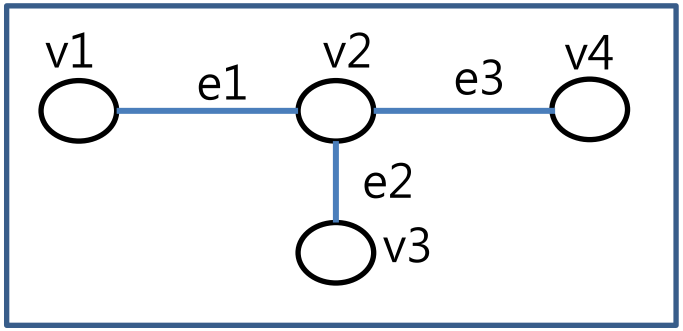
Chap 6. Graph



An **undirected** graph(**무방향 그래프**)

$G = (V, E)$, *where* $V(G) = \{v_1, v_2, v_3, v_4\}$, $E(G) = \{e_1, e_2, e_3\}$, where $e_1 = (v_1,v_2)$, $e_2 = (v_2,v_3)$, $e_3 = (v_2,v_4)$.

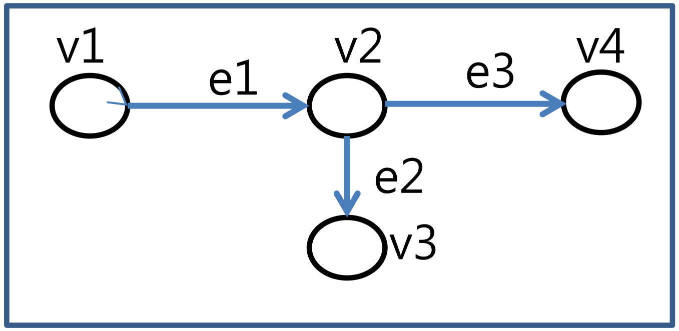
- **graph(그래프)란 vertex (정점<頂點[정수리 정, 점 점]>)와 edge (간선<幹線 [줄기 간, 줄 선]>)로 구성된 자료이다.**
- **각 간선 e는 한 쌍의 무순(unordered)의 정점 (u, v)에 부속(incident)되며 정점 u와 v는 반드시 다를 필요는 없다. 정점과 간선의 집합은 finite(유한<有限[있을 유, 한할 한]>)하다고 가정한다. 이 경우, 그 그래프는 유한하다.**

무방향 그래프에서 각 **edge**는 **(u, v)**의 쌍으로 표현된다. 이 경우, **v**와 **u**의 순서는 의미를 가지지 않는다. 따라서, **(u, v)** 와 **(v, u)**는 동일 **edge**를 나타낸다.

간선 *e*1은 정점 *v*1과 *v*2를 연결한다. 따라서,
(1) 인접(Adjacent, 인접<隣接[이웃 인, 이을 접]>)
- 정점 v1과 v2는 간선 e1에 의해 인접하다.
- 정점 v2에 인접한 정점은 {v1, v3, v4}.

(2) 부속(Incident, 부속 <附屬 [붙을 부 무리 속]>, be attached to, be affiliated with)
- 간선 *e*1은 정점 v1과 v2에 부속된다.
- 정점 v2에 부속된 간선은 {e1, e2, e3}.

Degree(차수<次數[버금 차, 셈 수]) : 정점에 부속된 간선의 수를 그 정점의 차수라 한다.
- 정점 v1의 차수는 1, 정점 v2의 차수는 3.



A **directed** graph(**방향 그래프**)

$G = (V, E)$, *where* $V(G) = \{v_1, v_2, v_3, v_4\}$, $E(G) = \{e_1, e_2, e_3\}$, where $e_1 = <v_1,v_2>$, $e_2 = <v_2,v_3>$, $e_3 = <v_2,v_4>$.

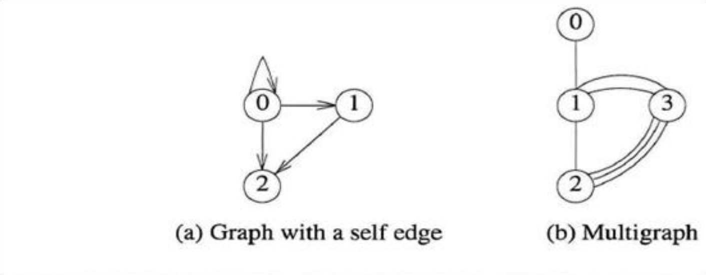
방향 그래프에서 각 **edge**는 **<u, v>**의 쌍으로 표현된다. 이 경우, **u**와 **v**를 각각 그 **edge**의 를 ***tail(꼬리)***와 ***head (머리)***라 한다. 따라서, **<u, v>**와 **<v, u>**는 두 개의 다른 **edge**를 나타낸다.

간선 *e*1은 정점 *v*1을 *v*2로 연결한다. 따라서,
(1) 인접(Adjacent)
- 정점 v1은 정점v2에 간선 e1으로 인접하다.
- 정점 v2는 정점v1으로부터 간선 e1으로 인접하다.
- 정점 v2에서 인접한 정점은 {v3, v4}.

(2) 부속(Incident)
- 간선 *e*1은 정점 v1과 v2에 부속된다.
- 정점 v2에 부속된 간선은 {e1, e2, e3}.

차수(Degree)
- **In-Degree(진입차수): 정점으로 들어오는 간선의 수**
- **Out-Degree(진출차수): 정점에서 나가는 간선의 수.**
- 정점 v2의 진입차수는 1, 진출차수는 2이다.
- 정점 v2의 차수는 3이다.

- A graph $G(V, E)$ is a structure which consists of a nonempty set of *vertices* $V = \{v_1, v_2, \ldots\}$ and a set of *edges* $E = \{e_1, e_2, \ldots\}$; each edge e is *incident* to the elements of an unordered pair of vertices (u, v) which are not necessarily distinct. Unless otherwise stated, both V and E are assumed to be finite. In this case we say that G is finite.
- For the edge e of (u, v), the two vertices are called *endpoints* of the edge e.
  - Edges having the same endpoints are called *parallel edges*. A graph with parallel edges is called a *multigraph*.
  - Once both endpoints of an edge are the same, such an edge is called a *self-loop* or *self-edge*.



(a) Graph with a self edge

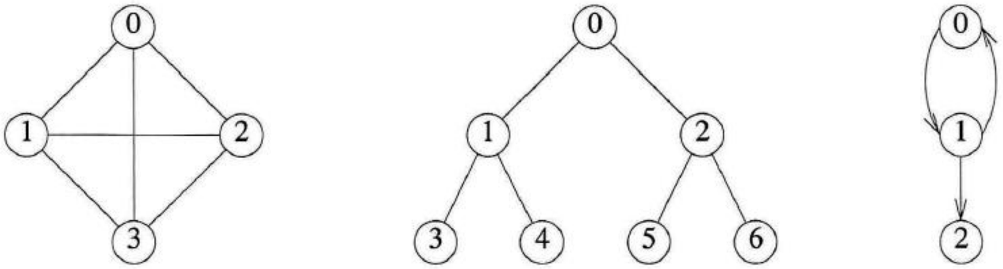
(b) Multigraph

Figure 6.3: Examples of graphlike structures

- In an *undirected graph* the pair of vertices representing any edge is unordered. Thus 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, <u, v> and <v, u> represent two different edges.
- If (u, v) is an edge in E(G), then vertices u and v are *adjacent* and the edge (u, v) is *incident* on vertices u and v. If <u, v> is a directed edge, vertex *u* is *adjacent to* *v*, and *v* is *adjacent from* *u*. The edge <*u*, *v*> is *incident* to u and v.
- The *degree* of a vertex is the number of edges incident to that vertex. 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.
- In the remainder of this chapter, we shall refer to a directed graph as a *digraph*. When we use the term graph, we assume that it is an undirected graph.

Often, both the **undirected edge (*i*, *j*)** and the **directed edge <*i*, *j*>** are written as (*i*, *j*). Which is meant is deduced from the context. In this book, we refrain from this practice.

- A ***Complete graph*****(완전 그래프: 完全)**
  - An *n*-vertex, undirected graph with ***n*(*n*-1)/2** edges is said to be ***complete***.

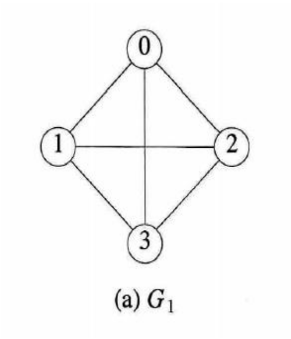

(a) G_1 C.G. (b) G_2 Not C.G. (c) G_3 Not C.G.

- In the case of directed graph on *n* vertices,
  - the maximum number of edges is ***n*(*n*-1)**.

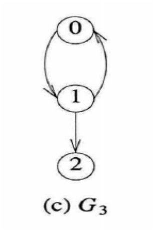
- If d_i is the degree of vertex *i* in *G* with *n* vertices and *e* edges, then the number of edges is $e = (\sum_{i=0}^{n-1} d_i)/2$

Subgraph(부분 그래프<部分, 떼 부, 나눌 분>)

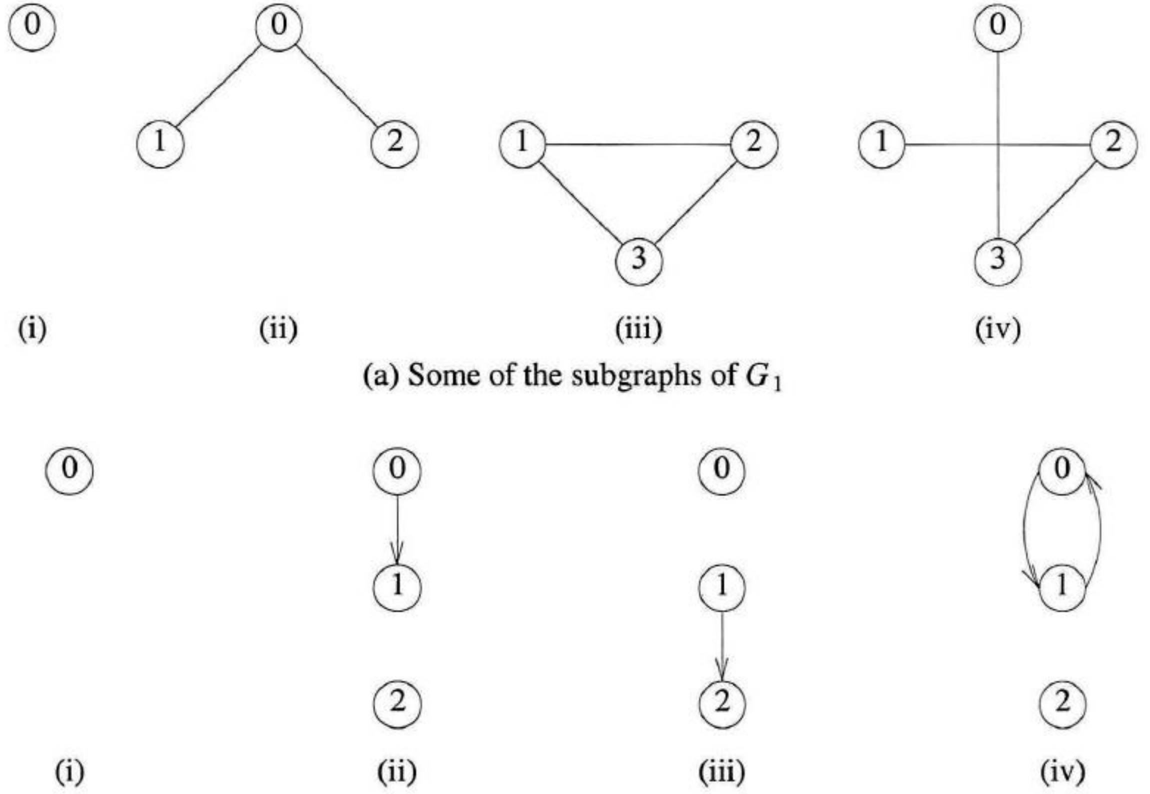
- A ***Subgraph*** of G is a graph G' such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$.



(a) G_1



(c) G_3



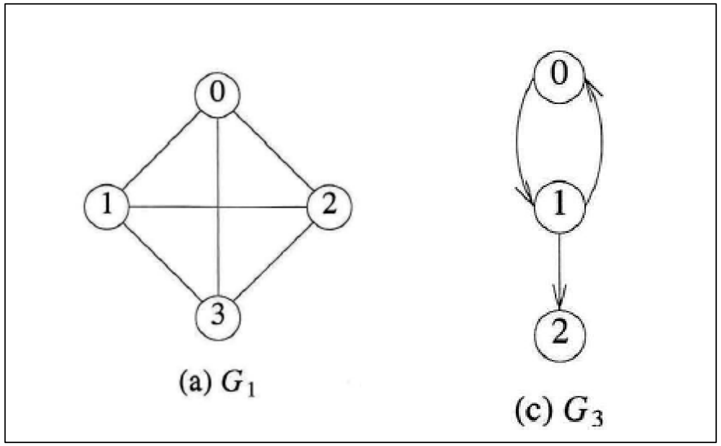
(a) Some of the subgraphs of G_1

(b) Some of the subgraphs of G_3

Figure 6.4: Some subgraphs

Path(경로<經路: 지날 경, 길 로>)

- A ***path*** from vertex u to vertex v in G is a sequence of vertices $u, i_1, i_2, \ldots, i_k, v$ such that $(u, i_1), (i_1, i_2), \ldots, (i_k, v)$ are 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 ***cycle*** is a simple path in which the first and last vertices are the same.
- 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) G_1

(c) G_3

Connected (연결,連結) and **Connected Component**(연결성분,連結成分)

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

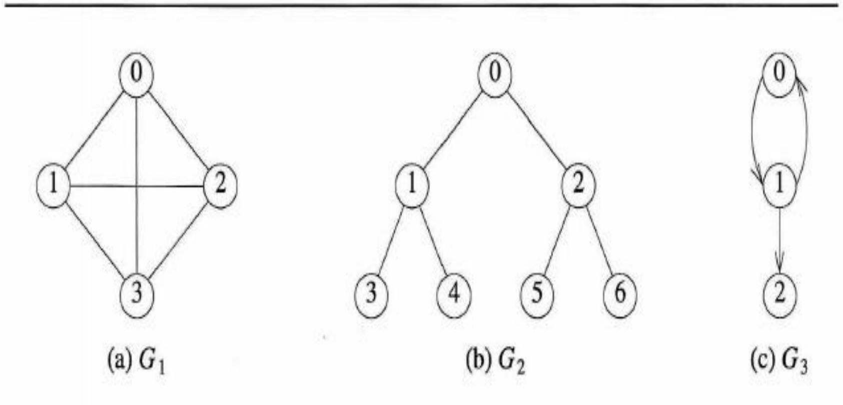


Figure 6.2: Three sample graphs

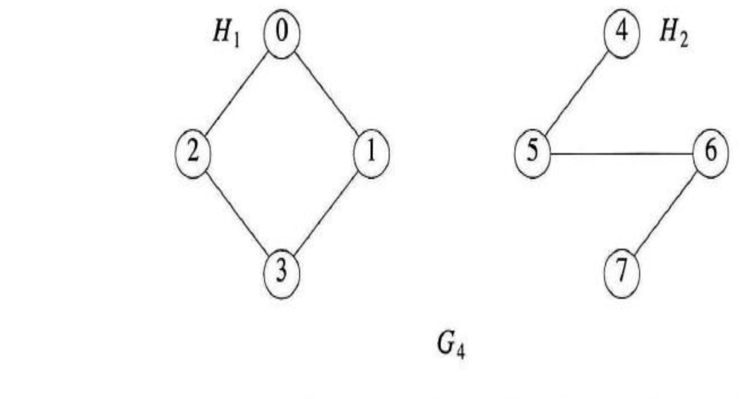
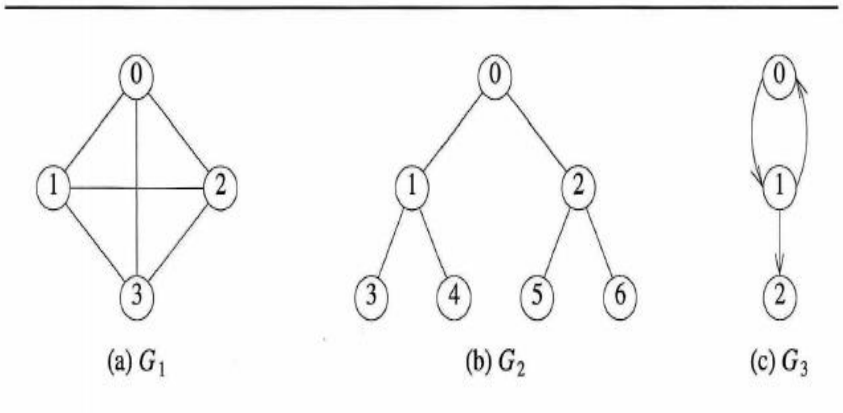


Figure 6.5: A graph with two connected components

Strongly Connected (강연결,强連結) and **Strongly Connected Component**(강연결성분,强連結成分)

- 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 (in Figure 6.2) 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 (in Figure 6.2) has two strongly connected components (see Figure 6.6).



(a) G_1 (b) G_2 (c) G_3

Figure 6.2: Three sample graphs

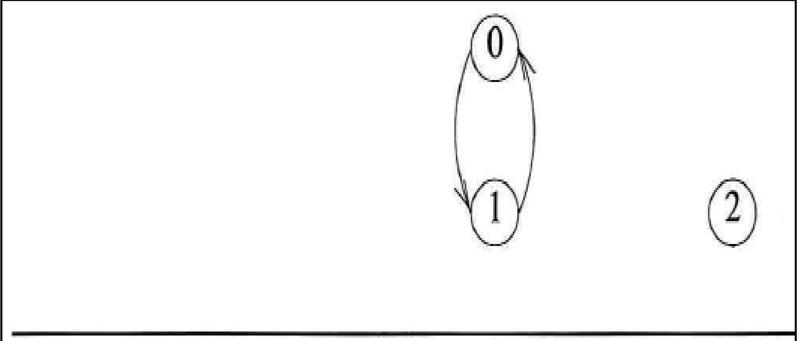


Figure 6.6: Strongly connected components of G_3

Contents

Part I

6.1 THE GRAPH ABSTRACT DATA TYPE

6.2 ELEMENTARY GRAPH OPERATIONS

Part II

6.3 MINIMUM COST SPANNING TREES

6.4 SHORTEST PATHS AND TRANSITIVE CLOSURE

Part III

6.5 ACTIVITY NETWORKS

그래프 부분에서 충분히 실습을 할 수 있도록 할 것
topological sorting (6.5. Activity Networks 에 관련 내용이 있음)

6.1 The Graph Abstract Data Type

6.1.1 introduction

- Königsberg bridge problem

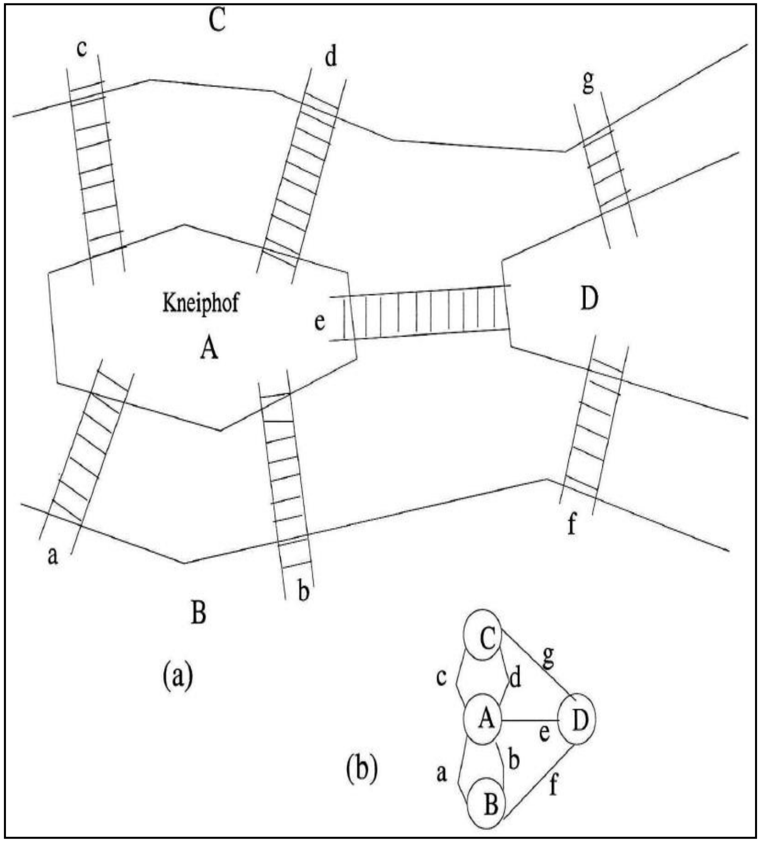
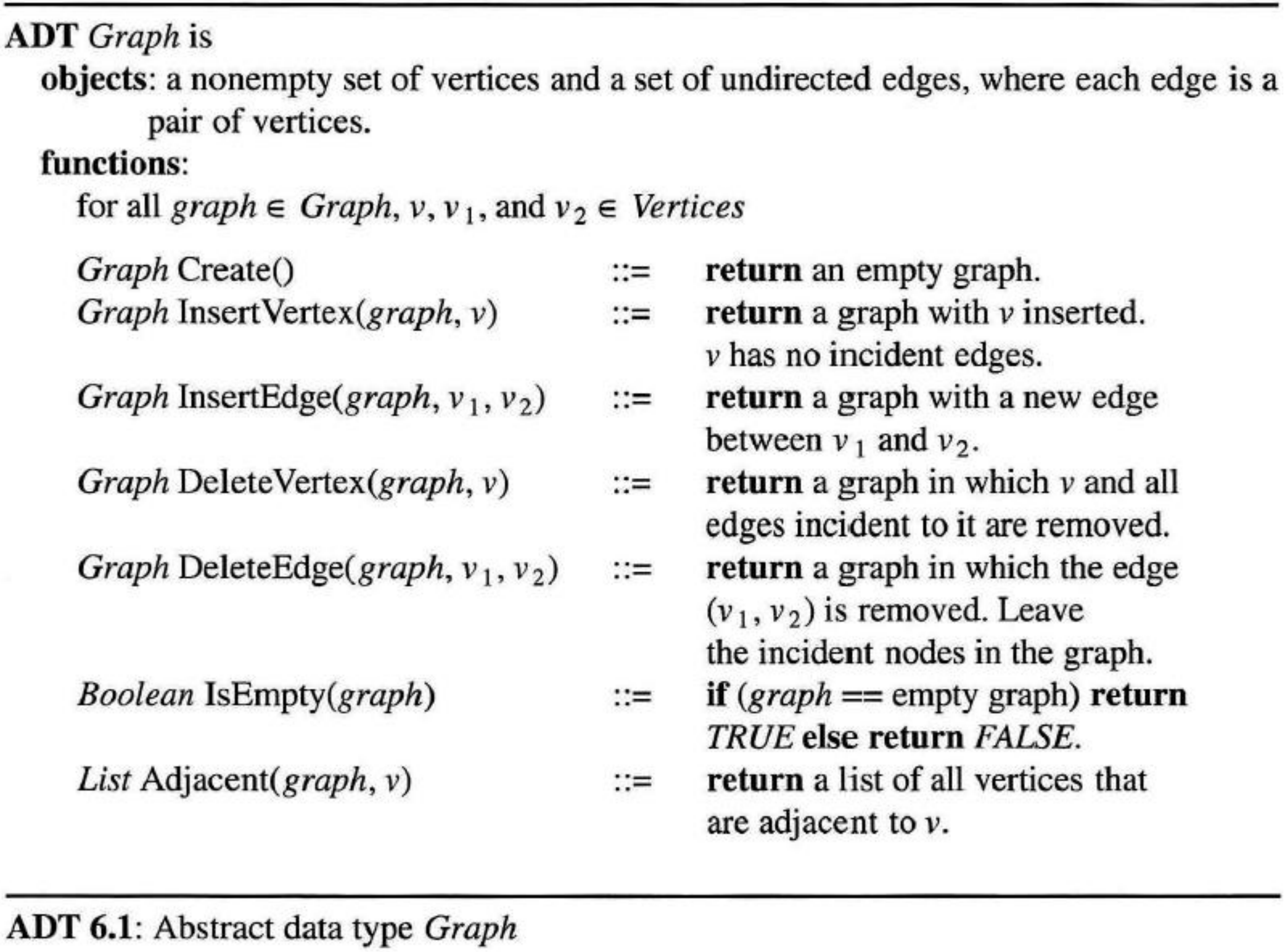


Figure 6.1: (a) Section of the river Pregel in Königsberg; (b) Euler's graph

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**. A walk that does this is called **Eulerian**. There is no **Eulerian walk** for the Konigsberg bridge problem, as all four vertices are of **odd degree**.



ADT *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*.

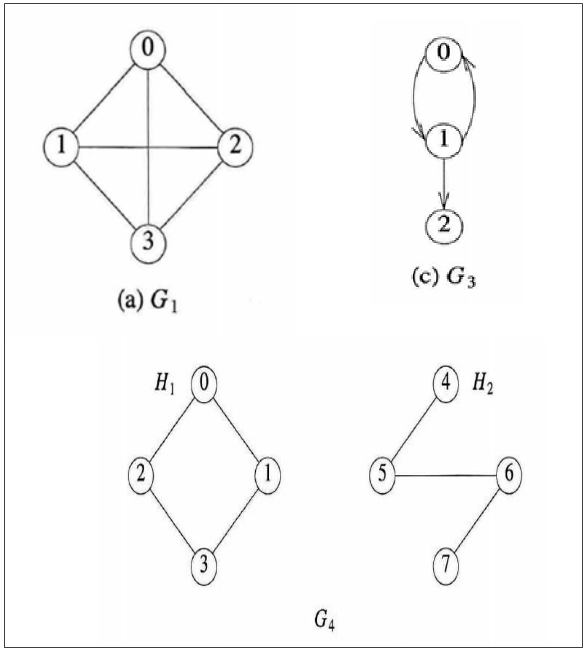
List Adjacent(*graph*, v) ::= **return** a list of all vertices that are adjacent to v.

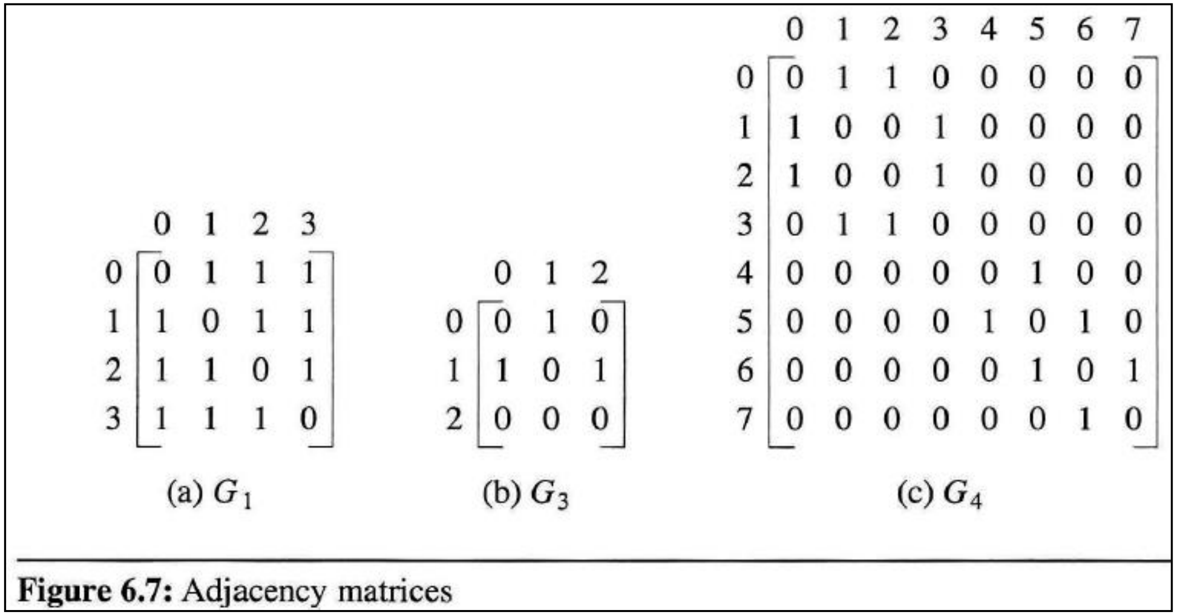
ADT 6.1: Abstract data type *Graph*

6.1.3 Graph Representation

6.1.3.1 Adjacency Matrix

Let *G*=(*V*, *E*) be a graph with *n* vertices, *n*≥1. The ***adjacency matrix*** of *G* is a two-dimensional *n*×*n* array, say *a*, with the property that ***a*[*i*][*j*]=1 iff edge(*i*, *j*) 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.





$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 2 & 1 & 1 & 0 & 1 \\ 3 & 1 & 1 & 1 & 0 \end{array}$$

(a) G_1

$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 2 & 0 & 0 & 0 \end{array}$$

(b) G_3

$$\begin{array}{c|cccccccc} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 3 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 7 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{array}$$

(c) G_4

Figure 6.7: Adjacency matrices

- 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.

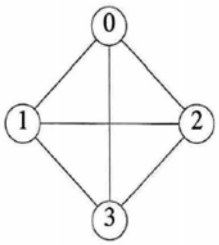
- For an undirected graph, the degree of vertex i is its *row sum*: $\sum_{j=0}^{n-1} a[i][j]$

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

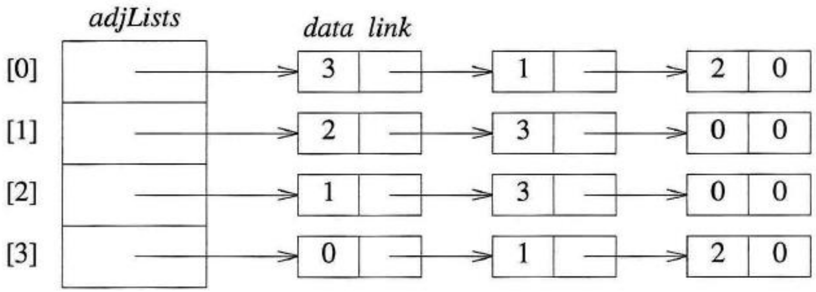
- Suppose we want to answer a nontrivial question about graphs, such as, How many edges are there in G? or, Is G connected? Adjacency matrices will require at least **O(n^2)** time, as n^2 - n entries of the matrix (diagonal entries are zero) have to be examined. When graphs are sparse (i.e., most of 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***.

6.1.3.2 Adjacency Lists

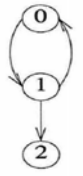
- In this representation of graphs, the *n* rows of the adjacency matrix are represented as ***n* chains** (though sequential lists could be used just as well). **There is one chain for each vertex in *G*.** The nodes in chain *i* represent the vertices that are ***adjacent from*** vertex *i*. The *data* 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*.
- For an undirected graph with *n* vertices and *e* edges, **the linked adjacency lists representation** requires an array of size *n* and 2*e* 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*.



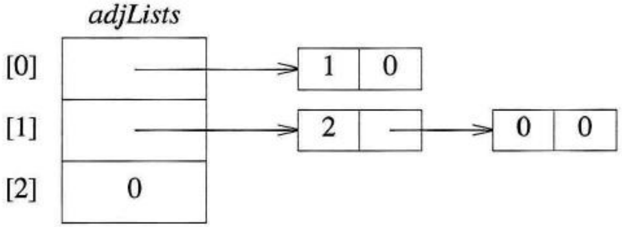
(a) G_1



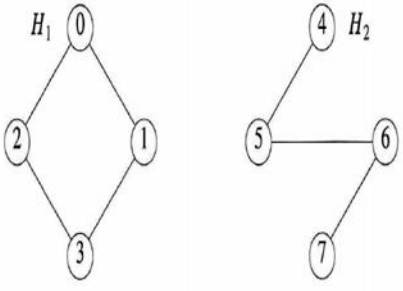
(a) G_1



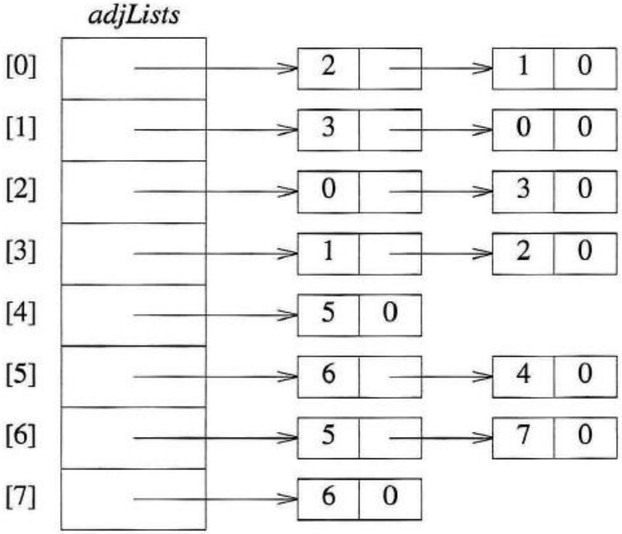
(c) G_3



(b) G_3



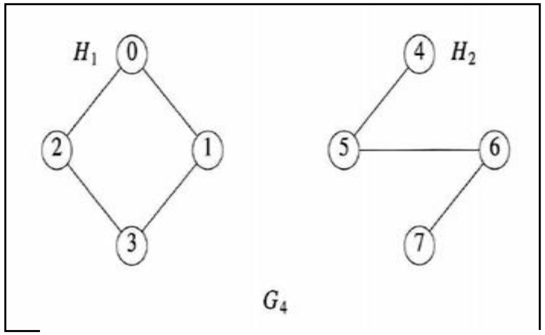
G_4



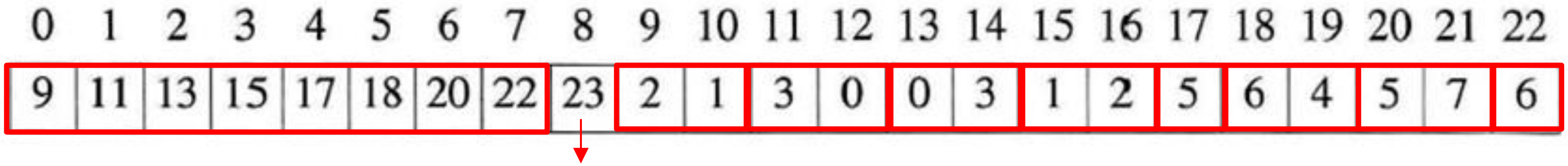
(c) G_4

Figure 6.8: Adjacency lists

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*[node[i]], · · · , *node*[*node*[i + 1]- 1], $0 \le i < n$. Figure 6.9 shows the representation for the graph G_4 of Figure 6.5.



```
int node [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

↓
to know the end of *node* array

Figure 6.9: Sequential representation of graph G_4

- For a digraph, the number of list nodes is only *e*. The out-degree of any vertex may be determined by counting the number of nodes on its adjacency list. Determining the in-degree of a vertex is a little more complex. If there is a need to access repeatedly all vertices adjacent to another vertex, then it may be worth the effort to keep another set of lists in addition to the adjacency lists. This set of lists, called ***inverse adjacency lists***, will contain one list for each vertex. Each list will contain a node for each vertex adjacent to the vertex it represents (see Figure 6.1 0).

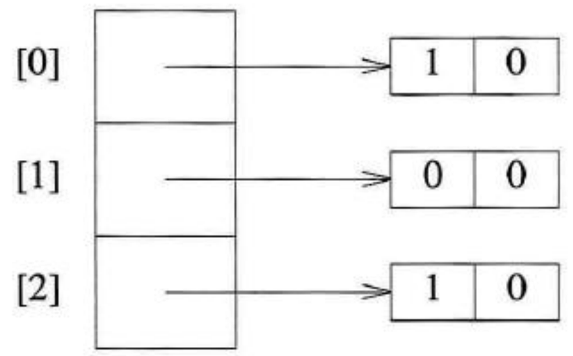
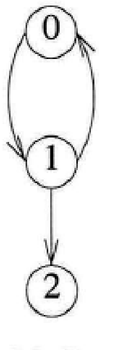


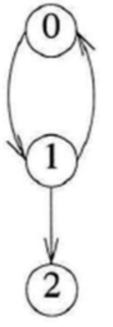
Figure 6.10: Inverse adjacency lists for G_3 (Figure 6.2(c))



(c) G_3

- Alternatively, one can adopt a simplified version of the list structure used for sparse matrix representation in Chapter 4.

4 fields : [tail, head, link for column chain, link for row chain]



(c) G_3

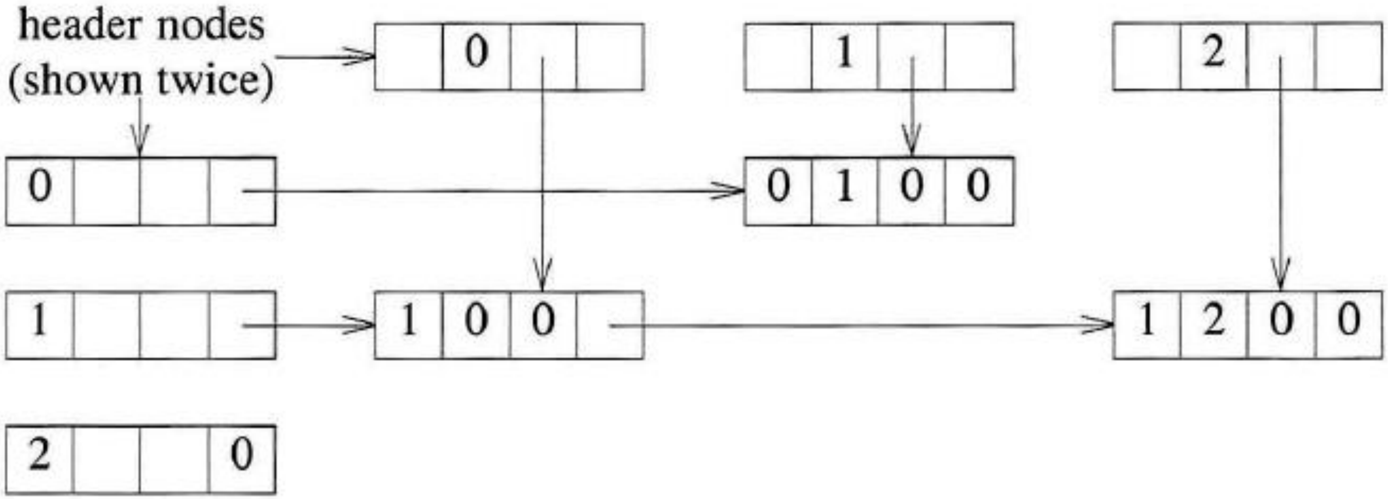
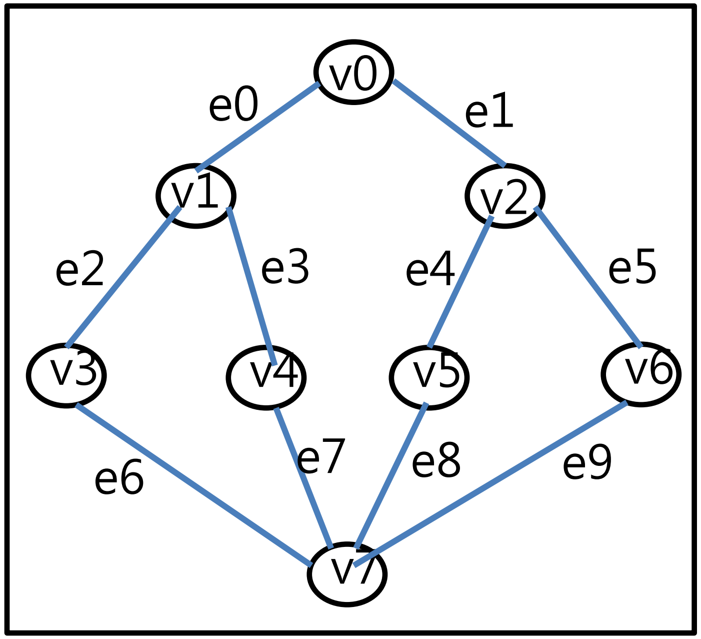


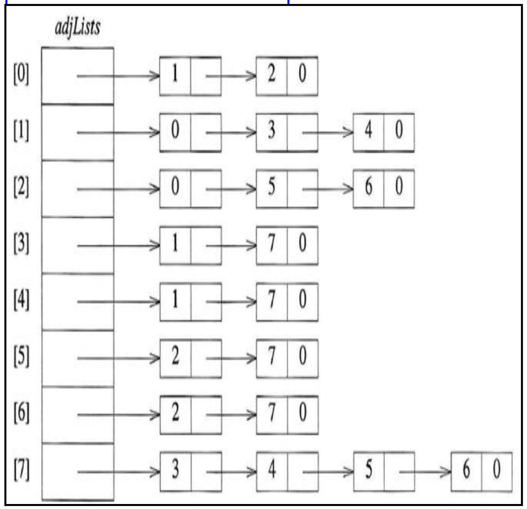
Figure 6.11: Orthogonal list representation for G_3 of Figure 6.2(c)



Adjacency Matrix

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

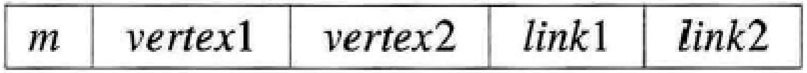
Adjacency List



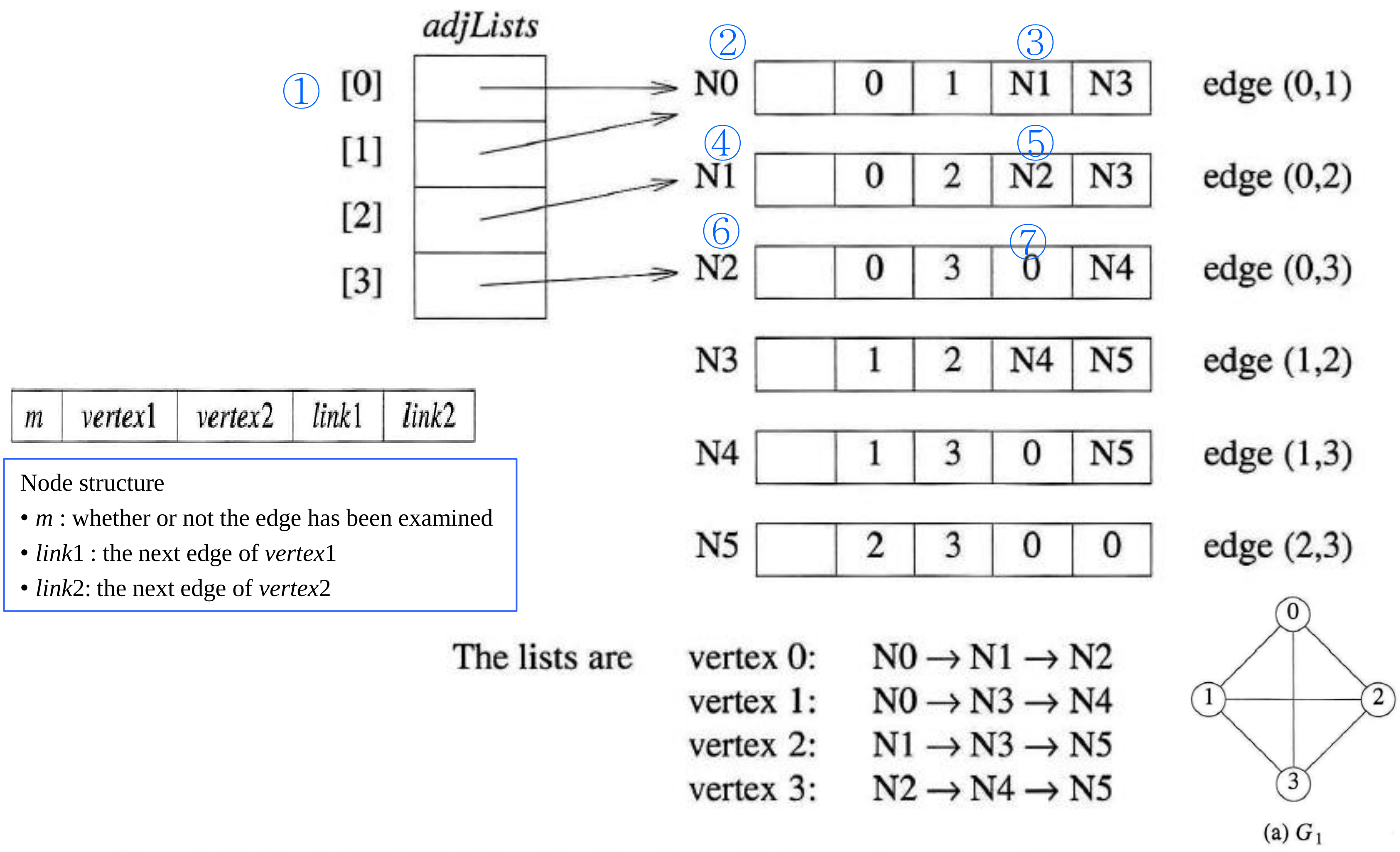
위 표현에서 edge number 정보는 표현되지 않았다.

6.1.3.3 Adjacency Multilists (생략)

- Adjacency list
  - each edge (*u*, *v*) is represented by two entries
- Multilist
  - lists in which nodes may be shared among several lists
- Adjacency multilists
  - For each edge, there will be exactly one node, but this node will be in two lists.
- Node structure
  - *m* : whether or not the edge has been examined
  - *link*1 : the next edge of *vertex*1
  - *link*2: the next edge of *vertex*2



m	*vertex*1	*vertex*2	*link*1	*link*2



N0		0	1	N1	N3	edge (0,1)
N1		0	2	N2	N3	edge (0,2)
N2		0	3	0	N4	edge (0,3)
N3		1	2	N4	N5	edge (1,2)
N4		1	3	0	N5	edge (1,3)
N5		2	3	0	0	edge (2,3)

m	*vertex*1	*vertex*2	*link*1	*link*2

Node structure
- *m* : whether or not the edge has been examined
- *link*1 : the next edge of *vertex*1
- *link*2: the next edge of *vertex*2

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)

Q1. How to find all edges incident for vertex 0 ? ① ~ ⑦

Q2. How to traversal G_1, visiting once an edge?

6.1.3.4 Weighted Edges (생략)

In many applications, the edges of a graph have weights assigned to them. These weights may represent the distance from one vertex to another or the cost of going from one vertex to an adjacent vertex. In these applications, the adjacency matrix entries $a[i][j]$ would keep this information too. When adjacency lists are used, the weight information may be kept in the list nodes by including an additional field, ***weight***. A graph with weighted edges is called a ***network***.

6.2 Elementary Graph Operations

Given an undirected graph, $G = (V, E)$, and a vertex, v, in $V(G)$ we wish to **visit all vertices in *G* that are reachable from *v***, that is, all vertices that are connected to v. We shall look at two ways of doing this: ***depth first search*** and ***breadth first search***. In our discussion of depth first search and breadth first search, we shall assume that the **linked adjacency list representation** for graphs is used. The excercises explore the use of other representations.

6.2.1 Depth First Search(깊이 우선 탐색)

We begin the search by visiting the start vertex, *v*. In this simple application, visiting consists of printing the node's vertex field. Next, we select an unvisited vertex, *w*, from *v*'s adjacency list and carry out a depth first search on *w*. We preserve our current position in *v*'s adjacency list by placing it on a **stack**. Eventually our search reaches a vertex, *u*, that has no unvisited vertices on its adjacency list. At this point, we remove a vertex from the stack and continue processing its adjacency list. Previously visited vertices are discarded; unvisited vertices are visited and placed on the stack. The search terminates when the stack is empty.

```
#define FALSE 0
#define TRUE 1
short visited[MAX_VERTICES];
void initialize_visited()
{
    int x;
    for (x=0; x<MAX_VERTICES; x++)
        visited[x] = FALSE;
}

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

Program 6.1: Depth first search

```
typedef struct node *nodePointer;
typedef struct node {
    int             vertex;
    struct node   *link;
} ;
typedef structure node node;
node *adjLists[MAX_VERTICES];
```

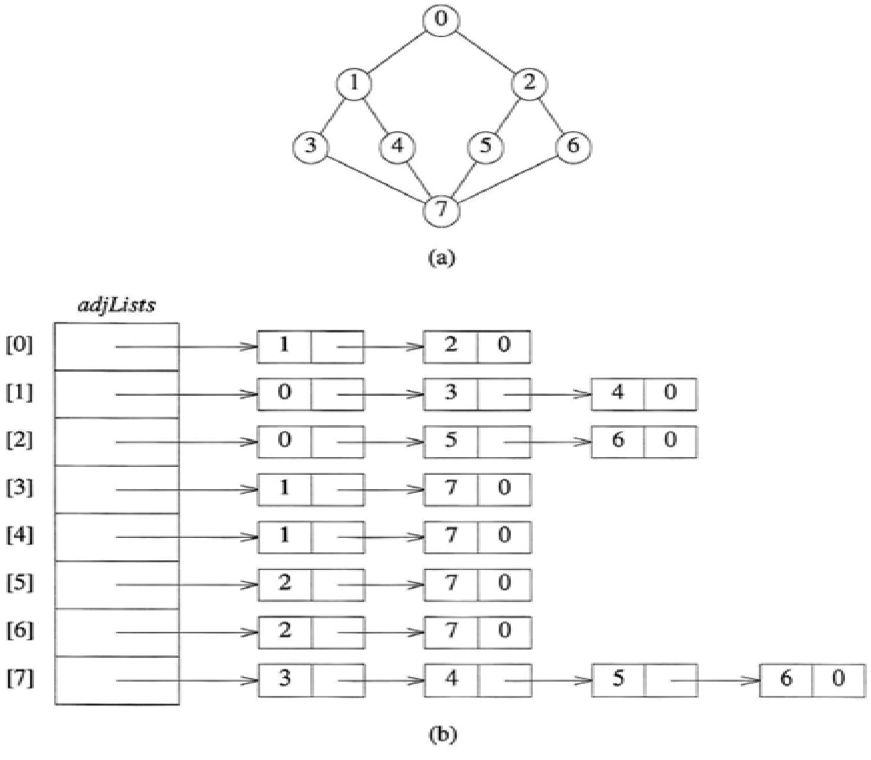


Figure 6.16: Graph *G* and its adjacency lists

- Analysis
- ➢ adjacency list : O(e)
- ➢ adjacency matrix : O(n^2)

dfs(v_0) : v_0, v_1, v_3, v_7, v_4, v_5, v_2, v_6

6.2.2 Breadth First Search(너비(폭: 幅) 우선 탐색)

Breadth first search starts at vertex *v* and marks it as visited. It then visits each of the vertices on *v*'s adjacency list. When we have visited all the vertices on *v*'s adjacency list, we visit all the unvisited vertices that are adjacent to the first vertex on *v*'s adjacency list. To implement this scheme, as we visit each vertex we place the vertex in a **queue**. When we have exhausted an adjacency list, we remove a vertex from the queue and proceed by examining each of the vertices on its adjacency list. Unvisited vertices are visited and then placed on the queue; visited vertices are ignored. We have finished the search when the queue is empty.

```
#define FALSE 0
#define TRUE 1
short visited[MAX_VERTICES];
void initialize_visited()
{
    int x;
    for (x=0; x<MAX_VERTICES; x++)
        visited[x] = FALSE;
}

void process_vertex(int v)
{
     printf("%5d",v);
     visited[v] = TRUE;
     add_queue(v);
}

/ * breath first search of a graph beginning at vertex v * /
void bfs(int v)
{
    node *search;
    initialize_queue();
    process_vertex(v);
    while (front) {
        v = delete_queue () ;
        for (search = adjLists[v]; search; search = search->link) {
            if (!visited[search->vertex])
                process_vertex(search->vertex);
        }
    }
}
```

Program 6.2: Breadth first search of a graph

```
typedef struct node *nodePointer;
typedef struct node {
    int             vertex;
    struct node   *link;
} ;
typedef structure node node;
node *adjLists[MAX_VERTICES];
```

```
typedef struct queue *queuePointer;
typedef struct queue {
    int vertex;
    queuePointer link;
} ;
typedef struct queue queue;
queue *front, *rear;
void add_queue(int);            // 작성할 것
int delete_queue();             // 작성할 것
void initialize_queue()
{
    front = rear = NULL
}
```

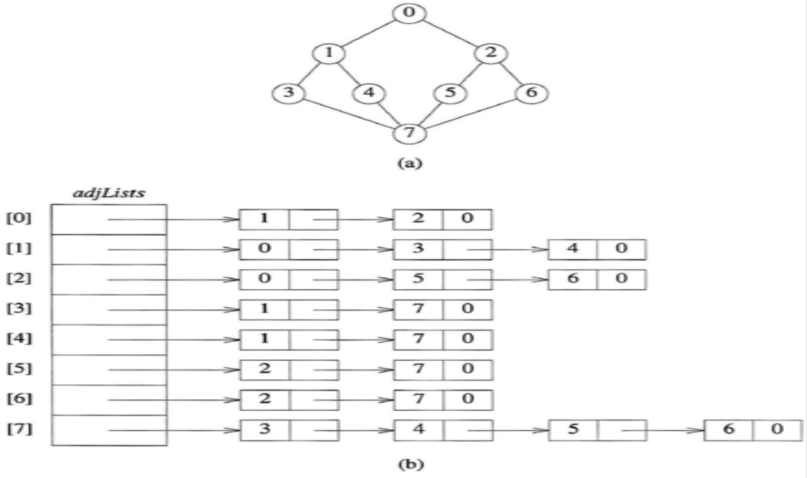


Figure 6.16: Graph *G* and its adjacency lists

- **Analysis**
  - adjacency matrix : $O(n^2)$
  - adjacency list : $O(e)$

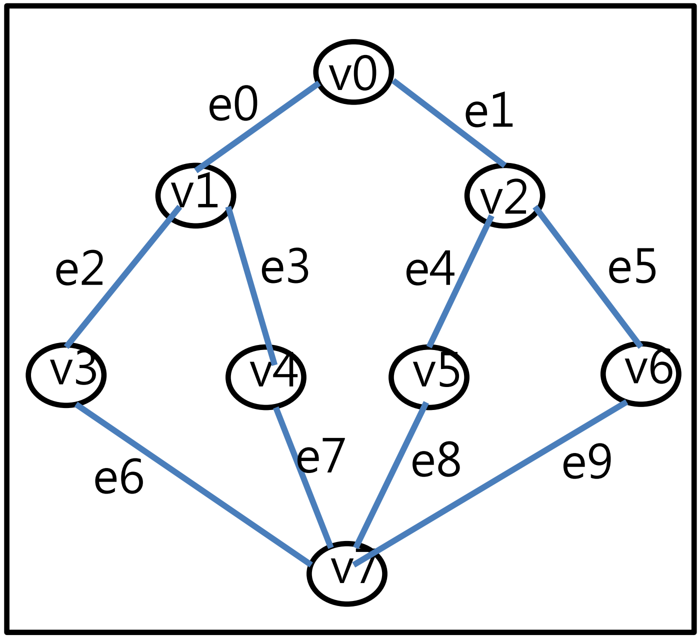
$bfs(v_0)$: $v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7$

과제 1 (6.2.1 DFS and 6.2.2 BFS)

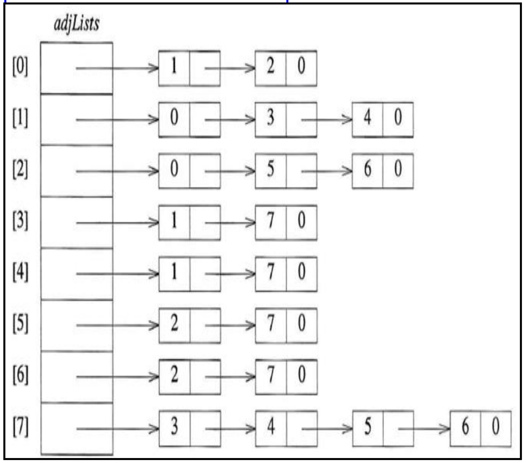
아래 작업을 수행하는 C 프로그램을 작성하시오.

1. 아래 graph의 edge 번호, vertex 번호, vertex 번호를 입력 받아 그를 Adjacency List로 표현하고 그 내용을 출력하시오.
2. 사용자로부터 starting vertex의 번호를 입력 받아 그 vertex에서 reachable한 모든 vertex들을 찾고자 한다.
   2.1 Program 6.1의 DFS는 recursive algorithm이다. 그를 이용하여 그를 수행하시오.
   2.2 Program 6.1의 DFS에 해당하는 iterative version을 구현하고 그를 이용하여 수행하시오.
   2.3 Program 6.2의 BFS는 iterative algorithm이다. 그를 이용하여 그를 수행하시오.
   2.4 Program 6.2의 BFS에 해당하는 recursive version을 구현하고 그를 이용하여 수행하시오.

3. 아래의 첨가된 10개의 vertex로 구성된 그래프에 대해 위의 1번과 2번을 수행하시오. 단, edge number는 각자가 알아서 정하면 된다.

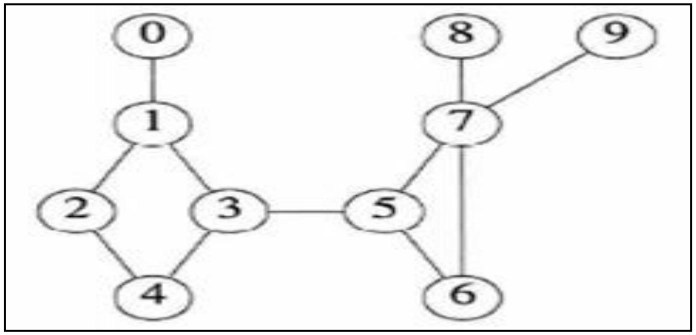


Adjacency List



위 표현에서 edge number 정보는 표현되지 않았다.

```
Graph를 입력하시오
(edge 번호 vertex 번호 vertex 번호) -1 -1 -1에 입력종료
0 0 1
1 0 2
2 1 3
3 1 4
4 2 5
5 2 6
6 3 7
7 4 7
8 5 7
9 6 7
-1 -1 -1
출력 형태: Vertex(Adjacent vertex list)
0 (1, 2)
1 (0, 3, 4)
2 (0, 5, 6)
3 (1, 7)
4 (1, 7)
5 (2, 7)
6 (2, 7)
7 (3, 4, 5, 6)
Starting vertex의 number를 입력하시오: 0
DFS(recursive version)의 결과: 0, 1, 3, 7, 4, 5, 2, 6
DFS(iterative version)의 결과: 0, 1, 3, 7, 4, 5, 2, 6
BFS(iterative version)의 결과: 0, 1, 2, 3, 4, 5, 6, 7
BFS(recursive version)의 결과: 0, 1, 2, 3, 4, 5, 6, 7
```



6.2.3 Connected Components (연결성분,連結成分)

(1) Determine whether or not an undirected graph is connected.

We can implement this operation by simply calling either *dfs*(0) or *bfs*(0) and then determining if there are any unvisited vertices.

For example, the call *dfs*(0) applied to graph G_4 of Figure 6.5 terminates without visiting vertices 4, 5, 6, and 7. Therefore, we can conclude that graph G_4 is not connected. The computing time for this operation is O($n + e$) if adjacency lists are used.

(2) List the connected components of a graph.

This is easily accomplished by making repeated calls to either *dfs*(v) or *bfs*(v) where v is an unvisited vertex. The function *connected* (Program 6.3) carries out this operation. Although we have used *dfs*, *bfs* may be used with no change in the time complexity.

```
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

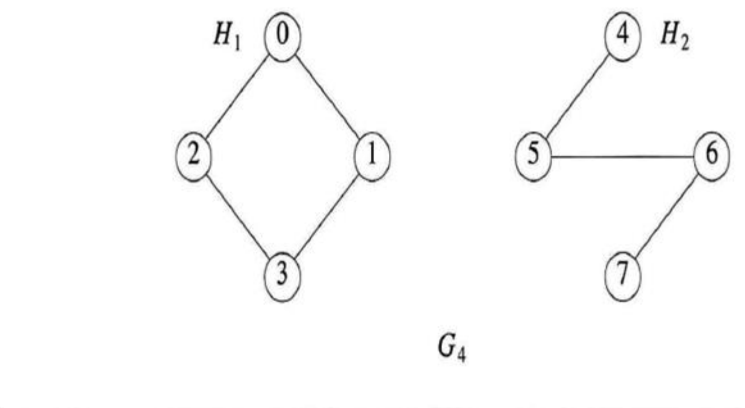


Figure 6.5: A graph with two connected components

6.2.4 Spanning Trees (신장 트리, <伸張, 펼 신 베풀 장>)

A *spanning tree* is any **tree** that consists solely of edges in *G* and that includes **all the vertices** in *G* (신장 트리는 어떤 그래프의 모든 정점을 포함하는 트리이며 그 그래프의 부분 그래프이다.).

Figure 6.17 shows a graph and three of its spanning trees.

We may use either *dfs* or *bfs* to create a spanning tree.

When *dfs* is used, the resulting spanning tree is known as ***a depth first spanning tree***. When *bfs* is used, the resulting spanning tree is called ***a breadth first spanning tree***. Figure 6.18 shows the spanning trees that result from a depth first and breadth first search starting at vertex v_0 in the graph of Figure 6.16.

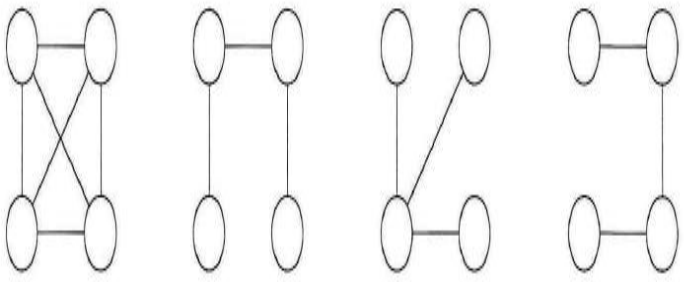
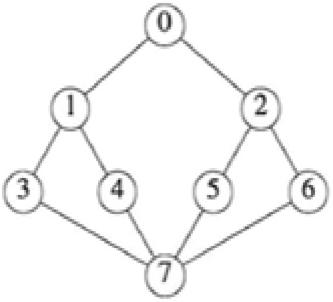
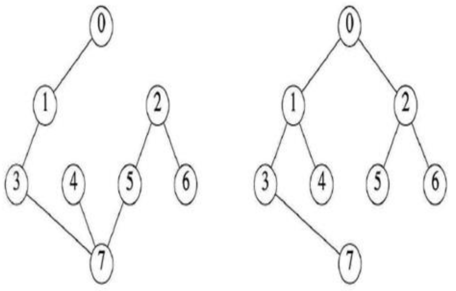


Figure 6.17: A complete graph and three of its spanning trees





(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

Properties of spanning trees.

(1) 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. For example, if we add the nontree edge (7, 6) to the dfs spanning tree of Figure 6.18(a), the resulting cycle is 7, 6, 2, 5, 7.

(2) 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 .)

6.2.5 Biconnected Components (이중연결성분,二重連結成分)

- An ***articulation point*** (단절점, 斷絶(유대나 연관관계를 끊음)點 ? 斷切(자르거나 베어서 끊음)點) is a vertex v of an undirected connected graph G such that the deletion of v, together with all edges incident on v, produces a graph, G', that has at least two connected components. For example, the connected graph of Figure 6.19 has four articulation points, vertices 1, 3, 5, and 7.

- A ***biconnected graph*** **(이중연결(二重連結) 그래프)** is a connected graph that has no articulation points. For example, the graph of Figure 6.16 is biconnected, while the graph of Figure 6.19 obviously is not.

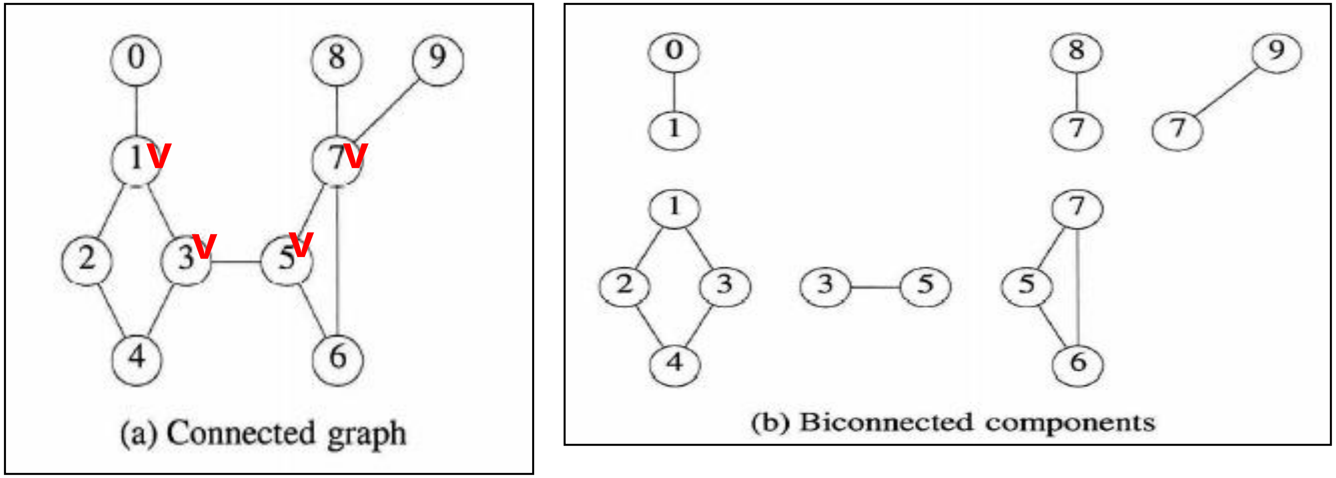


Figure 6.19: A connected graph and its biconnected components

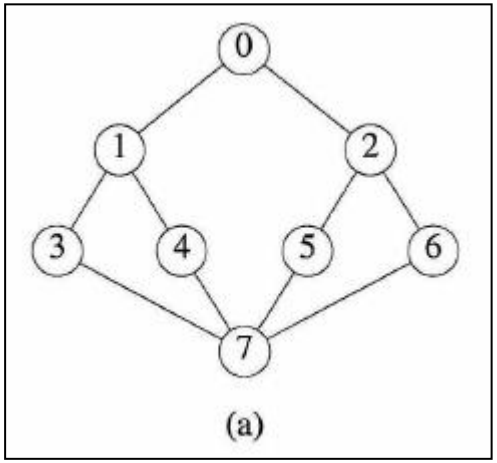
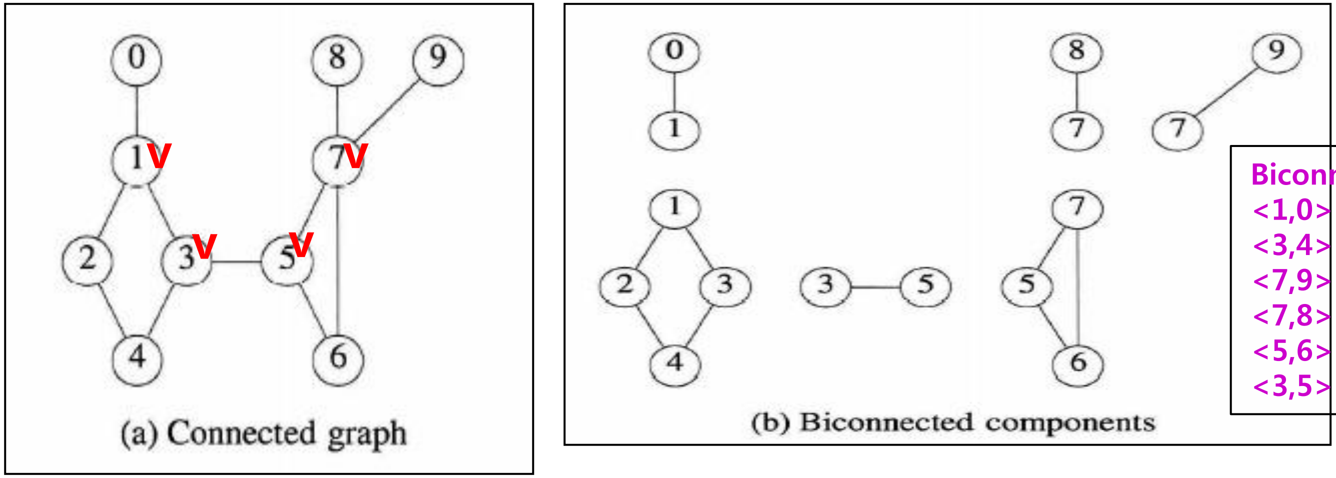


Figure 6.16: Graph *G* and its adjacency lists

A ***biconnected component*** (이중연결성분,二重連結成分) of a connected undirected graph 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. For example, the graph of Figure 6.19(a) contains the six biconnected components shown in Figure 6.19(b). The biconnected graph of Figure 6.16, however, contains just one biconnected component: the whole graph. It is easy to verify that two biconnected components of the same graph have no more than one vertex in common. This means that no edge can be in two or more biconnected components of a graph. Hence, the biconnected components of G partition the edges of G.



(a) Connected graph

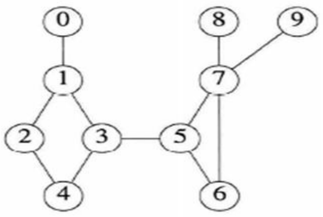
(b) Biconnected components

Biconnected Components:
<1,0>
<3,4> <4,2> <2,1> <1,3>
<7,9>
<7,8>
<5,6> <6,7> <7,5>
<3,5>

Figure 6.19: A connected graph and its biconnected components

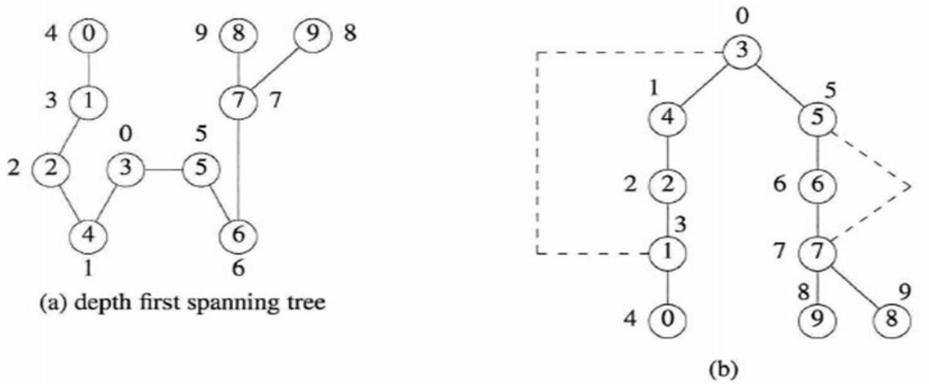
We can **find the biconnected components of a connected undirected graph, G,** by using any depth first spanning tree of G. For example, the function call dfs(3) applied to the graph of Figure 6.19(a) produces the spanning tree of Figure 6.20(a). We have redrawn the tree in Figure 6.20(b) to better reveal its tree structure. The numbers outside the vertices in either figure give the sequence in which the vertices are visited during the depth first search. We call this number the ***depth first number***, or ***dfn***, of the vertex. For example, dfn(3) = 0, dfn(0) = 4, and dfn(9) = 8. Notice that vertex 3, which is an ancestor of both vertices 0 and 9, has a lower dfn than either of these vertices. Generally, if u and v are two vertices, and u is an ancestor of v in the depth first spanning tree, then dfn(u)<dfn(v).

dfs(3) applied to the graph of Figure 6.19(a)



(a) Connected graph

Figure 6.19(a)



(a) depth first spanning tree

(b)

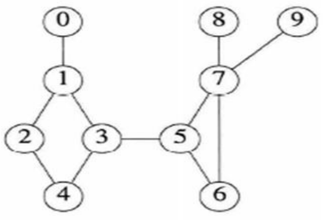
Figure 6.20: Depth first spanning tree of Figure 6.19(a)

dfn(x) : 현재의 DFS 경로와 방문순서에서 vertex x의 방문순서. 그 값은 0부터 시작한다.
low(x) : 현재의 DFS 경로와 방문순서에서 vertex x와 그의 후손들에서 back edge로 방문한 vertex x의 조상 중 가장 먼 조상의 방문순서.

The broken lines in Figure 6.20(b) represent ***nontree edges***. A nontree edge (u, v) is a ***back edge*** iff either u is an ancestor of v or v is an ancestor of u. From the definition of depth first search, it follows that all nontree edges are back edges. This means that the root of a depth first spanning tree is an articulation point iff it has at least two children (in the spanning tree. 내가 추가한 것임). In addition, any other vertex u is an articulation point iff it has at least one child *w* such that we cannot reach an ancestor of *u* using a path that consists of only *w*, descendants of *w*, and a single back edge. These observations lead us to define a value, ***low***, for each vertex of G such that ***low*(*u*)** is the lowest depth first number that we can reach from *u* using a path of descendants followed by at most one back edge:

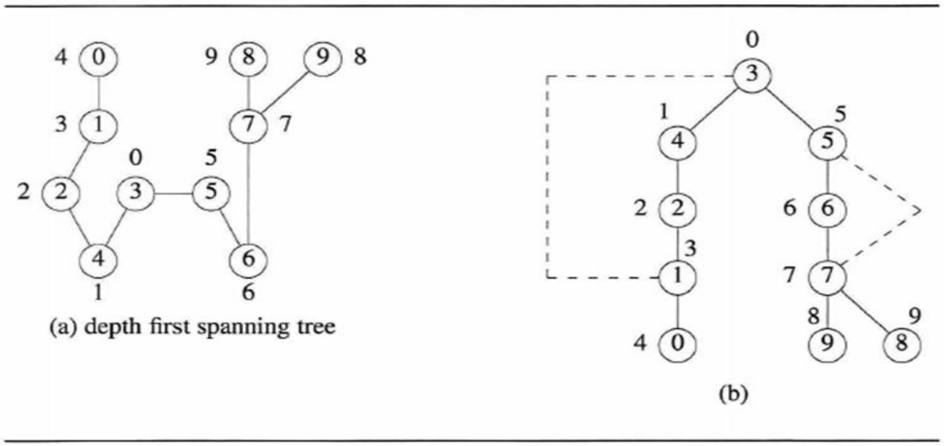
$$\begin{aligned}\textit{low}(u) = \min\{&\text{dfn}(u),\\ &\min\{\text{low}(w) \mid w \text{ is a child of } u\},\\ &\min\{\text{dfn}(w) \mid (u, w) \text{ is a back edge}\}\}\end{aligned}$$

dfs(3) applied to the graph of Figure 6.19(a)



(a) Connected graph

Figure 6.19(a)

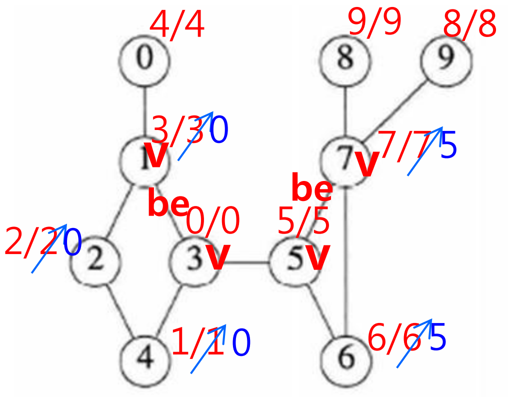


(a) depth first spanning tree

(b)

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

Therefore, we can say that *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*) ≥ *dfn*(*u*)**. Figure 6.21 shows the *dfn* and *low* values for each vertex of the spanning tree of Figure 6.20(b). From this table we can conclude that vertex 1 is an articulation point since it has a child 0 such that low(0) = 4 ≥ dfn(1) = 3. Vertex 7 is also an articulation point since low(8) = 9 ≥ dfn(7) = 7, as is vertex 5 since low(6) = 5 ≥ dfn(5) = 5. Finally, we note that the root, vertex 3, is an articulation point because it has more than one child.

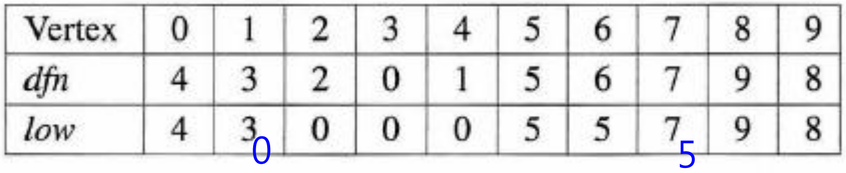


(a) Connected graph

dfn/low

be = back edge

call dfnlow(3, -1)



Vertex	0	1	2	3	4	5	6	7	8	9
dfn	4	3	2	0	1	5	6	7	9	8
low	4	3 0	0	0	0	5	5	7 5	9	8

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

We can easily modify dfs to compute dfn and low for each vertex of a connected undirected graph. The result is dfnlow (Program 6.4).

We invoke the function with the call dfnlow(x, -1), where x is the starting vertex for the depth first search. The function uses a MIN2 macro that returns the smaller of its two parameters. The results are returned as two global variables, dfn and low. We also use a global variable, num, to increment dfn and low. The function init (Program 6.5) contains the code to correctly initialize dfn, low, and num. The global declarations are:

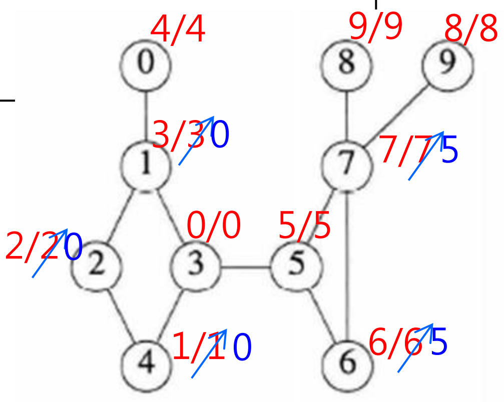
The root is a special case because it has no parent. The root is an articulation point if it has more than one child in the DFS tree. A non-root node v is an articulation point if and only if it has a subtree with no back edges pointing at ancestors of v.

```
#define MIN2 (x, y) ( (x) < (y) ? (x) : (y))
short int dfn[MAX_VERTICES];
short int low[MAX_VERTICES];
int num;

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

Program 6.4: Determining dfn and low

call dfnlow(3, -1)



(a) Connected graph

dfn/low

call dfnlow(x, -1), where x is the starting vertex for the depth first search.

```
void init(void)
{
    int i;
    for (i = 0; i < n; i++) {
        visited[i] = FALSE;
        dfn[i] = low[i] = -1;
    }
    num = 0;
}
```

Program 6.5: Initialization of dfn and low

```
#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 = adjLists[v]; w; w = w->link) {
        if ( !visited[w->vertex])
            dfs (w->vertex);
    }
}
```

Program 6.1: Depth first search

We can partition the edges of the connected graph into their biconnected components by adding some code to dfnlow. We know that low[w] has been computed following the return from the function call dfnlow(w, u). If low[w] ≥ dfn[u], then we have identified a new biconnected component. We can output all edges in a biconnected component if we use a stack to save the edges when we first encounter them. The function bicon (Program 6.6) contains the code. The same initialization function (Program 6.5) is used. The function call is bicon (x, -1), where x is the root of the spanning tree. Note that the parameters for the stack operations push and pop are slightly different from those used in Chapter 3.

The function call is bicon (x, -1), where x is the root of the spanning tree.

```
void bicon(int me, int parent)
{
    /* 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 child,x,y;
    dfn[me] = low[me] = num++;
    for (ptr = graph[me]; ptr; ptr = ptr->link) {
        child = ptr->vertex;
        if (parent != child && dfn[child] < dfn[me]) {// 이 부분이 교재의 오류
            // either an edge to an unvisited vertex or a back-edge
            push(me,child); /*add edge to stack*/
        }
        if (dfn[child] <0) { /* child has not been visited */
            bicon (child, me);
            low[me] = MIN2(low[me],low[child]);
            if (low[child] >= dfn[me]) {
                printf("New biconnected component: ");
                do { /* delete edge from stack */
                    pop(&x, &y);
                    printf(" <%d,%d>",x,y);
                } while (! ((x == me) && (y == child)));
                printf("\n");
            }
        } else if (child != parent) // a back-edge is found
            low[me] = MIN2(low[me],dfn[child]);
    }
}
```

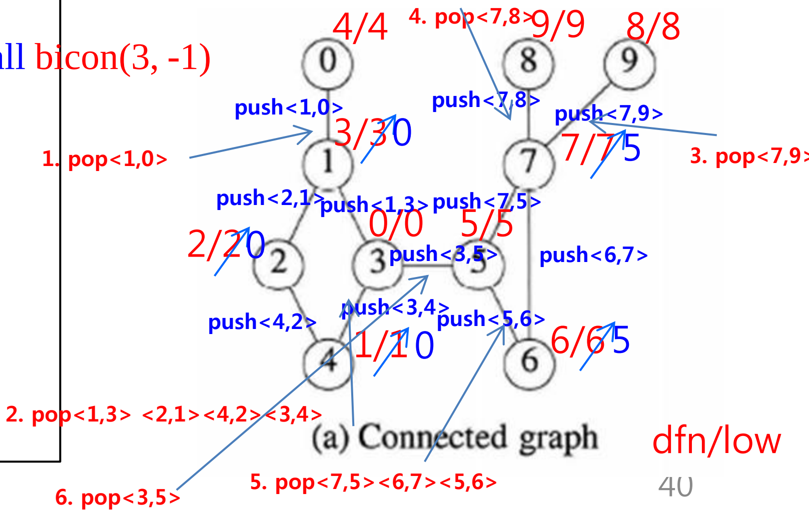
Program 6.6: Biconnected components of a graph

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

Program 6.4: Determining dfn and low

```
New biconnected component: <1,0>
New biconnected component: <1,3> <2,1><4,2><3,4>
New biconnected component: <7,9>
New biconnected component: <7,8>
New biconnected component: <7,5><6,7><5,6>
New biconnected component: <3,5>
```

call bicon(3, -1)



(a) Connected graph

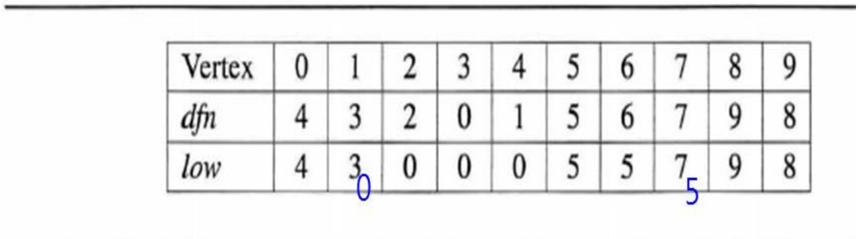
Analysis of bicon: The function bicon assumes that the connected graph has at least two vertices. Technically, a graph with one vertex and no edges is biconnected, but, our implementation does not handle this special case. The complexity of bicon is $O(n + e)$. We leave the proof of its correctness as an exercise. □

EXERCISE (6.2.5 Biconnected Components)

1. Figure 6.19(a)의 connected graph를 keyboard로 입력하여 그를 대상으로 Adjacency List를 만든 후 Program 6.4: Determining dfn and low를 Figure 6.19(a)의 connected graph를 대상으로 dfnlow(3, -1)을 호출하여 Figure 6.21의 표를 만들고 그 결과가 비교하시오.[2점]

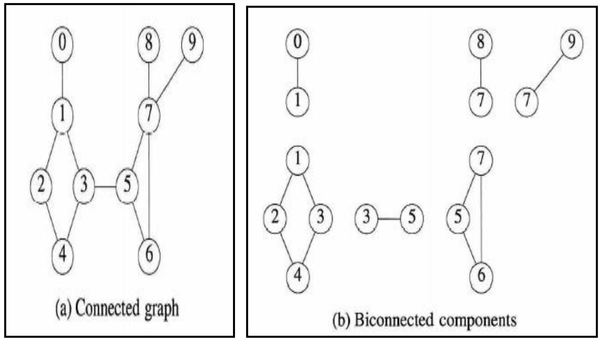
   Program 6.6: Biconnected components of a graph는 심각한 오류를 내포하고 있다. 그 오류를 찾아 정확하게 고치시오. (이미 내가 고쳐서 강의자료로 사용했음)
2. 수정한 Program 6.6을 Figure 6.19(a)의 connected graph를 대상으로 수행하여, Figure 6.20(a)와 같은 depth first spanning tree를 형성하여, 그 출력을 구하시오.[2점]
3. 우측 하단에 두 개의 붉은 색 edge가 Figure 6.19(a)의 connected graph에 추가되었다. 그 그래프를 대상으로 위의 1번과 2번을 수행하시오. 이 경우, 1번과 2번 각 2점씩이다.

```
New biconnected component: <1,0>
New biconnected component: <1,3> <2,1><4,2><3,4>
New biconnected component: <7,9>
New biconnected component: <7,8>
New biconnected component: <7,5><6,7><5,6>
New biconnected component: <3,5>
```



Vertex	0	1	2	3	4	5	6	7	8	9
dfn	4	3	2	0	1	5	6	7	9	8
low	4	3 0	0	0	0	5	5	7 5	9	8

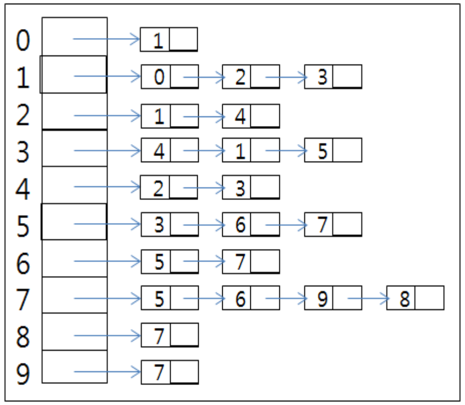
Figure 6.21: *dfn* and *low* values for *dfs* spanning tree with *root* = 3



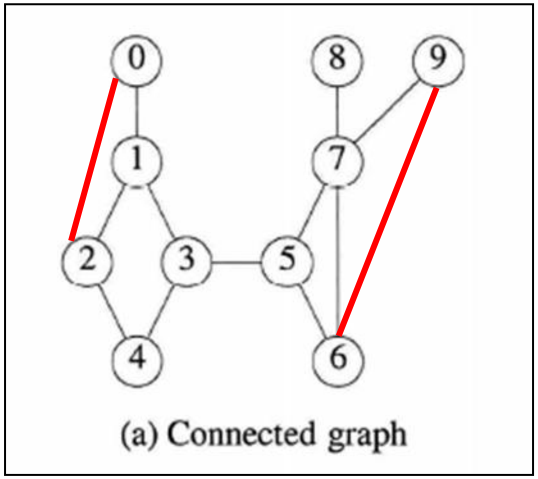
(a) Connected graph

(b) Biconnected components

Figure 6.19



0 → 1
1 → 0 → 2 → 3
2 → 1 → 4
3 → 4 → 1 → 5
4 → 2 → 3
5 → 3 → 6 → 7
6 → 5 → 7
7 → 5 → 6 → 9 → 8
8 → 7
9 → 7



(a) Connected graph

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.

greedy method(욕심쟁이 방법) : 주어진 조건 내에서 결정을 해야 할 때 마다, 그 순간 가장 좋다고 판단되는 것을 해답으로 선택함으로써 최종적인 해를 탐색하는 방법이다.

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. The greedy method can be applied to a wide variety of programming problems. Typically, the selection of an item at each stage is based on either a least cost or a highest profit criterion. A ***feasible solution*** is one which works within the constraints specified by the problem.

For spanning trees, we use **a least cost criterion**. Our solution must satisfy the following constraints:

(1) we must use only edges within the graph

(2) we must use exactly *n* - 1 edges

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

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*.

Example 6.3: We will construct a minimum cost spanning tree of the graph of Figure 6.22(a). Figure 6.23 shows the order in which the edges are considered for inclusion, as well as the result and the changes (if any) in the spanning tree.

The cost of the spanning tree is 99.

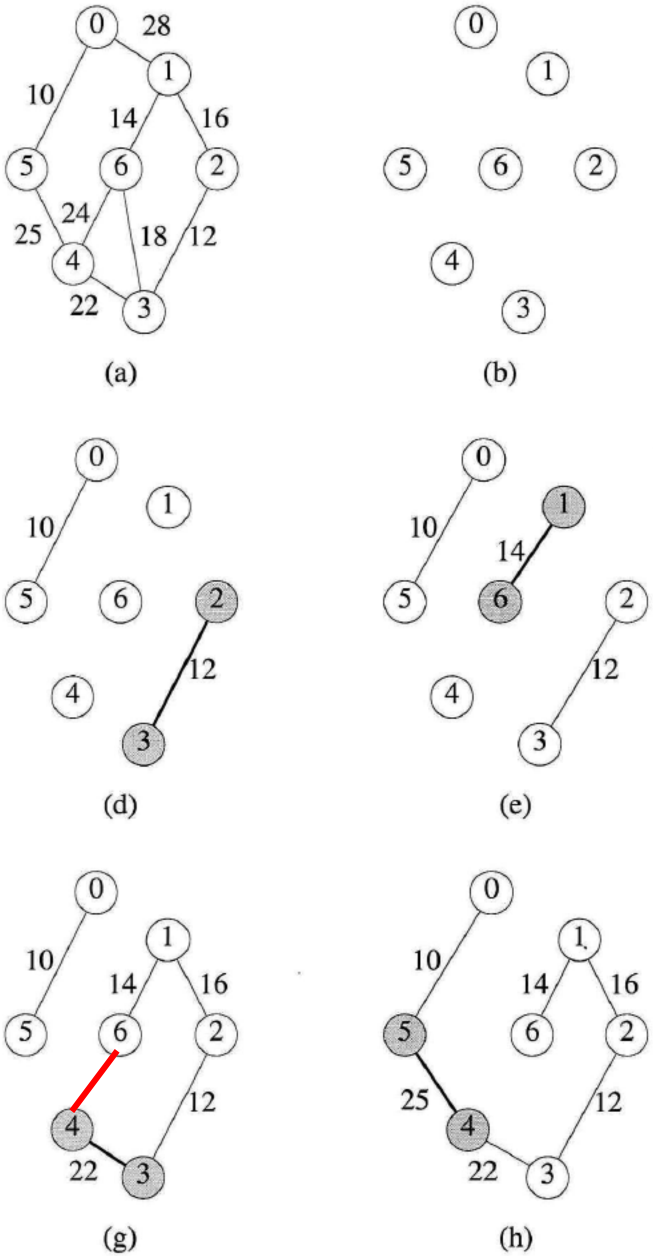
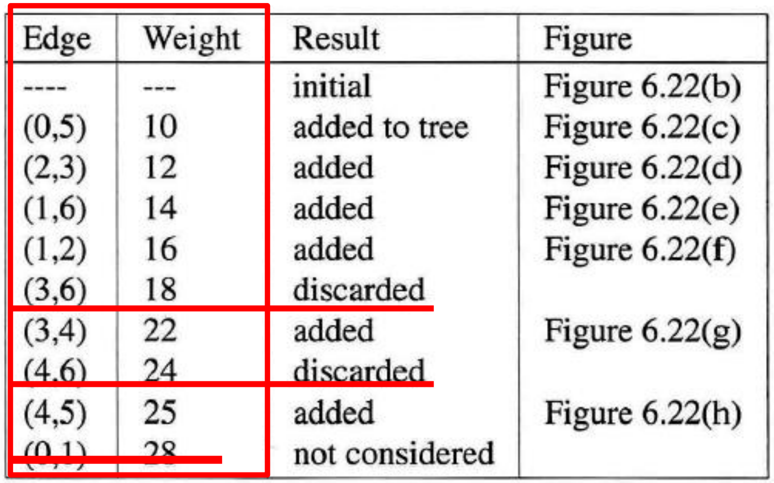


Figure 6.22: Stages in Kruskal's algorithm



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	

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

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

Program 6.7: Kruskal's algorithm

Theorem 6.1: Let *G* be any undirected, connected graph. Kruskal's algorithm generates a minimum-cost spanning tree.

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_EDGE* such that *T_EDGE* U {(*u*, *v*)} is also a tree**. We repeat this edge addition step until *T_EDGE* 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_EDGE*. Program 6.8 contains a formal description of Prim's algorithm. *T_EDGE* is the set of tree edges, and *T_VERTEX* is the set of tree vertices, that is, vertices that are currently in the tree. Figure 6.24 shows the progress of Prim's algorithm on the graph of Figure 6.22(a).

```
Minimum_Cost_Spanning_Tree()
{
    T_EDGE = {};
    T_VERTEX= {0}; /* start with vertex 0 and no edges */
    while (T_EDGE contains fewer than n-1 edges) {
        let (u, v) be a least cost edge such that u ϵ T_VERTEX and v is not in T_VERTEX;
        if (there is no such edge)  break;
        add v to T_VERTEX;
        add (u, v) to T_EDGE;
    }
    if (T_EDGE 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 *T_VERTEX* has a companion vertex, *near*(*v*), such that *near*(*v*) ϵ *T_VERTEX* and *cost*(*near*(*v*), *v*) is minimum over all such choices for *near*(*v*). (We assume that *cost*(*v*, *w*) = ∞ if (*v*, *w*) ∉ *E*). At each stage we select *v* so that *cost*(*near*(*v*), *v*) is minimum and *v* ∉ *T_VERTEX*. Using this strategy we can implement Prim's algorithm in **O(n^2)**, where *n* is the number of vertices in *G*. Asymptotically faster implementations are also possible. One of these results from the use of **Fibonacci heaps** which we examine in Chapter 9.

Vertex V_0에서 시작.

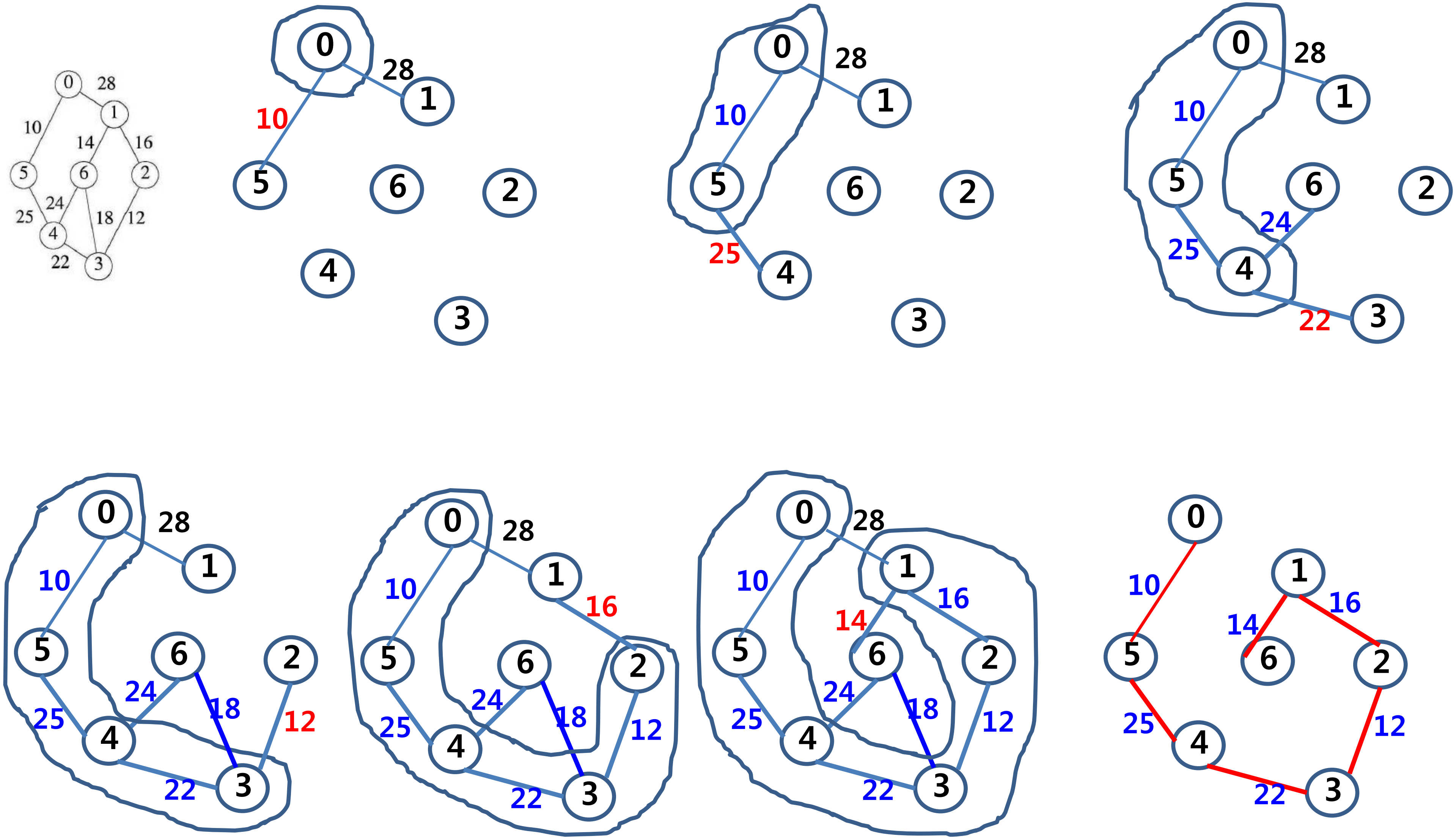


Figure 6.24: Stages in Prim's algorithm

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.

Figure 6.25 shows Sollin's algorithm applied to the graph of Figure 6.22(a). The initial configuration of zero selected edges is the same as that shown in Figure 6.22(b). Each tree in this forest is a single vertex. At the next stage, we select edges for each of the vertices. The edges selected are (0, 5), (1, 6), (2, 3), (3, 2), (4, 3), (5, 0), (6, 1). After eliminating the duplicate edges, we are left with edges (0, 5), (1, 6), (2, 3), and (4, 3). We add these edges to the set of selected edges, thereby producing the forest of Figure 6.25(a). In the next stage, the tree with vertex set {0, 5} selects edge (5, 4), and the two remaining trees select edge (1, 2). After these two edges are added, the spanning tree is complete, as shown in Figure 6.25(b). We leave the development of Sollin's algorithm into a C function and its correctness proof as exercises.

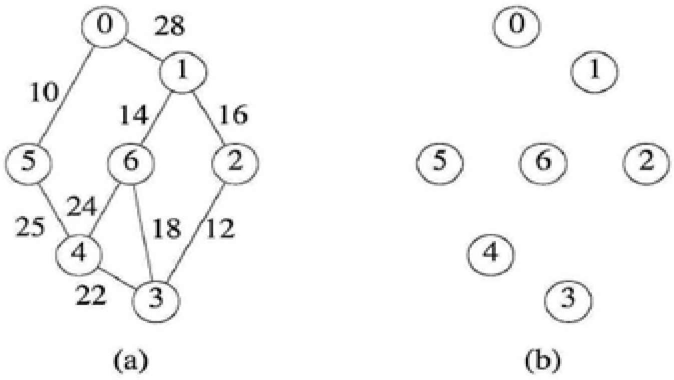


Figure 6.22

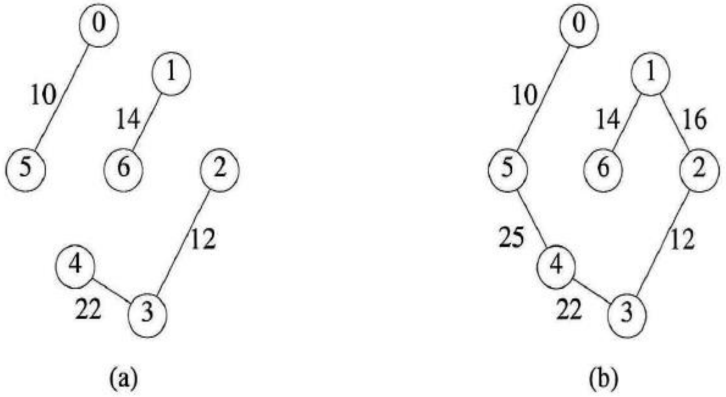


Figure 6.25: Stages in Sollin's algorithm

6.4 SHORTEST PATHS AND TRANSITIVE CLOSURE

MapQuest, Google Maps, Yahoo! Maps, and MapNation are a few of the many Web systems that **find a path between any two specified locations in the country**. **Path finding systems** generally use a graph to represent the highway system of a state or a country. In this graph, the vertices represent cities and the edges represent sections of the highway. Each edge has a weight representing the distance between the two cities connected by the edge. Alternatively, the weight could be an estimate of the time it takes to travel between the two cities. A motorist wishing to drive from city *A* to city *B* would be interested in answers to the following questions:

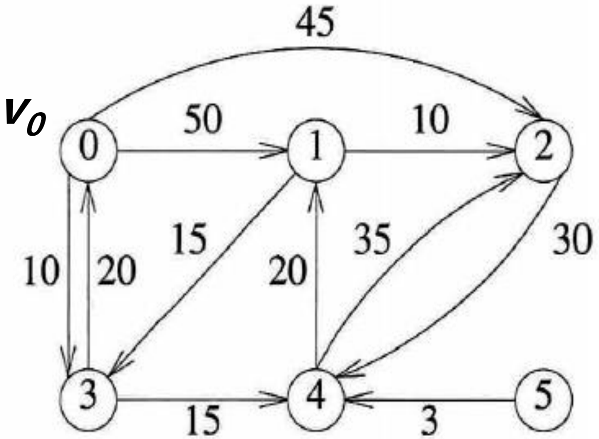
(1) Is there a path from *A* to *B*?

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

The problems defined by (1) and (2) above are special cases of the path problems we shall be studying in this section. An ***edge weight*** is also referred to as an ***edge length*** or ***edge cost***. We shall use the terms weight, cost, and length interchangeably. The length (cost, weight) of a path is now defined to be the sum of the lengths (costs, weights) of the edges on that path, rather than the number of edges. The starting vertex of the path will be referred to as the ***source*** and the last vertex the ***destination***. The graphs will be digraphs to allow for one-way streets.

6.4.1 Single Source/All Destinations: Nonnegative Edge Costs

In this problem we are given a directed graph, $G = (V, E)$, a weighting function, $w(e)$, $w(e) > 0$, for the edges of G, and a source vertex, v_0. We wish to determine a shortest path from v_0 to each of the remaining vertices of G. As an example, consider the graph of Figure 6.26(a). If v_0 is **the source vertex**, then the shortest path from v_0 to v_1 is v_0, v_3, v_4, v_1. The length of this path is 10 + 15 + 20 = 45. Although there are three edges on this path, it is shorter than the path $v_0\, v_1$, which has a length of 50. Figure 6.26(b) lists the shortest paths from v_0 to v_1, v_2, v_3, and v_4 in nondecreasing order of path length. There is no path from v_0 to v_5.



(a) Graph

	Path	*Length*
1)	0, 3	10
2)	0, 3, 4	25
3)	0, 3, 4, 1	45
4)	0, 2	45

(b) Shortest paths from 0

Figure 6.26: Graph and shortest paths from vertex 0 to all destinations

We may use **a greedy algorithm** to generate the shortest paths in the order indicated in Figure 6.26(b). Let S denote the set of vertices, including v_0, whose shortest paths have been found. For w not in S, let ***distance*[w]** be the length of the shortest path starting from v_0, going through vertices only in S, and ending in w. Generating the paths in nondecreasing order of length leads to the following observations:

(1) If the next shortest path is to vertex u, then the path from v_0 to u goes through only those vertices that are in S. To prove this we must show that all intermediate vertices on the shortest path from v_0 to u are already in S. Assume that there is a vertex w on this path that is not in S. Then, the path from v_0 to u also contains a path from v_0 to w which has a length that is less than the length of the path from v_0 to u. Since we assume that the shortest paths are generated in nondecreasing order of path length, we must have previously generated the path from v_0 to w. This is obviously a contradiction. Therefore, there cannot be any intermediate vertex that is not in S.

(2) Vertex u is chosen so that it has the minimum distance, *distance*[u], among all the vertices not in S. This follows from the definition of *distance* and observation (1). If there are several vertices not in S with the same distance, then we may select any one of them.

(3) Once we have selected u and generated the shortest path from v_0 to u, u becomes a member of S. Adding u to S can change the distance of shortest paths starting at v_0, going through vertices only in S, and ending at a vertex, w, that is not currently in S. If the distance changes, we have found a shorter such path from v_0 to w. This path goes through u. The intermediate vertices on this path are in S and its subpath from u to w can be chosen so as to have no intermediate vertices. The length of the shorter path is *distance* [u] + *length* (<u, w >).

We attribute these observations, along with the algorithm to determine the shortest paths from v_0 to all other vertices in G to **Edsger Dijkstra**. To implement **Dijkstra's algorithm**, we assume that the n vertices are numbered from 0 to n - 1. We maintain the set S as an array, *found*, with *found*[*i*] = FALSE if vertex *i* is not in S and *found*[*i*] = TRUE if vertex *i* is in S. We represent the graph by its cost **adjacency matrix**, with *cost*[*i*][*j*] being the weight of edge <*i*, *j*>. If the edge <*i*, *j*> is not in G, we set *cost*[*i*][*j*] to some large number. The choice of this number is arbitrary, although we make two stipulations regarding its value:

(1) The number must be larger than any of the values in the cost matrix.

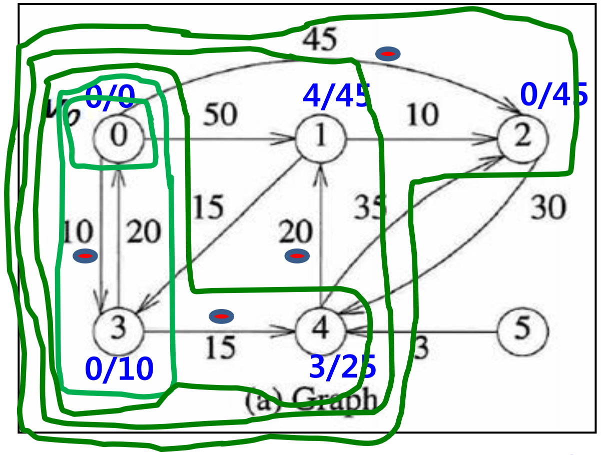
(2) The number must be chosen so that the statement *distance*[*u*] + *cost*[*u*][*w*] does not produce an overflow into the sign bit.

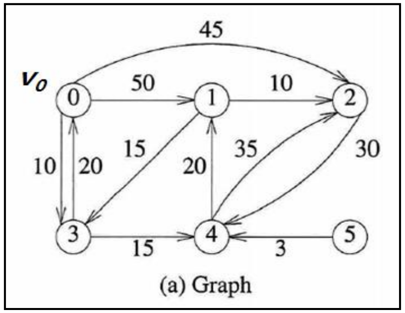
Restriction (2) makes *INT_MAX* (defined in <*limits.h*>) a poor choice for nonexistent edges. For $i = j$, we may set *cost*[*i*][*j*] to any nonnegative number without affecting the outcome. For the digraph of Figure 6.26(a), we may set the cost of a nonexistent edge with $i \neq j$ to 1000, for example. The function *shortestPath* (Program 6.9) contains our implementation of Dijkstra' s algorithm. This function uses *choose* (Program 6.10) to return a vertex, *u*, such that *u* has the minimum distance from the start vertex, *v*.

```
int cost[MAX_VERTICES][MAX_VERTICES]; // cost is the adjacency matrix
int distance[MAX_VERTICES];       // distance[x] represents the distance of the shortest path from vertex start_vertex to x
int parent[MAX_VERTICES];
short found[MAX_VERTICES]; //  found[x] is 0 if the shortest path from vertex start_vertex to x has not been found and a 1 if it has
int total_vertex_count;

void shortestPath(int start_vertex)
{
    int x, new_vertex;
    for (x = 0; x < total_vertex_count; x++) {
        found[x] = FALSE;
        distance[x] = cost[start_vertex][x]; parent[x] = start_vertex;
    }
    found[start_vertex] = TRUE;
    distance[start_vertex] = 0;
    for (x = 0; x < total_vertex_count-2; x++) {
        new_vertex = choose_min_distance_vertex();
        found[new_vertex] = TRUE;
        update_distance_parent(new_vertex);
    }
}
```

Program 6.9: Single source shortest paths





(a) Graph

cost

	0	1	2	3	4	5
0	0	50	45	10	-	-
1	-	0	10	15	-	-
2	-	-	0	-	30	-
3	20	-	-	0	15	-
4	-	20	35	-	0	-
5	-	-	-	-	3	-

- means INF

(#define INF INT_MAX)

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

Program 6.10: Choosing the least cost edge

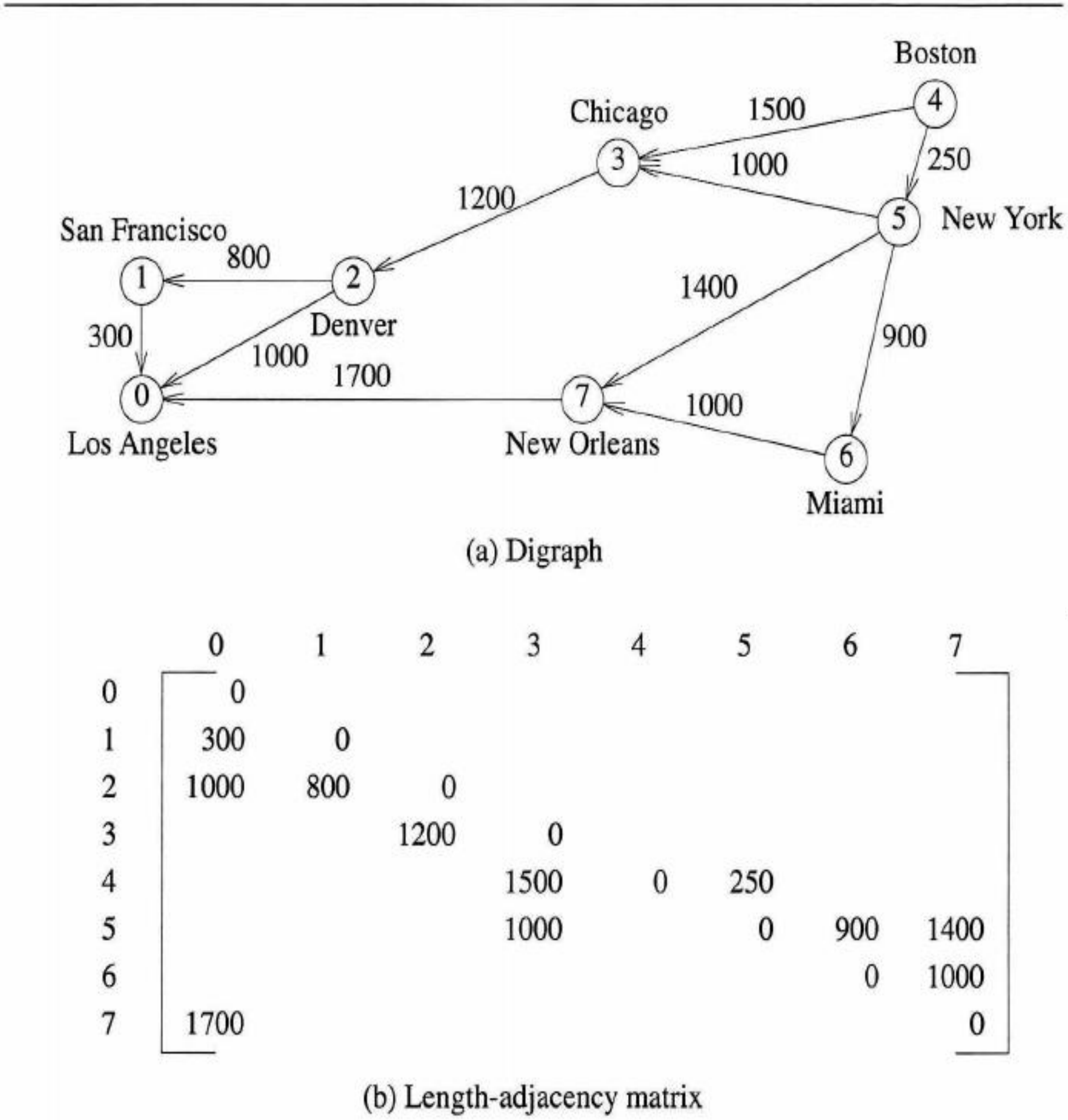
```
int update_distance_parent(int new_vertex)
{
    int x;
    int new_distance;

    for (x = 0; x < total_vertex_count; x++) {
        if (!found[x]) {
            new_distance=distance[new_vertex]+cost[new_vertex][x];
            if (new_distance < distance[x]) {
                distance[x] = new_distance;
                parent[x] = new_vertex;
            }
        }
    }
}
```

parent/distance

0	1	2	3	4	5
0/0	0/50	0/45	0/10	0/-	0/-
0/0	0/50	0/45	0/10	3/25	0/-
0/0	4/45	0/45	0/10	3/25	0/-
0/0	4/45	0/45	0/10	3/25	0/-
0/0	4/45	0/45	0/10	3/25	0/-

Example 6.4: Consider the eight-vertex **digraph** of Figure 6.27(a) with **length adjacency matrix** as in Figure 6.27(b). Suppose that the **source** vertex is Boston. The values of *dist* and the vertex *u* selected in each iteration of the outer **for** loop of Program 6.9 are shown in Figure 6.28. We use oo to denote the value LARGE. Note that the algorithm terminates after only 6 iterations of the for loop. By the definition of *dist*, the distance of the last vertex, in this case Los Angeles, is correct, as the shortest path from Boston to Los Angeles can go through only the remaining six vertices. □

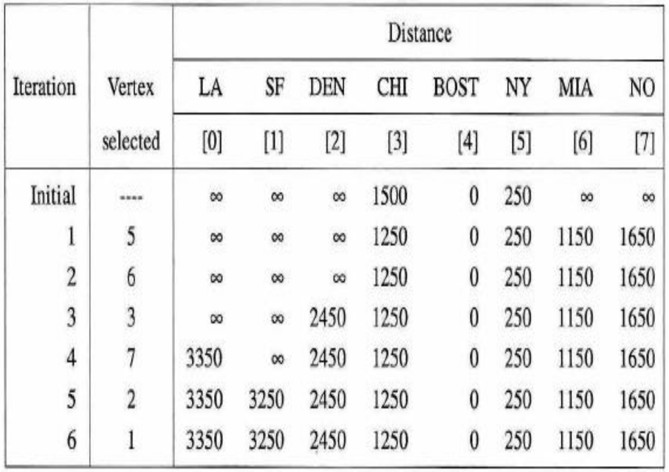


(a) Digraph

$$
\begin{array}{c|cccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
0 & 0 & & & & & & & \\
1 & 300 & 0 & & & & & & \\
2 & 1000 & 800 & 0 & & & & & \\
3 & & & 1200 & 0 & & & & \\
4 & & & & 1500 & 0 & 250 & & \\
5 & & & & 1000 & & 0 & 900 & 1400 \\
6 & & & & & & & 0 & 1000 \\
7 & 1700 & & & & & & & 0
\end{array}
$$

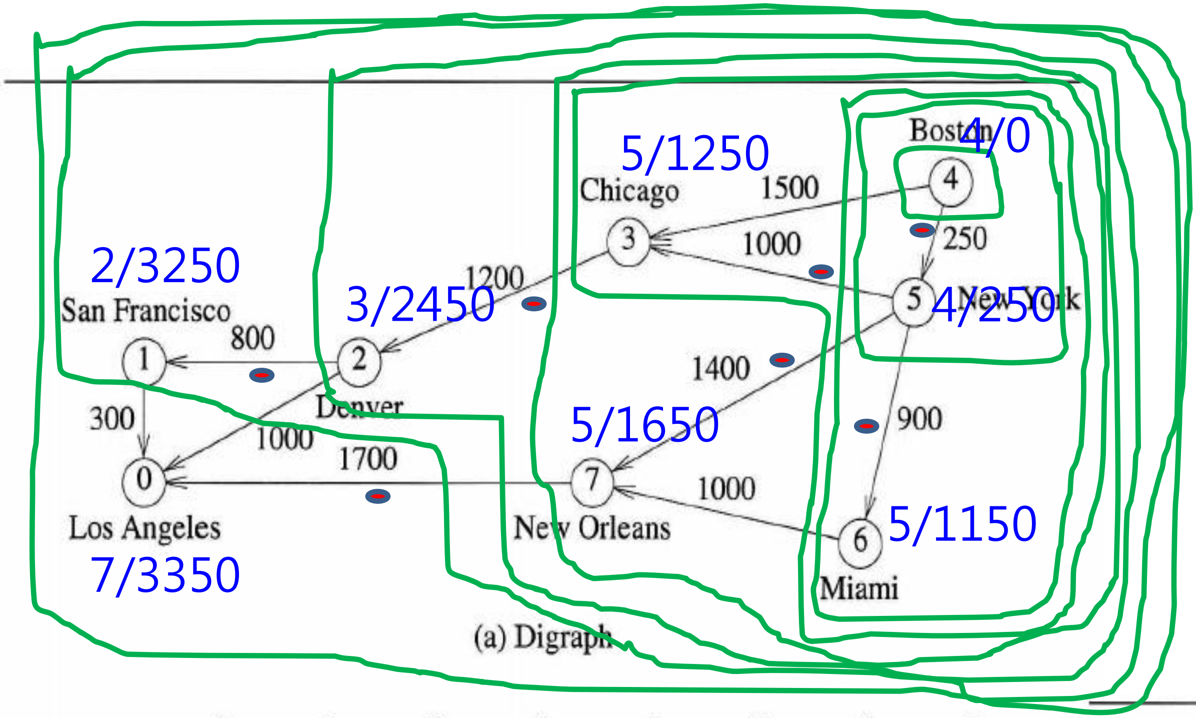
(b) Length-adjacency matrix

Figure 6.27: Digraph for Example 6.4



Iteration	Vertex	Distance							
		LA	SF	DEN	CHI	BOST	NY	MIA	NO
	selected	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
Initial	----	∞	∞	∞	1500	0	250	∞	∞
1	5	∞	∞	∞	1250	0	250	1150	1650
2	6	∞	∞	∞	1250	0	250	1150	1650
3	3	∞	∞	2450	1250	0	250	1150	1650
4	7	3350	∞	2450	1250	0	250	1150	1650
5	2	3350	3250	2450	1250	0	250	1150	1650
6	1	3350	3250	2450	1250	0	250	1150	1650

Figure 6.28: Action of *shortestPath* on digraph of Figure 6.27



(a) Digraph

$$
\begin{array}{c|cccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
0 & 0 & & & & & & & \\
1 & 300 & 0 & & & & & & \\
2 & 1000 & 800 & 0 & & & & & \\
3 & & & 1200 & 0 & & & & \\
4 & & & & 1500 & 0 & 250 & & \\
5 & & & & 1000 & & 0 & 900 & 1400 \\
6 & & & & & & & 0 & 1000 \\
7 & 1700 & & & & & & & 0
\end{array}
$$

(b) Length-adjacency matrix

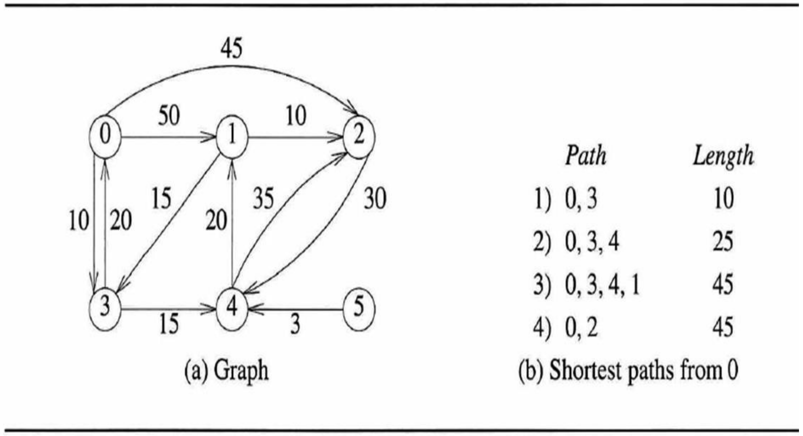
Figure 6.27: Digraph for Example 6.4

Iteration	Vertex	Distance							
		LA	SF	DEN	CHI	BOST	NY	MIA	NO
	selected	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
Initial	----	∞	∞	∞	1500	0	250	∞	∞
1	5	∞	∞	∞	1250	0	250	1150	1650
2	6	∞	∞	∞	1250	0	250	1150	1650
3	3	∞	∞	2450	1250	0	250	1150	1650
4	7	3350	∞	2450	1250	0	250	1150	1650
5	2	3350	3250	2450	1250	0	250	1150	1650
6	1	3350	3250	2450	1250	0	250	1150	1650

Figure 6.28: Action of *shortestPath* on digraph of Figure 6.27

EXERCISE 3 (6.4 Shortest Paths with/without negative edge lengths)

Modify *shortestpath* (Program 6.9) (to obtain single source all destination shortest paths with positive edge lengths) so that it obtains the shortest paths, in addition to the lengths of these paths. 🡐 source = vertex 0. For the graph on Figure 6.26(a), show all the shortest paths like the one in Figure 6.26(b).

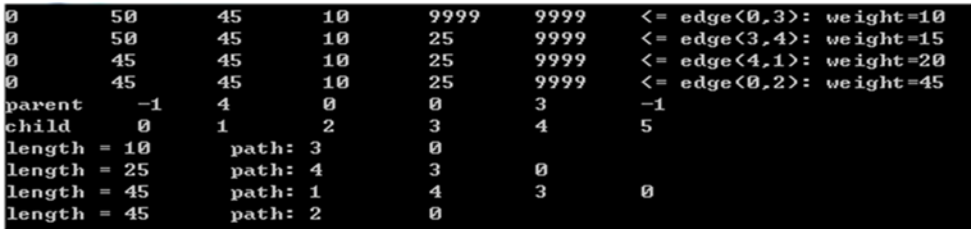


(a) Graph

Path	*Length*
1) 0, 3	10
2) 0, 3, 4	25
3) 0, 3, 4, 1	45
4) 0, 2	45

(b) Shortest paths from 0

Figure 6.26: Graph and shortest paths from vertex 0 to all destinations



```
0        50       45       10       9999     9999     <= edge<0,3>: weight=10
0        50       45       10       25       9999     <= edge<3,4>: weight=15
0        45       45       10       25       9999     <= edge<4,1>: weight=20
0        45       45       10       25       9999     <= edge<0,2>: weight=45
parent     -1     4        0        0        3        -1
child       0     1        2        3        4        5
length = 10         path: 3         0
length = 25         path: 4         3        0
length = 45         path: 1         4        3        0
length = 45         path: 2         0
```