# Data structure and Algorithms

## Graph

Thanh-Hai Tran

Electronics and Computer Engineering
School of Electronics and Telecommunications

Hanoi University of Science and Technology
1 Dai Co Viet - Hanoi - Vietnam

# Outline

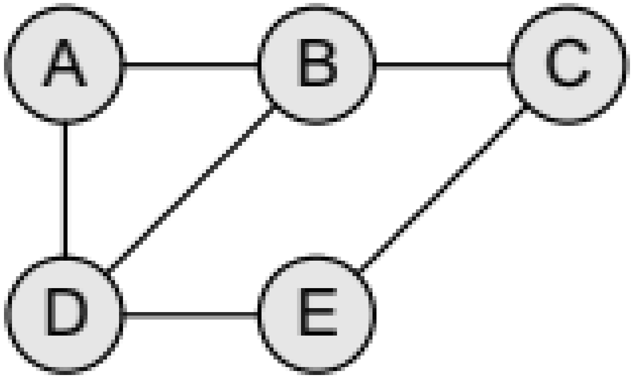
- **Basic concepts**
- **Graph representation**
  - Adjacency matrix
  - Adjacency list
- **Graph traversal  algorithm**
  - Breadth-First Search Algorithm
  - Depth-First Search Algorithm
- **References**

# Introduction

- A graph is an abstract data structure that is used to implement the mathematical concept of graphs.
- It is basically a collection of vertices (also called nodes) and edges that connect these vertices.
- A graph is often viewed as a generalization of the tree structure, where any kind of complex relationship can exist
- Why are Graphs Useful **?**
  - Graphs are widely used to model any situation where entities or things are related to each other in pairs
  - Examples:
    - Family trees
    - Transportation network

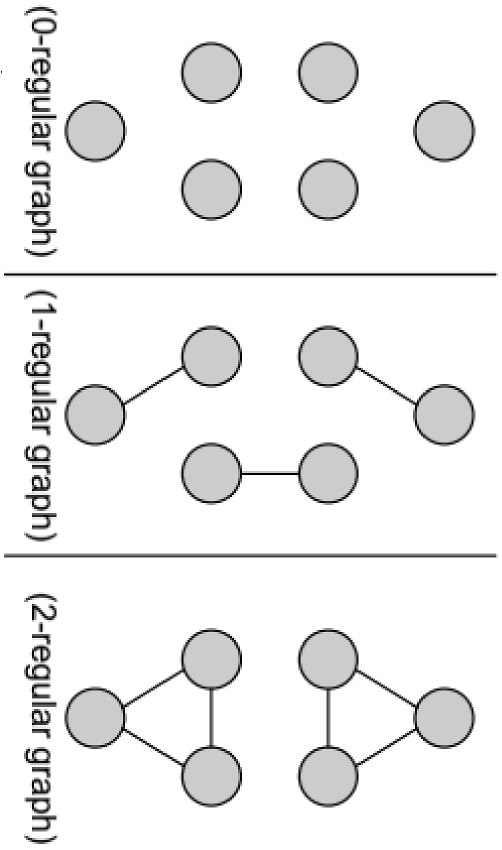
# Graph definition

- **Graph G is a ordered set (V, E), G = (V, E) where**
  - V(G) represents the set of vertices
  - E(G) represents the set of edges that connect these vertices
- **Example**
  - V(G) = {A, B, C, D, E} (5 vertices)
  - E(G) = {{A,B}, {B, C}, {C,E}, {D, E}, {A, D}, {D, B}} (6 edges)



# Graph terminology

- **Adjacent nodes or neighbours:**
  - For every edge e = (u, v) that connect nodes u and v
  - The nodes u and v are the end-points and said to be adjacent nodes / neighbor
- **Degree of a node:**
  - deg(u) is the total number of edges containing the node u
  - deg(u) = 0: u is an isolated node
- **Regular graph:** each node has the same degree
- **Size of a graph**: total number of edges



# Graph terminology

- **Path: P = {$v_0$, $v_1$, ..., $v_n$} of length n from node u to v is defined as:**
  - a sequence of n nodes (u = $v_0$, v = $v_n$) and
  - $v_i$ is adjacent to $v_i$ with i = 1,2,3,...
- **Close path:** path has the same end-points ($v_0 = v_n$)
- **Simple path:** all the nodes in the path are distinct with an exception that $v_0$ may be equal to $v_n$
- **Cycle:** A path in which the first and the last vertices are same

# Graph terminology

- **Connected graph:**
  - A graph is said to be connected if for any two vertices (u, v) in V there is a path from u to v.
  - There are no isolated nodes in a connected graph.
  - A connected graph that does not have any cycle is called a tree.

# Graph terminology

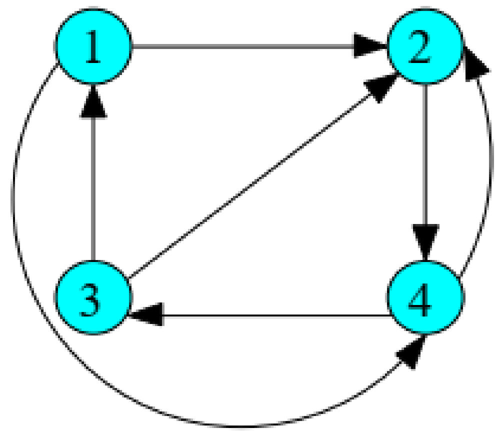
- **Complete graph:**
  - All its nodes are fully connected
  - There is a path from one node to every other node in the graph
  - A complete graph has n(n–1)/2 edges, where n is the number of nodes in G A complete graph has n(n–1)/2 edges,
    where n is the number of nodes in G

# Graph terminology

- **Weighted graph:**
  - A graph is said to be labelled if every edge in the graph is assigned some data.
  - In a weighted graph, the edges of the graph are assigned some weight or length.
  - The weight of an edge denoted by w(e) is a positive value which indicates the cost of traversing the edge.

# Graph terminology

- **Directed graph:** is a graph in which every edge has a direction assigned to it
- **Out-degree of a node:** outdeg(u) = number of edges that originates at u
- **In-degree of a node:** indeg(u) = number of edges that terminate at u



# Examples of graph

- **Social network (facebook) (directed or undirected)**

- **Problem: Suggest friends**

# Examples of graph

- **World wide web (directed graph)**

- **Problem: web-crawling  (graph traversal)**

# Examples of graph

- Intercity road network (weighted directed graph)

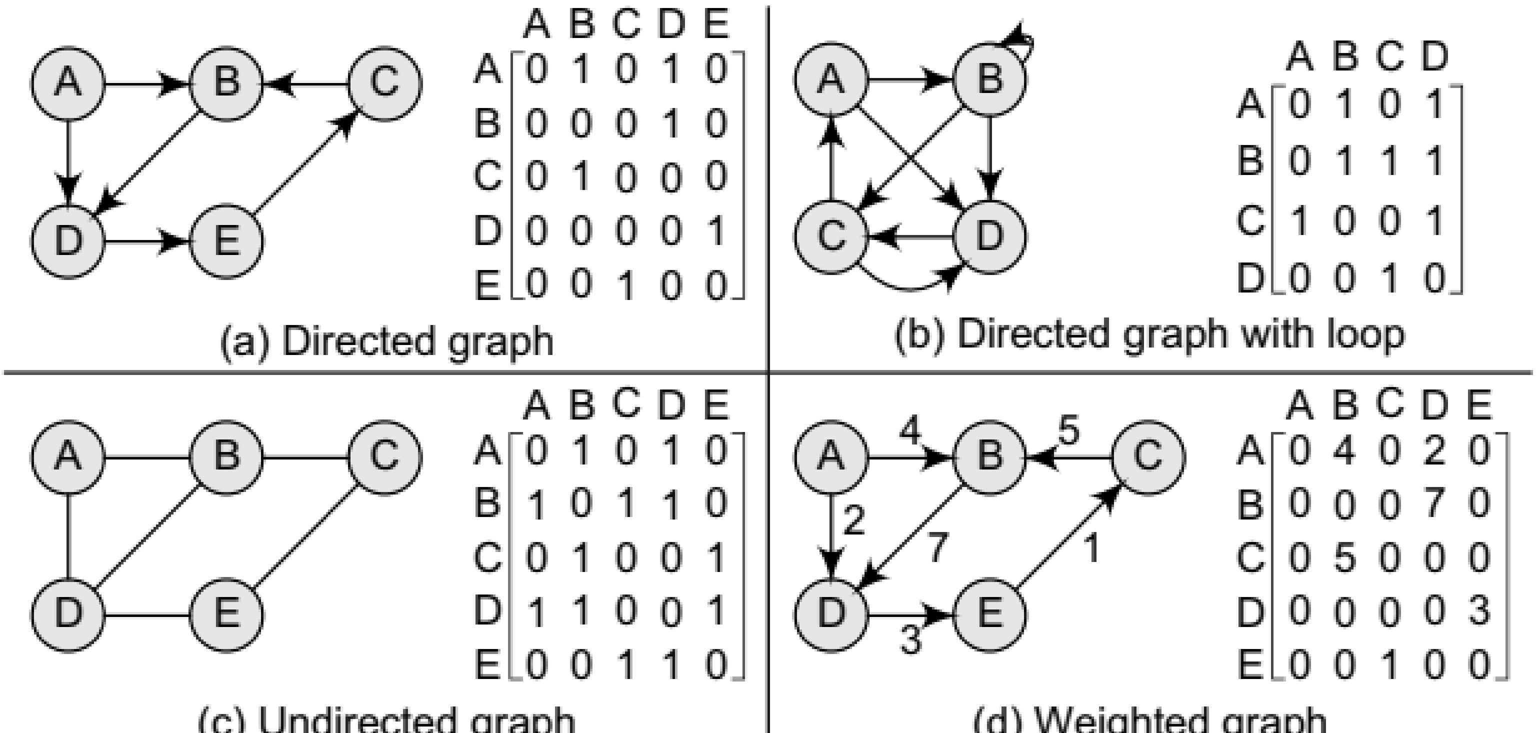
- Problem: shortest path from a city to another city

# Graph representation

- **Sequential representation** by using an adjacency matrix.
- **Linked representation** by using an adjacency list that stores the neighbors of a node using a linked list.
- **Adjacency multi-list** which is an extension of linked representation

# Adjacency matrix

$$a_{ij} = \begin{cases} 1 & \text{[if } v_i \text{ is adjacent to } v_j\text{, that is there is an edge } (v_i, v_j)\text{]A} \\ 0 & \text{[otherwise]} \end{cases}$$




(a) Directed graph

(b) Directed graph with loop

(c) Undirected graph

(d) Weighted graph

# Adjacency matrix

- For a simple graph (that has no loops), the adjacency matrix has 0s on the diagonal.
- The adjacency matrix of an undirected graph is symmetric.
- The memory use of an adjacency matrix is $O(n^2)$, where n is the number of nodes in the graph.
- Number of 1s (or non-zero entries) in an adjacency matrix is equal to the number of edges in the graph.
- The adjacency matrix for a weighted graph contains the weights of the edges connecting the nodes.

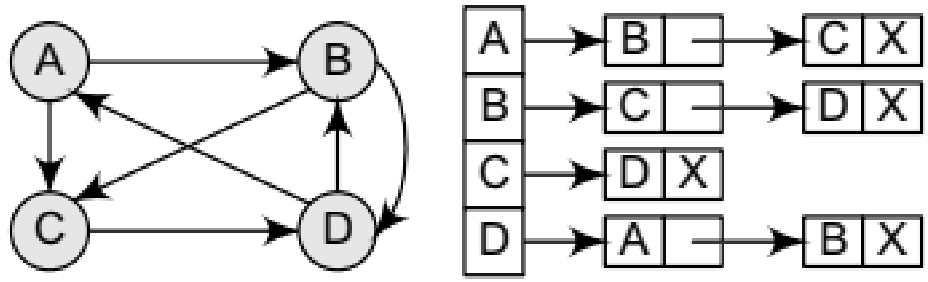
# Adjacency matrix

- **Operation:**
  - finding adjacent nodes
  - Complexity: O(n)
- **Operation:**
  - checking if two nodes are connected
  - Complexity O(1) (+ O(n) = O(n))
- **Memory space:**
  - $n^2$
  - If the graph is sparse => waste of memory

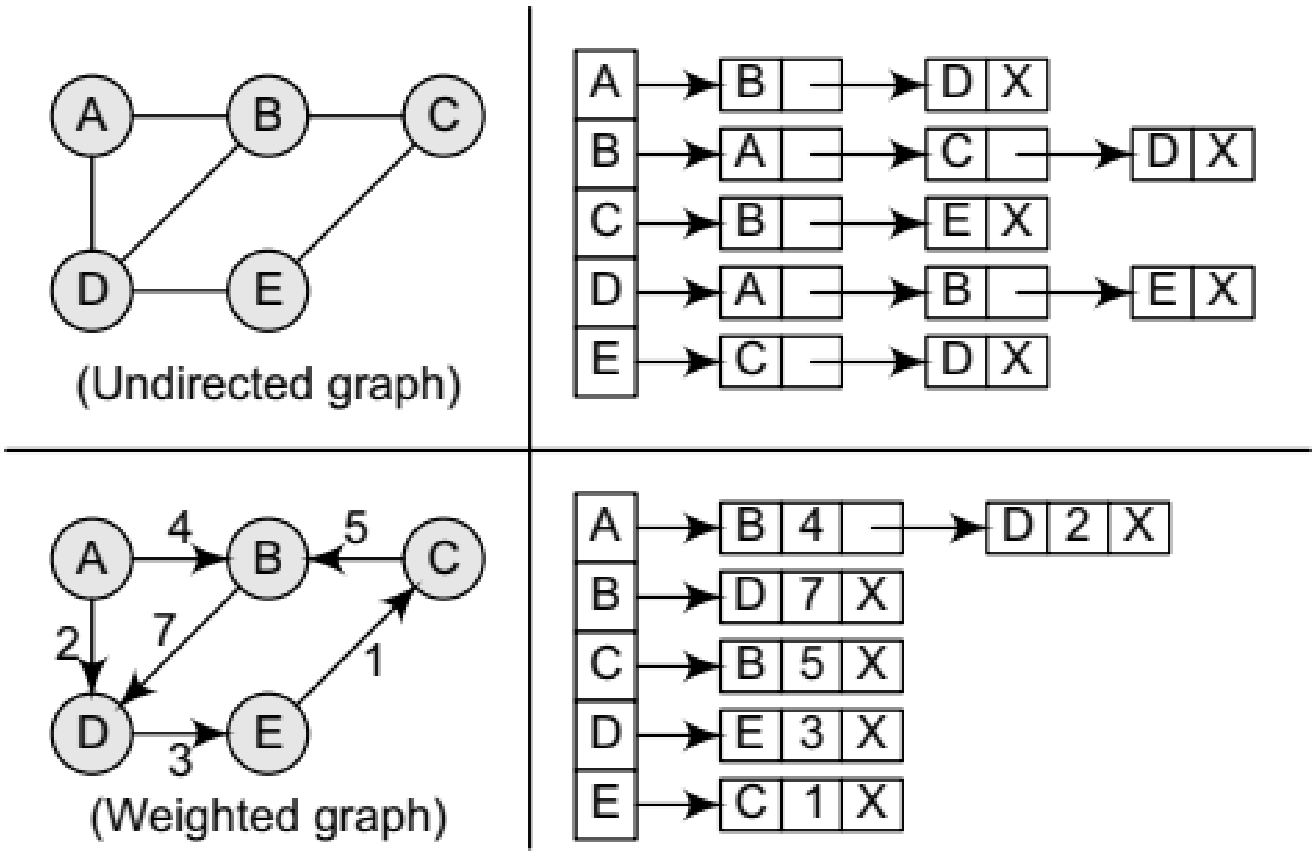
# Example

- **Social network**: facebook
  - $N = 10^9$
  - Adjacency matrix: $10^{18}$
- **If each person has 1000 = $10^3$ friends**
  - $|E| = 10^9 \times 10^3 /2 = 10^{12} = 5 \times 10^{11} \ll 10^{18}$
  - $10^{18}$ (bytes= 1000 PB)
  - $5 \times 10^{11}$(bytes) = 0.5TB
- **Finding adjacent nodes:**
  - If machine can scan $10^6$ cells/ seconds
  - Times: $10^9 / 10^6$ = 1000sec = 16.66mins

# Adjacency List Representation




```
struct node
{
        char vertex;
        struct node *next;
};
struct node *gnode;
```




(Undirected graph)

(Weighted graph)

# Graph traversal algorithms

- **Visit all nodes of a graph**
- **Algorithms:**
  - Breadth-first search: uses a queue as an auxiliary data structure to store nodes for further processing
  - Depth-first search: uses a stack
- **Status value of a node:**

| Status | State of the node | Description |
|---|---|---|
| 1 | Ready | The initial state of the node N |
| 2 | Waiting | Node N is placed on the queue or stack and waiting to be processed |
| 3 | Processed | Node N has been completely processed |

# Algorithm for breadth-first search

```
Step 1: SET STATUS = 1 (ready state)
        for each node in G
Step 2: Enqueue the starting node A
        and set its STATUS = 2
        (waiting state)
Step 3: Repeat Steps 4 and 5 until
        QUEUE is empty
Step 4: Dequeue a node N. Process it
        and set its STATUS = 3
        (processed state).
Step 5: Enqueue all the neighbours of
        N that are in the ready state
        (whose STATUS = 1) and set
        their STATUS = 2
        (waiting state)
        [END OF LOOP]
Step 6: EXIT
```

# Breadth-First search algorithm

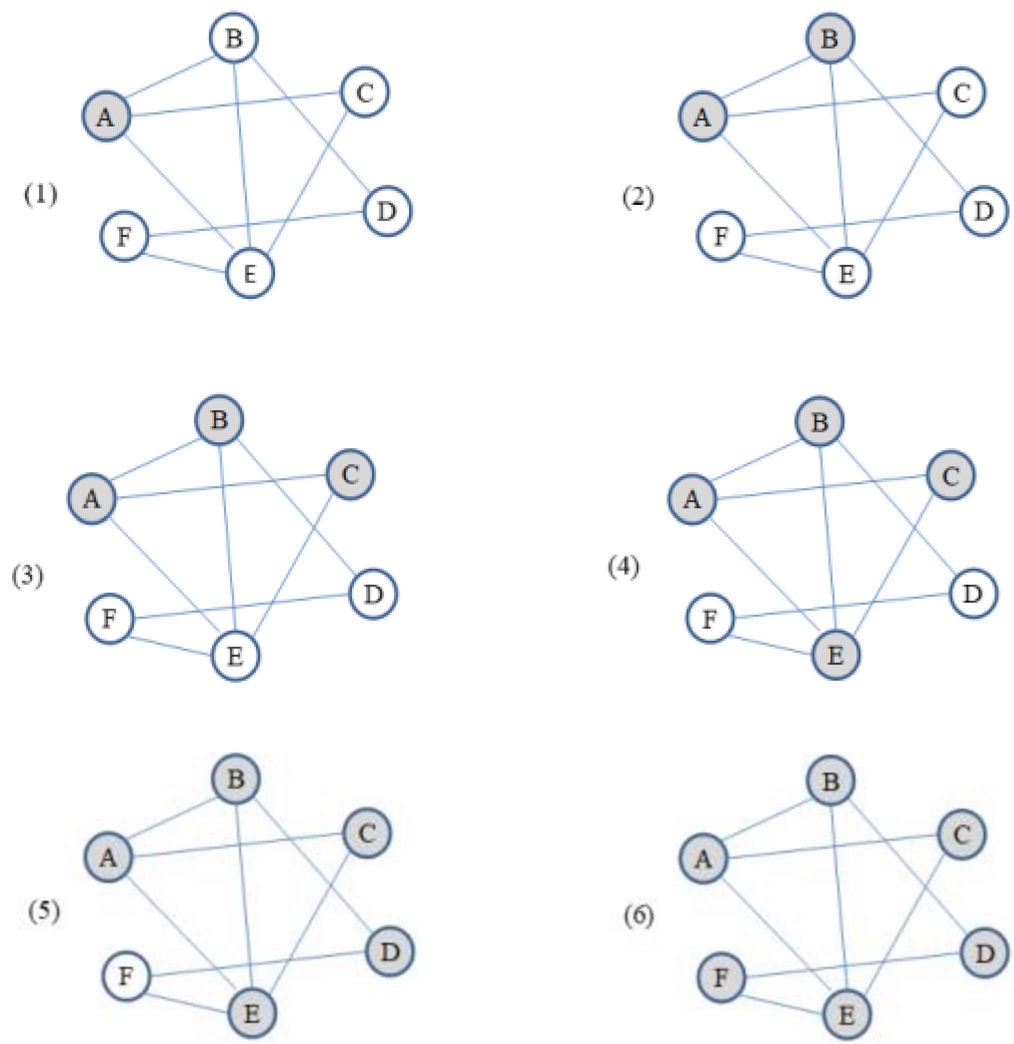
- **Breadth-frst search (BFS) is a graph search algorithm that**
  - begins at the root node
  - and explores all the neighbouring nodes.

# Breadth-First search algorithm

void BreathFirstSearch(G, vs)

```
{
    insert(vs, Q);
    while (!isEmpty(Q))
    {
        u = remove(Q);
        visit(u);
        for each u_a: (unvisited) and
                  (adjacent to u) and (u_a ∉ Q)
            insert(u_a, Q);
    }
}
```

# Breadth-First search algorithm



# Features of Breadth-First Search Algorithm

- **Space complexity:**
  - Given a graph with branching factor b (number of children at each node) and depth d, the asymptotic space complexity is the number of nodes at the deepest level $O(b^d)$
  - If the number of vertices and edges in the graph are known ahead of time, the space complexity can also be expressed as O ( | E | + | V | ), where | E | is the total number of edges in G and | V | is the number of nodes or vertices

# Features of Breadth-First Search Algorithm

- **Time complexity:**
  - In the worst case, breadth-frst search has to traverse through all paths to all possible nodes, thus the time complexity of this algorithm asymptotically approaches $O(b^d)$.
  - However, the time complexity can also be expressed as $O( | E | + | V | )$, since every vertex and every edge will be explored in the worst case

S E T
2020

# Applications of Breadth-First Search Algorithm

- Finding all connected components in a graph G.
- Finding all nodes within an individual connected component.
- Finding the shortest path between two nodes, u and v, of an unweighted graph.
- Finding the shortest path between two nodes, u and v, of a weighted graph

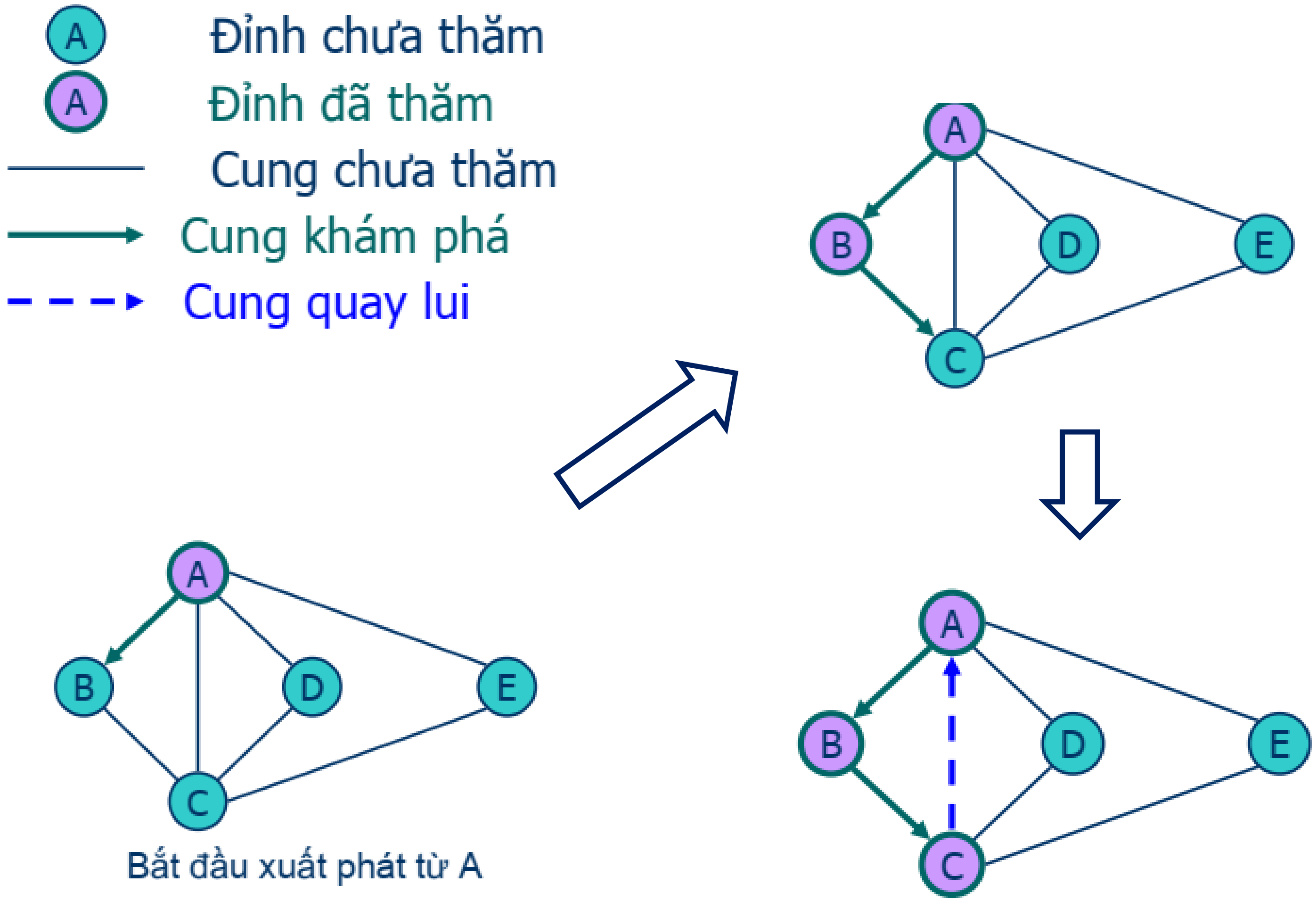
# Depth-first Search Algorithm

- The depth-frst search algorithm progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered
- When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored

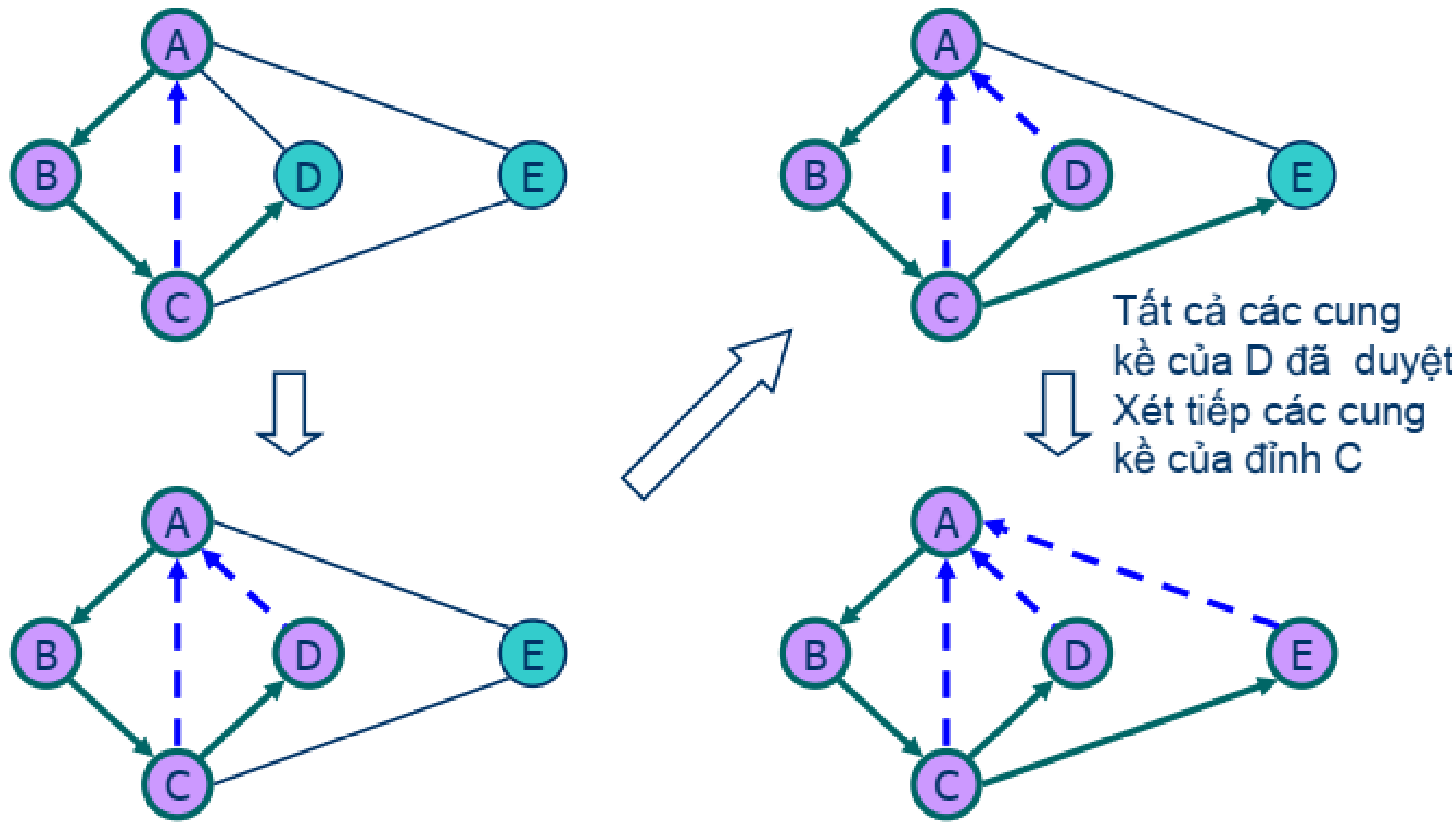
# Algorithm for depth-first search

```
Step 1: SET STATUS = 1 (ready state) for each node in G
Step 2: Push the starting node A on the stack and set
        its STATUS = 2 (waiting state)
Step 3: Repeat Steps 4 and 5 until STACK is empty
Step 4:   Pop the top node N. Process it and set its
          STATUS = 3 (processed state)
Step 5:   Push on the stack all the neighbours of N that
          are in the ready state (whose STATUS = 1) and
          set their STATUS = 2 (waiting state)
        [END OF LOOP]
Step 6: EXIT
```

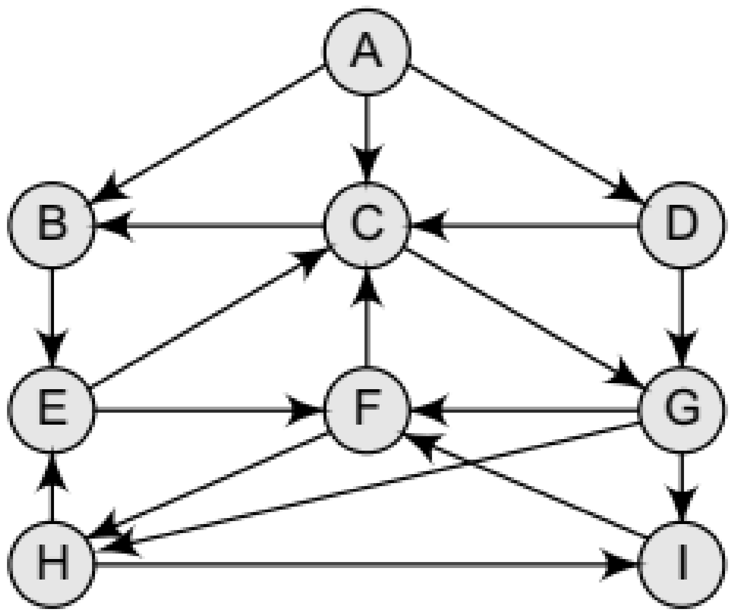
# Algorithm for depth-first search



# Algorithm for depth-first search



# Example




**Adjacency lists**

A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

Suppose we want to print all the nodes that can be reached from the node `H` (including `H` itself).
One alternative is to use a depth-first search of `G` starting at node `H`

# Applications of Depth-First Search Algorithm

- Finding a path between two specifed nodes, u and v, of an unweighted graph.
- Finding a path between two specifed nodes, u and v, of a weighted graph.
- Finding whether a graph is connected or not.
- Computing the spanning tree of a connected graph

# Algorithm for depth-first search

### Ý tưởng giải thuật

Đây là một giải thuật đệ quy có các bước cơ bản như sau:

1. Đặt $v = v_s$.
2. Thăm nút $v$.
3. Với mỗi nút $v_a$ là kề của $v$ mà chưa được thăm, lặp lại bước 2 cho $v_a$.
4. (Điểm dừng): giải thuật kết thúc khi đồ thị không còn nút nào cần phải thăm nữa.

# Algorithm for depth-first search

**Thủ tục**

Gọi thủ tục cài đặt cho giải thuật là *FullDepthSearch(G, vs)*, với *vs* là đỉnh bắt đầu mà thuộc tập các đỉnh V của đồ thị G.

**void FullDepthSearch(G, vs)**

```
{
    v = vs;
    DepthFirstSearch(G, v);
}
```

# DepthFirstSearch

DepthFirstSearch(G, v) là thủ tục đệ quy duyệt theo chiều sâu có dạng như sau:

**void DepthFirstSeach(G, v)**

```
{
    if (v == NULL) return; //Điểm dừng
    visit(v);
    for each unvisited adjacent va to v
        DepthFirstSearch(G, va);
}
```