제6장 Graphs

경북대학교 임경식 교수

6.1 Definition and representations

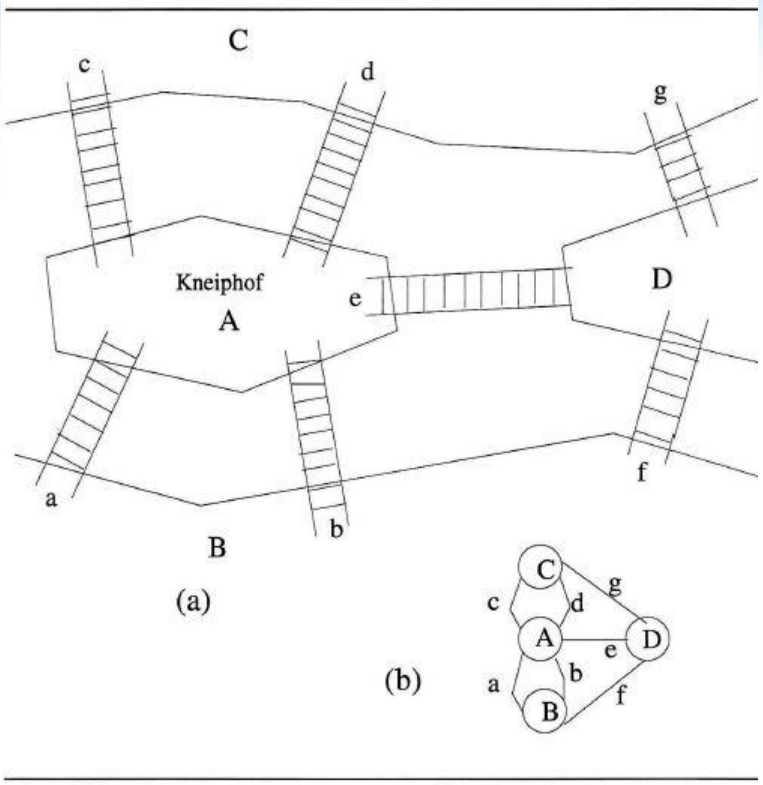


Figure 6.1: (a) Section of the river Pregel in Königsberg; (b) Euler's graph

The Konigsberg bridge problem is to determine whether, starting at one land area, it is possible to walk across all the bridges exactly once in returning to the starting land area.

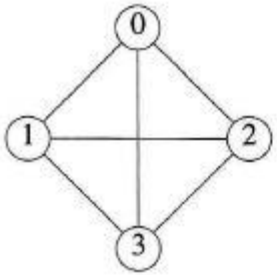
Defining the *degree* of a vertex to be the number of edges incident to it, Euler showed that there is a walk starting at any vertex, going through each edge exactly once and terminating at the start vertex iff the degree of each vertex is even.

6.1 Definition and representations

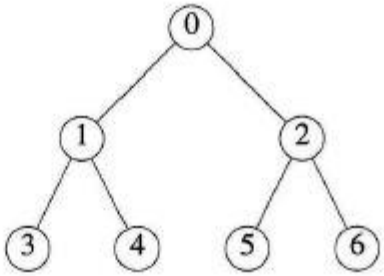
G = (V,E),
where V(G) is a finite, nonempty set of *vertices and*
E(G) is a set of pairs of vertices; these pairs are called *edges.*

In an *undirected graph*, the pairs *(u, v)* and *(v, u)* represent the same edge.

In a *directed graph*, each edge is represented by a directed pair *<u, v >*; *u* is the *tail* and *v* the *head* of the edge. Therefore, *<v, u >* and *<u, v >* represent two different edges.



(a) G_1



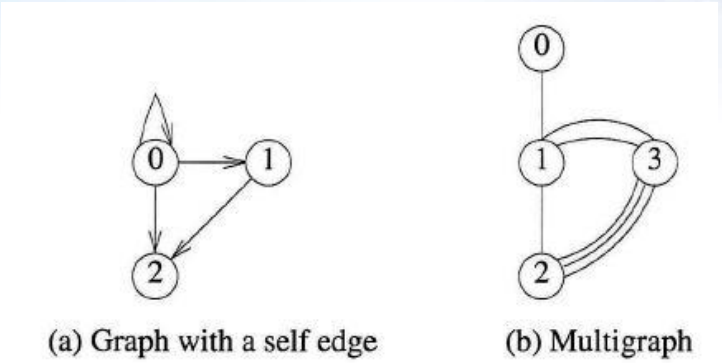
(b) G_2

(c) G_3

$V(G_1) = \{0,1,2,3\}$; $E(G_1) = \{(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)\}$
$V(G_{2)} = \{0,1,2,3,4,5,6\}$; $E(G_2) = \{(0,1),(0,2),(1,3),(1,4),(2,5),(2,6)\}$
$V(G_3) = \{0,1,2\}$; $E(G_3) = \{<0,1>,<1,0>,<1,2>\}$.

6.1 Definition and representations

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



(a) Graph with a self edge (b) Multigraph

The number of distinct unordered pairs *(u,* v) with *u !=* v in a graph with *n* vertices is *n (n* - 1) */* 2. This is the maximum number of edges in any n-vertex, undirected graph. An n-vertex, undirected graph with exactly *n(n* -1)/2 edges is said to be *complete.* In the case of a directed graph on *n* vertices, the maximum number of edges is *n (n*- 1).

6.1 Definition and representations

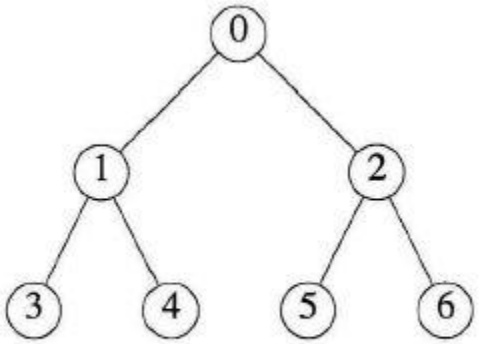
If *(u,* v) is an edge in *E* (G), then we shall say the vertices *u* and v are *adjacent* and that the edge *(u,* v) is *incident* on vertices *u* and *v.*

The vertices adjacent to vertex 1 in G2 are 3, 4, and 0.

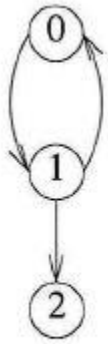
The edges incident on vertex 2 in G2 are (0,2), (2,5), and (2,6).

If <*u, v* > is a directed edge, then vertex *u* is *adjacent to v,* and *v* is *adjacent from u.* The edge <*u,v*> is incident to *u* and *v.*

In G3 , the edges incident to vertex 1 are <0,1>, <1,0>, and <1,2>.

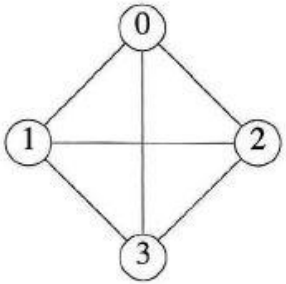


(b) G_2

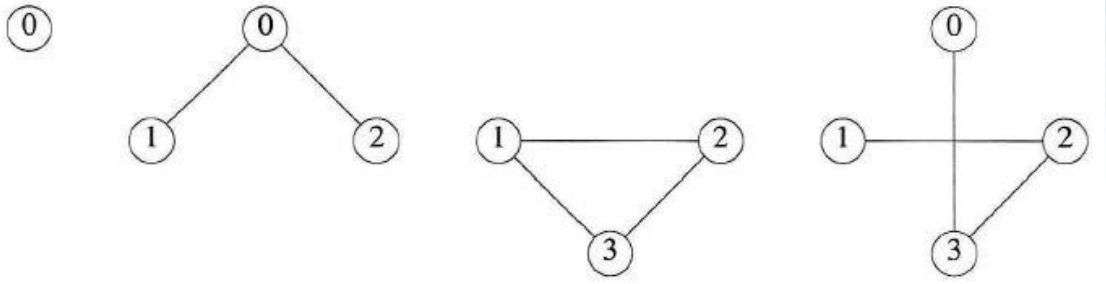


(c) G_3

6.1 Definition and representations

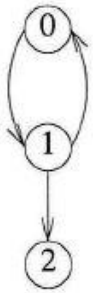


(a) G_1

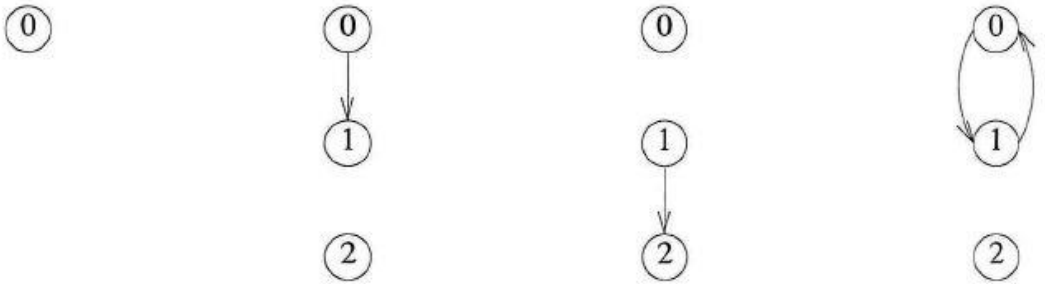


(i) (ii) (iii) (iv)

(a) Some of the subgraphs of G_1



(c) G_3



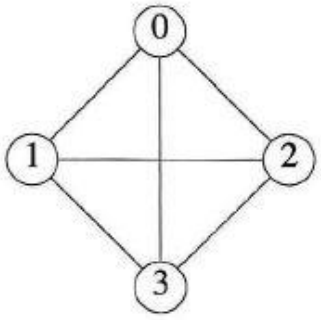
(i) (ii) (iii) (iv)

(b) Some of the subgraphs of G_3

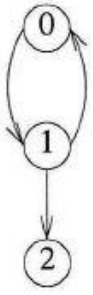
6.1 Definition and representations

A *path* from vertex u to vertex v in graph G is a sequence of vertices $u, i_1, i_2, \cdots, i_k, v$ such that $(u, i_1), (i_1, i_2), \cdots, (i_k, v)$ are edges in $E(G)$. If G' is directed, then the path consists of $<u, i_1>, <i_1, i_2>, \cdots, <i_k, v>$ edges in $E(G')$. 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 last are distinct. A path such as (0,1), (1,3), (3,2), is also written as 0,1,3,2. Paths 0,1,3,2 and 0,1,3,1 of G_1 are both of length 3. The first is a simple path; the second is not. 0,1,2 is a simple directed path in G_3. 0,1,2,1 is not a path in G_3, as the edge <2,1> is not in $E(G_3)$.

A *cycle* is a simple path in which the first and last vertices are the same. 0,1,2,0 is a cycle in G_1. 0,1,0 is a cycle in G_3. For the case of directed graphs we normally add the prefix "directed" to the terms cycle and path.



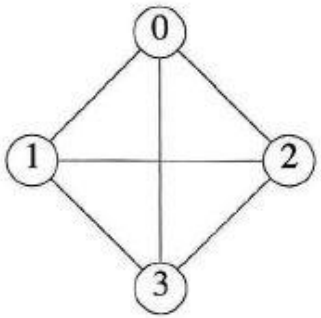
(a) G_1



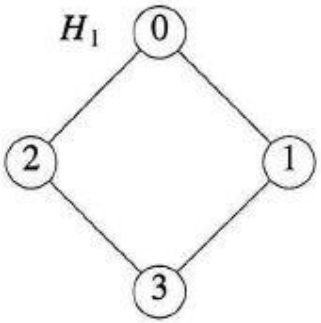
(c) G_3

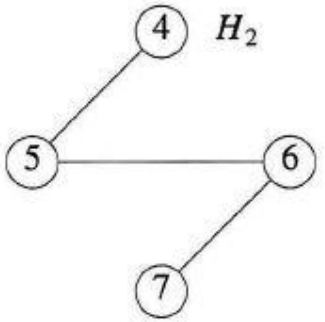
6.1 Definition and representations

In an undirected graph, G, two vertices u and v are said to be *connected* iff there is a path in G from u to v (since G is undirected, this means there must also be a path from v to u). An undirected graph is said to be connected iff for every pair of distinct vertices u and v in $V(G)$ there is a path from u to v in G. Graphs G_1 and G_2 are connected, whereas G_4 of Figure 6.5 is not. A *connected component* (or simply a component), H, of an undirected graph is a *maximal* connected subgraph. By maximal, we mean that G contains no other subgraph that is both connected and properly contains H. G_4 has two components, H_1 and H_2 (see Figure 6.5).



(a) G_1



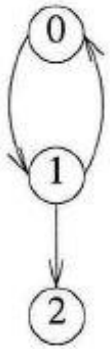


G_4

6.1 Definition and representations

A *tree* is a connected acyclic (i.e., has no cycles) graph.

A directed graph G is said to be *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. The graph G_3 is not strongly connected, as there is no path from vertex 2 to 1. A *strongly connected component* is a maximal subgraph that is strongly connected. G_3 has two strongly connected components (see Figure 6.6).

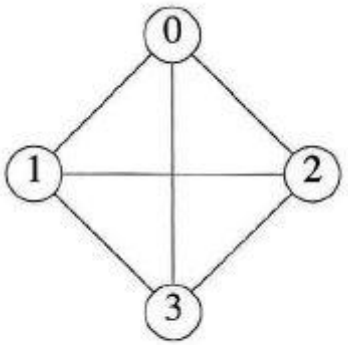


(c) G_3

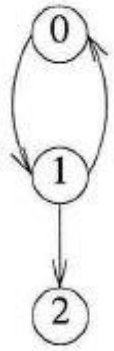
6.1 Definition and representations

The degree of a vertex is the number of edges incident to that vertex. The degree of vertex 0 in G_1 is 3. If G is a directed graph, we define the *in-degree* of a vertex v to be the number of edges for which v is the head. The *out-degree* is defined to be the number of edges for which v is the tail. Vertex 1 of G_3 has in-degree 1, out-degree 2, and degree 3. If d_i is the degree of vertex i in a graph G with n vertices and e edges, then the number of edges is

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



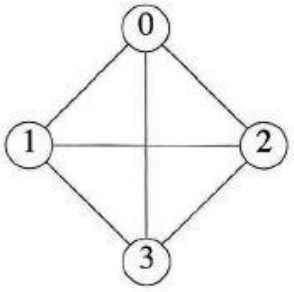
(a) G_1



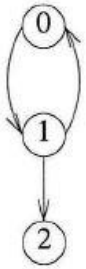
(c) G_3

6.1 Definition and representations

Let $G = (V, E)$ be a graph with n vertices, $n \geq 1$. The adjacency matrix of G is a two-dimensional $n \times n$ array, say a, with the property that $a[i][j] = 1$ iff the edge (i, j) ($<i, j>$ for a directed graph) is in $E(G)$. $a[i][j] = 0$ if there is no such edge in G. The adjacency matrices for the graphs G_1, G_3, and G_4 are shown in Figure 6.7. The adjacency matrix for an undirected graph is symmetric, as the edge (i, j) is in $E(G)$ iff the edge (j, i) is also in $E(G)$. The adjacency matrix for a directed graph may not be symmetric (as is the case for G_3). The space needed to represent a graph using its adjacency matrix is n^2 bits. About half this space can be saved in the case of undirected graphs by storing only the upper or lower triangle of the matrix.



(a) G_1



(c) G_3

	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

6.1 Definition and representations

From the adjacency matrix, one may readily determine if there is an edge connecting any two vertices i and j. For an undirected graph the degree of any vertex i is its row sum:

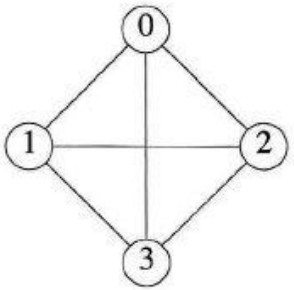
$$\sum_{j=0}^{n-1} a[i][j]$$

For a directed graph the row sum is the out-degree, and the column sum is the in-degree.

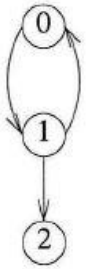
Suppose we want to answer a nontrivial question about graphs, such as, How many edges are there in G? or, Is G connected? Adjacency matrices will require at least $O(n^2)$ time, as $n^2 - n$ entries of the matrix (diagonal entries are zero) have to be examined. When graphs are sparse (i.e., most of the terms in the adjacency matrix are zero) one would expect that the former question could be answered in significantly less time, say $O(e + n)$, where e is the number of edges in G, and $e << n^2/2$. Such a speed-up can be made possible through the use of a representation in which only the edges that are in G are explicitly stored. This leads to the next representation for graphs, adjacency lists.

6.1 Definition and representations

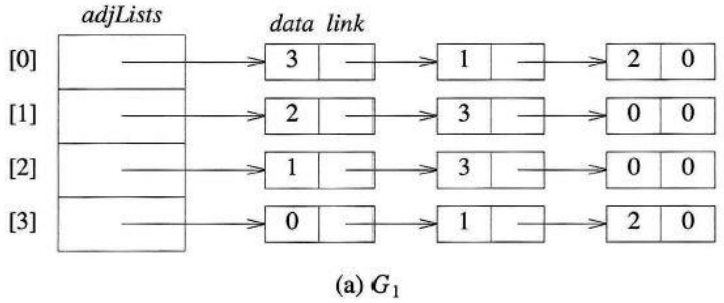
In this representation of graphs, the n rows of the adjacency matrix are represented as n chains (though sequential lists could be used just as well). There is one chain for each vertex in G. The nodes in chain i represent the vertices that are adjacent from vertex i. The *data* field of a chain node stores the index of an adjacent vertex. The adjacency lists for G_1, G_3, and G_4 are shown in Figure 6.8. Notice that the vertices in each chain are not required to be ordered. An array *adjLists* is used so that we can access the adjacency list for any vertex in O(1) time. *adjLists* [*i*] is a pointer to the first node in the adjacency list for vertex *i*.



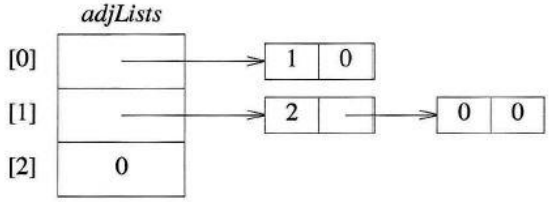
(a) G_1



(c) G_3



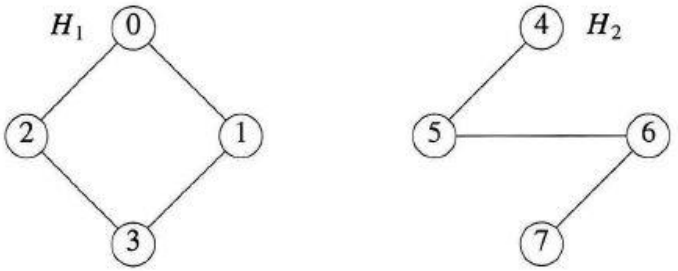
(a) G_1



(b) G_3

6.1 Definition and representations

For an undirected graph with n vertices and e edges, the linked adjacency lists representation requires an array of size n and $2e$ chain nodes. Each chain node has two fields. In terms of the number of bits of storage needed, the node count should be multiplied by $\log n$ for the array positions and $\log n + \log e$ for the chain nodes, as it takes $O(\log m)$ bits to represent a number of value m. If instead of chains, we use sequential lists, the adjacency lists may be packed into an integer array $node[n + 2e + 1]$. In one possible sequential mapping , $node[i]$ gives the starting point of the list for vertex i, $0 \le i < n$, and $node[n]$ is set to $n + 2e + 1$. The vertices adjacent from vertex i are stored in $node[i], \cdots, node[i + 1] - 1$, $0 \le i < n$. Figure 6.9 shows the representation for the graph G_4 of Figure 6.5.



G_4

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

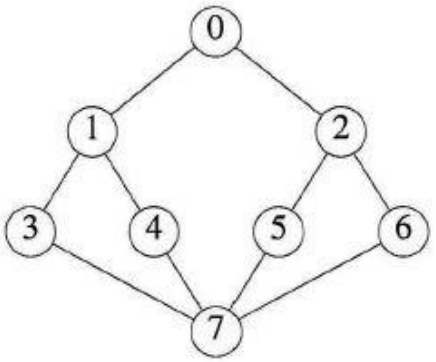
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
9	11	13	15	17	18	20	22	23	2	1	3	0	0	3	1	2	5	6	4	5	7	6

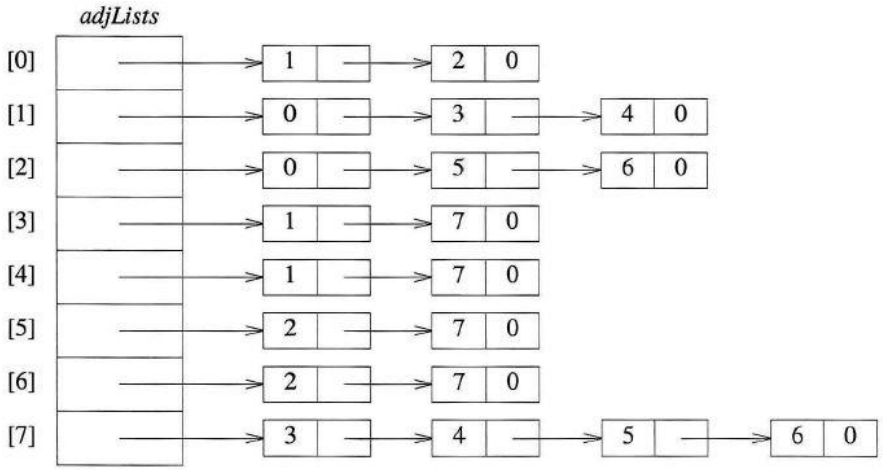
6.2 Elementary graph operations

```
#define FALSE 0
#define TRUE 1
short int visited[MAX_VERTICES];
void dfs(int v)
{/* depth first search of a graph beginning at v */
   nodePointer 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

How to form
a connected component of G?





v0 , v1, v3, v7, v4, v5, v2, v6.

6.2 Elementary graph operations

How to form a connected component of G?

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

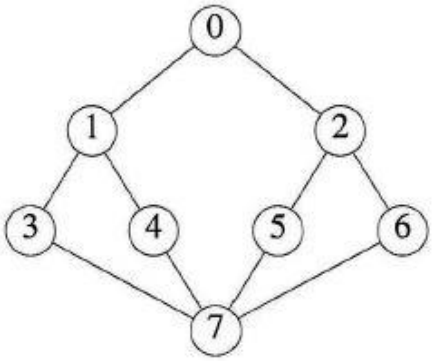
Program 6.1: Depth first search

Analysis of *dfs*: If we represent G by its adjacency lists, then we can determine the vertices adjacent to v by following a chain of links. Since *dfs* examines each node in the adjacency lists at most once, the time to complete the search is $O(e)$. If we represent G by its adjacency matrix, then determining all vertices adjacent to v requires $O(n)$ time. Since we visit at most n vertices, the total time is $O(n^2)$. □

6.2 Elementary graph operations

```
typedef struct queue *queuePointer;
typedef struct queue {
        int vertex;
        queuePointer link;
        };
queuePointer front, rear;
void addq(int);
int deleteq();
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



v0 , *v*1, *v*2, *v*3, *v*4, *v*5, v6, *v7.*

.

6.2 Elementary graph operations

How to form a connected component of G?

Analysis of *bfs*: Since each vertex is placed on the queue exactly once, the **while** loop is iterated at most n times. For the adjacency list representation, this loop has a total cost of $d_0 + \cdots + d_{n-1} = O(e)$, where $d_i = degree\ (v_i)$. For the adjacency matrix representation, the **while** loop takes $O(n)$ time for each vertex visited. Therefore, the total time is $O(n^2)$. As was true of *dfs*, all vertices visited, together with all edges incident to them, form a connected component of G. □

6.2 Elementary graph operations

How to list the connected components of a graph?

```
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");
   }
}
```

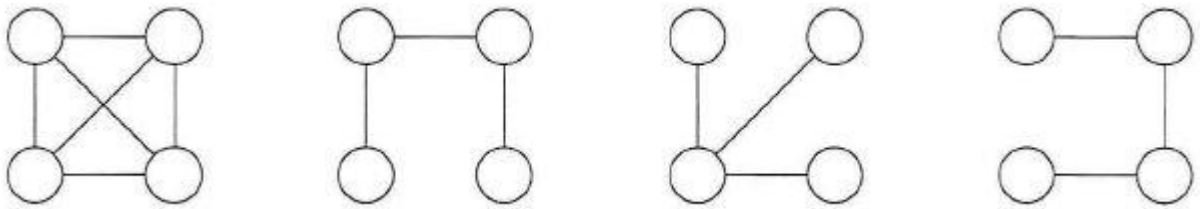
Program 6.3: Connected components

Analysis of *connected*: If G is represented by its adjacency lists, then the total time taken by *dfs* is $O(e)$. Since the **for** loop takes $O(n)$ time, the total time needed to generate all the connected components is $O(n+e)$.

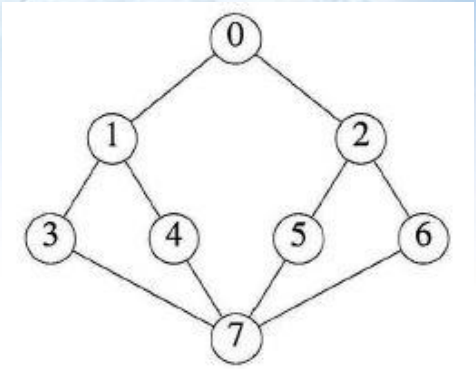
If G is represented by its adjacency matrix, then the time needed to determine the connected components is $O(n^2)$. □

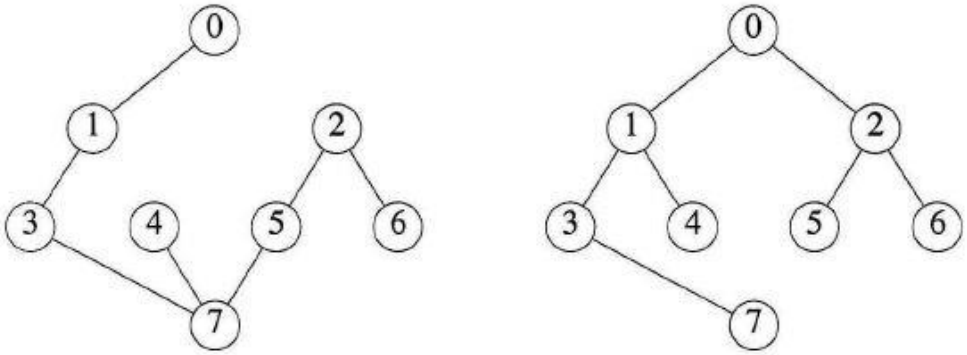
6.2 Elementary graph operations

When graph *G* is connected, a depth first or breadth first search starting at any vertex visits all the vertices in *G.* The search implicitly partitions the edges in *G* into two sets: *T* (for tree edges) and *N* (for nontree edges). T is the set of edges used or traversed during the search and *N* is the set of remaining edges. We can determine the set of tree edges by adding a statement to the **if** clause of either *dfs* or *bfs* that inserts the edge (v, *w)* into a linked list of edges. *(T* represents the head of this linked list.) The edges in *T* form a tree that includes all vertices of *G.* A *spanning tree* is any tree that consists solely of edges in *G* and that includes all the vertices in *G.*



6.2 Elementary graph operations

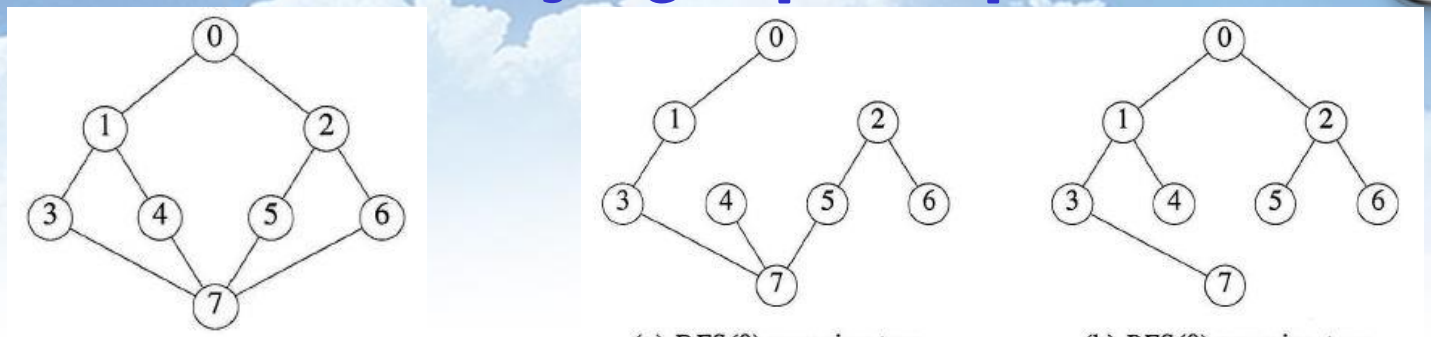




(a) *DFS* (0) spanning tree (b) *BFS* (0) spanning tree

Now suppose we add a nontree edge, *(v, w),* into any spanning tree, *T.* The result is a cycle that consists of the edge (v, *w)* and all the edges on the path from *w* to *v* in *T.*

6.2 Elementary graph operations



(a) *DFS* (0) spanning tree (b) *BFS* (0) spanning tree

Let us examine a second property of spanning trees. 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 edges. Any connected graph with n vertices must have at least $n - 1$ edges, and all connected graphs with $n - 1$ edges are trees. Therefore, we conclude that a spanning tree has $n - 1$ edges. (The exercises explore this property more fully.)

Constructing minimal subgraphs finds frequent application in the design of communication networks. Suppose that the vertices of a graph, G, represent cities and the edges represent communication links between cities. The minimum number of links needed to connect n cities is $n - 1$. Constructing the spanning trees of G gives us all feasible choices. However, we know that the cost of constructing communication links between cities is rarely the same. Therefore, in practical applications, we assign weights to the edges. These weights might represent the cost of constructing the communication link or the length of the link. Given such a weighted graph, we would like to select the spanning tree that represents either the lowest total cost or the lowest overall length. We assume that the cost of a spanning tree is the sum of the costs of the edges of that tree.

6.3 Minimum cost spanning trees

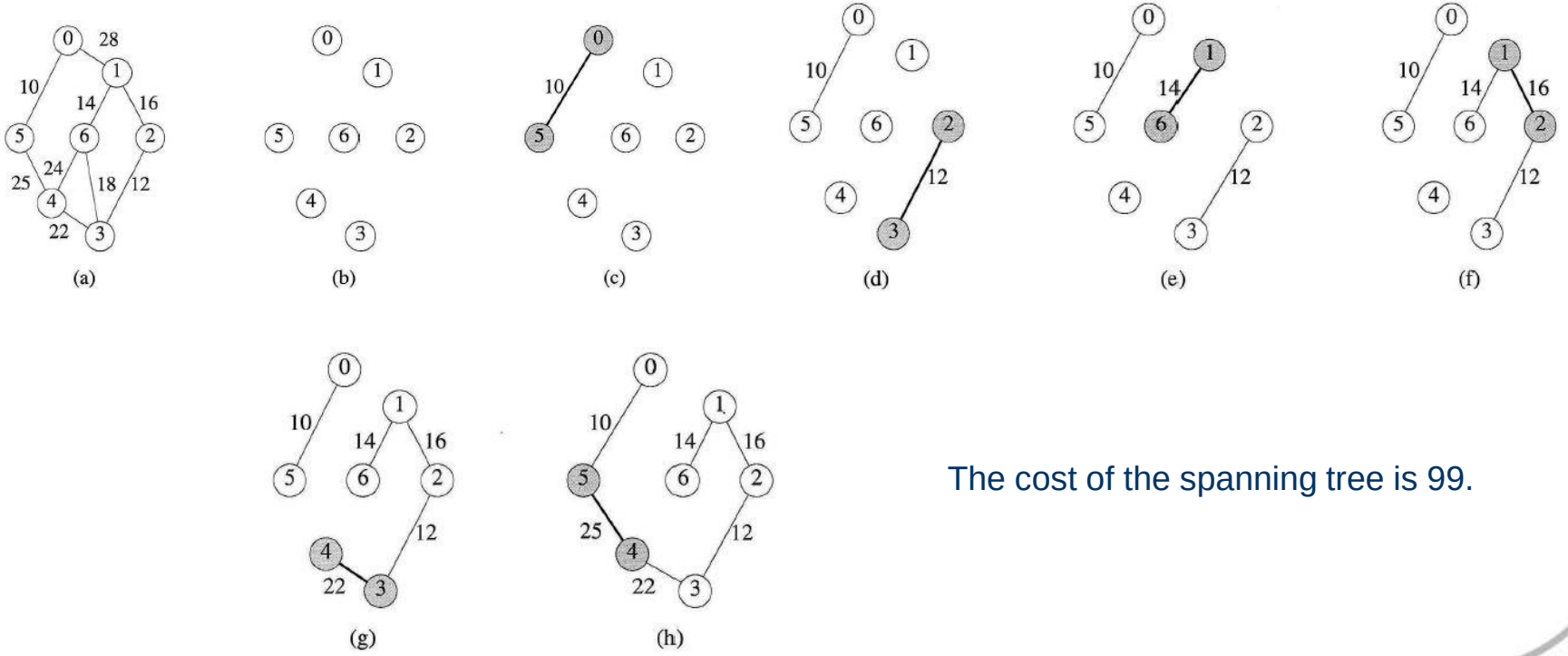
The *cost* of a spanning tree of a weighted undirected graph is the sum of the costs (weights) of the edges in the spanning tree. A *minimum cost spanning tree* is a spanning tree of least cost. Three different algorithms can be used to obtain a minimum cost spanning tree of a connected undirected graph. All three use an algorithm design strategy called the *greedy method*. We shall refer to the three algorithms as Kruskal's, Prim's, and Sollin's algorithms, respectively.

In the greedy method, we construct an optimal solution in stages. At each stage, we make a decision that is the best decision (using some criterion) at this time. Since we cannot change this decision later, we make sure that the decision will result in a feasible solution. A feasible solution is one which works within the constraints specified by the problem.

6.3 Minimum cost spanning trees

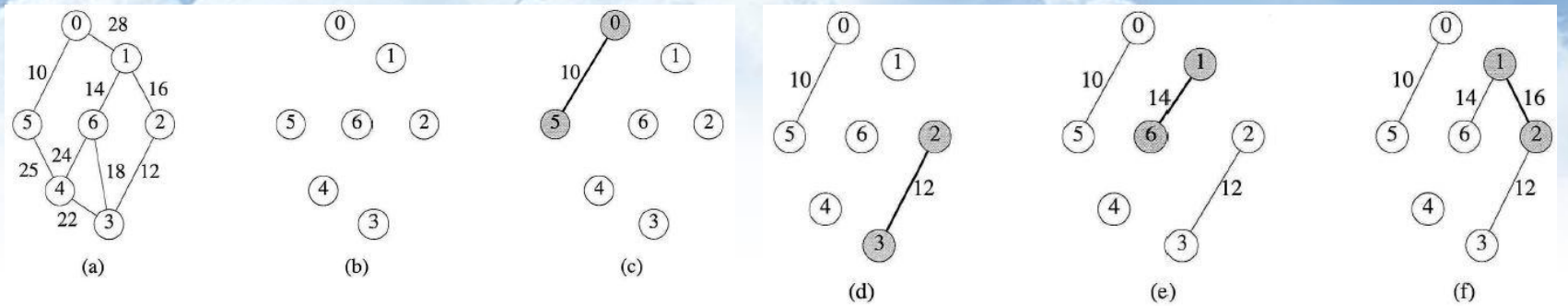
6.3.1 Kruskal's Algorithm

Kruskal's algorithm builds a minimum cost spanning tree T by adding edges to T one at a time. The algorithm selects the edges for inclusion in T in nondecreasing order of their cost. An edge is added to T if it does not form a cycle with the edges that are already in T. Since G is connected and has $n > 0$ vertices, exactly $n - 1$ edges will be selected for inclusion in T.

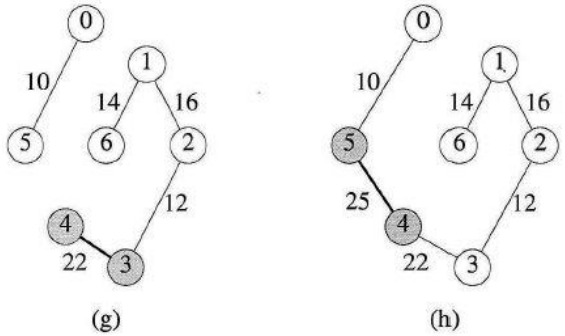


The cost of the spanning tree is 99.

6.3 Minimum cost spanning trees




```
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

6.3 Minimum cost spanning trees

```
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

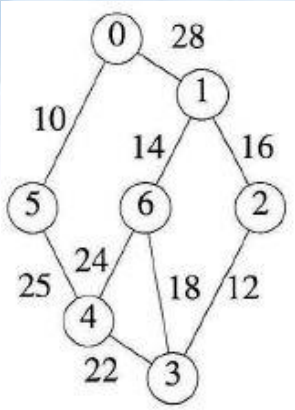
To determine an edge with minimum cost and delete that edge, we can handle both of these operations efficiently if we maintain the edges in *E* as a sorted sequential list in O(e log e) time. Actually, it is not necessary to sort the edges in *E* as long as we are able to find the next least cost edge quickly. Obviously a min heap is ideally suited for this task since we can determine and delete the next least cost edge in O(log e) time. Construction of the heap itself requires O(e log e) time.

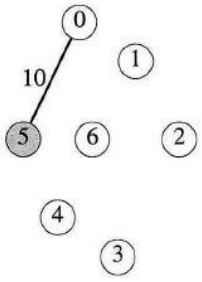
To check that the new edge, (v, w), does not form a cycle in *T* and to add such an edge to *T,* we may use the union-find operations. Since the union-find operations require less time than choosing and deleting an edge, the latter operations determine the total computing time of Kruskal's algorithm. Thus, the total computing time is O(e log e).

6.3 Minimum cost spanning trees

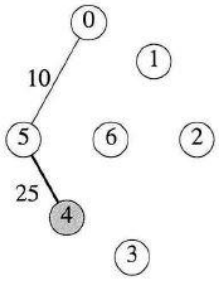
6.3.2 Prim's Algorithm

Prim's algorithm, like Kruskal's, constructs the minimum cost spanning tree one edge at a time. However, at each stage of the algorithm, the set of selected edges forms a tree. By contrast, the set of selected edges in Kruskal's algorithm forms a forest at each stage. Prim's algorithm begins with a tree, T, that contains a single vertex. This may be any of the vertices in the original graph. Next, we add a least cost edge (u, v) to T such that $T \cup \{(u, v)\}$ is also a tree. We repeat this edge addition step until T contains $n - 1$ edges. To make sure that the added edge does not form a cycle, at each step we choose the edge (u, v) such that exactly one of u or v is in T. Program 6.8 contains a formal description of Prim's algorithm. T is the set of tree edges, and TV is the set of tree vertices, that is, ver-

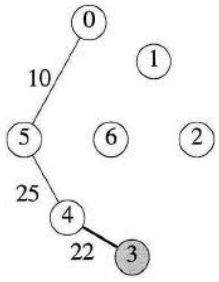




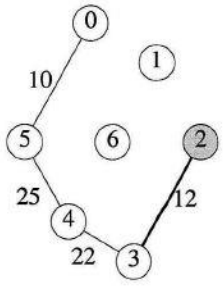
(a)



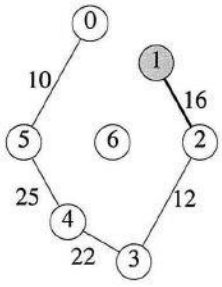
(b)



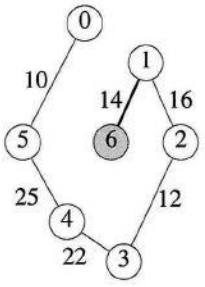
(c)



(d)



(e)

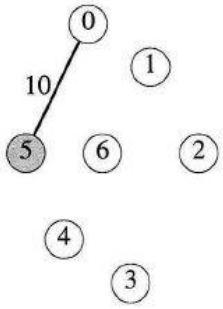


(f)

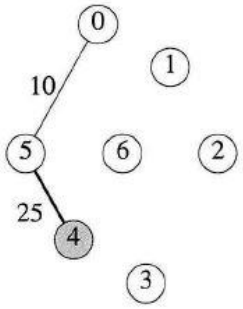
6.3 Minimum cost spanning trees

```
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");
```

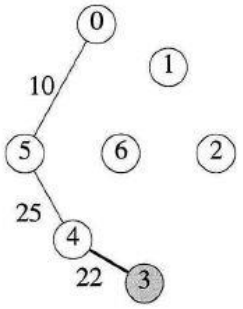
Program 6.8: Prim's algorithm



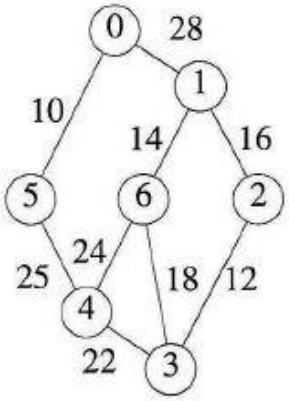
(a)

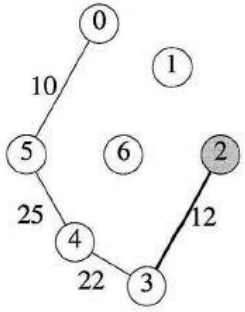


(b)

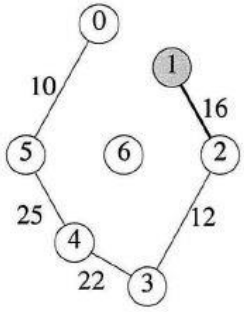


(c)

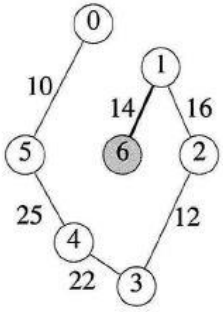




(d)



(e)



(f)

6.3 Minimum cost spanning trees

```
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");
```

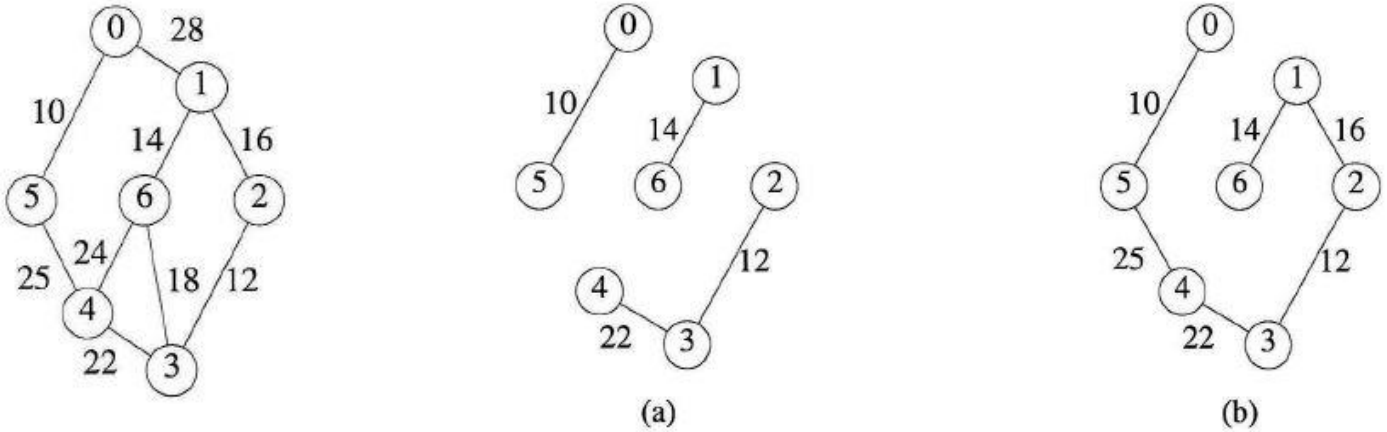
Program 6.8: Prim's algorithm

To implement Prim's algorithm, we assume that each vertex v that is not in TV has a companion vertex, $near(v)$, such that $near(v) \in TV$ and $cost(near(v), v)$ is minimum over all such choices for $near(v)$. (We assume that $cost(v, w) = \infty$ if $(v, w) \notin E$). At each stage we select v so that $cost(near(v), v)$ is minimum and $v \notin TV$. Using this strategy we can implement Prim's algorithm in $O(n^2)$, where n is the number of vertices in G.

6.3 Minimum cost spanning trees

6.3.3 Sollin's Algorithm

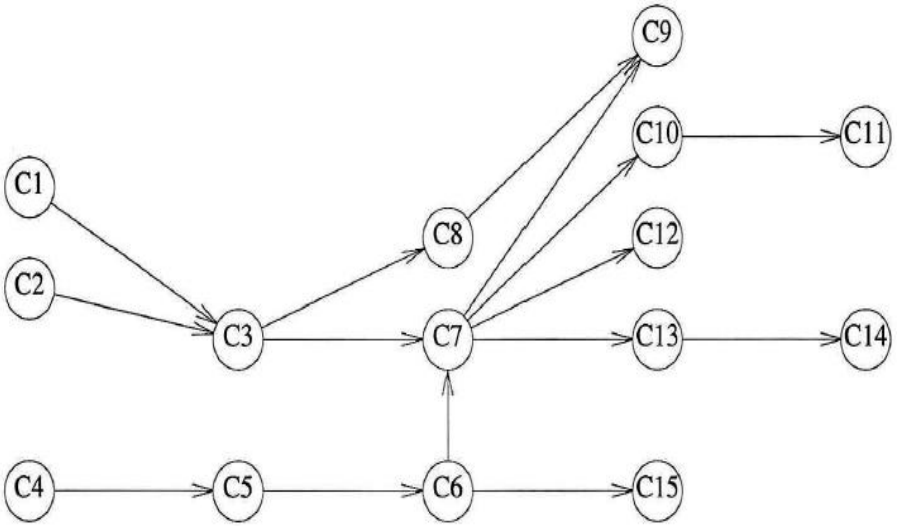
Unlike Kruskal's and Prim's algorithms, Sollin's algorithm selects several edges for inclusion in T at each stage. At the start of a stage, the selected edges, together with all n graph vertices, form a spanning forest. During a stage we select one edge for each tree in the forest. This edge is a minimum cost edge that has exactly one vertex in the tree. Since two trees in the forest could select the same edge, we need to eliminate multiple copies of edges. At the start of the first stage the set of selected edges is empty. The algorithm terminates when there is only one tree at the end of a stage or no edges remain for selection.



6.5 Activity Networks

Definition: A directed graph G in which the vertices represent tasks or activities and the edges represent precedence relations between tasks is an *activity-on-vertex network* or AOV network. □

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



6.5 Activity Networks

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. i is an *immediate predecessor* of 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. □

C3 and C6 are immediate predecessors of C7. C9, C10, C12, and C13 are immediate successors of C7. C14 is a successor, but not an immediate successor, of C3.

Definition: A relation $\cdot$ is *transitive* iff it is the case that 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 for no element x in S is it the case that $x \cdot x$. A precedence relation that is both transitive and irreflexive is a *partial order*. □

Generally, AOV networks are incompletely specified, and the edges needed to make the precedence relation transitive are implied. Given an AOV network, one of our concerns would be to determine whether or not the precedence relation defined by its edges is irreflexive. This is identical to determining whether or not the network contains any directed cycles. A directed graph with no directed cycles is an *acyclic* graph.

6.5 Activity Networks

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

There are several possible topological orders for the network of Figure 6.37(b). Two of these are

C1, C2, C4, C5, C3, C6, C8, C7, C10, C13, C12, C14, C15, C11, C9

and

C4, C5, C2, C1, C6, C3, C8, C15, C7, C9, C10, C11, C12, C13, C14

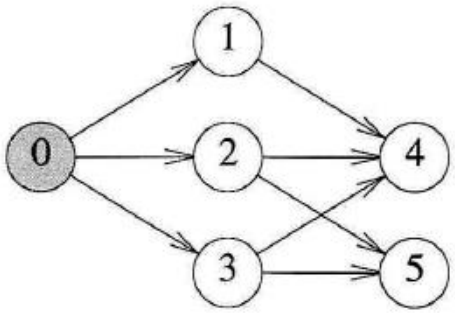
If a student were taking just one course per term, then she or he would have to take them in topological order. If the AOV network represented the different tasks involved in assembling an automobile, then these tasks would be carried out in topological order on an assembly line. The algorithm to sort the tasks into topological order is straightforward and proceeds by listing a vertex in the network that has no predecessor. Then, this vertex together with all edges leading out from it is deleted from the network. These two steps are repeated until all vertices have been listed or all remaining vertices in the network have predecessors, and so none can be removed. In this case there is a cycle in the network, and the project is infeasible. The algorithm is stated more formally in Program 6.13.

6.5 Activity Networks

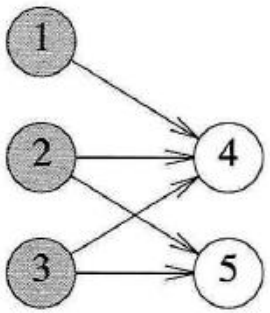
```
1   Input the AOV network.  Let n be the number of vertices.
2   for (i = 0; i < n; i++) /* output the vertices */
3   {
4     if (every vertex has a predecessor) return;
5        /* network has a cycle and is infeasible */
6     pick a vertex v that has no predecessors;
7     output v;
8     delete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

Program 6.13: Design of an algorithm for topological sorting

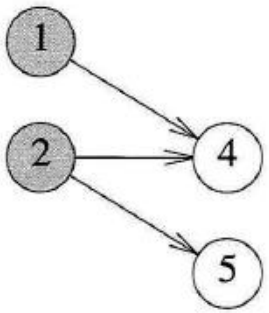
6.5 Activity Networks



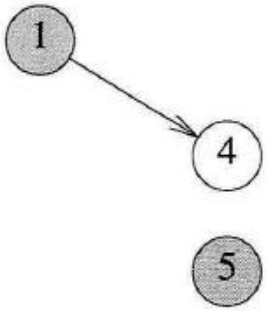
(a) Initial



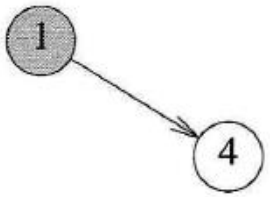
(b) Vertex 0 deleted



(c) Vertex 3 deleted



(d) Vertex 2 deleted



(e) Vertex 5 deleted

(f) Vertex 1 deleted

Topological order generated: 0, 3, 2, 5, 1, 4

Figure 6.38: Action of Program 6.13 on an AOV network (shaded vertices represent candidates for deletion)

6.5 Activity Networks

To obtain a complete algorithm that can be easily translated into a computer program, it is necessary to specify the data representation for the AOV network. The choice of a data representation, as always, depends on the functions you wish to perform. In this problem, the functions are

(1) decide whether a vertex has any predecessors (line 4)

(2) delete a vertex together with all its incident edges (line 8)

To perform the first task efficiently, we maintain a count of the number of immediate predecessors each vertex has. The second task is easily implemented if the network is represented by its adjacency lists. Then the deletion of all edges leading out of vertex v can be carried out by decreasing the predecessor count of all vertices on its adjacency list. Whenever the count of a vertex drops to zero, that vertex can be placed onto a list of vertices with a zero count. Then the selection in line 6 just requires removal of a vertex from this list.

6.5 Activity Networks

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

The *count* field contains the in-degree of that vertex and *link* is a pointer to the first node on the adjacency list. Each node has two fields, *vertex* and *link*. This can be done easily at the time of input. When edge <*i*,*j*> is input, the count of vertex *j* is incremented by 1. Figure 6.39(a) shows the internal representation of the network of Figure 6.38(a).

Inserting these details into Program 6.13, we obtain the C function *topSort* (Program 6.14). The list of vertices with zero count is maintained as a custom stack. A queue could have been used instead, but a stack is slightly simpler. The stack is linked through the *count* field of the header nodes, since this field is of no use after a vertex's

6.5 Activity Networks

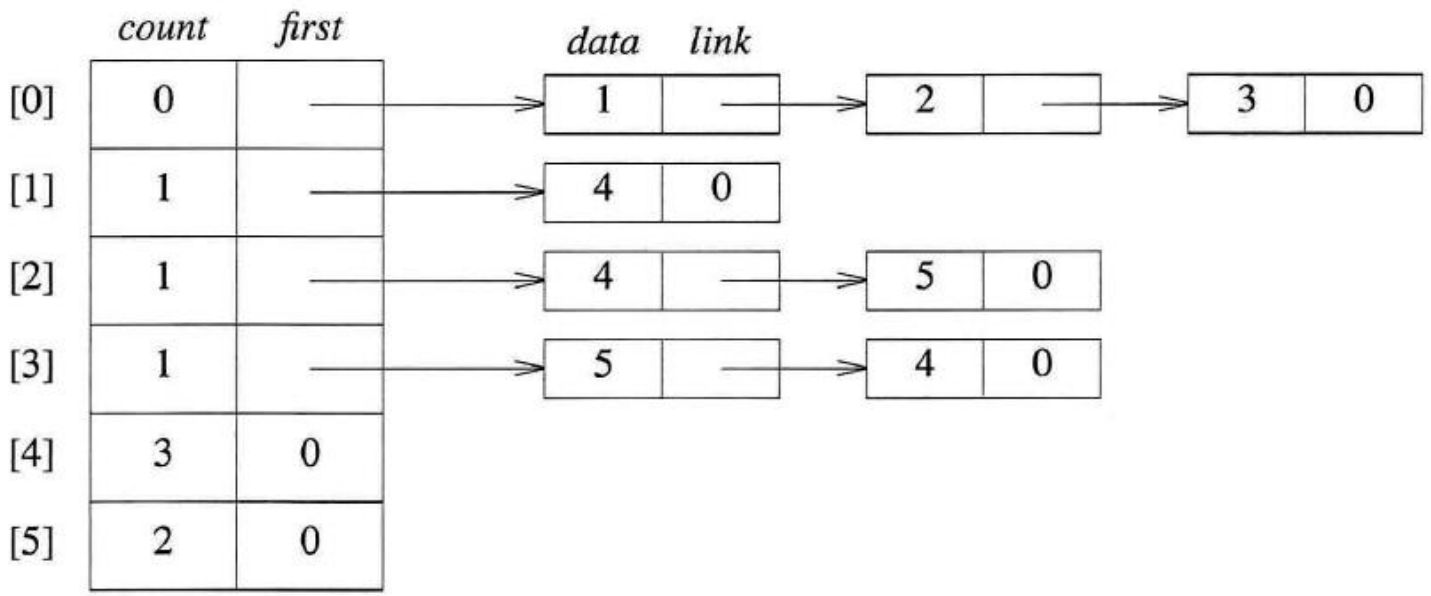


Figure 6.39: Internal representation used by topological sorting algorithm

6.5 Activity Networks

count has become zero.

Analysis of *topSort*: As a result of a judicious choice of data structures, *topSort* is very efficient. The first **for** loop takes $O(n)$ time, on a network with n vertices and e edges. The second **for** loop is iterated n times. The **if** clause is executed in constant time; the **for** loop within the **else** clause takes time $O(d_i)$, where d_i is the out-degree of vertex i. Since this loop is encountered once for each vertex that is printed, the total time for this part of the algorithm is:

$$O((\sum_{i=0}^{n-1} d_i) + n) = O(e + n)$$

Thus, the asymptotic computing time of the algorithm is $O(e + n)$. It is linear in the size of the problem! □

6.5 Activity Networks

```
void topSort(hdnodes graph[], int n)
{
   int i,j,k,top;
   nodePointer 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(EXIT_FAILURE);
      }
      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;
            }
         }
      }
}
```

Thank You !

노력 없이 이룰 수 있는 것 아무것도 없다.