

Chapter 6 GRAPHS

Data Structures Lecture Note
Prof. Jihoon Yang
Data Mining Research Laboratory

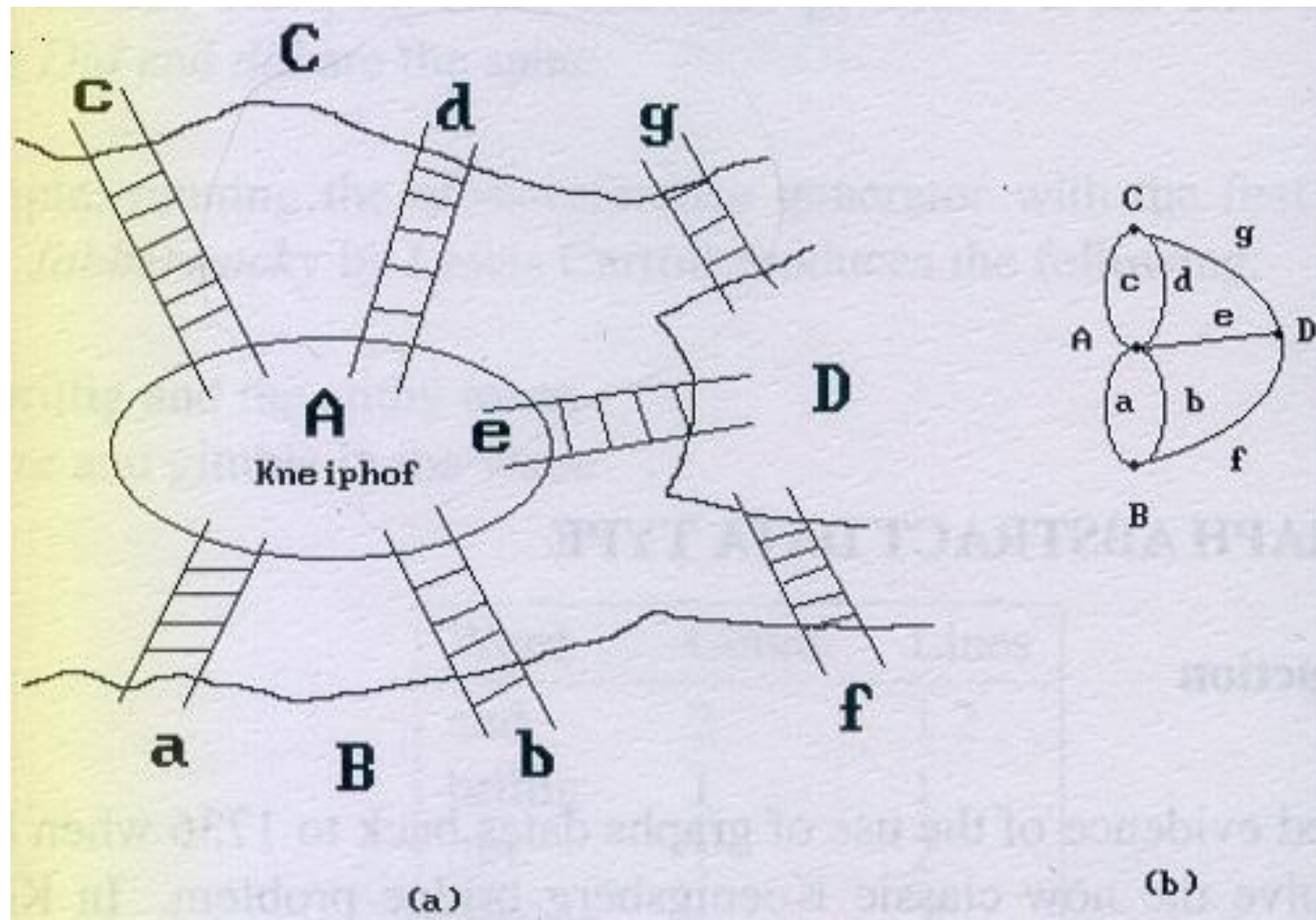
6.1 THE GRAPH ABSTRACT DATA TYPE

6.1.1 Introduction

The first recorded evidence of the use of graphs
dates back to 1736 when *Leonhard Euler* used them
to solve the classic *Koenigsberg* bridge problem.

See Figure 6.1.
(i.e., determining an Eulerian circuit(walk))

Euler proved that a graph has an *Eulerian walk*
iff the degree of each vertex is even.



Graphs are the most widely used mathematical structure

Applications

- Electrical circuit analysis
- Finding shortest routes
- Project planning
- Identification of chemical compounds
- Network flow design
- Gene/protein interactions, etc.

6.1.2 Definitions

a graph, G , consists of two sets :

$V(G)$, a finite, nonempty set of vertices, and

$E(G)$, a finite, possibly empty set of edges.

We may write $G=(V,E)$.

An undirected graph -- each edge is represented
as an unordered pair of vertices.

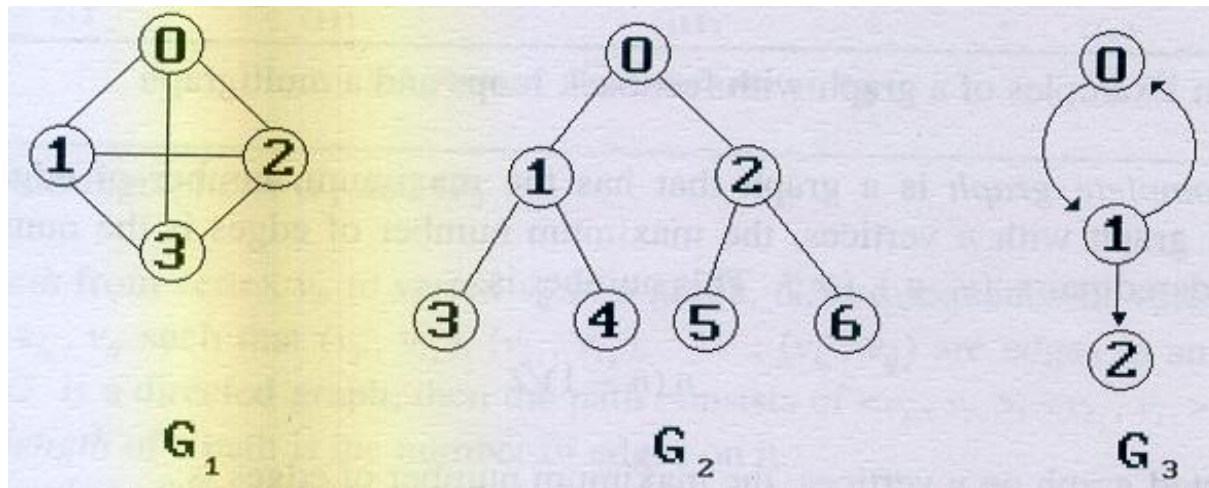
e.g., pairs (v_0, v_1) and (v_1, v_0) represent the same edge.

A directed graph -- each edge is represented
as a directed pair of vertices.

e.g., the pair $\langle v_0, v_1 \rangle$ represents an edge
in which v_0 is the tail and v_1 is the head.

Therefore $\langle v_0, v_1 \rangle$ and $\langle v_1, v_0 \rangle$ represent two different edges.

[Figure 6.2]



The set representation of each of these graphs is :

$$\begin{aligned}
 V(G_1) &= \{0, 1, 2, 3\} & E(G_1) &= \{(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)\} \\
 V(G_2) &= \{0, 1, 2, 3, 4, 5, 6\} & E(G_2) &= \{(0, 1), (0, 2), (1, 3), (1, 4), (2, 5), (2, 6)\} \\
 V(G_3) &= \{0, 1, 2\} & E(G_3) &= \{<0, 1>, <1, 0>, <1, 2>\}
 \end{aligned}$$

Notice that G_2 is a tree.

We can define trees as a special case of graphs.

Restrictions imposed on graphs :

1. No *self* loops - A graph may not have an edge from a vertex, i , back to itself.
2. No multiple edges - A graph may not have multiple occurrences of the same edge.

We refer a graph with multiple edge as a *multigraph*.

Complete graph --

For an undirected graph,
a complete graph with n vertices has
 $n(n-1)/2$ different edges.

For a directed graph,
a complete graph with n vertices has
 $n(n-1)$ different edges.

If (v_0, v_1) is an edge in an undirected graph,
then the vertices v_0 and v_1 are *adjacent*
and the edge (v_0, v_1) is *incident on* vertices v_0 and v_1 .

If $\langle v_0, v_1 \rangle$ is a directed edge,
then vertex v_0 is *adjacent to* vertex v_1 ,
while v_1 is *adjacent from* vertex v_0 .

The edge $\langle v_0, v_1 \rangle$ is *incident on* v_0 and v_1 .

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

A *path* from vertex v_p to vertex v_q in a graph, G ,
is a sequence of vertices, $v_p, v_{i_1}, v_{i_2}, \dots, v_{i_n}, v_q$,
such that $(v_p, v_{i_1}), (v_{i_1}, v_{i_2}), \dots, (v_{i_n}, v_q)$
are edges in an undirected graph.

If G' is a directed graph,
then the path consists of $\langle v_p, v_{i_1} \rangle, \langle v_{i_1}, v_{i_2} \rangle, \dots, \langle v_{i_n}, v_q \rangle$.

The *length* of a path is the number of edges on it.
simple path simple directed path

A *cycle* is a simple path
in which the first and the last vertices are the same.
directed cycle

**In an undirected graph G ,
two vertices, v_0 and v_1 are *connected*
if there is a path in G from v_0 to v_1 .**

**An undirected graph G is *connected*
if, for every pair of distinct vertices v_i , v_j ,
there is a path from v_i to v_j in G ,
i.e. they are connected.**

**A *connected component*, or simply a *component*, of an undirected graph
is a maximal connected subgraph.**

A *tree* is a graph that is connected and acyclic (it has no cycles).

A directed graph is *strongly* connected

if, for every pair of distinct vertices v_i, v_j in $V(G)$,
there is a directed path from v_i to v_j
and from v_j to v_i .

A *strongly connected component* is a maximal subgraph
which is strongly connected.

A *degree* of a vertex is

the number of edges incident to that vertex.

For a directed graph, we define

the *in-degree* of a vertex v

as the number of edges that have v as the head,

and the *out-degree* of a vertex v

as the number of edges that have v as the tail.

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

$$e = \frac{1}{2} \sum_{i=0}^{n-1} d_i .$$

[Structure 6.1] Abstract data type Graph

structure Graph is

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

functions : for all $\text{graph} \in \text{Graph}$, $v, v_1, v_2 \in \text{Vertices}$

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 vertices 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.

6.1.3 Graph Representations

Adjacency Matrix

Let $G = (V, E)$ be a graph with n vertices, $n \geq 1$.

Adjacency matrix of G is

a two-dimensional $n \times n$ array, say *adj_mat*, defined as:

$$\begin{aligned} \text{adj_mat}[i][j] &= 1 && \text{if the edge } (V_i, V_j) \text{ is in } E(G) \\ &= 0 && \text{if there is no such edge.} \end{aligned}$$

The same definition can be used for a directed graph

except that the edge $\langle V_i, V_j \rangle$ is directed.

The adjacency matrix for an undirected graph is symmetric,

since the edge (V_i, V_j) is in $E(G)$

iff the edge (V_j, V_i) is also in $E(G)$.

For undirected graphs,

we can save space by storing only the upper or lower triangle
of the matrix.

From the adjacency matrix, we can determine :

- if there is an edge connecting any two vertices.
- the degree of a vertex.

For an undirected graph,

the degree of any vertex i , is its row sum:

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

For a directed graph,

the row sum is the out-degree,

while the column sum is the in-degree.

Some questions or tasks require us
to examine (potentially) all edges of the graph,
e.g. How many edges are there in G ?
or, Is G connected?

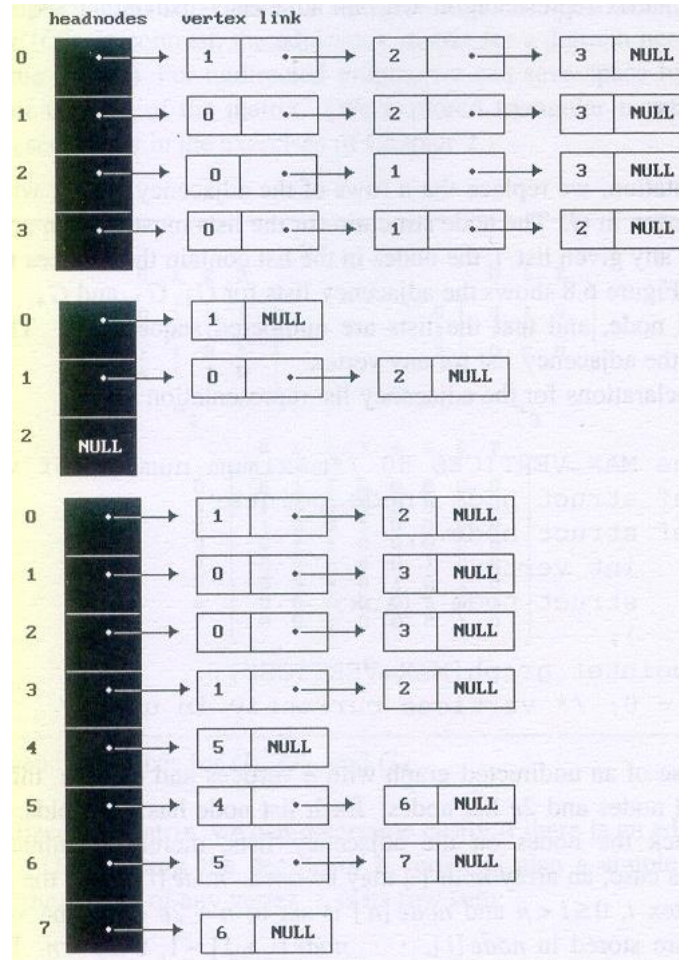
Using adjacency matrices,
all algorithms that answer these questions require at least $O(n^2)$ time
since we must examine $n^2 - n$ entries of the matrix
to determine the edges of the graph.

For a sparse graph, its adjacency matrix becomes sparse.
We might expect that the above questions would be answerable
in significantly less time, say $O(e+n)$ time.

Adjacency Lists

In this representation,
we replace the n rows of the adjacency matrix with n linked lists,
one for each vertex in G .

See Figure 6.8



The C declaration for the adjacency list representation :

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

In case of an undirected graph with n vertices and e edges,
this representation requires n head nodes and $2e$ list nodes.

Each list node has two fields.

Questions :

Determine the degree of any vertex, $0 \leq i < n$
equivalently, the number of edges incident on the vertex.

We can determine the total number of edges in G
in $O(n+e)$ time. $0 \leq i < n$

For a directed graph, we can determine the out-degree easily,
but finding the in-degree is more complex.

Maintaining the second list called inverse adjacency list.
See [Figure 6.10]

Weighted Edges

Need to modify our representations.

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 .

Depth First Search

Breadth First Search

6.2.1 Depth First Search

We begin by visiting the start vertex, v .

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.

This search terminates when the stack is empty.

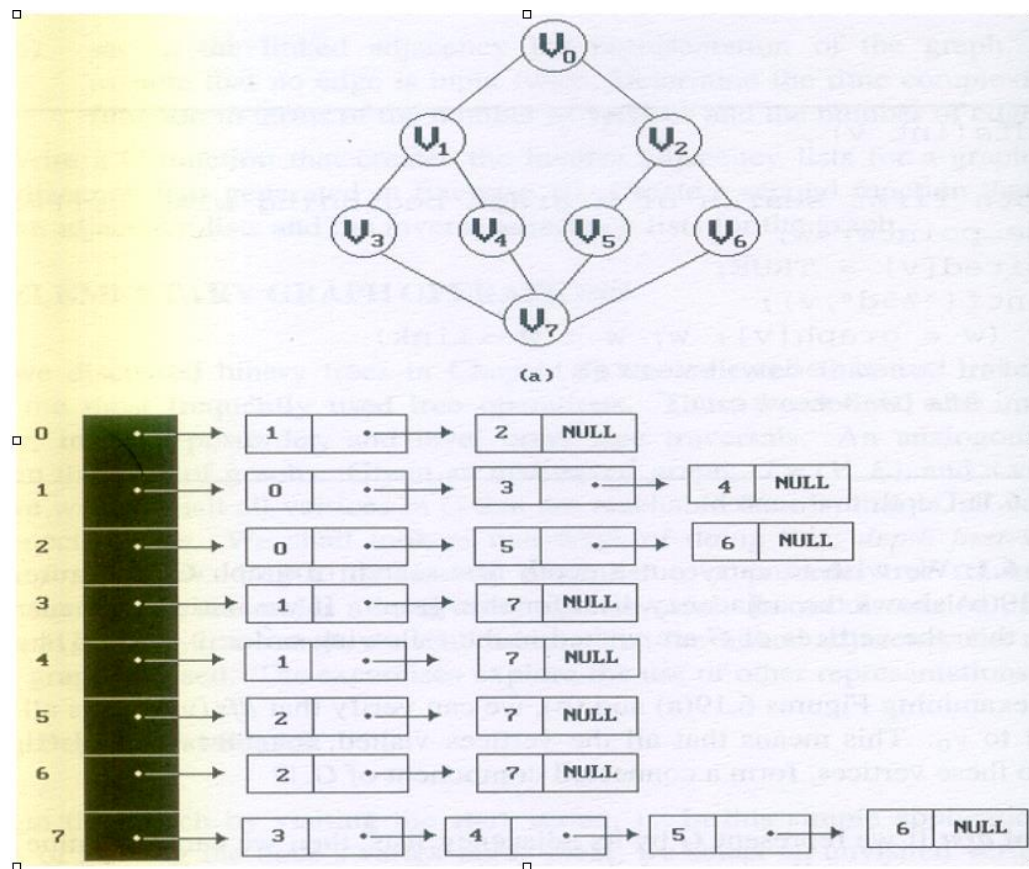
Declaration needed :

```
#define FALSE 0
#define TRUE 1
short int visited[MAX_VERTICES];
```

[Program 6.1] : Depth first search

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

Example 6.1 : See Figure 6.19.



Analysis of *dfs* :

If we use adjacency list,

since *dfs* examines each node in adjacency list at most once,
the time to complete the search is $O(e)$.

If we use adjacency matrix,

then determining all vertices adjacent to v requires $O(n)$ time.

Since we visit at most n vertices,

the total time is $O(n^2)$.

6.2.2 Breadth First Search

Starts at vertex 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,
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.

When the queue is empty, the search is finished.

To implement breadth first search,
we use a dynamically linked queue.

Necessary declarations:

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

[program 6.2] : Breadth first search

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

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 $d_0 + d_1 + \dots + d_{n-1} = O(e)$,
where $d_i = \text{degree}(v_i)$.

For the adjacency matrix representation,
the while loop takes $O(n)$ times for each vertex visited.
Therefore, the total time is $O(n^2)$.

6.2.3 Connected Components

Note that in both depth first search and breadth first search,
all vertices visited, together with all edges incident to them,
form a connected component of G .

This property allows us to determine
whether or not an directed graph is connected.

Simply calling either $\text{dfs}(0)$ or $\text{bfs}(0)$
and then determining if there are any unvisited vertices.

This takes $O(n+e)$ time if adjacency lists are used.

A closely related problem is
that of listing the connected components.

[Program 6.3] : Connected components

```
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");
        }
}
```

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,
the time needed to determine the connected components is $O(n^2)$.

6.2.4 Spanning Trees

When G is connected,
a depth first search 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) is the set of edges used or traversed
 during the search
 and N (for nontree edges) is the set of remaining edges.

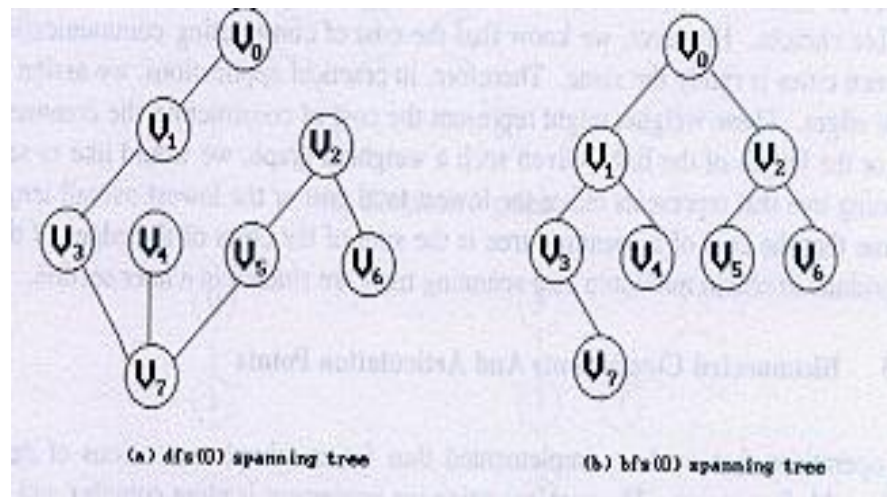
The edges in T form a tree that include all vertices of G .

A spanning tree is any tree that consists solely of edges in G
and that includes all the vertices in G .

See Figure 6.17 (page 284)

We may use either *dfs* or *bfs* to create a spanning tree;
depth first spanning tree
breadth first spanning tree

[Figure 6.18]



If we add a nontree edge (v,w) into any spanning tree T ,
it creates a cycle that consists of the edge (v,w)
and all edges on the path from w to v in T .

Another property of spanning tree:

A spanning tree is a minimal subgraph G' of G
such that $V(G') = V(G)$ and G' is connected.

Any connected graph with n vertices must have at least $n-1$ edges,
and all connected graph with $n-1$ edges are trees.

Therefore, the spanning tree has $n-1$ edges.

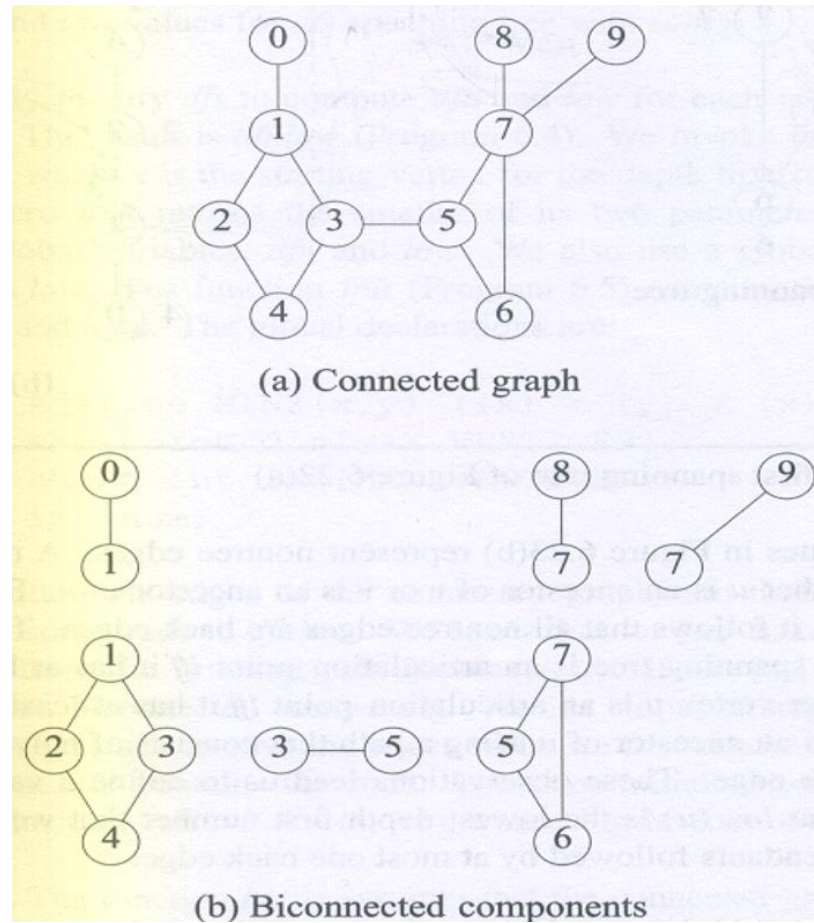
Constructing minimal subgraph finds frequent application
in the design of communication networks.

6.2.5 Biconnected Components and Articulation Points

For an undirected connected graph G ,
an articulation point is a vertex v of G such that
the deletion of v , together with all edges incident on v ,
produces a graph $G - v$
that has at least two connected components.

A biconnected graph is a connected graph
that has no articulation points.
See Figure 6.22 for an example

[Figure 6.22]



In many graph application,
articulation points are undesirable.

A *biconnected* component of a connected
undirected graph is a
maximal *biconnected subgraph* H of G .

It is easy to verify that
two biconnected components of the same graph
have no more than one vertex in common.

This means no edge can be
in two or more biconnected components
of a graph.

Hence, the biconnected components of G partition
the edges of G .

Finding the biconnected components of a connected undirected graph G by using a depth first spanning tree of G :

We number the vertices in the order in which the vertices are visited during the depth first search.

We call this number the *depth first number*, or *dfn*, of the vertex.

If u and v are two vertices, and u is an ancestor of v in the depth first spanning tree, then $dfn(u) < dfn(v)$.

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 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 dfn that we can reach from u
using a path of descendants followed by at most one back edge:

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

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 has a child w
such that $low(w) \geq dfn(u)$.

[Figure 6.24]

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

We can easily modify *dfs* to compute *dfn* and *low* for each vertex of a connected undirected graph. See Program 6.4.

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

In that program, we use a macro, *MIN2*, and global variables, *dfn*, *low* and *num*.

The function *init* (Program 6.5) contains the code to correctly initialize *dfn*, *low*, and *num*.

Declaration needed:

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

[Program 6.5] Initialization of dfn and low.

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

[Program 6.4]

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

If $\text{low}[w] \geq \text{dfn}[u]$,
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.

See Program 6.6.

Its initial call is *bicon*(*x*, -1),
where *x* is the root of the spanning tree.

The same initialization function (Program 6.5) is used.

[Program 6.6] Biconnected components of a graph

```
void bicon(int u, int v)
{
    node_pointer ptr;
    int w, x, y;
    dfn[u] = low[u] = num++;
    for (ptr=graph[u]; ptr; ptr=ptr->link) {
        w = ptr->vertex;
        if (v != w && dfn[w] < dfn[u])
            add(&top, u, w); /* add edge to stack */
        if (dfn[w] < 0) { /* w is an unvisited vertex */
            bicon(w, u);
            low[u] = MIN2(low[u], low[w]);
            if (low[w] >= dfn[u]) {
                printf("New biconnected component: ");
                do { /* delete edge from stack */
                    delete(&top, &x, &y);
                    printf("<%d, %d>", x, y);
                } while (!(x == u) && (y == w));
                printf("□\n");
            }
        }
        else if (w != v) low[u] = MIN2(low[u], dfn[w]);
    }
}
```

Analysis of *bicon* :

The function *bicon* assumes

that the connected graph has at least two vertices.

The time complexity of *bicon* is $O(n+e)$.

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 greedy algorithms :
Kruskal's algorithm,
Prim's algorithm,
Sollin's algorithm.

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.

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

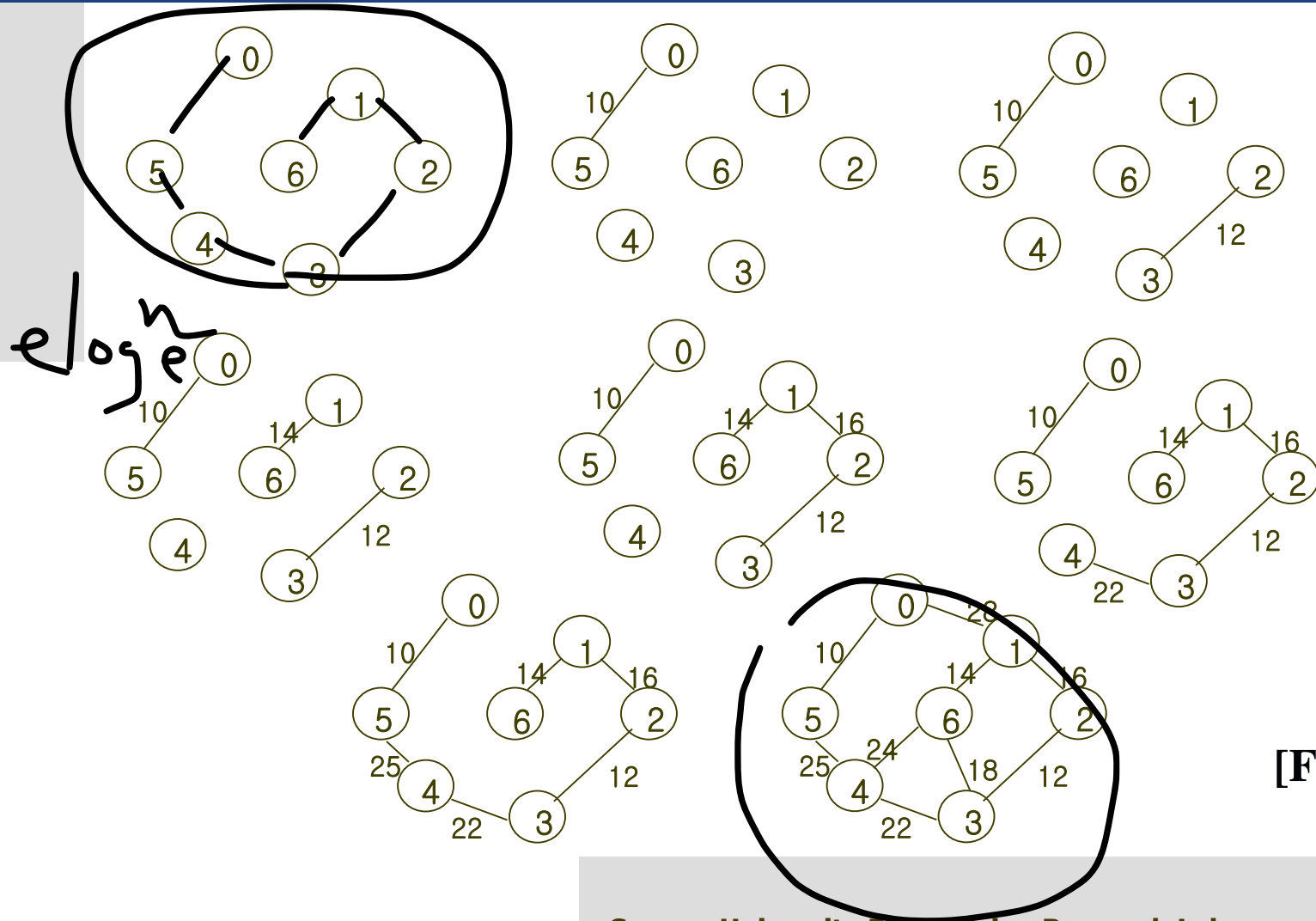
Kruskal's Algorithm

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 edges that are already in T .

If G is connected and has vertices $n > 0$, exactly $n-1$ edges will be selected for inclusion in T .



[Figure 6.25]

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

[Figure 6.26] Summary of Kruskal's algorithm applied to figure 6.25(a)

[Program 6.7] Kruskal's algorithm.

```
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");
```

Implementation :

How to determine an edge with minimum cost and delete that edge?
Sorting or Using a min heap.

How to check the new edge, (v,w) , does not form a cycle in T ?
We may use the union-find operations in Section 5.10.

The computing time of Kruskal's algorithm is $O(e \log e)$.

[Theorem 6.1] : Let G be an undirected connected graph,
Kruskal's algorithm generates a minimum cost spanning tree.

Prim's Algorithm

Prim's algorithm, like Kruskal's algorithm,
constructs the minimum cost spanning tree one edge at a time.

However, at each stage, the set of selected edges forms a tree.

Prim's algorithm begins with a tree, T ,
that contains a single vertex.

This may be any of the vertices.

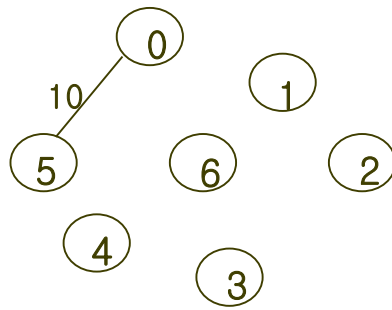
Next, we add a least cost edge (u,v) to T
such that $T \cup \{(u,v)\}$ is also a tree.

We repeat this edge addition step
until T contains $n-1$ edges.

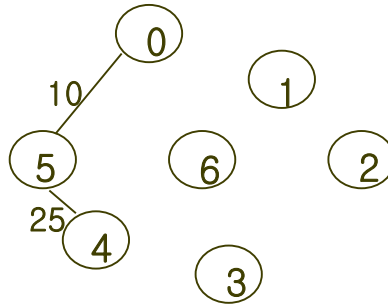
To make sure that the added edge does not form a cycle,
at each step we choose the edge (u,v) such that
exactly one of u or v in T .

[Program 6.8]: Prim's algorithm

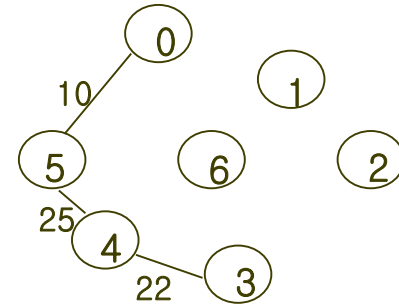
```
T = {};  
TV = {0}; /* start with vertex 0 and no edges */  
while (T contains fewer than n-1 edges) {  
    let (u,v) be a least cost edge such that  
        u ∈ TV and v ∉ TV;  
    if (there is no such edge)  
        break;  
    add v to TV;  
    add (u,v) to T;  
}  
if (T contains fewer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



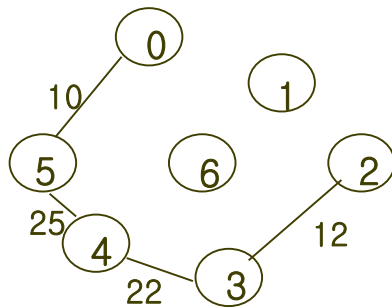
(a)



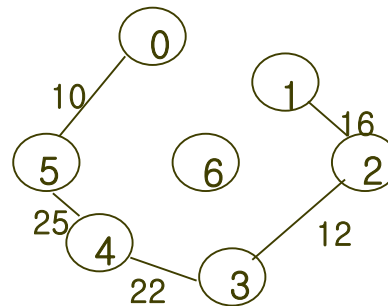
(b)



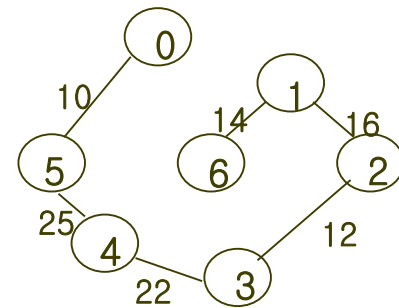
(c)



(d)



(e)



(f)

[Figure 6.27]

Implementation :

For each vertex v not in T ,
we keep a companion vertex, $\text{near}(v)$,
such that $\text{near}(v) \in T$ and
 $\text{cost}(\text{near}(v), v)$ is minimum over all such choices for $\text{near}(v)$.

Computing time is $O(n^2)$, where n is the number of vertices in G .

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.

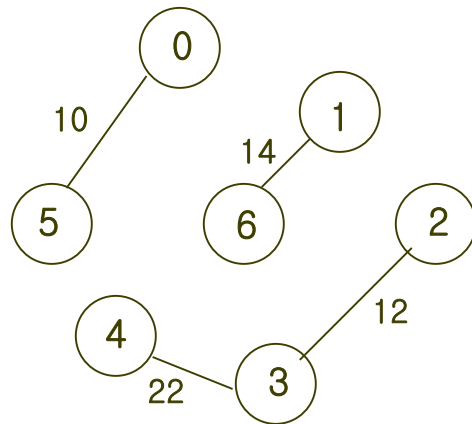
During a stage we select one edge for each tree in the forest.

This edge is a minimum cost edge

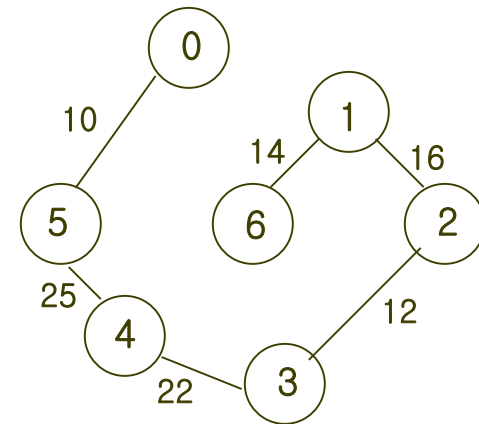
that has exactly one vertex in the tree.

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.



(a)



(b)

[Figure 6.28] Stages in Sollin's algorithm

Shortest Paths and Transitive Closure

Suppose we have a graph that represents the highway system.
In this graph, the vertices represent cities and
edges represent sections of the highway.
Each edge has a weight representing the distance
between the two cities connected by the edge.

Questions (from a motorist who wish to drive from city A to B):

- (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?

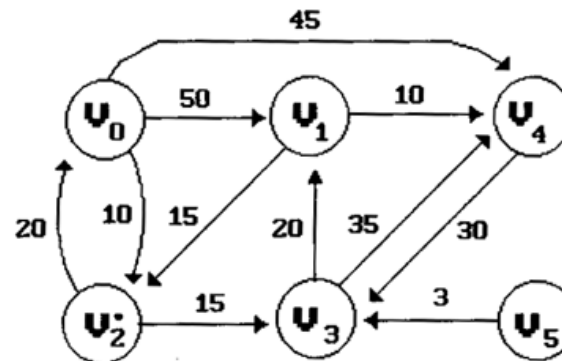
We define the length of a path
as the sum of the weights of the edges on that path.

We assume a directed graph.

Single Source All Destinations: Nonnegative Edge Costs

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 .



(a)

	path	length
1)	$v_0 \ v_2$	10
2)	$v_0 \ v_2 \ v_3$	25
3)	$v_0 \ v_2 \ v_3 \ v_1$	45
4)	$v_0 \ v_4$	45

(b)

We may use a greedy algorithm to generate the shortest paths
in nondecreasing order of their lengths.

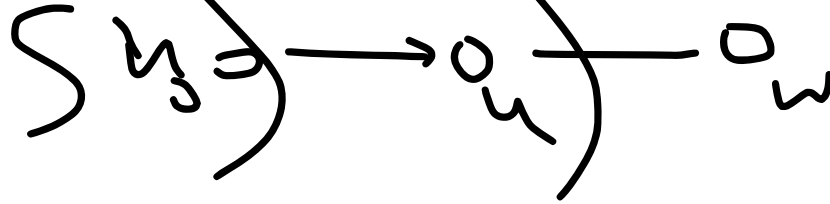
Let S be the set of vertices, including v_0 ,
whose shortest paths have been found.

For w not in S ,

let $\text{distance}[w]$ be the length of the shortest path starting from v_0 ,
going through vertices only in S , and ending in w .

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 .
- (2) Vertex u is chosen so that it has the minimum distance,
 $\text{distance}[u]$, among all the vertices not in S .
- (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 .



Implementing Dijkstra's algorithm

Assume that the n vertices are numbered from 0 to $n-1$.

Maintain the set S as an array, found,

found[i] = FALSE if vertex i is not in S and

found[i] = TRUE if vertex i is in S .

Graph is represented by its cost adjacency matrix,

with cost[i][j] being the weight of edge $\langle i, j \rangle$.

If the edge $\langle i, j \rangle$ is not in G ,

we set cost[i][j] to some large number.

The choice of this number is arbitrary, but we make two stipulations:

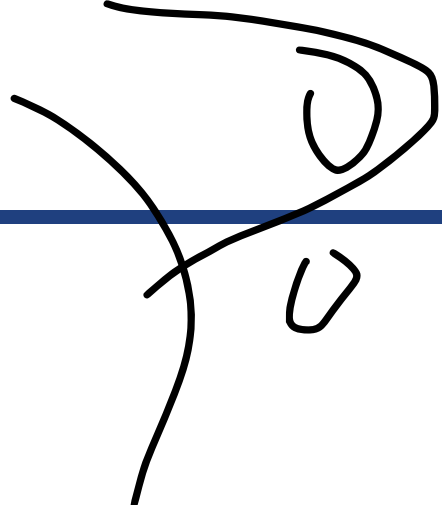
- (1) The number must be larger than any of the value
in the cost matrix.
- (2) The number must be chosen so that distance[u] + cost[u][w] does not produce
an overflow into the sign bit.

Program 6.9 : Declarations for the shortest path algorithm

```
#define MAX_VERTICES 6 /* maximum number of vertices*/
int cost[][MAX_VERTICES] =
    {{ 0, 50, 10, 1000, 45, 1000},
     { 1000, 0, 15, 1000, 10, 1000},
     { 20, 1000, 0, 15, 1000, 1000},
     { 1000, 20, 1000, 0, 35, 1000},
     { 1000, 1000, 30, 1000, 0, 1000},
     { 1000, 1000, 1000, 3, 1000, 0}};
int distance[MAX_VERTICES];
short int found[MAX_VERTICES];
int n = MAX_VERTICES;
```

Program 6.10 : Single source shortest paths

```
void shortestpath(int v, int cost[][MAX_VERTICES],
                 int distance[], int n, short int found[])
{
    /* distance[i] represents the shortest path from vertex v to i,
    found[i] holds a 0 if the shortest path from vertex i has not
    been found and a 1 if it has. cost is the adjacency matrix */
    int i, u, w;
    for (i=0; i<n; i++) {
        found[i] = FALSE;
        distance[i] = cost[v][i];
    }
}
```

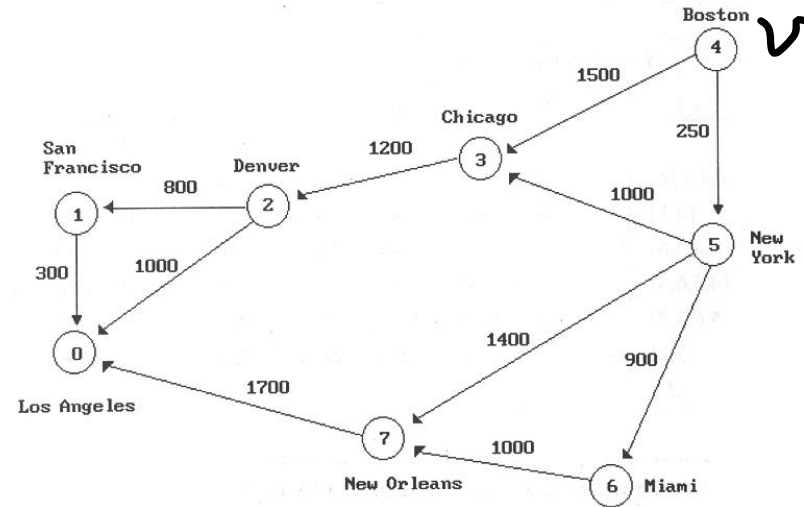
```
found[v] = TRUE;
distance[v] = 0;
for (i=0; i<n-2; i++) {
    u = choose(distance, n, found);
    found[u] = TRUE;
    for (w=0; w<n; w++)
        if (!found[w])
            if (distance[u] + cost[u][w] < distance[w])
                distance[w] = distance[u] + cost[u][w];
}
}
```

Program 6.10 : Single source shortest paths

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

Analysis of *shortestpath* : $O(n^2)$

Example 6.4 :



(a) Digraph of hypothetical airline routes

	0	1	2	3	4	5	6	7
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

(b) Cost adjacency matrix

Iteration	S	Vertex selected	Distance (4)							
			LA [0]	SF [1]	DEN [2]	CHI [3]	BOST [4]	NY [5]	MIA [6]	NO [7]
Initial	--	----	$+\infty$	$+\infty$	$+\infty$	1500	0	250	$+\infty$	$+\infty$
1	{4}	5	$+\infty$	$+\infty$	$+\infty$	1250	0	250	1150	1650
2	<u>{4,5}</u>	<u>6</u>	$+\infty$	$+\infty$	$+\infty$	1250	0	250	1150	1650
3	{4,5,6}	3	$+\infty$	$+\infty$	2450	1250	0	250	1150	1650
4	{4,5,6,3}	7	3350	$+\infty$	2450	1250	0	250	1150	1650
5	{4,5,6,3,7}	2	3350	3250	2450	1250	0	250	1150	1650
6	{4,5,6,3,7,2}	1	3350	3250	2450	1250	0	250	1150	1650
	{4,5,6,3,7,2,1}									

Figure 6.31: Action of *shortestpath* on the digraph of Figure 6.30

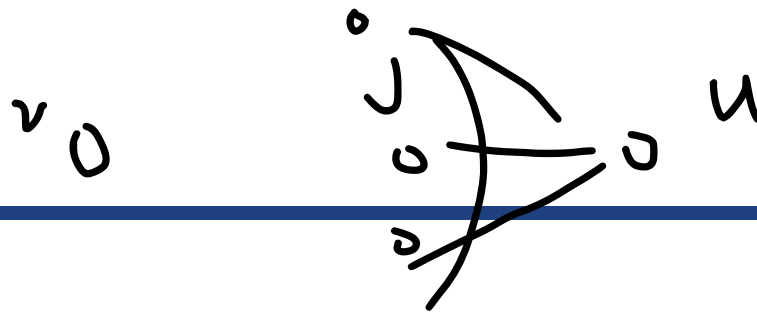
Single Source All Destinations: General Weights

Dijkstra's algorithm does not work for graphs with negative weights (e.g. Figure 6.29)

Bellman-Ford algorithm solves this problem (without cycles of negative length)

$dist^l[u]$: length of a shortest path from the source vertex v to vertex u
under the constraint that the shortest path contains at most l edges

Goal: compute $dist^{n-1}[u]$ for all u



Observations:

1. If the shortest path from v to u with at most k , $k > 1$, edges has no more than $k-1$ edges, then $dist^k[u] = dist^{k-1}[u]$
2. If the shortest path from v to u with at most k , $k > 1$, edges has exactly k edges, then it is comprised of a shortest path from v to some vertex j followed by the edge $\langle j, u \rangle$. The path from v to j has $k-1$ edges, and its length is $dist^{k-1}[j]$. All vertices i such that the edge $\langle i, u \rangle$ is in the graph are candidates for j . Since we are interested in a shortest path, the i that minimizes is the correct value for j

$$dist^k[u] = \min \{ dist^{k-1}[u], \min_i \{ dist^{k-1}[i] + length[i][u] \} \}$$

Bellman-Ford Algorithm

```
void BellmanFord(int n, int v)
{ // single source all destination shortest paths with
  // negative edge lengths
  for (int i=0; i<n; i++)
    dist[i] = length[v][i]; // initialize dist
  for (int k=2; k<=n-1; k++)
    for (each u such that u != v and u has at least one
         incoming edge)
      for (each <i, u> in the graph)
        if (dist[u] > dist[i] + length[i][u])
          dist[u] = dist[i] + length[i][u];
}
```

Time complexity: $O(n^3)$ with adjacency matrix, $O(ne)$ with list

All Pair Shortest Paths

We wish to find the shortest paths between all pairs of vertices, $v_i, v_j, i \neq j$.

We could solve this problem using shortest path
with each of the vertices in $V(G)$ as the source.

The total time required would be $O(n^3)$.

However, we can obtain a conceptually simpler algorithm that works correctly even if some edges in G have negative weights.

(We do require that G has no cycles with a negative length.)

Dynamic programming method.

Although this algorithm still has a computing time of $O(n^3)$,
it has a smaller constant factor.

We represent the graph G by its cost adjacency matrix.

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

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

$$A^{-1}[i][j] = \text{cost}[i][j].$$

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

If we have already generated A^{k-1} , then we may generate A^k by realizing that for any pair of vertices i, j one of the two rules below applies.

- (1) The shortest path from i to j going through no vertex with index greater than k does not go through the vertex with index k and so its cost is $A^{k-1}[i][j]$.
- (2) The shortest path go through the vertex k . Such a path consists of a path from i to k followed by one from k to j .

These rules yield the following formulas for $A^k[i][j]$:

$$A^k[i][j] = \min\{A^{k-1}[i][j], A^{k-1}[i][k] + A^{k-1}[k][j]\}, \quad k \geq 0$$

and

$$A^1[i][j] = \text{cost}[i][j].$$

Example 6.5: Look at $A^1[0][2]$.

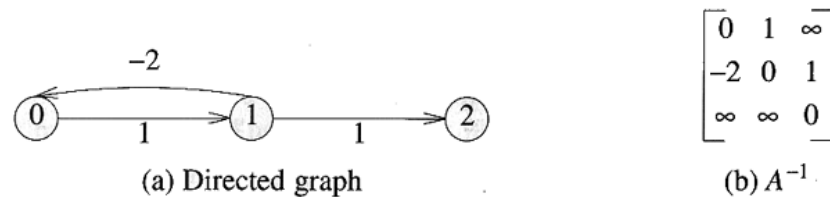


Figure 6.32: Graph with negative cycle

The function *allcosts* computes $A^{n-1}[i][j]$.

The computations are done in place using the array *distance*.

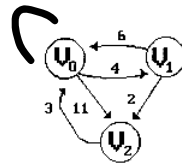
Note that $A^k[i][k] = A^{k-1}[i][k]$ and $A^k[k][j] = A^{k-1}[k][j]$.

Program 6.12 : All pairs, shortest paths function

```
void allcosts(int cost[][MAX_VERTICES],
              int distance[][MAX_VERTICES], int n)
{ /* determine the distances from each vertex to every other vertex, cost is
   the adjacency matrix, distance is the matrix of the distances. */
  int i, j, k;
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      distance[i][j] = cost[i][j];
  — for (k=0; k<n; k++)
    for (i=0; i<n; i++)
      — for (j=0; j<n; j++)
        — if (distance[i][k] + distance[k][j] < distance[i][j])
          distance[i][j] = distance[i][k] + distance[k][j];
}
```

Analysis of *allcosts* : The total time for *allcosts* is $O(n^3)$.

Example 6.6:



	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

(a) Digraph G

(b) Cost adjacency matrix for G

Figure 6.33: Directed graph and its cost matrix

A^{-1}	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

A^0	0	1	2
0	0	4	11
1	6	0	2
2	3	7	0

A^1	0	1	2
0	0	4	6
1	6	0	2
2	3	7	0

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