# HUST

**ĐẠI HỌC BÁCH KHOA HÀ NỘI**
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

ONE LOVE. ONE FUTURE.

# APPLIED ALGORITHMS

ĐẠI HỌC
BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY
OF SCIENCE AND TECHNOLOGY

# APPLIED ALGORITHMS

## DEPTH FIRST SEARCH (DFS) AND APPLICATIONS

ONE LOVE. ONE FUTURE.

# CONTENTS

- **Depth First Search (DFS)**
- DFS tree and và structure Num, Low
- Find bridges (cạnh cầu)
- Find articulation vertex (đỉnh khớp)
- Find strongly connected component (thành phần liên thông mạnh)

# Depth First Search (Tìm kiếm theo chiều sâu)

- Depth First Search is a basic graph traversal technique (visiting every vertex and every edge of the graph).
  - The algorithm can answer the question, does there exist a path from vertex $u$ to vertex $v$ on graph $G$ or not, if yes, indicate it.
  - The algorithm not only answers whether there is a path from $u$ to $v$, but it can answer which other vertices on the graph $G$ can be reached from u.
- The traversal order in DFS follows the ***Last In First Out*** (LIFO) mechanism, and starts from a certain starting vertex $u$.
  - Can use backtracking or stack to implement
- The complexity: $O(|V| + |E|)$, where $V$ is the vertex set and $E$ is the edge set of the graph $G$, as each vertex and edge of $G$ is visited exactly once.

# Implement idea

- Graph G = (V, E) is represented by an adjacent list
  - A[u]: list of adjacent nodes of u
- Marking array:
  - visited[u] = true means u was visited, and visited[u] = false means that u is not visited

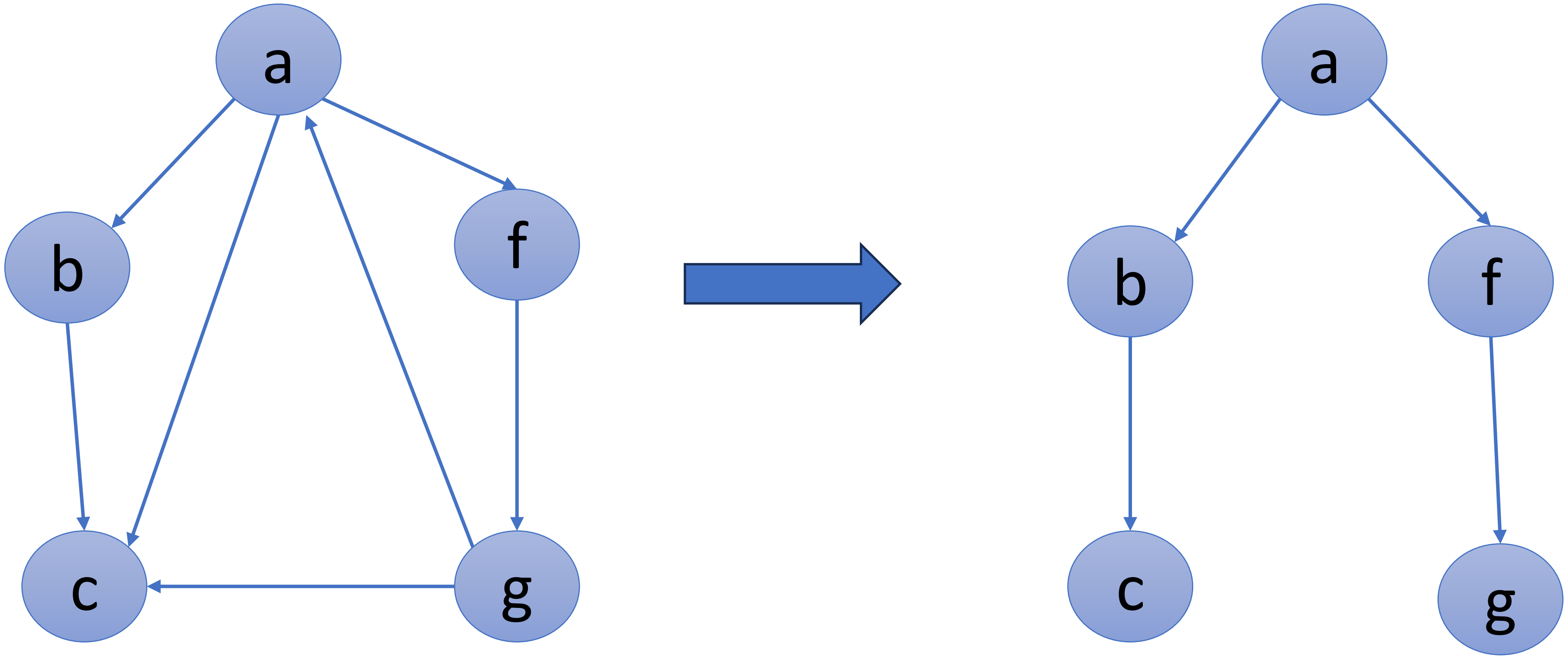
```
DFS(u, V, A) {
  visit(u); // assign visited[u] = true
  for v in A[u] do {
    if not visited[v] then {
        DFS(v, V, A);
    }
  }
}
DFS(V, A){
  for u in V  do {   visited[u] = false;   }
  for u in V do {
    if not visited[u] then
       DFS(u, V, A);
  }
}
```

# CONTENTS

- Depth First Search (DFS)
- **DFS tree and và structure Num, Low**
- Find bridges (cạnh cầu)
- Find articulation vertex (đỉnh khớp)
- Find strongly connected component (thành phần liên thông mạnh)

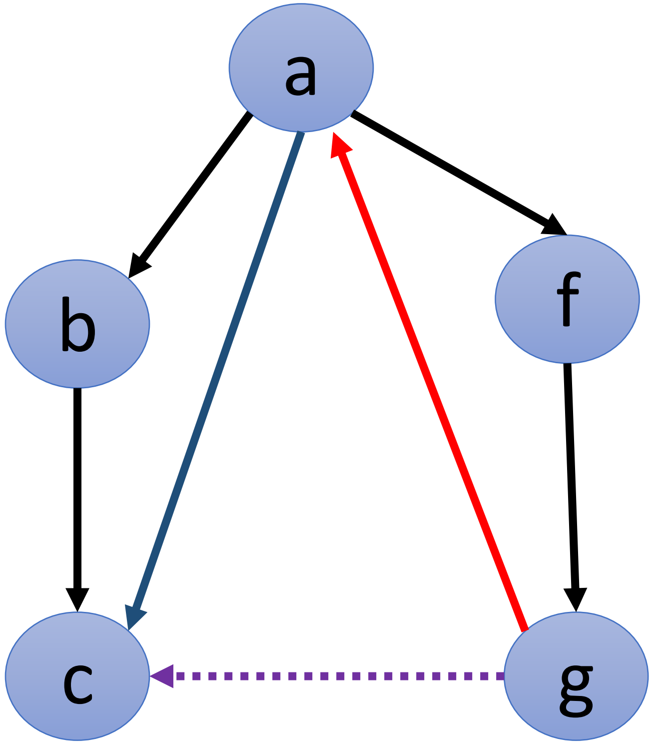
# DFS tree

- The trace of the depth-first search will form a tree



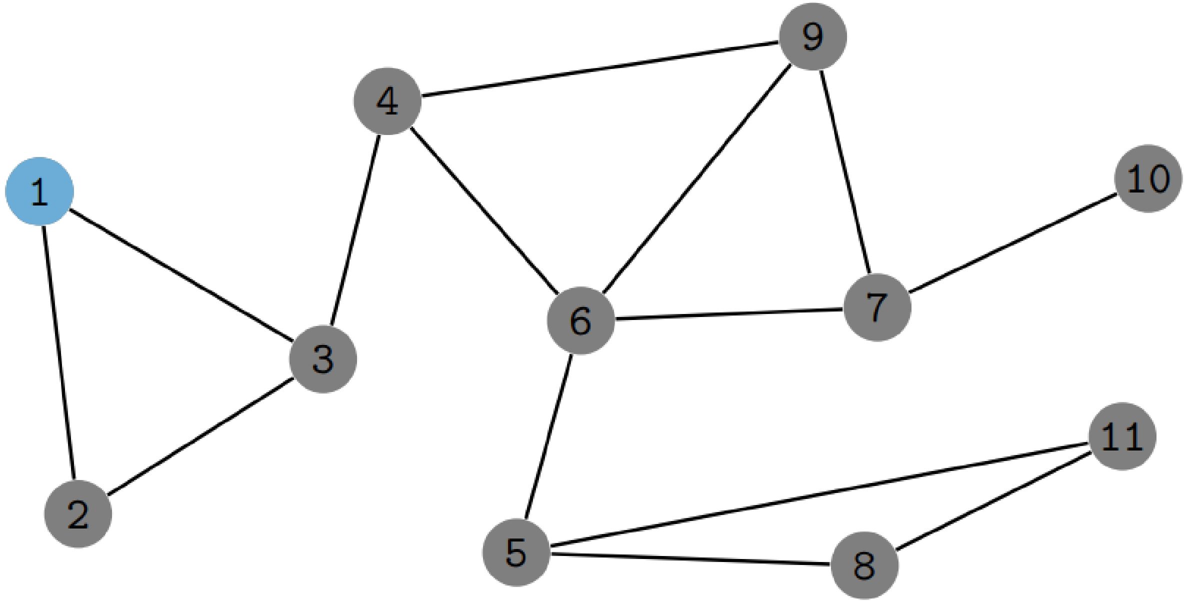
# DFS tree

- The trace of the depth-first search will form a tree
- Some kind of edge in the search process:
  - Tree Edge (Cạnh cây): The edge along which a new vertex is visited from one vertex, for example the black edge in the figure
  - Back Edge (Cạnh ngược): The edge going from descendant to ancestor, for example the red edge (*g*,*a*) in the figure
  - Forward Edge (Cạnh xuôi): The edge going from ancestors to descendants, for example the blue edge (*a*,*c*) in the figure
  - Crossing Edge (Cạnh vòng): The edge connecting two unrelated vertices, for example the dashed purple edge (*c*,*g*) in the figure



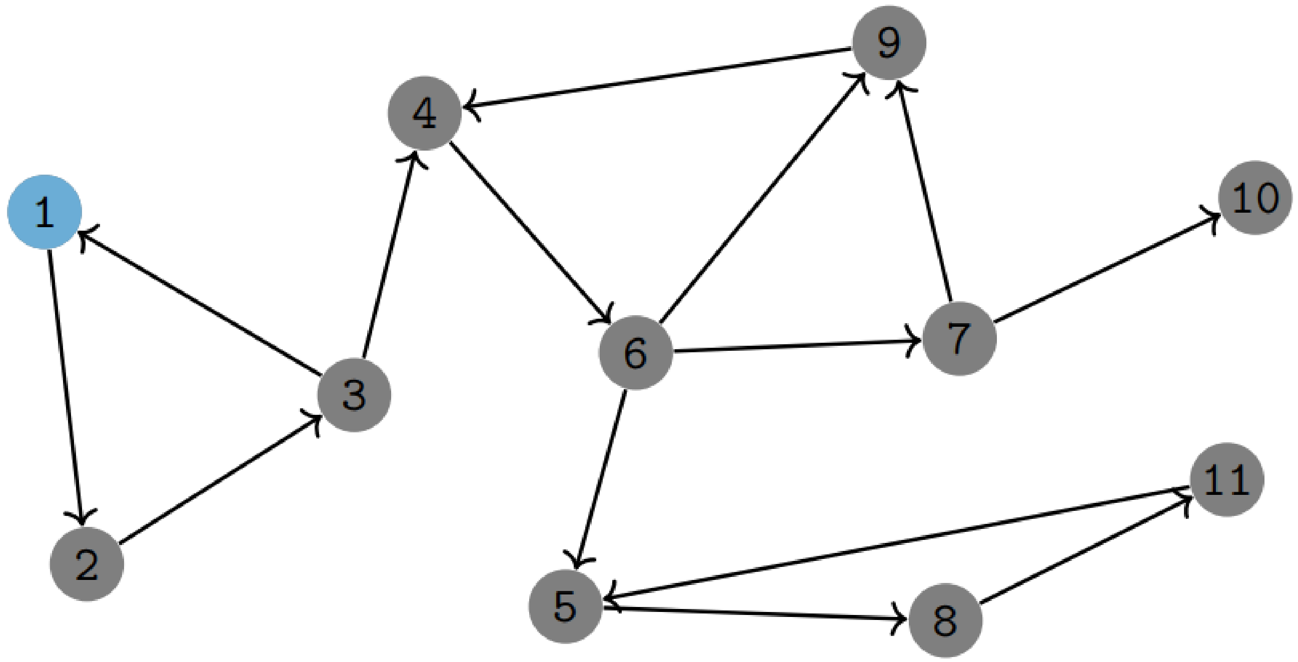
# DFS tree

- Arrays $Num$ and $Low$ store information of vertices in DFS tree:
  - $Num[u]$: visit order of vertex u in DFS
  - $Low[u]$: has the smallest value among the following values:
    - $Num[v]$ if $(v, u)$ is a back edge
    - $Low[v]$ if $v$ is a child of $u$ in DFS tree
    - $Num[u]$




| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Num[i] | 1 | 2 | 3 | 4 | 6 | 5 | 9 | 7 | 10 | 11 | 8 |
| Low[i] | 1 | 1 | 1 | 4 | 6 | 4 | 4 | 6 | 4 | 11 | 6 |

# Example




| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Num[i] | 1 | 2 | 3 | 4 | 6 | 5 | 9 | 7 | 10 | 11 | 8 |
| Low[i] | 1 | 1 | 1 | 4 | 6 | 4 | 4 | 6 | 4 | 11 | 6 |

# Implementation ideas

- p[v]: parent of  v discovered during the DFS
- Num[u] = 0: node u has not been visited yet
- Num[u] > 0: is visited, and Num[u] is the order that u is visited

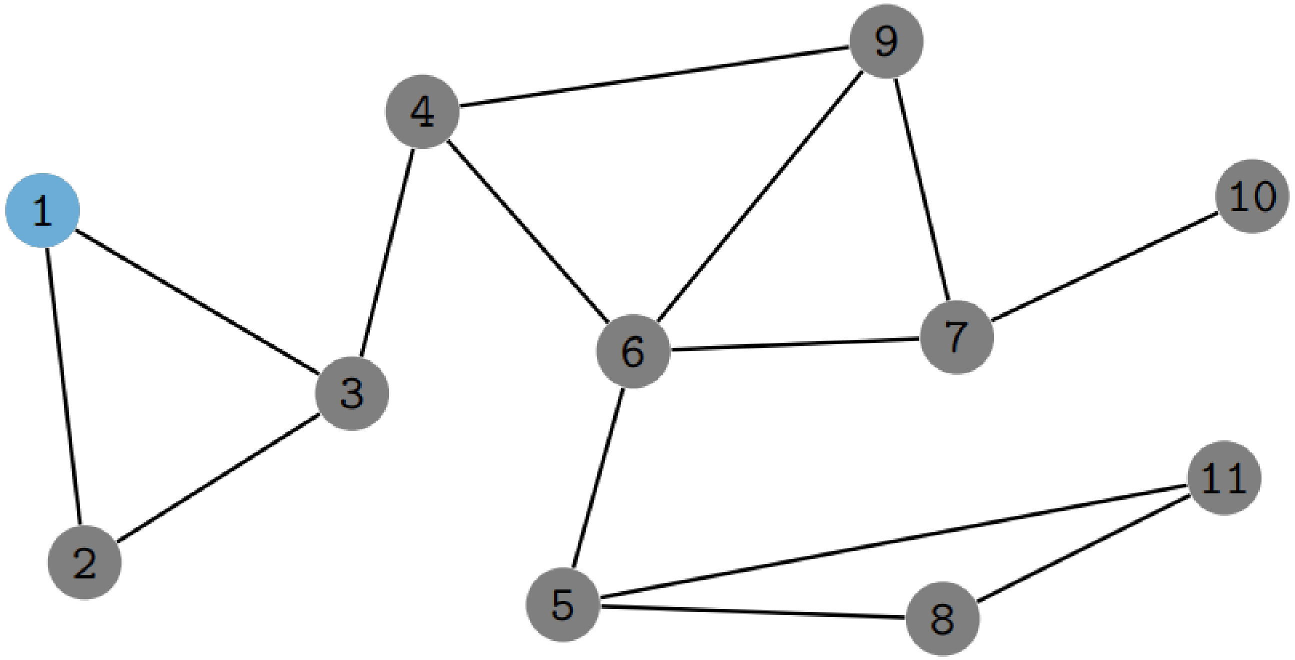
```
DFS(u, V, A, p) {
    T += 1; Num[u] = T; Low[u] = T;
    for v in A[u] do {
        if v = p[u] continue;
        if Num[v] > 0 then { // v was visited
            Low[u] = min(Low[u], Num[v]);
        } else {
            p[v] = u;
            DFS(v, V, A, p);
            Low[u] = min(Low[u], Low[v]);
        }
    }
}
```

# CONTENTS

- Depth First Search (DFS)
- DFS tree and và structure Num, Low
- **Find bridges (cạnh cầu)**
- Find articulation vertex (đỉnh khớp)
- Find strongly connected component (thành phần liên thông mạnh)

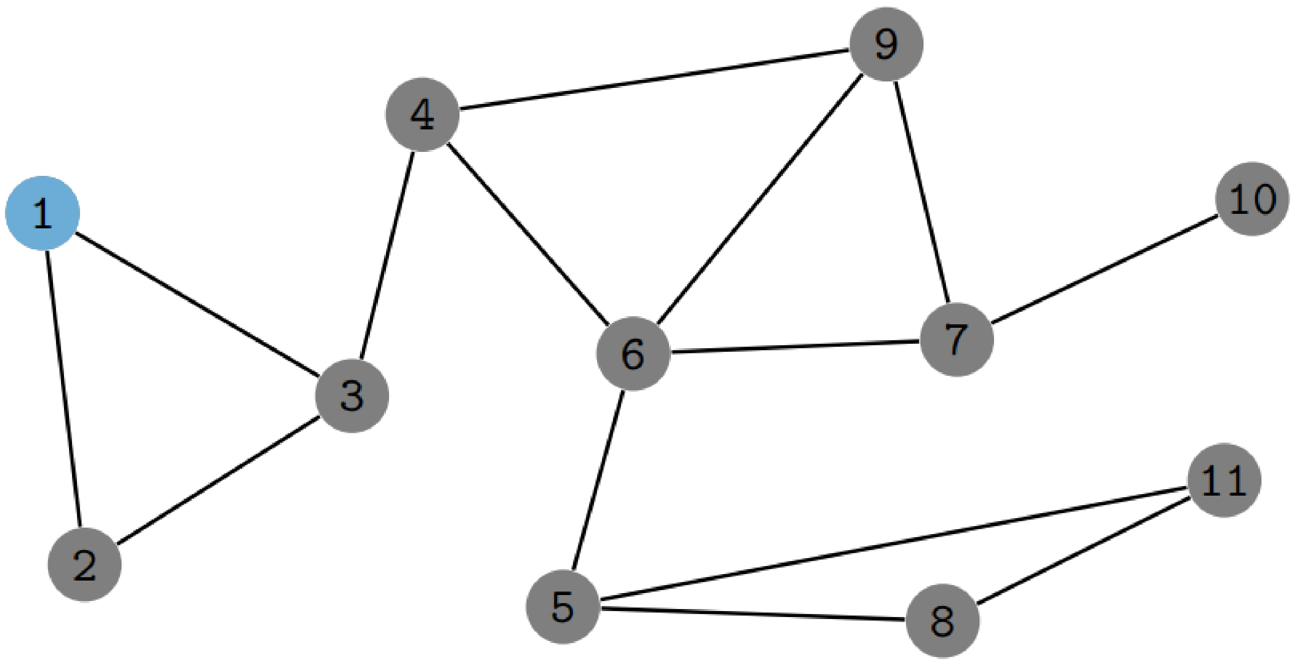
# Find bridge (cầu) in the graph

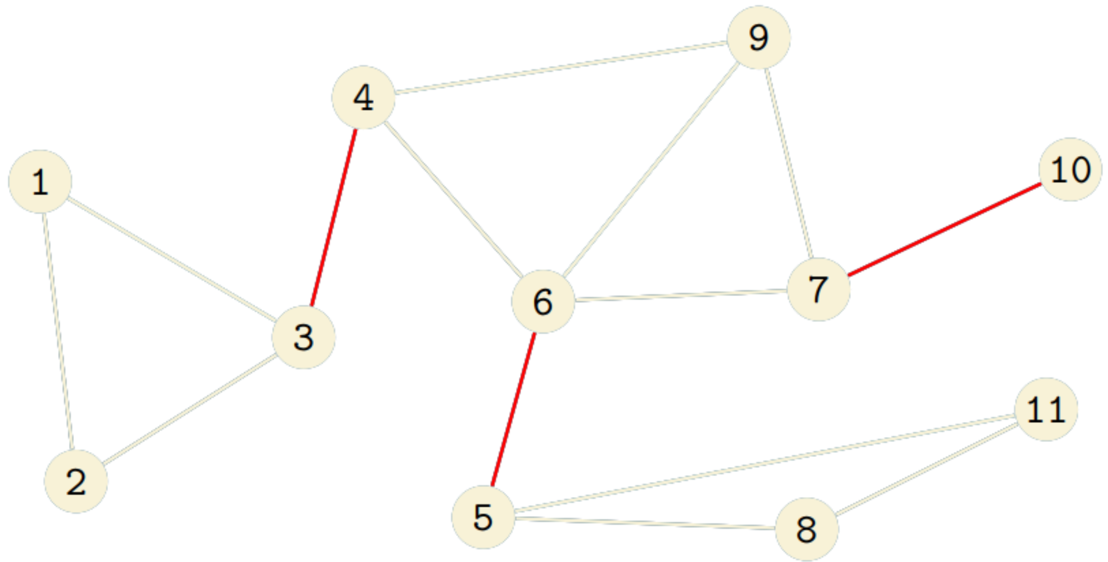
- **Definition:** A bridge is an edge of an undirected graph, so that removing this edge from the graph will increase the number of connected components.
- **Comment**: A forward edge $(u, v)$ is bridge if and only if $Low[v] > Num[u]$



# Find bridge (cầu) in the graph

- **Definition:** A bridge is an edge of an undirected graph, so that removing this edge from the graph will increase the number of connected components.
- **Comment**: A forward edge $(u, v)$ is bridge if and only if $Low[v] > Num[u]$




| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Num[i] | 1 | 2 | 3 | 4 | 6 | 5 | 9 | 7 | 10 | 11 | 8 |
| Low[i] | 1 | 1 | 1 | 4 | 6 | 4 | 4 | 6 | 4 | 11 | 6 |

# Implementation idea

- p[v]: parent of  v discovered during the DFS
- Num[u] = 0: node u has not been visited yet
- Num[u] > 0: is visited, and Num[u] is the order that u is visited

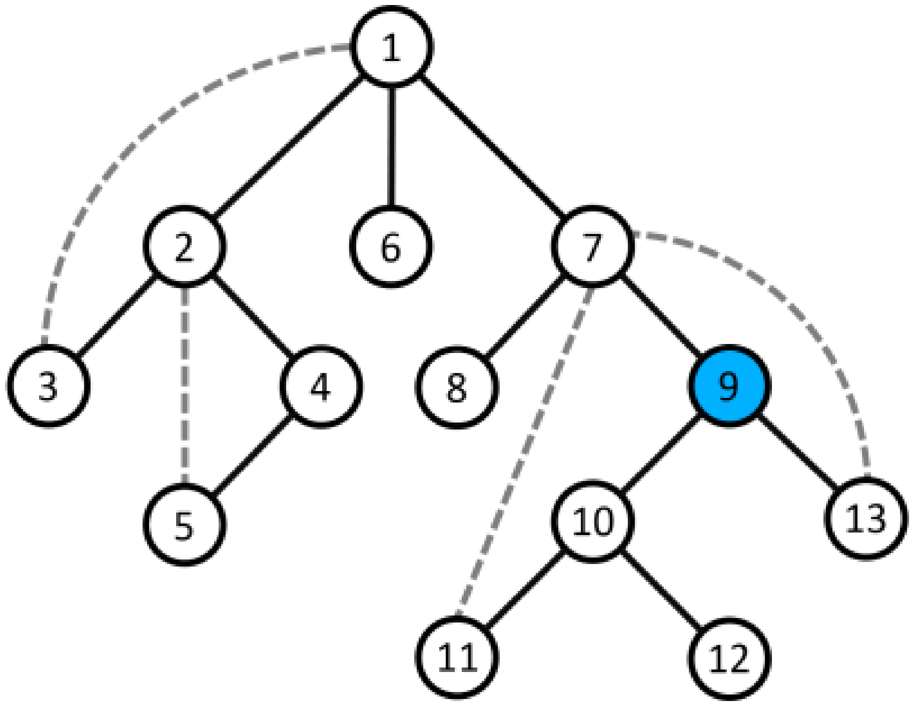
```
DFS(u) {
   T += 1; Num[u] = T; Low[u] = T;
   for v in A[u] do {
      if v = p[u] continue;
      if Num[v] > 0 then { // v was visited
         Low[u] = min(Low[u], Num[v]);
      } else {
         p[v] = u;
         DFS(v);
         Low[u] = min(Low[u], Low[v]);
         if Low[v] > Num[u] then (u,v) is a bridge;
      }
   }
}
```

# CONTENTS

- Depth First Search (DFS)
- DFS tree and và structure Num, Low
- Find bridges (cạnh cầu)
- **Find articulation vertex (đỉnh khớp)**
- Find strongly connected component (thành phần liên thông mạnh)

# Find articulation vertex (đỉnh khớp)

- **Definition**: In an undirected graph, a vertex is called an articulation vertex if removing this vertex and edges having it as end point from the graph will increase the number of connected components of the graph.
- **Comment**: Vertex $u$ vertex is called an articulation vertex if:
  - Either vertex $u$ is not the root of DFS tree and $Low[v] \geq Num[u]$ (where $v$ is any direct child of $u$ in the DFS tree);
  - Or vertex $u$ is the root of the DFS tree and has at least 2 direct children.



# Implementation idea

- p[v]: parent of v discovered during the DFS
- Num[u] = 0: node u has not been visited yet
- Num[u] > 0: is visited, and Num[u] is the order that u is visited
- numChild[u]: number of children of u

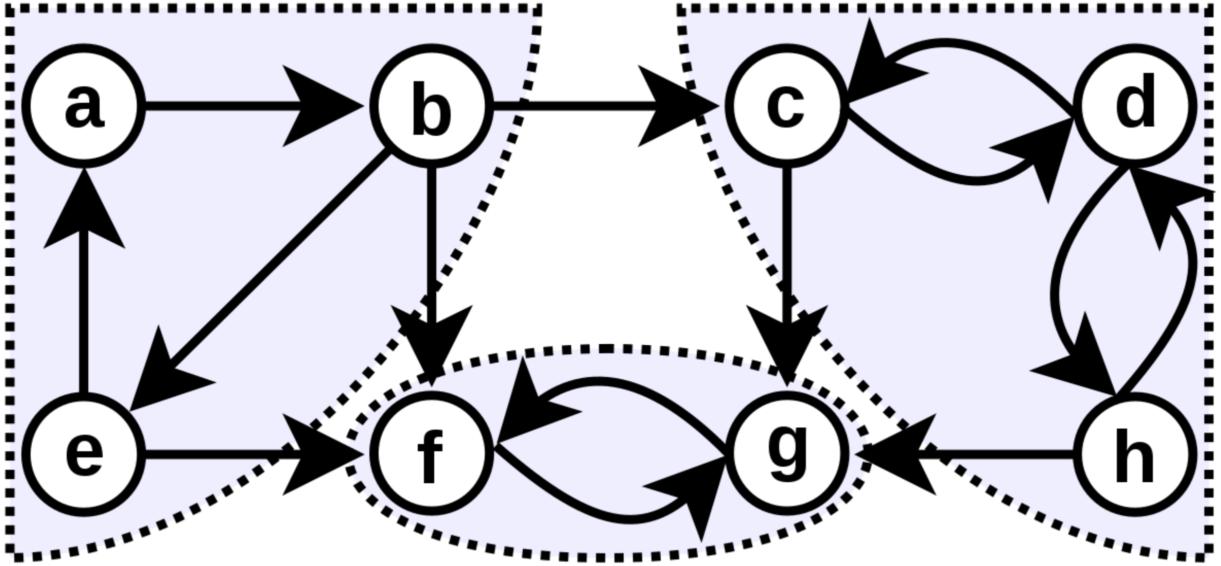
```
DFS(u) {
   T += 1; Num[u] = T; Low[u] = T;
   for v in A[u] do {
      if v = p[u] continue;
      if Num[v] > 0 then {  // v was visited
         Low[u] = min(Low[u], Num[v]);
      } else { // visit v
         p[v] = u;   numChild[u] += 1;
         DFS(v);
         Low[u] = min(Low[u], Low[v]);
         if u = p[u] then { // u là đỉnh xuất phát DFS (root)
            if numChild[u] >= 2 then { u is an articulation point; }
         } else { if Low[v] >= Num[u] then u is an articulation point; }
      }
   }
}
```

# CONTENTS

- Depth First Search (DFS)
- DFS tree and và structure Num, Low
- Find bridges (cạnh cầu)
- Find articulation vertex (đỉnh khớp)
- **Find strongly connected component (thành phần liên thông mạnh)**

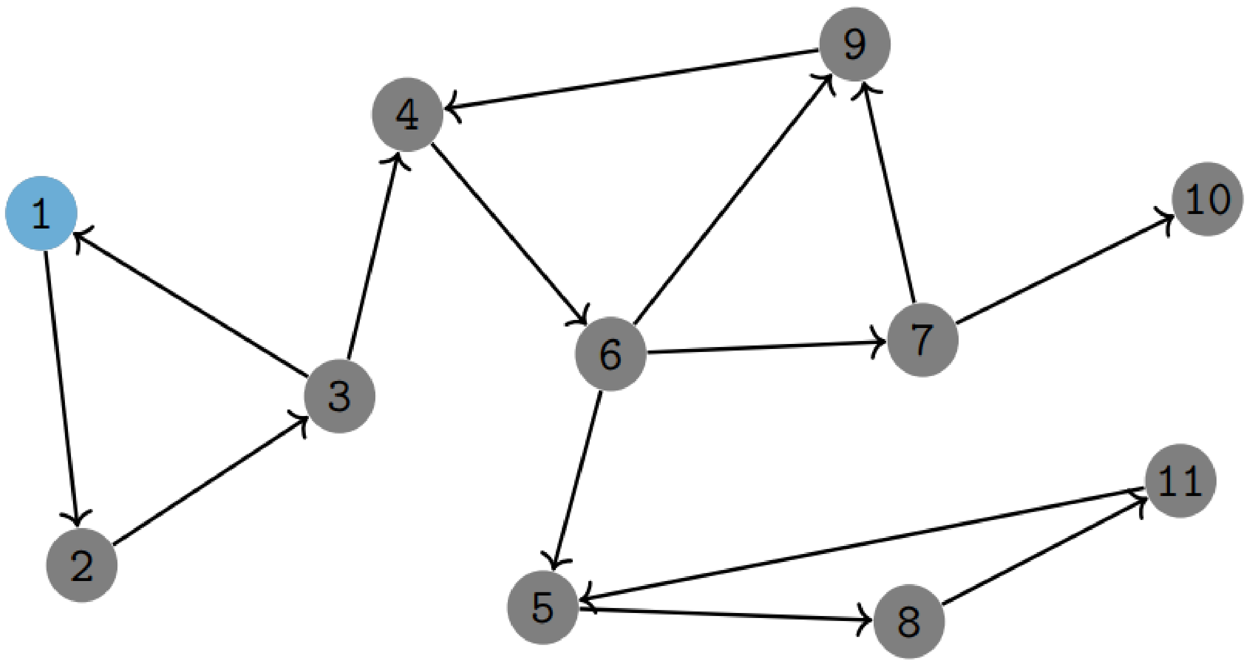
# Find Strongly Connected Components

- Breadth-First-Search (BFS) and Depth-First-Search (DFS) can easily find all connected components in an undirected graph. However, finding all strongly connected components is not simple for directed graph.
  - Note: A strongly connected component is a maximum subset of vertices such that between any two vertices there is always a path from one vertex to the other and vice versa.
- DFS search trees can be used to find
- all strongly connected components?



# Thuật toán Tarjan

- <u>**Observation**</u>: After analyzing the DFS tree, if at a vertex $u$, $Num[u]$ = $Low[u]$, then we have a strongly connected component following the process of traversing the tree from $u$.
- Use Stack ST to list vertices in a strongly connected component.
- Marking: inStack[u] = true means that u is in the Stack ST
  - After visiting u and the edge (u,v) is discovered in which v was visited → We can recognize the edge (u,v) is a back eddge or crossing edge:
    - inStack[v] = true: (u,v) is a back edge
    - inStack[v] = false: (u,v) is a crossing edge
- Complexity: $O(|V| + |E|)$




| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Num[i] | 1 | 2 | 3 | 4 | 6 | 5 | 9 | 7 | 10 | 11 | 8 |
| Low[i] | 1 | 1 | 1 | 4 | 6 | 4 | 4 | 6 | 4 | 11 | 6 |

# Implementation idea

```
DFS(u) {
   T += 1; Num[u] = T; Low[u] = T;  ST.push(u); inStack[u] = true;
   for v in A[u] do {
      if v = p[u] continue;
      if Num[v] > 0 then {  // v was visited
         if inStack[v] then Low[u] = min(Low[u], Num[v]);     // (u,v) is a back edge
      } else { // visit v
         p[v] = u;   DFS(v);   Low[u] = min(Low[u], Low[v]);
      }
   }
   if Low[u] = Num[u] then {// retrieve a strongly connected component stored in stack ST
      while ST not empty do {   x = ST.pop(); print(x); inStack[x] = false;  if x = u then break; }
   }
}
```

HUST

hust.edu.vn fb.com/dhbkhn

# THANK YOU !