# 7

# Basic Graph Algorithms




Samir Khuller
*University of Maryland*

Balaji Raghavachari
*University of Texas at Dallas*


## 7.1 Introduction

Graphs provide a powerful tool to model objects and relationships between objects. The study of graphs dates back to the eighteenth century, when Euler defined the Königsberg bridge problem, and since then has been pursued by many researchers. Graphs can be used to model problems in many areas such as transportation, scheduling, networks, robotics, VLSI design, compilers, mathematical biology, and software engineering. Many optimization problems from these and other diverse areas can be phrased in graph-theoretic terms, leading to algorithmic questions about graphs.

Graphs are defined by a set of vertices and a set of edges, where each edge connects two vertices. Graphs are further classified into directed and undirected graphs, depending on whether their edges are directed or not. An important subclass of directed graphs that arises in many applications, such

as precedence constrained scheduling problems, are **directed acyclic graphs** (DAG). Interesting subclasses of undirected graphs include **trees**, **bipartite graphs**, and **planar graphs**.

In this chapter, we focus on a few basic problems and algorithms dealing with graphs. Other chapters in this handbook provide details on specific algorithmic techniques and problem areas dealing with graphs, e.g., randomized algorithms (Chapter 12), combinatorial algorithms (Chapter 8), dynamic graph algorithms (Chapter 9), graph drawing (Chapter 6 of *Algorithms and Theory of Computation Handbook, Second Edition: Special Topics and Techniques*), and approximation algorithms (Chapter 34). Pointers into the literature are provided for various algorithmic results about graphs that are not covered in depth in this chapter.

## 7.2 Preliminaries

An undirected graph $G = (V, E)$ is defined as a set $V$ of vertices and a set $E$ of edges. An edge $e = (u, v)$ is an unordered pair of vertices. A directed graph is defined similarly, except that its edges are ordered pairs of vertices, i.e., for a directed graph, $E \subseteq V \times V$. The terms nodes and vertices are used interchangeably. In this chapter, it is assumed that the graph has neither self loops—edges of the form $(v, v)$—nor multiple edges connecting two given vertices. The number of vertices of a graph, $|V|$, is often denoted by $n$. A graph is a **sparse graph** if $|E| \ll |V|^2$.

Bipartite graphs form a subclass of graphs and are defined as follows. A graph $G = (V, E)$ is bipartite if the vertex set $V$ can be partitioned into two sets $X$ and $Y$ such that $E \subseteq X \times Y$. In other words, each edge of $G$ connects a vertex in $X$ with a vertex in $Y$. Such a graph is denoted by $G = (X, Y, E)$. Since bipartite graphs occur commonly in practice, often algorithms are designed specially for them. Planar graphs are graphs that can be drawn in the plane without any two edges crossing each other. Let $K_n$ be the complete graph on $n$ vertices, and $K_{x,y}$ be the complete bipartite graph with $x$ and $y$ vertices in either side of the bipartite graph, respectively. A homeomorph of a graph is obtained by subdividing an edge by adding new vertices.

A vertex $w$ is adjacent to another vertex $v$ if $(v, w) \in E$. An edge $(v, w)$ is said to be incident to vertices $v$ and $w$. The neighbors of a vertex $v$ are all vertices $w \in V$ such that $(v, w) \in E$. The number of edges incident to a vertex is called its **degree**. For a directed graph, if $(v, w)$ is an edge, then we say that the edge goes from $v$ to $w$. The out-degree of a vertex $v$ is the number of edges from $v$ to other vertices. The in-degree of $v$ is the number of edges from other vertices to $v$.

A **path** $p = [v_0, v_1, \ldots, v_k]$ from $v_0$ to $v_k$ is a sequence of vertices such that $(v_i, v_{i+1})$ is an edge in the graph for $0 \leq i < k$. Any edge may be used only once in a path. An intermediate vertex (or internal vertex) on a path $P[u, v]$, a path from $u$ to $v$, is a vertex incident to the path, other than $u$ and $v$. A path is simple if all of its internal vertices are distinct. A **cycle** is a path whose end vertices are the same, i.e., $v_0 = v_k$. A **walk** $w = [v_0, v_1, \ldots, v_k]$ from $v_0$ to $v_k$ is a sequence of vertices such that $(v_i, v_{i+1})$ is an edge in the graph for $0 \leq i < k$. A closed walk is one in which $v_0 = v_k$. A graph is said to be connected if there is a path between every pair of vertices. A directed graph is said to be **strongly connected** if there is a path between every pair of vertices in each direction. An acyclic, undirected graph is a **forest**, and a tree is a connected forest. A maximal forest $F$ of a graph $G$ is a forest of $G$ such that the addition of any other edge of $G$ to $F$ introduces a cycle. A directed graph that does not have any cycles is known as a DAG. Consider a binary relation $C$ between the vertices of an undirected graph $G$ such that for any two vertices $u$ and $v$, $uCv$ if and only if there is a path in $G$ between $u$ and $v$. $C$ is an equivalence relation, and it partitions the vertices of $G$ into equivalence classes, known as the connected components of $G$.

Graphs may have weights associated with edges or vertices. In the case of edge-weighted graphs (edge weights denoting lengths), the distance between two vertices is the length of a shortest path between them, where the length of a path is defined as the sum of the weights of its edges. The diameter of a graph is the maximum of the distance between all pairs of vertices.

There are two convenient ways of representing graphs on computers. In the adjacency list representation, each vertex has a linked list; there is one entry in the list for each of its adjacent vertices.

The graph is thus, represented as an array of linked lists, one list for each vertex. This representation uses $O(|V| + |E|)$ storage, which is good for sparse graphs. Such a storage scheme allows one to scan all vertices adjacent to a given vertex in time proportional to the degree of the vertex. In the adjacency matrix representation, an $n \times n$ array is used to represent the graph. The $[i, j]$ entry of this array is 1 if the graph has an edge between vertices $i$ and $j$, and 0 otherwise. This representation permits one to test if there is an edge between any pair of vertices in constant time. Both these representation schemes extend naturally to represent directed graphs. For all algorithms in this chapter except the all-pairs shortest paths problem, it is assumed that the given graph is represented by an adjacency list.

Section 7.3 discusses various tree traversal algorithms. Sections 7.4 and 7.5 discuss depth-first and breadth-first search techniques, respectively. Section 7.6 discusses the single-source shortest-path problem. Section 7.7 discusses **minimum spanning trees**. Section 7.8 discusses some traversal problems in graphs. Section 7.9 discusses various topics such as **planar graphs**, graph coloring, light approximate shortest path trees, and network decomposition, and Section 7.10 concludes with some pointers to current research on graph algorithms.

## 7.3 Tree Traversals

A tree is rooted if one of its vertices is designated as the root vertex and all edges of the tree are oriented (directed) to point away from the root. In a rooted tree, there is a directed path from the root to any vertex in the tree. For any directed edge $(u, v)$ in a rooted tree, $u$ is $v$'s parent and $v$ is $u$'s child. The *d*escendants of a vertex $w$ are all vertices in the tree (including $w$) that are reachable by directed paths starting at $w$. The ancestors of a vertex $w$ are those vertices for which $w$ is a descendant. Vertices that have no children are called **leaves**. A binary tree is a special case of a rooted tree in which each node has at most two children, namely the left child and the right child. The trees rooted at the two children of a node are called the left subtree and right subtree.

In this section we study techniques for processing the vertices of a given binary tree in various orders. It is assumed that each vertex of the binary tree is represented by a record that contains fields to hold attributes of that vertex and two special fields left and right that point to its left and right subtree respectively. Given a pointer to a record, the notation used for accessing its fields is similar to that used in the *C* programming language.

The three major tree traversal techniques are preorder, inorder, and postorder. These techniques are used as procedures in many tree algorithms where the vertices of the tree have to be processed in a specific order. In a preorder traversal, the root of any subtree has to be processed before any of its descendants. In a postorder traversal, the root of any subtree has to be processed after all of its descendants. In an inorder traversal, the root of a subtree is processed after all vertices in its left subtree have been processed, but before any of the vertices in its right subtree are processed. Preorder and postorder traversals generalize to arbitrary rooted trees. The algorithm below shows how postorder traversal of a binary tree can be used to count the number of descendants of each node and store the value in that node. The algorithm runs in linear time in the size of the tree.

```
PostOrder (T)
1  if T ≠ nil then
2     lc ← PostOrder(T → left).
3     rc ← PostOrder(T → right).
4     T → desc ← lc + rc + 1.
5     return (T → desc).
6  else
7     return 0.
8  end-if
end-proc
```

## 7.4 Depth-First Search

Depth-first search (DFS) is a fundamental graph searching technique developed by Hopcroft and Tarjan [16] and Tarjan [27]. Similar graph searching techniques were given earlier by Even [8]. The structure of DFS enables efficient algorithms for many other graph problems such as biconnectivity, triconnectivity, and planarity [8].

The algorithm first initializes all vertices of the graph as being unvisited. Processing of the graph starts from an arbitrary vertex, known as the root vertex. Each vertex is processed when it is first discovered (also referred to as visiting a vertex). It is first marked as visited, and its adjacency list is then scanned for unvisited vertices. Each time an unvisited vertex is discovered, it is processed recursively by DFS. After a node's entire adjacency list has been explored, that instance of the DFS procedure returns. This procedure eventually visits all vertices that are in the same connected component of the root vertex. Once DFS terminates, if there are still any unvisited vertices left in the graph, one of them is chosen as the root and the same procedure is repeated.

The set of edges that led to the discovery of new vertices forms a maximal forest of the graph, known as the **DFS forest**. The algorithm keeps track of this forest using parent-pointers; an array element $p[v]$ stores the parent of vertex $v$ in the tree. In each connected component, only the root vertex has a nil parent in the DFS tree.

### 7.4.1 The DFS Algorithm

DFS is illustrated using an algorithm that assigns labels to vertices such that vertices in the same component receive the same label, a useful preprocessing step in many problems. Each time the algorithm processes a new component, it numbers its vertices with a new label.

```
DFS-Connected-Component (G)
1   c ← 0.
2   for all vertices v in G do
3       visited[v] ← false.
4       finished[v] ← false.
5       p[v] ← nil.
6   end-for
7   for all vertices v in G do
8       if not visited[v] then
9           c ← c + 1.
10          DFS (v, c).
11      end-if
12  end-for
end-proc

DFS (v, c)
1   visited[v] ← true.
2   component[v] ← c.
3   for all vertices w in adj[v] do
4       if not visited[w] then
5           p[w] ← v.
6           DFS (w, c).
7       end-if
8   end-for
9   finished[v] ← true.
end-proc
```

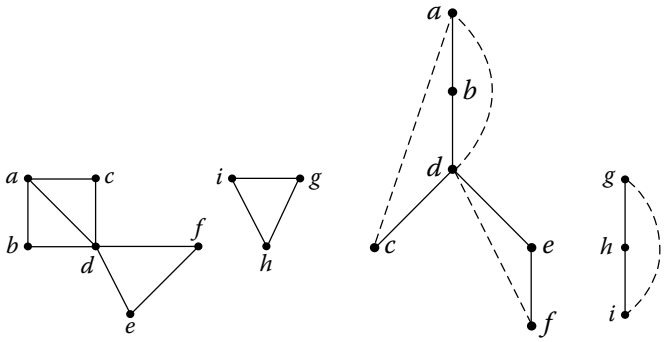



**FIGURE 7.1** Sample execution of DFS on a graph having two connected components.

### 7.4.2 Sample Execution

Figure 7.1 shows a graph having two connected components. DFS started execution at vertex $a$, and the DFS forest is shown on the right. DFS visited the vertices $b$, $d$, $c$, $e$, and $f$, in that order. It then continued with vertices $g$, $h$, and $i$. In each case, the recursive call returned when the vertex has no more unvisited neighbors. Edges $(d, a)$, $(c, a)$, $(f, d)$, and $(i, g)$ are called back edges, and these edges do not belong to the DFS forest.

### 7.4.3 Analysis

A vertex $v$ is processed as soon as it is encountered, and therefore at the start of DFS ($v$), *visited*[$v$] is false. Since *visited*[$v$] is set to true as soon as DFS starts execution, each vertex is visited exactly once. DFS processes each edge of the graph exactly twice, once from each of its incident vertices. Since the algorithm spends constant time processing each edge of $G$, it runs in $O(|V| + |E|)$ time.

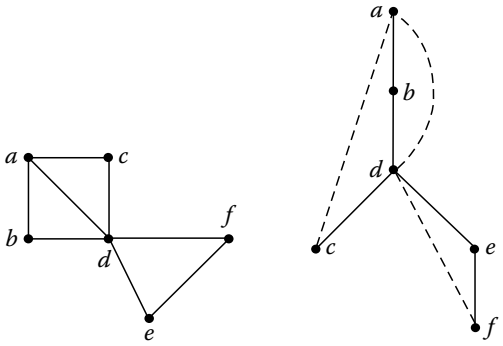
### 7.4.4 Classification of Edges

In the following discussion, there is no loss of generality in assuming that the input graph is connected. For a rooted DFS tree, vertices $u$ and $v$ are said to be related, if either $u$ is an ancestor of $v$, or vice versa.

DFS is useful due to the special nature by which the edges of the graph may be classified with respect to a DFS tree. Note that the DFS tree is not unique, and which edges are added to the tree depends on the order in which edges are explored while executing DFS. Edges of the DFS tree are known as tree edges. All other edges of the graph are known as back edges, and it can be shown that for any edge $(u, v)$, $u$ and $v$ must be related. The graph does not have any cross edges—edges that connect two vertices that are unrelated.

### 7.4.5 Articulation Vertices and Biconnected Components

One of the many applications of DFS is to decompose a graph into its biconnected components. In this section, it is assumed that the graph is connected. An **articulation vertex** (also known as cut vertex) is a vertex whose deletion along with its incident edges breaks up the remaining graph into two or more disconnected pieces. A graph is called biconnected if it has no articulation vertices. A biconnected component of a **connected graph** is a maximal subset of edges such that




| Vertex | *dis* | *fin* | *low* (*v*) |
|---|---|---|---|
| *a* | 1 | 12 | 1 |
| *b* | 2 | 11 | 1 |
| *c* | 4 | 5 | 1 |
| *d* | 3 | 10 | 1 |
| *e* | 6 | 9 | 3 |
| *f* | 7 | 8 | 3 |

Graph DFS tree

**FIGURE 7.2** Identifying cut vertices.

the corresponding induced subgraph is biconnected. Each edge of the graph belongs to exactly one biconnected component. Biconnected components can have cut vertices in common.

The graph in Figure 7.2 has two biconnected components, formed by the edge sets $\{(a,b),(a,c),(a,d),(b,d),(c,d)\}$ and $\{(d,e),(d,f),(e,f)\}$. There is a single cut vertex $d$ and it is shared by both biconnected components.

We now discuss a linear-time algorithm, developed by Hopcroft and Tarjan [16] and Tarjan [27], to identify the cut vertices and biconnected components of a connected graph. The algorithm uses the global variable time that is incremented every time a new vertex is visited or when DFS finishes visiting a vertex. Time is initially 0, and is $2|V|$ when the algorithm finally terminates. The algorithm records the value of time when a variable $v$ is first visited in the array location $dis[v]$ and the value of time when $DFS(v)$ completes execution in $fin[v]$. We refer to $dis[v]$ and $fin[v]$ as the discovery time and finish time of vertex $v$, respectively.

Let $T$ be a DFS tree of the given graph $G$. The notion of $low(v)$ of a vertex $v$ with respect to $T$ is defined as follows.

$$low(v) = \min(dis[v], dis[w] : (u,w) \text{ is a back edge for some descendant } u \text{ of } v)$$

$low(v)$ of a vertex is the discovery number of the vertex closest to the root that can be reached from $v$ by following zero or more tree edges downward, and at most one back edge upward. It captures how far high the subtree of $T$ rooted at $v$ can reach by using at most one back edge. Figure 7.2 shows an example of a graph, a DFS tree of the graph and a table listing the values of dis, fin, and low of each vertex corresponding to that DFS tree.

Let $T$ be the DFS tree generated by the algorithm, and let $r$ be its root vertex. First, $r$ is a cut vertex if and only if it has two or more children. This follows from the fact that there are no cross edges with respect to a DFS tree. Therefore the removal of $r$ from $G$ disconnects the remaining graph into as many components as the number of children of $r$. The low values of vertices can be used to find cut vertices that are nonroot vertices in the DFS tree. Let $v \neq r$ be a vertex in $G$. The following theorem characterizes precisely when $v$ is a cut vertex in $G$.

**THEOREM 7.1** *Let $T$ be a DFS tree of a connected graph $G$, and let $v$ be a nonroot vertex of $T$. Vertex $v$ is a cut vertex of $G$ if and only if there is a child $w$ of $v$ in $T$ with $low(w) \geq dis[v]$.*

Computing low values of a vertex and identifying all the biconnected components of a graph can be done efficiently with a single DFS scan. The algorithm uses a stack of edges. When an edge is encountered for the first time it is pushed into the stack irrespective of whether it is a tree edge or

a back edge. Each time a cut vertex $v$ is identified because $low(w) \geq dis[v]$ (as in Theorem 7.1), the stack contains the edges of the biconnected component as a contiguous block, with the edge $(v, w)$ at the bottom of this block. The algorithm pops the edges of this biconnected component from the stack, and sets $cut[v]$ to true to indicate that $v$ is a cut vertex.

```
BICONNECTED COMPONENTS (G)
1  time ← 0.
2  MAKEEMPTYSTACK (S).
3  for each u ∈ V do
4     visited[u] ← false.
5     cut[u] ← false.
6     p[u] ← nil.
7  end-for
8  Let v be an arbitrary vertex, DFS(v).
end-proc

DFS (v)
1  visited[v] ← true.
2  time ← time + 1.
3  dis[v] ← time.
4  low[v] ← dis[v].
5  for all vertices w in adj[v] do
6     if not visited[w] then
7        PUSH (S, (v, w)).
8        p[w] ← v.
9        DFS(w).
10       if (low[w] ≥ dis[v]) then
11          if (dis[v] ≠ 1) then cut[v] ← true.        (* v is not the root *)
12          else if (dis[w] > 2) then cut[v] ← true. (* v is root, and has at least 2 children *)
13          end-if
14          OUTPUTCOMP(v, w).
15       end-if
16       low[v] ← min(low[v], low[w]).
17    else if (p[v] ≠ w and dis[w] < dis[v]) then
18       PUSH (S, (v, w)).
19       low[v] ← min(low[v], dis[w]).
20    end-if
21 end-for
22 time ← time + 1.
23 fin[v] ← time.
end-proc

OUTPUTCOMP(v, w)
1  PRINT ("New Biconnected Component Found").
2  repeat
3     e ← POP (S).
4     PRINT (e).
5  until (e = (v, w)).
end-proc
```

In the example shown in Figure 7.2 when DFS($e$) finishes execution and returns control to DFS($d$), the algorithm discovers that $d$ is a cut vertex because $low(e) \geq dis[d]$. At this time, the stack contains the edges $(d,f)$, $(e,f)$, and $(d,e)$ at the top of the stack, which are output as one biconnected component.

**Remarks:** The notion of biconnectivity can be generalized to higher connectivities. A graph is said to be $k$-connected, if there is no subset of $(k-1)$ vertices whose removal will disconnect the graph. For example, a graph is triconnected if it does not have any separating pairs of vertices—pairs of vertices whose removal disconnects the graph. A linear-time algorithm for testing whether a given graph is triconnected was given by Hopcroft and Tarjan [15]. An $O(|V|^2)$ algorithm for testing if a graph is $k$-connected for any constant $k$ was given by Nagamochi and Ibaraki [25]. One can also define a corresponding notion of edge-connectivity, where edges are deleted from a graph rather than vertices. Galil and Italiano [11] showed how to reduce edge connectivity to vertex connectivity.

### 7.4.6 Directed Depth-First Search

The DFS algorithm extends naturally to directed graphs. Each vertex stores an adjacency list of its outgoing edges. During the processing of a vertex, the algorithm first marks the vertex as visited, and then scans its adjacency list for unvisited neighbors. Each time an unvisited vertex is discovered, it is processed recursively. Apart from tree edges and back edges (from vertices to their ancestors in the tree), directed graphs may also have forward edges (from vertices to their descendants) and cross edges (between unrelated vertices). There may be a cross edge $(u,v)$ in the graph only if $u$ is visited after the procedure call "DFS ($v$)" has completed execution. The following algorithm implements DFS in a directed graph. For each vertex $v$, the algorithm computes the discovery time of $v$ ($dis[v]$) and the time at which DFS($v$) finishes execution ($fin[v]$). In addition, each edge of the graph is classified as (1) tree edge or (2) back edge or (3) forward edge or (4) cross edge, with respect to the depth-first forest generated.

```
DIRECTED DFS (G)
1   for all vertices v in G do
2       visited[v] ← false.
3       finished[v] ← false.
4       p[v] ← nil.
5   end-for
6   time ← 0.
7   for all vertices v in G do
8       if not visited[v] then
9           DFS (v).
10      end-if
11      end-for
end-proc
```

```
DFS (v)
1   visited[v] ← true.
2   time ← time + 1.
3   dis[v] ← time.
4   for all vertices w in adj[j] do
5       if not visited[w] then
6           p[w] ← v.
```

```
7          PRINT ("Edge from" v "to" w "is a Tree edge").
8          DFS (w).
9       else if not finished[w] then
10         PRINT ("Edge from" v "to" w "is a Back edge").
11      else if dis[v] < dis[w] then
12         PRINT ("Edge from" v "to" w "is a Forward edge").
13      else
14         PRINT ("Edge from" v "to" w "is a Cross edge").
15      end-if
16   end-for
17   finished[v] ← true.
18   time ← time + 1.
19   fin[v] ← time.
end-proc
```

### 7.4.7 Sample Execution

A sample execution of the directed DFS algorithm is shown in Figure 7.3. DFS was started at vertex $a$, and the DFS forest is shown on the right. DFS visits vertices $b, d, f$, and $c$, in that order. DFS then returns and continues with $e$, and then $g$. From $g$, vertices $h$ and $i$ are visited in that order. Observe that $(d, a)$ and $(i, g)$ are back edges. Edges $(c, d)$, $(e, d)$, and $(e, f)$ are cross edges. There is a single forward edge $(g, i)$.

### 7.4.8 Applications of DFS

#### 7.4.8.1 Strong Connectivity

Directed DFS is used to design a linear-time algorithm that classifies the edges of a given directed graph into its **strongly connected** components—maximal subgraphs that have directed paths connecting any pair of vertices in them. The algorithm itself involves running DFS twice, once on the original graph, and then a second time on $G^R$, which is the graph obtained by reversing the direction of all edges in $G$. During the second DFS, the algorithm identifies all the strongly connected components. The proof is somewhat subtle, and the reader is referred to [7] for details. Cormen et al. [7] credit Kosaraju and Sharir for this algorithm. The original algorithm due to Tarjan [27] is more complicated.

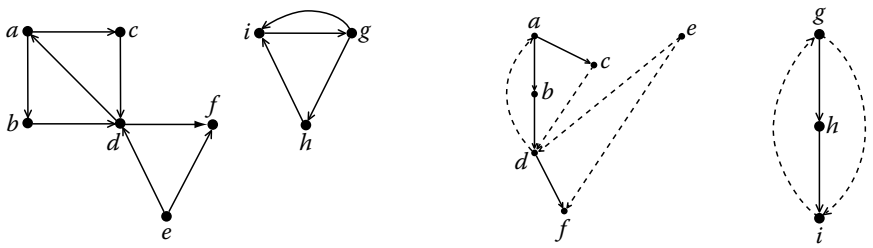



**FIGURE 7.3** Sample execution of DFS on a directed graph.

#### 7.4.8.2 Directed Acyclic Graphs

Checking if a graph is acyclic can be done in linear time using DFS. A graph has a cycle if and only if there exists a back edge relative to its DFS forest. A directed graph that does not have any cycles is known as a **directed acyclic graph** (DAG). DAGs are useful in modeling precedence constraints in scheduling problems, where nodes denote jobs/tasks, and a directed edge from $u$ to $v$ denotes the constraint that job $u$ must be completed before job $v$ can begin execution. Many problems on DAGs can be solved efficiently using dynamic programming (see Chapter 1).

#### 7.4.8.3 Topological Order

A useful concept in DAGs is that of a topological order: a linear ordering of the vertices that is consistent with the partial order defined by its edges. In other words, the vertices can be labeled with distinct integers in the range $[1 \cdots |V|]$ such that if there is a directed edge from a vertex labeled $i$ to a vertex labeled $j$, then $i < j$. Topological sort has applications in diverse areas such as project management, scheduling and circuit evaluation.

The vertices of a given DAG can be ordered topologically in linear time by a suitable modification of the DFS algorithm. It can be shown that ordering vertices by decreasing finish times (as computed by DFS) is a valid topological order. The DFS algorithm is modified as follows. A counter is initialized to $|V|$. As each vertex is marked finished, the counter value is assigned as its topological number, and the counter is decremented. Since there are no back edges in a DAG, for all edges $(u, v)$, $v$ will be marked finished before $u$. Thus, the topological number of $v$ will be higher than that of $u$.

The execution of the algorithm is illustrated with an example in Figure 7.4. Along with each vertex, we show the discovery and finish times, respectively. Vertices are given decreasing topological numbers as they are marked finished. Vertex $f$ finishes first and gets a topological number of 9 ($|V|$); $d$ finishes next and gets numbered 8, and so on. The topological order found by the DFS is $g, h, i, a, b, e, c, d, f$, which is the reverse of the finishing order. Note that a given graph may have many valid topological ordering of the vertices.

Other topological ordering algorithms work by identifying and deleting vertices of in-degree zero (i.e., vertices with no incoming edges) recursively. With some care, this algorithm can be implemented in linear time as well.

#### 7.4.8.4 Longest Path

In project scheduling, a DAG is used to model precedence constraints between tasks. A longest path in this graph is known as a critical path and its length is the least time that it takes to complete the project. The problem of computing the longest path in an arbitrary graph is NP-hard. However, longest paths in a DAG can be computed in linear time by using DFS. This method can be generalized to the case when vertices have weights denoting duration of tasks.

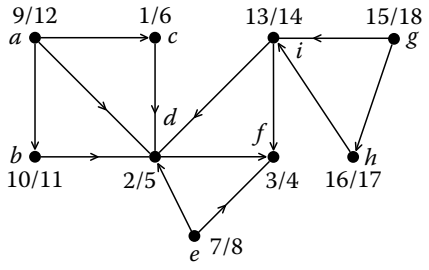



**FIGURE 7.4** Example for topological sort. Order in which vertices finish: $f, d, c, e, b, a, i, h, g$.

The algorithm processes the vertices in reverse topological order. Let $P(v)$ denote the length of a longest path coming out of vertex $v$. When vertex $v$ is processed, the algorithm computes the length of a longest path in the graph that starts at $v$.

$$P(v) = 1 + \max_{(v,w)\in E} P(w).$$

Since we are processing vertices in reverse topological order, $w$ is processed before $v$, if $(v, w)$ is an edge, and thus, $P(w)$ is computed before $P(v)$.

## 7.5 Breadth-First Search

Breadth-first search is another natural way of searching a graph. The search starts at a root vertex $r$. Vertices are added to a queue as they are discovered, and processed in first-in first-out (FIFO) order.

Initially, all vertices are marked as unvisited, and the queue consists of only the root vertex. The algorithm repeatedly removes the vertex at the front of the queue, and scans its neighbors in the graph. Any neighbor that is unvisited is added to the end of the queue. This process is repeated until the queue is empty. All vertices in the same connected component as the root vertex are scanned and the algorithm outputs a spanning tree of this component. This tree, known as a breadth-first tree, is made up of the edges that led to the discovery of new vertices. The algorithm labels each vertex $v$ by $d[v]$, the distance (length of a shortest path) from the root vertex to $v$, and stores the BFS tree in the array $p$, using parent-pointers. Vertices can be partitioned into levels based on their distance from the root. Observe that edges not in the BFS tree always go either between vertices in the same level, or between vertices in adjacent levels. This property is often useful.

### 7.5.1 The BFS Algorithm

```
BFS-Distance (G, r)
1   MakeEmptyQueue (Q).
2   for all vertices v in G do
3       visited[v] ← false.
4       d[v] ← ∞.
5       p[v] ← nil.
6   end-for
7   visited[r] ← true.
8   d[r] ← 0.
9   Enqueue (Q, r).
10  while not Empty (Q) do
11      v ← Dequeue (Q).
12      for all vertices w in adj[v] do
13          if not visited[w] then
14              visited[w] ← true.
15              p[w] ← v.
16              d[w] ← d[v] + 1.
17              Enqueue (w, Q).
18          end-if
19      end-for
20  end-while
end-proc
```

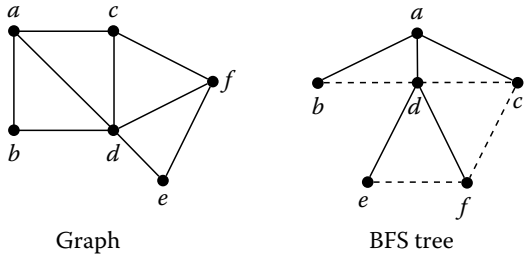



**FIGURE 7.5** Sample execution of BFS on a graph.

### 7.5.2 Sample Execution

Figure 7.5 shows a connected graph on which BFS was run with vertex $a$ as the root. When $a$ is processed, vertices $b$, $d$, and $c$ are added to the queue. When $b$ is processed nothing is done since all its neighbors have been visited. When $d$ is processed, $e$ and $f$ are added to the queue. Finally $c$, $e$, and $f$ are processed.

### 7.5.3 Analysis

There is no loss of generality in assuming that the graph $G$ is connected, since the algorithm can be repeated in each connected component, similar to the DFS algorithm. The algorithm processes each vertex exactly once, and each edge exactly twice. It spends a constant amount of time in processing each edge. Hence, the algorithm runs in $O(|V| + |E|)$ time.

### 7.5.4 Bipartite Graphs

A simple algorithm based on BFS can be designed to check if a given graph is bipartite: run BFS on each connected component of the graph, starting from an arbitrary vertex in each component as the root. The algorithm partitions the vertex set into the sets $X$ and $Y$ as follows. For a vertex $v$, if $d[v]$ is odd, then it inserts $v$ into $X$. Otherwise $d[v]$ is even and it inserts $v$ into $Y$. Now check to see if there is an edge in the graph that connects two vertices in the same set ($X$ or $Y$). If the graph contains an edge between two vertices of the same set, say $X$, then we conclude that the graph is not bipartite, since the graph contains an odd-length cycle; otherwise the algorithm has partitioned the vertex set into $X$ and $Y$ and all edges of the graph connect a vertex in $X$ with a vertex in $Y$, and therefore by definition, the graph is bipartite. (Note that it is known that a graph is bipartite if and only if it does not have a cycle of odd length.)

## 7.6 Single-Source Shortest Paths

A natural problem that often arises in practice is to compute the shortest paths from a specified node $r$ to all other nodes in a graph. BFS solves this problem if all edges in the graph have the same length. Consider the more general case when each edge is given an arbitrary, nonnegative length. In this case, the length of a path is defined to be the sum of the lengths of its edges. The distance between two nodes is the length of a shortest path between them. The objective of the shortest path problem is to compute the distance from $r$ to each vertex $v$ in the graph, and a path of that length from $r$ to $v$. The output is a tree, known as the shortest path tree, rooted at $r$. For any vertex $v$ in the graph, the unique path from $r$ to $v$ in this tree is a shortest path from $r$ to $v$ in the input graph.

### 7.6.1 Dijkstra's Algorithm

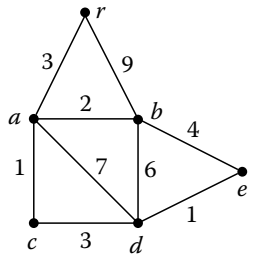
Dijkstra's algorithm provides an efficient solution to the shortest path problem. For each vertex $v$, the algorithm maintains an upper bound of the distance from the root to vertex $v$ in $d[v]$; initially $d[v]$ is set to infinity for all vertices except the root, which has $d$-value equal to zero. The algorithm maintains a set $S$ of vertices with the property that for each vertex $v \in S$, $d[v]$ is the length of a shortest path from the root to $v$. For each vertex $u$ in $V - S$, the algorithm maintains $d[u]$ to be the length of a shortest path from the root to $u$ that goes entirely within $S$, except for the last edge. It selects a vertex $u$ in $V - S$ with minimum $d[u]$ and adds it to $S$, and updates the distance estimates to the other vertices in $V - S$. In this update step it checks to see if there is a shorter path to any vertex in $V - S$ from the root that goes through $u$. Only the distance estimates of vertices that are adjacent to $u$ need to be updated in this step. Since the primary operation is the selection of a vertex with minimum distance estimate, a priority queue is used to maintain the $d$-values of vertices (for more information about priority queues, see Chapter 4). The priority queue should be able to handle the DecreaseKey operation to update the $d$-value in each iteration. The following algorithm implements Dijkstra's algorithm.

```
Dijkstra-Shortest Paths (G, r)
1   for all vertices v in G do
2       visited[v] ← false.
3       d[v] ← ∞.
4       p[v] ← nil.
5   end-for
6   d[r] ← 0.
7   BuildPQ (H, d).
8   while not Empty (H) do
9           u ← DeleteMin (H).
10          visited[u] ← true.
11          for all vertices v in adj[u] do
12              Relax (u, v).
13          end-for
14  end-while
end-proc
```

```
Relax (u, v)
1   if not visited[v] and d[v] > d[u] + w(u, v) then
2       d[v] ← d[u] + w(u, v).
3       p[v] ← u.
4       DecreaseKey (H, v, d[v]).
5   end-if
end-proc
```

### 7.6.2 Sample Execution

Figure 7.6 shows a sample execution of the algorithm. The column titled "Iter" specifies the number of iterations that the algorithm has executed through the while loop in Step 8. In iteration 0 the initial values of the distance estimates are $\infty$. In each subsequent line of the table, the column marked $u$ shows the vertex that was chosen in Step 9 of the algorithm, and the other columns show the change to the distance estimates at the end of that iteration of the while loop. In the first iteration, vertex $r$




| Iter | $u$ | $d(a)$ | $d(b)$ | $d(c)$ | $d(d)$ | $d(e)$ |
|---|---|---|---|---|---|---|
| 0 | – | ∞ | ∞ | ∞ | ∞ | ∞ |
| 1 | $r$ | 3 | 9 | ∞ | ∞ | ∞ |
| 2 | $a$ | 3 | 5 | 4 | 10 | ∞ |
| 3 | $c$ | 3 | 5 | 4 | 7 | ∞ |
| 4 | $b$ | 3 | 5 | 4 | 7 | 9 |
| 5 | $d$ | 3 | 5 | 4 | 7 | 8 |
| 6 | $e$ | 3 | 5 | 4 | 7 | 8 |

**FIGURE 7.6** Dijkstra's shortest path algorithm.

was chosen, after that $a$ was chosen since it had the minimum distance label among the unvisited vertices, and so on. The distance labels of the unvisited neighbors of the visited vertex are updated in each iteration.

### 7.6.3 Analysis

The running time of the algorithm depends on the data structure that is used to implement the priority queue $H$. The algorithm performs $|V|$ DeleteMin operations and at most $|E|$ DecreaseKey operations. If a binary heap is used to find the records of any given vertex, each of these operations run in $O(\log |V|)$ time. There is no loss of generality in assuming that the graph is connected. Hence, the algorithm runs in $O(|E| \log |V|)$. If a Fibonacci heap [10] is used to implement the priority queue, the running time of the algorithm is $O(|E| + |V| \log |V|)$. Even though the Fibonacci heap gives the best asymptotic running time, the binary heap implementation is likely to give better running times for most practical instances.

### 7.6.4 Extensions

Dijkstra's algorithm can be generalized to solve several problems that are related to the shortest path problem. For example, in the bottleneck shortest path problem, the objective is to find, for each vertex $v$, a path from the root to $v$ in which the length of the longest edge in that path is minimized. A small change to Dijkstra's algorithm (replacing the operation $+$ in Relax by max) solves this problem. Other problems that can be solved by suitably modifying Dijkstra's algorithm include the following:

- Finding most reliable paths from the root to every vertex in a graph where each edge is given a probability of failure (independent of the other edges)
- Finding the fastest way to get from a given point in a city to a specified location using public transportation, given the train/bus schedules

### 7.6.5 Bellman–Ford Algorithm

The shortest path algorithm described above directly generalizes to directed graphs, but it does not work if the graph has edges of negative length. For graphs that have edges of negative length, but no cycles of negative length, there is a different algorithm solves due to Bellman and Ford that solves the single-source shortest paths problem in $O(|V||E|)$ time.

In a single scan of the edges, the Relax operation is executed on each edge. The scan is then repeated $|V| - 1$ times. No special data structures are required to implement this algorithm, and the proof relies on the fact that a shortest path is simple and contains at most $|V| - 1$ edges.

This problem also finds applications in finding a feasible solution to a system of linear equations of a special form that arises in real-time applications: each equation specifies a bound on the difference

between two variables. Each constraint is modeled by an edge in a suitably defined directed graph. Shortest paths from the root of this graph capture feasible solutions to the system of equations (for more information, see [7, Chapter 24.5]).

### 7.6.6 The All-Pairs Shortest Paths Problem

Consider the problem of computing a shortest path between every pair of vertices in a directed graph with edge lengths. The problem can be solved in $O(|V|^3)$ time, even when some edges have negative lengths, as long as the graph has no negative length cycles. Let the lengths of the edges be stored in a matrix $A$; the array entry $A[i, j]$ stores the length of the edge from $i$ to $j$. If there is no edge from $i$ to $j$, then $A[i, j] = \infty$; also $A[i, i]$ is set to 0 for all $i$. A dynamic programming algorithm to solve the problem is discussed in this section. The algorithm is due to Floyd and builds on the work of Warshall.

Define $P_k[u, v]$ to be a shortest path from $u$ to $v$ that is restricted to using intermediate vertices only from the set $\{1, \ldots, k\}$. Let $D_k[u, v]$ be the length of $P_k[u, v]$. Note that $P_0[u, v] = (u, v)$ since the path is not allowed to use *any* intermediate vertices, and therefore $D_0[u, v] = A[u, v]$. Since there are no negative length cycles, there is no loss of generality in assuming that shortest paths are simple.

The structure of shortest paths leads to the following recursive formulation of $P_k$. Consider $P_k[i, j]$ for $k > 0$. Either vertex $k$ is on this path or not. If $P_k[i, j]$ does not pass through $k$, then the path uses only vertices from the set $\{1, \ldots, k-1\}$ as intermediate vertices, and is therefore the same as $P_{k-1}[i, j]$. If $k$ is a vertex on the path $P_k[i, j]$, then it passes through $k$ exactly once because the path is simple. Moreover, the subpath from $i$ to $k$ in $P_k[i, j]$ is a shortest path from $i$ to $k$ that uses intermediate vertices from the set $\{1, \ldots, k-1\}$, as does the subpath from $k$ to $j$ in $P_k[i, j]$. Thus, the path $P_k[i, j]$ is the union of $P_{k-1}[i, k]$ and $P_{k-1}[k, j]$. The above discussion leads to the following recursive formulation of $D_k$:

$$D_k[i, j] = \begin{cases} \min\left(D_{k-1}[i, j], D_{k-1}[i, k] + D_{k-1}[k, j]\right) & \text{if } k > 0 \\ A[i, j] & \text{if } k = 0 \end{cases}$$

Finally, since $P_n[i, j]$ is allowed to go through any vertex in the graph, $D_n[i, j]$ is the length of a shortest path from $i$ to $j$ in the graph.

In the algorithm described below, a matrix $D$ is used to store distances. It might appear at first glance that to compute the distance matrix $D_k$ from $D_{k-1}$, different arrays must be used for them. However, it can be shown that in the $k$th iteration, the entries in the $k$th row and column do not change, and thus, the same space can be reused.

```
FLOYD-SHORTEST-PATH (G)
1   for i = 1 to |V| do
2       for j = 1 to |V| do
3           D[i, j] ← A[i, j]
4       end-for
5   end-for
6   for k = 1 to |V| do
7       for i = 1 to |V| do
8           for j = 1 to |V| do
9               D[i, j] ← min(D[i, j], D[i, k] + D[k, j]).
10          end-for
11      end-for
12  end for
end-proc
```

## 7.7 Minimum Spanning Trees

The following fundamental problem arises in network design. A set of sites need to be connected by a network. This problem has a natural formulation in graph-theoretic terms. Each site is represented by a vertex. Edges between vertices represent a potential link connecting the corresponding nodes. Each edge is given a nonnegative cost corresponding to the cost of constructing that link. A tree is a minimal network that connects a set of nodes. The cost of a tree is the sum of the costs of its edges. A minimum-cost tree connecting the nodes of a given graph is called a minimum-cost spanning tree, or simply a **minimum spanning tree** (MST).

The problem of computing a MST arises in many areas, and as a subproblem in combinatorial and geometric problems. MSTs can be computed efficiently using algorithms that are greedy in nature, and there are several different algorithms for finding an MST. One of the first algorithms was due to Boruvka. Two algorithms, popularly known as Prim's algorithm and Kruskal's algorithm, are described here.

We first describe some rules that characterize edges belonging to a MST. The various algorithms are based on applying these rules in different orders. Tarjan [28] uses colors to describe these rules. Initially, all edges are uncolored. When an edge is colored blue it is marked for inclusion in the MST. When an edge is colored red it is marked to be excluded from the MST. The algorithms maintain the property that there is an MST containing all the blue edges but none of the red edges.

A cut is a partitioning of the vertex set into two subsets $S$ and $V - S$. An edge crosses the cut if it connects a vertex $x \in S$ to a vertex $y \in V - S$.

(**Blue rule**) Find a cut that is not crossed by any blue edge and color a minimum weight edge that crosses the cut to be blue.

(**Red rule**) Find a simple cycle containing no red edges and color a maximum weight edge on that cycle to be red.

The proofs that these rules work can be found in [28].

### 7.7.1 Prim's Algorithm

Prim's algorithm for finding an MST of a given graph is one of the oldest algorithms to solve the problem. The basic idea is to start from a single vertex and gradually "grow" a tree, which eventually spans the entire graph. At each step, the algorithm has a tree of blue edges that covers a set $S$ of vertices. The blue rule is applied by picking the cut $S, V - S$. This may be used to extend the tree to include a vertex that is currently not in the tree. The algorithm selects a minimum-cost edge from the edges crossing the cut and adds it to the current tree (implicitly coloring the edge blue), thereby adding another vertex to $S$.

As in the case of Dijkstra's algorithm, each vertex $u \in V - S$ can attach itself to only one vertex in the tree so that the current solution maintained by the algorithm is always a tree. Since the algorithm always chooses a minimum-cost edge, it needs to maintain a minimum-cost edge that connects $u$ to some vertex in $S$ as the candidate edge for including $u$ in the tree. A priority queue of vertices is used to select a vertex in $V - S$ that is incident to a minimum-cost candidate edge.

```
PRIM-MST (G, r)
1  for all vertices v in G do
2      visited[v] ← false.
3      d[v] ← ∞.
4      p[v] ← nil.
5  end-for
6  d[r] ← 0.
7  BUILDPQ (H, d).
```

```
8   while not Empty(H) do
9      u ← DeleteMin (H).
10     visited[u] ← true.
11     for all vertices v in adj[u] do
12        if not visited[v] and d[v] > w(u, v) then
13           d[v] ← w(u, v).
14           p[v] ← u.
15           DecreaseKey (H, v, d[v]).
16        end-if
17     end-for
18  end-while
end-proc
```

### 7.7.2 Analysis

First observe the similarity between Prim's and Dijkstra's algorithms. Both algorithms start building the tree from a single vertex and grow it by adding one vertex at a time. The only difference is the rule for deciding when the current label is updated for vertices outside the tree. Both algorithms have the same structure and therefore have similar running times. Prim's algorithm runs in $O(|E| \log |V|)$ time if the priority queue is implemented using binary heaps, and it runs in $O(|E| + |V| \log |V|)$ if the priority queue is implemented using Fibonacci heaps.

### 7.7.3 Kruskal's Algorithm

Kruskal's algorithm for finding an MST of a given graph is another classical algorithm for the problem, and is also greedy in nature. Unlike Prim's algorithm which grows a single tree, Kruskal's algorithm grows a forest. First the edges of the graph are sorted in nondecreasing order of their costs. The algorithm starts with an empty forest. The edges of the graph are scanned in sorted order, and if the addition of the current edge does not generate a cycle in the current forest, it is added to the forest. The main test at each step is: does the current edge connect two vertices in the same connected component of the current forest? Eventually the algorithm adds $n - 1$ edges to generate a spanning tree in the graph.

The following discussion explains the correctness of the algorithm based on the two rules described earlier. Suppose that as the algorithm progresses, the edges chosen by the algorithm are colored blue and the ones that it rejects are colored red. When an edge is considered and it forms a cycle with previously chosen edges, this is a cycle with no red edges. Since the algorithm considers the edges in nondecreasing order of weight, the last edge is the heaviest edge in the cycle and therefore it can be colored red by the red rule. If an edge connects two blue trees $T_1$ and $T_2$, then it is a lightest edge crossing the cut $T_1$ and $V - T_1$, because any other edge crossing the cut has not been considered yet and is therefore no lighter. Therefore it can be colored blue by the blue rule.

The main data structure needed to implement the algorithm is to maintain connected components. An abstract version of this problem is known as the union–find problem for collection of disjoint sets (Chapters 8, 9, and 34). Efficient algorithms are known for this problem, where an arbitrary sequence of Union and Find operations can be implemented to run in almost linear time (for more information, see [7,28]).

```
Kruskal-MST(G)
1   T ← ϕ.
2   for all vertices v in G do
```

```
3      p[v] ← v.
4    end-for
5    Sort the edges of G by nondecreasing order of costs.
6    for all edges e = (u, v) in G in sorted order do
7      if FIND (u) ≠ FIND (v) then
8        T ← T ∪ (u, v).
9        UNION (u, v).
10     end-if
11   end-for
end-proc
```

### 7.7.4 Analysis

The running time of the algorithm is dominated by Step 5 of the algorithm in which the edges of the graph are sorted by nondecreasing order of their costs. This takes $O(|E| \log |E|)$ (which is also $O(|E| \log |V|)$) time using an efficient sorting algorithm such as heap sort. Kruskal's algorithm runs faster in the following special cases: if the edges are presorted, if the edge costs are within a small range, or if the number of different edge costs is bounded. In all these cases, the edges can be sorted in linear time, and Kruskal's algorithm runs in the near-linear time of $O(|E|\alpha(|E|, |V|))$, where $\alpha(m, n)$ is the inverse Ackermann function [28].

### 7.7.5 Boruvka's Algorithm

Boruvka's algorithm also grows many trees simultaneously. Initially there are $|V|$ trees, where each vertex forms its own tree. At each stage the algorithm keeps a collection of blue trees (i.e., trees built using only blue edges). For convenience, assume that all edge weights are distinct. If two edges have the same weight, they may be ordered arbitrarily. Each tree selects a minimum cost edge that connects it to some other tree and colors it blue. At the end of this parallel coloring step, each tree merges with a collection of other trees. The number of trees decreases by at least a factor of 2 in each step, and therefore after $\log |V|$ iterations there is exactly one tree. In practice, many trees merge in a single step and the algorithm converges much faster. Each step can be implemented in $O(|E|)$ time, and hence, the algorithm runs in $O(|E| \log |V|)$. For the special case of planar graphs, the above algorithm actually runs in $O(|V|)$ time.

Almost linear-time deterministic algorithms for the MST problem in undirected graphs are known [5,10]. Recently, Karger et al. [18] showed that they can combine the approach of Boruvka's algorithm with a random sampling approach to obtain a randomized algorithm with an expected running time of $O(|E|)$. Their algorithm also needs to use as a subroutine a procedure to verify that a proposed tree is indeed an MST [20,21]. The equivalent of MSTs in directed graphs are known as minimum branchings and are discussed in Chapter 8.

## 7.8 Tour and Traversal Problems

There are many applications for finding certain kinds of paths and tours in graphs. We briefly discuss some of the basic problems.

The **traveling salesman problem** (TSP) is that of finding a shortest tour that visits all the vertices of a given graph with weights on the edges. It has received considerable attention in the literature [22]. The problem is known to be computationally intractable (NP-hard). Several heuristics are known to solve practical instances. Considerable progress has also been made in finding optimal solutions for graphs with a few thousand vertices.

One of the first graph-theoretic problems to be studied, the **Euler tour problem** asks for the existence of a closed walk in a given connected graph that traverses each edge exactly once. Euler proved that such a closed walk exists if and only if each vertex has even degree [12]. Such a graph is known as an **Eulerian graph**. Given an Eulerian graph, an Euler tour in it can be computed using an algorithm similar to DFS in linear time.

Given an edge-weighted graph, the **Chinese postman problem** is that of finding a shortest closed walk that traverses each edge at least once. Although the problem sounds very similar to the TSP problem, it can be solved optimally in polynomial time [1].

## 7.9 Assorted Topics

### 7.9.1 Planar Graphs

A graph is called planar if it can be drawn on the plane without any of its edges crossing each other. A planar embedding is a drawing of a planar graph on the plane with no crossing edges. An embedded planar graph is known as a plane graph. A *face* of a plane graph is a connected region of the plane surrounded by edges of the planar graph. The unbounded face is referred to as the exterior face. Euler's formula captures a fundamental property of planar graphs by relating the number of edges, the number of vertices and the number of faces of a plane graph: $|F| - |E| + |V| = 2$. One of the consequences of this formula is that a simple planar graph has at most $O(|V|)$ edges.

Extensive work has been done on the study of planar graphs and a recent book has been devoted to the subject [26]. A fundamental problem in this area is deciding whether a given graph is planar, and if so, finding a planar embedding for it. Kuratowski gave necessary and sufficient conditions for when a graph is planar, by showing that a graph is planar if and only if it has no subgraph that is a homeomorph of $K_5$ or $K_{3,3}$. Hopcroft and Tarjan [17] gave a linear-time algorithm to test if a graph is planar, and if it is, to find a planar embedding for the graph.

A balanced separator is a subset of vertices that disconnects the graph in such a way, that the resulting components each have at most a constant fraction of the number of vertices of the original graph. Balanced separators are useful in designing "divide-and-conquer" algorithms for graph problems, such as graph layout problems (Chapter 8 of *Algorithms and Theory of Computation Handbook, Second Edition: Special Topics and Techniques*). Such algorithms are possible when one is guaranteed to find separators that have very few vertices relative to the graph. Lipton and Tarjan [24] proved that every planar graph on $|V|$ vertices has a separator of size at most $\sqrt{8|V|}$, whose deletion breaks the graph into two or more disconnected graphs, each of which has at most $2/3|V|$ vertices. Using the property that planar graphs have small separators, Frederickson [9] has given faster shortest path algorithms for planar graphs. Recently, this was improved to a linear-time algorithm by Henzinger et al. [13].

### 7.9.2 Graph Coloring

A coloring of a graph is an assignment of colors to the vertices, so that any two adjacent vertices have distinct colors. Traditionally, the colors are not given names, but represented by positive integers. The vertex coloring problem is the following: given a graph, to color its vertices using the fewest number of colors (known as the chromatic number of the graph). This was one of the first problems that were shown to be intractable (NP-hard). Recently it has been shown that even the problem of approximating the chromatic number of the graph within any reasonable factor is intractable. But, the coloring problem needs to be solved in practice (such as in the channel assignment problem in cellular networks), and heuristics are used to generate solutions. We discuss a commonly used greedy heuristic below: the vertices of the graph are colored sequentially in an arbitrary order. When a vertex is being processed, the color assigned to it is the smallest positive number that is not used

by any of its neighbors that have been processed earlier. This scheme guarantees that if the degree of a vertex is $\Delta$, then its color is at most $\Delta + 1$. There are special classes of graphs, such as planar graphs, in which the vertices can be carefully ordered in such a way that the number of colors used is small. For example, the vertices of a planar graph can be ordered such that every vertex has at most five neighbors that appear earlier in the list. By coloring its vertices in that order yields a six-coloring. There is a different algorithm that colors any planar graph using only four colors.

### 7.9.3 Light Approximate Shortest Path Trees

To broadcast information from a specified vertex $r$ to all vertices of $G$, one may wish to send the information along a shortest path tree in order to reduce the time taken by the message to reach the nodes (i.e., minimizing delay). Though the shortest path tree may minimize delays, it may be a much costlier network to construct and considerably heavier than a MST, which leads to the question of whether there are trees that are light (like an MST) and yet capture distances like a shortest path tree. In this section, we consider the problem of computing a light subgraph that approximates a shortest path tree rooted at $r$.

Let $T_{\min}$ be a MST of $G$. For any vertex $v$, let $d(r, v)$ be the length of a shortest path from $r$ to $v$ in $G$. Let $\alpha > 1$ and $\beta > 1$ be arbitrary constants. An $(\alpha, \beta)$-light approximate shortest path tree ($(\alpha, \beta)$-LAST) of $G$ is a spanning tree $T$ of $G$ with the property that the distance from the root to any vertex $v$ in $T$ is at most $\alpha \cdot d(r, v)$ and the weight of $T$ is at most $\beta$ times the weight of $T_{\min}$.

Awerbuch et al. [3], motivated by applications in broadcast-network design, made a fundamental contribution by showing that every graph has a shallow-light tree—a tree whose diameter is at most a constant times the diameter of $G$ and whose total weight is at most a constant times the weight of a MST. Cong et al. [6] studied the same problem and showed that the problem has applications in VLSI-circuit design; they improved the approximation ratios obtained in [3] and also studied variations of the problem such as bounding the radius of the tree instead of the diameter.

Khuller et al. [19] modified the shallow-light tree algorithm and showed that the distance from the root to each vertex can be approximated within a constant factor. Their algorithm also runs in linear time if a MST and a shortest path tree are provided. The algorithm computes an $(\alpha, 1 + \frac{2}{\alpha - 1})$-LAST.

The basic idea is as follows: initialize a subgraph $H$ to be a MST $T_{\min}$. The vertices are processed in a preorder traversal of $T_{\min}$. When a vertex $v$ is processed, its distance from $r$ in $H$ is compared to $\alpha \cdot d(r, v)$. If the distance exceeds the required threshold, then the algorithm adds to $H$ a shortest path in $G$ from $r$ to $v$. When all the vertices have been processed, the distance in $H$ from $r$ to any vertex $v$ meets its distance requirement. A shortest path tree in $H$ is returned by the algorithm as the required LAST.

### 7.9.4 Network Decomposition

The problem of decomposing a graph into clusters, each of which has low diameter, has applications in distributed computing. Awerbuch [2] introduced an elegant algorithm for computing low diameter clusters, with the property that there are few inter-cluster edges (assuming that edges going between clusters are not counted multiply). This construction was further refined by Awerbuch and Peleg [4], and they showed that a graph can be decomposed into clusters of diameter $O(r \log |V|)$ with the property that each $r$ neighborhood of a vertex belongs to some cluster. (An $r$ neighborhood of a vertex is the set of nodes whose distance from the vertex is at most $r$.) In addition, each vertex belongs to at most $2 \log |V|$ clusters. Using a similar approach Linial and Saks [23] showed that a graph can be decomposed into $O(\log |V|)$ clusters, with the property that each connected component in a cluster has $O(\log |V|)$ diameter. These techniques have found several applications in the computation of approximate shortest paths, and in other distributed computing problems.

The basic idea behind these methods is to perform an "expanding BFS." The algorithm selects an arbitrary vertex, and executes BFS with that vertex as the root. The algorithm continues the search layer by layer, ensuring that the number of vertices in a layer is at least as large as the number of vertices currently in that BFS tree. Since the tree expands rapidly, this procedure generates a low diameter BFS tree (cluster). If the algorithm comes across a layer in which the number of nodes is not big enough, it rejects that layer and stops growing that tree. The set of nodes in the layer that was not added to the BFS tree that was being grown is guaranteed to be small. The algorithm continues by selecting a new vertex that was not chosen in any cluster and repeats the above procedure.

## 7.10 Research Issues and Summary

We have illustrated some of the fundamental techniques that are useful for manipulating graphs. These basic algorithms are used as tools in the design of algorithms for graphs. The problems studied in this chapter included representation of graphs, tree traversal techniques, search techniques for graphs, shortest path problems, MSTs, and tour problems on graphs.

Current research on graph algorithms focuses on dynamic algorithms, graph layout and drawing, and approximation algorithms. More information about these areas can be found in Chapters 8, 9, and 34 of this book. The methods illustrated in our chapter find use in the solution of almost any graph problem.

The graph isomorphism problem is an old problem in this area. The input to this problem is two graphs and the problem is to decide whether the two graphs are isomorphic, i.e., whether the rows and columns of the adjacency matrix of one of the graphs can be permuted so that it is identical to the adjacency matrix of the other graph. This problem is neither known to be polynomial-time solvable nor known to be NP-hard. This is in contrast to the subgraph isomorphism problem in which the problem is to decide whether there is a subgraph of the first graph that is isomorphic to the second graph. The subgraph isomorphism is known to be NP-complete. Special instances of the **graph isomorphism problem** are known to be polynomially solvable, such as when the graphs are planar, or more generally of bounded genus. For more information on the isomorphism problem, see Hoffman [14].

Another open problem is whether there exists a deterministic linear-time algorithm for computing a MST. Near-linear-time deterministic algorithms using Fibonacci heaps have been known for finding an MST. The newly discovered probabilistic algorithm uses random sampling to find an MST in expected linear time. Much of the recent research in this area is focusing on the design of approximation algorithms for NP-hard problems.

## 7.11 Further Information

The area of graph algorithms continues to be a very active field of research. There are several journals and conferences that discuss advances in the field. Here we name a partial list of some of the important meetings: ACM Symposium on Theory of Computing (STOC), IEEE Conference on Foundations of Computer Science (FOCS), ACM-SIAM Symposium on Discrete Algorithms (SODA), International Colloquium on Automata, Languages and Programming (ICALP), and European Symposium on Algorithms (ESA). There are many other regional algorithms/theory conferences that carry research papers on graph algorithms. The journals that carry articles on current research in graph algorithms are *Journal of the ACM, SIAM Journal on Computing, SIAM Journal on Discrete Mathematics, Journal of Algorithms, Algorithmica, Journal of Computer and System Sciences, Information and Computation, Information Processing Letters,* and *Theoretical Computer Science.* To find more details about some of

the graph algorithms described in this chapter we refer the reader to the books by Cormen et al. [7], Even [8], Gibbons [12], and Tarjan [28].

## Defining Terms

**Articulation vertex/cut vertex:** A vertex whose deletion disconnects a graph into two or more connected components.

**Biconnected graph:** A graph that has no articulation/cut vertices.

**Bipartite graph:** A graph in which the vertex set can be partitioned into two sets $X$ and $Y$, such that each edge connects a node in $X$ with a node in $Y$.

**Branching:** A rooted spanning tree in a directed graph, such that the root has a path in the tree to each vertex.

**Chinese postman problem:** Find a minimum length tour that traverses each edge at least once.

**Connected graph:** A graph in which there is a path between each pair of vertices.

**Cycle:** A path in which the start and end vertices of the path are identical.

**Degree:** The number of edges incident to a vertex in a graph.

**DFS forest:** A rooted forest formed by depth-first search.

**Directed acyclic graph:** A directed graph with no cycles.

**Euler tour problem:** Asks for a traversal of the edges that visits each edge exactly once.

**Eulerian graph:** A graph that has an Euler tour.

**Forest:** An acyclic graph.

**Graph isomorphism problem:** Deciding if two given graphs are isomorphic to each other.

**Leaves:** Vertices of degree one in a tree.

**Minimum spanning tree:** A spanning tree of minimum total weight.

**Path:** An ordered list of distinct edges, $\{e_i = (u_i, v_i) | i = 1, \ldots, k\}$, such that for any two consecutive edges $e_i$ and $e_{i+1}$, $v_i = u_{i+1}$.

**Planar graph:** A graph that can be drawn on the plane without any of its edges crossing each other.

**Sparse graph:** A graph in which $|E| \ll |V|^2$.

**Strongly connected graph:** A directed graph in which there is a directed path between each ordered pair of vertices.

**Topological order:** A numbering of the vertices of a DAG such that every edge in the graph that goes from a vertex numbered $i$ to a vertex numbered $j$ satisfies $i < j$.

**Traveling salesman problem:** Asks for a minimum length tour of a graph that visits all the vertices exactly once.

**Tree:** A connected forest.

**Walk:** A path in which edges may be repeated.

## Acknowledgments

Samir Khuller's research was supported by NSF Research Initiation Award CCR-9307462 and NSF CAREER Award CCR-9501355. Balaji Raghavachari's research was supported by NSF Research Initiation Award CCR-9409625.

## References


1. Ahuja, R.K., Magnanti, T.L., and Orlin, J.B., *Network Flows.* Prentice Hall, Englewood Cliffs, NJ, 1993.
2. Awerbuch, B., Complexity of network synchronization. *J. Assoc. Comput. Mach.,* 32(4), 804–823, 1985.
3. Awerbuch, B., Baratz, A., and Peleg, D., Cost-sensitive analysis of communication protocols. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing,* pp. 177–187, Quebec City, Quebec, Canada, August 22–24, 1990.
4. Awerbuch, B. and Peleg, D., Sparse partitions. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science,* pp. 503–513, St. Louis, MO, October 22–24, 1990.
5. Chazelle, B., A faster deterministic algorithm for minimum spanning trees. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science,* pp. 22–31, Miami, FL, October 20–22, 1997.
6. Cong, J., Kahng, A.B., Robins, G., Sarrafzadeh, M., and Wong, C.K., Provably good performance-driven global routing. *IEEE Trans. CAD,* 739–752, 1992.
7. Cormen, T.H., Leiserson, C.E., and Rivest, R.L., *Introduction to Algorithms.* The MIT Press, Cambridge, MA, 1989.
8. Even, S., *Graph Algorithms.* Computer Science Press, Rockville, MD, 1979.
9. Frederickson, G.N., Fast algorithms for shortest paths in planar graphs with applications. *SIAM J. Comput.,* 16(6), 1004–1022, 1987.
10. Fredman, M.L. and Tarjan, R.E., Fibonacci heaps and their uses in improved network optimization algorithms. *J. Assoc. Comput. Mach.,* 34(3), 596–615, 1987.
11. Galil, Z. and Italiano, G., Reducing edge connectivity to vertex connectivity. *SIGACT News,* 22(1), 57–61, 1991.
12. Gibbons, A.M., *Algorithmic Graph Theory.* Cambridge University Press, New York, 1985.
13. Henzinger, M.R., Klein, P.N., Rao, S., and Subramanian, S., Faster shortest-path algorithms for planar graphs. *J. Comput. Syst. Sci.,* 55(1), 3–23, 1997.
14. Hoffman, C.M., *Group-Theoretic Algorithms and Graph Isomorphism.* LNCS #136, Springer-Verlag, Berlin, 1982.
15. Hopcroft, J.E. and Tarjan, R.E., Dividing a graph into triconnected components. *SIAM J. Comput.,* 2(3), 135–158, 1973.
16. Hopcroft, J.E. and Tarjan, R.E., Efficient algorithms for graph manipulation. *Commun. ACM,* 16, 372–378, 1973.
17. Hopcroft, J.E. and Tarjan, R.E., Efficient planarity testing. *J. Assoc. Comput. Mach.,* 21(4), 549–568, 1974.
18. Karger, D.R., Klein, P.N., and Tarjan, R.E., A randomized linear-time algorithm to find minimum spanning trees. *J. Assoc. Comput. Mach.,* 42(2), 321–328, 1995.
19. Khuller, S., Raghavachari, B., and Young, N., Balancing minimum spanning trees and shortest-path trees. *Algorithmica,* 14(4), 305–321, 1995.
20. King, V., A simpler minimum spanning tree verification algorithm. *Algorithmica,* 18(2), 263–270, 1997.
21. Komls, J., Linear verification for spanning trees. *Combinatorica,* 5, 57–65, 1985.
22. Lawler, E.l., Lenstra, J.K., Rinnooy Kan, A.H.G., and Shmoys, D.B., *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization.* Wiley, New York, 1985.
23. Linial, M. and Saks, M., Low diameter graph decompositions. *Combinatorica,* 13(4), 441–454, 1993.
24. Lipton, R. and Tarjan, R.E., A separator theorem for planar graphs. *SIAM J. Appl. Math.,* 36, 177–189, 1979.
25. Nagamochi, H. and Ibaraki, T., Linear time algorithms for finding sparse $k$-connected spanning subgraph of a $k$-connected graph. *Algorithmica,* 7(5/6), 583–596, 1992.

26. Nishizeki, T. and Chiba, N., *Planar Graphs: Theory and Algorithms.* North-Holland, Amsterdam, the Netherlands, 1989.
27. Tarjan, R.E., Depth-first search and linear graph algorithms. *SIAM J. Comput.,* 1(2), 146–160, June 1972.
28. Tarjan, R.E., *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.