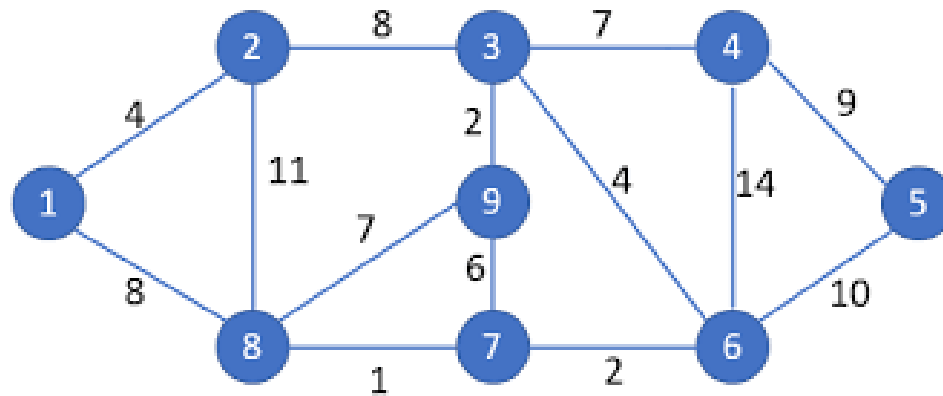
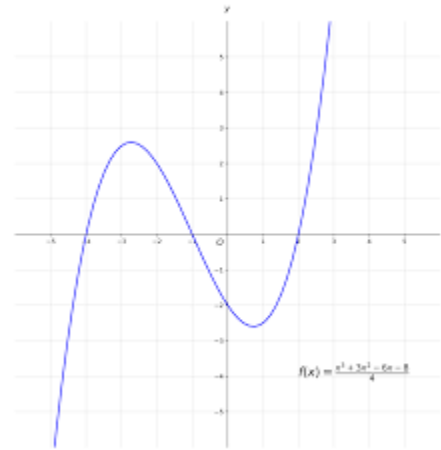


EE 1004 Foundation of Information Systems and Data Analysis

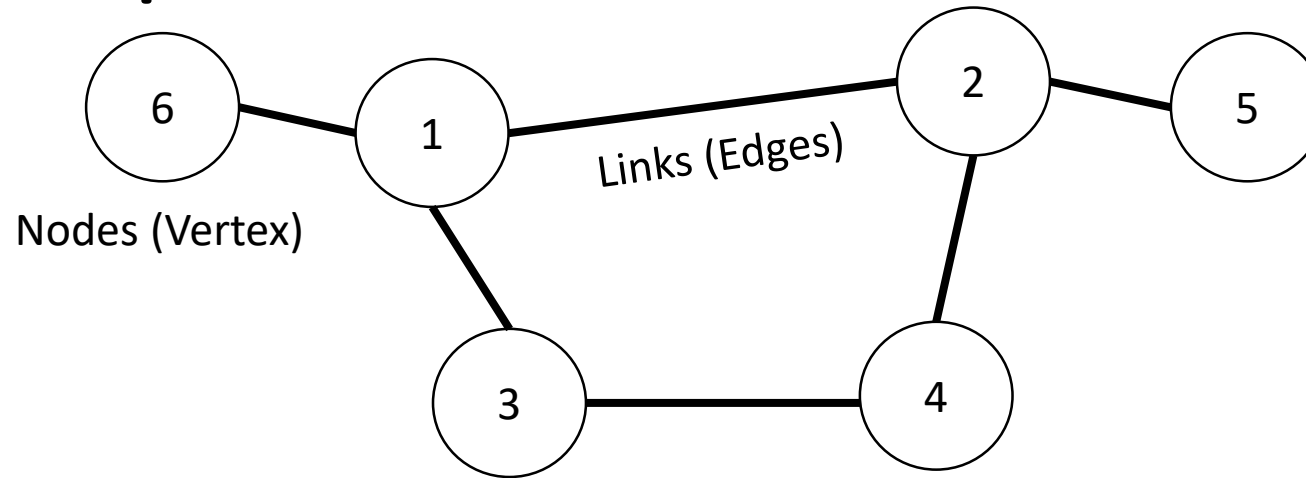
Part II

Chapter 7: Introduction of Graph Theory

What's Graph



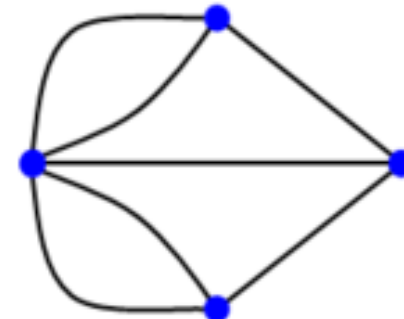
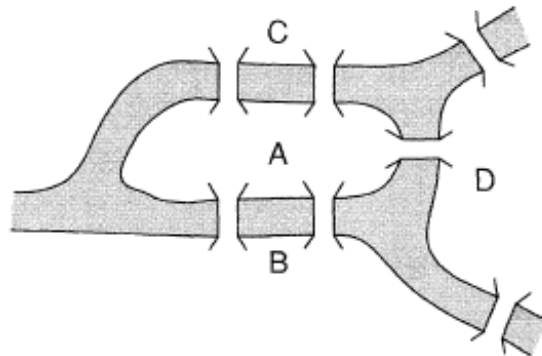
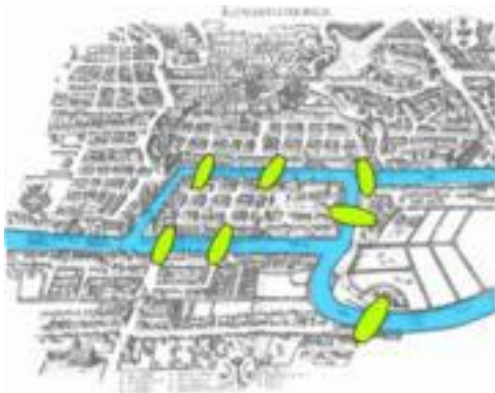
What's Graph



- Graph is a network that helps define and visualize relationships (**links**) between various components (**nodes**)
- Nodes: Locations, people, atoms...
- Links: Distance, cost, delay measures, strength of relationship...
- Many Practical problems can be simplified and represented by graphs

Historical background of Graph

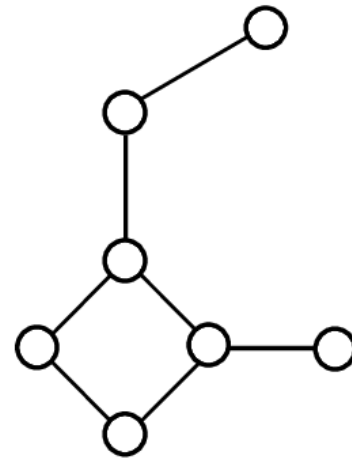
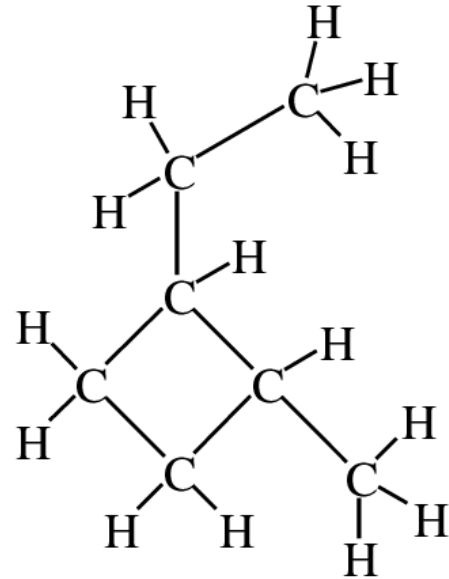
- In 1735, Leonhard Euler solved the Konigsberg Bridge Problem
- The possibility of finding a path over every one of seven bridges but without crossing any bridge twice
- First theorem in graph theory



Leonhard Euler
(1707 - 1783)

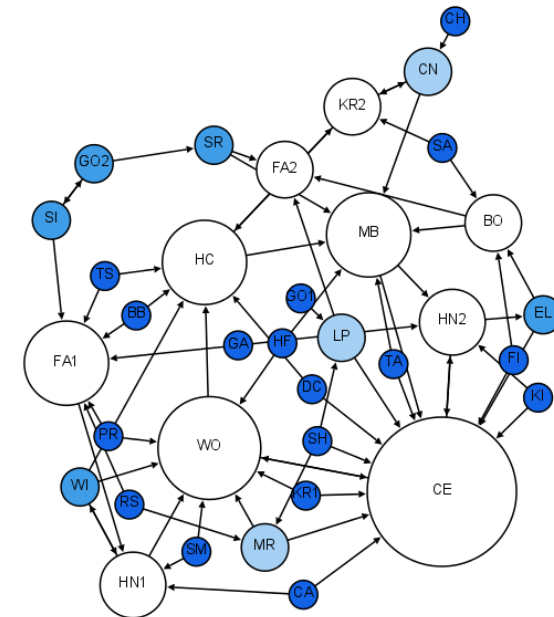
Practical Use Case of Graph

Chemistry
(Compositions)



1-ethyl-2-methylcyclobutane and the
corresponding hydrogen-depleted molecular

Social Science
(Sociogram)



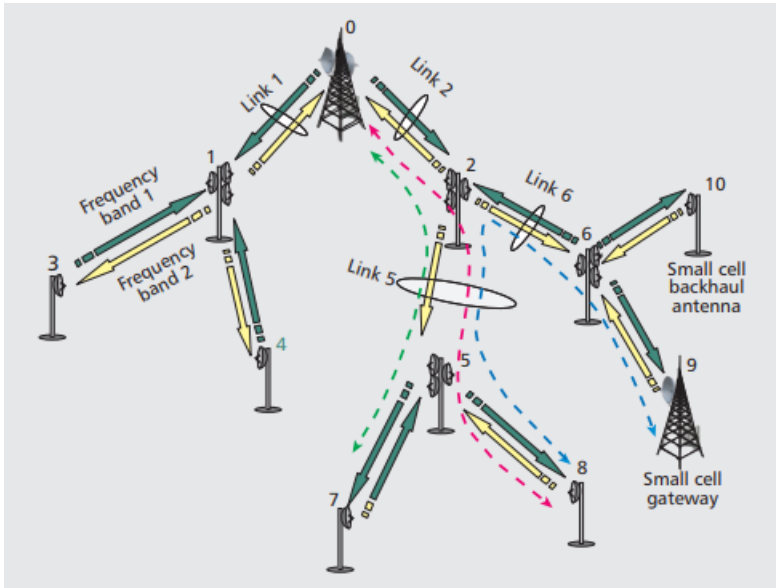
Moreno's Sociogram

Source:

- Janežič, Dušanka & Miličević, Ante & Nikolić, Sonja & Trinajstić, Nenad. (2007). Graph Theoretical Matrices in Chemistry. 10.1201/b18389.
- Brown, D. R., & Harvey, D. F. (2011). An experiential approach to organization development.

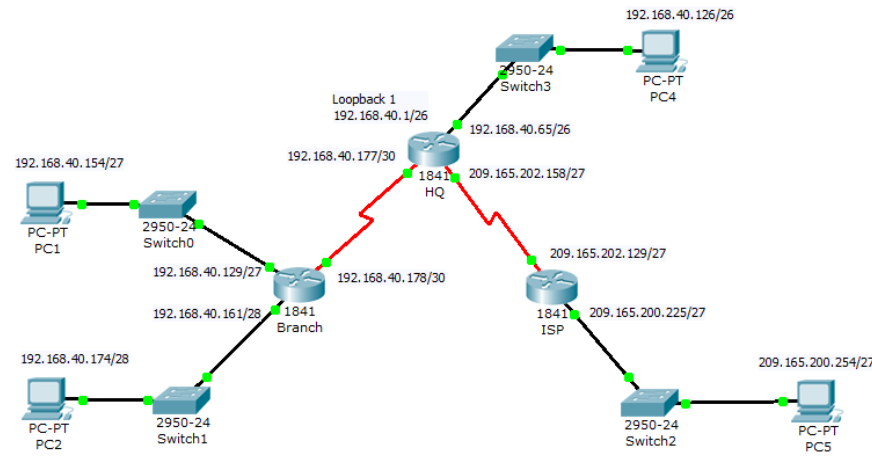
Practical Use Case of Graph (related to EE)

Operations Research
(Wireless Network)



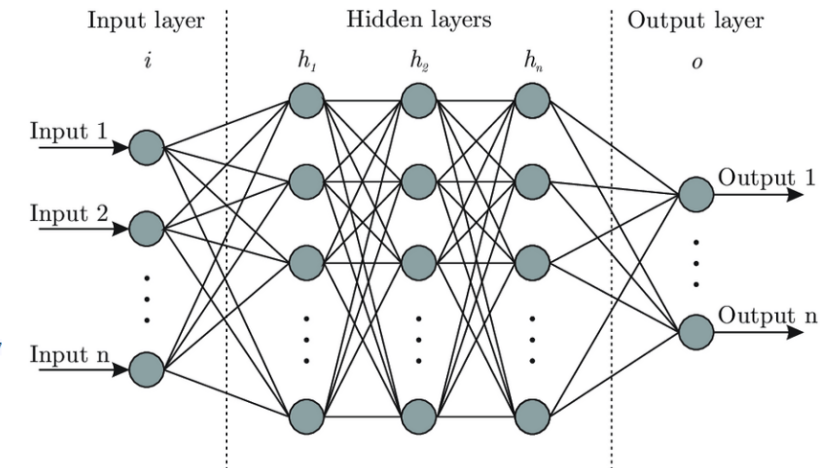
Graph-based Wireless Communication Network

Information Engineering
(Network Architecture)



Routing Protocol in networking

Computer Science
(Machine-Learning)



Feedforward Neural Network

Source:

- Ni, W., Collings, I.B., Lipman, J., Wang, X., Tao, M., & Abolhasan, M. (2015). Graph theory and its applications to future network planning: software-defined online small cell management. IEEE Wireless Communications, 22, 52-60.
- <https://computingforgeeks.com/what-is-ip-routing-and-routing-protocols-in-networking/>
- <https://towardsdatascience.com/graph-theory-and-deep-learning-know-hows-6556b0e9891b>

Content

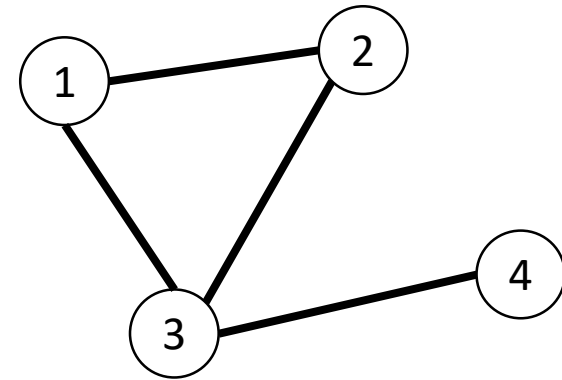
- Graph Terminologies
- Graph Classification
- Type of Graph
- Graph Measurement
- Shortest Path Problem and Algorithm
- Graph Representation

Graph Terminologies

- Vertex & Edge
- Order & Size
- Degree
- Neighborhood/Adjacency
- Connected and Isolated nodes (Connectivity)

Vertex & Edge

- A simple graph $G = \{V, E\}$ is a set of **vertices** $V(G)$ and **edges** $E(G)$ where each edge (u, v) is a connection between vertices, $u, v \in V(G)$
- *Given* $G = \{V, E\}$,
- $V(G) = \text{set of Vertices} = V = \{1, 2, 3, 4\}$
- $E(G) = \text{set of Edges} = E = \{(1, 2), (1, 3), (2, 3), (3, 4)\}$

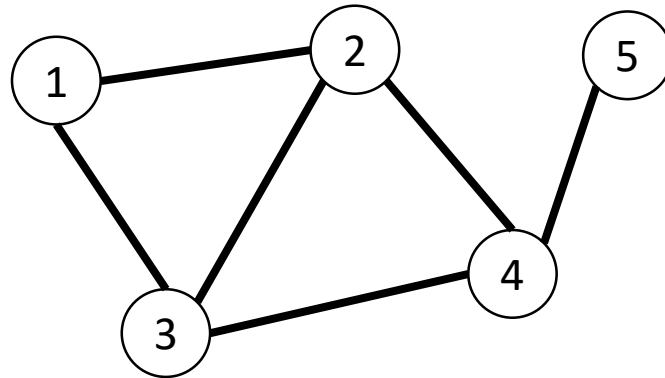


Order & Size

- **Order** = number of vertices of G , denoted by $|V|$

- **Size** = number of edges of G , denoted by $|E|$

- Example:



- $V = \{1,2,3,4,5\}$

- $E = \{(1,2), (1,3), (2,3), (2,4), (3,4), (4,5)\}$

- *such that* $|V| = 5, |E| = 6$

- Note that: if the order is given as $|V| = n$, then the size of G can be found within the range of:

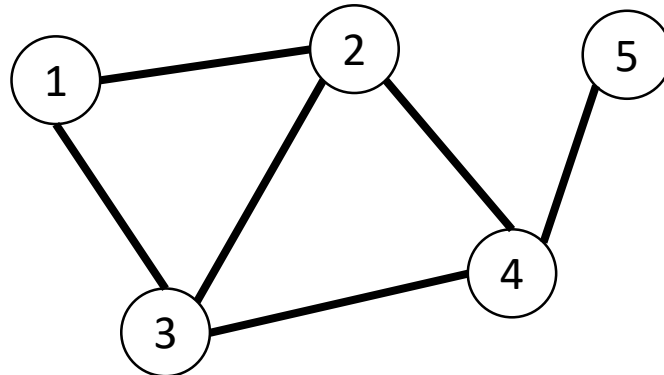
$$0 \leq |E| \leq C_2^n = \frac{n!}{2!(n-2)!} = \frac{n(n-1)}{2}$$

Degree

- **Degree** of vertices = number of edges connected to a vertex v
- **Total Degree** = the sum of degrees of all vertices/nodes in a graph
- Note that: Consider a **SIMPLE** graph,

$$\text{Total Degree} = \sum_{v_i \in V} \text{degree}(v_i) = 2|E|$$

- Example:

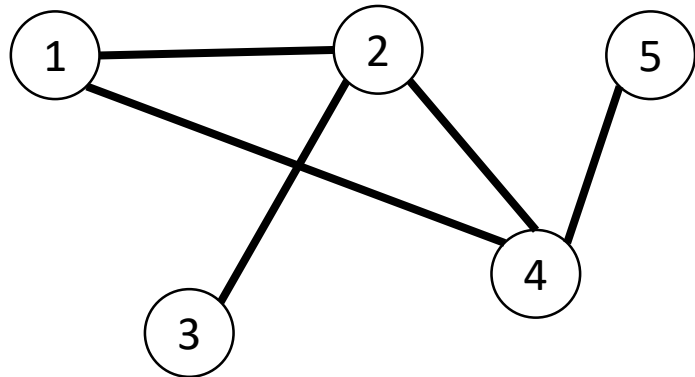


- For node 1, $E = \{(1,2), (1,3)\}$
- $\text{Degree}(1) = 2$
- For node 2, $E = \{(1,2), (2,3), (2,4)\}$
- $\text{Degree}(2) = 3$
- How about nodes 3, 4, 5?

Neighborhood/Adjacency

- **Neighbors** = the set of vertices connected to a vertex v of $G = \{V, E\}$ via an edge, denoted by $N(v)$
- Given a vertex subset of $A \subset V(G)$, $N(A)$ = the set of all vertices in G that are adjacent to at least one vertex in A

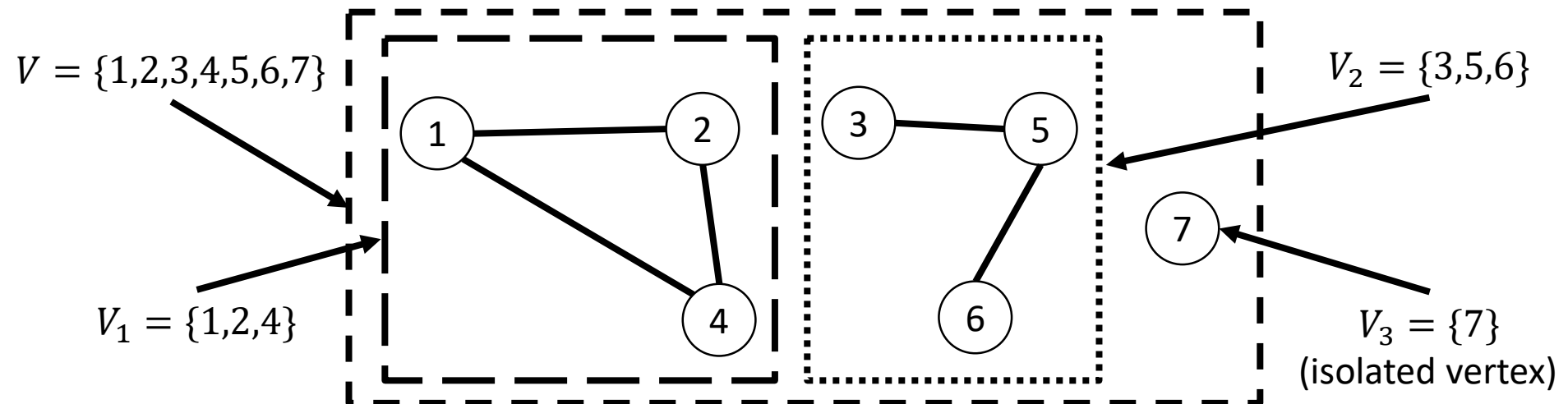
• Example:



- For node 1,
• $N(1) = \{2, 4\}$
- Given $A = \{1, 4\}$
• $N(A) = \{2, 5\}$

Connected and Isolated Vertex (Connectivity)

- **Connectivity** = a graph is connected if and only if there is a path between each pair of vertices
- **Connected Vertex** = a subset of vertices $V_i \subset V(G)$ that is connected
- **Isolated Vertex** = vertices which have no path exists



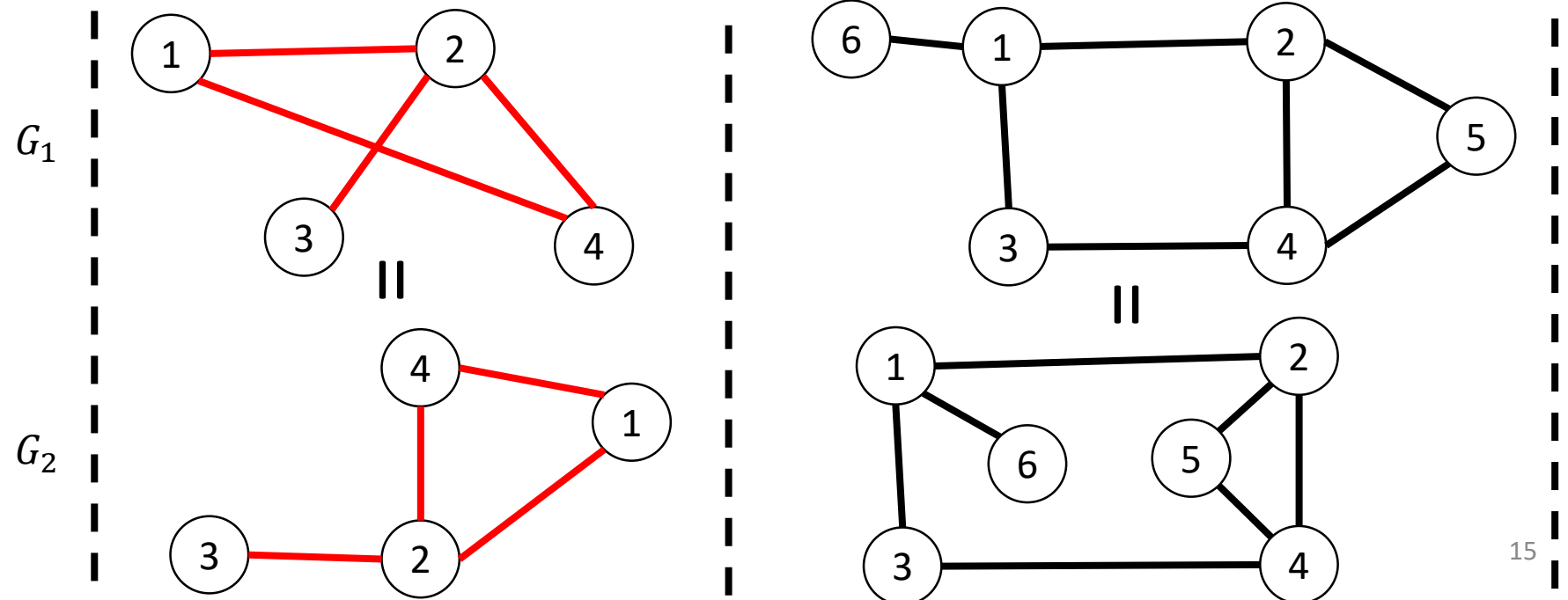
Classification of Graph

- Isomorphic /identical Graph
- Planar Graph
- Complete Graph
- Supergraph & Subgraph
- Bipartite Graph
- Tree

Isomorphism / Identical Graph

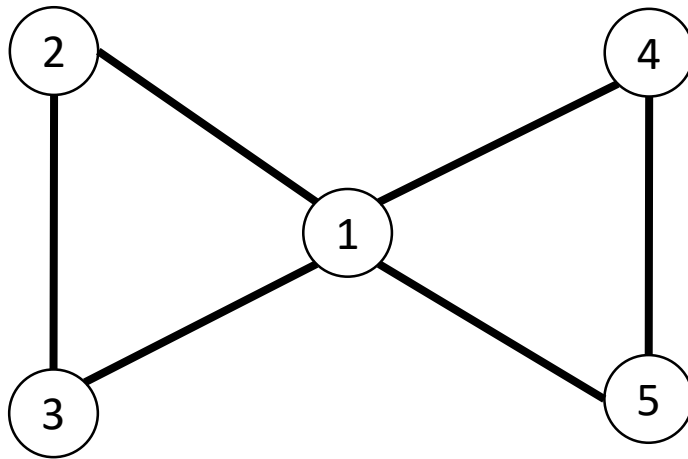
- **Isomorphism** = Two graphs G_1 & G_2 are **isomorphic** if there are:
 - Same number of nodes
 - Same number of edges
 - Preserved the vertex adjacency/edge connectivity (1-to-1 correspondence)

- Example:

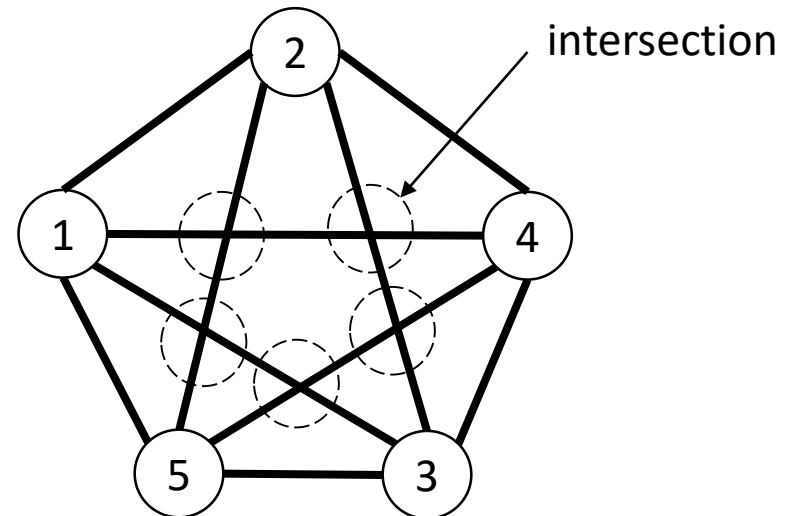


Planar Graph

- **Planar Graph** = A graph that can be drawn in the plane **without crossings**, implying that **no edges intersect** geometrically
- Example:



Planar Graph

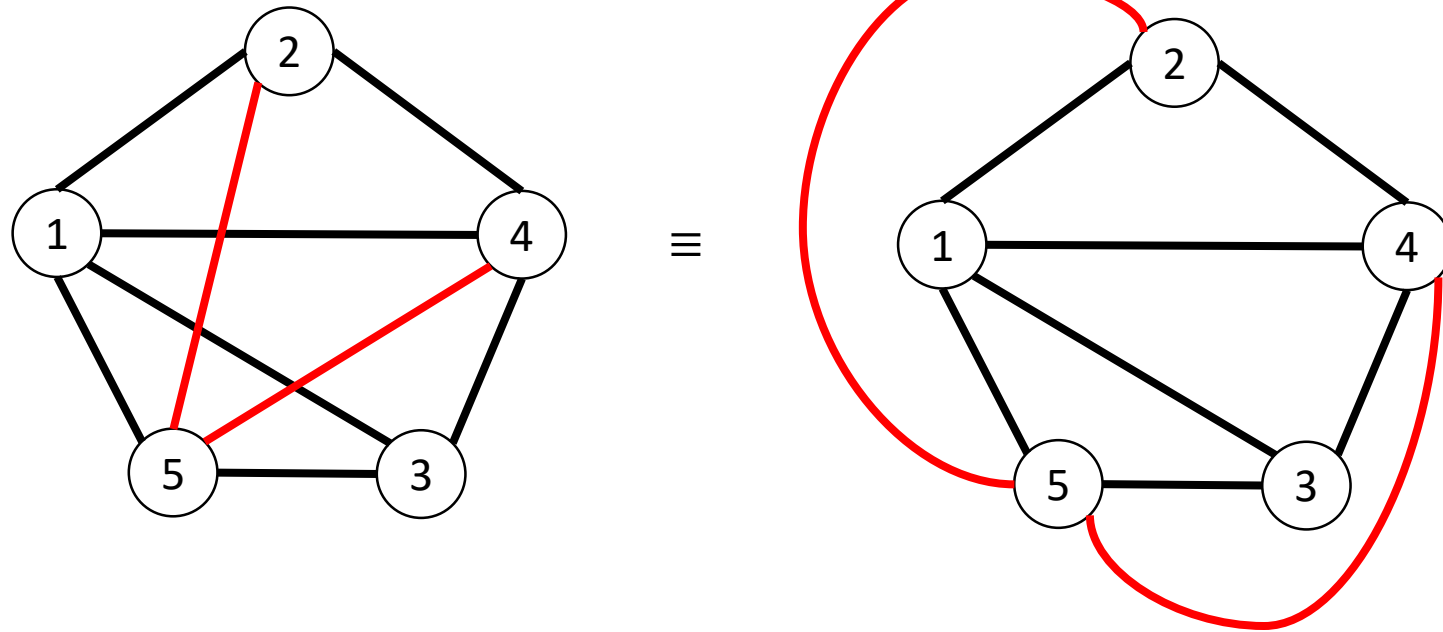


Non-planar Graph

Planar Graph

- Note that: A planar graph may be drawn like a non-planar graph

- Example:



It still a planar graph since they are isomorphic

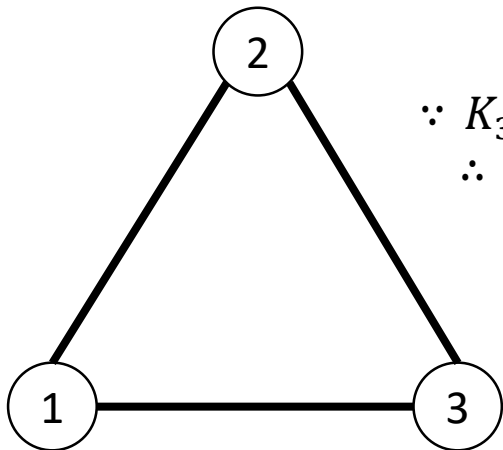
Completed Graph

- **Completed Graph** = A simple graph in which **each pair of distinct vertices are adjacent**, denoted by K_n

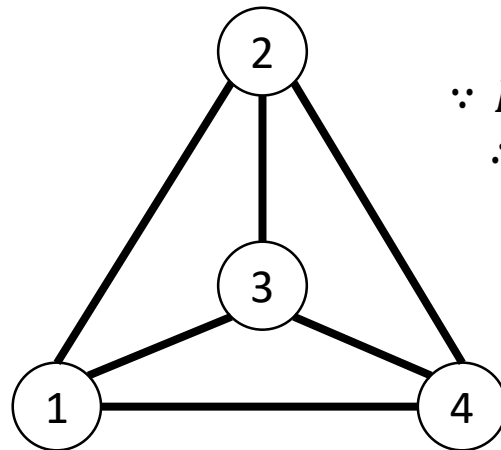
- Note that: Consider a complete graph on n vertices K_n ,

$$|E| = \frac{n(n-1)}{2}$$

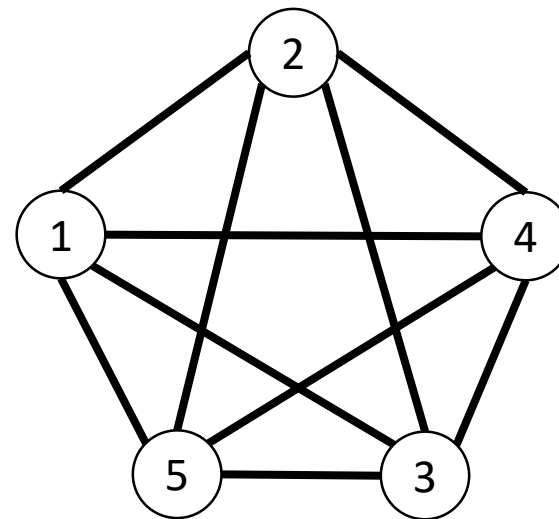
- Example:



$$\begin{aligned} \because K_3 \ (n = 3) \\ \therefore |E| = 3 \end{aligned}$$



$$\begin{aligned} \because K_4 \ (n = 4) \\ \therefore |E| = 6 \end{aligned}$$



$$\begin{aligned} \because K_5 \ (n = 5) \\ \therefore |E| = 10 \end{aligned}$$

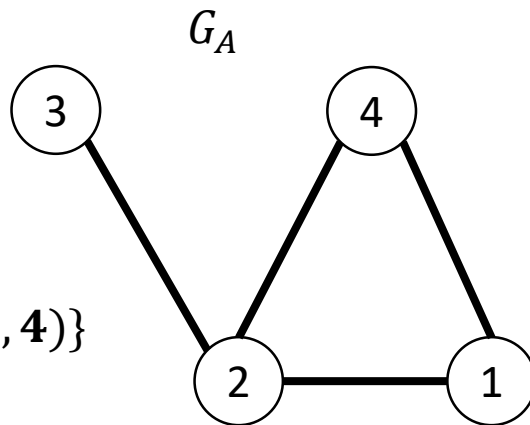
Supergraph / Subgraph

- Given a graph $G = \{V, E\}$
- **Subgraph of Graph G** = A graph in which each of whose vertices $v \in V(G)$ and each of whose edges $e \in E(G)$

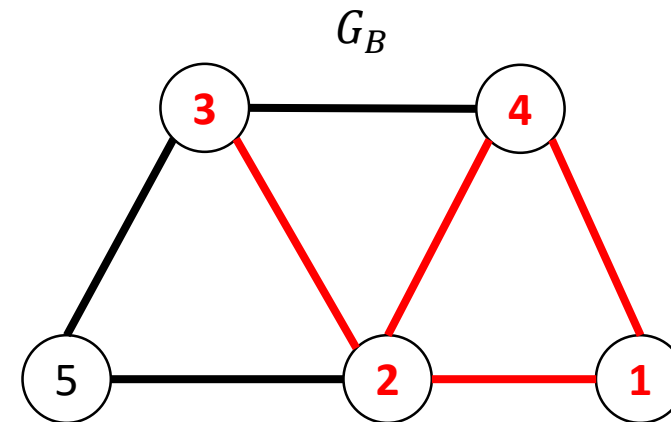
- Example:

$$v_A = \{1, 2, 3, 4\}$$

$$e_A = \{(1, 2), (1, 4), (2, 4), (3, 4)\}$$



G_A is the **subgraph** of G_B ;



G_B is the **supergraph** of G_A ;

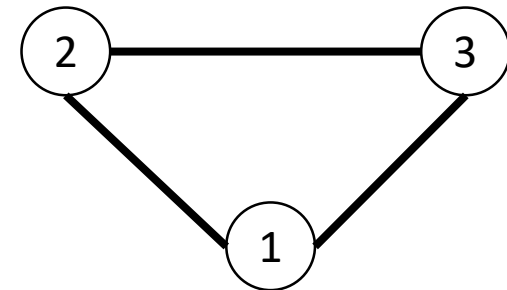
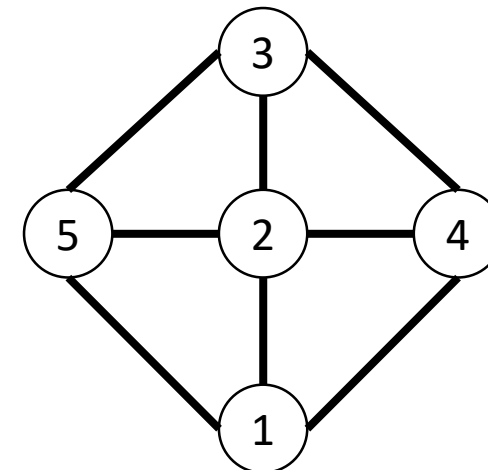
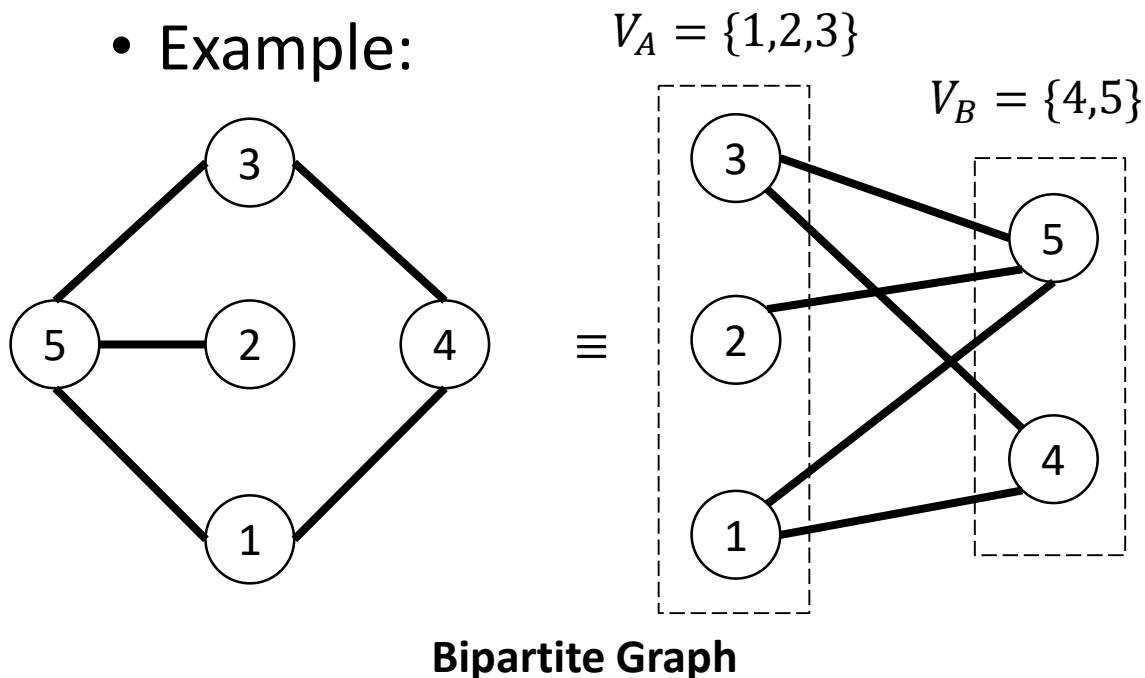
$$v_A \in v_B$$

$$e_A \in e_B$$

Bipartite Graph

- **Bipartite Graph** = A graph in which
 - Vertex set of G , $V(G)$, can be **split into two disjoint sets A and B** , $V_A(G)$ & $V_B(G)$
 - **Each edge of G joins a vertex of A and a vertex of B**
 - It is very commonly used in **in-and-output systems, where V_A and V_B are input and output vertices**

- Example:



Non-Bipartite Graph

Bipartite Graph

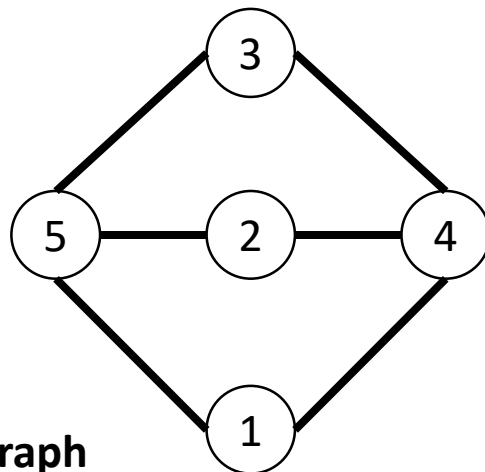
- **Complete Bipartite Graph** = A bipartite graph in which each vertex in A is joined to each vertex in B by just one edge, denoted by $K_{r,s}$

- Note that: Consider the complete bipartite graph $K_{r,s}$,

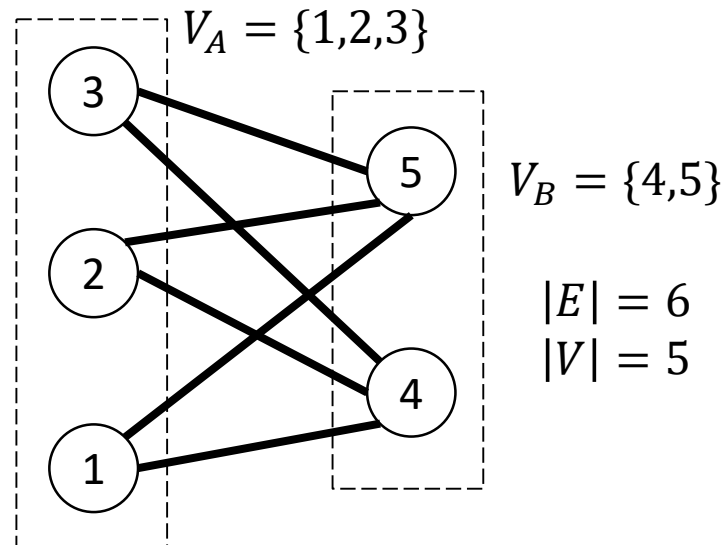
$$|E| = rs$$

$$|V| = r + s$$

- Example:



$\equiv$



$$|E| = 6$$

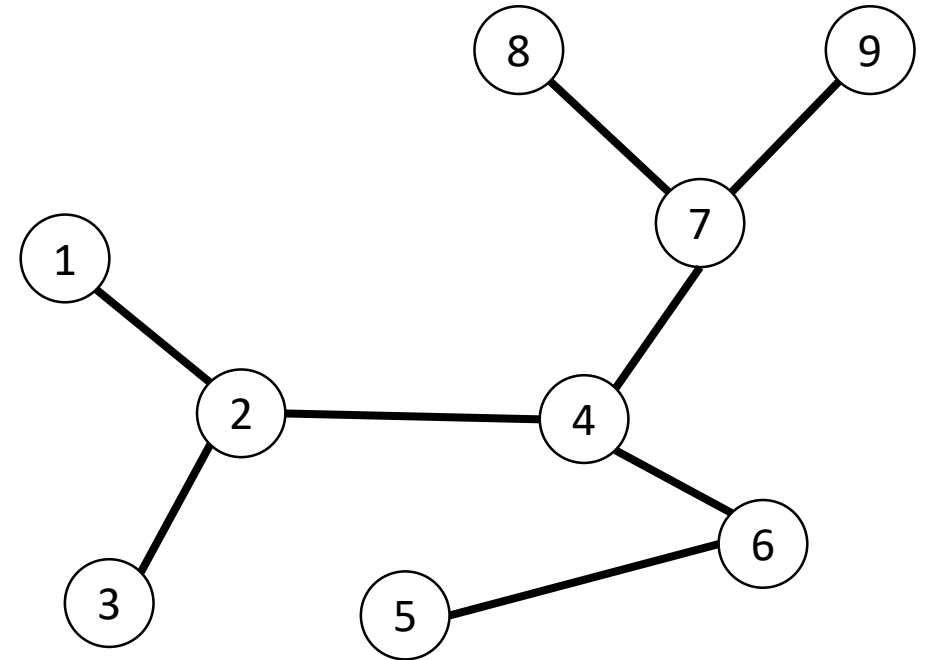
$$|V| = 5$$

Complete Bipartite Graph

Tree

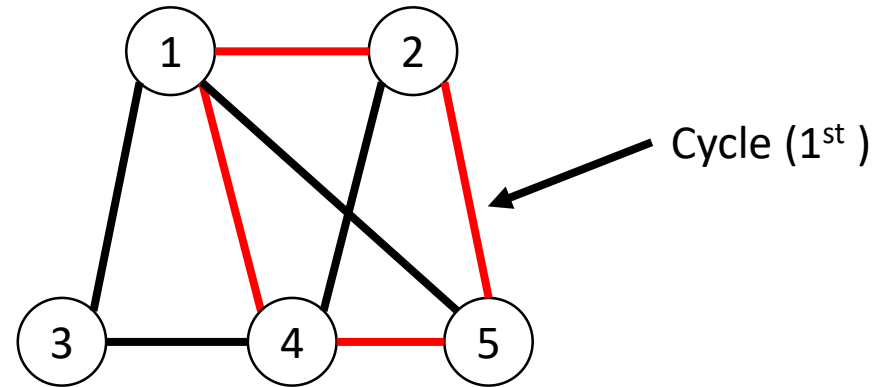
- **Tree** = A graph is **tree** if there is **exactly one path between any pairs** of vertices $\{u, v\}$

- **Criteria:**
 - Connected and acyclic (no cycles consists)
 - Removing edge disconnects graph
 - Adding edge creates a cycle

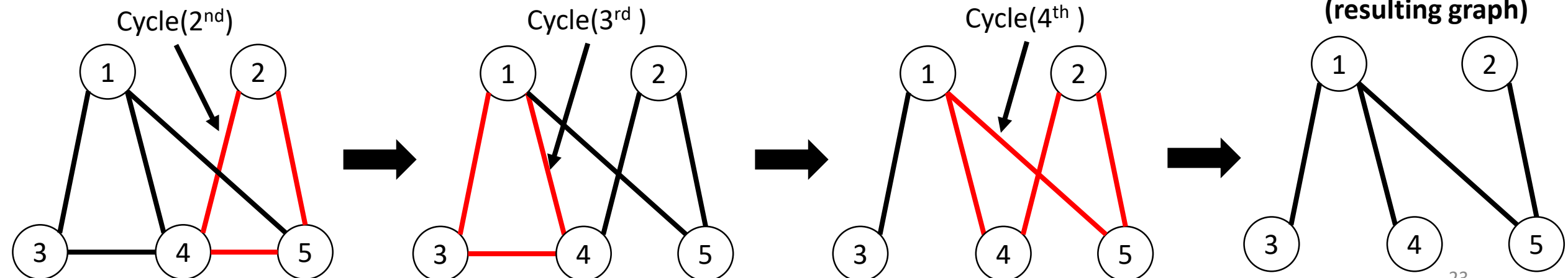


Tree

- Given any connected graph G ,



- Choosing a cycle and remove any one of its edge (the resulting graph must be connected)
 - Repeat the procedure with the remaining cycles until no cycles left
- Such that the remaining graph that connects all the vertices of G = **Spanning Tree** of G
- There is **not limited to single solution** of spanning tree on a given graph

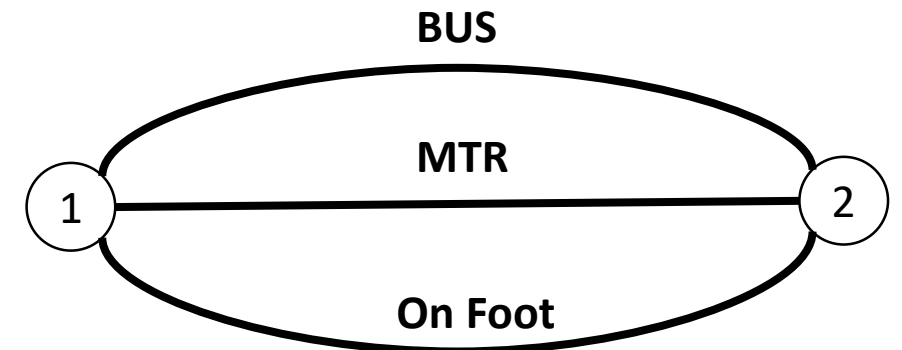
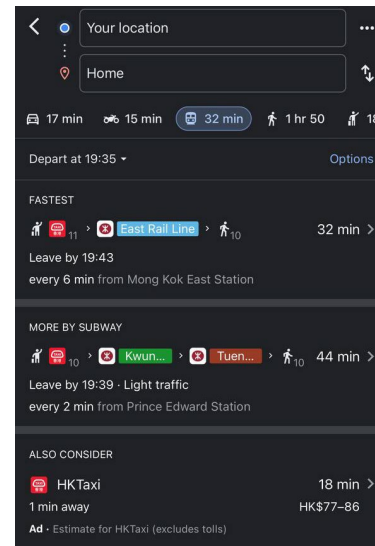
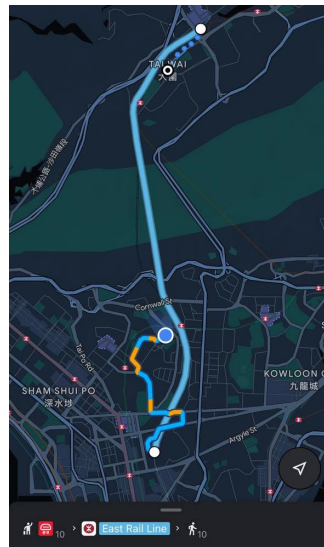
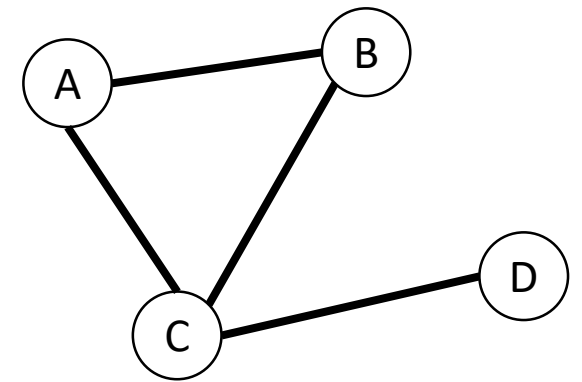


Type of Graph

- General Graph vs. Simple Graph
- Digraph (Directed Graph)
- Weighted Graph

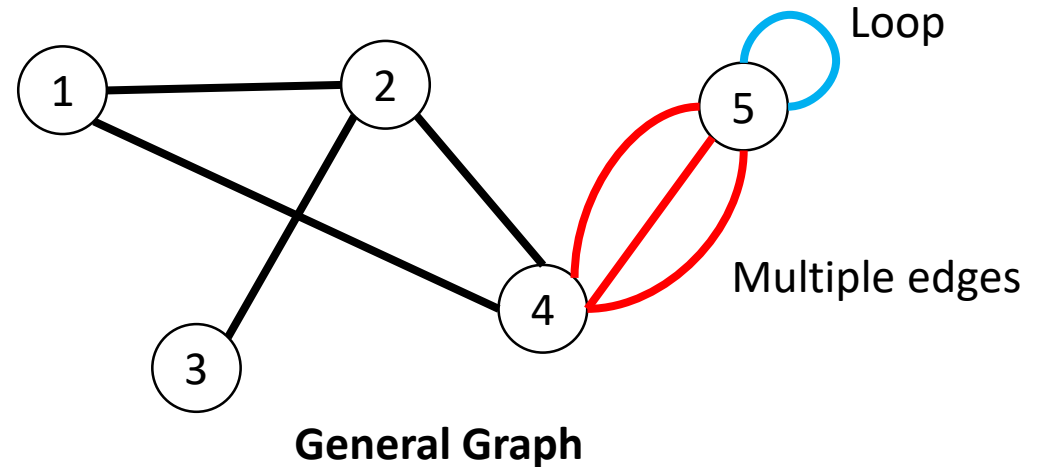
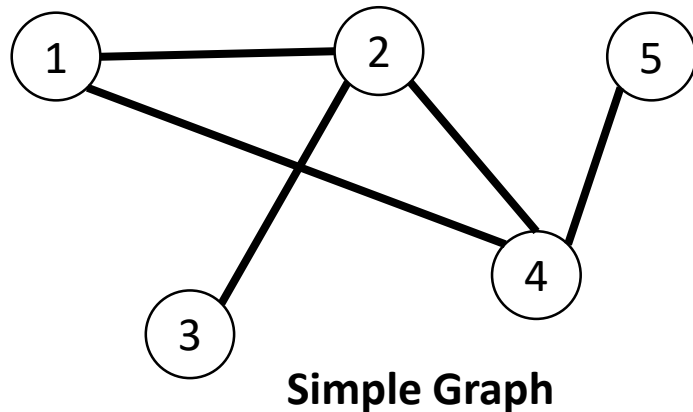
General Graph vs. Simple Graph

- The above-mentioned graph is limited to one edge joining each pair of vertices **BUT** not applicable to most of the cases
- Consider the journey of attending this class from your home to CityU,
 - **Nodes:** home sweet home (node 1), CityU (node2)
 - **Edges:** The route recommended by google map
- **There is not limited to a single route for this journey!**



General Graph vs. Simple Graph

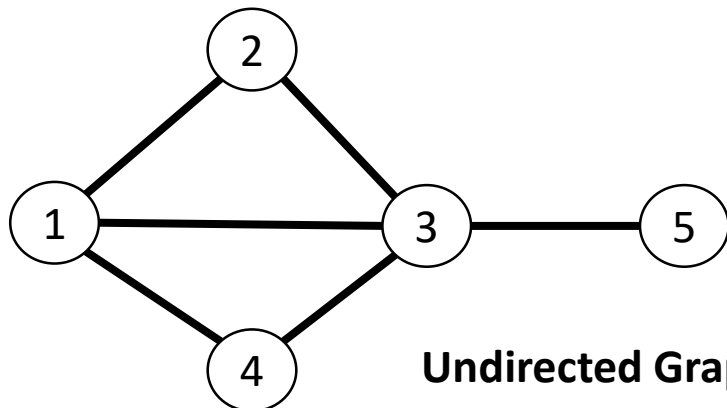
- **General Graph** = A graph in which allow **loops** and **multiple edges**
- Alternatively, **Simple Graph** = A graph with **no loops** and **multiple edges**



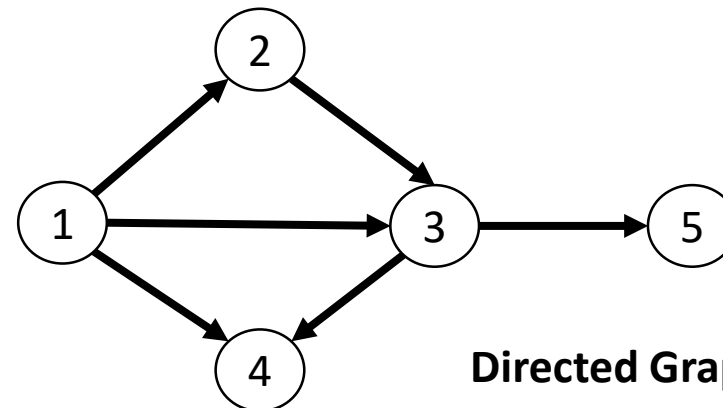
Directed Graph (Digraph)

- What if the edges are directional?
- Edges with order are called **Arcs / Arrows / Directed edges**
- **Directed Graph (Digraph)** = A graph in which consists of **directed edges**
- Note that: **Degree of directed graph = In-degree + out-degree**
 - **In-degree** = number of edges **entering** node
 - **Out-degree** = number of edges **leaving** node

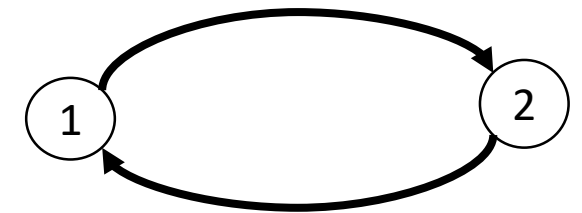
Two-way highway



Undirected Graph

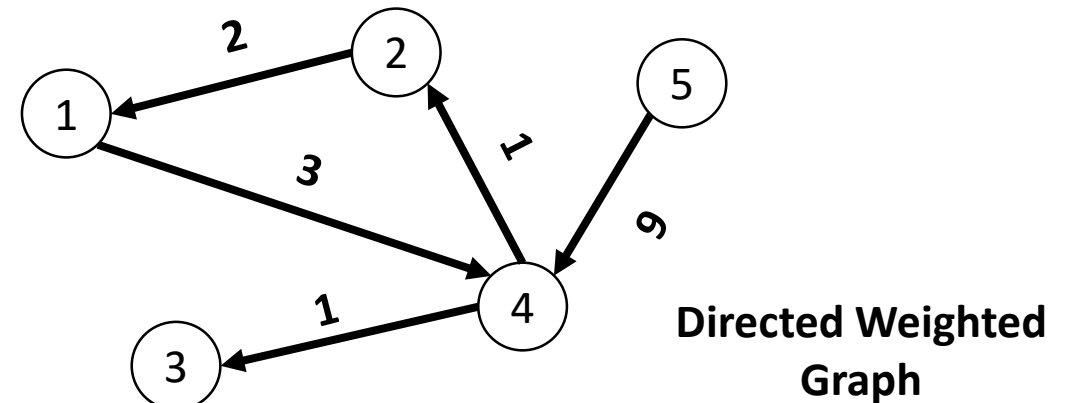
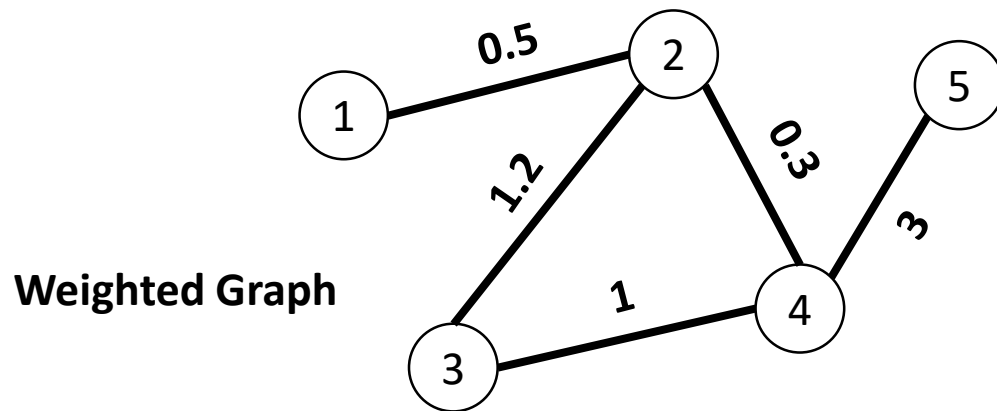


Directed Graph / Digraph



Weighted Graph

- **Weight** can be assigned to edges for describing the relation of vertices in detail
 - E.g., Transportation fees/costs, Distance, Time...
- **Weighted Graph** = A graph in which a **non-negative** number (**weight**) assigned to each edge



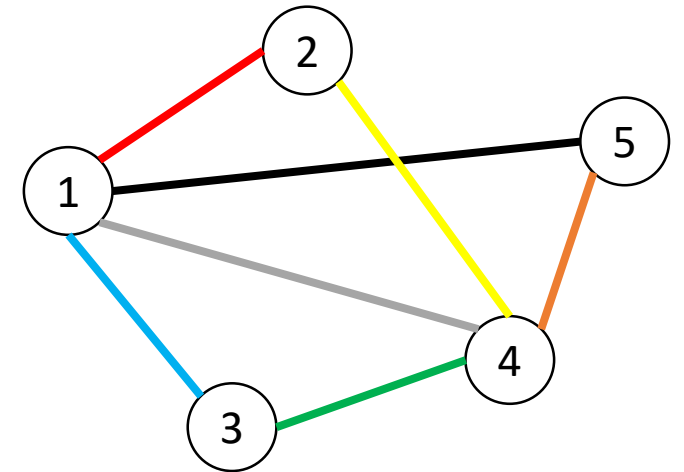
- And the number associated with each edge e is **weight**, denoted by $w(e)$

Graph Measurement

- Walk (Path & Trail) & Cycle
- Length & Distance
- Eccentricity
- Radius & Diameter

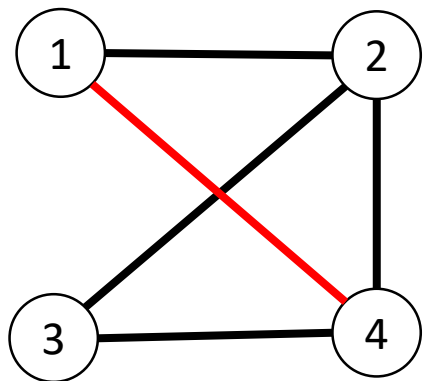
Walk (Path & Trail) & Cycle

- **Walk** = a **finite sequence** of edges/vertex of the form $v_0v_1, v_1v_2, \dots, v_{i-1}v_i$ / $v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_i$
 - E.g., $2 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 1 \rightarrow 3$
- **Trail** = a **walk** in which **all edges are distinct**
 - E.g., $2 \rightarrow 1 \rightarrow 5 \rightarrow 4 \rightarrow 1 \rightarrow 3$
- **Path** = a **walk** in which **all vertices are distinct**
 - E.g., $2 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 5$
- **Cycle** = a **closed path** in which **initial and final vertex are the same**
 - E.g., $2 \rightarrow 1 \rightarrow 4 \rightarrow 2$



Length & Distance

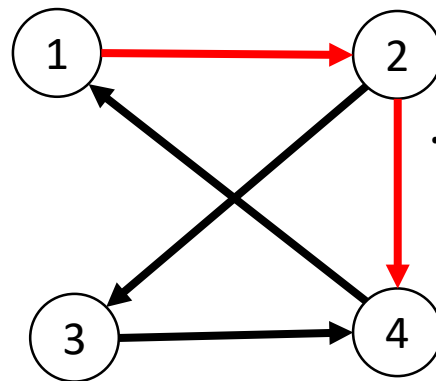
- **Length** = number of edges / sum of weights in a **walk/path/cycle**
- **Distance** = length of the **shortest path** between any pairs of vertices $\{u, v\} \in V(G)$, denoted by $d(u, v)$
- Example: Finding the **distance** from node 1 to node 4



Simple Graph

$\therefore \text{path} = 1 \rightarrow 4$

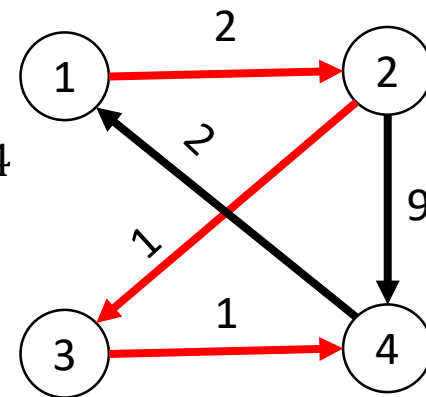
$\therefore d(1,4) = 1$



Directed Graph

$\therefore \text{path} = 1 \rightarrow 2 \rightarrow 4$

$\therefore d(1,4) = 2$



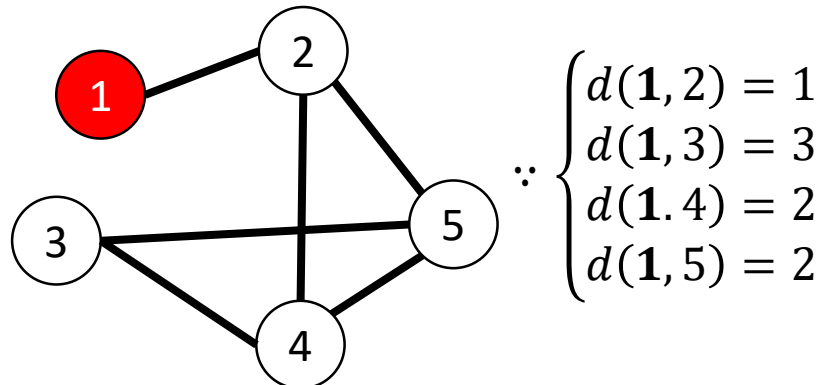
Weighted Graph

$\therefore \text{path} = 1 \rightarrow 2 \rightarrow 3 \rightarrow 4$

$\therefore d(1,4) = 2 + 1 + 1 = 4$

Eccentricity

- **Eccentricity** = maximum of all the distances from vertex $v \in V(G)$ to any other vertices $u \in V(G)$, denoted by $e(v) = \max\{d(v, u)\}$
- The **difference** between Eccentricity & Distance:
 - Eccentricity $e(v)$ defined on a **particular vertex** v
 - Distance $d(u, v)$ defined on a **particular pair of vertices** $\{u, v\}$
- Example: Find the **eccentricity** of **node 1** for the following graph



$$\therefore \begin{cases} d(1, 2) = 1 \\ d(1, 3) = 3 \\ d(1, 4) = 2 \\ d(1, 5) = 2 \end{cases}$$

$$\therefore e(1) = e\{1\} = \max\{d(1, u)\} = d(1, 3) = 3$$

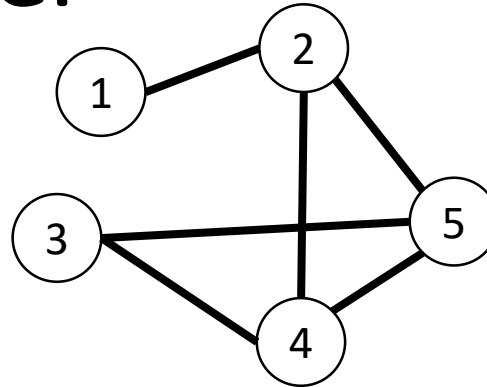
Please Try to find out the eccentricity for other nodes....

Radius & Diameter

- **Graphs with any sizes or completed graphs**, radius and diameter can be determined by a general form with eccentricity:
- **Radius = Minimum** over all the eccentricities in graph G , denoted by $r = \min\{e(v)\}$
- **Diameter = Maximum** over all the eccentricities in graph G , denoted by $d = \max\{e(v)\}$
- Alternatively, **Diameter** is also known as length of the **longest path** between any pairs of vertices $\{u, v\} \in V(G)$
- **However**, if the size of graph is small, we can **directly use relationship of degree/edges to identify radius and diameter**. The answer will be more efficient than using the general expression.

Radius & Diameter

Example:



For Small Graph only

$$\begin{aligned} \therefore \begin{cases} e(1) = 3 \\ e(2) = 2 \\ e(3) = 3 \\ e(4) = 2 \\ e(5) = 2 \end{cases} & \therefore \begin{cases} d = e(1) = e(3) = 3 \\ r = e(2) = e(4) = e(5) = 2 \end{cases} & \therefore \begin{cases} \text{Degree or } E(1) = 1 \\ \text{Degree or } E(2) = 3 \\ \text{Degree or } E(3) = 2 \\ \text{Degree or } E(4) = 3 \\ \text{Degree or } E(5) = 3 \end{cases} & \therefore \begin{cases} d = E(2) = E(4) = E(5) = 3 \\ r = E(1) = 1 \end{cases} \end{aligned}$$

General Expression by Eccentricity

Small Expression by degrees/edges

- The longest distances (diameter) are same but the r (radius) are different
- Why we have to reduce r for a small graph?

Parameters of Graph

We should know that

- Vertices and Edges are the fundamental elements of the graph
- They define order, size, degree, adjacent

Further....

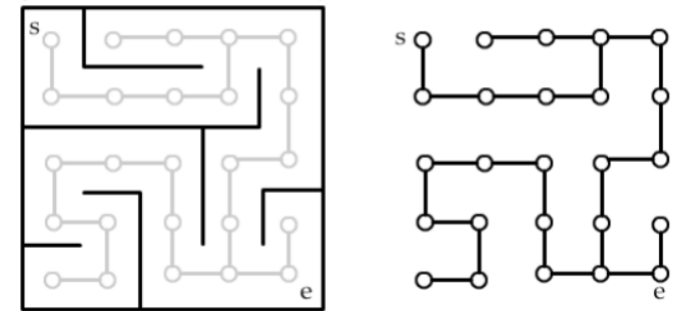
- To do analysis of the graph, we apply consideration of walk, trail, path, cycle
- During the analysis, we consider length, distance, eccentricity, radius, diameter
- The analyzed results help to provide efficient graph information for the systems, users, programming and computational resources

Shortest Path Problem and Algorithm

- Shortest Path Problem
- Dijkstra's Algorithm

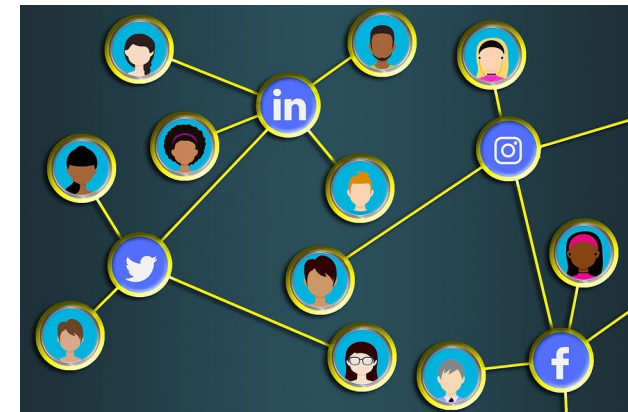
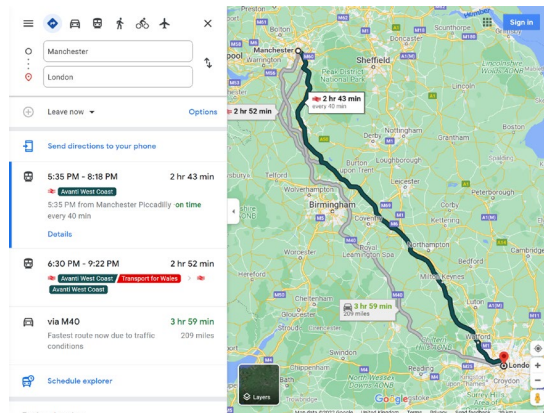
Shortest Path Problem

- The **problem of finding shortest paths** from a **single source vertex v** to **all other vertices** in the graph



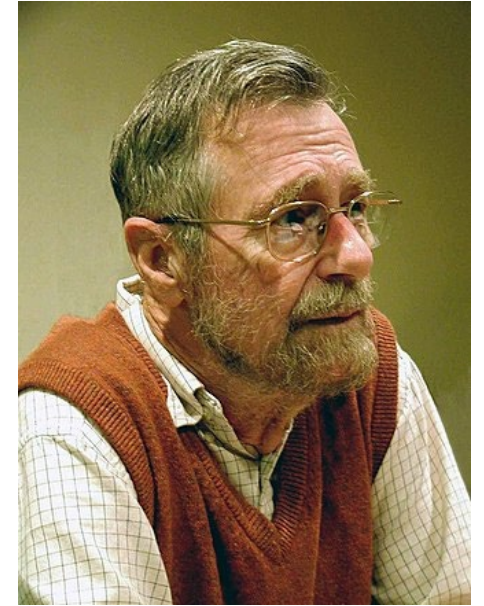
- **Applications:**

- Digital Mapping Services (Google Maps)
- Social Networking Applications (Facebook, Instagram, LinkedIn)
- Flighting Agenda
- Telephone Network
- IP routing
- Online Shopping
-



Dijkstra's Algorithm

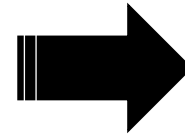
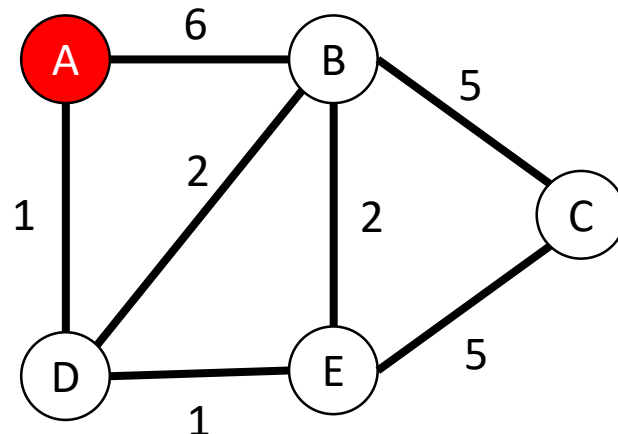
- Conceived by computer scientist **Edsger W. Dijkstra** in 1956
- It is a **solution** to the **single-source shortest path problem**
- **Characteristics:**
 - Applicable to both **directed and undirected graphs**
 - All edges must have **non-negative weights**



Edsger W. Dijkstra
(1930 - 2002)

Input

- Weighted (Un)directed Graph $G = \{V, E\}$
- Source vertex $v \in V(G)$ (in **red**)
- All non-negative edge weights $w(e)$



Output

- Lengths of shortest paths from a given source vertex $v \in V(G)$ (in **red**) to all other vertices
- Previous Vertex of specified vertex in shortest path

Vertex	Length of shortest path from A	Previous vertex
A	0	
B	3	D
C	7	E
D	1	A
E	2	D

General Procedure

To Begin,

- Set $\mathbf{A}$ is the initial vertex and $d(\mathbf{A}, \mathbf{A}) = 0$; other $d(\mathbf{A}, v_i) = \infty$
- Update $d(\mathbf{A}, v_i)$ and the previous vertices in the table

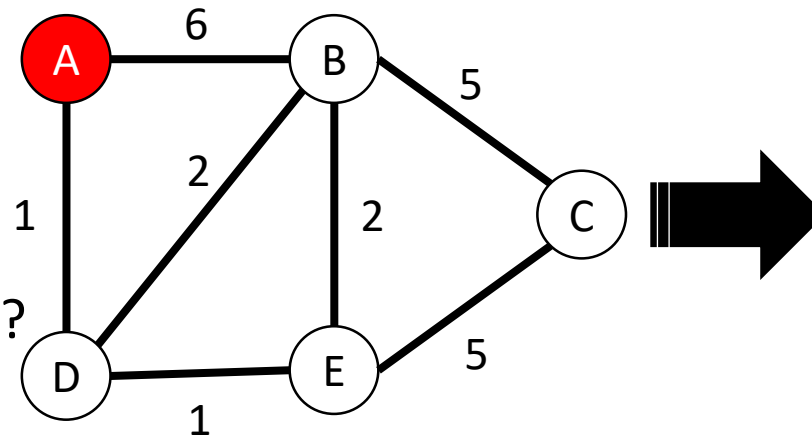
Start implement,

- Set v_i is the second initial vertex, update $d(\mathbf{A}, v_i)$ and the previous vertices in the table
- Checking min. $d(\mathbf{A}, v_i)$ for individual cases of v_i
- Update min. $d(\mathbf{A}, v_i)$ and the previous vertices in the table
- Repeat the implementation until all v_i have been visited

End

- The final table shows the shortest path for the graph

Dijkstra's Algorithm

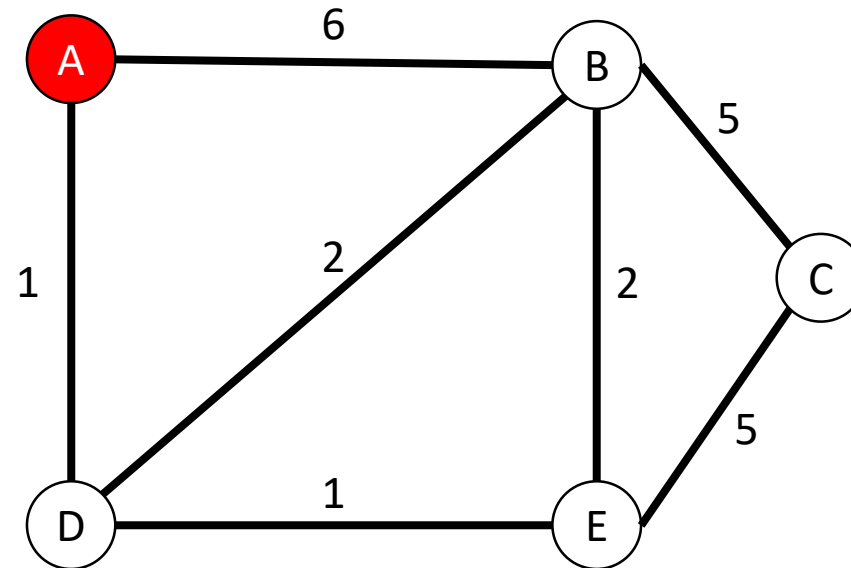


Vertex	Length of shortest path from A	Previous vertex
A	0	
B	3	D
C	7	E
D	1	A
E	2	D

- How does the Dijkstra's Algorithm work?
- Details:
 - Step 1: Mark the **distance from start vertex to all other vertices** are **infinity (∞)** and the **distance to itself is zero (0)**
 - Step 2: From the **start vertex/current vertex**, calculate the **accumulated** distance from the **start vertex to all its neighbors**
 - Step 3: For **each neighbors**, If the calculated distance is **smaller** than the previous result
 - **Update the calculated distance** in the table
 - **Mark the start vertex/current vertex** as the **previous vertex**
 - Step 4: Find **the vertex with a smallest accumulated distance** among the neighbors,
 - Mark the **start vertex/current vertex as visited vertex** (no longer consider as a neighbor)
 - mark the vertex found with smallest distance as **the current vertex for next visiting**
 - Step 5: **Repeat from Step 2** until **all the vertices are visited**

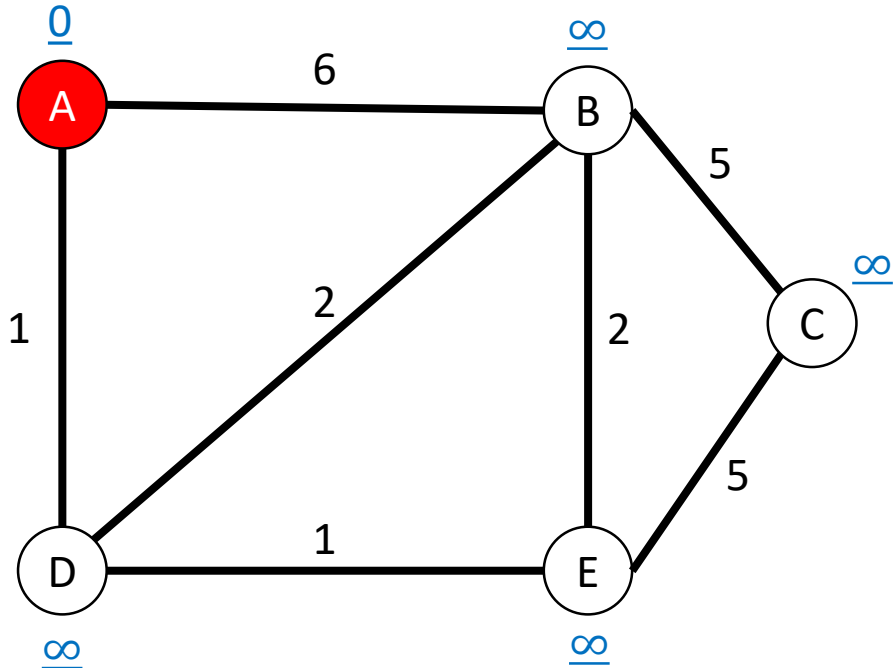
Dijkstra's Algorithm (Example)

- Example: Consider the following weighted graph, find the **shortest path from vertex A to every other vertex**



Dijkstra's Algorithm (Step 1)

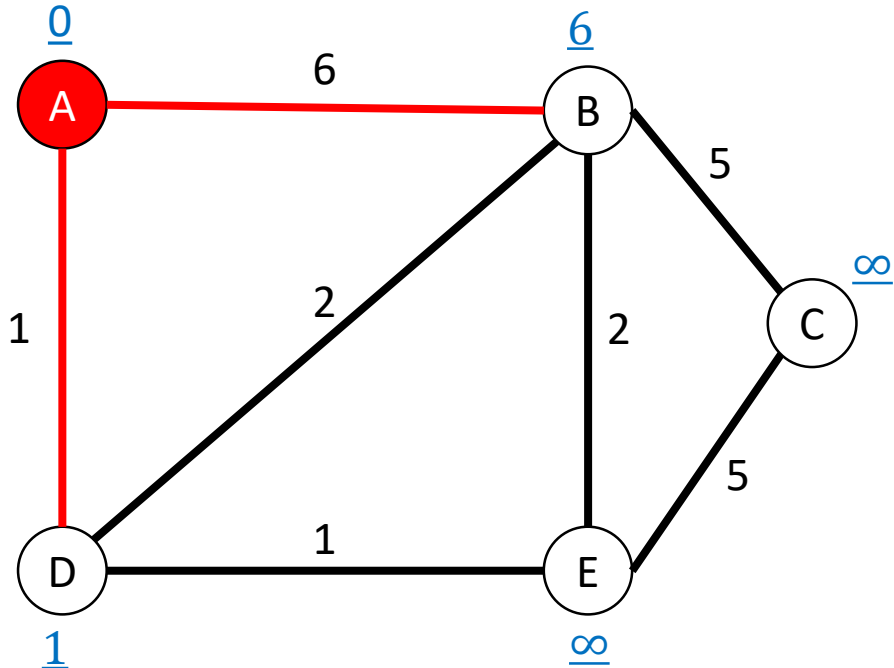
- Step 1: Mark the **distance from start vertex to all other vertices are infinity (∞)** and the **distance to itself is zero (0)**



Vertex v_i	$d(A, v_i)$	Previous vertex
A	<u>0 (visited)</u>	
B	∞	
C	∞	
D	∞	
E	∞	

Dijkstra's Algorithm (Step 2 @ vertex A)

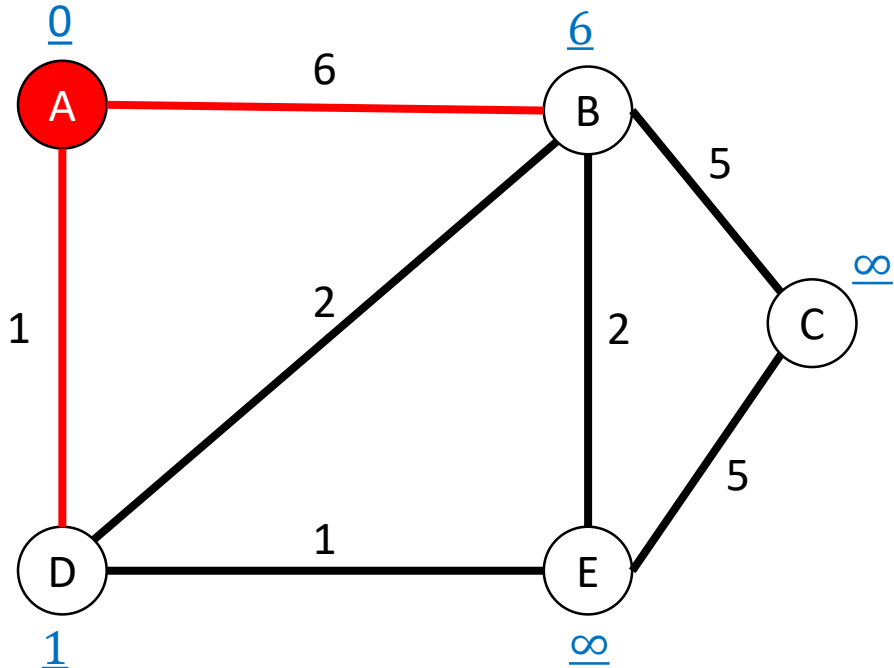
- Step 2: From the **start vertex/current vertex**, calculate the **accumulated** distance from the **start vertex to all its neighbors**



Vertex v_i	$d(A, v_i)$	@ vertex A	Previous vertex
A	<u>0 (visited)</u>		
B	∞	6	
C	∞	∞	
D	∞	1	
E	∞	∞	

Dijkstra's Algorithm (Step 3 @ vertex A)

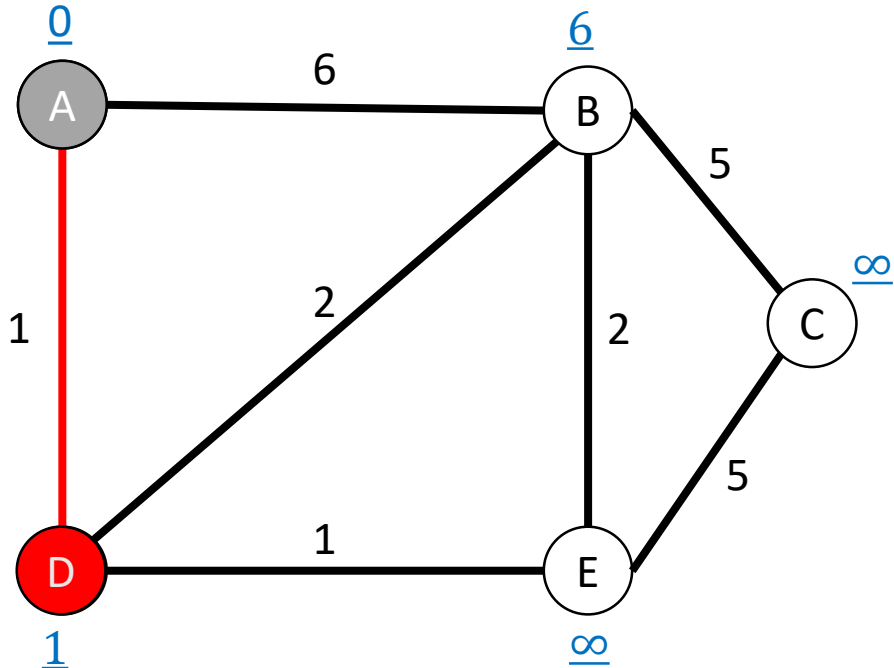
- Step 3: For **each neighbors**, If the calculated distance is **smaller** than the previous result
 - **Update the calculated distance** in the table
 - **Mark the start vertex/current vertex** as the **previous vertex**



Vertex v_i	$d(A, v_i)$	@ vertex A	Previous vertex
A	<u>0 (visited)</u>		
B	∞	6	A
C	∞	∞	
D	∞	1	A
E	∞	∞	

Dijkstra's Algorithm (Step 4 @ vertex A)

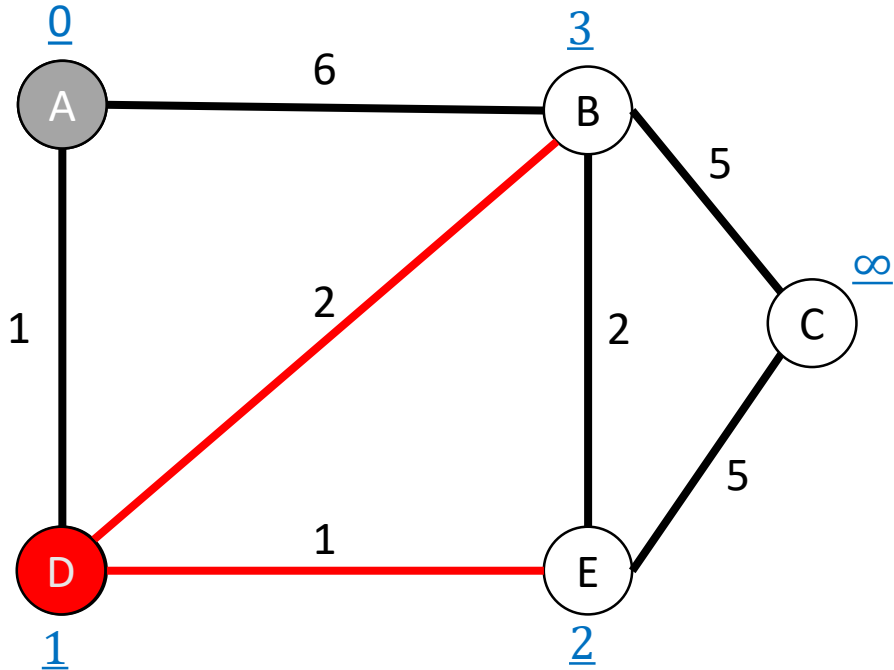
- Step 4: Find **the vertex with a smallest accumulated distance** among the neighbors,
 - Mark the **start vertex/current vertex as visited vertex** (no longer consider as a neighbor)
 - Mark the vertex found with smallest distance as **the current vertex for next visiting**



Vertex v_i	$d(A, v_i)$	@ vertex A	Previous vertex
A	<u>0 (visited)</u>		
B	∞	6	A
C	∞	∞	
D	∞	<u>1 (visited)</u>	A
E	∞	∞	

Dijkstra's Algorithm (Step 2 @ vertex D)

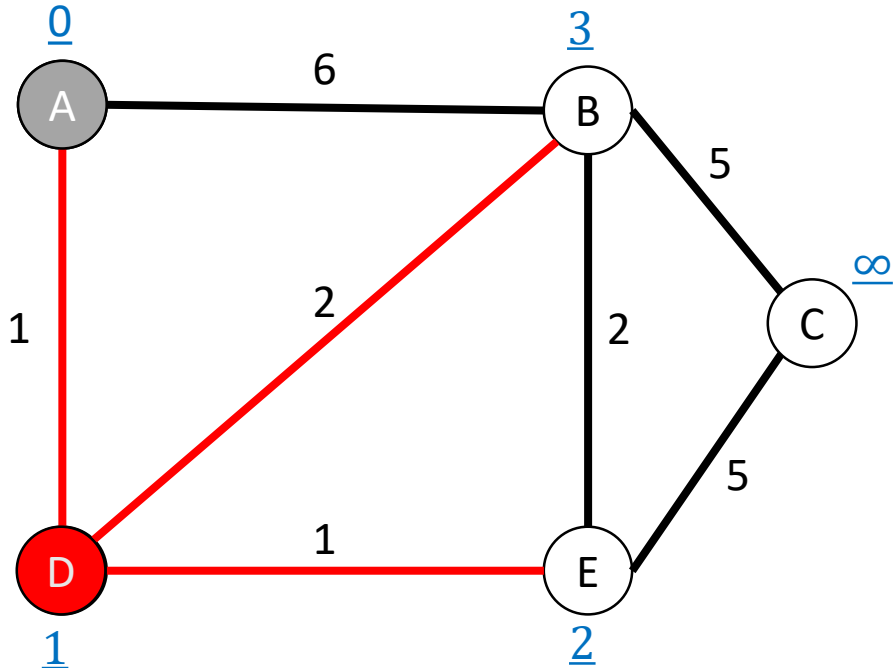
- Step 2: From the **start vertex/current vertex**, calculate the **accumulated** distance from the **start vertex to all its neighbors**



Vertex v_i	$d(A, v_i)$	@ vertex A	@ vertex D	Previous vertex
A	<u>0 (visited)</u>			
B	∞	6	3	A
C	∞	∞	∞	
D	∞	<u>1 (visited)</u>		A
E	∞	∞	2	

Dijkstra's Algorithm (Step 3 @ vertex D)

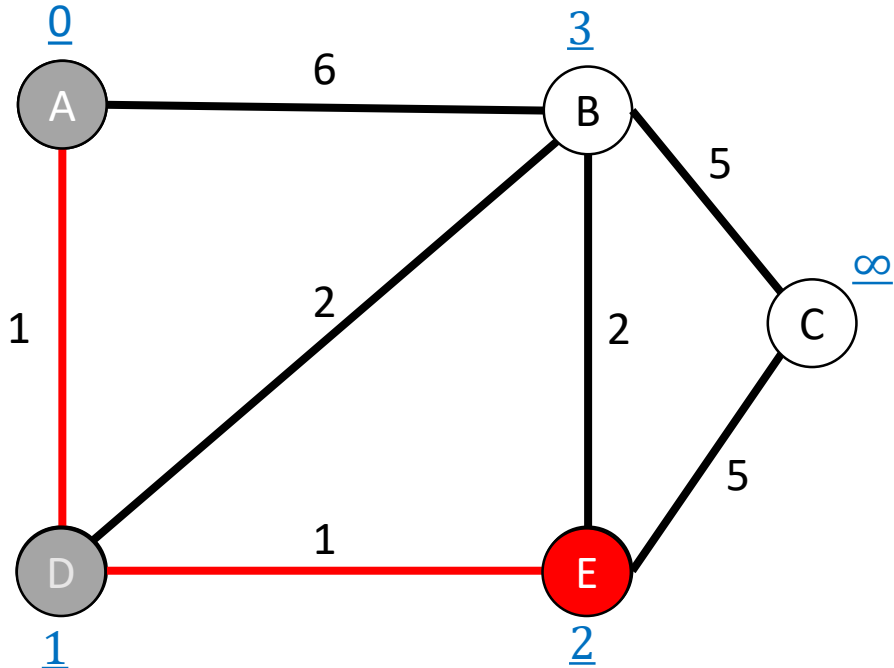
- Step 3: For **each neighbors**, If the calculated distance is **smaller** than the previous result
 - **Update the calculated distance** in the table
 - **Mark the start vertex/current vertex** as the **previous vertex**



Vertex v_i	$d(A, v_i)$	@ vertex A	@ vertex D	Previous vertex
A	<u>0 (visited)</u>			
B	∞	6	3	D
C	∞	∞	∞	
D	∞	<u>1 (visited)</u>		A
E	∞	∞	2	D

Dijkstra's Algorithm (Step 4 @ vertex D)

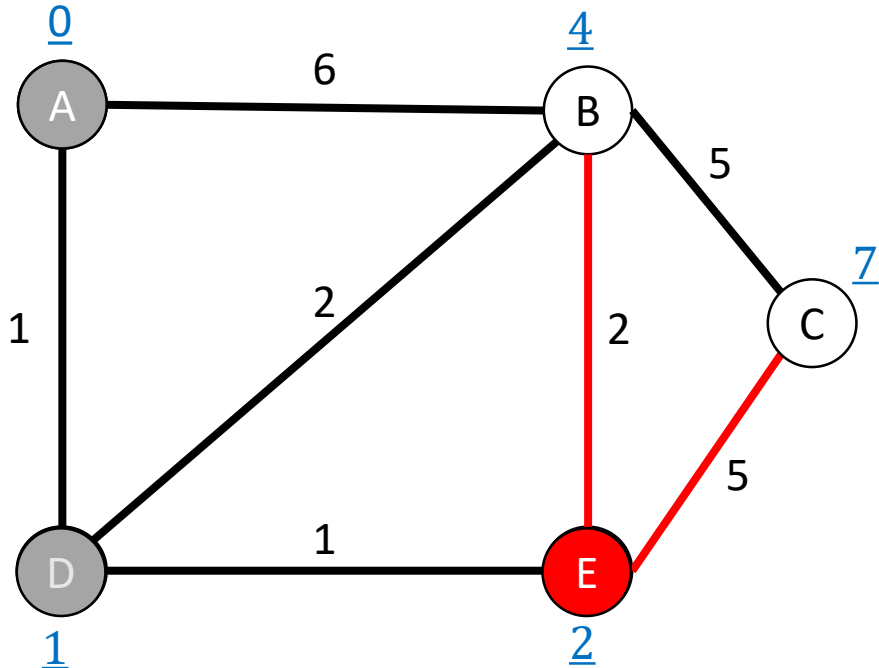
- Step 4: Find **the vertex with a smallest accumulated distance** among the neighbors,
 - Mark the **start vertex/current vertex as visited vertex** (no longer consider as a neighbor)
 - Mark the vertex found with smallest distance as **the current vertex for next visiting**



Vertex v_i	$d(A, v_i)$	@ vertex A	@ vertex D	Previous vertex
A	<u>0 (visited)</u>			
B	∞	6	3	D
C	∞	∞	∞	
D	∞	<u>1 (visited)</u>		A
E	∞	∞	<u>2 (visited)</u>	D

Dijkstra's Algorithm (Step 2 @ vertex E)

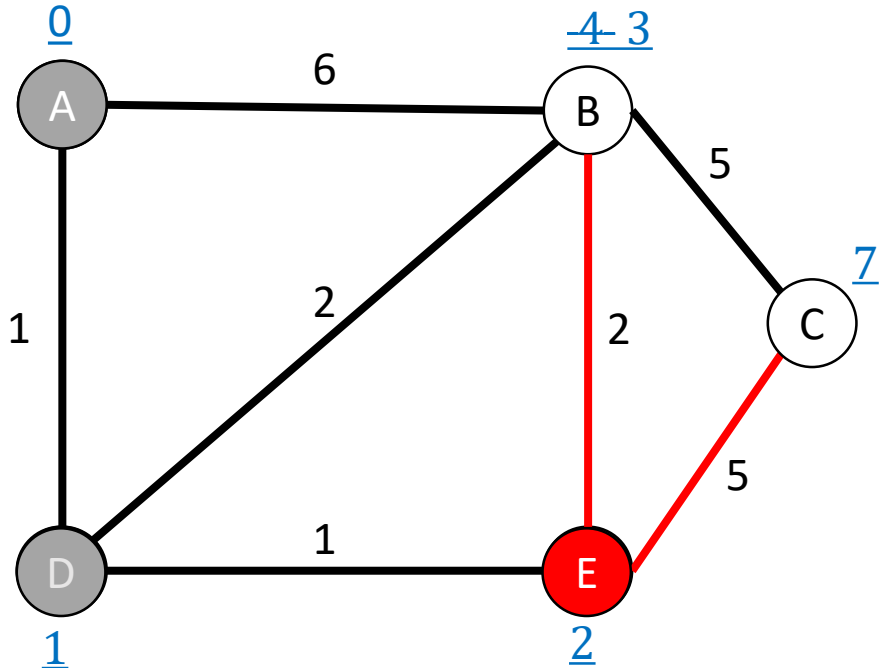
- Step 2: From the **start vertex/current vertex**, calculate the **accumulated** distance from the **start vertex to all its neighbors**



Vertex v_i	$d(A, v_i)$	@ vertex A	@ vertex D	@ vertex E	Previous vertex
A	<u>0 (visited)</u>				
B	∞	6	3	4	D
C	∞	∞	∞	7	
D	∞	<u>1 (visited)</u>			A
E	∞	∞	<u>2 (visited)</u>		D

Dijkstra's Algorithm (Step 3 @ vertex E)

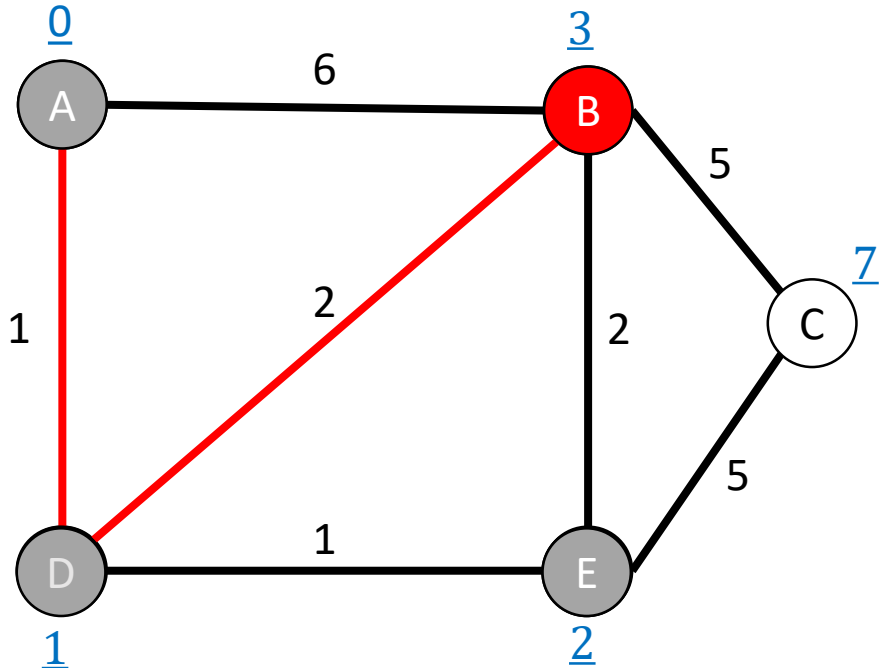
- Step 3: For **each neighbors**, If the calculated distance is **smaller** than the previous result
 - **Update the calculated distance** in the table
 - **Mark the start vertex/current vertex** as the **previous vertex**



Vertex v_i	$d(A, v_i)$	@ vertex A	@ vertex D	@ vertex E	Previous vertex
A	<u>0 (visited)</u>				
B	∞	6	3	~~4~~ 3	D
C	∞	∞	∞	7	E
D	∞	<u>1 (visited)</u>			A
E	∞	∞	<u>2 (visited)</u>		D

Dijkstra's Algorithm (Step 4 @ vertex E)

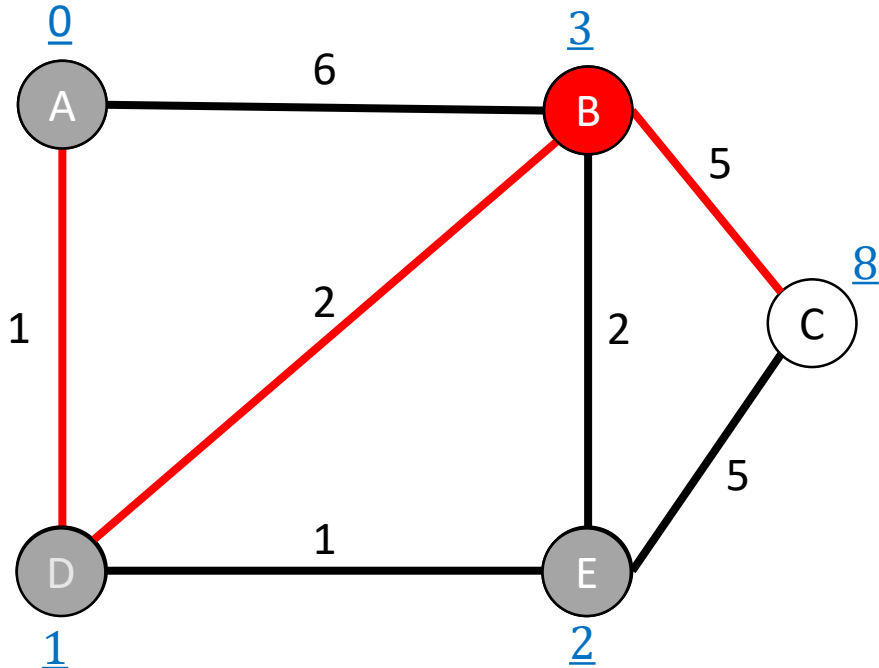
- Step 4: Find **the vertex with a smallest accumulated distance** among the neighbors,
 - Mark the **start vertex/current vertex as visited vertex** (no longer consider as a neighbor)
 - Mark the vertex found with smallest distance as **the current vertex for next visiting**



Vertex v_i	$d(A, v_i)$	@ vertex A	@ vertex D	@ vertex E	Previous vertex
A	<u>0 (visited)</u>				
B	∞	6	3	~~4~~ <u>3</u> (visited)	D
C	∞	∞	∞	7	E
D	∞	<u>1 (visited)</u>			A
E	∞	∞	<u>2 (visited)</u>		D

Dijkstra's Algorithm (Step 2 @ vertex B)

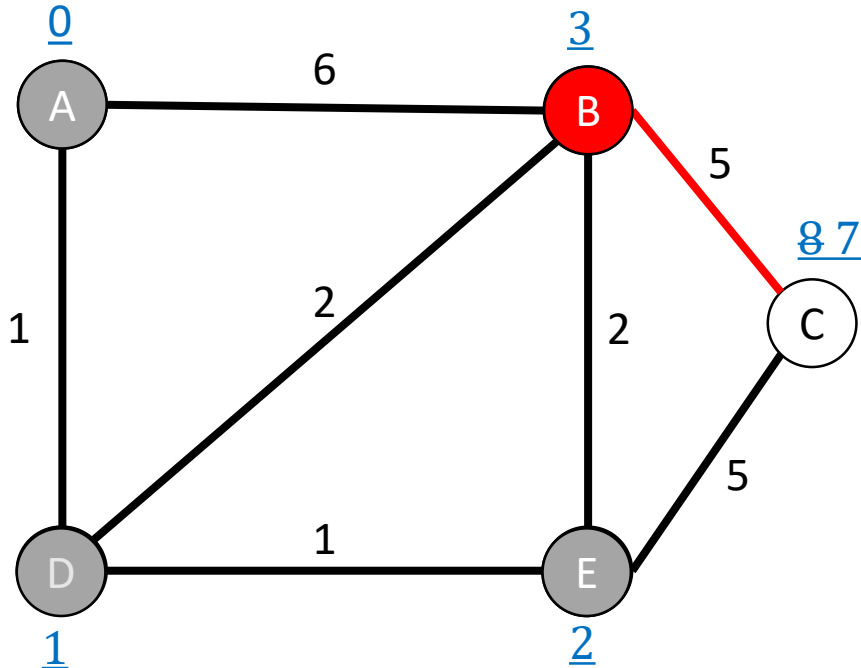
- Step 2: From the **start vertex/current vertex**, calculate the **accumulated** distance from the **start vertex to all its neighbors**



Vertex v_i	$d(A, v_i)$	@ vertex A	@ vertex D	@ vertex E	@ vertex B	Previous vertex
A	<u>0 (visited)</u>					
B	∞	6	3	~~5~~ <u>3 (visited)</u>		D
C	∞	∞	∞	7	8	E
D	∞	<u>1 (visited)</u>				A
E	∞	∞	<u>2 (visited)</u>			D

Dijkstra's Algorithm (Step 3 @ vertex B)

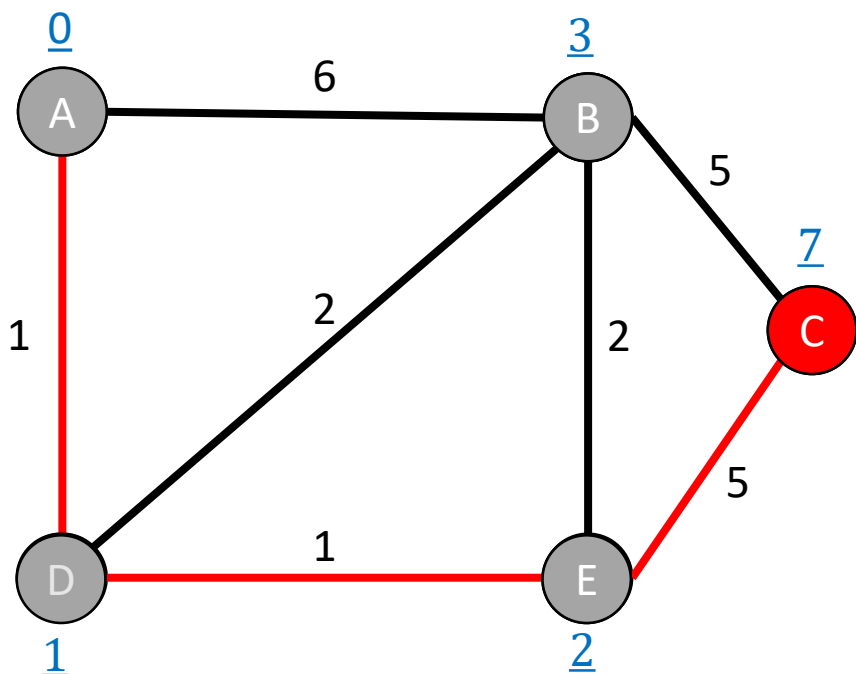
- Step 3: For **each neighbors**, If the calculated distance is **smaller** than the previous result
 - **Update the calculated distance** in the table
 - **Mark the start vertex/current vertex** as the **previous vertex**



Vertex v_i	$d(A, v_i)$	@ vertex A	@ vertex D	@ vertex E	@ vertex B	Previous vertex
A	<u>0 (visited)</u>					
B	∞	6	3	~~5~~ <u>3 (visited)</u>		D
C	∞	∞	∞	7	~~8~~ 7	E
D	∞	<u>1 (visited)</u>				A
E	∞	∞	<u>2 (visited)</u>			D

Dijkstra's Algorithm (Step 4 @ vertex B)

- Step 4: Find **the vertex with a smallest accumulated distance** among the neighbors,
 - Mark the **start vertex/current vertex as visited vertex** (no longer consider as a neighbor)
 - Mark the vertex found with smallest distance as **the current vertex for next visiting**



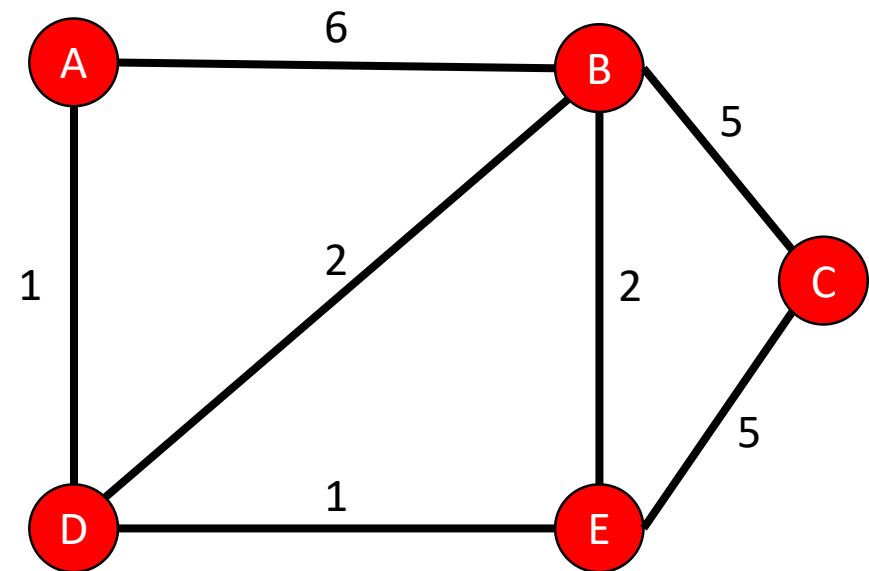
Vertex v_i	$d(A, v_i)$	@ vertex A	@ vertex D	@ vertex E	@ vertex B	Previous vertex
A	<u>0 (visited)</u>					
B	∞	6	3	~~5~~ <u>3 (visited)</u>		D
C	∞	∞	∞	7	~~8~~ <u>7 (visited)</u>	E
D	∞	<u>1 (visited)</u>				A
E	∞	∞	<u>2 (visited)</u>			D

Dijkstra's Algorithm (Step 5 - Final Result)

- Now, ALL the vertices are visited, and we can find the shortest path from vertex A to all other vertices!

Vertex	$d(A, v_i)$	Previous vertex
A	0	
B	3	D
C	7	E
D	1	A
E	2	D

- Path from A to B: $A \rightarrow D \rightarrow B$, $d(A, B) = 3$
- Path from A to C: $A \rightarrow D \rightarrow E \rightarrow C$, $d(A, C) = 7$
- Path from A to D: $A \rightarrow D$, $d(A, D) = 1$
- Path from A to E: $A \rightarrow D \rightarrow E$, $d(A, E) = 2$



Dijkstra's Algorithm (Summary)

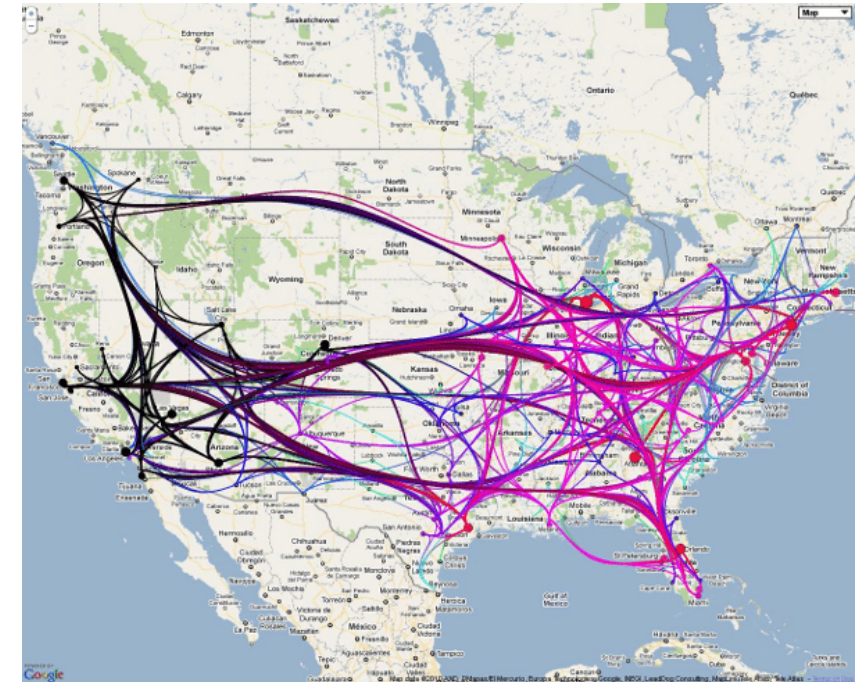
- Let **S** be the set of the vertices in which visited
 - Initialize the distance of start vertex from start vertex = 0
 - Initialize the distance of all other vertices from start = ∞ (*infinity*)
- Repeat the following until all vertices visited:
 - Visit the unvisited vertex with the smallest known distance from the start vertex
 - For the current vertex,
 - Examine its unvisited neighbors
 - Calculate distance of each neighbor from the start vertex
 - If the calculated distance of a vertex is less than the known distance
 - Update the shortest distance
 - Update the previous vertex for each of the updated vertices
 - Add the current vertex to the list of visited vertices **S**

Graph Representation

- Adjacency Matrix
- Incidence Matrix

How is graph recognized by computer?

- Consider the shortest path problem in the scale of a whole nation
 - Tremendous nodes and edges...
- Hand-written Graph may not be efficient to solve Shortest Path Problem
- **Solution:** Graph shall be converted into Mathematical representation (Matrix)
 - Adjacency Matrix
 - Incidence Matrix



Adjacency Matrix

- **Adjacency Matrix** $A_{n \times n}$ = Mapping **vertices adjacent to any other vertices** through a **n x n array**, where **n** is the order of the graph

$$A_{n \times n} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

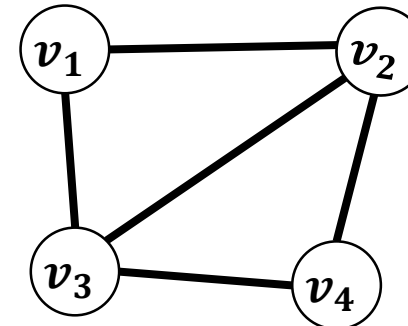
- **Rules** of determining the entries a_{ij} between any pairs of vertices $v_i \in V(G)$

- $a_{ij} = \begin{cases} 1, & \text{if } \text{edge}(v_i, v_j) \text{ exists} \\ 0, & \text{otherwise} \end{cases}$ for **undirected/directed** graph

- $a_{ij} = \begin{cases} \text{weight or no. of edges,} & \text{if } \text{edge}(v_i, v_j) \text{ exists} \\ 0, & \text{otherwise} \end{cases}$ for **weighted/general** graph

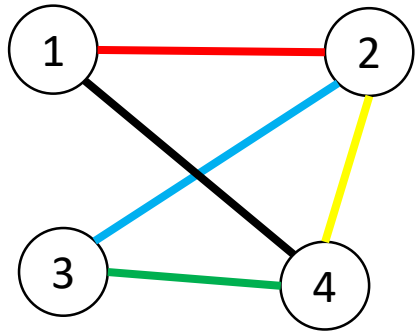
- **Characteristics:**

- Diagonal Entries for loops
- Diagonally Symmetric matrix for undirected graphs
- $a_{uv} \neq a_{vu}$ for directed graphs



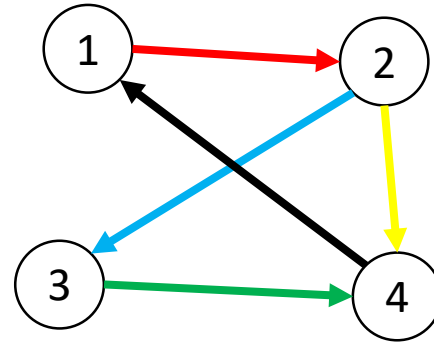
	v_1	v_2	$\cdots$	v_n
v_1	a_{11}	a_{12}	$\cdots$	a_{1n}
v_2	a_{21}	a_{22}	$\cdots$	a_{2n}
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
v_n	a_{n1}	a_{n2}	$\cdots$	a_{nn}

Adjacency Matrix



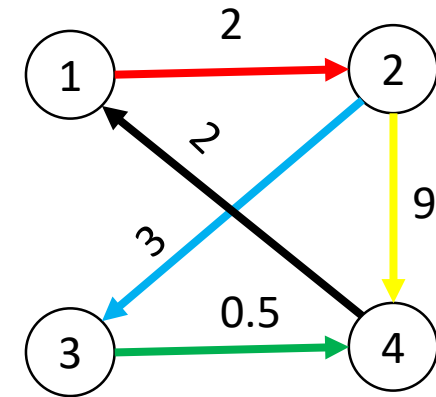
Simple (Undirected) Graph

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



Directed Graph

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



Weighted Graph

	1	2	3	4
1	0	2	0	0
2	0	0	3	9
3	0	0	0	0.5
4	2	0	0	0

Incidence Matrix

- Incidence Matrix $M_{n \times m}$ = Mapping **vertices incident with edges** through a **n x m array**, where **n** is the order of the graph and **m** is the size of the graph

$$M_{n \times m} = \begin{pmatrix} m_{11} & m_{12} & \dots & m_{1m} \\ m_{21} & m_{22} & \dots & m_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ m_{n1} & m_{n2} & \dots & m_{nm} \end{pmatrix}$$

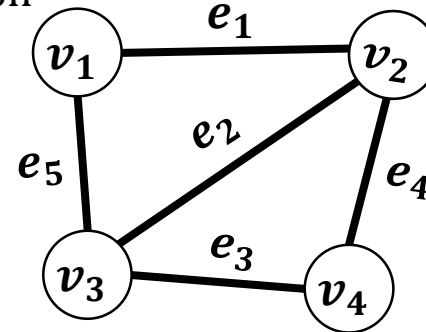
- Rules** of determining the entries m_{ij} between vertices $v_i \in V(G)$ and edges $e_j \in E(G)$,

- $m_{ij} = \begin{cases} 1, & \text{if } v_i \text{ is incident with } e_j \\ 0, & \text{otherwise} \end{cases}$ for **undirected** graph

- $m_{ij} = \begin{cases} 1, & \text{if } e_j \text{ leaves } v_i \\ -1, & \text{if } e_j \text{ enters } v_i \\ 0, & \text{otherwise} \end{cases}$ for **directed** graph

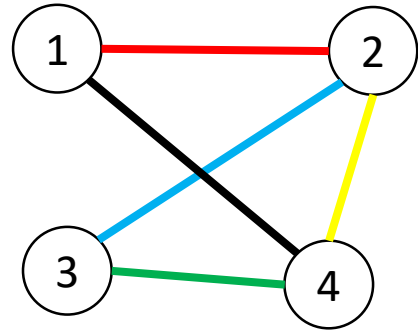
- Note that:

- Some reference book may follow opposite sign notation for directed graph cases



	e_1	e_2	$\dots$	e_m
v_1	m_{11}	m_{12}	$\dots$	m_{1m}
v_2	m_{21}	m_{22}	$\dots$	m_{2m}
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
v_n	m_{n1}	m_{n2}	$\dots$	m_{nm}

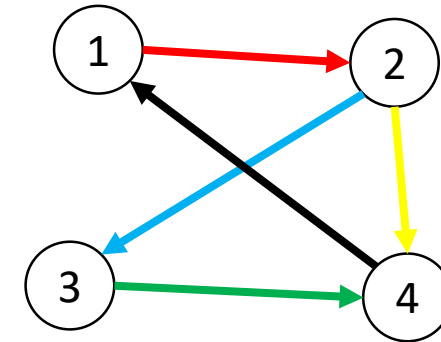
Incidence Matrix



Simple (Undirected) Graph

(1, 2) (1, 4) (2, 4) (2, 3) (3, 4)

	(1, 2)	(1, 4)	(2, 4)	(2, 3)	(3, 4)
1	1	1	0	0	0
2	1	0	1	1	0
3	0	0	0	1	1
4	0	1	1	0	1



Directed Graph

(1, 2) (1, 4) (2, 4) (2, 3) (3, 4)

	(1, 2)	(1, 4)	(2, 4)	(2, 3)	(3, 4)
1	-1	1	0	0	0
2	1	0	-1	-1	0
3	0	0	0	1	-1
4	0	-1	1	0	1