DATA STRUCTURES AND ALGORITHMS

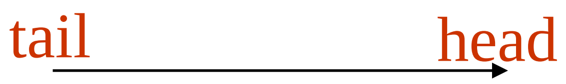
GRAPHS

Definition

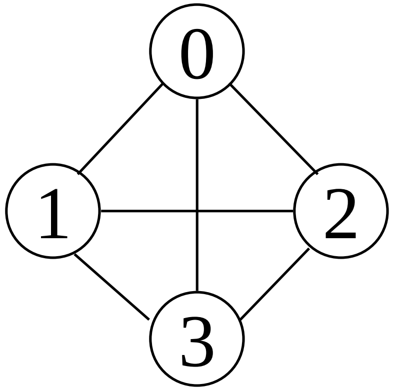
- A graph G consists of two sets
  - a finite, nonempty set of vertices V(G)
  - a finite, possible empty set of edges E(G)
  - G(V,E) represents a graph

- 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>

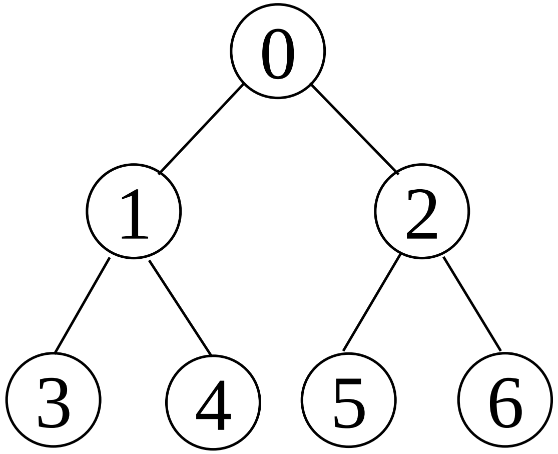


Examples for Graph



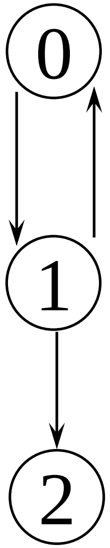
G_1

complete graph



G_2

incomplete graph



G_3

V(G1)={0,1,2,3} E(G1)={(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)}
V(G2)={0,1,2,3,4,5,6} E(G2)={(0,1),(0,2),(1,3),(1,4),(2,5),(2,6)}
V(G3)={0,1,2} E(G3)={<0,1>,<1,0>,<1,2>}

complete undirected graph: n(n-1)/2 edges
complete directed graph: n(n-1) edges

Complete Graph

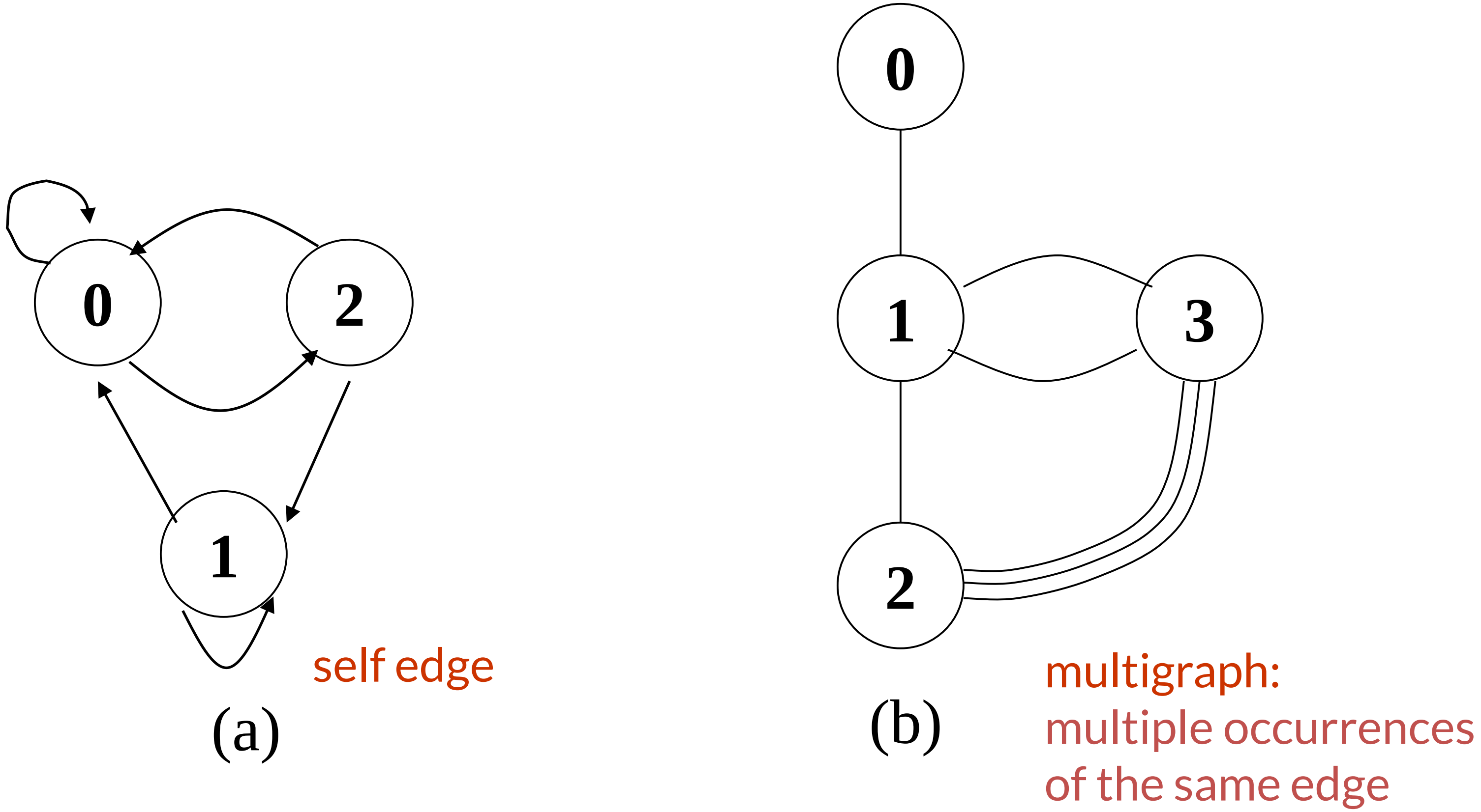
- A complete graph is a graph that has the maximum number of edges
  - for undirected graph with n vertices, the maximum number of edges is $n(n-1)/2$
  - for directed graph with n vertices, the maximum number of edges is $n(n-1)$
  - example: G1 is a complete graph

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

Example of a graph with feedback loops and a multigraph



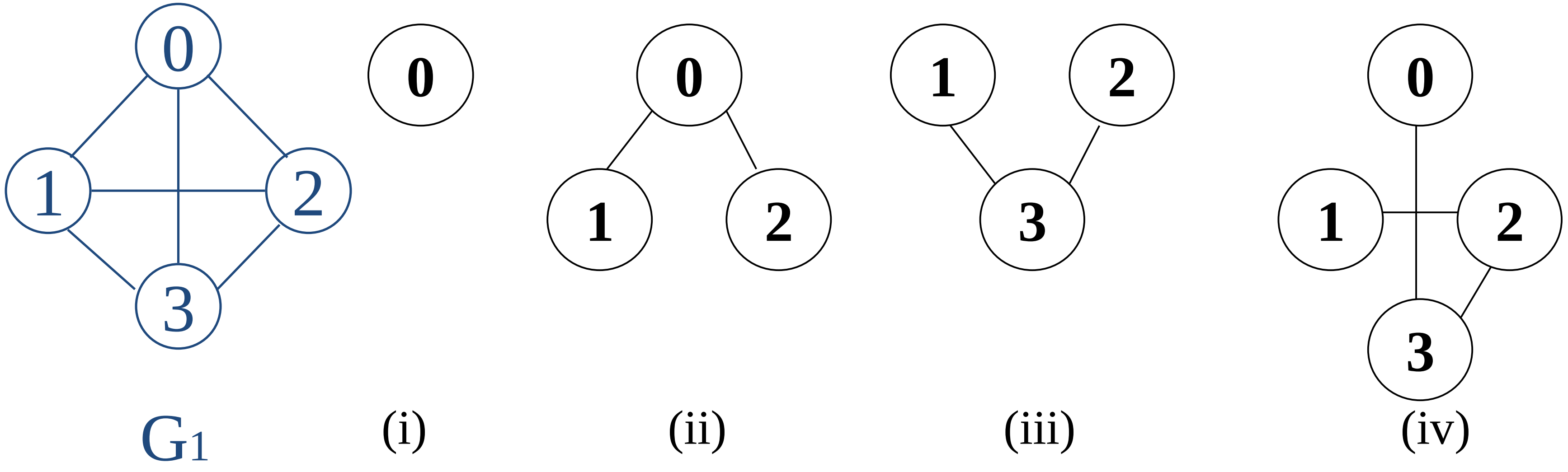
Subgraph and Path

- 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, vp, vi1, vi2, ..., vin, vq, such that (vp, vi1), (vi1, vi2), ..., (vin, vq) are edges in an undirected graph

- The length of a path is the number of edges on it

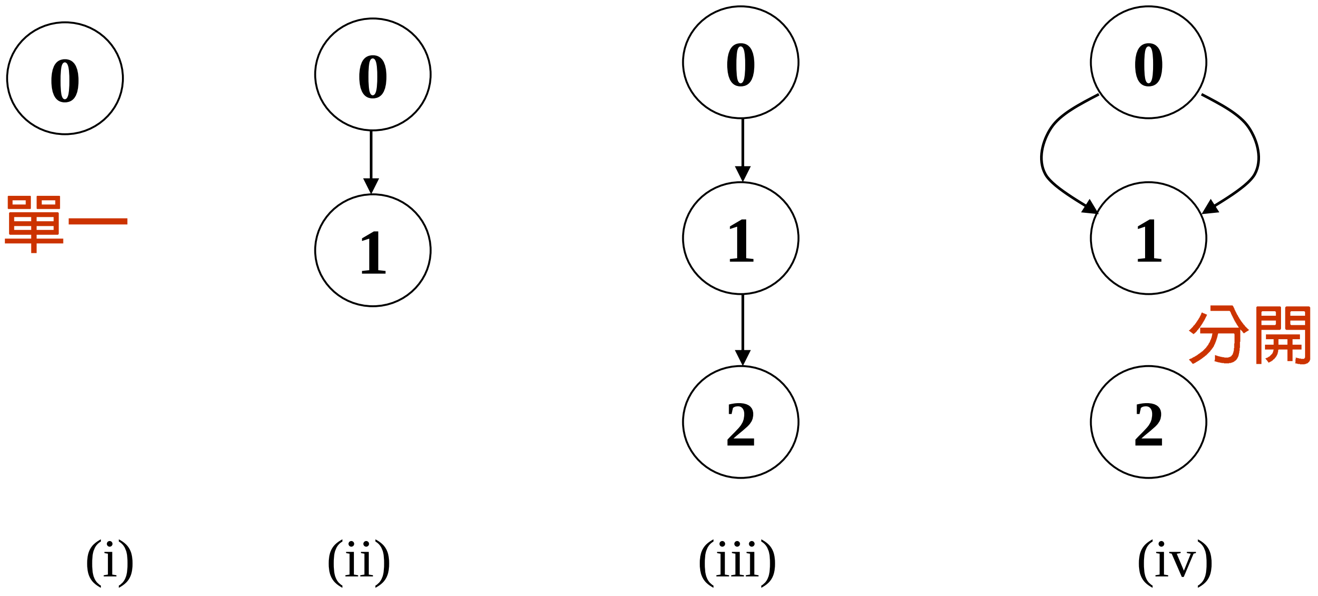
subgraphs of G1 and G3



(a) Some of the subgraph of G_1

subgraphs of G1 and G3 (contd.)

G3

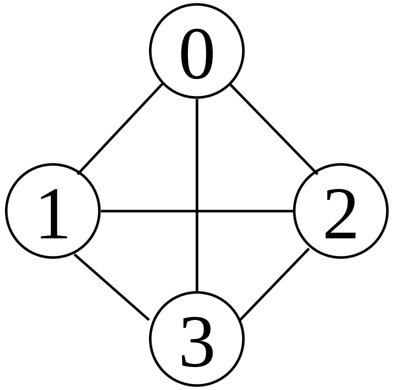


(b) Some of the subgraph of G_3

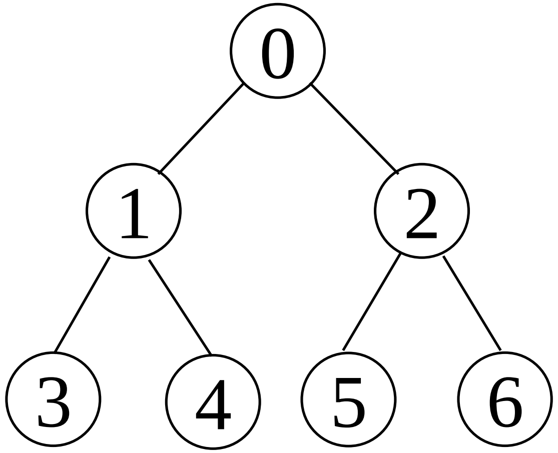
Simple Path and Style

- A simple path is a path in which all vertices, except possibly the first and the last, are distinct
- 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
- An undirected graph is connected if, for every pair of distinct vertices vi, vj, there is a path from vi to vj

Connected



G_1



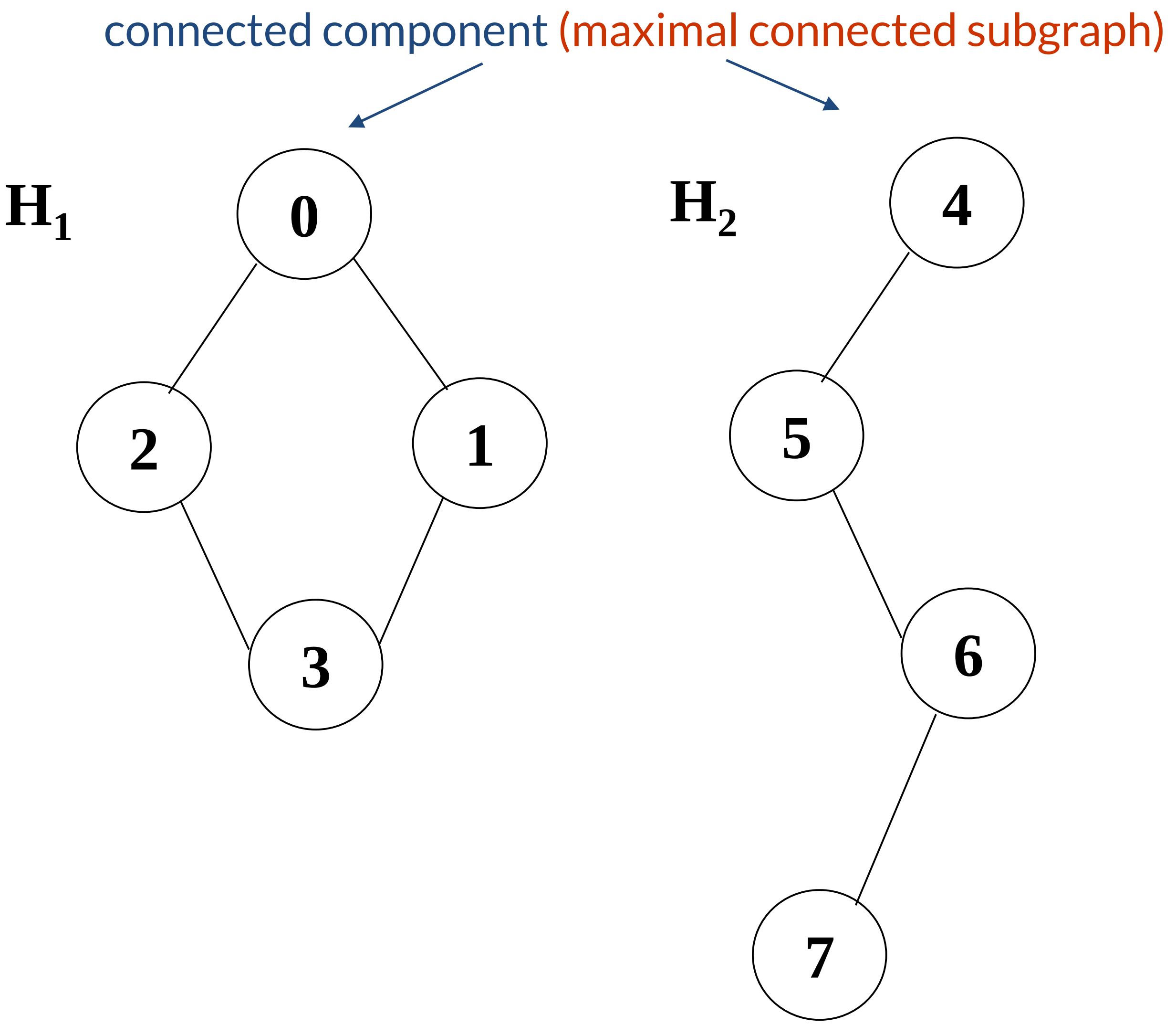
G_2

tree (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.

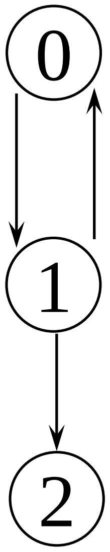
A graph with two connected components



G_4 (not connected)

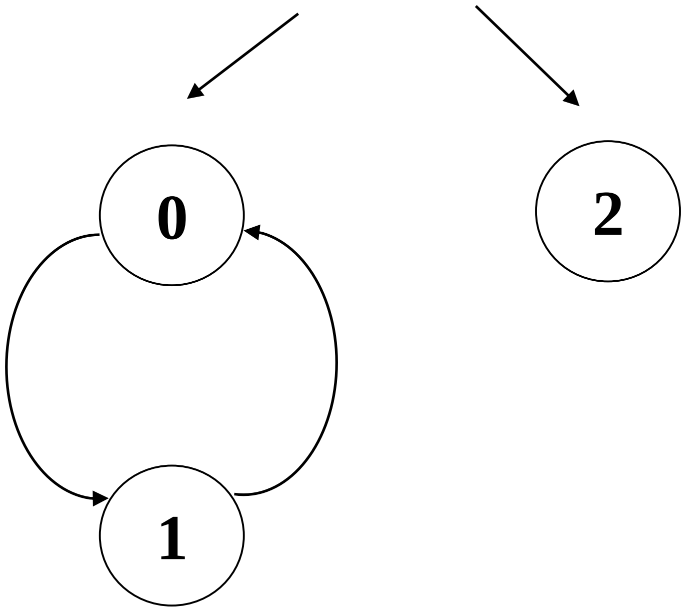
Strongly connected components of G3

not strongly connected



G_3

strongly connected component
(maximal strongly connected subgraph)



Degree

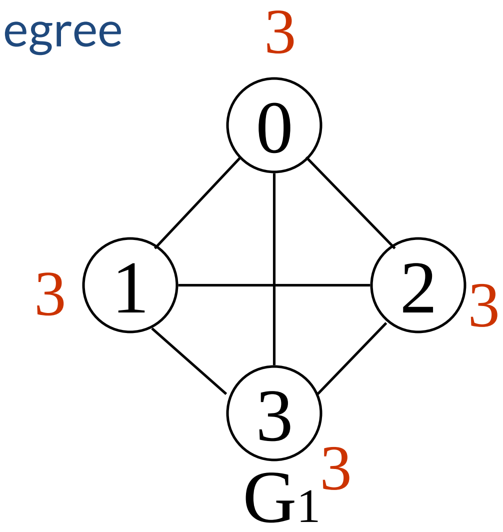
- The degree of a vertex is the number of edges incident to that vertex

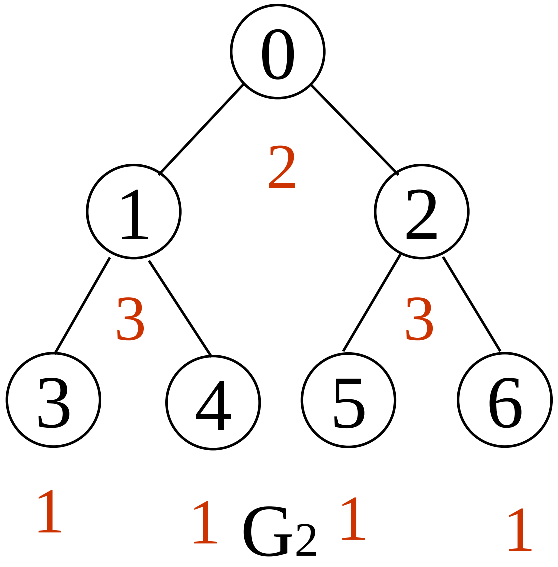
- For directed graph,
  - the in-degree of a vertex v is the number of edges that have v as the head
  - the out-degree of a vertex v is the number of edges that have v as the tail
  - if d_i is the degree of a vertex i in a graph G with n vertices and e edges, the number of edges is

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

undirected graph

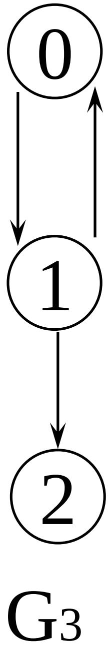
degree





directed graph

in-degree
out-degree



in:1, out: 1

in: 1, out: 2

in: 1, out: 0

ADT for Graph

structure Graph is

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

functions: for all *graph* $\in$ *Graph*, *v*, v_1 and v_2 $\in$ *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*

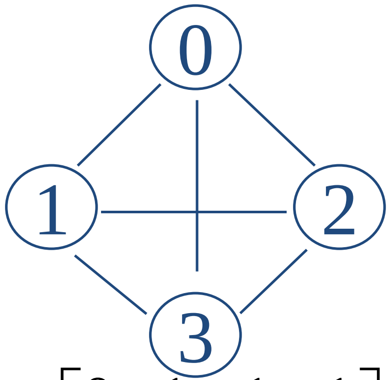
Graph Representations

- Adjacency Matrix
- Adjacency Lists
- 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

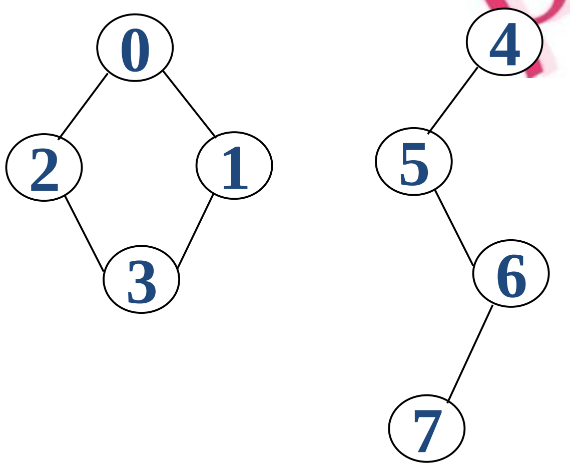


$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

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

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

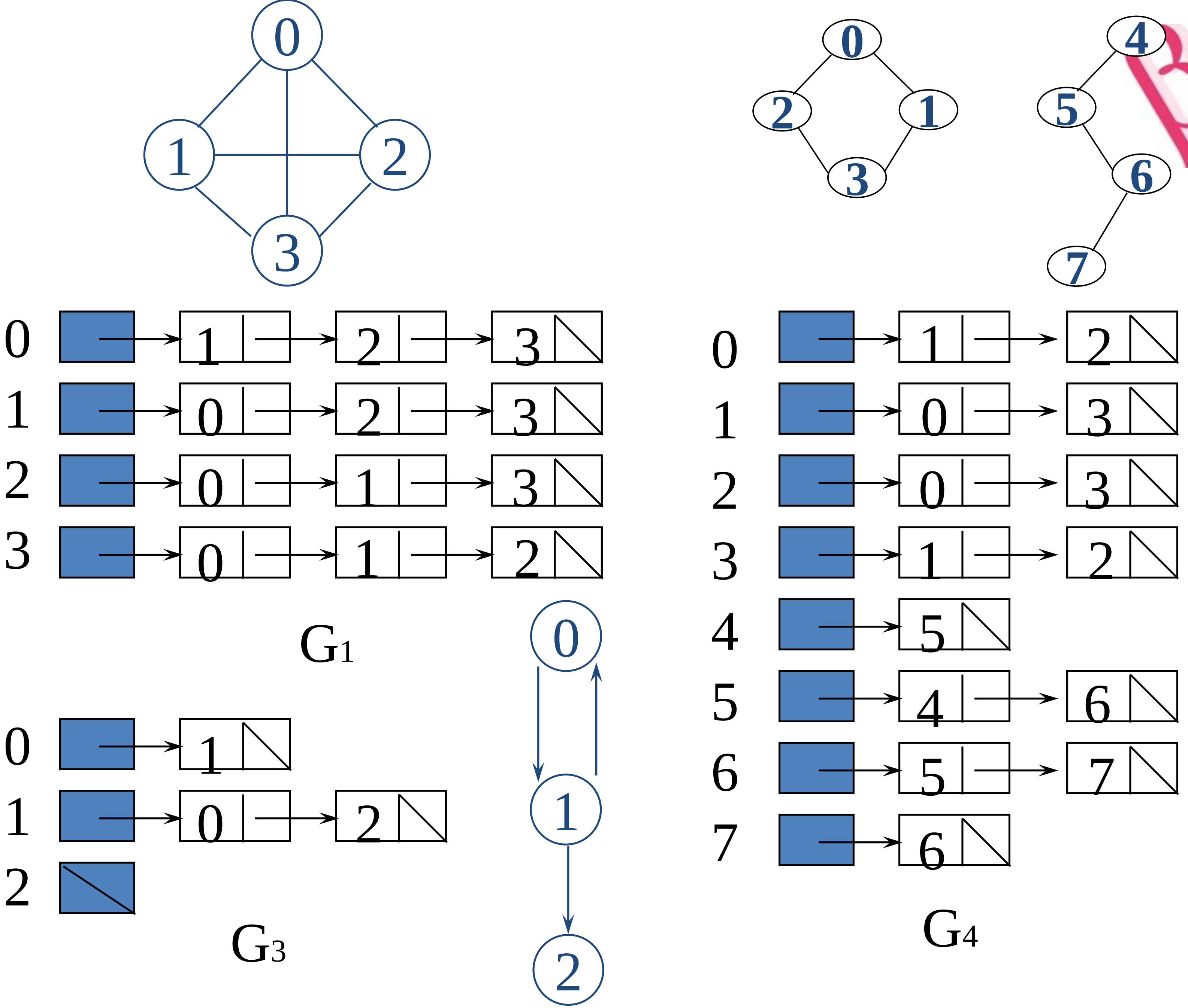
$$\sum_{j=0}^{n-1} adj\_mat[i][j]$$

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

Data Structures for 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 */
```



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

Interesting Operations

- degree of a vertex in an undirected graph

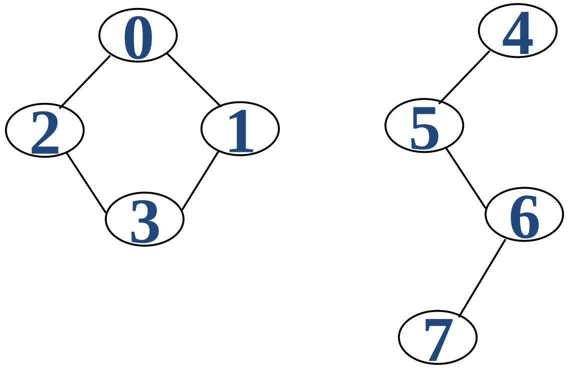
  # of nodes in adjacency list
- # of edges in a graph

  determined in O(n+e)
- out-degree of a vertex in a directed graph

  # of nodes in its adjacency list
- in-degree of a vertex in a directed graph

  traverse the whole data structure

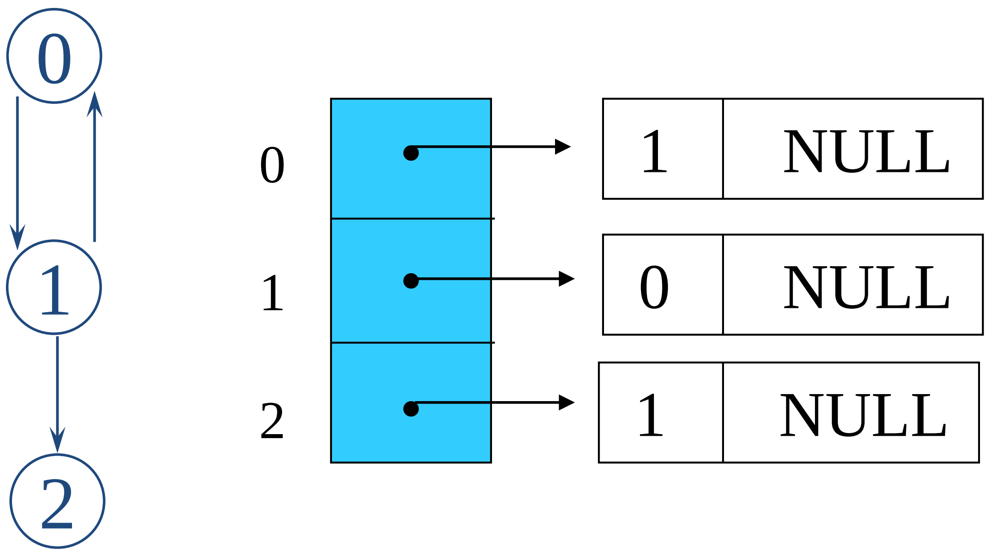
Compact Representation



node[0] … node[n-1]: starting point for vertices
node[n]: n+2e+1
node[n+1] … node[n+2e]: head node of edge

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

Inverse adjacency list for G3

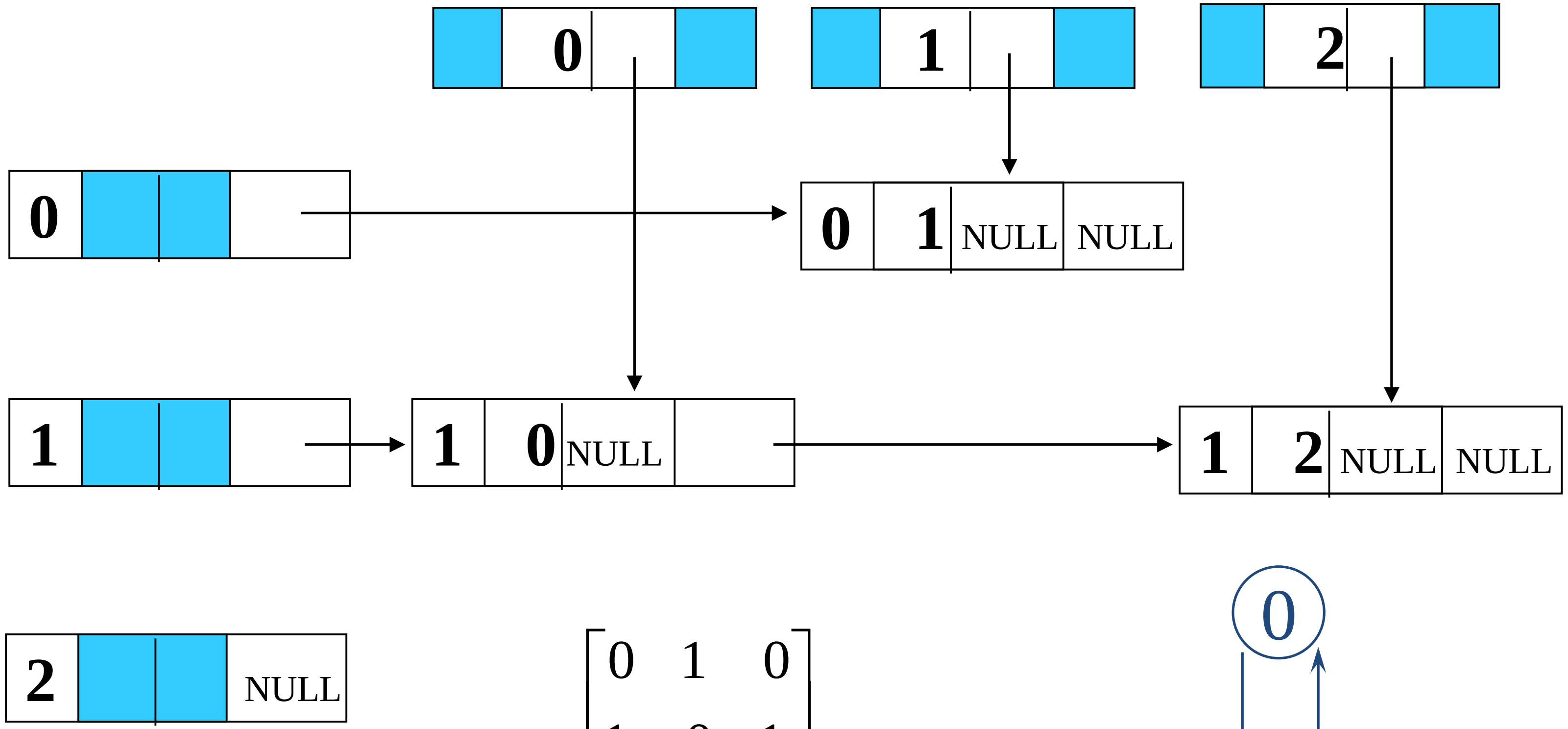


Determine in-degree of a vertex in a fast way.

Alternate node structure for adjacency lists

tail	head	column link for head	row link for tail

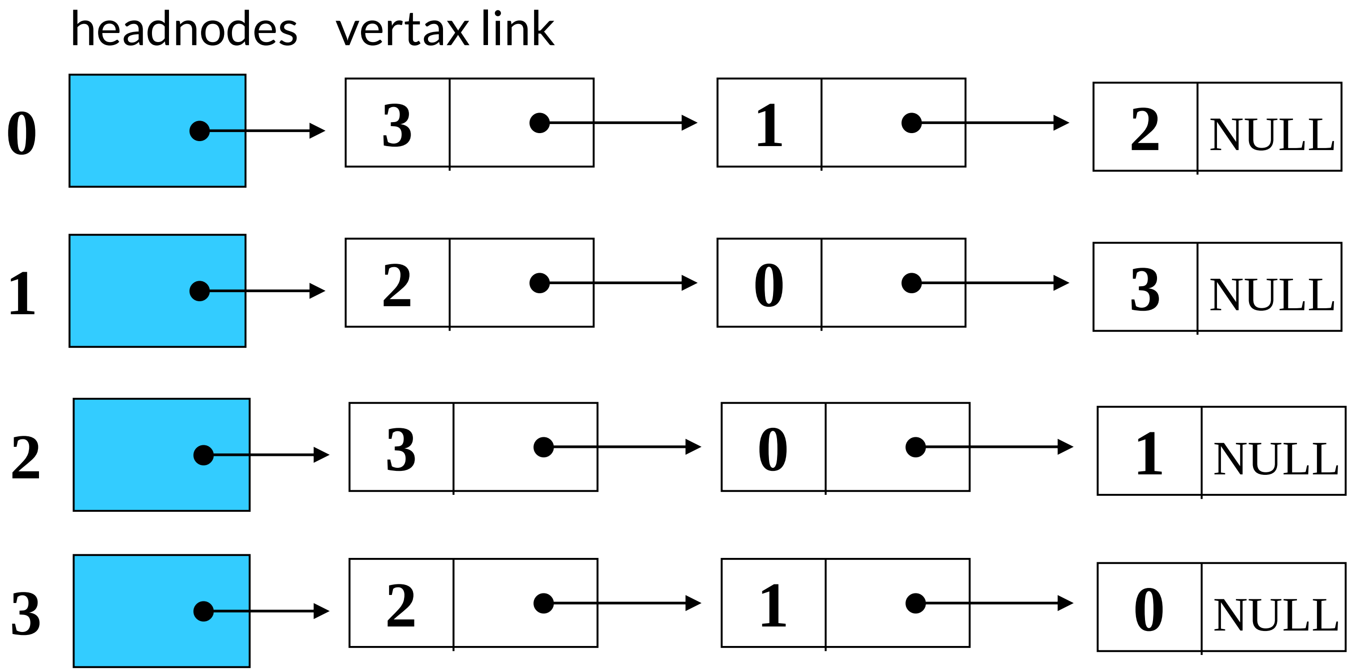
Orthogonal representation for graph G3

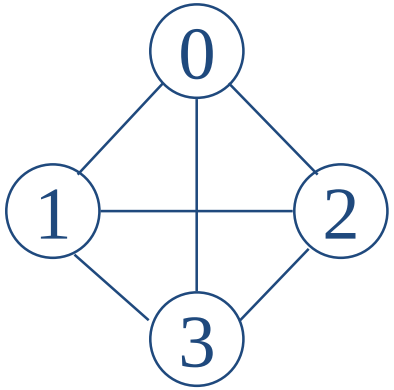


$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

Alternate order adjacency list for G1

Order is of no significance.





Adjacency Multilists

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

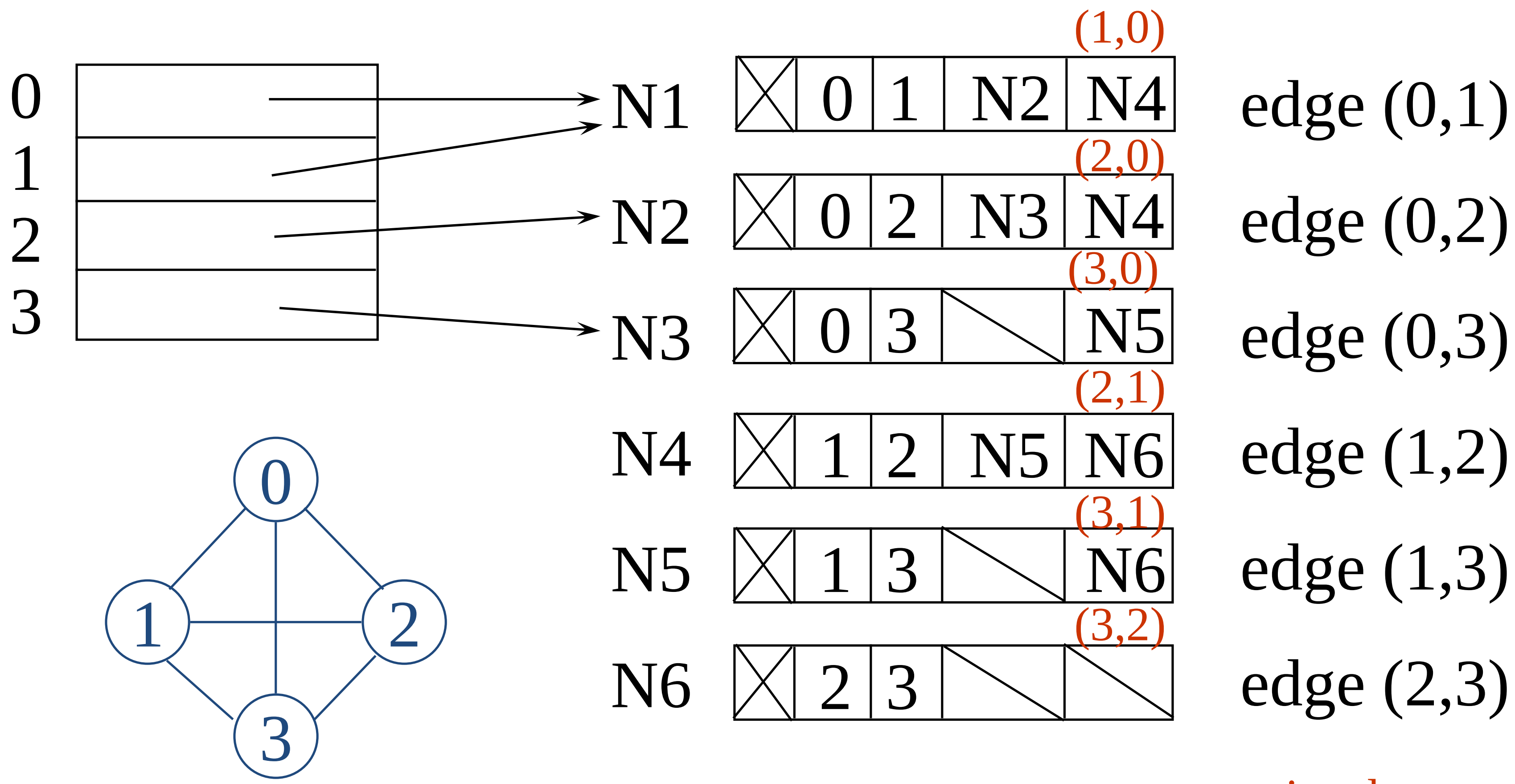
  lists in which nodes may be shared among several lists.

  (an edge is shared by two different paths)

marked	vertex1	vertex2	path1	path2

Example for Adjacency Multlists

Lists: vertex 0: M1->M2->M3, vertex 1: M1->M4->M5
vertex 2: M2->M4->M6, vertex 3: M3->M5->M6



six edges

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.

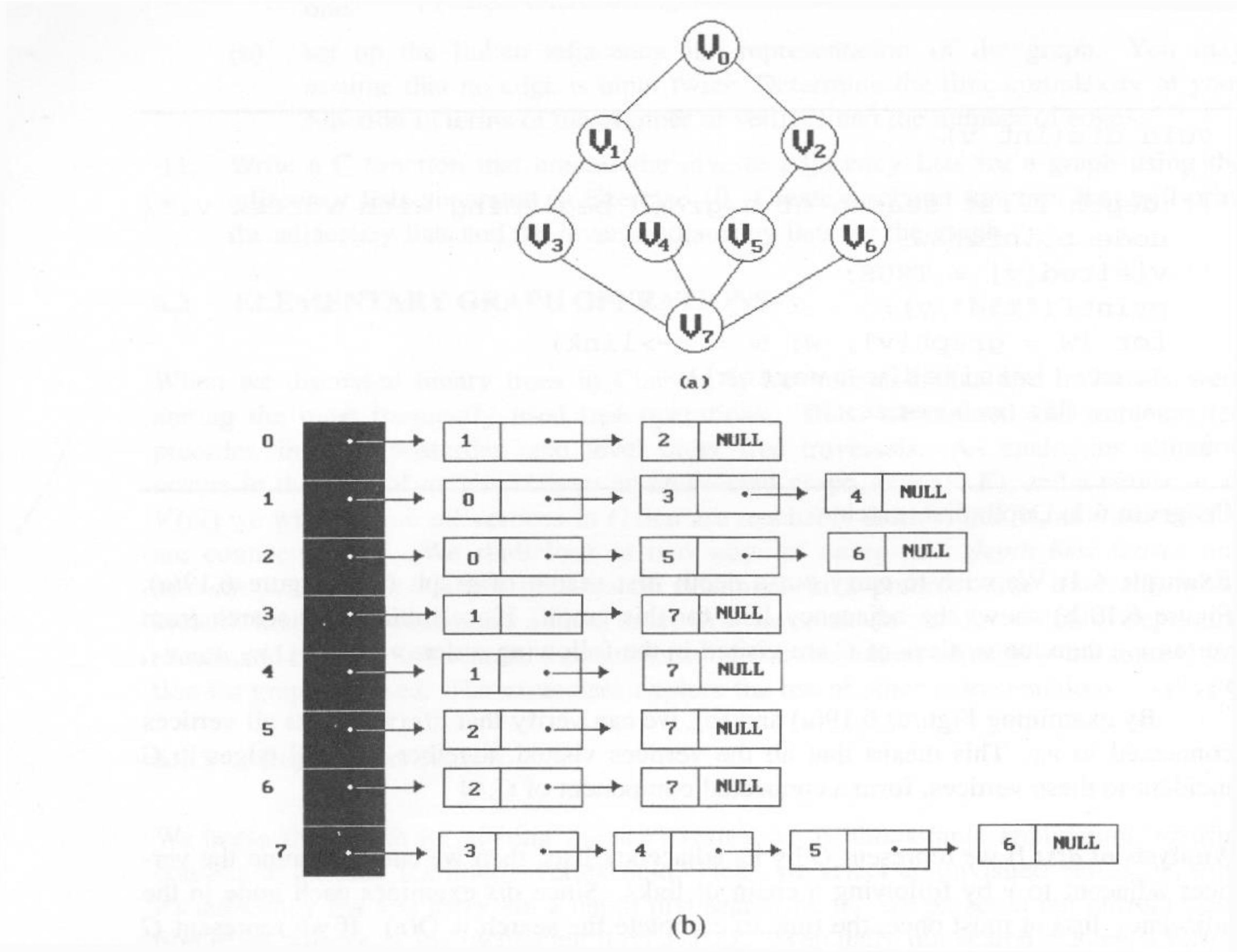
Depth First Search (DFS)
preorder tree traversal
Breadth First Search (BFS)
level order tree traversal

- Connected Components

- Spanning Trees

Graph G and its adjacency lists

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

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

```
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);
}
```

Data structure
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);
```

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

```
   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;
       }
   }
}
```

LearnOA

Connected Components

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

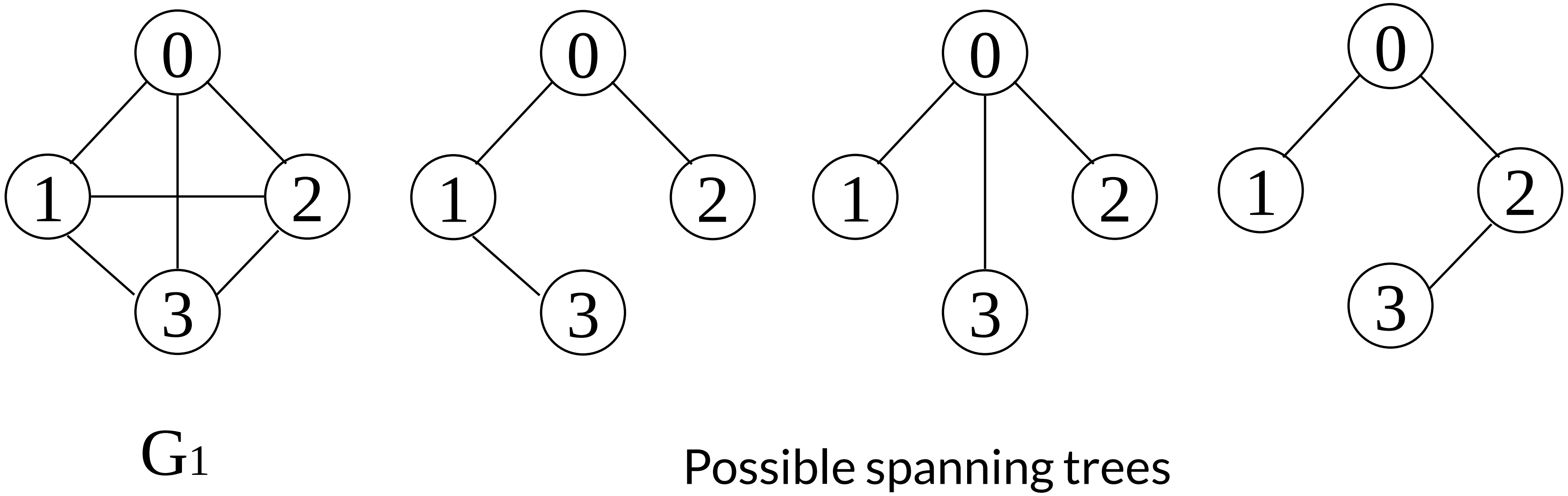
adjacency list: $O(n+e)$
adjacency matrix: $O(n^2)$

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

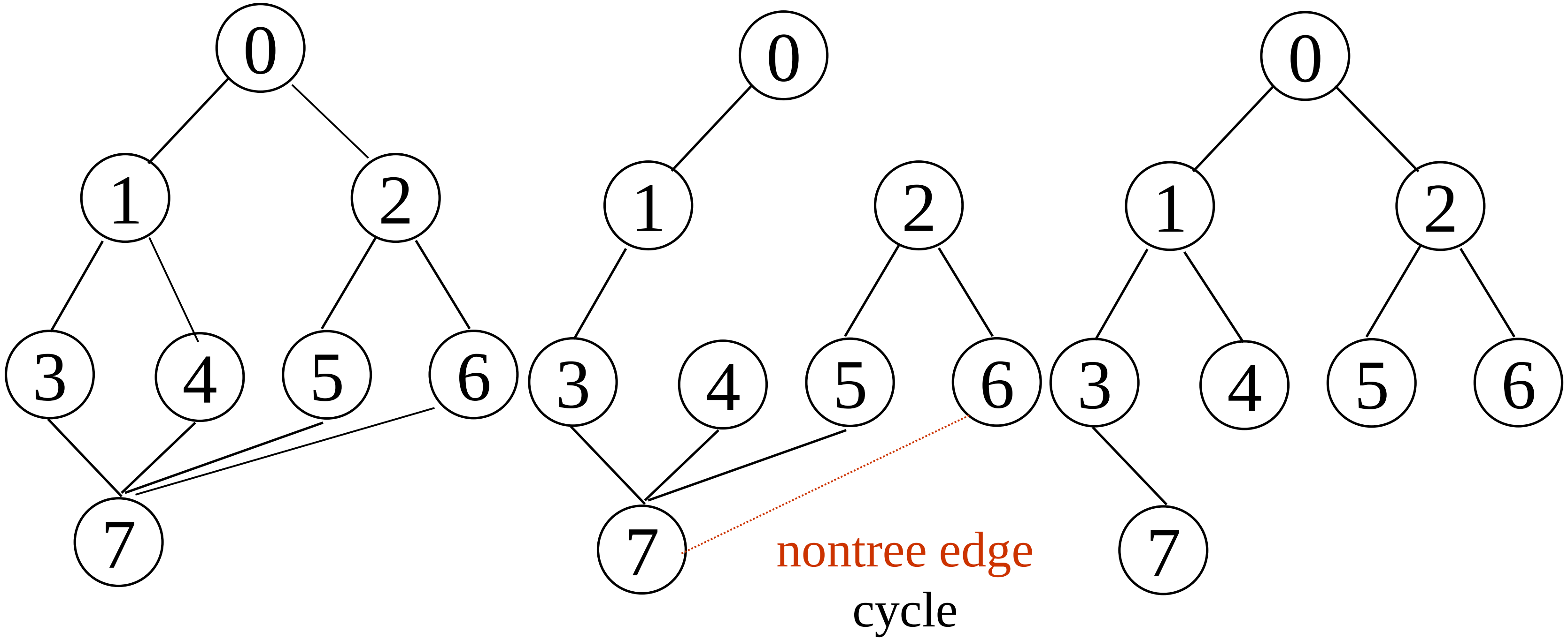
Examples of Spanning Tree



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 nontree edge into any spanning tree, this will create a cycle

DFS VS BFS Spanning Tree



DFS Spanning

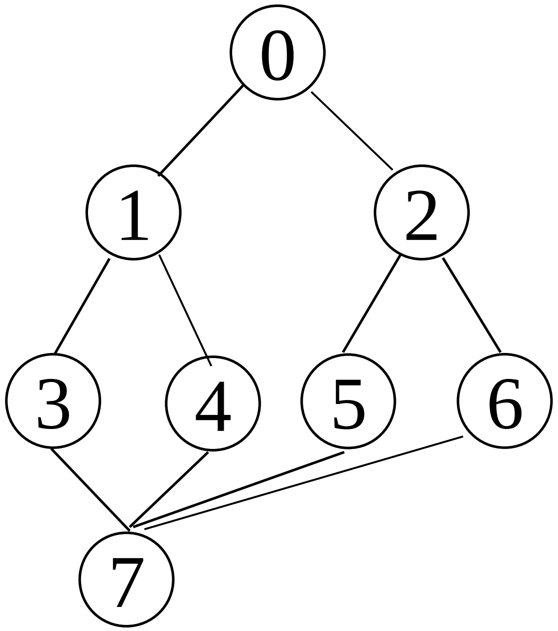
BFS Spanning

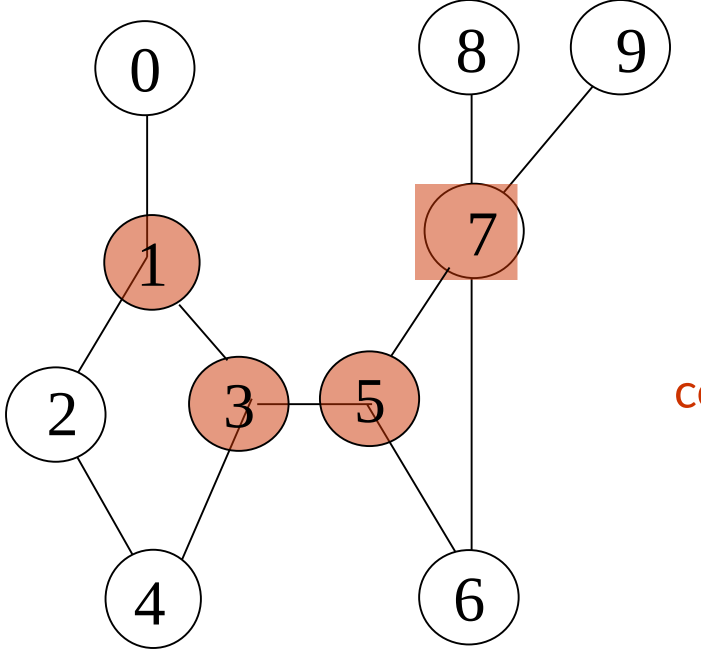
A spanning tree is a minimal subgraph, G', of G such that V(G')=V(G) and G' is connected.

Any connected graph with n vertices must have at least n-1 edges.

A biconnected graph is a connected graph that has no articulation points.

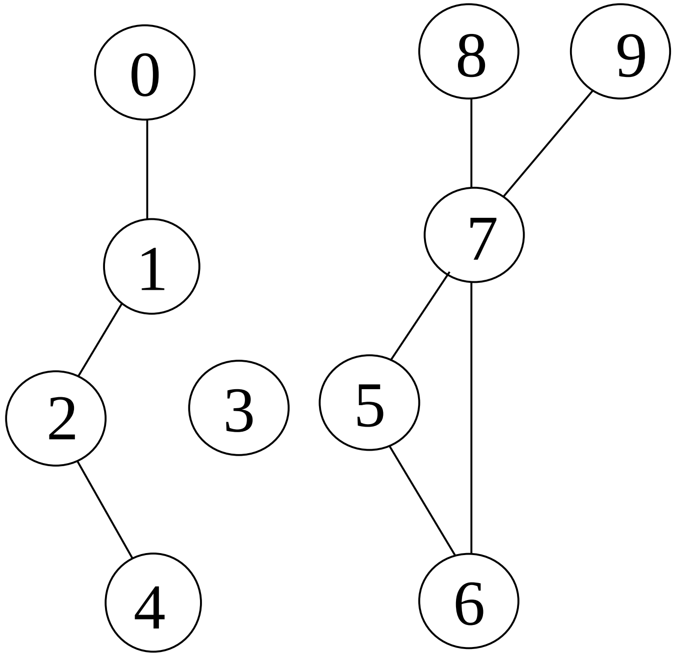
biconnected graph



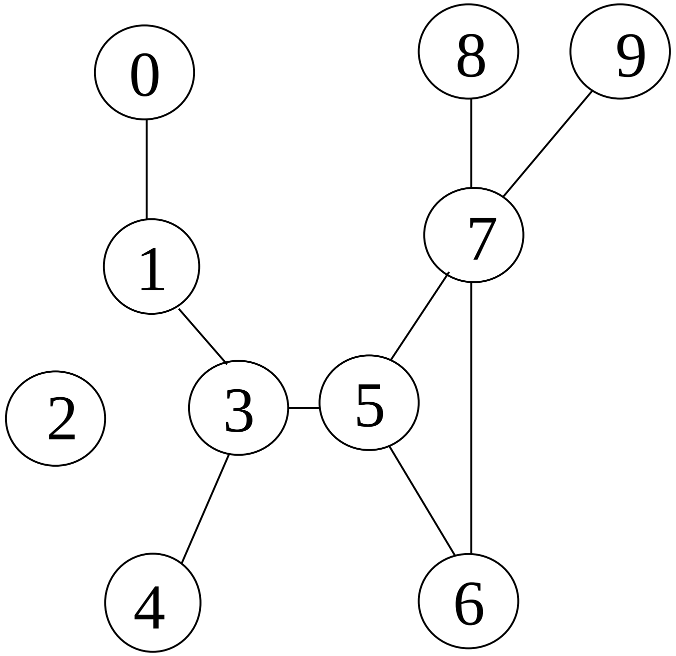


connected graph

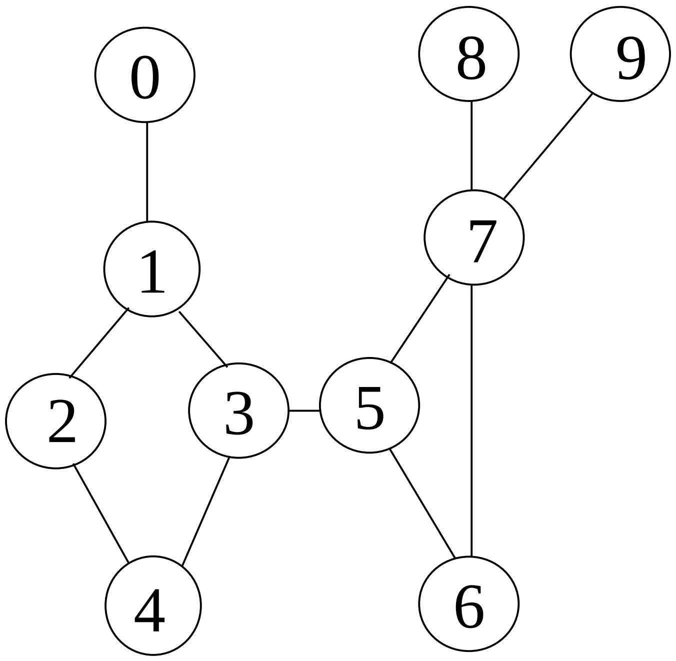
two connected components

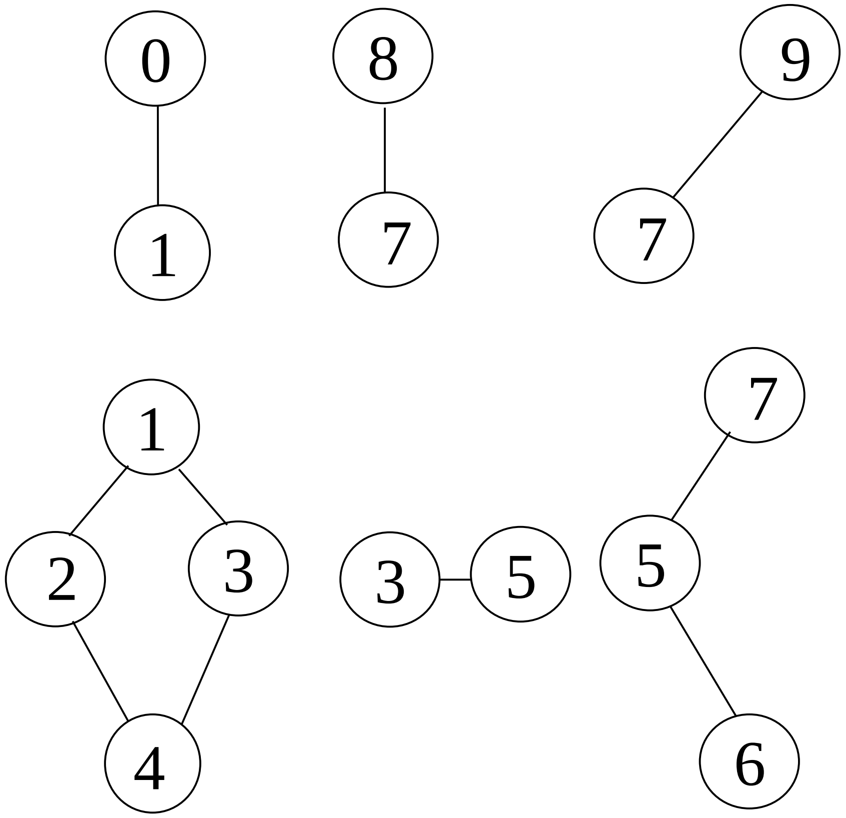


one connected graph



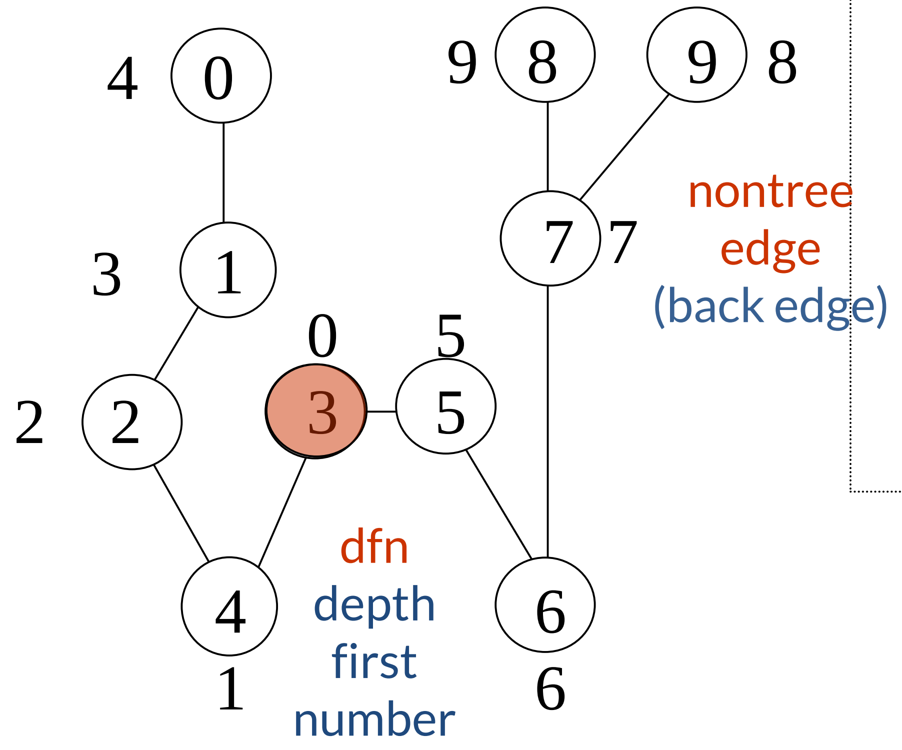
Biconnected component: a maximal connected subgraph H
(no subgraph that is both biconnected and properly contains H)



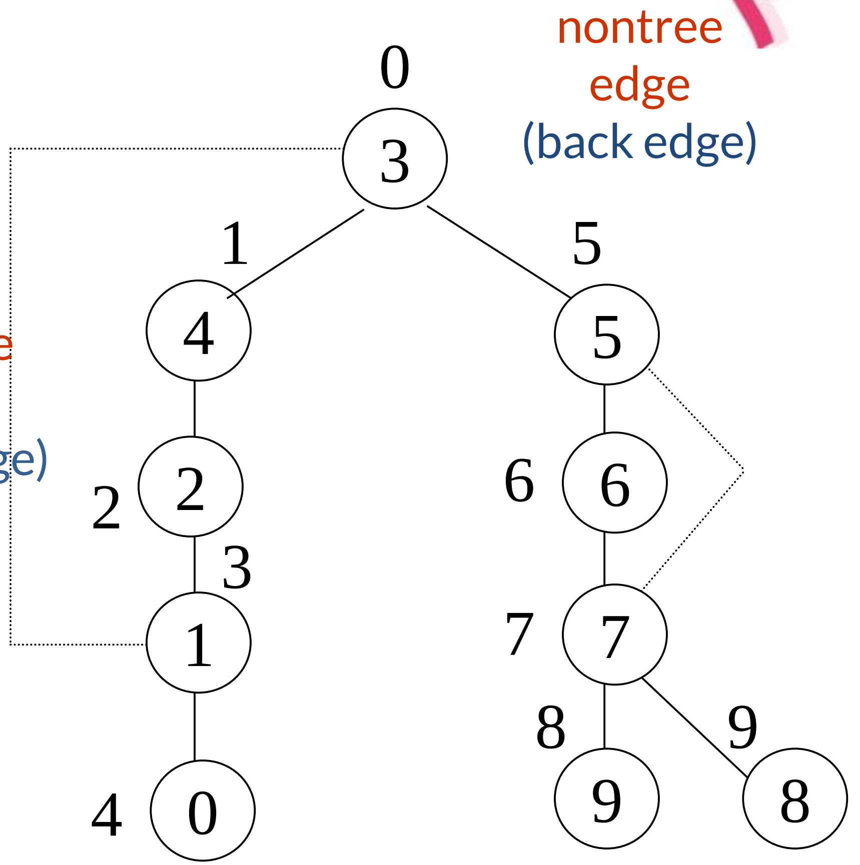


biconnected components

Find biconnected component of a connected undirected graph by depth first spanning tree



(a) depth first spanning tree



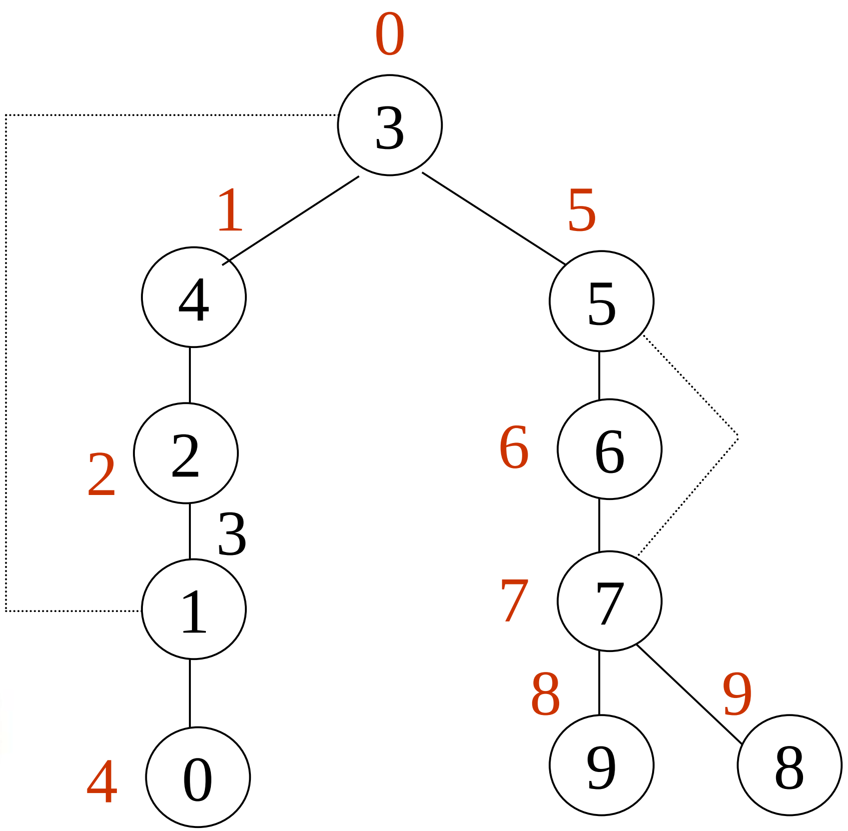
Why is cross edge impossible?

(b)

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

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

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



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

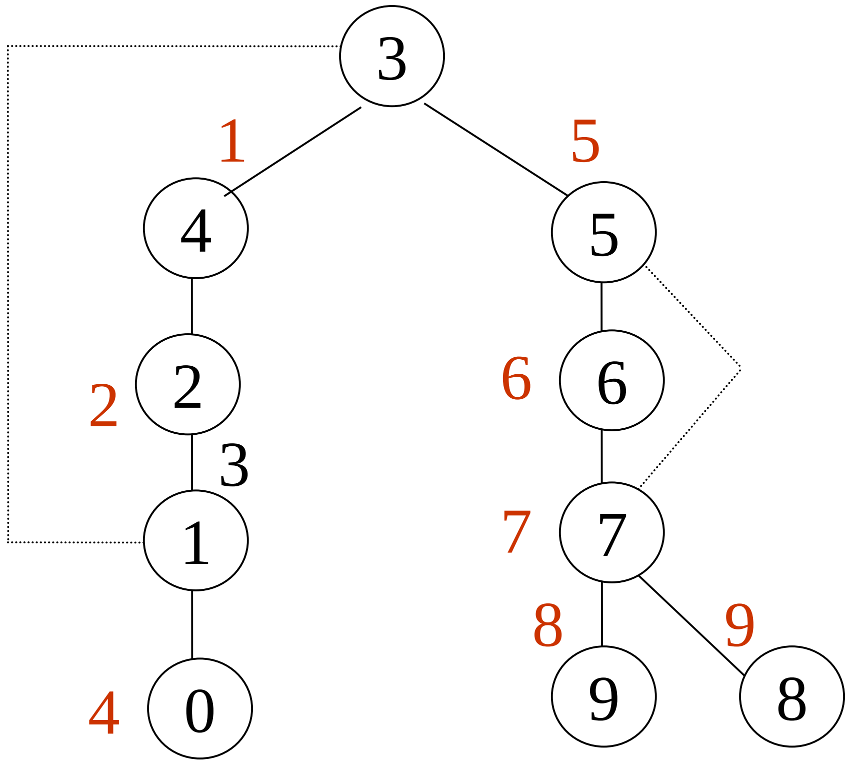
- Any other vertex u is an articulation point iff it has at least one child w such that we cannot reach an ancestor of u using a path that consists of

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

low(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) $\geq$ dfn(u)

vertex	dfn	low	child	low_child	low:dfn
0	4	4 (4,n,n)	null	null	null:4
1	3	0 (3,4,0)	0	4	4 ≥ 3 •
2	2	0 (2,0,n)	1	0	0 < 2
3	0	0 (0,0,n)	4,5	0,5	0,5 ≥ 0 •
4	1	0 (1,0,n)	2	0	0 < 1
5	5	5 (5,5,n)	6	5	5 ≥ 5 •
6	6	5 (6,5,n)	7	5	5 < 6
7	7	5 (7,8,5)	8,9	9,8	9,8 ≥ 7 •
8	9	9 (9,n,n)	null	null	null, 9
9	8	8 (8,n,n)	null	null	null, 8



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

Initializaiton of dfn and low

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

Determining dfn and low

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++;
```

low[u]=min{dfn(u), …}

```
    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]);
       }
```

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

```
     else if (w != v)
       low[u] =MIN2(low[u], dfn[w] );
  }
}
```

O X

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

Biconnected components of a graph

```
void bicon(int u, int v)
{
/* compute dfn and low, and output the edges of G by their
   biconnected  components , v is the parent ( if any) of the u
  (if any)  in the resulting spanning tree. It is assumed that all
 entries  of dfn[ ] have been initialized to -1, num has been
 initialized to 0,  and the stack has been set to empty */
   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 */
```

low[u]=min{dfn(u), …}

(1) dfn[w]=-1 第一次

(2) dfn[w]!=-1非第一次，藉back edge

```
if(dfn[w] < 0) {/* w has not been visited */
        bicon(w, 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{low(w)|w is a child of u}, …}

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

Minimum Cost Spanning Tree

- 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 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 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 nondecreasing 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

Examples for Kruskal's Algorithm

0 —10— 5

2 —12— 3

1 —14— 6

1 —16— 2

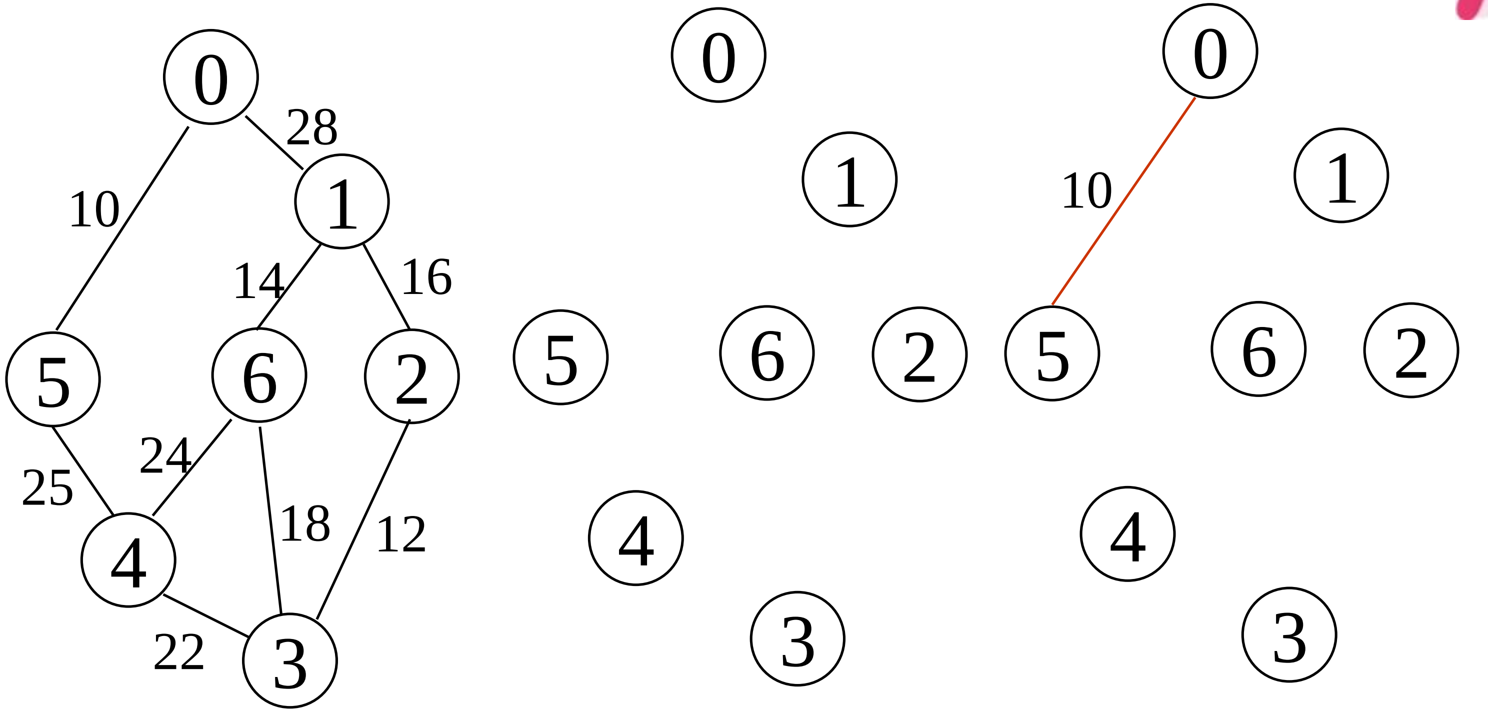
3 —18— 6

3 —22— 4

4 —24— 6

4 —25— 5

0 —28— 1



6/9

0 —10— 5

2 —12— 3

1 —14— 6

1 —16— 2

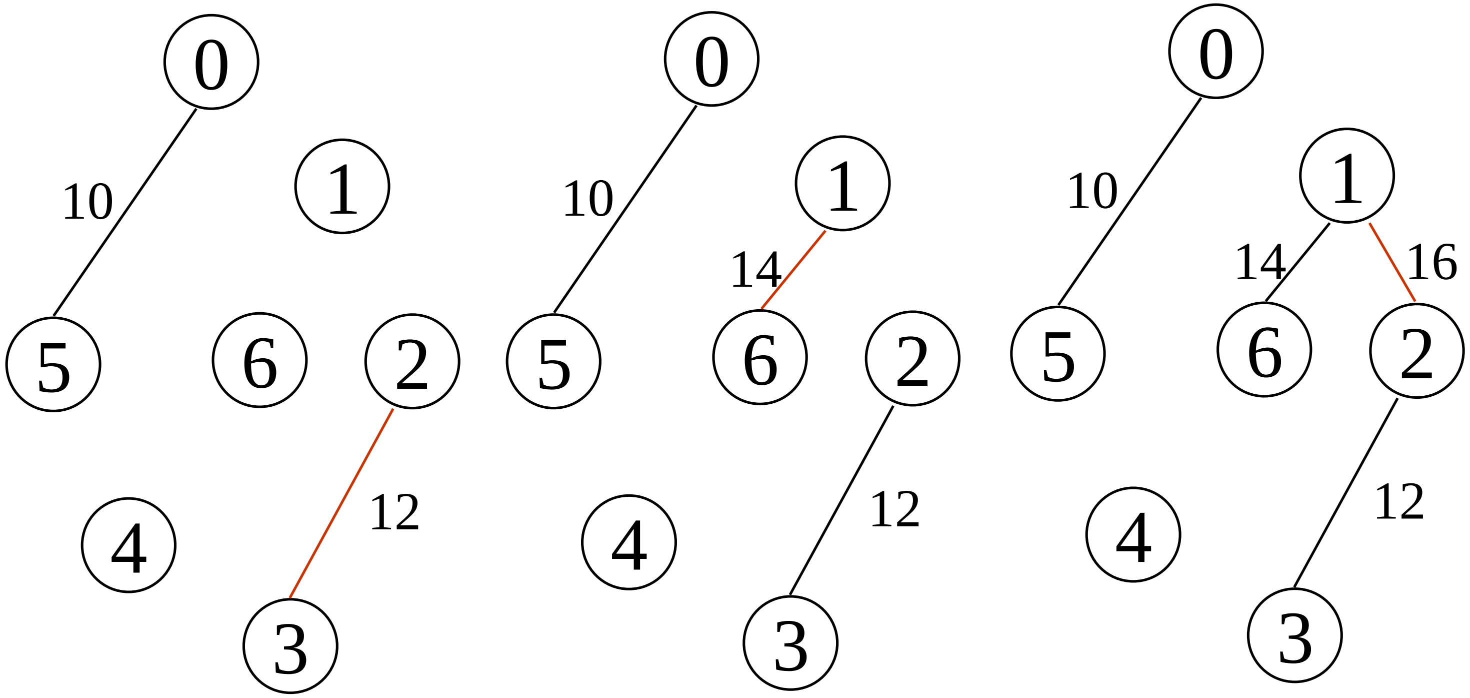
3 —18— 6

3 —22— 4

4 —24— 6

4 —25— 5

0 —28— 1



+ 3 — 6

cycle

0 —10— 5

2 —12— 3

1 —14— 6

1 —16— 2

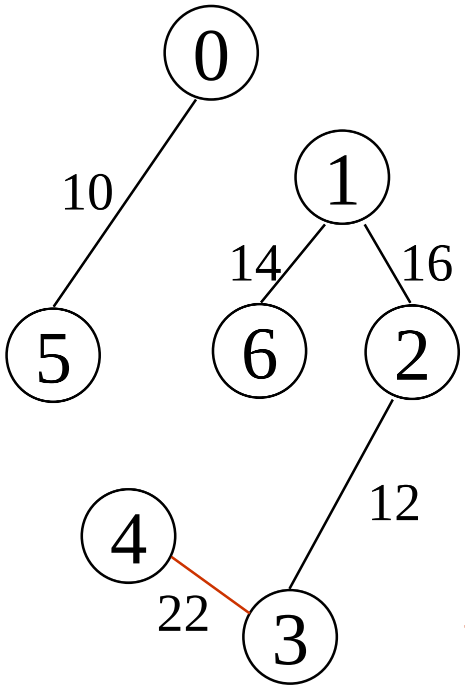
3 —18— 6

3 —22— 4

4 —24— 6

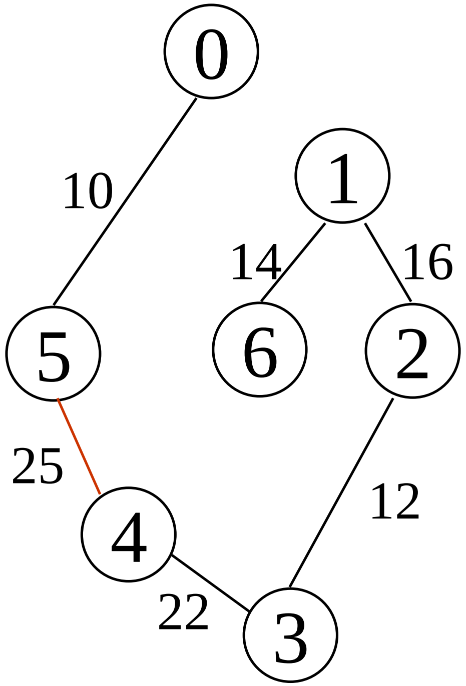
4 —25— 5

0 —28— 1



+ 4 —— 6

cycle



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 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}

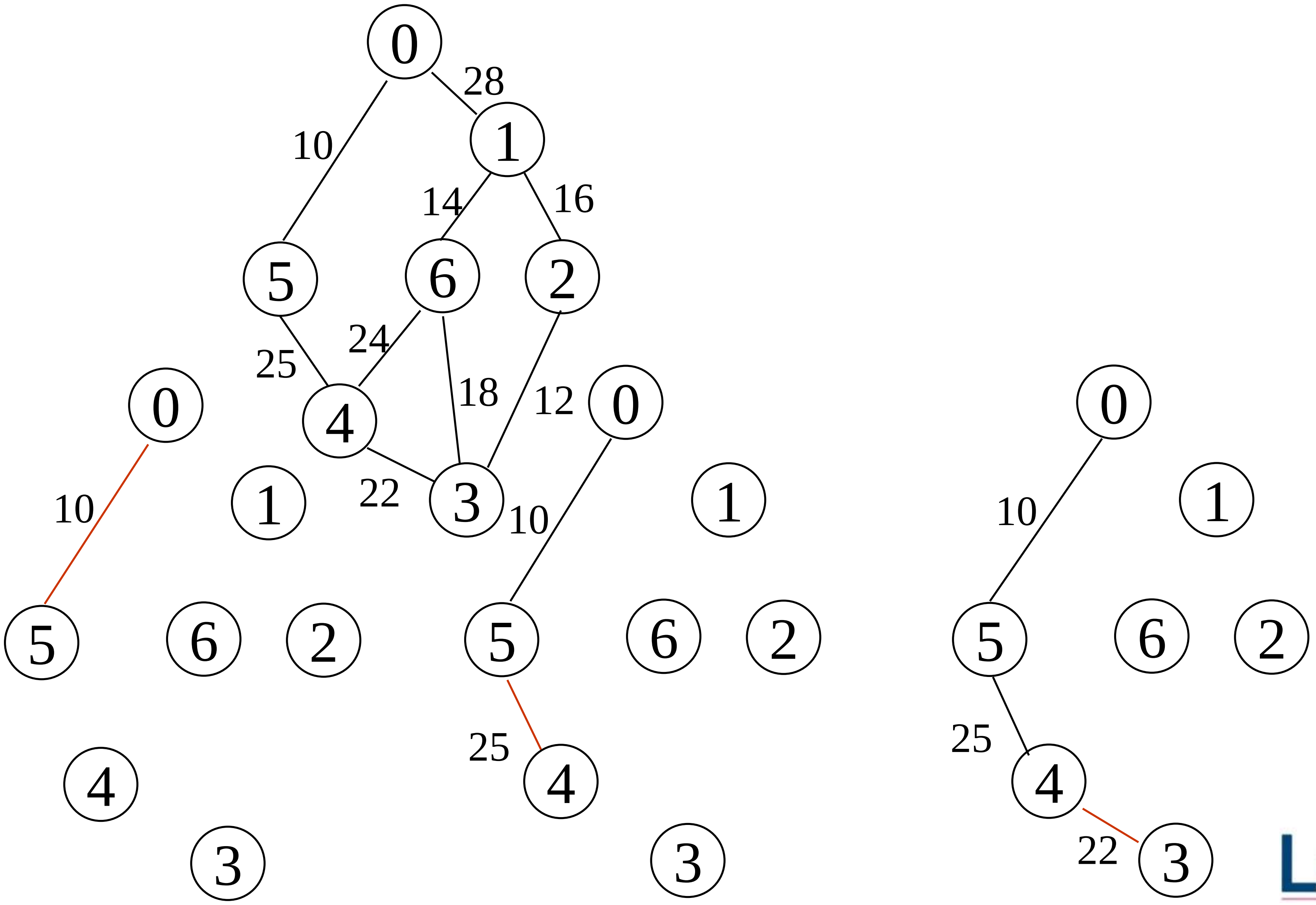
O(e log e)

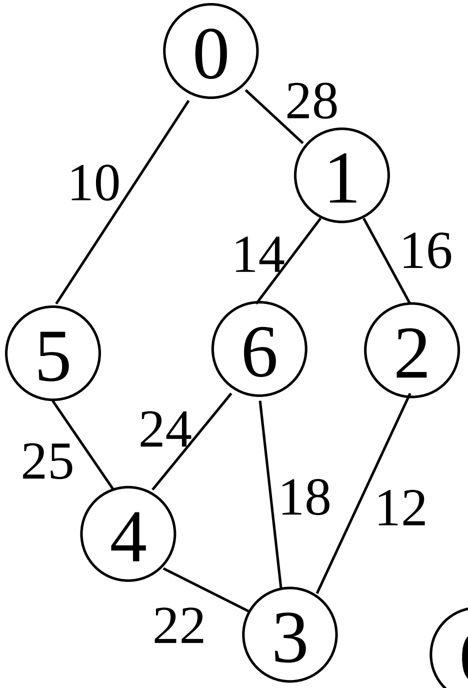
Prim's Algorithm

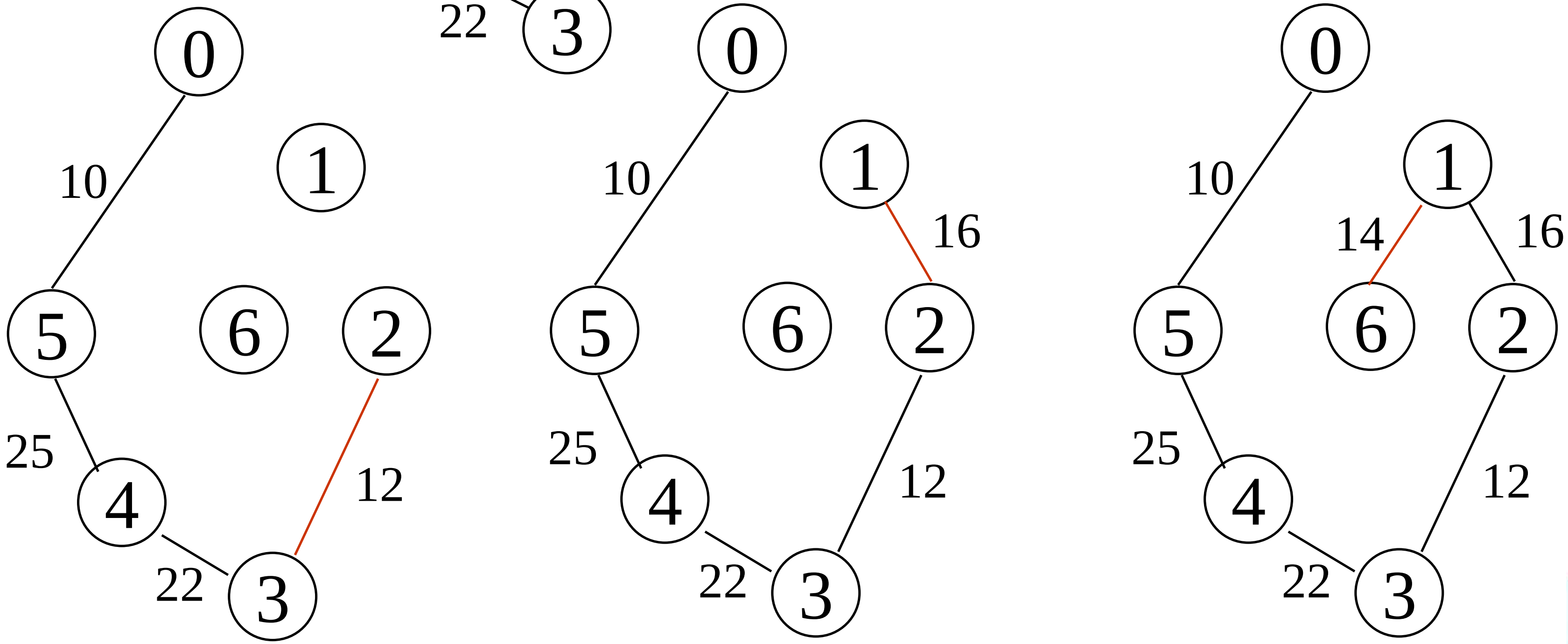
(tree all the time vs. forest)

```
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");
```

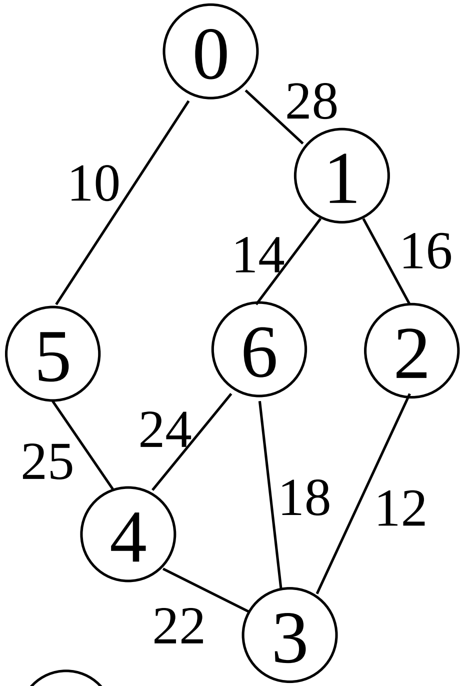
Examples for Prim’s Algorithm



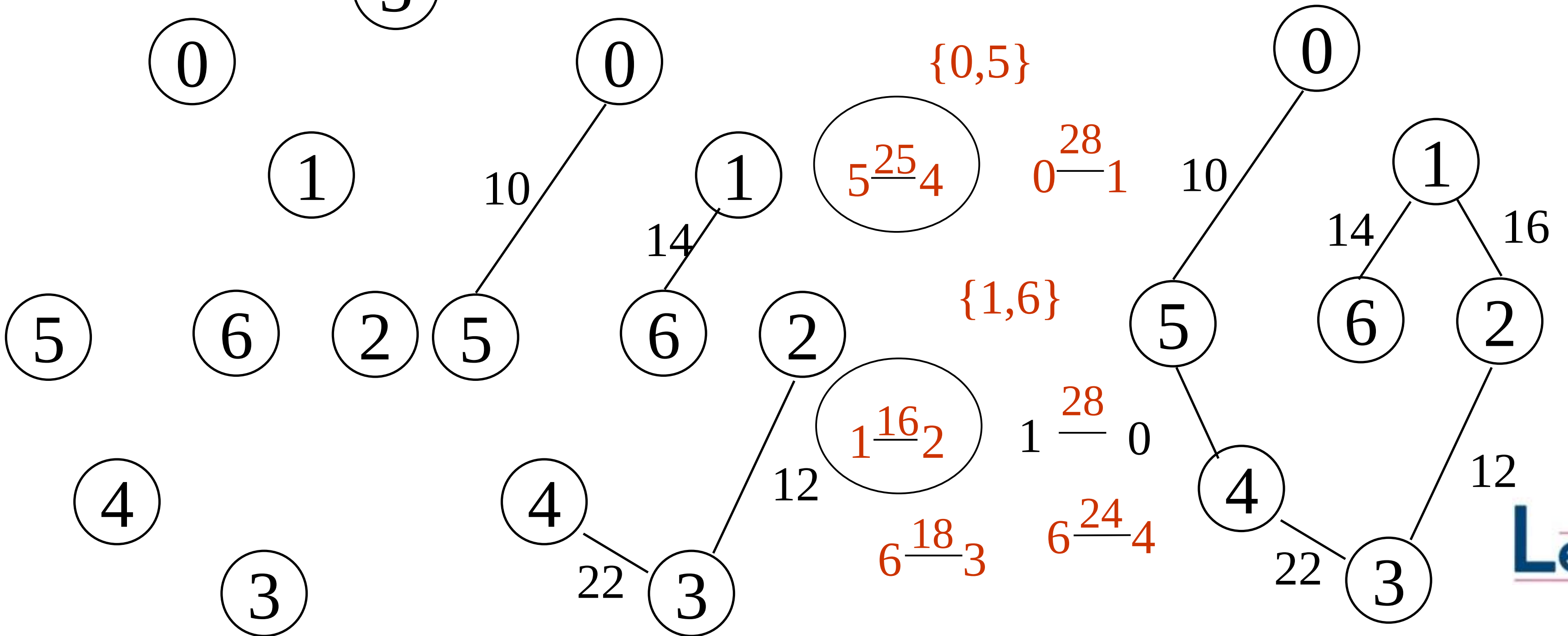

0
28
10
1
14
16
5
6
2
25
24
18
12
4
22
3


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

Sollin's Algorithm



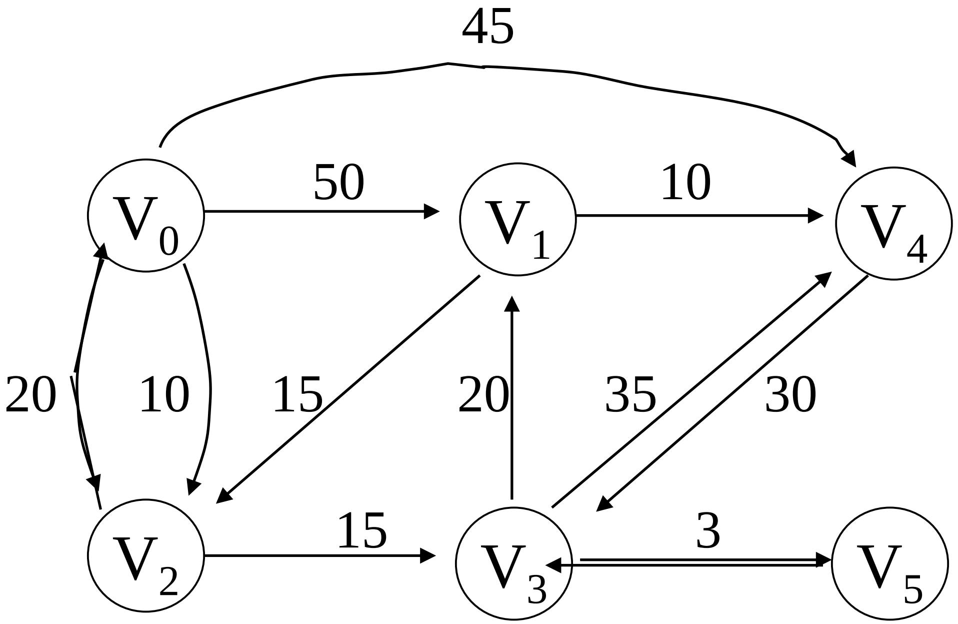
vertex	edge
0	0 -- 10 --> 5, 0 -- 28 --> 1
1	1 -- 14 --> 6, 1-- 16 --> 2, 1 -- 28 --> 0
2	2 -- 12 --> 3, 2 -- 16 --> 1
3	3 -- 12 --> 2, 3 -- 18 --> 6, 3 -- 22 --> 4
4	4 -- 22 --> 3, 4 -- 24 --> 6, 5 -- 25 --> 5
5	5 -- 10 --> 0, 5 -- 25 --> 4
6	6 -- 14 --> 1, 6 -- 18 --> 3, 6 -- 24 --> 4



Single Source All Destinations

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

*Figure 6.29: Graph and shortest paths from v_0 (p.293)

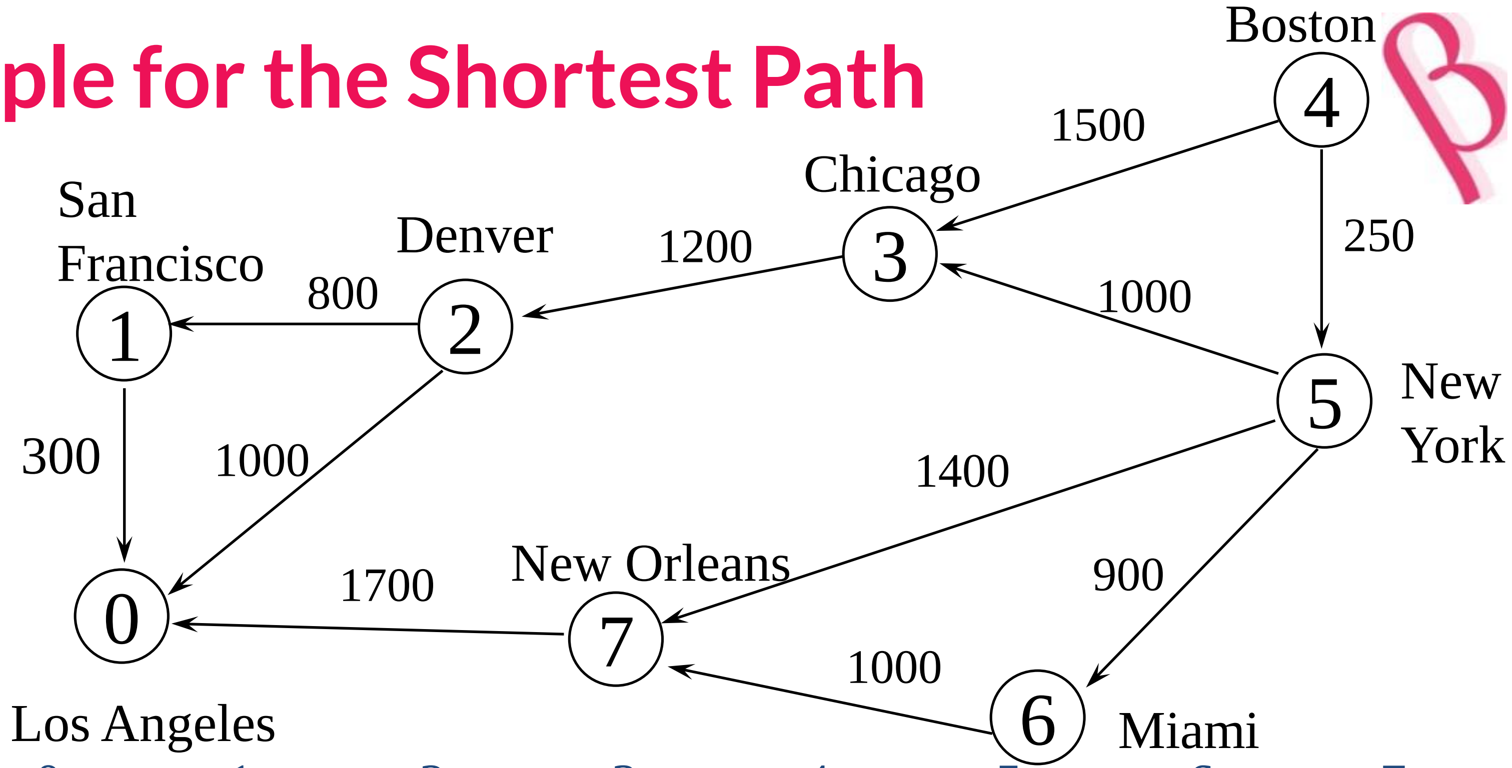


(a)

path	length
1) v0 v2	10
2) v0 v2 v3	25
3) v0 v2 v3 v1	45
4) v0 v4	45

(b)

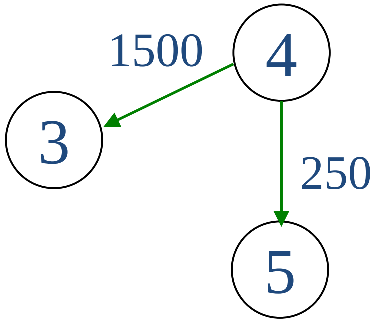
Example for the Shortest Path



	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

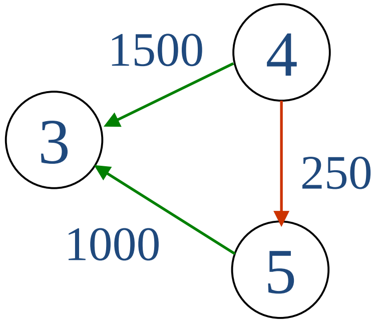
Cost adjacency matrix

(a)



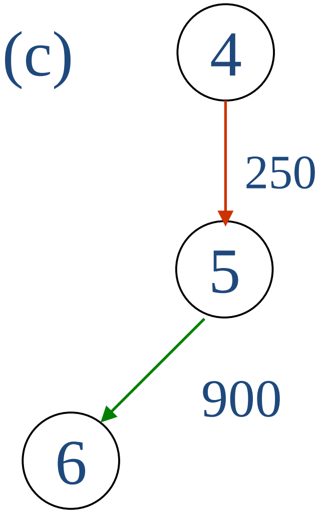
選5

(b)



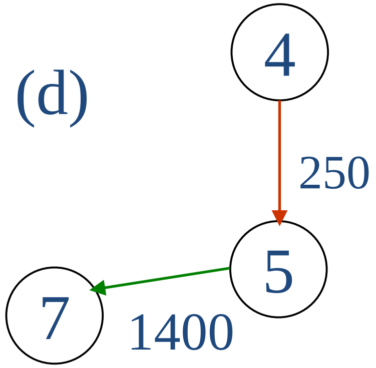
4到3由1500改成1250

(c)



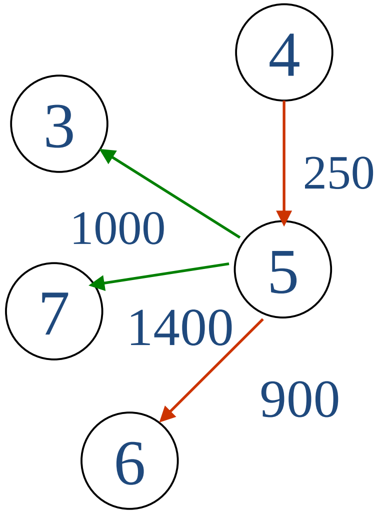
4到6由∞改成1250

(d)



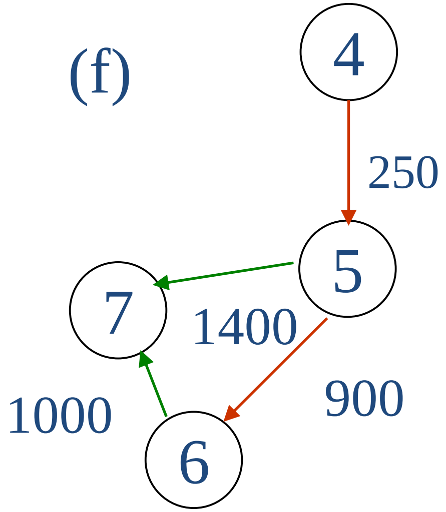
4到7由∞改成1650

(e)



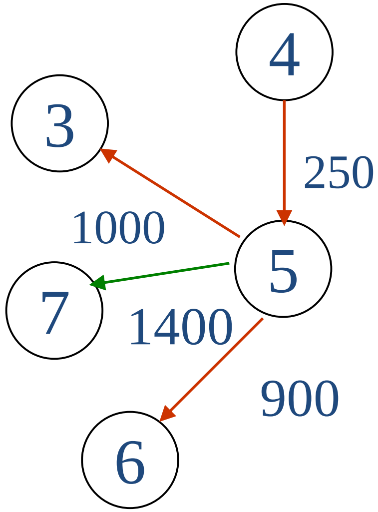
選6

(f)



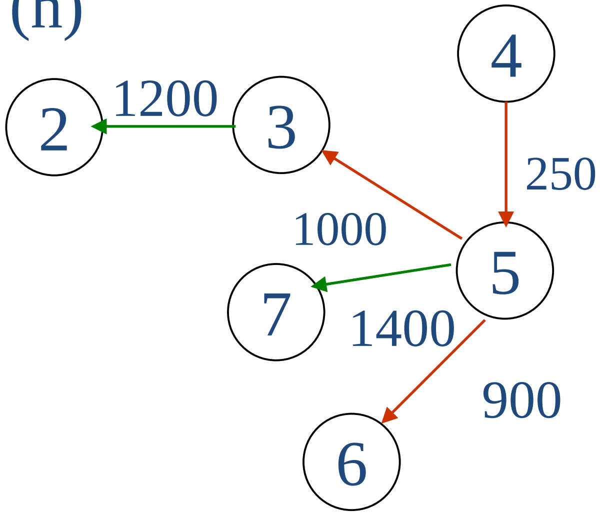
4-5-6-7比4-5-7長

(g)



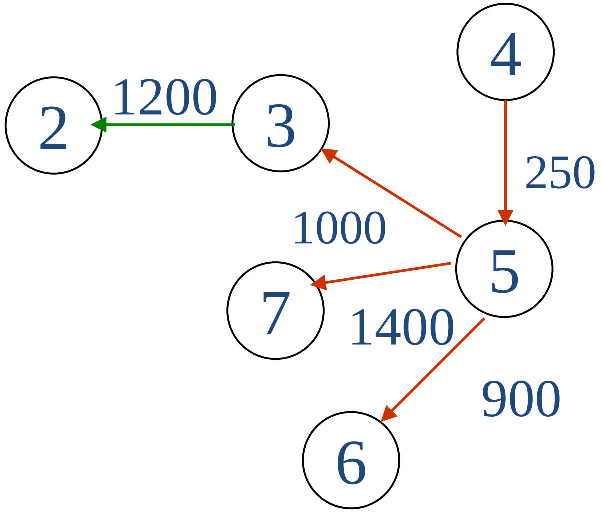
選3

(h)



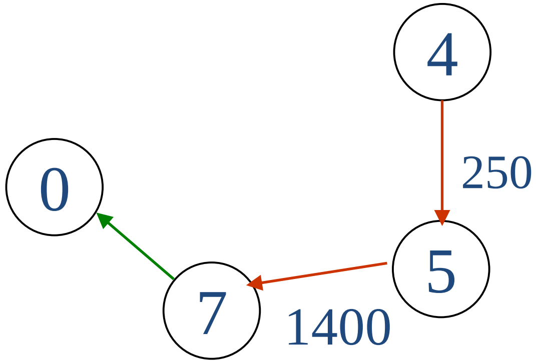
4到2由∞改成2450

(i)



選7

(j)



4到0由∞改成3350

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
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}									

Data Structure for SSAD

adjacency matrix

```
#define MAX_VERTICES 6
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;
```

Single Source All Destinations

```
void shortestpath(int v, int
  cost[][MAX_ERXTICES], int distance[], int n,
  short int found[])
{
  int i, u, w;
  for (i=0; i<n; i++) {
    found[i] = FALSE;
    distance[i] = cost[v][i];
  }
  found[v] = TRUE;
  distance[v] = 0;
```

O(n)

```
for (i=0; i<n-2; i++) {determine n-1 paths from v
  u = choose(distance, n, found);
  found[u] = TRUE;
  for (w=0; w<n; w++)
    if (!found[w]) 與u相連的端點w
      if (distance[u]+cost[u][w]<distance[w])
        distance[w] = distance[u]+cost[u][w];
 }
}
```

$O(n^2)$

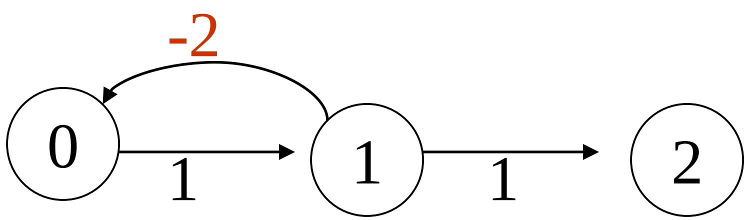
All Pairs Shortest Paths

- Find the shortest paths between all pairs of vertices.
- Solution 1
  - Apply shortest path n times with each vertex as source.
    $O(n^3)$
- Solution 2
  - Represent the graph G by its cost adjacency matrix with cost[i][j]
  - If the edge <i,j> is not in G, the cost[i][j] is set to some sufficiently large number
  - A[i][j] is the cost of the shortest path form i to j, using only those intermediate vertices with an index <= k

All Pairs Shortest Paths (Continued)

- The cost of the shortest path from i to j is $A^{n-1}[i][j]$, as no vertex in G has an index greater than n-1
- $A^{-1}[i][j]=cost[i][j]$
- Calculate the $A^0, A^1, A^2, \ldots, A^{n-1}$ from A^{-1} iteratively
- $A^k[i][j]=\min\{A^{k-1}[i][j], A^{k-1}[i][k]+A^{k-1}[k][j]\}, k>=0$

Graph with negative cycle



(a) Directed graph

$$\begin{bmatrix} 0 & 1 & \infty \\ -2 & 0 & 1 \\ \infty & \infty & 0 \end{bmatrix}$$

(b) A^{-1}

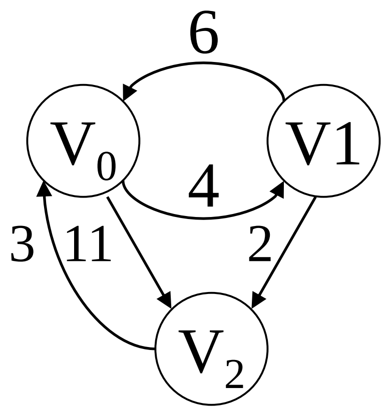
The length of the shortest path from vertex 0 to vertex 2 is $-\infty$.

0, 1, 0, 1,0, 1, …, 0, 1, 2

Algorithm for All Pairs Shortest Paths

```
void allcosts(int cost[][MAX_VERTICES],
              int distance[][MAX_VERTICES], int n)
{
  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];
}
```

Directed graph and its cost matrix

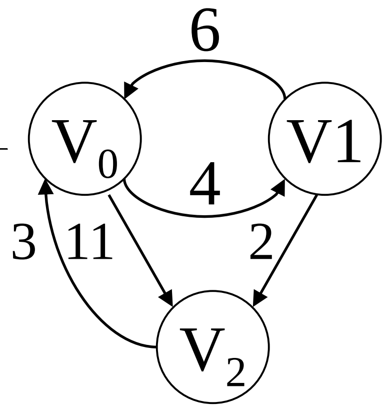


(a)Digraph G

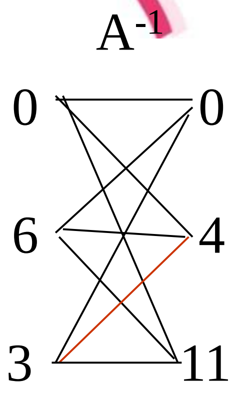
	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

(b)Cost adjacency matrix for G

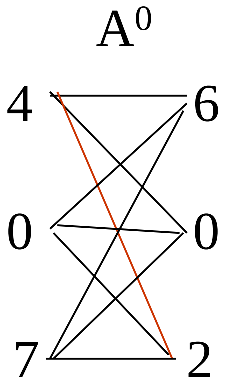
A^{-1}	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0



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



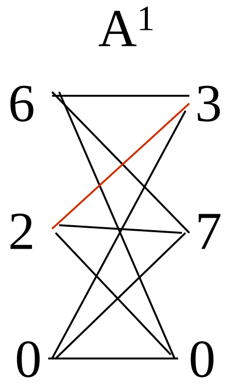
v0



v1

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



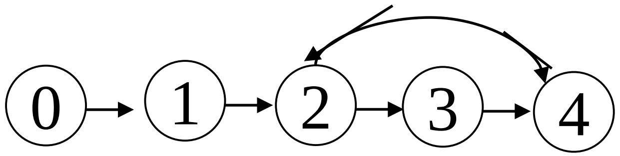
v2

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

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

(b) Adjacency matrix A for G

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

cycle

(c) transitive closure matrix A+

There is a path of length > 0

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

reflexive

(d) reflexive transitive closure matrix A*

There is a path of length ≥0

Thank You!