Graph Theory

2nd year computer science   L2

Pr. Cherif Foudil
Computer Science Department
Biskra University

2025

Course program

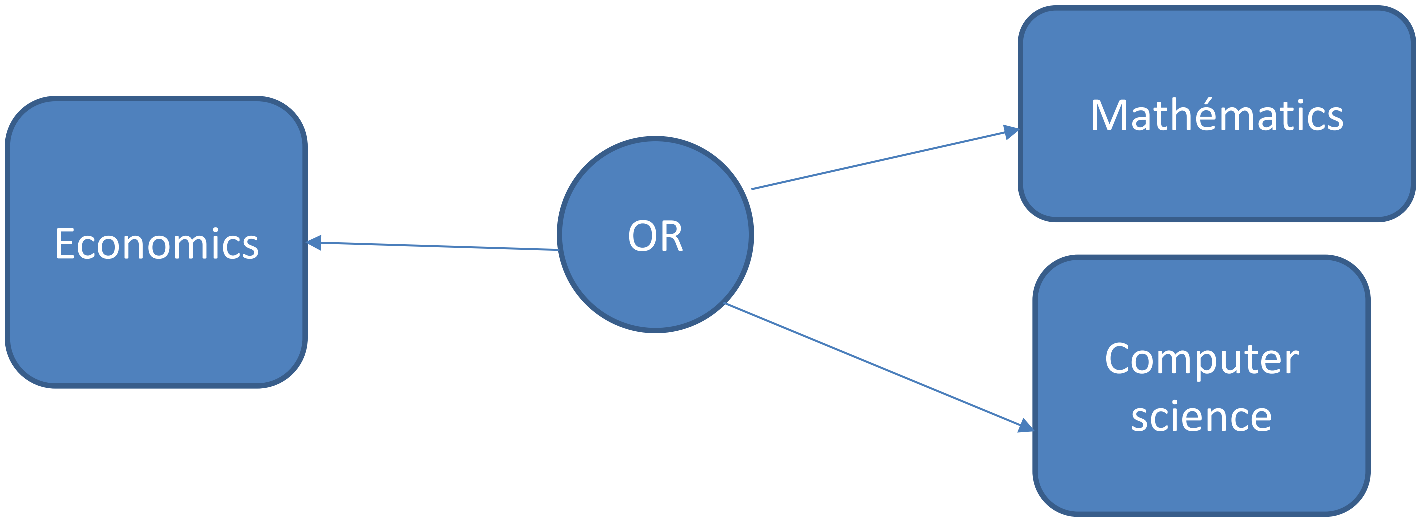
- Basic definitions
- Connectivity in a graph
- Notable paths
  - Hamiltonian
  - and Eulerian
- Trees and arborescences
- Pathfinding
- Maximum flow problem

Chapter 1

Basic definitions

1.1 Motivation

- **Graph theory** (GT) falls within the field of **operations research** (OR), which serves as a crossroads where **economics**, **mathematics**, and **computer science** intersect.

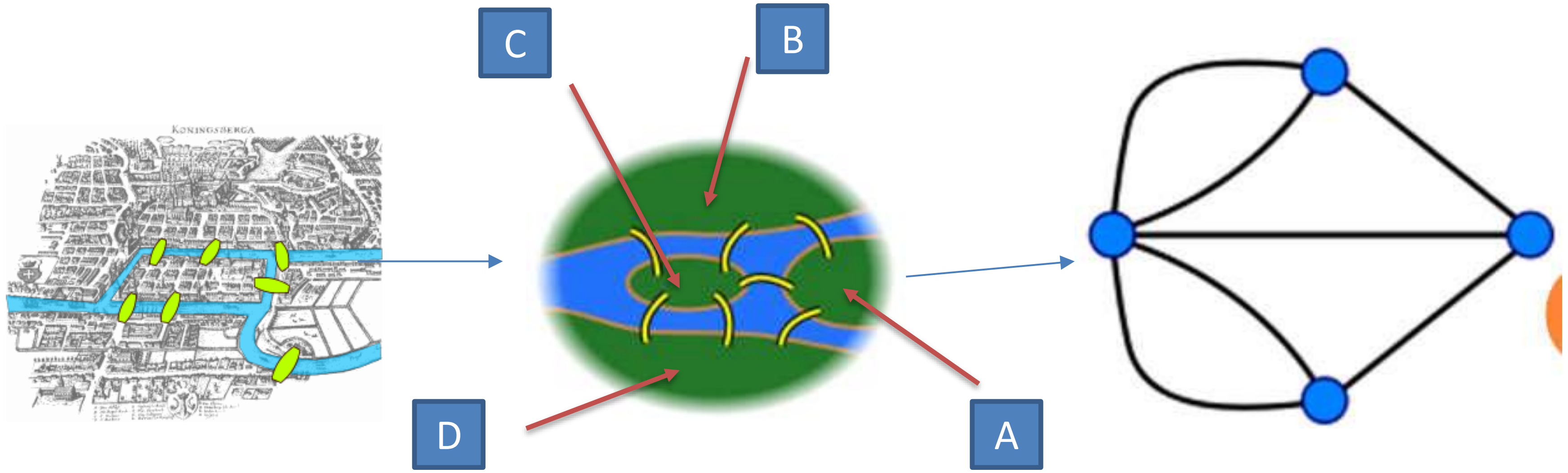


- The objective of **OR** is to find **optimal solutions** for economic problems using mathematical methods that can be programmed by computer.

1.2 History

***Question*:**

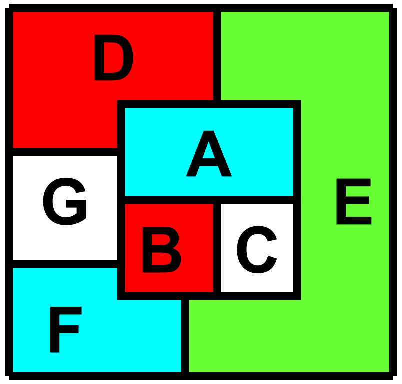
- Is it possible to walk through the city of Königsberg in such a way as to cross each bridge exactly one time and return to the starting district?

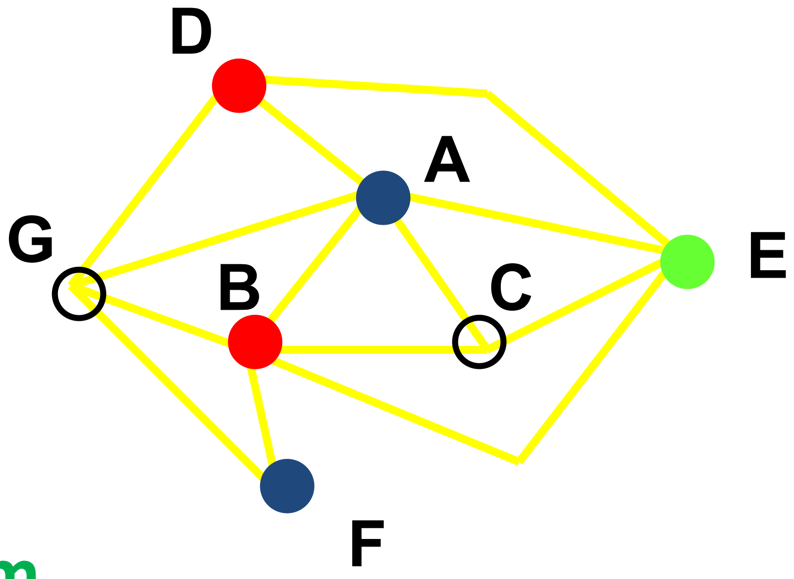


- The German mathematician **L. Euler** (1736) provided an answer to the problem faced by the residents of the city of Königsberg: how to cross the seven bridges of this city without ever crossing the same one twice. → **This marked the birth of graph theory**.

1.2 History

- In 1852, graph theory gained popularity thanks to the "**Four-Color Theorem.**"
- It was demonstrated that only four different colors are needed to color any geographical map in such a way that two adjacent regions (sharing a common boundary) always receive distinct colors.





- **The four-color problem...**

1.2 History

- Starting in 1946, GT (Graph Theory) experienced significant development thanks to researchers motivated by solving practical problems. Among them:
- **Edsger Dijkstra (1959**) for the pathfinding problem,
- **Ford and Fulkerson (1956)** for the maximum flow problem,
- **Bernard Roy (1958)** developed the MPM method for the scheduling problem.

1.2 History

- Graph theory has become essential in our daily lives for various networks:
- **Transportation networks**: road, air, rail, maritime, water, gas, electricity...
- **Data communication networks**: landline, mobile, WIFI...
- **Information networks**: databases, the web, social networks...

1.3 Graph concepts

1.3.1 definitions

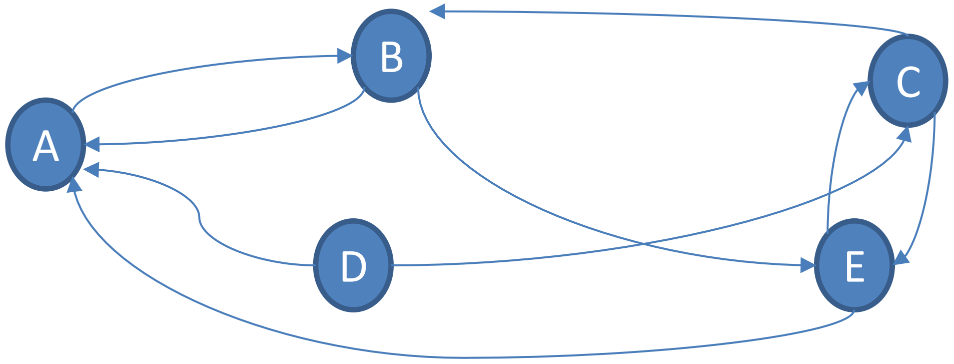
-A graph G, denoted as G=(V, E), is defined as:

- A finite set of vertices V = { v1,v2,…vn }
- A finite set of edges (or arcs) E, where an edge (or arc) connects a pair of vertices from V, E = { e_1,e_2,…e_m}

Each edge has two endpoints

It is not easy to visualize a given graph in this form → drawing it is more helpful.

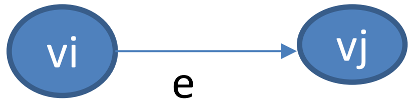
Example: A one-way traffic plan of a city, where each locality (vertex) and each road are represented by a directed arc indicating the direction of traffic.



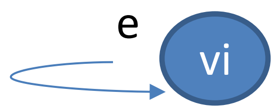
1.3 Graph concepts

1.3.2 Directed graph

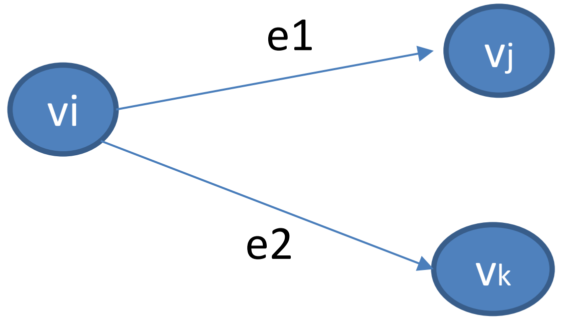
A directed graph is a graph in which if e = (v_i, v_j), v_i is the origin of the arc e, and v_j is the destination of e.



A loop, denoted as b = (v_i, v_i), is an arc where the origin coincides with the destination.



Two arcs are adjacent if they have at least one common endpoint.

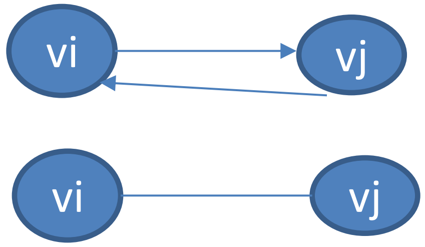


e_1 and e_2 are adjacent; they have v_i in common

1.3 Graph concepts

1.3.2 Undirected graph

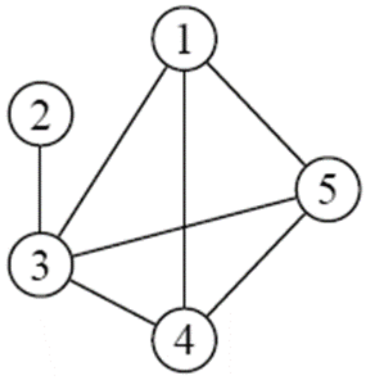
An undirected graph is a graph in which for all v_i, v_j ∈ V, if (v_i, v_j) ∈ E, then (v_j, v_i) ∈ E.



Example: G= (V, E),

V= (1, 2, 3, 4, 5) E= {(1, 5), (1, 4), (1, 3), (2, 3), (3, 4) ; (4, 5), (3, 5)}.

The graph G is represented in a schematic way as follows:



1.3 Graph concepts

1.3.2 Undirected graph

An edge e in E is defined by an unordered pair (v_1, v_2) of vertices called the endpoints of e.

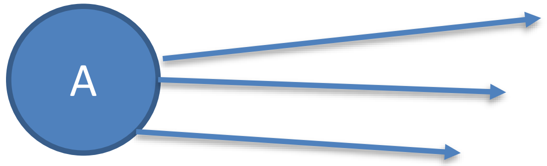
If the edge e connects vertices v_1 and v_2, we can say that these vertices are adjacent or incident to e, or that the edge e is incident to vertices v1 and v2.

The order of a graph is the number of vertices **n** in that graph.
$|V| = n$

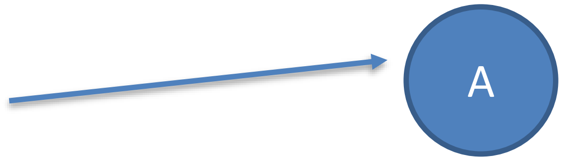
1.3 Graph concepts

1.3.3 The degree of a vertex:

- The vertex A is the initial endpoint of 3 arcs; we say that the **positive degree** of A is equal to 3, denoted by: $\mathbf{d^+_G (A) = 3}$



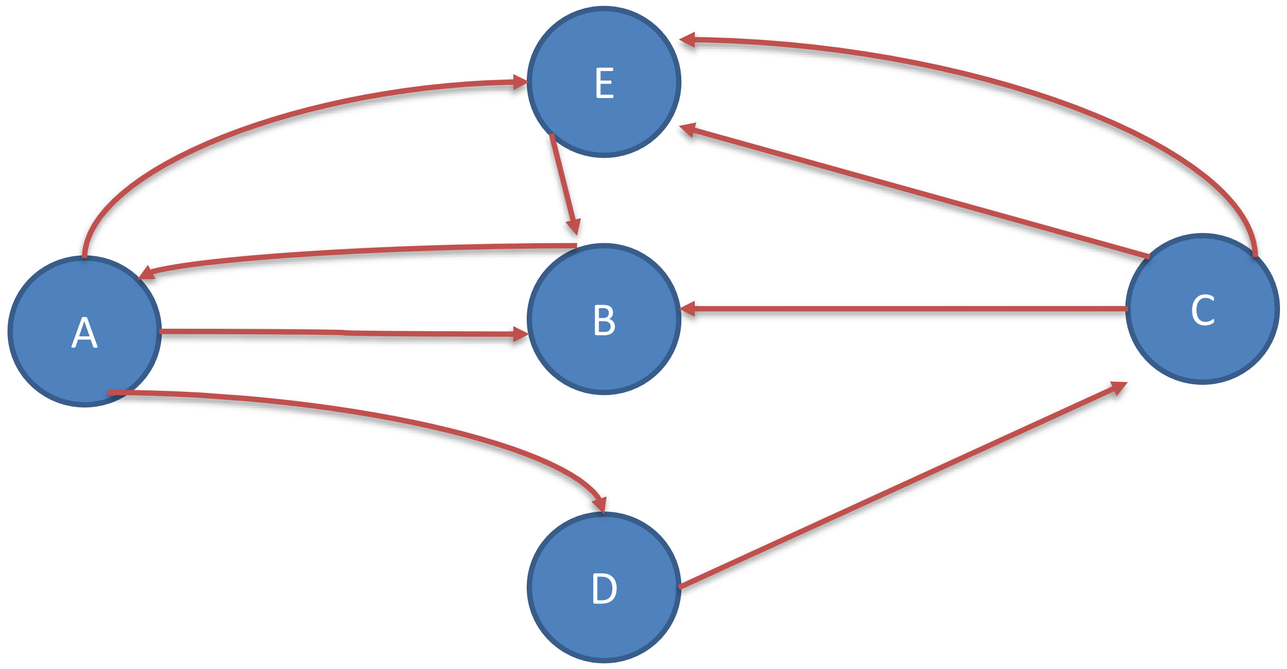
- The vertex A is the terminal endpoint of one arc; we say that **the negative degree** of A is equal to 1, denoted by: $\mathbf{d^-_G (A) = 1}$



- The degree $\mathbf{d_G (A) = d^+_G (A) + d^-_G (A)}$

1.3 Graph concepts

1.3.3 The degree of a vertex : Example



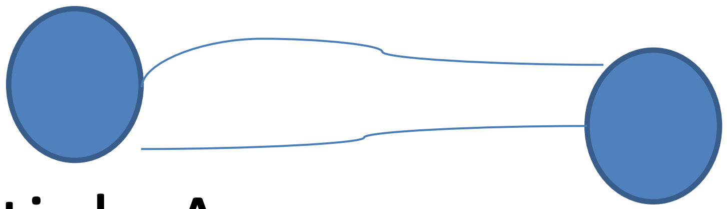
	A	B	C	D	E	
d^+_G (x)	3	1	3	1	1	9
d^-_G (x)	1	3	1	1	3	9
d_G (x)	4	4	4	2	4	18

1.3 Graph concepts

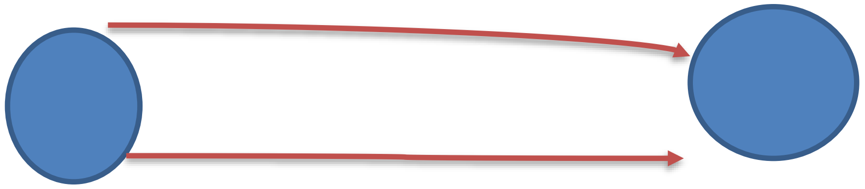
1.3.4 Types of graphs

a) **Multiple Graph**: G = (V, E) is a graph in which the set E of edges may contain more than one edge connecting two given vertices.

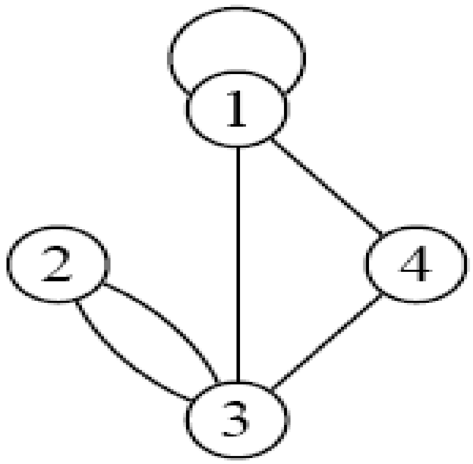
Multiple Edges



Multiple Arcs



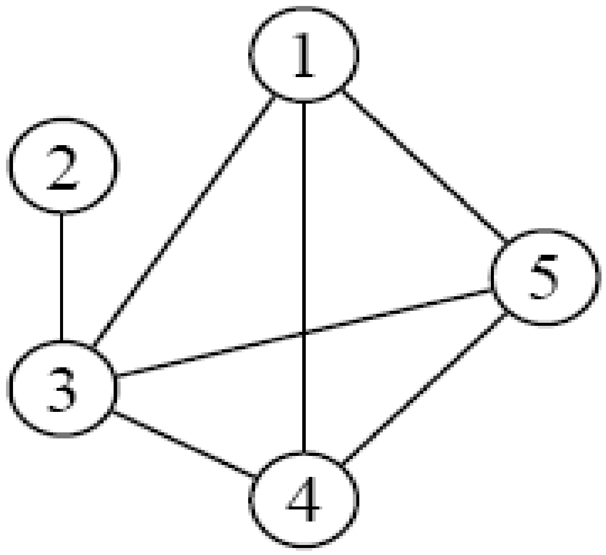
multiple graph



1.3 Graph concepts

1.3.4 Types of graphs

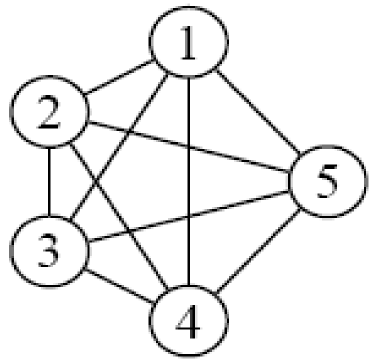
b) Simple Graph: A graph without loops or multiple edges (arcs).

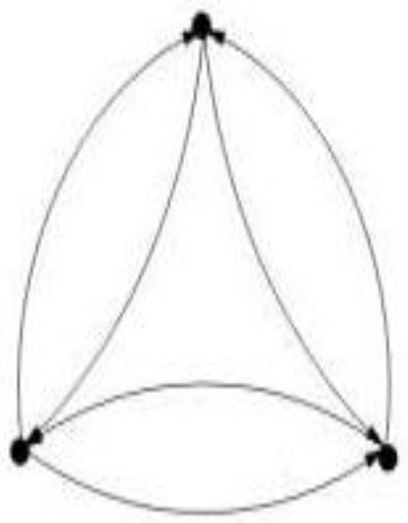


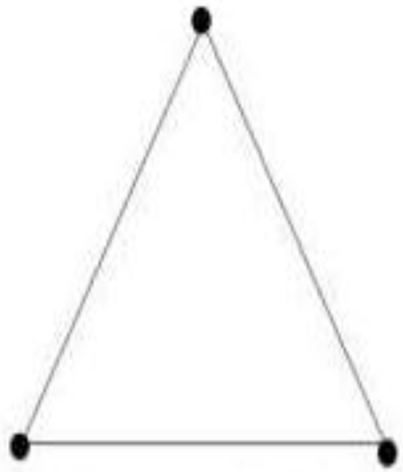
1.3 Graph concepts

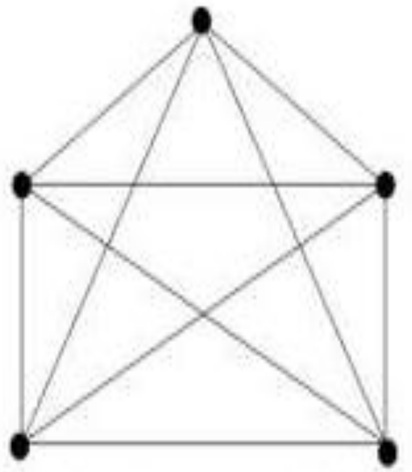
1.3.4 Types of graphs

c) Complete Graph: A graph in which every vertex is directly connected to every other vertex.




Graphe complet (orienté) sur trois sommets


Graphe complet (non-orienté) sur trois sommets


Graphe complet (non-orienté) sur cinq sommets

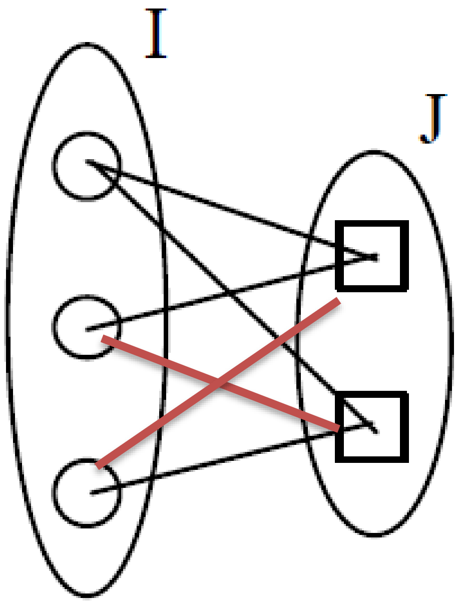
1.3 Graph concepts

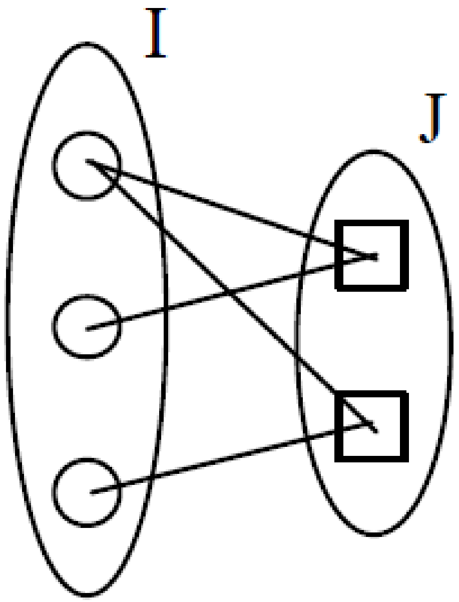
1.3.4 Types of graphs

d) Bipartite Graph: If its set of vertices can be divided into two distinct subsets I and J such that each edge has one endpoint in I and the other in J.

e) Complete Bipartite Graph :

If every vertex in set I is connected to every vertex in set J.

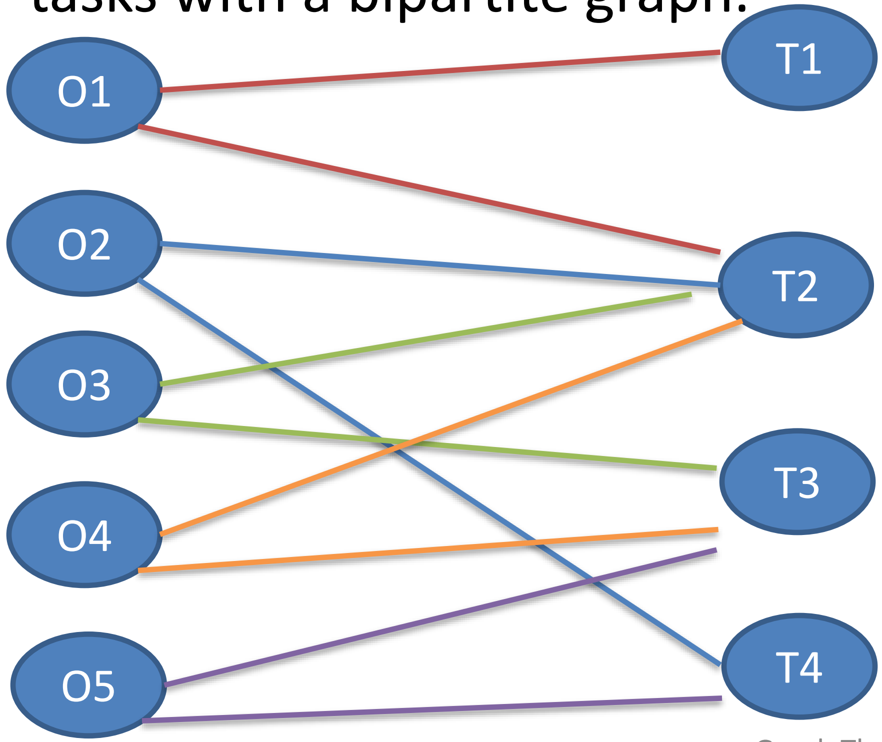




1.3 Graph concepts

1.3.4 Types of graphs

d) Example of a Bipartite Graph: In a workshop with 5 workers, each capable of performing 1 to 4 tasks, we represent the possibilities of assigning workers to different tasks with a bipartite graph.



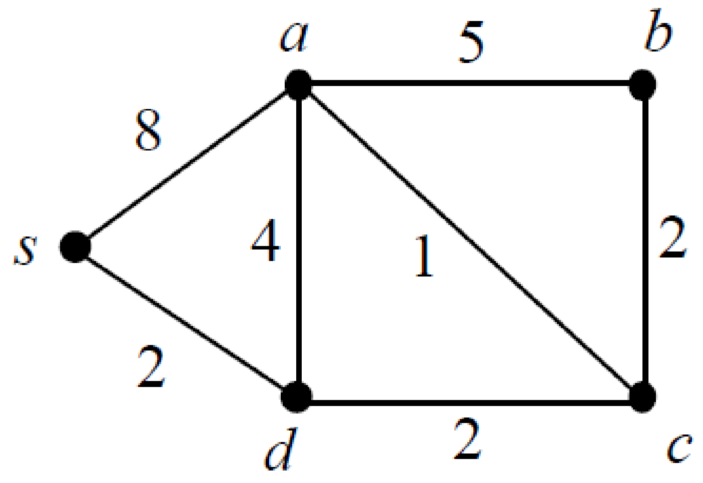
Note: If each worker can perform all tasks, a **complete bipartite graph** is obtained.

1.3 Graph concepts

1.3.4 Types of graphs

f) Weighted Graph:

When the edges represent a cost, they are assigned a number, creating a **weighted graph**.

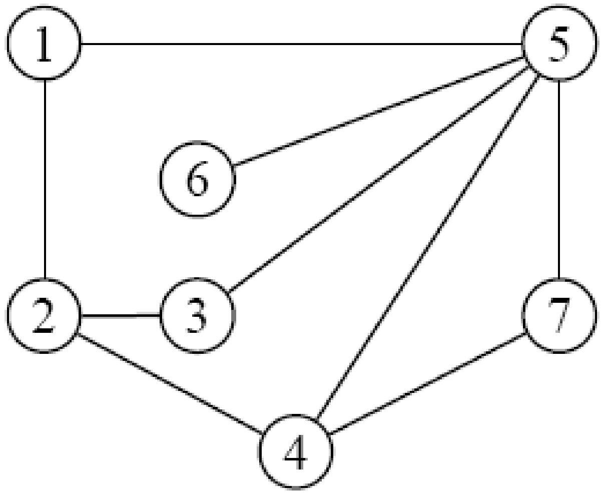


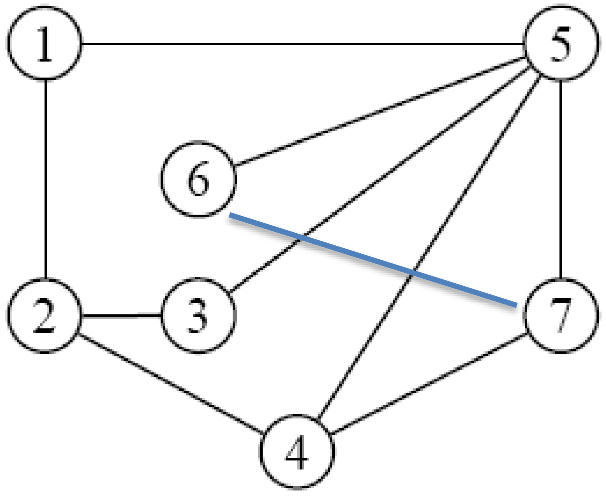
1.3 Graph concepts

1.3.4 Types of graphs

g) Planar Graph:

If its edges do not intersect, in other words, if it can be drawn in a plane in such a way that its edges do not cross.

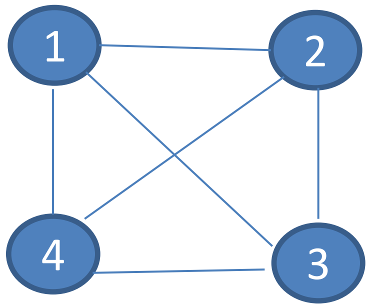


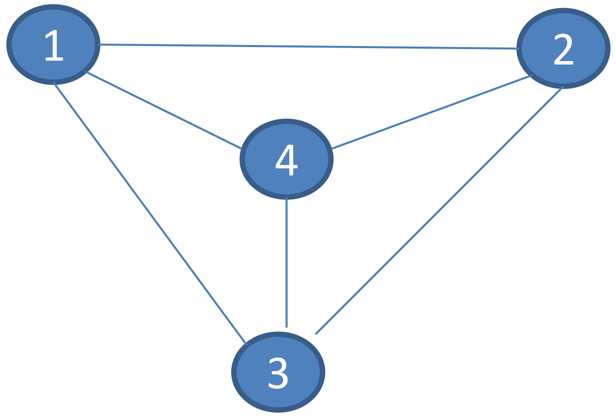


1.3 Graph concepts

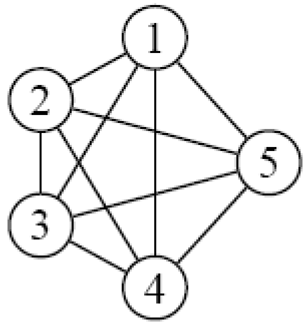
1.3.4 Types of graphs

Example 1: Complete graph with 4 vertices, it is planar if it can be transformed.



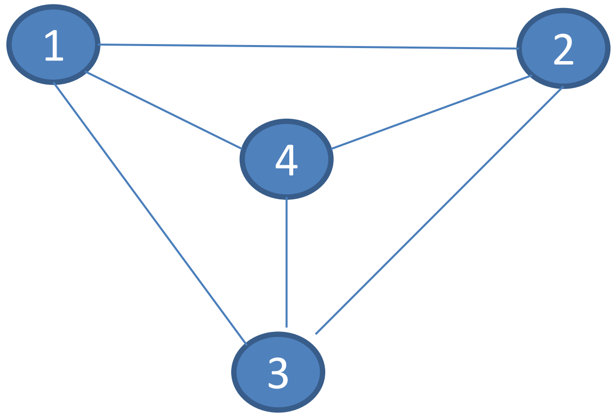


Example 2: Complete graph with 5 vertices, it is not planar, it cannot be transformed.



1.3 Graph concepts

1.3.4 Types of graphs G=(V,E)



Note: Planar graphs satisfy the formula(Euler formula):

$|V| + F = |E| + 2$ **4 + 4 = 6 + 2**

|V| number of vertices

|E| number of edges

F number of faces or regions

1.3 Graph concepts

1.3.4 Types of graphs

h) Graph - Isomorphism

G1 = (V1, E2) and G2 = (V2, E2) are isomorphic if:

There is a one-to-one function f from V1 to V2 with the property that a and b are adjacent in G1 if and only if f (a) and f (b) are adjacent in G2, for all a and b in V1.

Function f is called **isomorphism**

G1 = (V1, E1) , G2 = (V2, E2)

f(u1) = v1, f(u2) = v4, f(u3) = v3, f(u4) = v2,

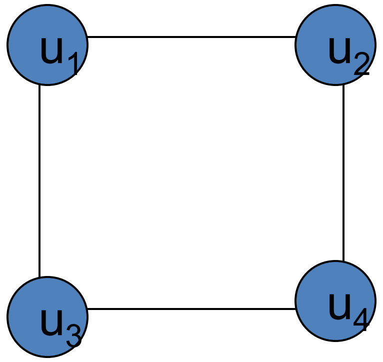
1.3 Graph concepts

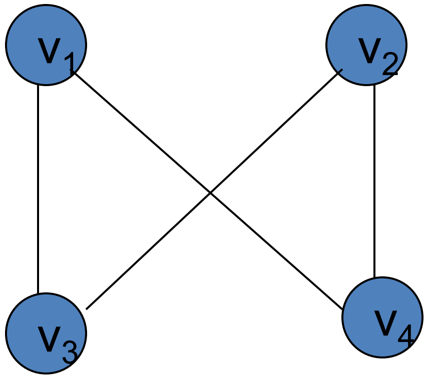
1.3.4 Types of graphs

i)Graph – Isomorphism example 1

Representation example: G1 = (V1, E1) , G2 = (V2, E2)

$f(u_1) = v_1$, $f(u_2) = v_4$, $f(u_3) = v_3$, $f(u_4) = v_2$,

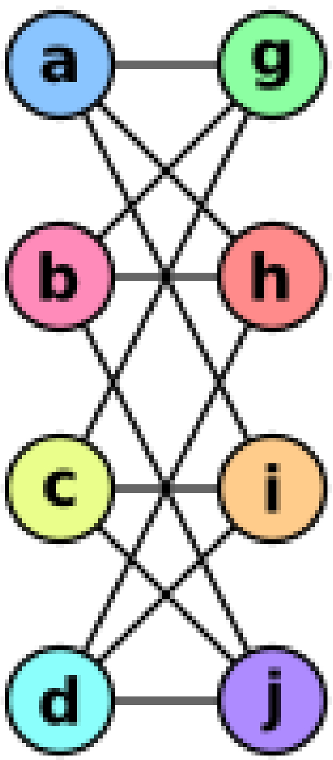


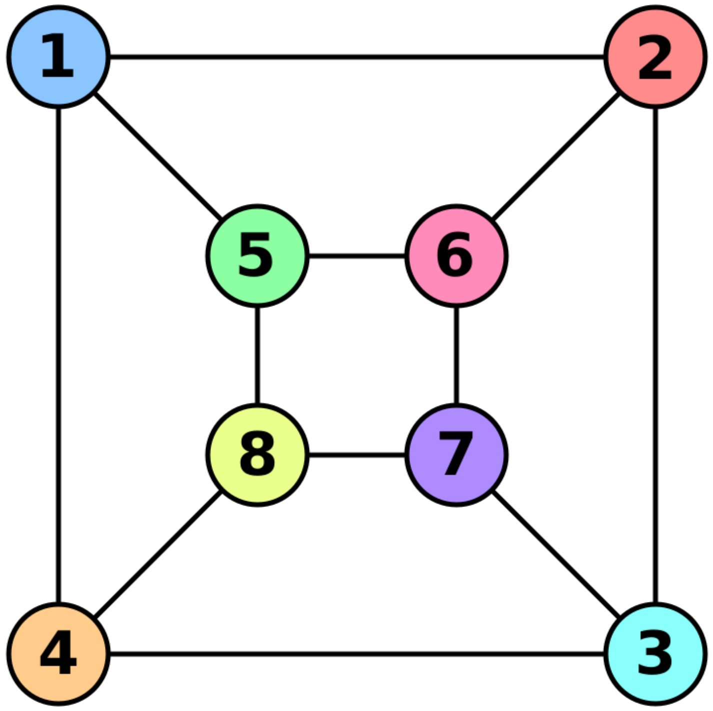


1.3 Graph concepts

i)Graph – Isomorphism (example 2)

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





1.3 Graph concepts

1.3.4 Types of graphs

j) Other graphs:

- A **reflexive** graph is a graph having a loop on each vertex
- A graph G = (V, E) is **symmetric** if,

  $\forall$ arc $e_1 = (v_i, v_j) \in E$, the arc $e_2 = (v_j, v_i) \in E$.

- A graph G = (V, E) is **antisymmetric** if,

  $\forall$ (For every) arc $e_1 = (v_i, v_j) \in E$, the arc $e_2 = (v_j, v_i) \notin E$.

- A graph G = (V, E) is **transitive** if,

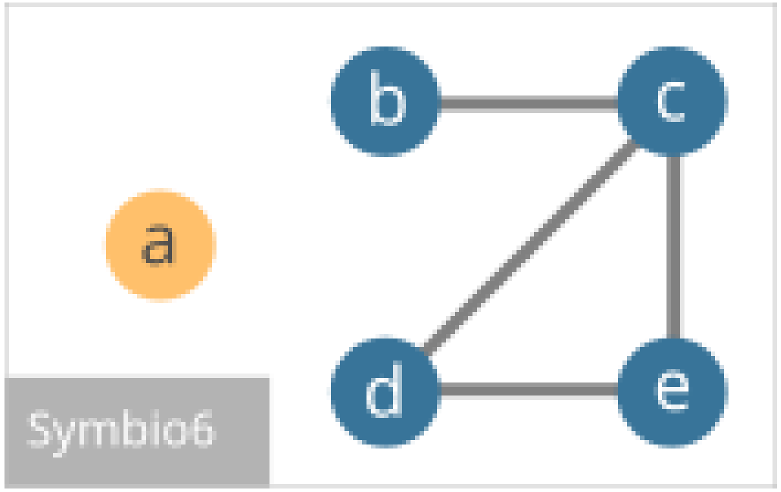
  $\forall$ arc $e_1 = (v_i, v_j) \in E$ and arc $e_2 = (v_j, v_k) \in E$ then the arc $e_3 = (v_i, v_k) \in E$.

1.3 Graph concepts

1.3.4 Types of graphs

k) Isolated vertex:

A vertex of a graph that has no edges is a vertex with degree zero.



Graph with isolated vertex a.

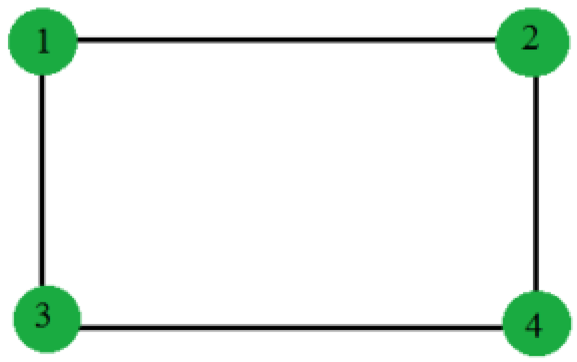
Null Graph: A null graph is defined as a graph which consists only the isolated vertices

1.3 Graph concepts

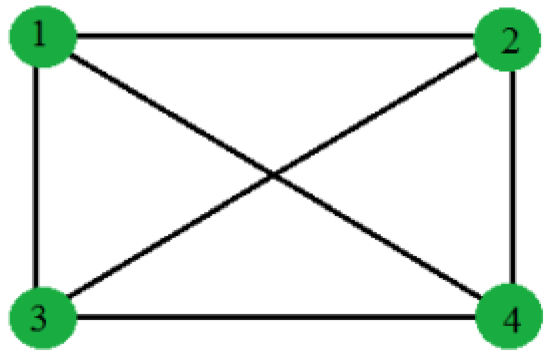
1.3.4 Types of graphs

l) Regular graph:

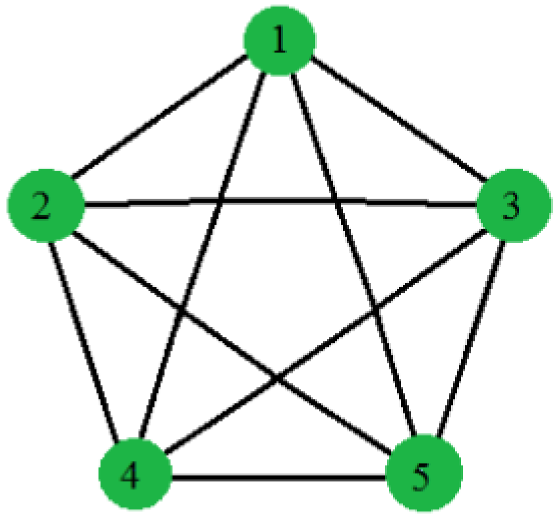
A graph is called regular graph if degree of each vertex is equal. A graph is called **K regular** if degree of each vertex in the graph is K.



2 Regular



3 Regular

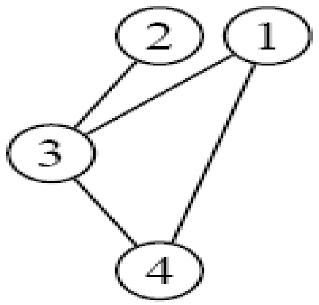


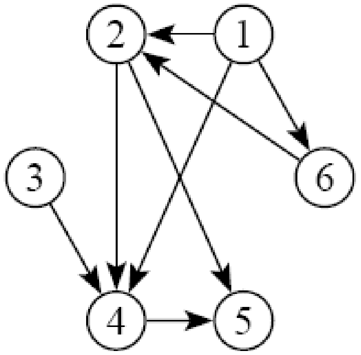
4 Regular

1.4 Representation of a graph

a) Sagittal representation (drawing):

Vertices are represented by circles, and the relationships are represented by lines or arrows,





b) Matrix representation.

For a given graph G=(V,E) with |V|=n and |E|=m (n vertices and m edges), we associate 4 types of matrices:

Adjacency matrix, Incidence matrix, Arc matrix, Associated matrix

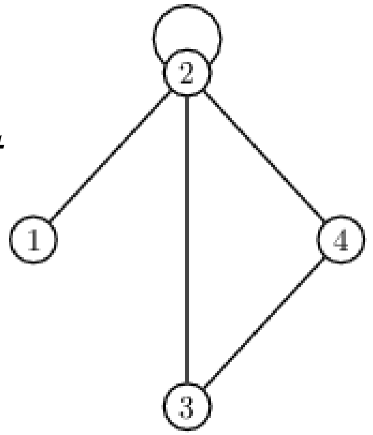
1.4 Representation of a graph

1- Adjacency Matrix: (undirected graph):

Let G be an undirected graph with n vertices numbered from 1 to n. The adjacency matrix of the graph is called matrix A=(ai,aj), where $a_{i,j}$ is the number of edges connecting vertex i to vertex j.

Example: a graph and its corresponding adjacency matrix:

Other definition: A square matrix where each row and column represent a vertex, and the entries indicate whether there is an edge between the vertices. It's often used for ***simple graphs.***



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

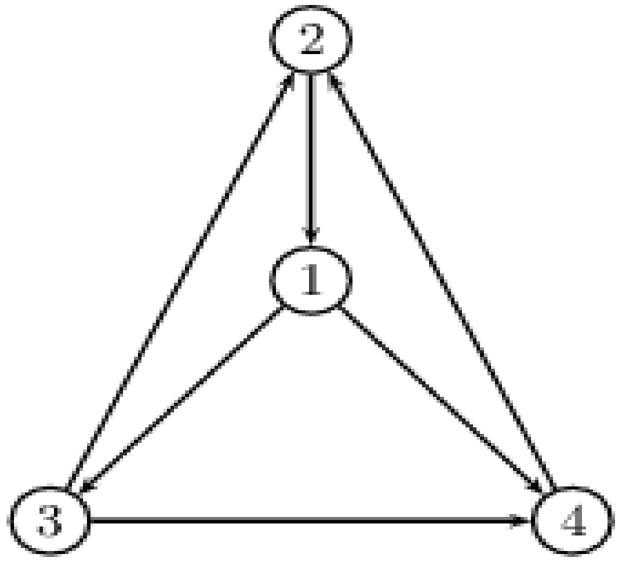
It is a symmetric matrix

1.4 Representation of a graph

Adjacency Matrix: (directed graph)

We can also define the adjacency matrix of a directed graph. This time, the coefficient $a_{i,j}$ represents the number of arcs originating from vertex i and ending at vertex j.

Example : For the following graph:



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

This matrix is no longer symmetric.

1.4 Representation of a graph

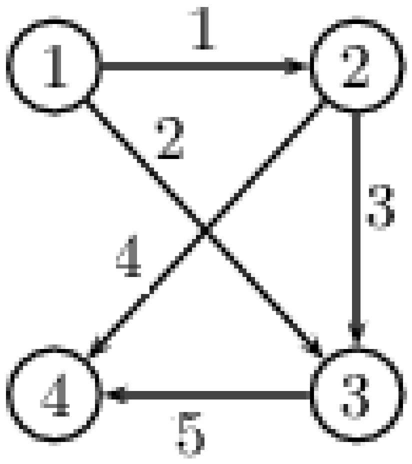
2- Incidence matrix: (directed graph)

Let G be a directed graph with n vertices numbered from 1 to n, and m arcs numbered from 1 to m. The **incidence matrix** of the graph is called matrix A=($a_{i,j}$) consisting of n rows and m columns such that:

-$a_{i,j}$ equals +1 if the arc numbered j has vertex i as its origin;

-$a_{i,j}$ equals -1 if the arc numbered j has vertex i as its destination;

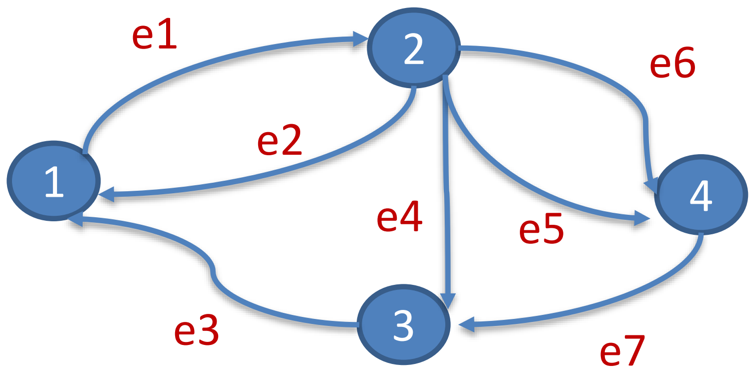
- $a_{i,j}$ equals 0 in all other cases. **A(vertex, arc)**



$$A = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & 1 & 0 \\ 0 & -1 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & -1 \end{pmatrix}.$$

1.4 Representation of a graph

Incidence matrix : Example: G=(V,E) of order 4 with 7 arcs.



Note: *You should start filling the matrix by beginning with the arcs.*

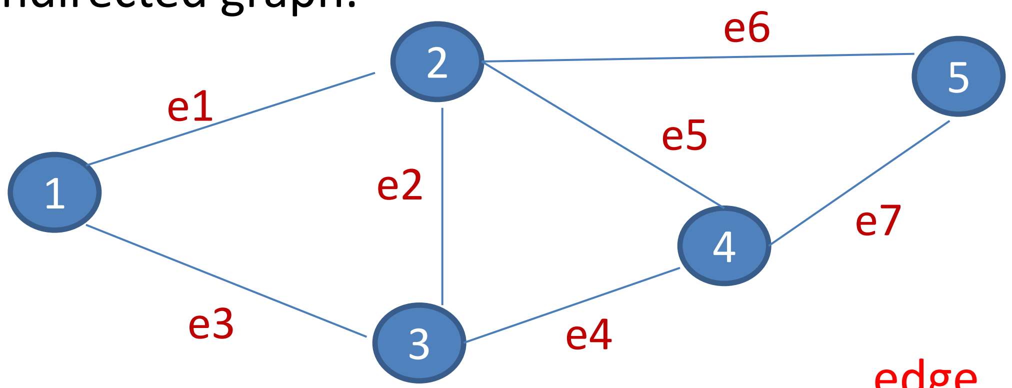
Arc

Vertex

	e1	e2	e3	e4	e5	e6	e7
1	1	-1	-1	0	0	0	0
2	-1	1	0	1	1	1	0
3	0	0	1	-1	0	0	-1
4	0	0	0	0	-1	-1	1

1.4 Representation of a graph

Incidence matrix : One can also define the incidence matrix (vertex/edge) for an undirected graph.



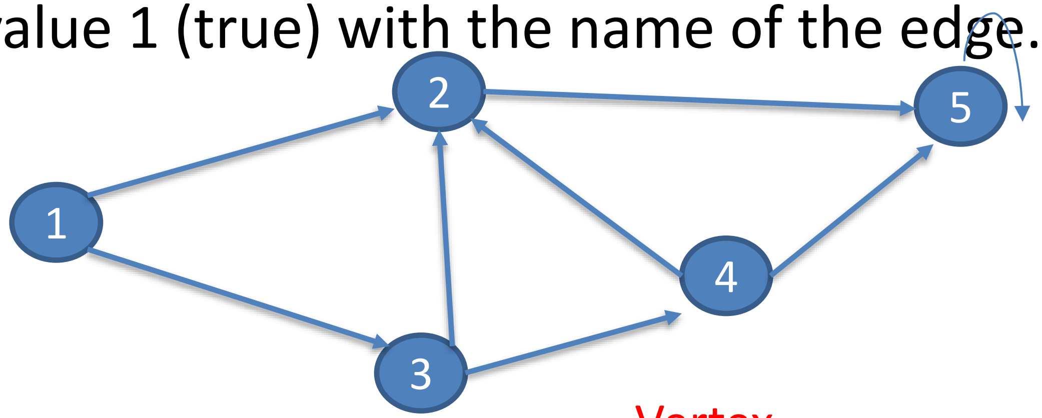
edge

Vertex

	e1	e2	e3	e4	e5	e6	e7
1	1	0	1	0	0	0	0
2	1	1	0	0	1	1	0
3	0	1	1	1	0	0	0
4	0	0	0	1	1	0	1
5	0	0	0	0	0	1	1

1.4 Representation of a graph

3- Edge Matrix: It is defined from the adjacency matrix by replacing the value 1 (true) with the name of the edge.



Vertex

Vertex

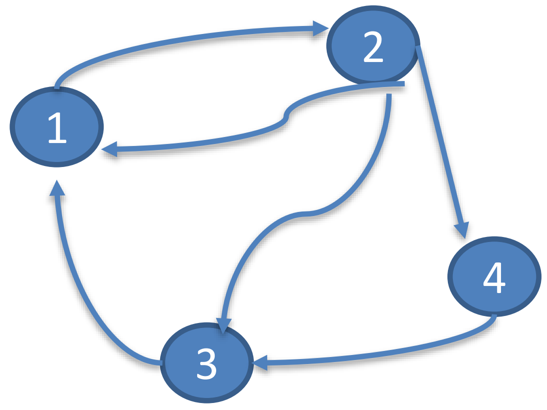
	1	**2**	**3**	**4**	**5**
1	0	12	13	0	0
2	0	0	0	0	25
3	0	32	0	34	0
4	0	42	0	0	45
5	0	0	0	0	55

1.4 Representation of a graph

4- Associated Matrix: We replace the name with the number of edges or arcs.

C) Representation by dictionaries:

The graph is represented by a table or dictionary of vertices. Each vertex has a list of **successor** (**predecessor**) vertices. We use a function Γ such that $\Gamma^{+}(x)$ is the list of successors and $\Gamma^{-}(x)$ is the list of predecessors.



Vertice	Successor $\Gamma^{+}(x)$	Predecessor $\Gamma^{-}(x)$
1	2	2,3
2	1,3,4	1
3	1	2,4
4	3	2

Chapter 2

Connectivity in a graph

2.1 Paths in a graph

Definitions: Let G=(V,E) be any graph, C={e1, e2, …, ep} a sequence of arcs with cardinality p such that two consecutive arcs ei and ei+1 share a common endpoint.

C is called a **chain** (walk)of cardinality p.

A **path** is a chain(walk) where the arcs are oriented in the same direction.

A **cycle** is a closed chain (the first vertex coincides with the last vertex).

A **circuit** is a closed path (the first vertex coincides with the last vertex).

2.1 Paths in a graph

A path is a **chain** (walk)or a **cycle** or a **route(way)** or a **circuit**.

A **path** is called **elementary** if it visits each of its **vertices exactly once** (except for the first vertex in the case of a cycle and circuit).

A **path** is called **simple** if it traverses each of its **edges exactly once**.

A **path** is called **Hamiltonian** if it visits each and every vertex of the graph exactly once (except for the first vertex in the case of a cycle and circuit).

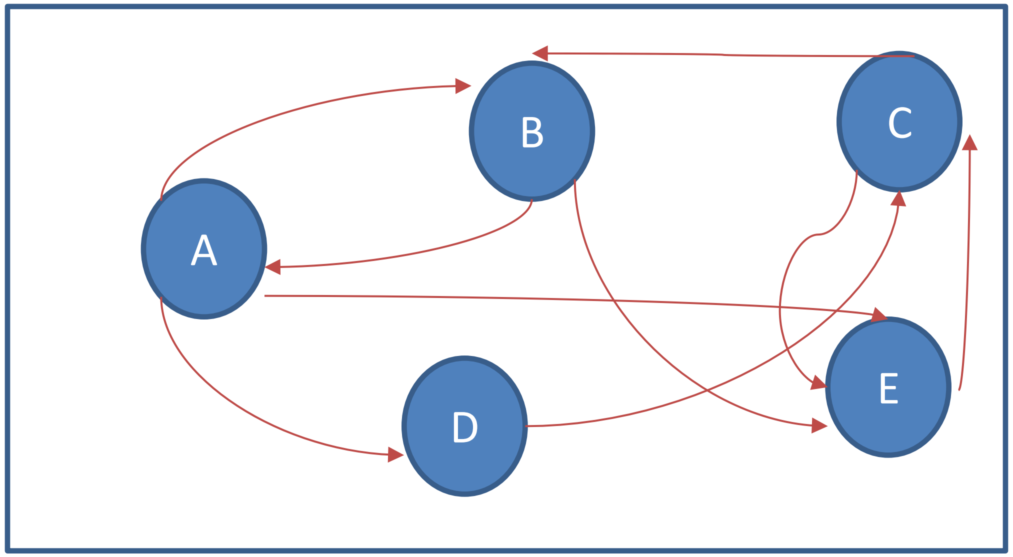
A **path** is called **Eulerian** if it traverses each and **every edge of the graph exactly once.**

2.1 Paths in a graph

2.1.1 Chain:

A chain connecting two vertices V0 and Vk in a graph G is a sequence of vertices connected by edges in such a way that two successive vertices share a common edge. It is denoted as (V0, V1, ..., Vk), and we refer to V0 and Vk as the endpoints of the chain.

Example: G=(V,E)



The sequence (A, B, C, D) is a **simple chain** connecting A to D.

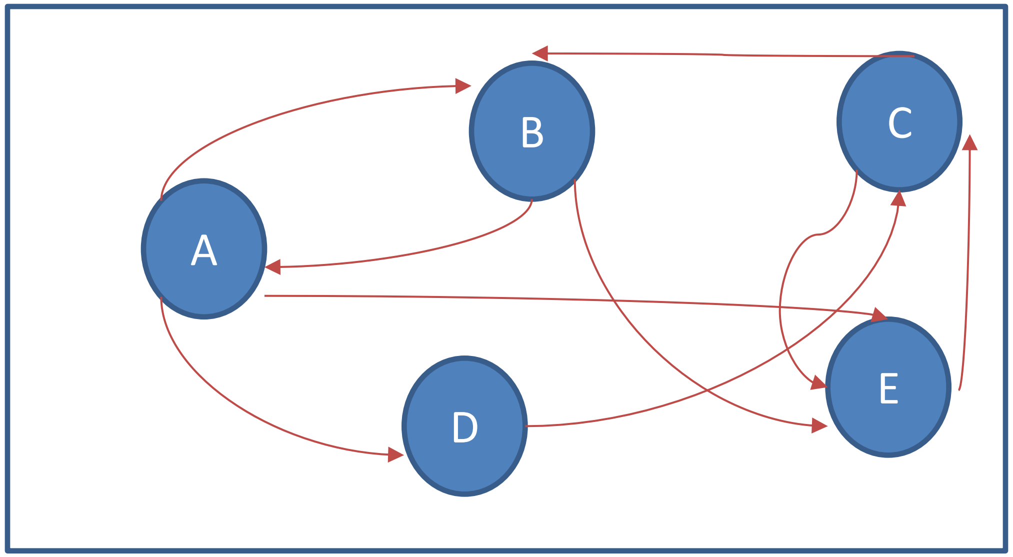
The sequence (A, E, B, A, E) is a path connecting A to E that **is not simple**: the edge (A, E) is traversed twice.

The sequence (A, B, E) is a **simple path** connecting A to E.

2.1 Paths in a graph

2.1.1 Chain:

- The length of the path is equal to the number of edges,
- A path **is elementary** if none of the **vertices** that make up the path appear more than once.
- A path is said to **be simple** if none of the **edges** appear more than once.



We call an **Eulerian path** of a graph G a path that passes **exactly once** through **each** of the edges of G.

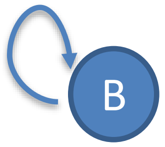
We call a **Hamiltonian path** of a graph G a path that passes **exactly once** through **each** of the vertices of G.

2.1 Paths in a graph

2.1.2 Cycle:

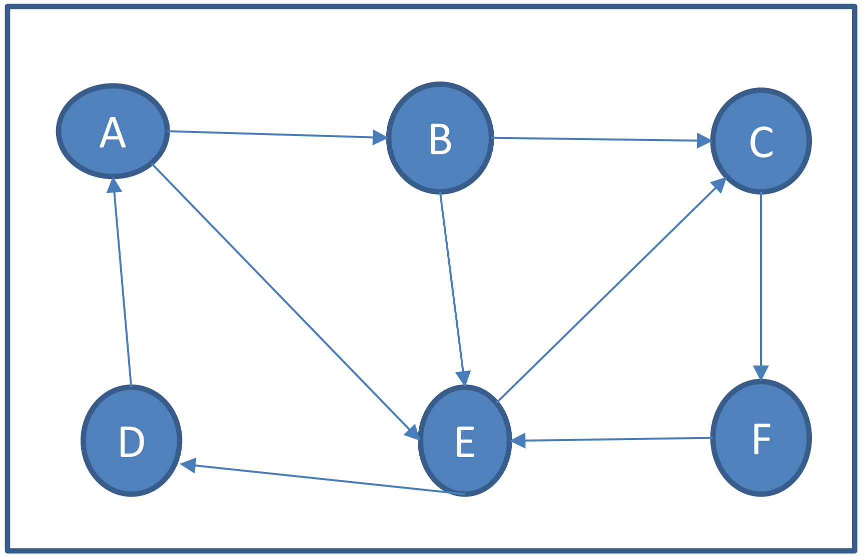
A **cycle** is a path in which the endpoints coincide (e0, e1, ..., ek = e0), and it consists of a succession of **distinct edges**.

Example: (A, B, C, E, A) is a cycle.



A **loop** is a specific type of cycle. (B, B)

A cycle is **elementary** if, while traversing it, one does not encounter the **vertices** multiple times.



C1 = (A, B, E, A) elementary cycle of length 3.

C2 = (A, B, C, F, E, D, A) elementary cycle of length 5.

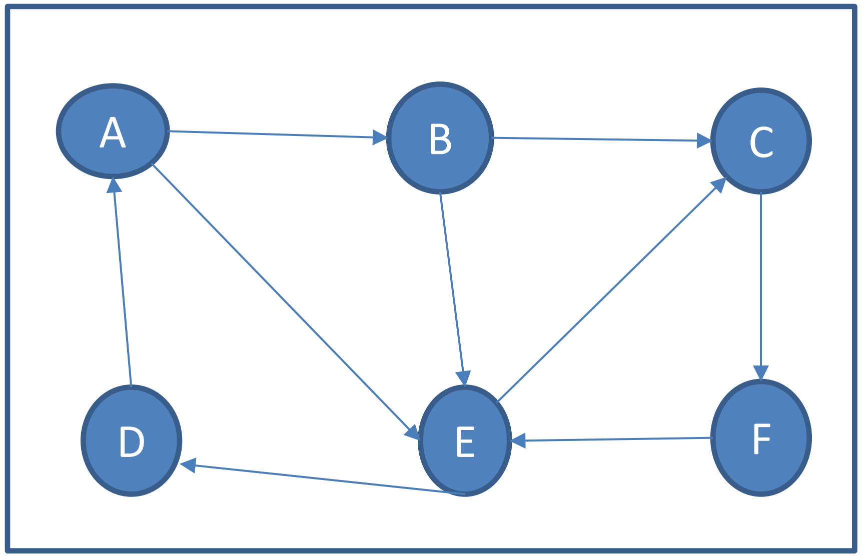
C3 = (D, A, E, F, C, E, D) cycle of length 6 that is not elementary.

C4 = (A, B, E, C, **B, E**, A) is not a cycle: the edge (B, E) is counted twice.

2.1 Paths in a graph

2.1.2 Cycle:

- We call a **Hamiltonian cycle** of a graph a cycle that passes exactly once through each of the vertices of the graph. C1 = (A, B, C, F, E, D, A) is a Hamiltonian cycle.
- A **graph is Hamiltonian** if it has a Hamiltonian cycle.
- We call an **Eulerian cycle** of a graph G a cycle that passes exactly once through each of the edges of G.
- A **graph is said to be Eulerian** if it has an Eulerian cycle.



The **distance** between two vertices in a graph G is the **length** of the **shortest** path connecting them.
Example: Distance between (A, F) = 2.

The **diameter** of a graph G is the **maximum** of the **distances** between its different vertices.
Diameter(G) = 2 ?? To be verified.

A graph without cycles is called **acyclic**.

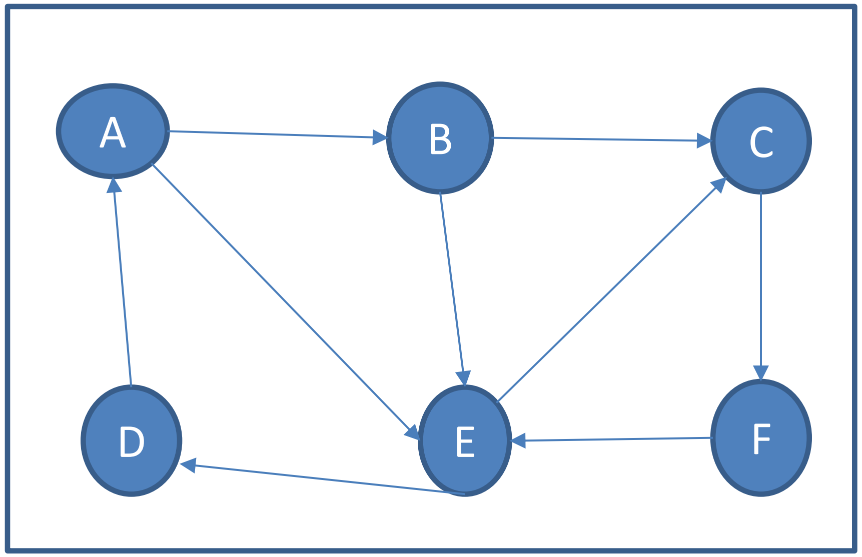
2.1 Paths in a graph

2.1.3 Path:

A path(a way) from vertex V0 to Vk in a graph G is a sequence of vertices connected successively by **directed edges** in the **same direction**.

Example: (A, E, C)

A path is called **elementary** if none of the **vertices** that make up the path appear more than once.



A path is called **simple** if none of the **edges** that make up the path appear more than once.

An **Eulerian path** is a simple path that passes exactly **once** through **each** edge of the graph.

A **Hamiltonian path** is a path that passes exactly **once** through each **vertex** of the graph.

2.1 Paths in a graph

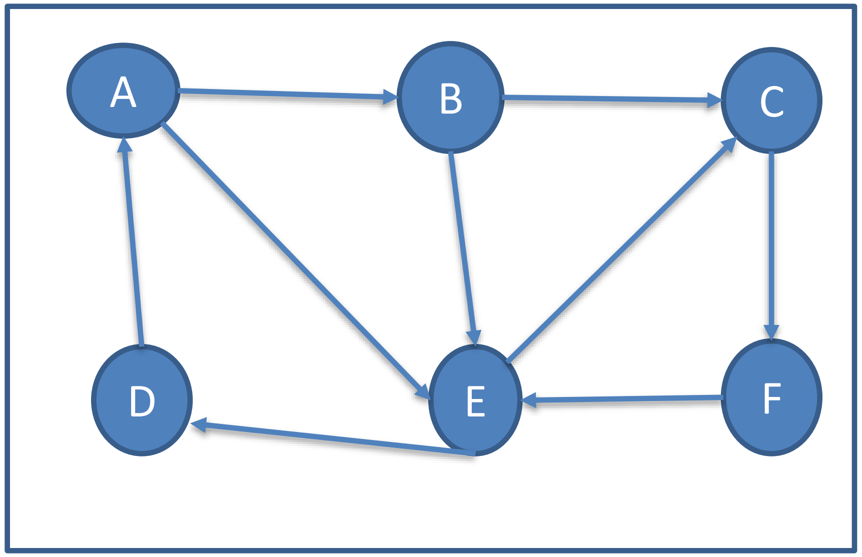
2.1.4 Circuit:

A **circuit** is a path in which the initial and final endpoints coincide.

Examples:

The sequence (A, B, C, F): elementary path.

The sequence (A, B, C, F, E): elementary path.

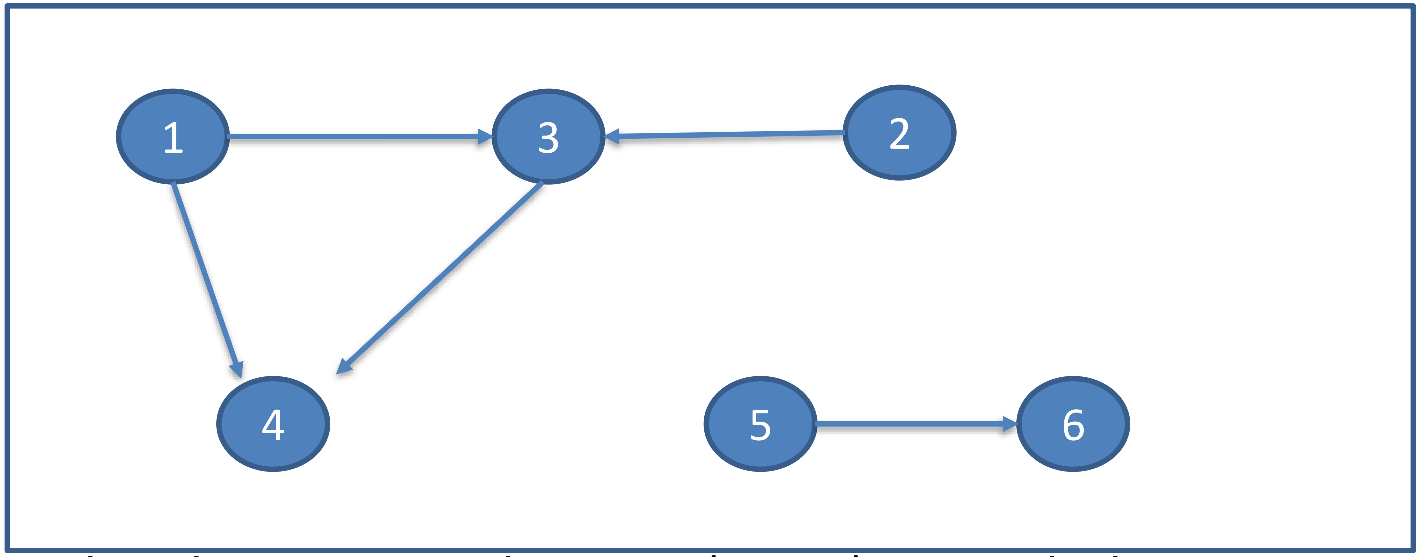


The sequence (A, B, E, A): elementary cycle.
The sequence (A, B, E, D, A): elementary circuit.
The sequence (A, B, C, F, E, D, A): Hamiltonian circuit.

2.2 Connectivity in a graph

2.2.1 The concept of connectivity:

Two vertices x and y have a connectivity relationship if and only if there exists a path(chain) between x and y



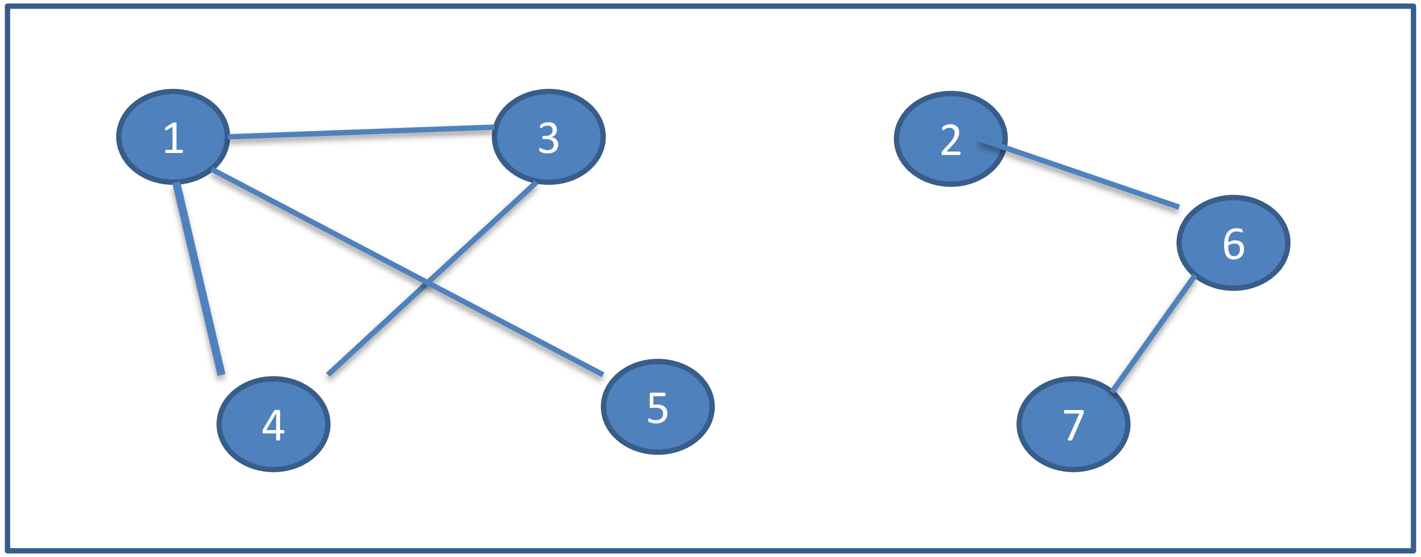
There exists a chain between 1 and 2 ⇒ C = (1, 3, 2) ⇒ 1 and 2 have a connectivity relationship.

There is no relationship between 1 and 5 ⇒ 1 and 5 do not have a connectivity relationship.

2.2 Connectivity in a graph

2.2.2 Connected graph

An undirected graph is connected if every vertex is reachable from any other vertex.



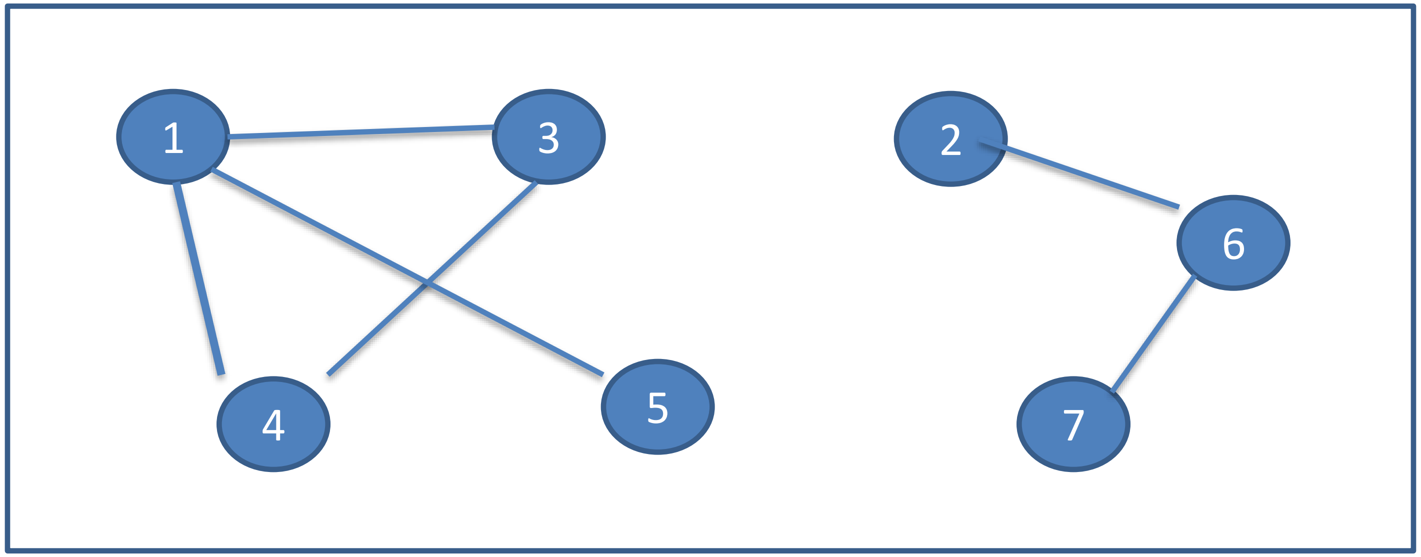
The graph is not connected.

Note: If we add an edge between 5 and 7, the graph becomes connected.

2.2 Connectivity in a graph

2.2.3 Connected component:

We call a **connected component** a set of vertices that are pairwise connected.

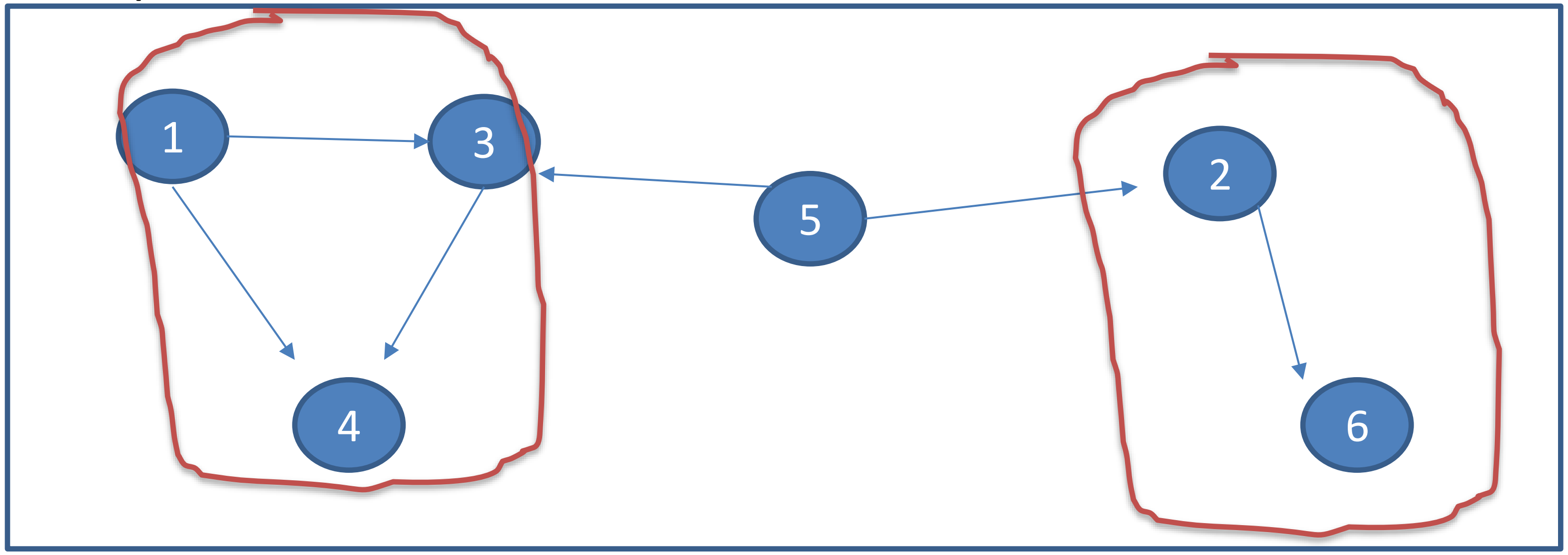


- The vertices (1, 3, 4, 5) form the first connected component.
- The vertices (2, 6, 7) form the second connected component.

2.2 Connectivity in a graph

2.2.4 Articulation point:

An **articulation point** is a vertex whose removal increases the number of connected components.



An **isthmus** is an edge whose removal has the same effect. Vertex 5 is an articulation point. Edge (5,2) is an isthmus.

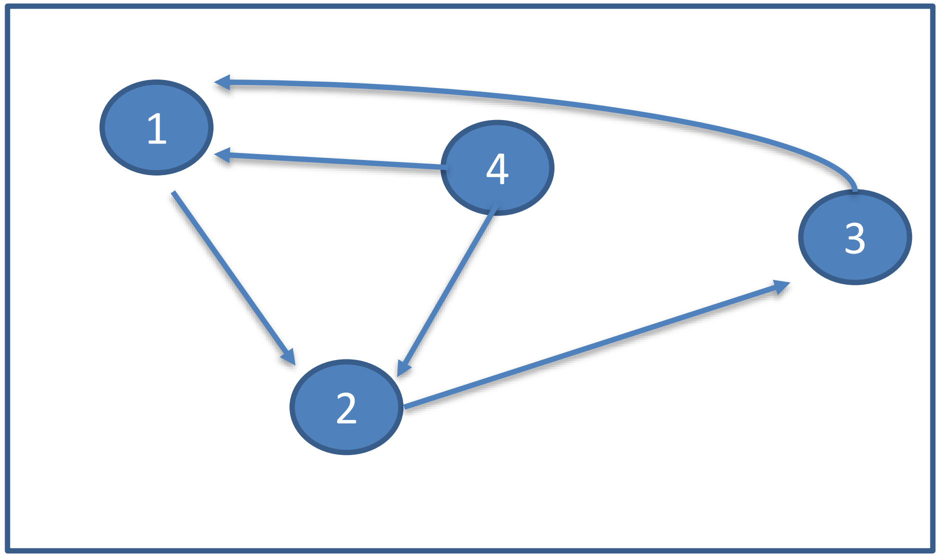
2.2 Connectivity in a graph

2.2.5 Strongly connected graph

We talk about **strong connectivity** in directed graphs.

Strong connectivity in a graph is defined as the relationship between two vertices as follows:

x and y are strongly connected ⇔ there exists a **path** from x to y **and** a **path** from y to x



Vertices 1 and 3 have a strong connectivity relationship.
We have a path from 1 to 3 and a path from 3 to 1.

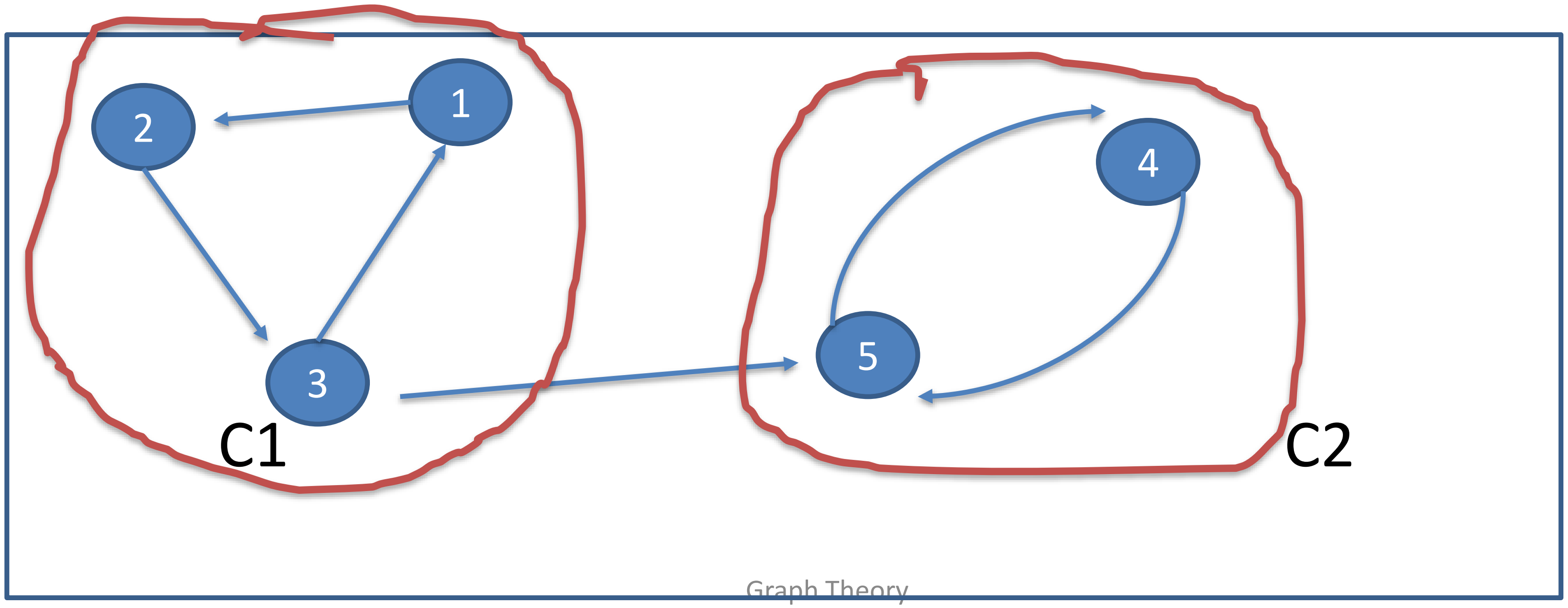
Vertices 4 and 3 do not have a strong connectivity relationship. There is a path from 4 to 3, but there is no path from 3 to 4

2.2 Connectivity in a graph

b) Strongly connected component:

We call a **strongly connected component** a set of vertices that are pairwise strongly connected.

- The set {1, 2, 3} forms a strongly connected component C1.
- The set {4, 5} forms another strongly connected component C2.
- The vertices in C1 do not have a strong connectivity relationship with the vertices in C2.



2.2 Connectivity in a graph

2.2.5 Strongly connected graph

c) Application:

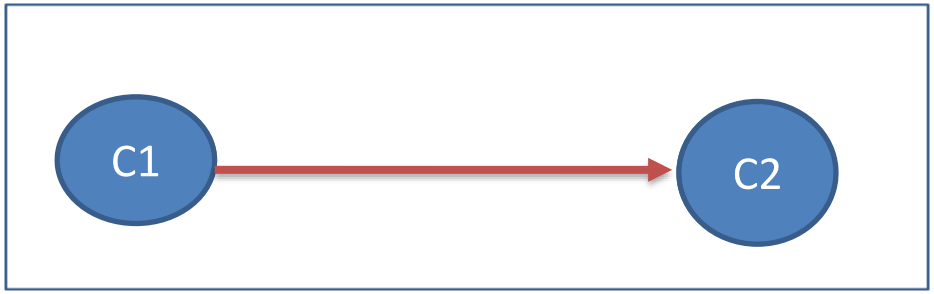
Traffic plan in a city, it is interesting to know if one can move from or to any point. One way to verify is to take the plan and check if it is strongly connected or not.

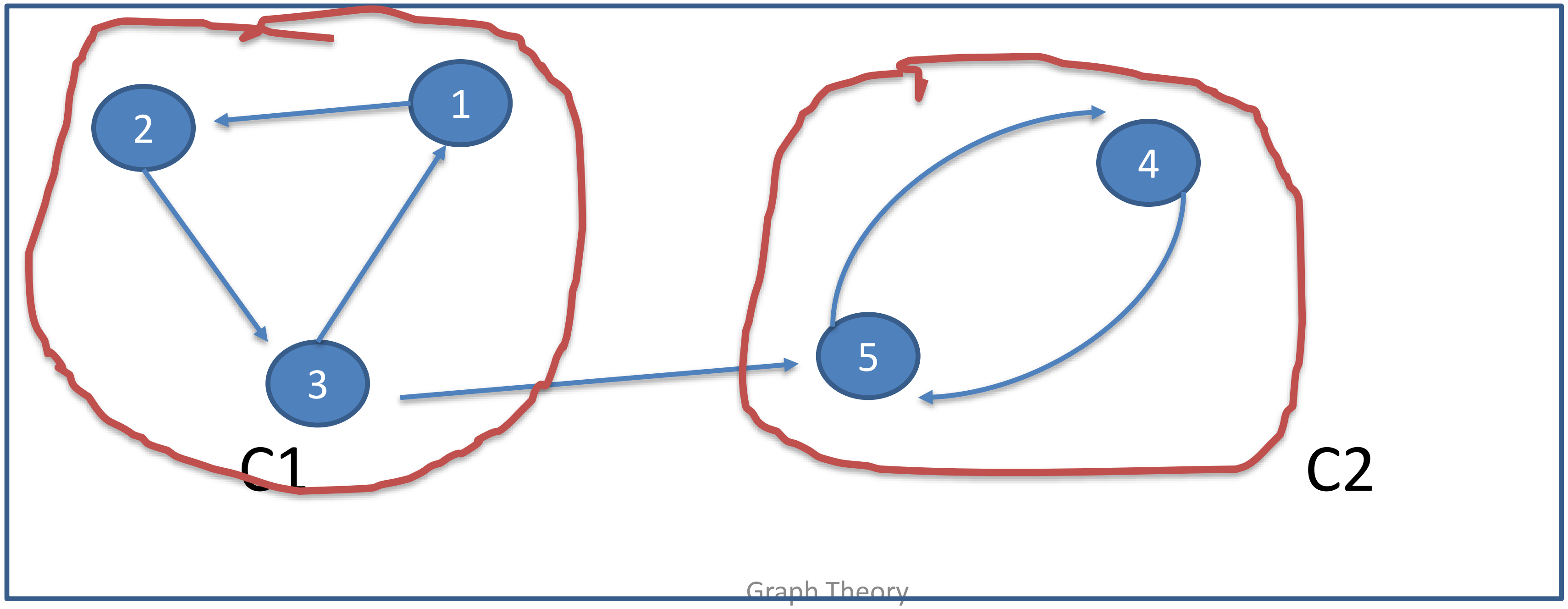
2.2 Connectivity in a graph

2.2.6 Reduced graph:

The vertices are represented by the strongly connected components.

Reduced graphe Gr





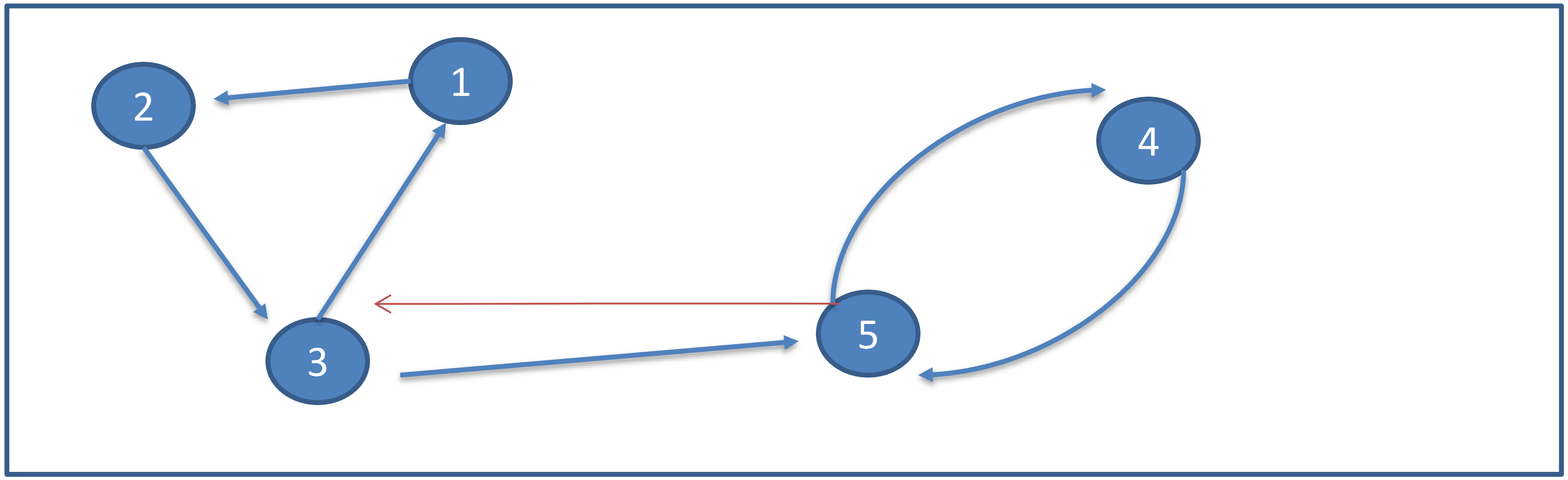
2.2 Connectivity in a graph

2.2.7 Strongly connected graph:

A graph G is said to be strongly connected if all its vertices are pairwise strongly connected, in other words, G contains a single strongly connected component.

This graph is not strongly connected; it contains 2 strongly connected components.

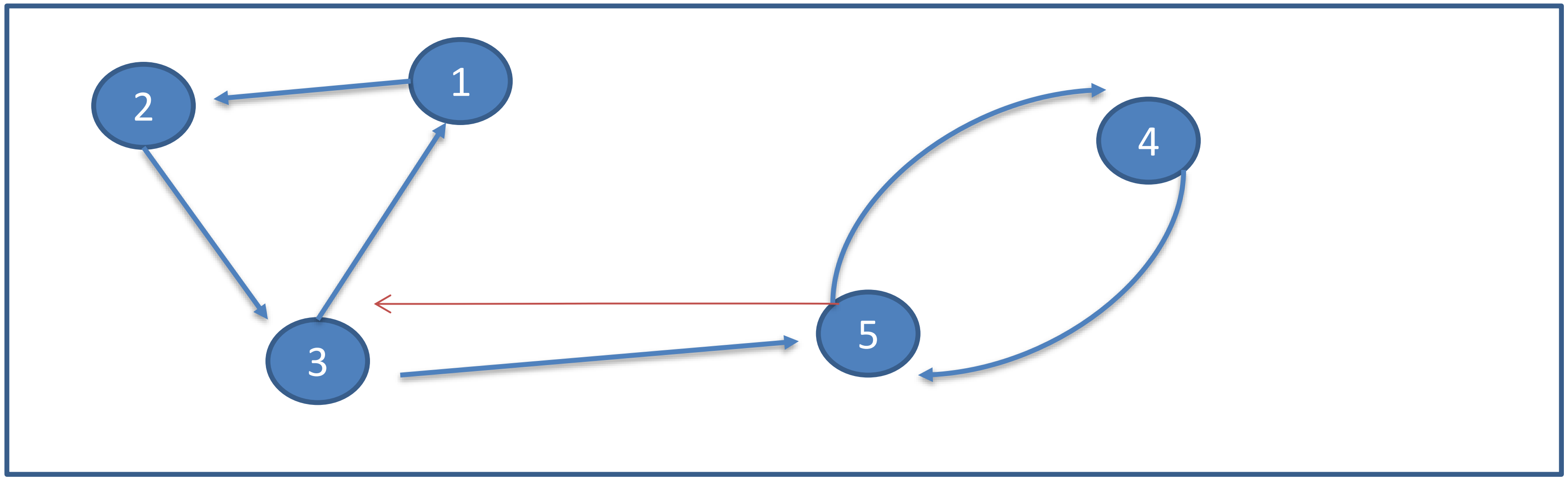
If we add an edge (5,3), it becomes a strongly connected graph



2.2 Connectivity in a graph

Vertex connectivity, denoted by the Greek letter kappa k), is the minimum number of vertices that must be removed to disconnect a graph. A graph is considered k-vertex-connected if its vertex connectivity is at least k.

The vertex connectivity of a complete graph Kn is n-1 because all but one vertex must be removed to make it trivial. (reduce it to a single vertex)



2.2 Connectivity in a graph

2.2.8 Search for strongly connected components:

Algorithm: Marking

Data: G=(V,E), directed graph

Results: the number K of strongly connected components and the list C1, C2, ..., Ck of strongly connected components

Step 0: Initialization $K \leftarrow 0$ $W \leftarrow V$

Step 1:

- Choose a vertex from W and mark it with a + and -
- Mark all direct and indirect successors with +
- Mark all direct and indirect predecessors with –
- Set K = K + 1 and Ck as the set of vertices marked with + and –
- Remove the vertices in Ck from W (W = W - Ck) and clear all marks
- **If** W = Ø **then** finish and proceed to **step 2**, otherwise go to **step 1**

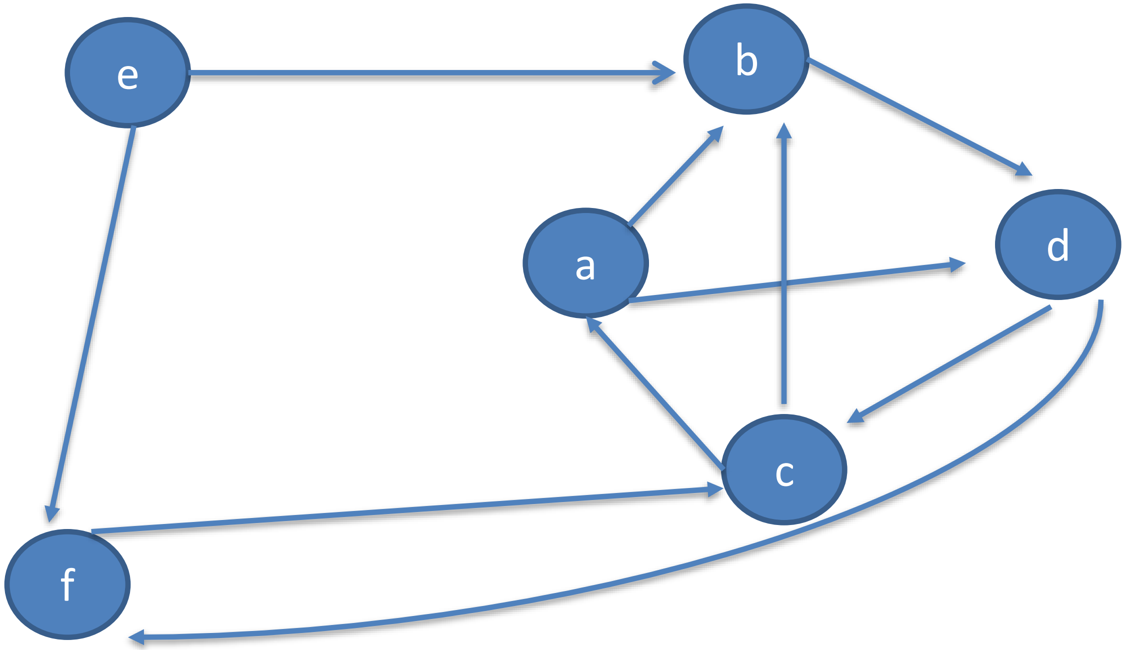
Step 2:

K is the number of strongly connected components, and each Ci is a component.

2.2 Connectivity in a graph

2.2.8 Search for strongly connected components: Example

0 : Initialisation: K = 0 W = V= {a,b,c,d,e,f}



1: We choose 'a', calculate the successors and predecessors of 'a', marked with + and -, we find C1 = {a, b, c, d, f}, and W = W - C1 = {e}.

W ≠ Ø, proceed to step 1 and continue.

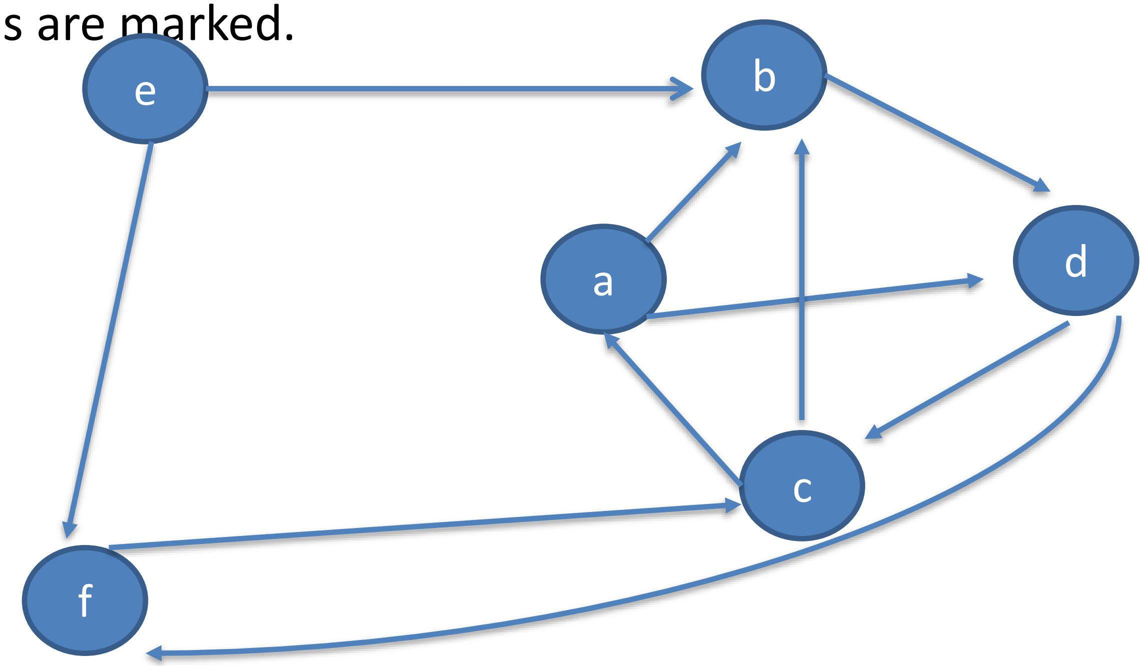
2.2 Connectivity in a graph

2.2.8 Search for strongly connected components: Example

Iteration 2: We mark vertex 'e' with + and -, so C2 = {e}.

No other vertices are marked.

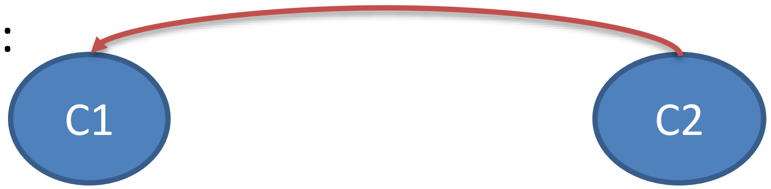
G:



W = W - C2 = ∅, so finish, the number of strongly connected components = 2

C1 = {a, b, c, d, f} and C2 = {e}. The graph G is associated with a reduced graph Gr represented by:

Gr :



2.2 Connectivity in a graph

2.2.9 Graph Scheduling:

This is the arrangement of a connected graph.

Principle: Scheduling a graph involves arranging its vertices in a specific order such that the edges follow the same direction.

Different levels of vertices are defined.

We schedule a graph to organize and plan tasks, menage resources and visualize project progress …

Algorithm: Graph Scheduling

Data: A connected directed graph G=(V,E)

Results: The different **levels** of vertices and the **scheduled graph**

2.2 Connectivity in a graph

2.2.9 Graph Scheduling: (Algorithm Continuation)

(0) Determine the dictionary of predecessors for G formed by (V, $\Gamma^{-}(x)$).

(1) Identify the vertices in $\Gamma^{-}(x)$ that have no predecessors. $\Gamma^{-}(x) = \emptyset$.

(1.1) Define N_0 as the set of vertices in G with no predecessors; this set is called **level 0.**

(1.2) Cross out in the column of $\Gamma^{-}(x)$ all the vertices from level 0, resulting in a new column $\Gamma_1(x)$ and the subgraph G_1 with vertices V/N_0.

(2) Identify in the column Γ-(x) the vertices with no predecessors; Γ-(x) = ∅.

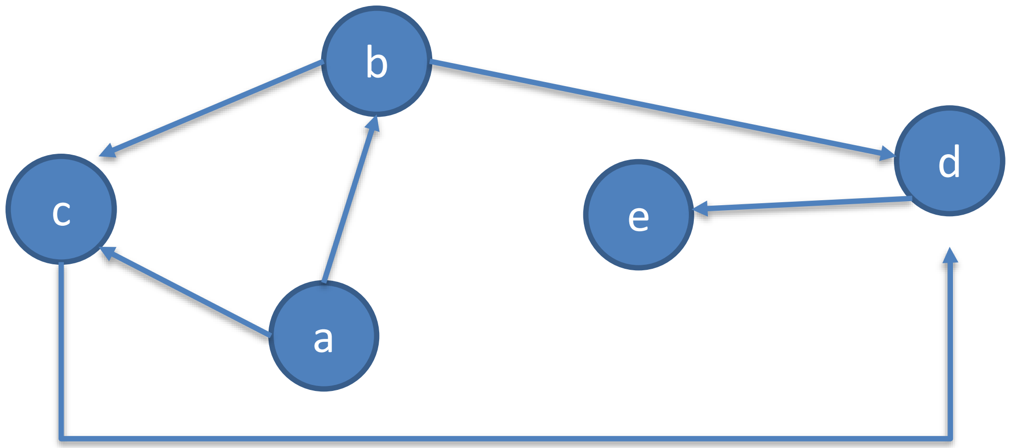
(2.1) Define N1 as the set of vertices in the graph with no predecessors.

(2.2) Cross out in the cell of Γ1(x) all the vertices at level N1, resulting in a new column Γ2(x) and the subgraph G2 generated by V/(N0 ∪ N1).

Continue this process until you complete the graph, and represent the ordered graph by levels of G.

2.2 Connectivity in a graph

2.2.9 Graph Scheduling: (example)



V	$\Gamma^-(x)$
a	∅
b	~~a~~
c	~~a~~,b
d	c,b
e	d

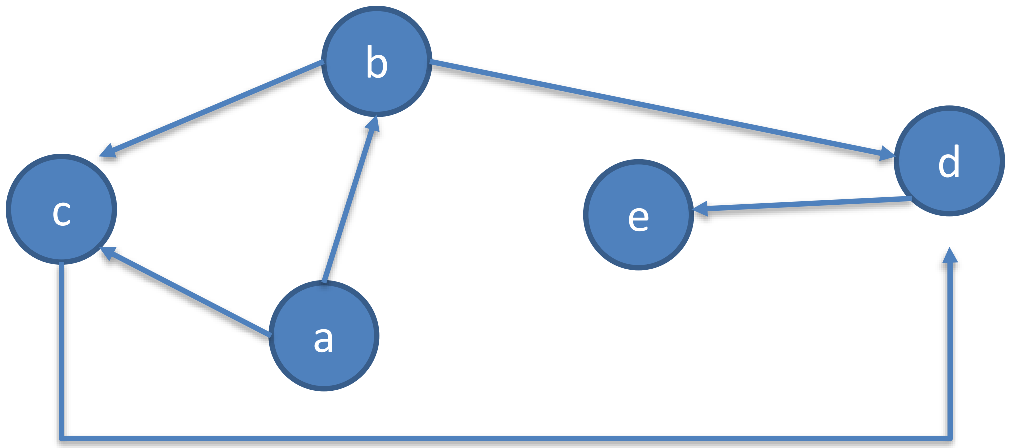
a: has no predecessor
N0={a}
Cross out vertex 'a' in Γ-(x)
And we obtain Γ-1(x)

V	$\Gamma^- 1(x)$
a	/
b	∅
c	~~b~~
d	c,~~b~~
e	d

b: has no predecessor
N1={b}
Cross out vertex 'b' in Γ-1(x)
And we obtain Γ-2(x)

2.2 Connectivity in a graph

2.2.9 Graph Scheduling: (example)



V	$\Gamma^{-}2(x)$
a	/
b	/
c	Ø
d	~~c~~
e	d

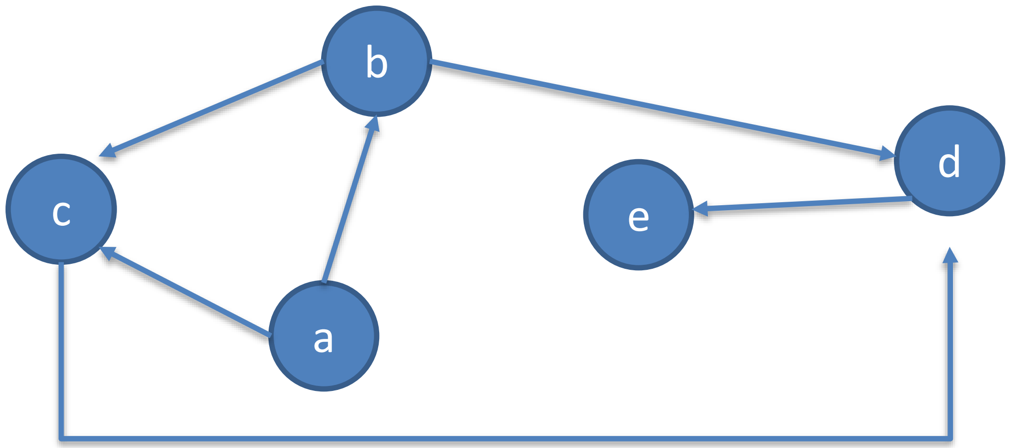
c: has no predecessor
N2={c}
Cross out vertex 'c' in Γ-2(x)
And we obtain Γ-3(x)

V	$\Gamma^{-}3(x)$
a	/
b	/
c	/
d	Ø
e	~~d~~

d: has no predecessor
N3={d}
Cross out vertex 'd' in Γ-3(x)
And we obtain Γ-4(x)

2.2 Connectivity in a graph

2.2.9 Graph Scheduling: (example)



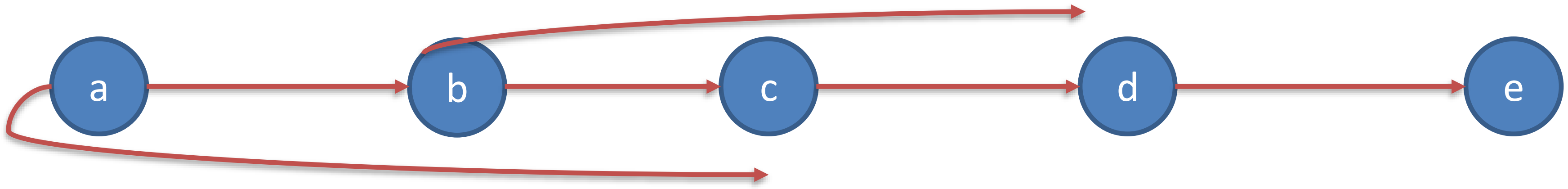
V	Γ^{-} 4(x)
a	/
b	/
c	/
d	/
e	Ø

e: has no predecessor
N4={e}

We have examined all the vertices of the graph, and thus, we have the ordered graph:
N0={a}, N1={b}, N2={c}, N3={d}, N4={e}

2.2 Connectivity in a graph

2.2.9 Graph Scheduling: (example)



N0	N1	N2	N3	N4

This is an ordered graph

2.2 Connectivity in a graph

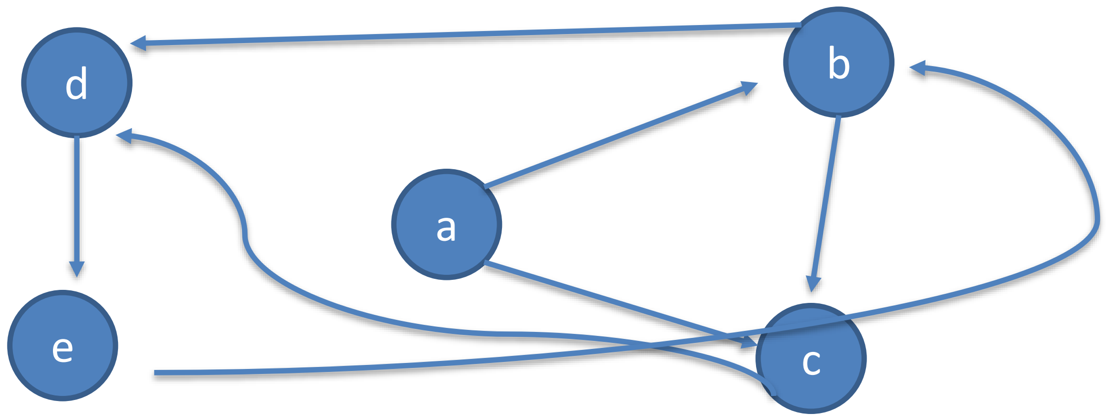
2.2.10 Searching for a circuit in a connected graph:

At a certain stage of scheduling a graph, the definition of levels becomes stuck (there are no vertices without predecessors).

➔ Consequently, ordering the graph is impossible, and we say that the graph contains a circuit.

2.2 Connectivity in a graph

2.2.10 Searching for a circuit in a connected graph: (application)



V	$\Gamma^{-}(x)$
a	∅
b	~~a~~,e
c	~~a~~,b
d	c,b
e	d

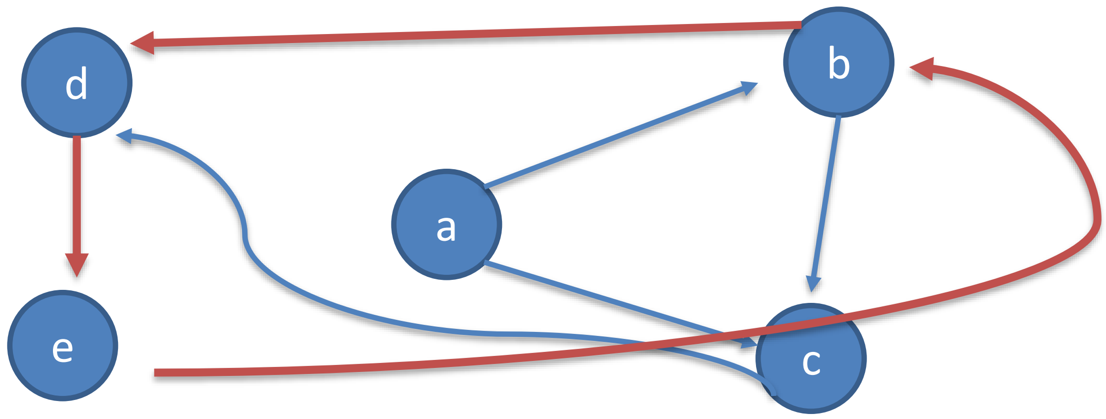
a: has no predecessor
N0={a}
We cross out vertex 'a' in Γ-(x)
And we obtain Γ1(x)

V	$\Gamma^{-}1(x)$
a	/
b	e
c	b
d	c,b
e	d

We cannot determine level N1; there are no vertices without predecessors.
⇨ G cannot be ordered.
⇨ G contains a **circuit**.

2.2 Connectivity in a graph

2.2.10 Searching for a circuit in a connected graph: (application)



V	Γ⁻ 1(x)
a	/
b	e
c	b
d	c,b
e	d

To determine this circuit, we choose a vertex in Γ1(x). Let e be this vertex; it leads us back to a line in V, and we stop because the vertex e has repeated.

⇨ The sequence (e, d, b, e)

⇨ The reverse sequence forms a circuit (e, b, d, e).

Chapter 3 : Remarkable Paths

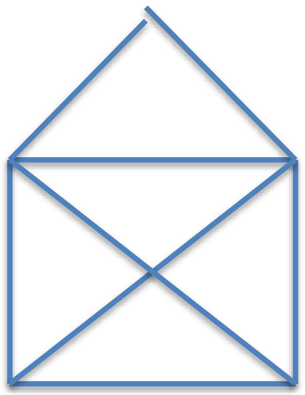
1- Eulerian paths
2- Hamiltonian paths
3- Graph Coloring

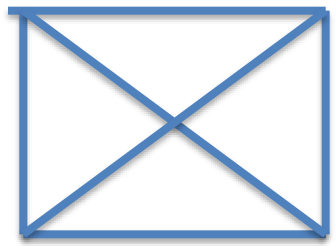
3.1 Eulerian Paths

<u>3.1.1 Eulerian Cycles and chains in a Multigraph</u>:

We ask two questions?

<u>Question 1:</u> Is it possible to draw these drawings without lifting the pencil or retracing any line?





After several attempts, we can make the first drawing; the second one is impossible.

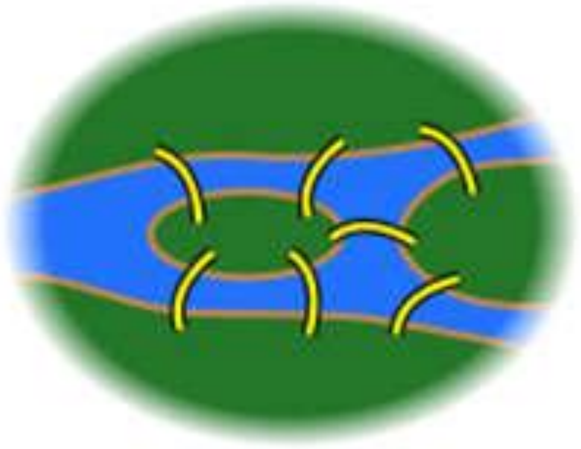
3.1 Eulerian Paths

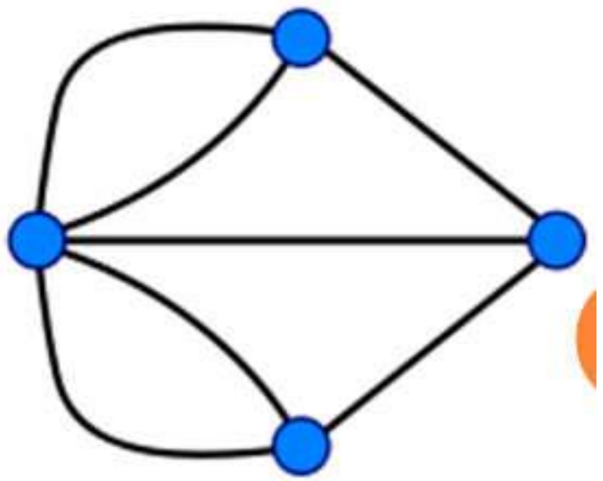
<u>3.1.1 Eulerian Cycles and chains in a Multigraph:</u>

We ask two questions?

<u>Question 2</u>: The inhabitants of the city of Königsberg were seeking to determine if it was possible to find a route that crosses each bridge once and only once.

The problem is to find an Eulerian path or cycle.





Definition: An Eulerian chain(cycle) is a chain (cycle) that uses each edge exactly once.

3.1 Eulerian Paths

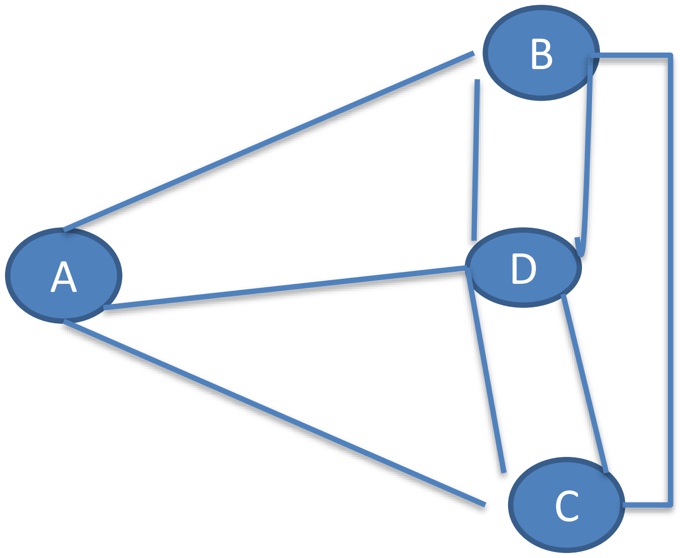
<u>3.1.1 Eulerian Cycles and chains in a Multigraph:</u>

a) Eulerien chain:

Let G = (V, E) be a multigraph.

A multigraph G has an Eulerian chain connecting the vertices V_0 and V_k of V if and only if G is **connected**, and **all its** vertices have **even degrees** except for V_0 and V_k.

Example 1:



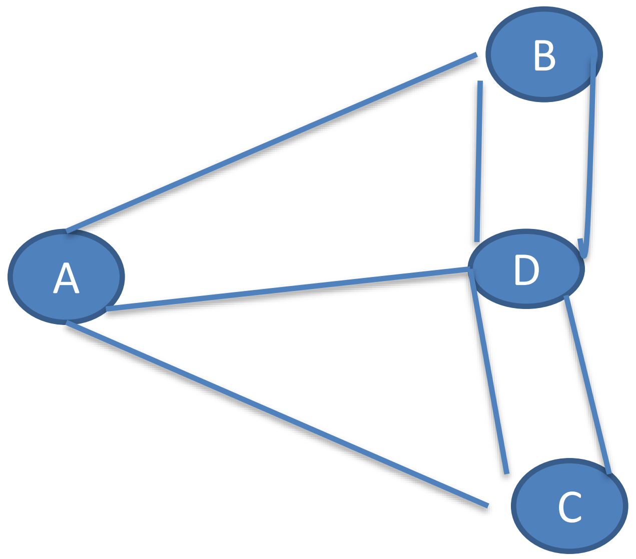
d(A)=3
d(B)=4
d(C)=4
d(D)=5
An Eulerian path connecting vertices A and D is defined as follows:
C=(A,B,C,A,D,B,D,C,D)

3.1 Eulerian Paths

<u>3.1.1 Eulerian Cycles and chains in a Multigraph:</u>

a) Eulerien chain:

Example 2:



d(A)=3
d(B)=3
d(C)=3
d(D)=5
All vertices have odd degrees
→ G does not have any Eulerian chain.

3.1 Eulerian Paths

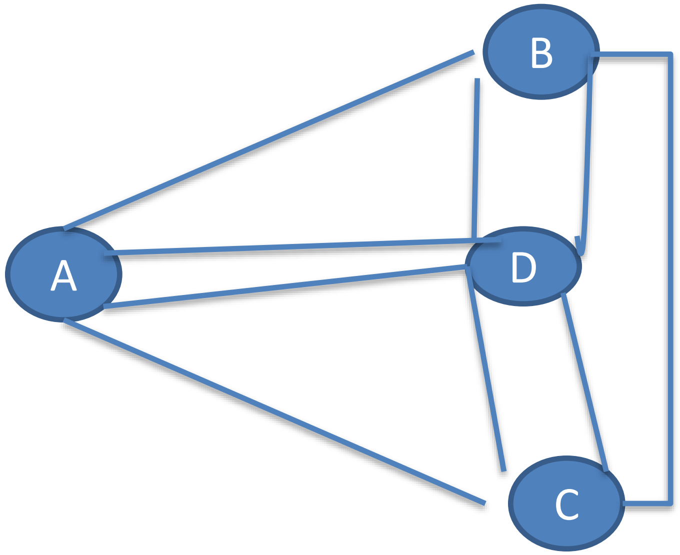
3.1.1 Eulerian Cycles and chains in a Multigraph:

b) Eulerien Cycles:

Let G = (V, E) be a multigraph.

A multigraph G has an **Eulerian cycle** if and only if G is **connected**, and **all its** vertices have **even degrees**.

Exemple :



d(A)=4
d(B)=4
d(C)=4
d(D)=6
All degrees are even → there exists a cycle.
C=(A,B,D,C,B,D,A,C,D,A)

3.1 Eulerian Paths

3.1.2 Eulerian Paths and circuits in a Graph:

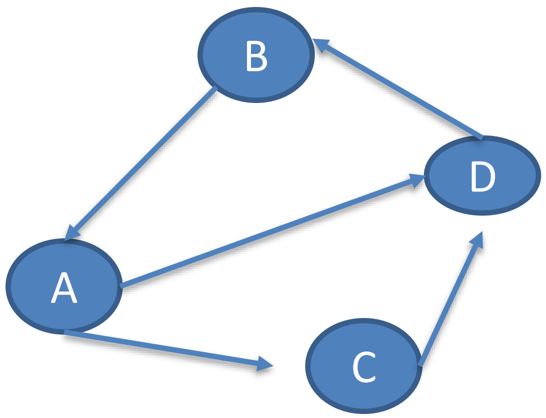
a) Eulerien paths:

Let G = (V, E) be a directed graph.

A directed graph G has an Eulerian path connecting vertices X_0 and X_k in V if and only if: 1- G is connected

2- $d^+(X_0) = d^-(X_0)+1 \qquad d^+(X_k) = d^-(X_k)-1$

3- $d^+(X) = d^-(X) \quad \forall \ X \in V / (X_0,X_k)$



G is connected

$d^+(A) = d^-(A)+1 \qquad 2 = 1 + 1$

$d^+(D) = d^-(D)-1 \qquad 1 = 2 - 1$

$d^+(B) = d^-(B) \quad = 1$

$d^+(C) = d^-(C) \quad = 1$

➔ G has an Eulerian path between A and D

C=(A,C,D,B,A,D)

3.1 Eulerian Paths

3.1.2 Eulerian Paths and circuits in a Graph:

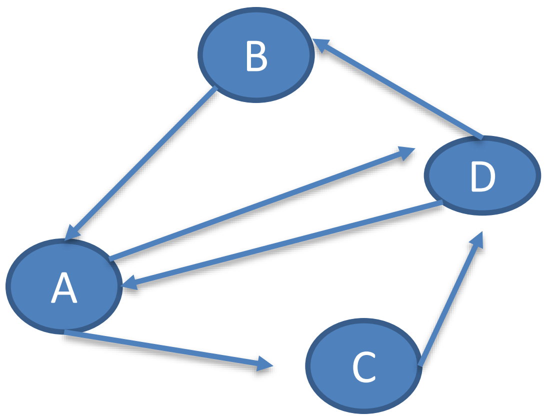
d) Eulerien Circuits:

Let G = (V, E) be a directed graph.

A directed graph G has an Eulerian circuit if and only if:

1- G is connected

2- $d^+(X) = d^-(X) \quad \forall \; X \in V$



G is connected

$d^+(A) = d^-(A) \quad = 2$

$d^+(B) = d^-(B) \quad = 1$

$d^+(C) = d^-(C) \quad = 1$

$d^+(D) = d^-(D \quad) \quad = 2$

➔ G has an Eulerian circuit.

➔ C=(A,D,B,A,C,D,A)

3.2 Hamiltonian Paths

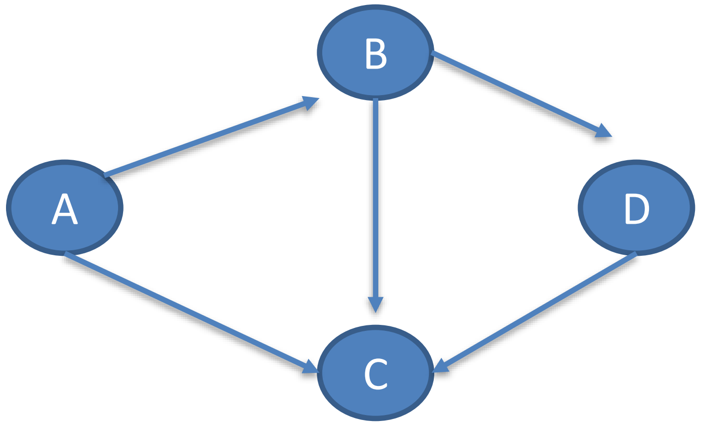
3.1.2 Hamiltonian paths of a graph:

a) Problem:

The traveling salesman problem consists of finding a route to visit all cities exactly once and return to the starting point.

In graph theory, it involves finding a Hamiltonian path with coinciding endpoints.

b) Definition: A path is said to be Hamiltonian if it passes through all the vertices exactly once.



Hamiltonian path : C=(A,B,D,C)

3.2 Hamiltonian Paths

<u>3.1.2 Hamiltonian paths of a graph:</u>

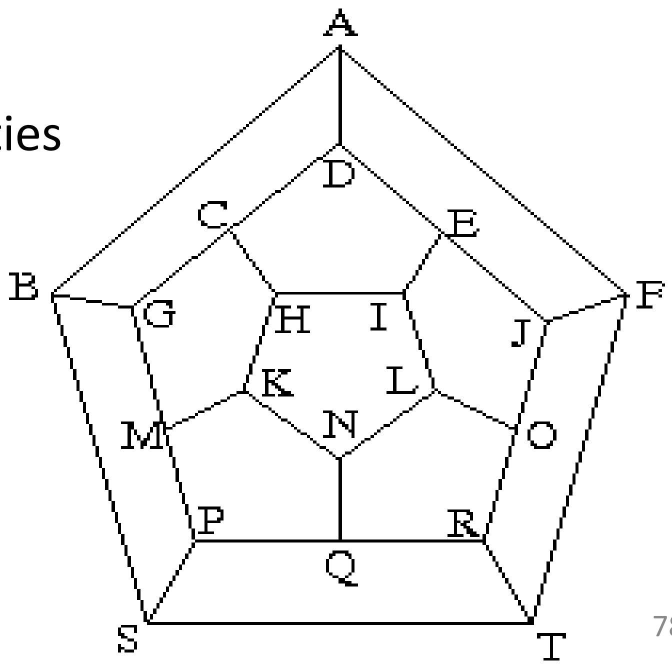
c) Algorithms:

It is not easy to verify whether a graph contains a Hamiltonian cycle, unlike the Eulerian cycle. This problem is NP-complete, which guarantees that there is no polynomial-time algorithm to solve it.

However, there are **necessary or sufficient conditions** for the existence of a Hamiltonian cycle

Example(traveling salesman problem):

for 20 (cities) vertices, ➔ $19!/2 = 6\ 10^{16}$ possibilities

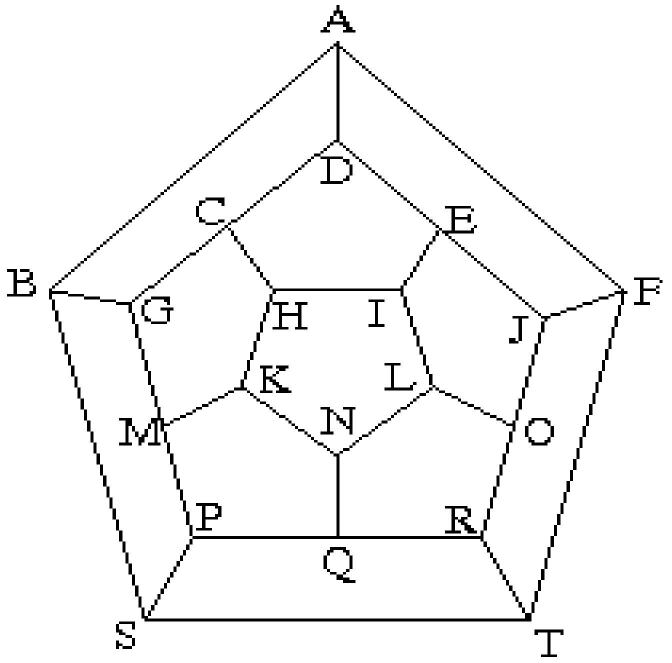


3.2 Hamiltonian Paths

3.1.2 Hamiltonian paths of a graph:

c-1)Dirac's Theorem (1952): For a simple graph G = (V, E) with N vertices, if the degree of every vertex is greater than or equal to N/2, then G contains a Hamiltonian cycle.

c-2)Ore's Theorem (1960): For a simple graph G = (V, E) with N vertices, if for every pair of non-adjacent vertices x and y, the inequality d(x) + d(y) ≥ N is satisfied, then G contains a Hamiltonian cycle.



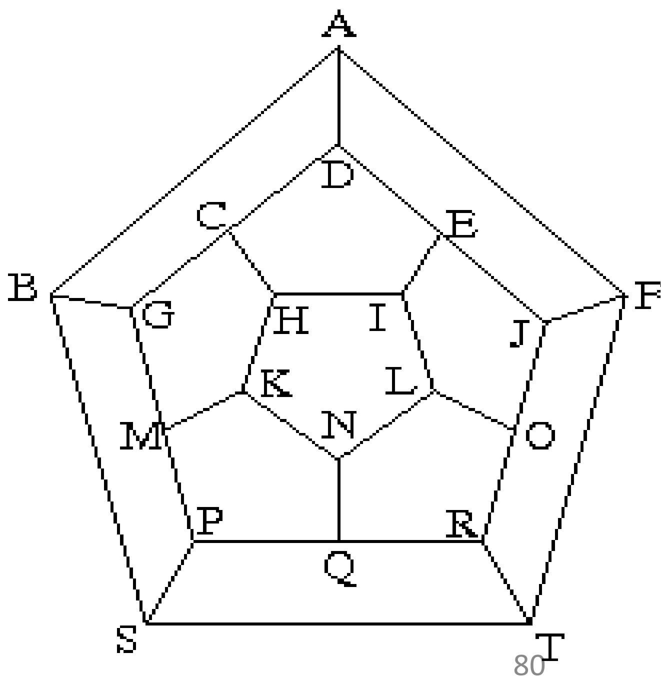
3.2 Hamiltonian Paths

<u>3.1.2 Hamiltonian paths of a graph:</u>

<u>c-3)The Pósa's Theorem</u>: A simple graph with n vertices, where $n \geq 3$, is Hamiltonian if:

1- For every integer k such that $1 \leq k < (n - 1)/2$, the number of vertices with a degree less than or equal to k is less than k.

2- The number of vertices with a degree less than or equal to $(n - 1)/2$ is less than or equal to $(n - 1)/2$.

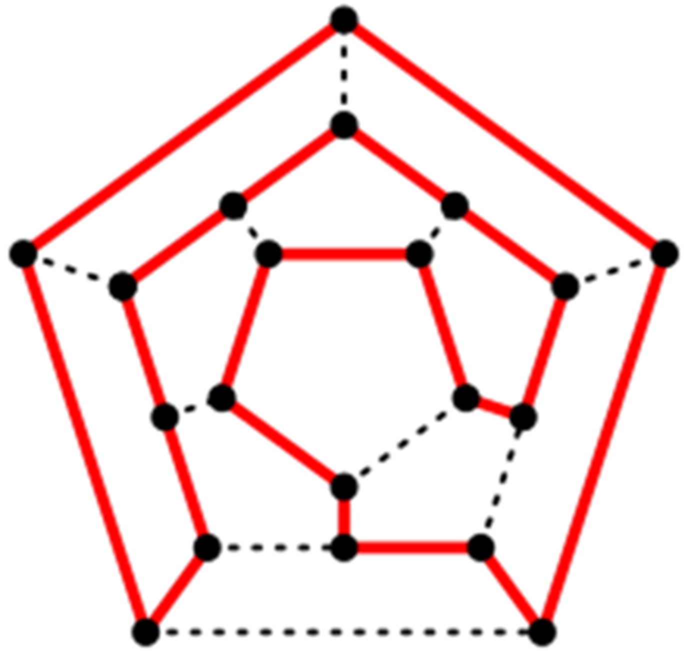


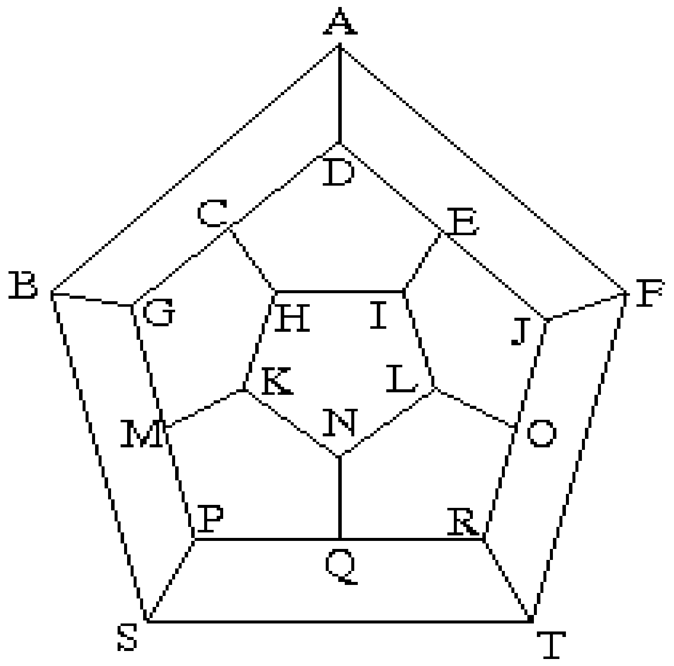
3.2 Hamiltonian Paths

3.1.2 Hamiltonian paths of a graph:

c-4)Chvátal-Erdős Theorem (1972):

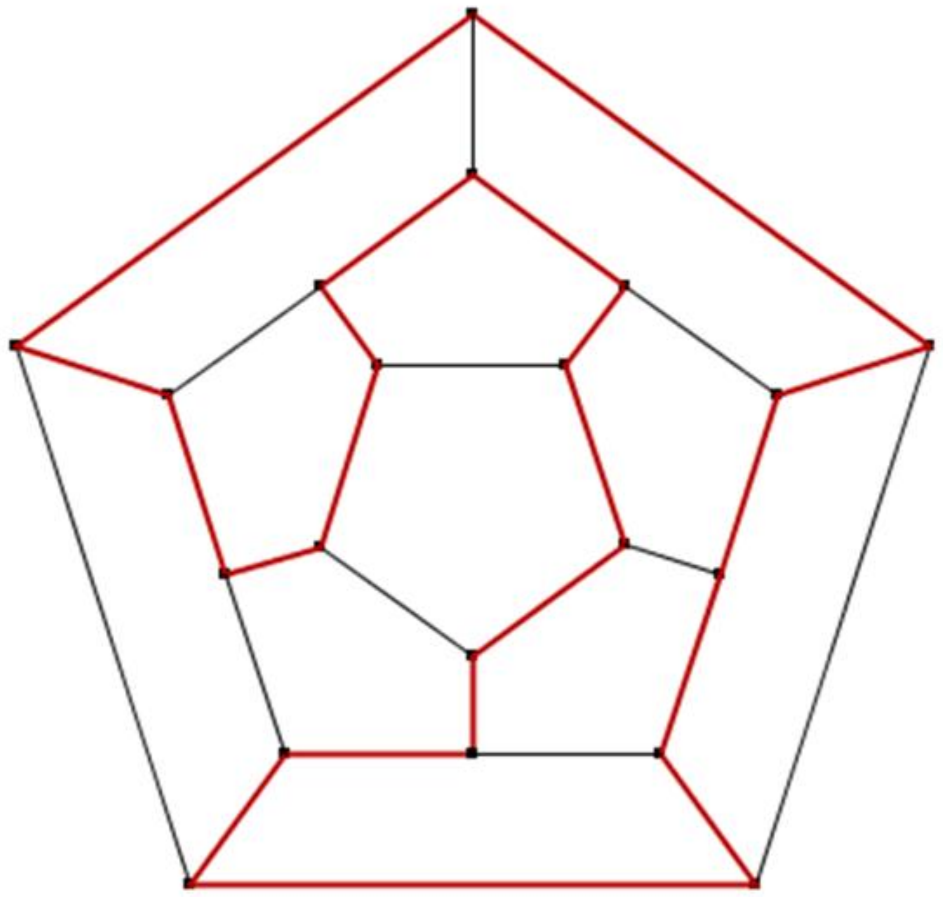
If the degree of each vertex in a simple graph G is at least n/2 (n is the number of vertices), and G is not bipartite, then G is Hamiltonian.





3.2 Hamiltonian Paths

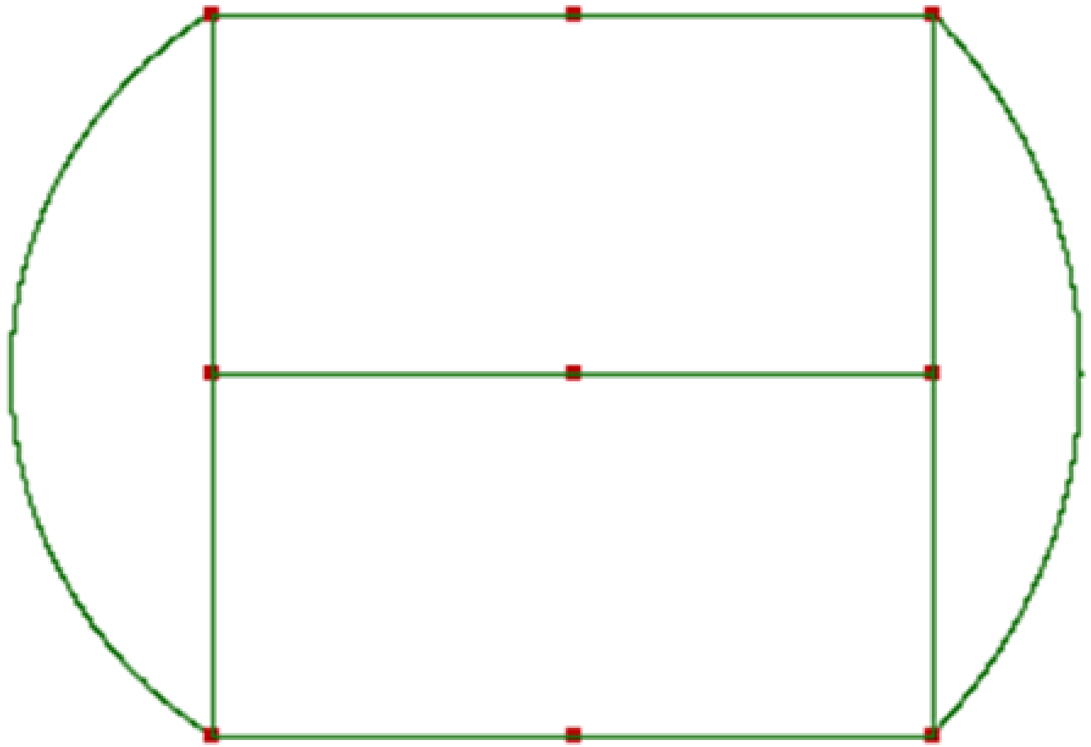
3.1.2 Hamiltonian paths of a graph:



Un graphe hamiltonien (un cycle hamiltonien est tracé en rouge)

3.2 Hamiltonian Paths

3.1.2 Hamiltonian paths of a graph:



Un graphe qui ne possède pas de cycle hamiltonien

3.2 Hamiltonian Paths

3.1.2 Hamiltonian paths of a graph:

- A complete graph with more than two vertices is Hamiltonian.

  The number of different Hamiltonian cycles on an **undirected complete** graph with n vertices is (n-1)! / 2,

  and on a **directed complete** graph with n vertices is (n-1)!

- For any **bipartite** graph G=(V,E) with V=XUY

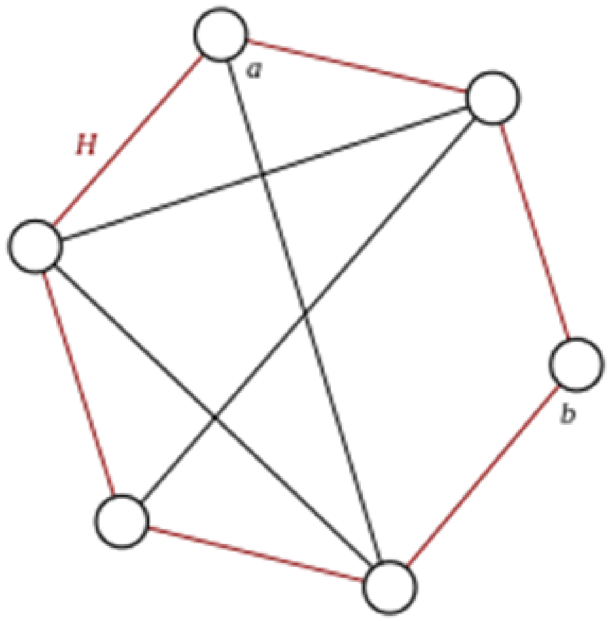
  If |X| ≠ |Y| then the graph cannot have a Hamiltonian cycle,

3.2 Hamiltonian Paths

3.1.2 Hamiltonian paths of a graph:

Example of application:

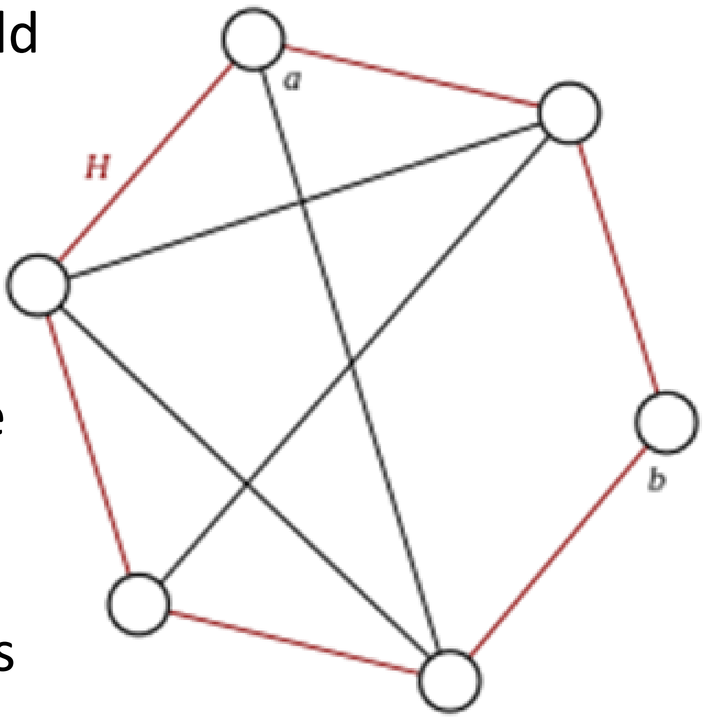
This graph has 6 vertices. It is Hamiltonian: the vertices have been ordered to highlight a Hamiltonian cycle, which is the red outer cycle H



3.2 Hamiltonian Paths

3.1.2 Hamiltonian paths of a graph:

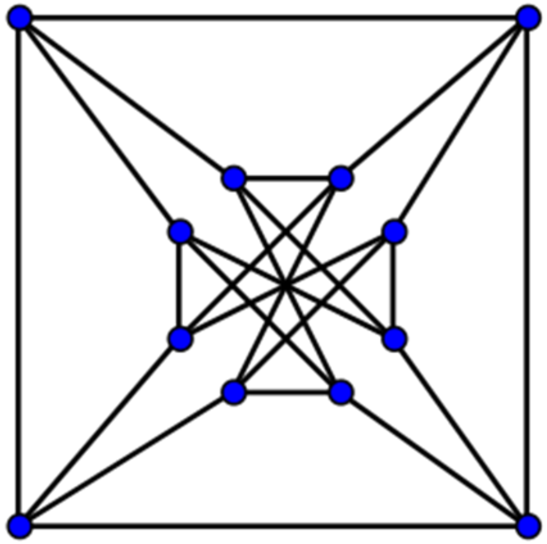
- **Dirac's theorem** does not prove that it is Hamiltonian. To do so, all vertices should have a degree of at least 3, but there is one vertex with a degree of 2.
- **Ore's theorem** is not more helpful because non-adjacent vertices a and b satisfy deg(a) + deg(b) = 3 + 2 = 5, while there should be at least 6.
- On the other hand, **Pósa's theorem** allows us to determine that the graph is Hamiltonian because there is 0 vertex with a degree of 1 and 1 vertex with a degree of 2: condition 1 is met (0 < 1), and 0 + 1 < 2.

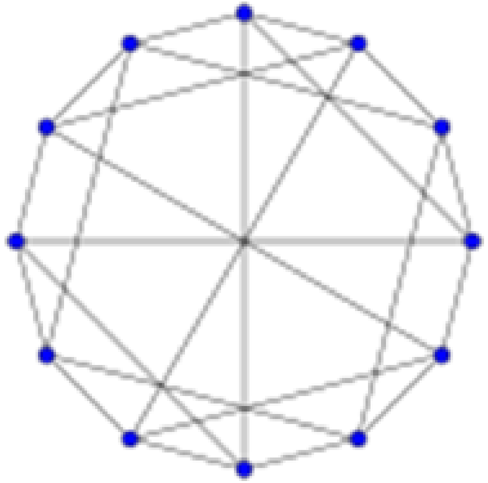


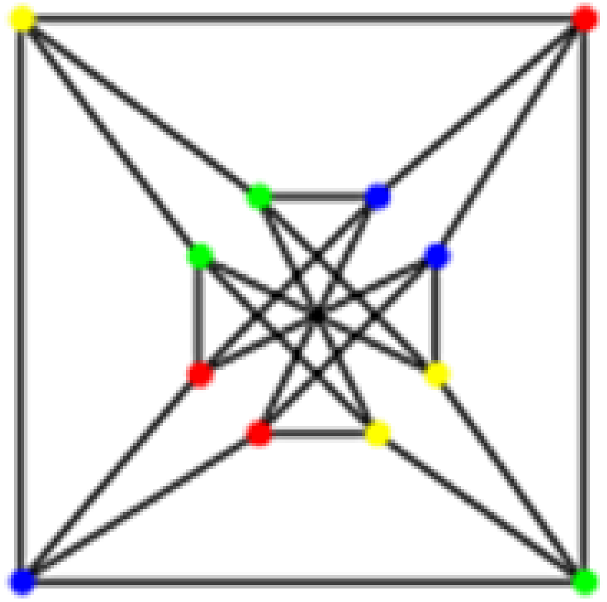
3.2 Hamiltonian Paths

3.1.2 Hamiltonian paths of a graph:

A Hamiltonian graph can contain multiple Hamiltonian cycles. For example, Chvátal's graph has 12 vertices, 24 edges, and 370 distinct Hamiltonian cycles.







3.2 Hamiltonian Paths

<u>3.1.2 Hamiltonian Path Algorithm: How it Works:</u>

The goal of this algorithm is to incrementally build paths and backtrack as soon as that a current path won't result in a Hamiltonian path.

<u>The algorithm in action:</u>

1- Begin at an arbitrary vertex.

2- Move to an adjacent, unvisited vertex, adding the edge to the path.

3- If all vertices have been visited, a Hamiltonian path has been found.

4- If no adjacent unvisited vertex can be found, backtrack to the previous vertex and try another path.

Note: we use a stack to save the previous vertices

3.2 Hamiltonian Paths

3.1.2 Hamiltonian paths of a graph:

d) Algorithms: Search for a Hamiltonian Path using degrees

Principle: Using the maximum out-degree (d^+)

Input: A graph G

Output: A Hamiltonian path in G

$W \leftarrow \emptyset$

(0): Choose a vertex with the maximum out-degree, let it be X_0. W = W U $\{X_0\}$

(1): Consider the subgraph G_1 generated by V - $\{X_0\}$ and choose a vertex with the maximum out-degree, X_1. W = W U $\{X_1\}$

(2): Consider the subgraph G_2 generated by V - $\{X_0, X_1\}$, and W = W U $\{X_0, X_1\}$

with **X_1 as a successor of X_0**.

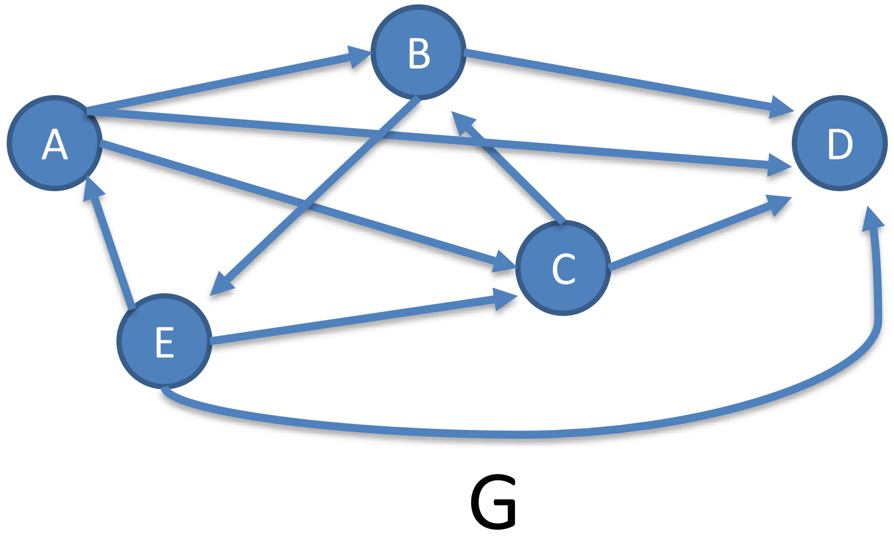
Continue this process until G is exhausted, obtaining a sequence of vertices in W that defines a Hamiltonian path C = $(X_0, X_1, \dots X_{k-1})$.

Note: It's possible to have multiple Hamiltonian paths.

3.2 Hamiltonian Paths

3.1.2 Hamiltonian paths of a graph:

d) Algorithms: Search for a Hamiltonian Path (Application)



(0)
d^+ (A)= 3
d^+ (B)= 2
d^+ (C)= 2
d^+ (D)= 0
d^+ (E)= 3

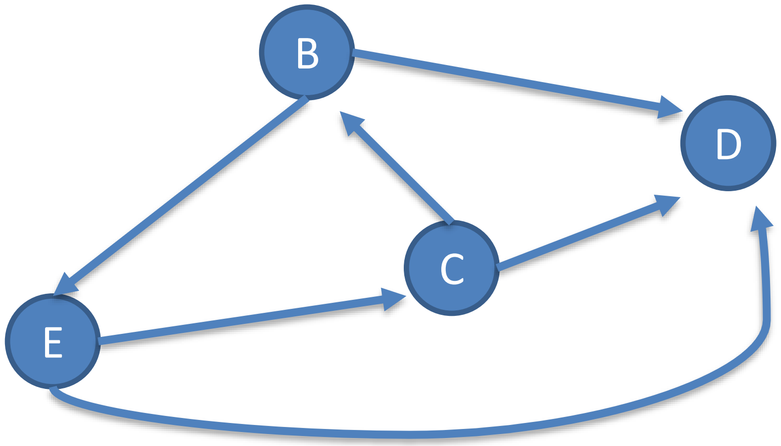
Max(3,2,2,0,3) = 3
We choose the vertex A or the vertex E
We choose A

X_0 = A et C=(A)

3.2 Hamiltonian Paths

<u>3.1.2 Chemins hamiltoniens d'un graphe</u>:

d) Algorithms: Search for a Hamiltonian Path (Application)



G1

d^+ (B)= 2
d^+ (C)= 2
d^+ (D)= 0
d^+ (E)= 2

Max(2,2,0,2) = 2
We choose les vertices B,C,E

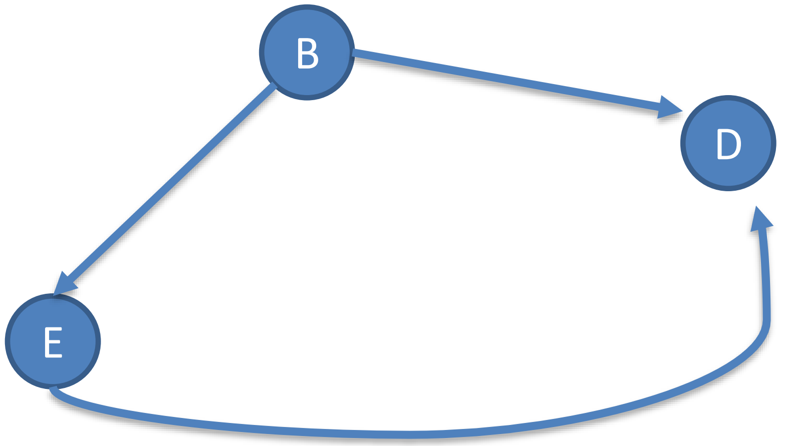
E is not successor of A
We choose B or C

X_1 = C et C=(A,C)

3.2 Hamiltonian Paths

3.1.2 Hamiltonian paths of a graph:

d) Algorithms: Search for a Hamiltonian Path (Application)



d^+ (B)= 2
d^+ (D)= 0
d^+ (E)= 1

Max(2,0,1) = 2
We choose le vertex B

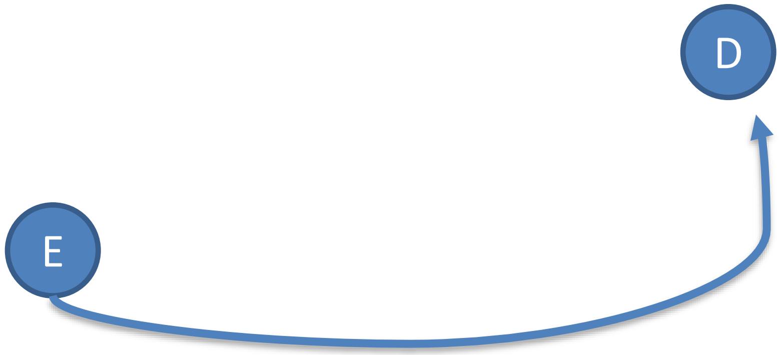
X_2 = B et C=(A,C,B)

G2

3.2 Hamiltonian Paths

3.1.2 Hamiltonian paths of a graph:

d) Algorithms: Search for a Hamiltonian Path (Application)



G3

d^+ (D)= 0
d^+ (E)= 1

Max(0,1) = 1
We choose the vertex E

X_3 = E et C=(A,C,B,E)

3.2 Hamiltonian Paths

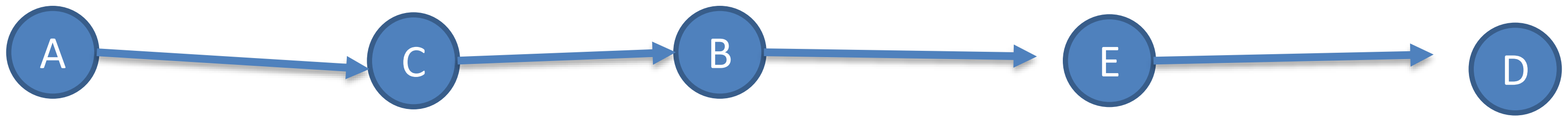
3.1.2 Hamiltonian paths of a graph:

d) Algorithms: Search for a Hamiltonian Path (Application)

X_4 = D et C=(A,C,B,E,D)

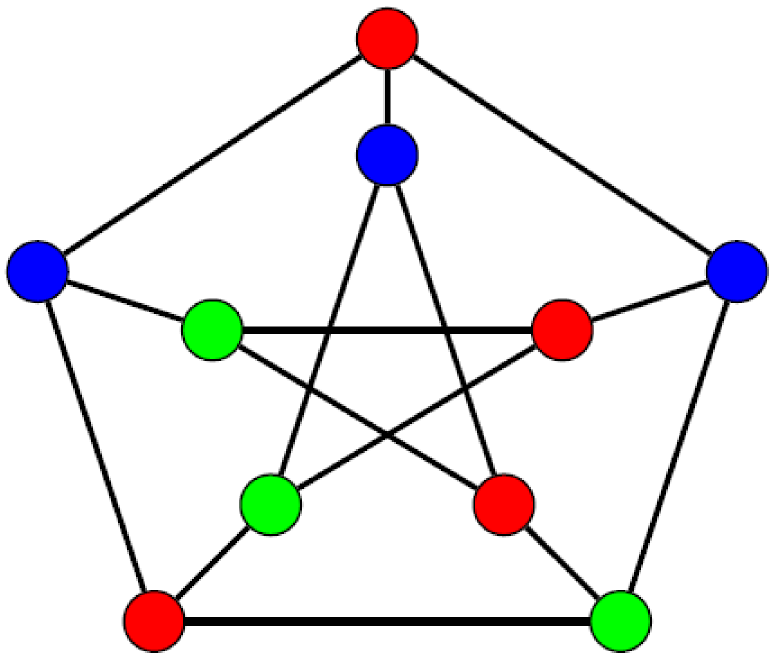
G4

hamiltonian path is C=(A,C,B,E,D)



3.3 Graph Coloring

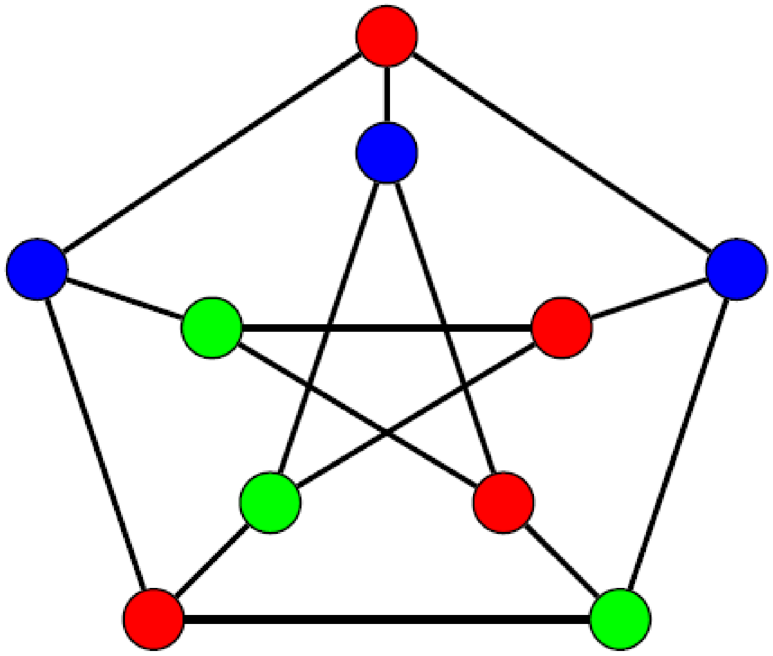
•


Une coloration du graphe de Petersen avec 3 couleurs

3.3 Graph Coloring

Definition 1:

Coloring a graph G is the act of coloring its vertices in such a way that two distinct, adjacent vertices always have different colors.

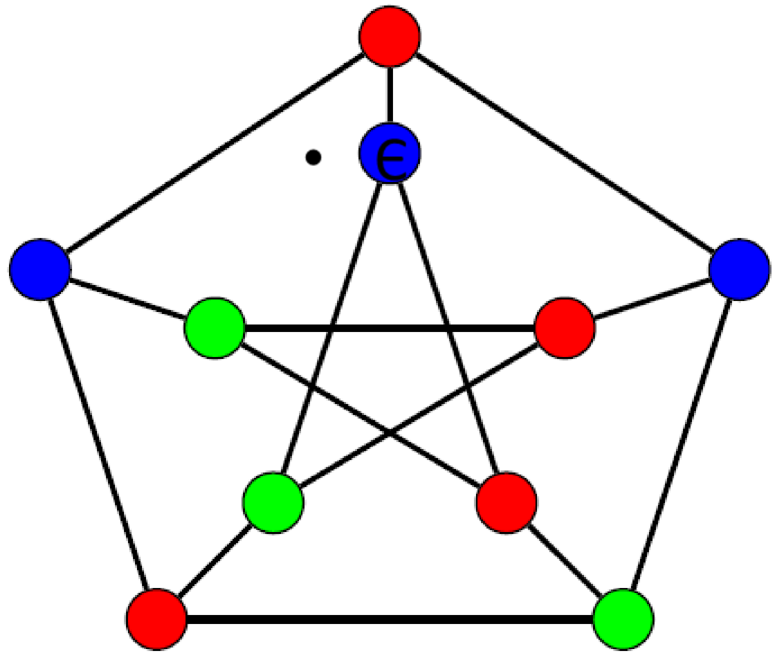


Une coloration du graphe de Petersen avec 3 couleurs

3.3 Graph Coloring

Definition 2:

Let G = (V, E) be a graph, and let Cv be the color of each vertex v, for all w ∈ V. For every (v, w) ∈ E then Cv ≠ Cw.



Une coloration du graphe de Petersen avec 3 couleurs

3.3 Graph Coloring (Applications)

Vertex coloring has numerous practical applications:

1. **Graph Theory and Combinatorics:** Vertex coloring is a fundamental concept in graph theory, and it helps solve problems related to graph coloring, chromatic numbers, and more.
2. **Circuit Design and VLSI Layouts**: In electronic chip design, vertex coloring is used to minimize the number of tracks and routes required for interconnections, reducing cost and improving efficiency.
3. **Map Coloring**: In cartography, vertex coloring is used to determine the minimum number of colors needed to color regions on a map so that no adjacent regions share the same color.
4. **wireless networks** where coloring forms the basis of protocols like MAC TDMA.
5. **In** general, vertex coloring is used as a means to break symmetries, a central theme in distributed computing **(operating systems).**
6. **Register allocation in compilation.**

3.3 Graph Coloring

7 - Graph Coloring Games: Some puzzles and games, like Sudoku and graph coloring puzzles, are based on vertex coloring concepts.

8- Course Scheduling: In educational institutions, vertex coloring can be applied to schedule classes, ensuring that no two classes with conflicting schedules are held simultaneously.

9- Job Scheduling and Task Assignment: In resource allocation problems, such as job scheduling and task assignment, vertex coloring can help allocate resources or tasks efficiently.

10- Social Network Analysis: In the analysis of social networks or complex networks, vertex coloring can be used to identify and analyze patterns of connections or interactions between individuals or entities.

3.3 Graph Coloring

Main use is Incompatibility Problems:

- Organizing an exam based on the subjects each student must take. How to schedule multiple exams in parallel without disadvantaging any candidate?
- Optimizing the use of work machines. How to schedule multiple manufacturing processes using several machines in parallel?
- How to accommodate people or animals while considering their incompatibility?

3.3 Graph Coloring

Coloring Problem:

The usual coloring problem consists of attempting to color a graph using as few colors as possible, where the number of colors used is equal to its chromatic number.

The chromatic number $\chi(G)$:

is the <u>**smallest number**</u> of colors needed to color the graph.

Coloring a planar map:

Coloring a planar map involves coloring the faces of the map in such a way that two faces that share an edge do not have the same color.

3.3 Graph Coloring

- **<u>Four-Color Theorem</u>**

The faces of a planar map can be colored with at most four colors.

The proof of this theorem is historical. Initially, many researchers proved it for numerous specific cases,

THEOREM 1*: Every simple planar graph is 6-colourable.*

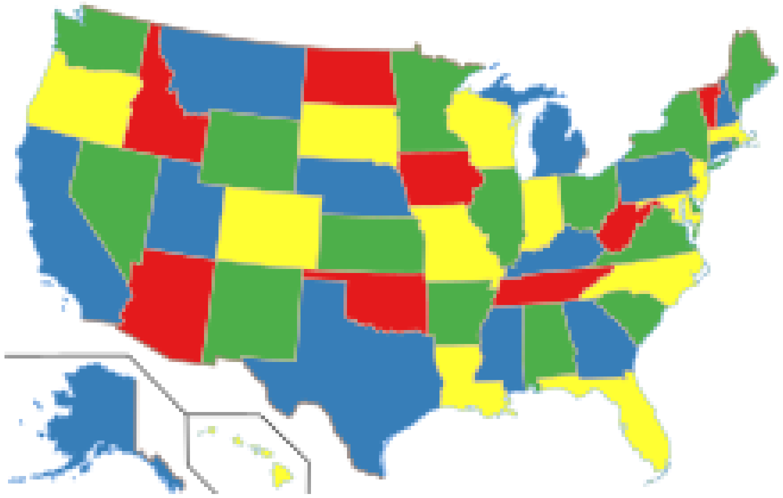
THEOREM 2*: Every simple planar graph is 5-colourable.*

and later, researchers used computer programs to verify that this theorem held true for all graphs.

THEOREM 3: (Appel and Haken, 1976).

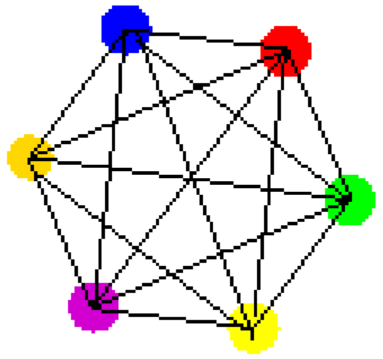
Every simple planar graph is 4-colourable.

Coloring the United States Map **in four colors**.



3.3 Graph Coloring

Important Remarks:



- If a graph has n vertices, then **$\chi(G) \leq n$**.
 (Different colors can be assigned to different vertices.)

- If a graph with n vertices is complete (K(n)), then **$\chi(G) = n$**.
(Distinct vertices must have different colors, so at least n colors are needed.)

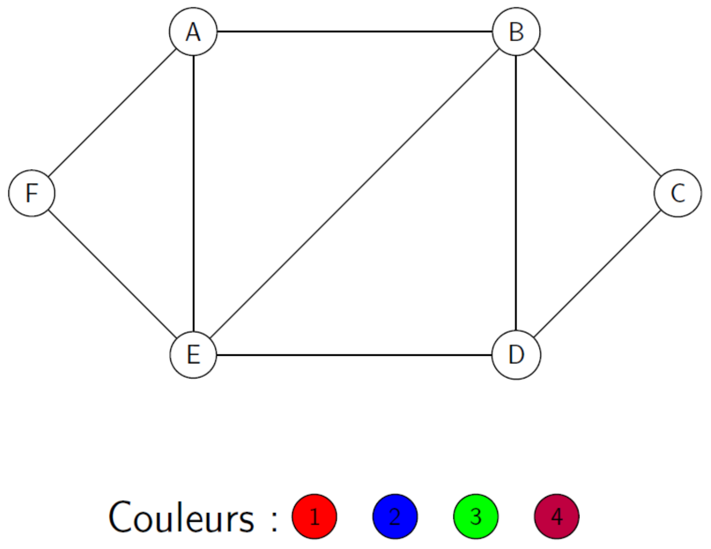
- If the maximum degree of a vertex is d, then **$\chi(G) \leq d + 1$**.
 (Select a vertex of degree d along with the vertices connected to it, and color these d + 1 vertices with d + 1 different colors.)
- When the graph is planar, then **$\chi(G) \leq 4$**.
- If a graph G of order n has a **complete** subgraph **with m** vertices,
 then **$m \leq \chi(G) \leq n$.**

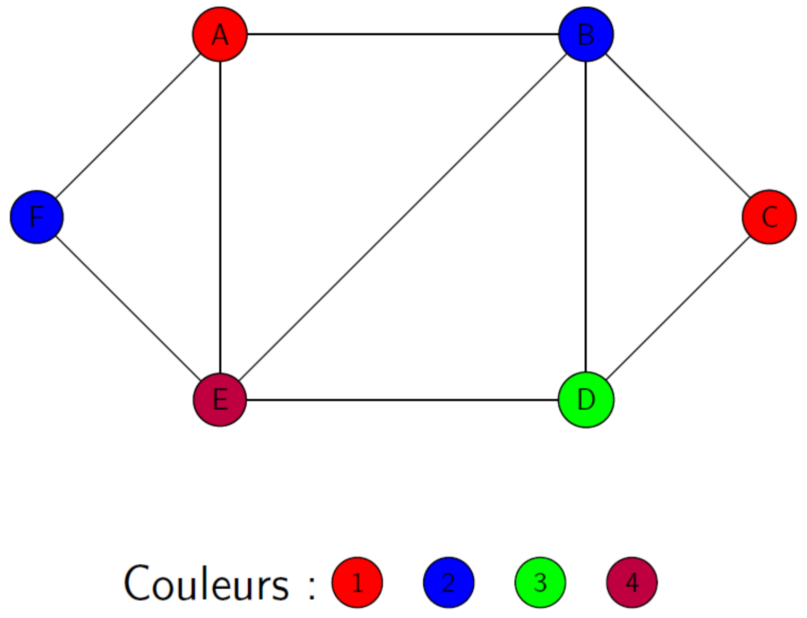
3.3 Graph Coloring

Chromatic Number Search:

Algorithm 1: Greedy Algorithm

```
While  there is an uncolored vertex v,  do
    Color v with the minimum color that does not conflict with its neighbors.
End while
```





3.3 Graph Coloring

Chromatic Number Search:

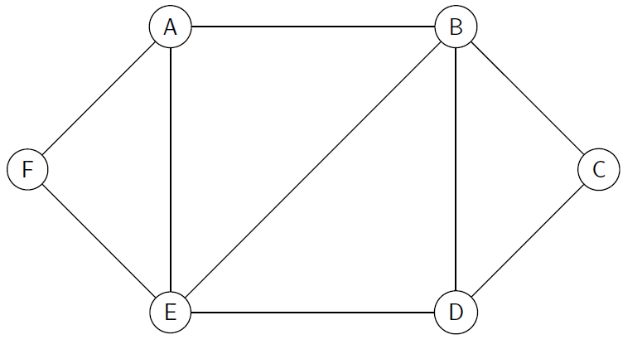
Algorithm 2: Welsh-Powel

1. Arrange the vertices in <u>descending</u> order of their <u>degrees</u>.
2. Choose a color.
3. Assign this color to the first uncolored vertex in the list.
4. Continue down the list, assigning the same color to any vertex that:
   * is not yet colored.
   * is not adjacent to a vertex colored with this color.

Continue until the list is completed.

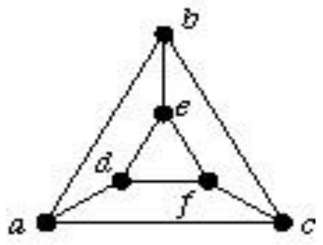
If not all vertices are colored, choose a color that has not been used yet and repeat steps 3 and 4.

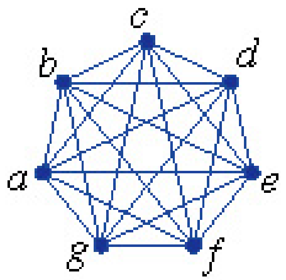
Note: By changing the order of the vertices in the list, you can sometimes achieve a lower number of colors.

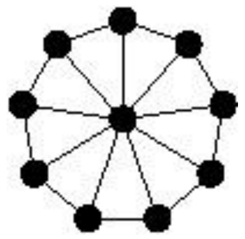
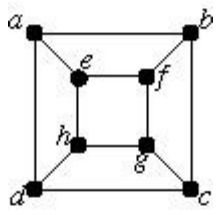


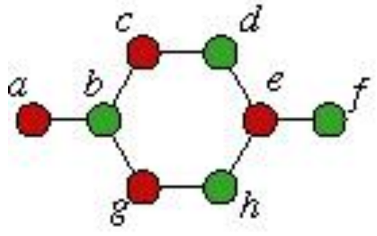
Couleurs : 🔴 1 🔵 2 🟢 3 🔴 4

3.3 Graph Coloring







G1(3 to 4) **G2 (7)** **G3 (3 to 4)** **G4 (2)** **G5(2 to3)**

To be verified for these 5 graphs:

1- Descending order of degrees

2- Numerical order

3- Arbitrary order

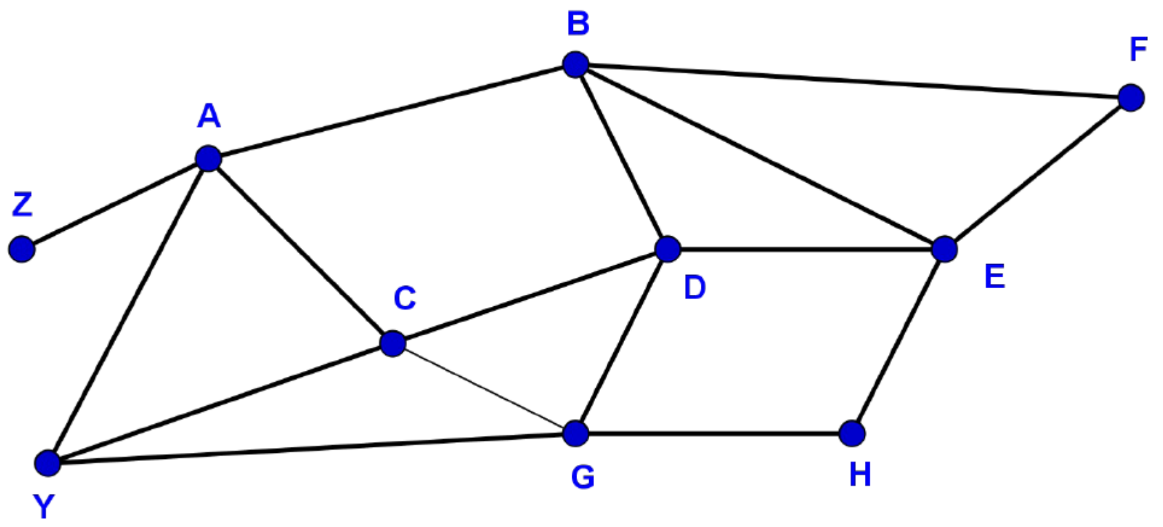
3.3 Graph Coloring

Exercise1:

1- Calculate the chromatic number of a bipartite graph.

2- Provide an algorithm to determine if a graph is bipartite.

3- Given the following connected graph:

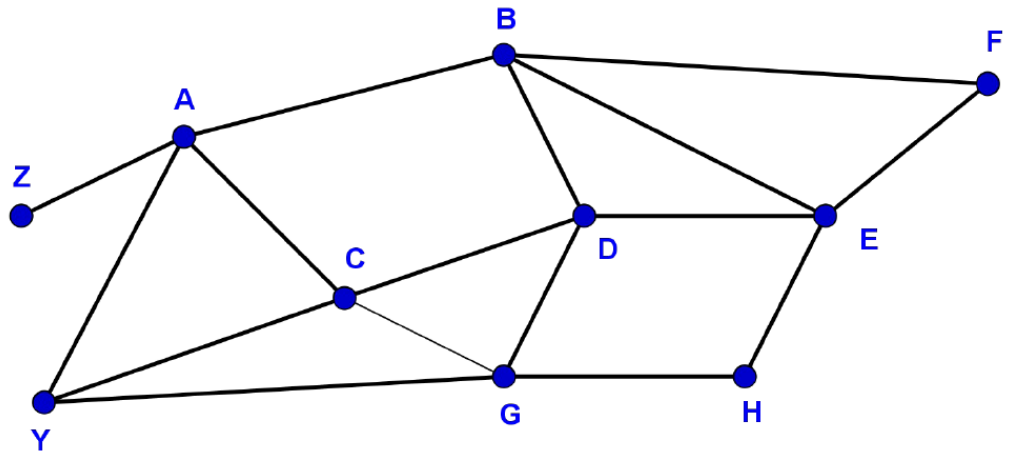


a- Determine an upper bound (encadrement)on the chromatic number of this graph.

b- Show that the chromatic number is equal to 3.

3.3 Graph Coloring

Exercice1:



a) Δ is the largest degree of the vertices in the graph.

Δ=4 and Δ+1=5.

There are complete subgraphs of order 3, and there are no complete subgraphs of order 4 contained in this graph. So, m=3.

The chromatic number is therefore between 3 and 5.

3.3 Graph Coloring

Exercise1:

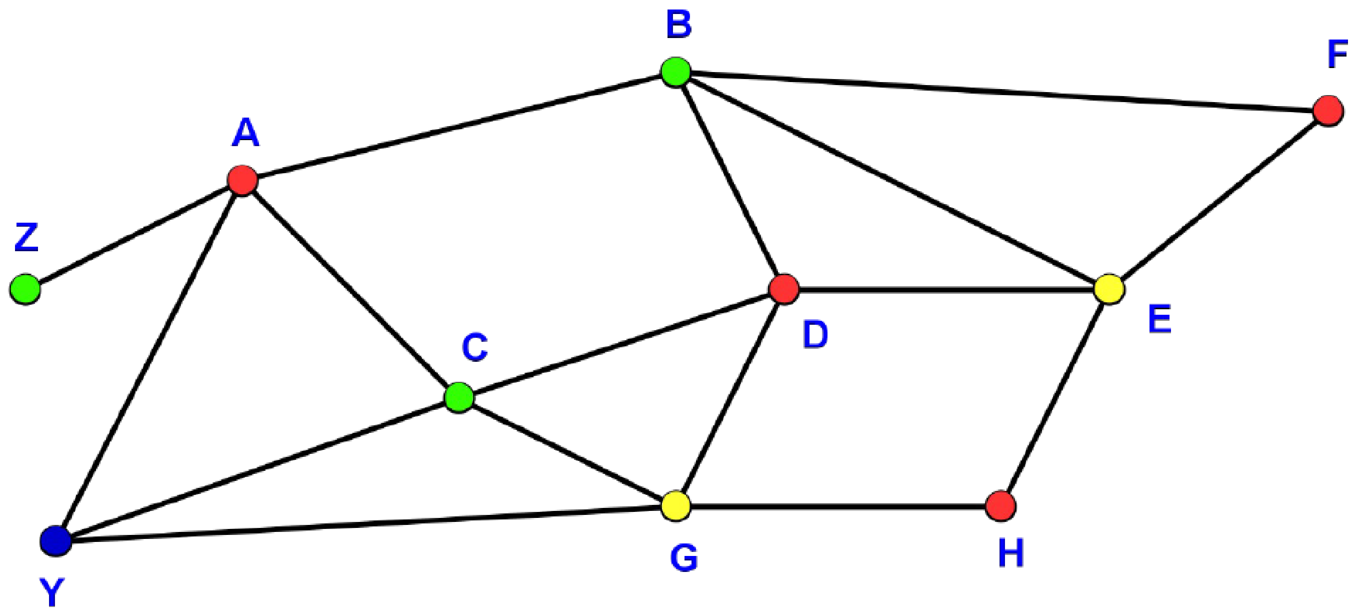
b. To color the graph, we use the previous algorithm. (**Welsh-Powel**)

degré	4	4	4	4	4	4	3	2	1
Vertex	A	B	C	D	E	G	Y	H	Z

List of vertices: A; B; C; D; E; G; Y; F; H; Z

List of colors: Red; Green; Yellow; Blue

We have used four colors, so we can assert that the chromatic number is **between 3 and 4**, but we cannot claim that the chromatic number is 4.



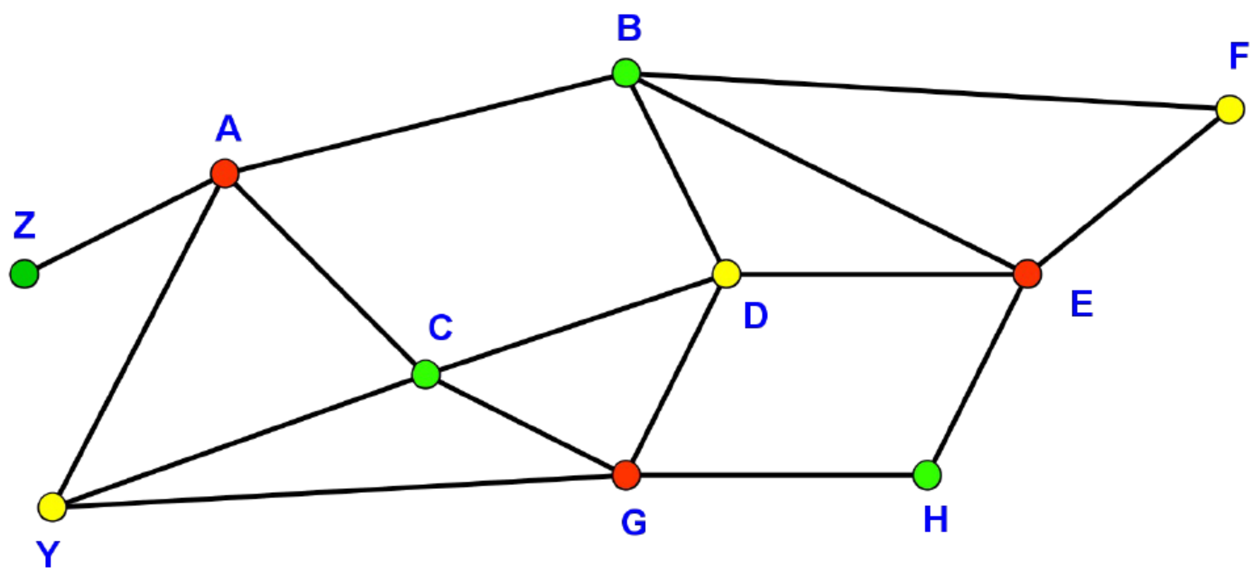
3.3 Graph Coloring

Note

If we change the order of the list of vertices (**we swap C and G**).

List of vertices: A; B; **G**; D; E; **C**; Y; F; H; Z

List of colors: Red; Green; Yellow

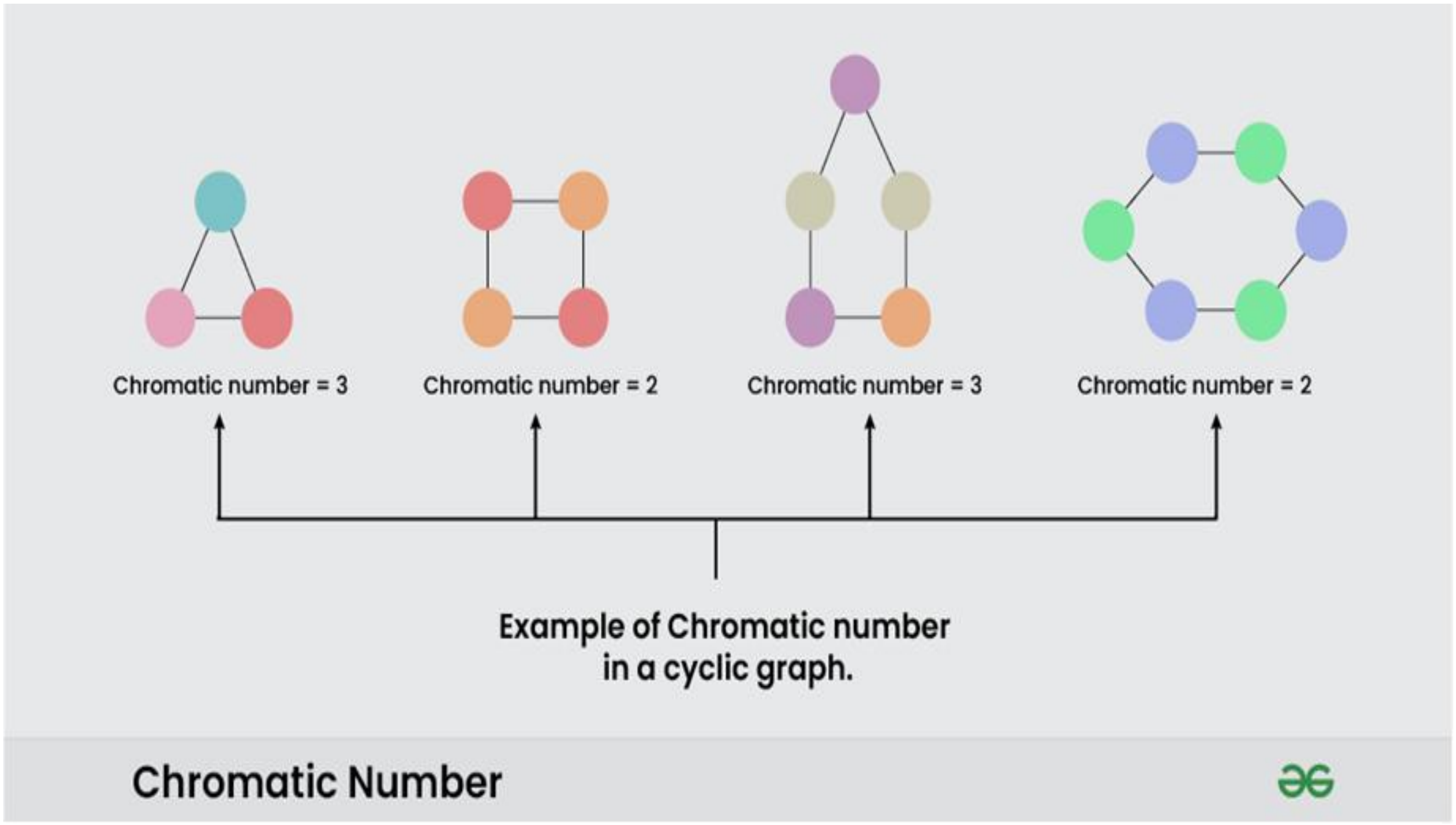


We have used 3 colors, so now, we can say that the chromatic number of this graph is 3.

3.3 Graph Coloring

Exercise 2: Cyclic graph

|V| = n if n is even then $\chi(G) = 2$ else (n is odd) $\chi(G) = 3$



3.3 Graph Coloring

Exercise 3:

The computer science department offering 8 courses.

The table shows with an x which pairs of courses have one or more students in common.

Only 2 air-conditionned lecture halls are available for use at any one time.

	F	M	H	P	E	I	S	C
F		X		X	X	X		X
M	X				X	X		
H						X	X	X
P	X							X
E	X	X				X		
I	X	X	X		X		X	
S			X			X		
C	X		X	X				

To design efficient way to schedule the final examinations, represent the information in this table by using a graph. In the graph, represent courses by vertices and join two courses by an edge if there is any student enrolled in both courses.

3.3 Graph Coloring

Exercise 3 :

- We have used 4 colours
- Time slot 1 : F, H
- Time slot 2 : P, I
- Time slot 3 : C, M, S
- Time slot 4 : E
- We remark that the second condition not verified (we have just 2 air-condionned)
- We swap S from slot 3 to slot 4

	F	M	H	P	E	I	S	C
F		X		X	X	X		X
M	X				X	X		
H					X	X	X	X
P	X							X
E	X	X			X	X		
I	X	X	X				X	
S			X		X	X		
C	X		X	X				

3.3 Graph Coloring

Example: homework ?

9 groups of students are learning 5 courses in a semester.

The course C1 is taken by groups 1,2,3

The course C2 is taken by groups 6,7

The course C3 is taken by groups 1,2,7,9

The course C4 is taken by groups 4,6,8

The course C5 is taken by groups 2,3,4,5

We want to schedule the exams such that no group will have more than one exam in one day, and the length of the exam period will be as short as possible.

Solution idea:

Create a graph where each group is a vertex, and there is an edge between two vertices if the corresponding groups share at least one course.

3.3 Graph Coloring

Exercises: **Kuratowski's theorem**

Provides a fundamental characterization of planar graphs:

A finite graph is planar if and only if it does not contain a subgraph that is homeomorphic to K_5 (the complete graph on 5 vertices) or $K_{3,3}$ (the complete bipartite graph with 3 vertices on each side).

In other words: If a graph contains a structure equivalent to K_5 or $K_{3,3}$, then it is not planar. Otherwise, it is planar.

3.3 Graph Coloring

Edge Coloring:

Edge coloring is the process of assigning a **color to each edge** of a graph so that **no two adjacent edges share the same color**.

Definitions

- Two **edges are adjacent** if they share a **common vertex**.
- A **proper edge coloring** is an assignment of colors to edges such that adjacent edges have **different colors**.
- The **edge chromatic number** of a graph G, denoted $\chi^{'}(G)$,is the **minimum number of colors** needed for such a coloring.

Simple Example

Consider a triangle K_3 with vertices A, B, C. Edges: AB, BC, CA

Each edge touches the other two (they all share vertices),

so we need **3 different colors**:

$\chi'(K3)=3$

3.3 Graph Coloring

Relation to Degrees — Vizing's Theorem

Vizing's Theorem (1964) gives the following bound:

$$\Delta(G) \leq \chi'(G) \leq \Delta(G) + 1$$

where $\Delta(G)$is the maximum degree of the vertices in the graph.

This means:

- The minimum number of colors needed is **either equal to** the maximum degree,
- **or** just **one more** than the maximum degree.

3.3 Graph Coloring

Examples

1. **Even cycle (C_4, C_6, …):**
   → Needs 2 colors
2. **Odd cycle (C_3, C_5, …):**
   → Needs 3 colors.
3. **Complete graph K_n:**
   1. If n is even → $\chi'(K_n) = n - 1$
   2. If n is odd → $\chi'(K_n) = n$

Example

Let's take a simple graph: a **square with a diagonal**.

Vertices: A, B, C, D

Edges: AB, BC, CD, DA, AC

We used **3 colors**.

3.3 Graph Coloring

Properties of Greedy algorithm

- The greedy algorithm is **simple and fast**.
- It **does not always give the optimal (minimum)** number of colors, but it guarantees that no two adjacent edges have the same color.
- In the worst case, it uses at most $\Delta(G) + 1$colors (as Vizing’s theorem states).

3.3 Graph Coloring

Dual Graph

Given a **planar graph** G , its **dual graph**, denoted G^* ,is constructed as follows:

1. **Draw one vertex in each face** of G (including the *outer* or *infinite* face).
2. **Connect two vertices in** G^* if the corresponding faces in G are **separated by an edge** in G.

   In other words, for every edge e in G, there is an edge e^* in G^* that crosses e.

Every region (face) of G becomes a vertex in G∗.

Every edge of G becomes an edge between the two faces it separates.

The dual of the dual brings you back to the original graph (G∗)∗=G

3.3 Graph Coloring

Example: Consider a simple planar graph

Original graph G:

- Vertices: A, B, C, D
- Edges: AB, BC, CD, DA, AC

This graph has **3 faces**:

1. Face 1: Triangle A, B, C
2. Face 2: Triangle A, C, D
3. Face 3: The outer face

Now, to create G^*:

- Place a vertex in each face → F_1, F_2, F_3
- Connect $F_1$and F_2because they share the edge AC
- Connect $F_1$and F_3)they share edge $AB$and BC)
- Connect $F_2$and F_3)they share edge $DA$and CD)

The resulting **dual graph** $G^*$is a **triangle** $F_1F_2F_3$.

Chapter 4

Trees and arborescences

4- Trees and arborescences

4.1 Definitions and Properties

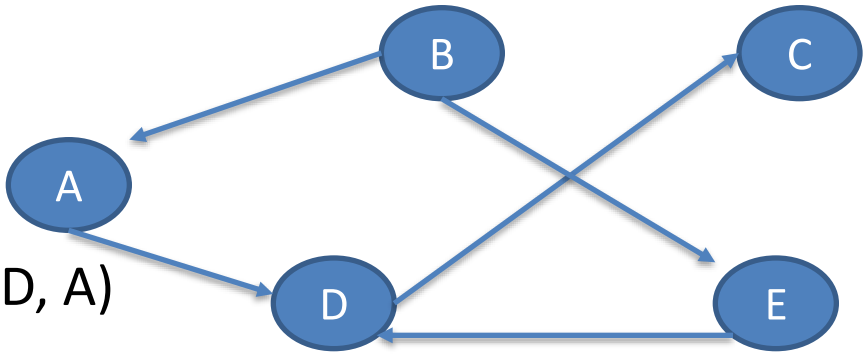
a) Property 1:

Let G = (V, E) be an undirected graph with |V| = n and |E| = m.

If G is connected ⇒ m ≥ n-1

If G is acyclic ⇒ m ≤ n-1

Example 1:

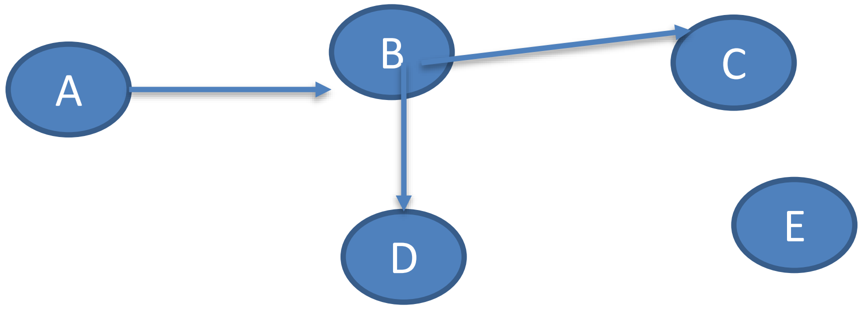


G is connected and

contains a cycle C = (A, B, E, D, A)

m ≥ n-1   5 ≥ 4 (5-1)

Example 2:



m < n   3 < 5

G is not connected

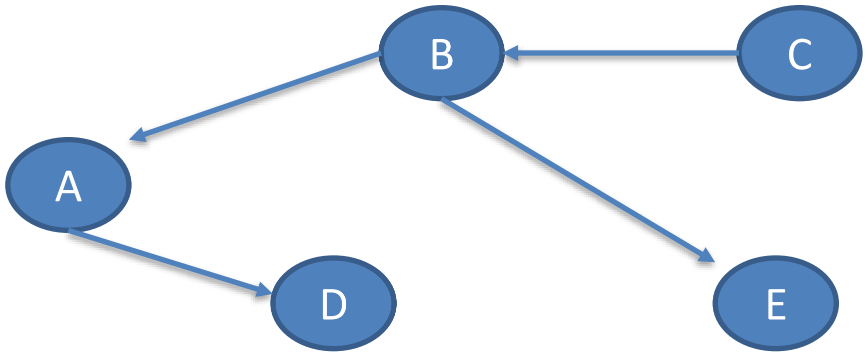
4- Trees and arborescences

<u>4.1 Definitions and Properties</u>

b) Definition 1: (tree)

- A tree is a connected graph with no cycles.
- A tree is a simple graph with no loops and having n-1 edges.

<u>Example</u>: tree



4- Trees and arborescences

4.1 Definitions and Properties

c) Property 2:

Let G = (V, E) be a graph with |V| = n ≥ 2.

The following properties are equivalent and characterize a tree:

1- G is connected and acyclic.

2- G is connected and minimal with respect to this condition (removing an edge from G would make it disconnected).

3- G is connected and has n-1 edges.

4- G is acyclic and maximal with respect to this property (adding an edge to G would introduce a cycle).

5- G is acyclic and has n-1 edges.

6- There is a unique path connecting every pair of vertices in G.

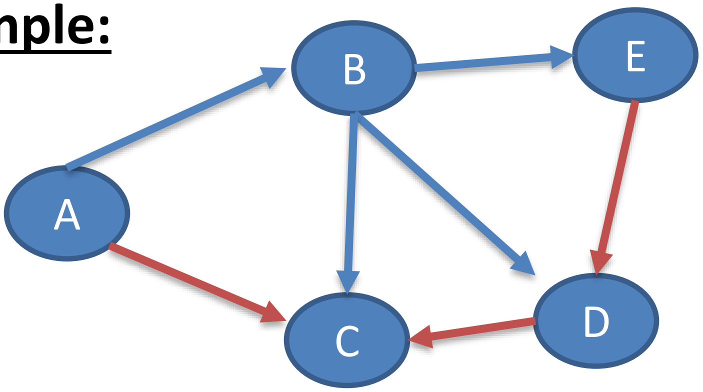
4- Trees and arborescences

<u>4.1 Definitions and Properties</u>

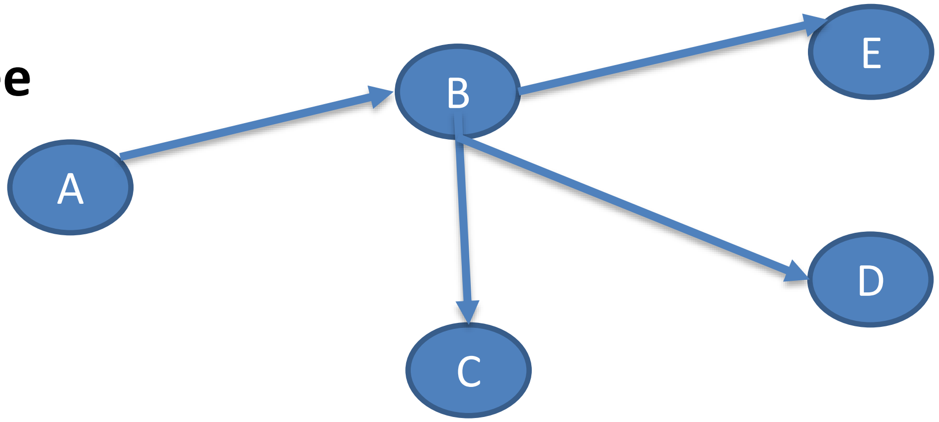
c) Property 3: In a connected graph G, it is possible to extract a subgraph that is a tree.

<u>Example:</u>

G



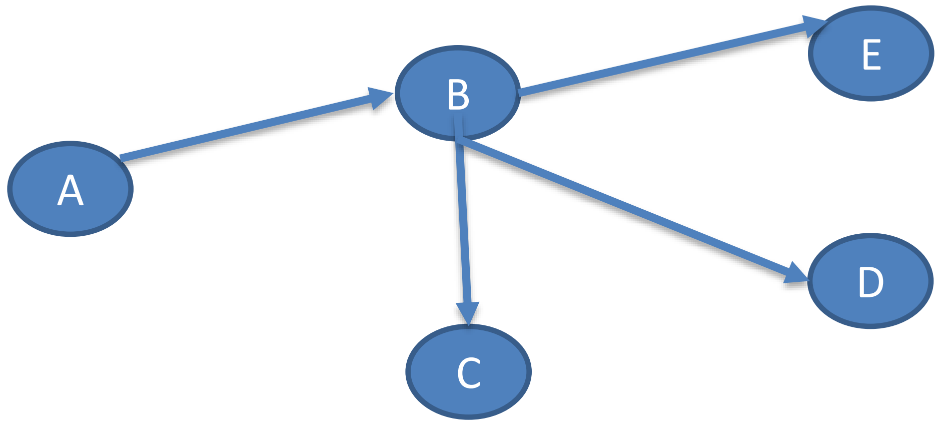
Gtr : tree



4- Trees and arborescences

4.1 Definitions and Properties

- The conditions 1, 3, and 5 are satisfied for the graph Gtr.
- If we remove an edge, Gtr is no longer connected (2).
- If we add an edge (e.g., (D, C)), Gtr will contain a cycle (4).
- For any pair of vertices, there is a unique path connecting them (6).



4- Trees and arborescences

<u>4.1 Definitions and Properties</u>

d) Definition: (Forest)

A forest is a graph in which each connected component is a tree. It is a graph without cycles.

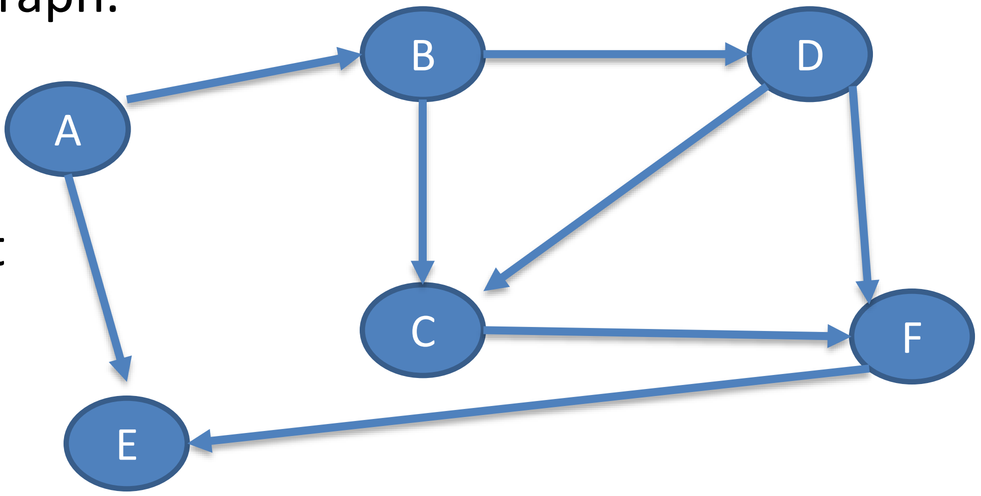
e) Definition: (Root)

A vertex S in a graph G is a root if there exists a path connecting S to every other vertex in the graph.

The vertex A is a root of G.

The vertex E is an anti-root

of G.



4- Trees and arborescences

4.1 Definitions and Properties

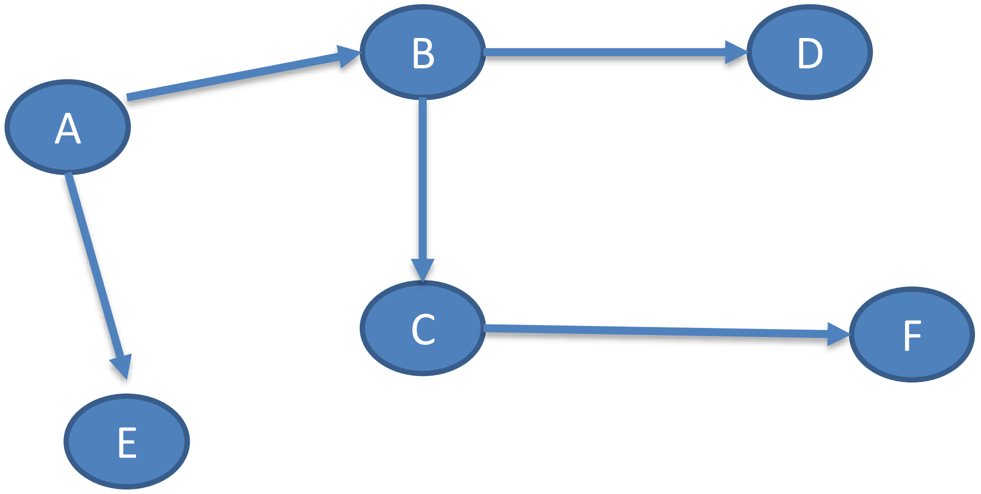
f) Definition: (Arborescence)

A graph G = (V, E), with n ≥ 2, is an arborescence with root S if:

- G is a tree.

- S is a root of G.

Example:



G is a tree and has the vertex A as its root.

⇨ G is an **arborescence**.

4- Trees and arborescences

4.1 Definitions and Properties

g) Examples of Arborescences:

- Dividing a book into chapters.
- Family tree.

h) Definition: (Anti-Arborescence)

A graph G = (V, E), with $n \geq 2$, is an **anti-arborescence** if:

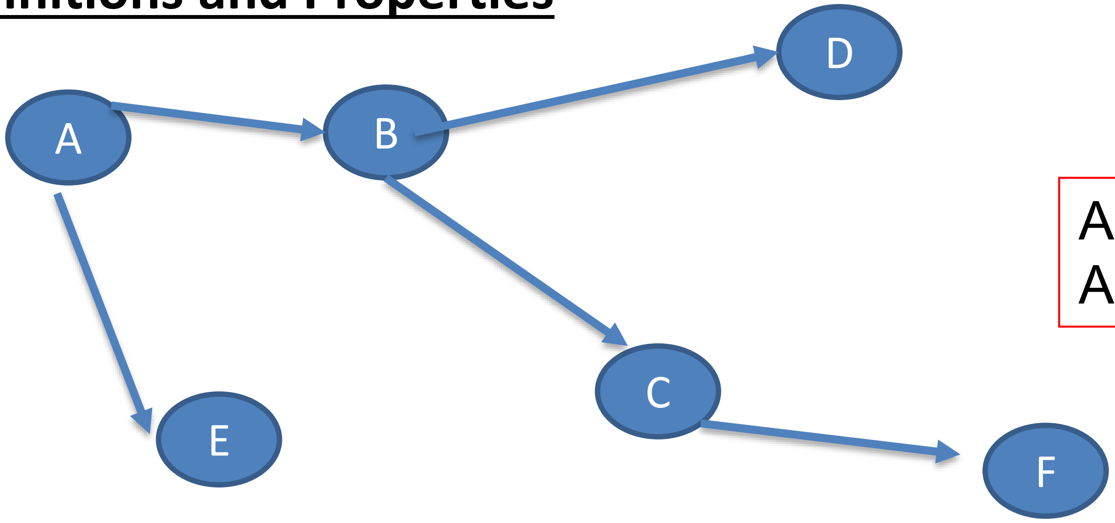
- G is a tree.
- S is an anti-root of G.

Note:

If the direction of edges in an arborescence is reversed, an anti-arborescence is obtained.

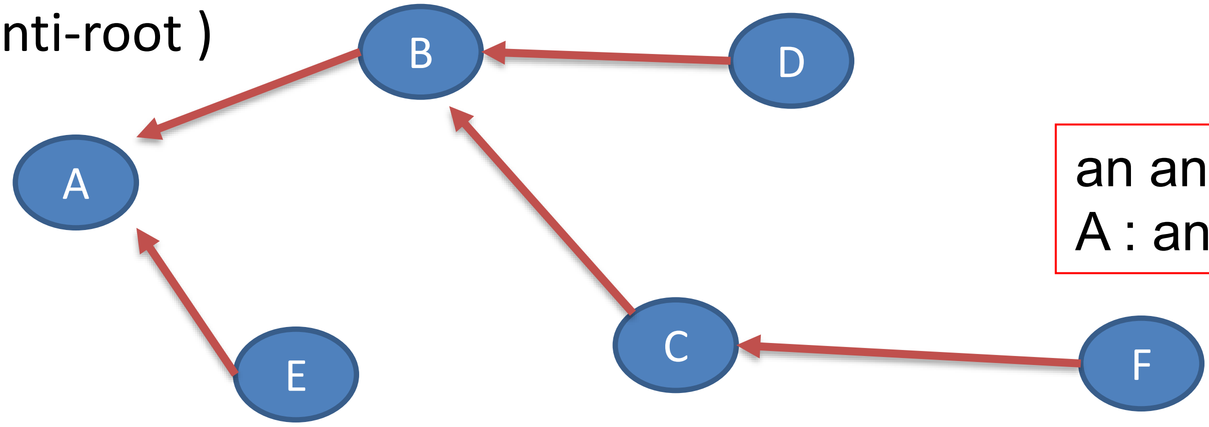
4- Trees and arborescences

4.1 Definitions and Properties



An arborescence
A : root

If the directions of the edges are reversed, an anti-arborescence is obtained.
(A : anti-root)



an anti-arborescence
A : anti-root

4- Trees and arborescences

i) Spanning Tree

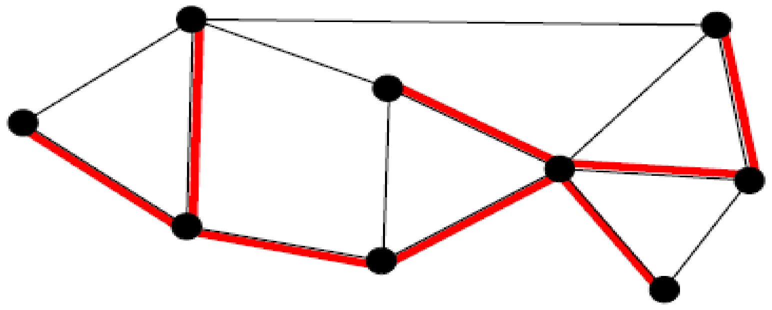
For an undirected graph G,

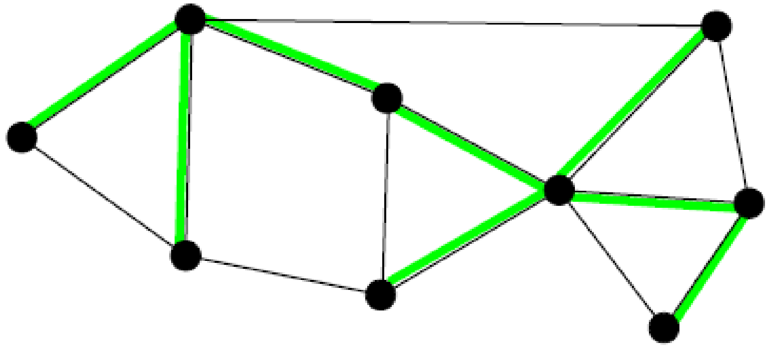
A spanning tree T of G:

- Contains all the vertices of G.
- Contains some edges from G.

Note:

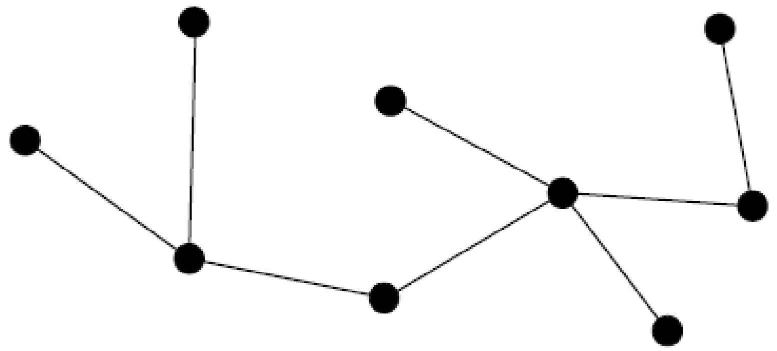
A graph can have multiple spanning trees.



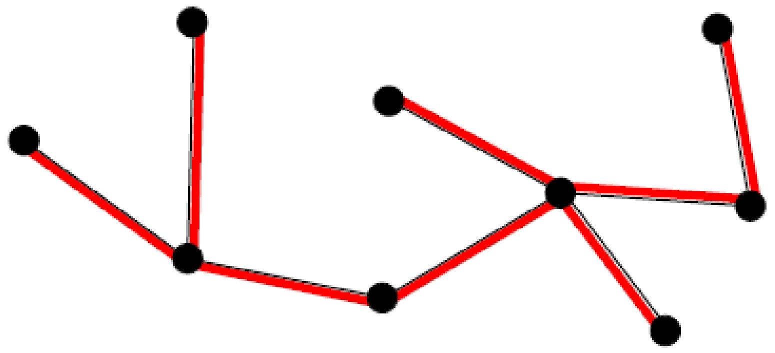


4- Trees and arborescences

i) Spanning Tree

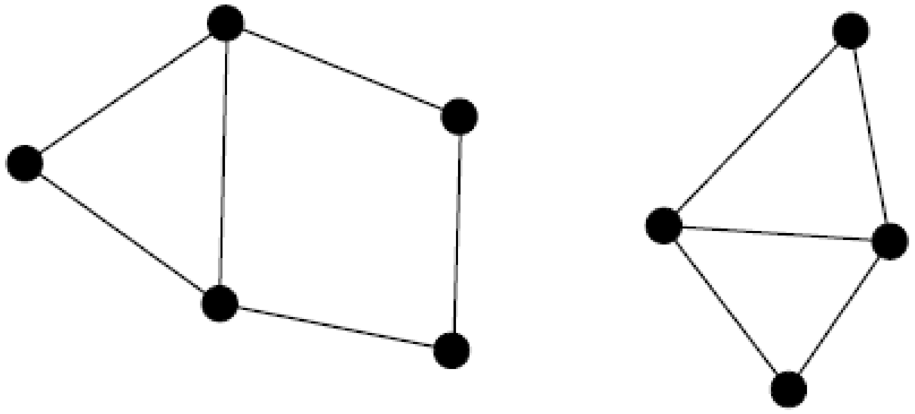


A tree has only one spanning tree, which is the tree itself.



4- Trees and arborescences

i) Spanning Tree



A non-connected graph does not have any spanning trees.

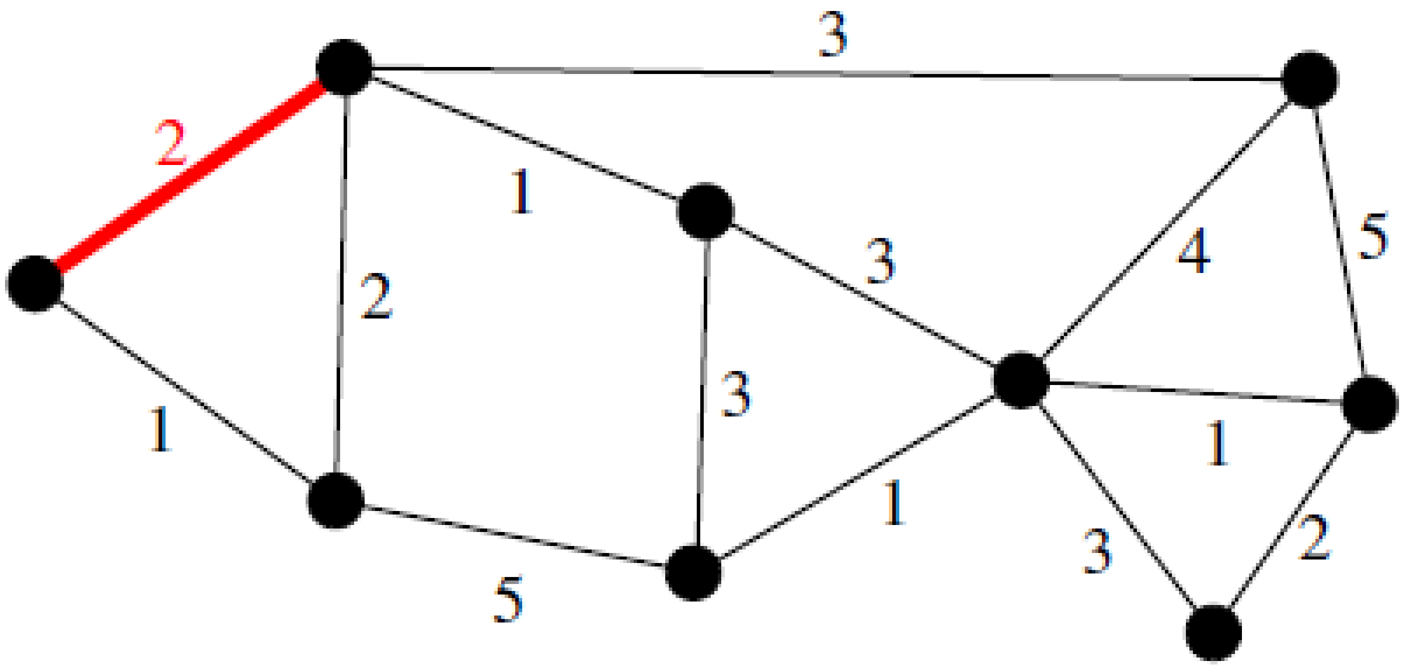
(In other words, a graph that has a spanning tree must be connected.)

A connected graph must have (at least) one spanning tree.

4- Trees and arborescences

j) Weighted graph

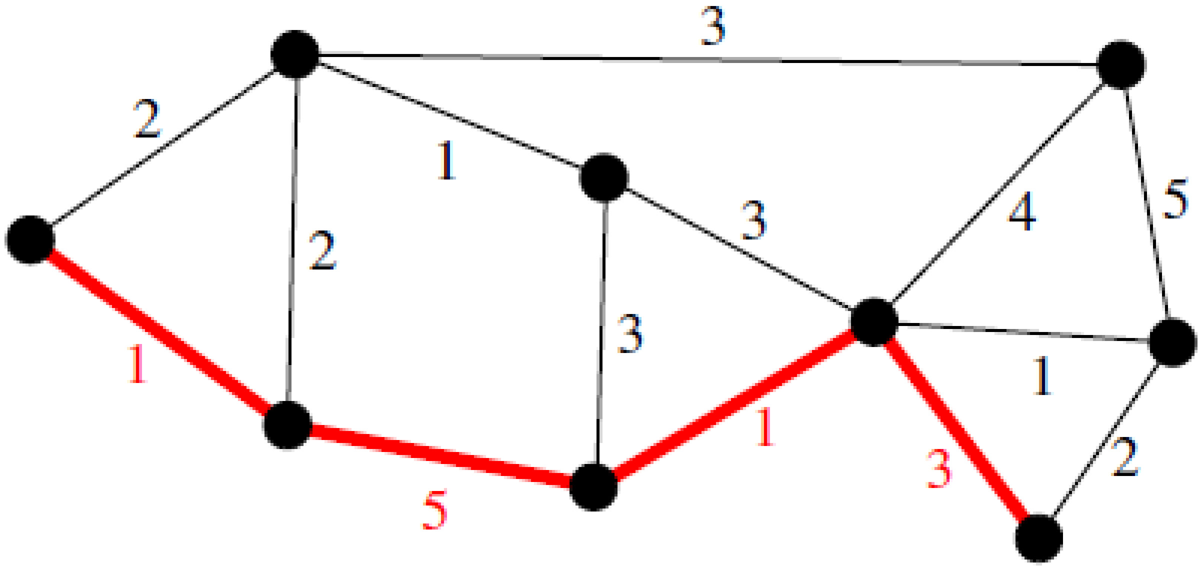
Each edge has a **weight** (> 0)



4- Trees and arborescences

j) Weighted graph

The weight of a path is the sum of the weights of the edges it comprises.



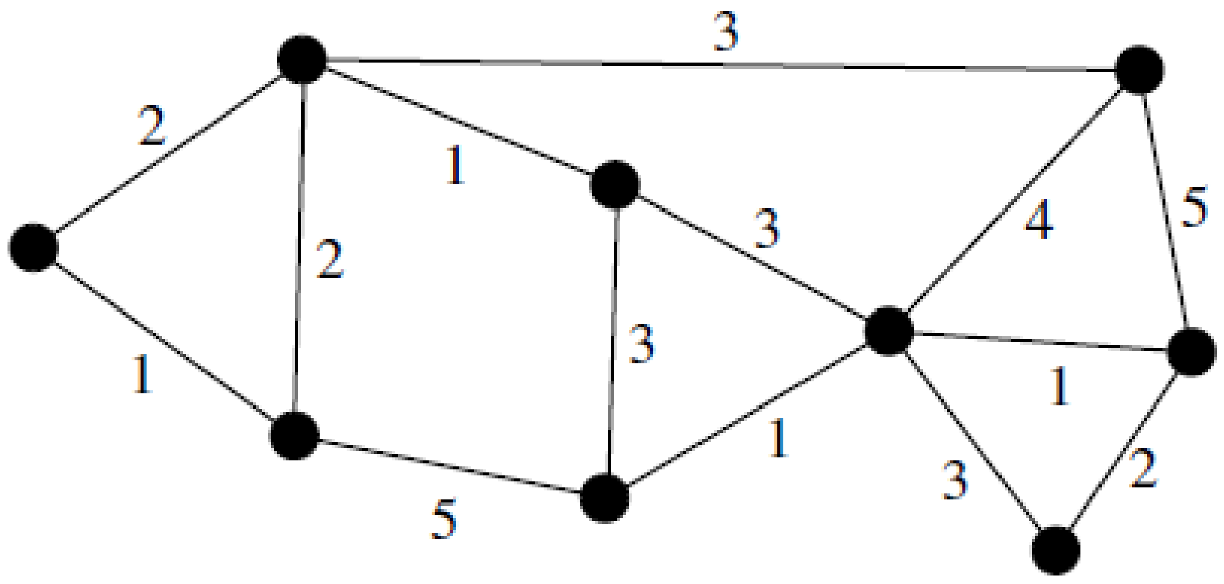
$$1 + 5 + 1 + 3 = 10$$

4- Trees and arborescences

j) Weighted graph

Given: A connected weighted graph

Question: We want to construct a spanning tree with the **smallest** possible total weight.

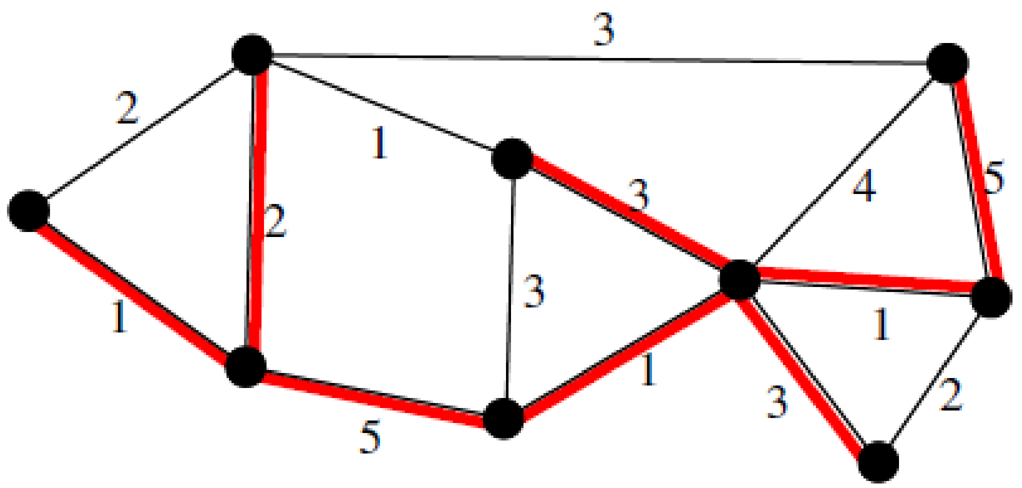


4- Trees and arborescences

j) Weighted graph

Given: A connected weighted graph

Question: We want to construct a spanning tree with the smallest possible total weight.



Can we do better?

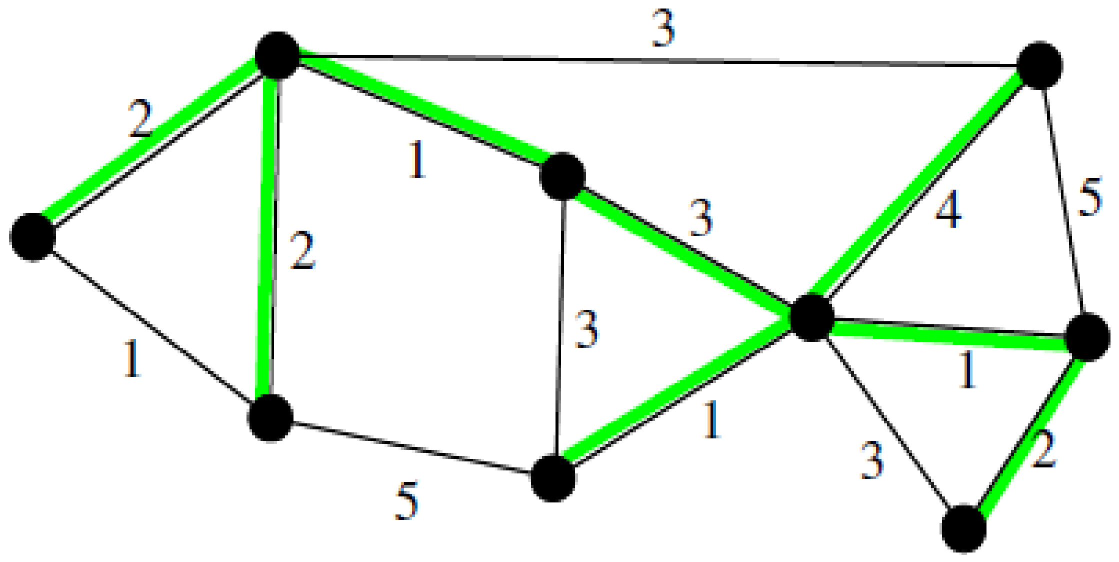
1 + 2 + 5 + 1 + 3 + 5 + 1 + 3 = 21

4- Trees and arborescences

j) Weighted graph

Given: A connected weighted graph

Question: We want to construct a spanning tree with the smallest possible total weight.



Can we do better?

2 + 2 + 1 + 3 + 1 + 4 + 1 + 2 = 16 ➔ **It's the Kruskal algorithm or the Prim algorithm.**

4- Trees and arborescences

4.2 Minimum Weight Tree Search Problem

Let G be a weighted graph.

The Minimum Cost Tree Problem consists of finding a subgraph that is a tree, with the minimum possible sum of the edge weights.

Example of applications 1: Telephone line installation

The goal is to connect all points without unnecessary lines, which involves finding a tree while minimizing the installation cost.

4- Trees and arborescences

<u>4.2 Minimum Weight Tree Search Problem</u>

Example of applications 2: Water network

Let’s consider the problem of connecting n cities with a drinking water network in the most economical way possible.

We assume that the length (l{ij} = l(a{ij})) is known, which is the length of a water pipe needed to connect cities i and j.

The network must obviously be connected and must not contain cycles to minimize costs; therefore, it is a tree, and it must be the most economical spanning tree.

4- Trees and arborescences

4.2 Minimum Weight Tree Search Problem

There are two famous algorithms to solve the problem of the minimum weight spanning tree.

Joseph Kruskal's algorithm (1956) (starts with an edge)

Prim's algorithm (starts with a vertex)

Both of these algorithms provide solutions to this problem.

A **minimum weight spanning tree** is generally **different** from the **shortest path** tree. A shortest path tree is indeed a spanning tree, but it minimizes the distance from the root to each vertex, rather than the sum of the weights of the edges.

4- Trees and arborescences

<u>4.2 Minimum Weight Tree Search Problem</u>

In a more formal manner, a minimum-cost spanning tree of a graph G = (V, E) is a partial graph G' = (V, E') of G such that:

- G' is connected and acyclic (G' is a tree),
- and the sum of the costs of the edges in E' is minimized.

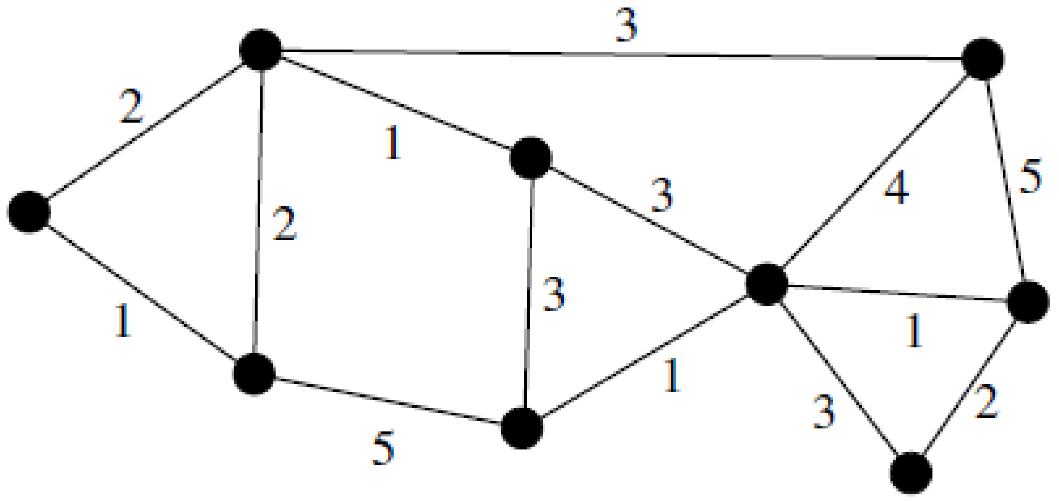
4- Trees and arborescences

a) Kruskal's Algorithm (General Idea)

We start with a forest of trees, each consisting of individual isolated vertices of the graph.

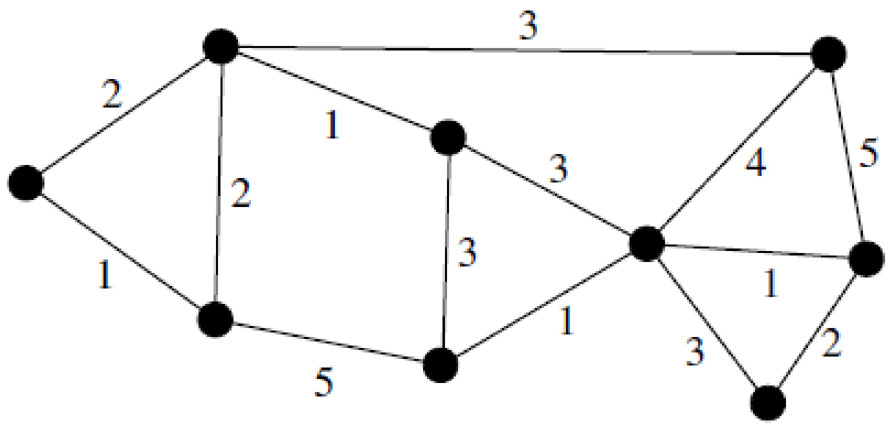
- At each iteration, we add to this forest the edge with the smallest weight that does not create a cycle with the already selected edges.

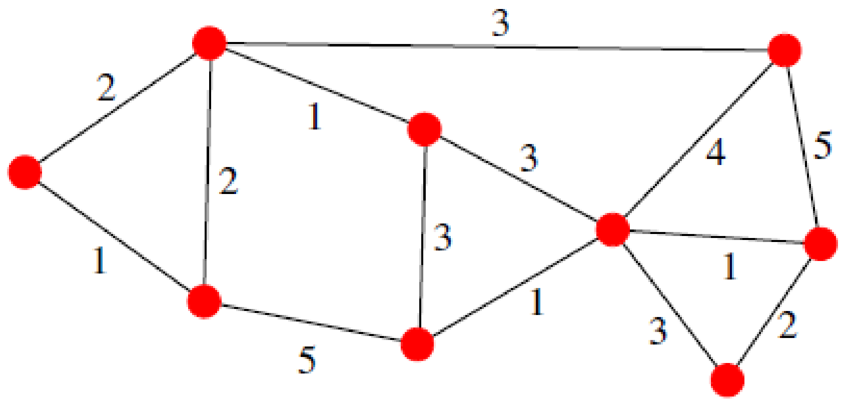
- We stop when we have examined all the edges.

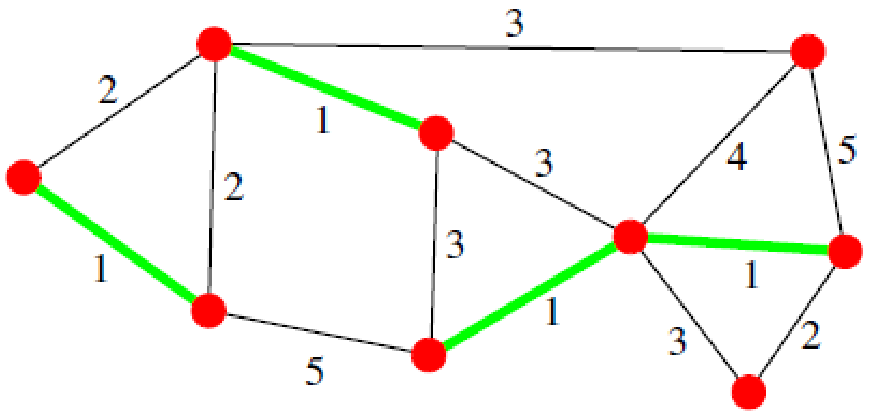


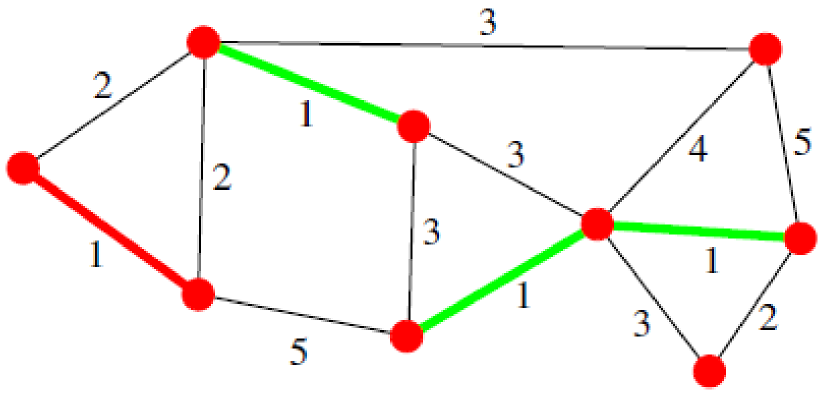
4- Trees and arborescences

b) Kruskal's Algorithm (Application)



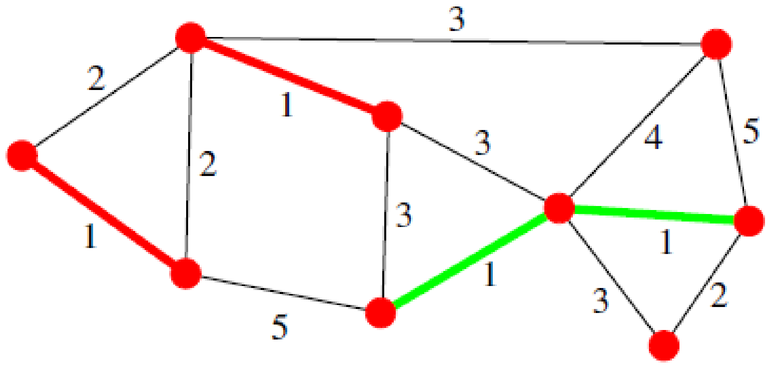


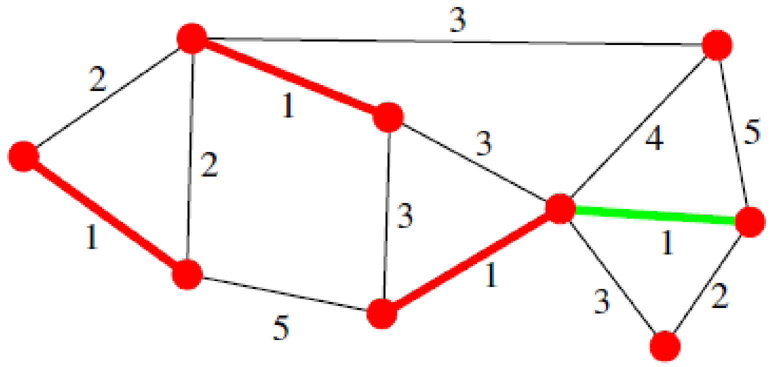


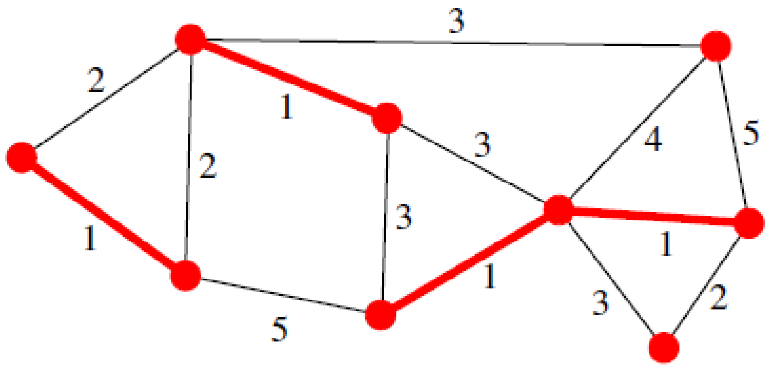


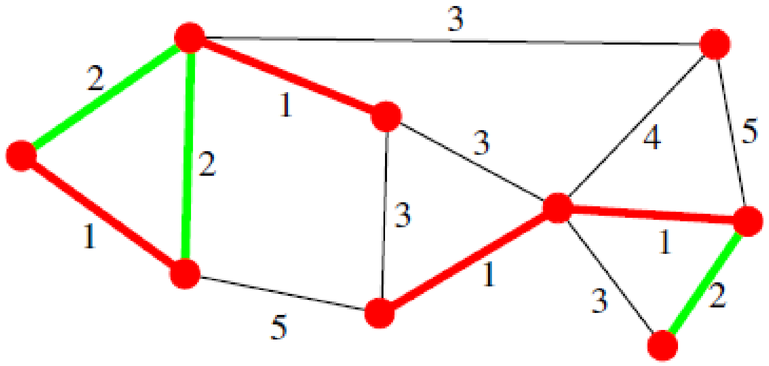
4- Trees and arborescences

b) Kruskal's Algorithm (Application)



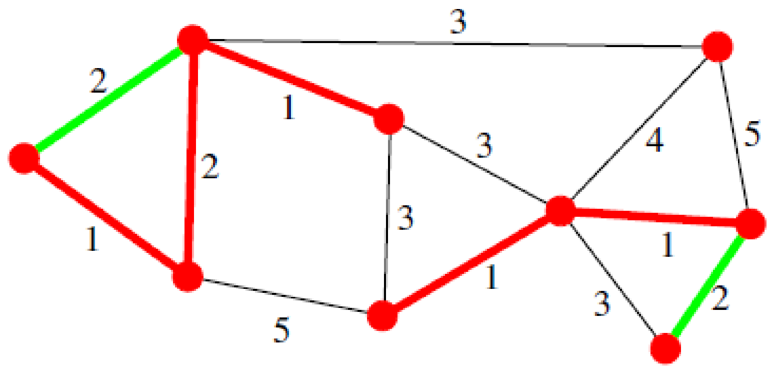


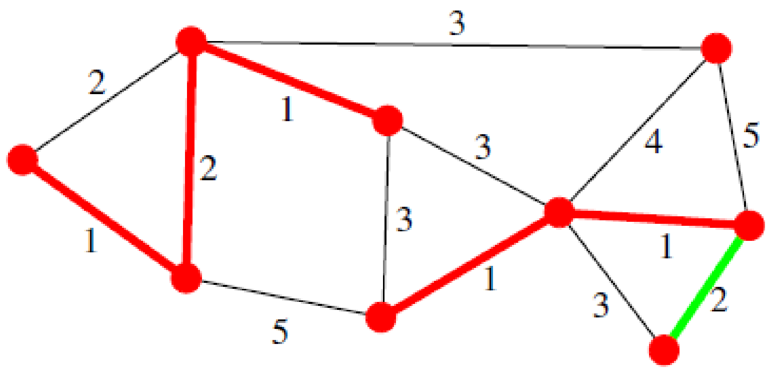


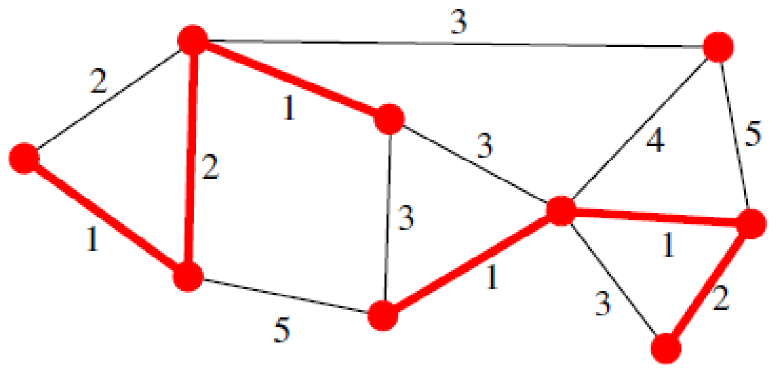


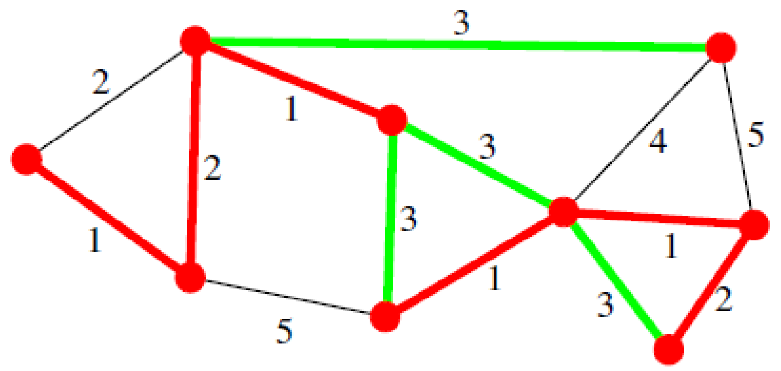
4- Trees and arborescences

b) Kruskal's Algorithm (Application)



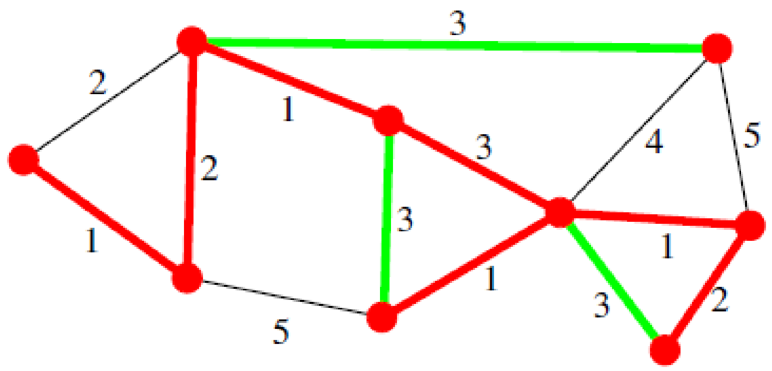


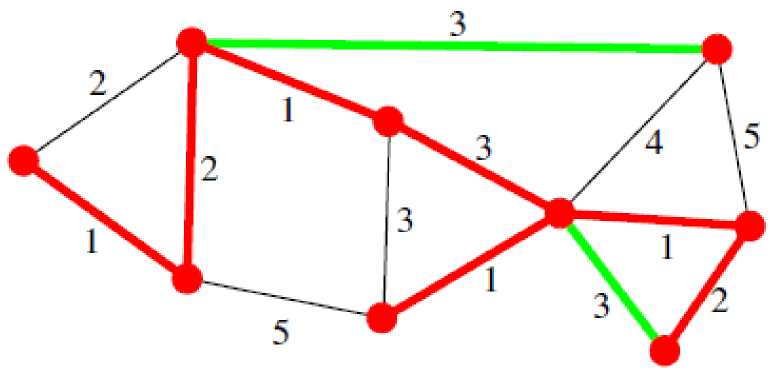


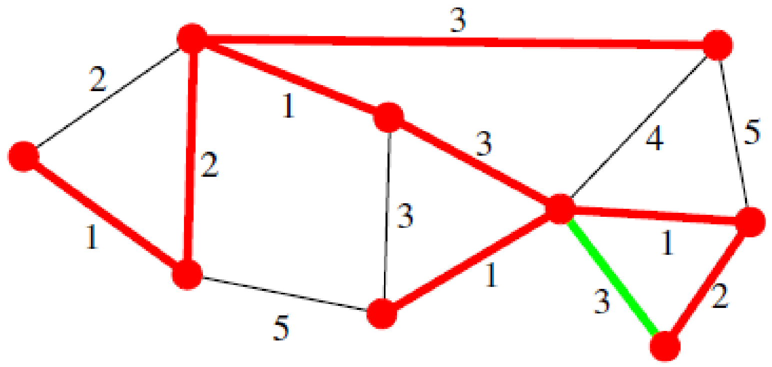


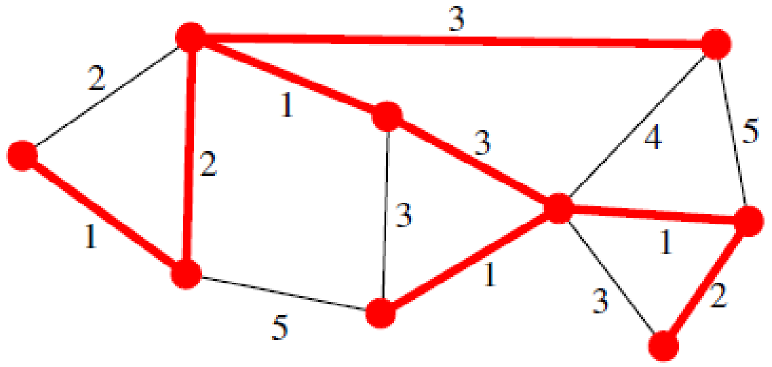
4- Trees and arborescences

b) Kruskal's Algorithm (Application)



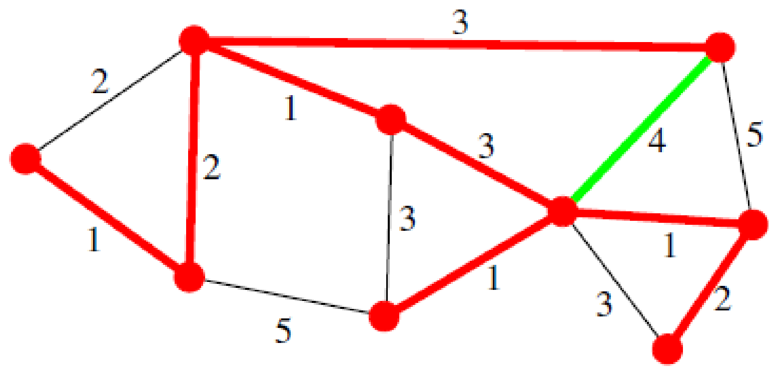


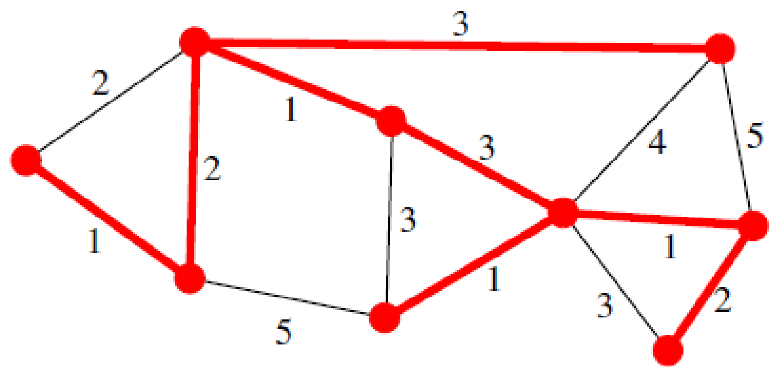


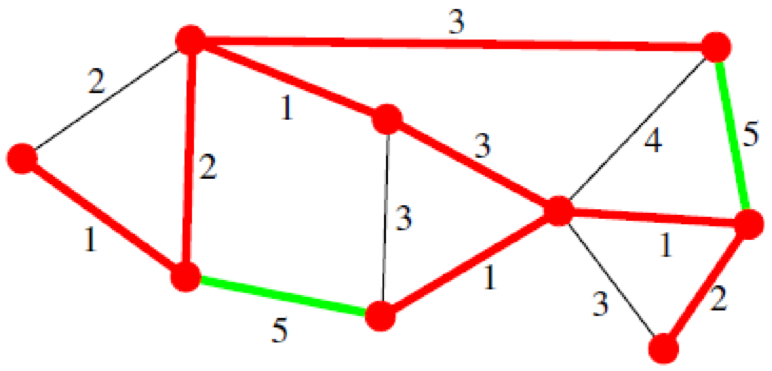


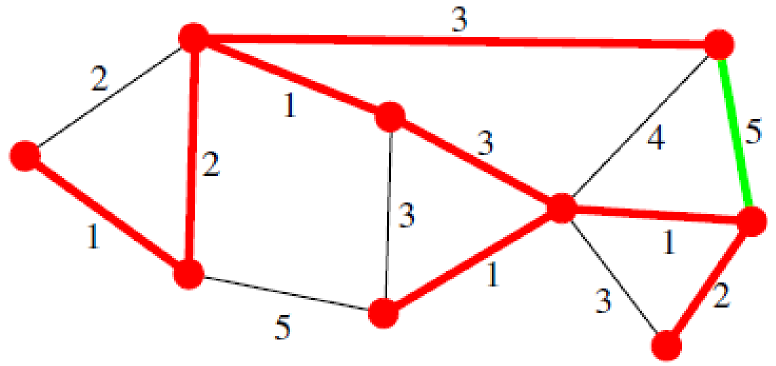
4- Trees and arborescences

b) Kruskal's Algorithm (Application)





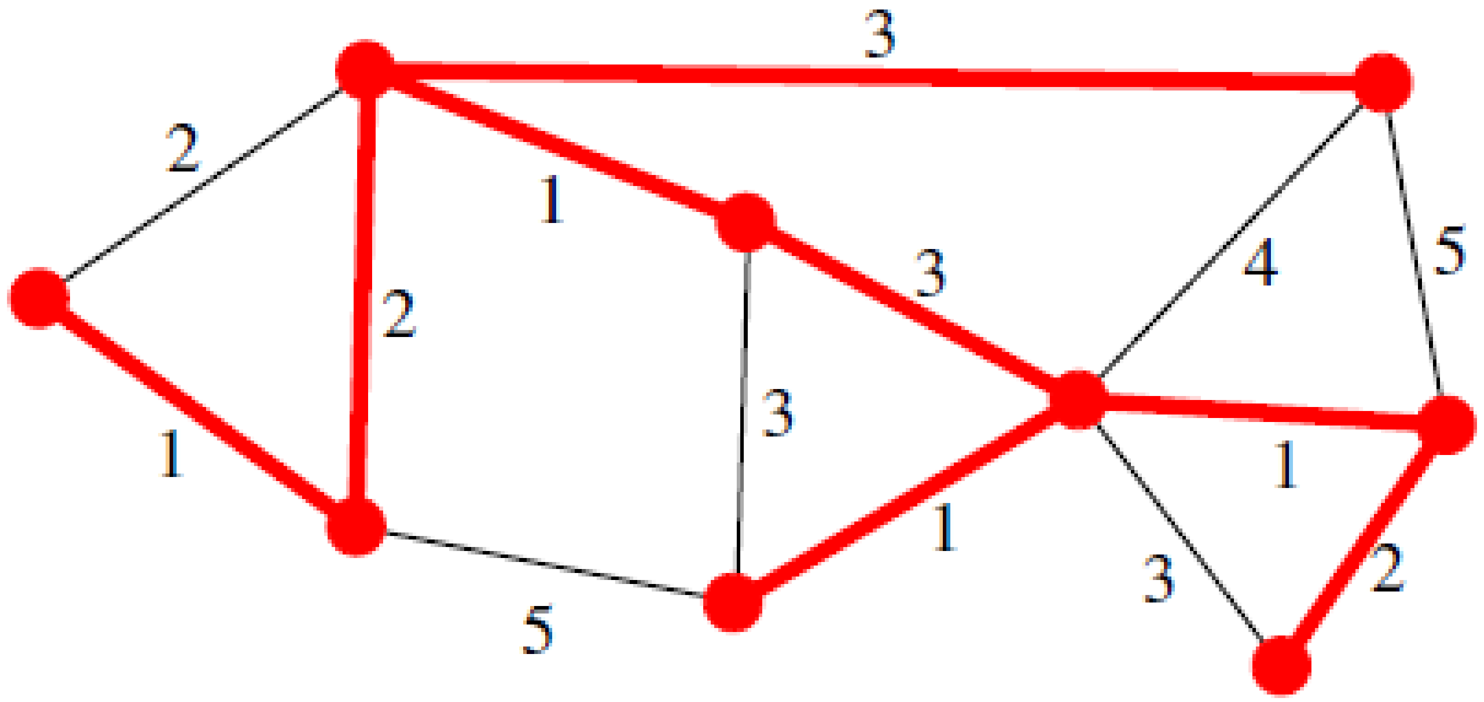




4- Trees and arborescences

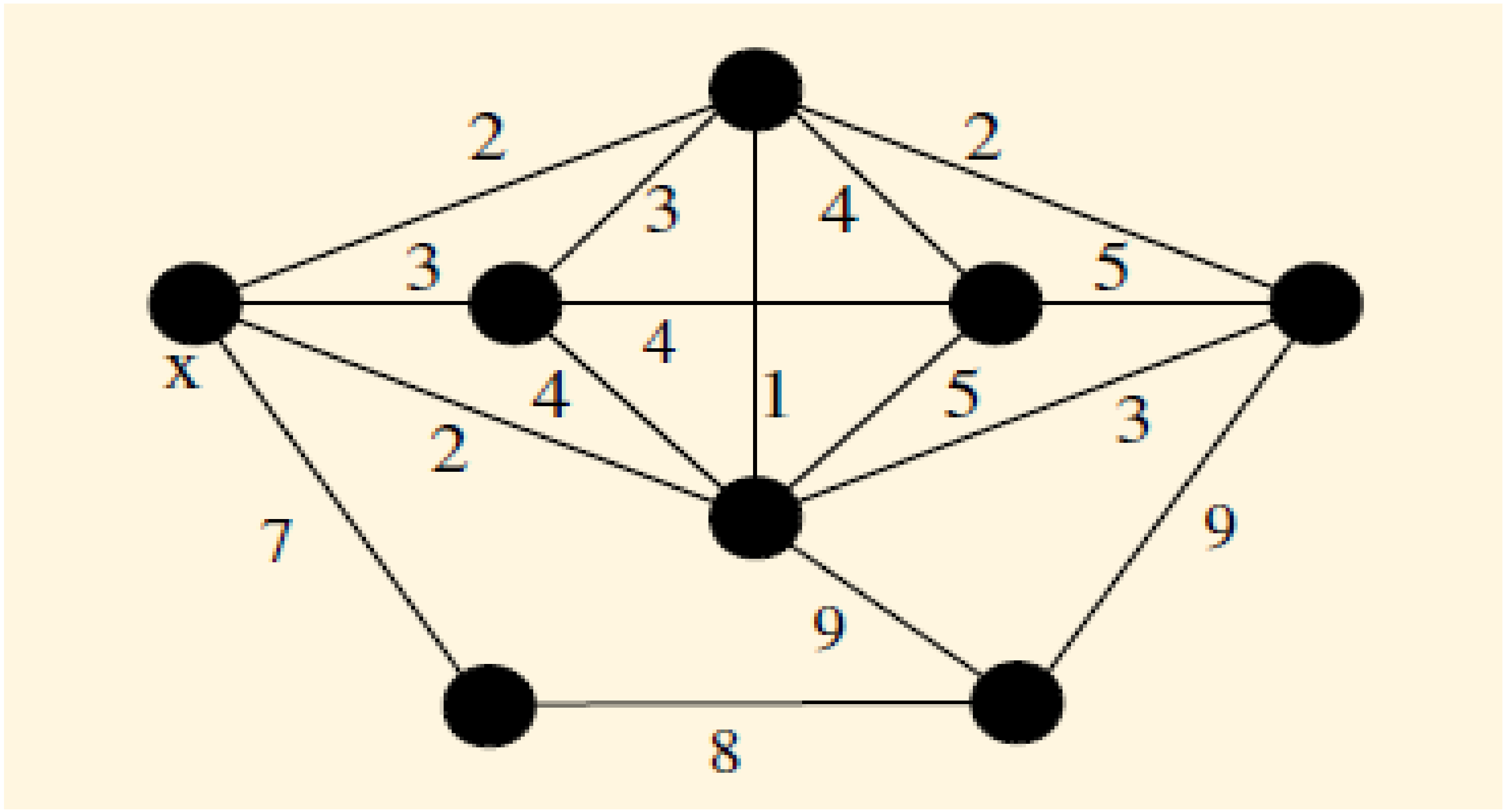
b) Kruskal's Algorithm (Application)

1 + 1 + 1 + 1 + 2 + 2 + 3 + 3 = 14



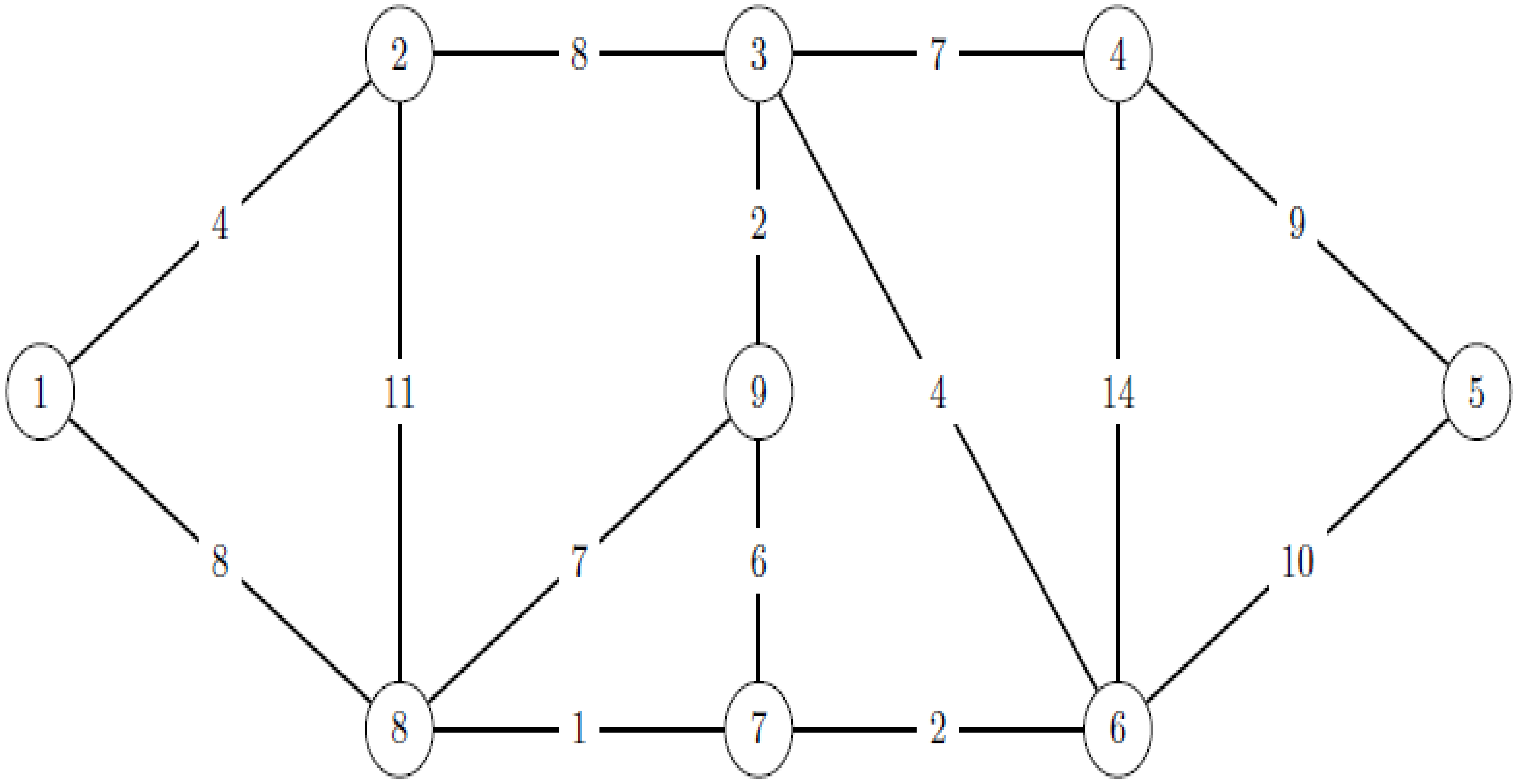
4- Trees and arborescences

c) Kruskal's Algorithm (homework)



4- Trees and arborescences

c) Kruskal's Algorithm (homework)



4- Trees and arborescences

d) Kruskal's Algorithm (details)

Principle:

- Number the edges in ascending order of their weights.
- Build the tree in the order of the edges.
- Add an edge without forming a cycle.

Algorithm:

Input: G = (V, E, C) an undirected weighted graph.

Output: The set of edges W.

4- Trees and arborescences

d) Kruskal's Algorithm(Algorithm)

```
(0): Initialization
Number the edges C(e1) ≤ C(e2) ≤ ... ≤ C(en); W ← Ø; I ← 1
(1): If (V, W U {ei}) forms a cycle, then    go to (3)
       else,    go to (2)
(2): W = W U {ei}    and go to (3)
(3): If i = m, then terminate (stop)   A = (V, W) minimum weight tree
     else, i ← i + 1    and    go to (1)
```

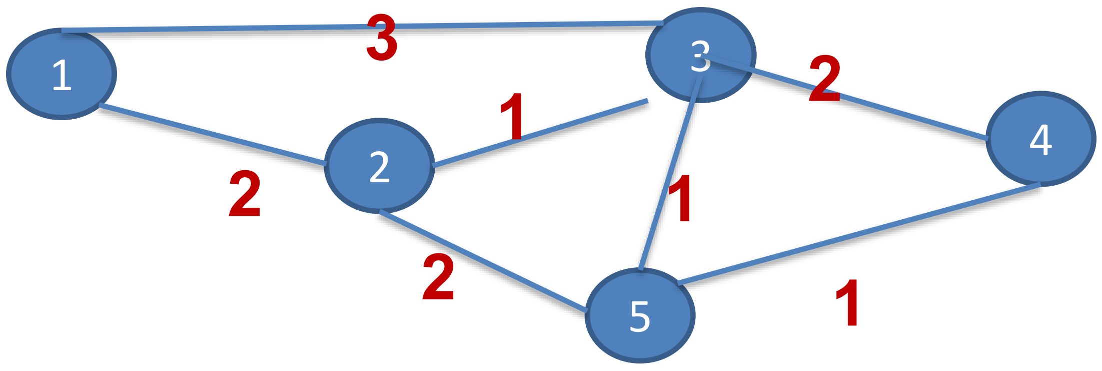
Note:

The algorithm terminates when the number of edges is equal to n-1.

To find a **maximum-weight** tree, use the same algorithm by multiplying the edge costs by **-1**.

4- Trees and arborescences

d) Kruskal's Algorithm (Application): Minimizing the Cost of Telephone Installation.



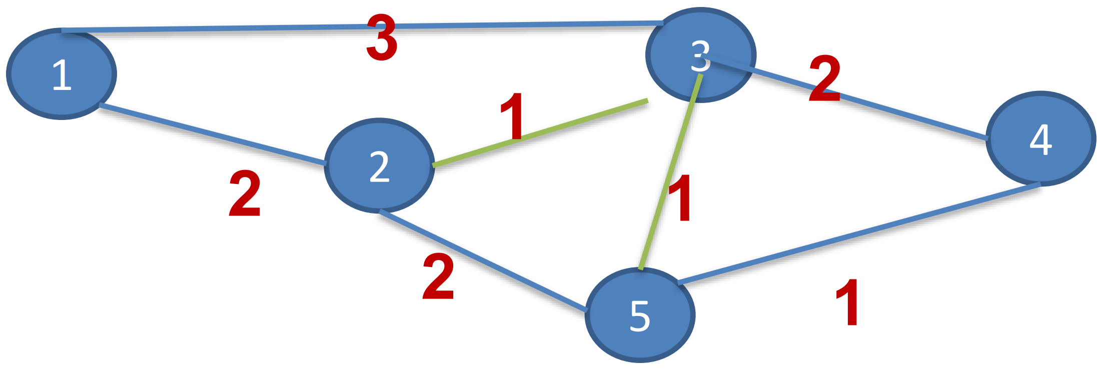
(0): initialisation:
We order the edges
W ← ∅; I ← 1
n=5 , m= 7

i	1	2	3	4	5	6	7
ei	(2,3)	(3,5)	(4,5)	(3,4)	(2,5)	(1,2)	(1,3)
C(ei)	1	1	1	2	2	2	3

Itération 1: e1=(2,3) W= W U {e1} = {(2,3)}
The graph (V,W) does not contain a cycle, I≠ m ➔ i = i+1 = 2

4- Trees and arborescences

d) Kruskal's Algorithm (Application): Minimizing the Cost of Telephone Installation.



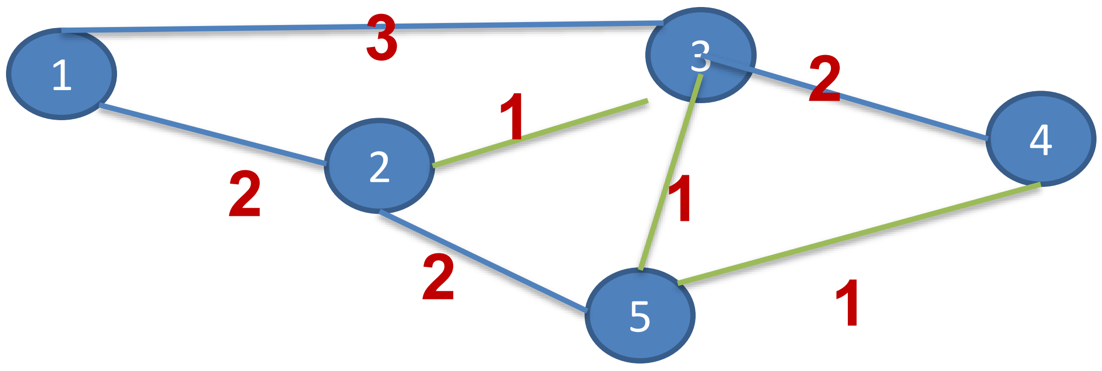
i	1	2	3	4	5	6	7
ei	(2,3)	(3,5)	(4,5)	(3,4)	(2,5)	(1,2)	(1,3)
C(ei)	1	1	1	2	2	2	3

Itération 2: e2=(3,5) W= W U {e2} = {(2,3), (3,5)}

The graph (V,W) does not contain a cycle, I≠ m ➔ i = i+1 = 3

4- Trees and arborescences

d) Kruskal's Algorithm (Application): Minimizing the Cost of Telephone Installation.

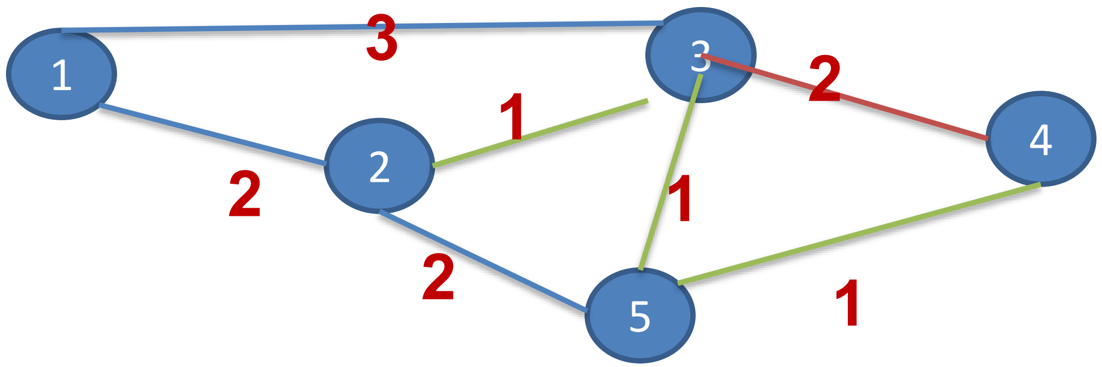


i	1	2	3	4	5	6	7
ei	(2,3)	(3,5)	(4,5)	(3,4)	(2,5)	(1,2)	(1,3)
C(ei)	1	1	1	2	2	2	3

Itération 3: e3=(4,5) W= W U {e3} = {(2,3), (3,5),(4,5)}
The graph (V,W) does not contain a cycle, I≠ m ➔ i = i+1 = 4

4- Trees and arborescences

d) Kruskal's Algorithm (Application): Minimizing the Cost of Telephone Installation.

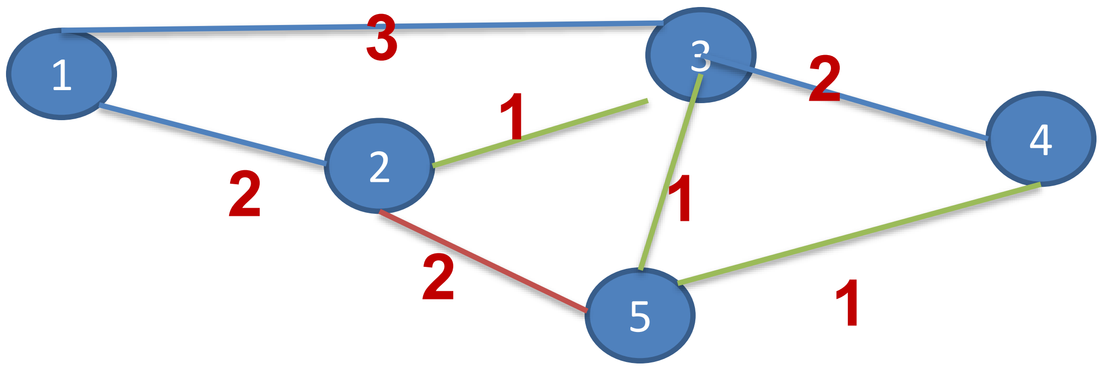


i	1	2	3	4	5	6	7
ei	(2,3)	(3,5)	(4,5)	(3,4)	(2,5)	(1,2)	(1,3)
C(ei)	1	1	1	2	2	2	3

Itération 4: e4=(3,4) W= W U {e4} = {(2,3), (3,5),(4,5),(3,4)}
The graph (V, W) contains a cycle, I≠ m ➔ i = i+1 = 5

4- Trees and arborescences

d) Kruskal's Algorithm (Application): Minimizing the Cost of Telephone Installation.

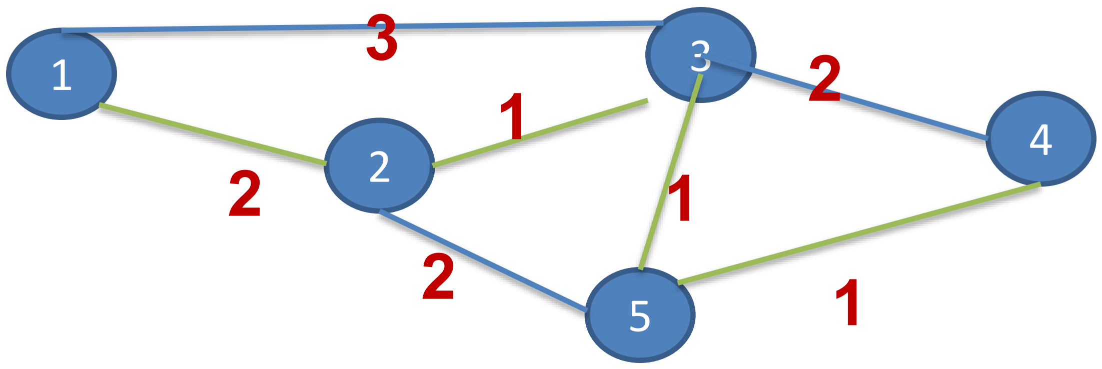


i	1	2	3	4	5	6	7
ei	(2,3)	(3,5)	(4,5)	(3,4)	(2,5)	(1,2)	(1,3)
C(ei)	1	1	1	2	2	2	3

Itération 5: e5=(2,5) W= W U {e5} = {(2,3), (3,5),(4,5),(2,5)}
The graph (V, W) contains a cycle, l≠ m ➔ i = i+1 = 6

4- Trees and arborescences

d) Kruskal's Algorithm (Application): Minimizing the Cost of Telephone Installation.



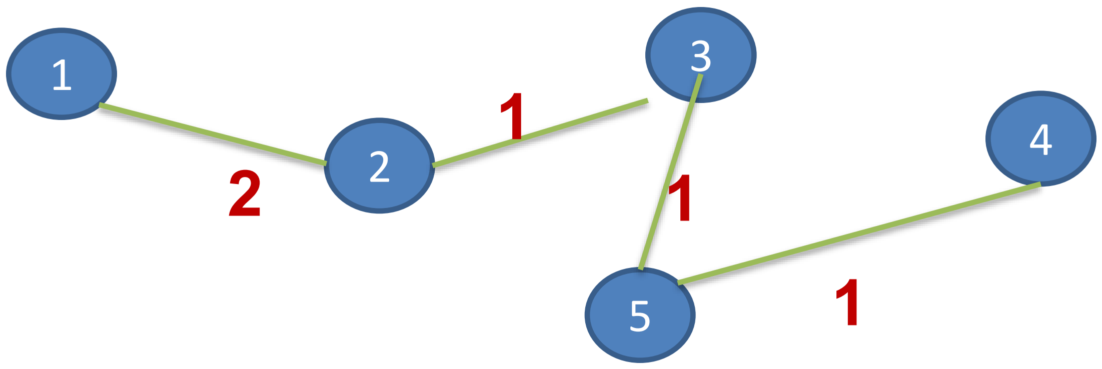
i	1	2	3	4	5	6	7
ei	(2,3)	(3,5)	(4,5)	(3,4)	(2,5)	(1,2)	(1,3)
C(ei)	1	1	1	2	2	2	3

Itération 6: e6=(1,2) W= W U {e6} = {(2,3), (3,5),(4,5),(1,2)}
The graph (V,W) does not contain a cycle , I≠ m ➔ **|W|=4=n-1**
Stop, the total installation cost= 1+1+1+2=5

4- Trees and arborescences

d) Kruskal's Algorithm (Application): Minimizing the Cost of Telephone Installation.

.



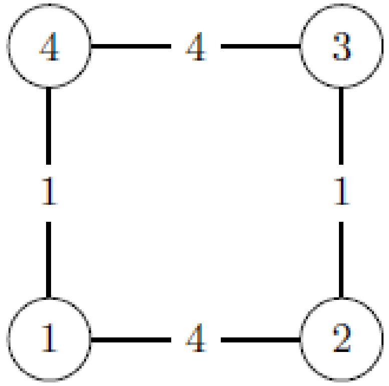
Stop: We have a loop with two conditions: l≠ m and |W|= n-1
Stop, the total installation cost = 1+1+1+2=5

4- Trees and arborescences

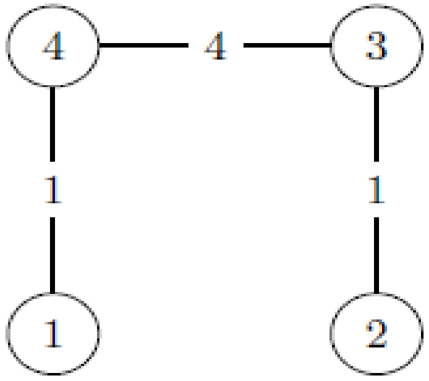
e) Uniqueness of the spanning tree:

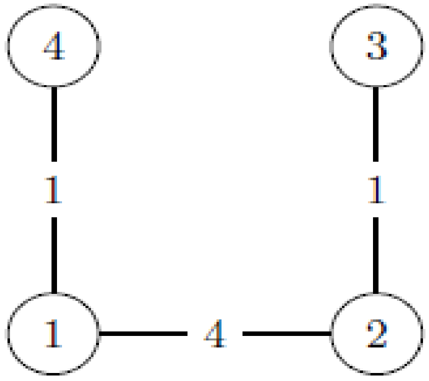
In a graph, there can be multiple different minimum-cost spanning trees.

Consider the following graph:



G has 2 different minimum-cost spanning trees.

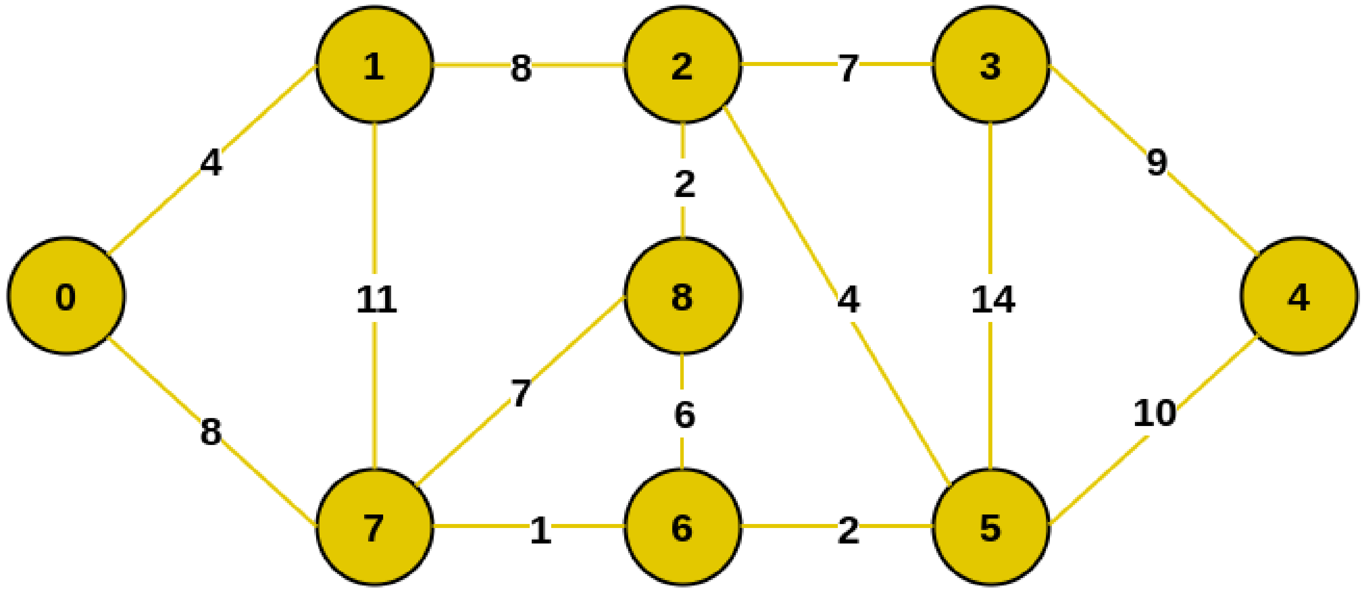




4- Trees and arborescences

f) Prim's Algorithm:

Similar to Kruskal's algorithm, we start with an arbitrary **vertex** and construct a spanning tree by choosing a minimum-cost edge among the edges **adjacent** to this vertex without forming a cycle.



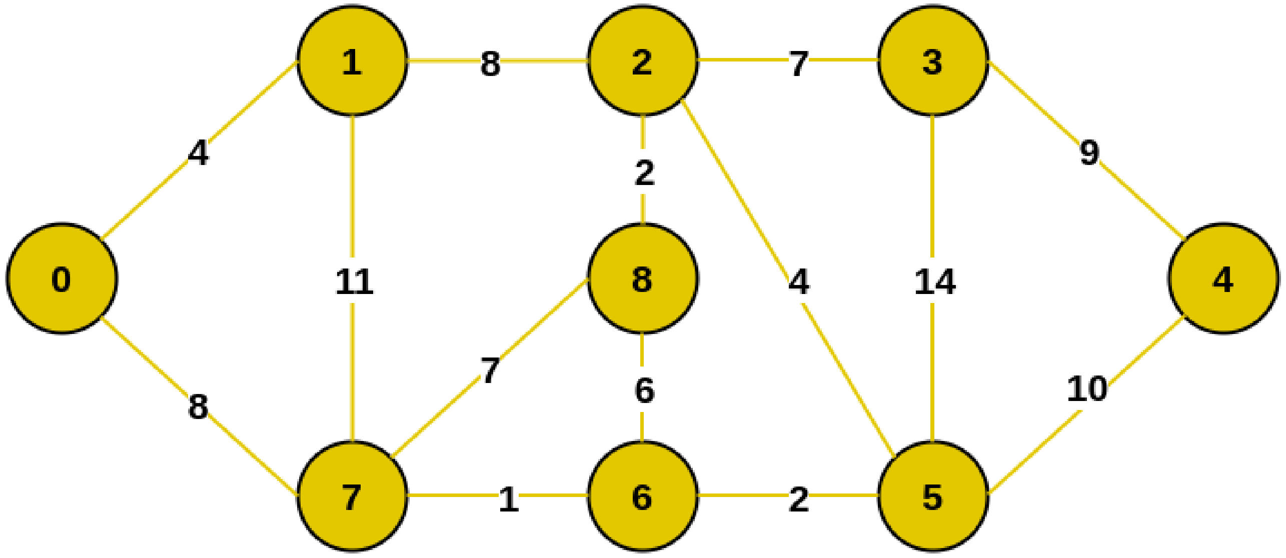
Example of a Graph

4- Trees and arborescences

f) Prim's Algorithm application:

Given a network of cities connected by roads, and each road has an associated cost (distance, travel time, or any other metric).

Your task is to find the minimum-cost way to connect all cities, ensuring that every city is reachable from any other city. This can be thought of as building a network of roads while minimizing the total cost.

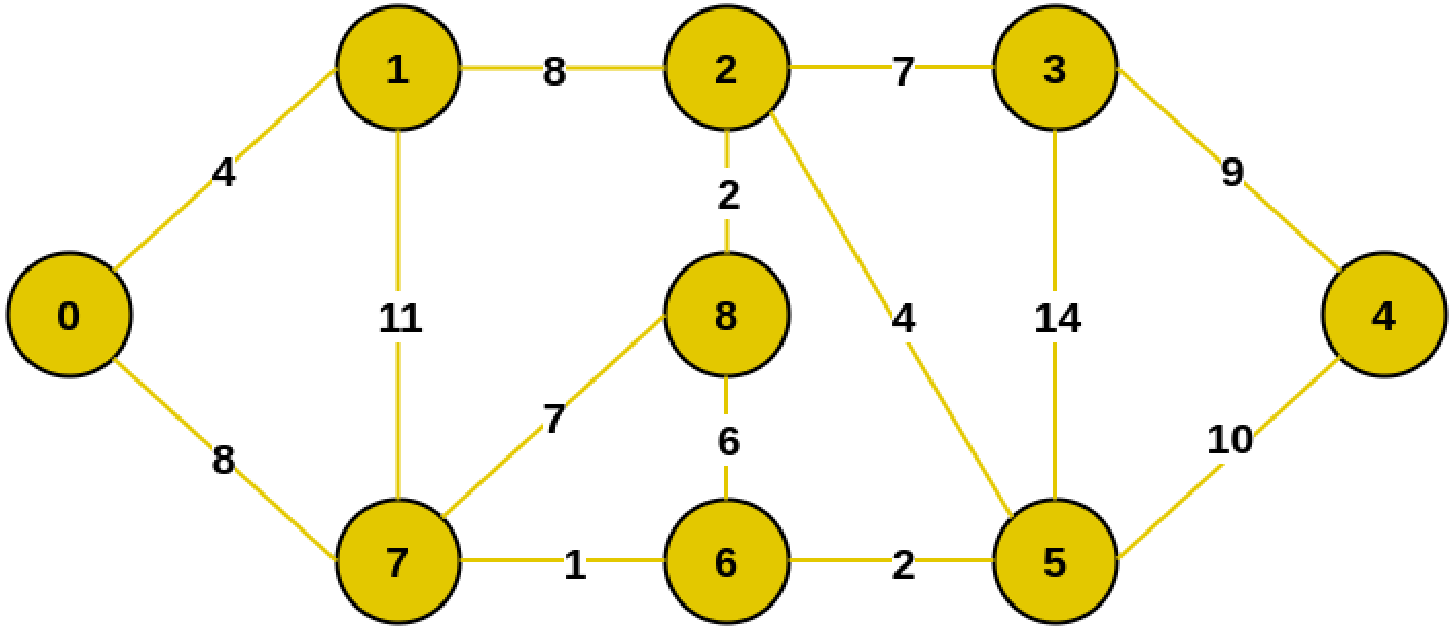


Example of a Graph

4- Trees and arborescences

f) Prim's Algorithm application:

Consider the following graph as an example for which we need to find the Minimum Spanning Tree (MST).

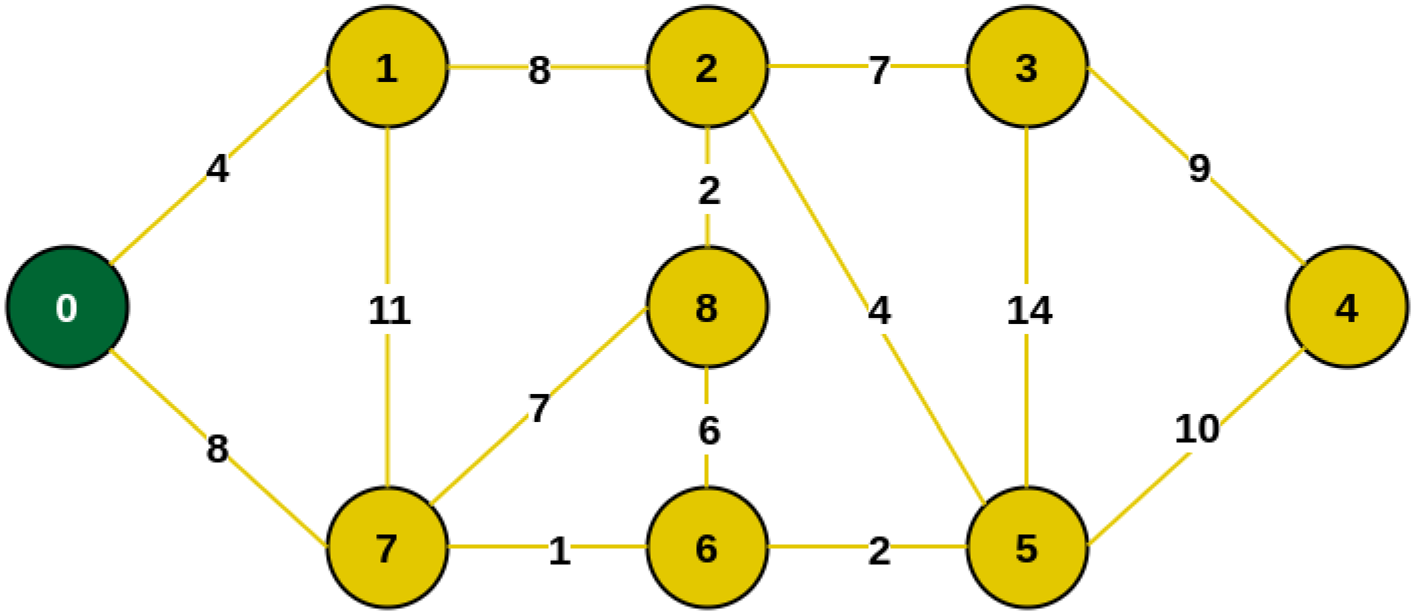


Example of a Graph

4- Trees and arborescences

f) <u>Prim's Algorithm</u> application:

Step 1: Firstly, we select an **arbitrary** vertex that acts as the starting vertex of the Minimum Spanning Tree. Here we have selected **vertex 0** as the starting vertex.

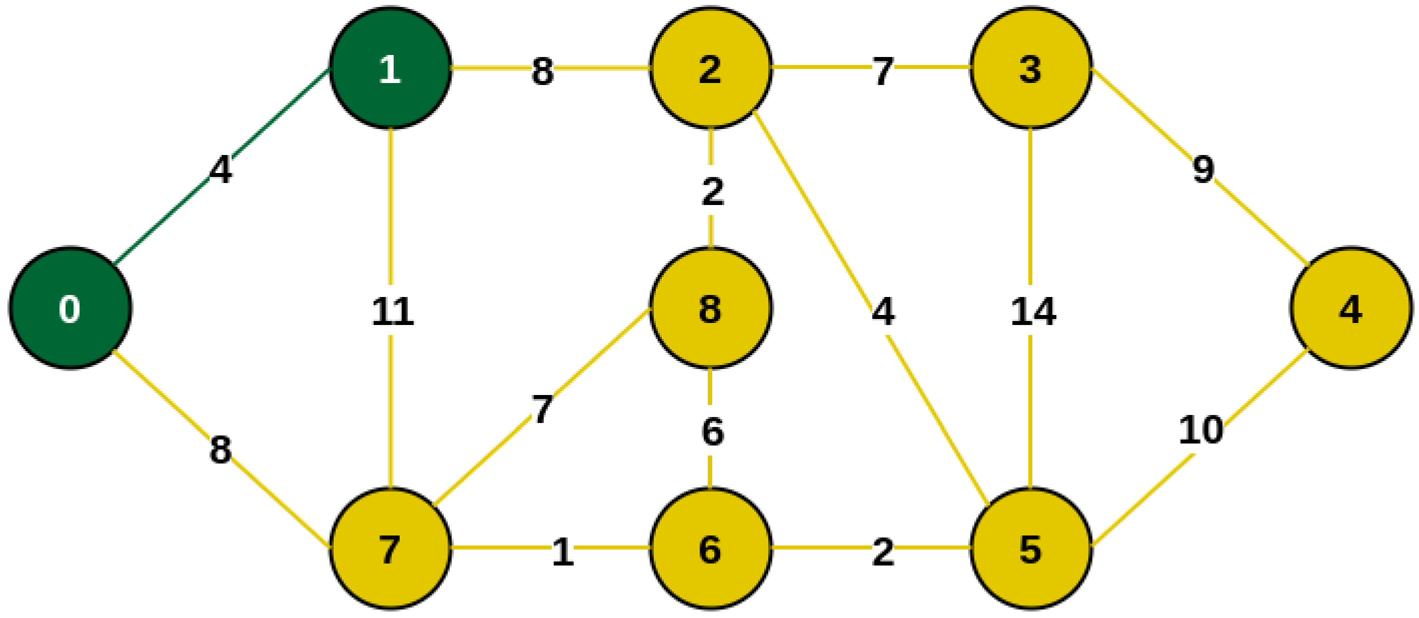


Select an arbitrary starting vertex. Here we have selected 0

4- Trees and arborescences

f) Prim's Algorithm application:

Step 2: All the edges connecting the incomplete MST and other vertices are the edges **{0, 1} and {0, 7}.** Between these two the edge with minimum weight is **{0, 1}**. So include the edge and vertex 1 in the MST.

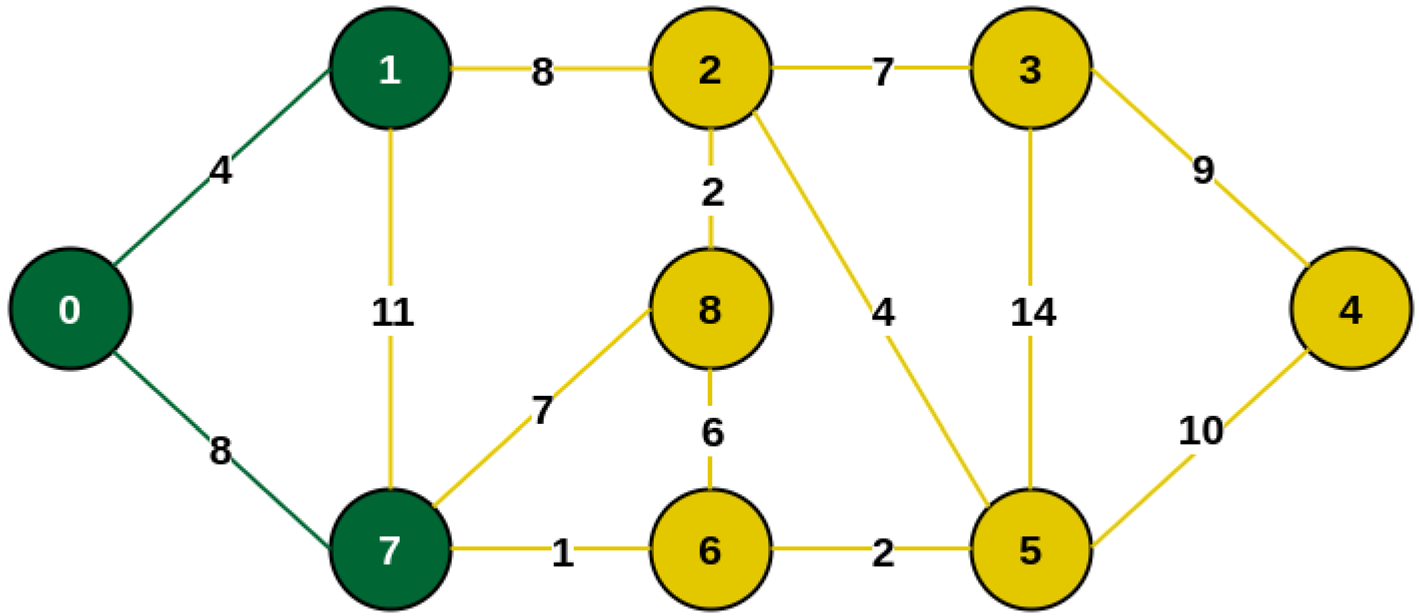


Minimum weighted edge from MST to other vertices is 0-1 with weight 4

4- Trees and arborescences

f) Prim's Algorithm application:

Step 3: The edges connecting the incomplete MST to other vertices are **{0, 7}, {1, 7} and {1, 2}.** Among these edges the minimum weight is 8 which is of the edges {0, 7} and {1, 2}. Let us here include the edge {0, 7} and the vertex 7 in the MST. **[We could have also included edge {1, 2} and vertex 2 in the MST].**

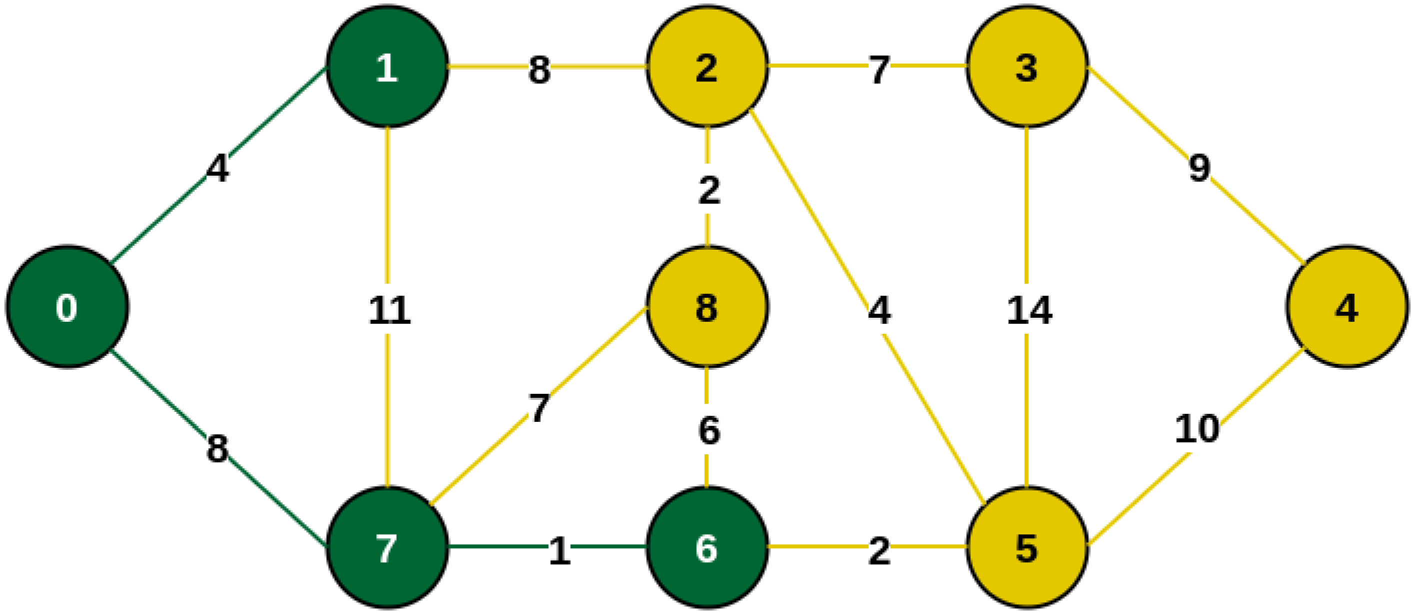


Minimum weighted edge from MST to other vertices is 0-7 with weight 8

4- Trees and arborescences

f) Prim's Algorithm application:

Step 4: The edges that connect the incomplete MST with the fringe vertices are **{1, 2}, {7, 6} and {7, 8}.** Add the edge **{7, 6}** and the vertex 6 in the MST.

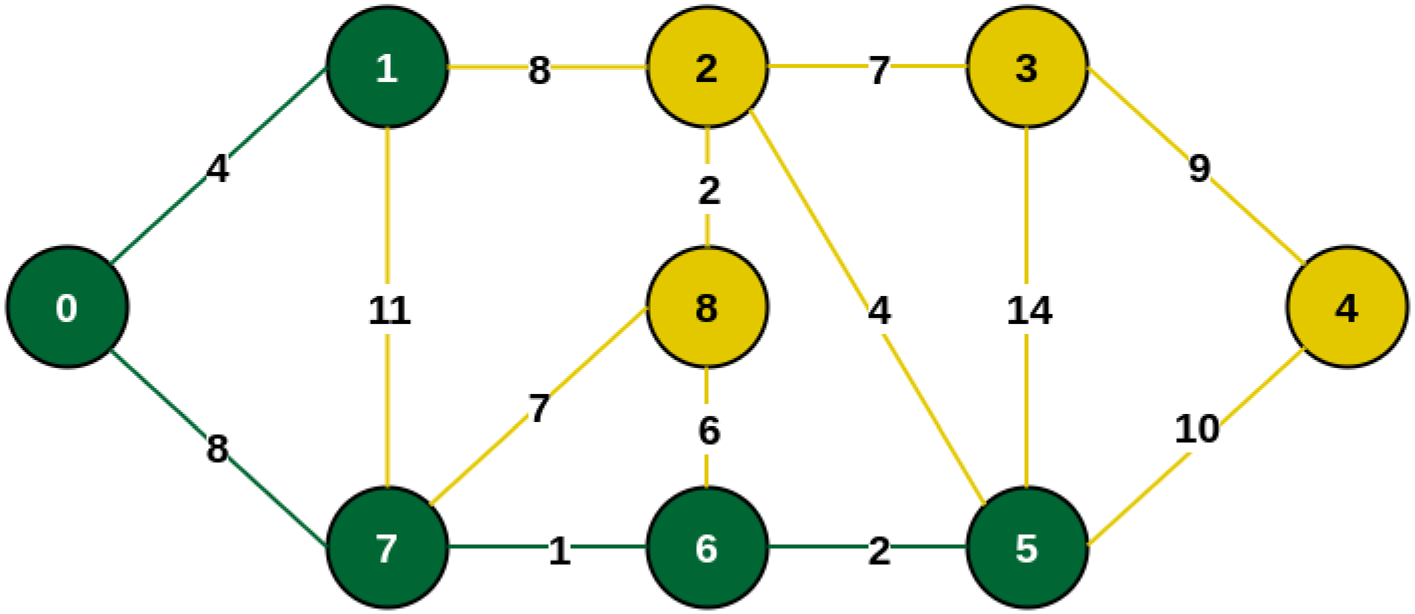


Minimum weighted edge from MST to other vertices is 7-6 with weight 1

4- Trees and arborescences

f) Prim's Algorithm application:

Step 5: The connecting edges now are **{7, 8}, {1, 2}, {6, 8} and {6, 5}.** Include edge **{6, 5}** and vertex **5** in the MST as the edge has the minimum weight.

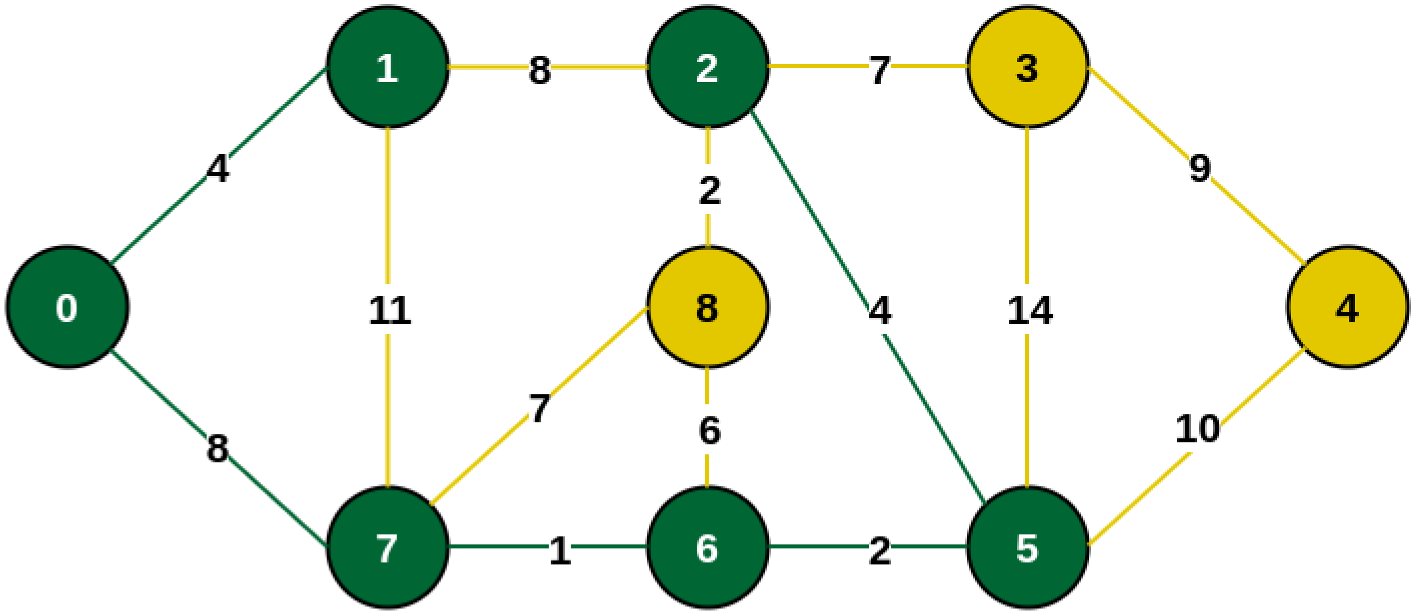


Minimum weighted edge from MST to other vertices is 6-5 with weight 2

4- Trees and arborescences

f) Prim's Algorithm application:

Step 6: Among the current connecting edges, the edge **{5, 2}** has the minimum weight. So include that edge and the vertex 2 in the MST.

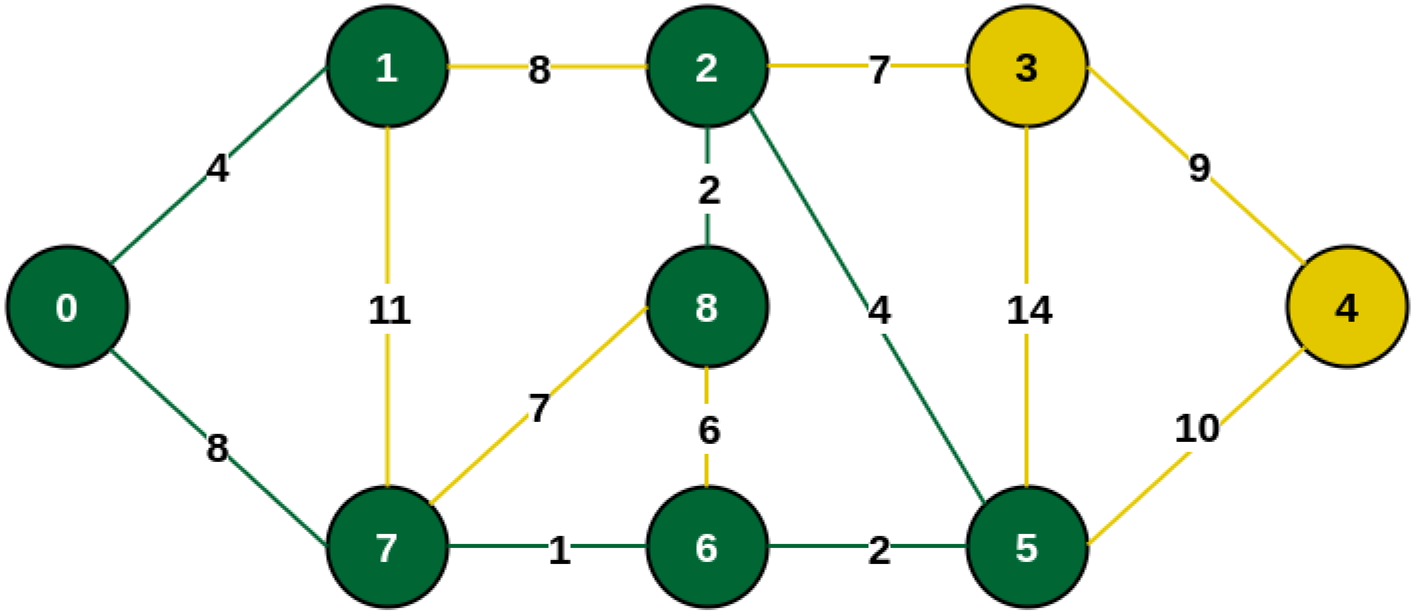


Minimum weighted edge from MST to other vertices is 5-2 with weight 4

4- Trees and arborescences

f) Prim's Algorithm application:

Step 7: The connecting edges between the incomplete MST and the other edges are **{2, 8}, {2, 3}, {5, 3} and {5, 4}.** The edge with minimum weight is edge **{2, 8}** which has weight 2. So include this edge and the vertex 8 in the MST.

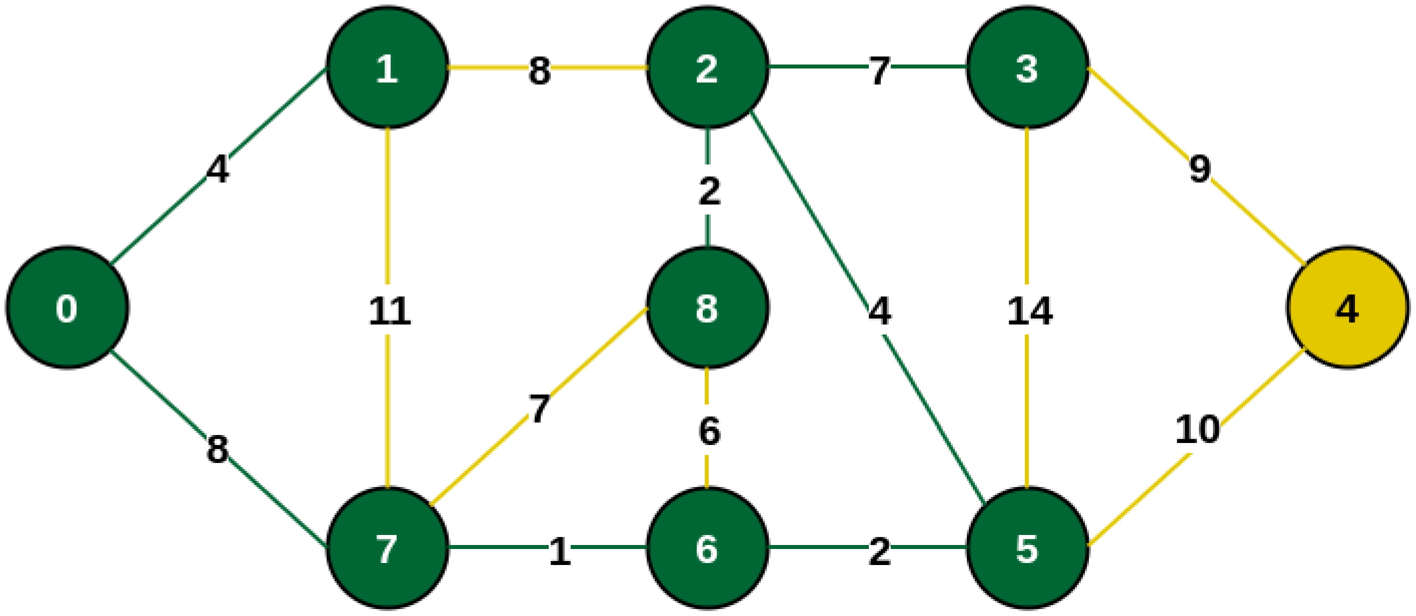


Minimum weighted edge from MST to other vertices is 2-8 with weight 2

4- Trees and arborescences

f) Prim's Algorithm application:

Step 8: See here that the edges **{7, 8} and {2, 3}** both have same weight which are minimum. But 7 is already part of MST. So we will consider the edge {2, 3} and include that edge and vertex 3 in the MST.

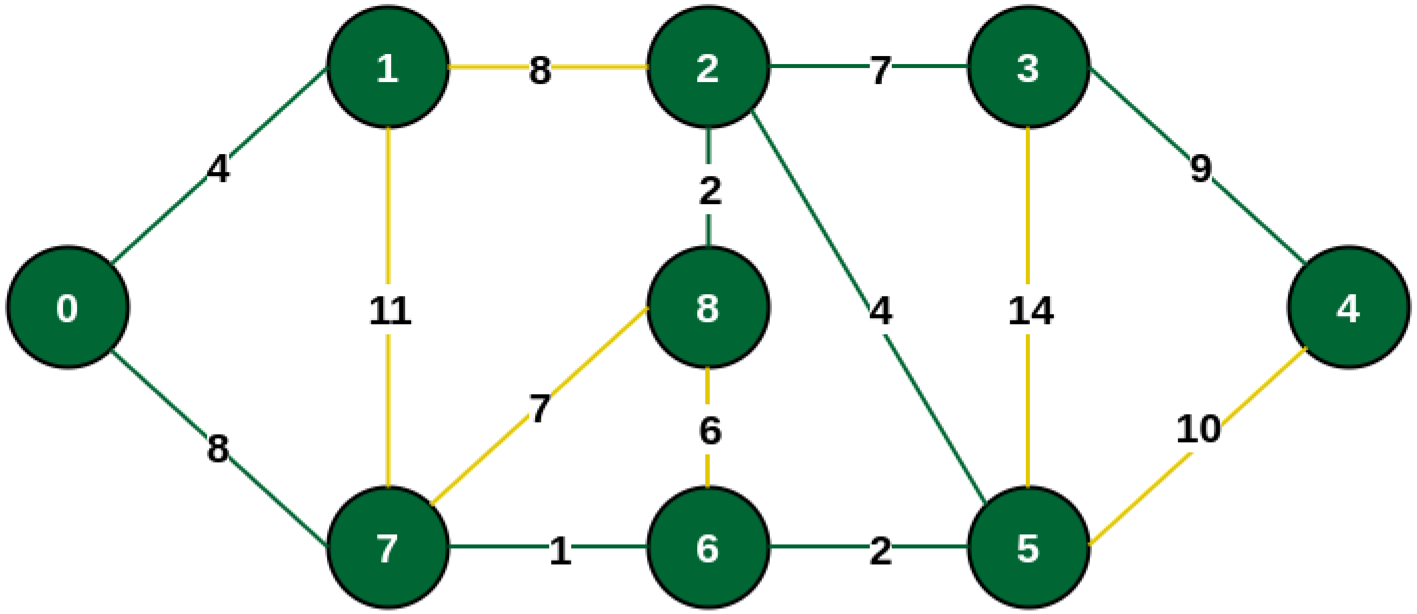


Minimum weighted edge from MST to other vertices is 2-3 with weight 7

4- Trees and arborescences

f) Prim's Algorithm application:

Step 9: Only the vertex 4 remains to be included. The minimum weighted edge from the incomplete MST to 4 is **{3, 4}.**

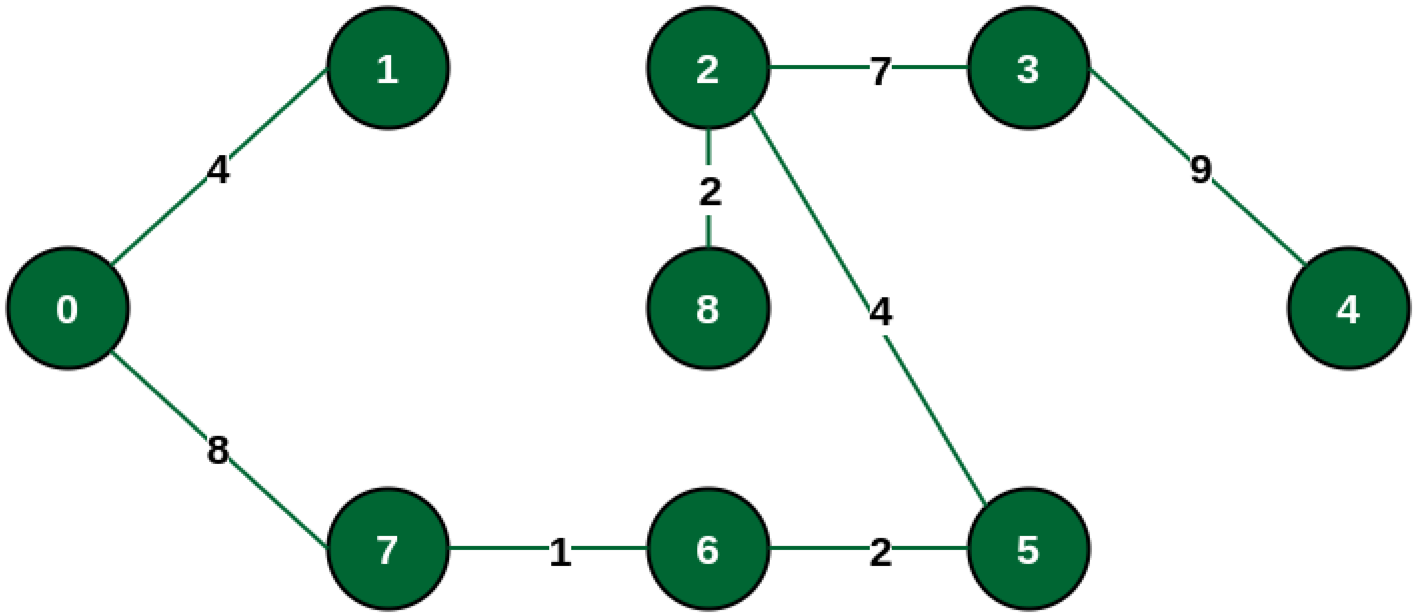


Minimum weighted edge from MST to other vertices is 3-4 with weight 9

4- Trees and arborescences

f) Prim's Algorithm application:

The final structure of the MST is as follows and the weight of the edges of the MST is **(4 + 8 + 1 + 2 + 4 + 2 + 7 + 9) = 37.**

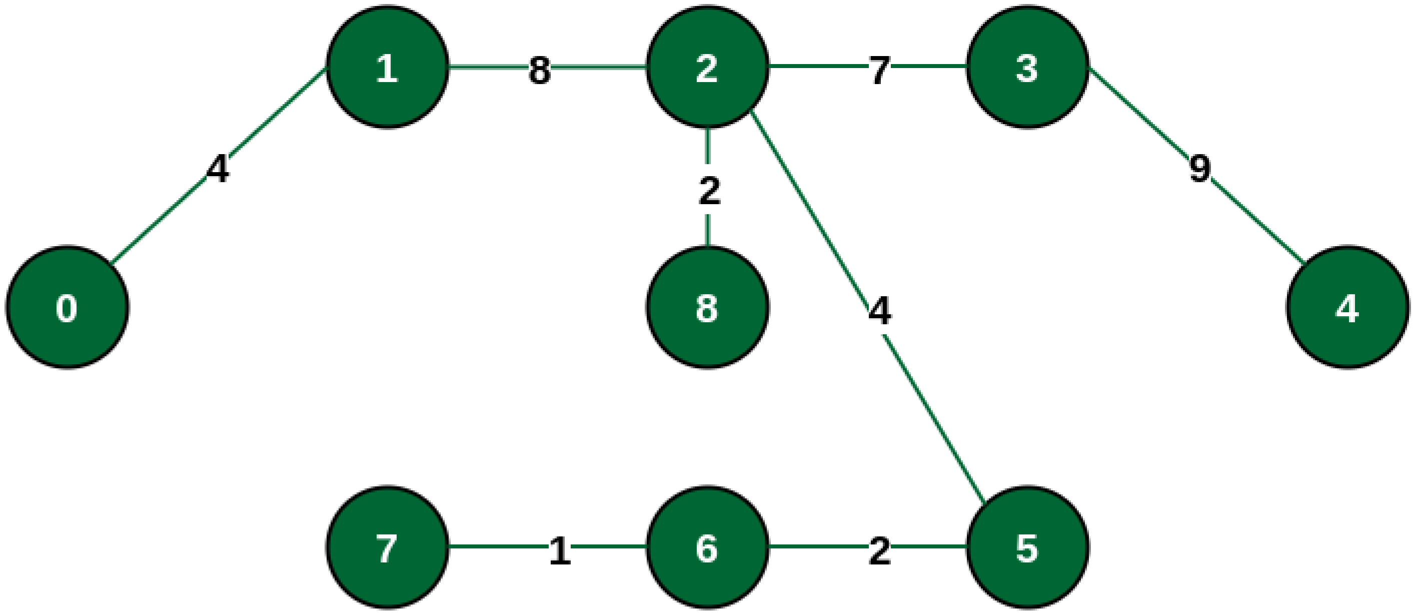


The final structure of MST

4- Trees and arborescences

f) Prim's Algorithm application:

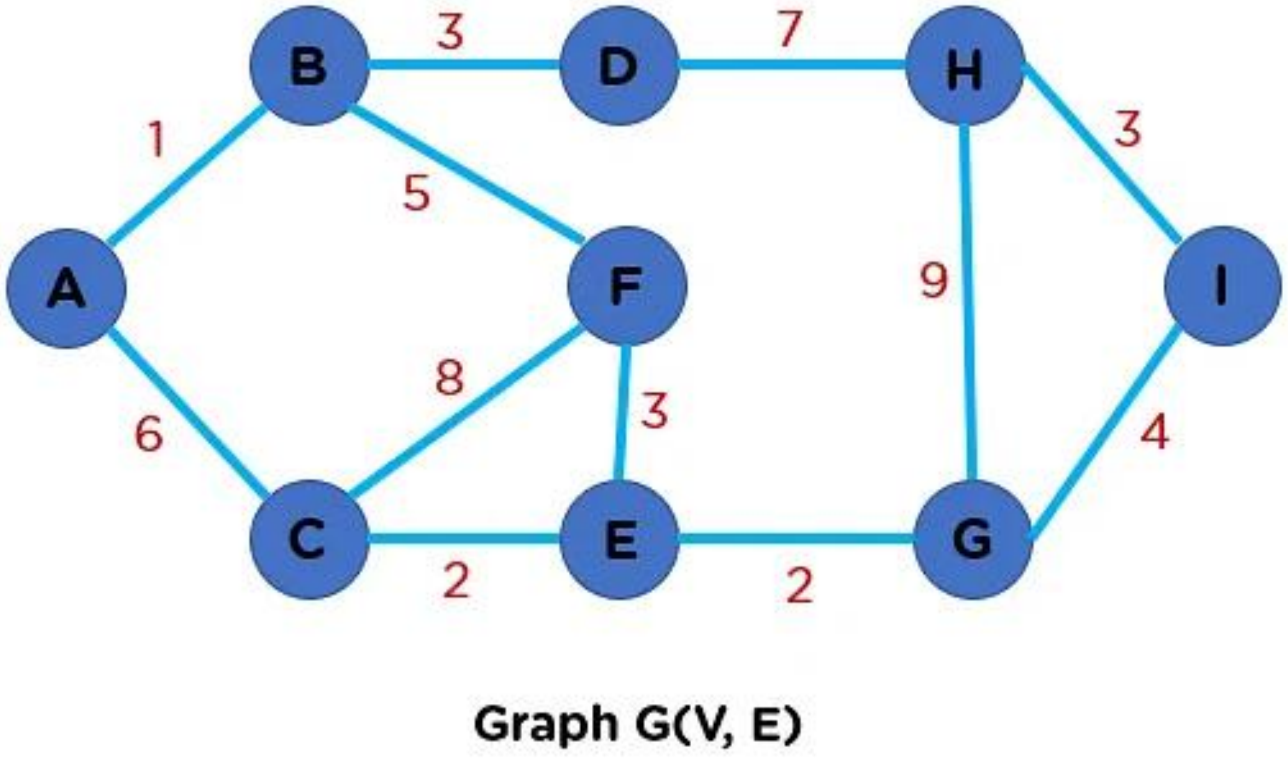
Note: If we had selected the edge **{1, 2}** in **the third step** then the MST would look like the following.



Alternative MST structure

4- Trees and arborescences

f) Prim's Algorithm other example(homework):



Graph G(V, E)

4- Trees and arborescences

g) Some exercises:

Exercise 1: Properties of MST

State and explain three important properties of a Minimum Spanning Tree (MST).

Solution:

Three important properties of a Minimum Spanning Tree (MST) are:

1- It is a spanning tree, which means it includes all vertices of the graph.

2- It is acyclic, ensuring that no cycles are formed.

3- It has the minimum total edge weight among all possible spanning trees for the graph.

4- Trees and arborescences

g) Some exercises:

Exercise 2: Unique MST

Under what conditions is a weighted graph guaranteed to have a unique Minimum Spanning Tree? Provide a proof or explanation.

Solution:

A weighted graph is guaranteed to have a unique MST when all its edge weights are distinct. This uniqueness is because there is only one way to construct the MST with the minimum total weight when all edge weights are distinct.

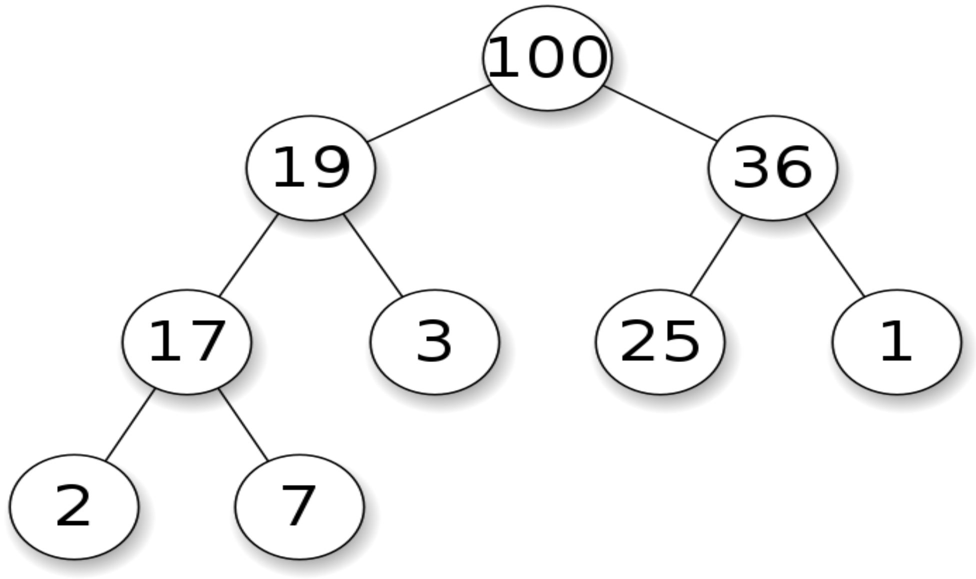
4- Trees and arborescences

Difference between Prims and Kruskal Algorithm

Prim	Kruskal
begins to construct the shortest spanning tree from any vertex in the graph	begins to construct the shortest spanning tree from the vertex having the lowest weight in the graph.
it traverses one node more than one time.	It crosses one node only one time.
The time complexity of Prim's algorithm is $O(n^2)$.	The time complexity of Kruskal's algorithm is $O(m \log n)$.
all the graph elements must be connected.	may have disconnected graphs.
When it comes to dense graphs, the Prim's algorithm runs faster.	When it comes to sparse graphs, Kruskal's algorithm runs faster
Il prefers list data structure	It prefers the heap data structure (complete binary tree and ordered)

An example of a heap. (Tas)

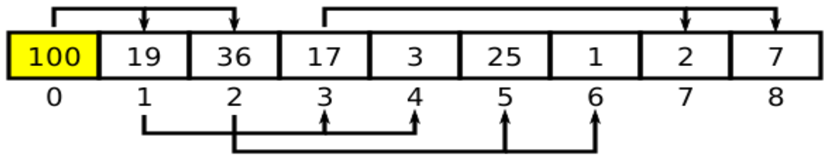
Tree representation



It contains 9 elements.

The highest-priority element (100) is at the root.

Array representation


100
19
36
17
3
25
1
2
7
0
1
2
3
4
5
6
7
8