

# ALGORITHMS AND DATA STRUCTURES II

## Lecture 3

Graphs - Definitions and  
Representations,  
Graph traversal algorithms.

1/37

Lecturer: K. Markov  
[markov@u-aizu.ac.jp](mailto:markov@u-aizu.ac.jp)

# GRAPHS

- A graph  $G(V, E)$  consists of a set of **vertices (nodes)**  $V$  and a set of **edges**  $E$ .
- If we have a set of vertices:

$$V = \{v_1, v_2, \dots, v_m\}, \quad m = |V|$$

and a set of edges:

$$E = \{e_1, e_2, \dots, e_n\}, \quad n = |E|$$

our graph  $G(V, E)$  can also be denoted as:

$$G(V = \{v_1, v_2, \dots, v_m\}, E = \{e_1, e_2, \dots, e_n\})$$

or as:

$$G(\{v_1, v_2, \dots, v_m\}, \{e_1, e_2, \dots, e_n\})$$

# GRAPHS

- Each edge  $e_i$  is defined by a pair of vertices  $v_j$  and  $v_k$  so we can write  $e_i = (v_j, v_k)$ . Without subscripts, a pair of vertices is also denoted by  $(v, w)$ .
- If the edges are **ordered pairs**  $(v, w)$  of vertices then the graph is *directed* (edges are also called **arcs**).
- If the edges are **unordered pairs** (sets) of distinct vertices (also denoted by  $(v, w)$ ) then the graph is *undirected*.

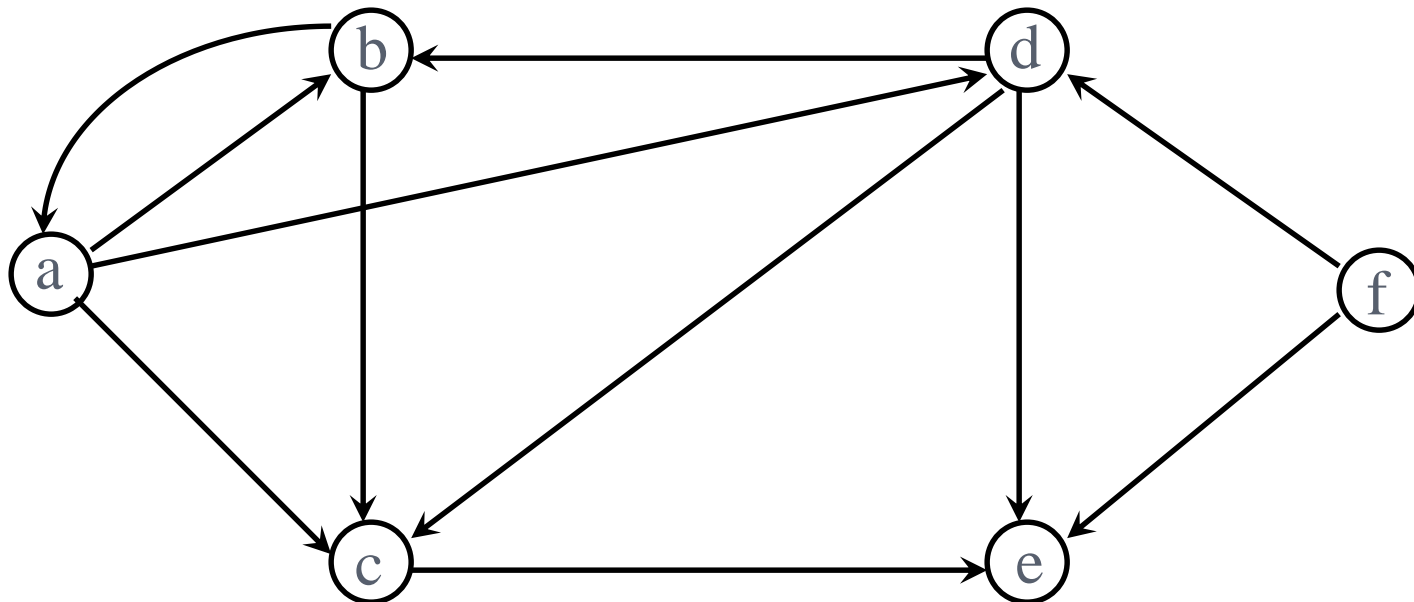
# GRAPHS

## ○ Example: A Directed Graph

$$G(V = \{a, b, c, d, e, f\},$$

$$E = \{(a, b), (a, c), (a, d), (b, a), (b, c), (c, e), \\ (d, b), (d, c), (d, e), (f, d), (f, e)\})$$

**Ordered pairs:** e.g. the two arcs connecting vertices  $a$  and  $b$  are denoted by  $(a, b)$  and  $(b, a)$ .



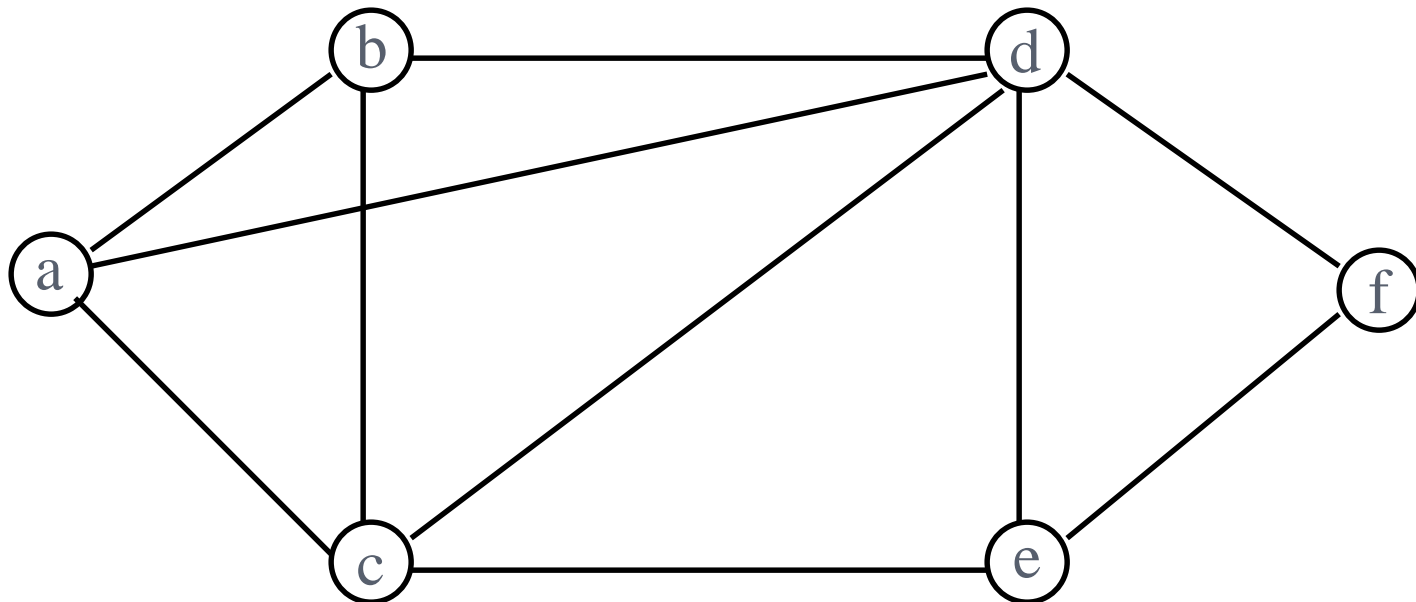
# GRAPHS

## ○ Example: An Undirected Graph

$$G(V = \{a, b, c, d, e, f\},$$

$$E = \{(a, b), (a, c), (a, d), (b, c), (b, d), \\ (c, d), (c, e), (d, e), (d, f), (e, f)\})$$

**Unordered pairs:** every edge is listed only once, e.g.  $(a, b)$  but **NOT**  $(b, a)$ .



# GRAPHS

## ○ Real life examples.

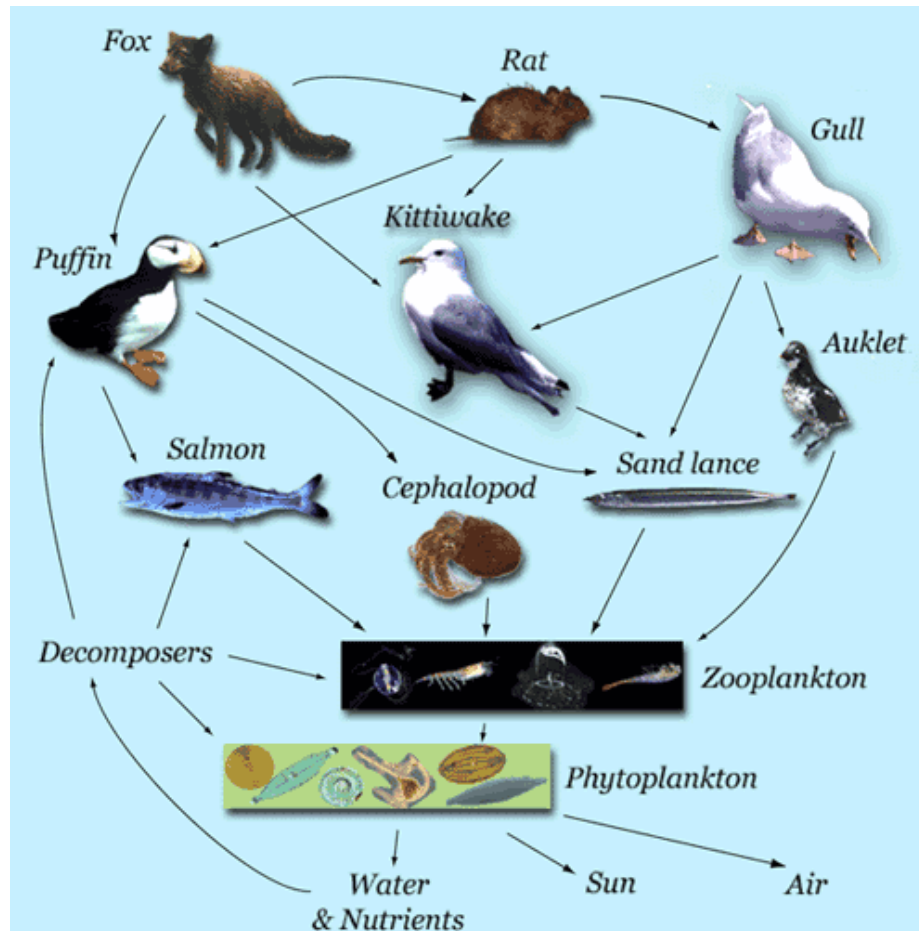
Graph	Vertices	Edges
transportation	street intersections	streets
communication	computers	cables
World Wide Web	web pages	hyperlinks
social	people	relationships
food web	species	who eats who
software system	functions	function calls
circuits	gates	wires
scheduling	tasks	precedence constraints

- A **web graph**: Node = web site, Edge = link.



# GRAPHS

- A **food graph**: Node = species, Edge = from predator to prey.

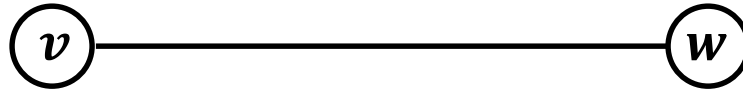




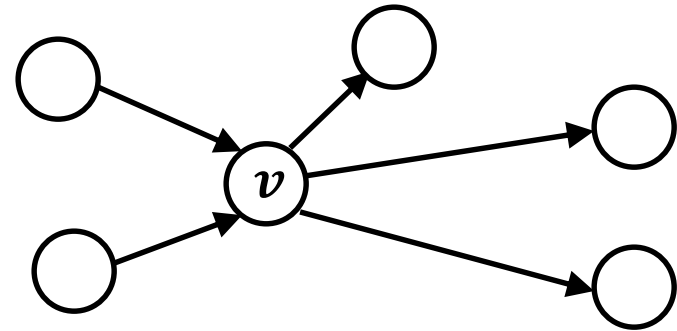
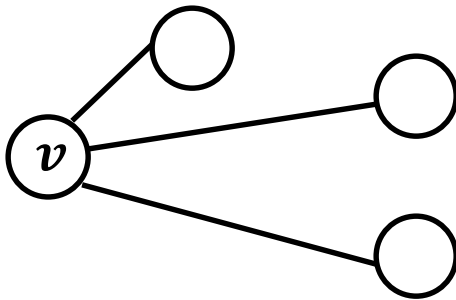
# GRAPHS

## Some definitions:

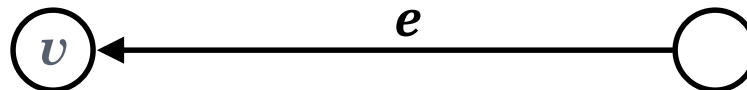
- Vertex  $v$  is **adjacent** to  $w$ , iff  $(v, w) \in E$ .



- Degree** ( $v$ ) = number of adjacent vertices.
  - In-degree, Out-degree.



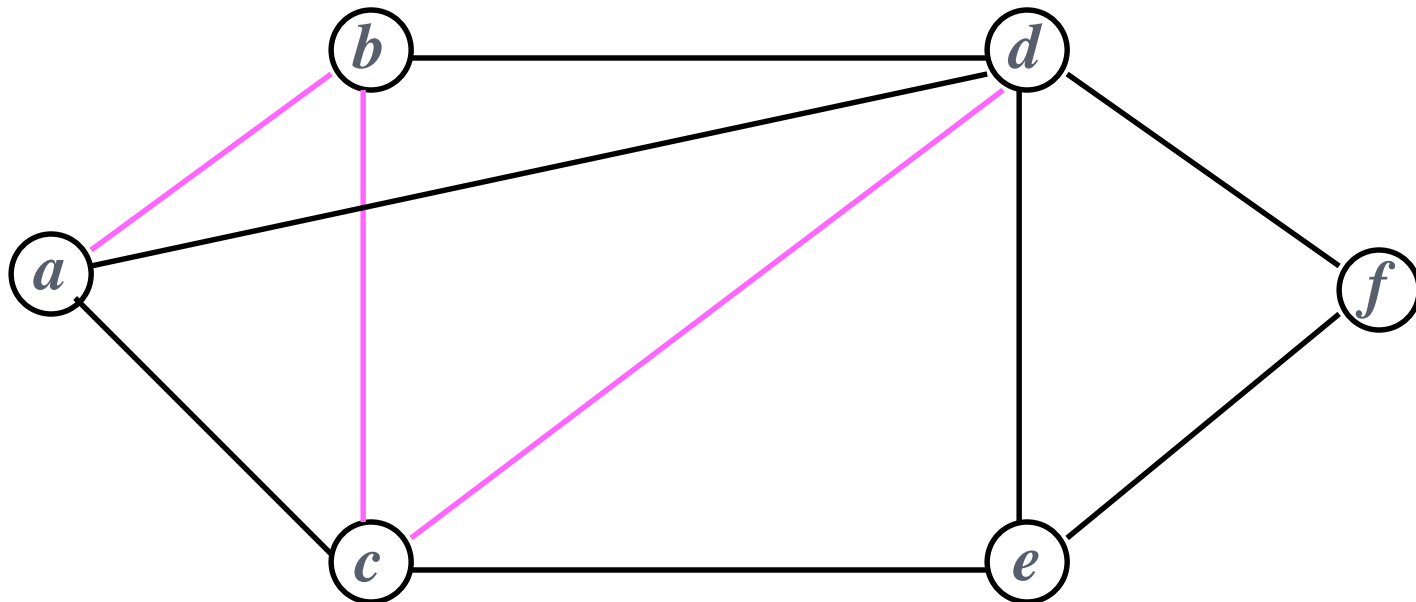
- Vertex  $v$  is **incident** to  $e$ , if  $v$  is the end vertex of  $e$ .



# GRAPHS

## Some definitions:

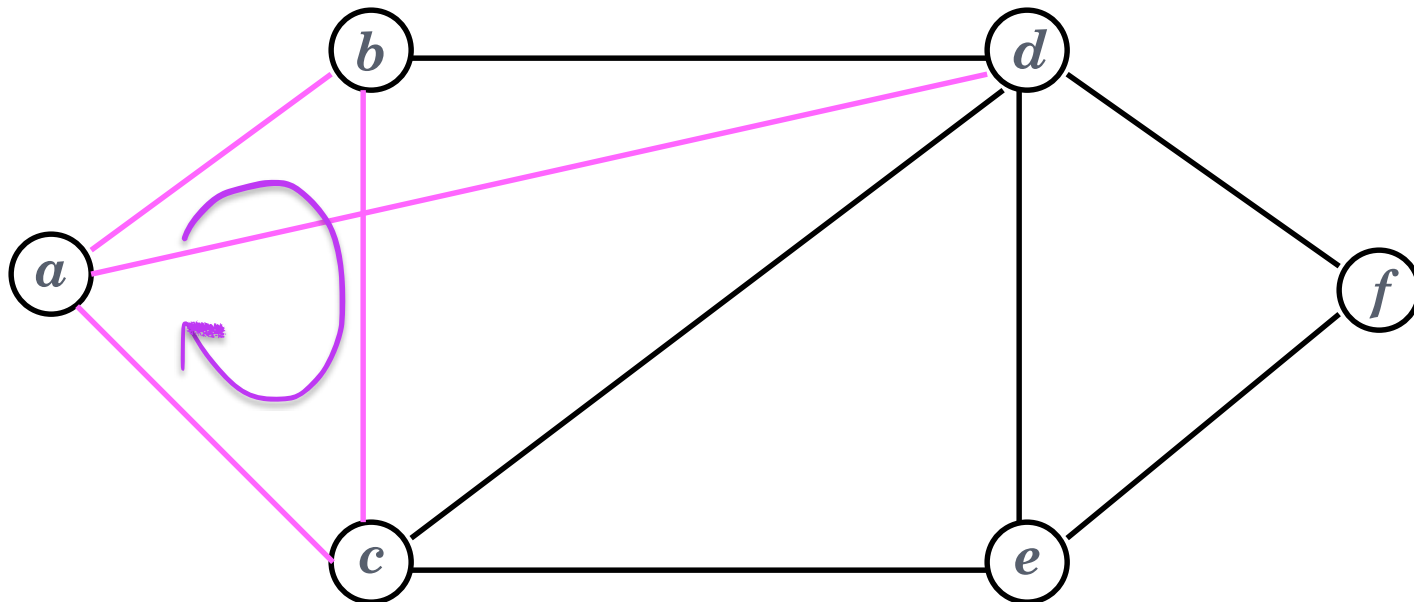
- A **path** in a graph is a sequence of edges of the form  $(v_1, v_2), (v_2, v_3), \dots, (v_{n-1}, v_n)$ . The path from  $v_1$  to  $v_n$  is of length  $n - 1$ .
  - As a *special case*, a single vertex denotes a path of length 0 from itself to itself.
- **Example** path:  $(a, b), (b, c), (c, d)$



# GRAPHS

## ○ Some definitions:

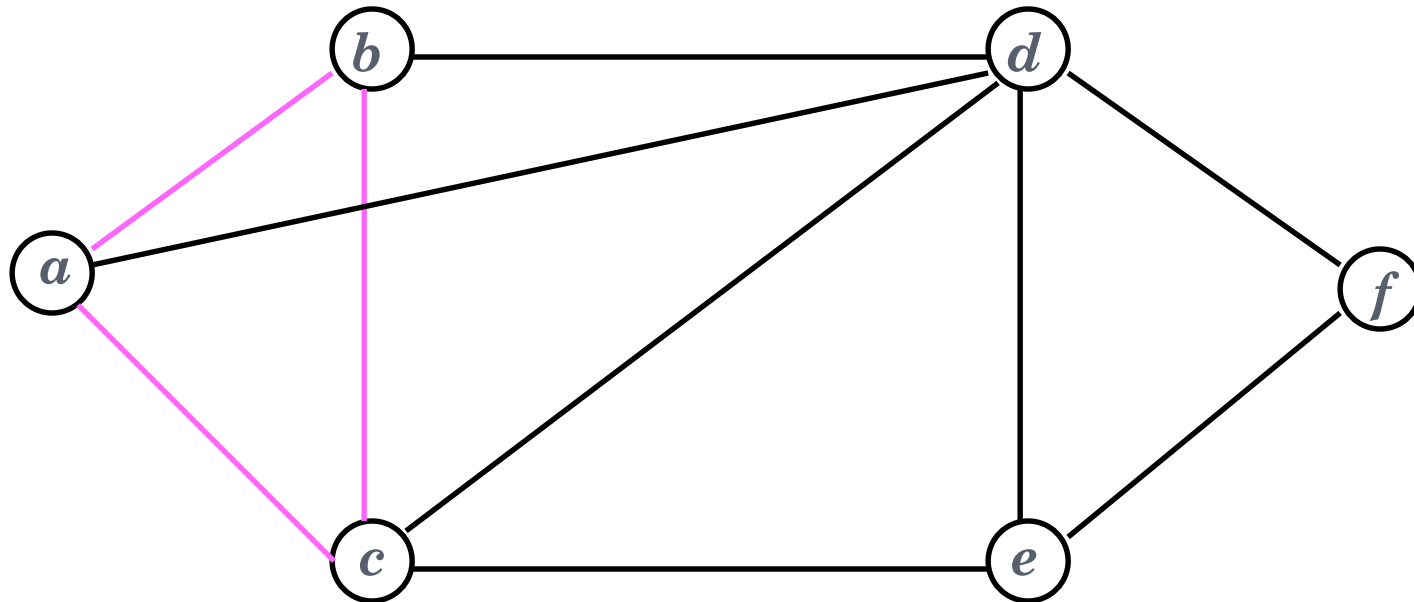
- A path is **simple** if all edges and all vertices on the path, except possibly the first and the last vertices, are distinct.
- **Example:**  $(a, b), (b, c), (c, a)$  is a simple path, but  $(a, b), (b, c), (c, a), (a, d)$  is **NOT** because of vertex  $a$ .



# GRAPHS



## ○ Some definitions:

- A **cycle** is a simple path of length at least 1 which begins and ends at the same vertex. In an un-directed graph, a cycle must be of length at least 3.
- **Example:**  $(a, b), (b, c), (c, a)$  is a simple path, which begins and ends at the same vertex  $a$ , so this is a cycle.



# GRAPHS

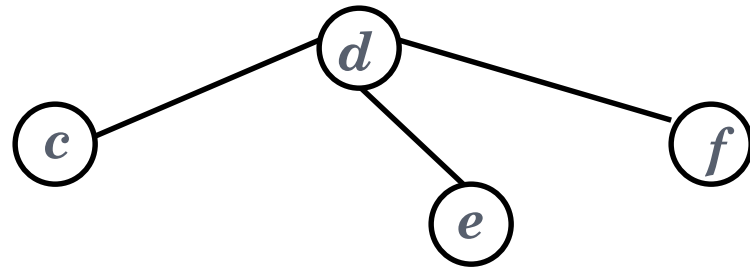
## ○ Some definitions:

- Number of edges (for an undirected graph) can range anywhere from 0 to  $V(V - 1)/2$ .
- Graphs with all edges present are called **complete graphs**.  
  
less than maximum
- Graphs with relatively few edges (less than  $V \log V$ ) are called **sparse graphs**.
- Graphs with most of the possible edges present are called **dense graphs**.

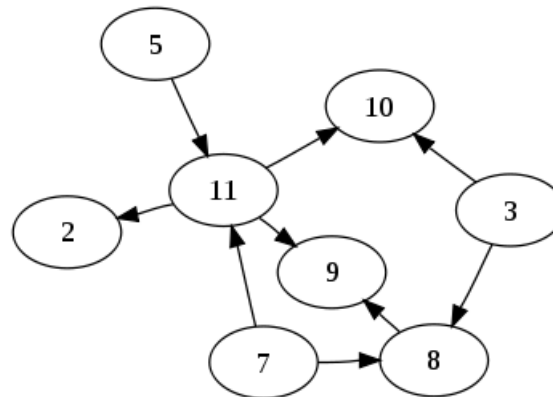
# GRAPHS

## ○ Some definitions:

- **Tree:** A connected graph with  $n$  vertices and  $n - 1$  edges.



- **Forest:** A collection of trees.
- **Directed Acyclic Graph (DAG):** A directed graph with no cycles.



# GRAPHS REPRESENTATION

- There are two popular computer representations of a graph. Both represent the *vertex set* and the *edge set*, but in different ways.

1. **Adjacency Matrix**

Use a 2D matrix to represent the graph.

2. **Adjacency List**

Use a 1D array of linked lists.

# GRAPHS REPRESENTATION

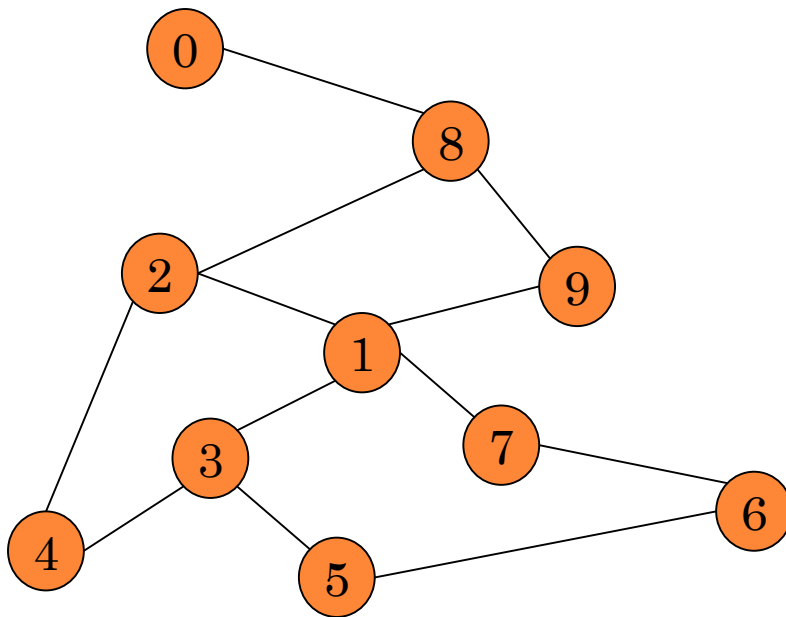
## ○ Adjacency Matrix

- 2D array  $A[0 \dots n - 1, 0 \dots n - 1]$ , where  $n$  is the number of vertices in the graph.
- Each row and column is indexed by the vertex id:  $a = 0, b = 1, c = 2, d = 3, c = 4$ .
- $A[i][j] = 1$  if there is an edge connecting vertices  $i$  and  $j$ ; otherwise,  $A[i][j] = 0$ .
- The storage requirement is  $\Theta(n^2)$ . An adjacency matrix is an appropriate representation if the graph is dense:  $|E| = \Theta(|V|^2)$



# GRAPHS REPRESENTATION

## Adjacency Matrix



	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	1	0
1	0	0	1	1	0	0	0	1	0	1
2	0	1	0	0	1	0	0	0	1	0
3	0	1	0	0	1	1	0	0	0	0
4	0	0	1	1	0	0	0	0	0	0
5	0	0	0	1	0	0	1	0	0	0
6	0	0	0	0	0	1	0	1	0	0
7	0	1	0	0	0	0	1	0	0	0
8	1	0	1	0	0	0	0	0	0	1
9	0	1	0	0	0	0	0	0	1	0

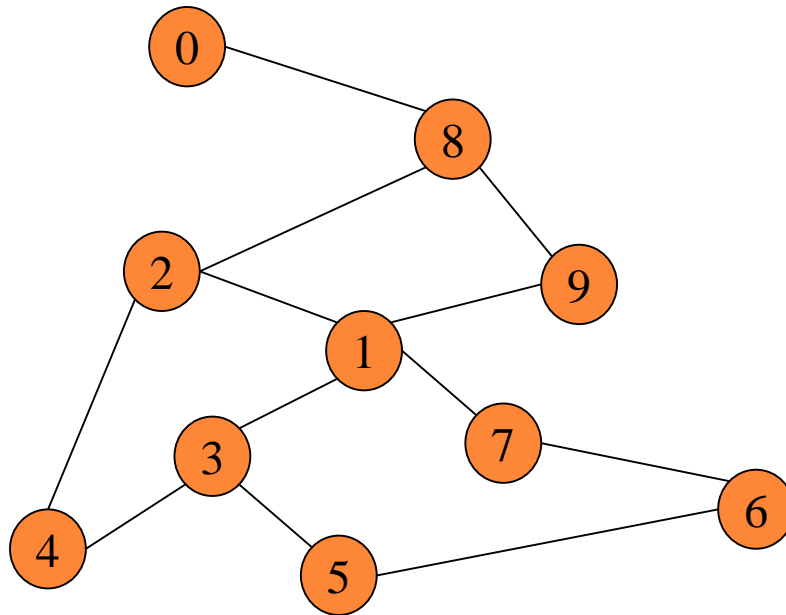
# GRAPHS REPRESENTATION

## ○ Adjacency List.

- If the graph is not dense, or in other words, **sparse**, a better solution is an adjacency list.
- The adjacency list is an array  $A[0 \dots n - 1]$  of lists, where  $n$  is the number of vertices in the graph.
- Each array entry is indexed by the vertex id.
- Each list  $A[i]$  stores the ids of the vertices adjacent to vertex  $i$ .
- The adjacency list has  $\Theta(n + m)$  space complexity.

# GRAPHS REPRESENTATION

## Adjacency List



0	→	8
1	→	2 3 7 9
2	→	1 4 8
3	→	1 4 5
4	→	2 3
5	→	3 6
6	→	5 7
7	→	1 6
8	→	0 2 9
9	→	1 8

# GRAPHS REPRESENTATION

## ○ Matrix vs. List representation.

### ● Adjacency Matrix.

- Always require  $n^2$  space.
  - This can waste a lot of space if the number of edges is small.
- Can quickly find if an edge exists.
- It's a matrix, some algorithms can use fast matrix computation!

### ● Adjacency List.

- More compact than adjacency matrices if graph has few edges.
- Requires more time to find if an edge exists.

# GRAPH TRAVERSAL

## ○ Graph questions:

- Given two nodes (vertices)  $v$  and  $w$ , is there a path between  $v$  and  $w$ ?
- Given two nodes (vertices)  $v$  and  $w$ , what is the length of the shortest path between  $v$  and  $w$ ?

## ○ Applications.

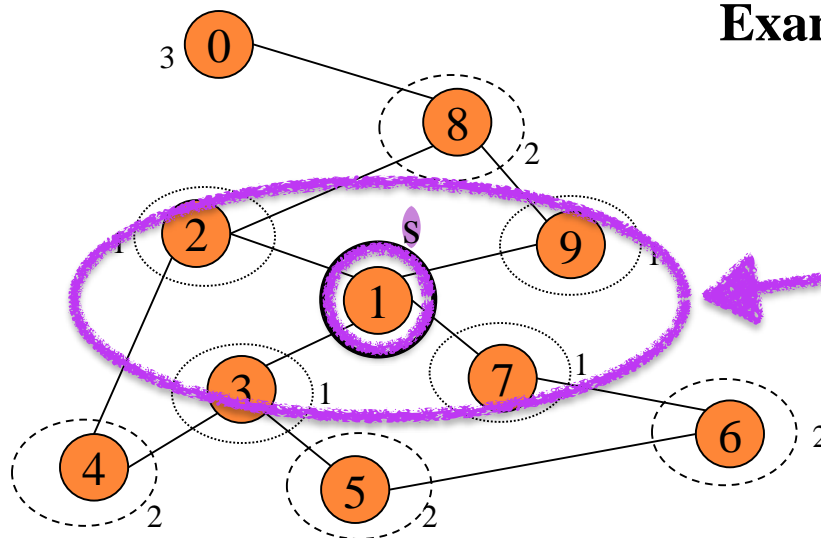
- Maze traversal.
- Fewest number of hops in a communication network.
- Navigation, etc.

# GRAPH TRAVERSAL

- **Two common graph traversal algorithms:**
  - **Breadth-First Search (BFS)**
    - Find the shortest paths in an un-weighted graph.
  - **Depth-First Search (DFS)**
    - Topological sort.
    - Find strongly connected components.

# BREADTH-FIRST SEARCH

- Given any source vertex  $s$ , BFS visits the other vertices at **increasing distances** away from  $s$ . In doing so, BFS discovers paths from  $s$  to other vertices.
- What do we mean by “**distance**”? The *number of edges on a path from  $s$* .



## Example

Consider  $s$ =vertex 1

Nodes at distance 1?

2, 3, 7, 9

Nodes at distance 2?

8, 6, 5, 4

Nodes at distance 3?

0

# BREADTH-FIRST SEARCH

- **BFS** algorithm implementation using **FIFO queue** and adjacency list.

- $G$  – graph
- $G.V$  – set of vertices.
- $G.Adj$  – adjacency list
- $s, u, v$  – vertices.
- $u.color$  – vertex color.
- $u.dist$  – vertex distance from root.
- $u.parent$  – parent vertex.
- $Q$  – FIFO queue.
- $Q.push(u)$  – inserts  $u$  into queue.
- $Q.pop()$  – retrieves next element.

initialized

start is gray

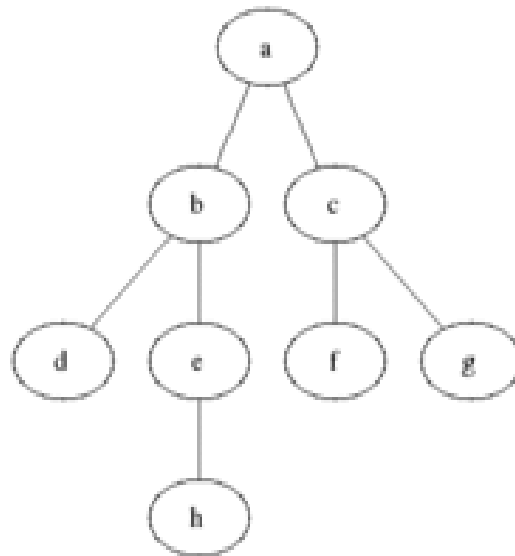
```
def BFS ( $G, s$ )  
    for each vertex  $u \in G.V$   
         $u.color = \text{WHITE}$   
         $u.dist = 0$   
         $u.parent = \text{None}$   
     $s.color = \text{GRAY}$   
     $Q = \emptyset$   
    Q.push( $s$ )  
    while  $Q \neq \emptyset$   
         $u = \text{Q.pop}()$   
        for each  $v \in G.Adj[u]$   
            if  $v.color == \text{WHITE}$   
                 $v.color = \text{GRAY}$   
                 $v.dist = u.dist + 1$   
                 $v.parent = u$   
                Q.push( $v$ )  
         $u.color = \text{BLACK}$ 
```



# BREADTH-FIRST SEARCH

## ○ BFS algorithm:

- Unvisited vertex -> WHITE color
- Vertex in the Queue -> GRAY color
- Visited vertex -> BLACK color



# BREADTH-FIRST SEARCH

## ○ BFS time complexity.

- Each vertex is enqueued once and dequeued once. These operations take  $O(1)$  time.
- Thus, all queue operations take  $O(n)$  time.
- Each vertex's adjacency list is scanned once.
- The sum of all of lengths of all adjacency lists is  $\Theta(m)$ .
- Total time complexity is  $O(n + m)$ !

# DEPTH-FIRST SEARCH

## ○ DFS algorithm.

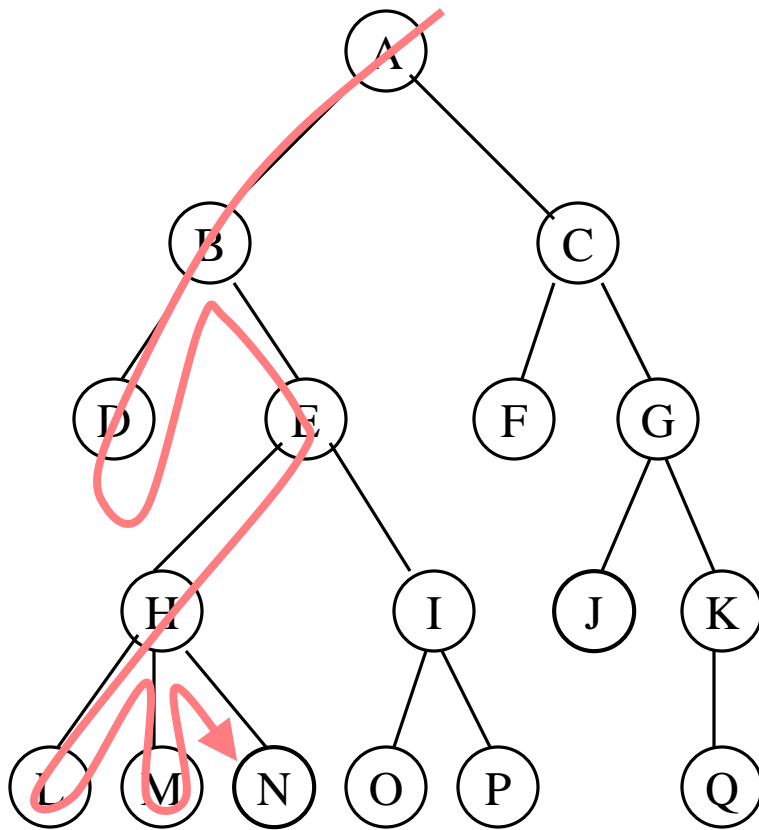
- DFS is another popular graph search strategy.
- It is a natural way to “visit” every vertex and check every edge in the graph systematically.
- Idea is similar to pre-order traversal (visit **children** first)
- It has applications for many graph problems, such as checking the connectivity, finding the connected components or cycles, and so on, in graphs.

# DEPTH-FIRST SEARCH

## ○ DFS algorithm.

- DFS will continue to visit neighbors in a **recursive** manner.
- Whenever we visit  $v$  from  $u$ , we recursively visit all **unvisited neighbors** of  $v$ . Then we backtrack (return) to  $u$ .
- Note: it is possible that  $v_j$  was unvisited when we recursively visit  $v_i$ , but became visited by the time we return from the recursive call.

# DEPTH-FIRST SEARCH



selecting the child till leaf => next child

- A **depth-first** search explores a path all the way to a leaf before **backtracking** and exploring another path.
- For example, after searching **A**, then **B**, then **D**, the search backtracks and tries another path from **B**.
- Nodes are explored in the order **A B D E H L M N I O P C F G J K Q**

# DEPTH-FIRST SEARCH

- DFS algorithm implementation using adjacency list.

```
def DFS (G)  
    // initialization  
    for each vertex  $u \in G.V$ :  
         $u.color = \text{WHITE}$   
     $time = 0$   
    // visit each node  
    for each vertex  $u \in G.V$ :  
        if  $u.color == \text{WHITE}$ :  
            DFS-VISIT (G, u)
```

```
def DFS-VISIT (G, u)  
     $time = time + 1$   
     $u.d = time$   
     $u.color = \text{GRAY}$   
    for each  $v \in G.Adj[u]$ :  
        if  $v.color == \text{WHITE}$ :  
            DFS-VISIT (G, v)  
     $u.color = \text{BLACK}$   
     $time = time + 1$   
     $u.f = time$ 
```

# DEPTH-FIRST SEARCH

- **DFS** algorithm explanation.
  - $G$  – graph,  $G.V$  – vertex set.
  - Unvisited vertex  $\rightarrow$  WHITE color.
  - Discovered vertex  $\rightarrow$  GRAY color.
    - Discovery time –  $v.d$
  - Finished vertex  $\rightarrow$  BLACK color.
    - Finished time –  $v.f$
  - $G.Adj$  – adjacency list of the graph.

# DEPTH-FIRST SEARCH

## ○ DFS time complexity.

- We never visit a vertex more than once.
- We examine all edges of the vertices.
- We know  $\sum_{vertex\ v} \textit{degree}(v) = 2m$  where  $m$  is the number of edges.
- So, the running time of DFS is proportional to the number of edges and number of vertices (same as BFS)  $O(n + m)$ .



# GRAPH CONNECTIVITY

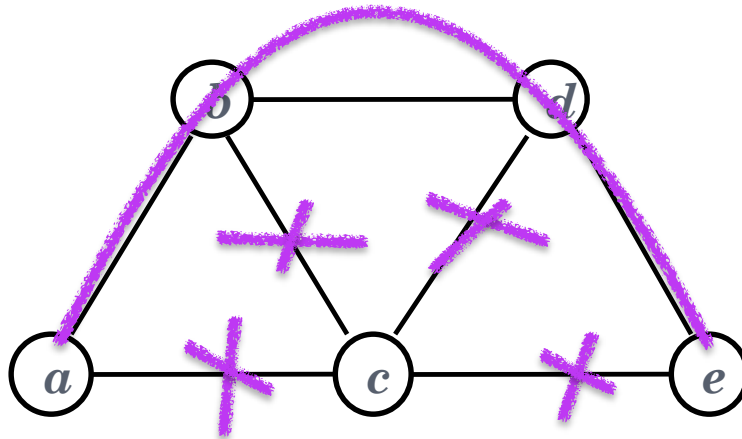
## ○ Connected graphs.

- An undirected graph  $G$  is called connected if there is a path between any pair of vertices in  $G$ .
- $G$  is called  **$k$ -connected** if the removal of any  $k - 1$  vertices leaves the remaining sub-graph connected.
- *1-connected* is connected;
- *2-connected* (bi-connected) means that one vertex failure can be tolerated.

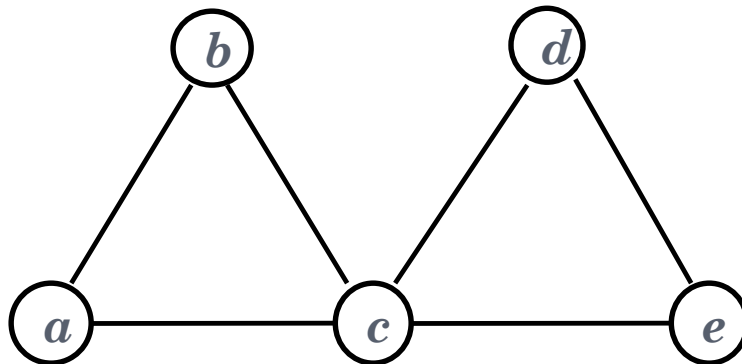
# GRAPH CONNECTIVITY

- **Bi-connected graphs.**

- Example:



Bi-connected graph.



**NOT** bi-connected graph.

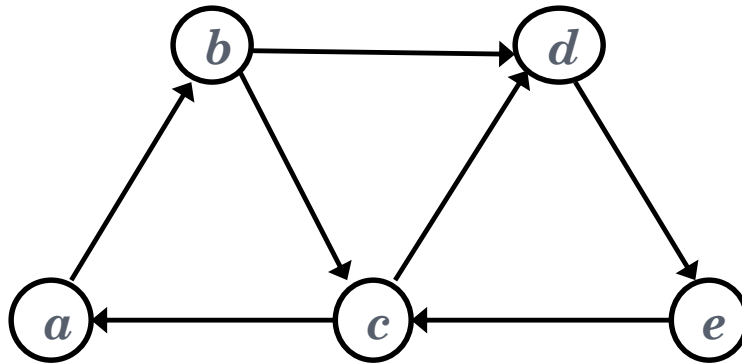
# GRAPH CONNECTIVITY

- **Strongly** connected graphs.
  - **Definition.** Node  $u$  and  $v$  are **mutually reachable** if there is a path from  $u$  to  $v$  and also a path from  $v$  to  $u$ .
  - **Definition.** A graph is **strongly connected** if every pair of nodes is mutually reachable.
  - **Lemma.** Let  $s$  be any node.  $G$  is strongly connected iff every node is reachable from  $s$ , and  $s$  is reachable from every node.
- The problem of finding the strongly connected components of  $G$  can be done by **DFS**.

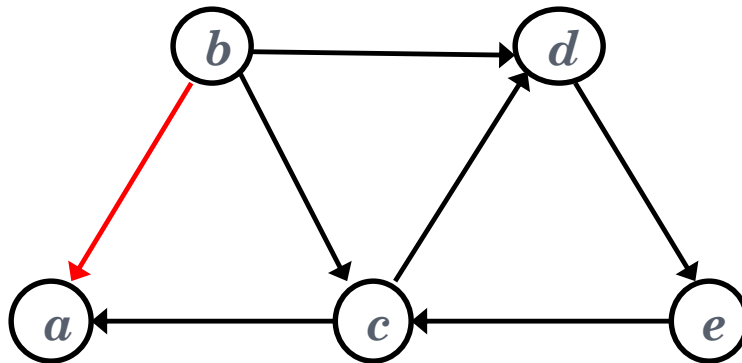
# GRAPH CONNECTIVITY

- Strongly connected graphs.

- Example:



Strongly connected.



**NOT** strongly connected.

**THAT'S ALL FOR TODAY!**