Graph Algorithm

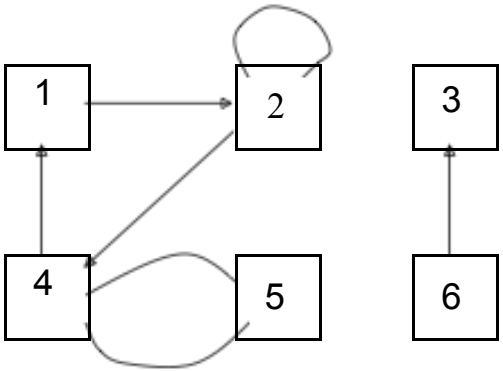
- IOI Training Oct. 2019

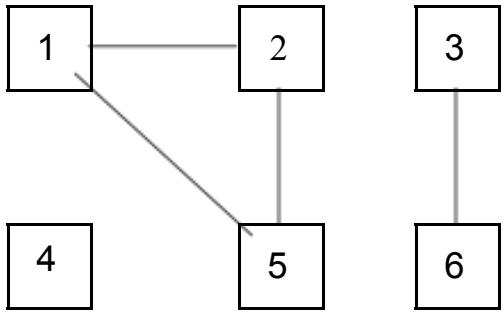
Outline

- Introduction
- Graph Data Structure
- Graph Theory
- BFS
- DFS
- Topological Sort
- Strongly Connected Component

What is a Graph?

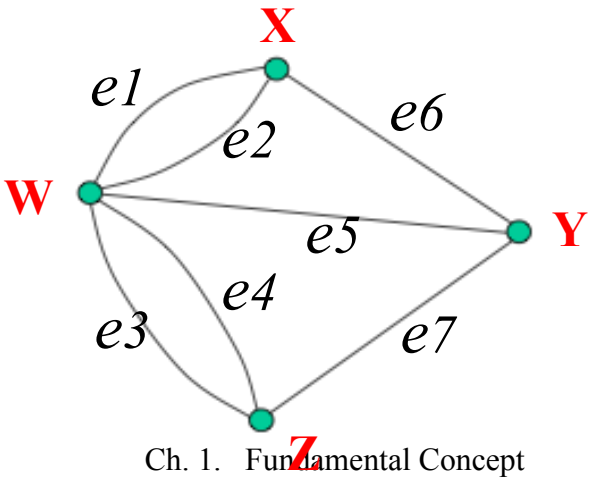
- Informally a *graph* is a set of nodes joined by a set of lines or arrows.





What Is a Graph?

- A graph G is a triple consisting of:
  - A vertex set $V(G)$
  - An edge set $E(G)$
  - A relation between an edge and a pair of vertices



Definitions

- Vertex
  - Basic Element
  - Drawn as a *node* or a *dot*.
  - V**ertex set** of *G* is usually denoted by *V*(*G*), or *V*
- Edge
  - A set of two elements
  - Drawn as a line connecting two vertices, called end vertices, or endpoints.
  - The edge set of G is usually denoted by E(G), or E.

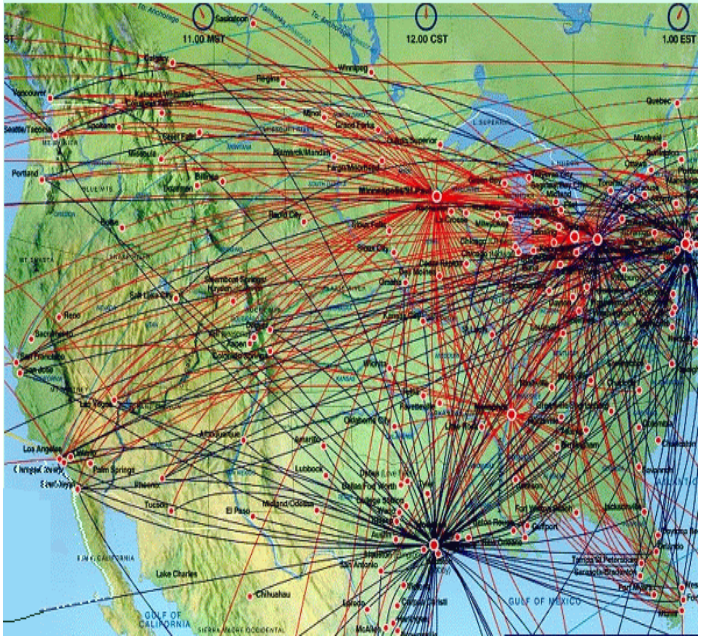
Graph-based representations

- Representing a problem as a graph can provide a different point of view
- Representing a problem as a graph can make a problem much simpler
  - More accurately, it can provide the appropriate tools for solving the problem

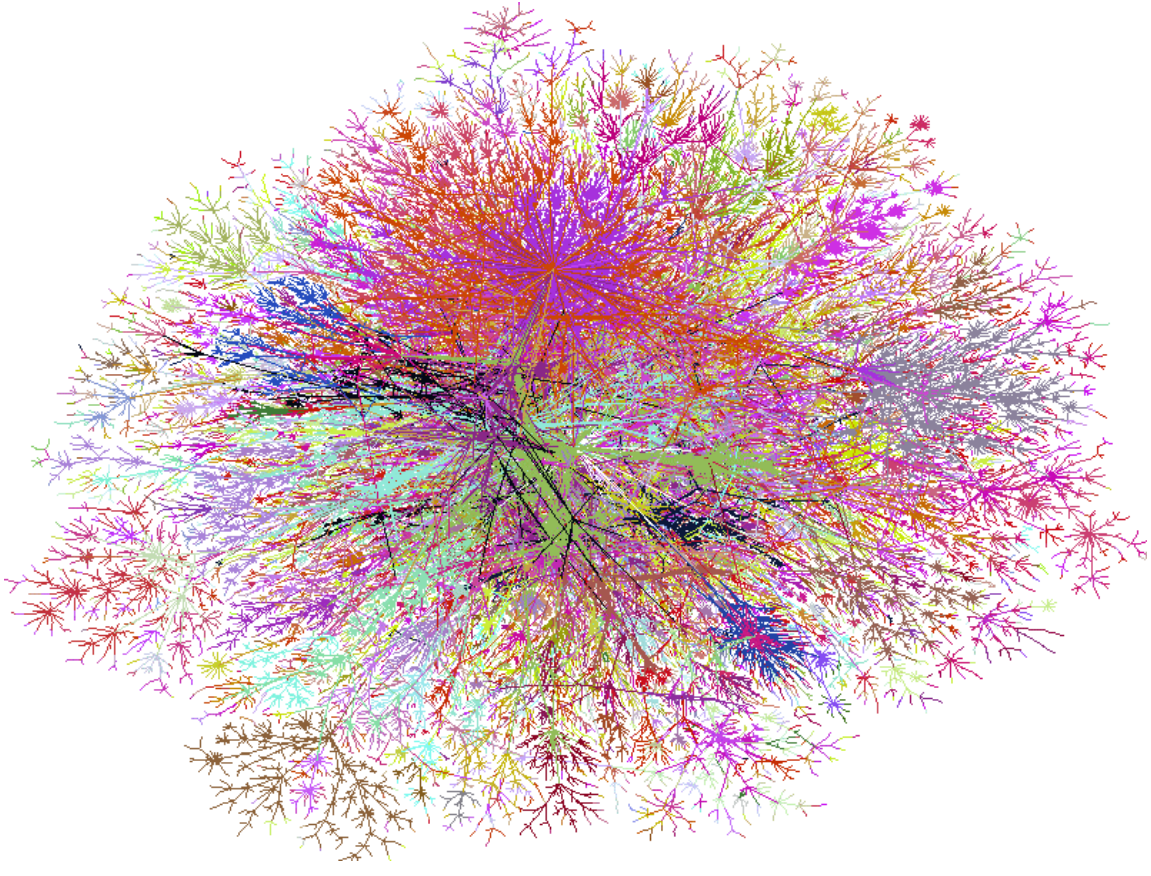
What makes a problem graph-like?

- There are two components to a graph
  - Nodes and edges
- In graph-like problems, these components have natural correspondences to problem elements
  - Entities are nodes and interactions between entities are edges
- Most complex systems are graph-like

Transportation Networks

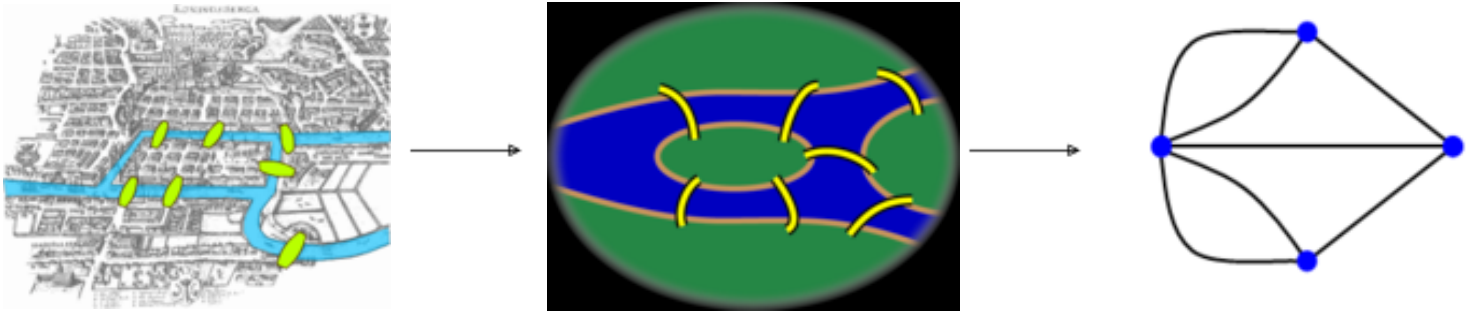


Internet



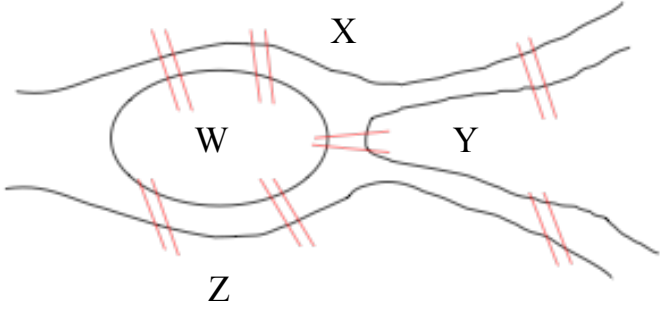
Graph Theory - History

Leonhard Euler's paper on "*Seven Bridges of Königsberg*", published in 1736.



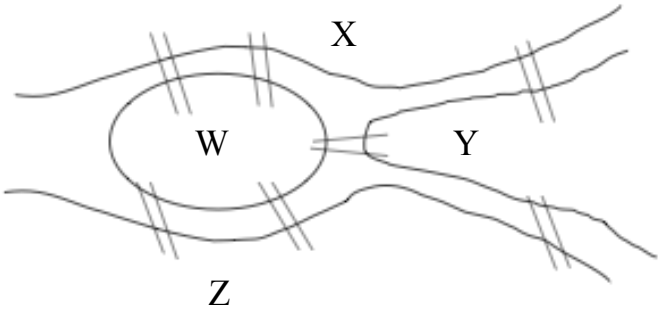
The Königsberg Bridge Problem

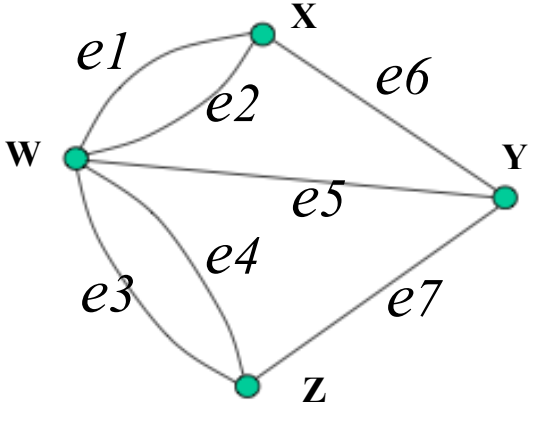
- Königsber is a city on the Pregel river in Prussia. The city occupied two islands plus areas on both banks
- Problem: Whether they could leave home, cross every bridge exactly once, and return home.



A Model

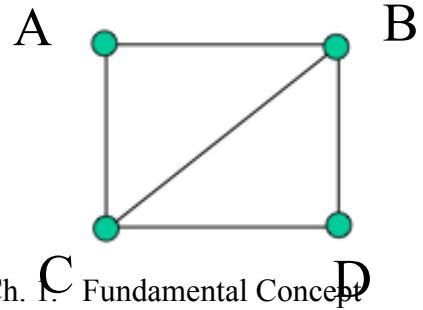
- A ***vertex*** : a region
- An ***edge*** : a path(bridge) between two regions





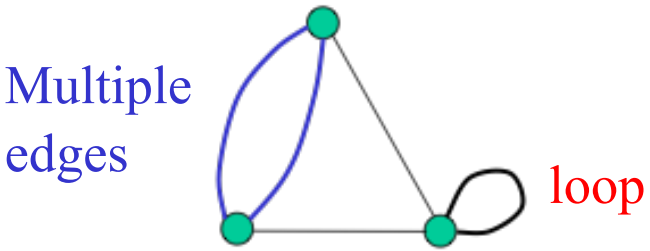
Adjacent, neighbors

- Two vertices are *adjacent* and are *neighbors* if they are the endpoints of an edge
- Example:
  - *A* and *B* are adjacent
  - *A* and *D* are not adjacent



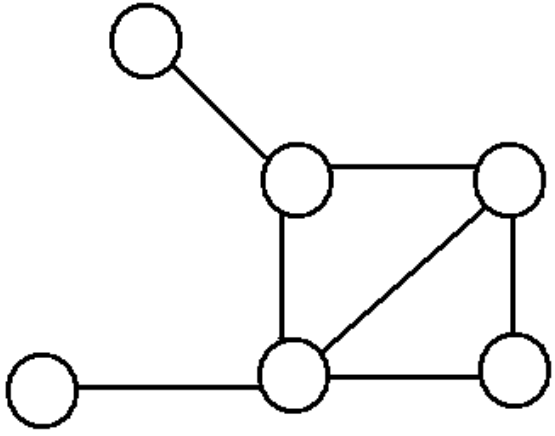
Loop, Multiple edges

- *Loop* : An edge whose endpoints are equal
- *Multiple edges* : Edges have the same pair of endpoints



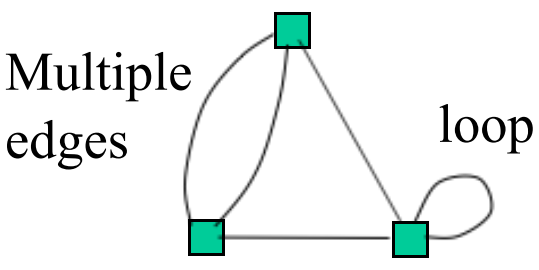
Simple Graphs

Simple graphs are graphs without multiple edges or self-loops.

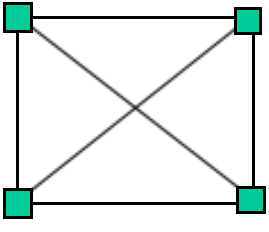


Simple Graph

- *Simple graph* : A graph has no loops or multiple edges



It is not simple.

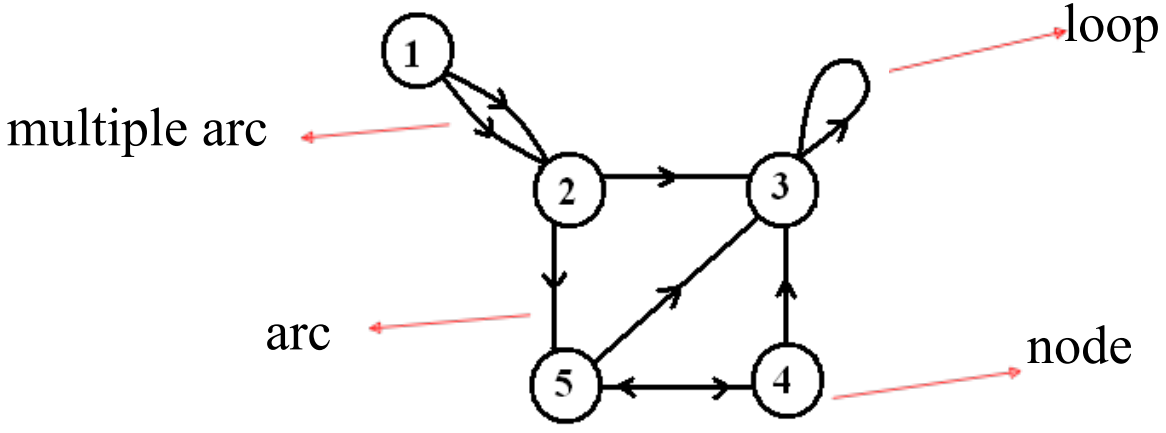


It is a simple graph.

Directed Graph (digraph)

Edges have directions

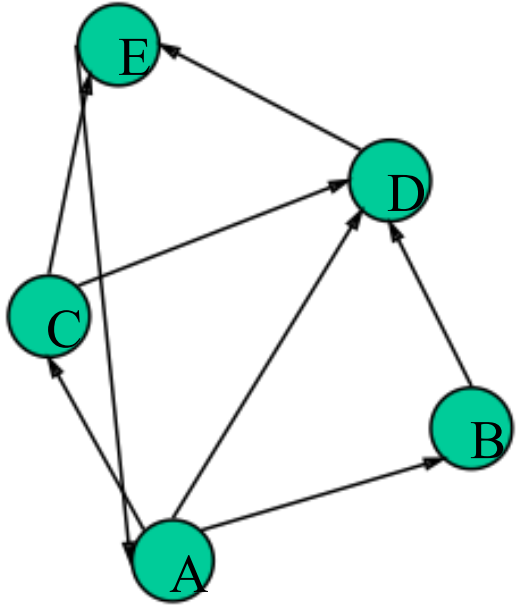
An edge is an *ordered* pair of nodes



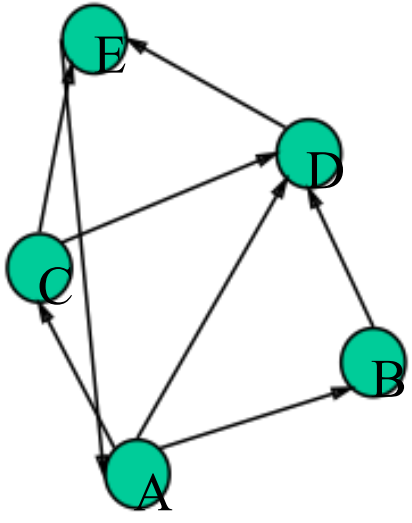
Digraphs

- Every undirected graph can be represented by a digraph.
- Applications
  - one-way streets
  - flights
  - task scheduling

Fundamental issue is reachability



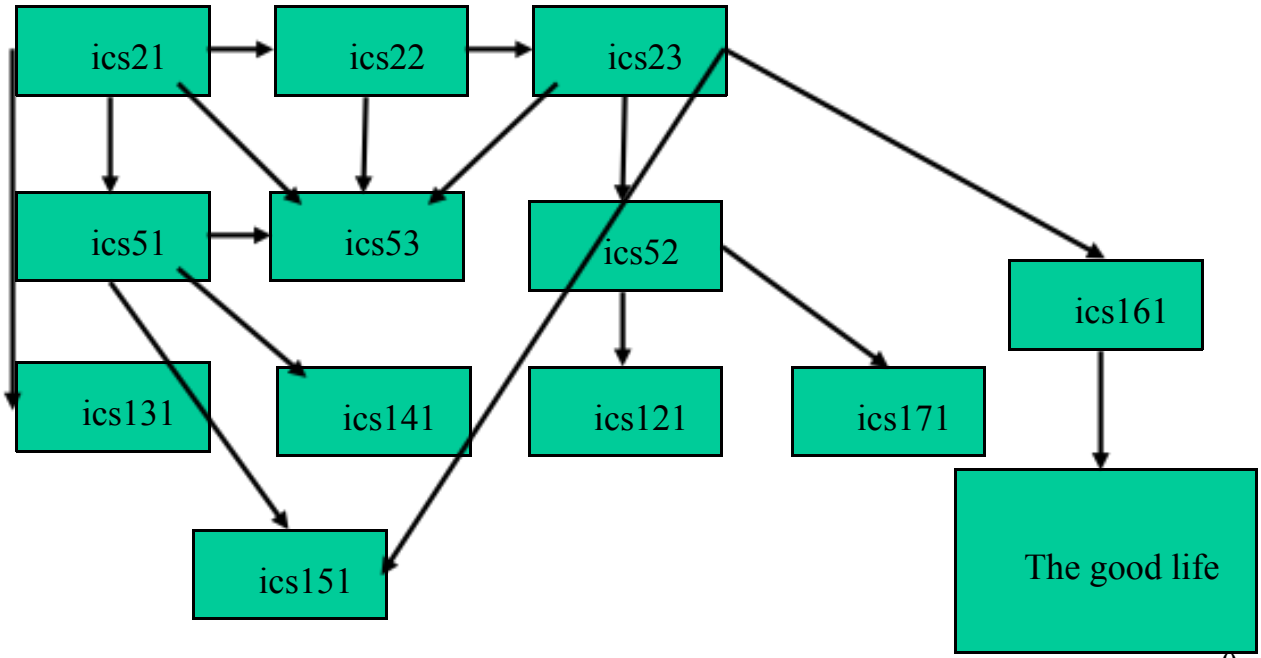
Digraph Properties



- A graph G=(V,E) such that
  - Each edge goes in one direction:
    - Edge (a,b) goes from a to b, but not b to a.
- If G is simple, $m \leq n*(n-1)$ – at most an edge between each node and every *other* node.
- If we keep in-edges and out-edges in separate adjacency lists, we can perform listing of in-edges and out-edges in time proportional to their size.

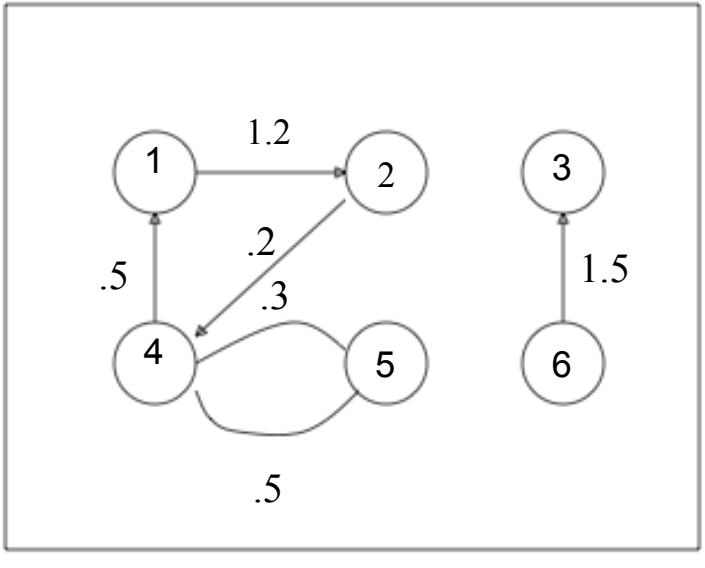
Digraph Application

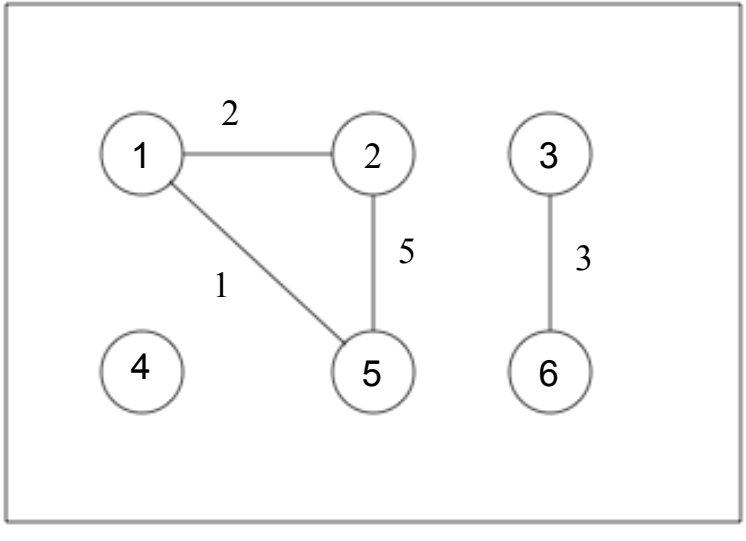
- Scheduling: edge (a,b) means task a must be completed before b can be started



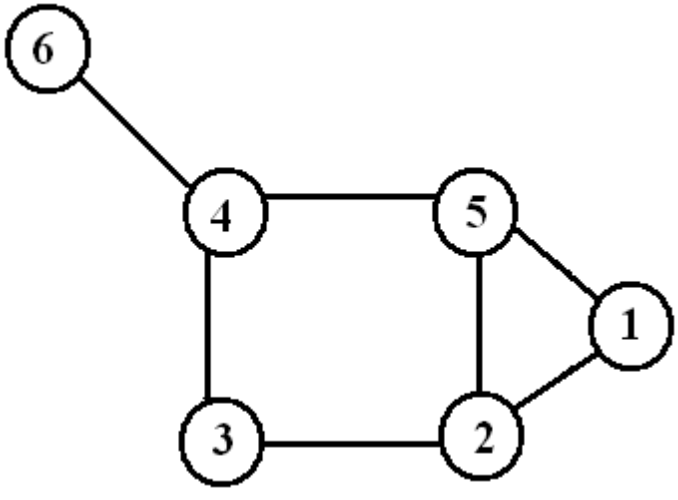
Weighted graphs

It is a graph for which each edge has an associated ***weight***, usually given by a ***weight function*** $w: E \to \mathbf{R}$.





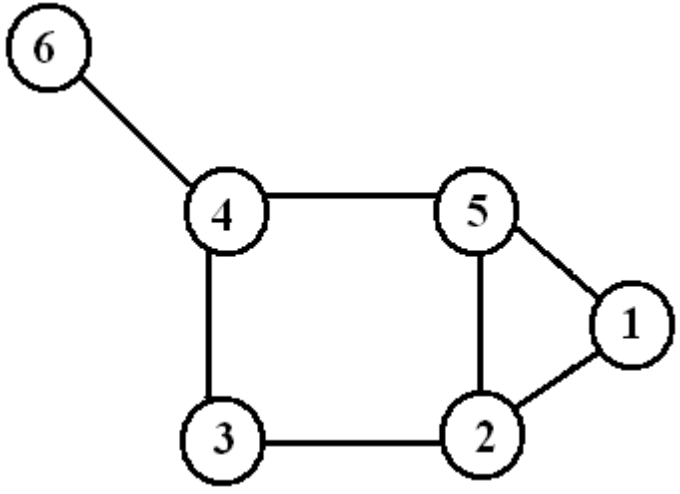
Example



- V:={1,2,3,4,5,6}
- E:={{1,2},{1,5},{2,3},{2,5},{3,4},{4,5},{4,6}}

Degree

Number of edges incident on a node

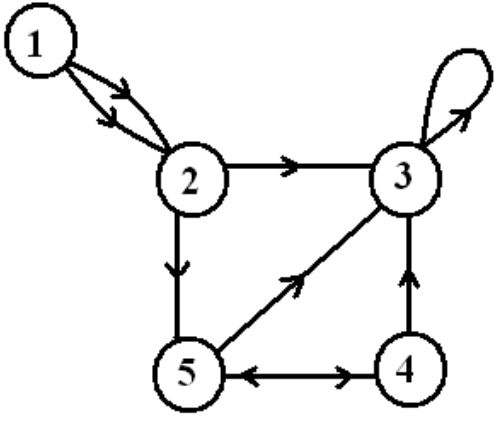


The degree of 5 is 3

Degree (Directed Graphs)

- In-degree: Number of edges entering
- Out-degree: Number of edges leaving

- Degree = indeg + outdeg



outdeg(1)=2
indeg(1)=0

outdeg(2)=2
indeg(2)=2

outdeg(3)=1
indeg(3)=4

Degree: Simple Facts

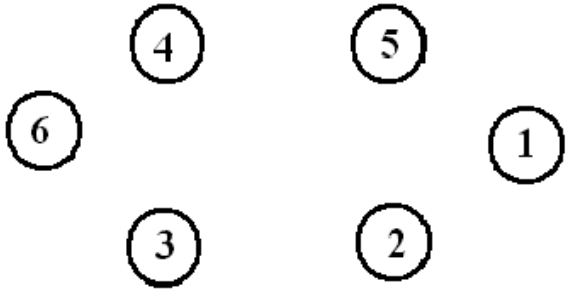
- If G is a graph with m edges, then
  $\sum \deg(v) = 2m = 2\,|E\,|$
- If G is a digraph then
  $\sum \text{indeg}(v) = \sum \text{outdeg}(v) = |E\,|$
- Number of Odd degree Nodes is even

Finite Graph, Null Graph

- *Finite graph* : an graph whose vertex set and edge set are finite
- *Null graph* : the graph whose vertex set and edges are empty

Special Types of Graphs

- **Empty Graph / Edgeless graph**
  - No edge

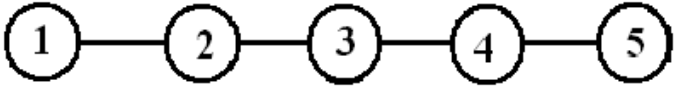


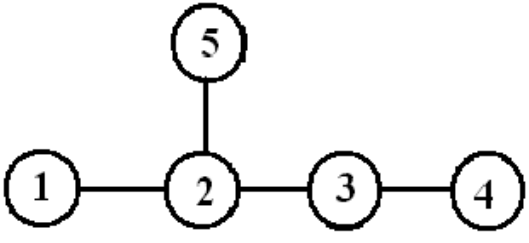
- **Null graph**
  - No nodes
  - Obviously no edge

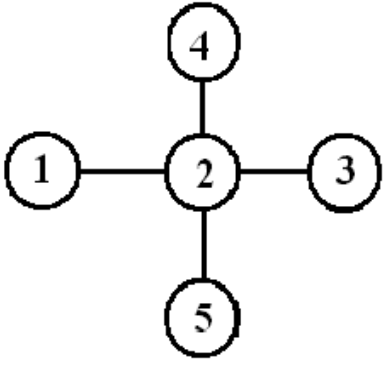
Trees

Connected Acyclic Graph

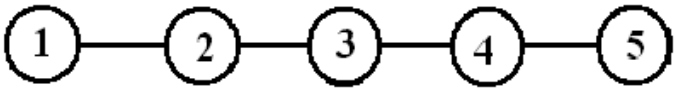
Two nodes have *exactly* one path between them



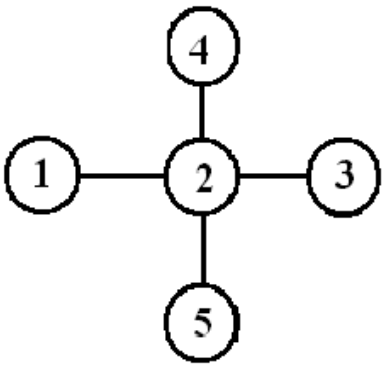




Special Trees



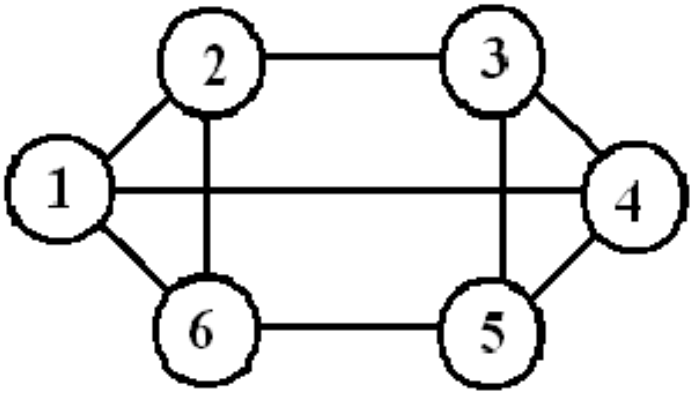
Paths



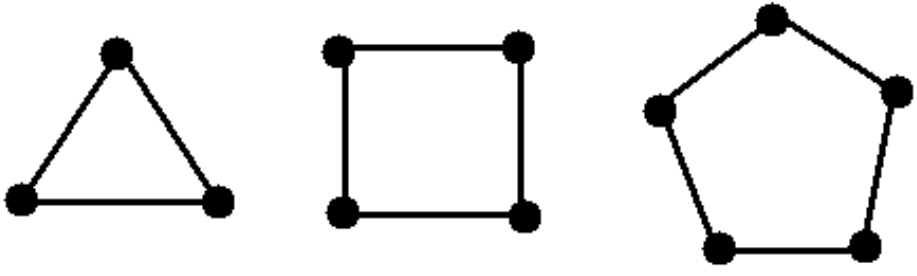
Stars

Regular

- Connected Graph
- All nodes have the same degree



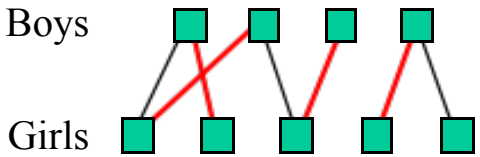
Special Regular Graphs: Cycles

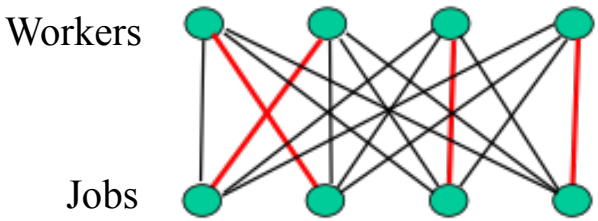


C3 C4 C5

Bipartite Graphs

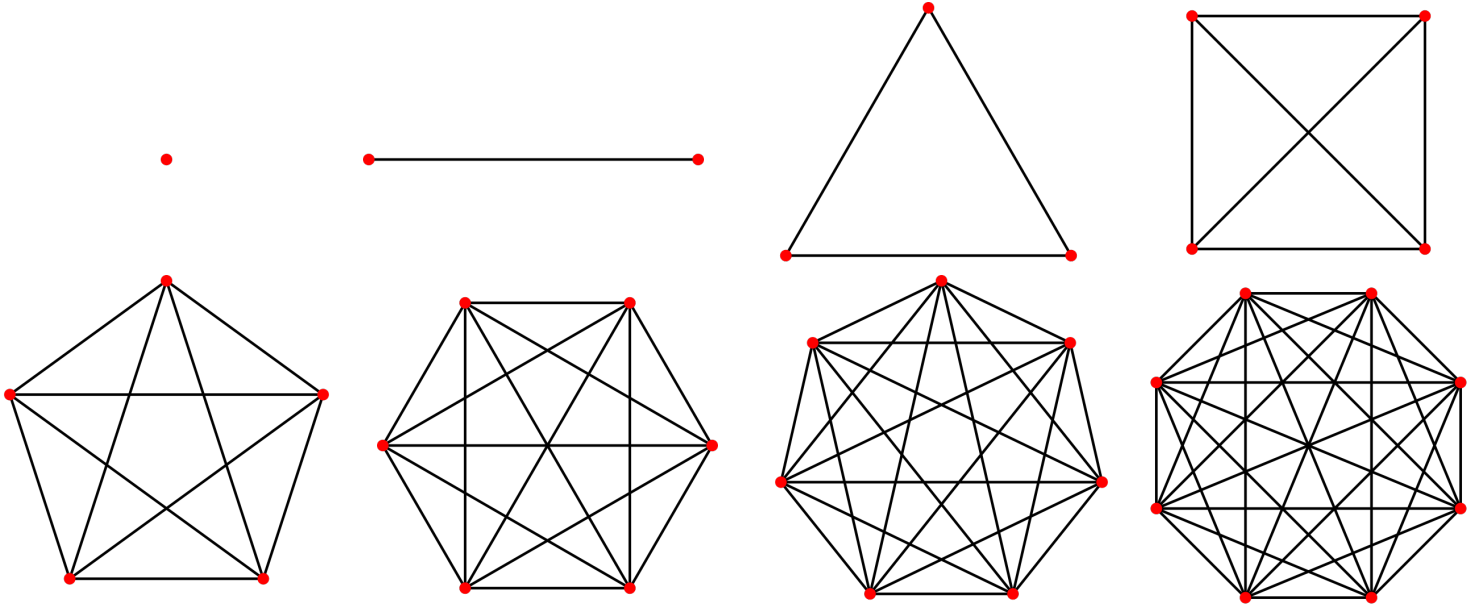
- A graph G is ***bipartite*** if $V(G)$ is the union of two disjoint independent sets called ***partite sets of G***
- Also: The vertices can be partitioned into two sets such that each set is independent
  - Matching Problem
  - Job Assignment Problem





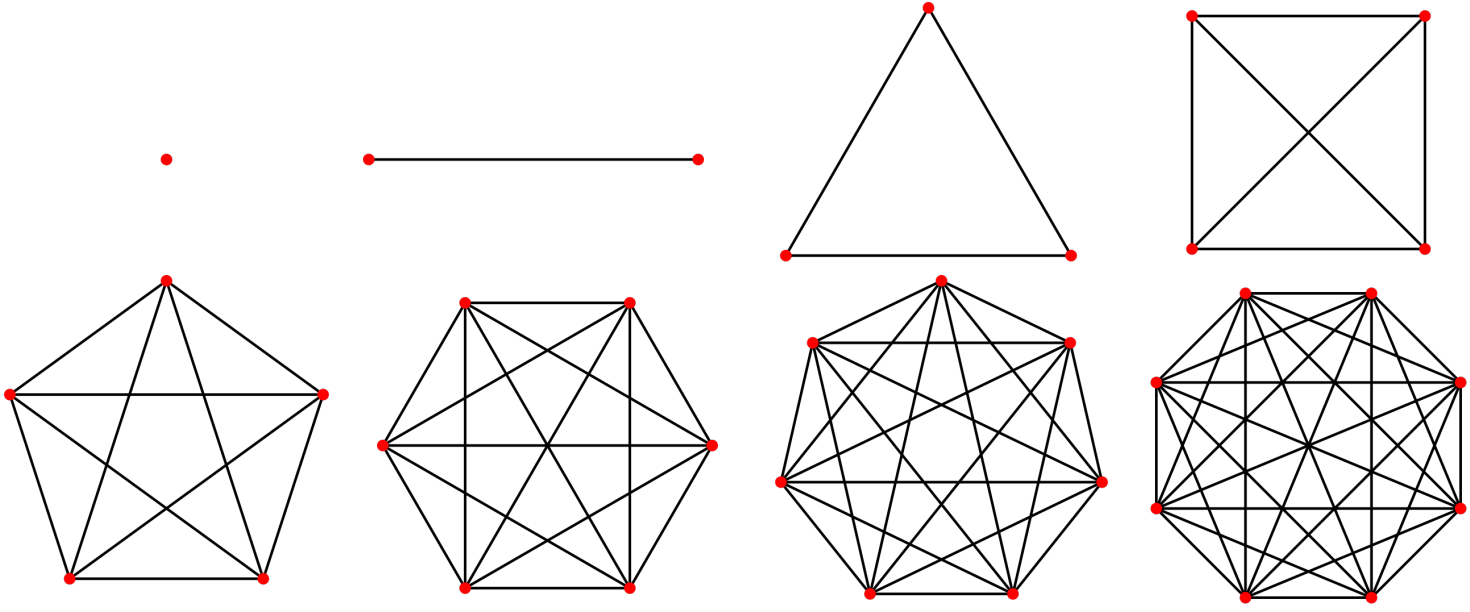
Complete Graph

- Every pair of vertices are adjacent
- For n nodes, how many edges are there?



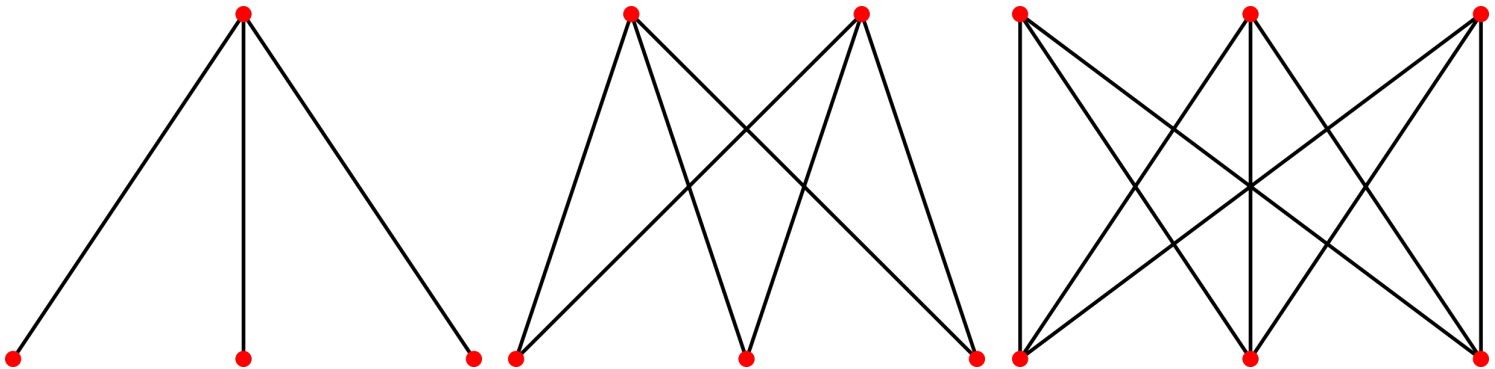
Complete Graph

- Every pair of vertices are adjacent
- Has n(n-1)/2 edges



Complete Bipartite Graph

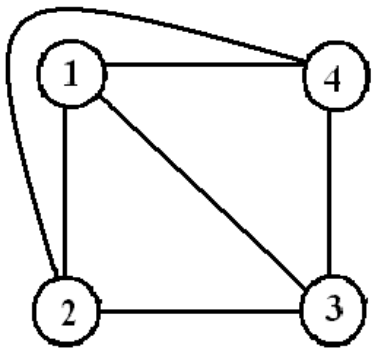
- Bipartite Variation of Complete Graph
- Every node of one set is connected to every other node on the other set

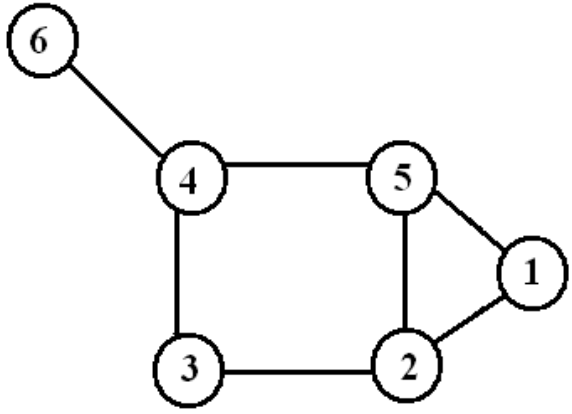


Stars

Planar Graphs

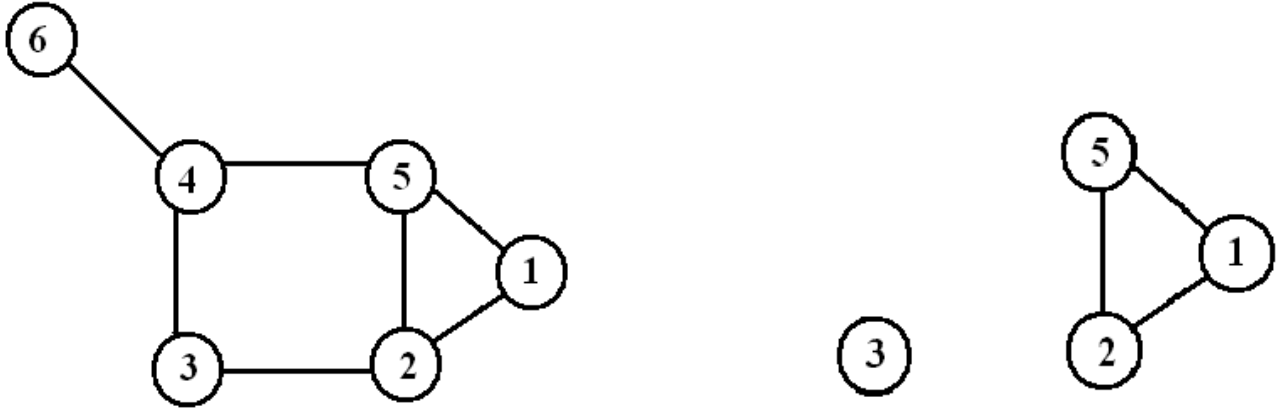
- Can be drawn on a plane such that no two edges intersect
- K4 is the largest complete graph that is planar





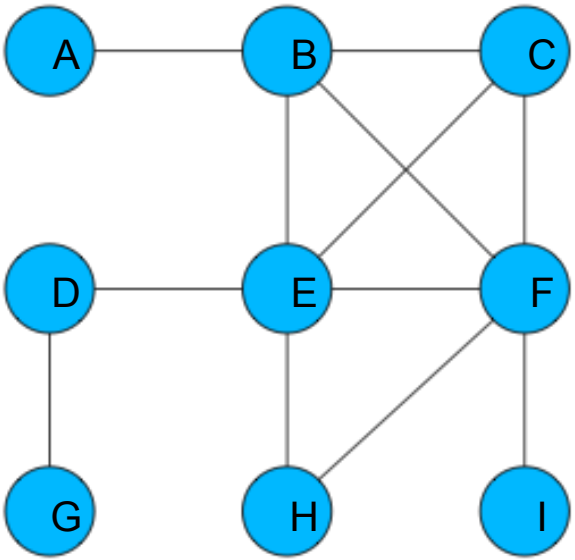
Subgraph

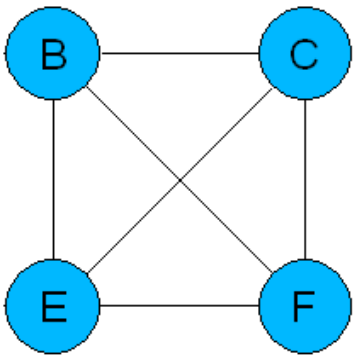
- Vertex and edge sets are subsets of those of G
  - a *supergraph* of a graph G is a graph that contains G as a subgraph.



Special Subgraphs: Cliques

A **clique** is a maximum complete connected subgraph.





GRAPH DATA STRUCTURE

Review

- Graph has two key components that we must keep:

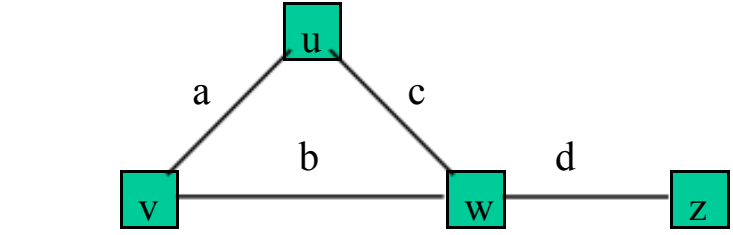
1. Node / Vertex
2. Edge
   - Can be directional
   - Can have weights or more information

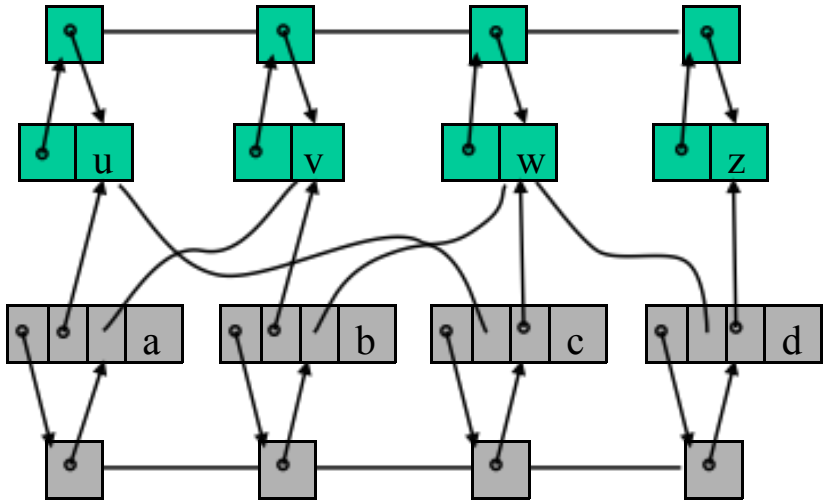
Representation (List)

- Edge List
  - pairs (ordered if directed) of vertices
  - Optionally weight and other data
- Adjacency List (node list)
- ***Adjacency-list representation***
  - an array of $|V|$ lists, one for each vertex in V.
  - For each $u \in V$, $ADJ[u]$ points to all its adjacent vertices.

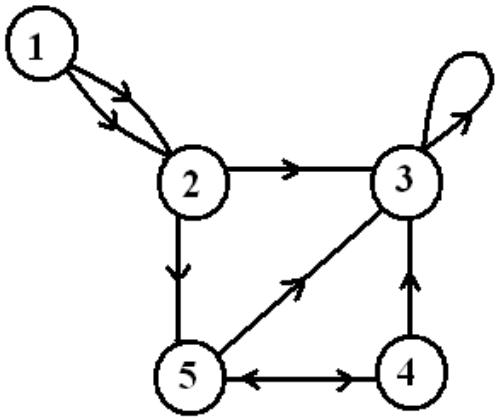
Edge List Structure

- Vertex object
  - element
  - reference to position in vertex sequence
- Edge object
  - element
  - origin vertex object
  - destination vertex object
  - reference to position in edge sequence
- Vertex sequence
  - sequence of vertex objects
- Edge sequence
  - sequence of edge objects



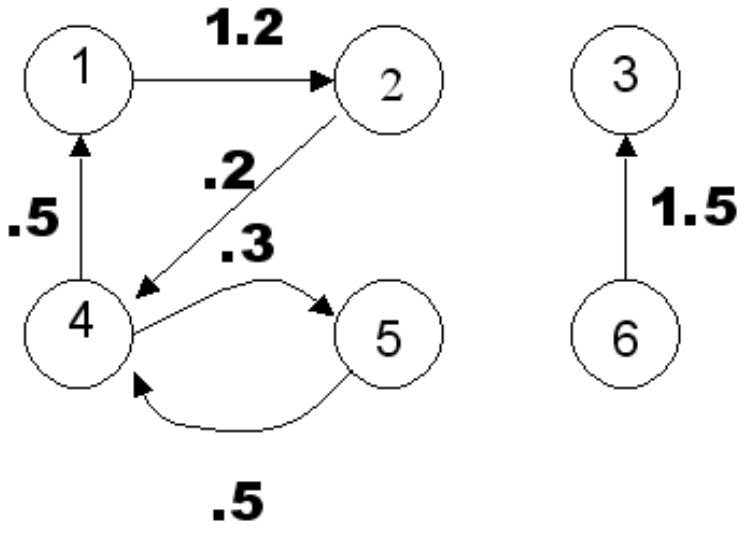


Edge and Node Lists



Edge List	Node List
1 2	1 2 2
1 2	2 3 5
2 3	3 3
2 5	4 3 5
3 3	5 3 4
4 3	
4 5	
5 3	
5 4	

Edge Lists for Weighted Graphs



Edge List

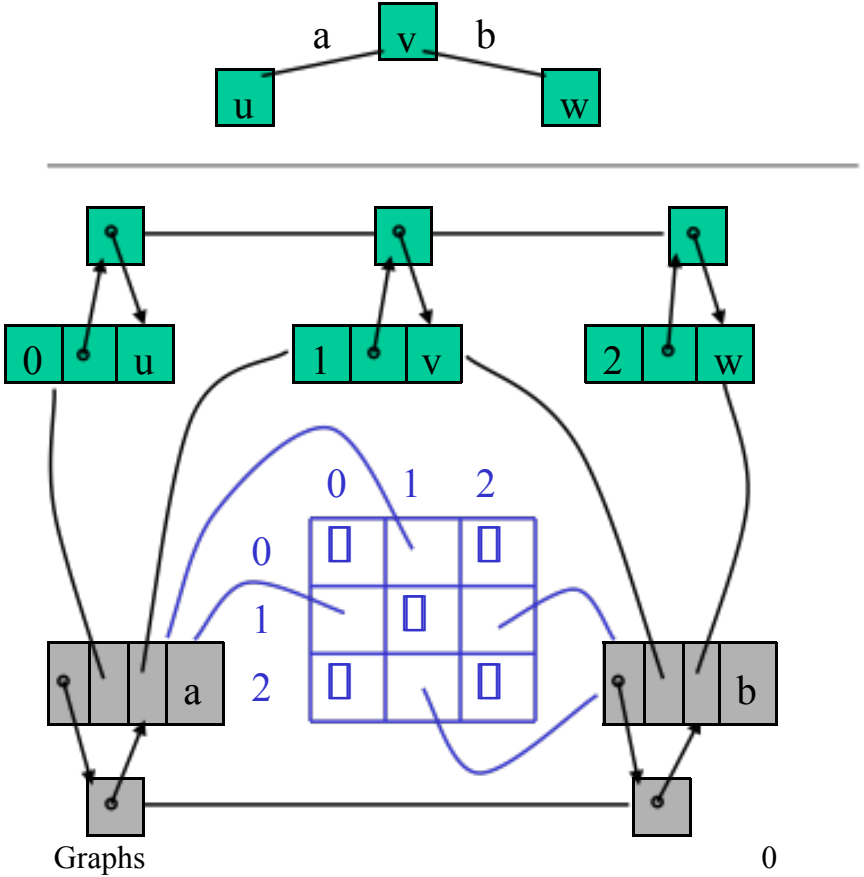
1 2 1.2
2 4 0.2
4 5 0.3
4 1 0.5
5 4 0.5
6 3 1.5

Representation (Matrix)

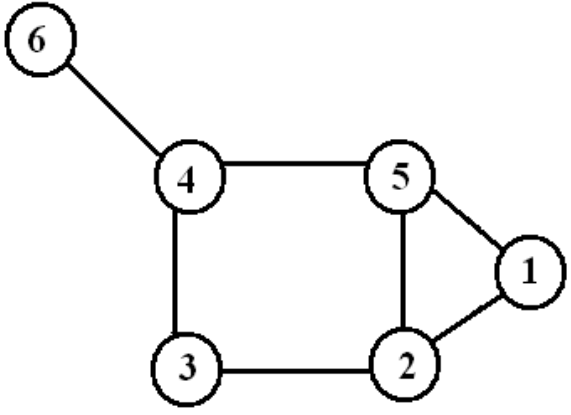
- Incidence Matrix
  - V x E
  - [vertex, edges] contains the edge's data
- Adjacency Matrix
  - V x V
  - Boolean values (adjacent or not)
  - Or Edge Weights

Adjacency Matrix Structure

- Edge list structure
- Augmented vertex objects
  - Integer key (index) associated with vertex
- 2D adjacency array
  - Reference to edge object for adjacent vertices
  - Null for non nonadjacent vertices
- The "old fashioned" version just has 0 for no edge and 1 for edge



Adjacency Matrices



	1,2	1,5	2,3	2,5	3,4	4,5	4,6
1	1	1	0	0	0	0	0
2	1	0	1	1	0	0	0
3	0	0	1	0	1	0	0
4	0	0	0	0	1	1	1
5	0	1	0	1	0	1	0
6	0	0	0	0	0	0	1

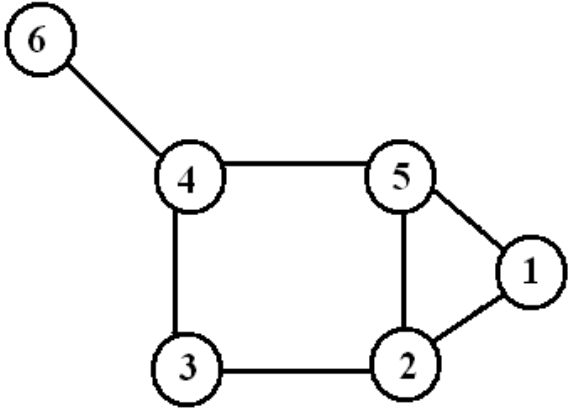
	1	2	3	4	5	6
1	0	1	0	0	1	0
2	1	0	1	0	1	0
3	0	1	0	1	0	0
4	0	0	1	0	1	1
5	1	1	0	1	0	0
6	0	0	0	1	0	0

Topological Distance

- A shortest path is the minimum path connecting two nodes.
- The number of edges in the shortest path connecting *p* and *q* is the ***topological distance*** between these two nodes, dp,q

Distance Matrix

$|V| \times |V|$ matrix $D \equiv (dij)$ such that
dij is the topological distance between *i* and *j*.



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

เรื่องที่ต้องคิดในทางปฏิบัติ

- เราเห็นการอธิบายกราฟด้วย Edge listing กับ Adjacency list มาแล้ว
  - แต่คำถามก็คือว่าเราจะเก็บข้อมูลพวกนี้ในหน่วยความจำอย่างไร (How to keep graph description in memory?)
  - แล้วเราจะทำอะไรกับข้อมูลพวกนี้บ้าง (What operations are we going to do with a graph?)
- ธรรมชาติของโครงสร้างข้อมูลที่ดีก็คือว่า มันต้องสามารถทำในสิ่งที่เราต้องการทำบ่อย ๆ ได้อย่างมีประสิทธิภาพ
  - เรามักถามว่าโหนดหมายเลข x มีเส้นเชื่อมต่อกับโหนดหมายเลข y หรือไม่
  - เรามักถามว่ามีเส้นทางจาก โหนดหมายเลข x ไปโหนดหมายเลข y หรือไม่

Ref: Aj. Pinyo Taeprasartsit, Silapakorn

เปรียบเทียบการใช้ Edge listing กับ Adjacency list

- ถ้าเราถามว่าโหนดหมายเลข x มีเส้นเชื่อมต่อกับโหนดหมายเลข y หรือไม่

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

  - Edge listing: เราต้องวิ่งไปทั้งลิสต์เพื่อหาว่ามีคู่เส้นเชื่อม (x, y) หรือ (y, x) หรือไม่
    - จากตัวอย่าง เรามีเส้นเชื่อมทั้งหมด 7 เส้น ถ้าถามแบบนี้ 100 รอบ ในกรณีที่เลวร้ายที่สุดก็ต้องวิ่งหาเส้นเชื่อมทั้งหมด 700 ครั้ง
  - Adjacency list: เราสร้างลิสต์แยกของแต่ละโหนดออกมา แล้วตรวจเฉพาะลิสต์ของโหนดที่สนใจ เช่นถ้า x = 1 และ y = 2
    - เราก็ตรวจเฉพาะลิสต์ของโหนด 1 หรือโหนด 2 อย่างใดอย่างหนึ่ง
    - ถ้าเราเลือกตรวจลิสต์โหนด 1 เราก็จะใช้เวลาตรวจรอบละ 3 ครั้ง ถ้าต้องตรวจ 100 รอบ ก็เสียเวลาแค่ 300 ครั้ง
- Adjacency list เร็วกว่ามาก ดังนั้นเรามาดูเรื่องวิธีเก็บในหน่วยความจำ

Ref: Aj. Pinyo Taeprasartsit, Silapakorn

การเก็บโครงสร้างกราฟในหน่วยความจำ

- อย่างที่บอกไว้ตั้งแต่ต้นว่ากราฟคือโครงสร้างข้อมูล

  → ถ้าเราไม่รู้ว่าจะเก็บโครงสร้างข้อมูลไว้ได้อย่างไรก็ไม่มีความหมาย

- สมมติว่าเรารู้จำนวนโหนดและเส้นเชื่อมของแต่ละโหนดว่าเป็นจำนวนที่แน่นอนตายตัวในกราฟที่เราสนใจ

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

- ดังนั้นโหนดแต่ละโหนดจะมีอาเรย์ของเลขจำนวนเต็มเพื่อเก็บว่ามันเชื่อมต่อกับใครอยู่บ้าง
- เราสามารถนำอาเรย์ของแต่ละโหนดมามัดรวมกันเป็นอาเรย์สองมิติ
  - มิติแรกระบุว่าเป็นลิสต์ของโหนดไหน
  - มิติที่สองระบุว่าโหนดที่สนใจเชื่อมกับใคร
  - ขนาดของมิติที่สองของแต่ละโหนดต่างกันได้ (แปลกใจมั้ย)

Ref: Aj. Pinyo Taeprasartsit, Silapakorn

ตัวอย่างการเก็บ Adjacency list

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

```
public class GraphStructDemo {
  int[][] arNode;

  private void prepareSpace() {
    arNode = new int[9][];

    arNode[0] = new int[1];
    arNode[1] = new int[3];
    arNode[2] = new int[2];
    arNode[3] = new int[3];
    arNode[4] = new int[1];

    arNode[5] = new int[0];
    arNode[6] = new int[2];
    arNode[7] = new int[1];
    arNode[8] = new int[1];
  }
```

ประกาศอาเรย์สำหรับเก็บข้อมูลการเชื่อมต่อไว้

เริ่มสร้างอาเรย์ แต่อย่าเพิ่งรีบบอกขนาดของมิติที่สอง เก็บไว้ทำทีหลังได้ (ทำแบบนี้ได้จริง ๆ นะ)

ระบุขนาดของลิสต์การเชื่อมต่อของแต่ละโหนดทีละอันตามปริมาณที่ต้องใช้จริง

ป้อนข้อมูลเข้า Adjacency list

```
private void insertData() {
    arNode[0][0] = 1;
    arNode[1][0] = 0;  arNode[1][1] = 2;  arNode[1][2] = 3;
    arNode[2][0] = 1;  arNode[2][1] = 3;
    arNode[3][0] = 1;  arNode[3][1] = 2;  arNode[3][2] = 4;
    arNode[4][0] = 3;

    //arNode[5][];  // No edge to insert
    arNode[6][0] = 7;  arNode[6][1] = 8;
    arNode[7][0] = 6;
    arNode[8][0] = 6;
}
```

ตรวจว่าโหนดเชื่อมกันหรือไม่

การตรวจดูว่าโหนด x เชื่อมกับโหนด y หรือไม่

```
public boolean isLinked(int x, int y) {
  int numNodes = arNode[x].length;

  for(int i = 0; i < numNodes; ++i) {
    if(arNode[x][i] == y) {
       return true;  // y is found
    }
  }

  return false;      // y not found
}
```

เนื่องจากอาเรย์ของแต่ละโหนดมีความยาวไม่เท่ากัน เราจึงต้องหาความยาวลิสต์ของโหนดมาเก็บไว้ก่อน

จากโครงสร้างของลูป เราเห็นได้ว่าการจะตรวจการเชื่อมต่อของโหนด ในกรณีที่แย่ที่สุดเกิดขึ้นเมื่อไม่พบการเชื่อมต่อ (return false) เพราะเราจะต้องหาตั้งแต่เริ่มจนจบ

มีวิธีที่ทำให้ฟังก์ชันนี้ทำงานเร็วขึ้นหรือไม่ ?

Ref: Aj. Pinyo Taeprasartsit, Silapakorn

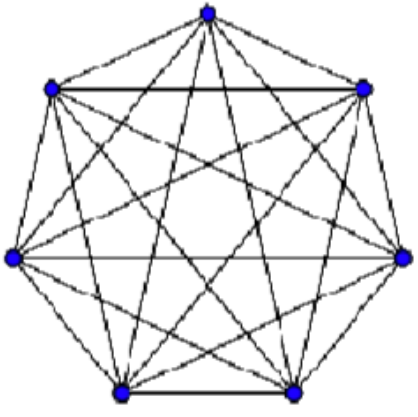
ทำอย่างไรจึงจะหาคำตอบเกี่ยวกับการเชื่อมต่อได้เร็ว ๆ

- จากฟังก์ชัน isLinked ลูปเป็นบริเวณที่ใช้เวลาในการค้นหานานที่สุด
- แต่เราก็รู้มาก่อนแล้วว่าวิธีที่ใช้ในการค้นหาที่เร็วนั้นมีอยู่
  - เช่น การใช้ binary search จะย่นเวลาในการหาข้อมูลในแต่ละลิสต์เหลือเพียง $O(\log M)$ เมื่อ M คือจำนวนโหนดในลิสต์ที่เราสนใจ
  - ดังนั้นถ้าเราจัดเรียงข้อมูลในลิสต์แต่ละอันก่อน เราก็จะสามารถใช้ binary search เพื่อทำให้การหาคำตอบเกี่ยวกับการเชื่อมต่อเป็นไปอย่างรวดเร็ว
- แล้วที่จริงยังมีทางทำให้เร็วกว่า $O(\log M)$ หรือไม่
  - มีเหมือนกัน แต่เราต้องเลิกใช้ Adjacency list แล้วไปใช้ของอย่างอื่นแทน

Ref: Aj. Pinyo Taeprasartsit, Silapakorn

หน่วยความจำที่ต้องใช้ในการอธิบายโครงสร้างกราฟ

- การใช้ Edge listing หรือ Adjacency list เหมาะกับกราฟที่มีเส้นเชื่อมน้อยเมื่อเทียบกับจำนวนโหนด (เหมาะกับ Sparse Graph)
- ลองพิจารณากราฟที่มีเส้นเชื่อมสมบูรณ์ (Complete Graph)



กราฟมี 7 โหนด 21 เส้นเชื่อม

Edge listing ใช้พื้นที่เก็บเลขจำนวนเต็ม 2 ตัวต่อเส้นเชื่อม

→ ใช้ integer 42 ตัว (168 bytes)

→ ถ้าเส้นเชื่อมมาก ปริมาณหน่วยความจำที่ต้องใช้จะเพิ่มขึ้นมากตาม

→ กราฟจำนวนมากมีปริมาณเส้นเชื่อมเป็น $O(|V|^2)$

[V คือเซ็ตของโหนด $|V|$ ก็คือจำนวนโหนดในเซ็ต]

ดังนั้นปริมาณเส้นเชื่อมและหน่วยความจำที่ต้องใช้จึงอยู่ในระดับ $O(|V|^2)$ ไปด้วย

Ref: Aj. Pinyo Taeprasartsit, Silapakorn

หน่วยความจำที่ต้องใช้ในการอธิบายโครงสร้างกราฟ (2)

- แล้วถ้าเป็น Adjacency matrix ล่ะ
  - เนื่องจากเป็นอาเรย์สองมิติ เราเห็นได้ชัดว่าใช้หน่วยความจำ $O(|V|^2)$
  - แต่ว่าเราไม่จำเป็นต้องเก็บเลขจำนวนเต็มเหมือนกับการใช้ลิสต์
  - ถ้าเราใช้ boolean เราจะเสียพื้นที่ต่อช่อง 1 ไบต์ (ถ้าเป็นจำนวนเต็มแบบลิสต์จะใช้ 4 ไบต์)
  - ถ้าเราใช้ **java.util.BitSet** เราจะใช้พื้นที่ต่อช่องแค่ 1 บิต
- ด้วยเทคนิคด้านการจัดการหน่วยความจำ ถ้ากราฟมีเส้นเชื่อมมาก ๆ การใช้ Adjacency matrix เป็นทางออกที่ดีที่สุดอย่างไม่ต้องสงสัย
  - ตอบคำถามเรื่องการเชื่อมต่อได้เร็วกว่า เข้าใจง่ายกว่า
  - ถ้าเส้นเชื่อมมากพอ มักจะใช้หน่วยความจำน้อยกว่าด้วย

Ref: Aj. Pinyo Taeprasartsit, Silapakorn

Asymptotic Performance

• n vertices, m edges • no parallel edges • no self-loops • Bounds are "big-Oh"	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$	$n + m$	n^2
incidentEdges(v)	m	deg(v)	n
areAdjacent (v, w)	m	min(deg(v), deg(w))	1
insertVertex(o)	1	1	n^2
insertEdge(v, w, o)	1	1	1
removeVertex(v)	m	deg(v)	n^2
removeEdge(e)	1	1	1

BASIC GRAPH THEORY

Walks

A ***walk of length k*** in a graph is a succession of k (not necessarily different) edges of the form

uv,vw,wx,…,yz.

This walk is denote by uvwx…xz, and is referred to as a ***walk between u and z***.

A walk is ***closed*** is u=z.

Walks VS. Trails

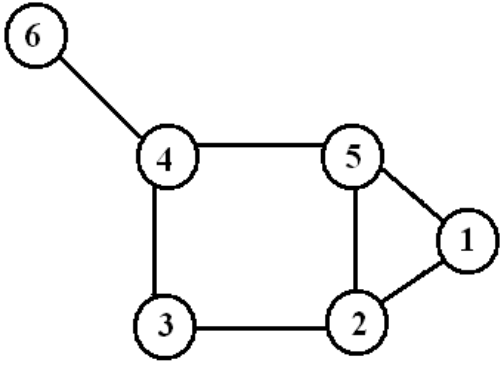
- A ***walk*** **:** a list of vertices and edges *v0*, *e1*, *v1*, ..., *ek*, *vk* such that, for $1 \leq i \leq k$, the edge *ei* has endpoints *vi-1* and *vi* .
- A ***trail*** : a walk with no repeated edge.

Circuit

- A circuit is a closed trail (A trail whose starting and ending nodes are the same).

Path

- A *path* is a walk in which all the edges and all the nodes are different.
- Note: The length of a walk, trail, path, or cycle is its number of edges.

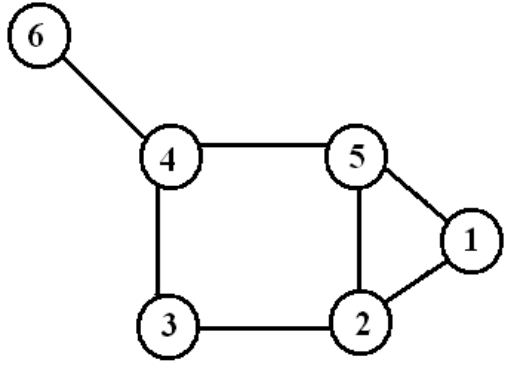


Walks and Paths

1,2,5,2,3,4	1,2,5,2,3,2,1	1,2,3,4,6
walk of length 5	CW of length 6	path of length 4

Cycle

- A ***cycle*** is a closed walk in which all the edges are different.

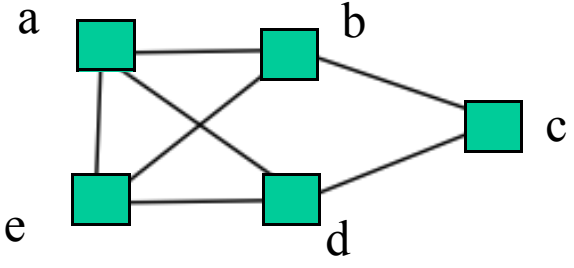


1,2,5,1 2,3,4,5,2

3-cycle 4-cycle

Checkpoint: Path and Cycle

- ***Path*** : a sequence of distinct vertices such that two consecutive vertices are adjacent
  - Example: **(*a, d, c, b, e*)** is a path
  - **(*a, b, e, d, c, b, e, d*)** is not a path; it is a walk
- ***Cycle*** : a closed Path
  - Example: **(*a, d, c, b, e, a*)** is a cycle



Lemma: Every u,v-walk contains a u,v-path 1.2.5

Proof:

- Use induction on the length of a u, v-walk W.
- Basis step: $l = 0$.
  - Having no edge, W consists of a single vertex ($u=v$).
  - This vertex is a u,v-path of length 0.

to be continued

Lemma: Every u,v-walk contains a u,v-path 1.2.5

Proof: Continue

- Induction step : $l \geq 1$.

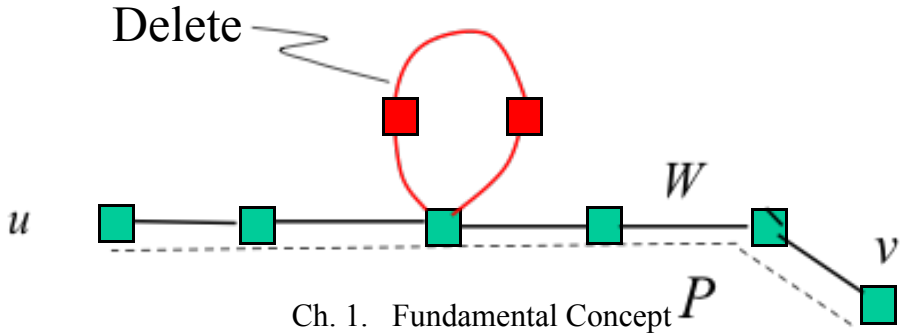
– Suppose that the claim holds for walks of length less than l.

– If W has no repeated vertex, then its vertices and edges form a u,v-path.

Lemma: Every u,v-walk contains a u,v-path 1.2.5

Proof: Continue

- Induction step : $l \geq 1$. Continue
  - If W has a repeated vertex w, then deleting the edges and vertices between appearances of w (leaving one copy of w) yields a shorter u,v-walk W' contained in W.
  - By the induction hypothesis, W' contains a u,v-path P,and this path P is contained in W.



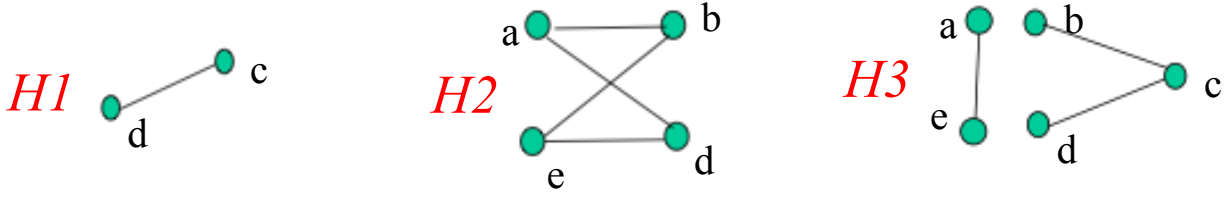
Connectivity

- a graph is ***connected*** if
  - you can get from any node to any other by following a sequence of edges OR
  - any two nodes are connected by a path.

- A directed graph is ***strongly connected*** if there is a directed path from any node to any other node.

Connected and Disconnected

- ***Connected*** : There exists at least one path between two vertices
- ***Disconnected*** : Otherwise
- Example:
  - *H1* and *H2* are connected
  - *H3* is disconnected

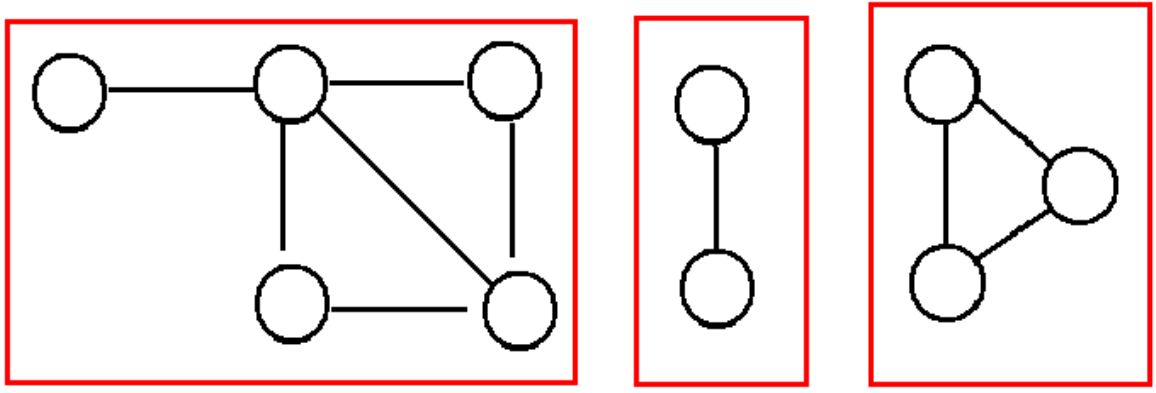


Theorem: *There is a simple path between every pair of distinct vertices of a connected undirected graph.*

Proof Idea: Let u and v be two distinct vertices of the connected undirected graph G = (V, E). Since G is connected, there is at least one path between u and v.

Component

- Every disconnected graph can be split up into a number of connected ***components***.



Theorem: Every graph with n vertices and k edges has at least n-k components 1.2.11

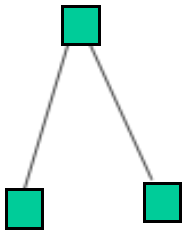
Proof:

- An n-vertex graph with no edges has n components
- Each edge added reduces this by at most 1
- If k edges are added, then the number of components is at least $n - k$

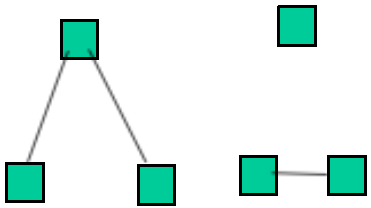
Theorem: Every graph with n vertices and k edges has at least n-k components 1.2.11

- Examples:

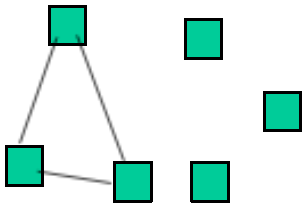
n =2, k =1,
1 component



n =3, k =2,
1 component



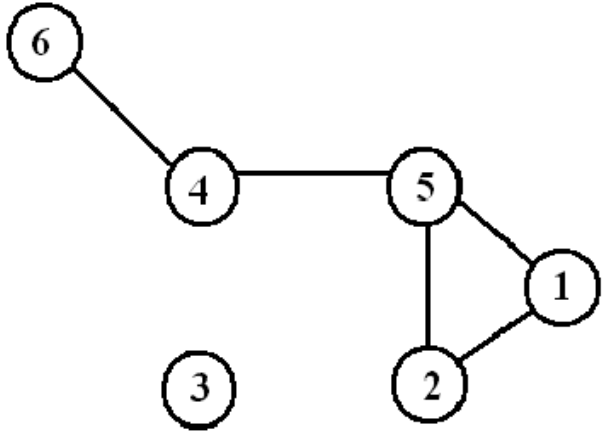
n =6, k =3,
3 components



n =6, k =3,
4 components

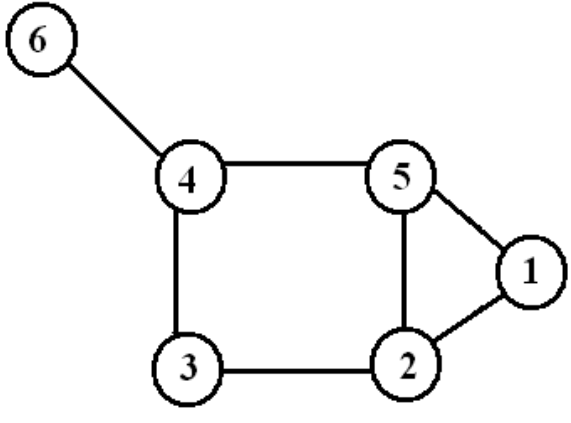
Spanning subgraph

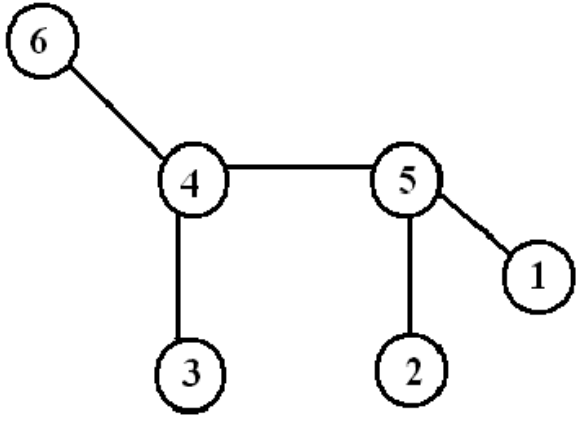
- Subgraph H has the same vertex set as G.
  - Possibly not all the edges
  - "H spans G".



Spanning tree

- Let G be a connected graph. Then a ***spanning tree*** in G is a subgraph of G that includes every node and is also a tree.





Isomorphism

- Bijection, i.e., a one-to-one mapping:

  f : V(G) -> V(H)

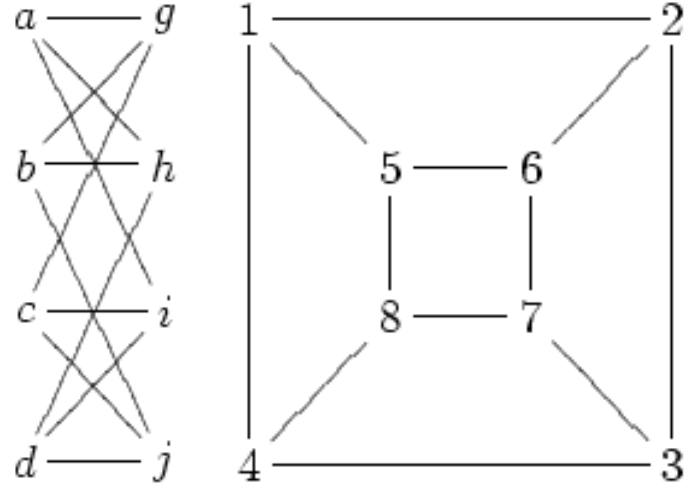
  u and v from G are adjacent if and only if f(u) and f(v) are adjacent in H.
- If an isomorphism can be constructed between two graphs, then we say those graphs are ***isomorphic***.

Isomorphism Problem

Determining whether two graphs are isomorphic

Although these graphs look very different, they are isomorphic; one isomorphism between them is

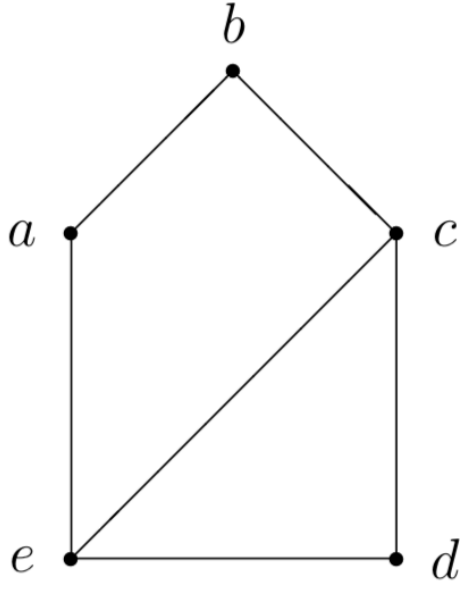
f(a)=1 f(b)=6 f(c)=8 f(d)=3
f(g)=5 f(h)=2 f(i)=4 f(j)=7

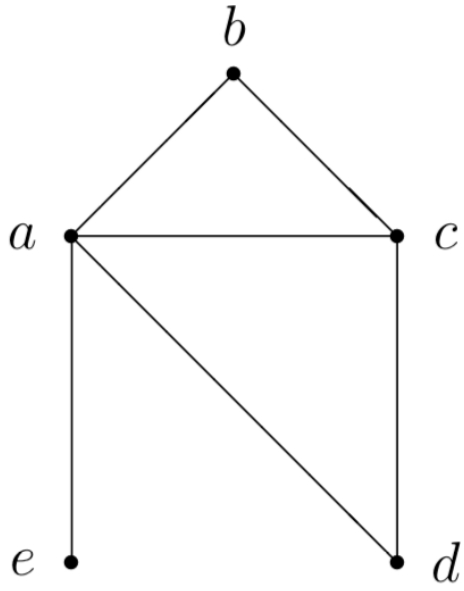


Two graphs are not isomorphic

- Show that they do not share a property that isomorphic simple graphs must have:
  - Have the same number of vertices
  - Have the same number of edges
  - The degrees of the vertices in isomorphic simple graphs must be the same.

Show that the following two graphs are not isomorphic.

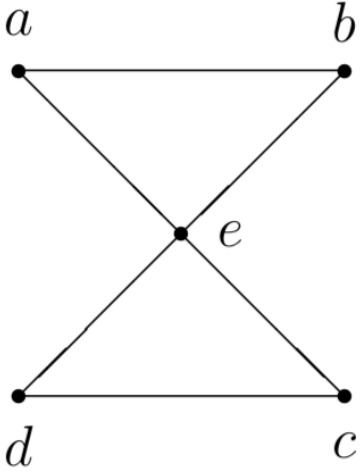


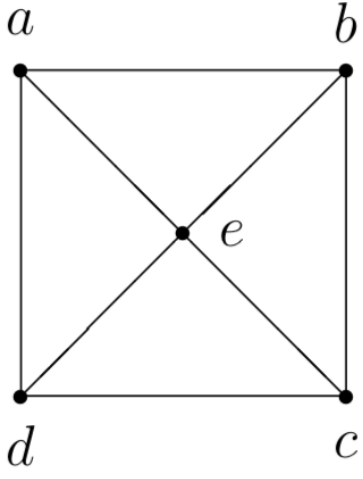


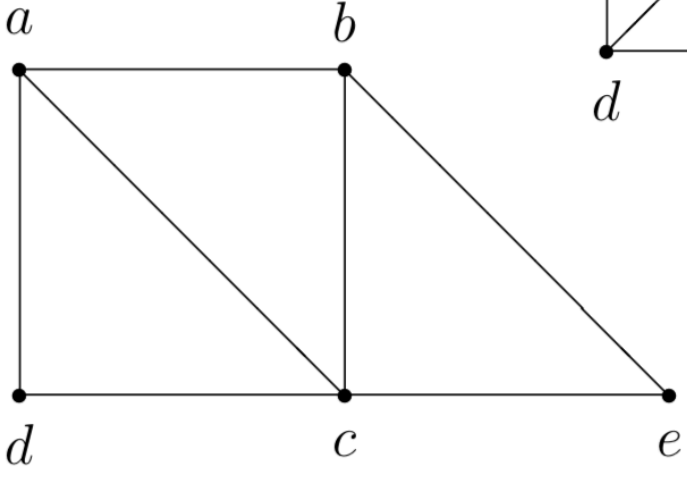
Euler circuit

- *Definition:* An **Euler circuit** in a graph ***G*** is a simple circuit containing every edge of ***G***. (An Euler path in ***G*** is a simple path containing every edge of ***G***).

Which of the following graphs have an Euler circuit?







Theorem: A connected multigraph has an Euler circuit if and only if each of its vertices has even degree.

- How should we construct an Euler Circuit from such graph?

Constructing Euler Circuits

procedure *Euler* (G: connected multigraph with all vertices of even degree)

circuit:= a circuit in G beginning at an arbitrarily chosen vertex with edges successively added to form a path that returns to this vertex

$H := G$ with the edges of this circuit removed

while H has edges

begin

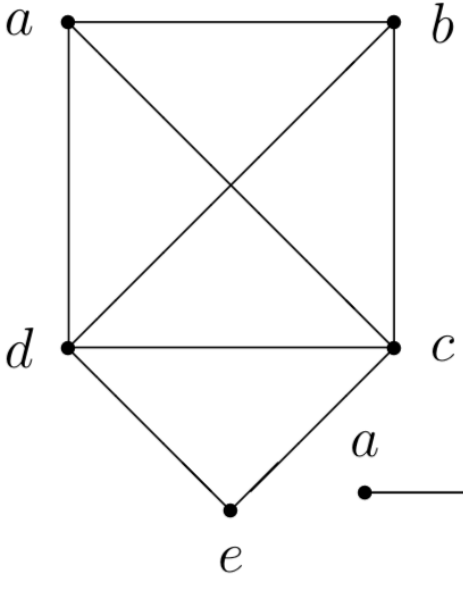
subcircuit := a circuit in H beginning at a vertex in H that also is an endpoint of an edge of circuit

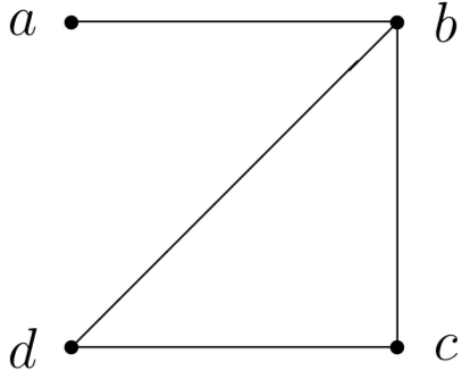
$H := H$ with edges of *subcircuit* and all isolated vertices removed

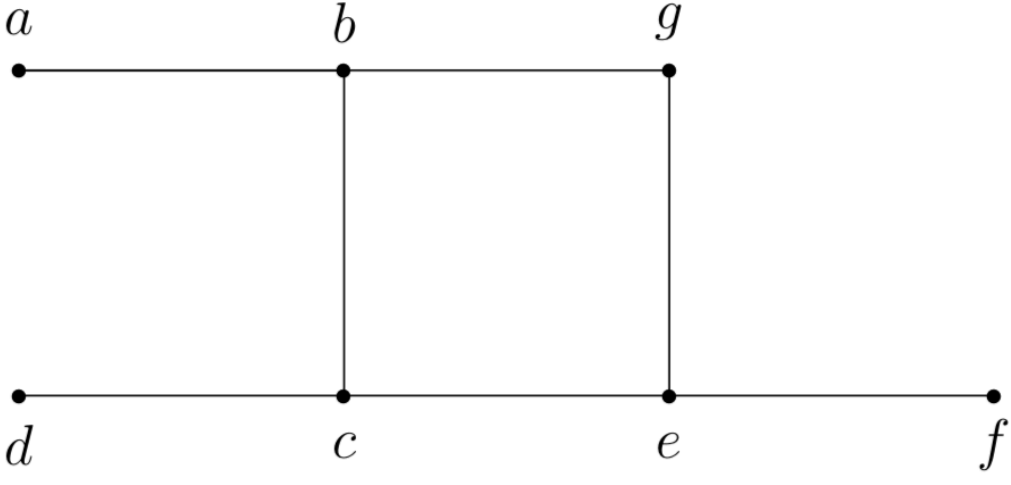
circuit := circuit with *subcircuit* inserted at the appropriate vertex

end { circuit is an Ruler circuit }

Definition 2.9 *A path $v_0, v_1, \cdots, v_n$ in the graph $G = (V, E)$ is called a Hamilton path if $V = \{v_0, v_1, \cdots, v_{n-1}, v_n$ and $v_i \neq x_j$ for $0 \leq i < j \leq n$. A circuit $v_0, v_1, \cdots, v_{n-1}, v_n, v_0$ (with $n > 1$) in a graph $G = (V, E)$ is called a Hamilton circuit if $v_0, v_1, \cdots, v_{n-1}, v_n$ is a Hamilton path.*







Theorem 2.10 *(Dirac) If G is a simple graph with n vertices with $n \geq 3$, such that the degree of each vertex is at least $n/2$, then G has a Hamilton circuit.*

Theorem 2.11 *(Ore) If G is a simple graph with n vertices with $n \geq 3$, such that $deg(u) + deg(v) \geq n$ for every pair of nonadjacent vertices u and v in G, then G has a Hamilton circuit.*

Check Point: Main Methods of the Graph ADT

- Vertices and edges
  - are positions
  - store elements
- Accessor methods
  - aVertex()
  - incidentEdges(v)
  - endVertices(e)
  - isDirected(e)
  - origin(e)
  - destination(e)
  - opposite(v, e)
  - areAdjacent(v, w)
- Update methods
  - insertVertex(o)
  - insertEdge(v, w, o)
  - insertDirectedEdge(v, w, o)
  - removeVertex(v)
  - removeEdge(e)
- Generic methods
  - numVertices()
  - numEdges()
  - vertices()
  - edges()

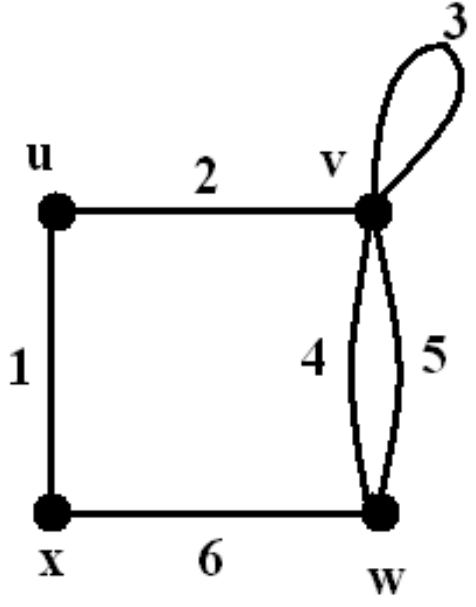
References

- Aldous & Wilson, *Graphs and Applications. An Introductory Approach*, Springer, 2000.
- WWasserman & Faust, *Social Network Analysis*, Cambridge University Press, 2008.

Exercise 1

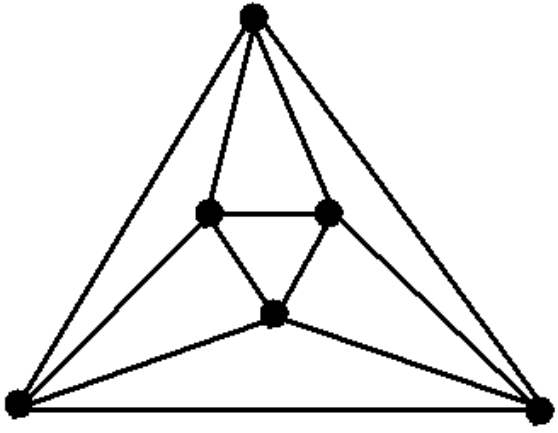
Which of the following statements hold for this graph?

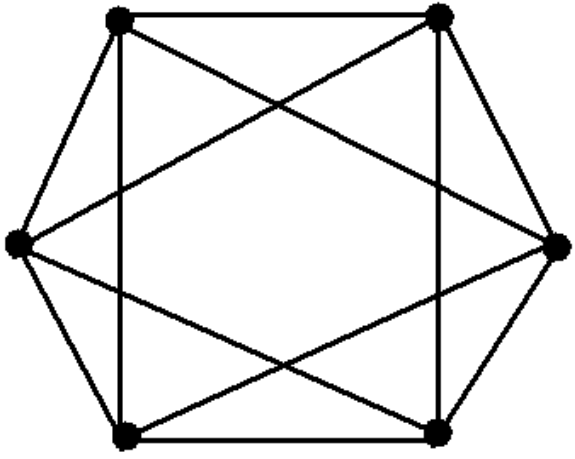
(a) nodes v and w are adjacent;
(b) nodes v and x are adjacent;
(c) node u is incident with edge 2;
(d) Edge 5 is incident with node x.



Exercise 2

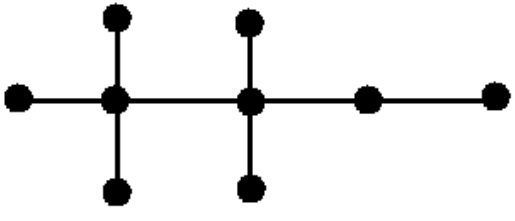
Are the following two graphs isomorphic?



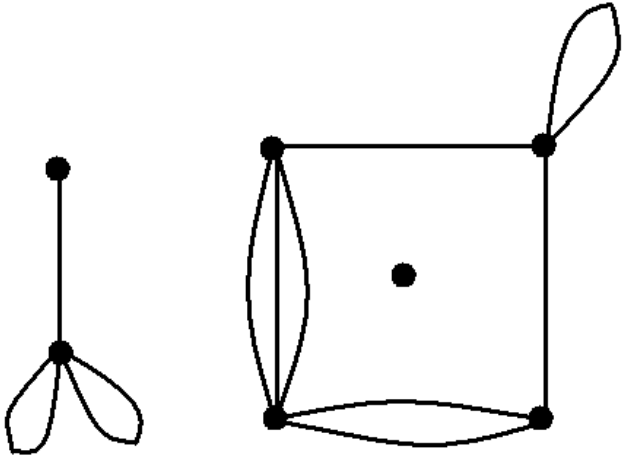


Exercise 3

Write down the degree sequence of each of the following graphs:



(a)



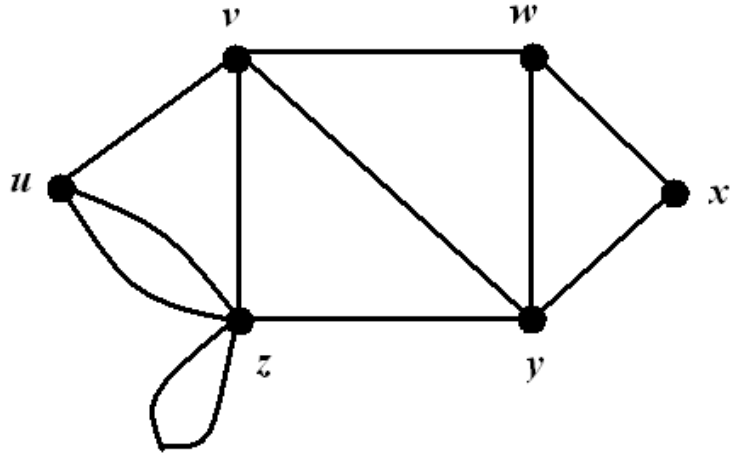
(b)

Exercise 3

Complete the following statements concerning the graph given below:

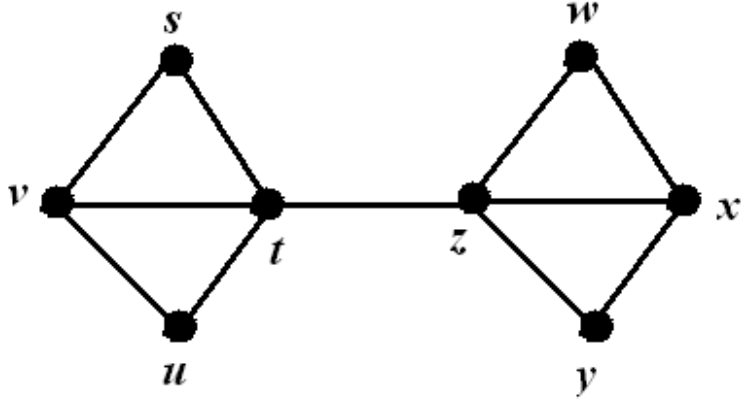
(a) *xyzzvy* is a _____of length____between___and__;

(b) *uvyz* is _____of length ____between___and__.



Exercise 4

Write down all the paths between s and y in the following graph. Build the distance matrix of the graph.



Exercise 5

Draw all the non-isomorphic trees with 6 nodes.

Exercise 6

Draw the graphs given by the following representations:

Node list

1 2 3 4
2 4
3 4
4 1 2 3
5 6
6 5

Edge list

1 2
1 4
2 2
2 4
2 4
3 2
4 3

Adjacency matrix

	1	2	3	4	5
1	0	2	0	1	1
2	2	0	0	1	1
3	0	0	0	0	0
4	1	1	0	0	2
5	1	1	0	2	0

Exercise 7*

(a) Complete the following tables for the number of walks of length 2 and 3 in the above digraph.

.

(b) Find the matrix products A2 and A3, where A is the Adjacency matrix of the above digraph.
(c) Comment on your results.