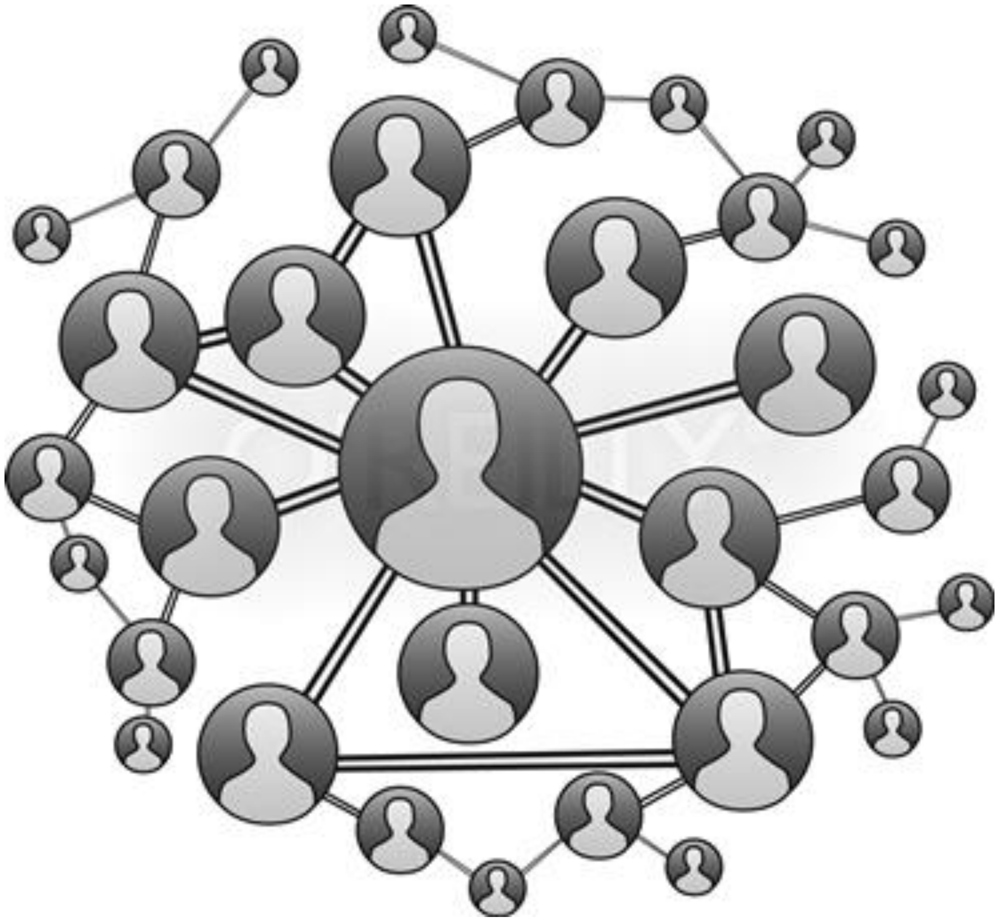
Data structures and algorithms Spring 2025

GRAPH (Part 1)

Lecturer: Do Thuy Duong

Graphs

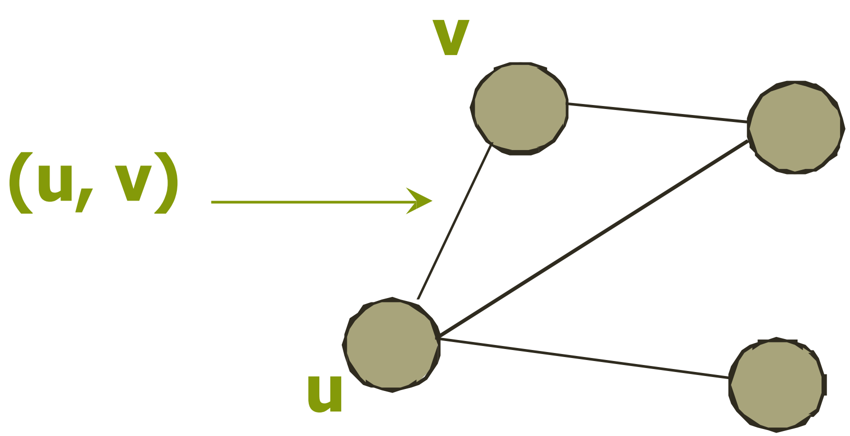


Content

- Graph definition and terminologies
- Graph types
- Graph Traversal
- Topological Sort

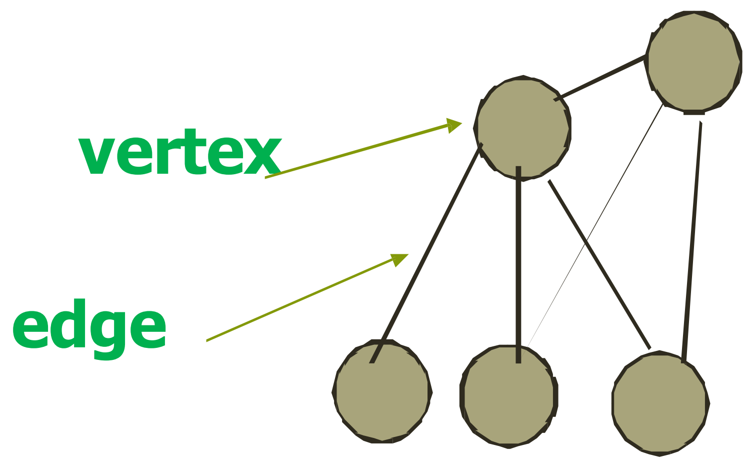
What is a graph? [1]

- **Definition:**
  - A graph **G** is defined as **a pair (V, E)** where
    - V is the **set of nodes (vertices)**
    - E is the **set of edges**
  - For any nodes u and v, if u and v are **connected** by an edge, such **edge** is denoted as **(u, v)**



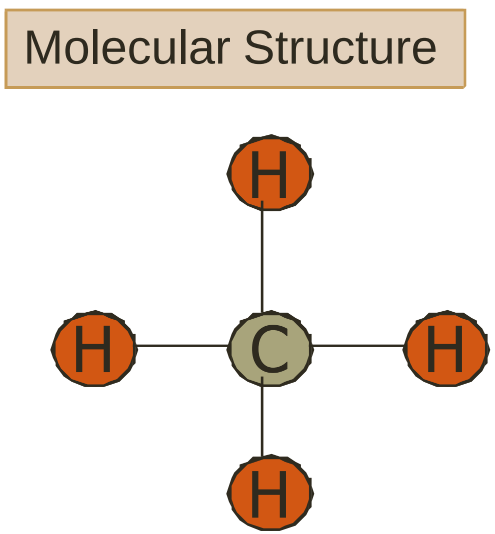
What is a graph? [2]

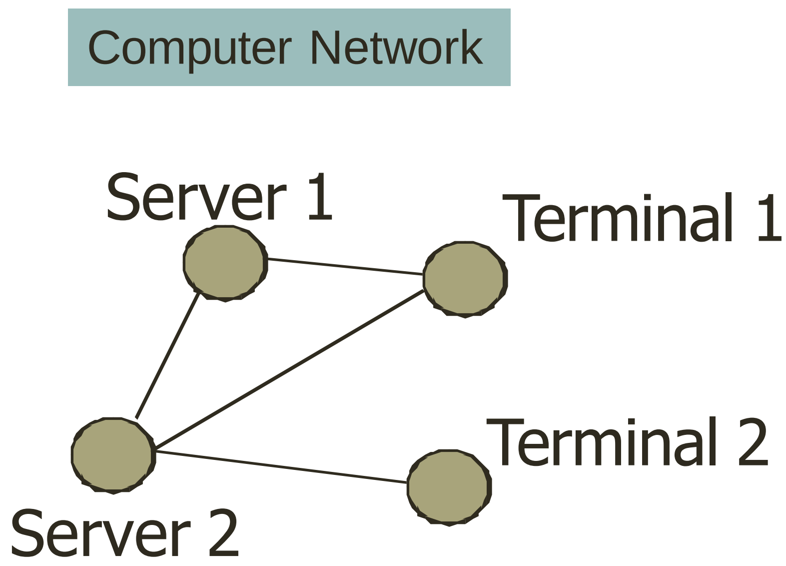
- Graphs also **represent the relationships** among data items:
  - Each node (vertex) represents an item
  - Each edge represents the relationship between two items



What is a graph? [3]

- **Some examples**
  - Molecular structure, network
  - Other examples: electrical and communication networks, airline routes, flow chart, graphs for planning projects



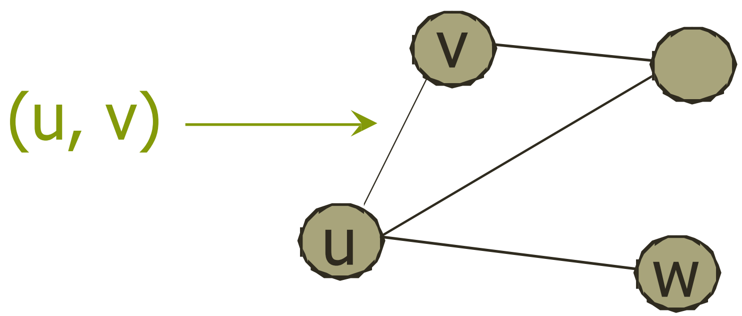


Main types of Graph

- Undirected Graph and Directed Graph
- Weighted Graph and Unweight graph
- And mix of them

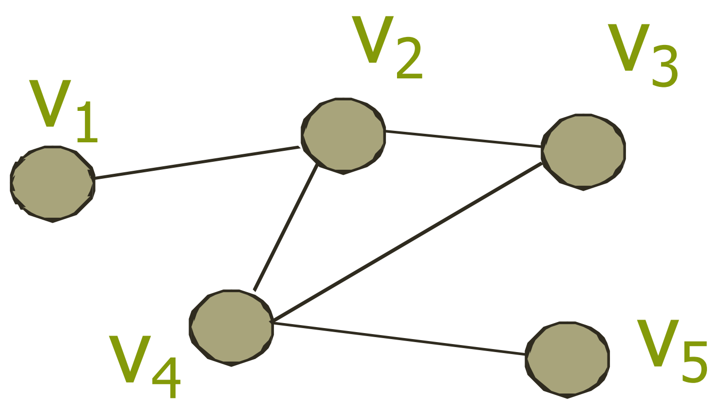
Terminology [1]

- **Adjacent**
  - Two nodes u and v are said to be **adjacent if (u, v) $\in$ E**
  - u and v are adjacent
  - v and w are not adjacent



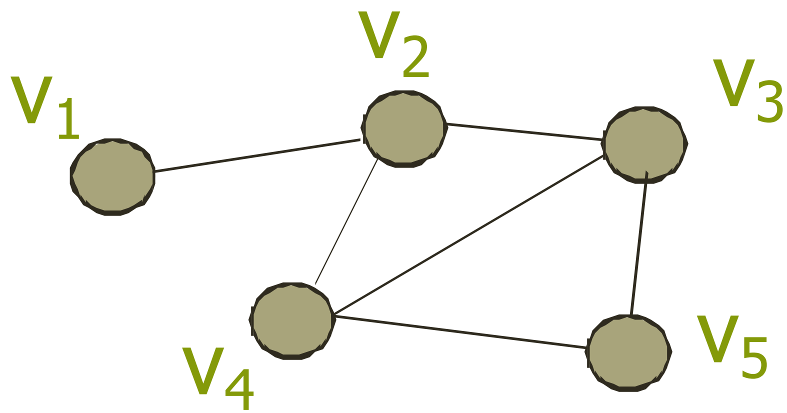
Terminology [2]

- A **path** from v_1 to v_k is a sequence of nodes $v_1, v_2, \ldots, v_k$ that are connected by edges: $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$
- A **simple path** is a path that every node **appears at most once**.
  - Eg: $V_1\ V_2\ V_3$



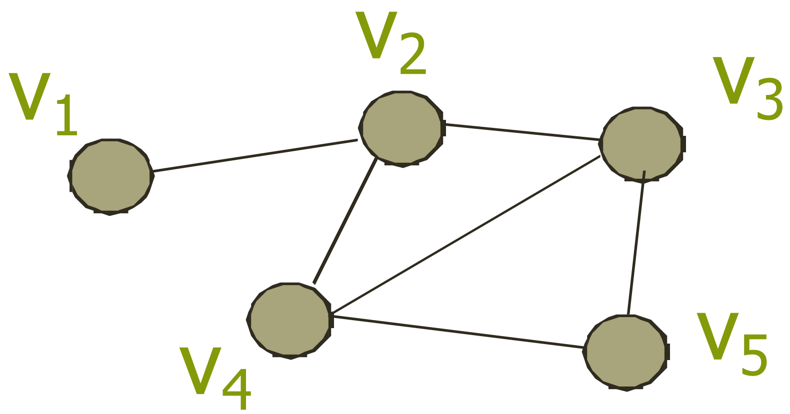
Terminology [3]

- A **cycle** is a path that begins and ends at the same node.
- A **simple cycle** is a cycle if every node **appears at most once**, except for the first and the last nodes
- E.g: $V_2 V_3 V_4 V_2$



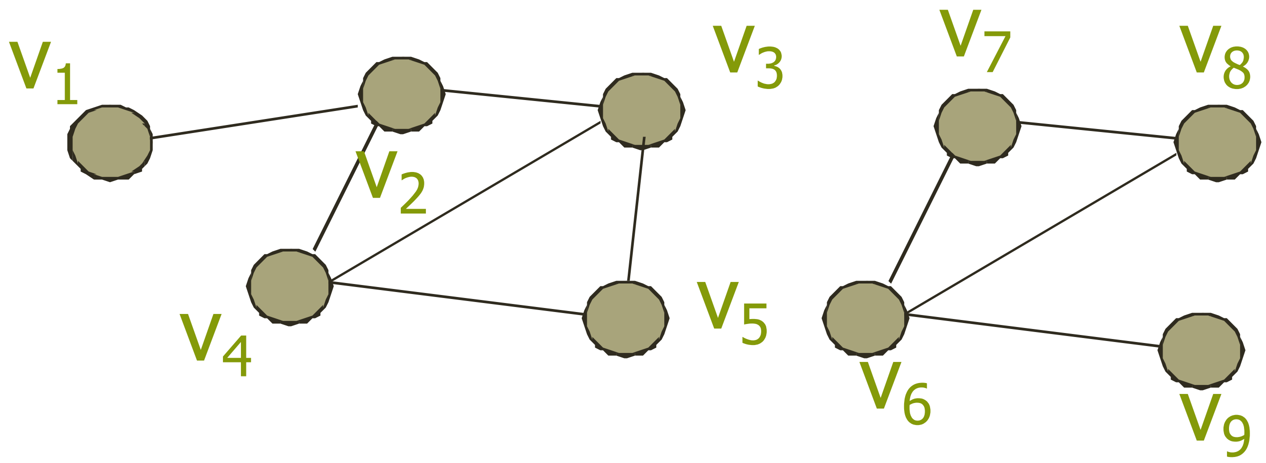
Terminology [4]

- **Connected graph**
  - A graph G is **connected** if there exists **path** between **every** pair of distinct nodes; otherwise, it is **disconnected**



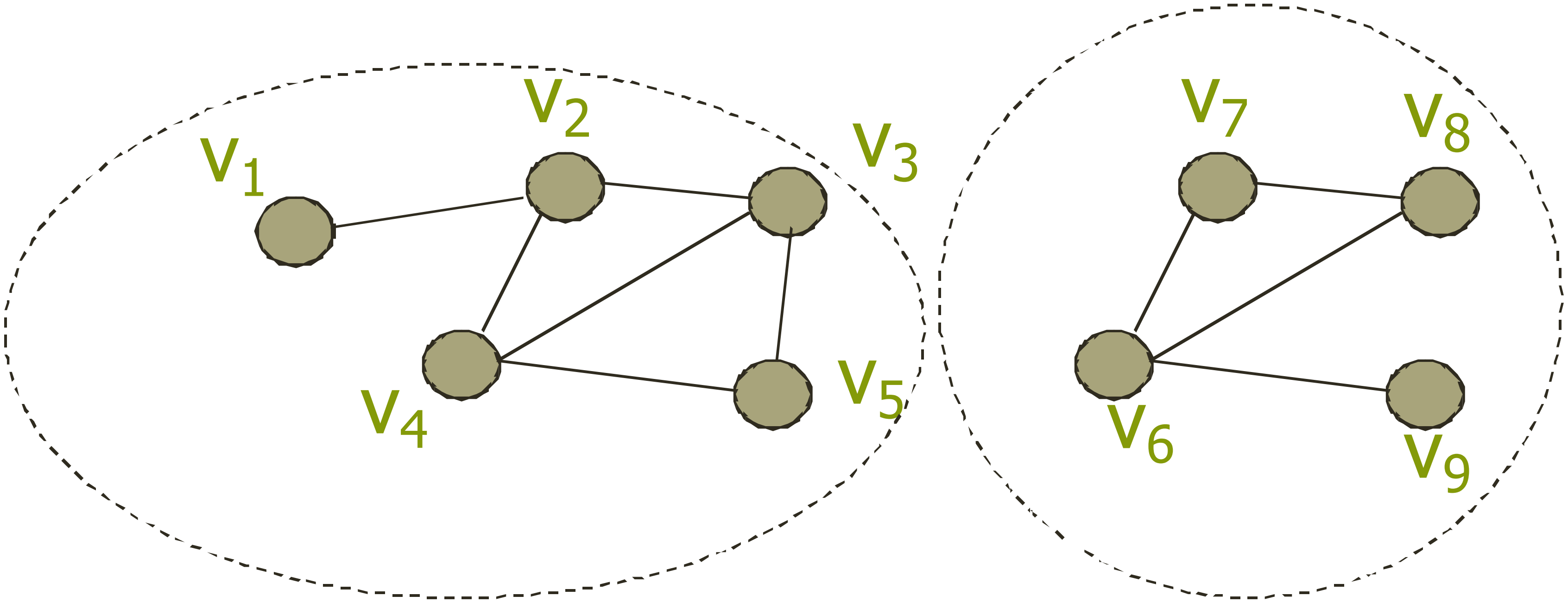
Terminology [5]

- Example of disconnected graph



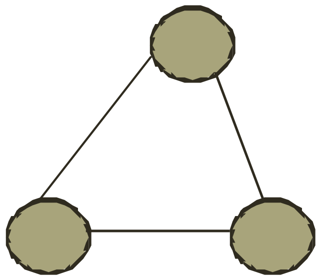
Terminology [6]

- **Connected component**
  - If a graph is disconnect, it can be partitioned into a number of graphs such that each of them is connected. Each such graph is called a **connected component**.

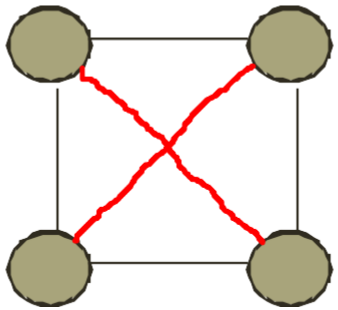


Terminology [7]

- **Complete graph**
  - A graph is **complete** if **each pair of distinct nodes has an edge**



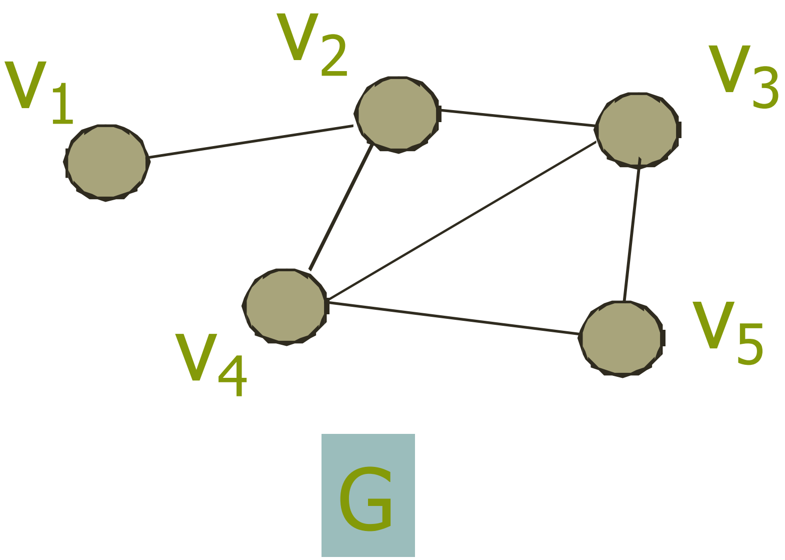
Complete graph with 3 nodes

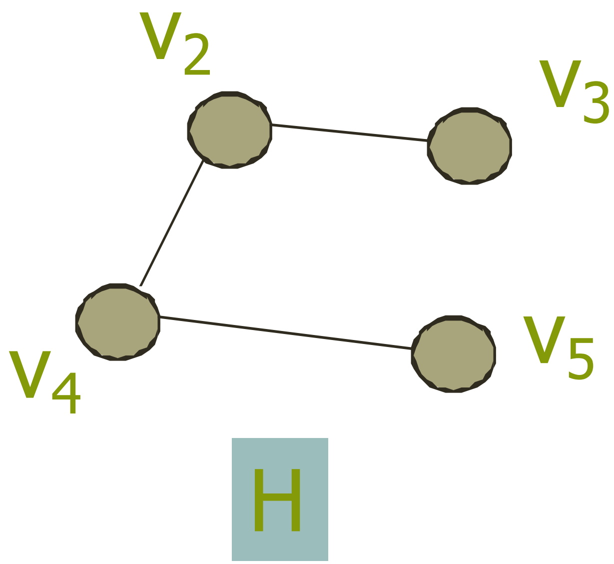


Complete graph with 4 nodes

Terminology [8]

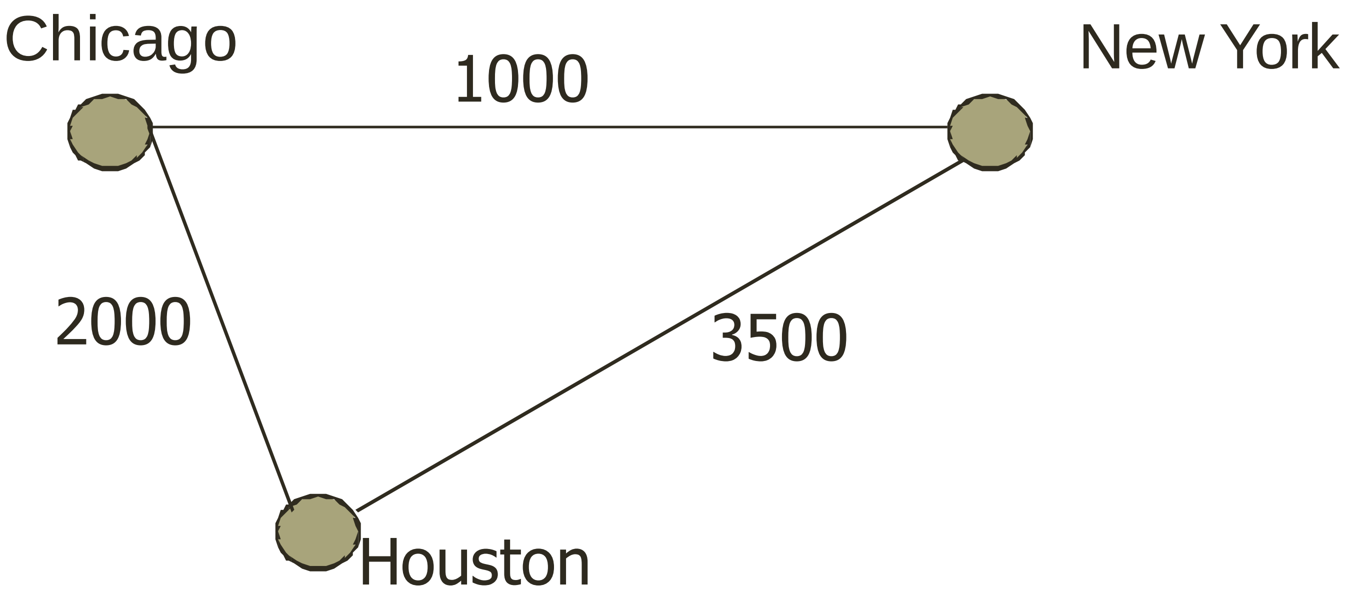
- A **subgraph** of a graph $G = (V, E)$ is a graph $H = (U, F)$ such that $U \subseteq V$ and $F \subseteq E$.





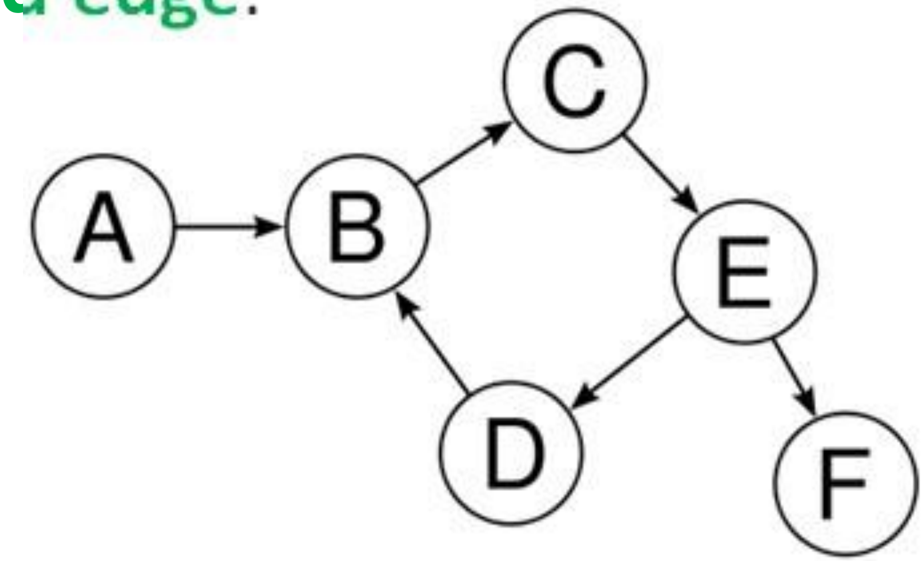
Terminology [9]

- **Weighted graph**
  - If **each edge** in G is **assigned a weight**, it is called a weighted graph

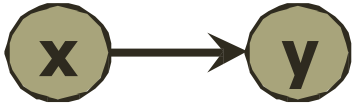


Terminology [10]

- **Directed graph (digraph)**
- If each edge in E has a **direction**, it is called a directed edge.
- A directed graph is a graph where **every edges is a directed edge**.



Terminology [11]



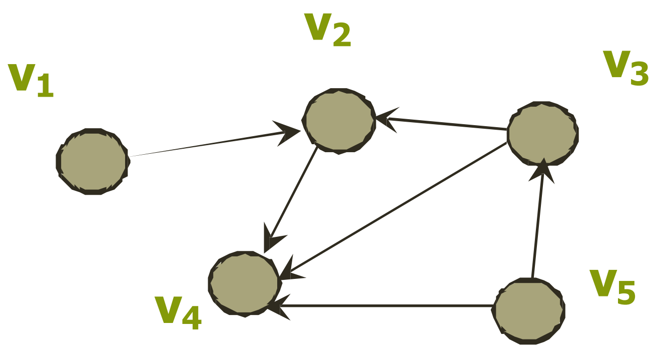
- If (x, y) is a directed edge, we say:
  - y is **adjacent** to x
  - y is **successor** of x
  - x is **predecessor** of y
- In a directed graph, **directed path**, **directed cycle** can be defined similarly.

Graph presentation

2 main ways to represent a graph

- **Adjacency matrix**
  - Represent a graph using **a two-dimensional array**.
- **Adjacency list**
  - Represent a graph using **n linked lists where n is the number of vertices**.

Adjacency matrix for directed graph



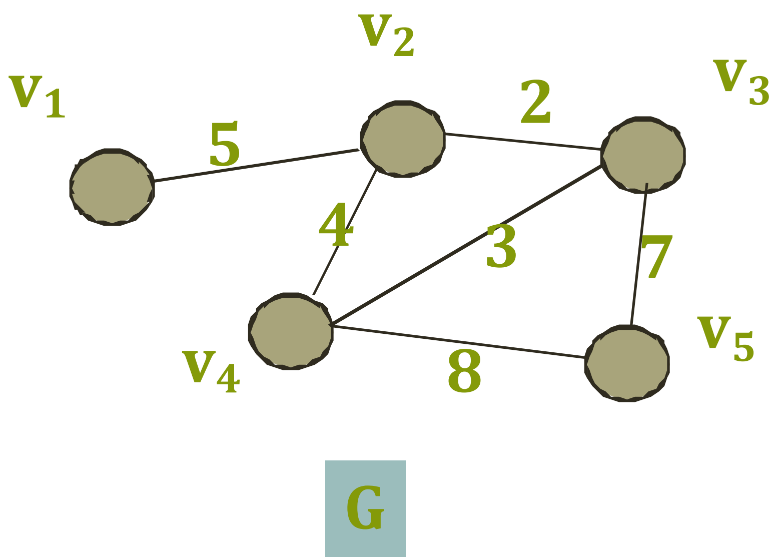
G

$$\text{Matrix } a[i][j] = \begin{cases} 1 & \text{if } (vi, vj) \in E \\ 0 & \text{if } (vi, vj) \notin E \end{cases}$$

		1	2	3	4	5
		v_1	v_2	v_3	v_4	v_5
1	v_1	0	**1**	0	0	0
2	v_2	0	0	0	**1**	0
3	v_3	0	**1**	0	**1**	0
4	v_4	0	0	0	0	0
5	v_5	0	0	**1**	**1**	0

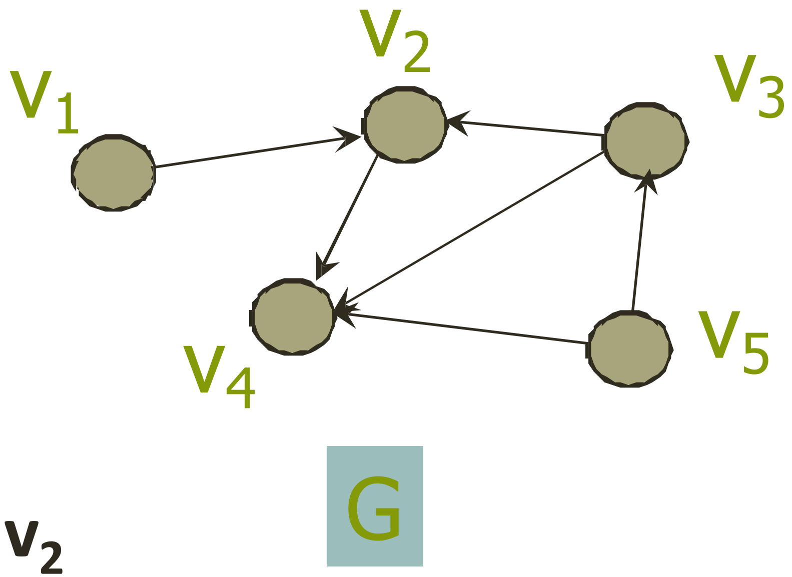
Adjacency matrix for weighted undirected graph

$$\text{Matrix } a[i][j] = \begin{cases} w(vi, vj) & \text{if } (vi, vj) \in E \text{ or } (vj, vi) \in E \\ \infty & \text{otherwise} \end{cases}$$



	v_1	v_2	v_3	v_4	v_5
v_1	∞	5	∞	∞	∞
v_2	5	∞	2	4	∞
v_3	∞	2	∞	3	7
v_4	∞	4	3	∞	8
v_5	∞	∞	7	8	∞

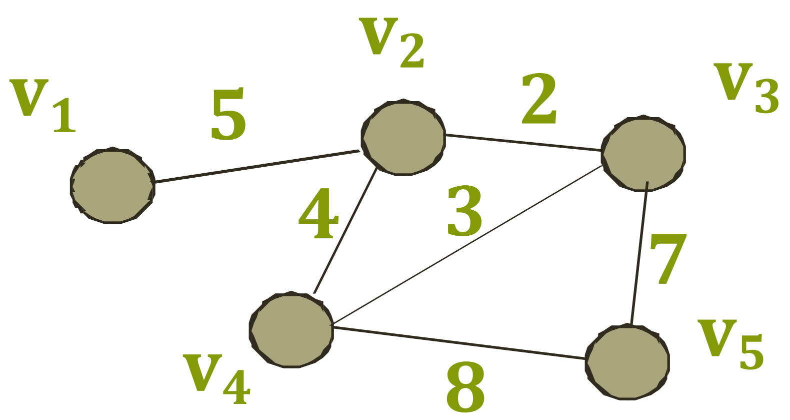
Adjacency list for directed graph



1	v_1	$\rightarrow$	v_2			
2	v_2	$\rightarrow$	v_4			
3	v_3	$\rightarrow$	v_2	$\rightarrow$	v_4
4	v_4					
5	v_5	$\rightarrow$	v_3	$\rightarrow$	v_4

Adjacency list for weighted undirected graph

1	v_1	$\rightarrow$	$v_2(5)$					
2	v_2	$\rightarrow$	$v_1(5)$	$\rightarrow$	$v_3(2)$	$\rightarrow$	$v_4(4)$
3	v_3	$\rightarrow$	$v_2(2)$	$\rightarrow$	$v_4(3)$	$\rightarrow$	$v_5(7)$
4	v_4	$\rightarrow$	$v_2(4)$	$\rightarrow$	$v_3(3)$	$\rightarrow$	$v_5(8)$
5	v_5	$\rightarrow$	$v_3(7)$	$\rightarrow$	$v_4(8)$		



G

Pros and Cons

- **Adjacency matrix**
  - Allows us to determine whether there is an edge from node **i** to node **j** in O(1) time

- **Adjacency list**
  - Allows us to find all nodes adjacent to a given node **j** efficiently
  - If the graph is sparse, adjacency list requires less space

Problems related to Graph

- **Graph Traversal**
- **Topological Sort**

Graph Traversal [1]

- A **graph traversal algorithm** tries to visit all the nodes it can reach.
- If a graph is disconnected, a graph traversal that begins at a node V will visit only a subset of nodes, that is, the connected component containing V.

Graph Traversal [2]

- Two basic graph traversal algorithms:
  - **Depth-first-search (DFS)**
    - After visit node V, DFS strategy proceeds along a path from V as deeply into the graph as possible before backing up.
    - **Supplementary textbook – page 603**.
  - **Breadth-first-search (BFS)**
    - After visit node V, BFS strategy visits every node adjacent to V before visiting any other nodes.
    - **Supplementary textbook – page 594**.

BFS: Breadth-first-search

- **Algorithm**
  - Given a starting vertex **s**
  - Visit all vertices **at increasing distance** from s
    - In the first stage, we visit all the vertices that are at the **distance of one edge away (level 1)**. When we visit there, we paint as "visited".
    - In the second stage, we visit all the new vertices we can reach at the **distance of two edges away (level 2)** from the source vertex s. These new vertices, which are **adjacent to level 1** vertices and not previously painted.
    - …
    - The BFS traversal terminates when every connected vertex has been visited.

BFS: Breadth-first-search

Algorithm BFS(V,E,s):

Input: A graph with V & E are the set of vertices and edges, s is the starting vertex.

Output: All connected vertices in V are visited.

```
For each v in V do Color[v] ← Back
Q ← new empty queue
Color[s] ← Yellow
Q.enqueue(s)
While Q is not empty do
        u ← Q.dequeue()
        For each v adjacent to u do
                If Color[v] is Black then
                        Color[v] ← Yellow
                        Q.enqueue(v)
Endwhile
```

BFS: Breadth-first-search

- https://www.youtube.com/watch?v=QRq6p9s8NVg

BFS: Breadth-first-search

Time complexity of BFS

- **See supplementary textbook, page 597**.
- Consider the graph **G=(V,E)**, where V is the set of vertices and E is the set of edges.
- BFS has the time complexity: **O(|V|+|E|)**.

DFS: Depth-first-search

- DFS is a systematic way to find all the vertices reachable from a source vertex s:
  - Explore every edge connected to S.
  - As soon as discovering a vertex, DFS starts exploring from it.
    - Unlike BFS, which puts a vertex on a queue so that it explores from it later
  - Can trackback and start over from a vertex as necessary.

DFS: Depth-first-search

- **DFS algorithm:**
  - With a current vertex **u**:
    - Examining an edge **(u,v)** that connects **u** to **v**.
    - If **v** is already painted, **back down** to **u** and examine another edge **(u,v1)**.
    - If **v** is unpainted, pain **v**, consider **v** as the current vertex. Repeat the process above.
  - A **dead-end** (**dead-lock**) situation:
    - All the edges from our current vertex u takes us to painted vertices.
    - Repeat backing down along the edge that brought us here to vertex u. Try with another vertex.

DFS: Depth-first-search

Code

Algorithm DFS(u):
Input: u is the current vertex in a graph G=(V,E).
Output: All connected vertices in V are visited.

Color[u]$\leftarrow$ Yellow

For each *v adjacent to u* **do**
If *Color[v] is Black* **then**
DFS(v)

DFS: Depth-first-search

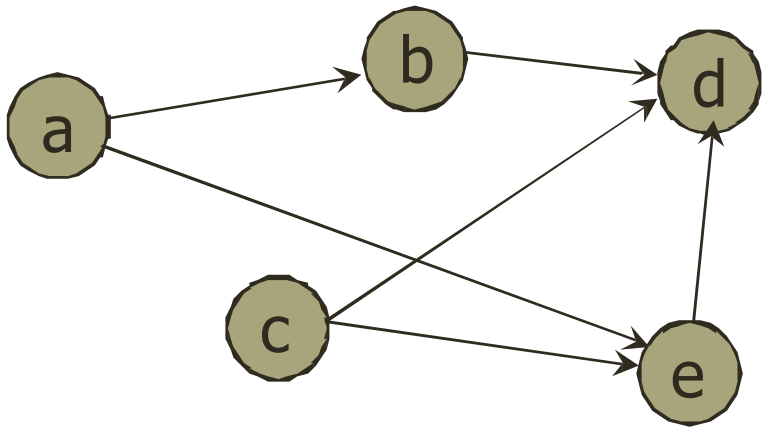
- https://www.youtube.com/watch?v=iaBEKo5sM7w

DFS: Depth-first-search

- **Time complexity of DFS**
  - **Supplementary textbook – page 606**.
  - Consider the graph **G=(V,E)**, where V is the set of vertices and E is the set of edges.
  - DFS has the time complexity: **O(|V|+|E|)**.

Topological order [1]

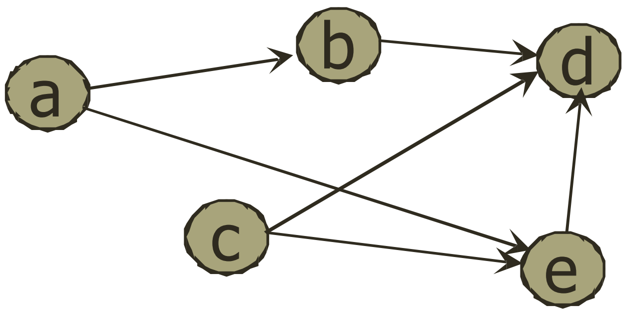
- Consider the prerequisite structure for courses:



1. Each **node x represents a course x**.
2. **(x, y)** represents that **course x is a prerequisite to course y**.
3. This graph should be a directed graph without cycles.

Topological order [2]

- Consider the prerequisite structure for courses:



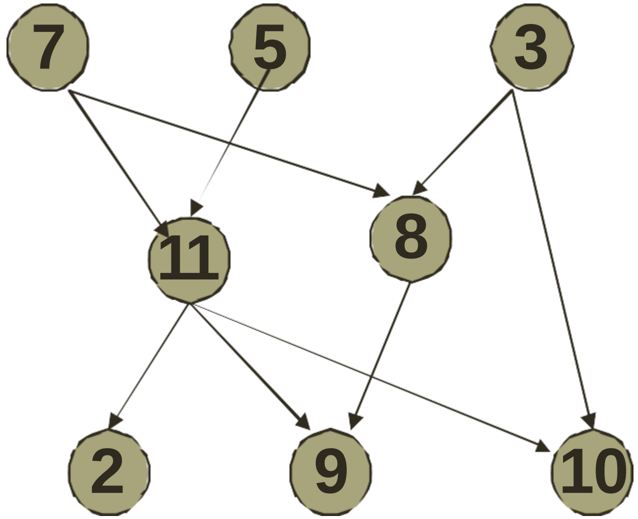
- A linear order to take all 5 courses while satisfying all prerequisites is called a **topological order**.
- Example:
  - a, c, b, e, d
  - c, a, b, e, d

Topological Sort [1]

- Topological sort:
  - **Ordering** of vertices in a directed graph such that if there is **a path** from v_i to v_j then v_j **appears after** v_i in the ordering.
- Application: scheduling jobs
  - Each job is a vertex in a graph, and there is an edge from *x* to *y* if job *x* must be completed before job *y* can be done.
  - Topological sort gives the order in which to perform the jobs.

Topological Sort [2]

- Topological sorts example:
- 7, 5, 3, 11, 8, 2, 10, 9
- 5, 7, 3, 8, 11, 2, 9, 10
- 5, 7, 11, 2, 3, 8, 9, 10



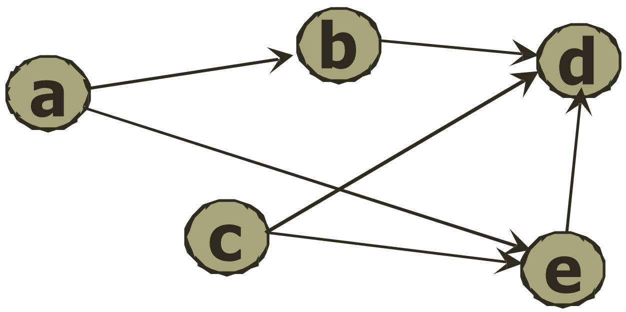
Topological Sort [3]

Code

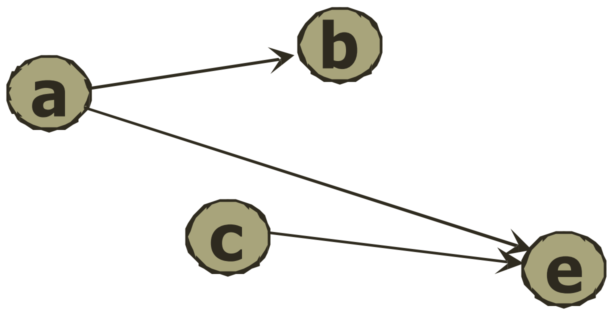
```
Algorithm  TopoSort1()
    Input: A graph G=(V,E).
    Output: A topological order.

    Do
      For each v in V do
        If  v has no successor  (no outgoing edges)  then
            Add v to the result set
            Delete v from V (remove all edges come to v)
    While V is not empty
```

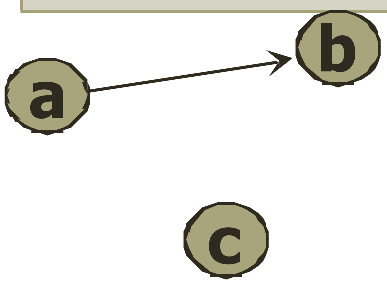
Topological Sort [4]



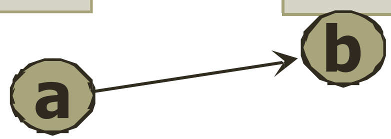
1. d has no successor! Choose d!



2. Both b and e have no successor! Choose e!



3. Both b and c have no successor! Choose c!



4. Only b has no successor! Choose b!

5. Choose a!

The topological order is a,b,c,e, d

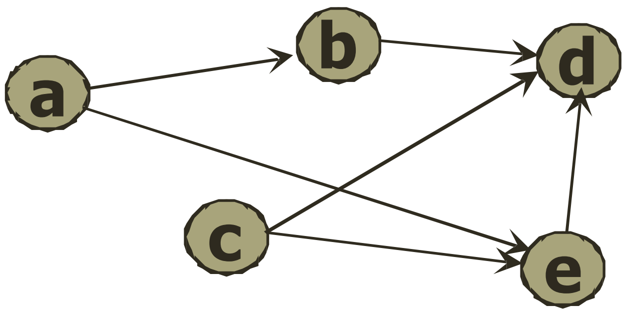
Topological Sort [5]

Code

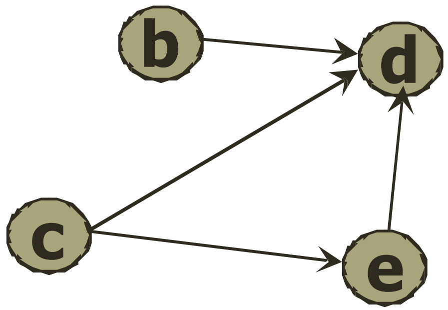
```
Algorithm  TopoSort2()
    Input: A graph G=(V,E).
    Output: A topological order.

    Do
      For each v in V do
        If  v has no ancestor  (no incoming edges) then
            Add v to the result set
            Delete v from V (remove all edges come to v)
    While V is not empty
```

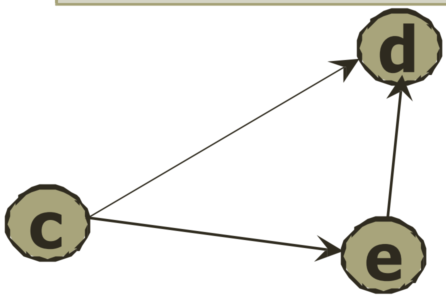
Topological Sort [6]



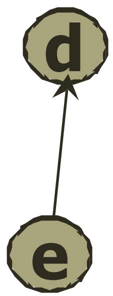
1. a, c has no ancestors! Choose a!



2. Both b and c have no ancestors! Choose b!



3. Only c has no ancestors! Choose c!



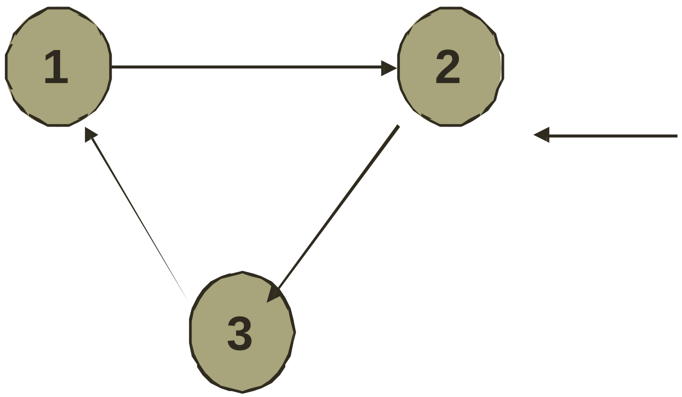
4. Only e has no ancestors! Choose e !

5. Choose d!

The topological order is **a,b,c,e,d**

Topological Sort [7]

- What happens if graph has a cycle?
  - Topological ordering is not possible
  - For two vertices v & w, v precedes w and w precedes v



Every edge has an incoming vertex so topological sort can not be performed

- Topological sorts can have more than one ordering

Tutorial & next topic

- **Preparing for the tutorial:**

Practice with examples and exercises in Tutorial 10

- **Preparing for next topic:**

- Read textbook chapter 9 (9.1 – 9.3): Graph algorithms.
- Read supplementary book chapter 22, 24 and chapter 25