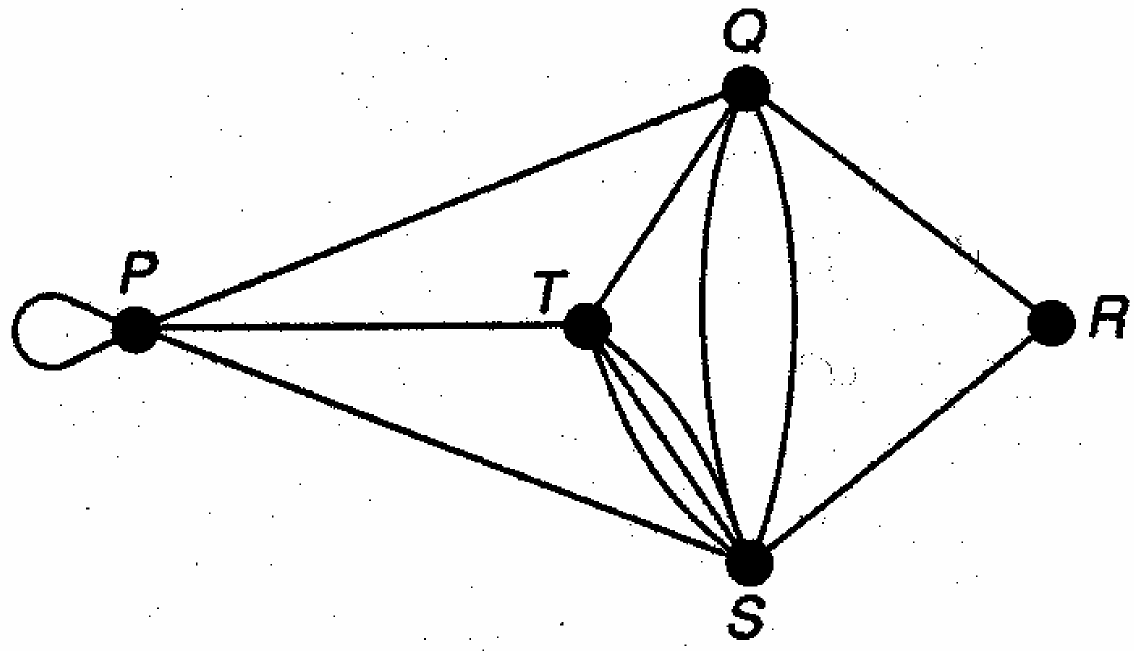
Graph properties and types

南台科大電子系
黎靖

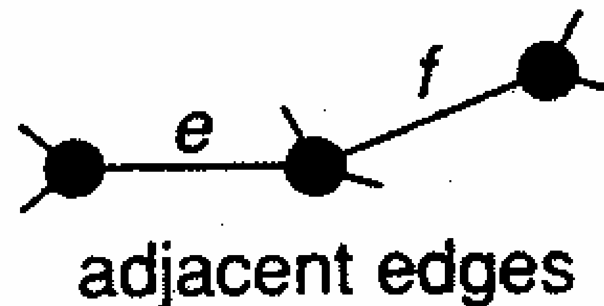
Glossary

- A **graph** is a pair of sets G = (V, E), where V is a set of **vertices**, and E is a set of pairs of vertices called **edges**. A **self-loop** is an edge whose endpoints are equal. **Multiple edges** (or **parallel edges**) are edges having the same pair of endpoints. Graphs that have parallel edges or self-loops are called **multigraphs**; graphs that have no parallel edges and self-loops are referred to as **simple graphs**.



Q
P
T
R
S

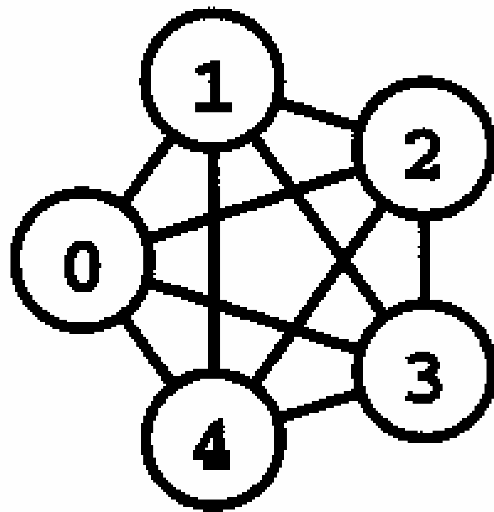
- If G is a graph, V and E is the vertex and edge sets of G, respectively. A vertex u is **adjacent to** a vertex v if (u, v) is an edge, i.e., $(u, v) \in E$. The set of vertices adjacent to v is Adj(v). An edge e = (u, v) is **incident with** the vertices u and v, which are the endpoints of e. Similarly, two distinct edges e and f are adjacent if they have a vertex in common.



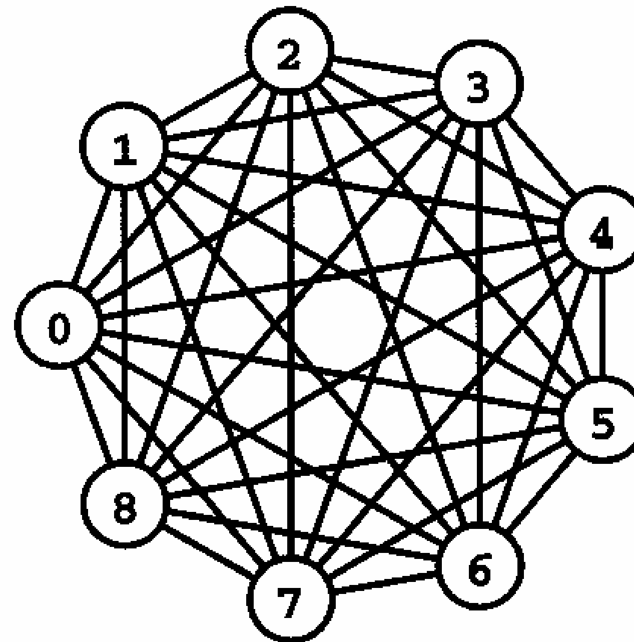
- The **degree** of a vertex u is the number of edges incident with the vertex u.

A **complete graph** on n vertices is a graph in which each vertex is adjacent to every other vertex. We use K_n to denote such a graph. A graph H is called the **complement** of graph G =(V, E) if H=(V, F), where,

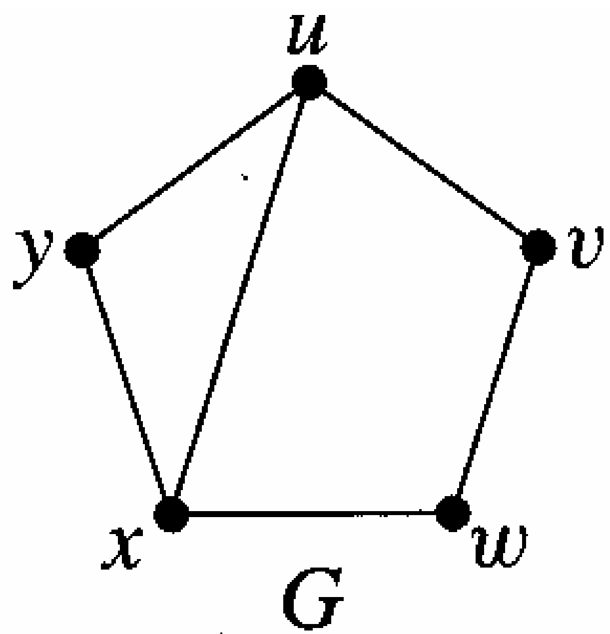
F=E($K_{|V|}$) − E.

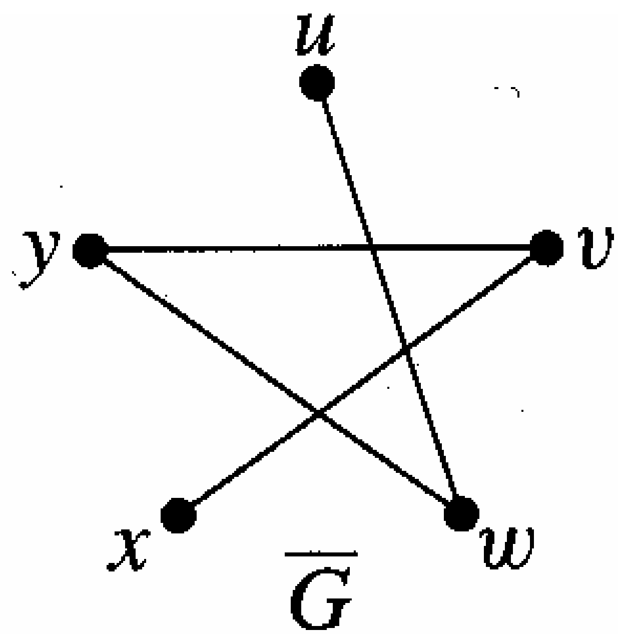


K_5

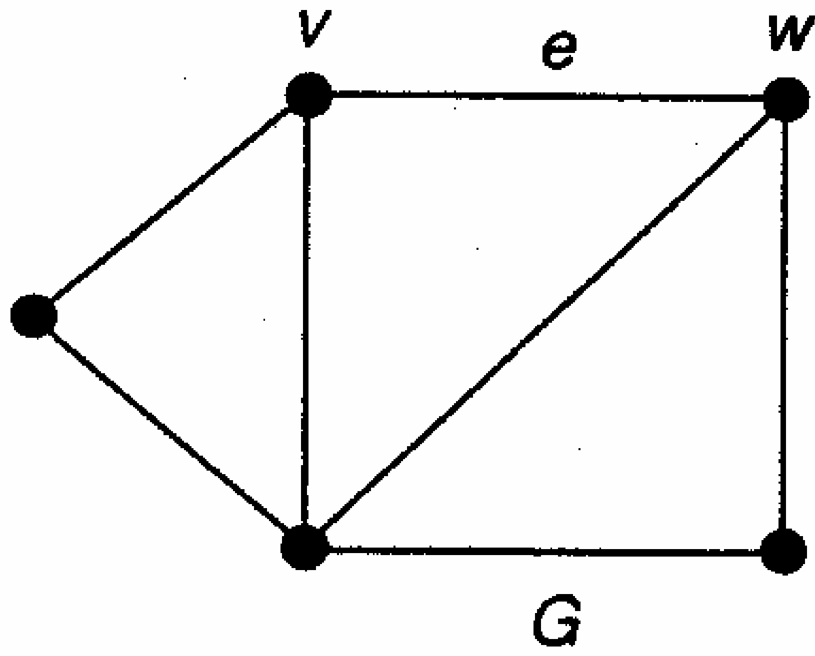


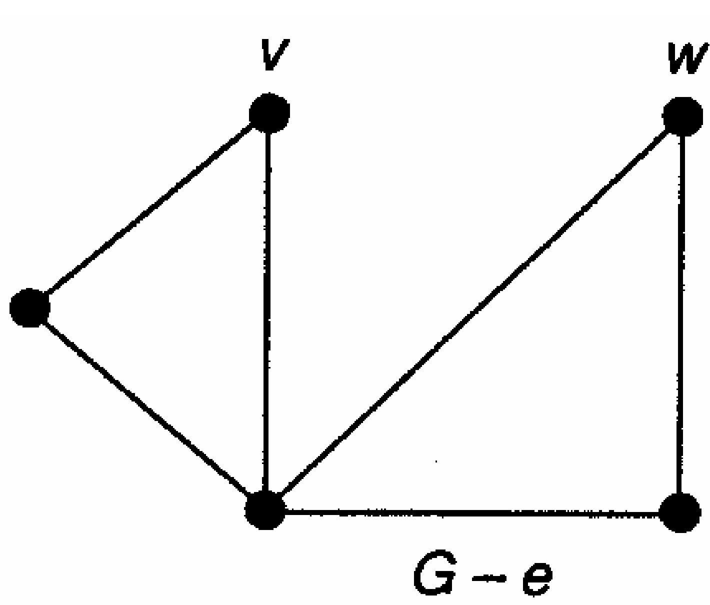
K_9

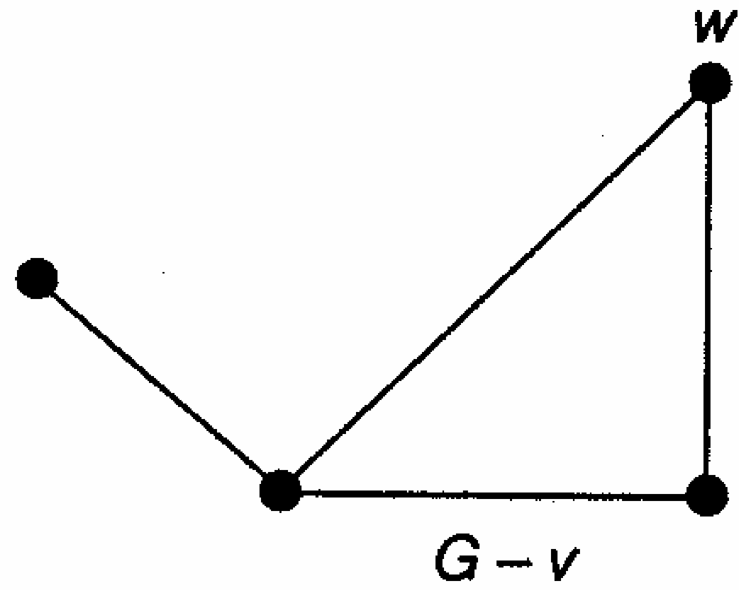

u
y
v
x
w
G


u
y
v
x
w
$\overline{G}$

- A graph $G'=(V', E')$ is a **subgraph** of a graph G if and only if $V' \subseteq V$ and $E' \subseteq E$.
- If $E' = \{(u, v) | (u, v) \in E \text{ and } u, v \subseteq V' \}$ then G' is a **vertex induced subgraph** of G. Unless otherwise stated, by subgraph we mean vertex induced subgraph.
- A complete subgraph of a graph is called a **clique**.


v
e
w
G

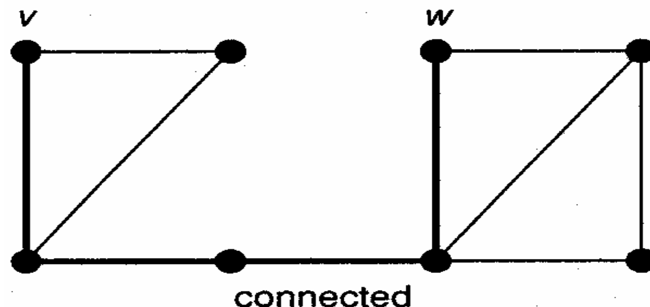

v
w
G – e


w
G – v

- A **walk** of a graph G is defined as a finite alternating sequence P= v0, e_1, …, v_{k-1}, e_k, v_k of vertices and edges, beginning and ending with vertices, such that each edge is incident with the vertices preceding and following it.
- A **tour** is a walk in which all edges are distinct. A walk is called an open walk if the terminal vertices are distinct. A path is a open walk in which no vertex appears more than once.

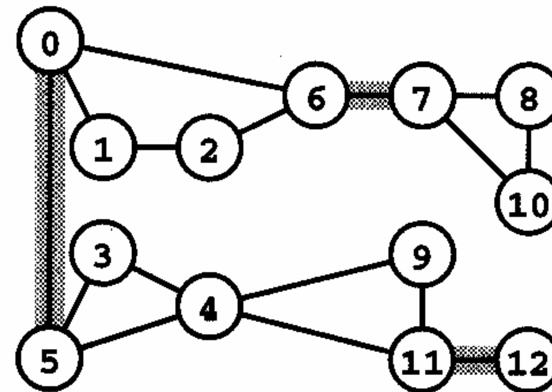
- The **length** of a path is the number of edges in it.
- A **path** is a (u, v) path if v_0= u and v_k=v.
- A **cycle** is a path of length k, k > 2 where v_0= v_k. A cycle is called **odd** if it's length k is odd, otherwise it is an **even cycle**.

- Two vertices u and v in G are **connected** if G has a (u, v) path.
- A graph is connected if all pairs of vertices are connected.
- The **distance** from u to v is the length of the shortest path from u to v.
- A **connected component** of G is a maximal connected subgraph of G.
- A **disconnecting set** in a connected graph G is a set of edges whose removal disconnects G. For example, in the right graph, the sets {e1, e2, e5} and {e3, e6, e7, e8} are both disconnecting sets of G.



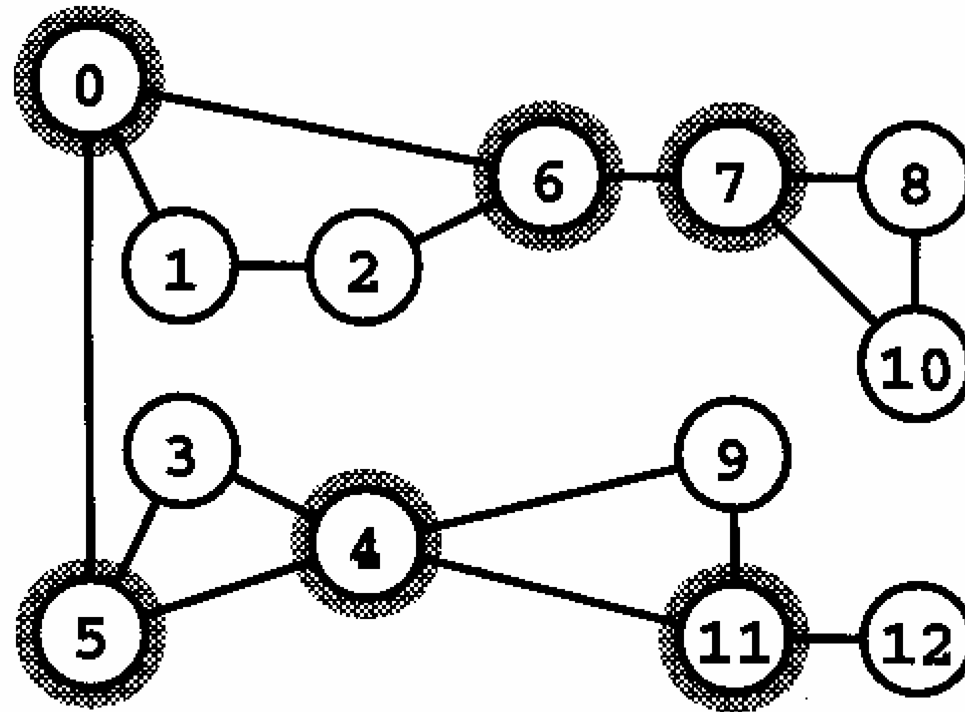
- A **cutset** is defined to be a disconnecting set, no proper subset of which is a disconnecting set. In the above example, only the second disconnecting set is a cutset. Note that the removal of the edges in a cutset always leaves a graph with exactly two components.
- If a cutset has only one edge e, we call e a **bridge**. These definitions can be extended to disconnected graph.

- If G is connected, its **edge connectivity** $\lambda(G)$ is the size of the smallest cutset in G. We also say that G is **k-edge connected** if $\lambda(G) \geq k$.
- A graph that has no bridges is said to be **edge-connected**. A graph that is not edge-connected is called as an **edge-separable** graph.



An edge-separable graph

- A **separating set** in a connected graph G is a set of vertices whose deletion disconnects G; recall that when we delete a vertex, we also remove its incident edges.
- If a separating set contains only one vertex v, we call v a **cut-vertex** (or **articulation point**, or **separation vertex**).
- A graph is **biconnected** if and only if it has no separation vertices.



Articulation points

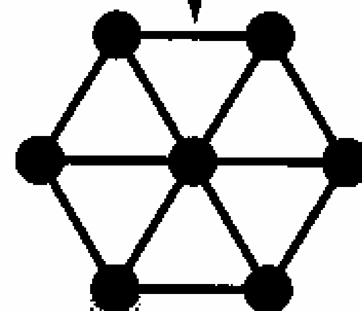
- An acyclic connected graph is called a **tree** (**free tree**). A graph is tree if it satisfies any of the following four conditions:

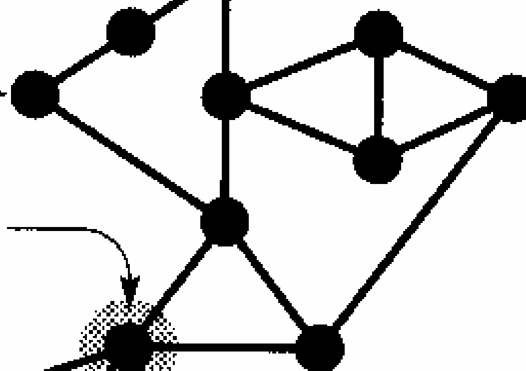
  (1) G has $|E|-1$ edges and no cycles.

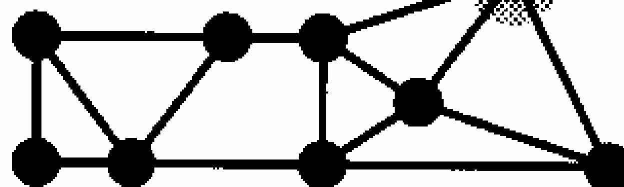
  (2) G has $|E|-1$ edges and is connected.

  (3) Exactly one simple path connects each pair of vertices in G.

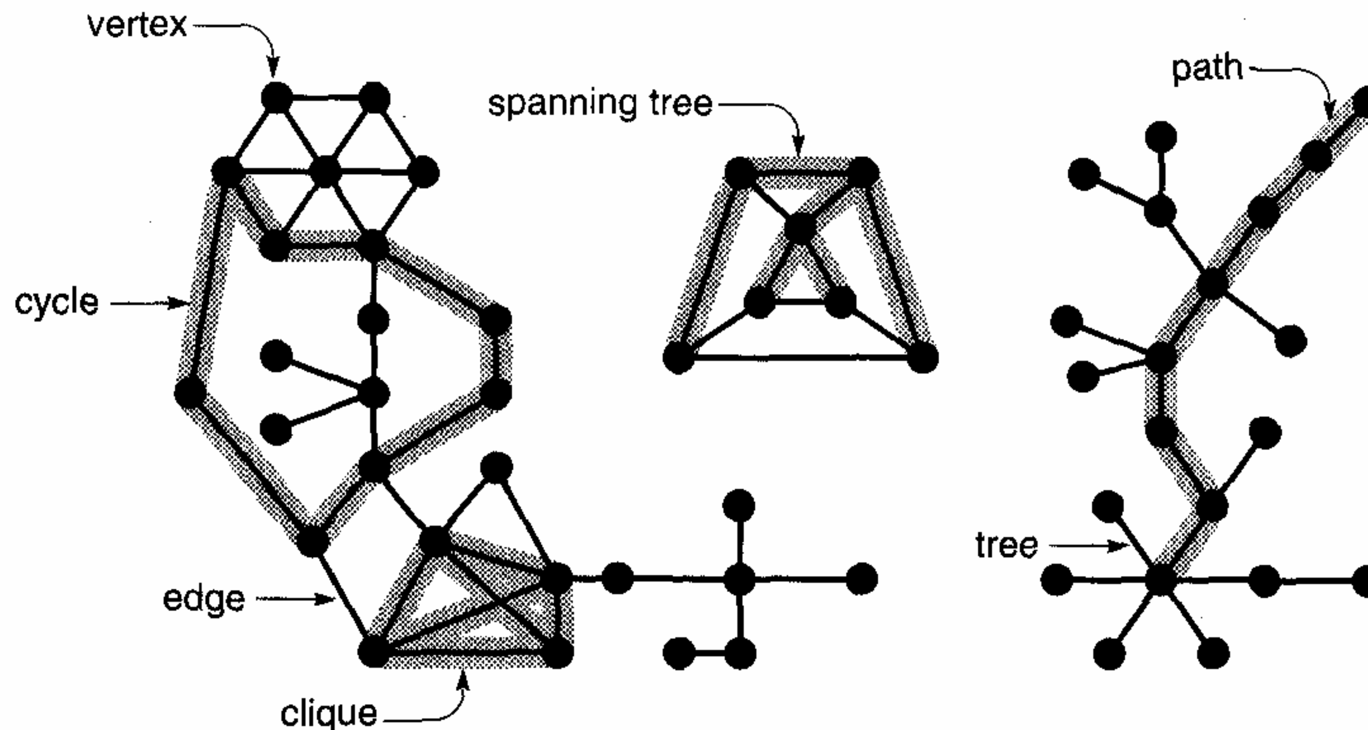
  (4) G is connected, but does not remain connected if any edge is removed.





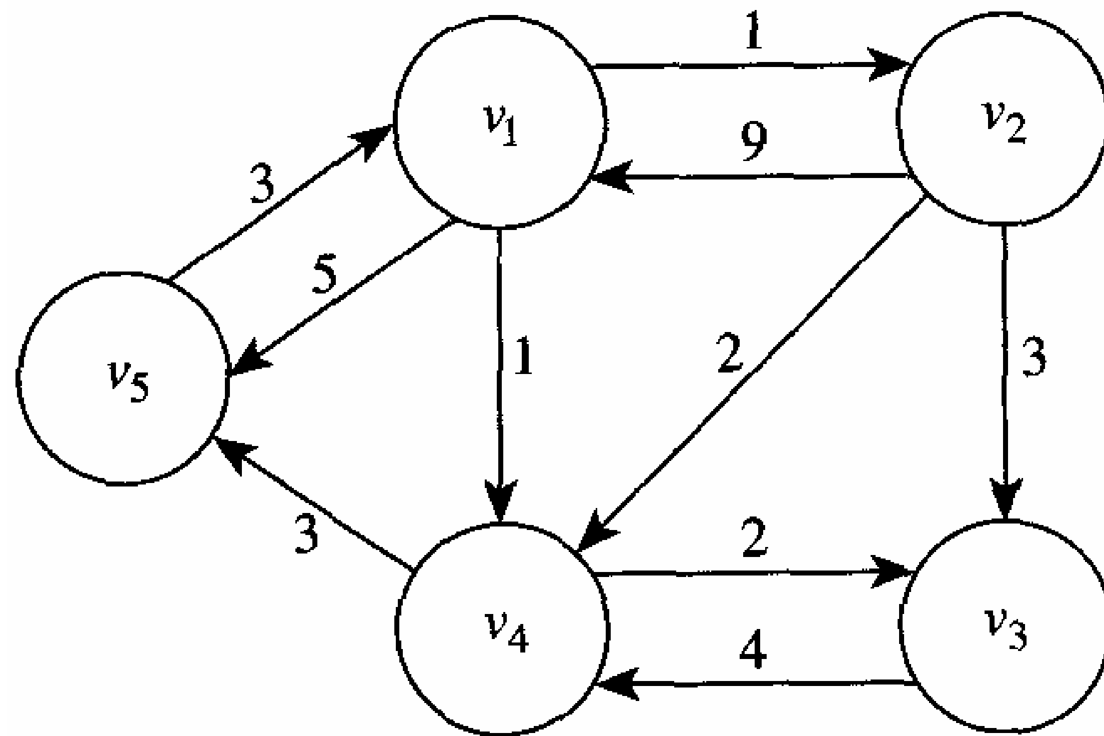

biconnected component
bridge
edge-connected
component
articulation point

- A set of tree is called a **forest**.
- A **spanning tree** of a connected graph is a subgraph that contains all of that graph's vertices and is a single tree.
- A **spanning forest** of a graph is a subgraph that contains all of that graph's vertices and is a forest.



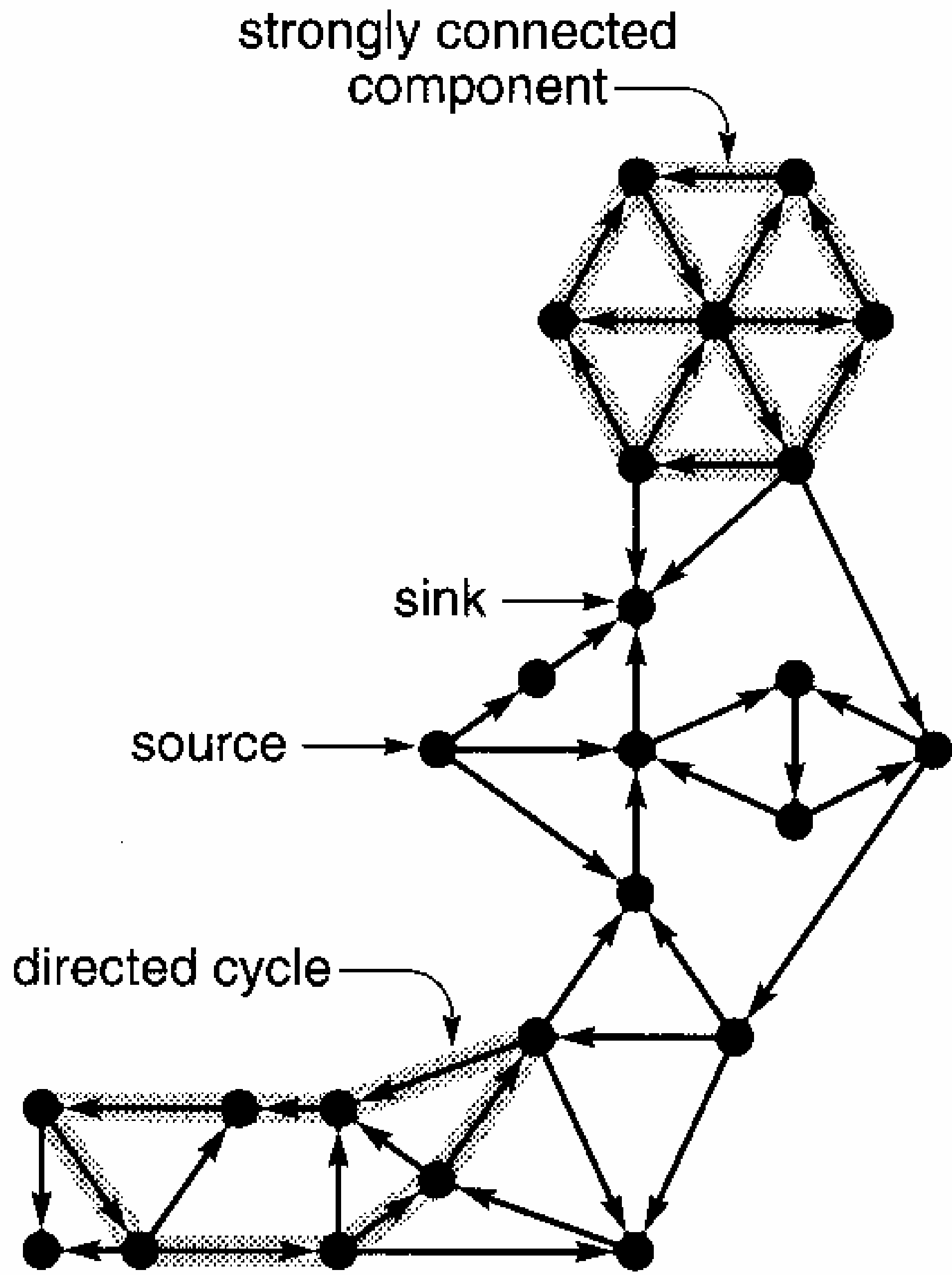
- A **directed graph** (**digraph**) is a pair of sets (V, $\vec{E}$), where V is a set of vertices and $\vec{E}$ is a set of ordered pairs of distinct vertices, called **directed edges**.
- A directed edge $\vec{e}$ = (u, v) is incident with u and v, the vertices u and v are the **head** and **tail** of $\vec{e}$, respectively; $\vec{e}$ is an **in-edge** of v and an **out-edge** of u.
- The **in-degree** of u denoted by $d^-(u)$ is equal to the number of in-edges of u, similarly the **out-degree** of u denoted by $d^+(u)$ is equal to the number of out-edges of u.

- An **orientation** for a graph G =(V,E) is an assignment of direction for each edge.
- An orientation is called **transitive** if, for each pair of edges (u, v) and (v, w), there exists an edge (u, w). If such a transitive orientation exists for a graph G, then G is called a **transitively orientable graph**.
- Definitions of subgraph, path, and walk are easily extended to directed graphs.

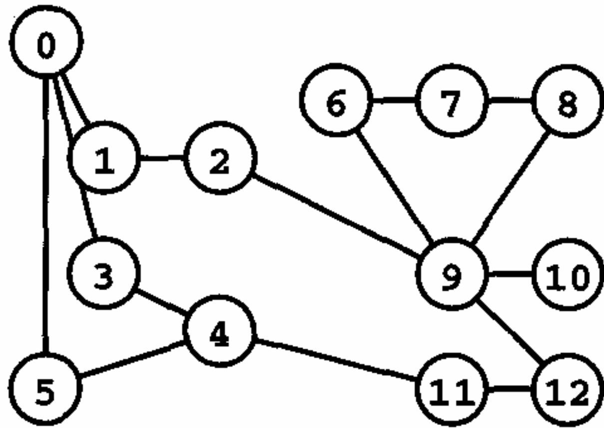


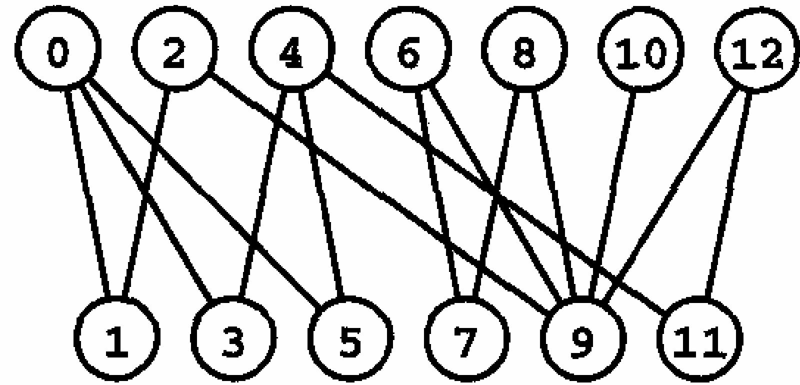
A weighted, directed graph.

- A **directed acyclic graph** is a directed graph with no directed cycles.
- A vertex u is an **ancestor** of v (and v is a **descendent** of u) if there is a (u, v) directed path in G.
- A **rooted tree** (or **directed tree**) is a directed acyclic graph in which all vertices have in-degree 1 except the root, which has in-degree 0. The root of a rooted tree T is denoted root(T).
- The **subtree** of tree T rooted at y is the subtree of T induced by the descendents of u.
- A **leaf** is a vertex in a directed acyclic graph with no descendents.

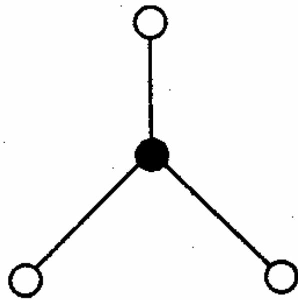

strongly connected
component
sink
source
directed cycle

- A **bipartite** graph is a graph G whose vertex set can be partitioned into two subsets X and Y, so that each edge has one end in X and one end in Y; such a partition (X,Y) is called bipartition of the graph.
- A **complete bipartite graph** is a bipartite graph with bipartition (X,Y) in which each vertex of X is adjacent to each vertex of Y; if |X|=m and lY|=n, such a graph is denoted by $\mathbf{K_{m,n}}$.
- An important characterization of bipartite graphs is in terms of odd cycles. A graph is bipartite if and only if it does not contain an odd cycle.

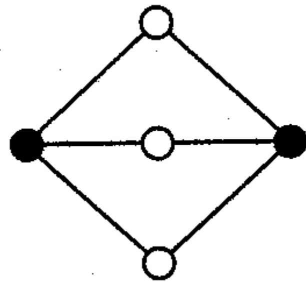




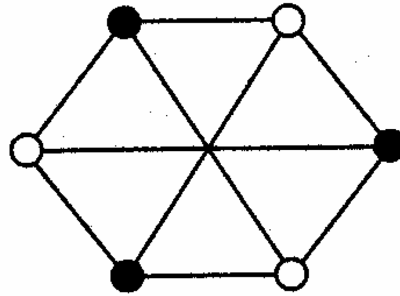
Bipartite graph



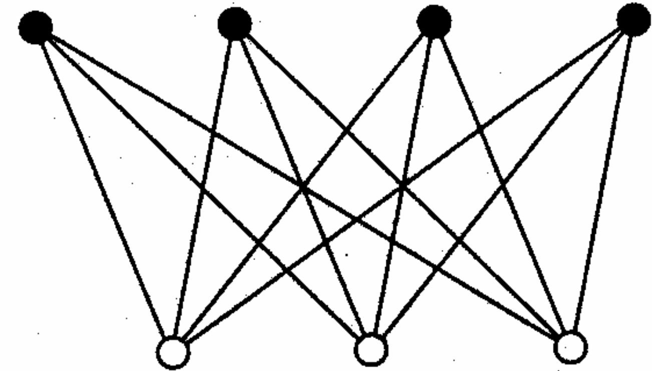
$K_{1,3}$



$K_{2,3}$



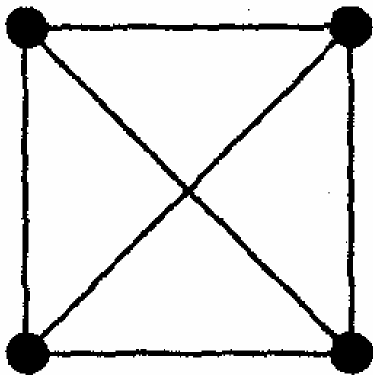
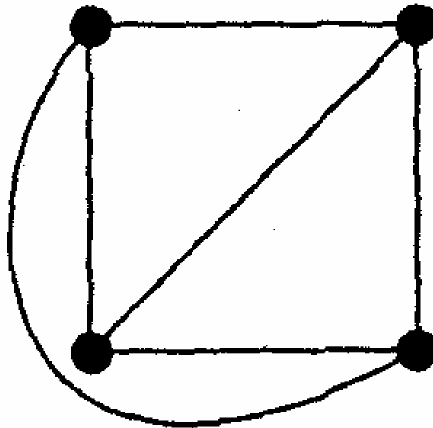
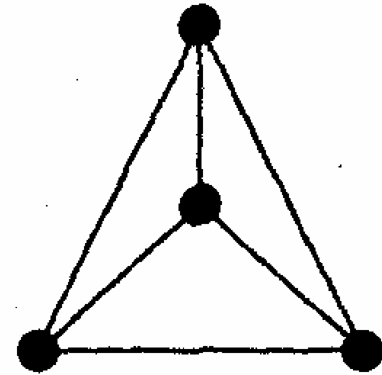
$K_{3,3}$



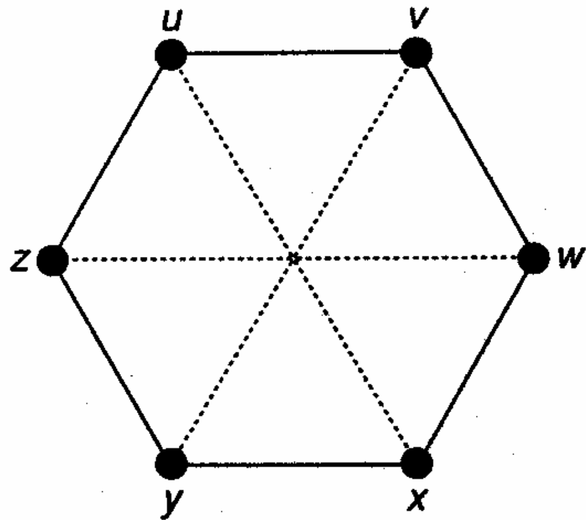
$K_{4,3}$

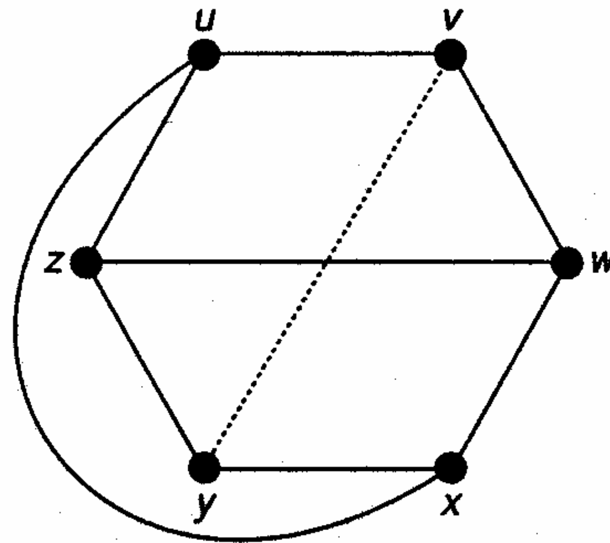
Complete bipartite graph

- A graph is called **planar** if it can be drawn in the plane without any two edges crossing. For example, K_4 is a planar graph, while $K5$ and $K_{3,3}$ is non-planar.

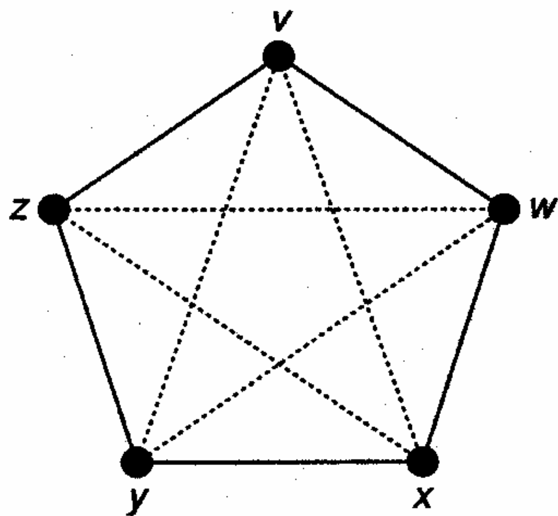




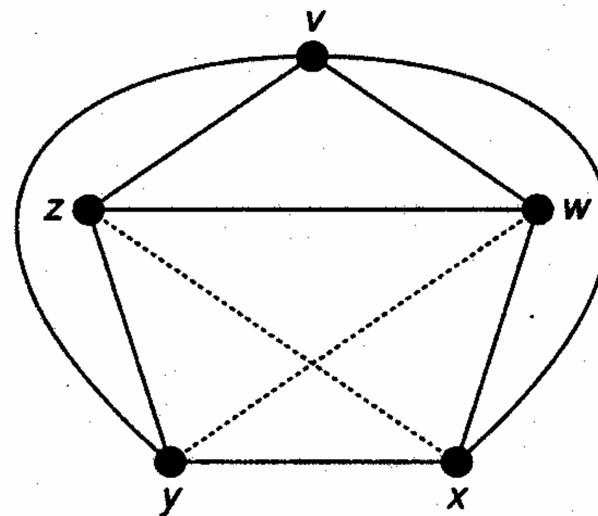
K_4





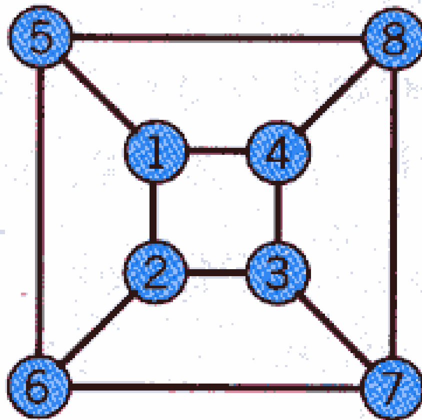
$K_{3,3}$



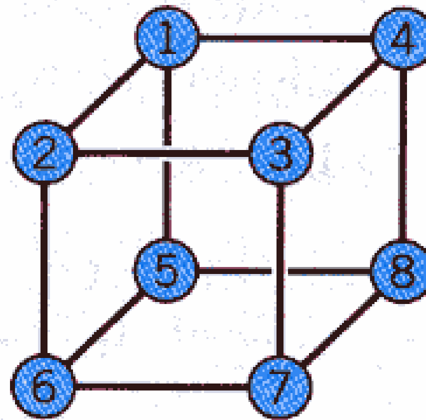


K_5

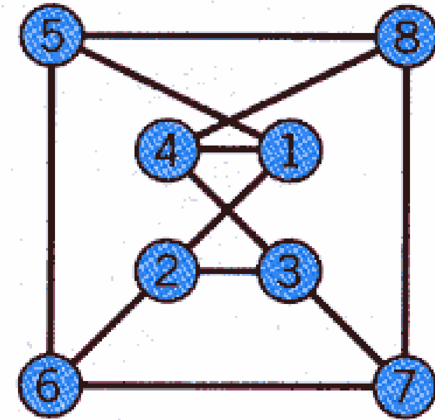
- Notice that there are many different ways of ‘drawing' a planar graph. A drawing may be obtained by mapping a vertex to a point in the plane and mapping edges to paths in the plane. Each such drawing is called an **embedding** of G.



(a)



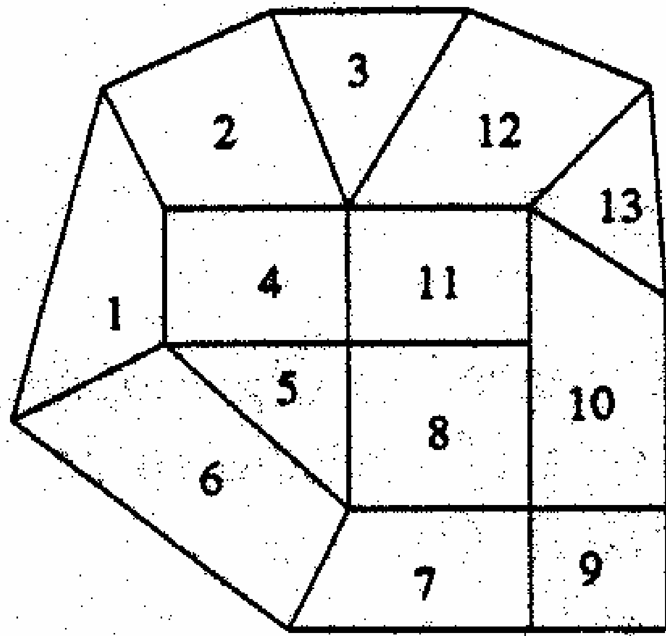
(b)

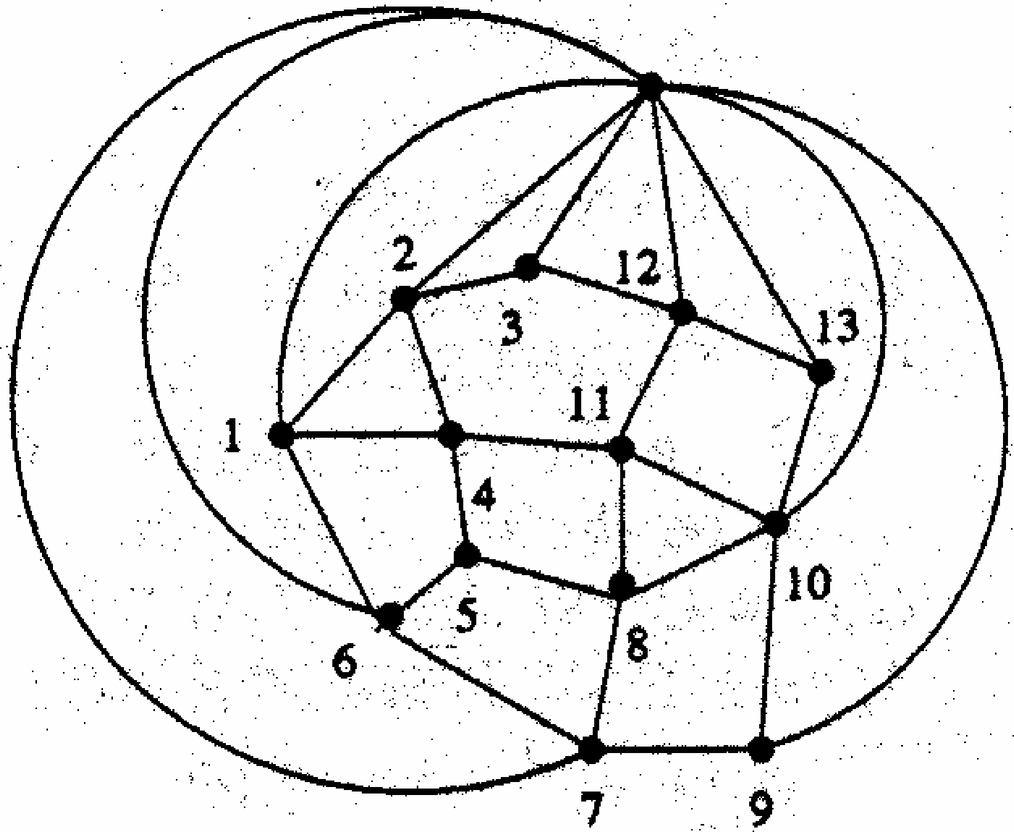


(c)

Different sketches of the same graph

- An embedding divides the plane into finite number of regions. The edges which bound a region define a **face**. The unbounded region is called the external or outside face.
- A face is called an **odd face** if it has odd number of edges. Similarly a face with even number of edges is called an **even face**.
- The **dual** of a planar embedding T is a graph $G_T=(V_T, E_T)$, such the $V_T=\{v \mid v \text{ is a face in } T\}$ and two vertices share an edge if their corresponding faces share an edge in T.


3
2
12
13
1
4
11
5
10
8
6
7
9


2
12
3
13
11
1
4
10
5
8
6
7
9

Density of a Graph

- Most graphs that we encounter in practice have relatively few of the possible edges present. To quantify this concept, we define the **density** of a graph to be the average vertex degree, or 2E/V.
- A **dense graph** is a graph whose average vertex degree is proportional to V; a **sparse graph** is a graph whose complement is dense. In other words, we consider a graph to be dense if E is proportional to V^2 and sparse otherwise.

- A **hypergraph** is a pair (V, E), where V is a set of vertices and E is a family of sets of vertices.
- A hypergraph is a sequence P= v_0, e_1, …, v_{k-1}, e_k, v_k of distinct vertices and distinct edges, such that vertices v_{i-1} and v_i are elements of the edge ei, $1 \leq i \leq k$.
- Two vertices u and v are connected in a hypergraph if the hypergraph has a (u, v) hyperpath.
- A hypergraph is **connected** if every pair of vertices is connected.

Graph ADT

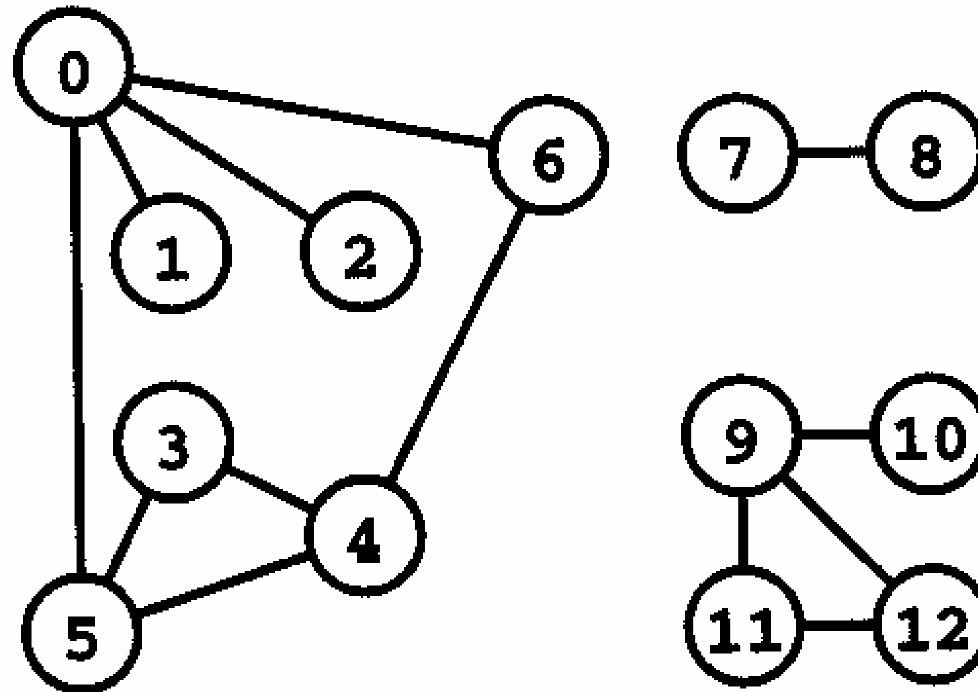
- typedef struct { int v; int w; } Edge;
- Edge EDGE(int, int);
- typedef struct graph *Graph;
- Graph GRAPHinit(int);
- void GRAPHinsertE(Graph, Edge);
- void GRAPHremoveE(Graph, Edge);
- int GRAPHedges(Edge [], Graph G);
- Graph GRAPHcopy(Graph);
- void GRAPHdestroy(Graph);

Example of a graph processing client

This program takes V and E from standard input, generates a random graph with V vertices and E edges, prints the graph if it is small, and computes (and prints) the number of connected components.

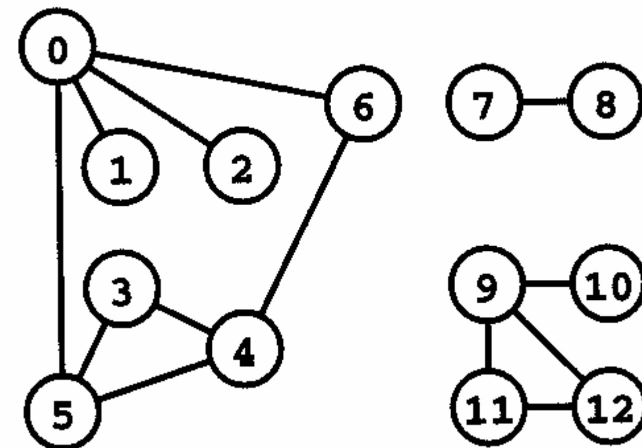
```
#include <stdio.h>
#include "GRAPH.h“
main(int argc, char *argv[])
 {
 int V = atoi(argv[1]), E = atoi(argv[2]);
 Graph G = GRAPHrand(V, E);
 if (V < 20)   GRAPHshow(G);
 else printf("%d vertices, %d edges, ", V, E);
  printf("%d component(s)\n", GRAPHcc(G));
 }
```

Adjacency-Matrix Representation



Adjacency matrix data structure

0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	0	1	0	0	1
12	0	0	0	0	0	0	0	0	0	1	0	1	0



Graph ADT implementation

```
#include <stdlib.h>
#include "GRAPH.h“
struct graph { int V; int E; int **adj; };
Graph GRAPHinit(int V)
 {
Graph G = malloc(sizeof *G);
G->V = V; G->E = 0;
G->adj = MATRIXint(V, V, 0);
return G;  }
```

```
void GRAPHinsertE(Graph G, Edge e)
{
int v = e.v, w = e.w;
if (G->adj[v][w] == 0) G->E++;
G->adj[v][w] = 1;
G->adj[w][v] = 1;
}
```

```
void GRAPHremoveE(Graph G, Edge e)
{
int v = e.v, w = e.w;
if (G->adj[v][w] == 1) G->E--;
G->adj[v][w] = 0;
G->adj[w][v] = 0;
}
```

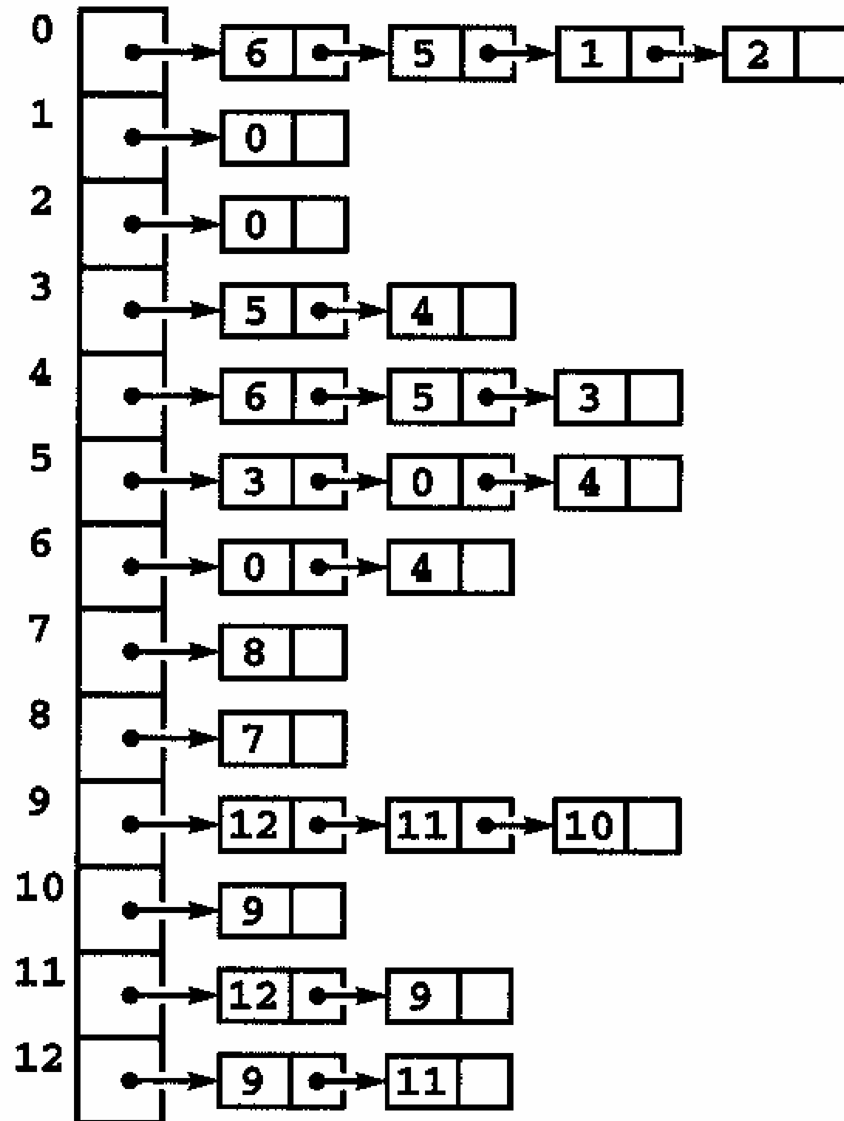
```
int GRAPHedges(Edge a[ ], Graph G)
{
int v, w, E = 0;
for (v = 0; v < G->V; v++)
   for (w = v+1; w < G->V; w++)
      if (G->adj[v][w] == 1)
         a[E++] = EDGE(v, w);
return E;
}
```

Adjacency-matrix allocation and initialization

```
int **MATRIXint(int r, int c, int val)
 { int i, j;
 int **t = malloc(r * sizeof(int *));
 for (i = 0; i < r; i++)
       t[i] = malloc(c * sizeof(int));
 for (i = 0; i < r; i++)
     for (j = 0; j < c; j++)        t[i][j] = val;
 return t;  }
```

Adjacency-Lists representation

Adjacency-lists data structure



Graph ADT implementation

```
#include <stdlib.h>
#include "GRAPH.h“
typedef struct node *link;
struct node { int v; link next; };
struct graph { int V; int E; link *adj; };
link NEW(int v, link next)
  {
  link x = malloc(sizeof *x);
  x->v = v; x->next = next;
  return x;                        }
```

```
Graph GRAPHinit(int V)
{
int v;    Graph G = malloc(sizeof *G);
G->V = V; G->E = 0;
G->adj = malloc(V*sizeof(link));
for (v = 0; v < V; v++) G->adj[v] = NULL;
return G;  }
```

```
void GRAPHinsertE(Graph G, Edge e)
 {
 int v = e.v, w = e.w;
 G->adj[v] = NEW(w, G->adj[v]);
 G->adj[w] = NEW(v, G->adj[w]);
 G->E++;
 }
```

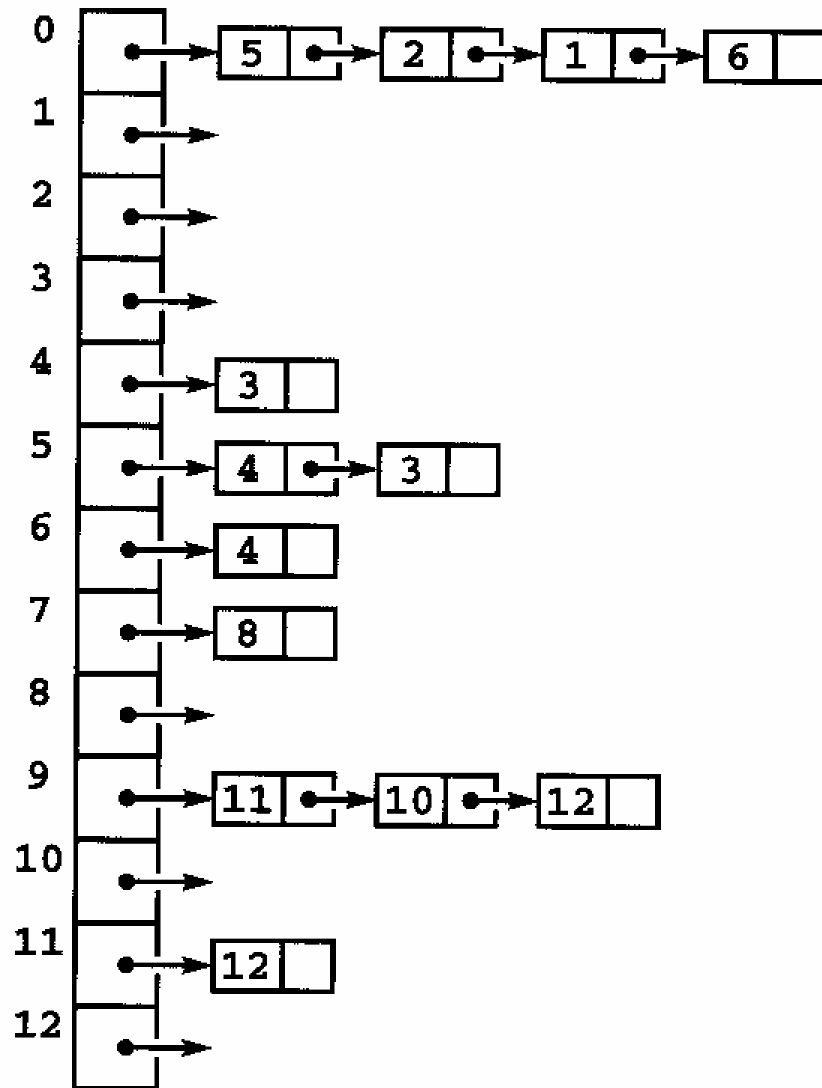
```
int GRAPHedges(Edge a[], Graph G)
 {
 int v, E = 0; link t;
 for (v = 0; v < G->V; v++)
   for (t = G->adj[v]; t != NULL; t = t->next)
        if (v < t->v) a[E++] = EDGE(v, t->v);
  return E;  }
```

Digraph representations

Adjacency-matrix data structure

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	1	1	0	0	1	1	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0	0	0	0	0
3	0	0	0	1	0	0	0	0	0	0	0	0	0
4	0	0	0	1	1	0	0	0	0	0	0	0	0
5	0	0	0	1	1	1	0	0	0	0	0	0	0
6	0	0	0	0	1	0	1	0	0	0	0	0	0
7	0	0	0	0	0	0	0	1	1	0	0	0	0
8	0	0	0	0	0	0	0	0	1	0	0	0	0
9	0	0	0	0	0	0	0	0	0	1	1	1	1
10	0	0	0	0	0	0	0	0	0	0	1	0	0
11	0	0	0	0	0	0	0	0	0	0	0	1	1
12	0	0	0	0	0	0	0	0	0	0	0	0	1

Adjacency-lists data structure



Graph Generators

Random graph generator (random edges)

```
int randV(Graph G)
{
return G->V * (rand() / (RAND_MAX + 1.0));
}

Graph GRAPHrand(int V, int E)
 {
 Graph G = GRAPHinit(V);
 while (G->E < E)
    GRAPHinsertE(G, EDGE(randV(G), randV(G)));
  return G;
 }
```

Random graph generator (random graph)

```
Graph GRAPHrand(int V, int E)
 {
 int i, j;
 double p = 2.0*E/V/(V-1);
 Graph G = GRAPHinit(V);
 for (i = 0; i < V; i++)
   for (j = 0; j < i; j++)
     if (rand() < p*RAND_MAX)
           GRAPHinsertE(G, EDGE(i, j));
 return G;
 }
```

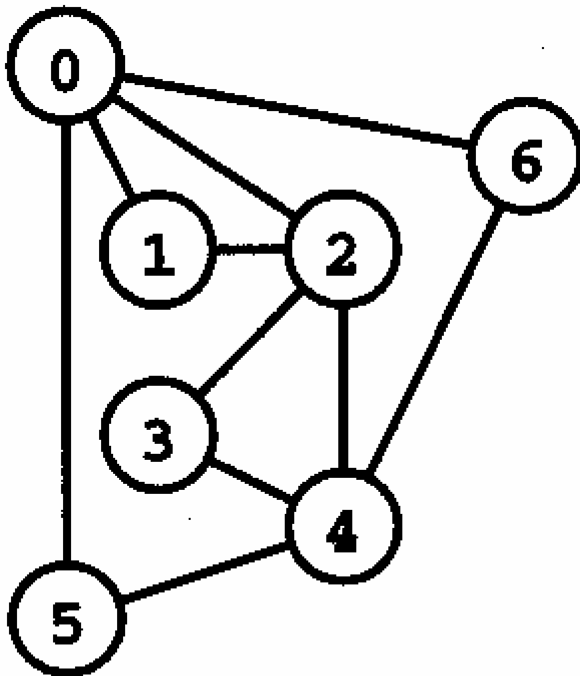
Simple, Euler, and Hamilton Paths

Simple path

```
static int visited[maxV];
int pathR(Graph G, int v, int w)
 { int t;
 if (v == w) return 1;
 visited[v] = 1;
 for (t = 0; t < G->V; t++)
    if (G->adj[v][t] == 1)
       if (visited[t] == 0)
         if (pathR(G, t, w)) return 1;
  return 0;  }

int GRAPHpath(Graph G, int v, int w)
   { int t;
  for (t = 0; t < G->V; t++) visited[t] = 0;
  return pathR(G, v, w);  }
```

Find a simple path from 2 to 6 in the graph by call pathR(G, 2, 6). Trace the search.



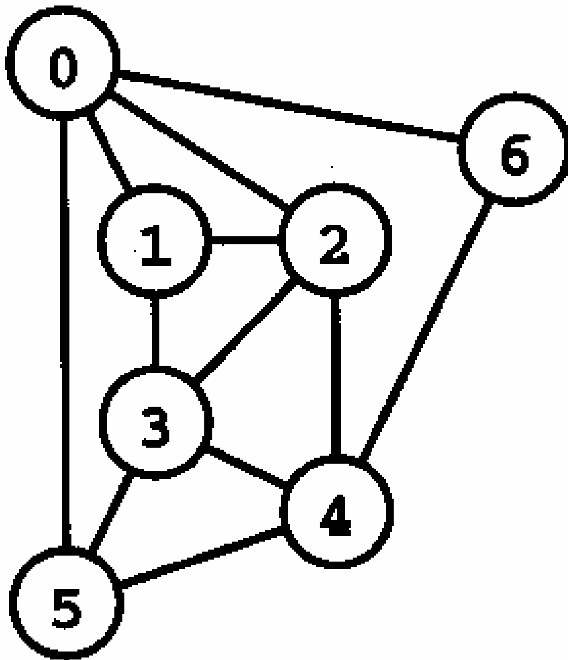

```
2-0 pathR(G, 0, 6)
  0-1 pathR(G, 1, 6)
    1-0
    1-2
  0-2
  0-5 pathR(G, 5, 6)
    5-0
    5-4 pathR(G, 4, 6)
      4-2
      4-3 pathR(G, 3, 6)
        3-2
        3-4
      4-6 pathR(G, 6, 6)
```

Hamilton paths and Hamiltonian cycles

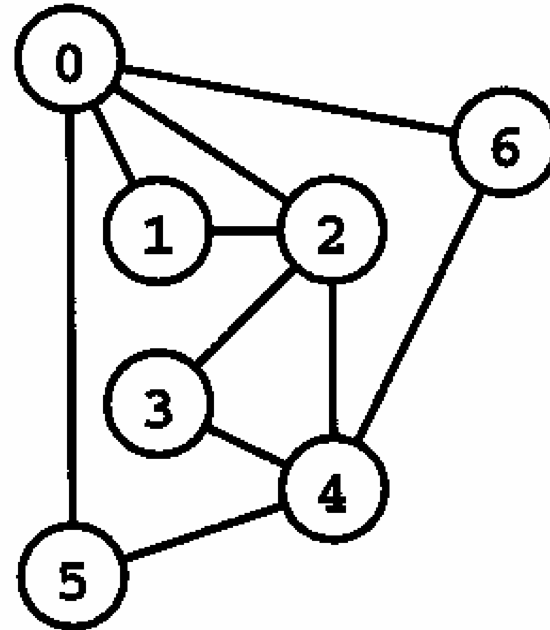
- A path between two vertices in a graph is a **Hamilton path** if it passes through each vertex exactly once.
- A closed path that passes through each vertex exactly once and in which all the edges are distinct is a **Hamiltonian cycle**.
- A graph is a **Hamiltonian graph** if it has a Hamiltonian cycle.
- In a digraph a directed path from a vertex to another vertex is a **directed Hamiltonian path** if it passes through each vertex exactly once.
- A closed directed Hamiltonian path is a **directed Hamiltonian cycle**.

Examples:

(1) **Graph with Hamilton tour**



(2) **Graph without Hamilton tour**



Program to search Hamilton path

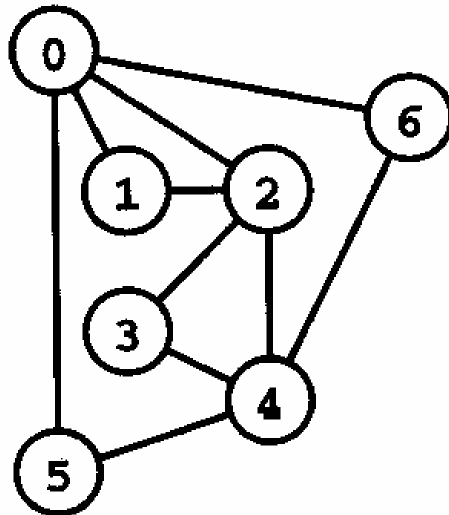
```
static int visited[maxV];
int pathR(Graph G, int v, int w, int d)
  { int t;
  if (v == w)
       { if (d == 0) return 1; else return 0; }
  visited[v] = 1;
  for (t = 0; t < G->V; t++)
     if (G->adj[v][t] == 1)
       if (visited[t] == 0)
          if (pathR(G, t, w, d-1)) return 1;
    visited[v] = 0;    return 0;  }

int GRAPHpathH(Graph G, int v, int w)
 { int t;
  for (t = 0; t < G->V; t++) visited[t] = 0;
  return pathR(G, v, w, G->V-1);  }
```

Property 17.3

A recursive search for a Hamilton tour could take exponential time.

Example: Hamilton-tour-search trace



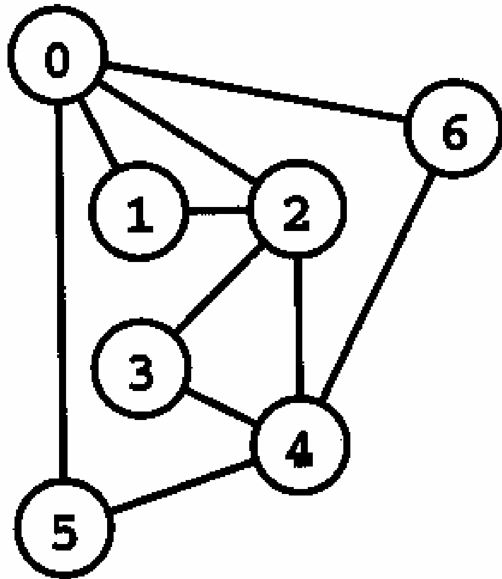

```
0-1
  1-2
    2-3
      3-4
        4-5
        4-6
    2-4
      4-3
      4-5
      4-6
0-2
  2-1
  2-3
    3-4
      4-5
      4-6
  2-4
    4-3
    4-5
    4-6
0-5
  5-4
    4-2
      2-1
      2-3
    4-3
      3-2
        2-1
    4-6
0-6
```

Euler path

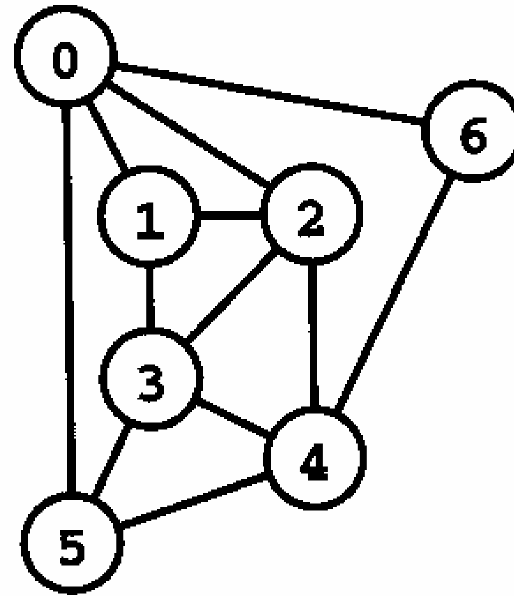
- A path in a graph is an Euler path if every edge of the graph appears as an edge in the path exactly once.
- A closed Euler path is an Euler circuit (Euler tour).
- A graph is said to be an Euler graph if it has an Euler circuit.
- There are analogous definitions in the case of digraphs.

Examples:

(1) Graph with Euler tour.



(2) Graph without Euler tour but has Euler path



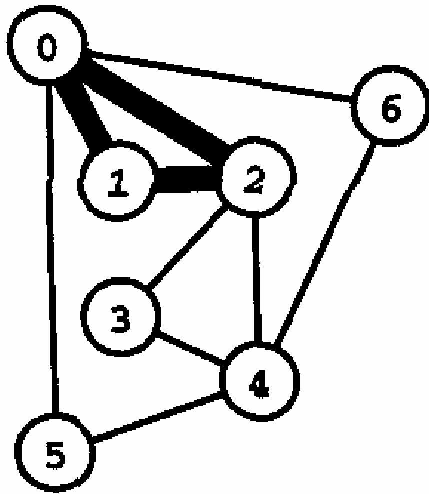
- **Property 17.4**

  A graph has a Euler tour if and only if it is connected and all its vertices are of even degree.

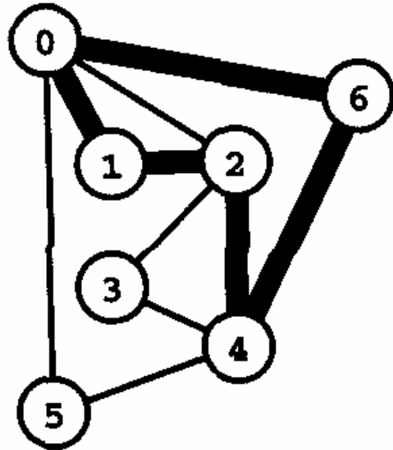
- **Corollary**

  A graph has a Euler path if and only if it is connected and exactly two of its vertices are of odd degree.

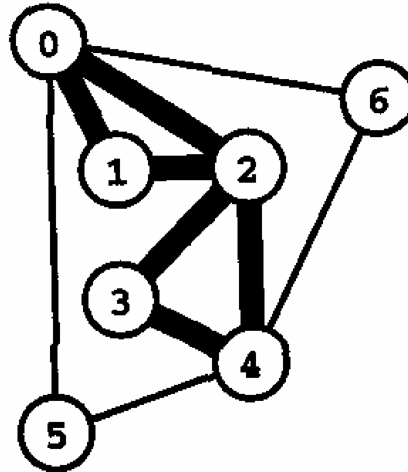
Examples: Partial tours



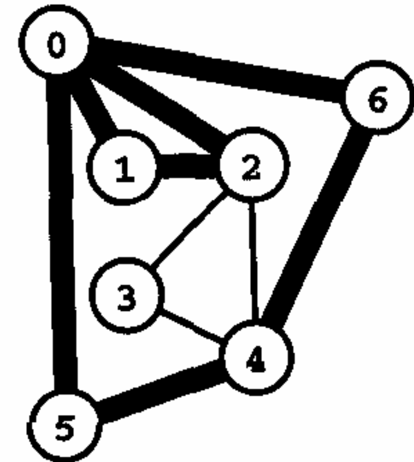
1-0-2-1



6-0-1-2-4-6



2-0-1-2-3-4-2



0-6-4-5-0-2-1-0

Euler path existence

```
int GRAPHpathE(Graph G, int v, int w)
  { int t;
   t = GRAPHdeg(G, v) + GRAPHdeg(G, w);
  if ((t % 2) != 0) return 0;
  for (t = 0; t < G->V; t++)
     if ((t != v) && (t != w))
       if ((GRAPHdeg(G, t) % 2) != 0) return 0;
  return 1;  }
```

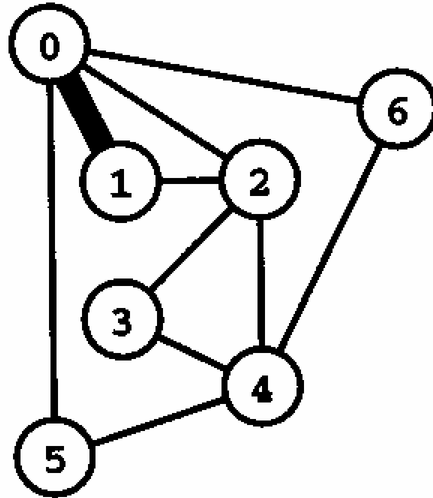
Linear-time Euler path (adjacency lists)

```
#include "STACK.h“
int path(Graph G, int v)
  { int w;
  for (; G->adj[v] != NULL; v = w)
      {        STACKpush(v);
      w = G->adj[v]->v;
      GRAPHremoveE(G, EDGE(v, w));        }
  return v;  }
```

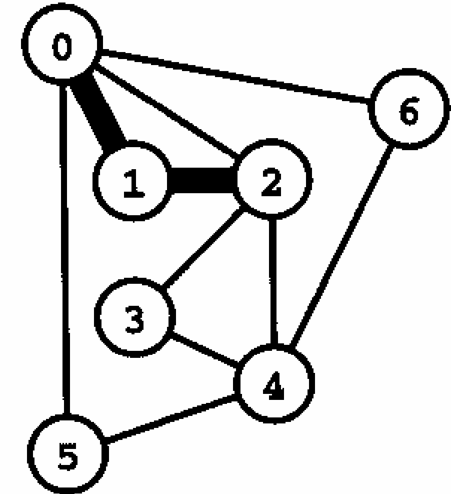
```
void pathEshow(Graph G, int v, int w)
  {
  STACKinit(G->E);
  printf("%d", w);
  while ((path(G, v) == v)&& !STACKempty())
    { v = STACKpop();
    printf("-%d", v); }
  printf("\n");  }
```

Euler tour by removing cycles

0: 2 5 6
1: 2
2: 0 3 4 1
3: 4 2
4: 6 5 3 2
5: 4 0
6: 4 0

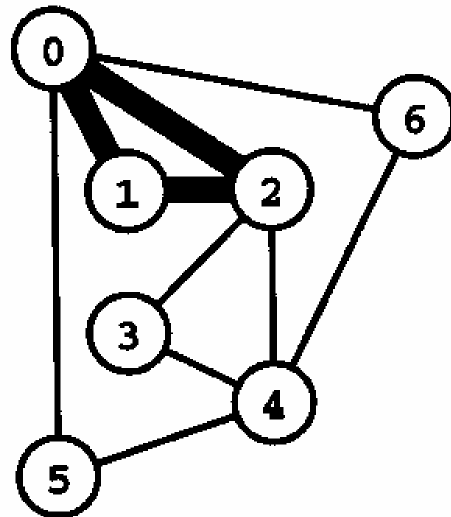


0: 2 5 6
1:
2: 0 3 4
3: 4 2
4: 6 5 3 2
5: 4 0
6: 4 0



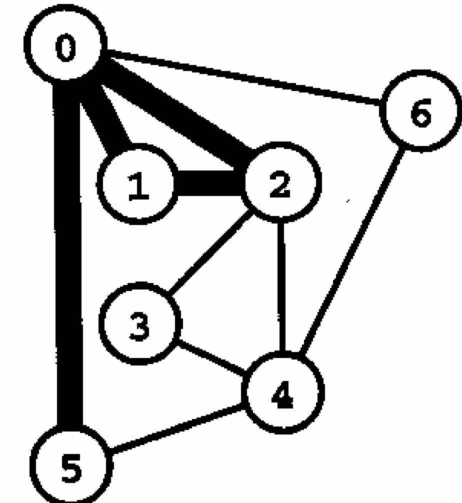
1 2

0: 5 6
1:
2: 3 4
3: 4 2
4: 6 5 3 2
5: 4 0
6: 4 0



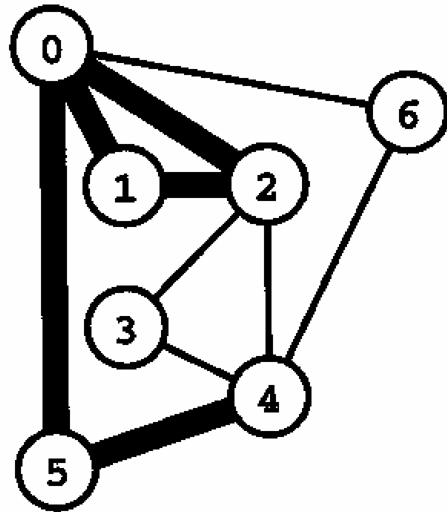
1 2 0

0: 6
1:
2: 3 4
3: 4 2
4: 6 5 3 2
5: 4
6: 4 0



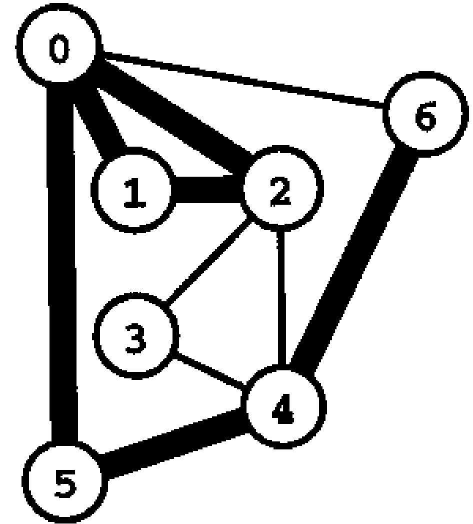
1 2 0 5

0: 6
1:
2: 3 4
3: 4 2
4: 6 3 2
5:
6: 4 0



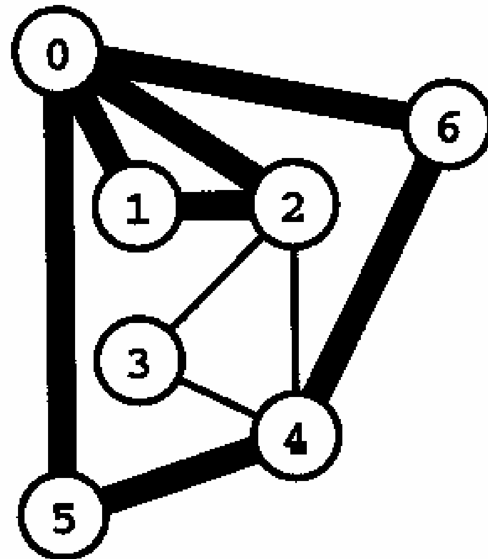
1 2 0 5 4

0: 6
1:
2: 3 4
3: 4 2
4: 3 2
5:
6: 0



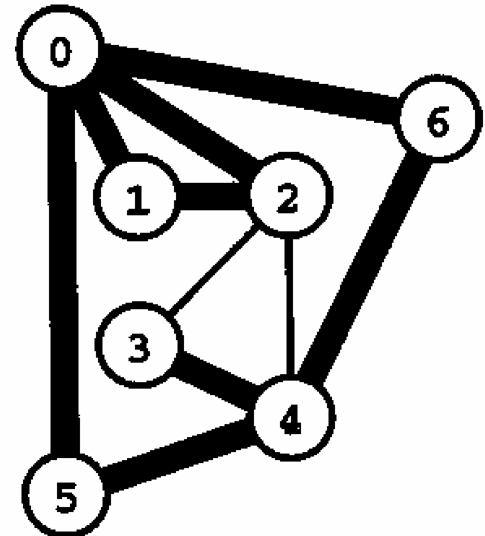
1 2 0 5 4 6

0:
1:
2: 3 4
3: 4 2
4: 3 2
5:
6:



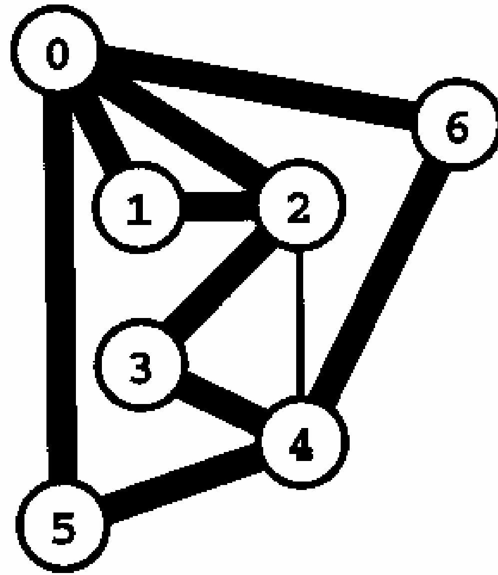
1 2 0 5 4 6 0

0:
1:
2: 3 4
3: 2
4: 2
5:
6:



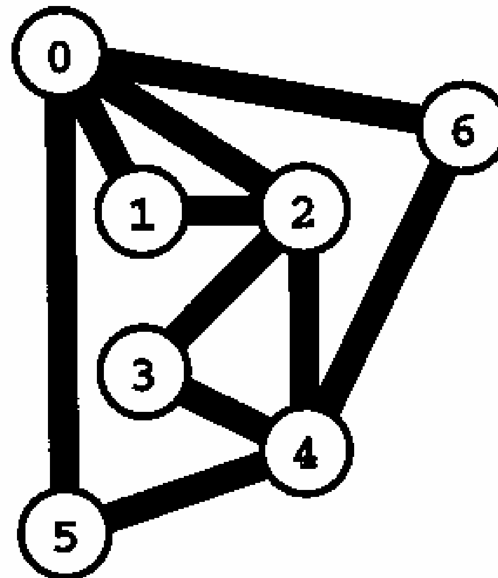
1 2 0 5 4 3

0:
1:
2: 4
3:
4: 2
5:
6:



1 2 0 5 4 3 2

0:
1:
2:
3:
4:
5:
6:



1 2 0 5 4 3 2 4

作業:

17.4, 17.5, 17.12, 17.17, 17.26, 17.31, 17.64, 17.91, 17.92, 17.107, 17.109, 17.112