CHAPTER 6:

GRAPHS

6.1 The Graph Abstract Data Type

- A graph problem example:
  - Köenigsberg bridge problem

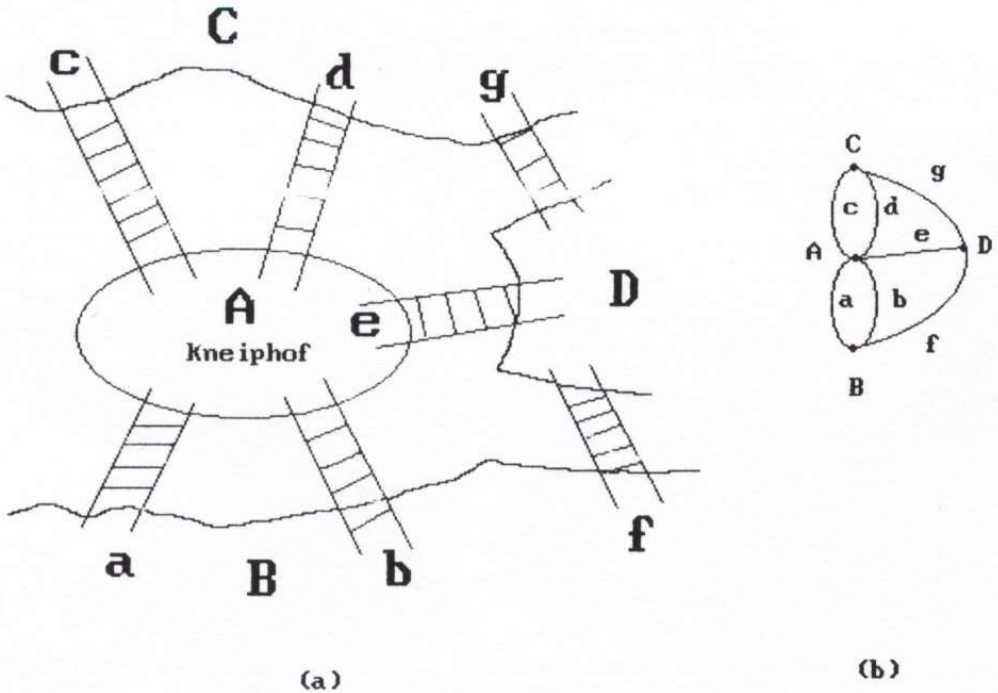


Figure 6.1: The bridges of Koenigsberg

- Definitions:
  - A graph, *G*, consists of two sets, *V* and *E*. *G* = (*V*, *E*)
    - *V* is a finite, nonempty set of *vertices*.
    - *E* is a set of pairs of vertices; these pairs are called *edges*.
    - *V(G)* and *E(G)* will represent the set of vertices and edges, respectively.
  - An *undirected graph* is one in which the pair of vertices representing any edge is unordered,
  - An *directed graph* is one in which we represent each edge as a directed pair of vertices.

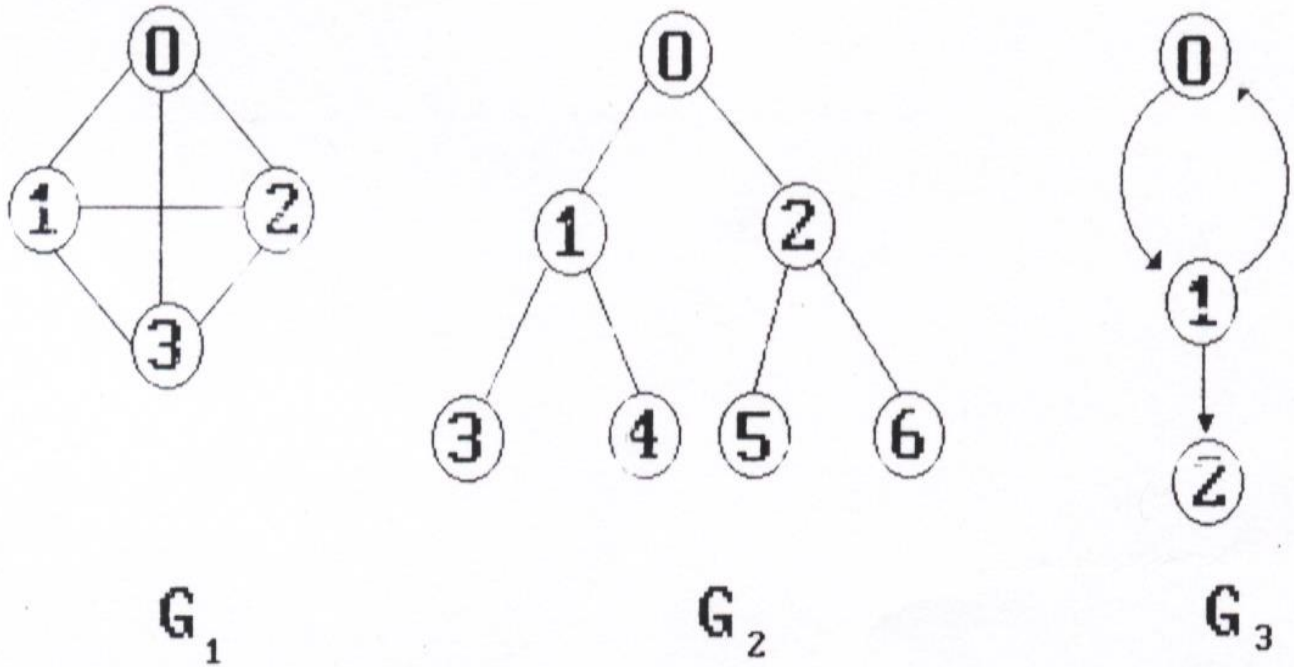


Figure 6.2: Three sample graphs

– Since we define the edges and vertices of a graph as sets, we impose the following restrictions on graphs:

(1) A graph may not have an edge from a vertex, *i*, back to itself. Such edges are known as *self loops*.

(2) A graph may not have multiple occurrences of the same edge. If we remove this restriction, we obtain a data referred to as a multigraph.

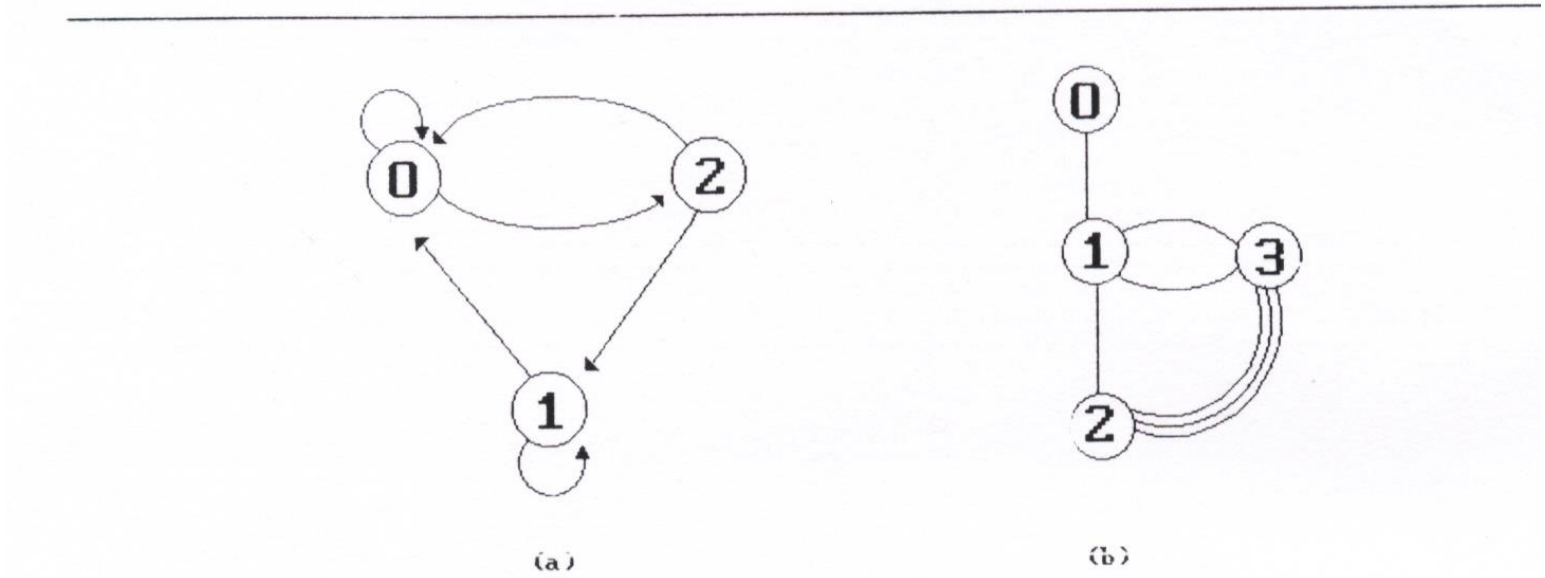
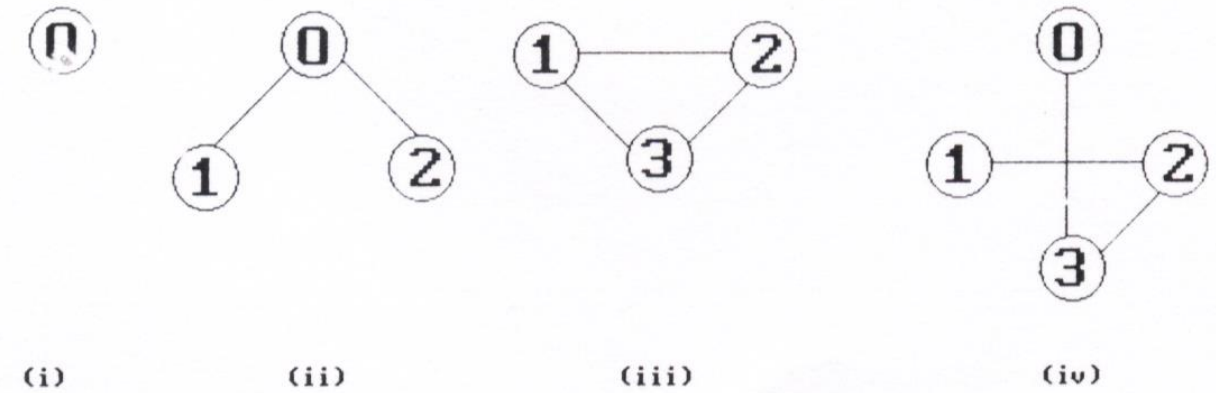
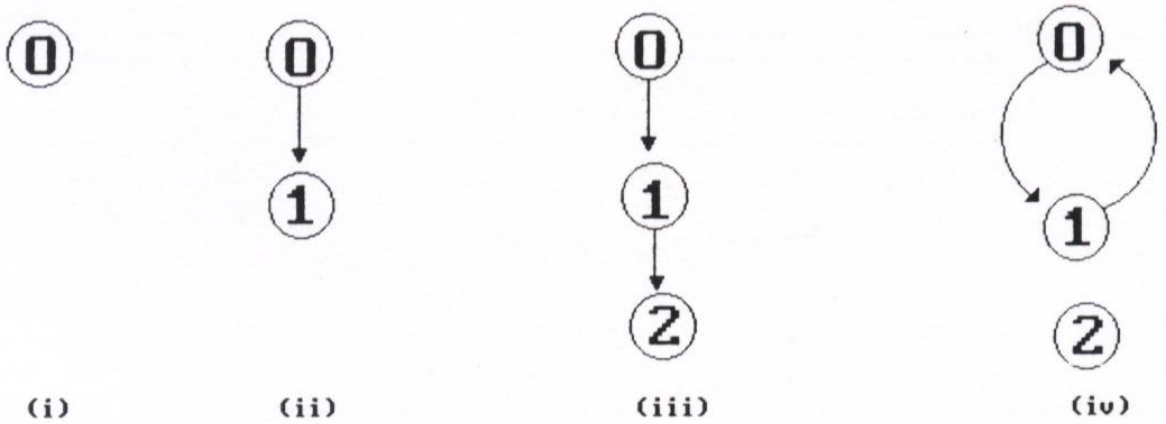


Figure 6.3: Examples of a graph with feedback loops and a multigraph

- A *complete graph* is a graph that has the maximum number of edges, ex: G_1 in Figure 6.2.
  - The number of distinct unordered pairs $<u,v>$ with $u \neq v$ in a graph with n vertices is $n(n-1)/2$.
  - The number of distinct ordered pairs $<u,v>$ with $u \neq v$ in a graph with n vertices is $n(n-1)$.
- If (u, v) is an edge in $E(G)$, then we shall say the vertices u and v are *adjacent* and that the edge (u, v) is *incident* on vertices u and v.
- A subgraph of G is a graph G' such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$.



(a) Some of the subgraphs of G_1



(b) Some of the subgraphs of G_3

Figure 6.4: Subgraphs of G_1 and G_3

– A *path* from vertex v_p to vertex v_q in a graph, G, is a sequence of vertices, v_p, v_{i1}, v_{i2}, … , v_{in}, v_q such that (v_p, v_{i1}), (v_{i1}, v_{i2}), … , (v_{in}, v_q) are edges in an undirected graph. If G' is a directed graph, then the path consists of $<v_p, v_{i1}>$, $< v_{i1}, v_{i2}>$, … , $< v_{in}, v_q>$.
  - A path such as (1, 2), (2, 4), (4, 3) is also written as 1, 2, 4, 3.
– The *length* of a path is the number of edges on it.
– A *simple path* is a path in which all vertices, except possibly the first and the last, are distinct.
  - Ex: In graph G_1 of Figure 6.2, the path 0, 1, 3, 2 is a simple path. In graph G_3, 0, 1, 2 is a *simple directed path*.

– A *cycle* is a simple path in which the first and the last vertices are the same.
  - Ex: 0, 1, 2, 0 is a cycle in G_1, and 0, 1, 0 is a cycle in G_3.
– In an undirected graph G, two vertices, u and v are *connected* if there is a path in G from u to v.
– An undirected graph is connected if, for every pair of distinct vertices u, v, there is a path from u to v in G.
  - Ex: graph G_1 and G_2 are connected, while graph G_4 in Figure 6.5 is not.

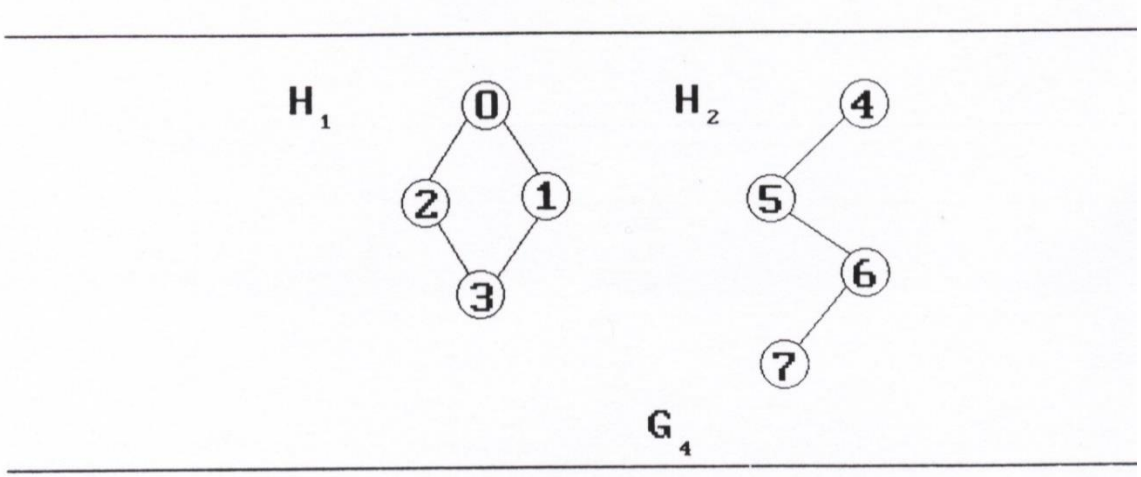


Figure 6.5: A graph with two connected components

- A *connected component*, or simply *component*, of an undirected graph is a maximum connected subgraph.
  - Ex: In Figure 6.5, G_4 has two components, H_1 and H_2.
- A tree is a *connected acyclic (i.e, has no cycle) graph*.
- A directed graph is *strongly connected* if, for every pair of vertices v_i, v_j in $V(G)$, there is a directed path from v_i to v_j and also from v_j to v_i.
- A *strongly connected component* is a maximal subgraph that is strongly connected.
- The *degree* of a vertex is the number of edges incident to that vertex.
  - The *in-degree* of a vertex v is the number of edges that have v as the head.
  - The *out-degree* of a vertex v is the number of edges that have v as the tail.

– If d_i is the degree of a vertex i in a graph G with n vertices and e edges, then the number of edges is:

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

– We shall refer to a directed graph as a *digraph*. When we use the term *graph*, we assume that it is an undirected graph.

– ADT of a graph

structure *Graph* is

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

functions:

for all *graph* ∈ *Graph*, v, v_1, and v_2 ∈ *Vertices*

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

Structure 6.1: Abstract data type *Graph*

- Graph representations
  - Adjacency matrix:
    - $adj\_mat[i][j] = 1$ *iff* $(v_i, v_i) \in E(G)$

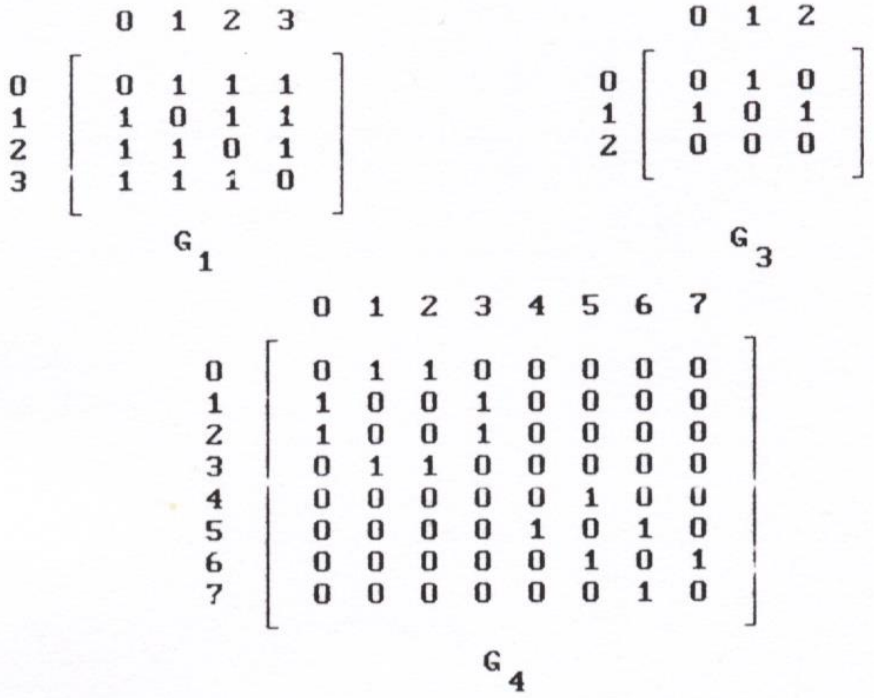


Figure 6.7: Adjacency matrices for G_1, G_3, and G_4

- For an undirected graph, the degree of any vertex, i, is its row sum:

$$\sum_{j=0}^{n-1} adj\_mat[i][j]$$

- For a directed graph, the row sum is the out-degree, while the column sum is the in-degree.
- The complexity of checking edge number or examining if G is connect: $O(n^2)$.

– Adjacency lists

- There is one list for each vertex in *G*. The nodes in list *i* represent the vertices that are adjacent from vertex *i*.

```
#define MAX_VERTICES 50 /*maximum number of vertices*/
typedef struct node *node_pointer;
typedef struct node {
        int vertex;
        struct node *link;
        };
node_pointer graph[MAX_VERTICES];
int n = 0; /* vertices currently in use */
```

- For an undirected graph with *n* vertices and *e* edges, this representation requires *n* head nodes and 2*e* list nodes.

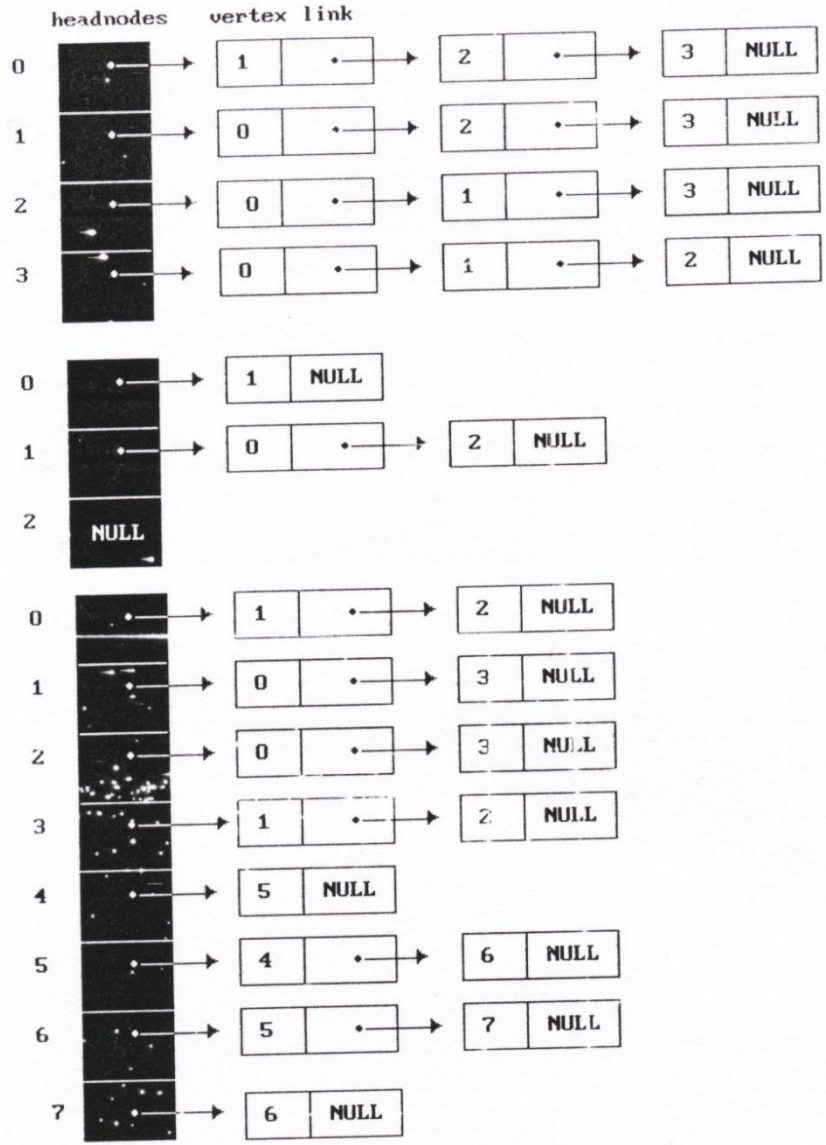


Figure 6.8: Adjacency lists for G_1, G_3, and G_4

- Often, we can sequentially pack the nodes on the adjacency lists to eliminate the use of pointers.
  - *node*[1] ~ *node*[*n*+2*e*+1] may be used.
  - *node*[*i*] gives the starting point of the list for vertex *i*, $0 \leqq i < n$ and *node*[n] is set to *n*+2*e*+1.
  - The vertices adjacent from vertex i are stored in *node*[*i*], … , *node*[*i*+1]-1, $0 \leqq i < n$.

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

Figure 6.9 Sequential representation of graph G4.

- The number of edge in undirected graph G may be determined in O(n+e) time.

– Adjacency multilists

- In some situations it is necessary to be able to determine the second entry for a particular edge and mark that edge as have been examined. It can be accomplished easily if the adjacency list are actually maintained as multilists.

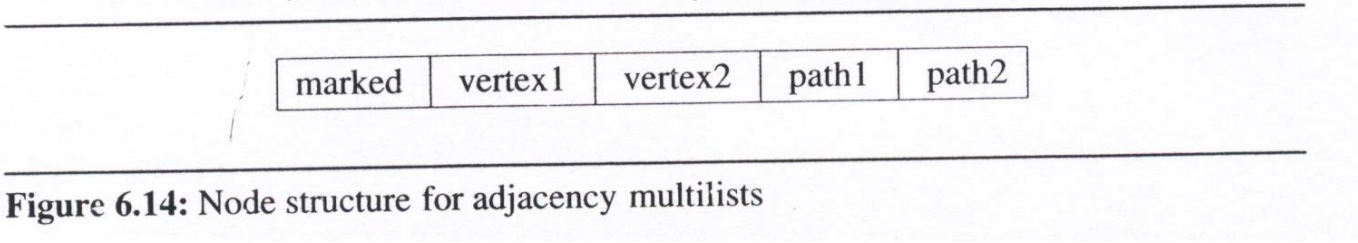
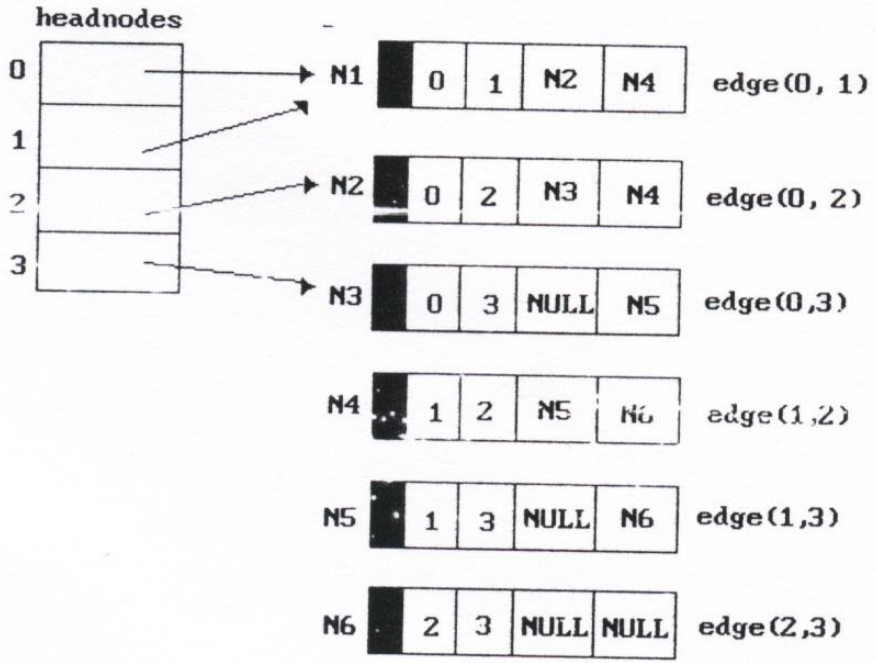


Figure 6.14: Node structure for adjacency multilists

```
typedef struct edge *edge_pointer;
typedef struct edge {
        short int marked;
        int vertex1;
        int vertex2;
        edge_pointer path1;
        edge_pointer path2;
        };
edge_pointer graph[MAX_VERTICES];
```



The lists are: vertex 0: M1 → M2 → M3
vertex 1: M1 → M4 → M5
vertex 2: M2 → M4 → M6
vertex 3: M3 → M5 → M6

Figure 6.15: Adjacency multilists for G_1

– Weighted edges

- The edges of a graph have weights assigned to them.

  Ex: these weights may represent the distance from one vertex to another or cost of going from one vertex to an adjacent vertex.

- adjacency matrix: *adj_mat*[*i*][*j*] would keep the weights.
- adjacency lists: add a *weight* field to the node structure.
- A graph with weighted edges is called a *network*.

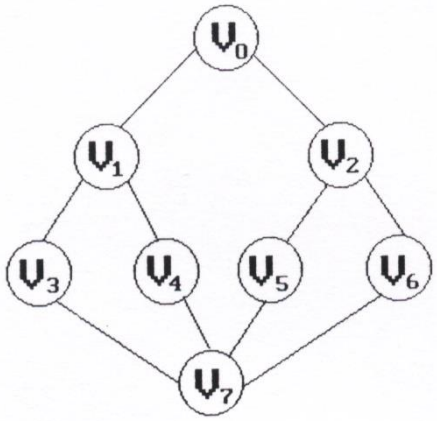
6.2 Elementary Graph Operations

- Depth-first search

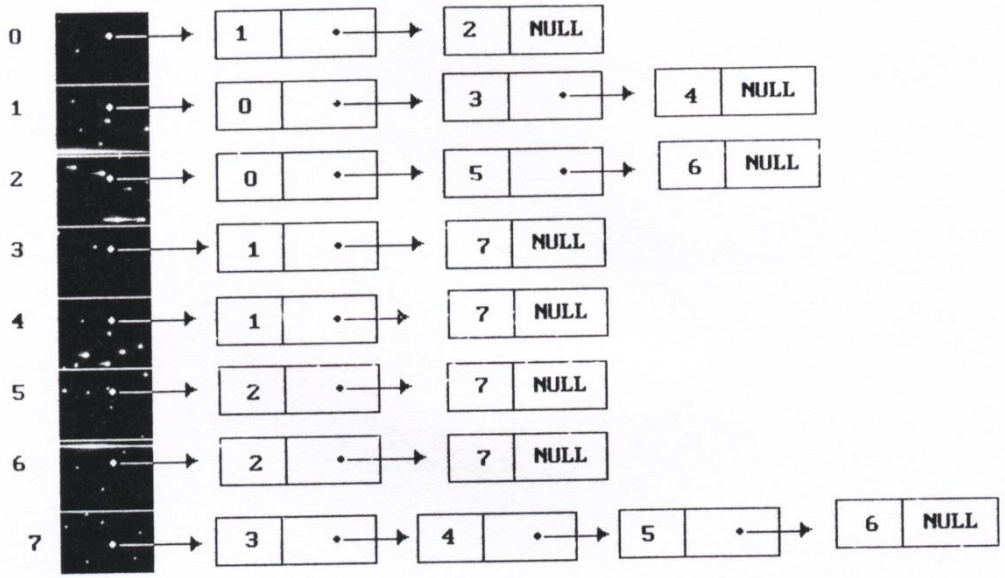
```
void dfs(int v)
{
/* depth first search of a graph beginning with vertex v.*/
   node_pointer w;
   visited[v] = TRUE;
   printf("%5d",v);
   for (w = graph[v]; w; w = w->link)
      if (!visited[w->vertex])
         dfs(w->vertex);
}
```

Program 6.1: Depth first search

Example 6.1: We wish to carry out a depth first search of graph G of Figure 6.19(a). Figure 6.19(b) shows the adjacency lists for this graph. If we initiate this search from vertex v_0, then the vertices of G are visited in the following order: $v_0, v_1, v_3, v_7, v_4, v_5, v_2, v_6$.



(a)



(b)

Figure 6.19: Graph *G* and its adjacency lists

- Analysis of dfs
  - When G is represented by it adjacency matrix, the complexity is $O(n^2)$.
  - When G is represented by it adjacency lists, the complexity is $O(e)$.

- Breadth-first search
  - It needs a queue to implement breadth-first search.

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

Program 6.2: Breadth first search of a graph

- Analysis of bfs
  - When G is represented by it adjacency matrix, the complexity is $O(n^2)$.
  - When G is represented by it adjacency lists, the complexity is $O(e)$.

- Connected components
  - If G is an undirected graph, then one can determine whether or not it is connected by simply making a call to either *dfs* or *bfs* and then determining if there is any unvisited vertex.

```
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

- Analysis of connected:
  - If G is represented by its adjacency lists, then total time to generate all the connected components is $O(n+e)$.
  - If G is represented by its adjacency matrix, then total time to generate all the connected components is $O(n^2)$.

- Spanning trees
  - **Definition:** A tree T is said to be a *spanning tree* of a connected graph G if T is a subgraph of G and T contains all vertices of G.

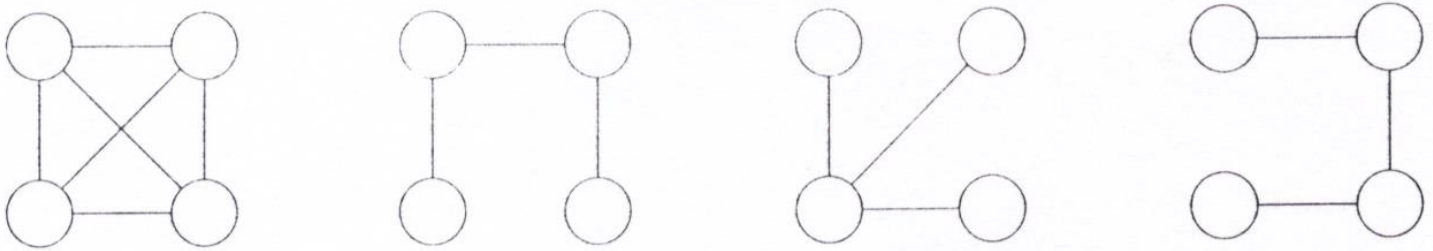


Figure 6.20: A complete graph and three of its spanning trees

- Classification:
  - *depth-first spanning tree:* the spanning tree resulting from a depth-first search.
  - *breadth-first spanning tree:* the spanning tree resulting from a breadth-first search.

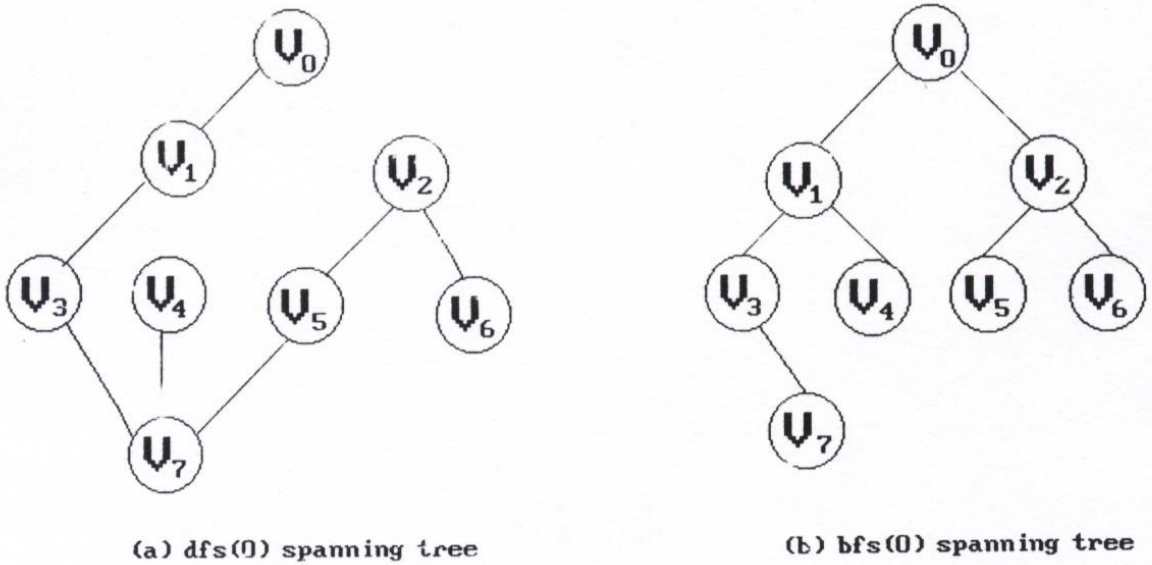
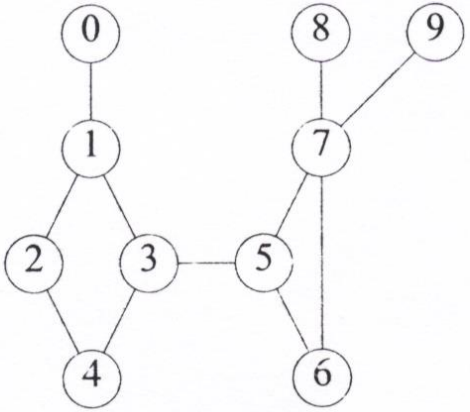


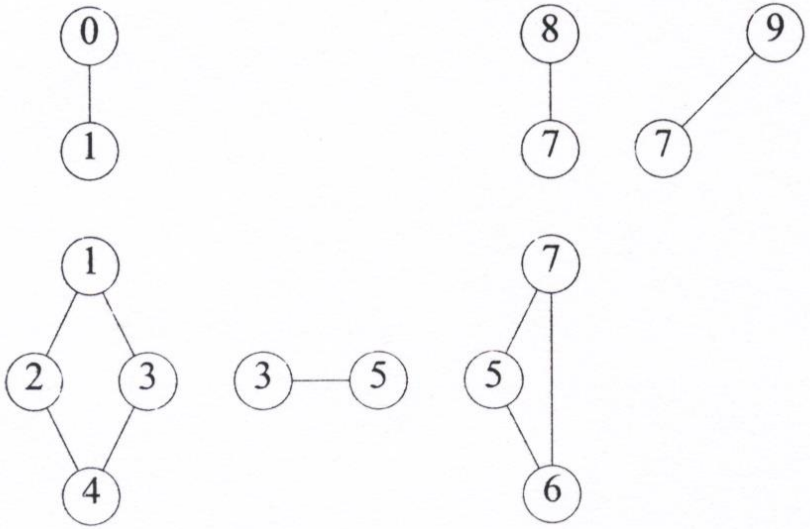
Figure 6.21: *dfs* and *bfs* spanning trees for graph of Figure 6.19

- Properties:
  - If a nontree edge (v, w) is introduced into any spanning tree T, then a cycle is formed.
  - A spanning tree is a minimal subgraph, G', of G such that $V(G') = V(G)$ and G' is connected.
    - » We define a minimal subgraph as one with the fewest number of edge
    - » A spanning tree has n-1 edges.

- Biconnected components and articulation points
  - assumption: G is an undirected, connected graph.
  - **Definition:** A vertex *v* of *G* is an *articulation point* iff the deletion of *v*, together with the deletion of all edges incident to *v*, leaves behind a graph that has at least two connected components.
  - **Definition:** A *biconnected graph* is a connected graph that has no articulation points.
  - **Definition:** A *biconnected component* of a connected graph *G* is a maximal biconnected subgraph *H* of *G*. By maximal, we mean that *G* contains no other subgraph that is both biconnected and properly contains *H*.



(a) Connected graph



(b) Biconnected components

Figure 6.22: A connected graph and its biconnected components

6.3 Minimum-cost Spanning Tree

- Introduction
  - The *cost* of a spanning tree of a weighted, undirected graph is 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.
  - We'll introduce three algorithms to obtain a minimum-cost spanning tree of a connected, undirected graph.
  - All three use a design strategy called the greedy method

- Greedy method: we construct an optimal solution in stages. At each stage, we make the best decision (using some criterion) possible at the time.

– To construct minimum-cost spanning trees, we use least-cost criterion. Our solution must satisfy the following constrains.

  (1) We must use only edges within the graph.

  (2) We must use exactly *n* - 1 edges.

  (3) We may not use edges that produce a cycle.

- Kruskal's algorithm:
  - Ex 6.3

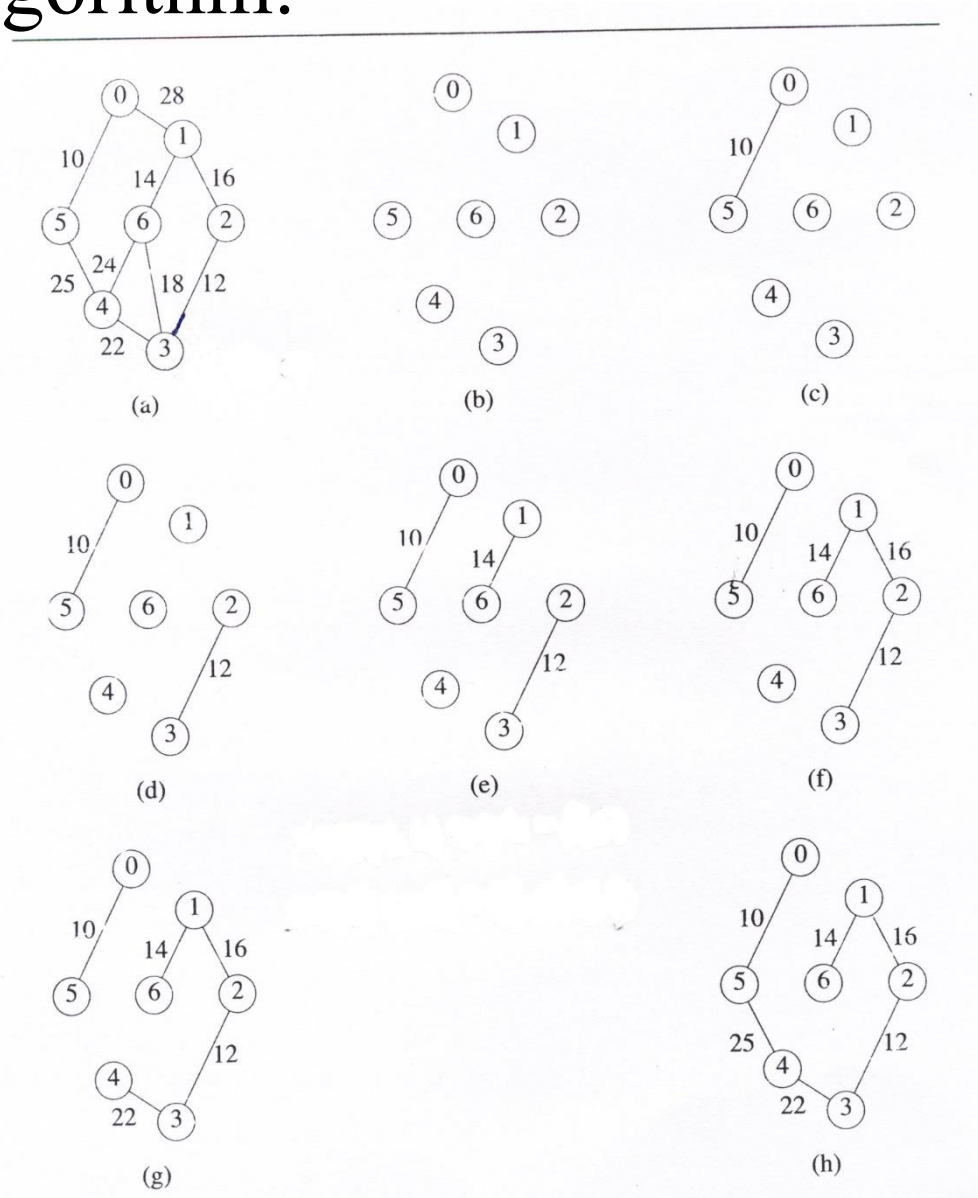


Figure 6.25: Stages in Kruskal's algorithm

Edge	Weight	Result	Figure
----	---	initial	Figure 6.25(b)
(0,5)	10	added to tree	Figure 6.25(c)
(2,3)	12	added	Figure 6.25(d)
(1,6)	14	added	Figure 6.25(e)
(1,2)	16	added	Figure 6.25(f)
(3,6)	18	discarded	
(3,4)	22	added	Figure 6.25(g)
(4,6)	24	discarded	
(4,5)	25	added	Figure 6.25(h)
(0,1)	28	not considered	

Figure 6.26: Summary of Kruskal's algorithm applied to Figure 6.25(a)

```
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

- Time complexity: O(e log e)
- **Theorem 6.1:** Let G be an undirected connected graph. Kruskal's algorithm generates a minimum cost spanning tree.

- **Prim's algorithm**

Idea: 任意選一 vertex 為起始點 (不必選最低 cost 的 edge之兩個端點)，之後每次加入連到這個vertex 的 edges 裡，有最低 cost 的 edge，但不構成 cycle 的；直到 end.

```
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

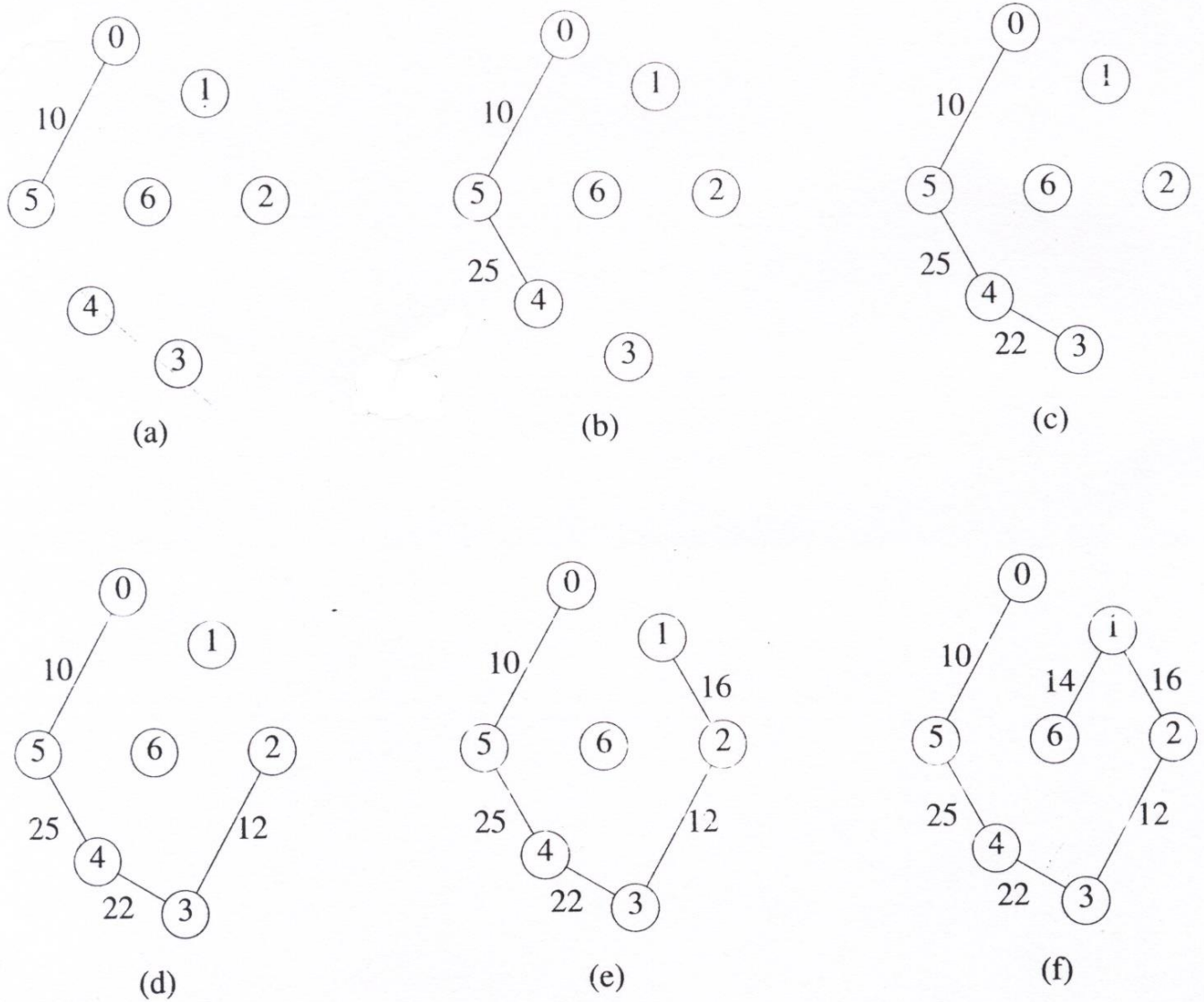


Figure 6.27: Stages in Prim's algorithm

- Sollin's algorithm

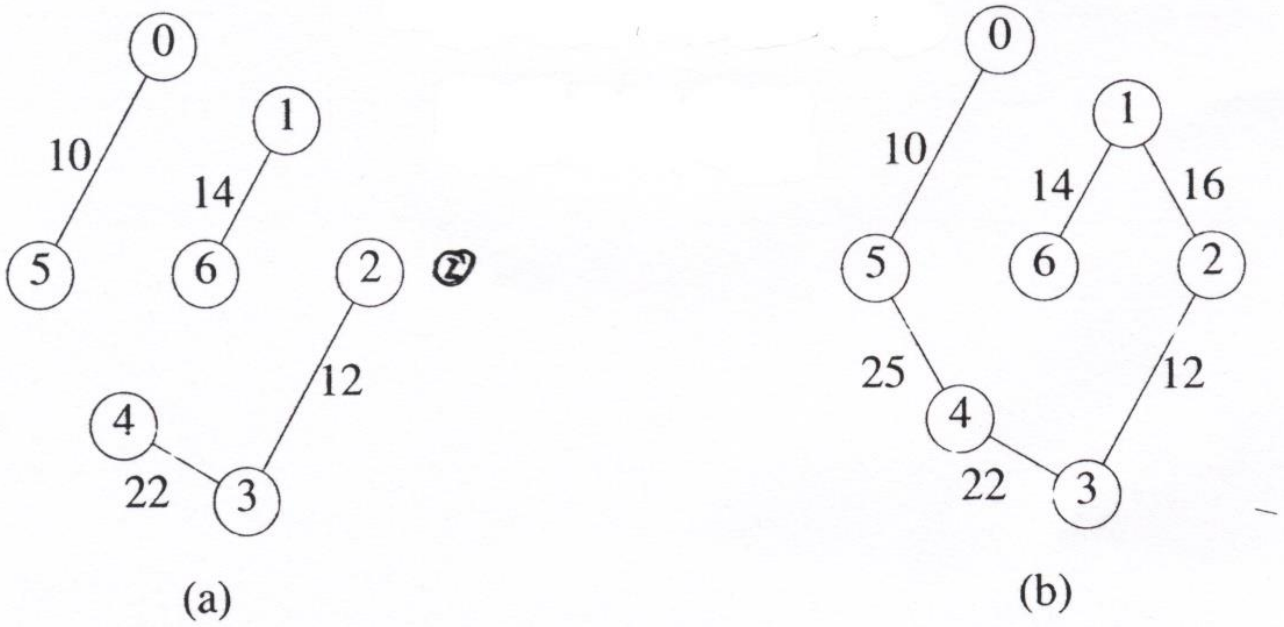


Figure 6.28: Stages in Sollin's algorithm

6.4 Shortest Path And Transitive Closure

- In this section, we shall study the path problems such as

  (1) Is there a path from city A to city B?

  (2) If there is more than one path from city A to city B, which path is the shortest?

- **Definition:**
  - *Source:* the starting vertex of the path.
  - *Destination:* the last vertex of the path.

- Single source/all destinations: nonnegative edge cost
  - **Problem:** given a directed graph $G = (V, E)$, a length function $length(i, j)$, $length(i, j) \geq 0$, for the edges of G, and a source vertex v.
  - **Need to solve:** determine a shortest path from v to each of the remaining vertices of G.
  - $dist[w]$: the length of shortest path starting from v, going through only the vertices that are in S, and ending at w.

Dijkastra's Algorithm

- Find the min cost of the path from a given source node to every other node.
- Given: the cost $e(v_i,v_j)$ of all edges;

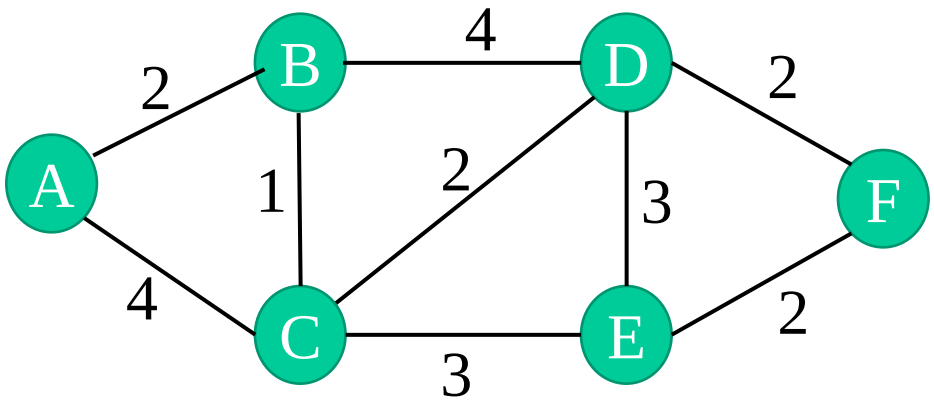
  v_0 is the source node;

  $e(v_i,v_j) = \infty$, if v_i and v_j are not adjacent

Dijkstra's Algorithm

1. $S \leftarrow \{v_0\}$;
2. $D[v_0] \leftarrow 0$; /* D[v] 是目前 v 到 v_0 之 min cost */
3. for each v in V-$\{v_0\}$ do $D[v] \leftarrow e(v_0,v)$;
4. while S≠V do
5. choose a vertex w in V-S such that D[w] is a minimum;
6. add w to S;
7. for each v in V-S do
8. $D[v] \leftarrow$ Min(D[v], D[w]+e(w,v));

Find the shortest path from A to every other nodes



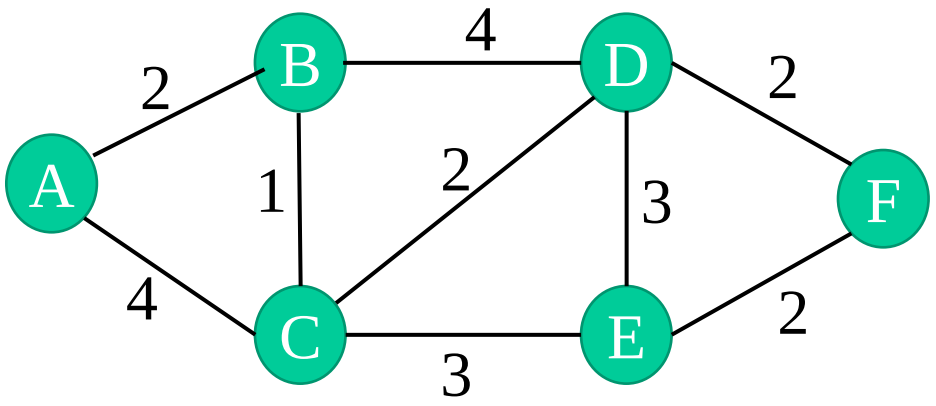
我們需要設定二個 set :
S and V
並要建立一個 Table

(1) S={A}, V={B, C, D, E, F}

找出 S 到 V 只走一步可達之處，取最小者。
若一步無法抵達之處，則為 ∞

	B	C	D	E	F
A	2^A	4^A	∞	∞	∞

Find the shortest path from A to every other nodes



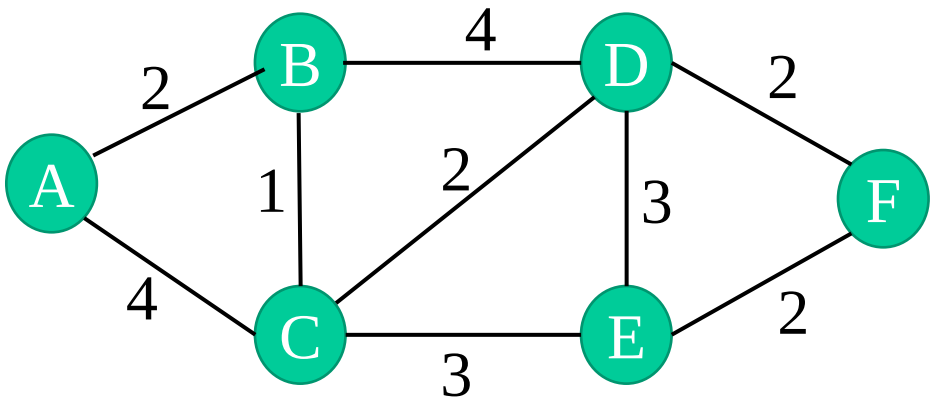
我們需要設定二個 set :
S and V
並要建立一個 Table

(2) 前一步找到 B and C. 取最小者（i.e., B)，從 V 移到 S.
故 S={A, B}, V={C, D, E, F}

找出 S 到 V 只走一步可達之處，取最小者。
若一步無法抵達之處，則為 ∞

	B	C	D	E	F
A	2^A	4^A	∞	∞	∞
		3^B	6^B		

Find the shortest path from A to every other nodes



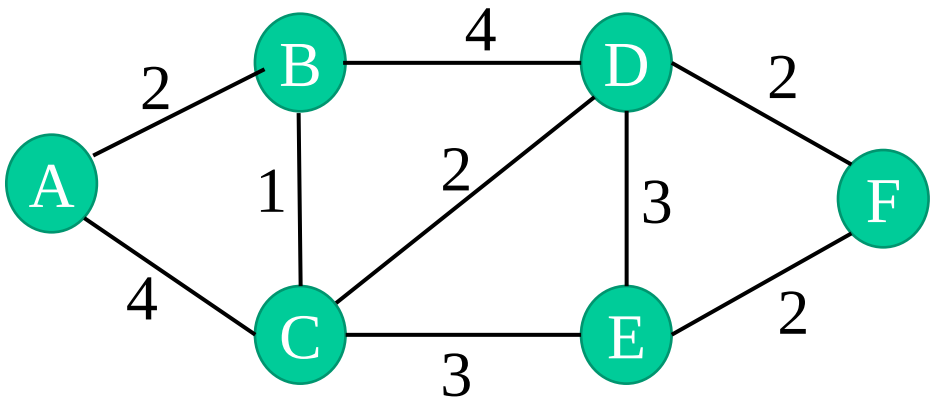
我們需要設定二個 set : S and V
並要建立一個 Table

(3) 前一步找到 C and D. 取最小者 （i.e., C)，從 V 移到 S.
故 S={A, B, C}, V={D, E, F}

找出 S 到 V 只走一步可達之處，取最小者。
若一步無法抵達之處，則為 ∞

	B	C	D	E	F
A	2^A	3^B	6^B	∞	∞
			5^C	6^C	

Find the shortest path from A to every other nodes



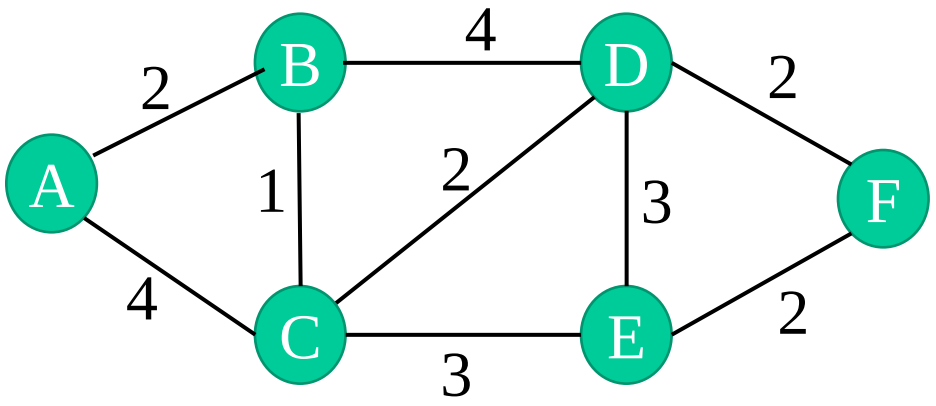
我們需要設定二個 set : S and V
並要建立一個 Table

(4) 前一步找到 D and E. 取最小者（i.e., D)，從 V 移到 S.
故 S={A, B, C, D}, V={E, F}

找出 S 到 V 只走一步可達之處，取最小者。
若一步無法抵達之處，則為 ∞

	B	C	D	E	F
A	2^{A}	3^{B}	5^{C}	6^{C}	∞
				↓ unchange	↓ 7^{D}

Find the shortest path from A to every other nodes



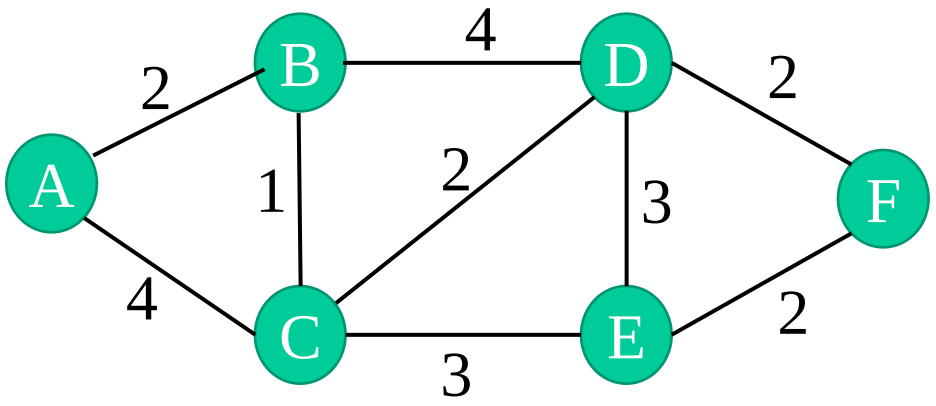
我們需要設定二個 set :
S and V
並要建立一個 Table

(5) 前一步找到 E and F. 取最小者 （i.e., E)，從 V 移到 S.
故 S={A, B, C, D, E}, V={F}

找出 S 到 V 只走一步可達之處，取最小者。
若一步無法抵達之處，則為 ∞

	B	C	D	E	F
A	2^A	3^B	5^C	6^C	7^D
				↓	↓
				unchange	unchange

Find the shortest path from A to every other nodes



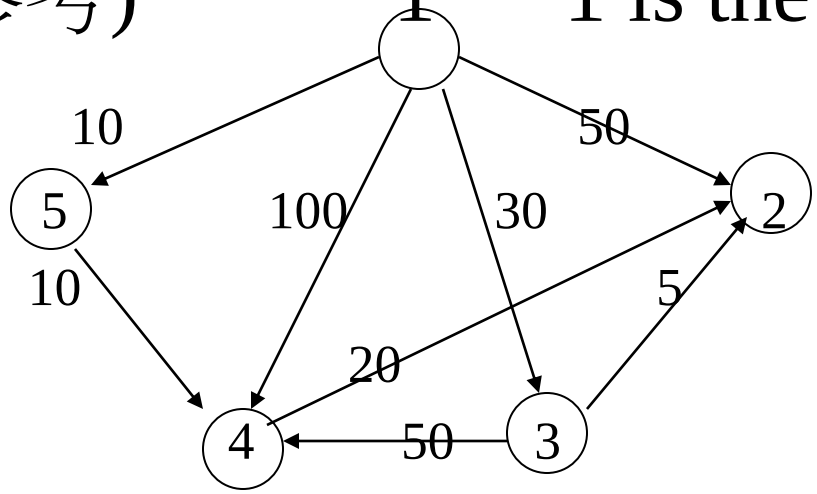
我們需要設定二個 set :
S and V
並要建立一個 Table

(6) 前一步找到 F. 取最小者（i.e., F)，從 V 移到 S.
故 S={A, B, C, D, E, F}, V={ }
V 空了，故 algorithm stops

	B	C	D	E	F
A	2^{A}	3^{B}	5^{C}	6^{C}	7^{D}

Finally, 從表中可知從 A 到任何其他點之 shortest path.

- Example(參考)

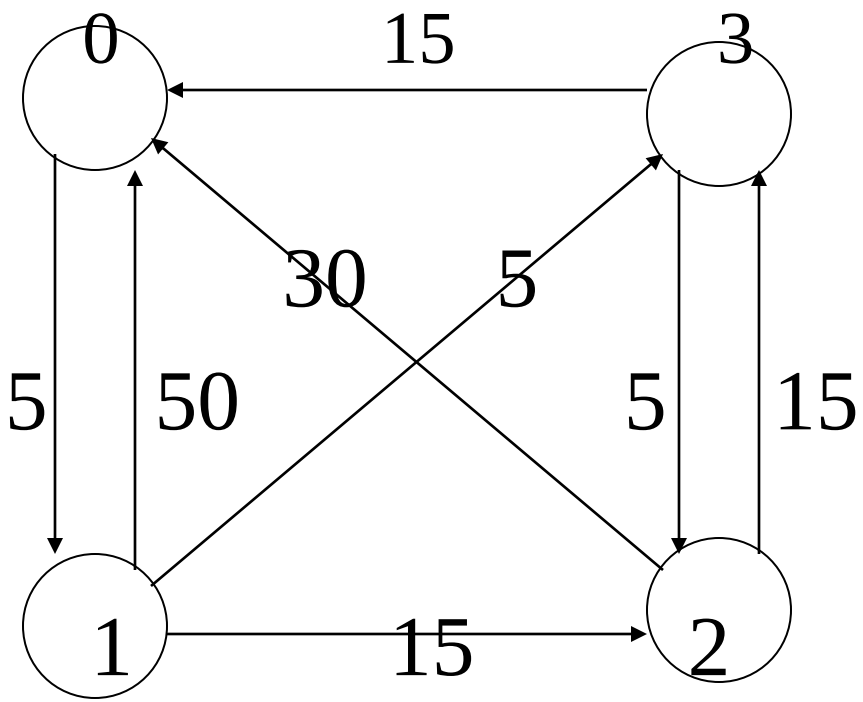


1 is the source node

step	w	S	V-S	D
initial	1	{1}	{2,3,4,5}	[50,30,100,10]
1	5	{1,5}	{2,3,4}	[50,30,20,10]
2	4	{1,5,4}	{2,3}	[40,30,20,10]
3	3	{1,5,4,3}	{2}	[35,30,20,10]
4	2	{1,5,4,3,2}	∅	[35,30,20,10]

- All pairs shortest paths
  - we could solve this problem using *shortestpath* with each of the vertices in $V(G)$ as the source. ($O(n^3)$)
  - Another algorithm to solve this problem
    - use dynamic programming method.
    - *cost*[*i*][*j*]: cost adjacency matrix of the graph *G*.
      - If $i = j$, *cost*[*i*][*j*] = 0.
      - If $<i, j>$ is not in *G*, *cost*[*i*][*j*] is set to some sufficiently large number.
    - Let $A^k[i][j]$ be the cost of shortest path from *i* to *j*, using only those intermediate vertices with an index $\leq k$.
      - The shortest path from *i* to *j* is $A^{n-1}[i][j]$ as no vertex in *G* has an index greater than $n - 1$.
      - $A^{-1}[i][j]$ = *cost*[*i*][*j*] since the only *i* to *j* paths allowed have no intermediate vertices on them.

- Algorithm concept
  - The basic idea in the all pairs algorithm is begin with the matrix A^{-1} and successively generated the matrices A^{-1}, A^0, A^2, …, A^n
  - How do we get A^k if we have A^{k-1}?
    - The shortest path from i to j going through no vertex with index greater than k does not go through the vertex with index k and so its cost is $A^{k-1}[i][j]$.
    - The shortest such path does go through vertex k. Such a path consists of a path from i to k followed by one from k to j. Neither of these goes through a vertex with index greater than k-1. Hence, their costs are $A^{k-1}[i][k]$ and $A^{k-1}[k][j]$.


0
15
3
30
5
5
50
5
15
1
15
2

$$A^{-1} = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix} \qquad A^{0} = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$A^{1} = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix} \qquad A^{2} = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix} \qquad A^{3} = \begin{pmatrix} 0 & 5 & 15 & 10 \\ 20 & 0 & 10 & 15 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

- Transitive closure

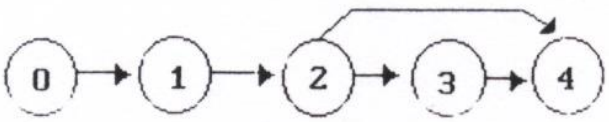
這是一個 All-pair-shortest path 的應用。

Transitive closure is to check whether there is a path from *i* to *j*, for all *i* and *j* in the graph.

  - **Definition:** The *transitive closure matrix*, denoted A^+, of a directed graph, *G*, is a matrix such that $A^+[i][j]$ = 1 if there is a path of length > 0 from *i* to *j*; otherwise, $A^+[i][j] = 0$. /* So, $A^+[i][i] = 0$ */

  - **Definition:** The *reflexive transitive closure matrix*, denoted A^*, of a directed graph, *G*, is a matrix such that $A^*[i][j] = 1$ if there is a path of length ≥ 0 from *i* to *j*; otherwise, $A^*[i][j] = 0$. /* So, $A^*[i][i] = 1$ */

更正：方向應相反，由4 到 2
（第二版有更正）



(a) Digraph G

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 for G

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	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^*

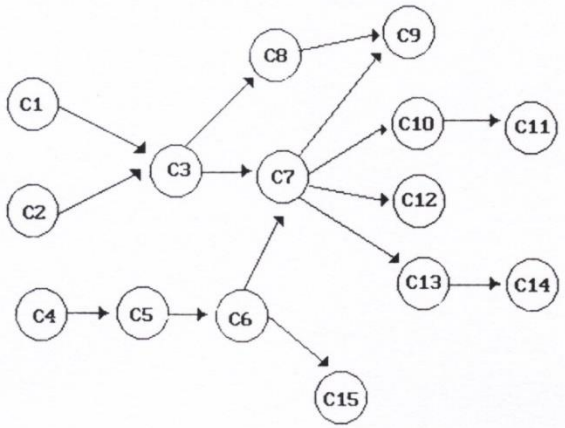
Figure 6.35: Graph G and its adjacency matrix A, A^+, A^*

6.5 Activity Networks

- Activity on vertex (AOV) networks
  - All but the simplest of projects can be divided into several subprojects called *activities*. The successful completion of these activities results in the completion of the entire project.
  - Example: A student working toward a degree in computer science must complete several courses successful.

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 edges as prerequisites

Figure 6.38: An AOV network

- **Definition:** An *activity on vertex*, or *AOV*, network, is a directed graph *G* in which the vertices represent tasks or activities and the edges represent precedence relations between tasks.
- **Definition:** Vertex *i* in an *AOV* network *G* is a *predecessor* of vertex *j iff* there is a directed path from vertex *i* to vertex *j*. Vertex *i* is an *immediate predecessor* of vertex *j iff* <*i*, *j*> is an edge in *G*. 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*.

– **Definition:** A relation • is transitive *iff* for all triples *i*, *j*, *k*, *i*•*j* and *j*•*k* ⇒ *i*•*k*. A relation • is irreflexive on a set *S* if *i*•*i* is false for all elements, *i*, in *S*. A *partial order* is a precedence relation that is both transitive and irreflexive.

– **Definition:** 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 order.

– Example 6.7

```
for (i = 0; i < n; i++) {
   if every vertex has a predecessor {
      fprintf(stderr,"Network has a cycle.\n");
      exit(1);
   }
   pick a vertex v that has no predecessors;
   output v;
   delete v and all edges leading out of v
   from the network;
}
```

Program 6.13: Topological sort

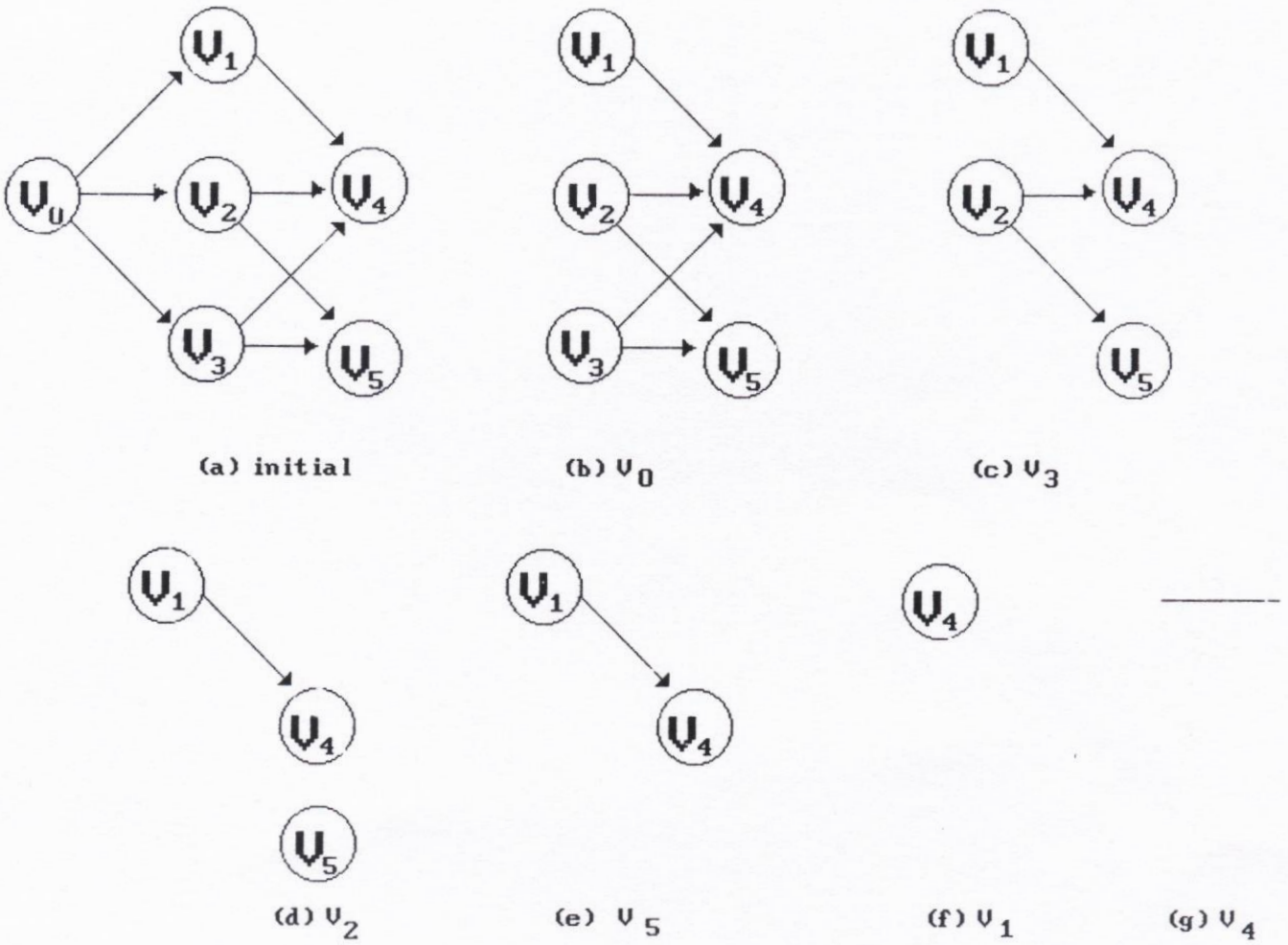


Figure 6.39: Simulation of Program 6.13 on an AOV network

– Topological sort

- data representation

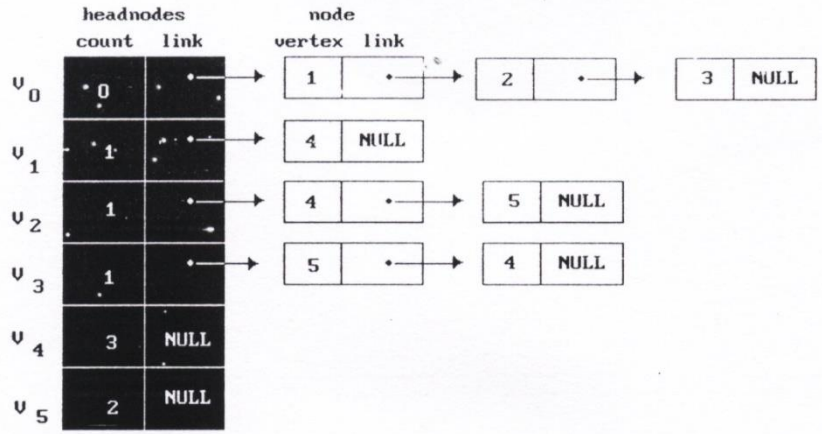


Figure 6.40: Adjacency list representation of Figure 6.39(a)

- The declarations used in topsort are:

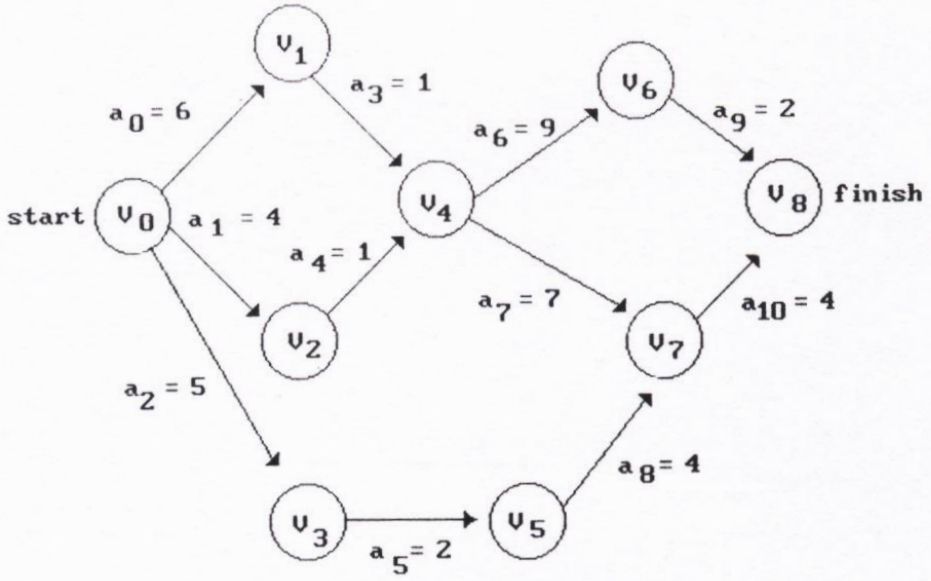
```
typedef struct node *node_pointer;
typedef struct node {
        int vertex;
        node_pointer link;
        };
typedef struct {
        int count;
        node_pointer link;
        } hdnodes;
hdnodes graph[MAX_VERTICES];
```

```
void topsort(hdnodes graph[], int n)
{
   int i,j,k,top;
   node_pointer ptr;
   /* create a stack of vertices with no predecessors */
   top = -1;
   for (i = 0; i < n; i++)
      if (!graph[i].count) {
         graph[i].count = top;
         top = i;
   }
   for (i = 0; i < n; i++)
      if (top == -1) {
         fprintf(stderr,"\nNetwork   has   a   cycle.   Sort
        terminated. \n");
         exit(1);
      }
      else {
         j = top;   /* unstack a vertex */
         top = graph[top].count;
         printf("v%d, ",j);
         for (ptr = graph[j].link; ptr; ptr = ptr->link) {
         /* decrease the count of the successor vertices
         of j */
            k = ptr->vertex;
            graph[k].count--;
            if (!graph[k].count) {
            /* add vertex k to the stack */
               graph[k].count = top;
               top = k;
            }
         }
      }
}
```

Program 6.14: Topological sort

- Analysis of *topsort*:
  - Time complexity: O(*n*+*e*)

- Activity on edge (AOE) networks
  - An activity on edge, or AOE, network is an activity network closely related to the AOV network. (ex: Figure 6.41)
  - AOE networks have proved very useful for evaluating the performance of many types of projects.
    - What is the least amount of time in which the project may be complete (assuming there are no cycle in the network)?
    - Which activities should be speeded to reduce project length?



(a) AOE network. Activity graph of a hypothetical project

event	interpretation
v_0	start of project
v_1	completion of activity a_0
v_4	completion of activities a_3 and a_4
v_7	completion of activities a_7 and a_8
v_8	completion of project

(b) Interpretation of some of the events in the activity graph of (a)

Figure 6.41: An AOE network

– A *critical path* is a path that has the longest length.
  - For example, the path v_0, v_1, v_4, v_7, v_8 is a critical path in the network of Figure 6.41(a).
– The *earliest time* an event, v_i, can occur is the length of the longest path from the start vertex v_0 to vertex v_i.
  - For example, the earliest time that event v_4 can occur is 7.
  - The earliest time an event can occur determines the earliest start time for all activities represented by edges leaving that vertex.
  - We denote *early*(i) as the earliest starting time of activity a_i. For example, *early*(6) = *early*(7) = 7.

– The *lastest time*, *late*(*i*) of activity a_i, is defined to be the latest time the activity may start without increasing the project duration.
  - For example, early(5) = 5 and late(5) = 8, early(7) = 7 and late(7) = 7.
– A *critical activity* is an activity for which *early*(*i*) = *late*(*i*).
  - The difference between *late*(*i*) and *early*(*i*) is a measure of how critical an activity is.
– The purpose of critical-path analysis is to identify critical activities so that resource may be concentrated on these activities in an attempt to reduce a project finish time.

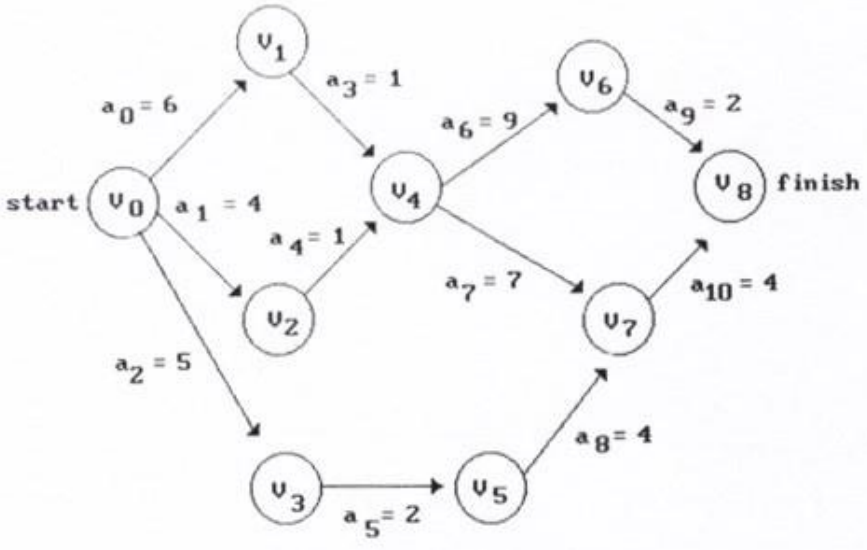
- Critical-path analysis can also be carried out with AOV network.

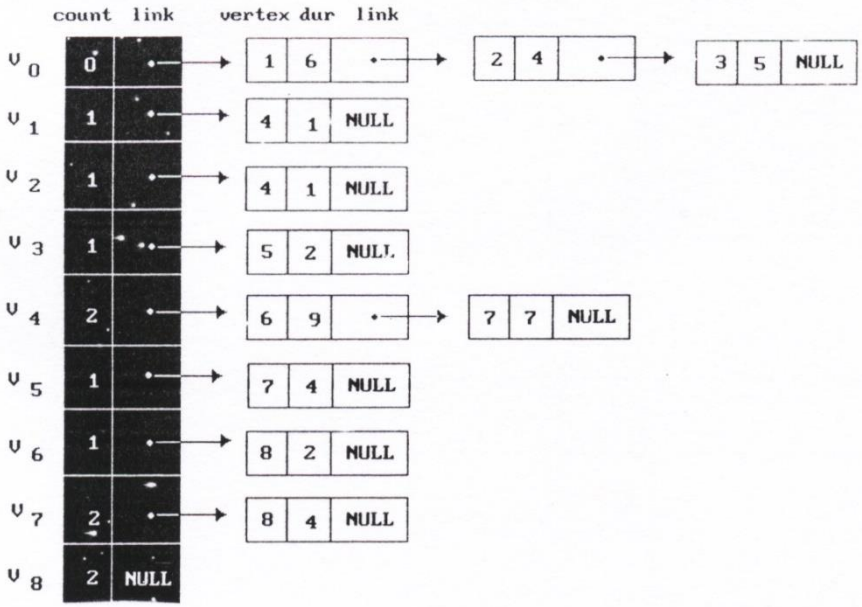
- Calculation of earliest times
  - *earliest*[*j*]: the earliest event occurrence time for event *j*.
  - *latest*[*j*]: the latest event occurrence time for event *j*.
  - If activity a_i is represented by edge <*k*, *l*>
    - *early*(*i*) = *earliest*[*k*]
    - *late*(*i*) = *latest*[*l*] - *duration of activity* a_i

– We compute the times *earliest*[*j*] and *latest*[*j*] in two stages: a forward stage and a backward stage.
– During the forwarding stage, we start with *earliest*[0] = 0 and compute the remaining start times using the formula:

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

Where $P(j)$ is the set of immediate predecessors of j.





(a) Adjacency lists for Figure 6.41(a)

Earliest	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	Stack
initial	0	0	0	0	0	0	0	0	0	[0]
output v_0	0	6	4	5	0	0	0	0	0	[3, 2, 1]
output v_3	0	6	4	5	0	7	0	0	0	[5, 2, 1]
output v_5	0	6	4	5	0	7	0	11	0	[2, 1]
output v_2	0	6	4	5	5	7	0	11	0	[1]
output v_1	0	6	4	5	7	7	0	11	0	[4]
output v_4	0	6	4	5	7	7	16	14	0	[7, 6]
output v_7	0	6	4	5	7	7	16	14	18	[6]
output v_6	0	6	4	5	7	7	16	14	18	[8]
output v_8										

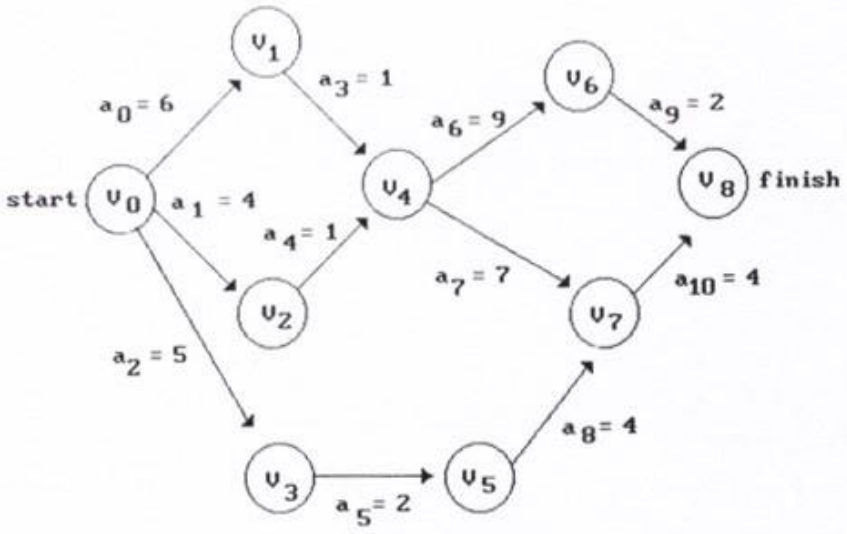
(b) Computation of *earliest*

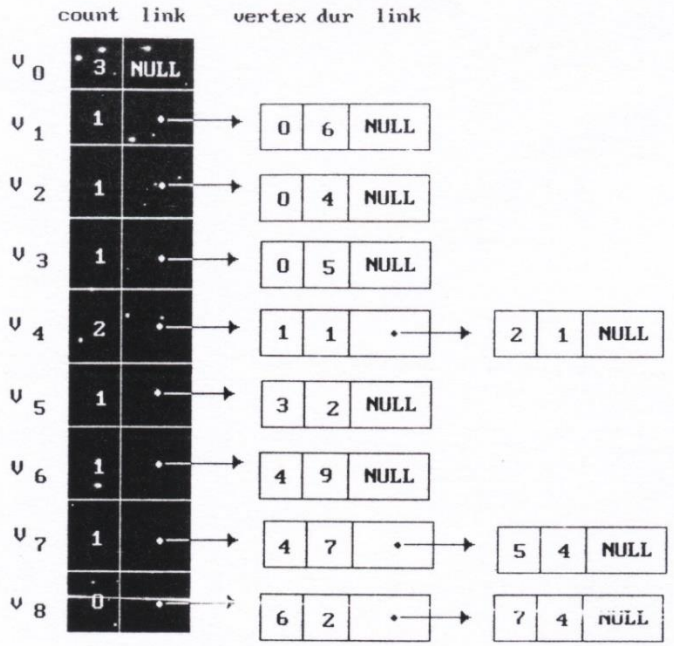
Figure 6.42: Computing *earliest* from topological sort

- Calculation of latest times
  - In the backward stage, we compute the values of *latest*[*i*] using a procedure analogous to that used in the forward stage.
  - We start with *latest*[*n*-1] = *earliest*[*n*-1] and use the equation:

$$latest[j] = \min_{i \in S(j)} \{ latest[i] - duration\ of \langle j, i \rangle \}$$

Where $S(j)$ is the set of vertice adjacent from vertex j.





(a) Inverted adjacency lists for AOE network of Figure 6.41(a)

Latest	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	Stack
initial	18	18	18	18	18	18	18	18	18	[8]
output v_8	18	18	18	18	18	18	16	14	18	[7, 6]
output v_7	18	18	18	18	7	10	16	14	18	[5, 6]
output v_5	18	18	18	8	7	10	16	14	18	[3, 6]
output v_3	3	18	18	8	7	10	16	14	18	[6]
output v_6	3	18	18	8	7	10	16	14	18	[4]
output v_4	3	6	6	8	7	10	16	14	18	[2, 1]
output v_2	2	6	6	8	7	10	16	14	18	[1]
output v_1	0	6	6	8	7	10	16	14	18	[0]

(b) Computation of *latest*

Figure 6.43: Computing *latest* for AOE network of Figure 6.41(a)

$$latest[8] = earliest[8] = 18$$
$$latest[6] = \min\{earliest[8] - 2\} = 16$$
$$latest[7] = \min\{earliest[8] - 4\} = 14$$
$$latest[4] = \min\{earliest[6] - 9;\ earliest[7] - 7\} = 7$$
$$latest[1] = \min\{earliest[4] - 1\} = 6$$
$$latest[2] = \min\{earliest[4] - 1\} = 6$$
$$latest[5] = \min\{earliest[7] - 4\} = 10$$
$$latest[3] = \min\{earliest[5] - 2\} = 8$$
$$latest[0] = \min\{earliest[1] - 6;\ earliest[2] - 4;\ earliest[3] - 5\} = 0$$

(c) Computation of *latest* from Equation (6.4) using a reverse topological order

Figure 6.43 (continued): Computing *latest* for AOE network of Figure 6.41(a)

– earliest and latest values $\Rightarrow$ *early*(i) and *late*(i) $\Rightarrow$ critical

Activity	Early	Late	Late – Early	Critical
a_0	0	0	0	yes
a_1	0	2	2	no
a_2	0	3	3	no
a_3	6	6	0	yes
a_4	4	6	2	no
a_5	5	8	3	no
a_6	7	7	0	yes
a_7	7	7	0	yes
a_8	7	10	3	no
a_9	16	16	0	yes
a_{10}	14	14	0	yes

Figure 6.44: Early, late, and critical values