ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Discrete Mathematics

Course preparation group:
Nguyễn Khánh Phương
Đỗ Phan Thuận
Phạm Quang Dũng
Huỳnh Thanh Bình
Trần Vĩnh Đức
Bùi Quốc Trung
Đinh Viết Sang
Ban Hà Bằng

Content of Part 2

PART 1
COMBINATORIAL THEORY
(Lý thuyết tổ hợp)

PART 2
GRAPH THEORY
(Lý thuyết đồ thị)

Graph traversal (Graph searching)

Searching a graph means systematically following the edges of the graph so as to visit the vertices.

2 algorithms:

- Breadth First Search – BFS
- Depth First Search – DFS

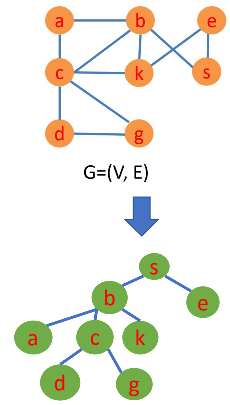
Breadth-first Search (BFS)

Breadth-first Search

```
BFS(s)
// Breadth first search starts from vertex s
    visited[s] ← 1; //visited
    Q ← ∅; enqueue(Q,s); // insert s into Q
    while (Q ≠ ∅)
    {
        u ← dequeue(Q); // Remove u from Q
        for v ∈Adj[u]
            if (visited[v] == 0) //not visited yet
            {
                visited[v] ← 1; //visited
                enqueue(Q,v)  // insert v into Q
            }
    }

(*Main Program*)
main ()
    for s ∈ V // Initialize
        visited[s] ← 0;

    for s ∈ V
        if (visited[s]==0) BFS(s);
```



Breadth First Search

- Given
  - a graph G=(V,E) – set of vertices and edges
  - a distinguished source vertex s
- Breadth first search systematically explores the edges of G to discover every vertex that is reachable from s.
- It also produces a 'breadth first tree' with root *s* that contains all the vertices reachable from *s*.
- For any vertex *v* reachable from *s*, the path in the breadth first tree corresponds to the shortest path in graph G from *s* to *v*.
- It works on both directed and undirected graphs.

Computation time of BFS


```
BFS(s)
// Breadth first search starts from vertex s
    visited[s] ← 1; //visited
    Q ← ∅; enqueue(Q,s); // insert s into Q
    while (Q ≠ ∅)
    {
        u ← dequeue(Q); // Remove u from Q
        for v ∈Adj[u]
            if (visited[v] == 0) //not visited yet
            {
                visited[v] ← 1; //visited
                enqueue(Q,v)  // insert v into Q
            }
    }

(*Main Program*)
main ()
    for s ∈ V // Initialize
        visited[s] ← 0;

    for s ∈ V
        if (visited[s]==0) BFS(s);
```


- The computation time of BFS(s) is $O(|V|+|E|)$, linear to the size of adjacency list that represents the graph.

Depth-first Search (DFS)

Computation time of DFS

```
(* Main Program*)
1.  for s ∈ V
2.      visited[s] ← false
3   for s ∈ V
4.      if (visited[s] == false)
5.          DFS(s)
```

```
DFS(s)
1.  visited[s] ← true  // Visit s
2.  for each v ∈ Adj[s]
3.      if (visited[v] == false)
4.          DFS(v)
```

- In `main`: the loops for on lines 1-2 and 3-5 require $O(|V|)$, excluding computation time of statement `DFS(s)`.
- In `DFS(s)`: the loop for on line 2 performs to traverse edges of graph:
  - Lines 3-4 of `DFS(s)` perform $|\mathrm{Adj}[s]|$ times
  - Each edge is traversed exactly once if graph is directed, otherwise exactly twice if graph is undirected

→ The total computation time of `DFS(s)` in the main program is $\sum_{s \in V}|\mathrm{Adj}[s]| = O(|E|)$

- Thus, computation time of DFS is $O(|V|+|E|)$.

Depth First Search (DFS)

- It searches 'deeper' the graph when possible.
- Starts at the selected node and explores as far as possible along each branch before backtracking.

```
(* Main Program*)
1. for s ∈ V
2.     visited[s] ← false
3. for s ∈ V
4.     if (visited[s] == false)
5.         DFS(s)
```

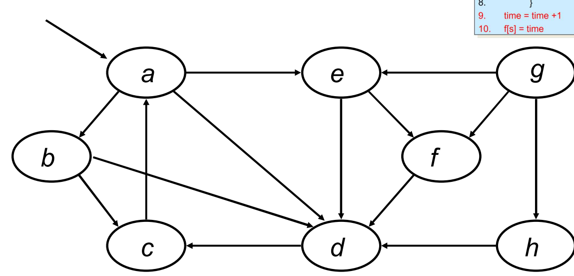
```
DFS(s)
1.  visited[s] ← true  // Visist s
2.  for each v ∈ Adj[s]
3.      if (visited[v] == false)
4.          DFS(v)
```


Example: DFS

```
(*Main program *)
1.  for each s ∈ V
2.      pred[s] = NULL;
3.      visited[s] = false;
4.  time = 0
5.  for each s ∈ V
6.    if (visited[s] == false) DFS(s);
```

```
DFS(s)
1.  visited[s]= true; //Visit s
2.  time = time +1
3.  d[s] = time
4.  for each v ∈ Adj[s]
5.      if (visited[v] == false) {
6.          pred[v] ← s;
7.          DFS(v);
8.      }
9.  time = time +1
10. f[s] = time
```

Source vertex

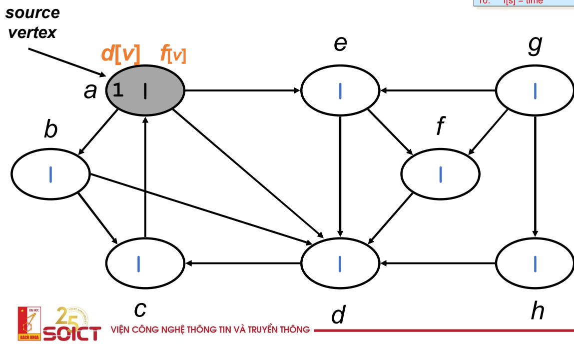


For the operation of the algorithm to be deterministic, assume that we traverse the vertices in the adjacency list of a vertex in lexical order.

Example: DFS

```
(*Main program *)
for each s ∈ V
      pred[s] = NULL;
      visited[s] = false;
time = 0
for each s ∈ V
   if (visited[s] == false) DFS(s);
```

```
DFS(s)
visited[s]= true; //Visit s
time = time +1
d[s] = time
for each v ∈ Adj[s]
      if (visited[v] == false) {
           pred[v] ← s;
           DFS(v);
      }
time = time +1
f[s] = time
```

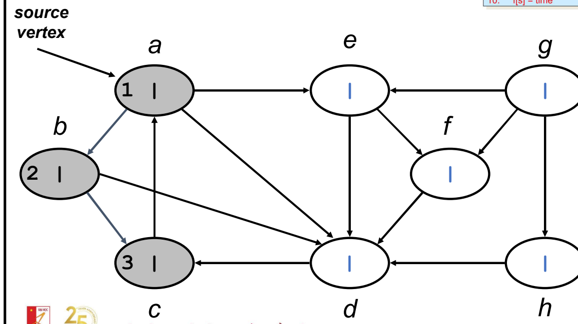


SOICT VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Example: DFS

```
(*Main program *)
for each s ∈ V
      pred[s] = NULL;
      visited[s] = false;
time = 0
for each s ∈ V
   if (visited[s] == false) DFS(s);
```

```
DFS(s)
visited[s]= true; //Visit s
time = time +1
d[s] = time
for each v ∈ Adj[s]
      if (visited[v] == false) {
           pred[v] ← s;
           DFS(v);
      }
time = time +1
f[s] = time
```

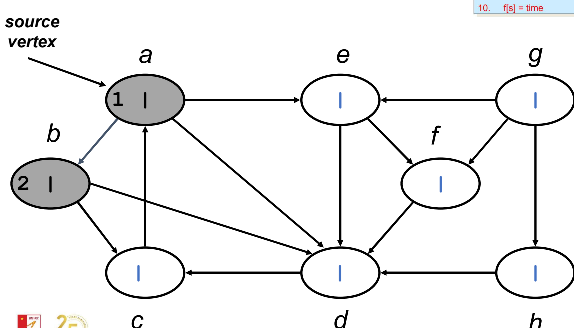


SOICT VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Example: DFS

```
(*Main program *)
for each s ∈ V
      pred[s] = NULL;
      visited[s] = false;
time = 0
for each s ∈ V
   if (visited[s] == false) DFS(s);
```

```
DFS(s)
visited[s]= true; //Visit s
time = time +1
d[s] = time
for each v ∈ Adj[s]
      if (visited[v] == false) {
           pred[v] ← s;
           DFS(v);
      }
time = time +1
f[s] = time
```

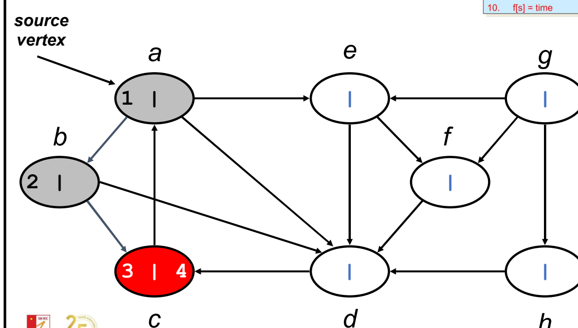


SOICT VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Example: DFS

```
(*Main program *)
for each s ∈ V
      pred[s] = NULL;
      visited[s] = false;
time = 0
for each s ∈ V
   if (visited[s] == false) DFS(s);
```

```
DFS(s)
visited[s]= true; //Visit s
time = time +1
d[s] = time
for each v ∈ Adj[s]
      if (visited[v] == false) {
           pred[v] ← s;
           DFS(v);
      }
time = time +1
f[s] = time
```

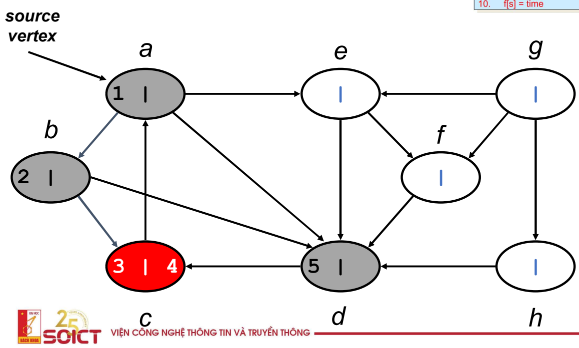


SOICT VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Example: DFS

```
(*Main program *)
for each s ∈ V
    pred[s] = NULL;
    visited[s] = false;
time = 0
for each s ∈ V
  if (visited[s] == false) DFS(s);
```

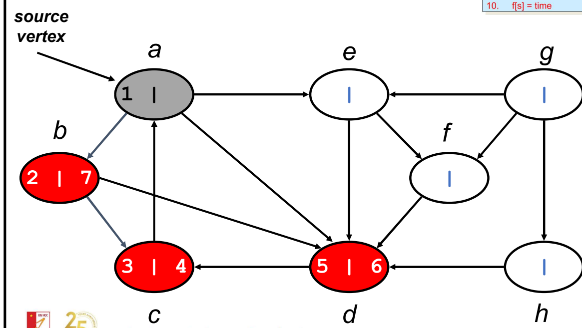
```
DFS(s)
visited[s]= true; //Visit s
time = time +1
d[s] = time
for each v ∈ Adj[s]
    if (visited[v] == false) {
        pred[v] ← s;
        DFS(v);
    }
time = time +1
f[s] = time
```



Example: DFS

```
(*Main program *)
for each s ∈ V
    pred[s] = NULL;
    visited[s] = false;
time = 0
for each s ∈ V
  if (visited[s] == false) DFS(s);
```

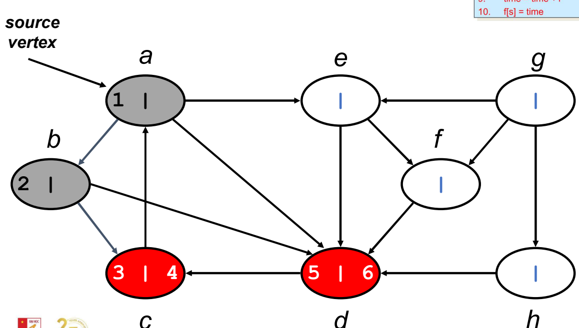
```
DFS(s)
visited[s]= true; //Visit s
time = time +1
d[s] = time
for each v ∈ Adj[s]
    if (visited[v] == false) {
        pred[v] ← s;
        DFS(v);
    }
time = time +1
f[s] = time
```



Example: DFS

```
(*Main program *)
for each s ∈ V
    pred[s] = NULL;
    visited[s] = false;
time = 0
for each s ∈ V
  if (visited[s] == false) DFS(s);
```

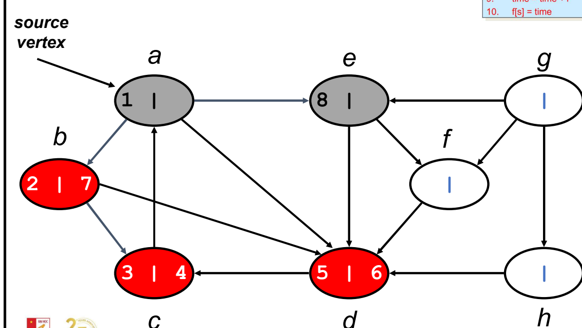
```
DFS(s)
visited[s]= true; //Visit s
time = time +1
d[s] = time
for each v ∈ Adj[s]
    if (visited[v] == false) {
        pred[v] ← s;
        DFS(v);
    }
time = time +1
f[s] = time
```



Example: DFS

```
(*Main program *)
for each s ∈ V
    pred[s] = NULL;
    visited[s] = false;
time = 0
for each s ∈ V
  if (visited[s] == false) DFS(s);
```

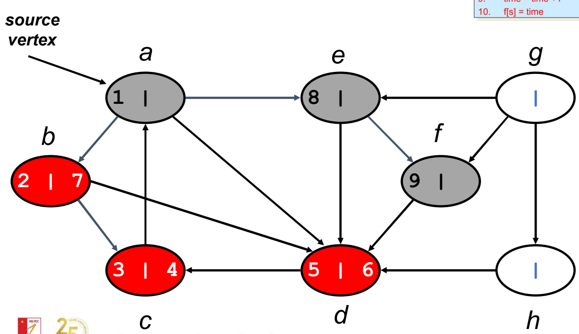
```
DFS(s)
visited[s]= true; //Visit s
time = time +1
d[s] = time
for each v ∈ Adj[s]
    if (visited[v] == false) {
        pred[v] ← s;
        DFS(v);
    }
time = time +1
f[s] = time
```



Example: DFS

```
(*Main program *)
for each s ∈ V
        pred[s] = NULL;
        visited[s] = false;
time = 0
for each s ∈ V
    if (visited[s] == false) DFS(s);
```

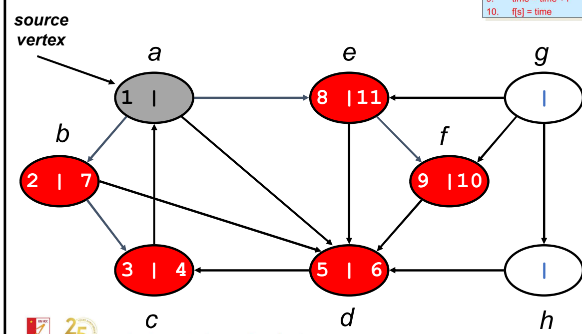
```
DFS(s)
visited[s]= true; //Visit s
time = time +1
d[s] = time
for each v ∈ Adj[s]
        if (visited[v] == false) {
                pred[v] ← s;
                DFS(v);
        }
time = time +1
f[s] = time
```



Example: DFS

```
(*Main program *)
for each s ∈ V
        pred[s] = NULL;
        visited[s] = false;
time = 0
for each s ∈ V
    if (visited[s] == false) DFS(s);
```

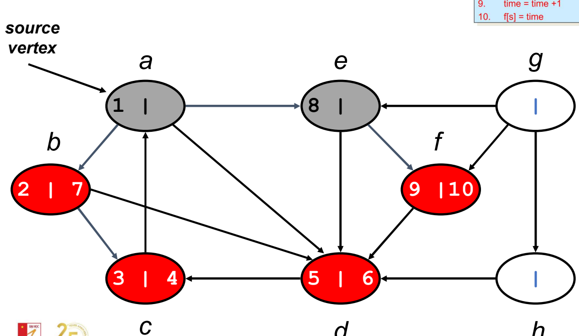
```
DFS(s)
visited[s]= true; //Visit s
time = time +1
d[s] = time
for each v ∈ Adj[s]
        if (visited[v] == false) {
                pred[v] ← s;
                DFS(v);
        }
time = time +1
f[s] = time
```

Example: DFS

```
(*Main program *)
for each s ∈ V
        pred[s] = NULL;
        visited[s] = false;
time = 0
for each s ∈ V
    if (visited[s] == false) DFS(s);
```

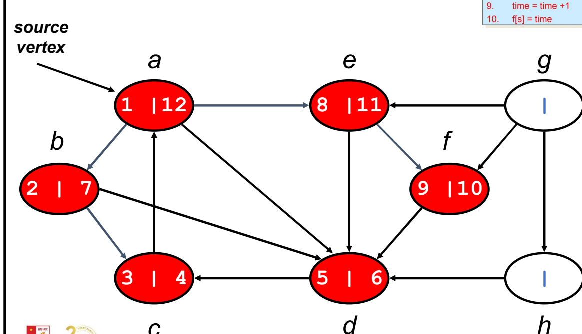
```
DFS(s)
visited[s]= true; //Visit s
time = time +1
d[s] = time
for each v ∈ Adj[s]
        if (visited[v] == false) {
                pred[v] ← s;
                DFS(v);
        }
time = time +1
f[s] = time
```

Example: DFS

```
(*Main program *)
for each s ∈ V
        pred[s] = NULL;
        visited[s] = false;
time = 0
for each s ∈ V
    if (visited[s] == false) DFS(s);
```

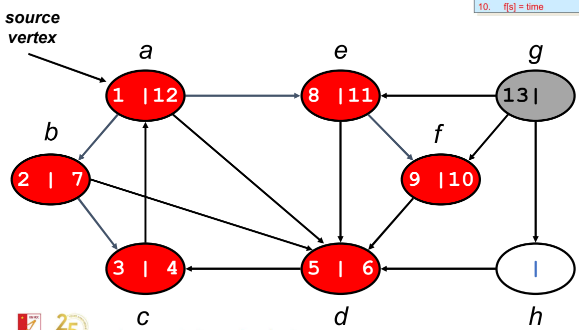
```
DFS(s)
visited[s]= true; //Visit s
time = time +1
d[s] = time
for each v ∈ Adj[s]
        if (visited[v] == false) {
                pred[v] ← s;
                DFS(v);
        }
time = time +1
f[s] = time
```

Example: DFS

```
(*Main program *)
for each s ∈ V
        pred[s] = NULL;
        visited[s] = false;
time = 0
for each s ∈ V
    if (visited[s] == false) DFS(s);
```

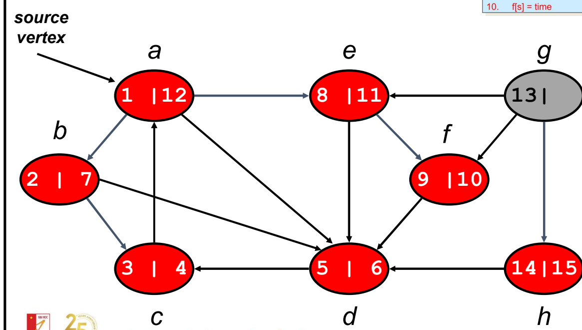
```
DFS(s)
visited[s]= true; //Visit s
time = time +1
d[s] = time
for each v ∈ Adj[s]
        if (visited[v] == false) {
            pred[v] ← s;
            DFS(v);
        }
time = time +1
f[s] = time
```



Example: DFS

```
(*Main program *)
for each s ∈ V
        pred[s] = NULL;
        visited[s] = false;
time = 0
for each s ∈ V
    if (visited[s] == false) DFS(s);
```

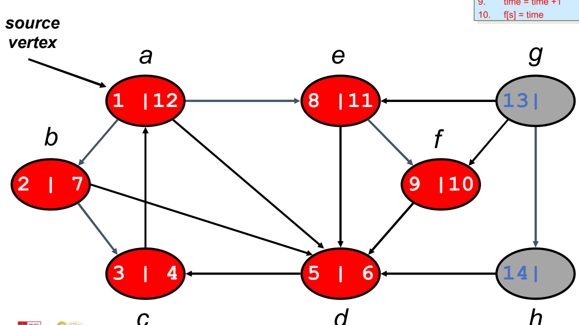
```
DFS(s)
visited[s]= true; //Visit s
time = time +1
d[s] = time
for each v ∈ Adj[s]
        if (visited[v] == false) {
            pred[v] ← s;
            DFS(v);
        }
time = time +1
f[s] = time
```



Example: DFS

```
(*Main program *)
for each s ∈ V
        pred[s] = NULL;
        visited[s] = false;
time = 0
for each s ∈ V
    if (visited[s] == false) DFS(s);
```

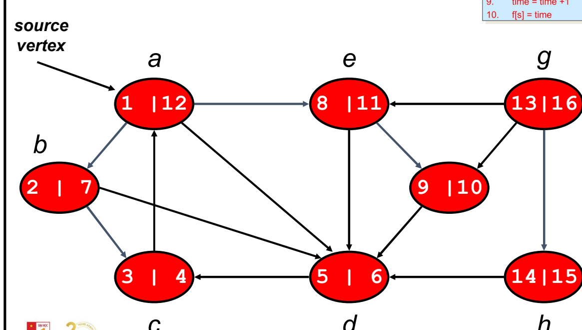
```
DFS(s)
visited[s]= true; //Visit s
time = time +1
d[s] = time
for each v ∈ Adj[s]
        if (visited[v] == false) {
            pred[v] ← s;
            DFS(v);
        }
time = time +1
f[s] = time
```



Example: DFS

```
(*Main program *)
for each s ∈ V
        pred[s] = NULL;
        visited[s] = false;
time = 0
for each s ∈ V
    if (visited[s] == false) DFS(s);
```

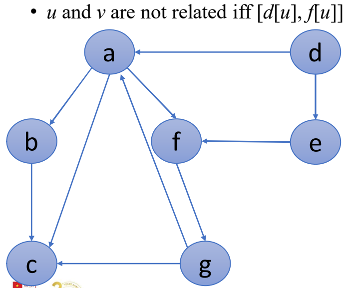
```
DFS(s)
visited[s]= true; //Visit s
time = time +1
d[s] = time
for each v ∈ Adj[s]
        if (visited[v] == false) {
            pred[v] ← s;
            DFS(v);
        }
time = time +1
f[s] = time
```



Lemma of nested intervals

Given directed graph $G = (V, E)$, and arbitrary DFS tree, 2 arbitrary vertices u, v of G. Then

- u is a descendant of v iff $[d[u], f[u]] \subseteq [d[v], f[v]]$
- u is ancestor of v iff $[d[u], f[u]] \supseteq [d[v], f[v]]$
- u and v are not related iff $[d[u], f[u]]$ and $[d[v], f[v]]$ are not intersecting.

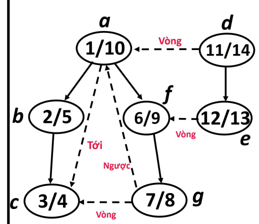


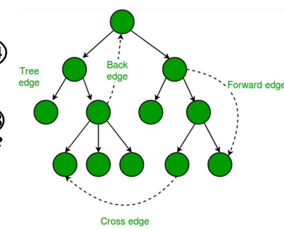
- *Tree edge (cạnh cây)*: the edge whereby from a vertex visits a new vertex (cạnh theo đó từ một đỉnh đến thăm đỉnh mới)
- *Back edge (cạnh ngược)*: going from descendants to ancestors (đi từ con cháu đến tổ tiên)
- *Forward edge (cạnh tới)*: going from ancestor to descendant (đi từ tổ tiên đến con cháu)
- *Cross edge (cạnh vòng)*: edge connecting 2 non-related vertices (giữa hai đỉnh không có họ hàng)

Lemma of nested intervals

Given directed graph $G = (V, E)$, and arbitrary DFS tree, 2 arbitrary vertices u, v of G. Then

- u is a descendant of v iff $[d[u], f[u]] \subseteq [d[v], f[v]]$
- u is ancestor of v iff $[d[u], f[u]] \supseteq [d[v], f[v]]$
- u and v are not related iff $[d[u], f[u]]$ and $[d[v], f[v]]$ are not intersecting.

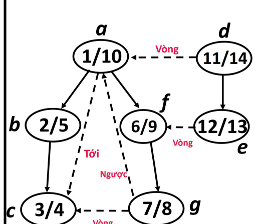


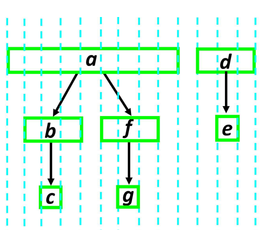


Lemma of nested intervals

Given directed graph $G = (V, E)$, and arbitrary DFS tree, 2 arbitrary vertices u, v of G. Then

- u is a descendant of v iff $[d[u], f[u]] \subseteq [d[v], f[v]]$
- u is ancestor of v iff $[d[u], f[u]] \supseteq [d[v], f[v]]$
- u and v are not related iff $[d[u], f[u]]$ and $[d[v], f[v]]$ are not intersecting.





DFS: Edges classification

- DFS creates a classification of the edges of given graph:
  - *Tree edge (cạnh cây)*: the edge whereby from a vertex visits a new vertex (cạnh theo đó từ một đỉnh đến thăm đỉnh mới)
  - *Back edge (cạnh ngược)*: going from descendants to ancestors (đi từ con cháu đến tổ tiên)
  - *Forward edge (cạnh tới)*: going from ancestor to descendant (đi từ tổ tiên đến con cháu)
  - *Cross edge (cạnh vòng)*: edge connecting 2 non-related vertices (giữa hai đỉnh không có họ hàng)
- **Note:** there are many applications using tree edges and back edges

Some applications

1. **Connectedness of graph**
2. Find the path from *s* to *t*
3. Cycle detection
4. Check strongly connectedness
5. Graph orientation

Some applications

1. Connectedness of graph
2. **Find the path from *s* to *t***
3. Cycle detection
4. Check strongly connectedness
5. Graph orientation

SOICT VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

The problem of connectivity

- **Problem:** Given undirected graph $G = (V,E)$. How many connected components are there in this graph, and each connected component consists of which vertices?
- Answer: Use DFS (BFS) :
  - Each time the function DFS (BFS) is called in the main program, there is one more connected component found in the graph

SOICT VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

The problem of finding the path

The problem of finding the path

- **Input:** Graph $G = (V,E)$ represents by adjacency list, and 2 vertices *s, t*.
- **Output:** Path from vertex *s* to vertex *t*, or confirm there is no path from *s* to *t*.

Algorithm: Perform DFS(s) (or BFS(s)).

- If `pred[t] == NULL` then there does not exist the path, otherwise there is the path from *s* to *t* and the path is:

t ← pred[t] ← pred[pred[t]] ← . . . ← *s*

SOICT VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Some applications

1. Connectedness of graph
2. Find the path from *s* to *t*
3. **Cycle detection**
4. Check strongly connectedness
5. Graph orientation

3. Cycle detection: using DFS

Problem: Given graph $G=(V,E)$. G contains cycle or not?

- ***Theorem:*** *Graph G does not contain cycle if and only if during the DFS execution, we don't not detect the back edge.*
  - **The way to detect the existence of back edge:**
    - 1st method: use lemma of nested intervals
    - 2nd method: mark the state for vertices

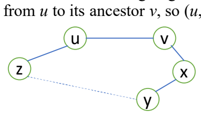
3. Cycle detection: using DFS

Problem: Given graph $G=(V,E)$. G contains cycle or not?

- ***Theorem:*** *Graph G does not contain cycle if and only if during the DFS execution, we don't not detect the back edge.*

Proof:

- ⇒) If G does not contain the cycle then there does not exist back edge. Obviously: the existence of back edge (going from descendants to ancestors) entails the existence of cycle.
- ⇐) We need to prove: if there does not exist back edge, then G *does not contain the cycle*. We prove by contrapositive: G has cycle ⇒ ∃ back edge. Let v be the vertex on the cycle that is the first visited in the DFS execution, and u is the preceding vertex of v on the cycle. When v is visited, the remaining vertices on cycle are all not visited yet. We need to visit all the vertices that are reachable from v before going back v when finishing DFS(v). Thus, the edge $u \rightarrow v$ is traversed from u to its ancestor v, so (u, v) is back edge.



Therefore, DFS can be used to solve the cycle detection problem.

Some applications

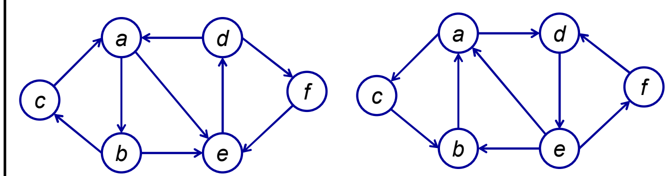
1. Connectedness of graph
2. Find the path from *s* to *t*
3. Cycle detection
4. **Check strongly connectedness**
5. Graph orientation

Check strongly connectedness of directed graph

Problem: Given directed graph $G=(V,E)$. Check if the graph G is strongly connected or not?

Proposition: A directed graph $G = (V, E)$ is strongly connected if and only if there always exists a path from a vertex v to all other vertices and always exists a path from all vertices of $V \setminus \{v\}$ to v.

Algorithm to check strongly connectedness of directed graph



Graph G Graph G^T

Algorithm to check strongly connectedness of directed graph

- Pick an arbitrary vertex $v \in V$.
- Perform DFS(v) on G. If there exists vertex u not visited yet, then G is not strongly connected and the algorithm finishes. Otherwise, the algorithm continues the following step:
  - Perform DFS(v) on $G^T = (V, E^T)$, where E^T is obtained from E by reversing the direction of edges. If exist vertex u not visited, then G is not strongly connected, otherwise G is strongly connected.
- Computation time: $O(|V|+|E|)$

Some applications

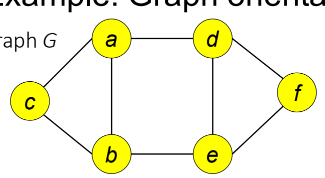
1. Connectedness of graph
2. Find the path from s to t
3. Cycle detection
4. Check strongly connectedness
5. **Graph orientation**

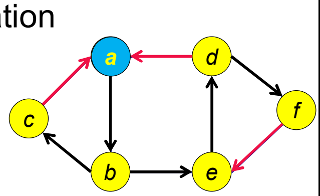
Graph orientation

- **Problem:** Given undirected connected graph $G= (V, E)$. Find the way to orient its edges such that the obtained directed graph is strongly connected or answer that G is non-directional (G là không định hướng được).
- **Orientation algorithm δ:** During the execution of DFS(G), we orient: (1) tree edges of DFS direct from the ancestor to the descendant, (2) back edges of DFS direct from descendant to ancestor. Denote the obtained graph by $G(\delta)$
- **Lemma. G is directional if and only if $G(\delta)$ is strongly connected.**

Example: Graph orientation

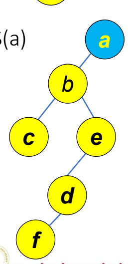
Graph G





Graph $G(\delta)$

DFS(a)



Orientation algorithm δ: During the execution of DFS(G), we orient: (1) tree edges of DFS direct from the ancestor to the descendant, (2) back edges of DFS direct from descendant to ancestor. Denote the obtained graph by $G(\delta)$

Step 2: Applying DFS to check whether G(δ) is strongly connected