vertex is considered as a walk of length 0. A walk is called a *trail* if all its edges are distinct and is called a *path* if all its vertices are distinct. A graph consisting of a path with $n$ vertices is denoted by $P_n$. A $v_0 - v_n$ walk is called *closed* if $v_0 = v_n$. A closed walk $v_0, v_1, v_2, \ldots, v_n = v_0$ in which $n \geq 3$ and $v_0, v_1, \ldots, v_{n-1}$ are distinct is called a *cycle* (or *circuit*) of length $n$. The graph consisting of a cycle of length $n$ is denoted by $C_n$.

Two vertices $u$ and $v$ of a graph $G$ are said to be *connected* if there exists a $u - v$ path in $G$. A graph $G$ is said to be *connected* if every pair of its vertices are connected. A graph which is not connected is said to be *disconnected*.

It is an easy exercise to verify that connectedness of vertices is an equivalence relation on the set of vertices $V$. Hence $V$ can be partitioned into nonempty subsets $V_1, V_2, \ldots, V_n$ such that two vertices $u$ and $v$ are connected if and only if both $u$ and $v$ belong to the same set $V_i$.

Let $G_i$ denote the induced subgraph of $G$ with vertex set $V_i$. Clearly the subgraphs $G_1, G_2, \ldots, G_n$ are connected and are called the *components* of $G$. Clearly a graph $G$ is connected if and only if it has exactly one component.

For any two vertices $u, v$ of a graph we define the *distance* between $u$ and $v$ by

$$d(u,v) = \begin{cases} \text{the length of a shortest } u - v \text{ path if such a path exists} \\ \infty \text{ otherwise.} \end{cases}$$

If $G$ is a connected graph, then $d(u,v)$ is always a nonnegative integer. In this case $d$ is actually a *metric* on the set of vertices $V$.

**Theorem 1.2** *A graph $G$ with at least two vertices is bipartite if and only if all its cycles are of even length.* ■

A *cutvertex* of a graph $G$ is a vertex whose removal increases the number of components. A *bridge* of a graph $G$ is an edge whose removal increases the number of components. Clearly if $v$ is a cutvertex of a connected graph, then $G - v$ is disconnected.

**Theorem 1.3** *Let $v$ be a vertex of a connected graph $G$. The following statements are equivalent:*

1. *$v$ is a cutvertex of $G$.*
2. *There exists a partition of $V - \{v\}$ into subsets $U$ and $W$ such that for each $u \in U$ and $w \in W$, the vertex $v$ is on every $u - w$ path.*
3. *There exist two vertices $u$ and $w$ distinct from $v$ such that $v$ is on every $u - w$ path.* ■

**Theorem 1.4** *Let $x$ be an edge of a connected graph $G$. The following statements are equivalent:*

1. *$x$ is bridge of $G$.*

2. *There exists a partition of $V$ into two subsets $U$ and $W$ such that for every vertex $u \in U$ and $w \in W$, the edge $x$ is on every $u - w$ path.*

3. *There exist two vertices $u, w$ such that the edge $x$ is on every $u - w$ path.* ■

**Theorem 1.5** *An edge $x$ of a connected graph $G$ is a bridge if and only if $x$ is not on any cycle of $G$.* ■

A *nonseparable graph* is a connected graph with no cutvertices. All other graphs are *separable*. A *block* of a separable graph $G$ is a maximal nonseparable subgraph of $G$.

Two $u - v$ paths are internally disjoint if they have no common vertices except $u$ and $v$.

**Theorem 1.6** *Every nontrivial connected graph contains at least two vertices that are not cutvertices.* ■

**Theorem 1.7** *A graph $G$ with $n \geq 3$ vertices is a block if and only if any two vertices of $G$ are connected by at least two internally disjoint paths.* ■

A block $G$ is also called 2-*connected* or *biconnected* because at least two vertices have to be removed from $G$ to disconnect it. The concepts of vertex and edge connectivities and generalized versions of the above theorem, called Menger's theorem, will be discussed further in Chapter 12 on connectivity.

## 1.5 OPERATIONS ON GRAPHS

Next we describe some binary operations defined on graphs. Let $G_1$ and $G_2$ be two graphs with disjoint vertex sets. The *union* $G = G_1 \cup G_2$ has $V(G) = V(G_1) \cup V(G_2)$ and $E(G) = E(G_1) \cup E(G_2)$. If a graph $G$ consists of $k(\geq 2)$ disjoint copies of a graph $H$, then we write $G = kH$. The *join* $G = G_1 + G_2$ has $V(G) = V(G_1) \cup V(G_2)$ and $E(G) = E(G_1) \cup E(G_2) \cup \{uv | u \in V(G_1)$ and $v \in V(G_2)\}$.

A pair of vertices $v_i$ and $v_j$ in a graph $G$ are said to be *identified* if the two vertices are replaced by a new vertex such that all the edges in $G$ incident on $v_i$ and $v_j$ are now incident on the new vertex.

By *contraction* of an edge $e$ we refer to the operation of removing $e$ and identifying its end vertices. A graph $G$ is *contractible* to a graph $H$ if $H$ can be obtained from $G$ by a sequence of contractions.

The *Cartesian product* $G = G_1 \Box G_2$ has $V(G) = V(G_1) \times V(G_2)$ and two vertices $(u_1, u_2)$ and $(v_1, v_2)$ of $G$ are adjacent if and only if either $u_1 = v_1$ and $u_2v_2 \in E(G_2)$ or $u_2 = v_2$ and $u_1v_1 \in E(G_1)$.

The *$n$-cube* $Q_n$ is the graph $K_2$ if $n = 1$, while for $n \geq 2, Q_n$ is defined recursively as $Q_{n-1} \Box K_2$. The $n$-cube $Q_n$ can also be considered as that graph whose vertices are labeled by the binary $n$-tuples $(a_1, a_2, \ldots, a_n)$ (i.e., $a_i$ is 0 or 1 for $1 \leq i \leq n$) and such that two vertices are adjacent if and only if their corresponding $n$-tuples differ at precisely one coordinate. The graph $Q_n$ is an $n$-regular graph of order $2^n$. The graph $Q_n$ is called a *hypercube*.

The *strong product* $G_1 \boxtimes G_2$ of $G_1$ and $G_2$ is the graph with vertex set $V(G_1) \times V(G_2)$ and two vertices $(u_1, u_2)$ and $(v_1, v_2)$ are adjacent in $G_1 \boxtimes G_2$ if $u_1 = v_1$ and $u_2v_2 \in E(G_2)$, or $u_1v_1 \in E(G_1)$ and $u_2 = v_2$, or $u_1v_1 \in E(G_1)$ and $u_2v_2 \in E(G_2)$.

The *direct product* $G_1 \times G_2$ of two graphs $G_1$ and $G_2$ is the graph with vertex set $V(G_1) \times V(G_2)$ and two vertices $(u_1, u_2)$ and $(v_1, v_2)$ are adjacent in $G_1 \times G_2$ if $u_1v_1 \in V(G_1)$ and $u_2v_2 \in E(G_2)$.

The *lexicographic product* $G_1 \circ G_2$ of two graphs $G_1$ and $G_2$ is the graph with vertex set $V(G_1) \times V(G_2)$ and two vertices $(u_1, u_2)$ and $(v_1, v_2)$ are adjacent in $G_1 \circ G_2$ if $u_1v_1 \in E(G_1)$ or $u_1 = v_1$ and $u_2v_2 \in E(G_2)$.

## 1.6 TREES

The graphs that are encountered in most of the applications are connected. Among connected graphs trees have the simplest structure and are perhaps the most important ones. A tree is the simplest nontrivial type of a graph and in trying to prove a general result or to test a conjecture in graph theory, it is sometimes convenient to first study the situation for trees.

A graph that contains no cycles is called an *acyclic graph*. A connected acyclic graph is called a *tree*. Any graph without cycles is also called a *forest* so that the components of a forest are trees.

A tree of a graph $G$ is a connected acyclic subgraph of $G$. A *spanning tree* of a graph $G$ is a tree of $G$ having all the vertices of $G$. A connected subgraph of a tree $T$ is called a *subtree* of $T$.

The *cospanning tree* $T^*$ of a spanning tree $T$ of a graph $G$ is the subgraph of $G$ having all the vertices of $G$ and exactly those edges of $G$ that are not in $T$. Note that a cospanning tree may not be connected.

The edges of a spanning tree $T$ are called the *branches* of $T$, and those of the corresponding cospanning tree $T^*$ are called *links* or *chords*.

A spanning tree $T$ uniquely determines its cospanning tree $T^*$. As such, we refer to the edges of $T^*$ as the chords or links of $T$.

**Theorem 1.8** *The following statements are equivalent for a graph $G$ with $n$ vertices and $m$ edges:*

1. *$G$ is a tree.*
2. *There exists exactly one path between any two vertices of $G$.*
3. *$G$ is connected and $m = n - 1$.*
4. *$G$ is acyclic and $m = n - 1$.*
5. *$G$ is acyclic, and if any two nonadjacent vertices of $G$ are connected by an edge, then the resulting graph has exactly one cycle.* ■

**Theorem 1.9** *A subgraph $G'$ of an $n$-vertex graph $G$ is a spanning tree of $G$ if and only if $G'$ is acyclic and has $n - 1$ edges.* ■

**Theorem 1.10** *A subgraph $G'$ of a connected graph $G$ is a subgraph of some spanning tree of $G$ if and only if $G'$ is acyclic.* ■

A graph is trivial if it has only one vertex.

**Theorem 1.11** *In a nontrivial tree there are at least two pendant vertices.* ■

For a graph $G$ with $p$ components a spanning forest is a collection of $p$ spanning trees, one for each component.

The *rank* $\rho(G)$ and *nullity* $\mu(G)$ of a graph $G$ of order $n$ and size $m$ are defined by $\rho(G) = n - k$ and $\mu(G) = m - n + k$, where $k$ is the number of components of $G$. Note that $\rho(G) + \mu(G) = m$.

The *arboricity* $a(G)$ of a graph $G$ is the minimum number of edge disjoint spanning forests into which $G$ can be decomposed.

## 1.7 CUTSETS AND CUTS

A *cutset* $S$ of a connected graph $G$ is a minimal set of edges of $G$ such that $G - S$ is disconnected. Equivalently, a cutset $S$ of a connected graph $G$ is a minimal set of edges of $G$ such that $G - S$ has exactly two components.

Let $G$ be a connected graph with vertex set $V$. Let $V_1$ and $V_2$ be two disjoint subsets of $V$ such that $V = V_1 \cup V_2$. Then the set $S$ of all those edges of $G$ having one end vertex in $V_1$ and the other in $V_2$ is called a *cut* of $G$. This is usually denoted by $[V_1, V_2]$.

**Theorem 1.12** *A cut in a connected graph $G$ is a cutset or union of edge-disjoint cutsets of $G$.* ■

Several results connecting spanning trees, circuits, and cutsets will be discussed in Chapters 7 and 8.

## 1.8 EULERIAN GRAPHS

Let $G$ be a connected graph. A closed trail containing all the edges of $G$ is called an *Eulerian trail*. A graph $G$ having an Eulerian trial is called an *Eulerian graph*.

The following theorem gives simple and useful characterizations of Eulerian graphs.

**Theorem 1.13** *The following statements are equivalent for a connected graph $G$.*

1. *$G$ is Eulerian.*
2. *The degree of every vertex in $G$ is even.*
3. *$G$ is the union of edge-disjoint circuits.* ■

**Corollary 1.2** *Let $G$ be a connected graph with exactly $2k (k \geq 1)$ odd vertices. Then the edge set of $G$ can be partitioned into $k$ open trails.* ■

**Corollary 1.3** *Let $G$ be a connected graph with exactly two odd vertices. Then $G$ has an open trail containing all the edges of $G$.* ■

## 1.9 HAMILTONIAN GRAPHS

A *Hamiltonian cycle* in a graph $G$ is a cycle containing all the vertices of $G$. A *Hamiltonian path* in $G$ is a path containing all the vertices of $G$. A graph $G$ is defined to be *Hamiltonian* if it has a Hamiltonian cycle.

An Euler trail is a closed walk passing through each edge exactly once and a Hamiltonian cycle is a closed walk passing through each vertex exactly once. Thus there is a striking similarity between an Eulerian graph and a Hamiltonian graph. This may lead one to expect that there exists a simple, useful, and elegant characterization of a Hamiltonian graph, as in the case of an Eulerian graph. Such is not the case; in fact, development of such a characterization is a major unsolved problem in graph theory. However, several sufficient conditions have been established for a graph to be Hamiltonian.

We present several necessary conditions and sufficient conditions for a graph to be Hamiltonian. We observe that any Hamiltonian graph has no cutvertex.

**Theorem 1.14** *If $G$ is Hamiltonian, then for every nonempty proper subset $S$ of $V(G), \omega(G-S) \leq |S|$ where $\omega(H)$ denotes the number of components in any graph $H$.* ■

A sequence $d_1 \leq d_2 \leq \cdots \leq d_n$ is said to be *graphic* if there is a graph $G$ with $n$ vertices $v_1, v_2, \ldots, v_n$ such that the degree $d(v_i)$ of $v_i$ equals $d_i$ for each $i$. Also $(d_1, d_2, \ldots, d_n)$ is then called the *degree sequence* of $G$.

If $S : d_1 \leq d_2 \leq \cdots \leq d_n$ and $S^* : d_1^* \leq d_2^* \leq \cdots \leq d_n^*$ are graphic sequences such that $d_i^* \geq d_i$ for $1 \leq i \leq n$, then $S^*$ is said to *majorize* $S$.

The following result is due to Chavátal [6].

**Theorem 1.15** *A simple graph $G = (V, E)$ of order $n$, with degree sequence $d_1 \leq d_2 \leq \cdots \leq d_n$ is Hamiltonian if $d_k \leq k < n/2 \Rightarrow d_{n-k} \geq n - k$.* ■

**Corollary 1.4** *A simple graph $G = (V, E)$ of order $n \geq 3$ with degree sequence $d_1 \leq d_2 \leq \cdots \leq d_n$ is Hamiltonian if one of the following conditions is satisfied:*

1. $1 \leq k \leq n \Rightarrow d_k \geq \frac{n}{2}$ [7].
2. $(u, v) \notin E \Rightarrow d(u) + d(v) \geq n$ [8].
3. $1 \leq k < \frac{n}{2} \Rightarrow d_k > k$ [6].
4. $j < k$, $d_j \leq j$ and $d_k \leq k - 1 \Rightarrow d_j + d_k \geq n$ [6]. ■

The *closure* of a graph $G$ with $n$ vertices is the graph obtained from $G$ by repeatedly joining pairs of nonadjacent vertices whose degree sum is at least $n$ until no such pair remains. The closure of $G$ is denoted by $c(G)$.

**Theorem 1.16** *A graph is Hamiltonian if and only if its closure is Hamiltonian.* ■

**Corollary 1.5** *Let $G$ be a graph with at least 3 vertices. If $c(G)$ is complete, then $G$ is Hamiltonian.* ■

## 1.10 GRAPH PARAMETERS

Several graph parameters relating to connectivity, matching, covering, coloring, and domination will be discussed in different chapters of this book. In this section we give a brief summary of some of the graph theoretic parameters.

**Definition 1.1** *The distance $d(u, v)$ between two vertices $u$ and $v$ in a connected graph is defined to be the length of a shortest $u - v$ path in $G$. The eccentricity $e(v)$ of a vertex $v$ is the number $\max_{u \in V(G)} d(u, v)$. Thus $e(v)$ is the distance between $v$ and a vertex farthest from $v$. The radius $rad\ G$ of $G$ is the minimum eccentricity among the vertices of $G$, while the diameter $diam\ G$ of $G$ is the maximum eccentricity. A vertex $v$ is a central vertex if $e(v) = rad(G)$ and the center $Cen(G)$ is the subgraph of $G$ induced by its central vertices. A vertex $v$ is a peripheral vertex if $e(v) = diam(G)$, while the periphery $Per(G)$ is the subgraph of $G$ induced by its peripheral vertices.*

**Definition 1.2** *If $G$ is a noncomplete graph and $t$ is a nonnegative real number such that $t \leq |S|/\omega(G-S)$ for every vertex-cut $S$ of $G$, where $\omega(G-S)$ is the number of components of $G-S$, then $G$ is defined to be $t$-tough. If $G$ is a $t$-tough graph and $s$ is a nonnegative real number such that $s < t$, then $G$ is also $s$-tough. The maximum real number $t$ for which a graph $G$ is $t$-tough is called the toughness of $G$ and is denoted by $t(G)$.*

**Definition 1.3** *A subset $S$ of vertices of a graph is called an independent set if no two vertices of $S$ are adjacent in $G$. The number of vertices in a maximum independent is called the independence number of $G$.*

**Definition 1.4** *The clique number $\omega(G)$ of a graph $G$ is the maximum order among the complete subgraphs of $G$.*

We observe that the clique number of $G$ is the independence number of its complement.

**Definition 1.5** *The girth of a graph $G$ having at least one cycle is the length of a shortest cycle in $G$. The circumference of $G$ is the length of a longest cycle in $G$.*

**Definition 1.6** *The vertex cover of a graph $G$ is a set $S$ of vertices such that every edge of $G$ has at least one end vertex in $S$. An edge cover of $G$ is a set $L$ of edges such that every vertex of $G$ is incident to some edge of $L$. The minimum size of a vertex cover is called the vertex covering number of $G$ and is denoted by $\beta(G)$. The minimum size of an edge cover is called the edge covering number of $G$ and is denoted by $\beta'(G)$.*

## 1.11 DIRECTED GRAPHS

A *directed graph* (or in short *digraph*) $D$ is a pair $(V, A)$ where $V$ is a finite nonempty set and $A$ is a subset of $V \times V - \{(x,x)/x \in V\}$. The elements of $V$ and $A$ are respectively called *vertices* and *arcs*. If $(u,v) \in A$ then the arc $(u,v)$ is said to have $u$ as its initial vertex (tail) and $v$ as its terminal vertex (head). Also the arc $(u,v)$ is said to join $u$ to $v$.

As in the case of graphs a digraph can also be represented by a diagram. A digraph is represented by a diagram of its underlying graph together with arrows on its edges, each arrow pointing toward the head of the corresponding arc. For example $D = (\{1,2,3,4\}, \{(1,2),(2,3),(1,3),(3,1)\})$ is a digraph. The diagram representing $D$ is given in Figure 1.2.

The *indegree* $d^-(v)$ of a vertex $v$ in a digraph $D$ is the number of arcs having $v$ as its terminal vertex. The *outdegree* $d^+(v)$ of $v$ is the number of arcs having $v$ as its initial vertex. The ordered pair $(d^+(v), d^-(v))$ is called the *degree pair* of $v$.

The degree pairs of the vertices 1, 2, 3 and 4 of the digraph in Figure 1.2 are (2,1), (1,1), (1,2), and (0,0) respectively.

**Theorem 1.17** *In a digraph $D$, the sum of the indegrees of all the vertices is equal to the sum of their outdegrees, each sum being equal to the number of arcs in $D$.* ■

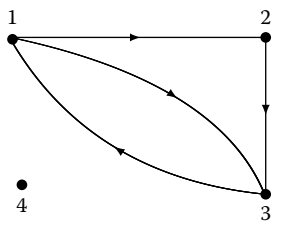


Figure 1.2 Example of a directed graph.

*Subgraphs* and *induced subgraphs* of a directed graph are defined as in the case of undirected graphs.

Let $D = (V, A)$ be a digraph. The *underlying graph* $G$ of $D$ is a graph having the same vertex set as $D$ and two vertices $u$ and $w$ are adjacent in $G$ whenever $(u, w)$ or $(w, u)$ is in $A$.

Similarly if we are given a graph $G$ we can obtain a digraph from $G$ by giving orientation to each edge of $G$. A digraph thus obtained from $G$ is called an *orientation* of $G$.

The *converse* digraph $D'$ of a digraph $D$ is obtained from $D$ by reversing the direction of each arc.

## 1.12 PATHS AND CONNECTIONS IN DIGRAPHS

A *walk* (*directed walk*) in a digraph is a finite alternating sequence $W = v_0x_1v_1 \dots x_nv_n$ of vertices and arcs in which $x_i = (v_{i-1}, v_i)$ for every arc $x_i$. $W$ is called a walk from $v_0$ to $v_n$ or a $v_0 - v_n$ walk. The vertices $v_0$ and $v_n$ are called *origin* and *terminus* of $W$, respectively, and $v_1, v_2, \dots, v_{n-1}$ are called its *internal vertices*. The *length* of a walk is the number of occurrences of arcs in it. A walk in which the origin and terminus coincide is called a *closed walk*.

A *path* (*directed path*) is a walk in which all the vertices are distinct. A *cycle* (*directed cycle* or *circuit*) is a nontrivial closed path whose origin and internal vertices are distinct.

If there is a path from $u$ to $v$ then $v$ is said to be *reachable* from $u$. A digraph is called *strongly connected* or *strong* if every pair of vertices are mutually reachable. A digraph is called *unilaterally connected* or *unilateral* if for every pair of vertices, at least one is reachable from the other. A digraph is called *weakly connected* or *weak* if the underlying graph is connected. A digraph is called *disconnected* if the underlying graph is disconnected.

**Theorem 1.18** *The edges of a connected graph $G = (V, E)$ can be oriented so that the resulting digraph is strongly connected if and only if every edge of $G$ is contained in at least one cycle.* ■

Let $D = (V, A)$ be a digraph.

1. Let $W_1$ be a maximal subset of $V$ such that for every pair of vertices $u, v \in W_1$, $u$ is reachable from $v$ and $v$ is reachable from $u$. Then the subdigraph of $D$ induced by $W_1$ is called a *strong component* of $D$.

2. Let $W_2$ be a maximal subset of $V$ such that for every pair of vertices $u, v \in W_2$, either $u$ is reachable from $v$ or $v$ is reachable from $u$. Then the subdigraph of $D$ induced by $W_2$ is called a *unilateral component* of $D$.

3. Let $W_3$ be a maximal subset of $V$ such that for every pair of vertices $u, v \in W_3, u$ and $v$ are joined by a path in the underlying graph of $D$. Then the subdigraph of $D$ induced by $W_3$ is called a *weak component* of $D$.

We observe that each vertex of $D$ is in exactly one strong component of $D$. An arc $x$ lies in exactly one strong component if it lies on a cycle. There is no strong component containing an arc that does not lie on any cycle.

The *condensation* $D^*$ of a digraph $D$ has the strong components $S_1, S_2, \dots, S_n$ of $D$ as its vertices with an arc from $S_i$ to $S_j$ whenever there is at least one arc from $S_i$ to $S_j$ in $D$.

A directed graph is said to be *quasi-strongly connected* if for every pair of vertices $v_1$ and $v_2$ there is a vertex $v_3$ from which there is a directed path to $v_1$ and a directed path to $v_2$. Note that $v_3$ need not be distinct from $v_1$ or $v_2$.

## 1.13 DIRECTED GRAPHS AND RELATIONS

A *binary relation* $R$ on a set $X = \{x_1, x_2, \ldots\}$ is a collection of ordered pairs of elements of $X$. If $(x_i, x_j) \in R$, then we write $x_i \; R \; x_j$. A most convenient way of representing a binary relation $R$ on a set $X$ is by a directed graph, the vertices of which stand for the elements of $X$ and the edges stand for the ordered pairs of elements of $X$ defining the relation $R$.

For example, the relation *is a factor of* on the set $X = \{2, 3, 4, 6, 9\}$ is shown in Figure 1.3.

Let $R$ be a relation on a set $X = \{x_1, x_2, \ldots\}$. The relation $R$ is called *reflexive* if $x_i R x_i$ for all $x_i \in X$. The relation $R$ is said to be *symmetric* if $x_i R x_j$ implies $x_j R x_i$. The relation $R$ is said to be *transitive* if $x_i R x_j$ and $x_j R x_k$ imply $x_i R x_k$. A relation $R$ which is reflexive, symmetric, and transitive is called an *equivalence relation*. If $R$ is an equivalence relation defined on a set $S$, then we can uniquely partition $S$ into subsets $S_1, S_2, \ldots, S_k$ such that two elements $x$ and $y$ of $S$ belong to $S_i$ if and only if $xRy$. The subsets $S_1, S_2, \ldots, S_k$ are all called the *equivalence classes* induced by the relation $R$ on the set $S$.

The directed graph representing a reflexive relation is called a *reflexive directed graph*. In a similar way *symmetric* and *transitive directed graphs* are defined. We now make the following observations about these graphs:

1. In a reflexive directed graph, there is a self-loop at each vertex.

2. In a symmetric directed graph, there are two oppositely oriented edges between any two adjacent vertices. Therefore, an undirected graph can be considered as representing a symmetric relation if we associate with each edge two oppositely oriented edges.

3. The edge $(v_1, v_2)$ is present in a transitive graph $G$ if there is a directed path in $G$ from $v_1$ to $v_2$.

## 1.14 DIRECTED TREES AND ARBORESCENCES

A vertex $v$ in a directed graph $D$ is a *root* of $D$ if there are directed paths from $v$ to all the remaining vertices of $D$.

**Theorem 1.19** *A directed graph $D$ has a root if and only if it is quasi-strongly connected.* ■

A directed graph $D$ is a *tree* if the underlying undirected graph is a tree. A directed graph $D$ is a *directed tree* or *arborescence* if $D$ is a tree and has a root.

We present in the next theorem a number of equivalent characterizations of a directed tree.

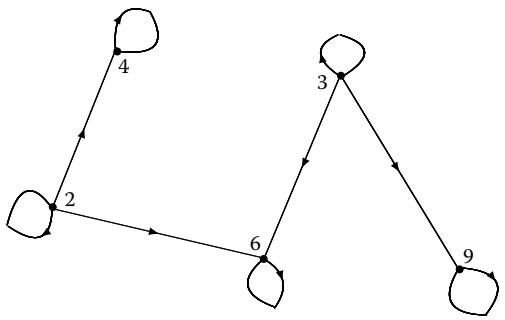


Figure 1.3 Directed graph representation of a relation on the set $X = \{2, 3, 4, 6, 9\}$.

**Theorem 1.20** *Let $D$ be a directed graph with $n > 1$ vertices. Then the following statements are equivalent:*

1. *$D$ is a directed tree.*
2. *There exists a vertex $r$ in $D$ such that there is exactly one directed path from $r$ to every other vertex of $D$.*
3. *$D$ is quasi-strongly connected and loses this property if any edge is removed from it.*
4. *$D$ is quasi-strongly connected and has a vertex $r$ such that $d^-(r) = 0$.* ■

**Theorem 1.21** *A directed graph $D$ has a directed spanning tree if and only if $D$ is quasi-strongly connected.* ■

## 1.15 DIRECTED EULERIAN GRAPHS

A *directed Euler trail* in a directed graph $D$ is a closed directed trail that contains all the arcs of $D$. An *open directed Euler trail* is an open directed trail containing all the arcs of $D$. A directed graph possessing a directed Euler trail is called a *directed Eulerian graph.*

The following theorem gives simple and useful characterizations of directed Eulerian graphs.

**Theorem 1.22** *The following statements are equivalent for a connected directed graph $D$.*

1. *$D$ is a directed Eulerian graph.*
2. *For every vertex $v$ of $D, d^-(v) = d^+(v)$.*
3. *$D$ is the union of some edge-disjoint directed cycles.* ■

**Theorem 1.23** *A directed connected graph $D$ possesses an open directed Euler trail if and only if the following conditions are satisfied:*

1. *In $D$ there are two vertices $v_1$ and $v_2$, such that $d^+(v_1) = d^-(v_1) + 1$ and $d^-(v_2) = d^+(v_2) + 1$.*
2. *For every vertex $v$ different from $v_1$ and $v_2$, we have $d^+(v) = d^-(v)$.* ■

**Theorem 1.24** *The number of directed Euler trails of a directed Eulerian graph $D$ without self-loops is $\tau_d(D) \prod_{p=1}^{n} (d^-(v_p) - 1)!$, where $n$ is the number of vertices in $D$ and $\tau_d(D)$ is the number of directed spanning trees of $G$ with $v_1$ as root.* ■

**Corollary 1.6** *The number of directed spanning trees of a directed Eulerian graph is the same for every choice of root.* ■

## 1.16 DIRECTED HAMILTONIAN GRAPHS

A directed circuit in a directed graph $D$ is a *directed Hamilton circuit* of $D$ if it contains all the vertices of $D$. A directed path in $D$ is a *directed Hamilton path* of $G$ if it contains all the vertices of $D$. A digraph is a *directed Hamiltonian graph* if it has a directed Hamilton circuit. A directed graph $D$ is *complete* if its underlying undirected graph is complete.

**Theorem 1.25** *Let $u$ be any vertex of a strongly connected complete directed graph with $n \geq 3$ vertices. For each $k, 3 \leq k \leq n$, there is a directed circuit of length $k$ containing $u$.* ■

**Corollary 1.7** *A strongly connected complete directed graph is Hamiltonian.* ■

**Theorem 1.26** *Let $D$ be a strongly connected $n$-vertex graph without parallel edges and self-loops. If for every vertex $v$ in $D$, $d^-(v) + d^+(v) \geq n$, then $D$ has a directed Hamilton circuit.* ■

**Corollary 1.8** *Let $D$ be a directed $n$-vertex graph without parallel edges or self-loops. If $\min(\delta^-, \delta^+) \geq n/2 > 1$, then $D$ contains a directed Hamilton circuit.* ■

**Theorem 1.27** *If a directed graph $D = (V, A)$ is complete, then it has a directed Hamilton path.* ■

## 1.17 ACYCLIC DIRECTED GRAPHS

A directed graph is *acyclic* if it has no directed circuits. Obviously the simplest example of an acyclic directed graph is a directed tree.

We can label the vertices of an $n$-vertex acyclic directed graph $D$ with integers from the set $\{1, 2, \ldots, n\}$ such that the presence of the edge $(i, j)$ in $D$ implies that $i < j$. Note that the edge $(i, j)$ is directed from vertex $i$ to vertex $j$. Ordering the vertices in this manner is called *topological sorting*.

**Theorem 1.28** *In an acyclic directed graph $D$ there exists at least one vertex with zero indegree and at least one vertex with zero outdegree.* ■

Select any vertex with zero outdegree. Since $D$ is acyclic, by Theorem 1.28, there is at least one such vertex in $D$. Label this vertex with the integer $n$. Now remove from $D$ this vertex and the edges incident on it. Let $D'$ be the resulting graph. Since $D'$ is also acyclic, we can now select a vertex whose outdegree in $D'$ is zero. Label this with the integer $n - 1$. Repeat this procedure until all the vertices are labeled. It is now easy to verify that this procedure results in a topological sorting of the vertices of $D$.

## 1.18 TOURNAMENTS

A *tournament* is an orientation of a complete graph. It derives its name from its application in the representation of structures of round-robin tournaments. In a round-robin tournament several teams play a game that cannot end in a tie, and each team plays every other team exactly once. In the directed graph representation of the round-robin tournament, vertices represent teams and an edge $(v_1, v_2)$ is present in the graph if the team represented by the vertex $v_1$ defeats the team represented by the vertex $v_2$. Clearly, such a directed graph has no parallel edges and self-loops, and there is exactly one edge between any two vertices. Thus it is a tournament.

The teams participating in a tournament can be ranked according to their scores. The *score* of a team $i$ is the number of teams it has defeated. This motivates the definition of the score sequence of a tournament.

The *score sequence* of an $n$-vertex tournament is the sequence $(s_1, s_2, \ldots, s_n)$ such that each $s_i$ is the outdegree of a vertex of the tournament. An interesting characterization of a tournament in terms of the score sequence is given in the following theorem.

**Theorem 1.29** *A sequence of nonnegative integers $s_1, s_2, \ldots, s_n$ is the score sequence of a tournament $G$ if and only if*

1. $\sum_{i=1}^{n} s_i = \frac{n(n-1)}{2}$.
2. $\sum_{i=1}^{k} s_i = \frac{k(k-1)}{2}$ *for all* $k < n$. ■

Suppose we can order the teams in a round-robin tournament such that each team precedes the one it has defeated. Then we can assign the integers $1, 2, \ldots, n$ to the teams to indicate their ranks in this order. Such a ranking is always possible since in a tournament there exists a directed Hamilton path and it is called *ranking* by a Hamilton path.

Note that ranking by a Hamilton path may not be the same as ranking by the score. Further, a tournament may have more than one directed Hamilton path. In such a case there will be more than one Hamilton path ranking. However, there exists exactly one directed Hamilton path in a transitive tournament. This is stated in the following theorem, which is easy to prove.

**Theorem 1.30** *In a transitive tournament there exists exactly one directed Hamilton path.* ■

## 1.19 COMPUTATIONAL COMPLEXITY AND COMPLETENESS

In assessing the efficiency of an algorithm two metrics are used; time complexity and space complexity. In this book we will be mainly concerned with time complexities of algorithms. The computational time complexity of an algorithm is a measure of the number of primitive operations (low level instructions) performed during the execution of the algorithm. We will use what is known as the random access machine (RAM) model in which it is assumed that the computer can perform any primitive operation in a constant number of steps which do not depend on the size of the input.

A function $t(n)$ is said to be $O(g(n))$ if there exist some constant $c$ and some nonnegative integer $n_0$ such that

$$t(n) \leq c\, g(n) \text{ for all } n \geq n_0$$

This definition is referred to as the *big-Oh* notation. Usually it is pronounced as $t(n)$ *is big Oh of* $g(n)$. We can also say as $t(n)$ *is order* $g(n)$. For example $5n^2 + 100n$ is $O(n^2)$.

A function $t(n)$ is said to be $\Omega(g(n))$, if there exist some positive constant $c$ and some nonnegative integer $n_0$ such that

$$t(n) \geq c\, g(n) \text{ for all } n \geq n_0$$

This definition is referred to as the *big-Omega* notation. Usually it is pronounced as $t(n)$ *is big-Omega of* $g(n)$. For example, $n^3$ is $\Omega(n^2)$.

A function $t(n)$ is said to be $\Theta(g(n))$, if $t(n)$ is $O(g(n))$ and $\Omega(g(n))$. It is pronounced as $t(n)$ *is big-Theta of* $g(n)$. For example, $(1/2)\, n(n-1)$ is $\Theta(n^2)$.

There are several texts that discuss the above and other asympototic notations as well as properties involving them. For example, see Levitin [9], Goodrich and Tamassia [10], and Cormen et al. [11].

An algorithm for a problem is efficient [12] if there exists a polynomial $p(k)$ such that an instance of the problem whose input length is $k$ takes at most $p(k)$ elementary computational steps to solve. In other words any algorithm of polynomial time complexity is accepted to be efficient. There are several problems that defy polynomial time algorithms in spite of massive efforts to solve them efficiently. This family includes several important problems such as the

traveling salesman problem, the graph coloring problem, the problem of simplifying Boolean functions, scheduling problems, and certain covering problems.

In this section we present a brief introduction to the theory of $NP$-completeness. A *decision* problem is one that asks only for a yes or no answer. For example the question *Can this graph be 5-colored?* is a decision problem. Many of the important optimization problems can be phrased as decision problems. Usually, if we find a fast algorithm for a decision problem, then we will be able to solve the corresponding original problem also efficiently. For instance, if we have a fast algorithm to solve the decision problem for graph coloring, by repeated applications (in fact, $n \log n$ applications) of this algorithm, we can find the chromatic number of an $n$-vertex graph.

A decision problem belongs to the class $P$ if there is a polynomial time algorithm to solve the problem. A verification algorithm is an algorithm $A$ which takes as input an instance of a problem and a candidate solution to the problem, called a *certificate* and verifies in polynomial time whether the certificate is a solution to the given problem instance. The class $NP$ is the class of problems which can be verified in polynomial time.

The fundamental open question in computational complexity is whether the class $P$ equals the class $NP$. By definition, the class $NP$ contains all problems in class $P$. It is not known, however, whether all problems in $NP$ can be solved in polynomial time.

In an effort to determine whether $P = NP$, Cook [13] defined the class of $NP$-complete problems. We say that a problem $P_1$ is polynomial-time reducible to a problem $P_2$, written $P_1 \leq_p P_2$, if

1. There exists a function $f$ which maps any instance $I_1$ of $P_1$ to an instance of $P_2$ in such a way that $I_1$ is a *yes* instance of $P_1$ if and only if $f(I_1)$ is a *yes* instance of $P_2$.

2. For any instance $I_1$, the instance $f(I_1)$ can be constructed in polynomial time.

If $P_1$ is polynomial-time reducible to $P_2$, then any algorithm for solving $P_2$ can be used to solve $P_1$. We define a problem $P$ to be $NP$-complete if (1) $P \in NP$, and (2) for every problem $P' \in NP, P' \leq_p P$. If a problem $P$ can be shown to satisfy condition (2), but not necessarily condition (1), then we say that it is *$NP$-hard*. Let *$NPc$* denote the class of $NP$-complete problems.

The relation $\leq_p$ is transitive. Because of this, a method frequently used in demonstrating that a given problem is $NP$-complete is the following:

1. Show that $P \in NP$.

2. Show that there exists a problem $P' \in NPc$, such that $P' \leq_p P$.

It follows from the definition of $NP$-completeness that if any problem in $NPc$ can be solved in polynomial time, then every problem in $NPc$ can be solved in polynomial time and $P = NP$. On the other hand, if there is some problem in $NPc$ that cannot be solved in polynomial time, then no problem in $NPc$ can be solved in polynomial time.

Cook [13] proved that there is an $NP$-complete problem. The satisfiability problem is defined as follows:

Let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of Boolean variables. A literal is either a variable $x_i$ or its complement $\overline{x_i}$. Thus the set of literals is $L = \{x_1, x_2, \ldots, x_n, \overline{x_1}, \overline{x_2}, \ldots, \overline{x_n}\}$. A clause $C$ is a subset of $L$. The *satisfiability problem* (SAT) is: Given a set of clauses, does there exist a set of truth values ($T$ or $F$), one for each variable, such that every clause contains at least one literal whose value is $T$. Cook's proof that SAT is $NP$-complete opened the way to demonstrate the $NP$-completeness of a vast number of problems. The second problem to be

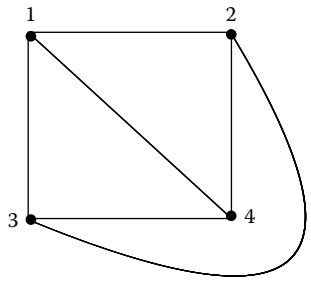


Figure 1.4 Graph for the illustration of the concept of reducibility.

proved $NP$-complete is 3-SAT the 3-*satisfiability problem*, which is the special case of SAT in that only three literals are permitted in each clause.

The list of $NP$-complete problems has grown very rapidly since Cook's work. Karp [14,15] demonstrated the $NP$-completeness of a number of combinatorial problems. Garey and Johnson [16] is the most complete reference on $NP$-completeness and is highly recommended. Other good textbooks recommended for this study include Horowitz and Sahni [17], Melhorn [18], and Aho et al. [19]. For updates on $NP$-completeness see the article titled "$NP$-completeness: An ongoing guide" in the *Journal of Algorithms*.

Since the concept of reducibility plays a dominant role in establishing the $NP$-completeness of a problem, we shall illustrate this with an example.

Consider the graph $G = (V, E)$ in Figure 1.4 and the decision problem *Can the vertices of G be 3-colored?* We now reduce this problem to an instance of SAT.

We define 12 Boolean variables $x_{i,j} (i = 1, 2, 3, 4; j = 1, 2, 3)$ where the variable $x_{i,j}$ corresponds to the assertion that *vertex i has been assigned color j*. The clauses are defined as follows:

$$\begin{aligned}
C(i) &= \{x_{i,1}, x_{i,2}, x_{i,3}\}, 1 \leq i \leq 4; \\
T(i) &= \{\overline{x}_{i,1}, \overline{x}_{i,2}\}, 1 \leq i \leq 4; \\
U(i) &= \{\overline{x}_{i,1}, \overline{x}_{i,3}\}, 1 \leq i \leq 4; \\
V(i) &= \{\overline{x}_{i,2}, \overline{x}_{i,3}\}, 1 \leq i \leq 4; \\
D(e, j) &= \{\overline{x}_{u,j}, \overline{x}_{v,j}\}, \text{ for every } e = (u, v) \in E, 1 \leq j \leq 3.
\end{aligned}$$

Whereas, $C(i)$ asserts that each vertex $i$ has been assigned at least one color, the clauses $T(i), U(i)$ and $V(i)$ together assert that no vertex has been assigned more than one color. The clauses $D(e, j)$'s guarantee that the coloring is proper (adjacent vertices have been assigned distinct colors).

Thus, the graph of Figure 1.4 is 3-colorable if and only if there exists an assignment of truth values $T$ and $F$ to the 12 Boolean variables $x_{1,1}, x_{1,2}, \ldots, x_{4,3}$ such that the each of the clauses contains at least one literal whose value is $T$.

## References

[1] J. A. Bondy and U. S. R. Murty, *Graph Theory*, Springer, Berlin, Germany, 2008.

[2] G. Chartrand and L. Lesniak, *Graphs and Digraphs*, 4th Edition, CRC Press, Boca Raton, FL, 2005.

[3] M. N. S. Swamy and K. Thulasiraman, *Graphs, Networks and Algorithms*, Wiley-Interscience, New York, 1981.

[4] K. Thulasiraman and M. N. S. Swamy, *Graphs: Theory and Algorithms*, Wiley-Interscience, New York, 1992.

[5] D. B. West, *Introduction to Graph Theory*, 2nd Edition, Prentice Hall, Upper Saddle River, NJ, 2001.

[6] V. Chavátal, On Hamilton's ideals, *J. Comb. Th. B*, **12** (1972), 163–168.

[7] G. A. Dirac, Some theorems on abstract graphs, *Proc. London Math. Soc.*, **2** (1952), 69–81.

[8] O. Ore, Note on Hamilton circuits, *Amer. Math. Monthly*, **67** (1960), 55.

[9] A. Levitin, *Introduction to the Design and Analysis of Algorithms*, Pearson, Boston, MA, 2012.

[10] M. T. Goodrich and R. Tamassia, *Algorithm Design: Foundations, Analysis and Internet Examples*, John Wiley & Sons, New York, 2002.

[11] T. H. Cormen, C. E. Liercersona, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1990.

[12] J. Edmonds, Paths, trees and flowers, *Canad. J. Math.*, **17** (1965), 449–467.

[13] S. A. Cook, The complexity of theorem proving procedures, *Proc. 3rd ACM Symp. on Theory of Computing*, ACM, New York, 1971, 151–158.

[14] R. M. Karp, Reducibility among combinatorial problems, *Complexity of Computer Communications*, R. E. Miller and J. W. Thatcher, Eds., Plenum Press, New York, 1972, 85–104.

[15] R. M. Karp, On the computational complexity of combinatorial problems, *Networks*, **5** (1975), 45–68.

[16] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, CA, 1979.

[17] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, Potomac, MD, 1978.

[18] K. Melhorn, *Graph Algorithms and NP-Completeness*, Springer-Verlag, New York, 1984.

[19] A. V. Aho, J. E. Hopcroft, and J. D. Ullman *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

CHAPTER 2

# Basic Graph Algorithms*

**Krishnaiyan "KT" Thulasiraman**

## CONTENTS



## 2.1 INTRODUCTION

Graphs arise in the study of several practical problems. The first step in such studies is to discover graph theoretic properties of the problem under consideration that would help us in the formulation of a method of solution to the problem. Usually solving a problem involves analysis of a graph or testing a graph for some specified property. Graphs that arise in the study of real-life problems are very large and complicated. Analysis of such graphs in an efficient manner, therefore, involves the design of efficient computer algorithms.

In this and the next chapter we discuss several basic graph algorithms. We consider these algorithms to be basic in the sense that they serve as building blocks in the design of more complex algorithms. While our main concern is to develop the theoretical foundation on which the design of the algorithms is based, we also develop results concerning the computational complexity of these algorithms.

The computational complexity of an algorithm is a measure of the running time of the algorithm. Thus it is a function of the size of the input. In the case of graph algorithms, complexity results will be in terms of the number of vertices and the number of edges in the graph. In the following function $g(n)$ is said to be $O(f(n))$ if and only if there exist constants $c$ and $n_0$ such that $|g(n)| \leq c|f(n)|$ for all $n \geq n_0$. Furthermore, all our complexity results will be with respect to the worst-case analysis.

There are different methods of representing a graph on a computer. Two of the most common methods use the adjacency matrix and the adjacency list. Adjacency matrix representation is not a very efficient one in the case of sparse graphs. In the adjacency

*This chapter is an edited version of Sections 14.1, 14.2, 15.1, and 15.8 in Swamy and Thulasiraman [17].

list representation, we associate with each vertex a list that contains all the edges incident on it. A detailed discussion of data structures for representing a graph may be found in some of the references listed at the end of this chapter.

## 2.2 MINIMUM WEIGHT SPANNING TREE

Consider a weighted connected undirected graph $G$ with a nonnegative real weight $w(e)$ associated with each edge $e$ of $G$. The weight of a subgraph of $G$ will refer to the sum of the weights of the edges of the subgraph. In this section we discuss the problem of constructing a minimum weight spanning tree of $G$. We present two algorithms for this problem, namely, Kruskal's algorithm [1] and the one due to Prim [2]. Theorems 2.1 through 2.3 provide the basis of these algorithms. In the following, $T + e$ denotes the subgraph $T \cup \{e\}$. Similarly, $T - e$ denotes the subgraph that results after deleting edge $e$ from $T$. Also, contraction of an edge $e = (u, v)$ results in a new graph $G'$ in which the vertices $u$ and $v$ are replaced by a single vertex $w$ and all the edges in $G$ that are incident on $u$ or $v$ are incident on vertex $w$ in $G'$. The self loop on $w$ that results because of the coalescing of $u$ and $v$ will not be present in $G'$.

**Theorem 2.1** *Consider a vertex $v$ in a weighted connected graph $G$. Among all the edges incident on $v$, let $e$ be one of minimum weight. Then, $G$ has a minimum weight spanning tree that contains $e$.*

*Proof.* Let $T_{\min}$ be a minimum weight spanning tree of $G$. If $T_{\min}$ does not contain $e$, then a circuit $C$ is created when $e$ is added to $T_{\min}$. Let $e'$ be the edge of $C$ that is adjacent to $e$. Clearly $e' \in T_{\min}$. Also $T' = T_{\min} - e' + e$ is a spanning tree of $G$. Since $e$ and $e'$ are both incident on $v$, we get $w(e) \leq w(e')$. But $w(e) \geq w(e')$ because $w(T') = w(T_{\min}) - w(e') + w(e) \geq w(T_{\min})$. So, $w(e) = w(e')$ and $w(T') = w(T_{\min})$. Thus we have found a minimum weight spanning tree, namely $T'$, containing $e$. ■

If the fundamental circuit with respect to a chord $c$ of a spanning tree $T$ contains branch $b$, then $T - b + c$ is also a spanning tree of $G$ (see Chapter 7). Using this result we can prove the statements in Theorem 2.2 along the same lines as the proof of Theorem 2.1.

**Theorem 2.2** *Let $e$ be a minimum weight edge in a weighted connected graph $G$. Then*

1. *$G$ has a minimum weight spanning tree that contains $e$.*
2. *If $T_{min}$ is a minimum weight spanning tree of $G$, then for every chord $c$ $w(b) \leq w(c)$, for every branch $b$ in the fundamental circuit of $T_{min}$ with respect to $c$.*
3. *If $T_{min}$ is a minimum weight spanning tree of $G$, then for every branch $b \in T_{min}$ $w(b) \leq w(c)$, for every chord $c$ in the fundamental cutset of $T_{min}$ with respect to $b$.* ■

**Theorem 2.3** *Let $T$ be an acyclic subgraph of a weighted connected graph $G$ such that there exists a minimum weight spanning tree containing $T$. If $G'$ denotes the graph obtained by contracting the edges of $T$, and $T'_{min}$ is a minimum weight spanning tree of $G'$, then $T'_{min} \cup T$ is a minimum weight spanning tree of $G$.*

*Proof.* Let $T_{\min}$ be a minimum weight spanning tree of $G$ containing $T$. Let $T_{\min} = T \cup T'$. Then $T'$ is a spanning tree of $G'$. Therefore

$$w(T') \geq w(T'_{\min}). \tag{2.1}$$

It is easy to see that $T'_{\min} \cup T$ is also a spanning tree of $G$. So

$$w(T'_{\min} \cup T) \geq w(T_{\min}) = w(T) + w(T'). \tag{2.2}$$

From this we get

$$w(T'_{\min}) \geq w(T'). \tag{2.3}$$

Combining (2.1) and (2.3) we get $w(T'_{\min}) = w(T')$, and so $w(T'_{\min} \cup T) = w(T' \cup T) = w(T_{\min})$. Thus $T'_{\min} \cup T$ is a minimum weight spanning tree of $G$. ■

We now present Kruskal's algorithm.

**Algorithm 2.1 Minimum weight spanning tree (Kruskal)**

***Input:*** $G = (V, E)$ is the given nontrivial $n$-vertex weighted connected graph with $m$ edges. The edges are ordered according to their weights, that is, $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$.
***Output:*** A minimum weight spanning tree of $G$. The edges $e'_1, e'_2, \ldots, e'_{n-1}$ will be the required spanning tree.

```
begin
    k ← 0;
    T_0 ← ϕ;
    for i = 1 to m do
        If T_k + e_i is acyclic then
            begin
                k ← k + 1;
                e'_k ← e_i;
                T_k ← T_{k−1} + e'_k;
            end
end
```

Kruskal's algorithm essentially proceeds as follows. Edges are first sorted in the order of nondecreasing weights and then examined, one at a time, for inclusion in a minimum weight spanning tree. An edge is included if it does not form a circuit with the edges already selected.

Next we prove the correctness of Kruskal's algorithm.

**Theorem 2.4** *Kruskal's algorithm constructs a minimum weight spanning tree of a weighted connected graph.*

*Proof.* Let $G$ be the given nontrivial weighted connected graph. Clearly, when Kruskal's algorithm terminates, the edges selected will form a spanning tree of $G$. Thus the algorithm terminates with $k = n - 1$, and $T_{n-1}$ as a spanning tree of $G$.

We next establish that $T_{n-1}$ is indeed a minimum weight spanning tree of $G$ by proving that every $T_k$, $k \geq 1$ constructed in the course of Kruskal's algorithm is contained in a minimum weight spanning tree of $G$. Our proof is by induction on $k$.

Clearly by Theorem 2.1, $G$ has a minimum weight spanning tree that contains the edge $e'_1 = e_1$. In other words $T_1 = \{e_1\}$ is contained in a minimum weight spanning tree of $G$. As inductive hypothesis, assume that $T_k$, for some $k \geq 1$ is contained in a minimum weight spanning tree of $G$. Let $G'$ be the graph obtained by contracting the edges of $T_k$. Then the edge $e'_{k+1}$ selected by the algorithm will be a minimum weight edge in $G'$. So, by Theorem 2.2, the edge $e'_{k+1}$ is contained in a minimum weight spanning tree $T'_{\min}$ of $G'$. By Theorem 2.3, $T_k \cup T'_{\min}$ is a minimum weight spanning tree of $G$. More specifically $T_{k+1} = T_k + e'_{k+1}$ is contained in a minimum weight spanning tree of $G$ and the correctness of Kruskal's algorithm follows. ■

We next present another algorithm due to Prim [2] to construct a minimum weight spanning tree of a weighted connected graph.

**Algorithm 2.2 Minimum weight spanning tree (Prim)**

***Input:*** $G = (V, E)$ is the given nontrivial $n$-vertex weighted connected graph.
***Output:*** A minimum weight spanning tree of $G$. The edges $e_1, e_2, \ldots, e_{n-1}$ will form the required minimum spanning tree.

```
begin
    i ← 1;
    T_0 ← ϕ;
    Select any vertex v ∈ V;
    S ← {v};
    while i ≤ n − 1 do
        begin
            Select an edge e_i = (p, q) of minimum weight such that e_i has exactly one
            end vertex, say p, in S;
            T_i ← T_{i−1} + e_i;
            Add vertex q to the set S;
            i ← i + 1;
        end
end
```

As in Kruskal's algorithm, Prim's algorithm also constructs a sequence of acyclic subgraphs $T_1, T_2, \ldots$, and terminates with $T_{n-1}$, a minimum weight spanning tree of $G$. The subgraph $T_{i+1}$ is constructed from $T_i$ by adding an edge of minimum weight with exactly one end vertex in $T_i$. This construction ensures that all $T_i$'s are connected. If $G'$ denotes the graph obtained by contracting the edges of $T_i$, and $W$ denotes the vertex of $G'$, which corresponds to the vertex set of $T_i$, then $e_{i+1}$ is in fact a minimum weight edge incident on $W$ in $G'$. This observation and Theorems 2.1 and 2.3 can be used (as in the proof of Theorem 2.4) to prove that each $T_i$ is contained in a minimum weight spanning tree of $G$. This would then establish that the spanning tree produced by Prim's algorithm is a minimum weight spanning tree of $G$.

Both Prim's and Kruskal's algorithms can be viewed as special cases of a more general version. Both these algorithms construct a sequence of acyclic subgraphs $T_1, T_2, \ldots, T_{n-1}$ with $T_{n-1}$ as a minimum weight spanning tree. Each $T_{i+1}$, is constructed by adding an edge $e_{i+1}$ to $T_i$. They differ in the way $e_{i+1}$ is selected. As before, let $G'$ denote the graph obtained by contracting the edges of $T_i$, and $W$ denote the (super) vertex of $G'$, which corresponds to the vertex set of $T_i$. Then the two algorithms select weight edge $e_{i+1}$ as follows.

**Prim:** $e_{i+1}$ is a minimum weight edge incident on $W$ in $G'$.

Note: This is a restricted application of Theorem 2.1 since this theorem is applicable to any vertex in $G'$.

**Kruskal:** $e_{i+1}$ is a minimum weight edge in $G'$.

Note: This is also a restricted application of Theorem 2.1 since this theorem only requires $e_{i+1}$ to be the minimum weight edge incident on some vertex in $G'$.

Both these algorithms result in a minimum weight spanning tree since the edge $e_{i+1}$ selected as above is in a minimum weight spanning tree of $G'$ as proved in Theorems 2.1 and 2.3.

The following algorithm which unifies both Prim's and Kruskal's algorithms is slightly more general than both.

**Algorithm 2.3 Minimum weight spanning tree (Unified version)**

***Input:*** $G = (V, E)$ is the given nontrivial $n$-vertex weighted connected graph.
***Output:*** A minimum weight spanning tree of $G$. The edges $e_1, e_2, \ldots, e_{n-1}$ will form the required minimum spanning tree.

```
begin
    i ← 1;
    T_0 ← ϕ;
    G' ← G;
    while i ≤ n − 1 do
        begin
            Select an edge e_i = (p, q) such that e_{i+1} is a minimum weight edge incident
            on any vertex in G';
            Contract the edge e_i in G' and let G' denote the contracted graph;
            T_i ← T_{i−1} + e_i;
            i ← i + 1;
        end
end
```

For complexity results relating to the minimum weight spanning tree enumeration problem see Kerschenbaum and Van Slyke [3], Yao [4], and Cheriton and Tarjan [5]. For sensitivity analysis of minimum weight spanning trees and shortest path trees see Tarjan [6]. See Papadimitriou and Yannakakis [7] for a discussion of the complexity of restricted minimum weight spanning tree problems. For a history of the minimum weight spanning tree problem see Graham and Hall [8]. The complexity of Kruskal's algorithm is $O(m \log n)$ (see Korte and Vygen [9]). Clearly the complexity of Prim's algorithm is $O(n^2)$. For more sophisticated implementations see [10–13]. See also [14] for an algorithm due to Jarnik.

## 2.3 OPTIMUM BRANCHINGS

Consider a weighted directed graph $G = (V, E)$. Let $w(e)$ be the weight of edge $e$. Recall that the weight of a subgraph of $G$ is defined to be equal to the sum of the weights of all the edges in the subgraph.

A subgraph $G_s$ of $G$ is a branching in $G$ if $G_s$ has no directed circuits and the in-degree of each vertex of $G_s$ is at most 1. Clearly each component of $G_s$ is a directed tree*. A branching of maximum weight is called an *optimum branching*.

In this section we discuss an algorithm due to Edmonds [15] for computing an optimum branching of $G$. Our discussion here is based on Karp [16].

An edge $e = (i, j)$ directed from vertex $i$ to vertex $j$ is critical if

1. $w(e) > 0$.
2. $w(e) \geq w(e')$ for every edge $e' = (k, j)$ incident into $j$.

A spanning subgraph $H$ of $G$ is a critical subgraph of $G$ if

1. Every edge of $H$ is critical.
2. The in-degree of every vertex of $H$ is at most 1.

*See Chapter 1 for the definition of a directed tree, also called an arborescence.

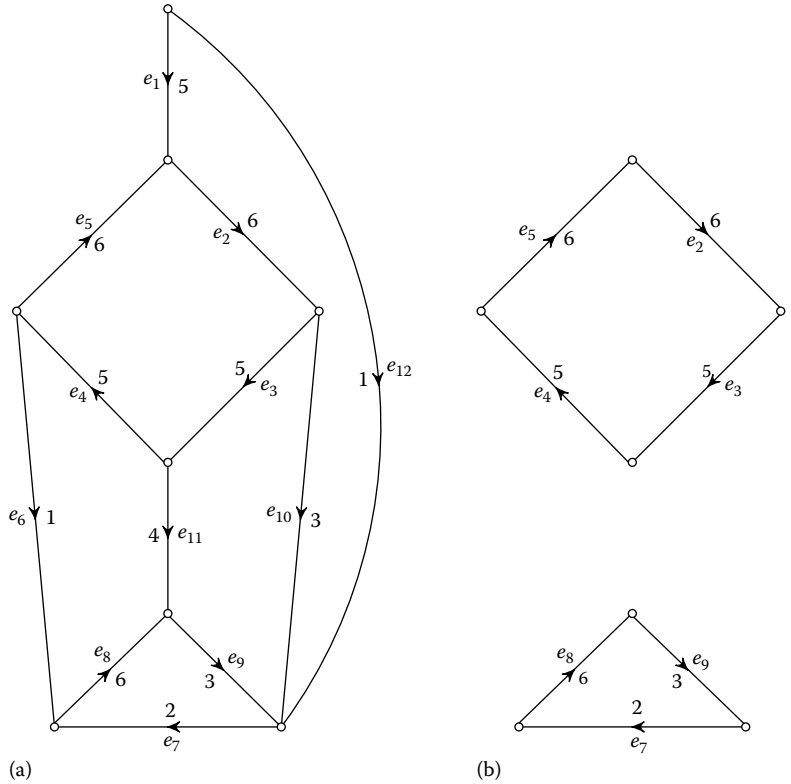


Figure 2.1 (a) A directed graph $G$. (b) A critical subgraph of $G$. (Data from M. N. Swamy and K. Thulasiraman, *Graphs, Networks and Algorithms*, Wiley-Interscience, 1981.)

A directed graph $G$ and a critical subgraph $H$ of $G$ are shown in Figure 2.1. It is easy to see that

1. Each component of a critical subgraph contains at most one circuit, and such a circuit will be a directed circuit.

2. A critical subgraph with no circuits is an optimum branching of $G$.

Consider a branching $B$. Let $e = (i, j)$ be an edge not in $B$, and let $e'$ be the edge of $B$ incident into vertex $j$. Then $e$ is eligible relative to $B$ if

$$B' = (B \cup e) - e'$$

is a branching.

For example, the edges $\{e_1, e_2, e_3, e_4, e_7, e_8\}$ form a branching $B$ of the graph of Figure 2.1. The edge $e_6$, not in $B$, is eligible relative to $B$ since

$$(B \cup e_6) - e_7$$

is a branching of this graph.

Lemmas 2.1 and 2.2 are easy to prove, and they lead to Theorem 2.5, which forms the basis of Karp's proof of the correctness of Edmonds' algorithm. In the following the edge set of a subgraph $H$ will also be denoted by $H$.

**Lemma 2.1** *Let $B$ be a branching, and let $e = (i, j)$ be an edge not in $B$. Then $e$ is eligible relative to $B$ if and only if in $B$ there is no directed path from $j$ to $i$.* ■

**Lemma 2.2** *Let $B$ be a branching and let $C$ be a directed circuit such that no edge of $C - B$ is eligible relative to $B$. Then $|C - B| = 1$.* ■

**Theorem 2.5** *Let $H$ be a critical subgraph. Then there exists an optimum branching $B$ such that, for every directed circuit $C$ in $H$, $|C - B| = 1$.*

*Proof.* Let $B$ be an optimum branching that, among all optimum branchings, contains a maximum number of edges of the critical subgraph.

Consider any edge $e \in H - B$ that is incident into vertex $j$, and let $e'$ be the edge of $B$ incident into $j$. If $e$ were eligible, then

$$(B \cup e) - e'$$

would also be an optimum branching, containing a larger number of edges of $H$ than $B$ does; a contradiction. Thus no edge of $H - B$ is eligible relative to $B$. So, by Lemma 2.2, for each circuit $C$ in $H$, $|C - B| = 1$. ■

Let $C_1, C_2, \ldots, C_k$ be the directed circuits in $H$. Note that no two circuits of $H$ can have a common edge. In other words, these circuits are edge-disjoint. Let $e_i^0$ be an edge of minimum weight in $C_i$, $i \geq 1$.

**Corollary 2.1** *There exists an optimum branching $B$ such that:*

1. $|C_i - B| = 1$, $i = 1, 2, \ldots, k$.
2. *If no edge of $B - C_i$ is incident into a vertex in $C_i$, $i = 1, 2, \ldots, k$, then*

$$C_i - B = e_i^0. \tag{2.4}$$

*Proof.* Among all optimum branchings that satisfy item 1, let $B$ be a branching containing a minimum number of edges from the set $\{e_1^0, e_2^0, \ldots, e_k^0\}$. We now show that $B$ satisfies item 2.

If not, suppose that, for some $i$, $e_i^0 \in B$, but that no edge of $B - C_i$ is incident into a vertex in $C_i$. Let $e = C_i - B$. Then $(B - e_i^0) \cup e$ is clearly an optimum branching that satisfies item 1 but has fewer edges than $B$ from the set $\{e_1^0, e_2^0, \ldots, e_k^0\}$. This is a contradiction. ■

This result is very crucial in the development of Edmonds' algorithm. It suggests that we can restrict our search for optimum branchings to those that satisfy (2.4).

Next we construct, from the given graph $G$, a simpler graph $G'$ and show how to construct from an optimum branching of $G'$ an optimum branching of $G$ that satisfies (2.4).

As before, let $H$ be the critical subgraph of $G$ and let $C_1, C_2, \ldots, C_k$ be the directed circuits in $H$. The graph $G'$ is constructed by contracting all the edges in each $C_i = 1, 2, \ldots, k$. In $G'$, vertices of each circuit $C_i$ are represented by a single vertex $a_i$, called a *pseudo-vertex*. The weights of the edges of $G'$ are the same as those of $G$, except for the weights of the edges incident into the pseudo-vertices. These weights are modified as follows.

Let $e = (i, j)$ be an edge of $G$ such that $j$ is a vertex of some circuit $C_r$ and $i$ is not in $C_r$. Then in $G'$, $e$ is incident into the pseudo-vertex $a_r$. Define $\tilde{e}$ as the unique edge in $C_r$ that is incident into vertex $j$. Then in $G'$ the weight of $e$, denoted by $w'(e)$, is given by

$$w'(e) = w(e) - w(\tilde{e}) + w(e_r^0). \tag{2.5}$$

For example, consider the edge $e_1$ incident into the directed circuit $\{e_2, e_3, e_4, e_5\}$ of the critical subgraph of the graph $G$ of Figure 2.1. Then $\tilde{e}_1 = e_5$, and the weight of $e_1$ in $G'$ is given by

$$\begin{aligned} w'(e_1) &= w(e_1) - w(e_5) + w(e_4) \\ &= 5 - 6 + 5 \\ &= 4. \end{aligned}$$

Note that $e_4$ is a minimum weight edge in the circuit $\{e_2, e_3, e_4, e_5\}$.

Let $E$ and $E'$, respectively, denote the edge sets of $G$ and $G'$. We now show how to construct from a branching $B'$ of $G'$ a branching $B$ of $G$ that satisfies (2.4) and vice versa.

For any branching $B$ of $G$ that satisfies (2.4) it is easy to see that

$$B' = B \cap E' \tag{2.6}$$

is a branching of $G'$. Furthermore, $B'$ as defined is unique for a given $B$.

Next consider a branching $B'$ of $G'$. For each $C_i$, let us define $C_i'$ as follows:

1. If the in-degree in $B'$ of a pseudo-vertex $a_i$ is zero, then

$$C_i' = C_i - e_i^0.$$

2. If the in-degree in $B'$ of $a_i$ is nonzero, and $e$ is the edge of $B'$ incident into $a_i$, then

$$C_i' = C_i - \tilde{e}.$$

Now it is easy to see that

$$B = B' \bigcup_{i=1}^{k} C_i'. \tag{2.7}$$

is a branching of $G$ that satisfies (2.4). Furthermore, $B$ as defined is unique for a given $B'$.

Thus we conclude that there is a one-to-one correspondence between the set of branchings of $G$ that satisfy (2.4) and the set of branchings of $G'$. Also, the weights of the corresponding branchings $B$ and $B'$ satisfy

$$w(B) - w(B') = \sum_{i=1}^{k} w(C_i) - \sum_{i=1}^{k} w(e_i^0). \tag{2.8}$$

This property of $B$ and $B'$ implies that if $B$ is an optimum branching of $G$ that satisfies (2.4), then $B'$ is an optimum branching of $G'$ and vice versa. Thus we have proved the following theorem.

**Theorem 2.6** *There exists a one-to-one correspondence between the set of all optimum branchings in $G$ that satisfy* (2.4) *and the set of all optimum branchings in $G'$.* ■

Edmonds' algorithm for constructing an optimum branching is based on Theorem 2.6 and is as follows:

**Algorithm 2.4 Optimum branching (Edmonds)**

**S1.** From the given graph $G = G_0$ construct a sequence of graphs $G_0, G_1, G_2, \ldots, G_k$, where
1. $G_k$ is the first graph in the sequence whose critical subgraph is acyclic and
2. $G_i$, $1 \leq i \leq k$, is obtained from $G_{i-1}$ by contracting the circuits in the critical subgraph $H_{i-1}$ of $G_{i-1}$ and altering the weights as in (2.5).

**S2.** Since $H_k$ is acyclic, it is an optimum branching in $G_k$. Let $B_k = H_k$.
Construct the sequence $B_{k-1}$, $B_{k-2}, \ldots, B_0$, where
1. $B_i$, $0 \leq i \leq k-1$, is an optimum branching of $G_i$.
2. $B_i$, for $i \geq 0$, is constructed by expanding, as in (2.7), pseudo-vertices in $B_{i+1}$.

As an example let $G_0$ be the graph in Figure 2.1a, and let $H_0$ be the graph in Figure 2.1b. $H_0$ is the critical subgraph of $G_0$. After contracting the edges of the circuits in $H_0$ and modifying the weights, we obtain the graph $G_1$ shown in Figure 2.2a. The critical subgraph $H_1$ of $G_1$ is shown in Figure 2.2b. $H_1$ is acyclic. So it is an optimum branching of $G_1$. An optimum branching of $G_0$ is obtained from $H_1$ by expanding the pseudo-vertices $a_1$ and $a_2$ (which correspond to the two directed circuits in $H_0$), and it is shown in Figure 2.2c.

The running time of Edmonds' optimum branching algorithm is $O(mn)$, where $m$ is the number of edges and $n$ is the number of vertices. Tarjan [18] gives an $O(mn \log n)$

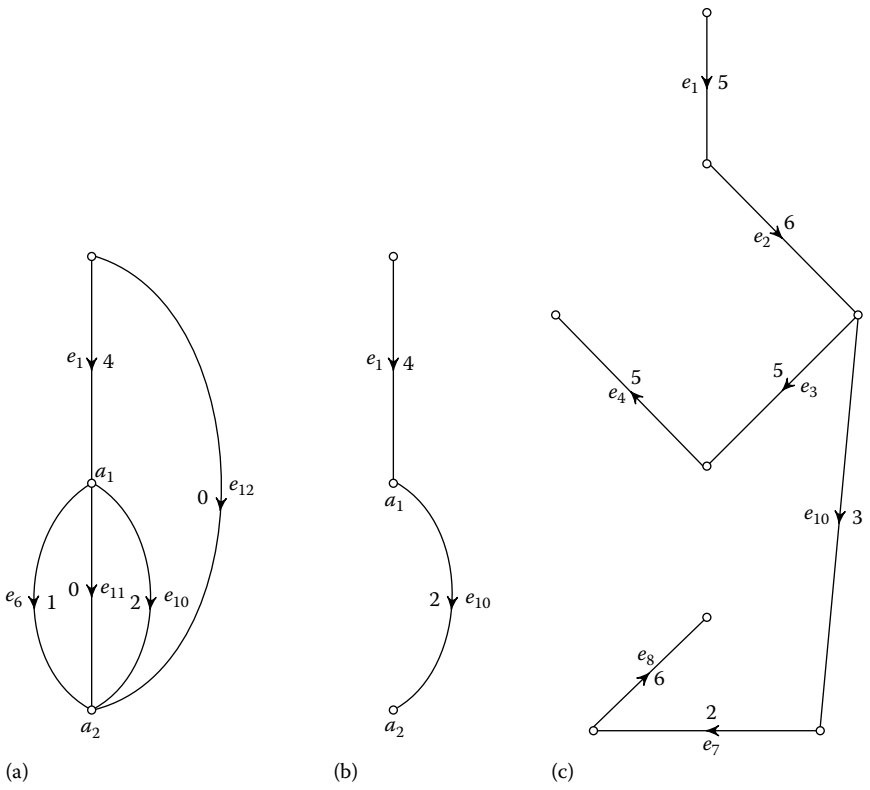


Figure 2.2 (a) Graph $G_1$. (b) $H_1$ a critical subgraph of $G_1$. (c) An optimum branching of graph $G$ of Figure 2.1a. (Data from M. N. Swamy and K. Thulasiraman, *Graphs, Networks and Algorithms*, Wiley-Interscience, 1981.)

implementation of Edmonds' algorithm. Gabow et al. [13] have given an $O(m+n\log n)$ time implementation using Fibonacci heaps (see also Camerini et al. [19]). Bock [20] and Chu and Liu [21] have also independently discovered Edmonds' algorithm.

## 2.4 TRANSITIVE CLOSURE

A binary relation $R$ on a set is a collection of ordered pairs of the elements of the set. If $(x, y) \in R$, then we say that $x$ is related to $y$ and denote this relationship by $x\ R\ y$. A relation $R$ is transitive if $x\ R\ y$ whenever there exists a sequence

$$x_0 = x, x_1, x_2, \ldots, x_k = y$$

such that $k > 0$ and $x_0\ R\ x_1$, $x_1\ R\ x_2, \ldots$, and $x_{k-1}\ R\ x_k$.

The *transitive closure* of a binary relation $R$ is a relation $R^*$ defined as follows: $x\ R^*\ y$ if and only if there exists a sequence

$$x_0 = x, x_1, x_2, \ldots, x_k = y$$

such that $k > 0$ and $x_0\ R\ x_1$, $x_1\ R\ x_2, \ldots$, and $x_{k-1}\ R\ x_k$.

Clearly, if $x\ R\ y$, then $x\ R^*\ y$. Hence $R \subseteq R^*$. Further, it can be easily shown that $R^*$ is transitive. In fact, it is the smallest transitive relation containing $R$. So if $R$ is transitive, then $R^* = R$.

A binary relation $R$ on a set $S$ can be represented by a directed graph $G$ in which each vertex corresponds to an element of $S$ and $(x, y)$ is a directed edge of $G$ if an only if $x\ R\ y$. The directed graph $G^*$ representing the transitive closure $R^*$ of $R$ is called the transitive closure of $G$. It follows from the definition of $R^*$ that the edge $(x, y)$, $x \neq y$, is in $G^*$ if and only if there exists in $G$ a directed path from the vertex $x$ to the vertex $y$. Similarly the self-loop $(x, x)$ at vertex $x$ is in $G^*$ if and only if there exists in $G$ a directed circuit containing $x$. For example, the graph shown in Figure 2.3b is the transitive closure of the graph of Figure 2.3a.

Suppose that we define the *reachability matrix* of an $n$-vertex directed graph $G$ as an $n \times n$ (0, 1) matrix in which the $(i, j)$ entry is equal to 1 if and only if there exists a directed path from vertex $i$ to vertex $j$ when $i \neq j$, or a directed circuit containing vertex $i$ when $i = j$. In other words the $(i, j)$ entry of the reachability matrix is equal to 1 if and only if vertex $j$ is reachable from vertex $i$ through a path of directed edges in $G$. The problem of constructing the transitive closure of a directed graph arises in several applications (e.g., see Gries [22]). In this section we discuss an elegant and computationally efficient algorithm due to Warshall [23] for computing the transitive closure of a directed graph. We also discuss a variation of Warshall's algorithm given by Warren [24].

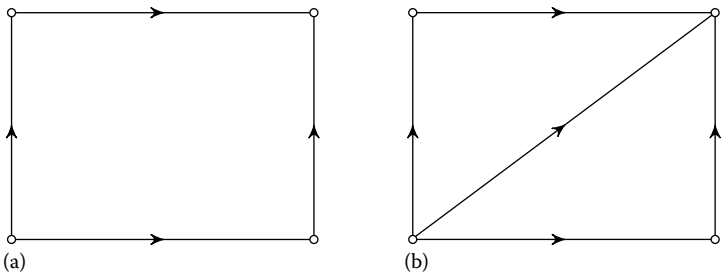


Figure 2.3 (a) Graph $G$. (b) $G^*$, transitive closure of $G$. (Data from M. N. Swamy and K. Thulasiraman, *Graphs, Networks and Algorithms*, Wiley-Interscience, 1981.)

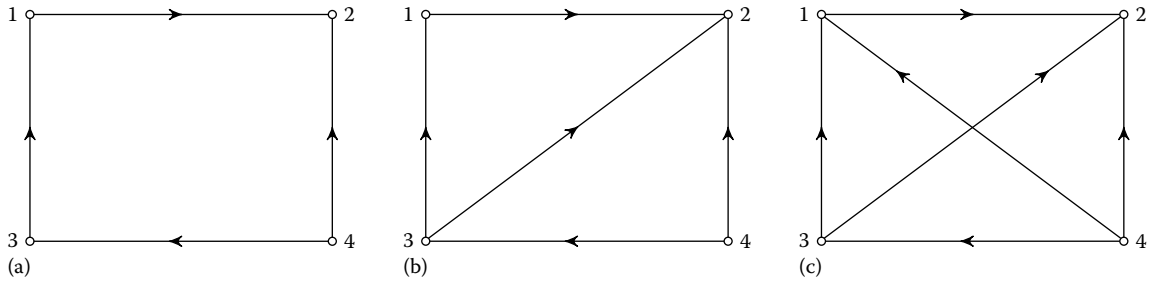


Figure 2.4 Illustration of Warshall's algorithm. (a) $G^0$. (b) $G^1 = G^2$. (c) $G^3 = G^4$. (Data from M. N. Swamy and K. Thulasiraman, *Graphs, Networks and Algorithms*, Wiley-Interscience, 1981.)

Let $G$ be an $n$-vertex directed graph with its vertices denoted by the integers $1, 2, \ldots, n$. Let $G^0 = G$. Warshall's algorithm constructs a sequence of graphs so that $G^i \subseteq G^{i+1}$, $0 \leq i \leq n-1$, and $G^n$ is the transitive closure of $G$. The graph $G^i$, $i \geq 1$, is obtained from $G^{i-1}$ by processing vertex $i$ in $G^{i-1}$. Processing vertex $i$ in $G^{i-1}$ involves addition of new edges to $G^{i-1}$ as described next.

Let, in $G^{i-1}$, the edges $(i,k)$, $(i,l)$, $(i,m),\ldots$ be incident out of vertex $i$. Then for each edge $(j,i)$ incident into vertex $i$, add to $G^{i-1}$ the edges $(j,k)$, $(j,l)$, $(j,m),\ldots$ if these edges are not already present in $G^{i-1}$. The graph that results after vertex $i$ is processed is denoted as $G^i$. Warshall's algorithm is illustrated in Figure 2.4.

It is clear that $G^i \subseteq G^{i+1}$, $i \geq 0$. To show that $G^n$ is the transitive closure of $G$ we need to prove the following result.

**Theorem 2.7**

1. *Suppose that, for any two vertices $s$ and $t$, there exists in $G$ a directed path $P$ from vertex $s$ to vertex $t$ such that all its vertices other than $s$ and $t$ are from the set $\{1, 2, \ldots, i\}$. Then $G^i$ contains the edge $(s,t)$.*

2. *Suppose that, for any vertex $s$, there exists in $G$ a directed circuit $C$ containing $s$ such that all its vertices other than $s$ are from the set $\{1, 2, \ldots, i\}$. Then $G^i$ contains the self-loop $(s,s)$.*

*Proof.*

1. Proof is by induction on $i$.

   Clearly the result is true for $G^1$ since Warshall's construction, while processing vertex 1, introduces the edge $(s,t)$ if $G^0 (= G)$ contains the edges $(s,1)$ and $(1,t)$.

   Let the result be true for all $G^k$, $k < i$.

   Suppose that $i$ is not an internal vertex of $P$. Then it follows from the induction hypothesis that $G^{i-1}$ contains the edge $(s,t)$. Hence $G^i$ also contains $(s,t)$ because $G^{i-1} \subseteq G^i$.

   Suppose that $i$ is an internal vertex of $P$. Then again from the induction hypothesis it follows that $G^{i-1}$ contains the edges $(s,i)$ and $(i,t)$. Therefore, while processing vertex $i$ in $G^{i-1}$, the edge $(s,t)$ is added to $G^i$.

2. Proof follows along the same lines as in (1). ■

As an immediate consequence of this theorem we get the following corollary.

**Corollary 2.2** *$G^n$ is the transitive closure of $G$.* ■

We next give a formal description of Warshall's algorithm. In this description the graph $G$ is represented by its adjacency matrix $M$ and the symbol $\vee$ stands for Boolean addition.

**Algorithm 2.5 Transitive closure (Warshall)**

***Input:*** $M$ is the adjacency matrix of $G$.
***Output:*** Transitive closure of $G$.

```
begin
for i = 1, 2, ..., n do
    begin
        for j = 1, 2, ..., n do
            if M(j, i) = 1 then
                begin
                    for k = 1, 2, ..., n do
                        M(j, k) ← M(j, k) ∨ M(i, k);
                end
    end
end
```

A few observations are now in order:

1. Warshall's algorithm transforms the adjacency matrix $M$ of a graph $G$ to the adjacency matrix of the transitive closure of $G$ by suitably overwriting on $M$. It is for this reason that the algorithm is said to work *in place*.

2. The algorithm processes all the edges incident into a vertex before it begins to process the next vertex. In other words it processes the matrix $M$ column-wise. Hence, we describe Warshall's algorithm as *column-oriented*.

3. While processing a vertex no new edge (i.e., an edge that is not present when the processing of that vertex begins) incident into the vertex is added to the graph. This means that while processing a vertex we can choose the edges incident into the vertex in any arbitrary order.

4. Suppose that the edge $(j, i)$ incident into the vertex $i$ is not present while vertex $i$ is processed, but that it is added subsequently while processing some vertex $k$, $k > i$. Clearly this edge was not processed while processing vertex $i$. Neither will it be processed later since no vertex is processed more than once. In fact, such an edge will not result in adding any new edges.

5. Warshall's algorithm is said to work in one pass since each vertex is processed exactly once.

Suppose that we wish to modify Warshall's algorithm so that it becomes *row-oriented*. In a row-oriented algorithm, while processing a vertex, all the edges incident out of the vertex are to be processed. The processing of the edge $(i, j)$ introduces the edges $(i, k)$ for every edge $(j, k)$ incident out of vertex $j$. Therefore new edges incident out of a vertex may be added while processing a vertex *row-wise*. Some of these newly added edges may not be processed before the processing of the vertex under consideration is completed. If the processing of these edges is necessary for the computation of the transitive closure, then such a processing

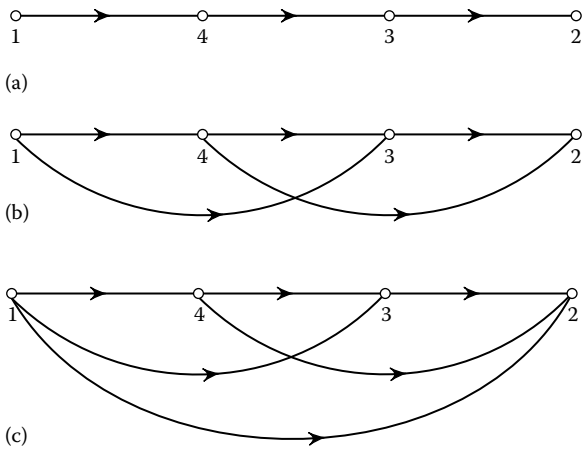


Figure 2.5 Example of row-oriented transitive closure algorithm: (a) $G$, (b) $G'$, and (c) $G^*$. (Data from M. N. Swamy and K. Thulasiraman, *Graphs, Networks and Algorithms*, Wiley-Interscience, 1981.)

can be done only in a second pass. Thus, in general, a row-oriented algorithm may require more than one pass to compute the transitive closure.

For example, consider the graph $G$ of Figure 2.5a. After processing row-wise the vertices of $G$ we obtain the graph $G'$ shown in Figure 2.5b. Clearly, $G'$ is not the transitive closure of $G$ since the edge (1,2) is yet to be added. It may be noted that the edge (1,3) is not processed in this pass because it is added only while processing the edge (1,4). The same is the case with the edge (4,2).

Suppose we next process the vertices of $G'$. In this second pass the edge (1,2) is added while processing vertex 1 and we get the transitive closure $G^*$ shown in Figure 2.5c. Thus in the case of the graph of Figure 2.5a two passes of the row-oriented algorithm are required.

Now the question arises whether two passes always suffice. The answer is in the affirmative, and Warren [24] has demonstrated this by devising a clever two-pass row-oriented algorithm. In this algorithm, while processing a vertex, say vertex $i$, in the first pass only edges connected to vertices less than $i$ are processed, and in the second pass only edges connected to vertices greater than $i$ are processed. In other words the algorithm transforms the adjacency matrix $M$ of the graph $G$ to the adjacency matrix of $G^*$ by processing in the first pass only entries below the main diagonal of $M$ and in the second pass only entries above the main diagonal. Thus during each pass at most $n(n-1)/2$ edges are processed. A description of Warren's modification of Warshall's algorithm now follows.

**Algorithm 2.6 Transitive closure (Warren)**

***Input:*** $M$ is the adjacency matrix of $G$.
***Output:*** Transitive closure of $G$.

```
begin
    for i = 2, 3, ..., n do {Pass 1 begins}
        begin
            for j = 1, 2, ..., i − 1 do
                if M(i, j) = 1 then
                    begin
                        for k = 1, 2, ..., n do
                            M(i, k) ← M(i, k) ∨ M(j, k);
                    end
```

```
        end
        {Pass 1 ends}
    for i = 1, 2, ..., n − 1 do {Pass 2 begins}
        begin
            for j = i + 1, i + 2, ..., n do
                if M(i, j) = 1 then
                    begin
                        for k = 1, 2, ..., n do
                            M(i, k) ← M(i, k) ∨ M(j, k);
                    end
        end
        {Pass 2 ends}
end
```

As an example, consider again the graph shown in Figure 2.5a. At the end of the first pass of Warren's algorithm we obtain the graph shown in Figure 2.6a, and at the end of the second pass we get the transitive closure $G^*$ shown in Figure 2.6b. The proof of correctness of Warren's algorithm is based on the following lemma.

**Lemma 2.3** *Suppose that, for any two vertices $s$ and $t$, there exists in $G$ a directed path $P$ from $s$ to $t$. Then the graph that results after processing vertex $s$ in the first pass of Warren's algorithm contains an edge $(s, r)$, where $r$ is a successor of $s$ on $P$ and either $r > s$ or $r = t$.*

*Proof.* Proof is by induction on $s$.

If $s = 1$, then the lemma is clearly true because all the successors of 1 on $P$ are greater than 1. Assume that the lemma is true for all $s < k$ and let $s = k$. Suppose $(s, i_1)$ is the first edge on $P$. If $i_1 > s$, then clearly the lemma is true.

If $i_1 < s$, then by the induction hypothesis the graph that results after processing vertex $i_1$ in the first pass contains an edge $(i_1, i_2)$, where $i_2$ is a successor of $i_1$ on $P$ and either $i_2 > i_1$ or $i_2 = t$.

If $i_2 \neq t$ and $i_2 < s$, then again by the induction hypothesis the graph that results after processing vertex $i_2$ in the first pass contains an edge $(i_2, i_3)$ where $i_3$ is a successor of $i_2$ on $P$, and either $i_3 > i_2$ or $i_3 = t$.

If $i_3 \neq t$ and $i_3 < s$, we repeat the arguments on $i_3$ until we locate an $i_m$ such that either $i_m > s$ or $i_m = t$. Thus the graph that we have before the processing of vertex $s$ begins contains the edges $(s, i_1)$, $(i_1, i_2), \ldots, (i_{m-1}, i_m)$ such that the following conditions are satisfied:

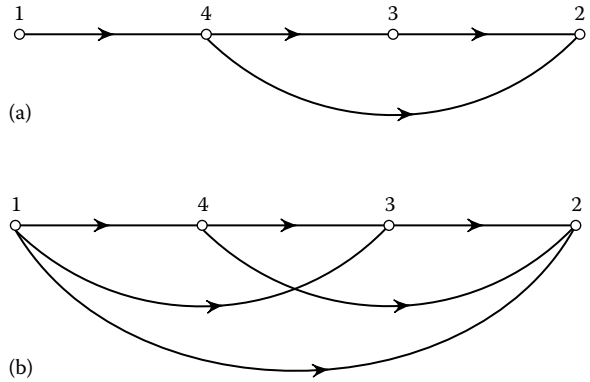


Figure 2.6 Illustration of Warren's algorithm. (Data from M. N. Swamy and K. Thulasiraman, *Graphs, Networks and Algorithms*, Wiley-Interscience, 1981.)

1. $i_p$ is a successor of $i_{p-1}$ on $P$, $p \geq 2$.
2. $i_{m-1} > i_{m-2} > i_{m-3} > \cdots > i_1$, and $i_k < s$ for $k \neq m$.
3. $i_m = t$ or $i_m > s$.

We now begin to process vertex $s$. Processing of $(s, i_1)$ introduces the edge $(s, i_2)$ because of the presence of $(i_1, i_2)$. Since $i_2 > i_1$, the edge $(s, i_2)$ is subsequently processed. Processing of this edge introduces $(s, i_3)$ because of the presence of $(i_2, i_3)$, and so on. Thus when the processing of $s$ is completed, the required edge $(s, i_m)$ is present in the resulting graph. ■

**Theorem 2.8** *Warren's algorithm computes the transitive closure of a directed graph $G$.*

*Proof.* We need to consider the following two cases.

*Case* 1 For any two distinct vertices $s$ and $t$ there exists in $G$ a directed path $P$ from $s$ to $t$.

Let $(i, j)$ be the first edge on $P$ (as we proceed from $s$ to $t$) such that $i > j$. Then it follows from Lemma 2.3 that the graph that we have, before the second pass of Warren's algorithm begins, contains an edge $(i, k)$, where $k$ is a successor of $i$ on $P$ and either $k = t$ or $k > i$. Thus after the first pass is completed there exists a path $P'$ : $s$, $i_1$, $i_2, \ldots$, $i_m$, $t$ such that $s < i_1 < i_2 < \cdots < i_m$ and each $i_{j+1}$ is a successor of $i_j$ on $P$.

When in the second pass we process vertex $s$, the edge $(s, i_1)$ is first encountered. The processing of this edge introduces the edge $(s, i_2)$ because of the presence of the edge $(i_1, i_2)$. Since $i_2 > i_1$, the edge $(s, i_2)$ is processed subsequently. This, in turn, introduces the edge $(s, i_3)$, and so on. Thus when the processing of $s$ is completed, we have the edge $(s, t)$ in the resulting graph.

*Case* 2 There exists in $G$ a directed circuit containing a vertex $s$.

In this case we can prove along the same lines as before that when the processing of vertex $s$ is completed in the second pass, the resulting graph contains the self-loop $(s, s)$. ■

Clearly both Warshall's and Warren's algorithms have the worst-case complexity $O(n^3)$. Warren [24] refers to other row-oriented algorithms. For some of the other transitive closure algorithms (see [25–33]). Syslo and Dzikiewicz [34] discuss computational experiences with several of the transitive closure algorithms. Melhorn [35] discusses the transitive closure problem in the context of general path problems in graphs. Several additional references on this topic can also be found in [35].

## 2.5 SHORTEST PATHS

Let $G$ be a connected directed graph in which each directed edge is associated with a finite real number called the length of the edge. The length of an edge directed from a vertex $i$ to a vertex $j$ is denoted by $w(i, j)$. If there is no edge directed from vertex $i$ to vertex $j$, then $w(i, j) = \infty$. The length of a directed path in $G$ is the sum of the lengths of the edges in the path. A minimum length directed $s - t$ path is called a shortest path from $s$ to $t$. The length of a shortest directed $s - t$ path, if it exists, is called the distance from $s$ to $t$, and it is denoted as $d(s, t)$. We assume that the vertices are denoted as $1, 2, \ldots, n$.

In this section we consider the following two problems:

1. *Single Source Shortest Paths Problem*: Find the shortest paths from a specified vertex $s$ to all other vertices in $G$.
2. *All-Pairs Shortest Paths Problem*: Find the shortest paths between all the ordered pairs of vertices in $G$.

These two problems arise in several optimization problems. For example, finding a minimum cost flow in a transport network involves finding a shortest path from the source to the sink in the network [36] (see also Chapter 5).

### 2.5.1 Single Source Shortest Paths: Bellman–Ford–Moore Algorithm

We first make a few observations and assumptions that do not cause any loss of generality of the algorithms to be discussed in this section.

- Vertex 1 will serve as the source vertex. Algorithms for the shortest path problems start with an initial estimate of the distance of each vertex from the source vertex 1. We will denote this estimate for vertex $i$ by $\lambda(i)$. Initially, $\lambda(1) = 0$ and $\lambda(i) = \infty$ for all $i \neq 1$. Algorithms repeatedly update these estimates until they all become the required distance values. That is, at termination, $d(1, i) = \lambda(i)$.
- We assume that all vertices are reachable from the source vertex. In other words, we assume that there is a directed path from the source vertex to every other vertex. Thus, at termination of the algorithms, $\lambda(i)$ values will be finite.
- Algorithms search for a shortest directed walk from the source vertex to every other vertex. We assume that the graph has no directed circuit of negative length. If such a circuit $C$ were present, then a directed walk $P$ of arbitrarily small length from the source vertex to every vertex in $C$ can be found by repeatedly traversing the circuit $C$.

The following theorem is the basis of all shortest path algorithms that we will be discussing in this section. In this theorem and the subsequent discussions, an $i - j$ directed path refers to a directed path from vertex $i$ to vertex $j$.

**Theorem 2.9 (Optimality conditions)** *Consider a connected directed graph $G = (V, E)$ with each edge $e \in E$ associated with a finite real number $w(e)$. Let $\lambda(i)$, $i \geq 1$ denote the length of a $1 - i$ directed path. Then for all $i \geq 1$, $\lambda(i)$ is the distance from vertex 1 to vertex $i$, that is $\lambda(i) = d(1, i)$, if and only if the following condition is satisfied.*

$$\lambda(i) + w(i, j) \geq \lambda(j), \text{for every edge } e = (i, j). \tag{2.9}$$

*Proof.*
*Necessity*: Let $\lambda(i)$, $i \geq 1$ be the distance from vertex 1 to vertex $i$. Suppose condition (2.9) is violated for some edge $e = (i, j)$. Then, $\lambda(i) + w(e) < \lambda(j)$. Since there is no directed circuit of negative length, concatenating the edge $e$ to a shortest path from vertex 1 to vertex $i$ will give a path of length less than $\lambda(j)$, contradicting that $\lambda(j)$ is the length of a shortest path from vertex 1 to vertex $i$.

*Sufficiency*: Consider a directed path $P$ from vertex 1 to vertex $j$ and let

$$P : 1 = i_0, i_1, \ldots, i_{k-1}, i_k = j.$$

Assume that the $\lambda(i)$'s satisfy (2.9).

Then

$$\begin{aligned}
\lambda(j) = \lambda(i_k) &\leq \lambda(i_{k-1}) + w(i_{k-1}, i_k) \\
&\leq \lambda(i_{k-2}) + w(i_{k-2}, i_{k-1}) + w(i_{k-1}, i_k) \ldots \\
&\leq \lambda(i_0) + w(i_0, i_1) + w(i_1, i_2) + \cdots + w(i_{k-2}, i_{k-1}) + w(i_{k-1}, i_k) \\
&= w(i_0, i_1) + w(i_1, i_2) + \cdots + w(i_{k-2}, i_{k-1}) + w(i_{k-1}, i_k), \\
&\qquad \text{since} \qquad i_0 = 1 \text{ and } \lambda(1) = 0. \\
&= \text{length of path } P.
\end{aligned}$$

Since the length of every directed path $P$ from vertex 1 to vertex $j$ is at least $\lambda(j)$, and $\lambda(j)$ is the length of a directed path from vertex 1 to vertex $j$, it follows that $\lambda(j)$ is the length of a shortest path from vertex 1 to vertex $j$. ■

We are now ready to present a shortest path algorithm due to Ford [37]. This algorithm is also attributed to Bellman [38] and Moore [39]. So we shall call this Bellman–Ford–Moore (in short, BFM) algorithm.

**Algorithm 2.7 Shortest paths in graphs with negative length edges (Bellman–Ford–Moore)**

***Input:*** A connected graph $G = (V, E)$ with length $w(e) = w(i, j)$ for each edge $e = (i, j)$.
***Output:*** Shortest paths and their lengths from vertex 1 to all other vertices.

```
begin
  λ(1) ← 0;
  λ(i) ← ∞, for each i ≠ 1;
  PRED (i) ← i, for every vertex i;
  while there exists an edge e = (i, j) satisfying λ(i) + w(i, j) < λ(j) do
    begin
      λ(j) ← λ(i) + w(i, j);
      PRED (j) ← i;
    end
end
```

Note that the algorithm starts by assigning $\lambda(1) = 0$ and $\lambda(j) = \infty$ for all $j \neq 1$. If there exists an edge $(i, j)$ which violates the optimality condition (2.9) then the $\lambda(j)$ value is updated to $\lambda(j) = \lambda(i) + w(i, j)$ and the predecessor of $j$ is set to $i$. If for every edge $(i, j)$ the optimality condition is satisfied then the algorithm terminates.

A few observations are now in order.

- If at any iteration $\lambda(j)$ is finite and $\text{PRED}(j) = i$ then it means that vertex $j$ received its current label $\lambda(j)$ while examining the edge $(i, j)$ and $\lambda(j)$ is updated to $\lambda(i) + w(i, j)$. Since there are no directed circuits of negative length in the graph, a directed $1-j$ path of length $\lambda(i) + w(i, j)$ can then be obtained by tracing the path backward from vertex $j$ along the predecessors. For example, if $\text{PRED}(8) = 6$, $\text{PRED}(6) = 9$ and $\text{PRED}(9) = 1$, then the 1 – 8 path along the edges (1,9), (9,6), and (6,8) is of length $\lambda(8)$.

- Since there are no directed circuits of negative length, the number of times the label of a vertex $j$ is updated is no more than the number of directed paths from vertex 1 to vertex $j$. Since the number of directed paths is finite, the BFM algorithm will terminate, and at termination the label values will satisfy (2.9).

This establishes the correctness of the BFM algorithm. We summarize this result in the following theorem.

**Theorem 2.10** *For a directed graph $G$ with no directed circuits of negative length, the Bellman–Ford–Moore Algorithm terminates in a finite number of steps, and upon termination, $\lambda(j) = d(1, j)$ for every vertex $j$.* ■

Note that the number of directed paths from vertex 1 to another vertex could be exponential in the number of vertices of the graph. So, if in the implementation of Algorithm 2.7 edges are selected in an arbitrary manner, it is possible that the algorithm may perform an exponential number of operations before terminating. We next present an implementation which results in $O(mn)$ time complexity, where $m$ denotes the number of edges and $n$ denotes the number of vertices in the network.

**Step 1**: First arrange the edges in any specified order as follows: $e_1, e_2, e_3, \ldots, e_m$

**Step 2**: Scan the edges one by one and check if the condition $\lambda(j) > \lambda(i) + w(i,j)$ is satisfied for edge $(i,j)$. If so, update $\lambda(j) = \lambda(i) + w(i,j)$.

**Step 3**: Stop if no $\lambda(j)$ is updated in step 2. Otherwise, repeat step 2.

In the above implementation Step 2 is called a *sweep* or a *phase*. During a phase, $m$ edges are scanned and the $\lambda(j)$ values are updated if necessary. Thus a sweep takes $O(m)$ time. We now show that the algorithm terminates in atmost $n$ sweeps. We first prove the following.

**Theorem 2.11** *If there exists a shortest path from vertex* 1 *to vertex* $j$ *having* $k$ *edges, then* $\lambda(j)$ *will have reached its final value by the end of the* $k$*th sweep.*

*Proof.* Proof is by induction on $k$. Clearly the result is true for $k = 1$, because the vertex $j$ will get its final $\lambda(j)$ value when the edge $(1,j)$ is scanned during the first sweep. As induction hypothesis, assume that the result is true for some $k \geq 1$. In other words, we assume that if there exists a shortest path from vertex 1 to vertex $j$ having $k$ edges, then at the end of the $k$th sweep, $\lambda(j)$ will have reached its final value.

Consider a vertex $j$ which is connected to 1 by a shortest path having $(k+1)$ edges.

$$P : 1, i_1, i_2, i_3, \ldots, i_k, i_{k+1} = j$$

Since the subpath from 1 to $i_k$ is a shortest path to $i_k$ having $k$ edges, then by the induction hypothesis $\lambda(i_k)$ will have reached its final value by the end of the $k$th sweep. During the $(k+1)$th sweep when edge $(i_k, i_{k+1})$ is scanned $\lambda(i_{k+1})$ will be updated to $\lambda(i_{k+1}) = \lambda(i_k) + w(i_k, i_{k+1})$ which is the length of $P$. Thus at the end of the $(k+1)$th sweep, $\lambda(i_{k+1}) = d(1,j)$ which is its final value. ■

Now we can see that the BFM algorithm will terminate in at most $n$ sweeps because every path has at most $(n-1)$ edges. So the complexity of the above implementation of the BFM algorithm is $O(mn)$. We call this implementation of the BFM as edge-based implementation.

In a similar manner, we can design a vertex-based implementation of the BFM algorithm. In this implementation, we first order the vertices as $1, 2, \ldots, n$. Recall that vertex 1 is the source vertex. During a sweep each vertex is examined in the order specified by the vertex ordering. Here, examining a vertex $j$ involves examining all the edges incident on and directed away from $j$ and updating the labels of the neighbors according to step 2. We can again show that no more than $n$ sweeps will be required and that the algorithm terminates in $O(mn)$ time. We will call this vertex-based implementation.

#### 2.5.1.1 Negative Cycle Detection

As we noted earlier, $\lambda(j)$'s satisfying the optimality conditions do not exist if there exists a directed circuit of negative length. So, if a negative directed were present, the BFM algorithm will not terminate. Suppose $C$ denotes the maximum of the absolute values of all edge lengths, no path can have a cost smaller than $-nC$. Thus if the $\lambda(j)$ value falls below $-nC$ for some node $j$, then we can terminate the algorithm. The negative length circuit can be obtained by tracing the predecessor values starting at node $j$.

#### 2.5.1.2 Shortest Path Tree

At the end of the shortest path algorithm each node $j$ has a predecessor PRED$(j)$. The set of edges $(j, \text{PRED}(j))$ will form a tree called the shortest path tree. Each path from 1 to

node $j$ in the tree gives a shortest length path from 1 to $j$. Also, $\lambda(j) = \lambda(i) + w(i, j)$ for every edge $(i, j)$ on this tree.

Sometimes we may be interested in getting the second, third, or higher shortest paths. These and related problems are discussed in Christofides [40], Dreyfus [41], Frank and Frisch [42], Gordon and Minoux [43], Hu [44], Lawler [45], Minieka [46], and Spira and Pan [47]. For algorithms designed for sparse networks see Johnson [48] and Wagner [49] (see also Edmonds and Karp [50], Fredman [51], and Johnson [52]). For a shortest path problem that arises in solving a special system of linear inequalities and its applications see Comeau and Thulasiraman [53], Lengauer [54], and Liao and Wong [55].

### 2.5.2 Single Source Shortest Paths in Graphs with No Negative Length Edges: Dijkstra's Algorithm

In this section we consider a special case of the shortest path problem where all the edge lengths are nonnegative. This restricted version of the shortest path problem admits a very efficient algorithm due to Dijkstra [56]. This algorithm may be viewed as a vertex-based implementation of the BFM algorithm. This implementation requires only one sweep of all the vertices, but in contrast to the BFM algorithm the order in which the vertices are selected for scanning cannot be arbitrary. This order depends on the edge lengths. Following is an informal description of Dijkstra's algorithm.

Like the BFM algorithm, Dijkstra's algorithm starts by assigning $\lambda(1) = 0$ and $\lambda(j) = \infty$ for all $j \neq 1$. Initially, all vertices are unlabeled.

**A general iteration:** In iteration $i$

- An unlabeled vertex $i$ with minimum $\lambda$-value is labeled *Permanent*. Let this vertex be denoted as $u_i$, that is, among all unlabeled vertices vertex $u_i$ has the smallest $\lambda$-value.
- Then every edge $(u_i, j)$ where $j$ is unlabeled is examined for violation of the optimality condition (2.9). If there exists such an edge $(u_i, j)$, then the $\lambda(j)$ value is updated to $\lambda(j) = \lambda(u_i) + w(u_i, j)$ and the predecessor of $j$ is set to $u_i$.

The algorithm terminates when all the vertices are permanently labeled. A formal description of Dijkstra's algorithm is given next. In this description, we use an array PERM to indicate which of the vertices are permanently labeled. If PERM$(v) = 1$, then $v$ is a permanently labeled vertex. We start with PERM$(v) = 0$ for all $v$. PRED is an array that keeps a record of the vertices from which the vertices get permanently labeled. If a vertex $v$ is permanently labeled, then $v$, PRED$(v)$, PRED(PRED$(v)$), ..., 1 are the vertices in a shortest directed 1-$v$ path.

**Algorithm 2.8 Shortest paths in graphs with non-negative edge lengths (Dijkstra)**

***Input:*** A connected graph $G = (V, E)$ with length $w(e) = w(i, j) \geq 0$ for each edge $e = (i, j)$.
***Output:*** Shortest paths and their lengths from vertex 1 to all other vertices.
**begin**
  $\lambda(1) \leftarrow 0$;
  $\lambda(i) \leftarrow \infty$, for each $i \neq 1$;
  PRED$(i) \leftarrow i$, for every vertex $i$;
  PERM$(i) \leftarrow 0$, for every vertex $i$;
  $S_0 \leftarrow \phi$;
  **for** $k = 1, 2, \ldots, n$ **do**

**begin (iteration $k$ begins)**
Let $u_k$ be a vertex that is not yet labeled permanently and has minimum λ-value;
PERM$(u_k) \leftarrow 1$;
$S_k \leftarrow S_{k-1} \cup \{u_k\}$;
**for** every edge $e = (u_k, j)$ **do**
**If** $\lambda(j) > \lambda(u_k) + w(u_k, j)$, **then** $\lambda(j) \leftarrow \lambda(u_k) + w(u_k, j)$ and
PRED$(j) \leftarrow u_k$;
**end (iteration $k$ ends)**
**end**

Note that in a computer program $\infty$ is represented by as high a number as necessary. Further, if the final λ value of a vertex $v$ is equal to $\infty$, then it means that there is no directed path from $s$ to $v$.

To illustrate Dijkstra's algorithm, consider the graph $G$ in Figure 2.7 in which the length of an edge is shown next to the edge. In Figure 2.8 we have shown the λ values of vertices and the entries of the PRED array.

For any $i$ the circled entries correspond to the permanently labeled vertices. The entry with the mark * is the label of the latest permanently labeled vertex $u_i$. The shortest paths from $s$ and the corresponding distances are obtained from the final λ values and the entries in the PRED array.

**Theorem 2.12** *For $i \geq 1$, let $u_i$ denote the vertex that is labeled permanently in iteration $i$ of Dijkstra's algorithm, and $d(u_i)$ be the λ-value of $u_i$ at the termination of the algorithm. Then*

1. $d(u_1) \leq d(u_2) \leq \cdots \leq d(u_n)$; *and*
2. $d(1, u_j) = d(u_i)$ *for all* $j \geq 1$.

Note: In Dijkstra's algorithm the λ-value of a vertex does not change once the vertex is labeled permanently.

*Proof.* Proof depends on the following observations based on the way labels are updated and how a vertex is selected for permanent labeling.

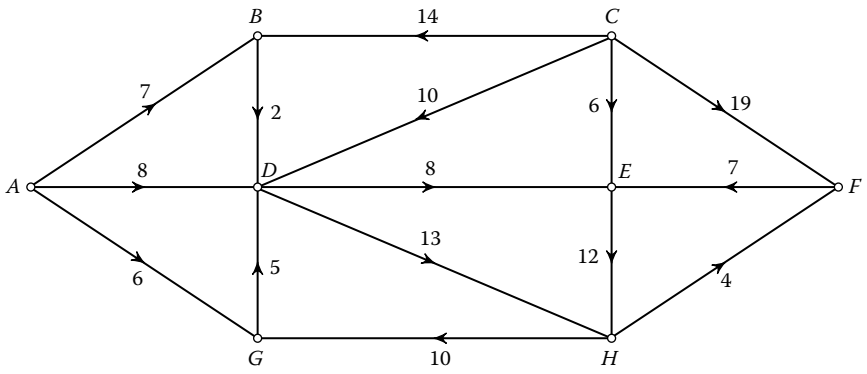


Figure 2.7 Graph for illustrating Dijkstra's algorithm. (Data from M. N. Swamy and K. Thulasiraman, *Graphs, Networks and Algorithms*, Wiley-Interscience, 1981.)

| | Vertices | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $i$ | $A$ | $B$ | $C$ | $D$ | $E$ | $F$ | $G$ | $H$ |
| 1 | ⓪* | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| 2 | ⓪ | 7 | ∞ | 8 | ∞ | ∞ | ⑥* | ∞ |
| 3 | ⓪ | ⑦* | ∞ | 8 | ∞ | ∞ | ⑥ | ∞ |
| 4 | ⓪ | ⑦ | ∞ | ⑧* | ∞ | ∞ | ⑥ | ∞ |
| 5 | ⓪ | ⑦ | ∞ | ⑧ | ⑯* | ∞ | ⑥ | 21 |
| 6 | ⓪ | ⑦ | ∞ | ⑧ | ⑯ | ∞ | ⑥ | ㉑* |
| 7 | ⓪ | ⑦ | ∞ | ⑧ | ⑯ | ㉕* | ⑥ | ㉑ |
| 8 | ⓪ | ⑦ | ∞* | ⑧ | ⑯ | ㉕ | ⑥ | ㉑ |

(a)

PRED($A$) = $A$ PRED($E$) = $D$
PRED($B$) = $A$ PRED($F$) = $H$
PRED($C$) = $C$ PRED($G$) = $A$
PRED($D$) = $A$ PRED($H$) = $D$

| From | To | Shortest path |
|---|---|---|
| $A$ | $B$ | $A, B$ |
| $A$ | $C$ | No Path |
| $A$ | $D$ | $A, D$ |
| $A$ | $E$ | $A, D, E$ |
| $A$ | $F$ | $A, D, H, F$ |
| $A$ | $G$ | $A, G$ |
| $A$ | $H$ | $A, D, H$ |

(b)

Figure 2.8 Illustration of Dijkstra's algorithm. The λ values are shown in (a). (Data from M. N. Swamy and K. Thulasiraman, *Graphs, Networks and Algorithms*, Wiley-Interscience, 1981.)

1. Let $S_i$ denote the set of all the vertices permanently labeled at the end of iteration $i$. That is $S_i = \{u_1, u_2, \ldots, u_i\}$. Then, at the end of iteration $i$, $\lambda(j)$ of a vertex $j \neq S_i$, if it is finite, is the minimum of the labels considered for assignment to $j$ while scanning the edges directed into $j$ from the permanently labeled vertices $u_1, u_2, \ldots, u_i$. So,

$$\lambda(j) \leq d(u_k) + w(u_k, j), \text{ for all } \; j \neq S_i \text{ and } k \leq i \tag{2.10}$$

2. $u_{i+1}$ is the vertex not in $S_i$ with minimum λ-value. This means that

$$\lambda(u_{i+1}) = d(u_{i+1}) \leq d(u_k) + w(u_k, u_{i+1}), \text{ for all } k \leq i \tag{2.11}$$

PRED($u_{i+1}$) is the vertex $u_k$ for which the above minimum is achieved. So, if PRED($u_{i+1}$) = $u_a$, then

$$d(u_{i+1}) = d(u_a) + w(u_a, u_{i+1}). \tag{2.12}$$

3. It follows from (2.10) that for $j > i$

$$d(u_j) \leq d(u_i) + w(u_i, u_j) \tag{2.13}$$

because λ-values of vertices do not increase from iteration to iteration.

*Proof of* (1): If $a = i$, then from (2.12) we get $d(u_{i+1}) = d(u_i) + w(u_i, u_{i+1})$ and so $d(u_i) \leq d(u_{i+1})$.

If $a < i$, then vertex $u_{i+1}$ must have reached its final label value $d(u_{i+1}) = d(u_a) + w(u_a, u_{i+1})$ in iteration $a$.

So, in the iteration $i$ when $u_i$ with label $d(u_i)$ was selected for permanent labeling, the vertex $u_{i+1}$ with label $d(u_{i+1}) = d(u_a) + w(u_a, u_{i+1})$ was also available for consideration. But vertex $u_i$ was permanently labeled. So, $d(u_i) \leq d(u_{i+1})$.

Result (1) follows since the above argument is valid for all $i \geq 1$.

*Proof of* (2): We prove the result by showing that $d(u_1), d(u_2), \ldots, d(u_n)$ satisfy the optimality condition for shortest path lengths given in Theorem 2.9.

Consider any edge $(u_i, u_j)$. If $i > j$, then $d(u_j) \leq d(u_i)$, by the result (1) of the theorem. So $d(u_i)+w(u_i, u_j) \geq d(u_j)$. Suppose that $i < j$. Then, we see from (2.13) that $d(u_i)+w(u_i, u_j) \geq d(u_j)$.

Thus, the labels $d(u_1), d(u_2), \ldots, d(u_n)$ at the termination of Dijkstra's algorithm satisfy the optimality condition for shortest path lengths and the result (2) follows. ■

Dijkstra's algorithm requires examination of at most $m$ edges and at most $n$ minimum computations, leading to a complexity of $O(m + n \log n)$. In our discussions thus far we have assumed that all the lengths are nonnegative. Dijkstra's algorithm is not valid if some of the lengths are negative. (Why?)

### 2.5.3 All Pairs Shortest Paths

Suppose that we are interested in finding the shortest paths between all the $n(n-1)$ ordered pairs of vertices in an $n$-vertex directed graph. A straight-forward approach to get these paths would be to use the BFM algorithm $n$ times. However, there are algorithms that are computationally more efficient than this. These algorithms are applicable when there are no negative-length directed circuits. Now we discuss one of these algorithms. This algorithm, due to Floyd [57], is based on Warshall's algorithm (Algorithm 2.5) for computing transitive closure.

Consider an $n$-vertex directed graph $G$ with lengths associated with its edges. Let the vertices of $G$ be denoted as $1, 2, \ldots, n$. Assume that there are no negative-length directed circuits in $G$. Let $W = [w_{ij}]$ be the $n \times n$ matrix of direct lengths in $G$, that is, $w_{ij}$ is the length of the directed edge $(i, j)$ in $G$. We set $w_{ij} = \infty$ if there is no edge $(i, j)$ directed from $i$ to $j$. We also set $w_{ii} = 0$ for all $i$.

Starting with the matrix $W^{(0)} = W$, Floyd's algorithm constructs a sequence $W^{(1)}$, $W^{(2)}, \ldots, W^{(n)}$ of $n \times n$ matrices so that the entry $w_{ij}^{(n)}$ in $W^{(n)}$ would give the distance from $i$ to $j$ in $G$. The matrix $W^{(k)} = [w_{ij}^{(k)}]$ is constructed from the matrix $W^{(k-1)} = [w_{ij}^{(k-1)}]$ according to the following rule:

$$w_{ij}^{(k)} = \min \left\{ w_{ij}^{(k-1)}, w_{ik}^{(k-1)} + w_{kj}^{(k-1)} \right\} \tag{2.14}$$

Let $P_{ij}^{(k)}$ denote a path of minimum length among all the directed $i-j$ paths, which use as internal vertices only those from the set $\{1, 2, \ldots, k\}$. The following theorem proves the correctness of Floyd's algorithm.

**Theorem 2.13** *For $0 \leq k \leq n$, $w_{ij}^{(k)}$ is equal to the length of $P_{ij}^{(k)}$.*

*Proof.* Proof follows from the following observations.

1. If $P_{ij}^{(k)}$ does not contain vertex $k$ then $w_{ij}^{(k)} = w_{ij}^{(k-1)}$.
2. If $P_{ij}^{(k)}$ contains vertex $k$ then $w_{ij}^{(k)} = w_{ik}^{(k-1)} + w_{kj}^{(k-1)}$ because a subpath of a shortest path is a shortest path between the end vertices of the subpath.

Note: See the similarity between the proof of this theorem and the proof of correctness of Warshall's algorithm for transitive closure. ■

Usually, in addition to the shortest lengths, we are also interested in obtaining the paths that have these lengths. Recall that in Dijkstra's algorithm we use the PRED array to keep a

record of the vertices that occur in the shortest paths. This is achieved in Floyd's algorithm as described next.

As we construct the sequence $W^{(0)}, W^{(1)}, \ldots, W^{(n)}$, we also construct another sequence $Z^{(0)}, Z^{(1)}, \ldots, Z^{(n)}$ of matrices such that the entry $z_{ij}^{(k)}$ of $Z^{(k)}$ gives the vertex that immediately follows vertex $i$ in $P_{ij}^{(k)}$. Clearly, initially we set

$$z_{ij}^{(0)} = \begin{cases} j, & \text{if } w_{ij} \neq \infty; \\ 0, & \text{if } w_{ij} = \infty. \end{cases} \tag{2.15}$$

Given $Z^{(k-1)} = [z_{ij}^{(k-1)}]$, $Z^{(k)} = [z_{ij}^{(k)}]$ is obtained according to the following rule: Let

$$M = \min\left\{w_{ij}^{(k-1)}, w_{ik}^{(k-1)} + w_{kj}^{(k-1)}\right\}$$

Then

$$z_{ij}^{(k)} = \begin{cases} z_{ij}^{(k-1)}, & \text{if } M = w_{ij}^{(k-1)}; \\ z_{ik}^{(k-1)}, & \text{if } M < w_{ij}^{(k-1)}. \end{cases} \tag{2.16}$$

It should be clear that the shortest $i-j$ path is given by the sequence $i$, $i_1$, $i_2, \ldots$, $i_p$, $j$ of vertices, where

$$i_1 = z_{ij}^{(n)}, i_2 = z_{i_1 j}^{(n)}, i_3 = z_{i_2 j}^{(n)}, \ldots j = z_{i_p j}^{(n)}. \tag{2.17}$$

**Algorithm 2.9 Shortest paths between all pairs of vertices (Floyd)**

***Input:*** $W = [w_{ij}]$ is the $n \times n$ matrix of direct lengths in the given directed graph $G = (V, E)$. Here $w_{ii} = 0$ for all $i = 1, 2, \ldots, n$. The graph does not have negative-length directed circuits.
***Output:*** Shortest directed paths and their lengths between every pair of vertices.

```
begin
    for every (i, j) ∈ V × V do
        if (i, j) ∈ E then z_ij ← j else z_ij ← 0;
    for k = 1, 2, ..., n do
        for each (i, j) ∈ V × V do
            begin
            if w_ij > w_ik + w_kj then
                begin
                w_ij ← w_ik + w_kj;
                z_ij ← z_ik;
                end
            end
end
```

Suppose the graph has some negative-length directed circuits. Then, during the execution of Floyd's algorithm, $w_{ii}$ becomes negative for some $i$. This means that the vertex $i$ is in some negative-length directed circuit and so the algorithm can be terminated at that stage.

It is easy to see that Floyd's algorithm is of complexity $O(n^3)$. This algorithm is also valid for finding shortest paths in a network with negative lengths provided the network does not have a directed circuit of negative length. Dantzig [58] proposed a variant of Floyd's algorithm that is also of complexity $O(n^3)$.

For some of the other shortest path algorithms see Tabourier [59], Williams and White [60], and Yen [61]. Deo and Pang [62] and Pierce [63] give exhaustive bibliographies of algorithms for the shortest path and related problems. A discussion of complexity results for shortest path problems can be found in Melhorn [35] and Tarjan [64]. Discussions of shortest path problems in a more general setting can be found in [43] and [35], Carré [65], and Tarjan [66]. For some other developments see Moffat and Takoka [67] and Frederickson [68].

## 2.6 TRANSITIVE ORIENTATION

An undirected graph $G$ is *transitively orientable* if we can assign orientations to the edges of $G$ so that the resulting directed graph is transitive. If $G$ is transitively orientable, then $\vec{G}$ will denote a *transitive orientation* of $G$.

For example, the graph shown in Figure 2.9a is transitively orientable. A transitive orientation of this graph is shown in Figure 2.9b.

In this section we discuss an algorithm due to Pnueli et al. [69] to test whether a simple undirected graph $G$ is transitively orientable and obtain a transitive orientation $\vec{G}$ if one exists. To aid the development and presentation of this algorithm, we introduce some notations:

1. $i \rightarrow j$ means that vertex $i$ is connected to vertex $j$ by an edge oriented from $i$ to $j$.
2. $i \leftarrow j$ is similarly defined.
3. $i$—$j$ means that there is an edge connecting vertex $i$ and vertex $j$.
4. $i \not$—$j$ means that there is no edge connecting vertex $i$ and vertex $j$.
5. $i \nrightarrow j$ means that either $i \not$—$j$ or $i \leftarrow j$ or the edge $i$—$j$ is not oriented.

$i \rightarrow j$, $i \leftarrow j$, and $i$—$j$ will also be used to denote the corresponding edges.

Consider now an undirected graph $G = (V, E)$ which is transitively orientable. Let $\vec{G} = (V, \vec{E})$ denote a transitive orientation of $G$.

Suppose that there exist three vertices $i, j, k \in V$ such that $i \rightarrow j$, $j$—$k$, and $i \not$—$k$, then transitivity of $\vec{G}$ requires that $j \leftarrow k$. Similarly, if $i, j, k \in V$, $i \rightarrow j$, $i$—$k$, and $j \not$—$k$, then transitivity of $\vec{G}$ requires that $i \rightarrow k$.

These two observations lead to the following simple rules which form the basis of Pnueli, Lempel and Even's algorithm.

**Rule $R_1$** For $i, j, k \in V$, if $i \rightarrow j$, $j$—$k$, and $i \not$—$k$, then orient the edge $j$—$k$ as $j \leftarrow k$.

**Rule $R_2$** For $i, j, k \in V$, if $i \rightarrow j$, $i$—$k$, and $j \not$—$k$, then orient the edge $i$—$k$ as $i \rightarrow k$.

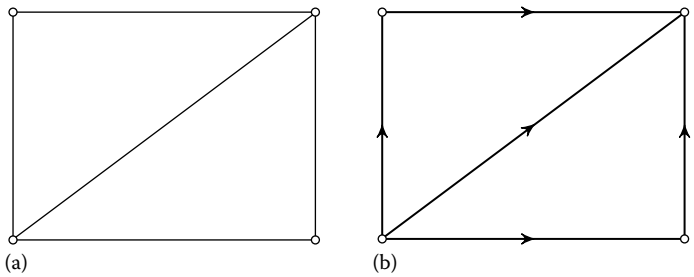


Figure 2.9 (a) Graph $G$. (b) A transitive orientation of $G$. (Data from M. N. Swamy and K. Thulasiraman, *Graphs, Networks and Algorithms*, Wiley-Interscience, 1981.)

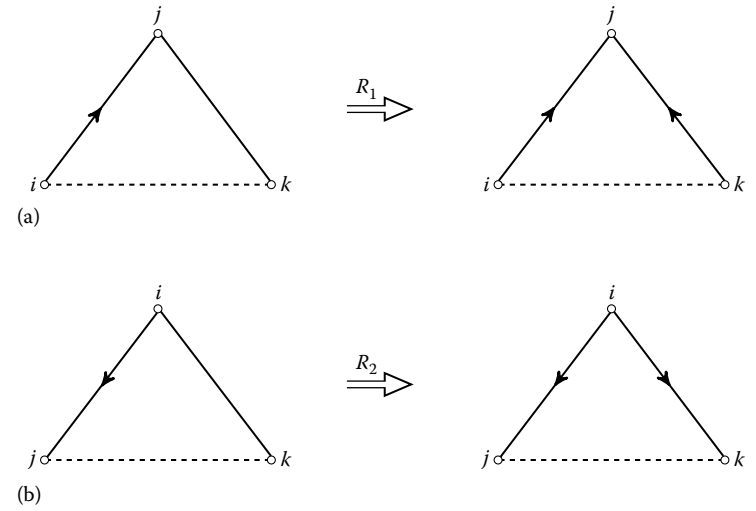


Figure 2.10 (a) Rule $R_1$. (b) Rule $R_2$. (Data from M. N. Swamy and K. Thulasiraman, *Graphs, Networks and Algorithms*, Wiley-Interscience, 1981.)

These two rules are illustrated in Figure 2.10, where a dashed line indicates the absence of the corresponding edge.

A description of the transitive orientation algorithm now follows.

**Algorithm 2.10 Transitive orientation (Pnueli, Lempel, and Even)**

**S1.** $G$ is the given simple undirected graph $i \leftarrow 1$.

**S2.** (Phase $i$ begins.) Select an edge $e$ of the graph $G$ and assign an arbitrary orientation to $e$. Assign, whenever possible, orientations to the edges in $G$ adjacent to $e$, using Rule $R_1$ or Rule $R_2$. The directed edge $e$ is now labeled *examined*.

**S3.** Test if there exists in $G$ a directed edge which has not been labeled *examined*. If yes, go to step S4. Otherwise go to step S6.

**S4.** Let $i \rightarrow j$ be a directed edge in $G$ which has not been labeled *examined*. Now do the following, whenever applicable for each edge in $G$ incident on $i$ or $j$, and then label $i \rightarrow j$ *examined*:

*Case* 1 Let the edge under consideration be $j$—$k$.

a. (Applicability of rule $R_1$.) If $i \neq k$ and the edge $j$—$k$ is not oriented, then orient $j$—$k$ as $j \leftarrow k$.

b. (Contradiction of rule $R_1$.) If $i \neq k$ and the edge $j$—$k$ is already oriented as $j \rightarrow k$, then a contradiction of rule $R_1$ has occurred. Go to step S9.

*Case* 2 Let the edge under consideration be $i$—$k$.

a. (Applicability of rule $R_2$.) If $j \neq k$ and the edge $i$—$k$ is not oriented, then orient $i$—$k$ as $i \rightarrow k$.

b. (Contradiction of rule $R_2$.) If $j \neq k$ and edge $i$—$k$ is already oriented as $i \leftarrow k$, then a contradiction of rule $R_2$ has occurred. Go to step S9.

**S5.** Go to step S3.

**S6.** (Phase $i$ has ended successfully.) Test whether all the edges of $G$ have been assigned orientations. If yes, go to step S8. Otherwise remove from $G$ all its directed edges and let $G'$ be the resulting graph.

**S7.** $G \leftarrow G'$ and $i \leftarrow i+1$. Go to Step S2.

**S8.** (All the edges of the given graph have been assigned orientations which are consistent with Rules $R_1$ and $R_2$. These orientations define a transitive orientation of the given graph.) HALT.

**S9.** (The graph $G$ is not transitively orientable.) HALT.

The main step in the above algorithm is S4. In this step we examine each edge adjacent to a directed edge, say, edge $i \rightarrow j$. If such an edge is already oriented, then we test whether its orientation and that of $i \rightarrow j$ are consistent with Rule $R_1$ or Rule $R_2$. If an edge under examination is not yet oriented, then we assign to it, if possible, an orientation using Rule $R_1$ or Rule $R_2$.

As we can see, the algorithm consists of different phases. Each phase involves execution of Step S2 and repeated executions of step S4 as more and more edges get oriented. If a phase ends without detecting any contradiction of Rule $R_1$ or Rule $R_2$, then it means that no more edges can be assigned orientations in this phase by application of the two rules, and that all the orientations assigned in this phase are consistent with these rules.

The algorithm terminates either (1) by detecting a contradiction of Rule $R_1$ or Rule $R_2$, or (2) by assigning orientations to all the edges of the given graph such that these orientations are consistent with Rules $R_1$ and $R_2$. In the former case the graph is not transitively orientable, and in the latter case the graph is transitively orientable with the resulting directed graph defining a transitive orientation.

The complexity of the algorithm depends on the complexity of executing step S4. This step is executed at most $m$ times, where $m$ is the number of edges in the given graph. Each execution of step S4 involves examining all the edges adjacent to an oriented edge. So the number of operations required to execute step S4 is proportional to $2\Delta$, where $\Delta$ is the maximum degree in the given graph. Thus the overall complexity of the algorithm is $O(2m\Delta)$.

Next we illustrate the algorithm with two examples. Consider first the graph $G$ shown in Figure 2.11a.

*Phase* 1: We begin by orienting edge 7—2 as $7 \rightarrow 2$. By Rule $R_2$, $7 \rightarrow 2$ implies that $7 \rightarrow 4$ and $7 \rightarrow 5$, and $7 \rightarrow 4$ implies that $7 \rightarrow 6$. Rule $R_1$ is not applicable to any edge adjacent to $7 \rightarrow 2$.

By Rule $R_2$, $7 \rightarrow 5$ and $7 \rightarrow 6$ imply that $7 \rightarrow 1$ and $7 \rightarrow 3$, respectively. In this phase no more edges can be assigned orientations. We can also check that all the assigned orientations are consistent with Rules $R_1$ and $R_2$. Phase 1 now terminates, and the edges oriented in this phase are shown in Figure 2.11b.

Now remove from the graph $G$ of Figure 2.11a all the edges which are oriented in Phase 1. The resulting graph $G'$ is shown in Figure 2.11c. Phase 2 now begins, and $G'$ is the graph under consideration.

*Phase* 2: We begin by orienting edge 1—2 as $1 \rightarrow 2$. This results in the following sequences of implications:

$$(1 \rightarrow 2) \underset{R_1}{\Rightarrow} (2 \leftarrow 3) \underset{R_2}{\Rightarrow} (3 \rightarrow 5) \underset{R_1}{\Rightarrow} (6 \rightarrow 5) \underset{R_1}{\Rightarrow} (4 \rightarrow 5)$$

$$(2 \leftarrow 3) \underset{R_2}{\Rightarrow} (3 \rightarrow 4)$$

$$(6 \rightarrow 5) \underset{R_2}{\Rightarrow} (6 \rightarrow 1)$$

$$(6 \rightarrow 5) \underset{R_2}{\Rightarrow} (6 \rightarrow 2)$$

Thus all the edges of $G'$ have now been oriented as shown in Figure 2.11d.

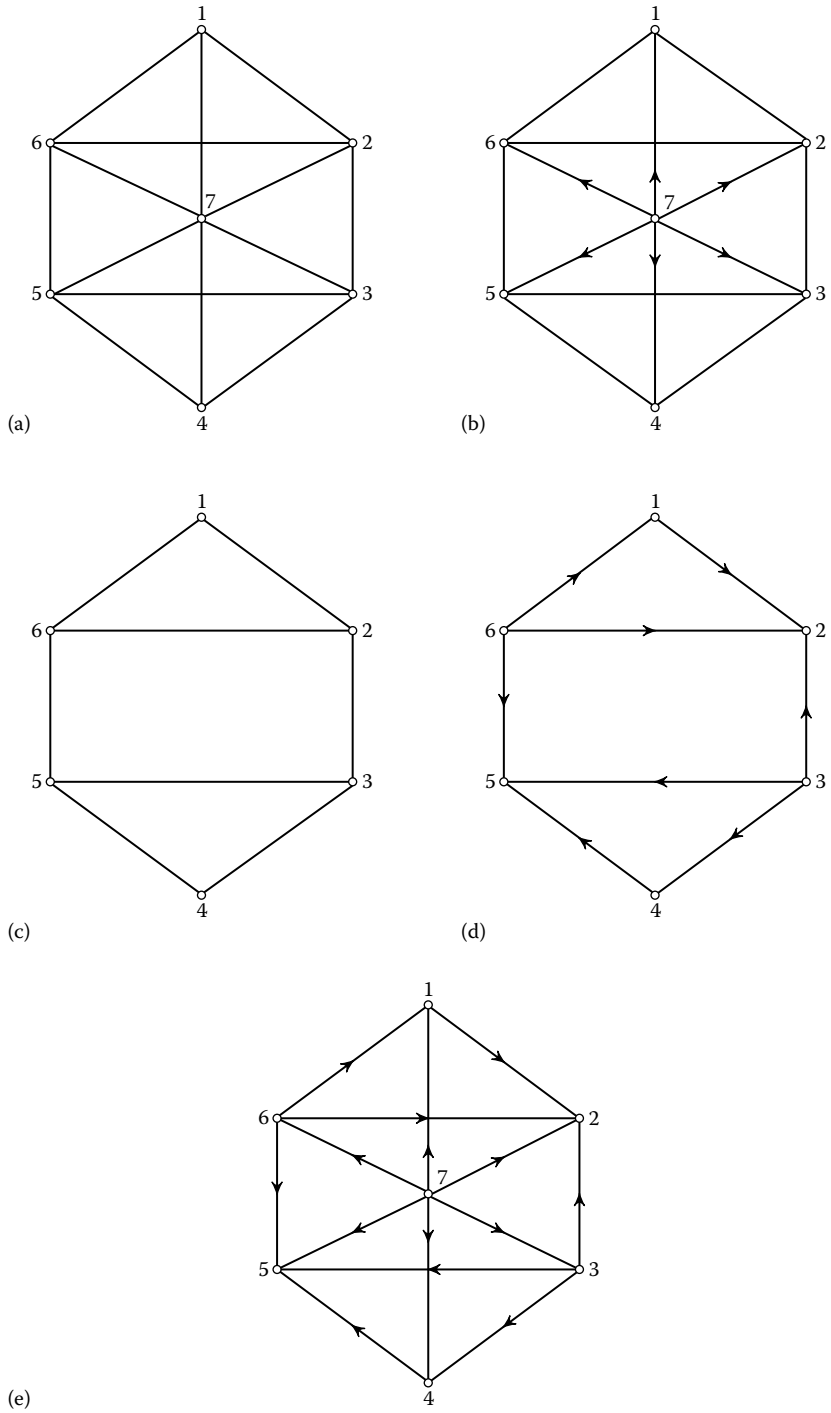


Figure 2.11 (a–e) Illustration of transitive orientation algorithm.

These orientations of $G'$ are also consistent with the Rules $R_1$ and $R_2$.

Therefore phase 2 terminates successfully.

The resulting transitive orientation of $G$ is shown in Figure 2.11e.

Consider next the graph shown in Figure 2.12. We begin by orienting edge 1—2 as $1 \rightarrow 2$. This leads to the following sequence of implications:

$$(1 \rightarrow 2) \underset{R_1}{\Rightarrow} (2 \leftarrow 3) \underset{R_2}{\Rightarrow} (3 \rightarrow 4) \underset{R_1}{\Rightarrow} (4 \leftarrow 5) \underset{R_2}{\Rightarrow} (5 \rightarrow 1) \underset{R_1}{\Rightarrow} (1 \leftarrow 2), \qquad (2.18)$$

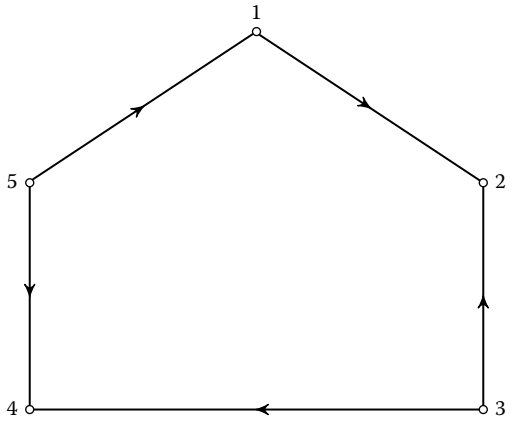


Figure 2.12 A non-transitively orientable graph.

which requires that 1—2 be directed as $1 \leftarrow 2$, contrary to the orientation we have already assigned to the edge 1—2. Thus a contradiction of Rule $R_1$ is observed. Hence the graph of Figure 2.12 is not transitively orientable.

We now proceed to prove the correctness of Algorithm 2.10. To do so we need to prove the following two assertions.

**Assertion 2.1** If Algorithm 2.10 terminates successfully (step S8), then the resulting directed graph is a transitive orientation of the given graph.

**Assertion 2.2** If the given graph is transitively orientable, then Algorithm 2.10 terminates successfully.

We first consider Assertion 1.

Given an undirected graph $G = (V, E)$. Suppose that the algorithm terminates successfully.

Consider now any two edges $i \rightarrow j$ and $k \rightarrow l$ which are assigned orientations in the same phase of the algorithm. Then we can construct a sequence of implications, which starts with the directed edge $i \rightarrow j$ and ends orienting the edge $k$—$l$ as $k \rightarrow l$. Such a sequence will be called a *derivation chain* from $i \rightarrow j$ to $k \rightarrow l$. For example, in the directed graph of Figure 2.11d the following are two of the derivation chains from $2 \leftarrow 6$ to $3 \rightarrow 4$:

$$(2 \leftarrow 6) \Rightarrow (3 \rightarrow 2) \Rightarrow (3 \rightarrow 4)$$
$$(2 \leftarrow 6) \Rightarrow (5 \leftarrow 6) \Rightarrow (3 \rightarrow 5) \Rightarrow (3 \rightarrow 2) \Rightarrow (3 \rightarrow 4)$$

Thus it is clear that it is meaningful to talk about a shortest derivation chain between any pair of directed edges which are assigned orientations in the same phase of Algorithm 2.10. Proof of Assertion 1 is based on the following important lemma.

**Lemma 2.4** *After a successful completion of phase* 1 *it is impossible to have three vertices* $i$, $j$, $k$ *such that* $i \rightarrow j$ *and* $j \rightarrow k$ *with* $i \nrightarrow k$.

*Proof.* Note that "$i \nrightarrow k$" means that either $i \not- k$ or $i \leftarrow k$ or edge $i$—$k$ is not oriented.

It is clear that there is an edge connecting $i$ and $k$. For otherwise we get a contradiction because by Rule $R_1$, $i \rightarrow j$ implies that $j \leftarrow k$.

Now assume that forbidden situations of the type $i \rightarrow j$ and $j \rightarrow k$ with $i \nrightarrow k$ exist after phase 1 of Algorithm 2.10. Then select, from among all the derivation chains which lead to a forbidden situation, a chain which is shortest with the minimum number of directed edges

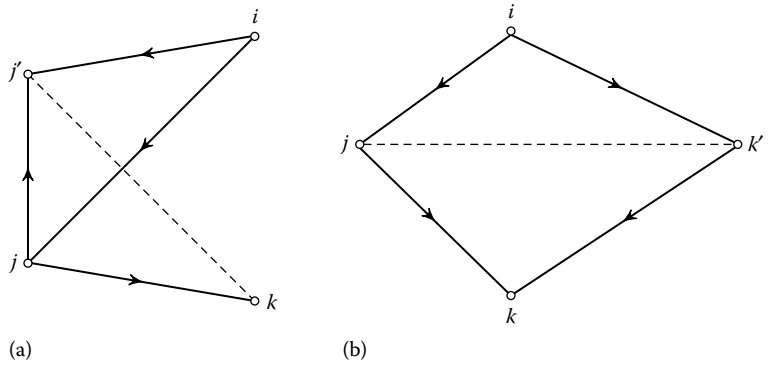


Figure 2.13 Illustration of the proof of Lemma 2.19.

incident into $k$. Clearly, any such chain must be of length at least 3. Let one such chain be as follows:

$$(i \to j) \Rightarrow (\alpha_1) \Rightarrow (\alpha_2) \Rightarrow \cdots \Rightarrow (\alpha_{p-1}) \Rightarrow (j \to k).$$

Now $\alpha_{p-1}$ must be either $j \to j'$ for some $j'$ or $k' \to k$ for some $k'$. Thus we need to consider two cases.

*Case* 1 Let $\alpha_{p-1}$ be $j \to j'$.

Then the derivation $(\alpha_{p-1}) \Rightarrow (j \to k)$ requires that $j' \not\to k$. Further $i \to j'$, for otherwise the derivation chain

$$(i \to j) \Rightarrow (\alpha_1) \Rightarrow \cdots \Rightarrow (\alpha_{p-2}) \Rightarrow (j \to j')$$

would lead to the forbidden situation $i \to j$ and $j \to j'$ with $i \nrightarrow j'$. But this chain is shorter than our chain, contradicting its minimality.

The situation arising out of the above arguments is depicted in Figure 2.13a, where a dashed line indicates the absence of the corresponding edge.

Now $i \to j'$ and $j' \not\to k$ imply that $i \to k$ by Rule $R_2$. But this contradicts our assumption that $i \nrightarrow k$.

*Case* 2 Let $\alpha_{p-1}$ be $k' \to k$.

As in the previous case the derivation $(\alpha_{p-1}) \Rightarrow (j \to k)$ requires that $j \not\to k'$. Further $i$—$k'$, for otherwise $k' \to k$ would imply that $i \to k$, contrary to our assumption that $i \nrightarrow k$. In addition, $i \to j$ and $j \not\to k'$ imply that $i \to k'$.

The situation resulting from the above arguments is depicted in Figure 2.13b.

Now the derivation chain

$$(i \to k') \Rightarrow (i \to j) \Rightarrow (\alpha_1) \Rightarrow \cdots \Rightarrow (\alpha_{p-2}) \Rightarrow (k' \to k),$$

leading to the forbidden situation $i \to k'$ and $k' \to k$ with $i \nrightarrow k$ is of the same length as our chain, but with one less edge entering $k$. This is again a contradiction of the assumption we have made about the choice of our chain. ■

Let $E'$ be the set of edges which are assigned orientations in the first phase, and let $\vec{E}'$ be the corresponding set of directed edges. The following result is an immediate consequence of Lemma 2.4.

**Theorem 2.14** *The subgraph* $\vec{G'} = (V, \vec{E'})$ *of the directed edges of the set* $\vec{E'}$ *is transitive.* ■

**Theorem 2.15** *If Algorithm* 2.10 *terminates successfully, then it gives a transitive orientation.*

*Proof.* Proof is by induction on the number of phases in the algorithm. If the algorithm orients all the edges of the given graph in phase 1, then, by Theorem 2.14, the resulting orientation is transitive.

Let the given graph $G = (V, E)$ be oriented in $p$ phases. Let $E'$ be the set of edges which are assigned orientations in the first phase. Then, by Theorem 2.14, the subgraph $\vec{G'} = (V, \vec{E'})$ is transitive. Further, since the subgraph $G'' = (V, E - E')$ is orientable in $p - 1$ phases, it follows from the induction hypothesis that $\vec{G''} = (V, \vec{E} - \vec{E'})$ is transitive. Now we prove that the directed graph $\vec{G} = (V, \vec{E})$ is transitive.

Suppose that $\vec{G}$ is not transitive, that is, there exist in $\vec{G}$ three vertices $i, j, k$ such that $i \to j$ and $j \to k$, but $i \not\to k$. Then both $i \to j$ and $j \to k$ cannot belong to $\vec{E'}$ or to $\vec{E} - \vec{E'}$ because $\vec{G'}$ and $\vec{G''}$ are both transitive.

Without loss of generality, assume that $i \to j$ is in $\vec{E'}$ and $j \to k$ is in $\vec{E} - \vec{E'}$. Then there must be an edge $i$—$k$ connecting $i$ and $k$; for otherwise $i \to j$ would imply $j \leftarrow k$, by Rule $R_1$.

Suppose that the edge $i$—$k$ is oriented as $i \leftarrow k$ in Phase 1. Then $\vec{G'}$ is not transitive, resulting in a contradiction. On the other hand, if it is oriented as $i \leftarrow k$ in a latter phase, then $\vec{G''}$ is not transitive, again resulting in a contradiction.

Thus it is impossible to have in $G$ three vertices $i$, $j$, $k$ such that $i \to j$ and $j \to k$, but $i \not\to k$. Hence $\vec{G}$ is transitive. ■

Thus Assertion 1 is established. We next proceed to establish Assertion 2.

Consider a graph $G = (V, E)$ which is transitively orientable. It is clear that if we reverse the orientations of all the edges in any transitive orientation of $G$, then the resulting directed graph is also a transitive orientation of $G$.

Suppose that we pick an edge of $G$ and assign an arbitrary orientation to it. Let this edge be $i \to j$. If we now proceed to assign orientations to additional edges using Rules $R_1$ and $R_2$, then the edges so oriented will have the same orientations in all possible transitive orientations in which the edge $i$—$j$ is oriented as $i \to j$. This is because once the orientation of $i$—$j$ is specified, the orientation derived by Rules $R_1$ and $R_2$ are necessary for transitive orientability. It therefore follows that if we apply Algorithm 2.10 to the transitively orientable graph $G$, then Phase 1 will terminate successfully without encountering any contradiction of Rule $R_1$ or Rule $R_2$. Further the edges oriented in the first phase will have the same orientations in some transitive orientation of $G$.

If we can prove that graph $G'' = (V, E - E')$, where $E'$ is the set of edges oriented in the first phase, is also transitively orientable, then it would follow that the second phase and also all other phases will terminate successfully giving a transitive orientation of $G$. Thus proving Assertion 2 is the same as establishing the transitive orientability of $G'' = (V, E - E')$. Toward this end we proceed as follows.

Let the edges of the set $E'$ be called *marked edges*, and the end vertices of these edges be called *marked vertices*. Let $V'$ denote the set of marked vertices. Note that an unmarked edge may be incident on a marked vertex.

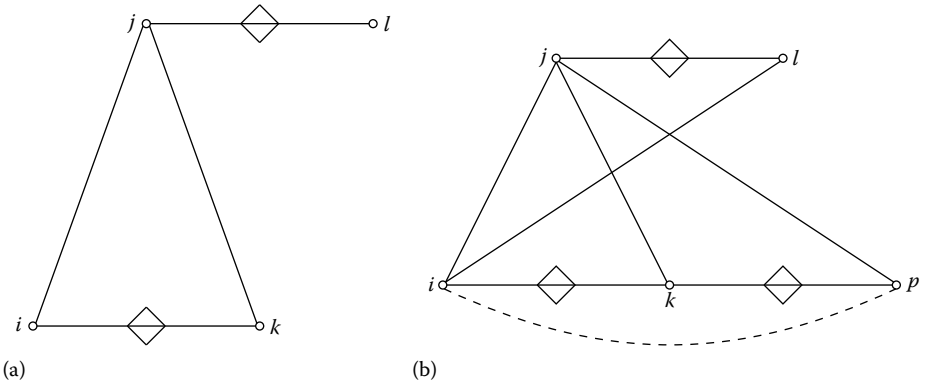


Figure 2.14 Illustration of the proof of Lemma 2.5.

**Lemma 2.5** *It is impossible to have three marked vertices $i$, $j$, $k$ such that edge $i$—$j$ and $j$—$k$ are unmarked, and edge $i$—$k$ is marked.*

*Proof.* Assume that forbidden situations of the type mentioned in the lemma exist. In other words, assume that there exist triples of marked vertices $i$, $j$, $k$ such that edges $i$—$j$ and $j$—$k$ are unmarked, and edge $i$—$k$ is marked. For each such triple $i$, $j$, $k$ there exists a marked edge $j$—$l$ for some $l$, because $j$ is a marked vertex. Therefore there exists a derivation chain from the marked edge $i$—$k$ to the marked edge $j$—$l$.

Now select a forbidden situation with $i$, $j$, and $k$ as the marked vertices such that there is a derivation chain $P$ from $i$—$k$ to $j$—$l$, which is a shortest one among all such chains that lead to a forbidden situation. This is shown in Figure 2.14a, where a diamond on an edge indicates that the edge is marked, and a dashed line indicates the absence of the corresponding edge.

The next marked edge after $i$—$k$ in the shortest chain $P$ is either $i$—$p$ or $k$—$p$, for some $p$. We assume, without loss of generality, that it is $k$—$p$. Hence $i$/—$p$, for otherwise edge $k$—$p$ would not have been marked from edge $i$—$k$. Further there exists the edge $i$—$l$ connecting $i$ and $l$, for otherwise edge $i$—$j$ would have been marked. Thus $p$ and $l$ are distinct. Also there exists the edge $j$—$p$, for otherwise edge $j$—$k$ would have been marked. The relations established so far are shown in Figure 2.14b. Now $j$—$p$ cannot be a marked edge, for marking it would result in marking $i$—$j$. Now we have a shorter derivation chain from edge $k$—$p$ to edge $j$—$l$, leading to another forbidden situation where edges $k$—$j$ and $j$—$p$ are unmarked with edge $k$—$p$ marked. A contradiction. ■

**Theorem 2.16** *If $G = (V, E)$ is transitively orientable, then $G'' = (V, E - E')$ is also transitively orientable.*

*Proof.* Since the Rules $R_1$ and $R_2$ mark only adjacent edges, it follows that the graph $G' = (V', E')$ is connected.

Consider now a vertex $v \in V - V'$. If $v$ is connected to any vertex $v' \in V'$, then $v$ should be connected to all the vertices of $V'$ which are adjacent to $v'$, for otherwise the edge $v - v'$ would have been marked. Since the graph $G'$ is connected, it would then follow that $v$ should be connected to all the vertices in $V'$.

Let $\vec{G}$ be a transitive orientation of $G$ such that the orientations of the edges of $\vec{E'}$ agree in $\vec{G}$.

Next we partition the set $V - V'$ into four subsets as follows:

$$A = \{i | i \in V - V' \text{ and for all } j \in V',\ i \to j \text{ in } \vec{G}\},$$
$$B = \{i | i \in V - V' \text{ and for all } j \in V',\ j \to i \text{ in } \vec{G}\},$$
$$C = \{i | i \in V - V' \text{ and for all } j \in V',\ i \not\to \text{ in } \vec{G}\},$$
$$D = V - (V' \cup A \cup B \cup C).$$

Note that $D$ consists of all those vertices of $V - V'$ which are connected to all the vertices of $V'$, but not all the edges connecting a vertex in $D$ to the vertices in $V'$ are oriented in the same direction.

Transitivity of $\vec{G}$ implies the following connections between the different subsets of $V$:

1. For every $i \in A$, $j \in D$, $k \in B$, $i \to j$, $j \to k$, and $i \to k$.
2. For all $i \in C$ and $j \in D$, $i \not\to j$.
3. All edges connecting $A$ and $C$ are directed from $A$ to $C$.
4. All edges connecting $B$ and $C$ are directed from $C$ to $B$.

The situation so far is depicted in Figure 2.15a.

Now reverse the orientations of all the edges directed from $V'$ to $D$ so that all the edges connecting $V'$ and $D$ are directed from $D$ to $V'$. The resulting orientation is as shown in Figure 2.15b. We now claim that this orientation is transitive.

To prove this claim we have to show that, in the graph of Figure 2.15b, if $i \to j$ and $j \to k$, then $i \to k$ for all $i$, $j$, and $k$. Clearly, this is true if none of these three edges is among those edges whose directions have just been reversed.

Thus we need to consider only the following four cases:

1. $i \in D$, $j \in V'$, and $k \in V'$.
2. $i \in D$, $j \in V'$, and $k \in B$.
3. $j \in D$, $k \in V'$, and $i \in A$.
4. $j \in D$, $k \in V'$, and $i \in D$.

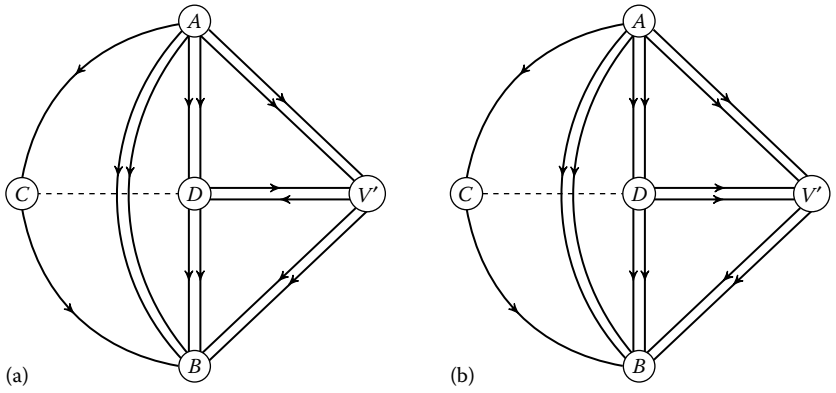


Figure 2.15 Illustration of the proof of Theorem 2.16.

In all these four cases $i \to k$ as shown in Figure 2.15b. Thus the orientation of Figure 2.15b is transitive.

Now remove from the graph of Figure 2.15b all the edges of $E'$, namely, all the marked edges.

Suppose that in the resulting graph there exist vertices $i$, $j$, and $k$ such that $i \to j$ and $j \to k$, but $i \nrightarrow k$. Here $i \nrightarrow k$ means only that $i$—$k$, for $i \leftarrow k$ would imply that the orientation in Figure 2.15b is not transitive. If edge $i$—$k$ is not in $E - E'$, it should be in $E'$, for otherwise the orientation in Figure 2.15b is not transitive. Thus $i$ and $k$ are in $V'$.

Since there is no vertex outside $V'$ which has both an edge into it from $V'$ and an edge from it into $V'$, it follows that $j$ is also in $V'$.

Thus we have $i$, $j$, $k$ marked, edges $i$—$j$ and $j$—$k$ unmarked, and edge $i$—$k$ marked. This is not possible by Lemma 2.5.

Therefore the directed graph which results after removing the edges of $\vec{E'}$ from the graph of Figure 2.15b is transitive. Hence $G'' = (V, E - E')$ is transitively orientable. ■

Thus we have established Assertion 2 and hence the correctness of Algorithm 2.10.

It should now be clear that the transitive orientation algorithm discussed above is an example of an algorithm which is simple but whose proof of correctness is very much involved. For an earlier algorithm on this problem see Gilmore and Hoffman [70].

Pnueli et al. [69] and Even et al. [71] introduced permutation graphs and established a structural relationship between these graphs and transitively orientable graphs. They also discussed an algorithm to test whether a given graph is a permutation graph. For perhaps the most recent work on transitive orientation and related problems (see McConnell and Spinard [72]).

Certain graph problems which are in general very hard to solve become simple when the graph under consideration is transitively orientable. Problems of finding a maximum clique and a minimum coloration are examples of such problems. These problems arise in the study of memory relocation and circuit layout problems [71] (see also Liu [73] and Even [74]).

## References

[1] J. B. Kruskal, Jr., On the shortest spanning subtree of a graph and the travelling salesman problem, *Proc. Am. Math. Soc.*, **7** (1956), 48–50.

[2] R. C. Prim, Shortest connection networks and some generalizations, *Bell Sys. Tech. J.*, **36** (1957), 1389–1401.

[3] A. Kerschenbaum and R. Van Slyke, Computing minimum spanning trees efficiently, *Proceedings of the 25th Annual Conference of the ACM*, 1972, 518–527.

[4] A. C. Yao, An $O(|E| \log \log |V|)$ algorithm for finding minimum spanning trees, *Inform. Process. Lett.*, **4** (1975), 21–23.

[5] D. Cheriton and R. E. Tarjan, Finding minimum spanning trees, *SIAM J. Comput.*, **5** (1976), 724–742.

[6] R. E. Tarjan, Sensitivity analysis of minimum spanning trees and shortest path trees, *Inform. Process. Lett.*, **14** (1982), 30–33.

[7] C. H. Papadimitriou and M. Yannakakis, The Complexity of restricted minimum spanning tree problems, *J. ACM*, **29** (1982), 285–309.

[8] R. L. Graham and P. Hall, On the history of the minimum spanning tree problem, *Mimeographed*, Bell Laboratories, Murray Hill, NJ, 1982.

[9] B. Korte and J. Vygen, *Combinatorial Optimization: Theory and Algorithms*, Springer, New York, 2000.

[10] B. Chazelle, A fast deterministic algorithm for minimum spanning trees, *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science*, 1997, 22–31.

[11] M. L. Fredman and R. E. Tarjan, Fibonacci heaps and their uses in improved network optimization problems, *J. ACM*, **34** (1987), 596–615.

[12] H. N. Gabow, Z. Galil, and T. Spencer, Efficient implementation of graph algorithms using contraction, *Proceedings of the 25th Annual IEEE Symposium on Foundations of Computer Science*, 1984, 338–346.

[13] H. N. Gabow, Z. Galil, T. Spencer, and R. E. Tarjan, Efficient algorithms for finding minimum spanning trees in undirected and directed graphs, *Combinatorica*, **6** (1986), 109–122.

[14] B. Korte and J. Nesetril, Vojtech Jarnik's work in combinatorial optimization, *Report No. 97855-0R*, Research Institute for Discrete Mathematics, University of Bonn, Germany, 1997.

[15] J. Edmonds, Optimum branchings, *J. Res. Nat. Bur. Std.*, **71B** (1967), 233–240.

[16] R. M. Karp, A simple derivation of Edmonds' algorithm for optimum branchings, *Networks*, **1** (1972), 265–272.

[17] M. N. Swamy and K. Thulasiraman, *Graphs, Networks and Algorithms*, Wiley-Interscience, New York, 1981.

[18] R. E. Tarjan, Finding optimum branchings, *Networks*, **7** (1977), 25–35.

[19] P. M. Camerini, L. Fratta, and F. Maffioli, A note on finding optimum branchings, *Networks*, **9** (1979), 309–312.

[20] F. C. Bock, An algorithm to construct a minimum directed spanning tree in a directed network, *Developments in Operations Research*, B. Avi-Itzak, Ed., Gordon & Breach, New York, 1971, 29–44.

[21] Y. Chu and T. Liu, On the shortest arborescence of a directed graph, *Scientia Sinica* [Peking], **4**, (1965), 1396–1400; *Math. Rev.*, **33**, 1245 (D. W. Walkup).

[22] D. Gries, *Compiler Construction for Digital Computers*, Wiley, New York, 1971.

[23] S. Warshall, A theorem on boolean matrices, *J. ACM*, **9** (1962), 11–12.

[24] H. S. Warren, A modification of Warshall's algorithm for the transitive closure of binary relations, *Comm. ACM*, **18** (1975), 218–220.

[25] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradzev, On economical construction of the transitive closure of a directed graph, *Soviet Math. Dokl.*, **11** (1970), 1209–1210.

[26] J. Eve and R. Kurki-Suonio, On computing the transitive closure of a relation, *Acta Inform.*, **8** (1977), 303–314.