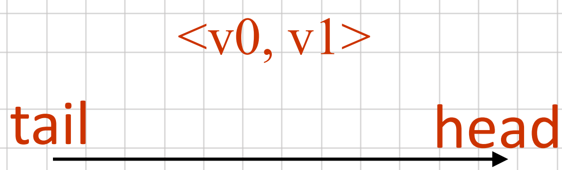
CHAPTER 6
GRAPHS

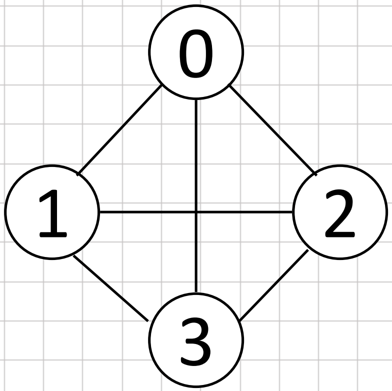
All the programs in this file are selected from
Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed
“Fundamentals of Data Structures in C”,

Definition

- A graph G consists of two sets
  - a finite, nonempty set of **vertices** V(G)
  - a finite, possible empty set of **edges** E(G)
  - G(V,E) represents a graph
- An undirected graph is one in which the pair of vertices in a edge is unordered, (v0, v1) = (v1,v0)
- A directed graph is one in which each edge is a directed pair of vertices, <v0, v1> != <v1,v0>

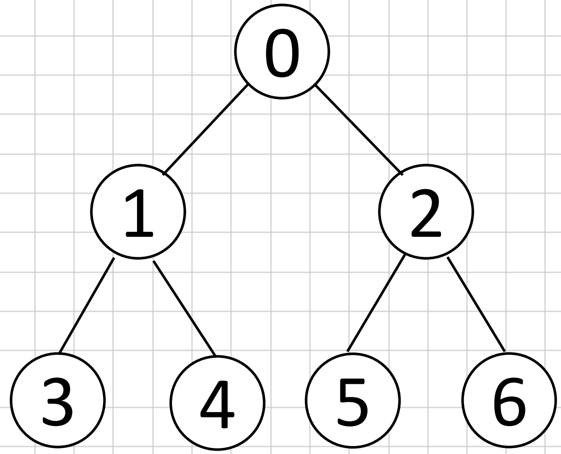


Examples for Graph



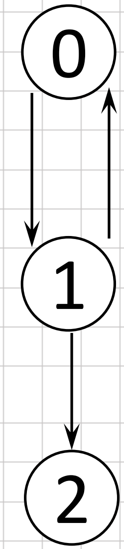
G_1

complete graph



G_2

incomplete graph



G_3

$V(G_1)=\{0,1,2,3\}$ $E(G_1)=\{(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)\}$

$V(G_2)=\{0,1,2,3,4,5,6\}$ $E(G_2)=\{(0,1),(0,2),(1,3),(1,4),(2,5),(2,6)\}$

$V(G_3)=\{0,1,2\}$ $E(G_3)=\{<0,1>,<1,0>,<1,2>\}$

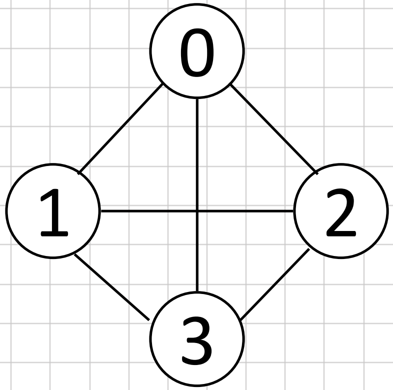
Complete Graph

- A complete graph is a graph that has the maximum number of edges

  - for **undirected graph** with n vertices, the maximum number of edges is **n(n-1)/2**

  - for **directed graph** with n vertices, the maximum number of edges is **n(n-1)**

  - example: G1 is a complete graph

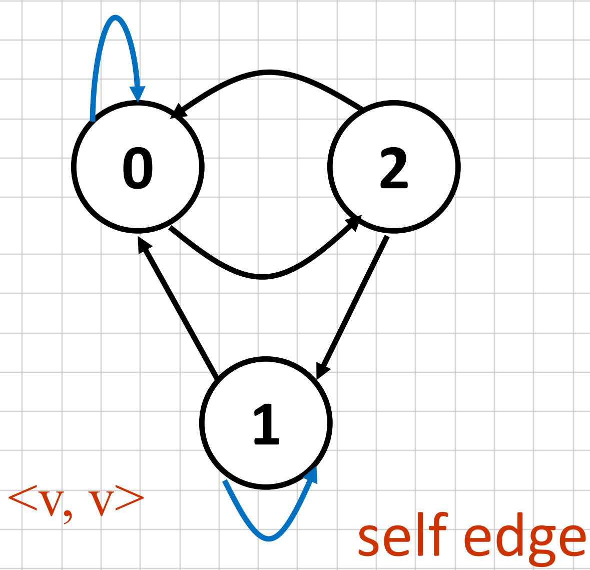


G_1

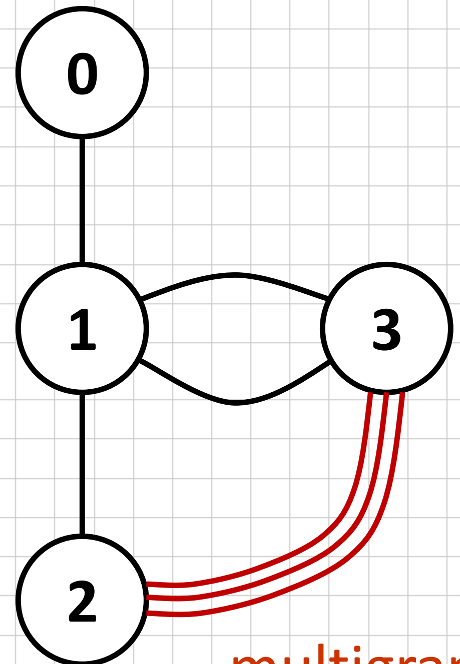
Adjacent and Incident

- If (v0, v1) is an edge in an undirected graph,
  - v0 and v1 are **adjacent**
  - The edge (v0, v1) is **incident** on vertices v0 and v1
- If <v0, v1> is an edge in a directed graph
  - v0 is adjacent to v1, and v1 is **adjacent from** v0
  - The edge <v0, v1> is incident on v0 and v1

Figure 6.3: Example of a graph with feedback loops and a multigraph (p.268)



(a)



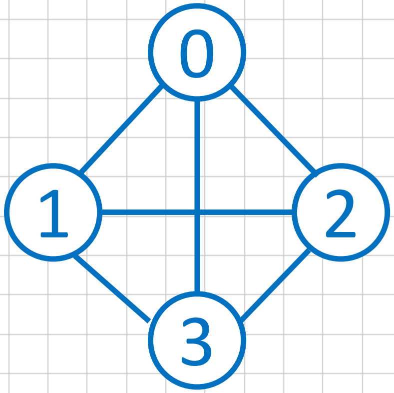
(b)

multigraph:
multiple occurrences of the same edge

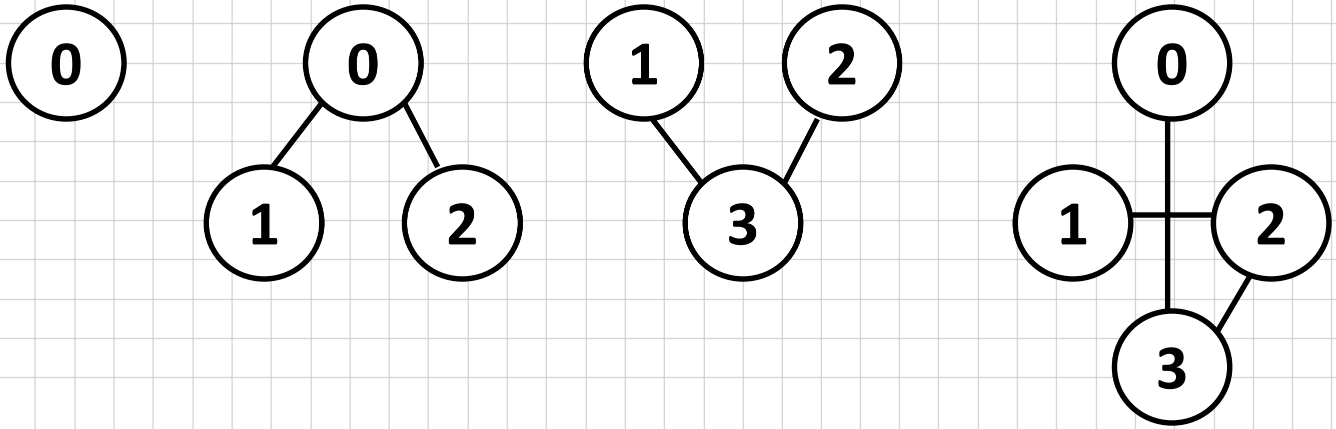
Subgraph and Path

- A subgraph of G is a graph G' such that V(G') is a subset of V(G) and E(G') is a subset of E(G)
- A path from vertex vp to vertex vq in a graph G, is a sequence of vertices, v_p, v_{i1}, v_{i2}, ..., v_{in}, v_q, such that (v_p, v_{i1}), (v_{i1}, v_{i2}), ..., (v_{in}, v_q) are edges in E(G'), if G' is directed
- The length of a path is the number of edges on it

Subgraph

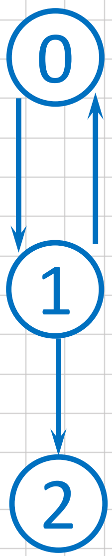


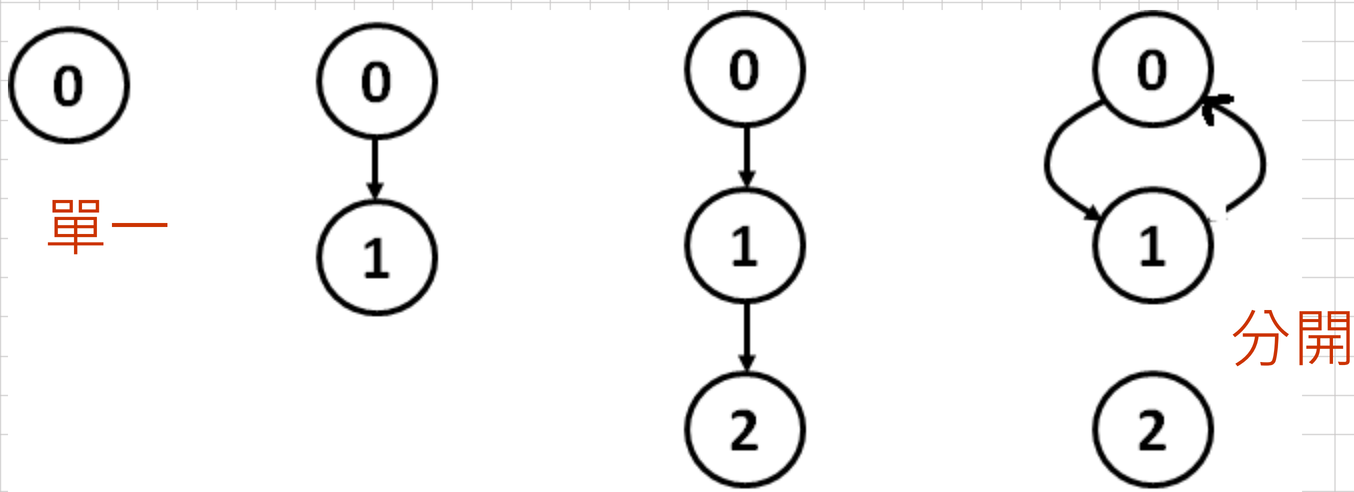
G_1



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

(a) Some of the subgraph of G_1





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

(b) Some of the subgraph of G_3

G_3

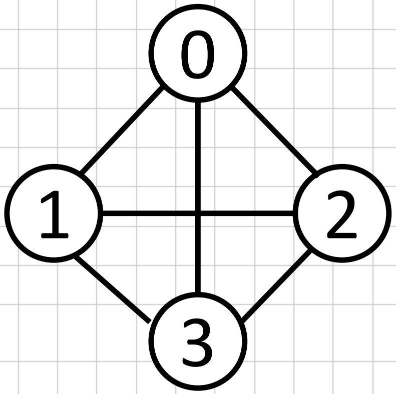
Figure 6.4: subgraphs of G_1 and G_3 (p.269)

Simple Path and Cycle

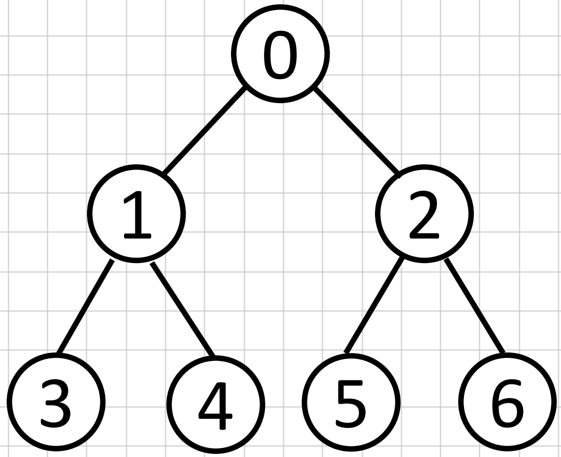
- A simple path is a path in which all vertices, except possibly the first and the last, are distinct

  (0,1), (1,3),(3,2) is also written as 0,1,3,2
- A cycle is a simple path in which the first and the last vertices are the same
- In an undirected graph G, two vertices, v0 and v1, are connected if there is a **path** in G from v0 to v1

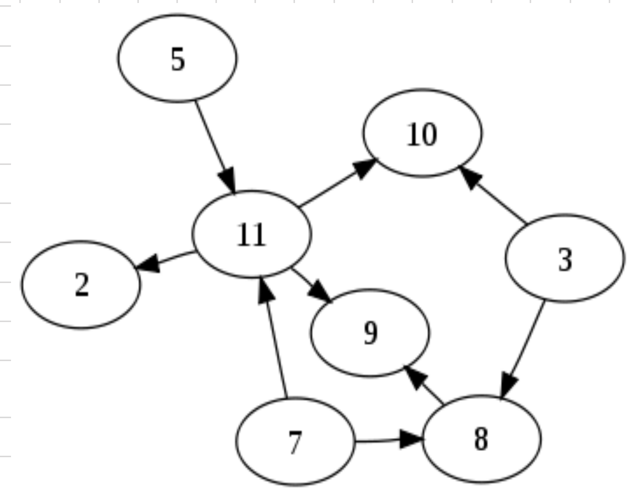
connected



G1



G2



G3

tree (acyclic graph)

Directed acyclic graph

Connected Component

- A connected component of an undirected graph is a maximal connected subgraph.
- A tree is a graph that is connected and acyclic.
- A directed graph is strongly connected if there is a directed path from vi to vj and also from vj to vi.
- A strongly connected component is a maximal subgraph that is strongly connected.

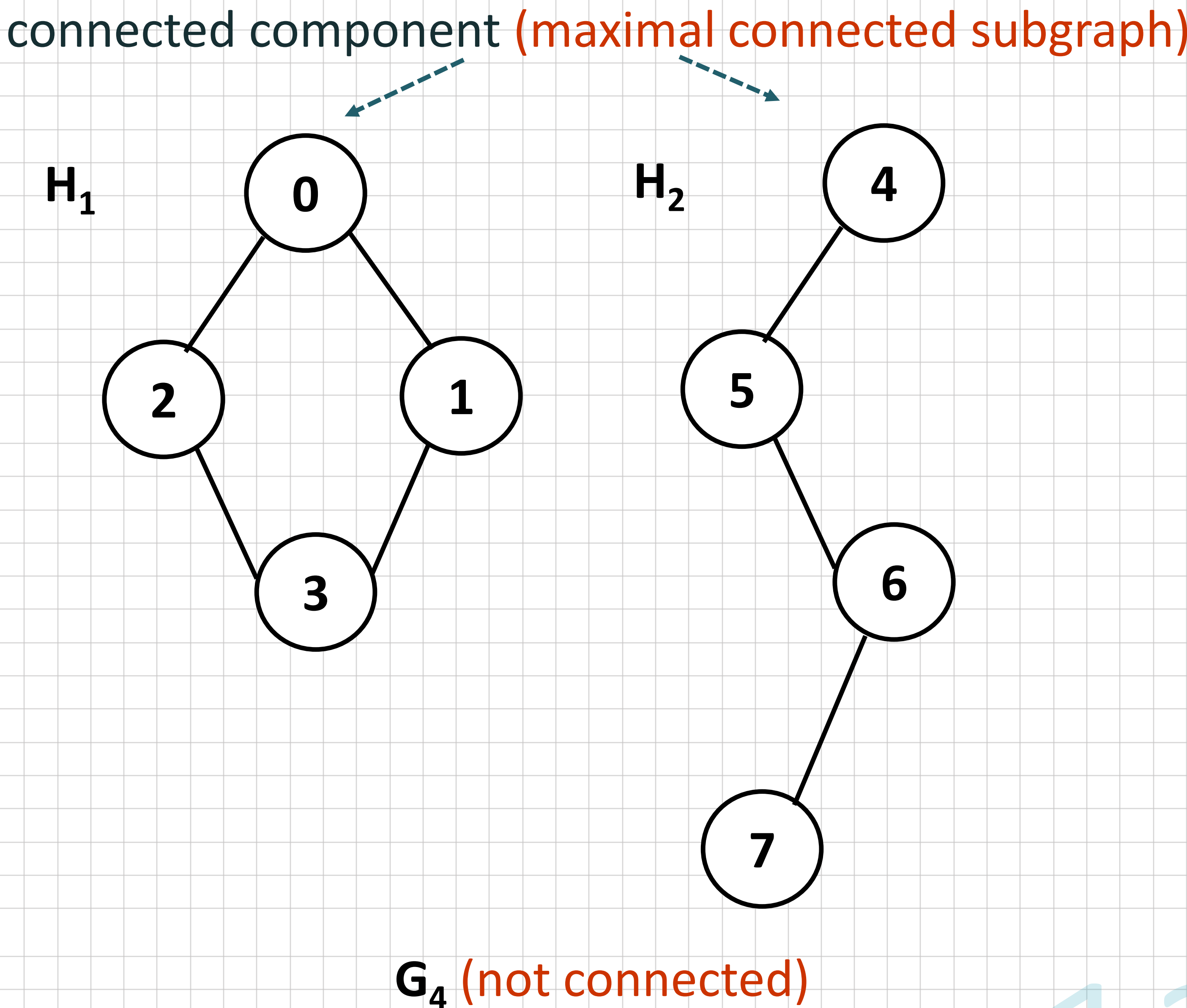


Figure 6.5: A graph with two connected components (p.262)

G3

strongly connected component
(maximal strongly connected subgraph)

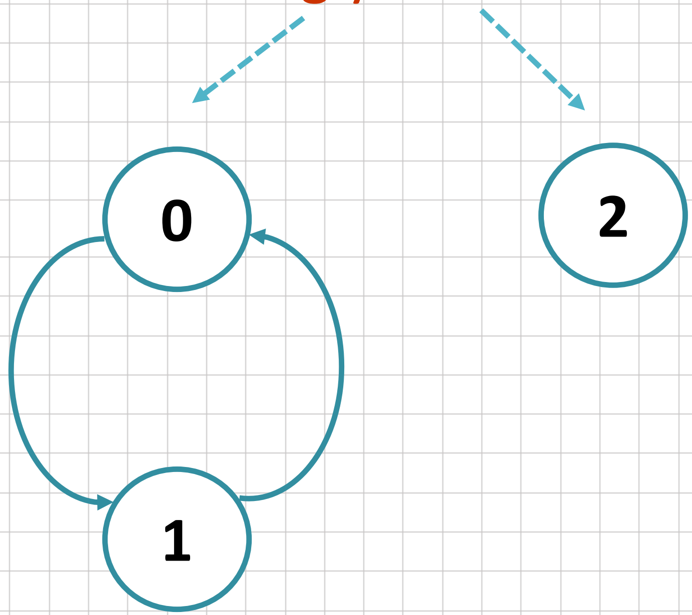
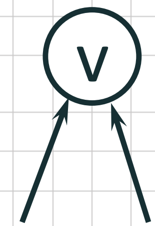


Figure 6.6: Strongly connected components of G_3 (p.262)

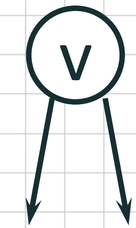
Degree

- The degree of a vertex is the number of edges incident to that vertex
- For directed graph,

  - the in-degree of a vertex v is the number of edges that have v as the head



  - the out-degree of a vertex v is the number of edges that have v as the tail



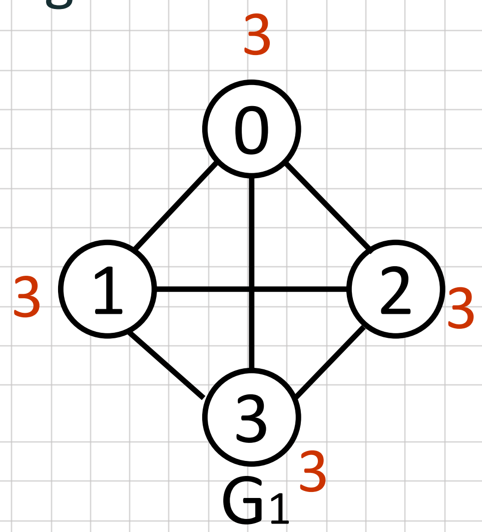
- if di is the degree of a vertex i in a graph G with n vertices and e edges, the number of edges is :

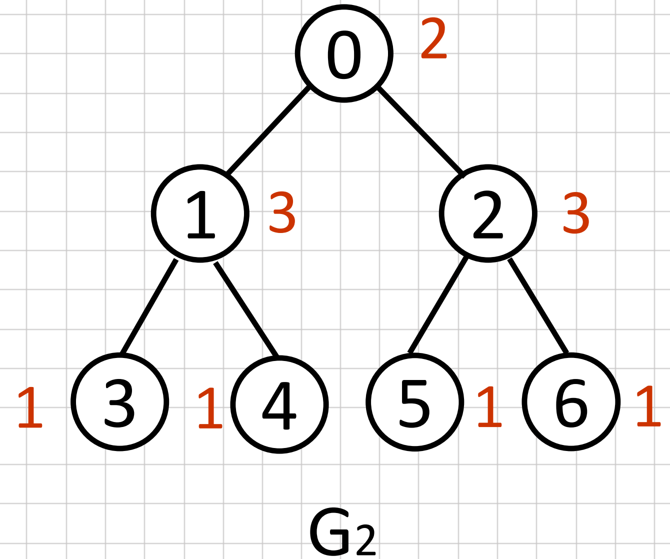
$$e = \left(\sum_{0}^{n-1} d_i\right) / 2$$

For an undirected graph

undirected graph

degree

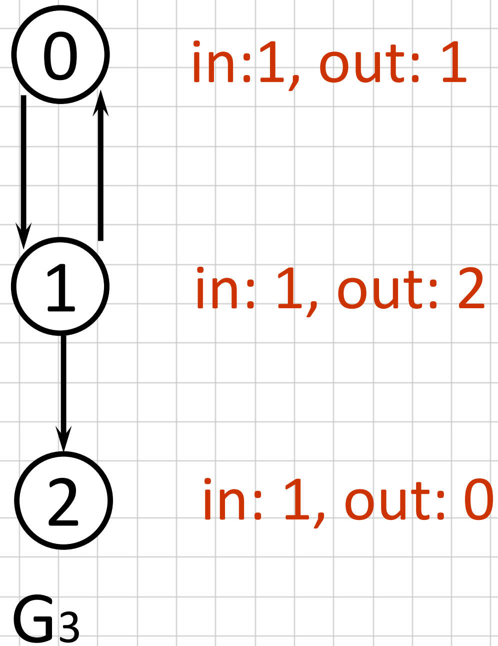




directed graph

in-degree

out-degree



ADT for Graph

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 $\in$ Graph, v, v1 and v2 $\in$ Vertices

Graph Create()

*::=**return** an empty graph*

Graph InsertVertex(graph, v)

*::= **return** a graph with v inserted. v has no incident edge.*

Graph InsertEdge(graph, v1,v2)

*::= **return** a graph with new edge between v1 and v2*

Graph DeleteVertex(graph, v)

*::= **return** a graph in which v and all edges incident to it are removed*

Graph DeleteEdge(graph, v1, v2)

*::=**return** a graph in which the edge (v1, v2) is removed*

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*

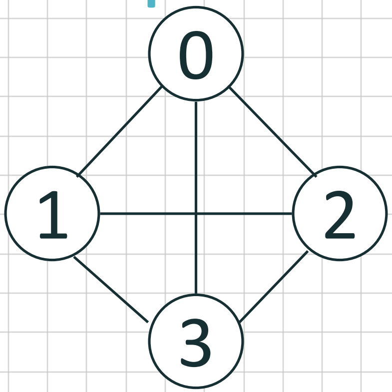
Graph Representations

- Adjacency Matrix
- Adjacency Lists
- Sequential Representation
- Adjacency Multilists

Adjacency Matrix

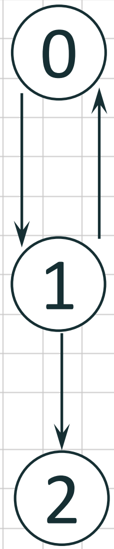
- Let G=(V,E) be a graph with **n** vertices.
- The adjacency matrix of G is a two-dimensional n by n array, say adj_mat
- If the edge (vi, vj) is in E(G), adj_mat[i][j]=1
- If there is no such edge in E(G), adj_mat[i][j]=0
- The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph need not be symmetric

Examples for Adjacency Matrix



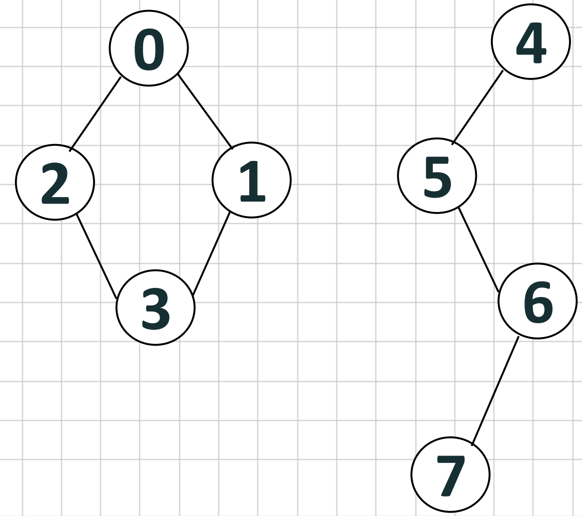
$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

G_1



$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

G_2



$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

G_4

symmetric

<u>Space usage</u>

undirected: $n^2/2$

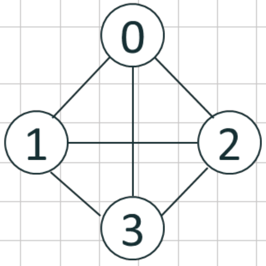
directed: n^2

Merits of Adjacency Matrix

From the adjacency matrix, to determine the connection of vertices is easy.

The degree of a vertex is:

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



$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

For a **digraph**, the **row sum** is the out_degree, while the **column** sum is the in_degree

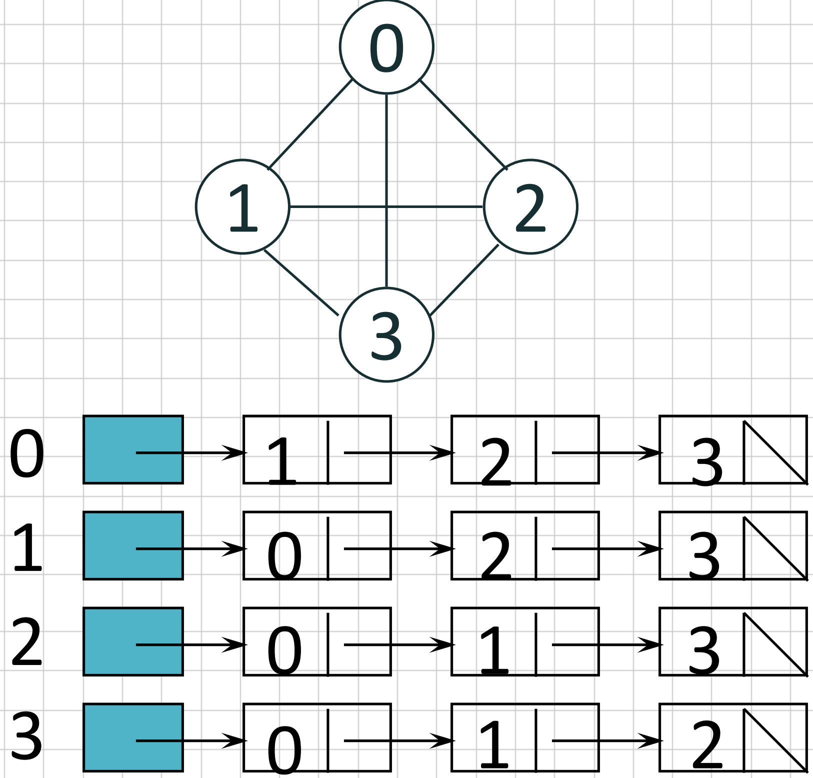
$$outd(vi) = \sum_{j=0}^{n-1} A[i,j] \quad ind(vi) = \sum_{j=0}^{n-1} A[j,i]$$

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

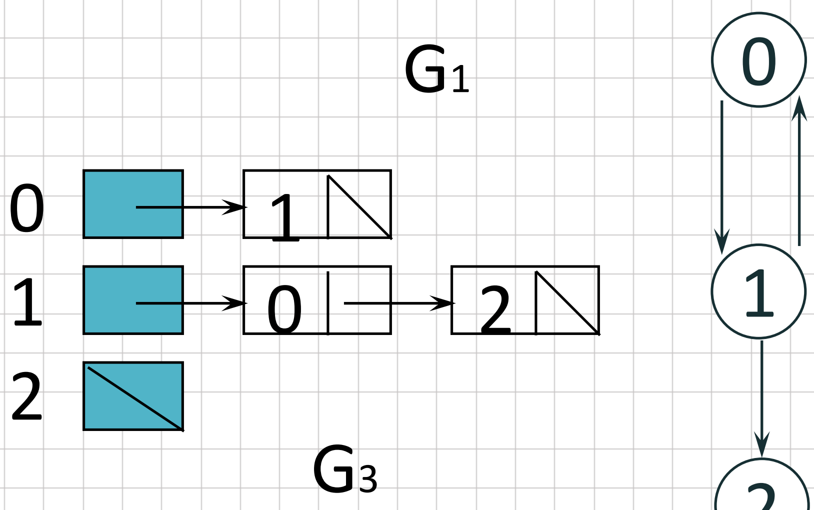
Adjacency Lists

Each row in adjacency matrix is represented as an adjacency list.

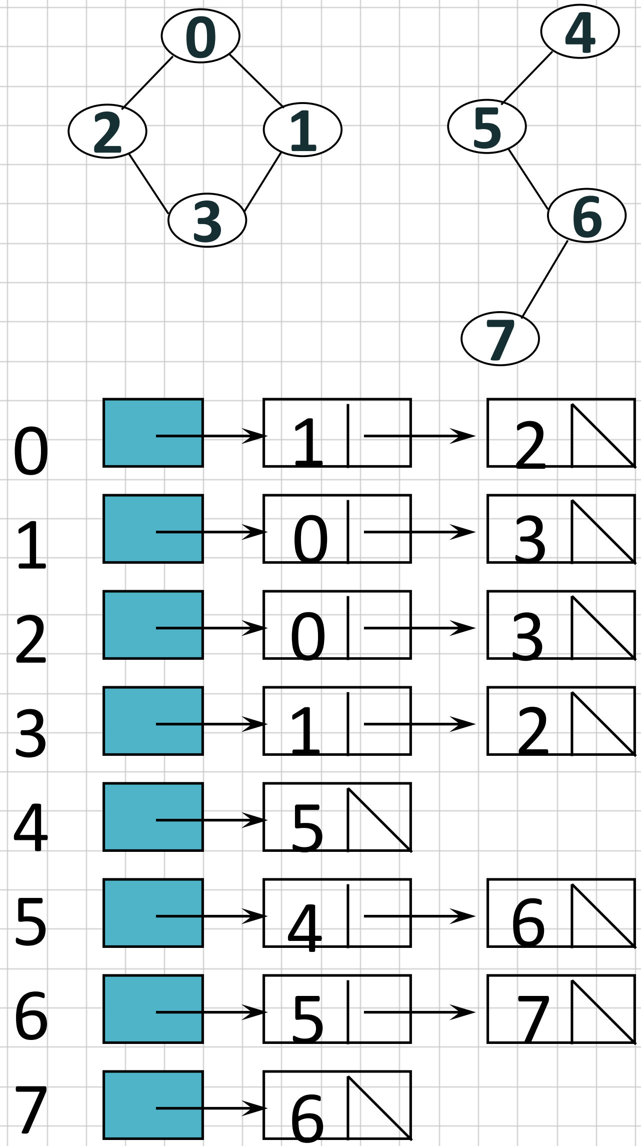
```
#define MAX_VERTICES 50
typedef struct node *node_pointer;
typedef struct node {
    int vertex;
    struct node *link;
};
node_pointer graph[MAX_VERTICES];
int n=0; /* vertices currently in use */
```



G_1



G_3



G_4

Space complexity: O(n + 2e)

An **undirected graph** with n vertices and e edges ==> n head nodes and 2e list nodes

Sequential Representation of Graph G4

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

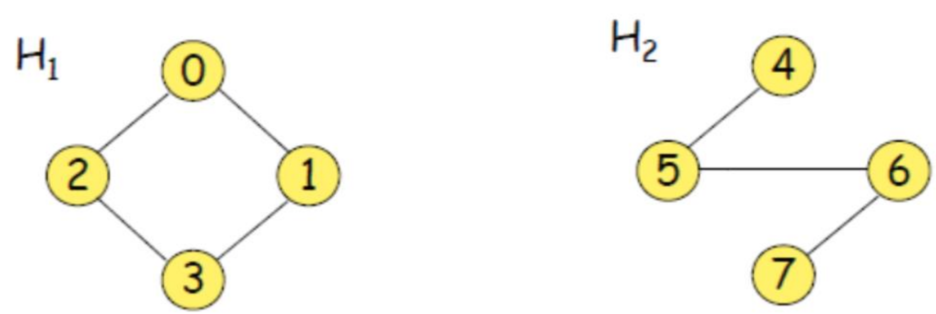
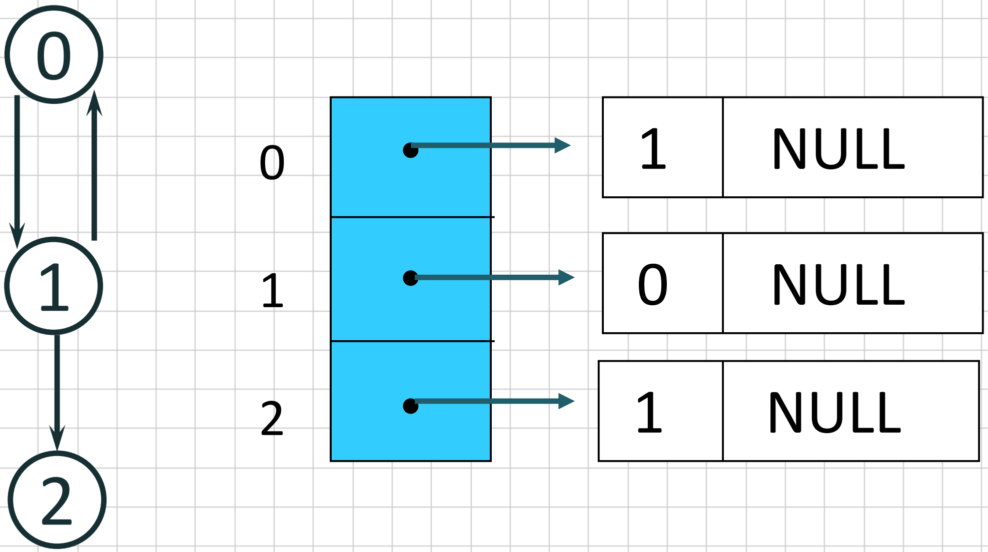


Figure 6.10: Inverse adjacency list for G_3



Determine in-degree of a vertex in a fast way.

Adjacency Multilists

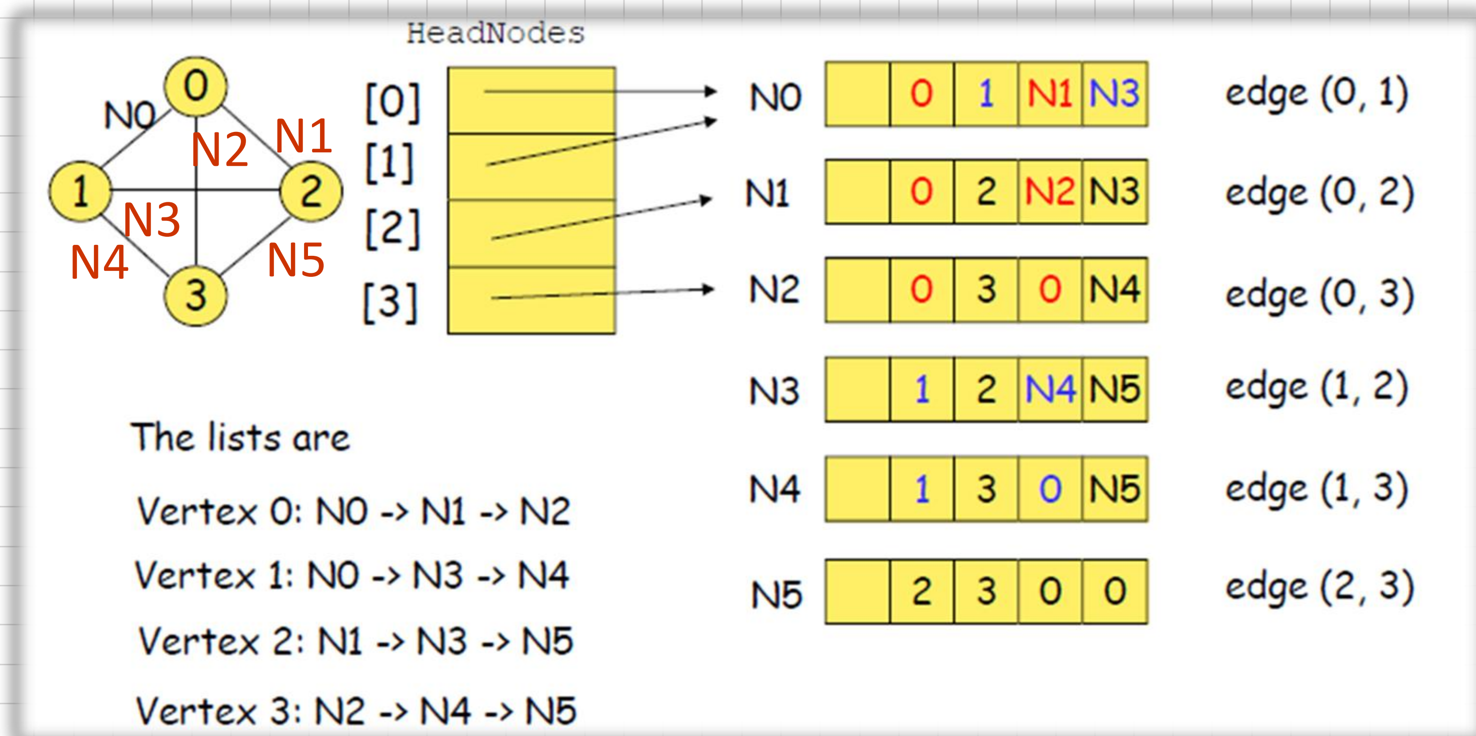
- An edge in an undirected graph is represented by two nodes in adjacency list representation.
- Adjacency Multilists

  lists in which nodes may be shared among several lists.

  (an edge is shared by two different paths)

marked	**vertex1**	**vertex2**	**path1**	**path2**

Example for Adjacency Multlists



Adjacency Multilists

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

marked	vertex1	vertex2	path1	path2

Some Graph Operations

- Traversal

  Given G=(V,E) and vertex v, find all w∈V, such that w connects v.

  - Depth First Search (DFS)
    preorder tree traversal
  - Breadth First Search (BFS)
    level order tree traversal
- Connected Components
- Spanning Trees

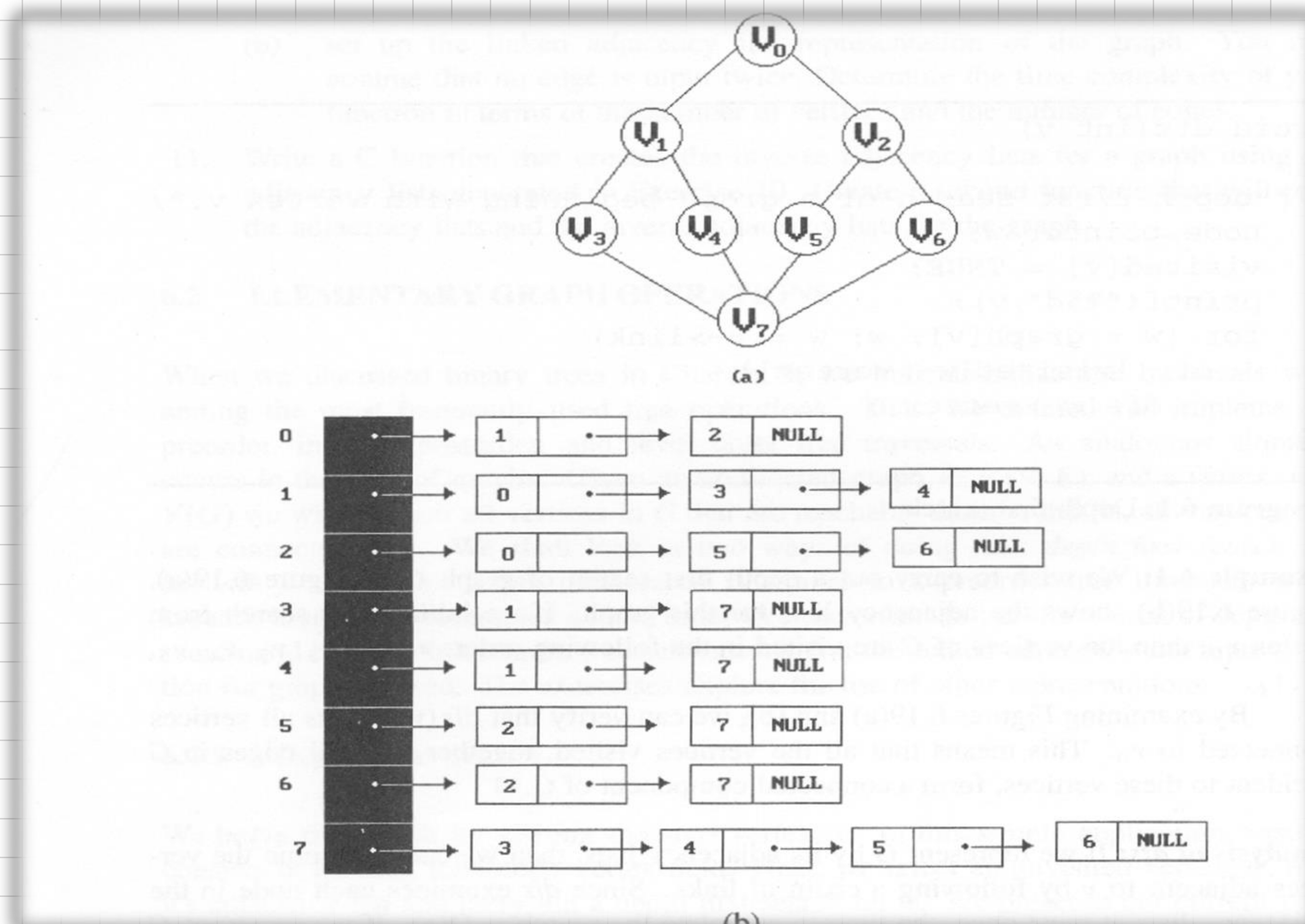
Elementary Graph Operations

Chapter 6.2

Page 279

Figure 6.16: Graph *G* and its adjacency lists (p.281)

depth first search: v0, v1, v3, v7, v4, v5, v2, v6



breadth first search: v0, v1, v2, v3, v4, v5, v6, v7

Depth First Search

```
void dfs(int 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);
}
```

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

Time complexity:

adjacency list: $O(e)$

adjacency matrix: $O(n^2)$

Breadth First Search

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

Breadth First Search (Continued)

```
void bfs(int v)
{
  node_pointer w;
  queue_pointer front, rear;
  front = rear = NULL;
  printf("%5d", v);
  visited[v] = TRUE;
  addq(&front, &rear, v);
```

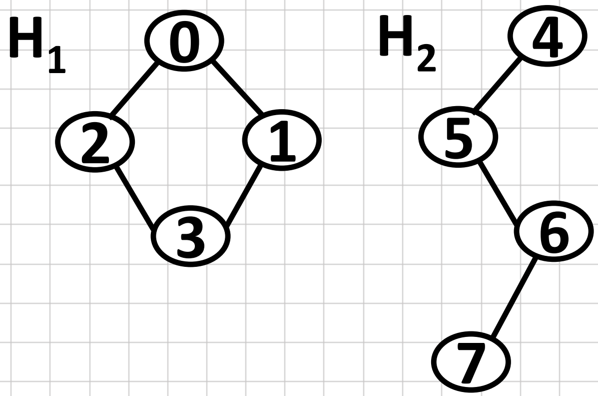
```
  while (front) {
    v= deleteq(&front);
    for (w=graph[v]; w; w=w->link)
      if (!visited[w->vertex]) {
        printf("%5d", w->vertex);
        addq(&front, &rear, w->vertex);
        visited[w->vertex] = TRUE;
      }
  }
}
```

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

Connected Components

- Determine a graph is connected by calling DFS or BFS and checking if there is any unvisited vertex

```
void connected(void)
{
    for (i=0; i<n; i++) {
        if (!visited[i]) {
            dfs(i);
            printf("\n");
        }
    }
}
```



G_4 (not connected)

adjacency list: $O(n+e)$
adjacency matrix: $O(n^2)$

Spanning Trees

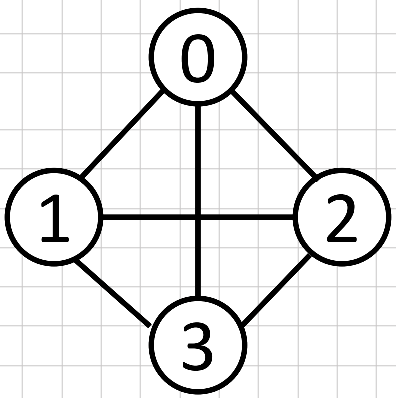
- When graph G is connected, a depth first or breadth first search starting at any vertex will visit all vertices in G
- A spanning tree is any tree that consists solely of edges in G and that includes all the vertices
- E(G): T (tree edges) + N (nontree edges)

where

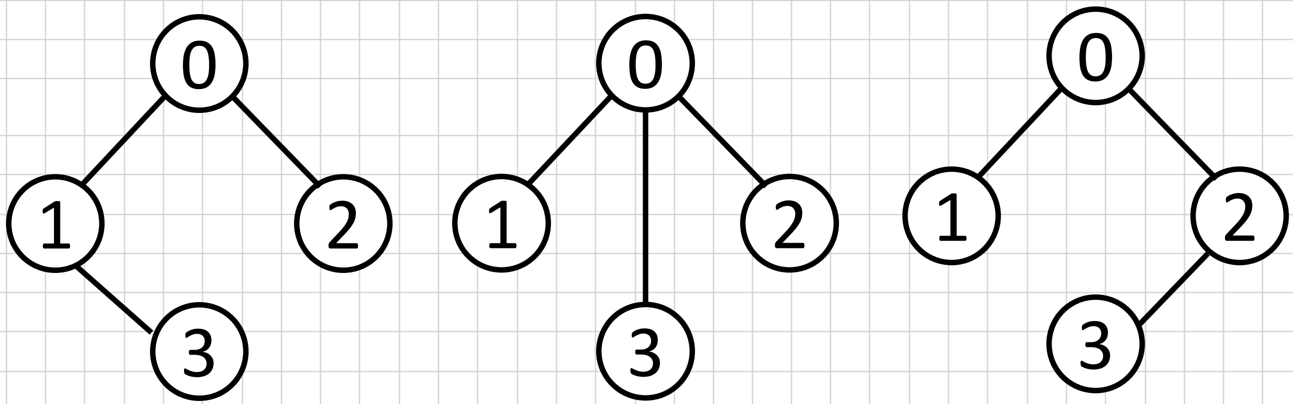
T: set of edges used during search

N: set of remaining edges

Examples of Spanning Tree



G_1



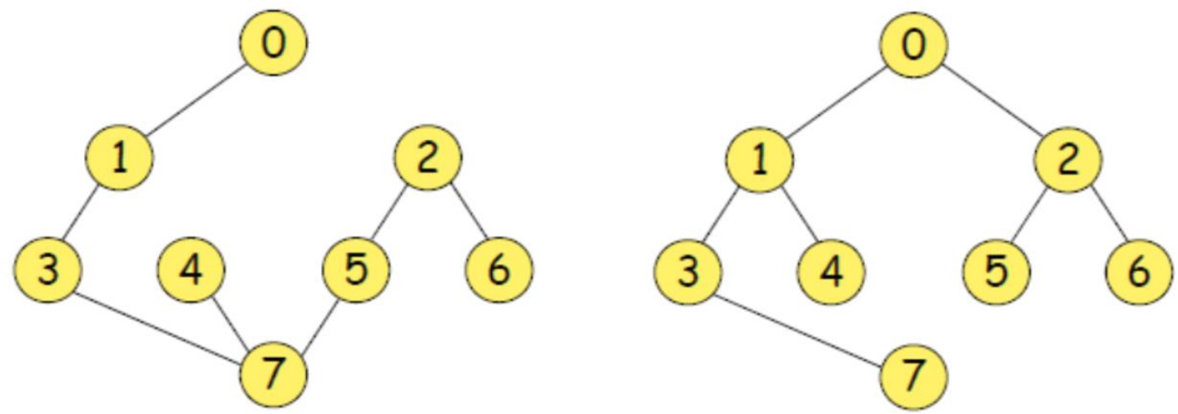
Possible spanning trees

Spanning Trees

- Either dfs or bfs can be used 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 known as a **breadth first spanning tree**

- While adding a nont-ree edge into any spanning tree, this will create a cycle

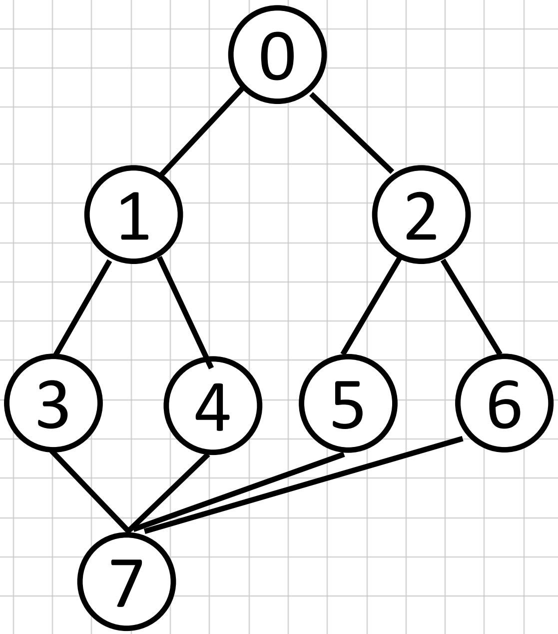
Spanning Trees



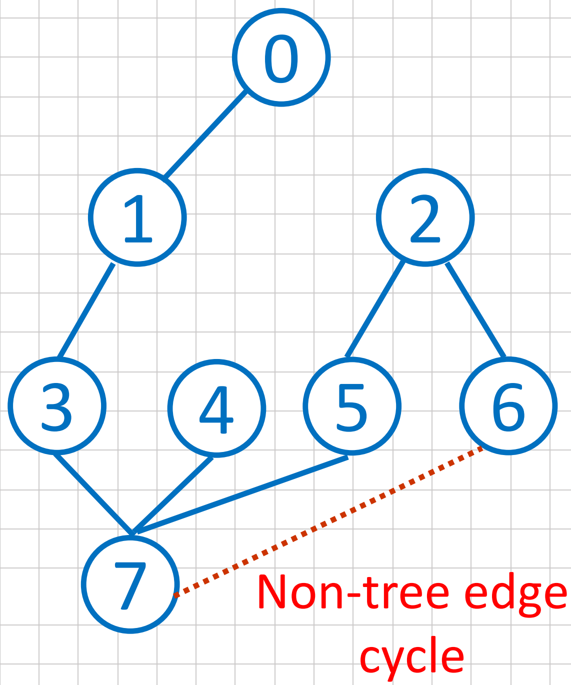
(a) DFS (0) spanning tree

(b) BFS (0) spanning tree

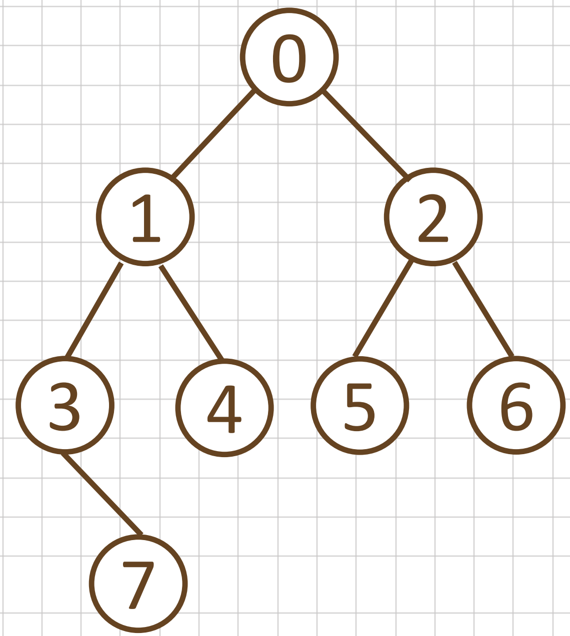
DFS vs BFS Spanning Tree



DFS Spanning



BFS Spanning



Minimum Cost Spanning Trees

Chapter 6.3

Page 292

- The cost of a spanning tree of a weighted undirected graph is the **sum of the costs of the edges** in the spanning tree
- A minimum cost spanning tree is a spanning tree of least cost
- Three different **greedy** algorithms can be used
  - Kruskal
  - Prim
  - Sollin

Select n-1 edges from a weighted graph of n vertices with minimum cost.

Greedy Strategy

- An optimal solution is constructed in stages
- At each stage, the best decision is made at this time
- Since this decision cannot be changed 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 a least cost or a highest profit criterion

Kruskal's Algorithm

Page 292

Kruskal's Idea

- Build a minimum cost spanning tree T by adding edges to T one at a time
- Select the edges for inclusion in T in ascending order of the cost
- An edge is added to T if it does not form a cycle
- Since G is connected and has $n > 0$ vertices, exactly n-1 edges will be selected

0 $\underline{\ 10\ }$ 5

2 $\underline{\ 12\ }$ 3

1 $\underline{\ 14\ }$ 6

1 $\underline{\ 16\ }$ 2

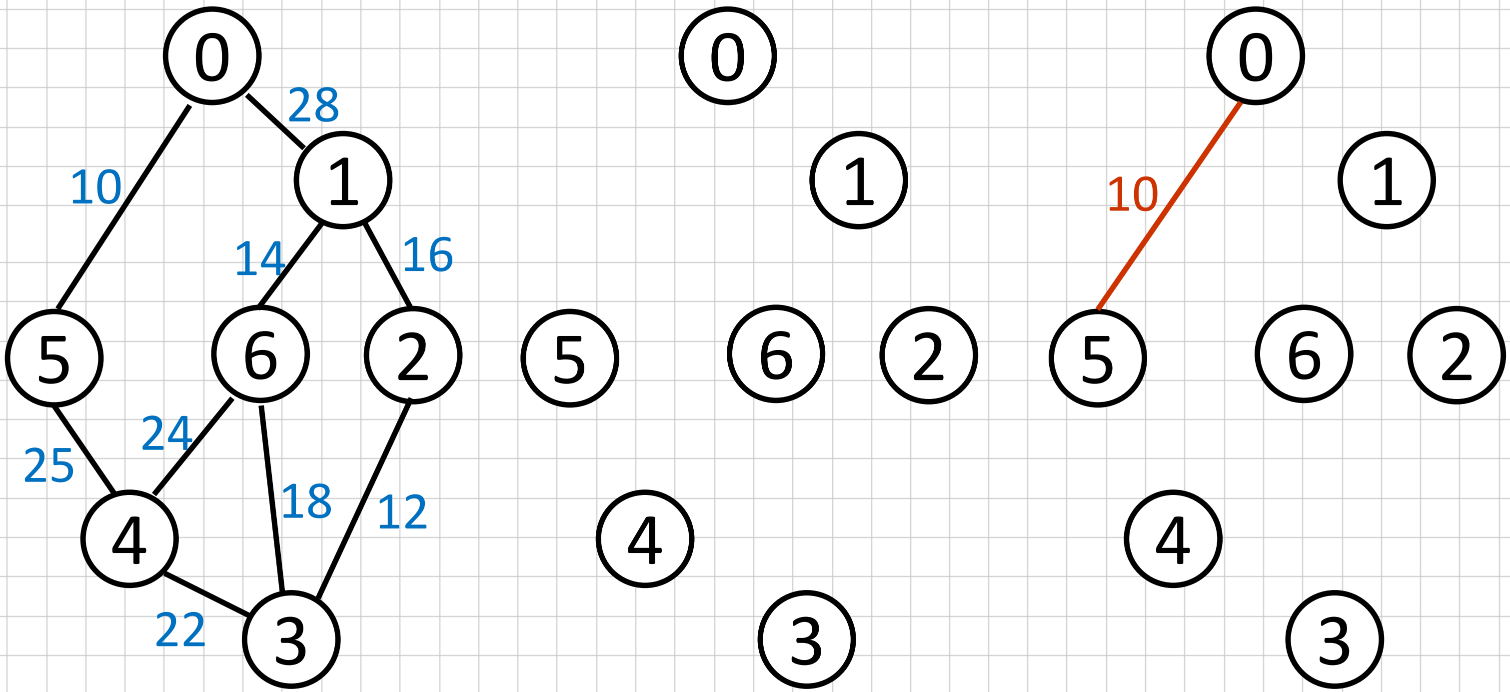
3 $\underline{\ 18\ }$ 6

3 $\underline{\ 22\ }$ 4

4 $\underline{\ 24\ }$ 6

4 $\underline{\ 25\ }$ 5

0 $\underline{\ 28\ }$ 1



6/9

0 —10— 5

2 —12— 3

1 —14— 6

1 —16— 2

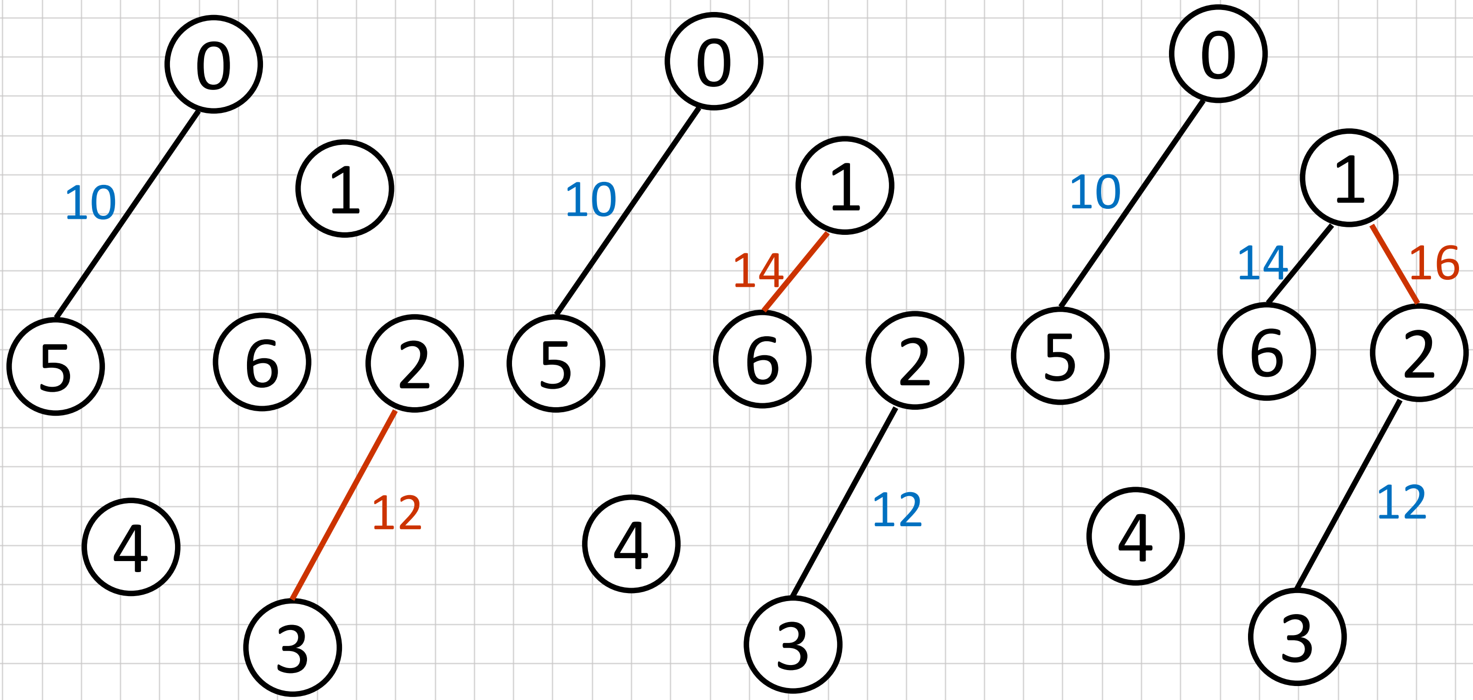
3 —18— 6

3 —22— 4

4 —24— 6

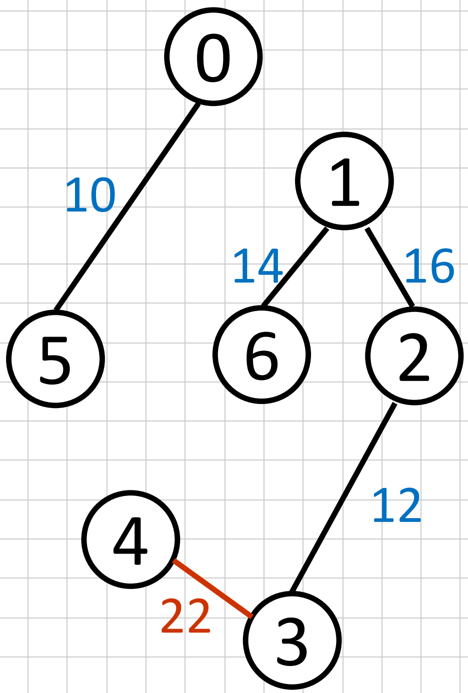
4 —25— 5

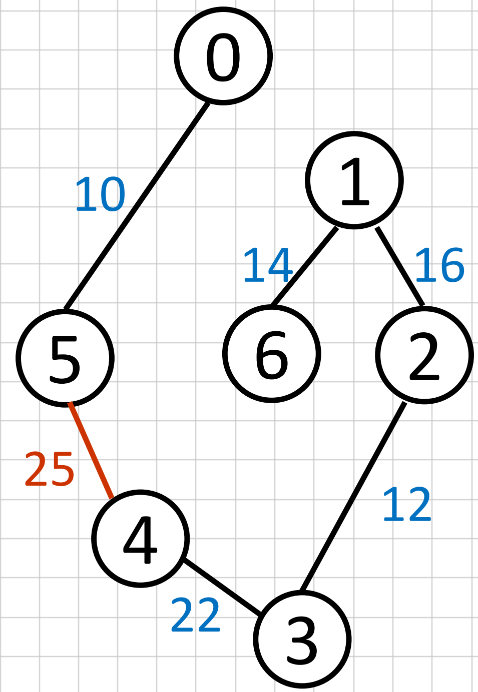
0 —28— 1



add 3——6 有 cycle

0	10	5
2	12	3
1	14	6
1	16	2
3	18	6
3	22	4
4	24	6
4	25	5
0	28	1





cost = 10 +25+22+12+16+14

Kruskal's Algorithm 目標：取出n-1條edges

```
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");
```

min heap construction time O(e)

choose and delete O(log e)

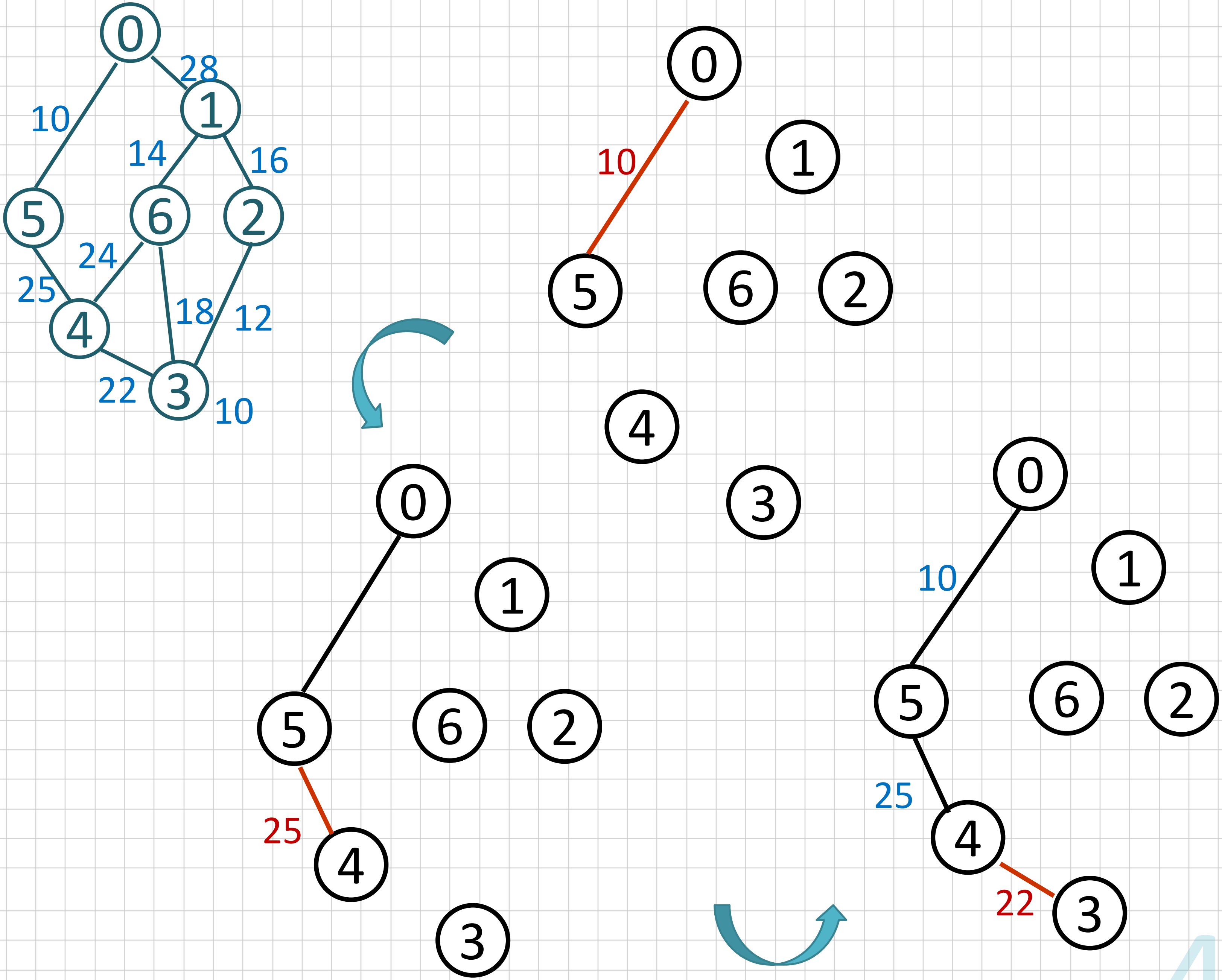
find & union O(log e)

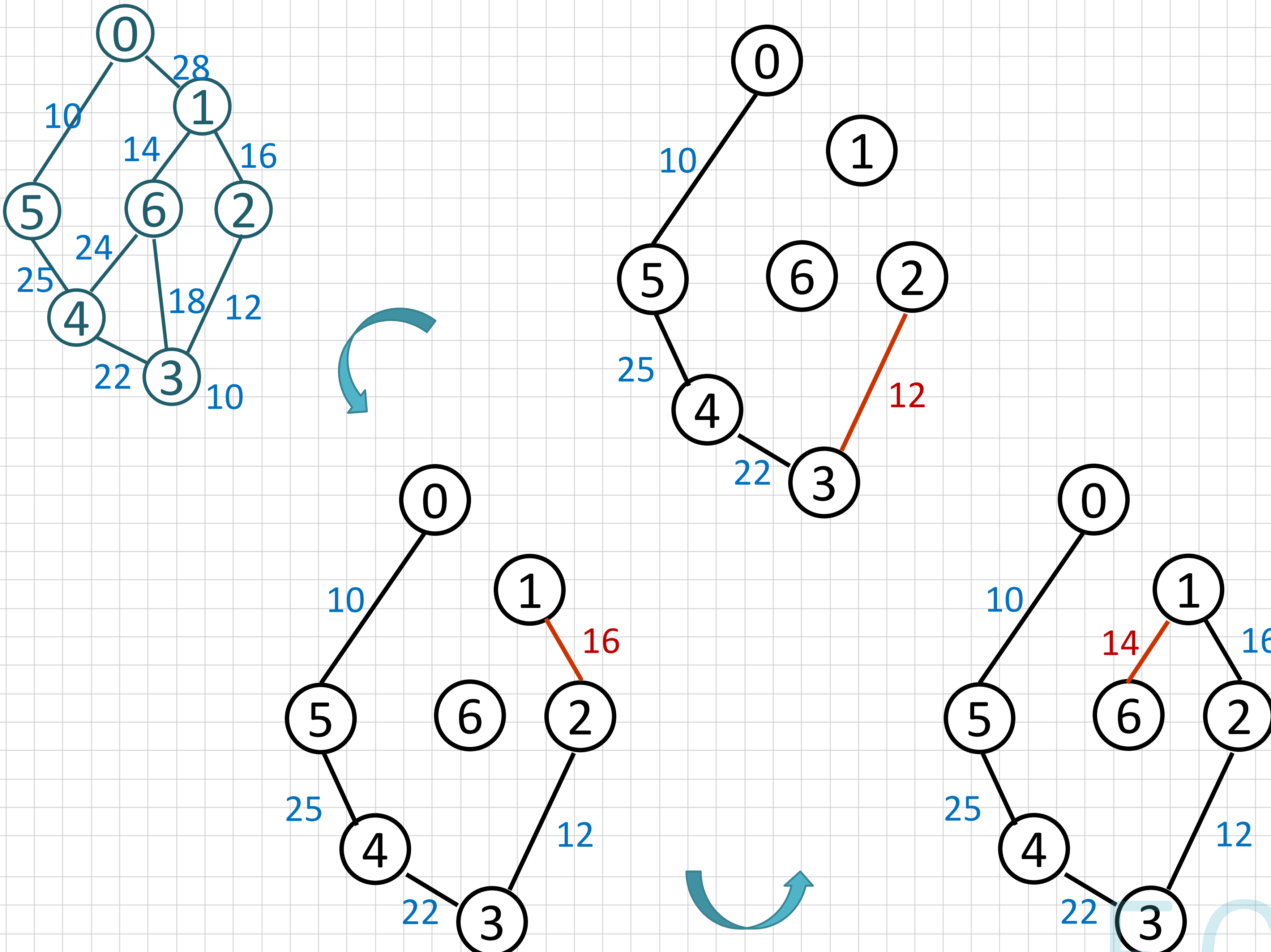
{0,5}, {1,2,3,6}, {4} + edge(3,6) X + edge(3,4) --> {0,5},{1,2,3,4,6}

O(e log e)

Prim's Algorithm

Page 296


0
28
1
10
14
16
5
6
2
24
25
18
12
4
22
3
10
0
10
1
5
6
2
4
3
0
1
5
6
2
25
4
3
0
10
1
5
6
2
25
4
22
3

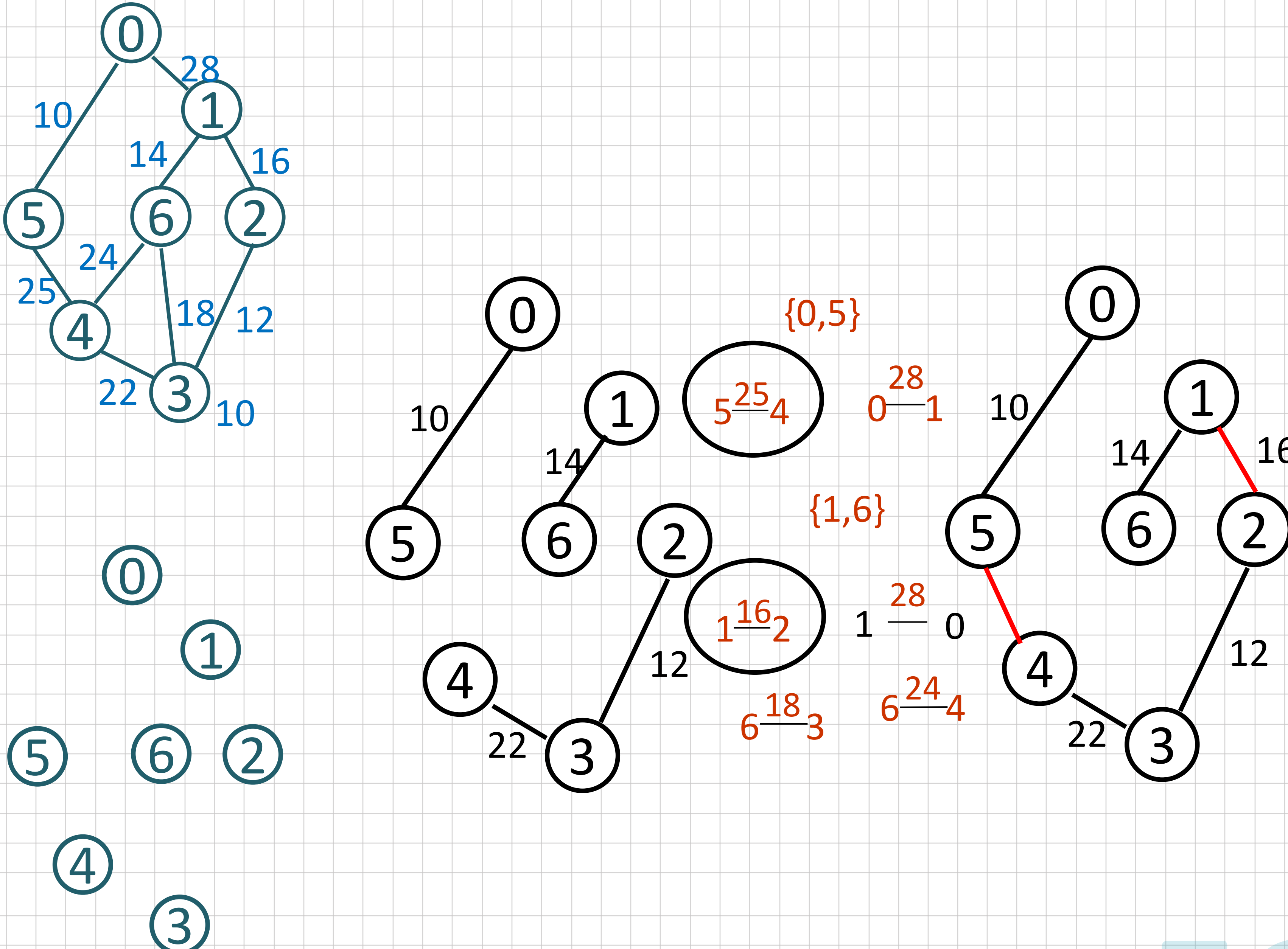

0
28
1
10
14
16
5
6
2
24
25
4
18
12
22
3
10

Prim's Algorithm (tree all the time vs. forest)

```
T={};
TV={0};
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");
```

Sollin’s Algorithm

Page 297



{0,5}
5 25 4
0 28 1
{1,6}
1 16 2
1 28 0
6 18 3
6 24 4

Kruskal

以edge為依據，每回合都自剩下的edge中選出最小者

每一次選edge，都要檢查是否會形成cycle

Prim

先選一個最小的edge

以verticx為依據，選出它所有edges中最小的那一個

~~每一次選edge，都要檢查是否會形成cycle~~

Sollin

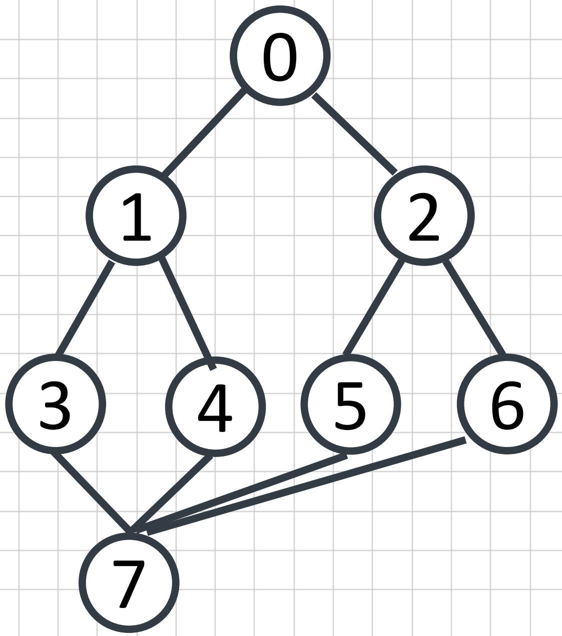
針對每一個vertex，選出它所有edges中最小的那一個

得到k個component

再從剩下的edge中找出(k – 1)個最小的edge

A biconnected graph :

a connected graph that has no **articulation points**.



biconnected graph

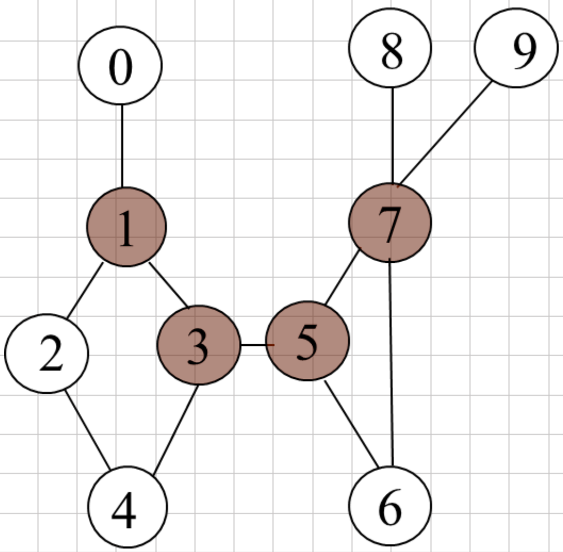
Chapter 6.2.5

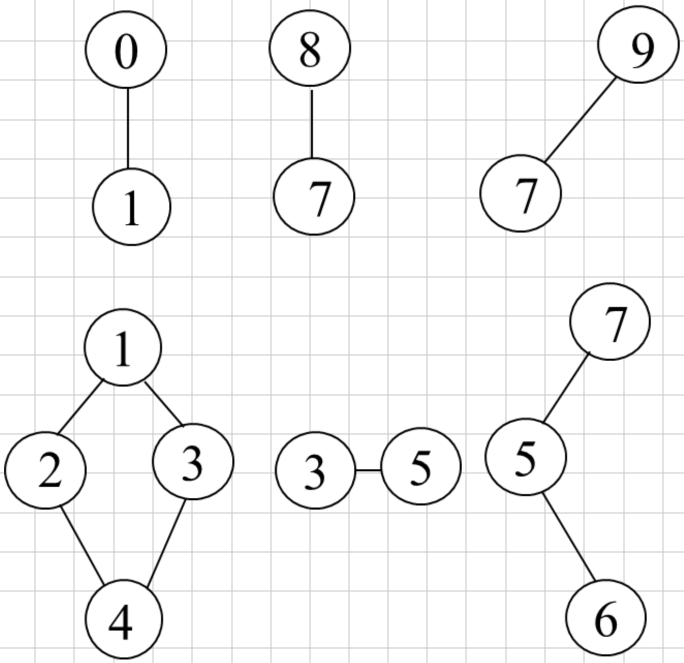
Page 286

biconnected component

a maximal connected subgraph H of G

no other subgraph that is both biconnected and properly contains H



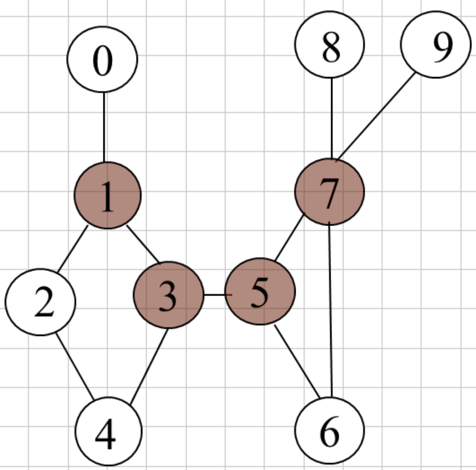


biconnected components

Find biconnected component of a connected undirected graph

by depth first spanning tree

Dfn: depth first number



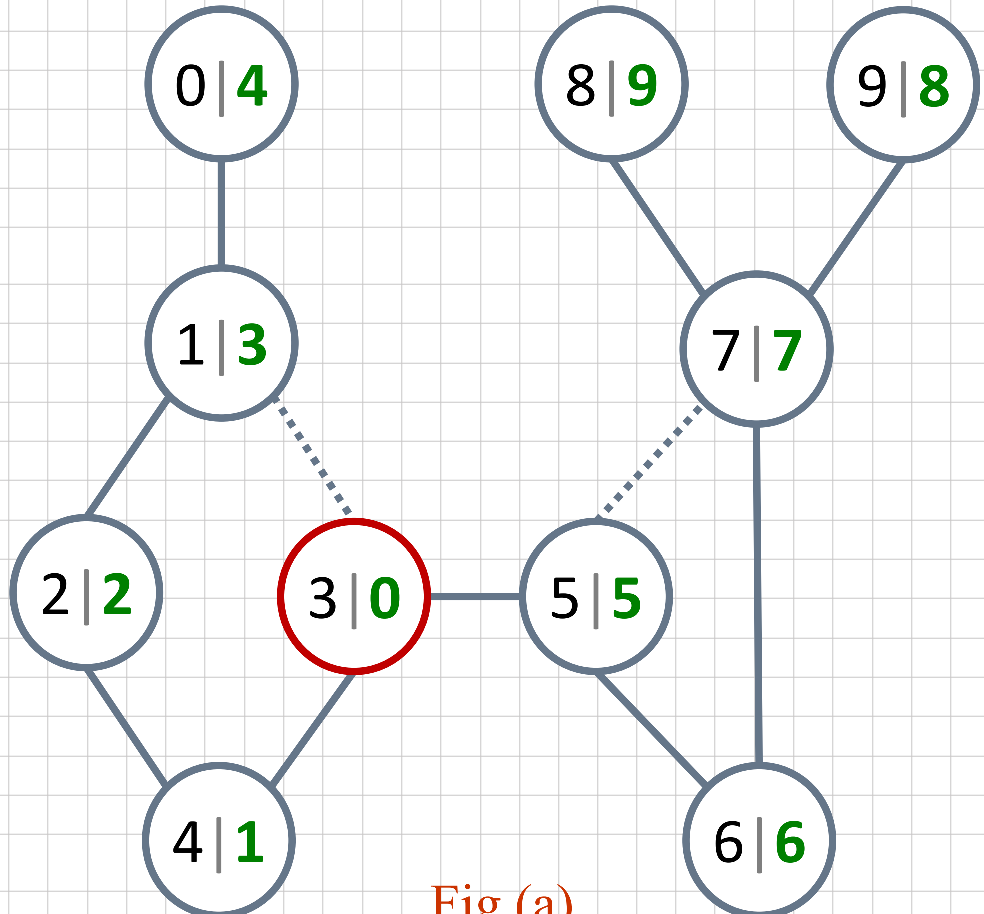
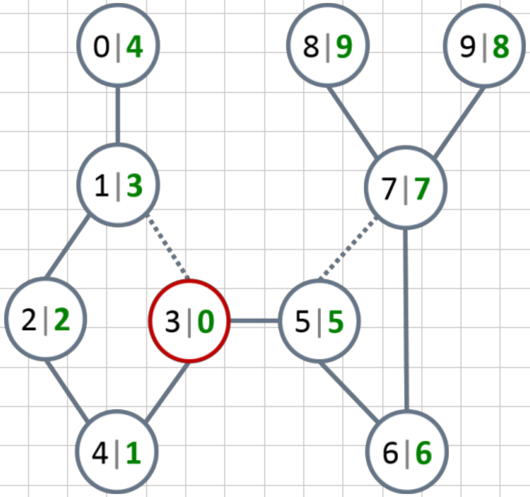


Fig (a)

depth first spanning tree

If u is an ancestor of v then dfn(u) < dfn(v).



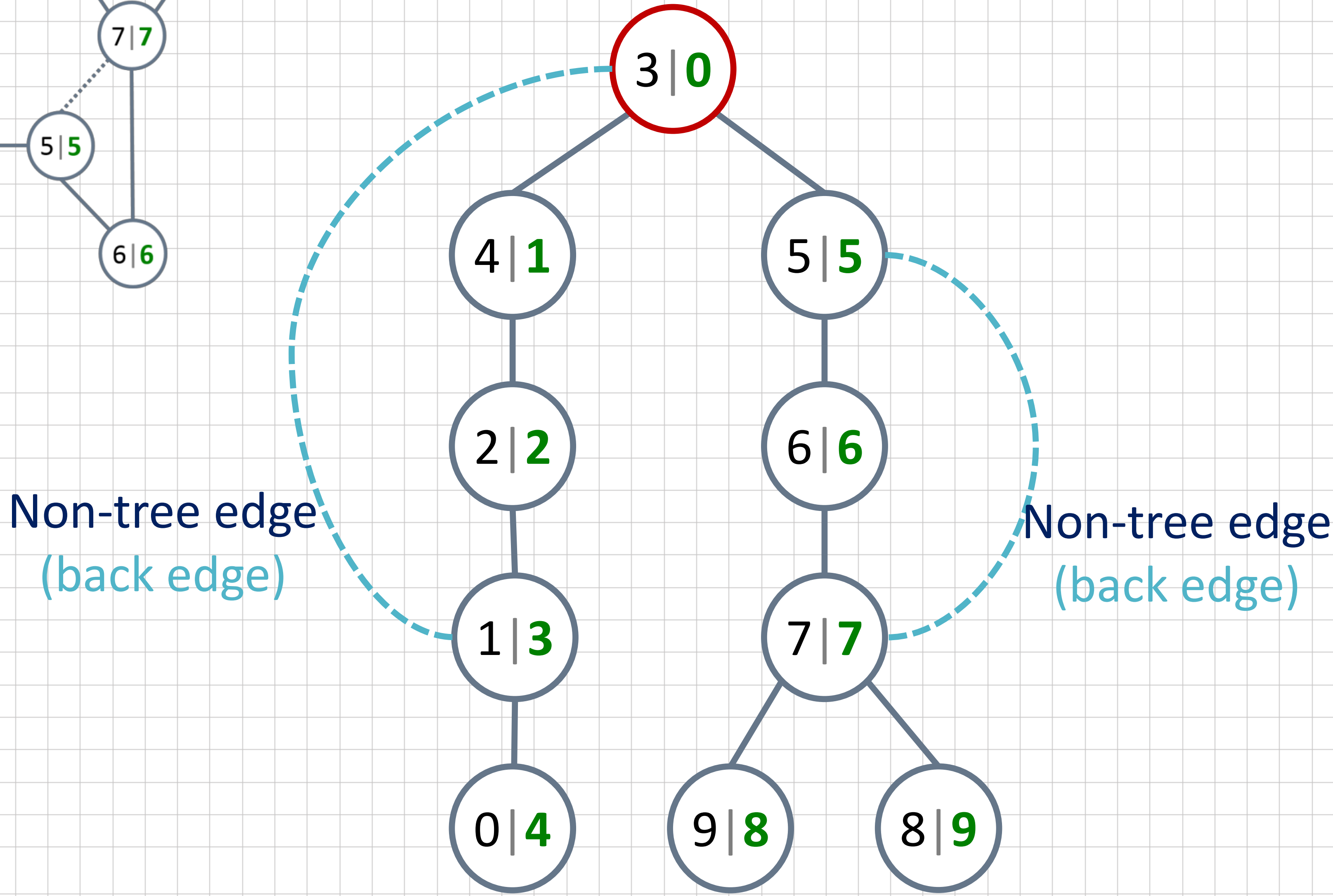


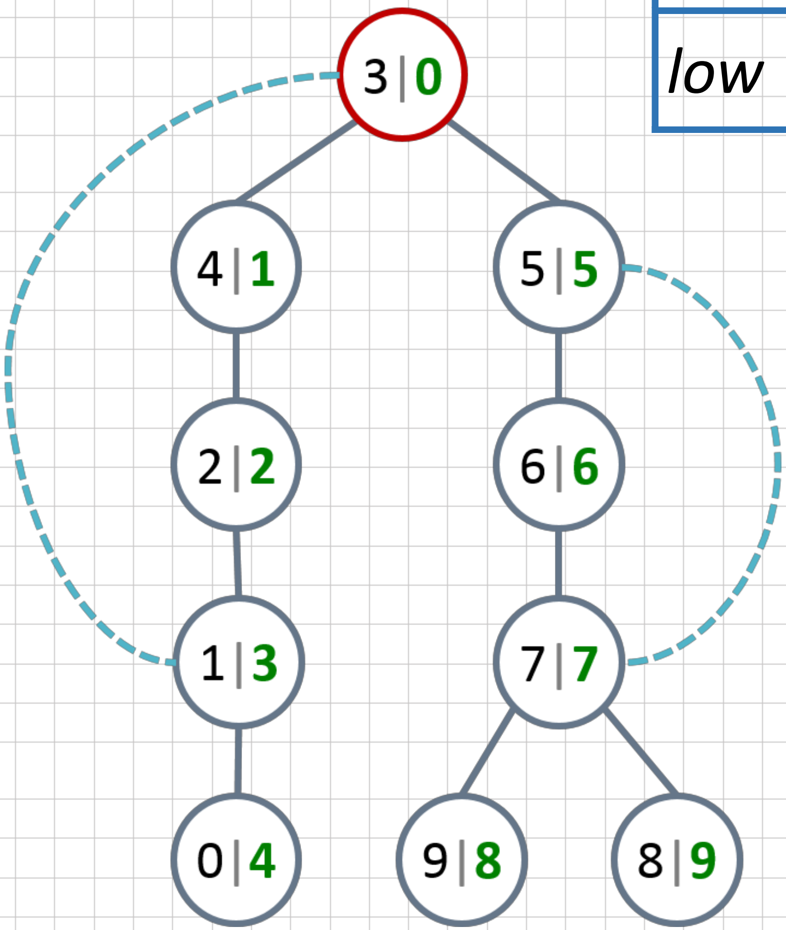
Fig (b)

Biconnected Components

- The root of the depth-first spanning tree is an articulation point iff it has at least two children.
- Define low(w) as the **lowest depth-first number** that can be reached from w using a path of descendants followed by, **at most**, one back edge.

Figure 6.21: *dfn* and *low* values for *dfs* spanning tree with *root* =3(p.288)

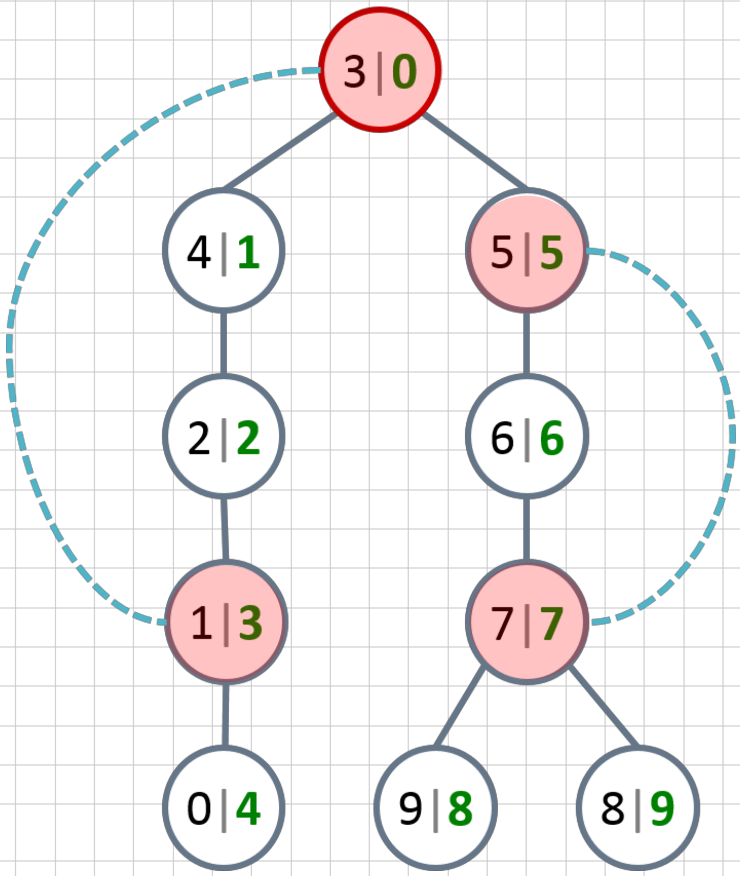
Vertax	0	1	2	3	4	5	6	7	8	9
dfn	4	3	2	0	1	5	6	7	9	8
low	4	0	0	0	0	5	5	5	9	8



請訂正課本錯誤

low(u)=min{dfn(u),
min{low(w)|w is a child of u},
min{dfn(w)|(u,w) is a back edge}

u: articulation point
low(child) ≥ dfn(u)

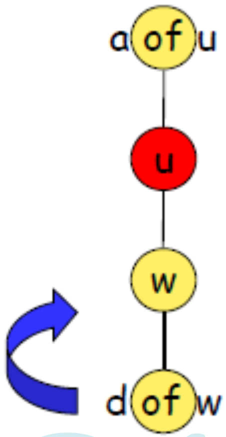


*The root of a depth first panning tree is an articulation point iff it has at least two children.

*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

(1) only w
(2) descendants of w
(3)single back edge.

low(w) >= dfn (u)



Program 6.5: Initializaiton of *dfn* and *low* (p.289)

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

Program 6.4: Determining *dfn* and *low* (p.289)

Initial call: dfn(x,-1)

```
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] );
    }
}
```

low[u]=min{dfn(u), …}

v

u

w

v

u

low[u]=min{…, min{low(w)|w is a child of u}, …}

dfn[w]≠0 非第一次，表示藉back edge

O X

low[u]=min{…,…,min{dfn(w)|(u,w) is a back edge}

Program 6.6: Biconnected components of a graph (p.290)

```
void bicon(int u, int v)
{
node_pointer ptr;
    int w, x, y;
    dfn[u] = low[u] = num ++;    low[u]=min{dfn(u), …}
    for (ptr = graph[u]; ptr; ptr = ptr->link) {
      w = ptr ->vertex;                    (1) dfn[w]=-1 第一次
      if ( v != w && dfn[w] < dfn[u] ) (2) dfn[w]!=-1非第一次，藉back edge
         add(&top, u, w);       /* add edge to stack */

 if(dfn[w] < 0) {/* w has not been visited */
         bicon(w, u);  low[u]=min{…, min{low(w)|w is a child of u}, …}
         low[u] = MIN2(low[u], low[w]);
         if (low[w] >= dfn[u] ){ articulation point
            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]);
    }  low[u]=min{…, …, min{dfn(w)|(u,w) is a back edge}}
  }
```

Shortest path and transitive closure

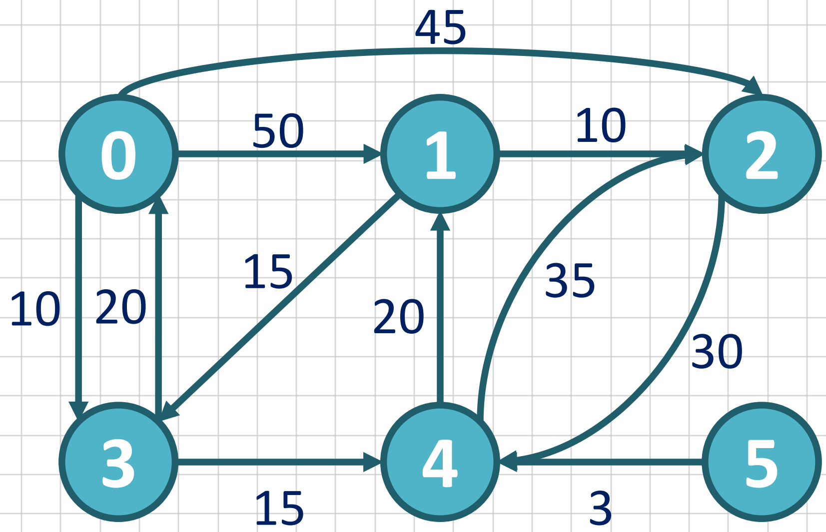
Chapter 6.4

Page 299

- Single Source All Destinations
- Single Source/All Destinations: General Weights
- All Pairs Shortest Paths
- Transitive Closure

Single Source All Destinations

Determine the **shortest paths** from v0 to all the remaining vertices.



	path	*length*
1)	0, 3	10
2)	0, 3, 4	25
3)	0, 3, 4, 1	45
4)	0, 2	45

Figure 6.26: Graph and shortest paths from v_0 (p.300)

Dijkstra Shortest Path Algo.

	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

Cost adjacency matrix

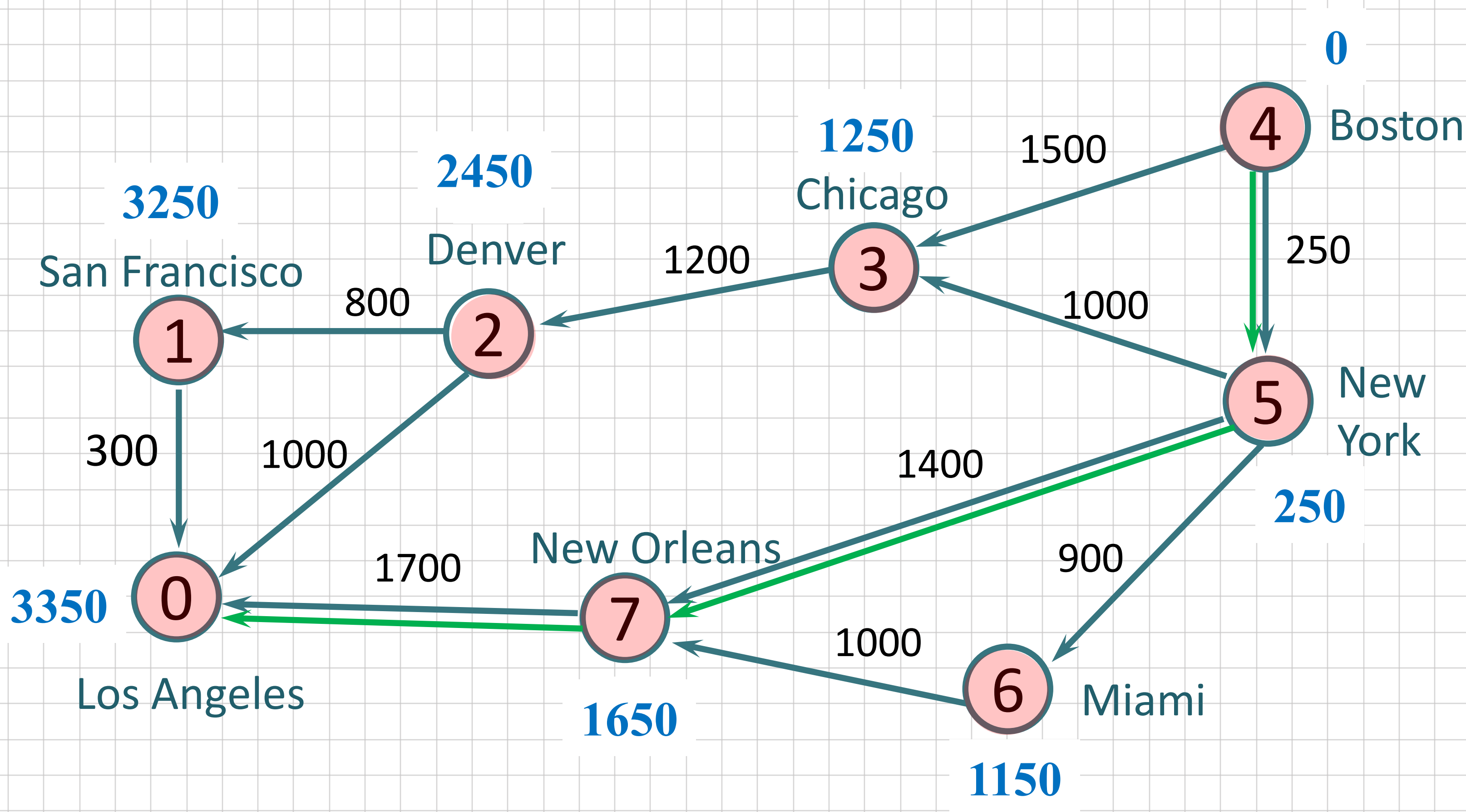
```
/**
 * Dijkstra's algorithm
 * @param d matrix of legths, position [0 1] = 2 means that from node 0 leads an edge to node 1 of length 2
 * @param from root node
 * @return tree an ancestors (denotes path from the node to the root node)
 */
procedure int[] doDijkstra(d, from) {
    //insert all nodes to the priority queue, node from has a distance 0, all others infinity
    Q = InsertAllNodesToTheQueue(d, from)
    CLOSED = {} //closed nodes - empty set
    predecessors = new array[d.nodeCount] //array of ancestors

    while !Q.isEmpty() do
        node = Q.extractMin()
        CLOSED.add(node) //close the node

        //contract distances
        for a in Adj(node) do //for all descendants
            if !CLOSED.contains(a) //if the descendatn was not closed yet
                //and his distance has decreased
                if Q[node].distance + d[node][a] < Q[a].distance
                    //zmen prioritu (vzdalenost) uzlu
                    Q[a].distance = Q[node].distance + d[node][a]
                    //change its ancestor
                    predecessors[a] = node

    return predecessors
```

Dijkstra Shortest Path Algo.

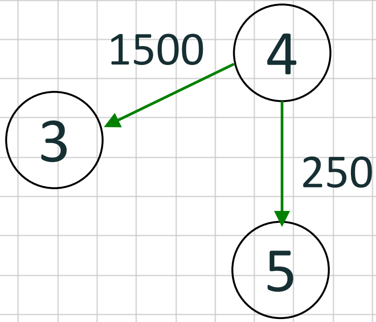


Example for the Shortest Path

(Continued)

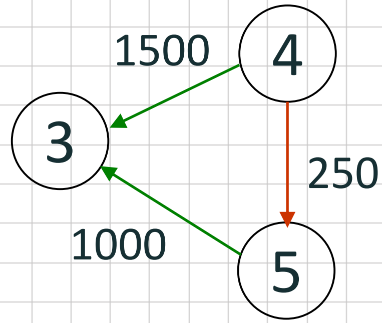
Iteration	S	Vertex Selected	LA [0]	SF [1]	DEN [2]	CHI [3]	BO [4]	NY [5]	MIA [6]	NO [7]
Initial	--	----	+∞	+∞	+∞	1500	0	250	+∞	+∞
1	{4}	(a) 5	+∞	+∞	+∞	(b) 1250	0	250	(c) 1150	(d) 1650
2	{4,5}	(e) 6	+∞	+∞	+∞	1250	0	250	1150	(f) 1650
3	{4,5,6}	(g) 3	+∞	+∞	(h) 2450	1250	0	250	1150	1650
4	{4,5,6,3}	(i) 7	(j) 3350	+∞	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
7	{4,5,6,3,7,2,1}									

(a)



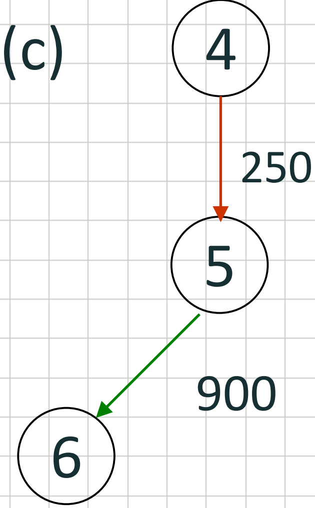
選5

(b)



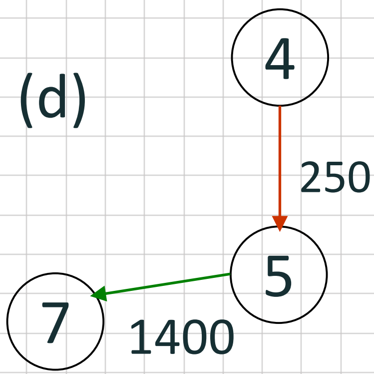
4到3由1500改成1250

(c)



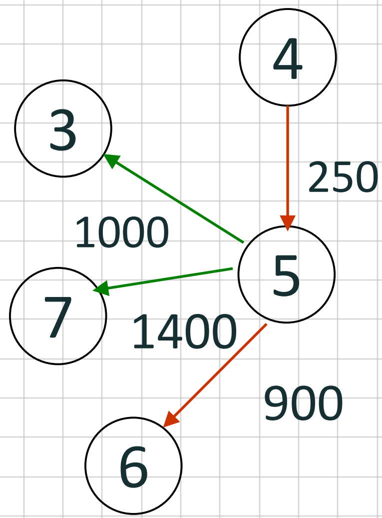
4到6由∞改成1250

(d)



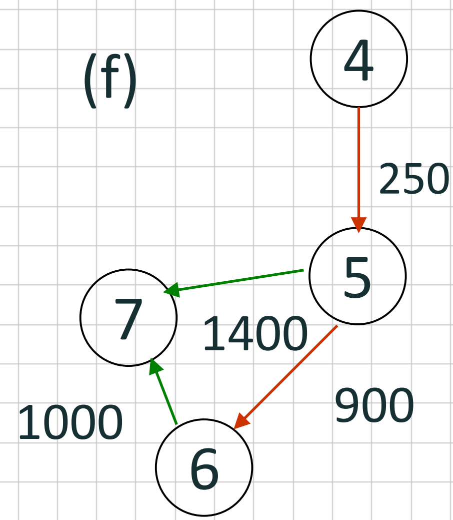
4到7由∞改成1650

(e)



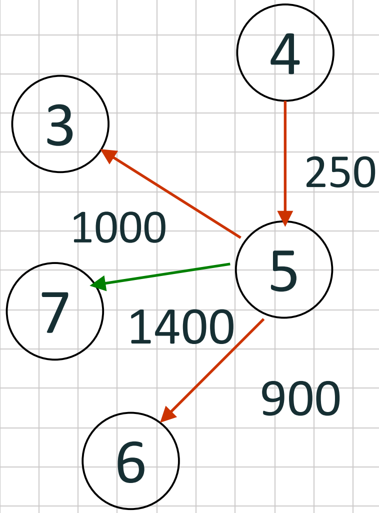
選6

(f)



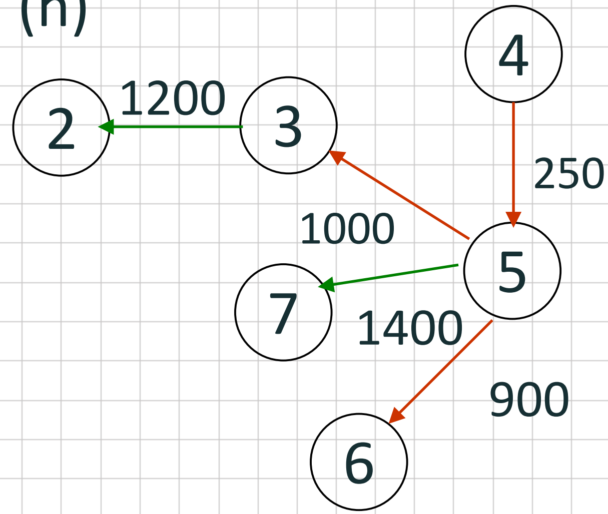
4-5-6-7比4-5-7長

(g)



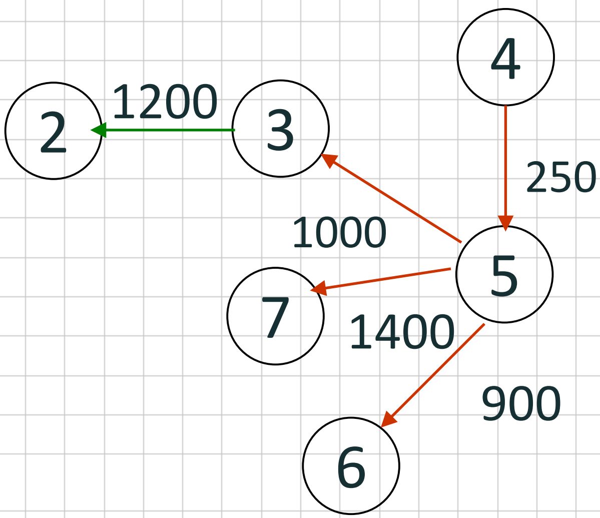
選3

(h)



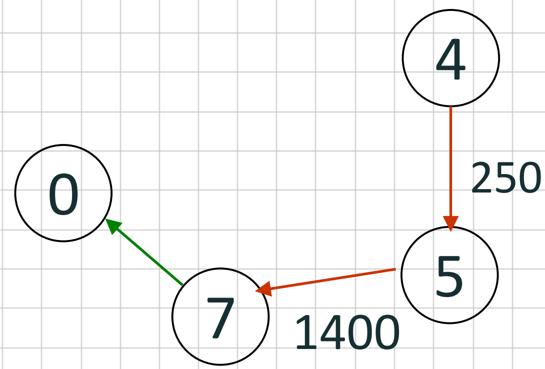
4到2由∞改成2450 (=1250+1200)

(i)



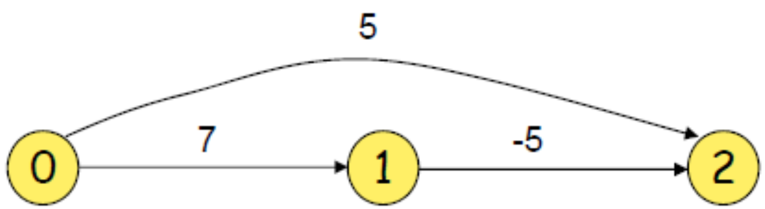
選7

(j)



4到0由∞改成3350 (= 1650+1700)

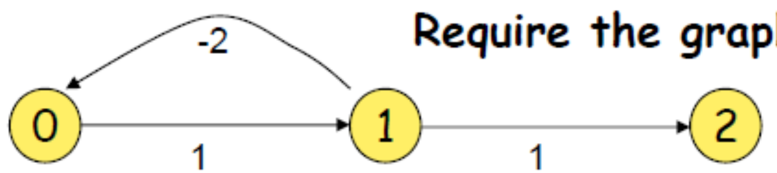
Directed Graphs



S	1	2
	7	5
2	7	5

(a) Directed graph with a negative-length edge

Require the graph has no cycles



(b) Directed graph with a cycle of negative length

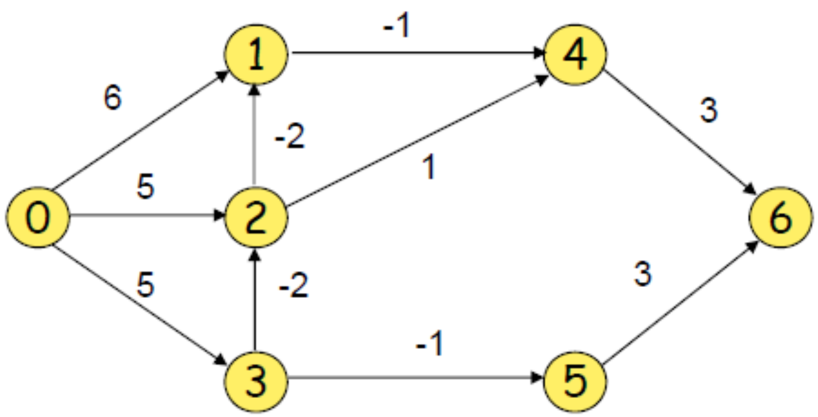
Single Source/All Destinations: General Weights

- When there are no cycles of negative length, there is a shortest path between any two vertices of an n-vertex graph that has at most n-1 edges on it.
  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 i followed by the edge <i, u>. The path from v to i has k – 1 edges, and its length is $dist^{k-1}[i]$.
- The distance can be computed in recurrence by the following:

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

- The algorithm is also referred to as the Bellman and Ford Algorithm.

Shortest Paths with Negative Edge Lengths



(a) A directed graph

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

(b) $dist^k$

All Pairs Shortest Paths

- Find the shortest paths between all pairs of vertices.
- Solution 1

  - Apply Bellman and Ford Algorithm n times with each vertex as source.

    $O(n^3)$

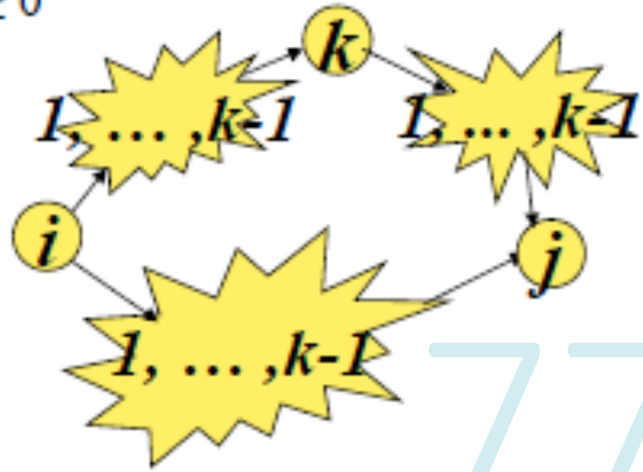
All Pairs Shortest Paths (Continued)

- Solution 2

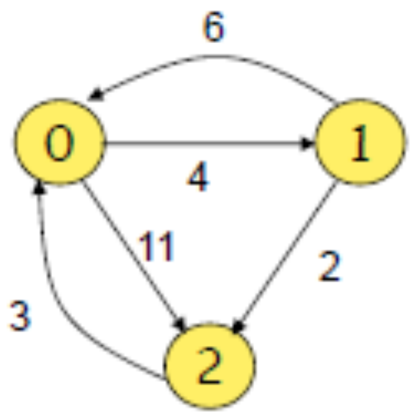
- Notations
  - $A^{-1}[i][j]$: is just the length[i][j]
  - $A^{n-1}[i][j]$: the length of the shortest i-to-j path in G
  - $A^{k}[i][j]$: the length of the shortest path from i to j going through **no intermediate vertex of index greater than k.**

How to determine the value of $A^{k}[i][j]$

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



Example for All-Pairs Shortest-Paths Problem



A	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

(a) A^{-1}

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

(b) A^{0}

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

(c) A^{1}

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

(d) A^{2}

Transitive Closure

Goal:

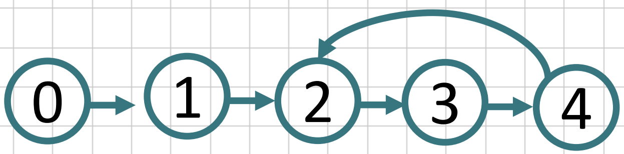
given a graph with unweighted edges, determine if there is a path from i to j for all i and j.

1) Require positive path (> 0) lengths.

   → transitive closure matrix

2) Require nonnegative path (≥0) lengths.

   → reflexive transitive closure matrix



(a) Digraph G

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

(b) Adjacency matrix A for G

$$\begin{array}{c c} 0 & \\ 1 & \\ 2 & \\ 3 & \\ 4 & \end{array}\begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

cycle

(c) transitive closure matrix A^+

There is a path of **length > 0**

$$\begin{array}{c c} 0 & \\ 1 & \\ 2 & \\ 3 & \\ 4 & \end{array}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

reflexive

(d) reflexive transitive closure matrix A^*

There is a path of **length ≥0**

Activity Network

Chapter 6.5

Page 315

- Activity on Vertex (AOV) Network
- Activity on Edge (AOE) Networks

Activity on Vertex (AOV) Network

- Definition

A directed graph in which the vertices represent tasks or activities and the edges represent precedence relations between tasks.

- predecessor (successor)

vertex i is a predecessor of vertex j iff there is a directed path from i to j. j is a successor of i.

- partial order

a precedence relation which is both transitive ($\forall$i, j, k, i•j & j•k => i•k) and irreflexive (no x•x).

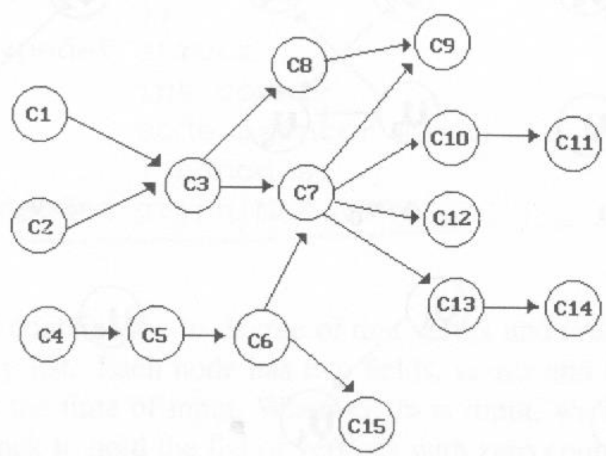
- acylic graph

a directed graph with no directed cycles

Figure 6.37: An AOV network (p.316)

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

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



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

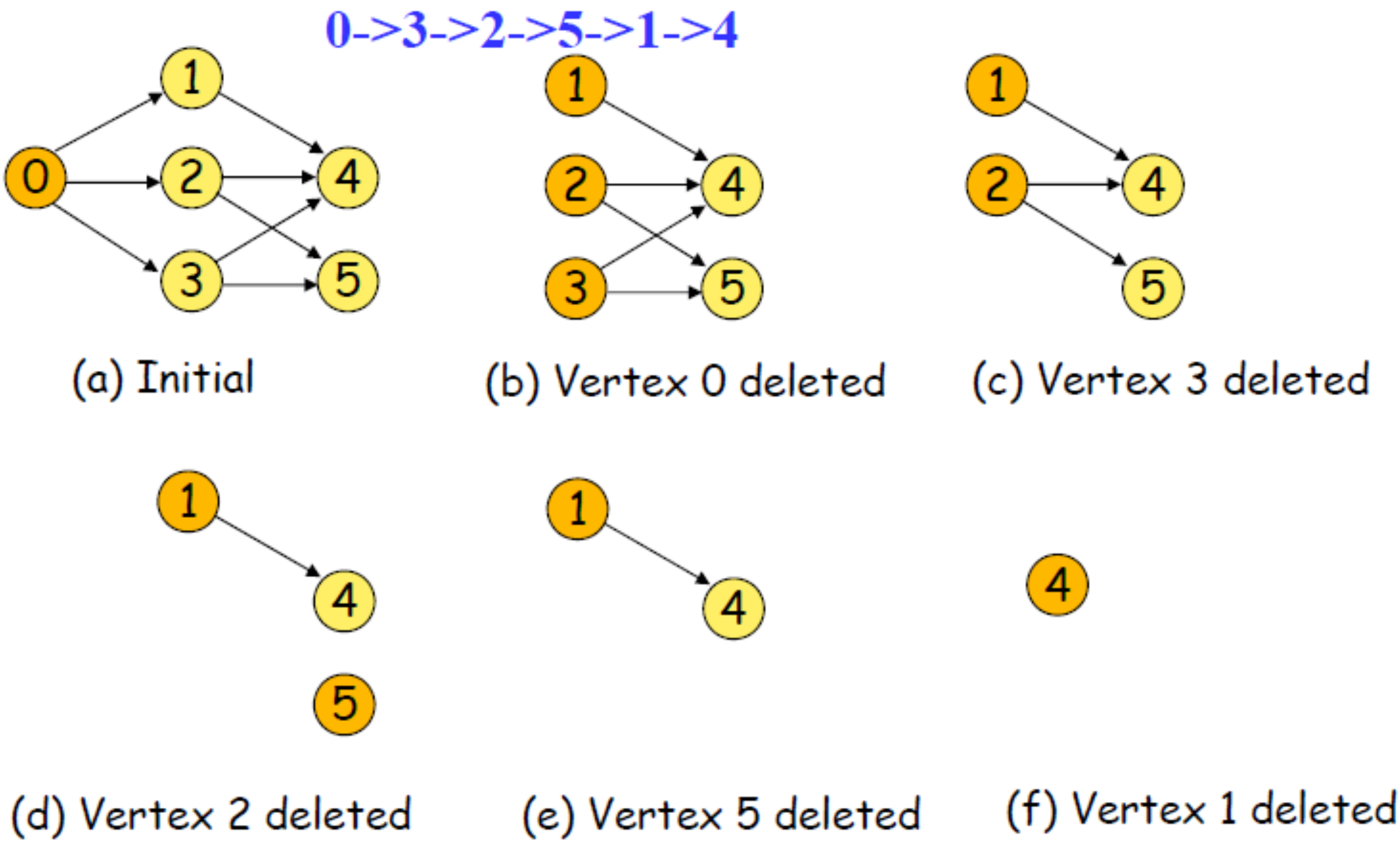
Topological order:
linear ordering of vertices of a graph

$\forall$i, j if i is a predecessor of j, then i precedes j in the linear ordering

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

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

Figure 6.38: Action of Program 6.13 on an AOV network (p.318)



Issues in Data Structure Consideration

- Decide whether a vertex has any predecessors.
  - Each vertex has a count.
- Decide a vertex together with all its incident edges.
  - Adjacency list

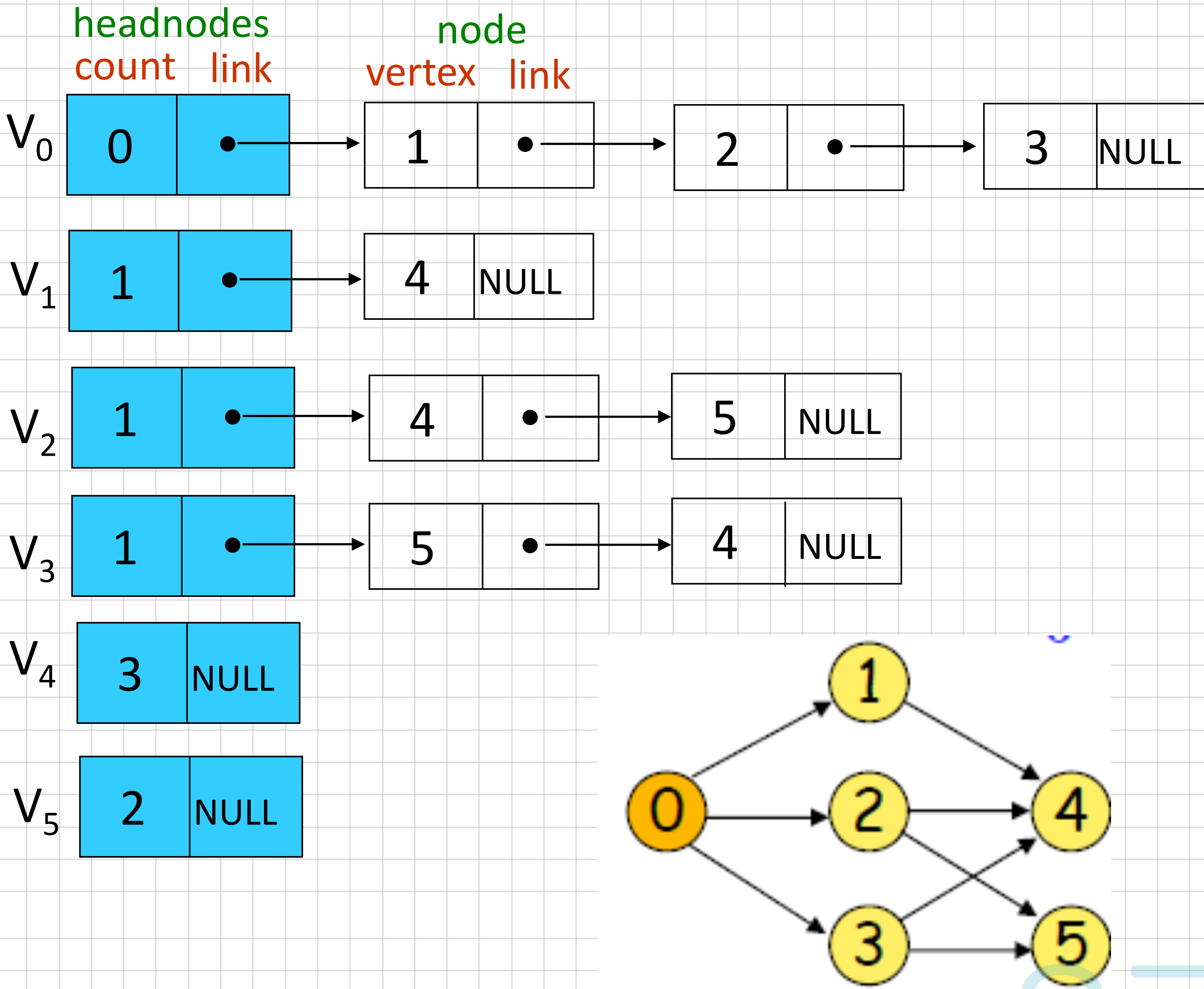
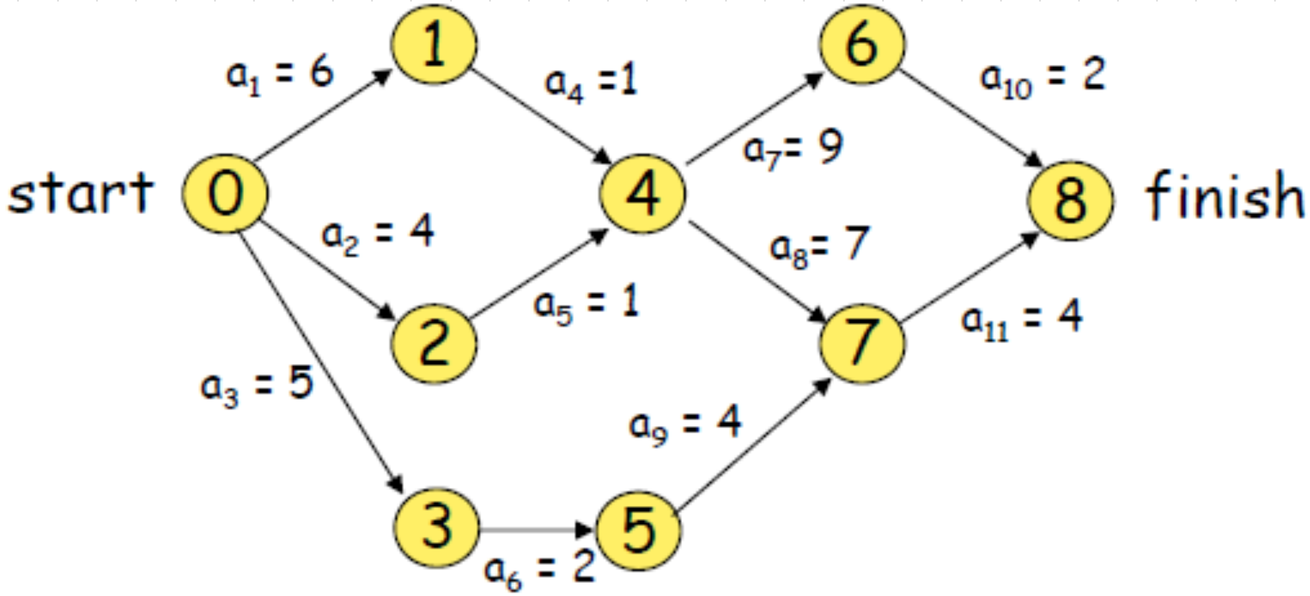


Figure 6.40: Adjacency list representation of Figure 6.30(a) (p.309)

Activity on Edge (AOE) Networks

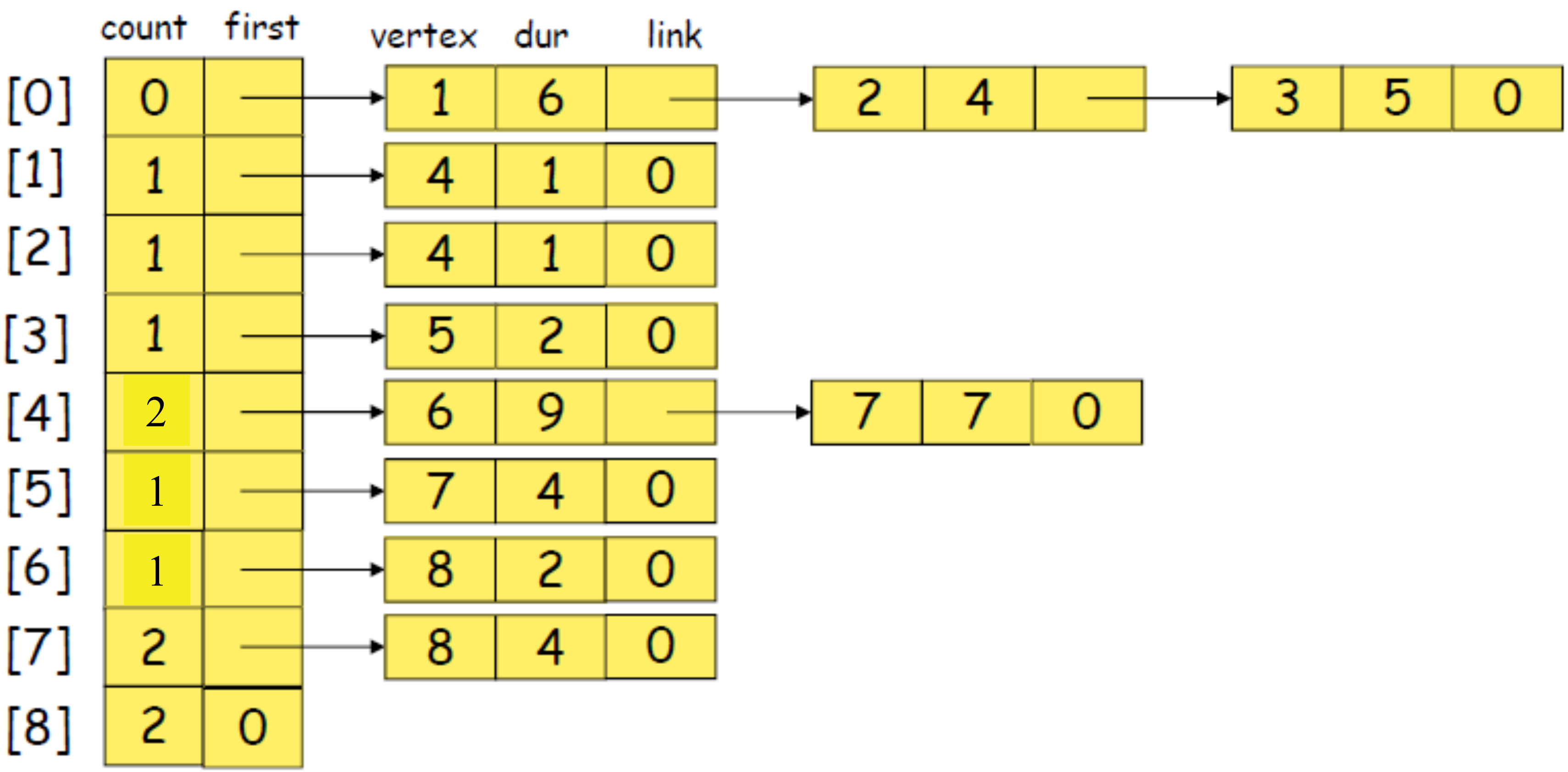
- directed edge
  - –tasks or activities to be performed
- vertex
  - –events which signal the completion of certain activities
- number on an edge
  - –time required to perform the activity

*Figure 6.40: An AOE network(p.322)

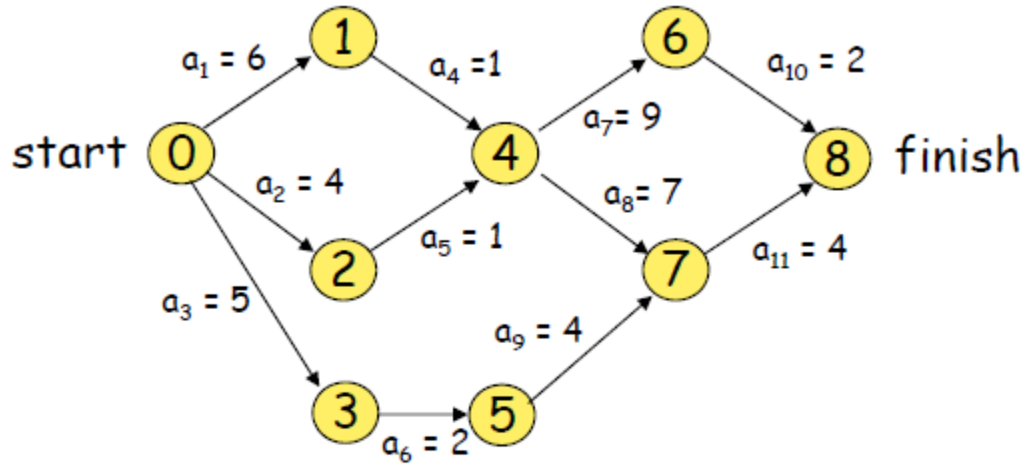


event	interpretation
0	Start of project
1	Completion of activity a_1
4	Completion of activities a_4 and a_5
7	Completion of activities a_8 and a_9
8	Completion of project

Adjacency lists



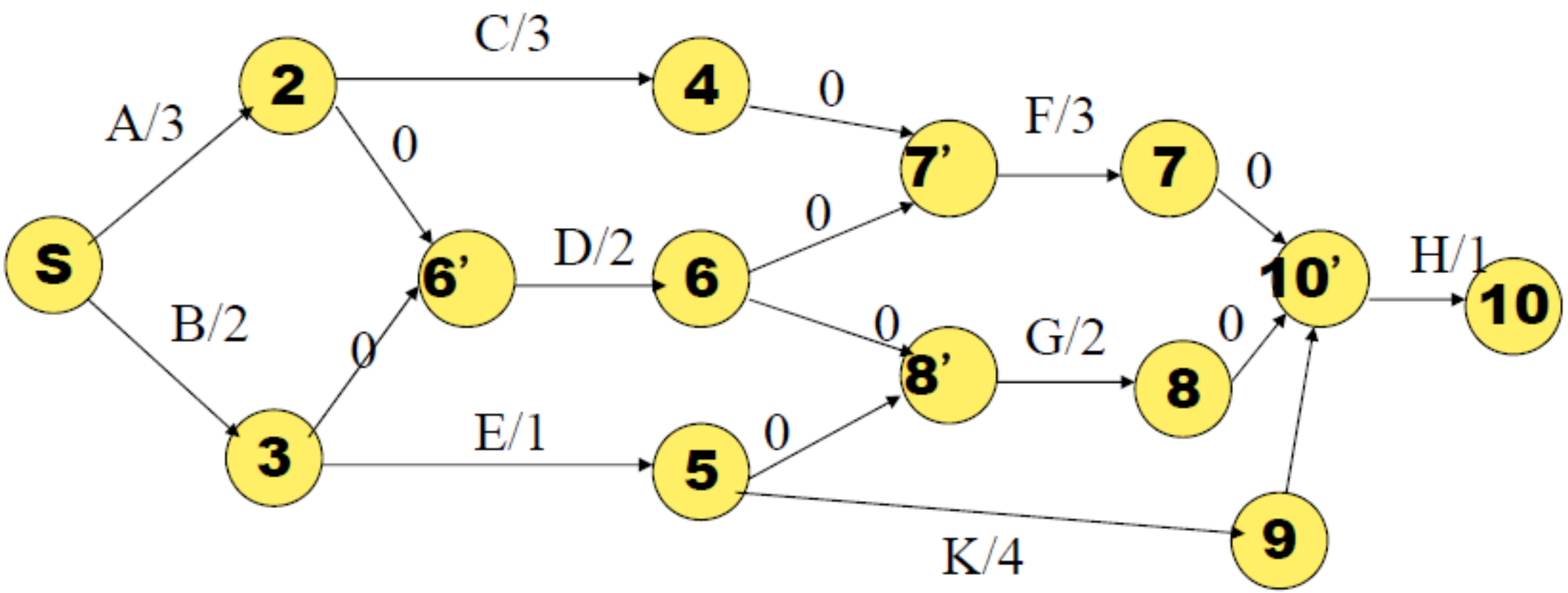
Compute Earliest Time



ee	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	Stack
Initial	0	0	0	0	0	0	0	0	0	[0]
output 0	0	6	4	5	0	0	0	0	0	[3,2,1]
output 3	0	6	4	5	0	7	0	0	0	[5,2,1]
output 5	0	6	4	5	0	7	0	11	0	[2,1]
output 2	0	6	4	5	5	7	0	11	0	[1]
output 1	0	6	4	5	7	7	0	11	0	[4]
output 4	0	6	4	5	7	7	16	14	0	[7,6]
output 7	0	6	4	5	7	7	16	14	18	[6]
output 6	0	6	4	5	7	7	16	14	18	[8]
output 8										

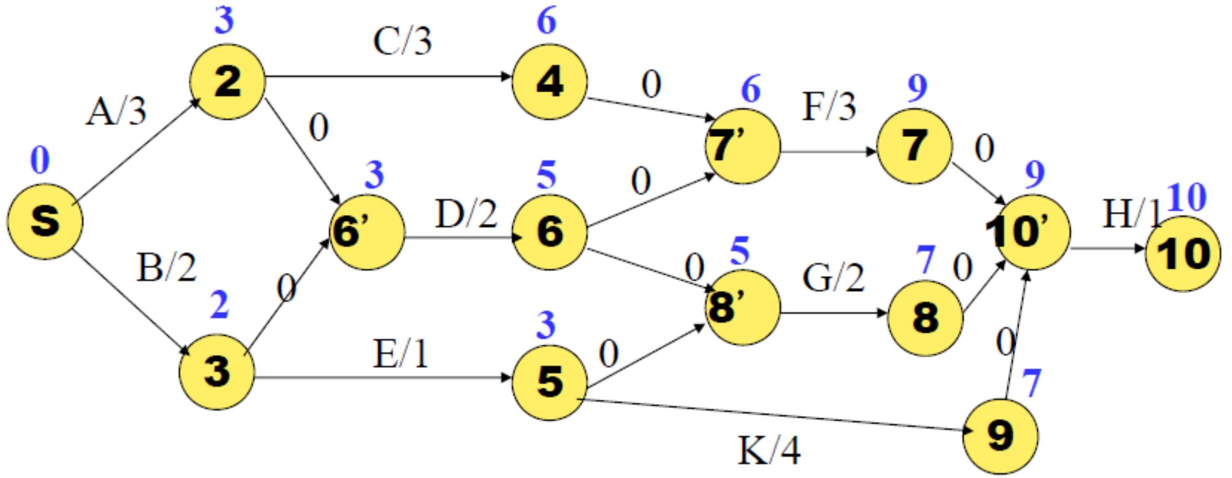
Critical Path Analysis

- AOE graph



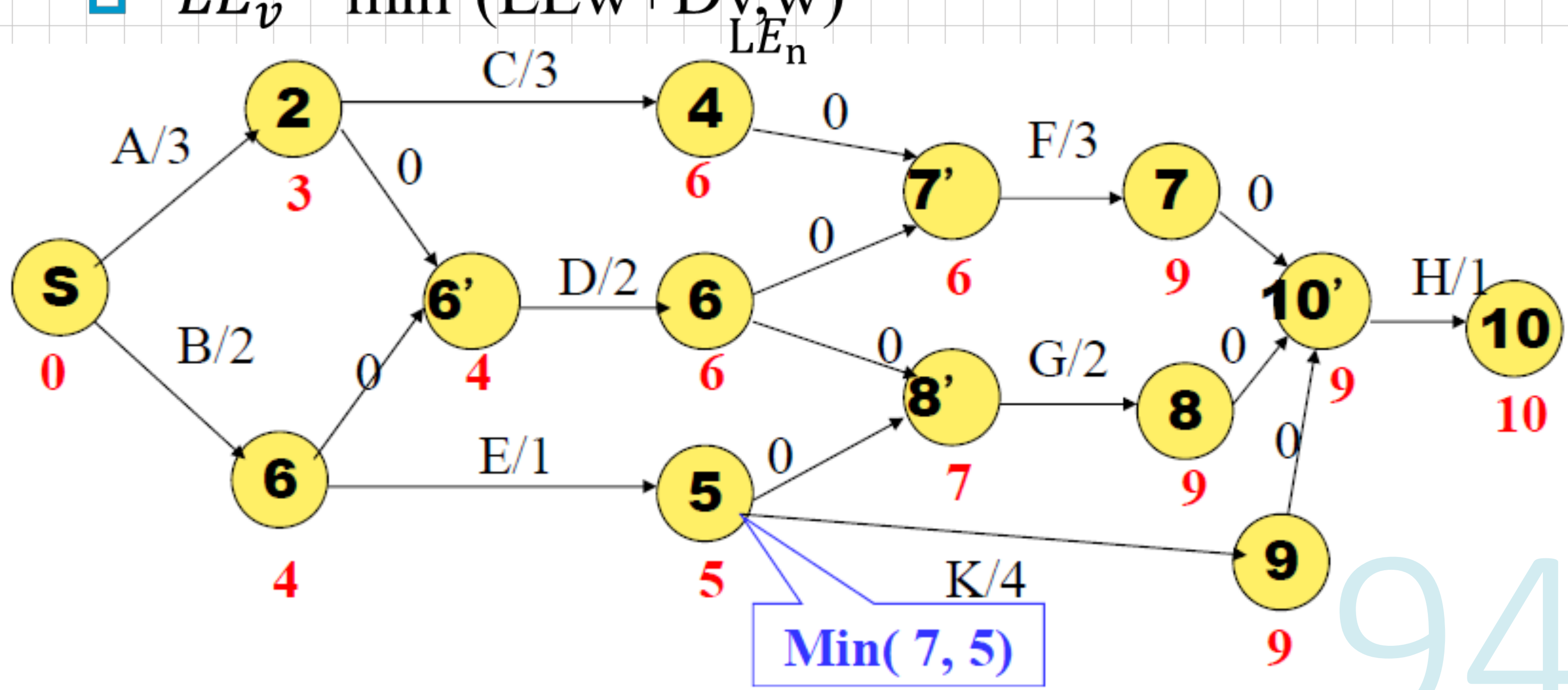
Critical Path Analysis

- Earliest completion times: longest path
  - computed by topological order
  - $EE_1 = 0$
  - EE_w=max (EEv+Dv,w)



Critical Path Analysis

- Latest completion times:
  - latest time without affecting final completion time
  - computed by **reverse** topological order
  - $LE_n = EE_n$
  - $LE_v = \min(LE_w + D_{v,w})$



Critical Path Analysis

- Slack time(v,w)=LEw-EEw
- Critical path = zero slack time