

Graphs



Data Structures

Ching-Fang Hsu

Department of Computer Science and Information Engineering

National Cheng Kung University



Definitions

❖ A graph G consists of two sets.

- The set of vertices

 - ◆ $V(G)$

 - ◆ Finite and nonempty

- The set of edges

 - ◆ $E(G)$

 - ◆ Possibly empty

- $G = (V, E)$

❖ An undirected graph

- The pair of vertices representing an edge is unordered.



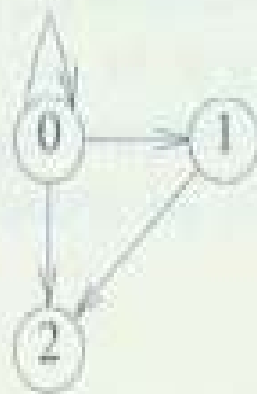
Definitions (contd.)

❖ A directed graph

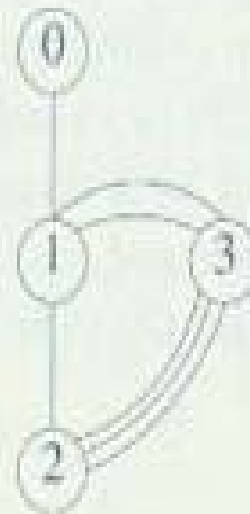
- ❑ Each edge is represented as a directed pair of vertices.
- ❑ Ex. The pair $\langle u, v \rangle$ represents an edge in which u is the tail and v is the head.

❖ The restrictions on graphs

- ❑ No self loops (aka self edges; p. 268, Fig. 6.3(a))
- ❑ A graph may not have multiple occurrences of the same edge.
 - ◆ Multigraphs (p. 268, Fig. 6.3(b))



(a) Graph with a self edge

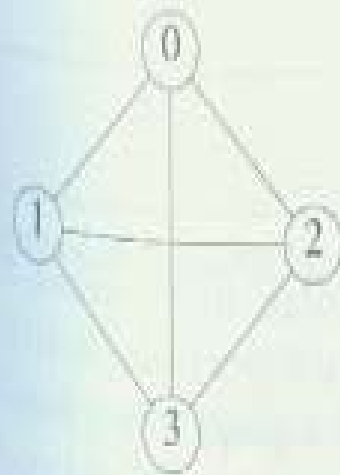


(b) Multigraph

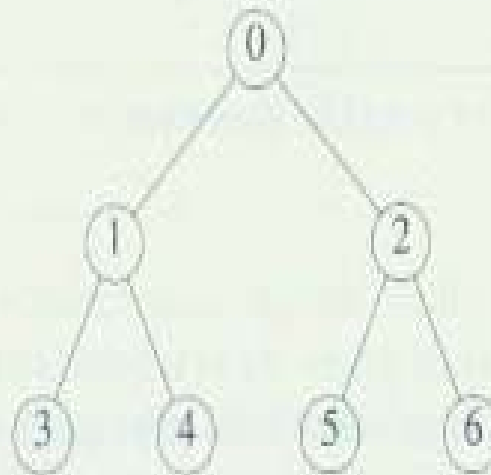
Figure 6.3: Examples of graphlike structures

Definitions (contd.)

- ❖ A complete graph (p. 267, Fig. 6.2)
 - A graph that has the maximum # of edges
 - The maximum # of edges for a graph with $|V| = n$
 - ◆ Undirected $\Rightarrow n(n-1)/2$, directed $\Rightarrow n(n-1)$
- ❖ Given an edge $(u, v) \in E(G)$, then
 - The vertices u and v are adjacent.
 - The edge (u, v) is incident on vertices u and v .
- ❖ A subgraph of G (p. 269, Fig. 6.4)
 - A graph G' such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$



(a) G_1



(b) G_2



(c) G_3

Figure 6.2: Three sample graphs

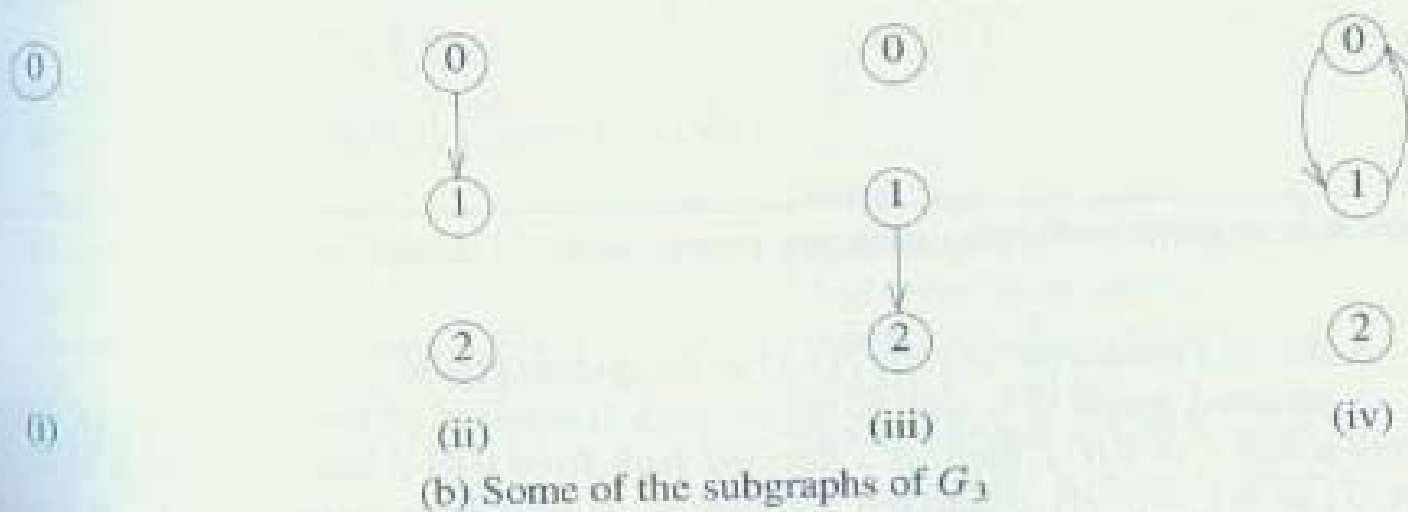
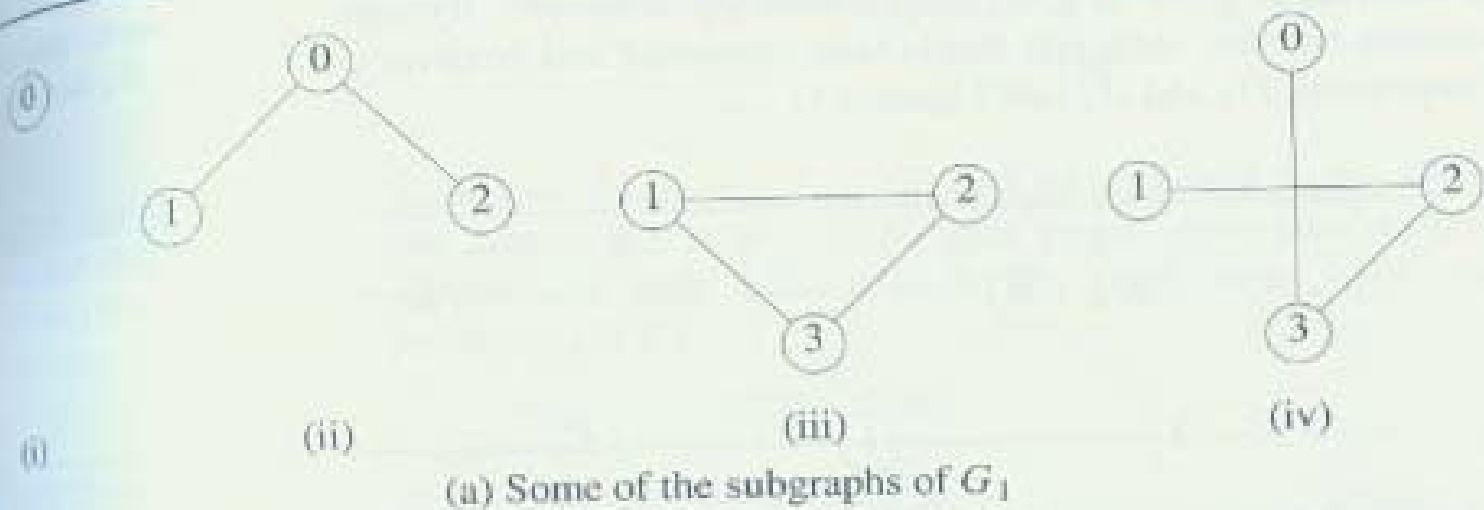


Figure 6.4: Some subgraphs

Definitions (contd.)

- ❖ A path from u to v in a graph G
 - ❑ A sequence of vertices, $u, i_1, i_2, \dots, i_k, v$, such that $(u, i_1), (i_1, i_2), \dots, (i_k, v)$ are edges in an undirected graph G .
- ❖ The length of a path
 - ❑ The number of edges on the path
- ❖ A simple path
 - ❑ A path in which all vertices, except possibly the first and the last, are distinct
 - ❑ Ex. 0, 1, 3, 2 in p. 267, Fig. 6.2 G_1

Definitions (contd.)

❖ A cycle

- ❑ A simple path in which the first and the last vertices are the same

❖ Two vertices u and v are connected in an undirected graph G if there is a path in G from u to v .

- ❑ An undirected graph is connected iff $\forall \langle u, v \rangle, u \neq v, \exists$ a path from u to v in G .

❖ A connected component (\equiv component) of an undirected graph

- ❑ A maximal connected subgraph
- ❑ Ex. p. 270, Fig. 6.5

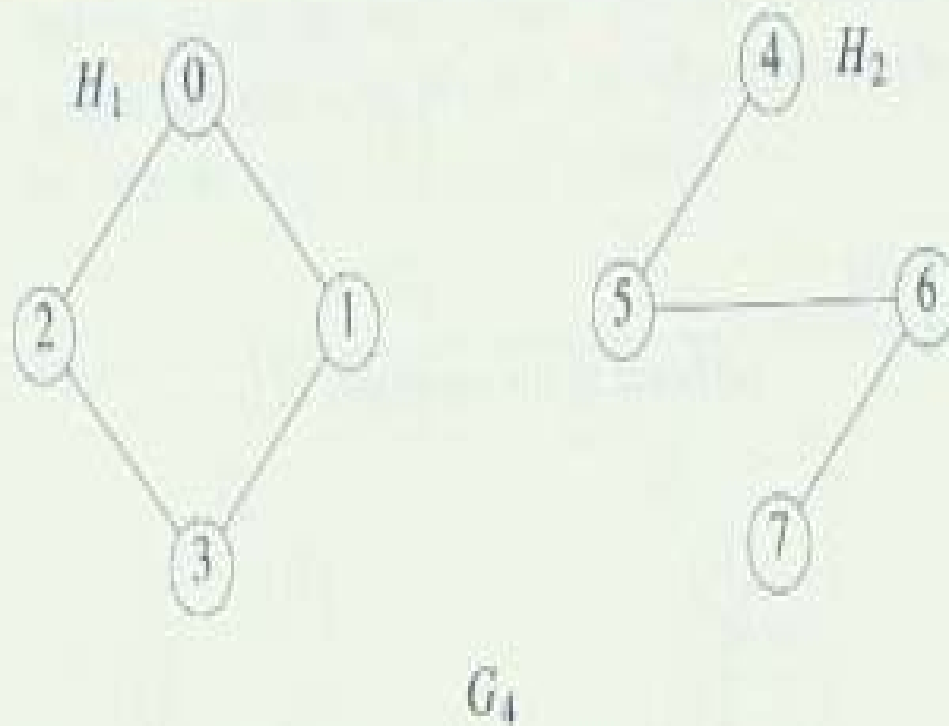


Figure 6.5: A graph with two connected components



Definitions (contd.)

- ❖ A tree

- A connected and acyclic graph

- ❖ A directed graph is strongly connected iff for every pair of distinct vertices u and v in $V(G)$, there is a directed path from u to v and also from v to u .

- ❖ A strongly connected component

- A maximal subgraph that is strongly connected

- ❖ The degree of a vertex

- The number of edges incident to that vertex

Definitions (contd.)

□ For a directed graph

◆ in-degree vs. out-degree

❖ Let d_i be the degree of vertex i in a graph G with $|V(G)| = n$ and $|E(G)| = e$, then

$$e = \left(\sum_{i=0}^{n-1} d_i \right) / 2$$

❖ Directed graphs \equiv digraphs

❖ The ADT of a graph (p. 271, ADT 6.1)

ADT *Graph* is

objects: a nonempty set of vertices and a set of undirected edges, where each edge is pair of vertices.

functions:

for all $graph \in Graph$, v , v_1 , and $v_2 \in Vertices$

<i>Graph</i> Create()	::=	return an empty graph.
<i>Graph</i> InsertVertex(<i>graph</i> , v)	::=	return a graph with v inserted. v has no incident edges.
<i>Graph</i> InsertEdge(<i>graph</i> , v_1 , v_2)	::=	return a graph with a new edge between v_1 and v_2 .
<i>Graph</i> DeleteVertex(<i>graph</i> , v)	::=	return a graph in which v and all edges incident to it are removed.
<i>Graph</i> DeleteEdge(<i>graph</i> , v_1 , v_2)	::=	return a graph in which the edge (v_1 , v_2) is removed. Leave the incident nodes in the graph.
<i>Boolean</i> IsEmpty(<i>graph</i>)	::=	if (<i>graph</i> == empty graph) return <i>TRUE</i> else return <i>FALSE</i> .
<i>List</i> Adjacent(<i>graph</i> , v)	::=	return a list of all vertices that are adjacent to v .

ADT 6.1: Abstract data type *Graph*



Graph Representation

- ❖ Three most commonly used representations
 - ❑ Adjacency matrices
 - ❑ Adjacency lists
 - ❑ Adjacency multilists

Graph Representation -- Adjacency Matrices

- ❖ A two-dimensional $n \times n$ array a
- ❖ $a[i][j] = 1$ iff the edge $(i, j) \in E(G)$; otherwise, $a[i][j] = 0$.
 - Ex. p. 272, Fig. 6.7
- ❖ The main advantage
 - Determining the degree of a vertex i is a simple task.
 - ◆ For an undirected graph: the sum of row i $\sum_{j=0}^{n-1} a[i][j]$
 - ◆ For a directed graph: the row sum \Rightarrow the out-degree
the column sum \Rightarrow the in-degree

	0	1	2	3
0	0	1	1	1
1	1	0	1	1
2	1	1	0	1
3	1	1	1	0

(a) G_1

	0	1	2
0	0	1	0
1	1	0	1
2	0	0	0

(b) G_3

	0	1	2	3	4	5	6	7
0	0	1	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0
2	1	0	0	1	0	0	0	0
3	0	1	1	0	0	0	0	0
4	0	0	0	0	0	1	0	0
5	0	0	0	0	1	0	1	0
6	0	0	0	0	0	1	0	1
7	0	0	0	0	0	0	1	0

(c) G_4

Figure 6.7: Adjacency matrices

Graph Representation -- Adjacency Matrices (contd.)

❖ The main disadvantage

- ❑ All algorithms that need to examine all edges require $O(n^2)$ time.

- ◆ $n^2 - n$ entries

- ◆ Examples

- ⇒ How many edges are there in G ?

- ⇒ Is G connected?

- ◆ Not efficient for sparse graphs

Graph Representation -- Adjacency Lists

- ❖ One chain for each vertex in G , instead of one row of the adjacency matrix
- ❖ The nodes in chain i represent the vertices that are adjacent from vertex i .
- ❖ For an undirected graph with n vertices and e edges, an array of size n and $2e$ chain nodes are required.
 - ❑ Two fields in a chain node
 - ❑ An alternative: sequentially packing the nodes on the adjacency lists

Graph Representation -- Adjacency Lists (contd.)

- ◆ No pointers; an array `node[]`

 - ⇒ `node[i]`: the starting point of the list for vertex i , $0 \leq i < n$

 - ⇒ `node[n] = n + 2e + 1`

 - ⇒ The vertices adjacent from vertex i : `node[i] ~ node[i+1] - 1`

- ◆ Example: p. 275, Fig. 6.9

- For an undirected graph

 - ◆ The degree of any vertex can be determined by counting the # of nodes in its adjacency list.

 - ◆ The total # of edges can be determined in $O(n + e)$ time.

- For a digraph

 - ◆ Out-degree \Rightarrow the total # of edges (in $O(n + e)$)

`int nodes [$n + 2 * e + 1$];`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
9	11	13	15	17	18	20	22	23	2	1	3	0	0	3	1	2	5	6	4	5	7	6

Figure 6.9: Sequential representation of graph G_4

Graph Representation -- Adjacency Lists (contd.)

◆ In-degree? More complex!

⇒ Solution 1. -- Maintaining a second set of lists called inverse adjacency lists

★ Ex. p. 275, Fig. 6.10

⇒ Solution 2. -- Changing the node structure of the adjacency lists

★ Ex. p. 276, Fig. 6.11 Lists in which nodes are shared among several lists

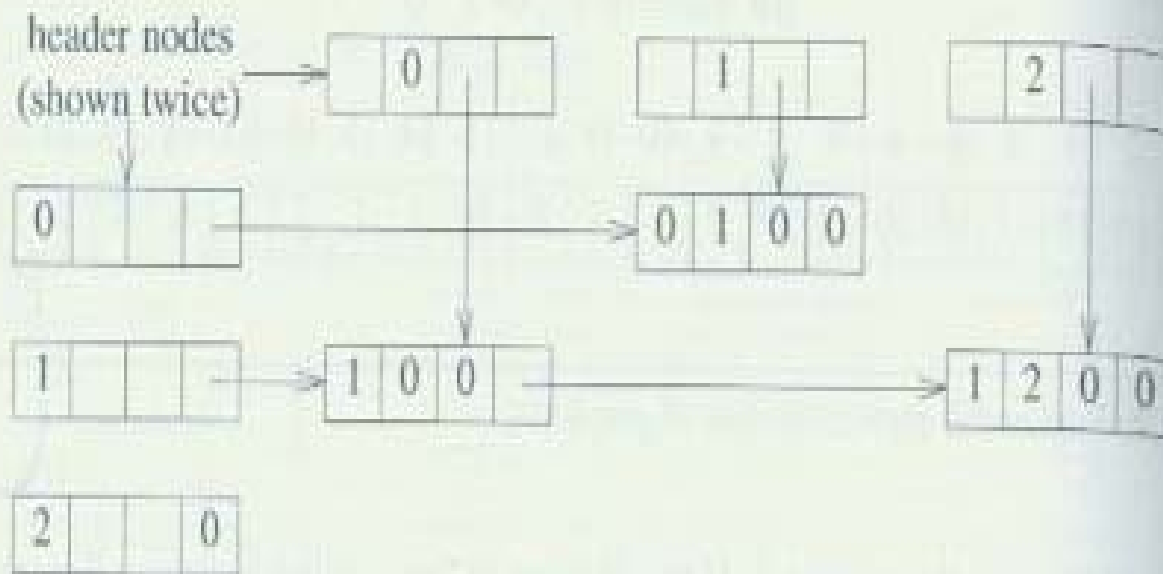


Figure 6.11: Orthogonal list representation for G_3 of Figure 6.2(c)

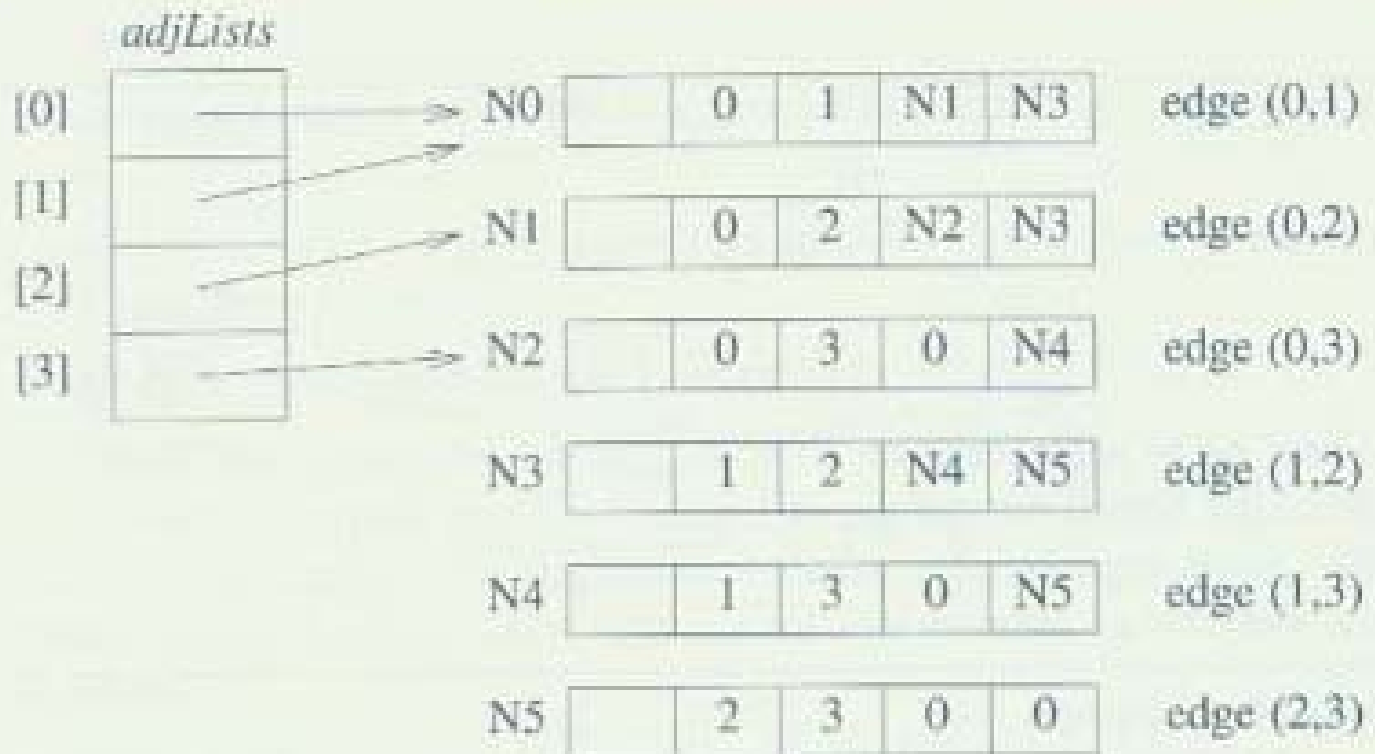


Graph Representation -- Adjacency Multilists

❖ For each edge, there is exactly one node, but this node is on the adjacency list for each of the two vertices it is incident to.

□ p. 276

□ Ex. p. 277, Fig. 6.12



The lists are

vertex 0:	N0 → N1 → N2
vertex 1:	N0 → N3 → N4
vertex 2:	N1 → N3 → N5
vertex 3:	N2 → N4 → N5

Figure 6.12: Adjacency multilists for G_1 of Figure 6.2(a)

Graph Representation -- Weighted Edges

- ❖ A graph with weighted edges is called a network.
- ❖ For an adjacency matrix
 - ❑ Replace the 1 used to signify an edge with the weight of the edge
- ❖ For adjacency lists and adjacency multilists
 - ❑ A weight field is needed!

Elementary Graph Operations

❖ Given an undirected graph, $G = (V, E)$, and a vertex $v \in V(G)$, visit all vertices reachable from v .

- ❑ Depth first search

- ◆ Similar to a preorder tree traversal

- ❑ Breadth first search

- ◆ Similar to a level order tree traversal

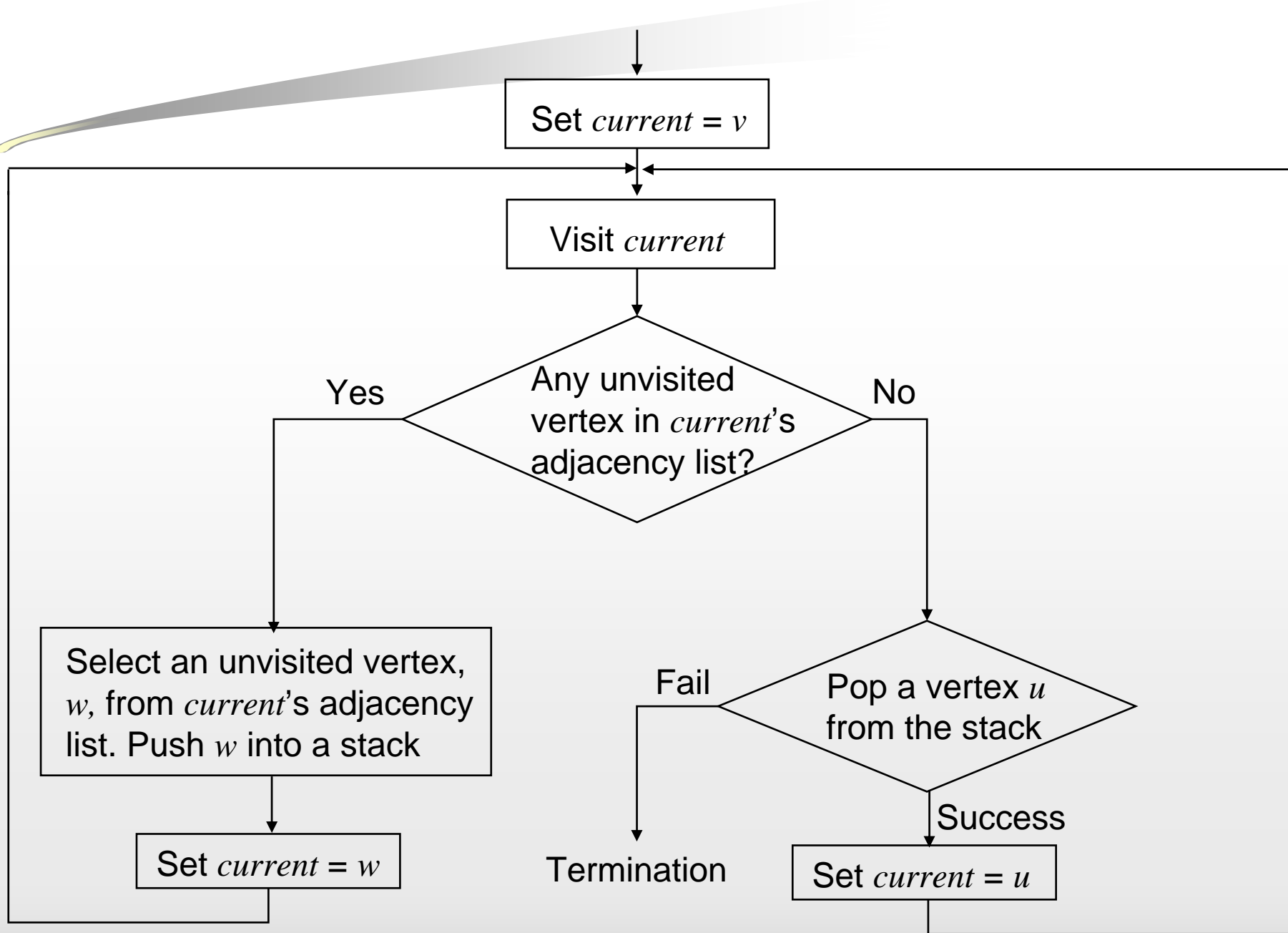
Elementary Graph Operations -- Depth First Search

❖ Assume that the start vertex is v .

```
void dfs (int v)
{
    nodePointer w;
    visited[v] = TRUE;
    printf ("%5d", v);
    for (w = graph[v]; w; w = w->link)
        if (!visited[w->vertex])
            dfs (w->vertex);
}
```

❖ Time complexity

- ❑ Adjacency lists: $O(e)$
- ❑ An adjacency matrix: $O(n^2)$



Elementary Graph Operations -- Breadth First Search

- ❖ Assume that the start vertex is v .
 - ❑ Visit each of the vertices on v 's adjacency list
 - ❑ When all the vertices on v 's adjacency list have been visited, visit all the unvisited vertices adjacent to the first vertex on v 's adjacency list.
 - ❑ p. 282, Program 6.2
 - ❑ A dynamically linked queue is exploited.
- ❖ Time complexity
 - ❑ Adjacency lists: $O(e)$
 - ❑ An adjacency matrix: $O(n^2)$

```

void bfs(int v)
/* breadth first traversal of a graph, starting at v
the global array visited is initialized to 0, the queue
operations are similar to those described in
Chapter 4, front and rear are global */
nodePointer w;
front = rear = NULL; /* initialize queue */
printf("%5d",v);
visited[v] = TRUE;
addq(v);
while (front) {
    v = deleteq();
    for (w = graph[v]; w; w = w->link)
        if (!visited[w->vertex]) {
            printf("%5d", w->vertex);
            addq(w->vertex);
            visited[w->vertex] = TRUE;
        }
    }
}

```

Program 6.2: Breadth first search of a graph

Elementary Graph Operations -- Connected Components

- ❖ Based on graph searches
- ❖ Problem 1: Determine whether or not an undirected graph is connected
 - ❑ Call $dfs(0)$ or $bfs(0)$ and then determine if there are any unvisited vertices
 - ❑ Time complexity: $O(n+e)$
- ❖ Problem 2: List the connected components of a graph
 - ❑ p. 283, Program 6.3
 - ❑ Time complexity: $O(n+e)$ for adjacency lists while $O(n^2)$ for an adjacency matrix

```
void connected(void)
/* determine the connected components of a graph */
int i;
for (i = 0; i < n; i++)
    if(!visited[i]) {
        dfs(i);
        printf("\n");
    }
}
```

Program 6.3: Connected components

Elementary Graph Operations -- Spanning Trees

❖ What is a spanning tree?

- ❑ Any tree that consists solely of edges in graph G and that includes all the vertices in G
- ❑ A minimal subgraph, G' , of G such that $V(G') = V(G)$ and G' is connected
- ❑ Given that $|V(G)| = n$, a spanning tree has $n - 1$ edges.
- ❑ p. 284, Fig. 6.17

❖ How to create a spanning tree?

- ❑ A depth first spanning tree vs. a breadth first spanning tree
- ❑ p. 285, Fig. 6.18

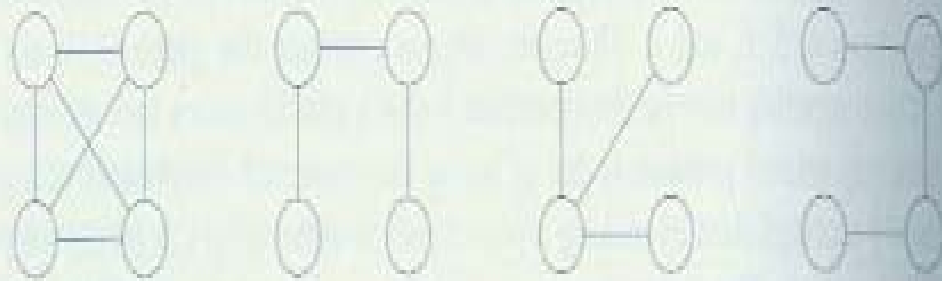
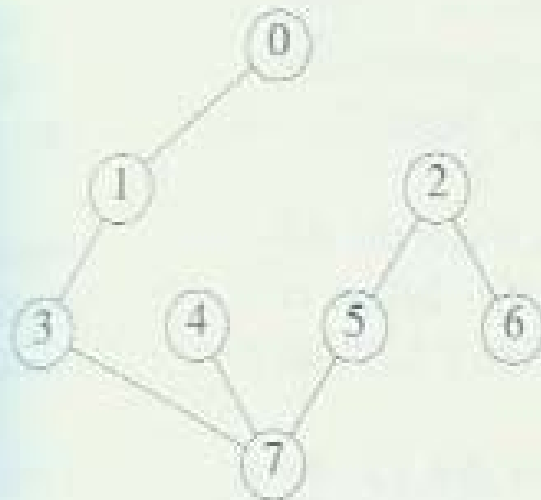
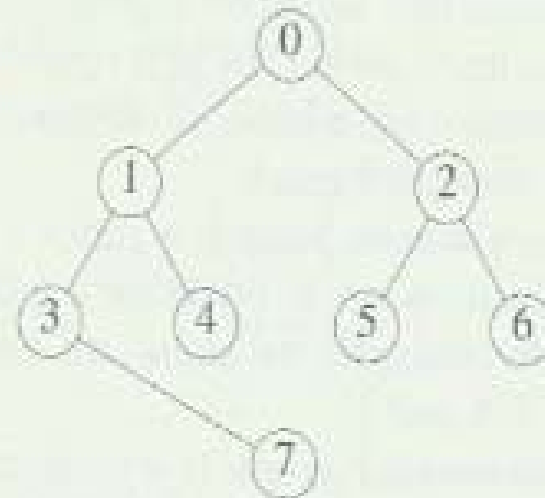


Figure 6.17: A complete graph and three of its spanning trees



(a) *DFS*(0) spanning tree



(b) *BFS*(0) spanning tree

Figure 6.18: Depth-first and breadth-first spanning trees for graph of Figure 6.16

Elementary Graph Operations -- Biconnected Components

- ❖ Assuming that G is an undirected graph
- ❖ An articulation point
 - ❑ A vertex v such that the deletion of v , together with all edges incident on v , produces a graph G' having at least two connected components
- ❖ A biconnected graph
 - ❑ A connected graph that has no articulation points.
- ❖ A biconnected component H of a connected graph G
 - ❑ A maximal biconnected subgraph of G ; G contains no other subgraph that is both biconnected and properly contains H .
 - ❑ No edge can be in two or more biconnected components of a graph.

Elementary Graph Operations -- Biconnected Components & Articulation Points (contd.)

❖ How to find the biconnected components of G ?

□ Using any depth first spanning tree of G

□ Example: p. 287, Fig. 6.19(a) and Fig. 6.20

◆ Depth first number of v ($dfn(v)$)

⇒ The sequence in which v is visited during the DFS

⇒ Generally, if u is an ancestor of v in the depth first spanning tree, then $dfn(u) < dfn(v)$.

◆ A back edge (u, v)

⇒ A nontree edge *iff* either u is an ancestor of v or v is an ancestor of u

⇒ All nontree edges are back edges.

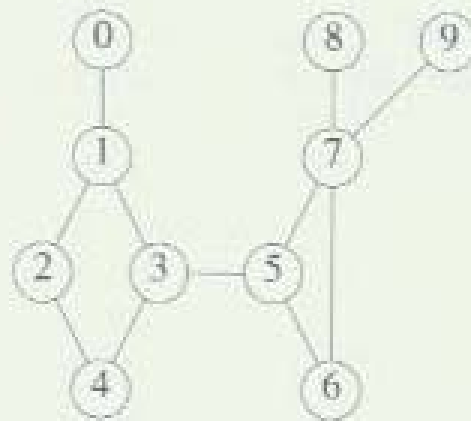
◆ A low value of a vertex u ($low(u)$)

⇒ The lowest depth dfn that we can reach from u using a path of descendants followed by at most one back edge

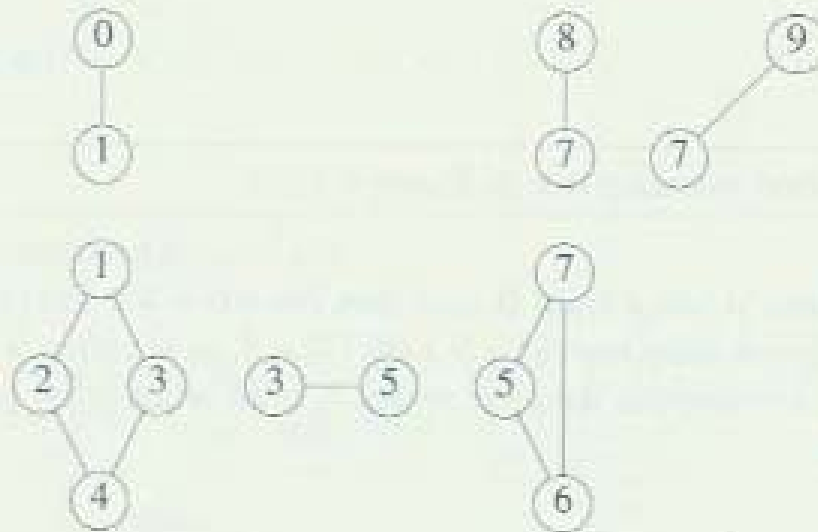
⇒ $low(u) = \min\{dfn(u), \min\{low(w) \mid w \text{ is a child of } u\}, \min\{dfn(w) \mid (u, w) \text{ is a back edge}\}\}$

Elementary Graph Operations -- Biconnected Components & Articulation Points (contd.)

- ◆ A vertex u is an articulation point iff
 - ⇒ u is either the root of the spanning tree and has two or more children, or
 - ⇒ u is not the root and u has a child w such that $low(w) \geq dfn(u)$.
- ◆ Example: p. 288, Fig. 6.21
- ◆ p. 289, Program 6.4 for dfn and low value determination
- ◆ p. 290, Program 6.6 for finding biconnected components of a graph
 - ⇒ Assuming that the graph contains at least two vertices

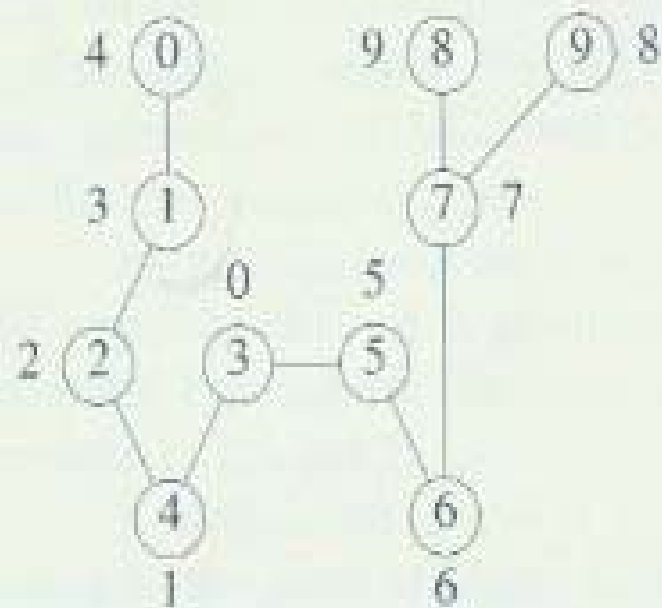


(a) Connected graph

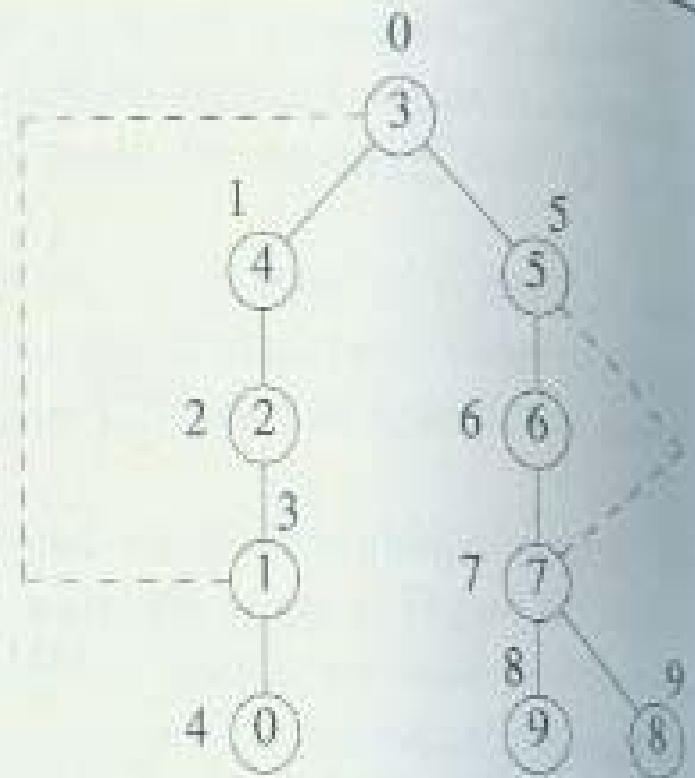


(b) Biconnected components

Figure 6.19: A connected graph and its biconnected components



(a) depth first spanning tree



(b)

Figure 6.20: Depth first spanning tree of Figure 6.19(a)

Vertex	0	1	2	3	4	5	6	7	8	9
<i>dfn</i>	4	3	2	0	1	5	6	7	9	8
<i>low</i>	4	3	0	0	0	5	5	7	9	8

Figure 6.21: *dfn* and *low* values for *dfs* spanning tree with *root* = 3


```

void dfnlow(int u, int v)
/* compute dfn and low while performing a dfs search
beginning at vertex u, v is the parent of u (if any) */
nodePointer ptr;
int w;
dfn[u] = low[u] = num++;
for (ptr = graph[u]; ptr; ptr = ptr->link) {
    w = ptr->vertex;
    if (dfn[w] < 0) { /* w is an unvisited vertex */
        dfnlow(w, u);
        low[u] = MIN2(low[u], low[w]);
    }
    else if (w != v)
        low[u] = MIN2(low[u], dfn[w]);
}
}

```

Program 6.4: Determining *dfn* and *low*

```

void bicon(int u, int v)
/* compute dfn and low, and output the edges of G by their
   biconnected components, v is the parent (if any) of u
   in the resulting spanning tree. It is assumed that all
   entries of dfn[] have been initialized to -1, num is
   initially to 0, and the stack is initially empty */
nodePointer ptr;
int w,x,y;
dfn[u] = low[u] = num++;
for (ptr = graph[u]; ptr; ptr = ptr->link) {
    w = ptr->vertex;
    if (v != w && dfn[w] < dfn[u])
        push(u,w); /* add edge to stack */
    if (dfn[w] < 0) { /* w has not been visited */
        bicon(w,u);
        low[u] = MIN2(low[u],low[w]);
        if (low[w] >= dfn[u]) {
            printf("New biconnected component: ");
            do { /* delete edge from stack */
                pop(&x, &y);
                printf(" <%d,%d>",x,y);
            } while (!(x == u) && (y == w));
            printf("\n");
        }
    }
    else if (w != v) low[u] = MIN2(low[u],dfn[w]);
}
}

```



Minimum Cost Spanning Trees

- ❖ The cost of a spanning tree of a weighted undirected graph
 - ❑ The sum of the costs (weights) of the edges in the spanning tree
- ❖ A minimum cost spanning tree is a spanning tree of least cost.
- ❖ How to obtain a minimum cost spanning tree?
 - ❑ Based on a design strategy called the greedy method

Minimum Cost Spanning Trees (contd.)

- ◆ Construct an optimal solution in stages
 - ◆ At each stage, make a decision that is the best (using some criterion) at this time
 - ◆ Make sure that the decision will result in a feasible solution
 - ◆ The selection of an item at each stage is typically based on either a least cost (e.g., minimum spanning trees problem) or a highest profit criterion.
- A feasible solution must satisfy the following constraints
- ◆ We must use only edges within the graph
 - ◆ We must use exactly $n - 1$ edges
 - ◆ We may not use edges that would produce a cycle

Minimum Cost Spanning Trees (contd.)

- ❖ Three different algorithms are introduced.
 - ❑ Kruskal's algorithm
 - ❑ Prim's algorithm
 - ❑ Sollin's algorithm

Minimum Cost Spanning Trees -- Kruskal's Algorithm

- ❖ Build a minimum cost spanning tree T by adding edges to T one at a time
 - Step 1: Select the edges in nondecreasing order of their cost
 - ◆ Maintain the edges in E as a sorted sequential list for more efficient processing
 - ◆ Time complexity of sorting the edges in E : $O(e \log e)$
 - ◆ A min heap is ideally suited!
 - Step 2: Check if an edge forms a cycle with the edges that are already in T if it is added to T
 - ◆ Use the union-find operations in Section 5.9

Minimum Cost Spanning Trees -- Kruskal's Algorithm (contd.)

- ❑ Exactly $n - 1$ edges will be selected
- ❖ p. 295, Program 6.7
 - ❑ The union-find operations require less time than choosing and deleting an edge.
 - ⇒ The total time complexity is $O(e \log e)$.
- ❖ p. 294~295, Fig. 6.22 and Fig. 6.23

```
T = {};  
while (T contains less than n-1 edges && E is not empty) {  
    choose a least cost edge (v,w) from E;  
    delete (v,w) from E;  
    if ((v,w) does not create a cycle in T)  
        add (v,w) to T;  
    else  
        discard (v,w);  
}  
if (T contains fewer than n-1 edges)  
    printf("No spanning tree\n");
```

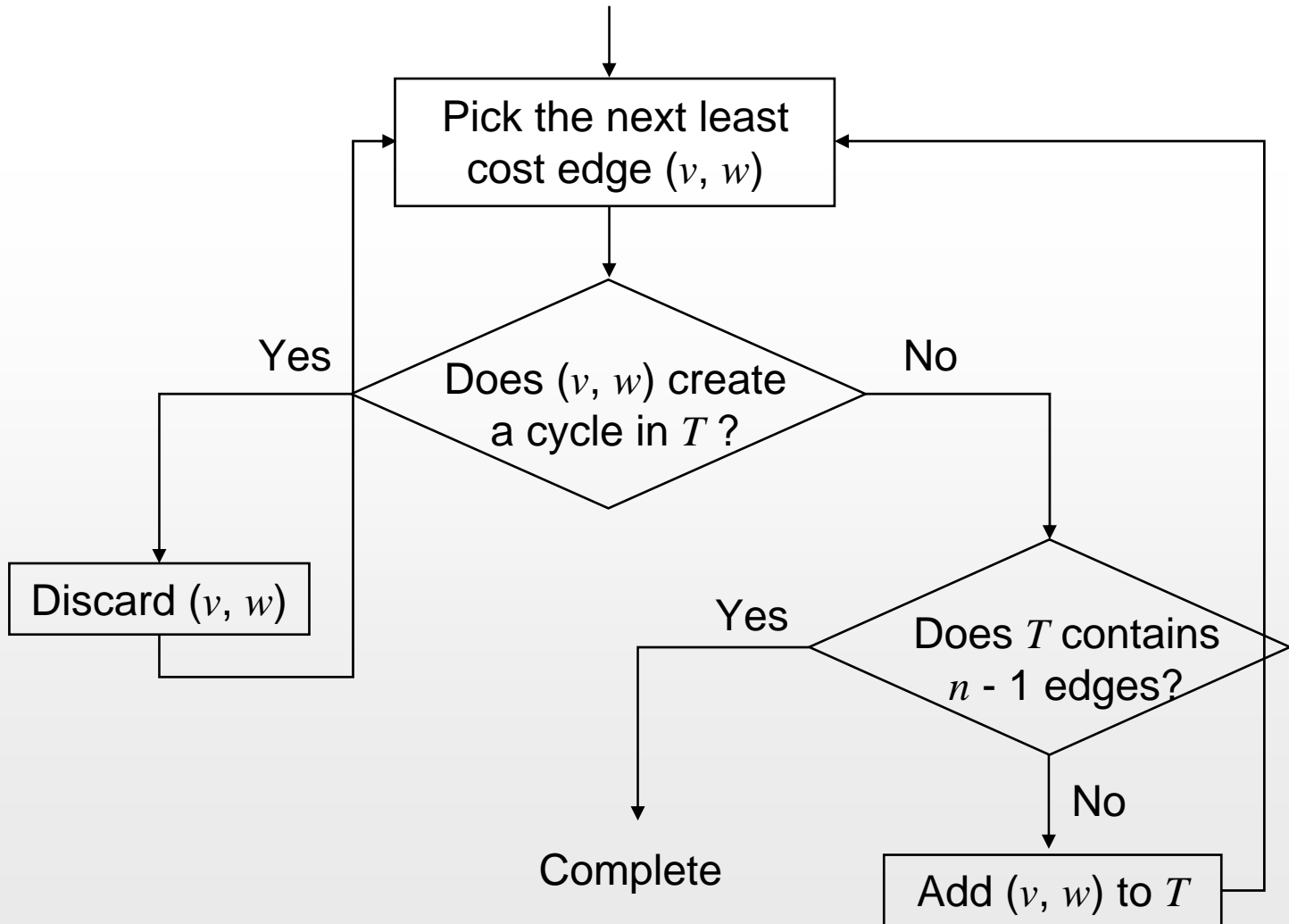
Program 6.7: Kruskal's algorithm



Figure 6.22: Stages in Kruskal's algorithm

Edge	Weight	Result	Figure
---	---	initial	Figure 6.22(b)
(0,5)	10	added to tree	Figure 6.22(c)
(2,3)	12	added	Figure 6.22(d)
(1,6)	14	added	Figure 6.22(e)
(1,2)	16	added	Figure 6.22(f)
(3,6)	18	discarded	
(3,4)	22	added	Figure 6.22(g)
(4,6)	24	discarded	
(4,5)	25	added	Figure 6.22(h)
(0,1)	28	not considered	

Figure 6.23: Summary of Kruskal's algorithm applied to Figure 6.22(a)



Minimum Cost Spanning Trees -- Prim's Algorithm

- ❖ Build a minimum cost spanning tree T by adding edges to T one at a time
 - ❑ At each stage, the set of selected edges forms a tree.
 - ◆ cf. Kruskal's (a forest at each stage)
 - ❑ Step 1: Begin with a tree T containing a single vertex
 - ◆ Any vertex in the original graph
 - ❑ Step 2: Add a least cost edge (u, v) to T such that $T \cup \{(u, v)\}$ is also a tree.
 - ◆ Exactly one of u or v is in T .

Minimum Cost Spanning Trees -- Prim's Algorithm (contd.)

- ❖ p. 297, Program 6.8
 - Time complexity: $O(n^2)$
- ❖ p. 298, Fig. 6.24

```
T = {};  
TV = {0}; /* start with vertex 0 and no edges */  
while (T contains fewer than n-1 edges) {  
    let (u, v) be a least cost edge such that u ∈ TV and  
    v ∉ TV;  
    if (there is no such edge)  
        break;  
    add v to TV;  
    add (u, v) to T;  
}  
if (T contains fewer than n-1 edges)  
    printf("No spanning tree\n");
```

Program 6.8: Prim's algorithm

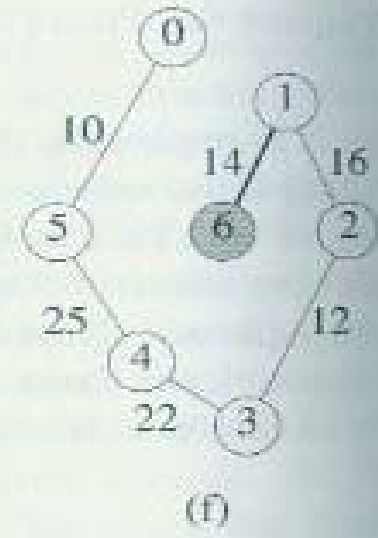
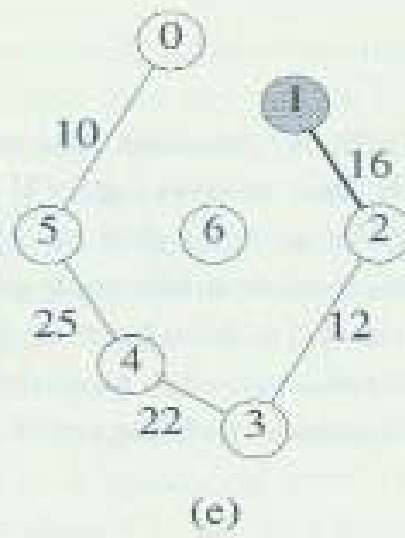
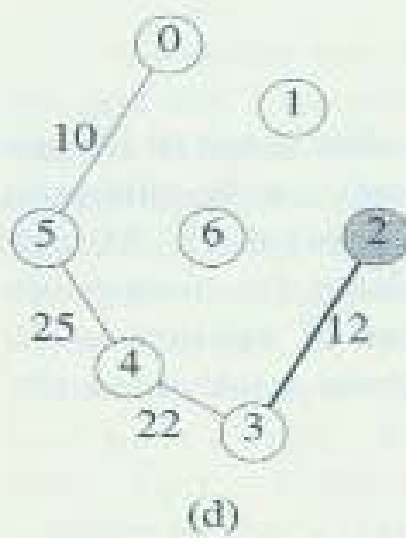
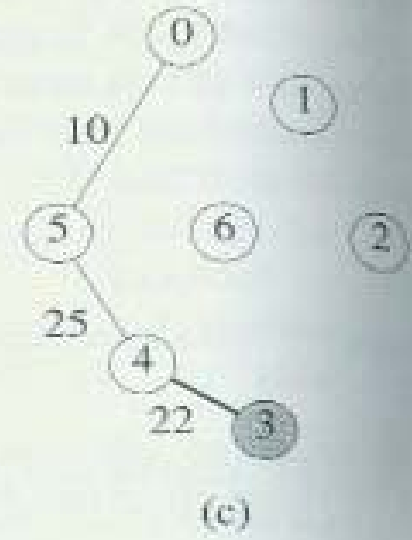
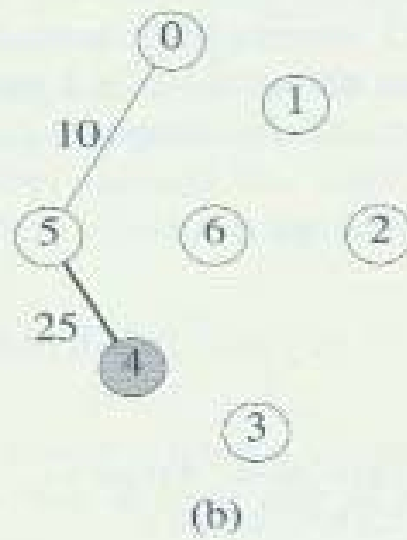
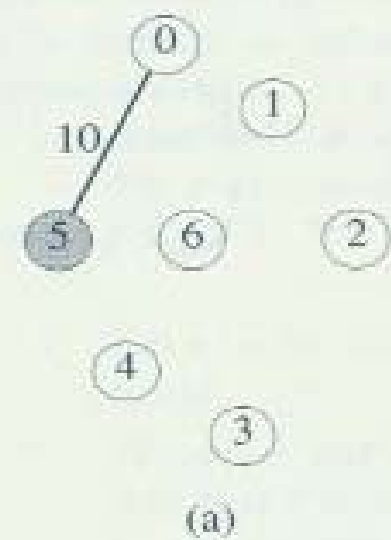
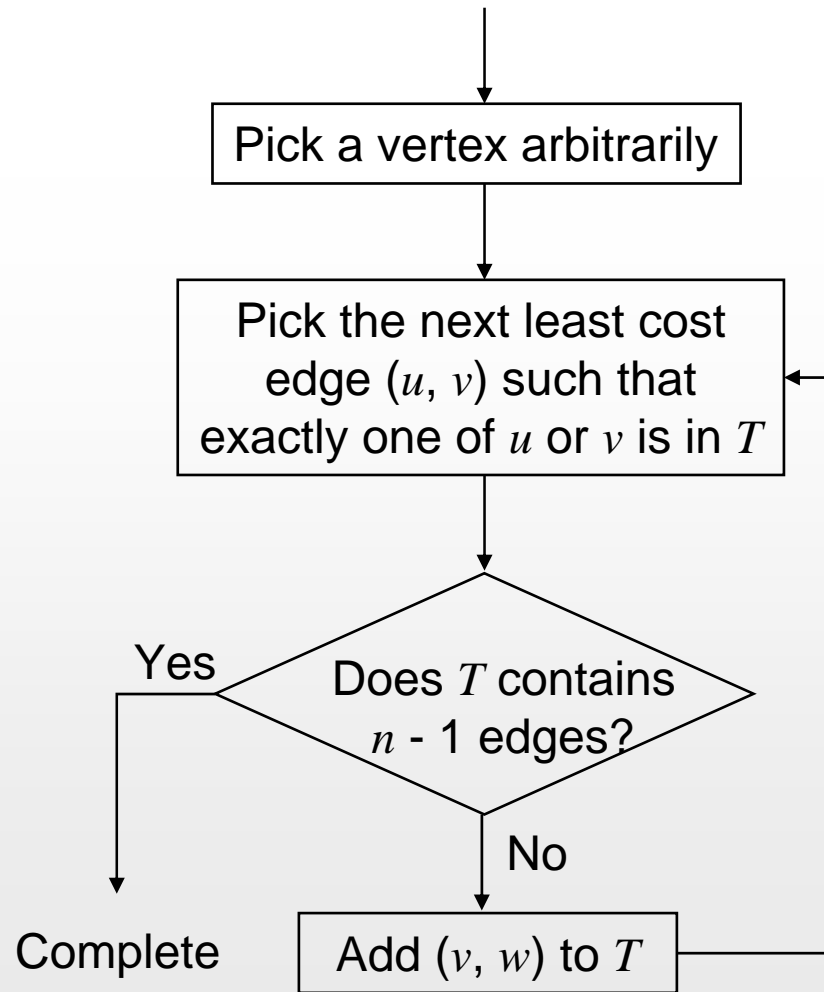


Figure 6.24: Stages in Prim's algorithm



Minimum Cost Spanning Trees -- Sollin's Algorithm

- ❖ At the start of a stage, the set of selected edges (E'), together with all n graph vertices, form a spanning forest.
 - ❑ The initial configuration: $E' = \emptyset$
- ❖ During a stage, select one edge for each tree in the forest.
 - ❑ A minimum cost edge that has exactly one vertex in the tree
 - ❑ Eliminate multiple copies of edges
- ❖ p. 299, Fig. 6.25

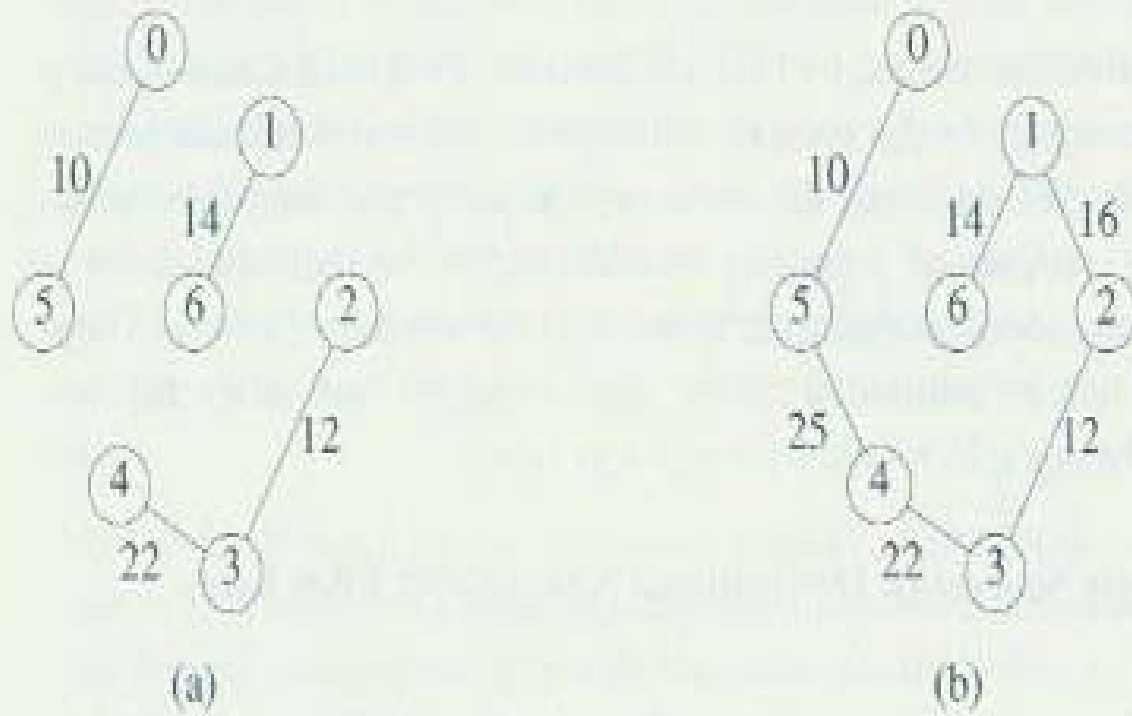
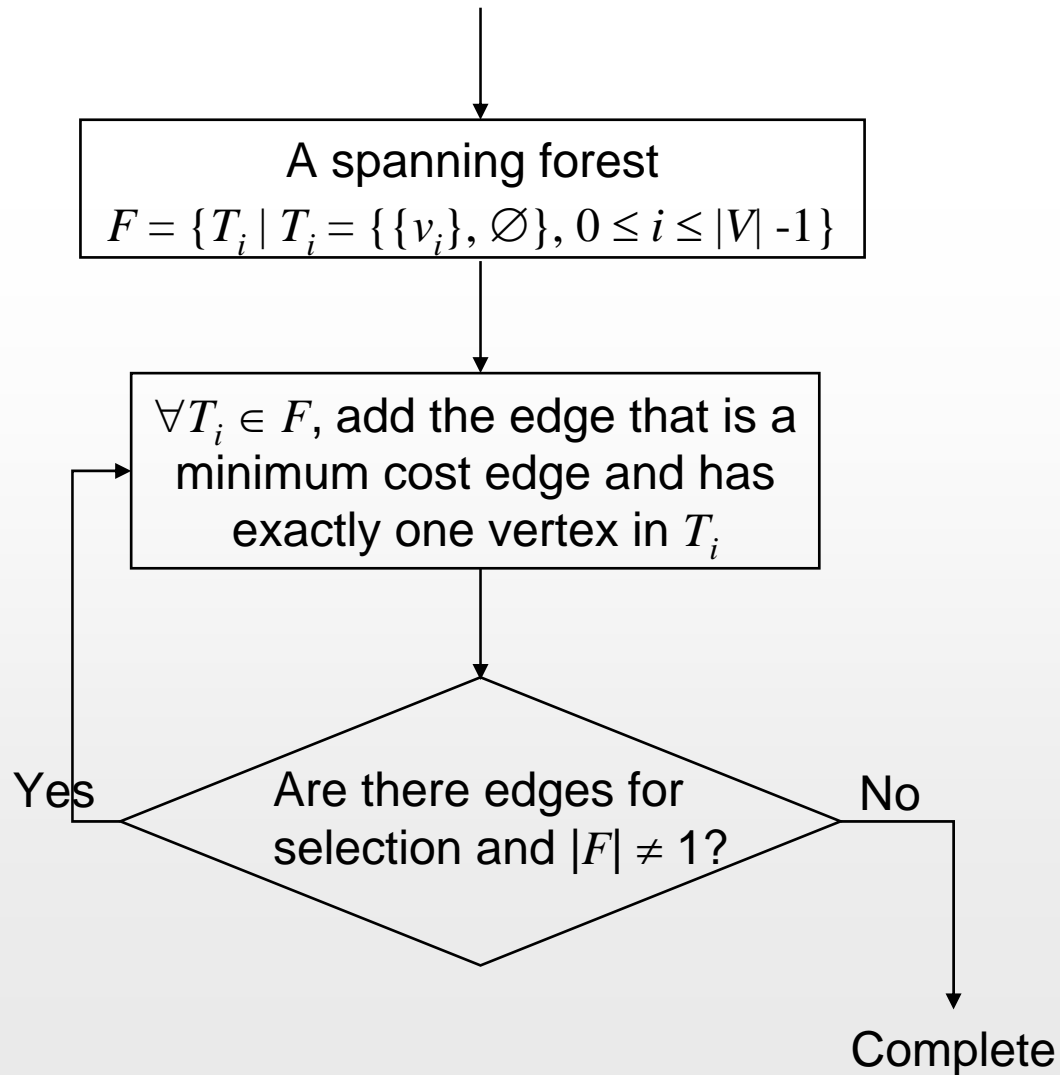


Figure 6.25: Stages in Sollin's algorithm



Shortest Paths and Transitive Closure

❖ Assumptions

- ❑ The length of a path is defined as the sum of the weights of the edges on that path.
- ❑ Directed graphs
- ❑ All weights are positive.

❖ Three problems related to finding shortest paths

- ❑ Single source all destinations
- ❑ All pairs shortest paths
- ❑ Transitive closure

Shortest Paths and Transitive Closure

-- Single Source All Destinations: Nonnegative Edge Costs

❖ Givens

- ❑ A directed graph, $G = (V, E)$
- ❑ A weighting function, $w(e)$, $w(e) > 0 \ \forall e \in E$
- ❑ A source vertex, v_0
- ❑ Let $v_0 \in S$, $S \subseteq V$ and the shortest paths of all vertices in S have been found. $\forall w \notin S$, $distance[w]$ is the length of the shortest path starting from v_0 , going through vertices only in S , and ending in w .

❖ The goal

- ❑ Determine a shortest path from v_0 to each of the remaining vertices of G

Shortest Paths and Transitive Closure

-- Single Source All Destinations:

Nonnegative Edge Costs (**contd.**)

❖ Example: p. 300, Fig. 6.26

❖ Dijkstra's algorithm

□ Initial: $S = \{v_0\}$ and $\forall i \in V, \text{distance}[i] = \text{cost}[v_0][i]$,

$$\text{cost}[u][v] = \begin{cases} w(\langle u, v \rangle) & \langle u, v \rangle \in E \\ \infty & \langle u, v \rangle \notin E \text{ and } u \neq v \\ 0 & u = v \end{cases}$$

□ Step 1: Find $u \notin S$ whose $\text{distance}[u]$ is the smallest among $\text{distance}[i]$, $\forall i \notin S$, and set $S = S \cup \{u\}$

Shortest Paths and Transitive Closure

-- Single Source All Destinations:

Nonnegative Edge Costs (contd.)

- Step 2: $\forall v \notin S$, if $distance[v] > distance[u] + cost[u][v]$,
set $distance[v] = distance[u] + cost[u][v]$
- If the shortest paths of all vertices have been
found already, stop; otherwise, go to Step 1.
- p. 302~303, Program 6.9, Program 6.10
 - ◆ Time complexity: $O(n^2)$

```

void shortestPath(int v, int cost[][MAX-VERTICES],
                 int distance[], int n, short int found[])
/* distance[i] represents the shortest path from vertex v
   to i, found[i] is 0 if the shortest path from i
   has not been found and a 1 if it has, cost is the
   adjacency matrix */
int i,u,w;
for (i = 0; i < n; i++) {
    found[i] = FALSE;
    distance[i] = cost[v][i];
}
found[v] = TRUE;
distance[v] = 0;
for (i = 0; i < n-2; i++) {
    u = choose(distance,n,found);
    found[u] = TRUE;
    for (w = 0; w < n; w++)
        if (!found[w])
            if (distance[u] + cost[u][w] < distance[w])
                distance[w] = distance[u] + cost[u][w];
}

```

Diagram 6.9: Single source shortest paths


```
int choose(int distance[], int n, short int found[])
/* find the smallest distance not yet checked */
    int i, min, minpos;
    min = INT_MAX;
    minpos = -1;
    for (i = 0; i < n; i++)
        if (distance[i] < min && !found[i]) {
            min = distance[i];
            minpos = i;
        }
    return minpos;
}
```

Program 6.10: Choosing the least cost edge

Shortest Paths and Transitive Closure

-- Single Source/ All Destinations: General Weights

- ❖ Program 6.9 does not necessarily give the correct results on graphs with negatively-weighted edges.
 - ❑ Ex. Fig. 6.29 on p. 305
- ❖ When negative edge lengths are permitted, we require that the graph have no cycles of negative length.
 - ❑ To ensure that shortest paths consist of a finite # of edges
 - ◆ Ex. Fig. 6.30 on p. 305

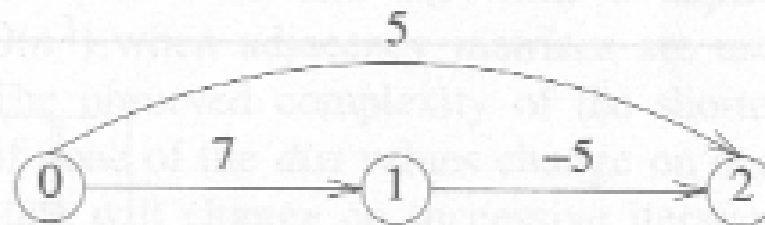


Figure 6.29: Directed graph with a negative-length edge

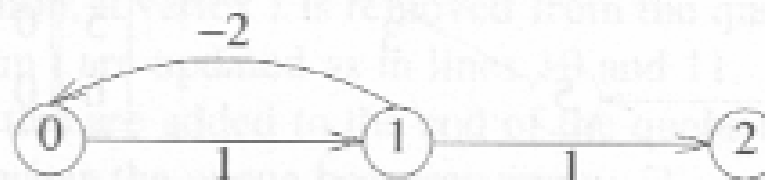


Figure 6.30: Directed graph with a cycle of negative length

Shortest Paths and Transitive Closure

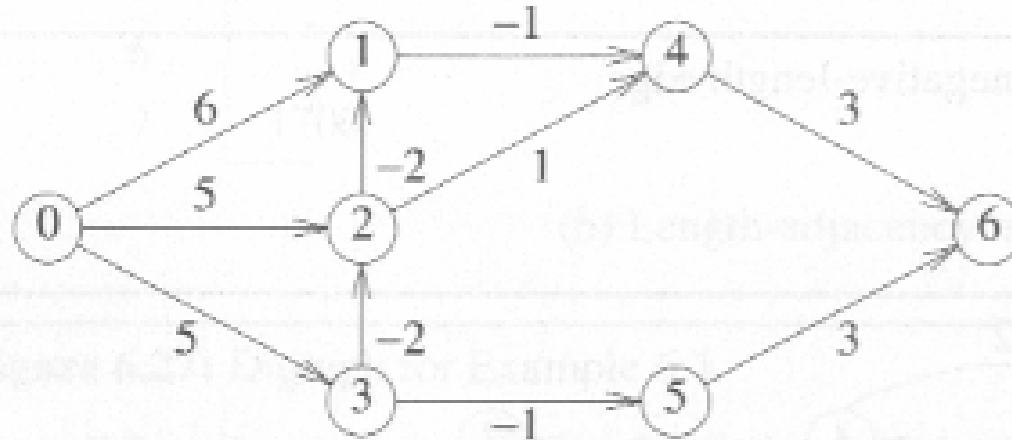
-- Single Source/ All Destinations: General Weights (contd.)

- ❖ With abovementioned assumption, there is a shortest path between any two vertices of an n -vertex graph that has at most $n-1$ edges on it.
- ❖ $dist^l[v][u]$: the length of a shortest path from the source vertex v to vertex u under the constraint that the shortest path contains at most l edges
 - $dist^1[v][u] = length[v][u], 0 \leq u < n$
 - $dist^{n-1}[v][u]$: the length of an undirected shortest path from v to u

Shortest Paths and Transitive Closure

-- Single Source/ All Destinations: General Weights (contd.)

- ❖ So, our goal then is to compute $dist^{n-1}[u]$ for all u .
 - Using dynamic programming methodology
- ❖ $dist^k[u] = \min\{dist^{k-1}[u], \min_i\{dist^{k-1}[i] + length[i][u]\}\}$
 - Ex. Fig. 6.31 on p. 306
- ❖ Bellman and Ford algorithm




(a) A directed graph

	$dist^k[7]$						
k	0	1	2	3	4	5	6
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

(b) $dist^k$

Figure 6.31: Shortest paths with negative edge lengths



```
void BellmanFord (int n, int v))
{
    for (int i = 0; i < n; i++)
        dist[i] = length[v][i];
    for (int k = 2; k <= n-1; k++)
        for (each u such that u!= v and u has
            at least one incoming edge)
            for (each <i, u> in the graph)
                if (dist[u] > dist[i]+length[i][u])
                    dist[u] = dist[i] + length[i][u]
}
```

Shortest Paths and Transitive Closure

-- Single Source/ All Destinations: General Weights (contd.)

❖ Program 6.11 on p. 307

- ❑ $O(n^3)$ when adjacency matrices are used

- ❑ $O(ne)$ when adjacency lists are used

- ❑ Complexity reduction

- ◆ Terminate the **for** loop either after $n-1$ iterations or after the first iteration in which no *dist* values are changed

- ◆ Maintain a queue of vertices i whose *dist* value changed on the previous iteration of the **for** loop.

- ⇒ The only values for i that need to be considered during the next iteration

Shortest Paths and Transitive Closure -- All Pairs Shortest Paths

- ❖ Find the shortest paths between all pairs of vertices, $v_i, v_j, i \neq j$
- ❖ Solution 1: Apply Dijkstra's algorithm for n vertices
 - ❑ Time complexity: $O(n^3)$
- ❖ Solution 2: Use the dynamic programming method
 - ❑ The basic principle: Decompose a problem into subproblems and each subproblem will be solved by the same approach recursively

Shortest Paths and Transitive Closure -- All Pairs Shortest Paths (contd.)

□ $A^k[i][j]$: The cost of the shortest path from i to j , using only those intermediate vertices with an index $\leq k$.

◆ $A^{-1}[i][j] = cost[i][j]$

◆ The cost of the shortest path from i to j is $A^{n-1}[i][j]$.

□ The basic idea is to begin with the matrix A^{-1} and successively generate the matrices $A^0, A^1, A^2, \dots, A^{n-1}$.

Shortest Paths and Transitive Closure -- All Pairs Shortest Paths (contd.)

- Given A^{k-1} , $\forall i$ and j , $A^k[i][j]$ can be derived by applying one of the following two rules
 - ◆ $A^k[i][j] = A^{k-1}[i][j]$; does not go through the vertex with index k
 - ◆ $A^k[i][j] = A^{k-1}[i][k] + A^{k-1}[k][j]$
- $\Rightarrow A^k[i][j] = \min \{A^{k-1}[i][j], A^{k-1}[i][k] + A^{k-1}[k][j]\}, k \geq 0$
- ◆ No cycle with a negative length is allowed!
 - \Rightarrow p. 303,
- ◆ p. 309, Program 6.12
 - \Rightarrow Time complexity: $O(n^3)$

```

void allCosts(int cost[][MAX-VERTICES],
              int distance[][MAX-VERTICES], int n)
/* compute the shortest distance from each vertex
   to every other, cost is the adjacency matrix,
   distance is the matrix of computed distances */
int i,j,k;
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        distance[i][j] = cost[i][j];
for (k = 0; k < n; k++)
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            if (distance[i][k] + distance[k][j] <
                distance[i][j])
                distance[i][j] =
                    distance[i][k] + distance[k][j];
}

```

Program 6.12: All pairs, shortest paths function

Shortest Paths and Transitive Closure -- Transitive Closure

- ❖ Closely related to the all pairs, shortest paths problem
- ❖ Given a digraph $G = (V, E)$, determine if there is a path from i to j , $\forall i, j \in V$
 - Two cases are of interest
 - ◆ The transitive closure of a graph
 - ◆ The reflexive transitive closure of a graph
 - **Definition** (The transitive closure matrix of a digraph; A^+)
 - ◆ $A^+[i][j] = \begin{cases} 1 & \text{if there is a path of length } > 0 \text{ from } i \text{ to } j \\ 0 & \text{otherwise} \end{cases}$

Shortest Paths and Transitive Closure -- Transitive Closure (contd.)

□ **Definition** (The reflexive transitive closure matrix of a digraph; A^*)

$$\blacklozenge A^*[i][j] = \begin{cases} 1 & \text{if there is a path of length } \geq 0 \text{ from } i \text{ to } j \\ 0 & \text{otherwise} \end{cases}$$

□ **Example:** p. 311, Fig. 6.34

◆ A^+ and A^* differ only on the diagonal.

❖ Find A^+ with minor modifications on the function for the all pairs shortest paths problem (*allcosts* on p. 309)



(a) Digraph G

	0	1	2	3	4
0	0	1	0	0	0
1	0	0	1	0	0
2	0	0	0	1	0
3	0	0	0	0	1
4	0	0	1	0	0

(b) Adjacency matrix A

	0	1	2	3	4
0	0	1	1	1	1
1	0	0	1	1	1
2	0	0	1	1	1
3	0	0	1	1	1
4	0	0	1	1	1

(c) A^+

	0	1	2	3	4
0	1	1	1	1	1
1	0	1	1	1	1
2	0	0	1	1	1
3	0	0	1	1	1
4	0	0	1	1	1

(d) A^*

Figure 6.34: Graph G and its adjacency matrix A , A^+ , and A^*

Shortest Paths and Transitive Closure -- Transitive Closure (contd.)

$$\text{cost}[i][j] = \begin{cases} 1 & \text{if } \langle i, j \rangle \in E \\ +\infty & \text{otherwise} \end{cases}$$

□ When *allcosts* terminates, obtain A^+ and A^* as follows:

$$A^+[i][j] = \begin{cases} 1 & \text{iff } \text{distance}[i][j] < +\infty \\ 0 & \text{otherwise} \end{cases}$$

$$A^*[i][j] = \begin{cases} A^+[i][j] & \text{if } i \neq j \\ 1 & \text{otherwise} \end{cases}$$

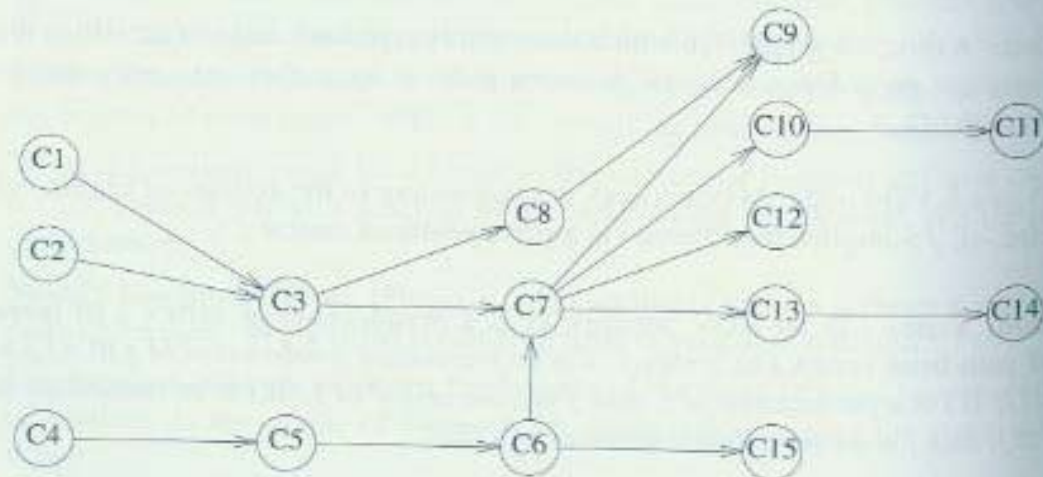
□ Time complexity: $O(n^3)$

Activity on Vertex (AOV) Networks

- ❖ Many projects can be divided into several subprojects called activities.
 - ❑ Ex. p. 316, Fig. 6.37
- ❖ Precedence relations among activities
 - ❑ Can be represented as a directed graph
- ❖ **Definition** (AOV networks)
 - ❑ An AOV network is a directed graph G in which the vertices represent activities and the edges represent precedence relations between activities.

Course number	Course name	Prerequisites
C1	Programming I	None
C2	Discrete Mathematics	None
C3	Data Structures	C1, C2
C4	Calculus I	None
C5	Calculus II	C4
C6	Linear Algebra	C5
C7	Analysis of Algorithms	C3, C6
C8	Assembly Language	C3
C9	Operating Systems	C7, C8
C10	Programming Languages	C7
C11	Compiler Design	C10
C12	Artificial Intelligence	C7
C13	Computational Theory	C7
C14	Parallel Algorithms	C13
C15	Numerical Analysis	C5

(a) Courses needed for a computer science degree at a hypothetical university



(b) AOV network representing courses as vertices and prerequisites as edges

Figure 6.37: An activity-on-vertex (AOV) network

Activity on Vertex (AOV) Networks (contd.)

❖ Definition (Predecessors)

- Vertex i in an AOV network G is a predecessor of vertex j iff there is a directed path from i to j .
Vertex i is an immediate predecessor of vertex j iff $\langle i, j \rangle$ is an edge in G .

❖ Definition (Successors)

- If i is a predecessor of j , then j is a successor of i .
If i is an immediate predecessor of j , then j is an immediate successor of i .

❖ Ex. p. 316, Fig. 6.37

Activity on Vertex (AOV) Networks (contd.)

- ❖ If an AOV network represents a feasible project, the precedence relations must be both transitive and irreflexive.

❖ Definition

□ A relation \cdot is transitive iff for all triples i, j, k , $i \cdot j$ and $j \cdot k \Rightarrow i \cdot k$. A relation \cdot is irreflexive on a set S if $i \cdot i$ is false $\forall i \in S$.

- ❖ We can show that a precedence relation is irreflexive by proving that the network is a directed acyclic graph (dag).

Activity on Vertex (AOV) Networks (contd.)

❖ Definition (Topological order)

- A topological order is a linear ordering of the vertices of a graph such that, for any two vertices, i, j , if i is a predecessor of j in the network then i precedes j in the linear ordering.

❖ Ex. p. 317

❖ An algorithm to sort activities into topological order

- p. 318, Program 6.13
- Ex. p. 318, Fig. 6.39

```
1  Input the AOV network. Let n be the number of vertices.
2  for (i = 0; i < n; i++) /* output the vertices */
3  {
4      if (every vertex has a predecessor) return;
5          /* network has a cycle and is infeasible */
6      pick a vertex v that has no predecessors;
7      output v;
8      delete v and all edges leading out of v;
9  }
```

Program 6.13: Design of an algorithm for topological sorting

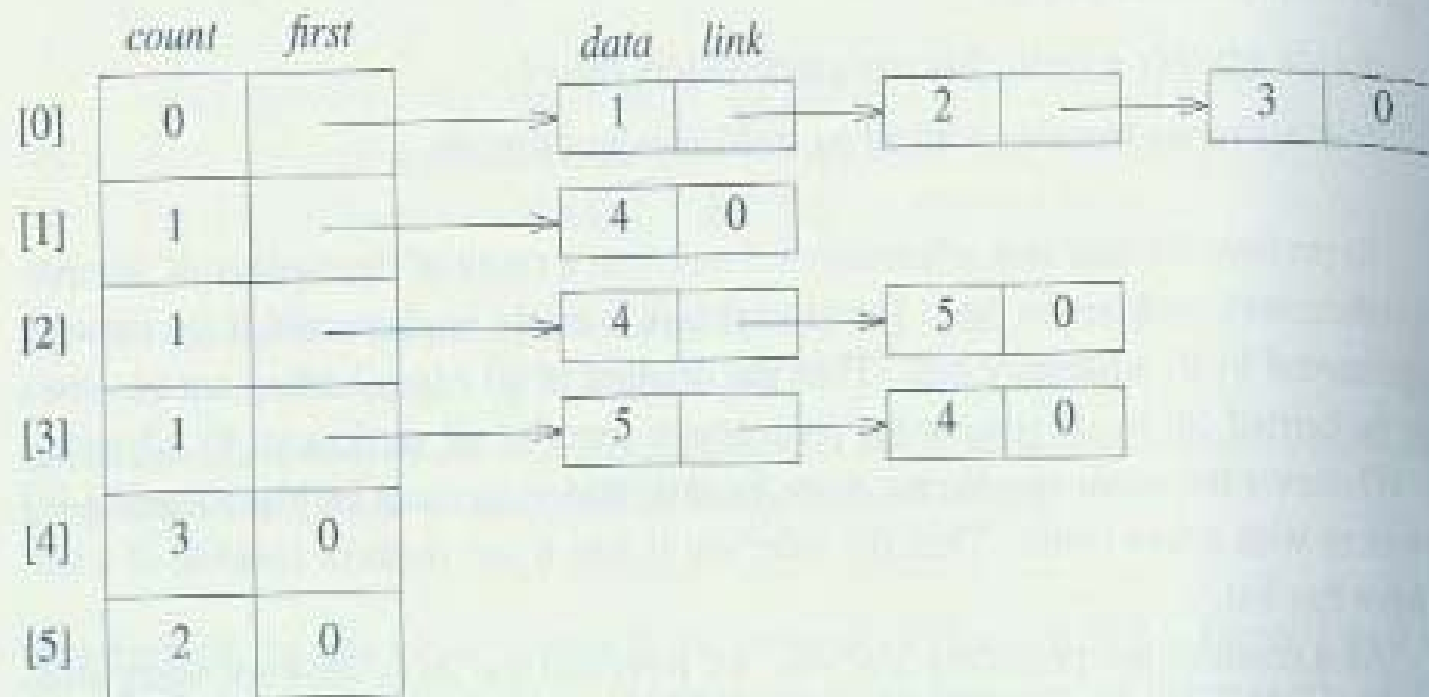
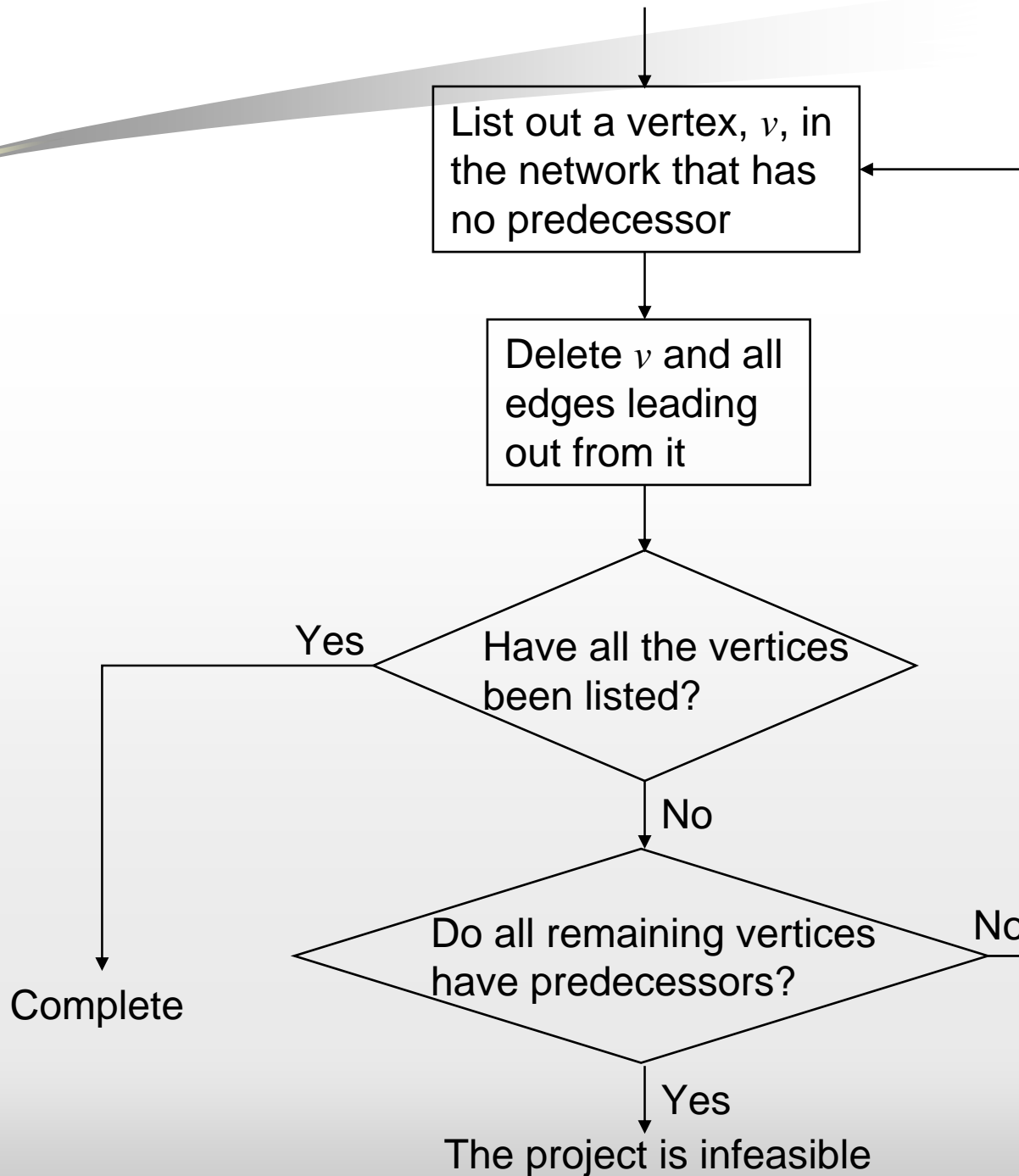


Figure 6.39: Internal representation used by topological sorting algorithm



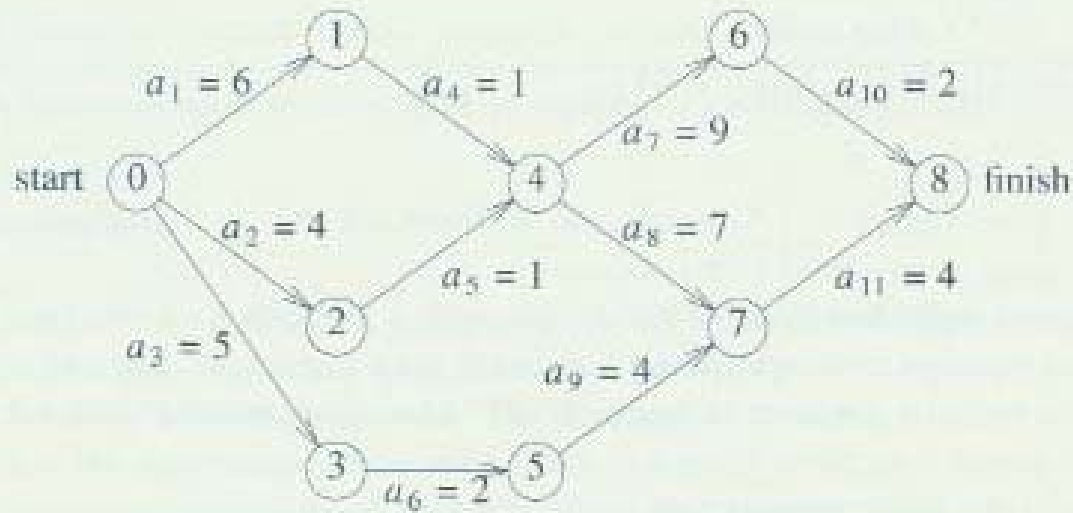


Activity on Edge (AOE) Networks

- ❖ The directed edges represent tasks or activities to be performed on a project.
- ❖ The vertices represent events which signal the completion of certain activities.
 - ❑ Activities represented by edges leaving a vertex cannot be started until the event at that vertex has occurred.
- ❖ “Start project”; “finish project”
- ❖ A critical path is a path that has the longest length from the start vertex to the finish vertex.
 - ❑ A network may have more than one critical path.

Activity on Edge (AOE) Networks (contd.)

- ❖ The earliest time an event v_i can occur = the length of the longest path from the start vertex 0 to i .
 - ❑ Example: p. 322, Fig. 6.40
 - ❑ It determines the earliest start time for all activities represented by edges leaving that vertex.
 - ◆ $e(i)$ for activity a_i ; $e(7) = e(8) = 7$
- ❖ The latest time of activity $a_i \equiv l(i)$
 - ❑ The latest time the activity may start without increasing the project duration
 - ◆ $l(6) = 8, l(8) = 7$



(a) Activity network of a hypothetical project

<i>event</i>	<i>interpretation</i>
0	start of project
1	completion of activity a_1
4	completion of activities a_4 and a_5
7	completion of activities a_8 and a_9
8	completion of project

(b) Interpretation of some of the events in the network of (a)

Figure 6.40: An AOE network

Activity on Edge (AOE) Networks (contd.)

❖ A critical activity

- ❑ An activity for which $e(i) = l(i)$

- ❑ $l(i) - e(i)$

 - ◆ A measure of how critical an activity is

- ❑ Critical paths determination

 - ◆ Delete all noncritical activities from the AOE networks

 - ◆ Generate all the paths from the start to finish vertex

Activity on Edge (AOE) Networks (contd.)

❖ How to calculate $e(i)$ and $l(i)$, $\forall i$?

□ Assume that activity a_i is represented by edge $\langle k, l \rangle$.

$$e(i) = ee[k]$$

$$l(i) = le[l] - \text{duration of activity } a_i$$

□ It is easier to first obtain $ee[j]$ and $le[j]$, for all events, j .

◆ A forward stage -- computing $ee[j]$

⇒ Starting with $ee[0] = 0$

⇒ $ee[j] = \max_{i \in P(j)} \{ee[i] + \text{duration of } \langle i, j \rangle\}$

⇒ $P(j)$ – the set of all vertices adjacent to vertex j

Activity on Edge (AOE) Networks (contd.)

- ◆ A backward stage – computing $le[j]$
 - ⇒ Starting with $le[n - 1] = ee[n - 1]$
 - ⇒ $le[j] = \min_{i \in S(j)} \{le[i] - \text{duration of } \langle j, i \rangle\}$
 - ⇒ $S(j)$ – the set of vertices adjacent from vertex j
- Using the values of ee and of le to compute $e[i]$ and $l[i]$ and the degree of criticality (also called slack) for each task