# DET102 Data Structures and Algorithms

Lecture 8: Graph

# Outlines

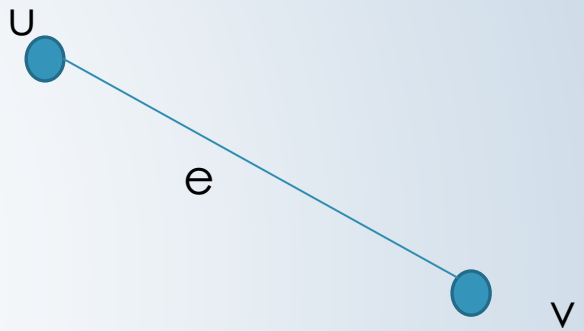
- Terms & Definitions
- Representations of graphs
- Graph traversals
  - BFS – Breath-first search
  - DFS - Depth-first search

# Definition

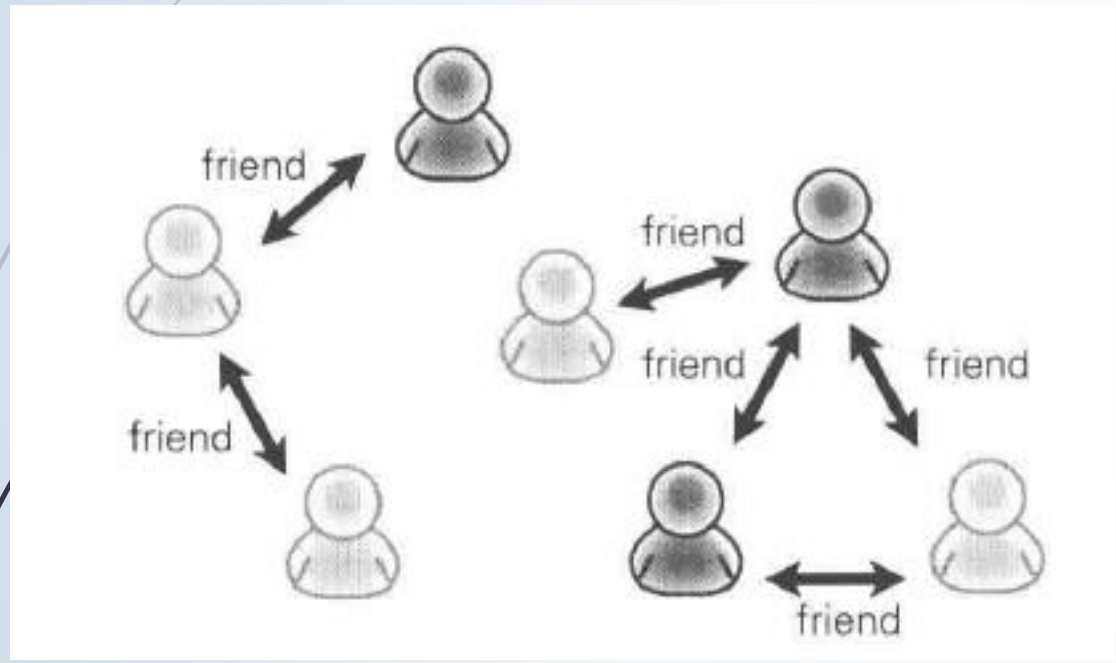
- G=(V,E)
  - Vertex: v, |V|=n
  - Edge: e, |E|=m

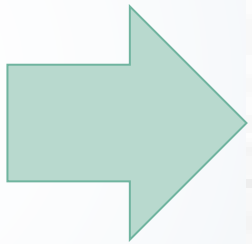
Adjacency(鄰接): u ~ v

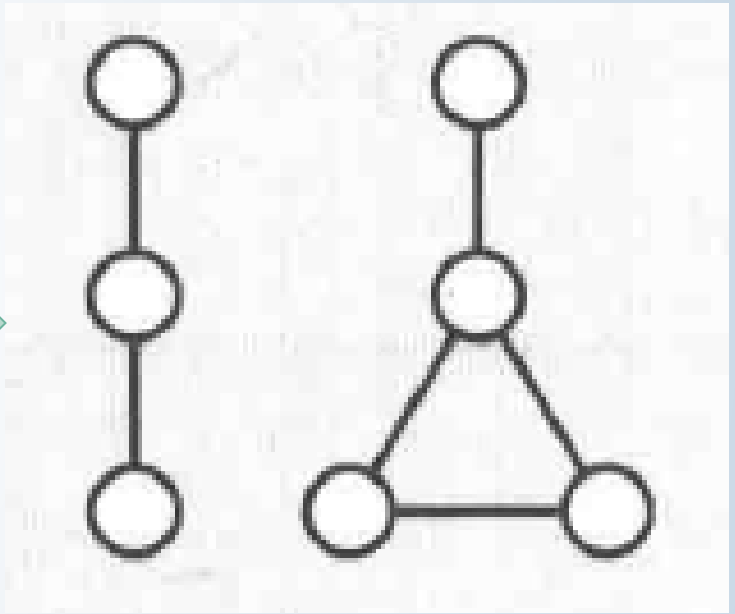
Incidence (關聯): u ~ e, v ~ e



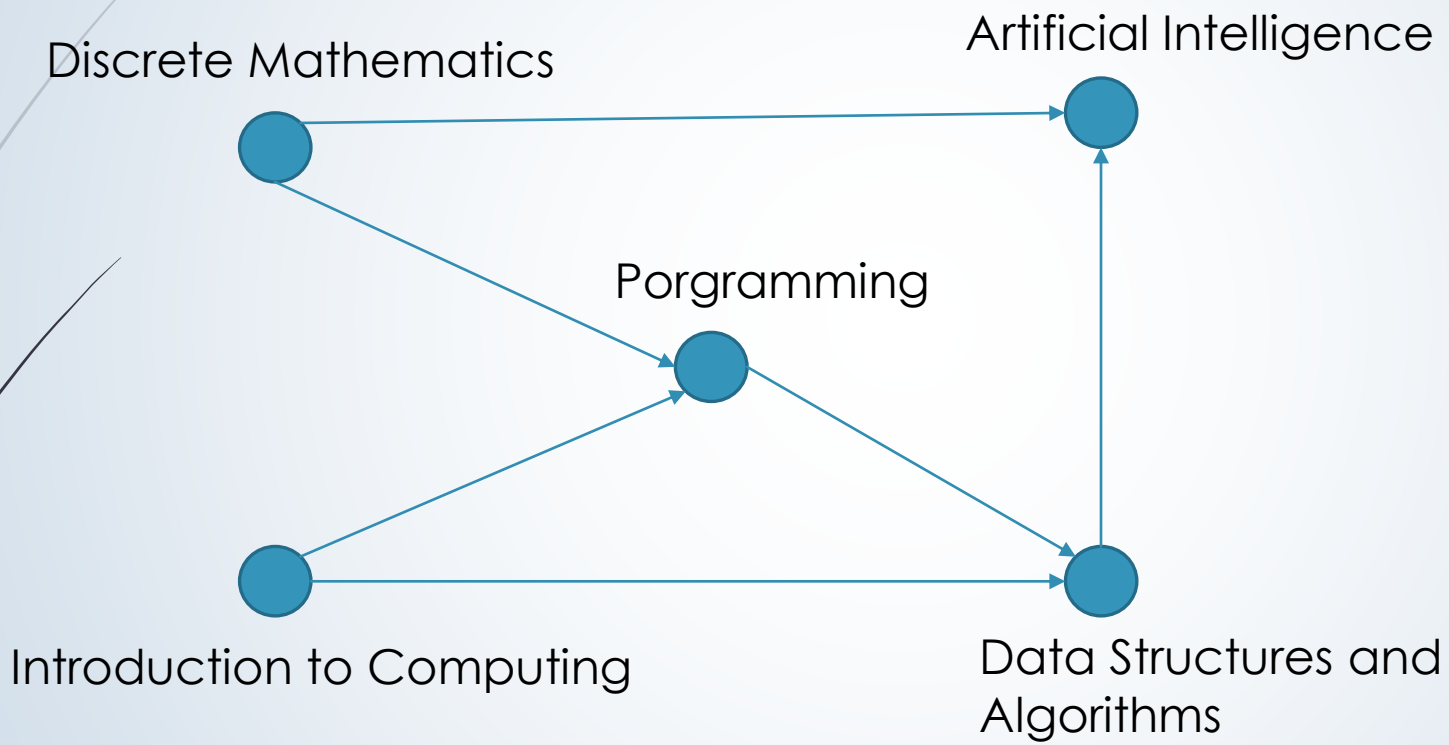
# Undirected graphs

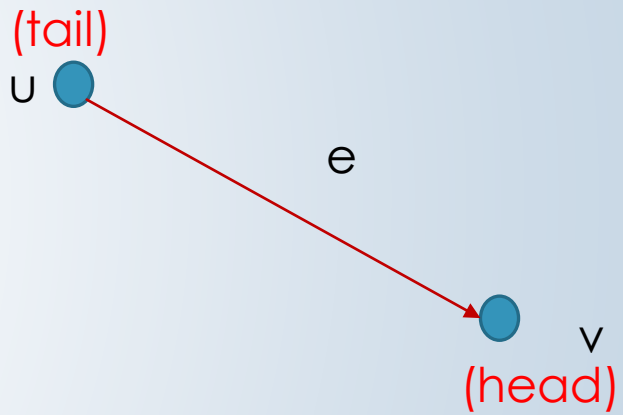




# Directed graph

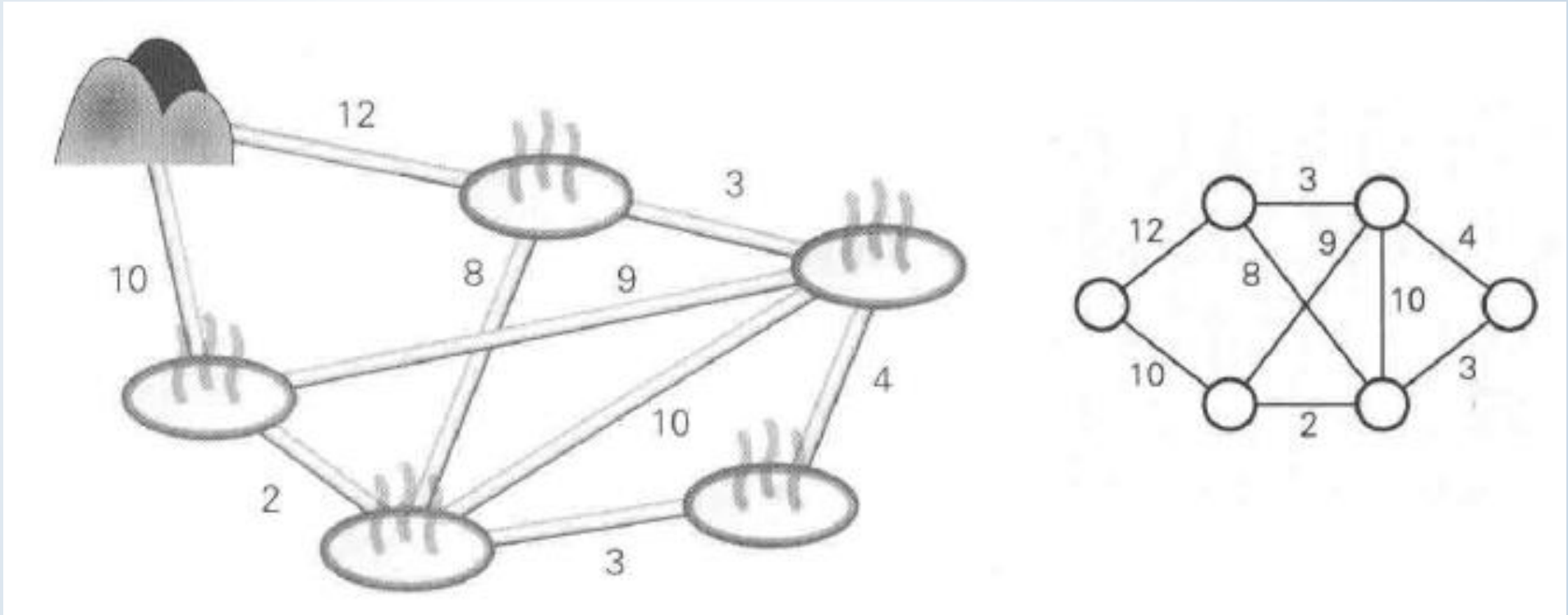


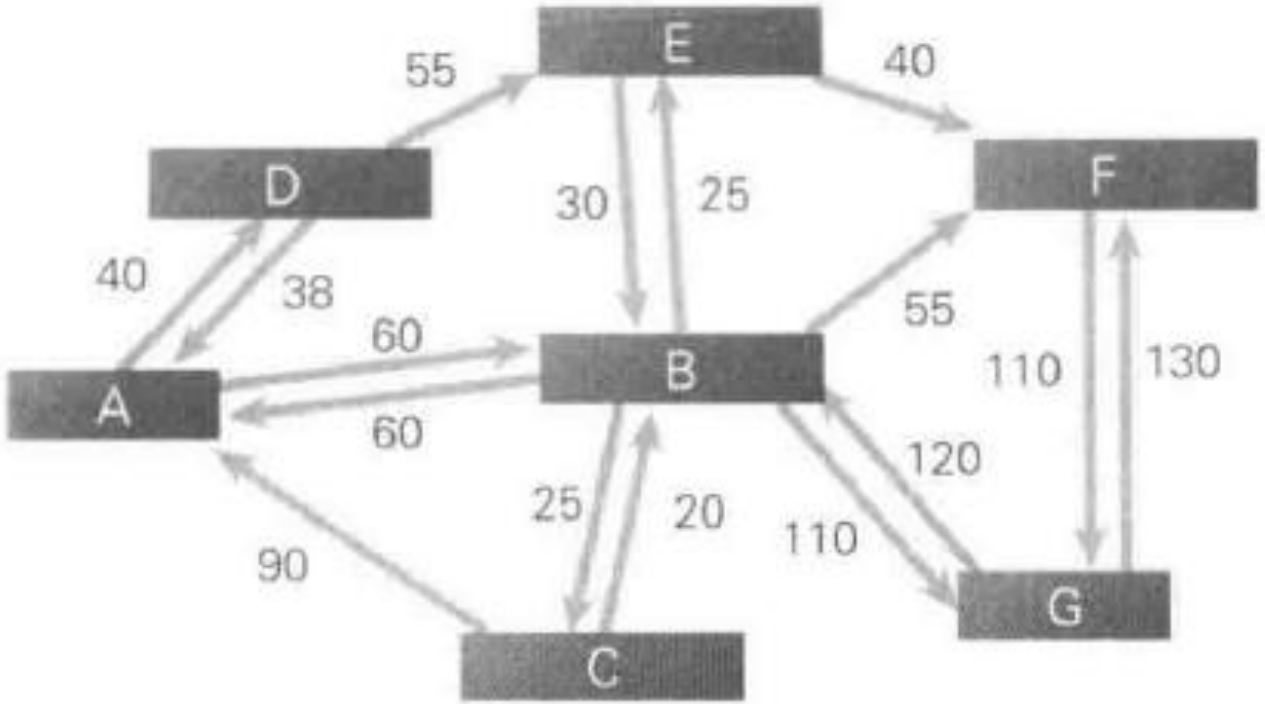

How to schedule your study plan ?

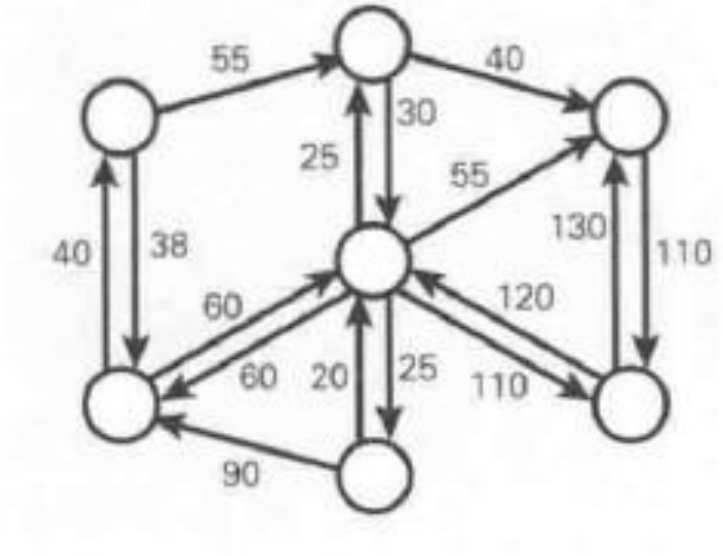
# Mixed Graph

# Weighted Undirected Graph



# Weighted Directed Graph



# Types of Graphs

| Graph | Features |
| --- | --- |
| Undirected graph | Edges are undirected |
| Directed graph | Edges are directed |
| Weighted undirected graph | Edges have weights but no directions |
| Unweighted directed graph | Edges have weights and directions |

| V | : number of vertices
| E | : number of edges

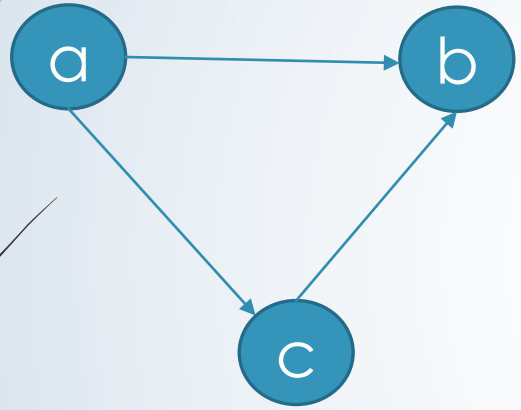
# Terms and definitions

- A graph *G* consists of:
  - A non-empty set of vertices (or nodes): *V*
  - A set of edges: *E*
  - *E* & *V* are related in a way that the vertices on both ends of an edge are members of *V*
  - Usually written as *G* = (*V*, *E*)

❖ Usually Vertices are used to represent a position, an object or state meanwhile Edges are used to represent a transaction or relationship

# Edge

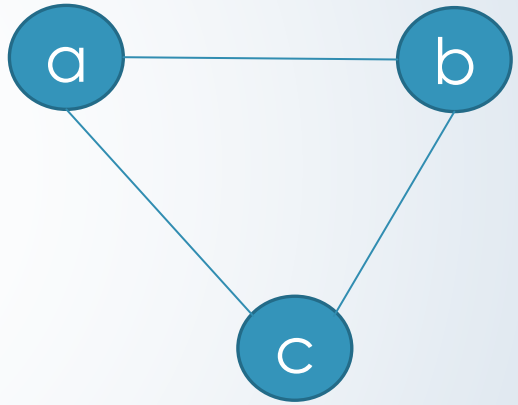
- Ordered pair (u,v)




E={ (a, b), (c, b), (a, c) }

in the Directed Graph

- Unordered pair {u,v}



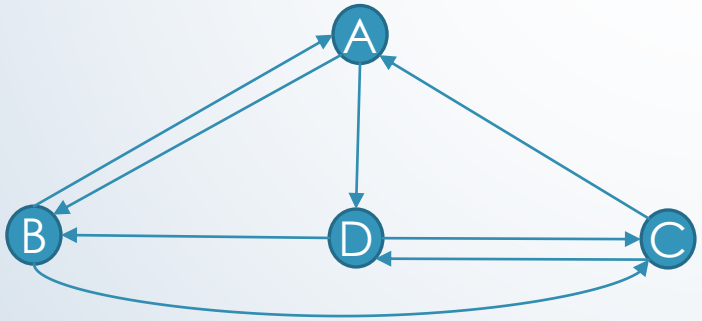

E={ (a, b), (a, c) , (b, c) }

in the Undirected Graph

(a,b) and (b,a) represent the same edge. a and b are **adjacent.**

# Path

- A *path* is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex.
  - Simple path: each vertex in the path is distinct
  - directed path: all edges are directed and are traversed along their direction

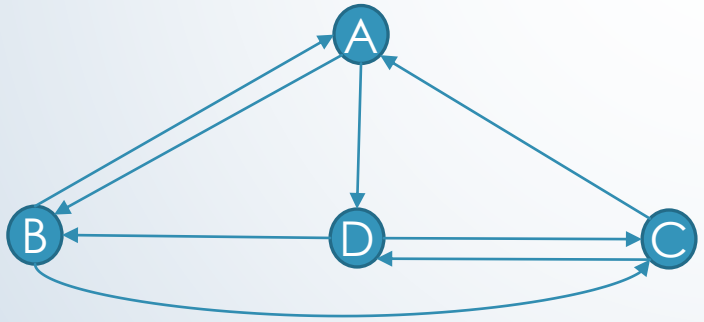



Simple path: <C,A,D,B>

Path: <C,A,B,A,D>

# Cycle

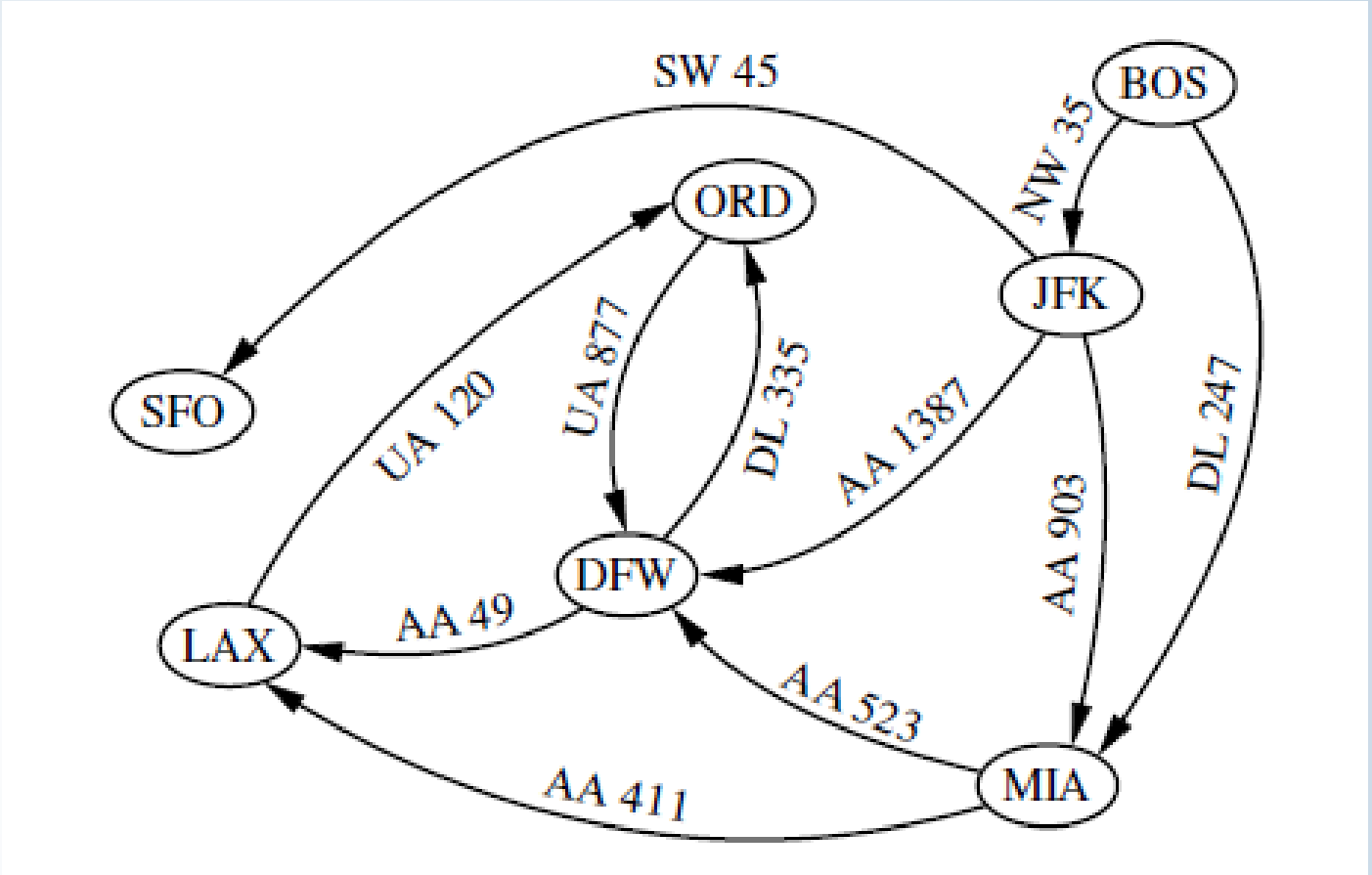
- A *cycle* is a path that starts and ends at the same vertex, and that includes at least one edge.
  - Simple cycle: each vertex in the cycle is distinct, except for the first and last one.
  - directed cycle: all edges are directed and are traversed along their direction




Simple Cycle: <C,A,B,C>
Cycle: <C, A, D, B, A, B, C>

Eulerian tour: <C,A,D, C, D, B,A,B,C>
Hamiltonian tour: <C, A, D, B, C>

HKG

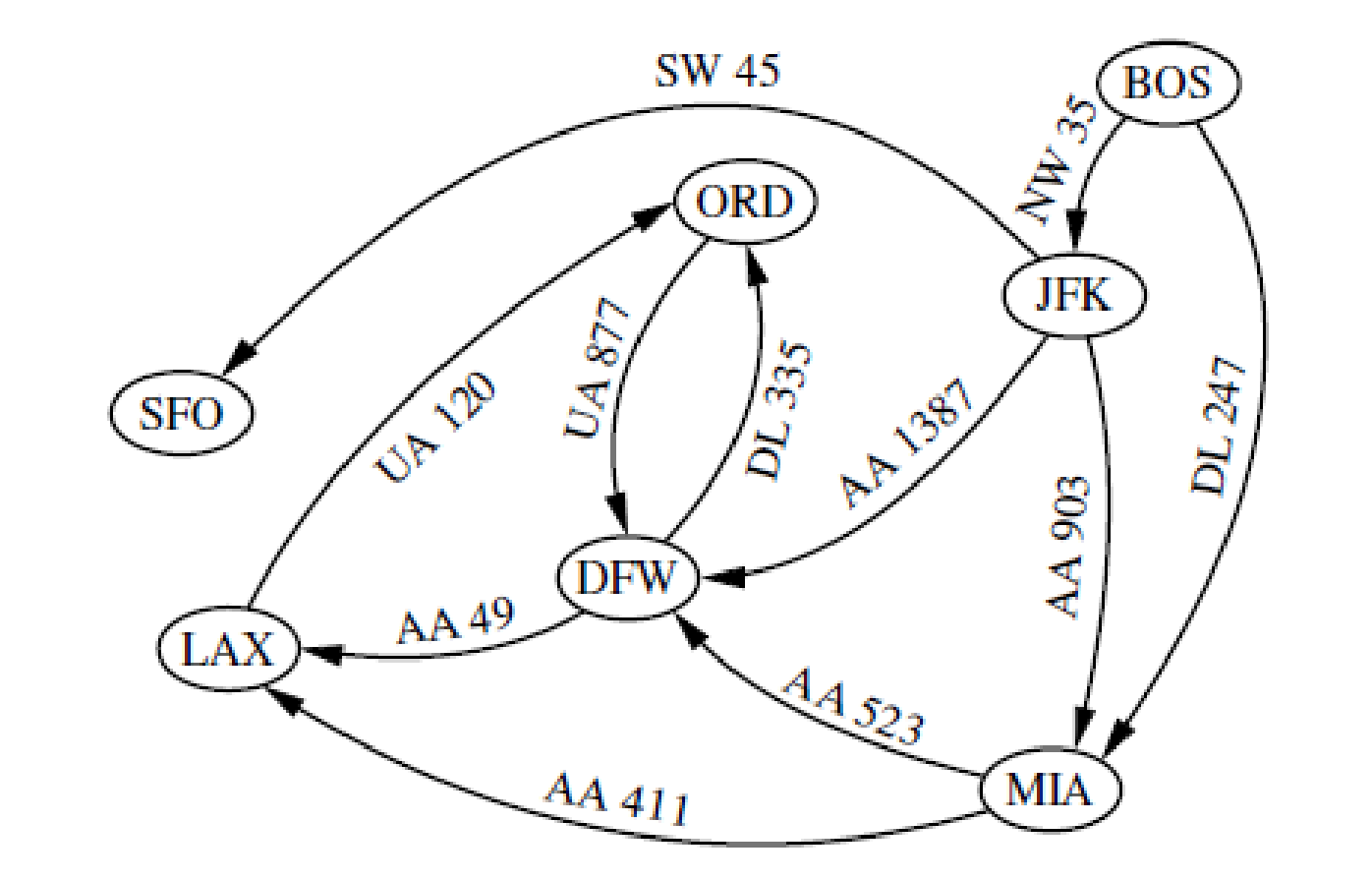



How to represent the two flights from HKG to SFO?

parallel edges or multiple edges

How to represent flycation in Hong Kong?

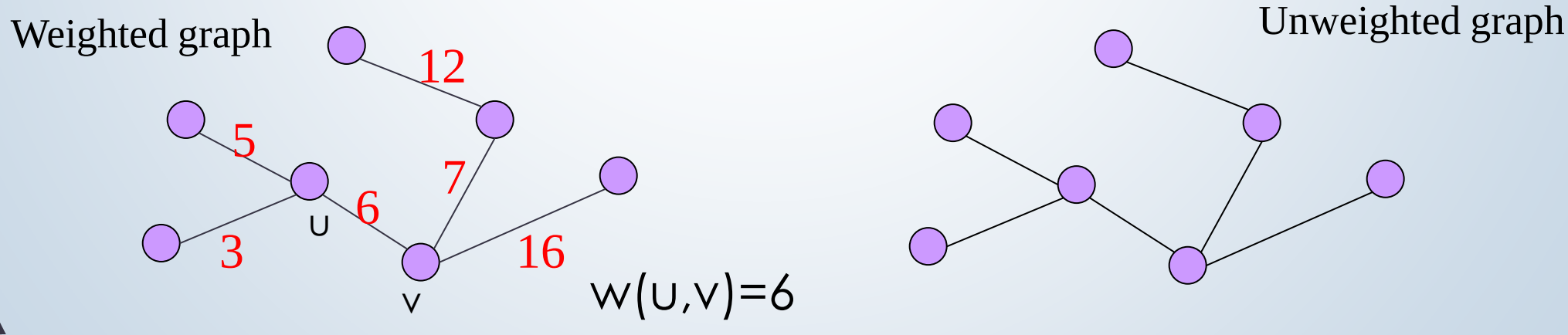
Self-loop




Find a path and a cycle from the above graph.

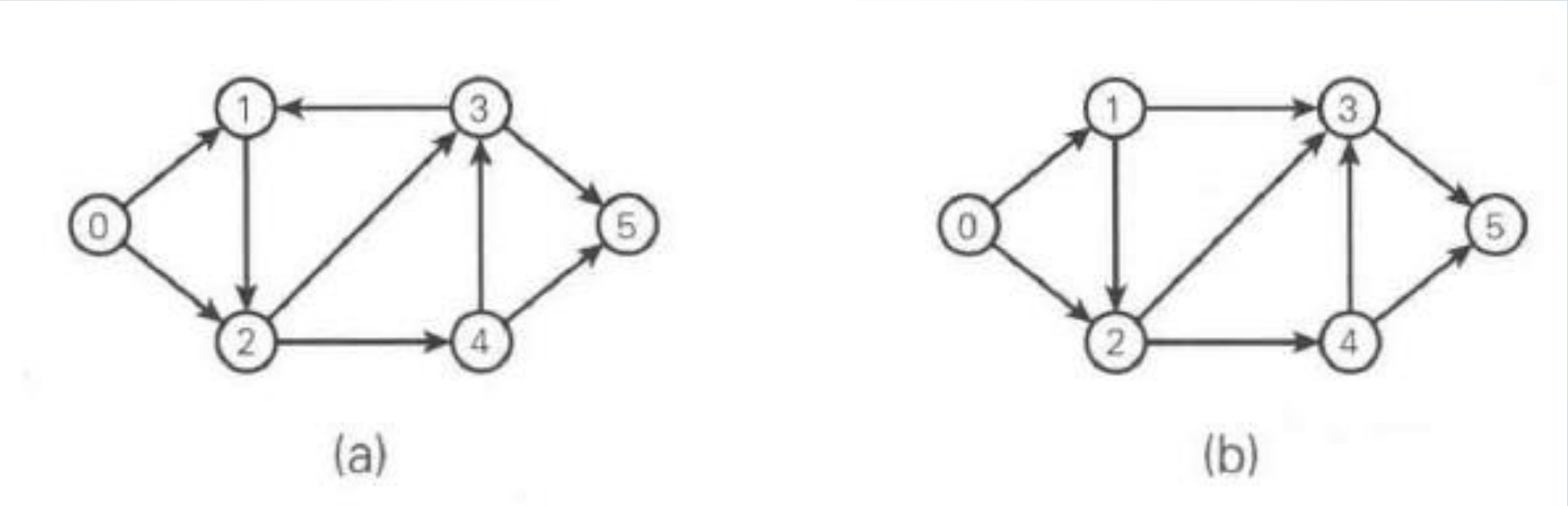
# Weight

- Graph can be un-weighted or weighted, in which a value is associated with each edge.
- In directed graph, the weights of edges going in opposite directions can be different.
- For example:
  - *Whether a bus can go from TKO to Shatin:* Unweighted (=1...)
  - *The bus fee it takes from TKO to Shatin:* Weighted



# Directed Acyclic Graph (DAG)

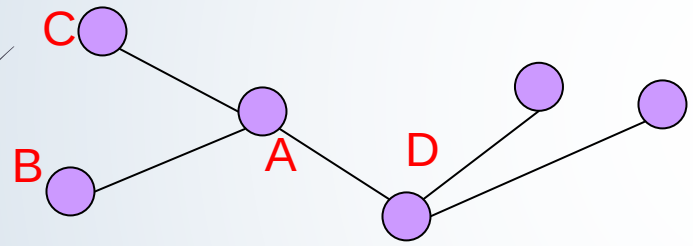
no cycle




Which is a DAG ?

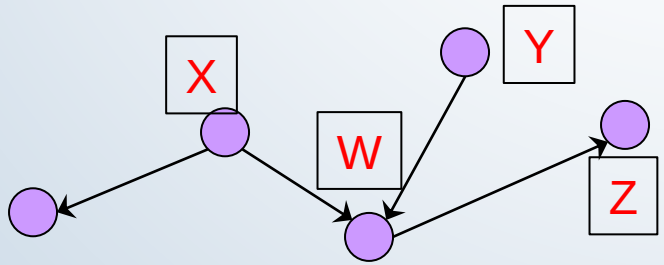
# Degree

- Degree of a vertex is the number of edges connecting to it




Node A is having a degree of **3** (connects B, C, and D)

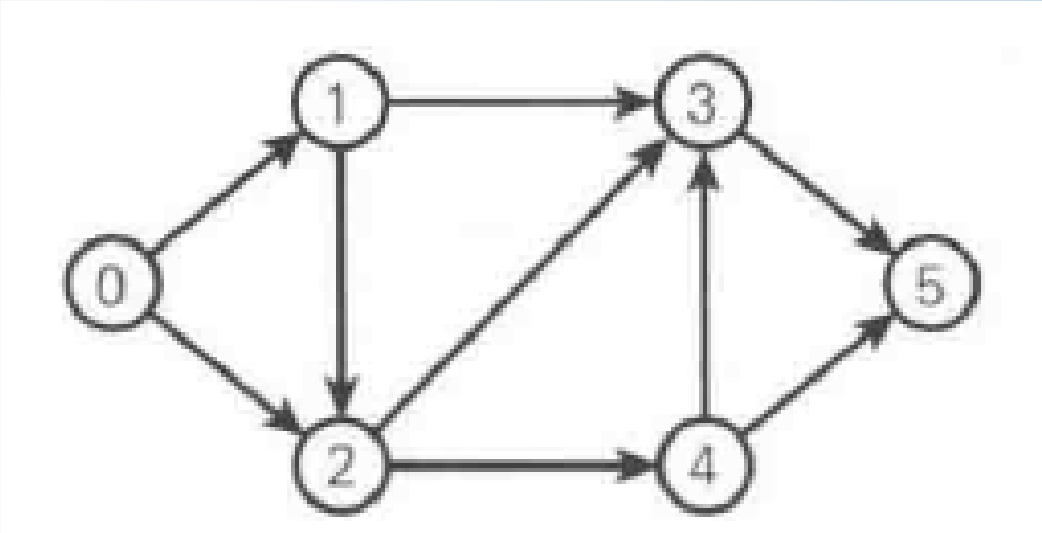
- For directed graph, degree is further classified as in-degree *(to this vertex)* & out-degree *(from this vertex)*




Node W is having an in-degree of **2** (X, Y) and an out-degree of **1** (Z)

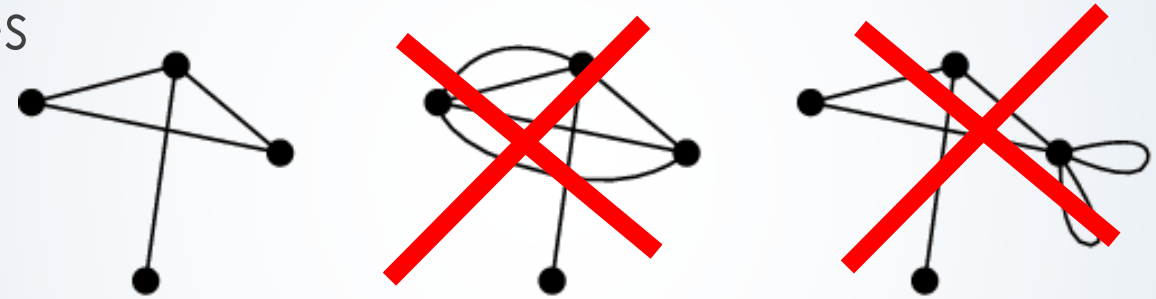
# Exercise

| Vertex | In-degree | Out-degree |
|---|---|---|
| 0 | | |
| 1 | | |
| 2 | | |
| 3 | | |
| 4 | | |
| 5 | | |

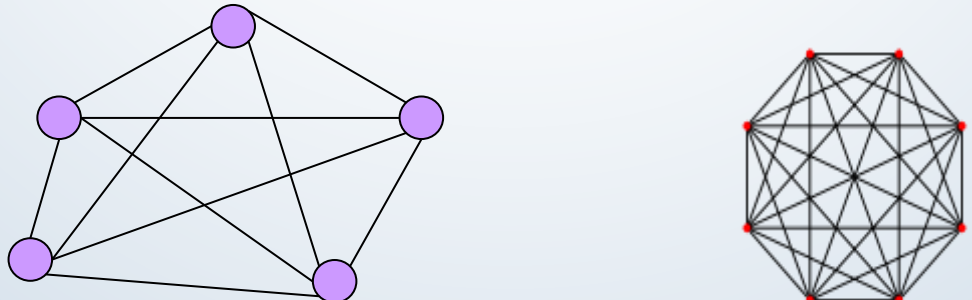


# Simple Graph V.S. Complete graph

- Simple graph:
  - an un-weighted, undirected graph containing no graph loops or multiple edges



- Complete graph:
  - A simple graph in which every pair of vertices are connected directly.
  - If number of vertices = V, number of edges = ?



# Connected Components

- Subgraph is a graph whose vertices and edges are subsets of another graph.
- A graph G'=(V', E') is a subgraph of another graph G=(V, E) iff
  - $V' \subseteq V$, and
  - $E' \subseteq E \wedge ( (v_1, v_2) \in E' \rightarrow v_1, v_2 \in V')$.
- Note: In general, a subgraph need not have all possible edges. If a subgraph has every possible edge, it is an induced subgraph.
- A connected component or simply component of an undirected graph is a subgraph in which each pair of nodes is connected with each other via a path.

# Properties of graph



- If $G$ is an undirected graph with $m$ edges and vertex set $V$, *then*

$$\sum_{v \text{ in } V} deg(v) = 2m$$

- If $G$ is a directed graph with $m$ edges and vertex set $V$, then

$$\sum_{v \text{ in } V} indeg(v) = \sum_{v \text{ in } V} outdeg(v) = m.$$

- Let $G$ be a simple graph with $n$ vertices and $m$ edges. If $G$ is undirected, then $m \leq n(n-1)/2$, and if $G$ is directed, then $m \leq n(n-1)$.
- Let $G$ be an undirected graph with $n$ vertices and $m$ edges.
- If $G$ is connected, then $m \geq n-1$.
- If $G$ is a tree, then $m = n-1$.
- If $G$ is a forest, then $m < n-1$.

# Graph Representation

# Vertex ADT

- A graph is a collection of vertices and edges.
- We model the abstraction as a combination of three data types: Vertex, Edge, and Graph.

- Vertex: stores an arbitrary element provided by the user (e.g., an airport code);
  - we assume it supports a method, element(), to retrieve the stored element.

# Edge ADT

- Edge: stores an associated object (e.g., a flight number, travel distance, cost),
  - retrieved with the element() method.
- In addition, we assume that an Edge supports the following methods:
  - endpoints( ): Return a tuple (*u*,*v*) such that vertex *u* is the origin of the edge and vertex *v* is the destination; for an undirected graph, the orientation is arbitrary.
  - opposite(v): Assuming vertex *v* is one endpoint of the edge (either origin or destination), return the other endpoint.

# Representations of graphs

- When working with graph, we always perform one of the following operation:
  - Get the list of vertices connecting a given vertex.
  - Are vertices A  and B connected?
  - What is the weight of edge from A to B?
  - What is the in/out degree of a vertex?

# Graph ADT

- Methods:
  - vertex count(): Return the number of vertices of the graph.
  - vertices( ): Return an iteration of all the vertices of the graph.
  - edge count(): Return the number of edges of the graph.
  - edges( ): Return an iteration of all the edges of the graph.
  - get edge(u,v): Return the edge from vertex u to vertex v, if one exists;
  - degree(v, out=True): For an undirected graph, return the number of edges incident to vertex v. For a directed graph, return the number of outgoing (resp. incoming) edges incident to vertex v, as designated by the optional parameter.

# Graph ADT

- incident edges(v, out=True): Return an iteration of all edges incident to vertex *v*. In the case of a directed graph, report outgoing edges by default; report incoming edges if the optional parameter is set to False.
- insert vertex(x=None): Create and return a new Vertex storing element x.
- insert edge(u, v, x=None): Create and return a new Edge from vertex *u* to vertex *v*, storing element x (None by default).
- remove vertex(v): Remove vertex *v* and all its incident edges from the graph.
- remove edge(e): Remove edge *e* from the graph.

# Representations of graphs

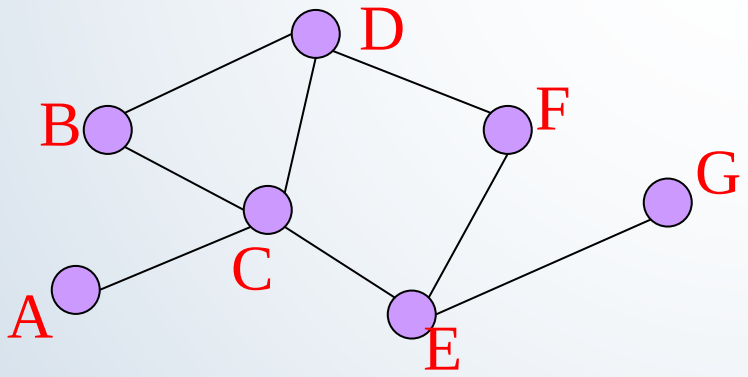
- 2 standard representations:
  - Adjacency Matrix
  - Adjacency List
- 3 more representations:
  - Edge list
  - Adjacency map
  - Incidence matrix

In each representation, we maintain a collection to store the vertices of a graph.
However, the four representations differ greatly in the way they organize the edges.

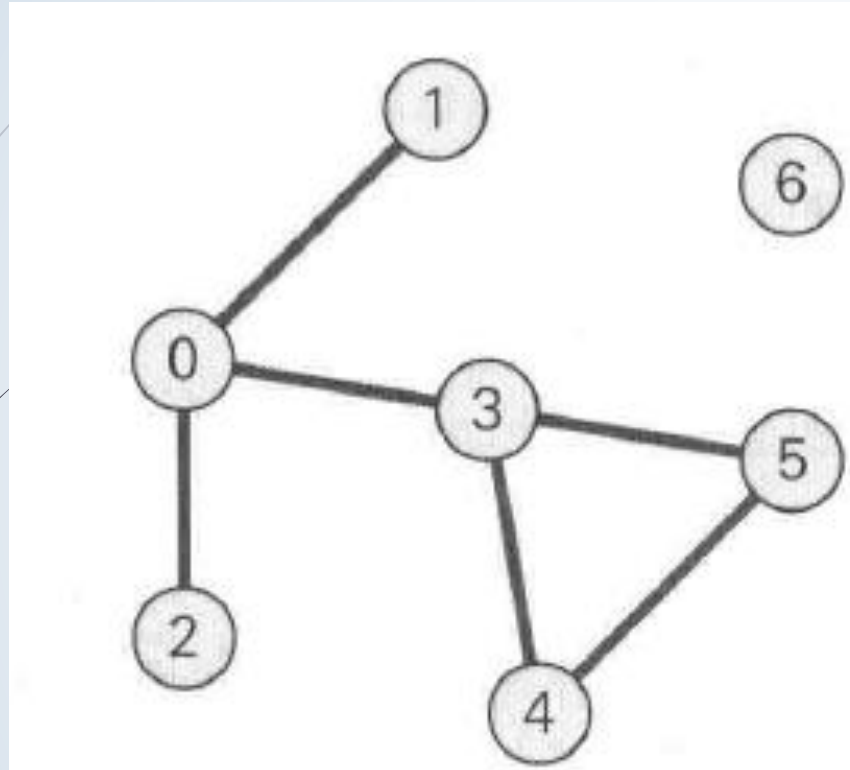
# Adjacency Matrix

- Use N*N 2D array to represent the weight (or T/F) of one vertex to another

An undirected graph




| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| A | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| B | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| C | 1 | 1 | 0 | 1 | 1 | 0 | 0 |
| D | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| E | 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| F | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| G | 0 | 0 | 0 | 0 | 1 | 0 | 0 |



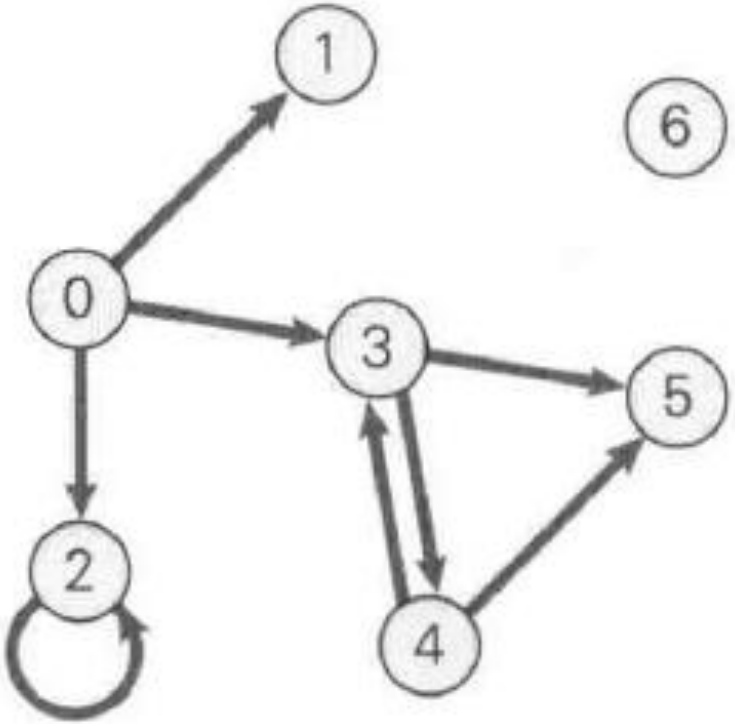

| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 3 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| | 4 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| | 5 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| | 6 | 0 | 0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

*j* (horizontal) *i* (vertical)

If M is the adjacency matrix, M[i][j]=M[j][i] for undirected graph.

M[i][j]=1 or true if there is an edge (i,j)
M[i][j]=0 or false if there is no edge between i and j.

# Adjacency Matrix for Directed Graph




*j*

| *i* | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| 4 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

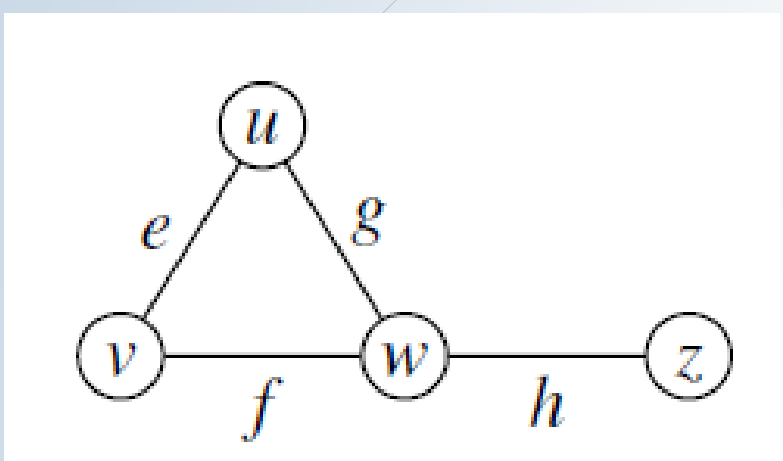
# Adjacency Matrix

- For undirected graph, only half of the array is used
- Fast query on edge weight and connection: O(1)
- Total memory used: $N^2$ *(what if the number of vertices = 10K ?)*
- Waste memory if the graph is sparse *i.e. #Edge is much smaller than half of (#Vertex)$^2$ a large proportion of the array will be zero*
- Slow when querying the list of neighboring vertices if the graph is sparse

# Adjacency List

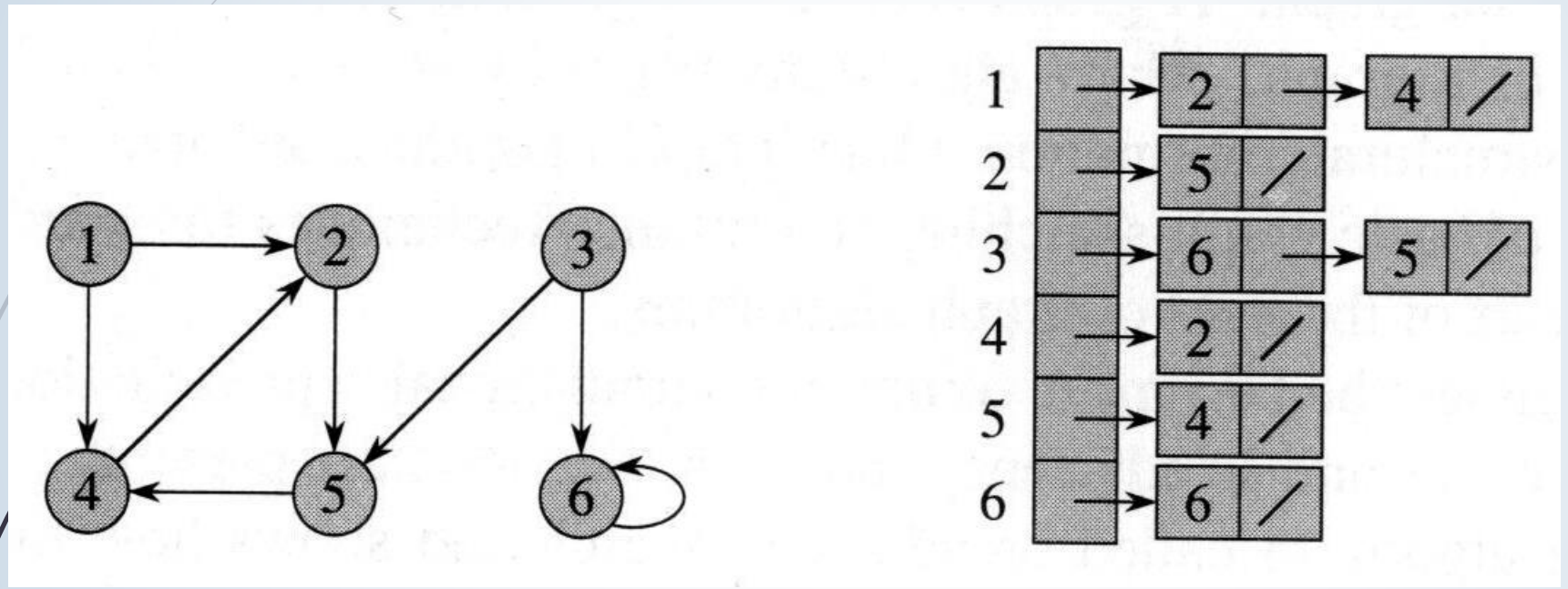


- The ***adjacency list*** structure groups the edges of a graph by storing them in smaller, secondary containers that are associated with each individual vertex.
- For each vertex $v$, we maintain a collection $I(v)$, called the ***incidence collection*** of $v$, whose entries are edges incident to $v$.
- In the case of a directed graph, outgoing and incoming edges can be respectively stored in two separate collections, $I_{out}(v)$ and $I_{in}(v)$.
- Traditionally, the incidence collection $I(v)$ for a vertex $v$ is a list, which is why we call this way of representing a graph the ***adjacency list*** structure.




| | | 0 | 1 | 2 | 3 |
|---|---|---|---|---|---|
| $u$ | 0 | | $e$ | $g$ | |
| $v$ | 1 | $e$ | | $f$ | |
| $w$ | 2 | $g$ | $f$ | | $h$ |
| $z$ | 3 | | | $h$ | |

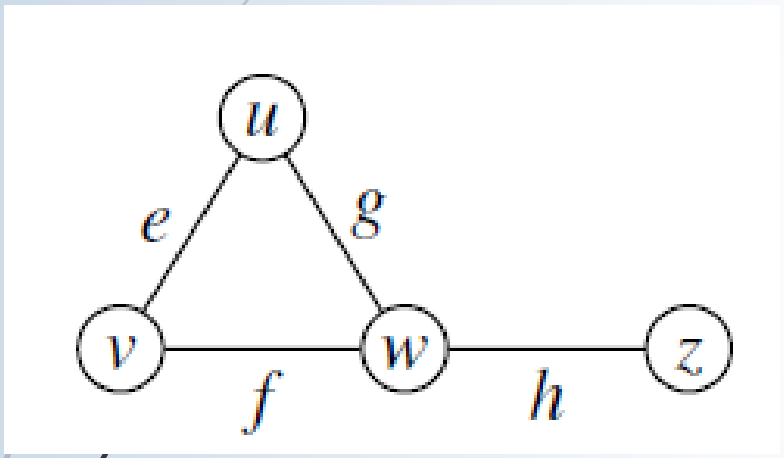
# Adjacency List

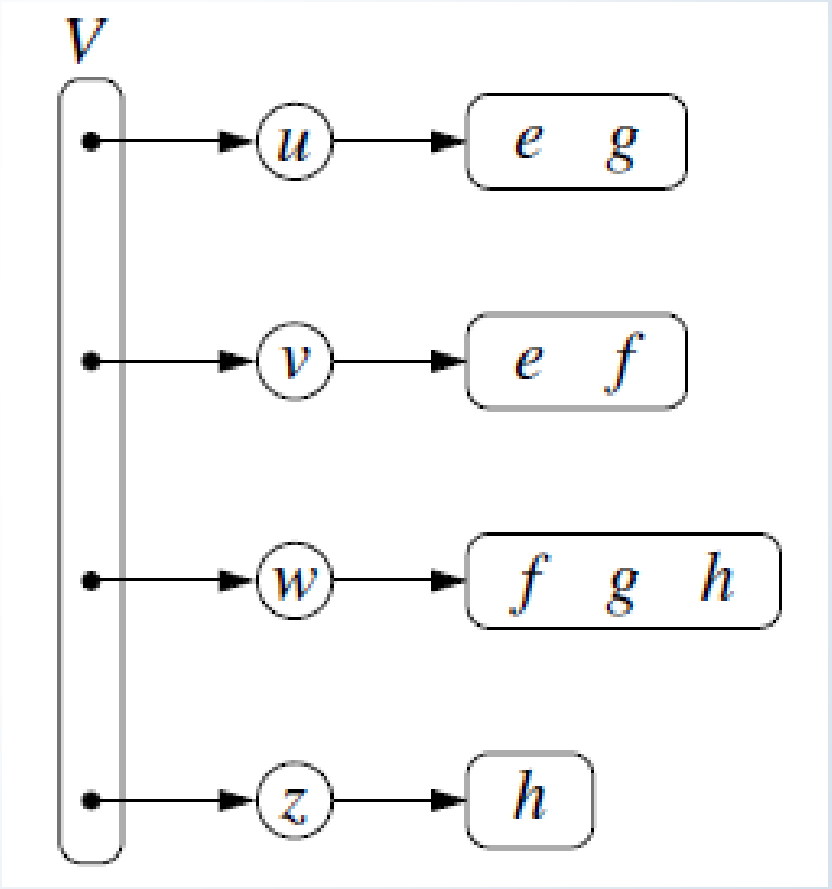



Space: O(V+E)

For each vertex v, store u's neighbors; if in directed graph, just outgoing edges.

# Adjacency List



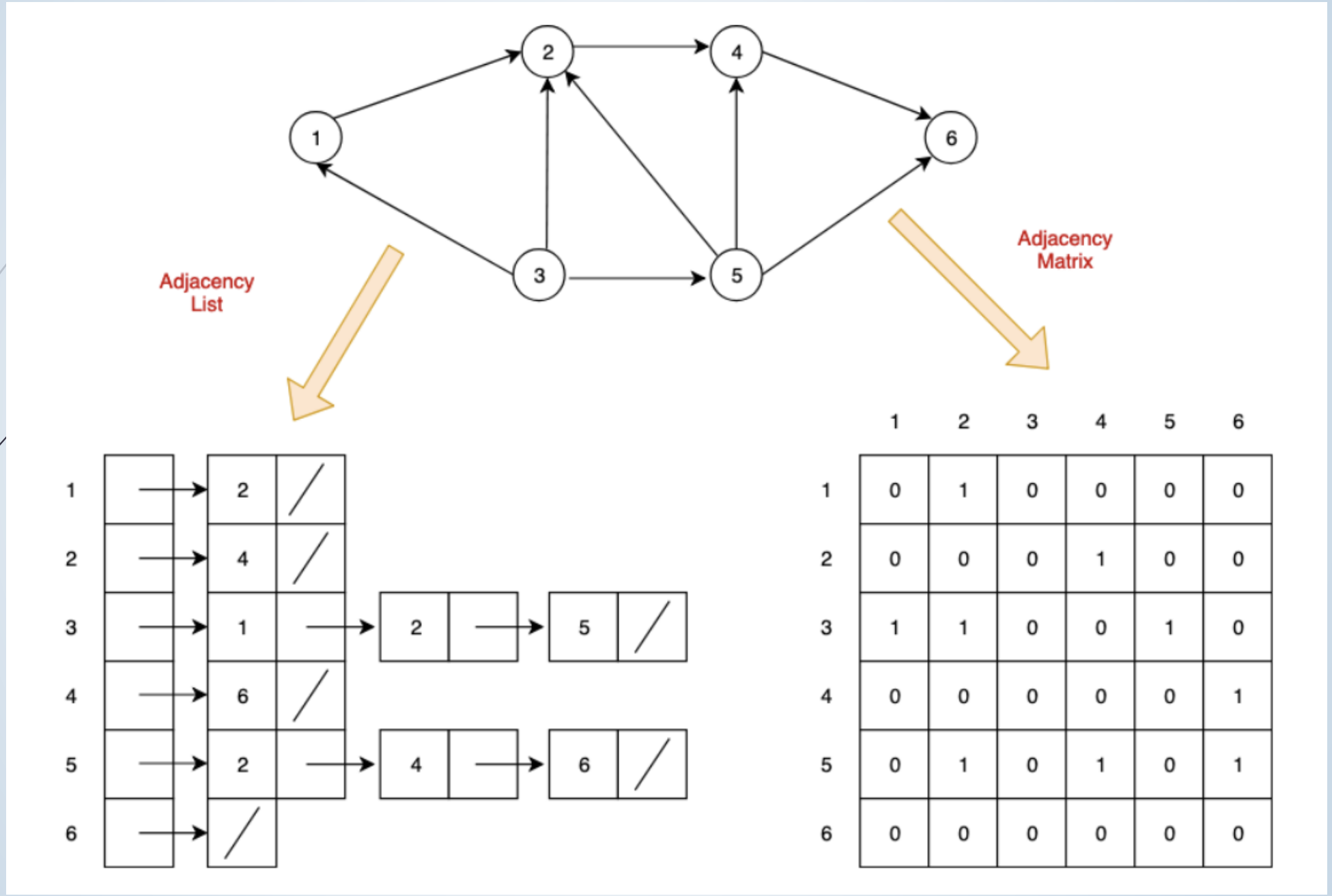



# Adjacency List

- Use link list *(or equivalent)* to store the list of neighboring vertex (or edge).
- Save memory if the graph is sparse:  O(V+E)
- Query on edge weight / connection can be slow
- Graph update is slow *(especially if one have to maintain order of neighbors)*
- Enumeration of all neighbors is fast.

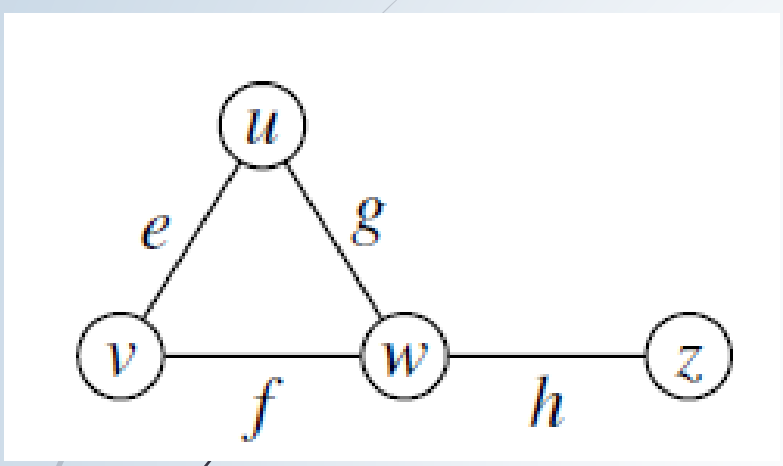
# Representations of graph

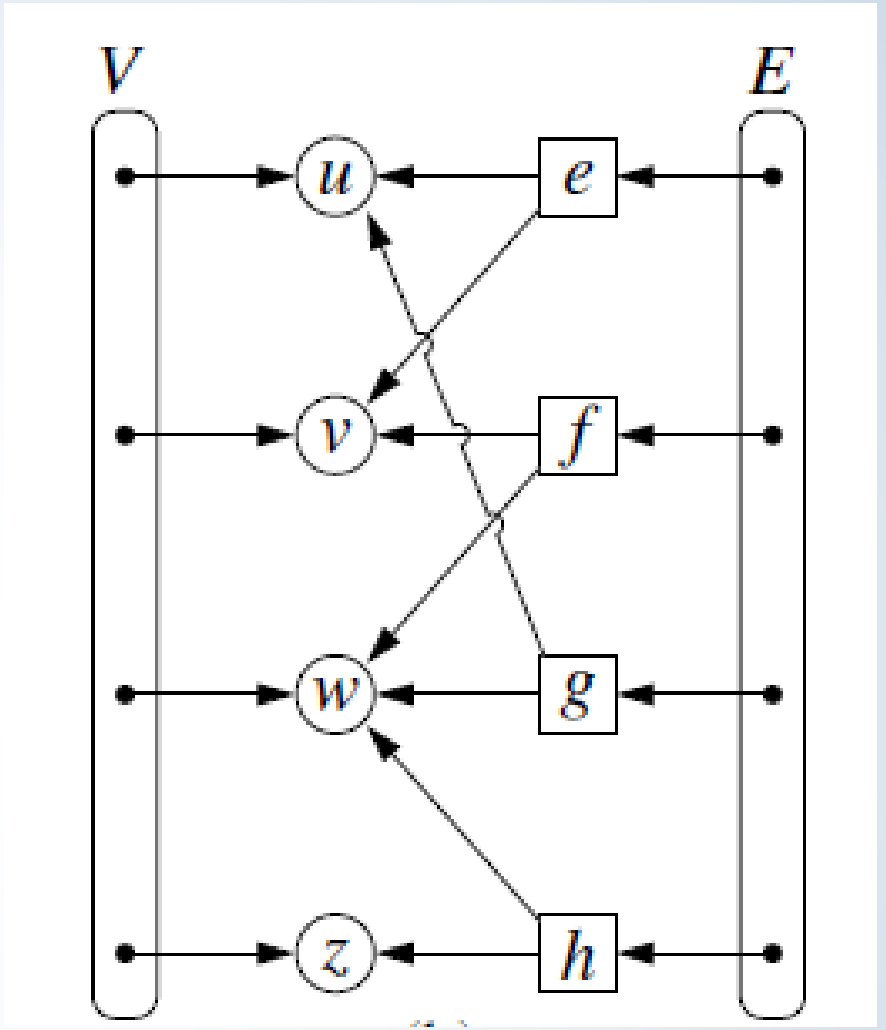
- If memory is sufficient and graph update is in-frequent, can represent the graph in both methods at the same time…
  - *If you need fast enumeration of neighbors together with fast query of weight/connection*
  - *e.g. List out all the neighbors of vertex A which do not connect to vertex B or vertex C…*
- Link-list can be replaced by 1D array with count *(enumeration of neighbor and query on degree will be fast)*


1
2
3
4
5
6
Adjacency List
Adjacency Matrix
1 2 3 4 5 6
1 0 1 0 0 0 0
2 0 0 0 1 0 0
3 1 1 0 0 1 0
4 0 0 0 0 0 1
5 0 1 0 1 0 1
6 0 0 0 0 0 0

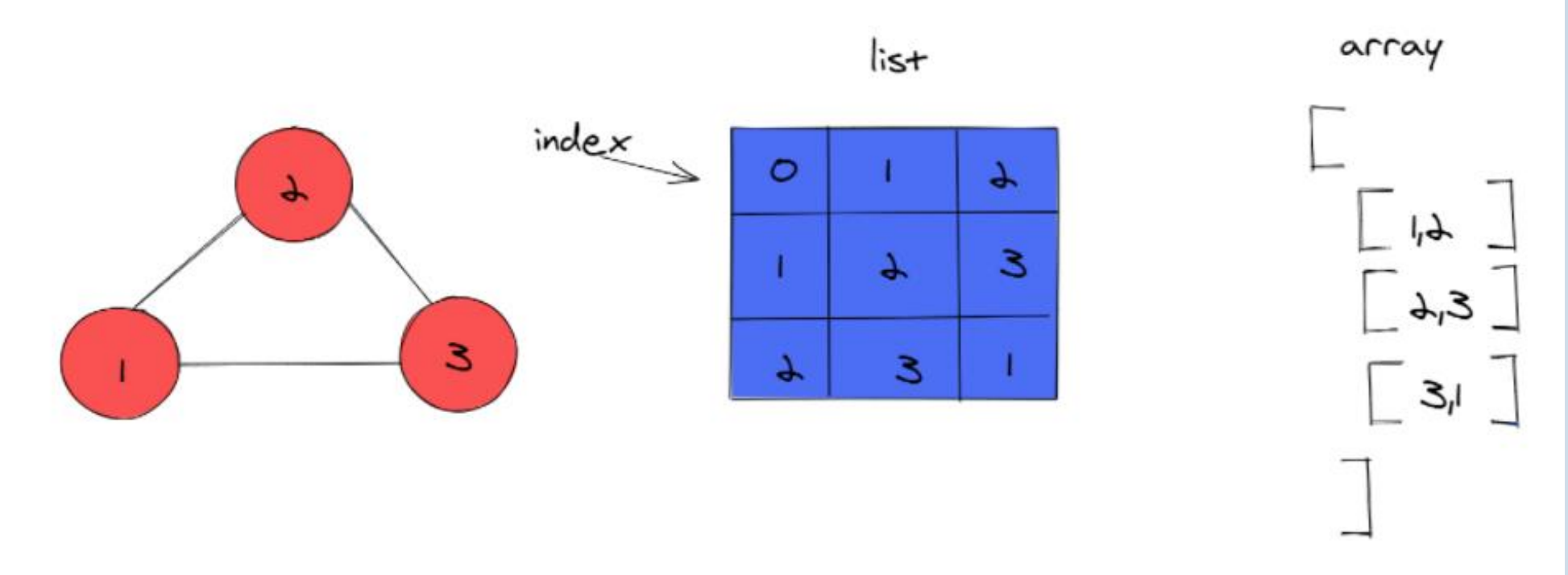
# Edge List

- All vertex objects are stored in an unordered list *V*, and all edge objects are stored in an unordered list *E*. Collections *V* and *E* are represented with doubly linked lists
- Performance:
  - Space: O(m+n)
  - Advantages: insert_edge, insert_vertex, remove_edge, vertex_count, edge_count O(1)
  - Limitation: degree(v), incident_edge(v), remove_vertex(v) O(m)




an edge object refers to the two vertex objects that correspond to its endpoints, but that vertices do not refer to incident edges.

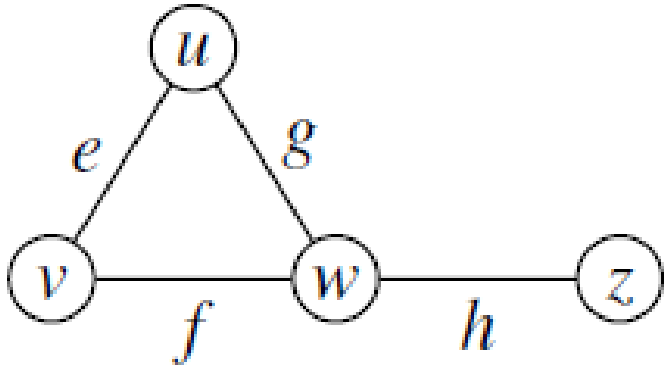



In the graph above, we have three nodes: `1`, `2`, and `3`. Each edge is given an index and represents a reference from one node to another.

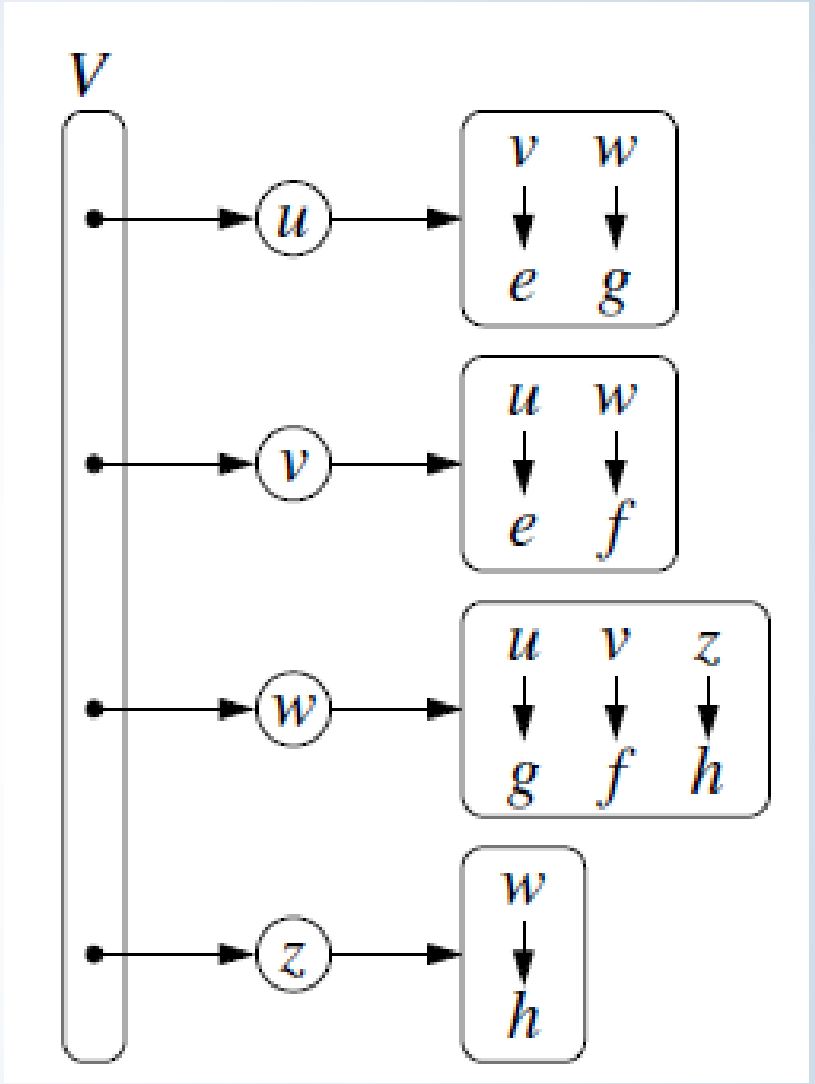
https://algodaily.com/lessons/implementing-graphs-edge-list-adjacency-list-adjacency-matrix

# Adjacency Map

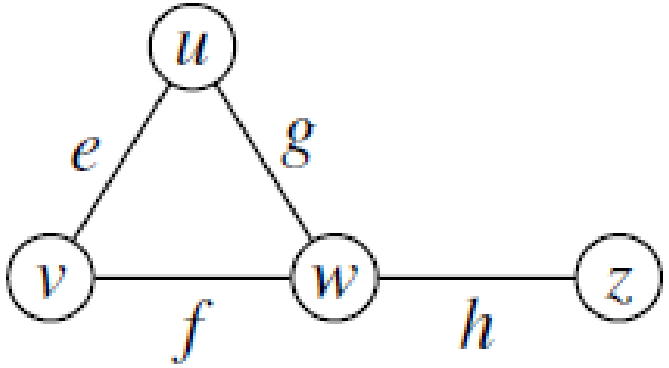
- Using adjacency list, it is not easy to check whether there is an edge from u to v because we have to search through either I(u) or I(v) (incidence collection of u or v).
- We can improve the performance by using a hash-based map to implement I(v) for each vertex v.




Get_edge(u,v) method can be implemented in ***expected*** $O(1)$ time by searching for vertex $u$ as a key in $I(v)$, or vice versa.




https://graphie.readthedocs.io/en/latest/Implementations/adjacencymap.html




# Incidence Matrix

edge

vertex

| 0 | e | g | f | h |
|---|---|---|---|---|
| U | 1 | 1 | | |
| V | 1 | | 1 | |
| W | | 1 | 1 | 1 |
| Z | | | | 1 |

# Graph Traversals

# Graph Searching

- To determine whether two vertices are connected (indirectly via some intermediate)
  - A is a relative of B, B is a relative of C, are A & C relative?
- To list out all members of a *connected-component*
  - List out all the direct/indirect family members of A…
- To find the shortest path (of un-weighted graph) from one vertex to another
  - Travel from Shatin to Central with minimum number of changes of transportation…
- TWO algorithms:
  - DFS (Depth First Search)
  - BFS (Breadth First Search)

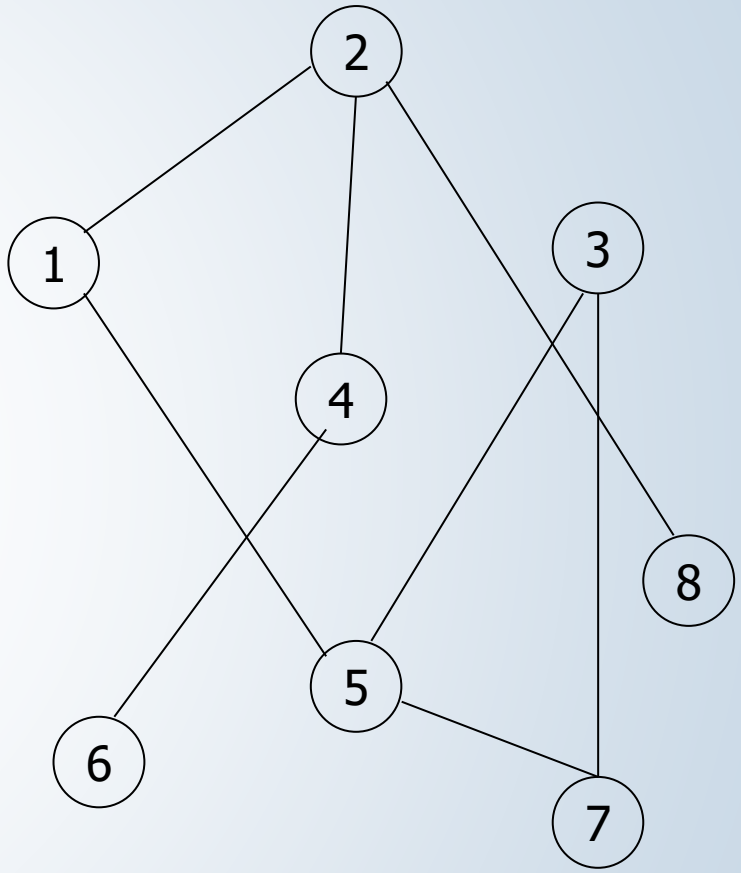
# DFS on Graphs



*Go as deep as you can*

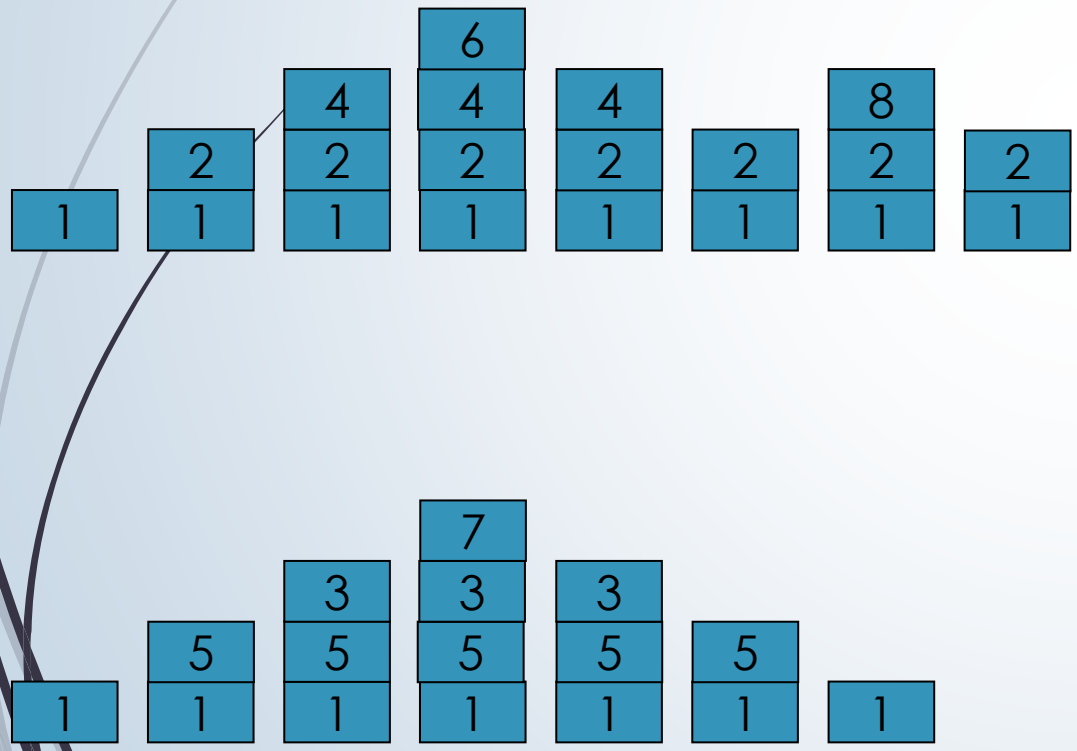
- Example DFS order (starting from 1):
  - 1,2,4,6,8,5,3,7
  - 1,5,7,3,2,8,4,6
- Using **Stack** to store nodes
  - Put the starting node into the stack
  - Repeat checking the top
- If top is unvisited
  - Print this element, mark as visited
- If top has unvisited neighbors
  - Push one of the neighbors on stack

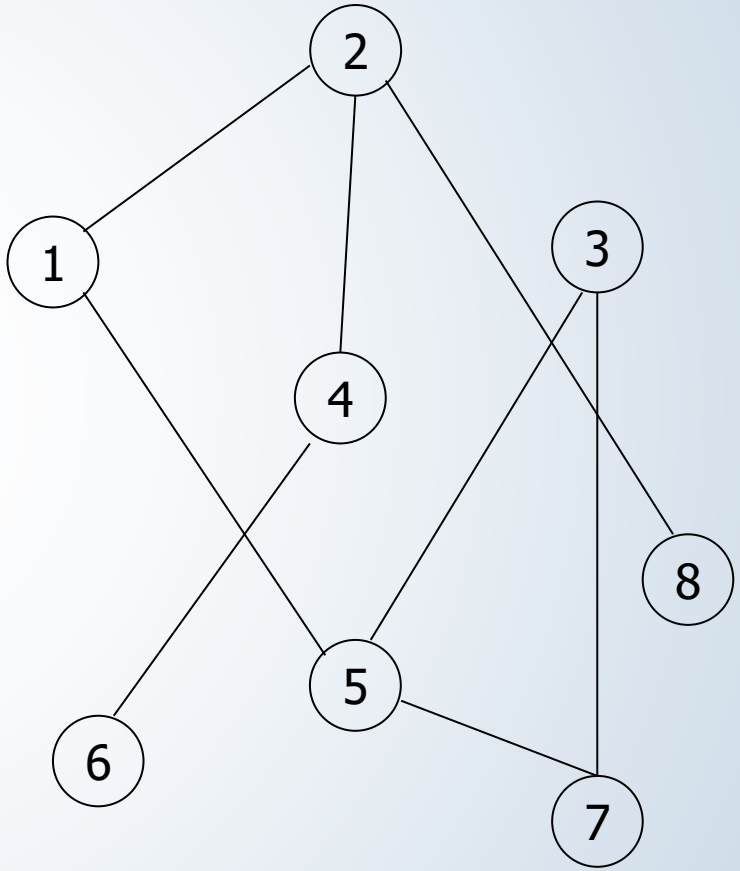
  else //top has no unvisited neighbors
  - Pop one element from the stack



# Example

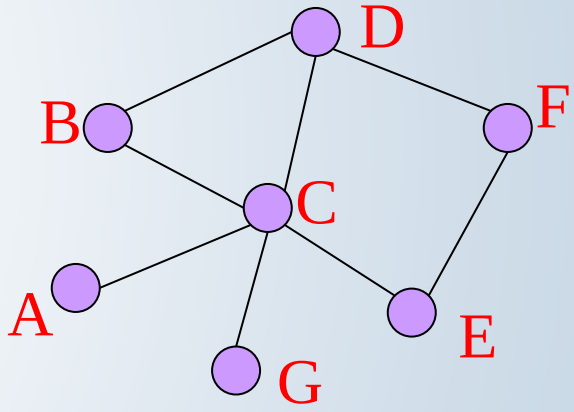
If Top has no unvisited neighbor, pop()



# Depth first search (DFS)

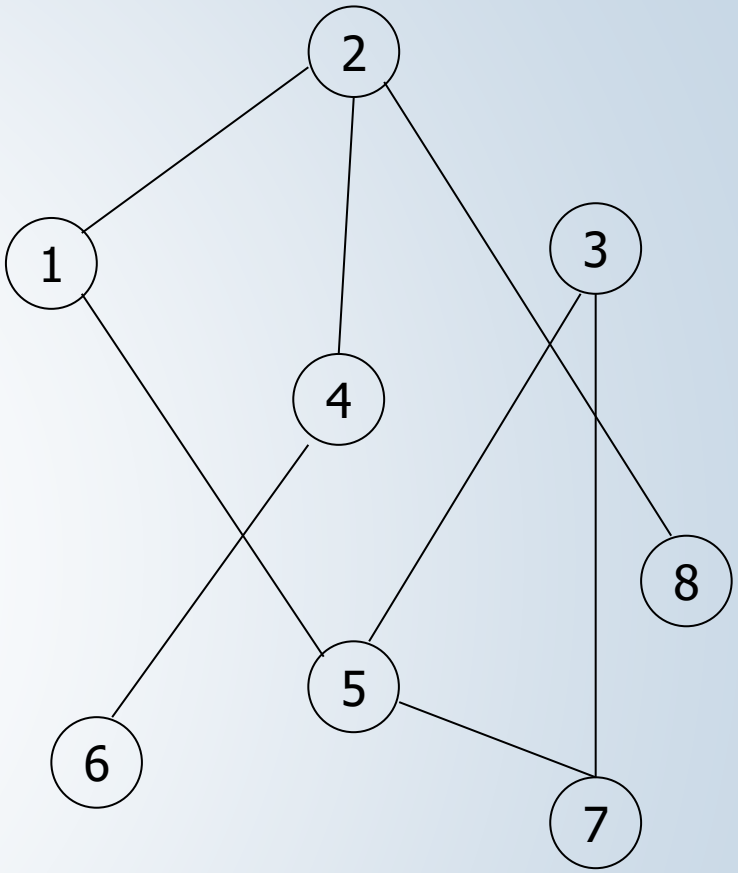
```
DFS (v) {                          //Starts with vertex v:
    visited[v] = true;
    for each vertex w adjacent to v {
        if (! visited[w])
            DFS (w);                        //Recursion
    }
}
```



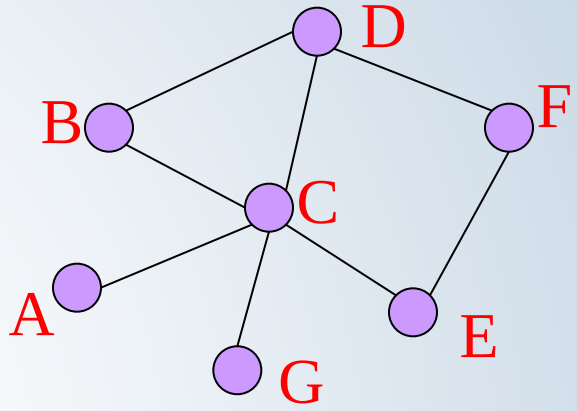

*DFS(A) = A, C, B, D, F, E, G*

# BFS on Graphs

- Go as broad as you can
- Example BFS order (starting from 1):
  - 1,2,5,4,8,3,7,6
  - 1,5,2,7,3,8,4,6
- Using **Queue** to store nodes
  - Put the starting node into queue
  - Repeat the following "Remove"
- Remove:
  - Remove a node from the queue
  - Print this element
  - Insert all his unvisited (haven't been in the queue) neighbors into the queue



```
BFS(v) {
     visited[v] = true;
    Enqueue(v);
    While queue not empty {
              x = Dequeue();
    for each vertex w adjacent to x {
          if (! visited[w]) {
        Enqueue (w);
        visited[w] = true;
            }
             }
    }
}
```




*BFS(A) = A, C, B, D, E, G, F*

# Example

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | | | | | | | |
| | 2 | 5 | | | | | |
| | | 5 | 4 | 8 | | | |
| | | | 4 | 8 | 3 | 7 | |
| | | | | 8 | 3 | 7 | 6 |
| | | | | | 3 | 7 | 6 |
| | | | | | | 7 | 6 |
| | | | | | | | 6 |
| | | | | | | | |