Chapter 0

Graph Algorithm

Graph Algorithms

❑ Elementary Definitions:

- Graphs and digraphs are useful abstractions for numerous problems and structures in operations research, computer science, electrical engineering, economics, mathematics, physics, chemistry, communications, game theory, and many other areas.

Graph

❑ Example 1

- A (hypothetical) map of airline routes between several California cities.

◆ What is the cheapest way to fly from Stockton to San Diego?

◆ Which route involves the least flying time?

◆ If one city's airport is closed by bad whether, can you still fly between any other pair of cities?

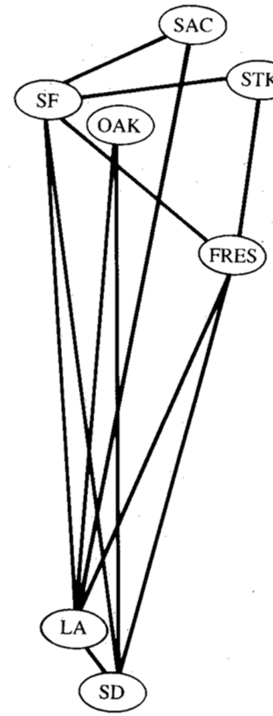


Figure 4.1 A (hypothetical) graph of nonstop airline flights between California cities.

Graph

❑ Example 2

- The flow of control in a flowchart.

◆ Does a given flowchart have any loops?

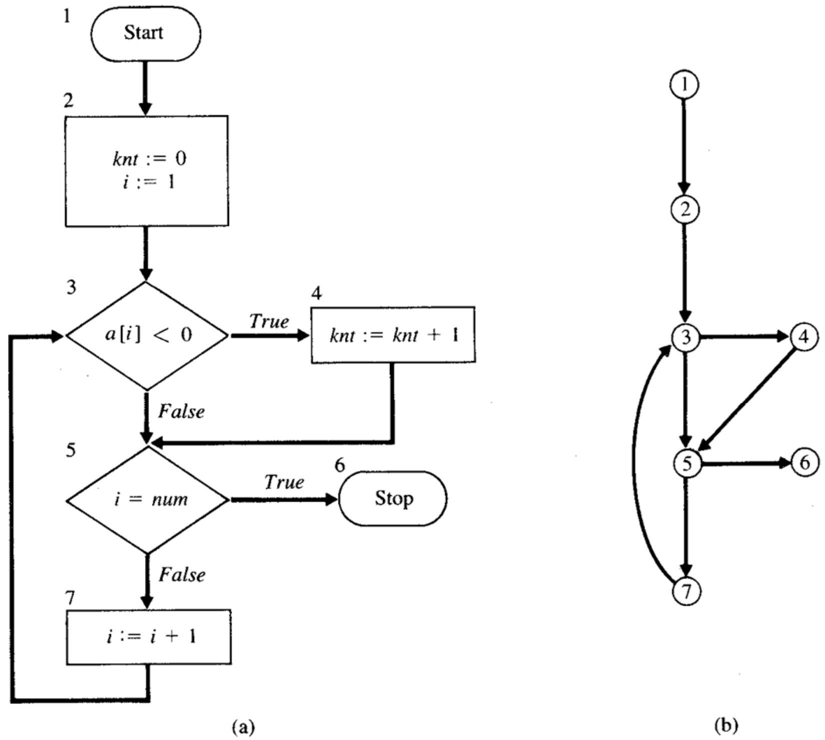


Figure 4.2 (a) A flowchart. (b) A directed graph. Arrows indicate the direction of flow.

Graph

❑ Example 3

- A binary relation:

  Let S be the set {1,2,...,9,10} and let R be the relation defined by xRy if and only if x divides y and x is not equal to y.

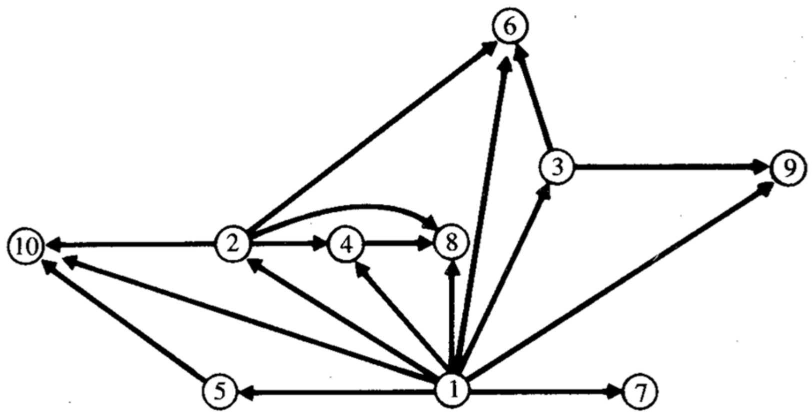
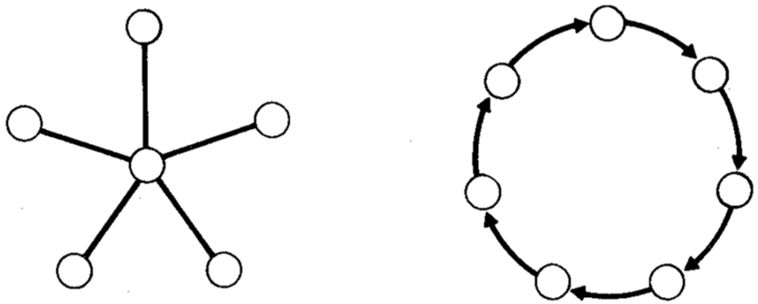


Figure 4.3 The relation *R* in Example 4.3.

◆ Is a given binary relation transitive?

Graph

- Example 4
  - Computer networks



(a) A star network.

(b) A ring network.

Figure 4.4 Computer networks.

- If one computer in a network goes down, can messages be sent between any other pair of computers in the network?

Graph

- **Definition**
  - Graph, G=(V,E)
    - V is a finite, nonempty set, elements are called vertices
    - E is a set of subsets of V of order two. elements are called edges
- **Example: in Fig 4.1**
  - V={SF, OAK, SAC, STK, FRES, LA, SD}
  - E={ {SF, STK}, {SF, SAC}, {SF, LA}, {SF, SD},
    {SF, FRES}, {SD, OAK}, {SAC, LA}, {LA, OAK},
    {LA, FRES}, {LA, SD}, {FRES, STK}, {SD, FRES} }
- An edge {u,w} in a graph will be written uw.
  Therefore uw=wu.

Digraph

- Definition
  - Digraph, G=(V,E)
    - V is a finite, nonempty set, elements are called vertices
    - E is a set of ordered pairs of distinct elements of V. Elements are called edges (or sometimes arcs)
    - for example, $(v, w) \in E$
      - v is called the tail and w the head of (v,w).
      - (v,w) is represented in the diagram as $v \rightarrow w$.
- Example: in Fig 2
  - V={1,2,3,4,5,6,7}
  - E={(1,2), (2,3), (3,4), (3,5), (4,5), (5,6), (5,7), (7,3)}
- An edge (u,w) in a digraph will be written uw.

Graph

- Note that (for all graphs and digraphs in this chapter)
  - There cannot be an edge that connects a vertex to itself in a graph or a digraph.
  - There cannot be two edges between one pair of vertices in a graph.
  - There cannot be two edges with the same orientation (direction), between one pair of vertices in a digraph.

Subgraph

- A subgraph of a graph or digraph G=(V,E)
  - a graph (or digraph) G'=(V',E') such that
    - $V' \subseteq V$
    - $E' \subseteq E$

Terms

- A *complete graph* is a graph with an edge between each pair of vertices.

- Vertices v and w are said to be *incident* with edges vw and vice versa.

- The edges of a graph or digraph G=(V,E) induce a relation called the *adjacency relation*, A, on the set of vertices.
  - Let v and w be elements of V. Then vAw (read "w is *adjacent* to v") if and only if vw is in E.

Terms

- A path from *v* to *w* in a graph or digraph G=(V,E) is a sequence of edges $v_0 v_1, v_1 v_2, \ldots, v_{k-1} v_k$
  - such that $v_0 = v,\ v_k = w$
  - $v_0, v_1, \ldots, v_k$ are all distinct.
  - The length of the path is *k*.
- A vertex *v* alone is considered to be a path of length zero from *v* to itself.
- A graph is connected if for each pair of vertices, *v* and *w*, there is a path from *v* to *w*.
- Let v and w be elements of V. Then vAw (read “w is *adjacent* to v”) if and only if vw is in E.

Terms

- A cycle in a graph or digraph G=(V,E) is a path $v_0, v_1, \ldots, v_k$ with
  - $k \geq 2$
  - $v_0 = v_k$
- A graph or digraph is acyclic if it has no cycles.
- A tree may be defined as a connected, acyclic graph;

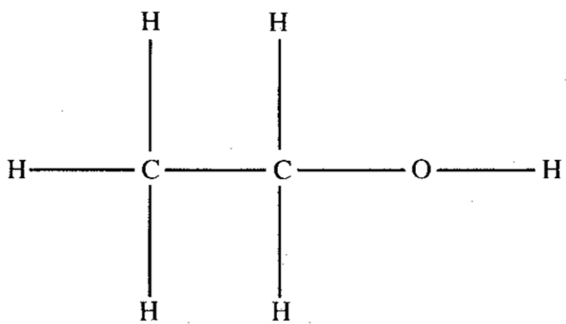


Figure 4.6 A tree: an alcohol molecule.

Terms

- A rooted tree is a tree with one vertex designated as the root.
  - The parent and child relations often used with trees can be derived once a root is specified.
- If a graph is not connected, it may be partitioned into separate connected pieces:
  - a connected component of a graph G is a maximal connected subgraph of G.

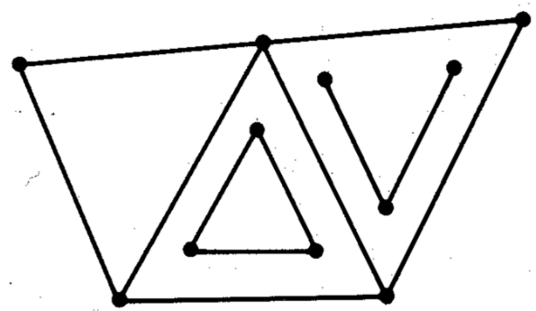


Figure 4.7 A graph with three connected components.

Terms

- A weighted graph (weighted digraph) (V,E,W) a graph (or digraph)
  - W is a function from E into R, the real numbers.
  - For an edge e, W(e) is called the weight of e.

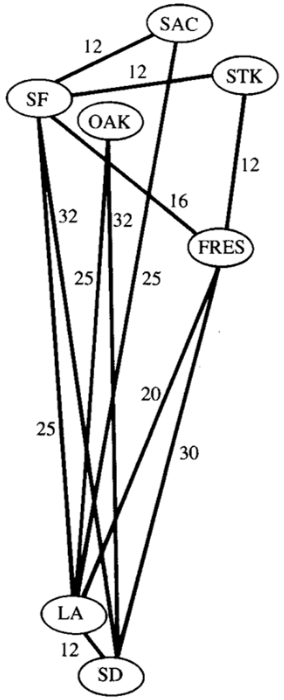


Figure 4.8 A weighted graph showing airline fares.

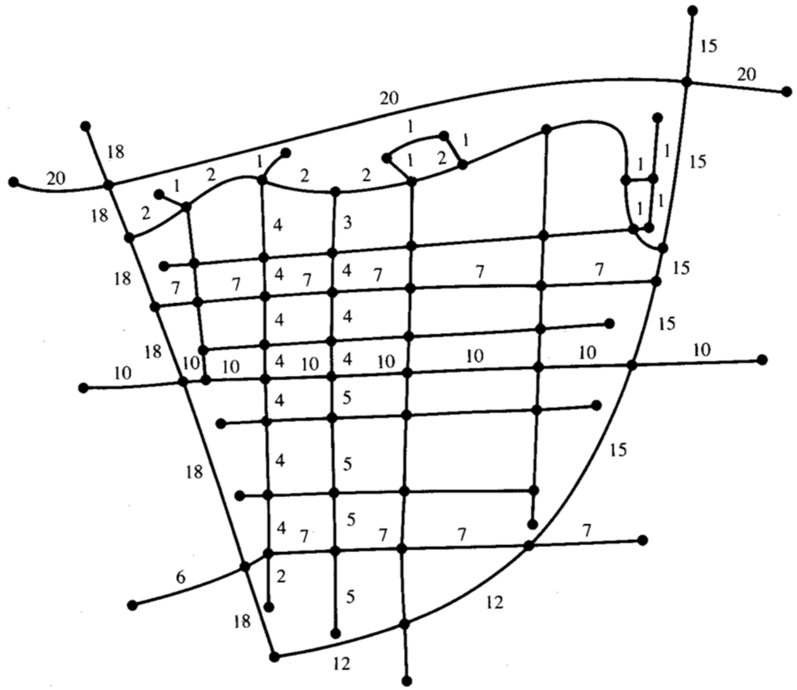


Figure 4.9 A street map showing traffic capacities.

Graph Representation

- Let G=(V,E) be a graph or digraph with |V|=n, |E|=m, and $V = \{v_0, v_1, \ldots, v_k\}$

Adjacency Matrix

- G can be represented by an *n* x *n* matrix.

$$A = (a_{ij}), \quad 1 \le i, j \le n$$

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

  - The adjacency matrix for a graph is symmetric.

Adjacency Matrix

- If G=(V,E,W) is a weighted graph or digraph, the weights can be stored in the adjacency matrix as follows:

$$A = (a_{ij}), \quad 1 \le i, j \le n$$

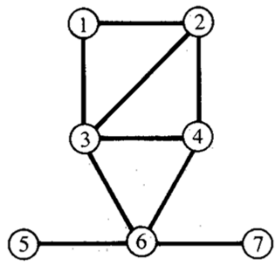
$$a_{ij} = \begin{cases} W(v_i v_j) & \text{if } v_i v_j \in E \\ c & \text{otherwise} \end{cases}$$

  - The constant *c* which depends on the interpretation of the weights and the problem to be solved.
  - If the weights are thought of as costs, (or some very high number) may be chosen for c because the cost of traversing a nonexistent edge is prohibitively high.

Adjacency List

- a data structure containing, for each vertex, *v*, a linked list indicating which vertices are adjacent to v.

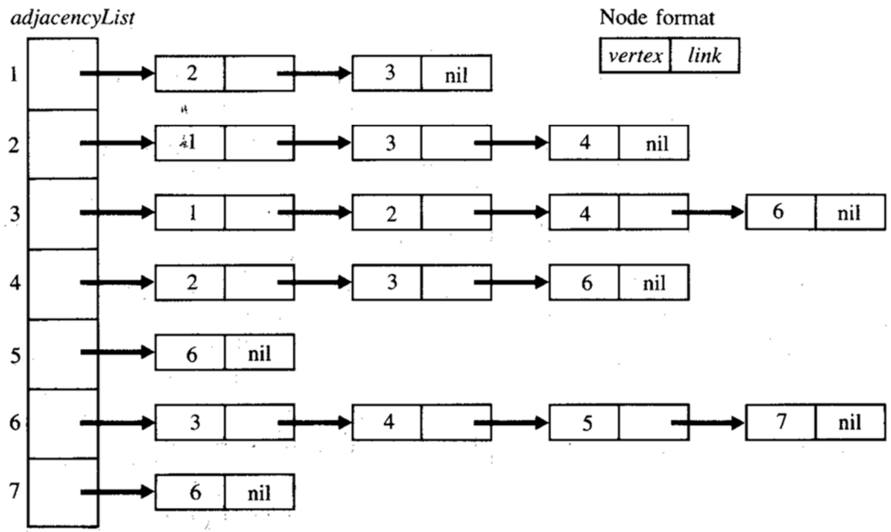
Representation for a Graph



(a) A graph

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

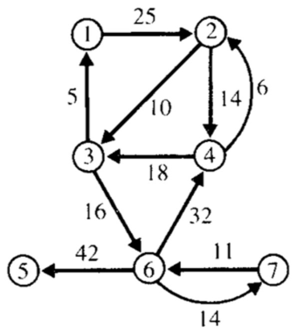
(b) Its adjacency matrix.



(c) Its adjacency list structure.

Figure 4.10 Representations for a graph.

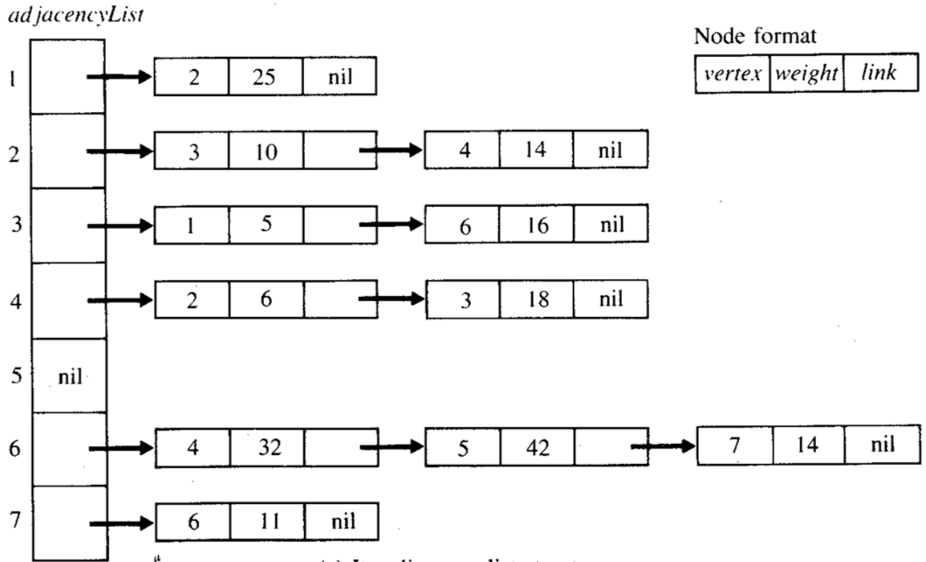
Representation of a Weighted Digraph



(a) A weighted digraph.

$$\begin{pmatrix} 0 & 25 & \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 10 & 14 & \infty & \infty & \infty \\ 5 & \infty & 0 & \infty & \infty & 16 & \infty \\ \infty & 6 & 18 & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & 32 & 42 & 0 & 14 \\ \infty & \infty & \infty & \infty & \infty & 11 & 0 \end{pmatrix}$$

(b) Its adjacency matrix.



(c) Its adjacency list structure.

Figure 4.11 Representations for a weighted digraph.

Graph Algorithms

- Algorithms for solving some problems on graphs and digraphs require that every edge be examined and processed in some way at least once.
- Since the number of edges in an adjacency matrix representation is $n(n-1)/2$, in a graph, or $n(n-1)$ in a digraph, the complexity of such algorithms will be in $\Omega(n^2)$

Traversing Graphs and Digraphs

❑ Depth First Search (DFS)

- Recursive version

Algorithm 4.5 Depth-first Search (Recursive)

```
procedure DFS(v: VertexType);
var
    w: VertexType;
begin
    visit and mark v;
    while there is an unmarked vertex w adjacent to v do
        DFS(w)
    end { while }
end { DFS }
```

Traversing Graphs and Digraphs

❑ Depth First Search (DFS)

- Iterative version : use a stack

Algorithm 4.4 Depth-first Search

Input: $G = (V, E)$, a graph or digraph represented by an adjacency list structure as described in Section 4.1.2 with $V = \{1, 2, \ldots, n\}$; $v \in V$, the vertex from which the search begins.

Comment: For a stack S, we assume that the function call $Top(S)$ returns the value of the top item on S (without popping it).

```
procedure DepthFirstSearch (adjacencyList: HeaderList; v: VertexType);
var
    S: Stack;
    w: VertexType;

begin
    initialize S to be empty;
    visit, mark, and stack v;
    while S is nonempty do
        while there is an unmarked vertex w adjacent to Top(S) do
            visit, mark, and stack w
        end { while there's an unmarked vertex ... };
        Pop S
    end { while S is nonempty }
end { DepthFirstSearch }
```

Traversing Graphs and Digraphs

❑ Breadth First Search (BFS)

- Iterative version : use a queue

Algorithm 4.6 Breadth-first Search

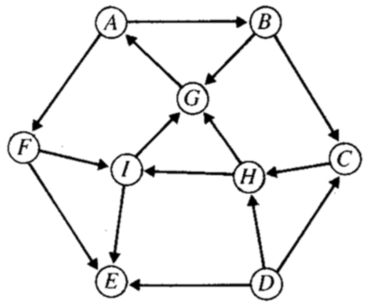
Input: $G = (V, E)$, a graph or digraph represented by an adjacency list structure as described in Section 4.1.2 with $V = \{1, 2, \ldots, n\}$; $v \in V$, the vertex from which the search begins.

Comment: For a queue Q, we assume that the function call *RemoveFromQ*(Q) returns the value of the front item on Q and removes that item from Q.

```
procedure BreadthFirstSearch (adjacencyList: HeaderList; v: VertexType);
var
    Q: Queue;
    w: VertexType;
begin
    initialize Q to be empty;
    visit and mark v; insert v in Q.
    while Q is nonempty do
        x := RemoveFromQ(Q);
        for each unmarked vertex w adjacent to x do
            visit and mark w;
            insert w in Q
        end { for }
    end { while Q is nonempty }
end { BreadthFirstSearch }
```

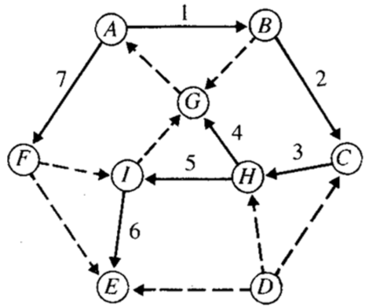
Traversing Graphs and Digraphs

- Example:

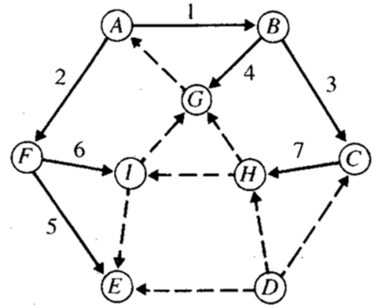


(a) A digraph.

Edges are numbered in the order traversed.



(b) Depth-first search beginning at *A*; order in which vertices are visited: *A B C H G I E F*



(c) Breadth-first search beginning at *A*; order in which vertices are visited: *A B F C G E I H*

Figure 4.21 Examples of depth-first and breadth-first searches.

DFS / BFS

- Driver routines for DFS / BFS

```
procedure Driver
    mark all vertices unvisited
    while there is an unvisited node v in V(G)
    do
          mark v “a root”
          DFS(v);         // or BFS(v)
```

Applications of DFS

- Connected Component

Algorithm 4.7 Connected Components

Input: $G = (V, E)$, a graph (not directed) represented by the adjacency list structure described in Section 4.1.2 with $V = \{1, 2, \ldots, n\}$.

Output: Lists of edges in each connected component. Also each vertex is numbered to indicate which component it is in.

```
procedure ConnectedComponents (adjacencyList: HeaderList; n: integer);
var
    mark: array[VertexType] of integer;
    { Each vertex will be marked with the number of the component it is in. }
    v: VertexType;
    componentNumber: integer;
```

```
procedure DFS(v: VertexType);
{ Does a depth-first search beginning at the vertex v }
var
    w: VertexType;
    ptr: NodePointer;
begin
    mark[v] := componentNumber;
    ptr := adjacencyList[v];
    while ptr ≠ nil do
        w := ptr↑.vertex;
        output (v,w);
        if mark[w] = 0 then DFS(w) end
        ptr := ptr↑.link
    end { while }
end { DFS }
```

```
begin { ConnectedComponents }
    { Initialize mark array. }
    for v := 1 to n do mark[v] := 0 end;
    { Find and number the connected components. }
    componentNumber := 0;
    for v := 1 to n do
        if mark[v] = 0 then
            componentNumber := componentNumber+1;
            output heading for a new component;
            DFS(v)
        end { if v was unmarked }
    end { for }
end { ConnectedComponents }
```

Depth First Search Trees

- DFS trees
  - The edges that lead to new, i.e., unmarked, vertices during a DFS of a graph or digraph G form a rooted tree called a DFS tree.
  - Those edges are called tree edges.
  - A vertex v is an ancestor of a vertex of w in a tree if v is on the path from a root to w; v is a proper ancestor of w if v is not w. If v is a (proper) ancestor of w, then w is a (proper) descendent of v.

Depth First Search Trees

- DFS trees for an undirected graph
  - the search provides an orientation for each of its edges.
  - an edge of G that is traversed from a vertex to one of its ancestors in DFS is called a back edge.
  - each edge of E(G) will be a tree edge or a back edge.

Depth First Search Trees

- DFS trees for a directed graph
  - its edges may be traversed only in the direction of their preassigned orientation.
  - There are tree edges, back edges, cross edges and descendent edges.
  - descendent edges go from a vertex to one of its descendents other than a child.
  - cross edges go from a vertex to that is not its descendents nor ancestors.

Depth First Search Trees

- Why there are no descendent edges and cross edges in if G is undirected?

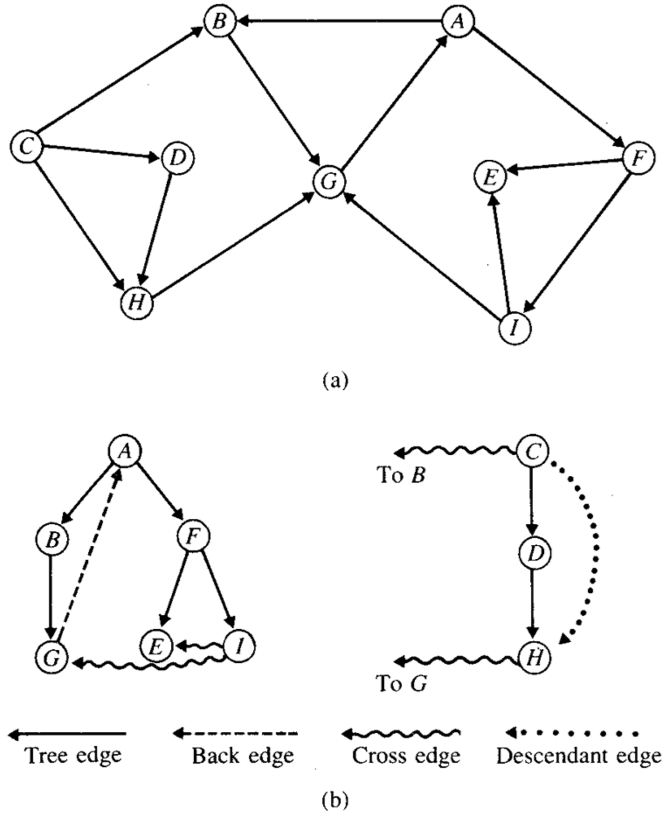


Figure 4.24 (a) A digraph. (b) Depth-first search trees for the digraph.

A Generalized DFS Skeleton

- DFS provides the structure for many elegant and efficient algorithms.
- A DFS encounters each vertex several time:
  - the vertex is first visited and becomes part of the DFS tree.
  - visited several times when the search backs up to it and attempts to branch out in a different directions.
  - the last encounters, when the search backs up from the vertex and does no pass through it or any of its descendents again.
- Depending on the problem to be solved, an algorithm will process the vertices differently when they are encountered at various stages of the traversal.

A Generalized DFS Skeleton

Algorithm 4.8 General Depth-first Search Skeleton

Input: $G = (V, E)$, a graph or digraph represented by the adjacency list structure described in Section 4.1.2 with $V = \{1, 2, \ldots, n\}$.

```
var
    mark: array[VertexType] of integer;
    markValue: integer;

procedure DFS (v: VertexType);
{ Does a depth-first search beginning at the vertex v, marking
  the vertices with markValue. }
```

A ... on

```
var
    w: VertexType;
    ptr: NodePointer;
begin
    { Process vertex when first encountered (like preorder). }
    mark[v] := markValue;
    ptr := adjacencyList[v];
    while ptr ≠ nil do
        w := ptr↑.vertex;
        { Processing for every edge.
          (If G is undirected, each edge is encountered
          twice; an algorithm may have to distinguish the
          two encounters.)                              }
        if mark[w] = 0 { unmarked } then
            { Processing for tree edges, vw. }
            DFS(w);
            { Processing when backing up to v (like inorder) }
        else
            { Processing for nontree edges.
              (If G is undirected, an algorithm may have
              to distinguish the case where w is the
              parent of v.)                      }
        end { if };
        ptr := ptr↑.link
    end { while };
    { Processing when backing up from v (like postorder) }
end { DFS }
```

A Generalized DFS Skeleton

- Sometimes we have to order in which vertices are encountered for the first time. W simply number the vertices as they are encountered by incrementing markValue. The number is called its <u>depth-first search number</u>.

Biconnected Components of a Graph

- ❑ Articulation Points and Biconnected components
- ❑ Question:
  - If any one vertex (and the edges incident with it) is removed from a connected undirected graph, is the remaining subgraph is still connected?
  - This graph property is important in graphs representing all kinds of communication or transportation networks.
- ❑ A vertex v is an <u>articulation point</u> (also called a <u>cutpoint</u>) for a graph if there are distinct vertices w and x (which are not equal to v) such that v is in every path from w to x.

Biconnected Components of a Graph

- Clearly, the removal of an articulation point would leave an unconnected graph.
- A connected graph is biconnected if and only if it has no articulation points.
- A biconnected component (called bicomponent) of a graph is a maximal biconnected subgraph.

Biconnected Components of a Graph

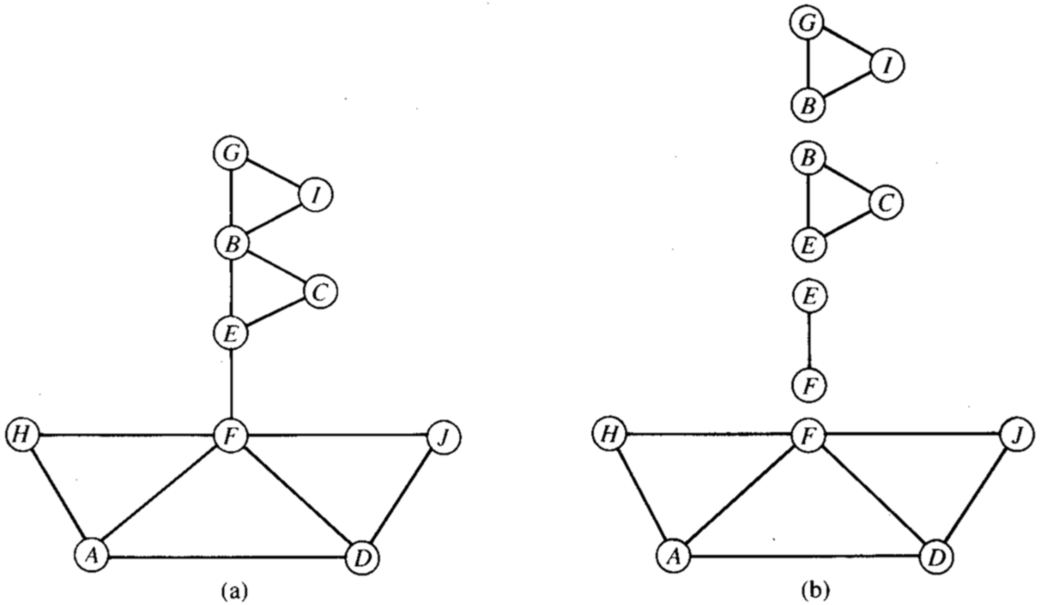


Figure 4.25 (a) A graph. (b) Its biconnected components.

Bicomponent Algorithm

- This algorithm uses
  - the depth-first search skeleton
  - the idea of a depth-first search tree.

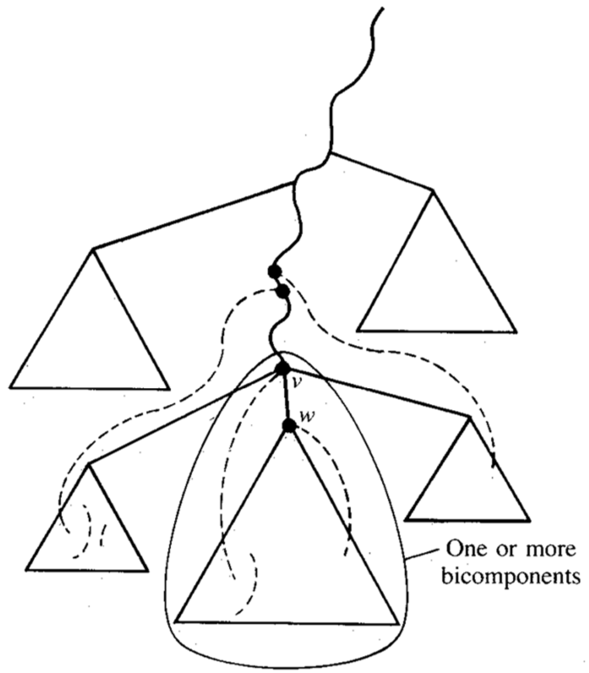


Figure 4.26 An articulation point *v* in a depth-first search tree. Every path from the root to *w* passes through *v*.

Bicomponent Algorithm

- Each vertex v keeps track of
  - dfsNumber[v]: number ordered in which they are first visited.
  - back[v]: the vertex closest to the root in the tree that one can get from v by following tree edges and certain back edges.

Bicomponent Algorithm

- Use the depth-first search skeleton
  - when a vertex v is first visited

    back[v] := dfsNumber[v]

    := depth-first search number.
  - when a vertex w is visited from v with back edge

    back[v] := min(dfsNumber[w], back[v])
  - when the search backs up from w to v.

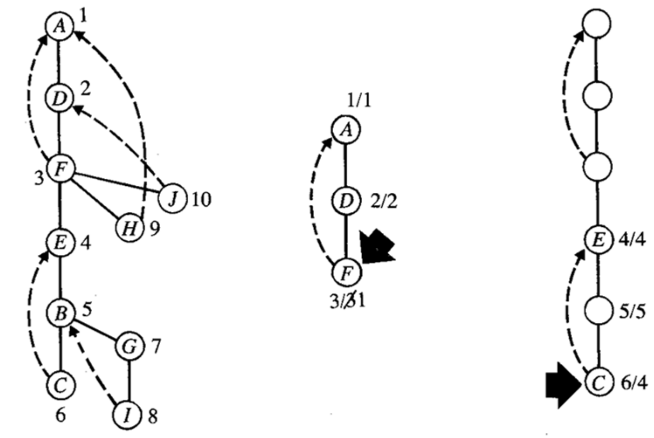
    check if v is an articulation point.

    v is an articulation point if

    back[w] >= dfsNumber[v].

    If it is not an articulation point,

    back[v] := min{back[v], back[w]}

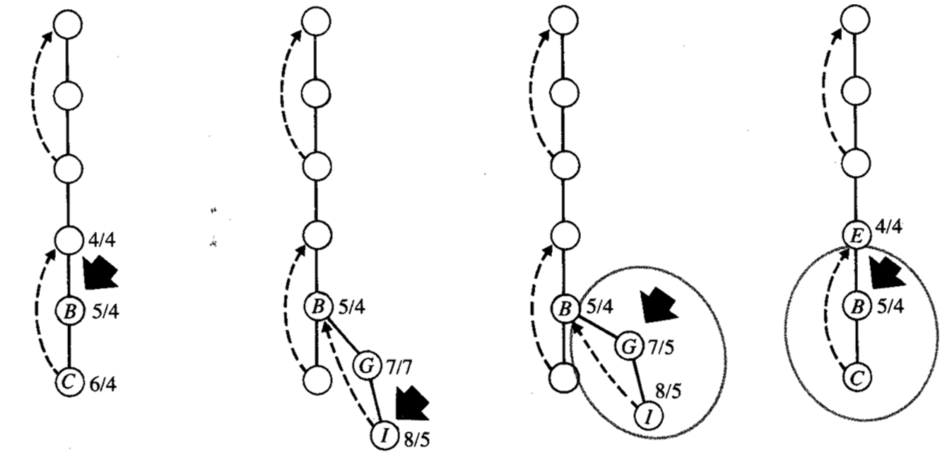


Vertex labels for (b)–(g) are *dfsNumber*/*back*.

(a)
The complete depth-first search tree.

(b)
Proceed forward; initialize values of *back*. Detect back edge *FA*; update *back*[*F*]

(c)
Continue forward. Detect back edge *CE*; update *back*[*C*]



(d)
back[*C*] < *dfsNumber*[*B*] so back up to *B* updating *back*[*B*].

(e)
Forward to *G* and *I*; detect back edge *IB*; update *back*[*I*]; *back*[*I*] < *dfsNumber*[*G*].

(f)
Back up to *G* updating *back*[*G*]; *back*[*G*] = *dfsNumber*[*B*]; remove bicomponent.

(g)
Back up to *B*; *back*[*B*] = *dfsNumber*[*E*]; remove bicomponent.

Figure 4.27 The action of the bicomponent algorithm on the graph in Fig. 4.25 (detecting the first two bicomponents).

Bicomponent Algorithm

- Theorem 4.3
  - In a depth-first search tree, a vertex v, other than the root, is an articulation point if and only if v is not a leaf and some subtree of v has no back edge incident with a proper ancestor of v.

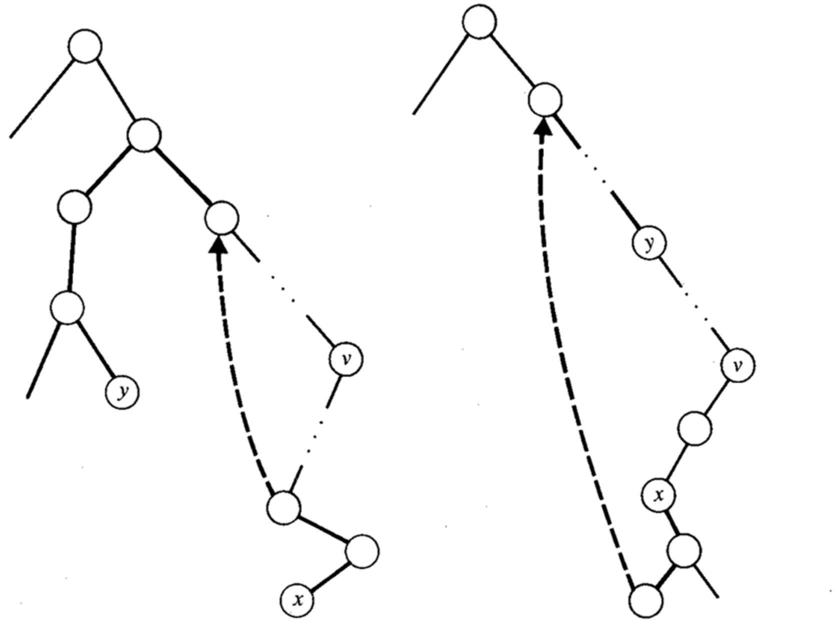


Figure 4.28 Examples for the proof of Theorem 4.3.

Bicomponent Algorithm

❑ Outline of the algorithm:

```
procedure BicompDFS(v: VertexType); { outline }
begin
    number v and initialize back[v];
    while there is an untraversed edge vw incident with v do
        if w is unmarked then
            BicompDFS(w);
            { Now backing up to v }
            if back[w] ≥ dfsNumber[v] then
                output a new bicomponent { the subtree rooted at w
                and incident edges };
            else { haven't found a new bicomponent }
                back[v] := min(back[v], back[w])
            end { of backing up from w to v }
        else { w is already in the tree }
            back[v] := min(dfsNumber[w], back[v])
        end { of processing w };
    end { while };
end { BicompDFS }
```

Bicomponent Algorithm

❑ algorithm:

Algorithm 4.9 Biconnected Components

Input: $G = (V, E)$, a connected graph (not directed) represented by linked adjacency lists with $V = \{1, 2, \ldots, n\}$.

Output: Lists of the edges in each biconnected component of G.

```
procedure Bicomponents (adjacencyList: HeaderList; n: integer);
var
    dfsNumber: array[VertexType] of integer;
    back: array[VertexType] of integer;
```

Bi m

```
    dfn: integer;
    v: VertexType;
    EdgeStack: Stack;
    { We assume that Top is a function that returns the top item on a
    stack (without popping it). }
procedure BicompDFS(v: VertexType);
var
    w: VertexType;
    ptr: NodePointer;
begin { BicompDFS }
    { Process vertex when first encountered. }
    dfn := dfn+1;
    dfsNumber[v] := dfn;  back[v] := dfn;
    ptr := adjacencyList[v];
    while ptr≠nil do
        w := ptr↑.vertex;
        if dfsNumber[w] < dfsNumber[v] then
            push vw on EdgeStack
            { else wv was a backedge already examined }
        end { if };
        if dfsNumber[w] = 0 { unmarked } then
            BicompDFS(w);
            { Now backing up to v }
            if back[w] ≥ dfsNumber[v] then
                output a heading for a new bicomponent;
                repeat
                    output Top(EdgeStack);
                    pop EdgeStack
                until vw is popped;
            else { haven't found a new bicomponent }
                back[v] := min(back[v], back[w])
            end { of backing up from w to v }
        else { w is already in the tree }
            back[v] := min(dfsNumber[w], back[v])
        end { of processing w };
        ptr := ptr↑.link;
    end { while };
end { BicompDFS }

begin { Bicomponents }
    for v := 1 to n do dfsNumber[v] := 0;
    dfn := 0;
    BicompDFS(1)
end { Bicomponents }
```

Bicomponent Algorithm

- Analysis
  - The Bicomponents algorithm takes time in

$$\Theta(\max(n,m)) = \Theta(m)$$

- Generalization
  - We can define tri-connectivity (and, in general, k-connectivity) to denote the property of having three (in general, k) vertex-disjoint paths between any pair of vertices.
  - Find an efficient algorithm to find the tri-connected (or k-connected) components of a graph.