Hanoi University of Science and Technology
School of Information and Communications Technology

Discrete Mathematics

Nguyễn Khánh Phương

Department of Computer Science
School of Information and Communication Technology
E-mail: phuongnk@soict.hust.edu.vn

PART 1
COMBINATORIAL THEORY
(Lý thuyết tổ hợp)

PART 2
GRAPH THEORY
(Lý thuyết đồ thị)

Contents of Part 2

Chapter 1. Fundamental concepts
Chapter 2. Graph representation
Chapter 3. Graph Traversal
Chapter 4. Tree and Spanning tree
Chapter 5. Shortest path problem
Chapter 6. Maximum flow problem

Hanoi University of Science and Technology
School of Information and Communications Technology

Chapter 3
Graph traversal

NGUYỄN KHÁNH PHƯƠNG
CS-SOICT-HUST

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

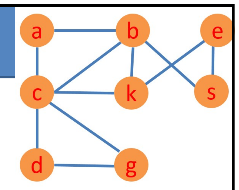
Breadth-first Search (BFS)

(Tìm kiếm theo chiều rộng)

Breadth First Search

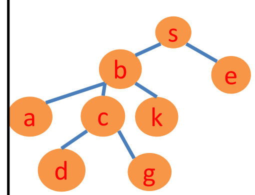
- Given
  - a graph G=(V,E) – set of vertices and edges
  - a distinguished source vertex s
- Breadth first search systematically explores the edges of G to discover every vertex that is reachable from s.
- It also produces a 'breadth first tree' with root *s* that contains all the vertices reachable 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.

Breadth First Search



G=(V, E)

- 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.
- 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*.



BFS(s) tree

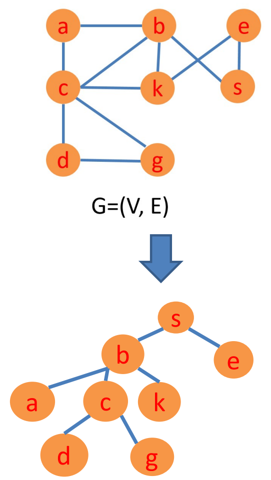
BFS creates a BFS tree containing s as the root and all vertices that is reachable from s

Adjacency list of s

- From s: can go to b and e. Visit them and insert them into queue: Q = {b, e}
- Dequeue (Q): remove b out of Q, then Q = {e}
  - From b: can go to a, c, k, s. But s was visited, so we visit only a, c, k; and insert them into queue: Q = {e, a, c, k}
- Dequeue(Q): remove e out of Q, then Q = {a, c, k}
  - From e: can go to k, s. But all of them were visited.
- Dequeue(Q): remove a out of Q, then Q = {c, k}
  - From a: can go to b, c. But these vertices were all visited.
- Dequeue(Q): remove c out of Q, then Q = {k}.
  - From c: can go to a, b, d, g, k. But a, b, k were visited, so we visit only d, g; and insert them into queue: Q = {k, d, g}
- Dequeue(Q): remove k out of Q, then Q = {d, g}.
  - From k: can go to b, c, e. But these vertices were all visited.
- Dequeue (Q): remove d from Q, then Q = {g}
  - From d: can go to c, g. But these vertices were all visited.
- Dequeue(Q): remove g from Q, then Q = empty
  - From g: can go to d, c. But these vertices were all visited.
- Q is now empty. All vertices of the graph were visited. Algorithm is finished

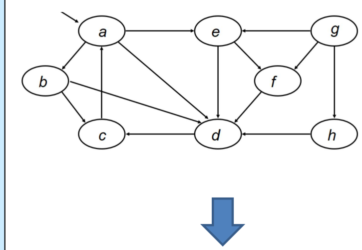
Breadth-first Search

```
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

```
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);
```



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);
```

- Initialize: need $O(|V|)$.
- The loop for
  - Each vertex is inserted into and removed from queue exactly once, each operation needs $O(1)$. So, the total computation time with queue is $O(|V|)$.
  - The adjacency list of each vertex is traversed exactly once. The total length of all adjacency list is $O(|E|)$.
- In total, the computation time of BFS(s) is $O(|V|+|E|)$, linear to the size of adjacency list that represents the graph.

Breadth-first Search

- **Input:** Graph $G = (V, E)$, either undirected or directed graph.

  vertex $s \in V$: source vertex
- **Output:**
  - $d[v]$ = distance (length of shortest path) from s to v, where $v \in V$. $d[v] = \infty$ if v is not reachable from s.
  - $pred[v] = u$ the vertex that is preceding v in the path from s to v with length of d[v].
  - Build BFS tree consisting of root s and all vertices that are reachable from s.

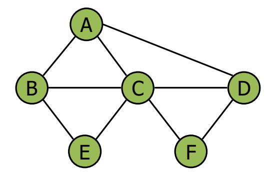
Breadth-first Search

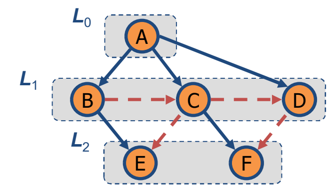
```
BFS(s)
//Breadth first search starts from vertex s
      visited[s] ← 1; //visited
      d[s] ← 0;  pred[s] ← NULL;
      Q ← ∅; enqueue(Q,s);  //Insert s into Q
      while (Q ≠ ∅)
      {
          u ← dequeue(Q);  //Remove u from Q
          for  v ∈Adj[u]
              if (visited[v] == 0)  //not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

(*Main Program*)
      for  s  ∈ V  //initialize
      {
           visited[s] ← 0;  d[s] ← ∞;  pred[s] ← NULL;
      }
      for  s  ∈ V
        if (visited[s]==0)  BFS(s);
```

Q: queue contain all visited vertices
visited[*v*]: mark visit state of vertex v
d[*v*]: distance from to v
pred[*v*]: vertex that is preceding v

BFS

- Sequence of vertices by performing BFS (A):





Depth-first Search (DFS)

(Tìm kiếm theo chiều sâu)

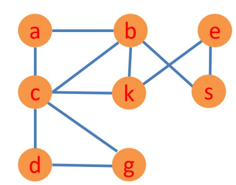
Depth First Search (DFS)

- 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*)
for s ∈ V
    visited[s] ← false
for s ∈ V
    if (visited[s] == false)
        DFS(s)
```

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

DFS(a)



G=(V, E)

16

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 $O(|V|+|E|)$.
- So, DFS and BFS have the same computation time

DFS: If we want to show the path from vertex s to all other vertices of graph

```
(* 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)
```

⬇

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

```
DFS(s)
1.   visited[s]= true; //Visit s
2.   for each v ∈ Adj[s]
3.        if (visited[v] == false) {
4.             pred[v] ← s;
5.             DFS(v);
6.        }
```


DFS: Edges classification

```
(*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
```

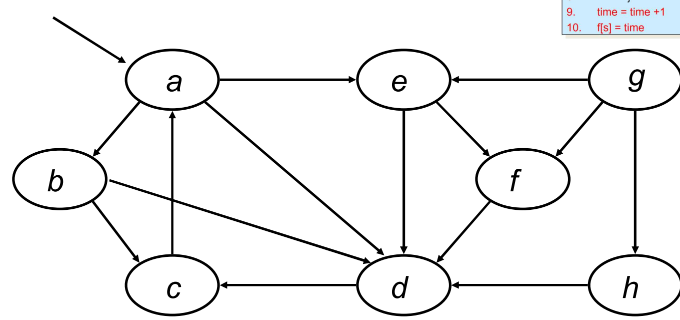
- Also records timestamps for each vertex
  - d[v] when the vertex v is first discovered
  - f[v] when the vertex v is finished

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

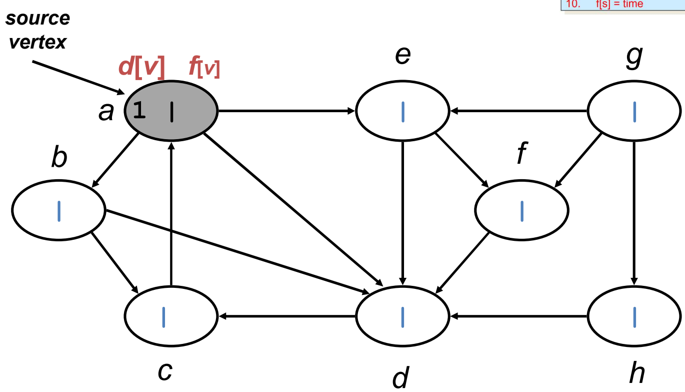


For the operation of the algorithm to be deterministic, assume that we traverse the vertices in the adjacency list of a vertex in lexical order.

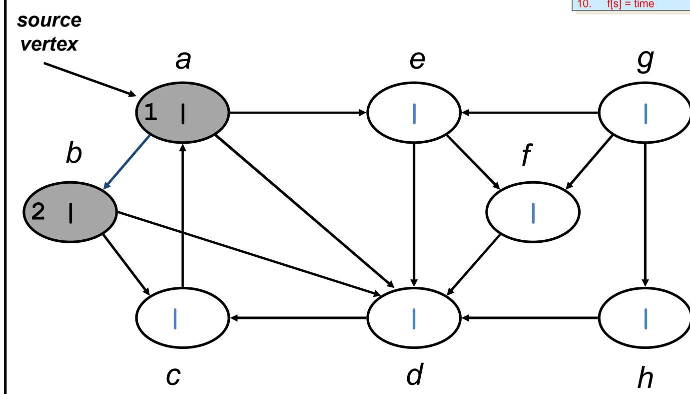
Example: DFS

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

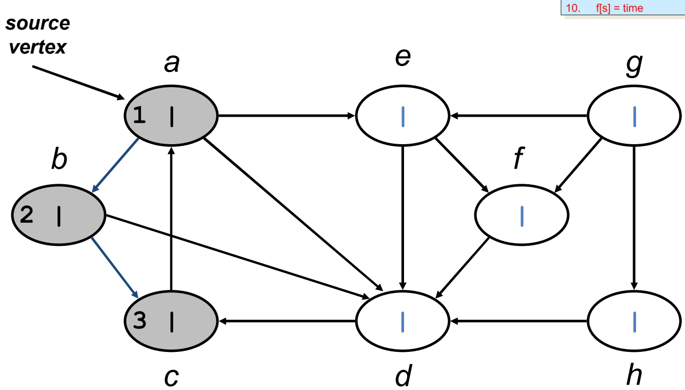
```
DFS(s)
    visited[s]= true; //Visit s
    time = time +1
    d[s] = time
    for each v ∈ Adj[s]
        if (visited[v] == false) {
            pred[v] ← s;
            DFS(v);
        }
    time = time +1
    f[s] = time
```



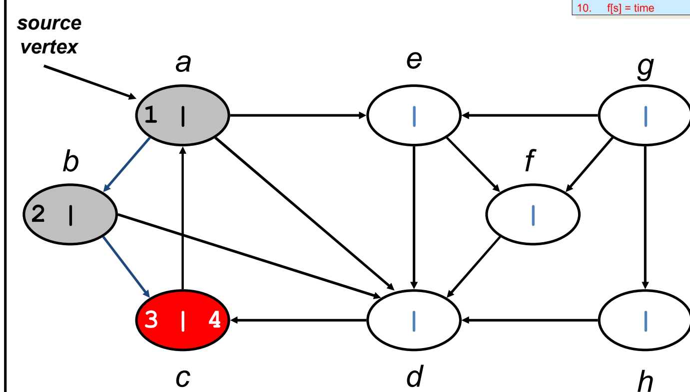
Example: DFS
(*Main program *)
for each s ∈ V
pred[s] = NULL;
visited[s] = false;
time = 0
for each s ∈ V
if (visited[s] == false) DFS(s);
DFS(s)
visited[s]= true; //Visit s
time = time +1
d[s] = time
for each v ∈ Adj[s]
if (visited[v] == false) {
pred[v] ← s;
DFS(v);
}
time = time +1
f[s] = time



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



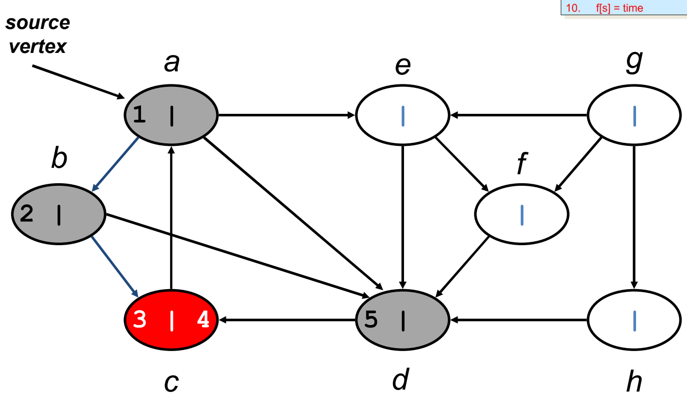
Example: DFS
(*Main program *)
for each s ∈ V
pred[s] = NULL;
visited[s] = false;
time = 0
for each s ∈ V
if (visited[s] == false) DFS(s);
DFS(s)
visited[s]= true; //Visit s
time = time +1
d[s] = time
for each v ∈ Adj[s]
if (visited[v] == false) {
pred[v] ← s;
DFS(v);
}
time = time +1
f[s] = time



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);
```

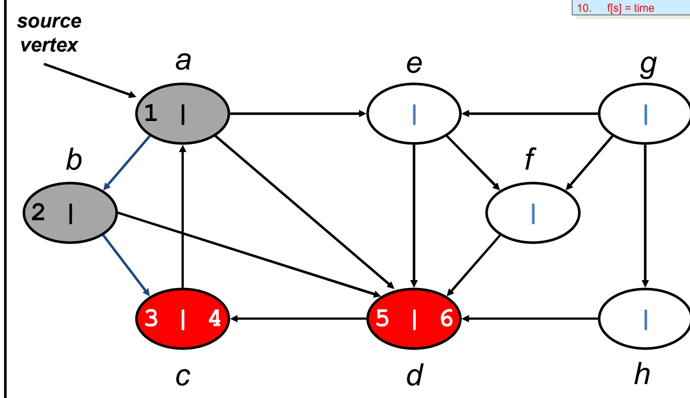
```
DFS(s)
    visited[s]= true; //Visit s
    time = time +1
    d[s] = time
    for each v ∈ Adj[s]
        if (visited[v] == false) {
            pred[v] ← s;
            DFS(v);
        }
    time = time +1
    f[s] = time
```



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);
```

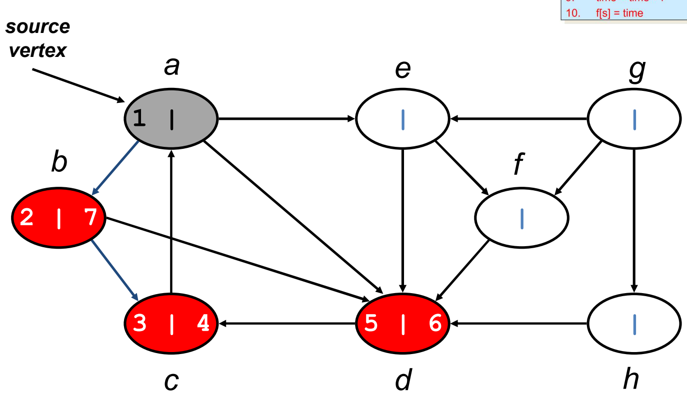
```
DFS(s)
    visited[s]= true; //Visit s
    time = time +1
    d[s] = time
    for each v ∈ Adj[s]
        if (visited[v] == false) {
            pred[v] ← s;
            DFS(v);
        }
    time = time +1
    f[s] = time
```



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);
```

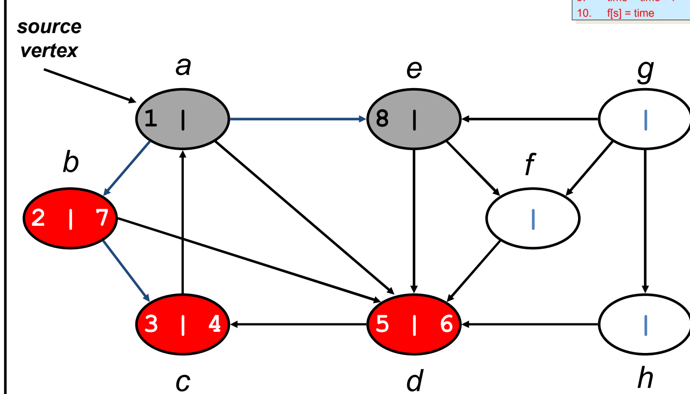
```
DFS(s)
    visited[s]= true; //Visit s
    time = time +1
    d[s] = time
    for each v ∈ Adj[s]
        if (visited[v] == false) {
            pred[v] ← s;
            DFS(v);
        }
    time = time +1
    f[s] = time
```



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);
```

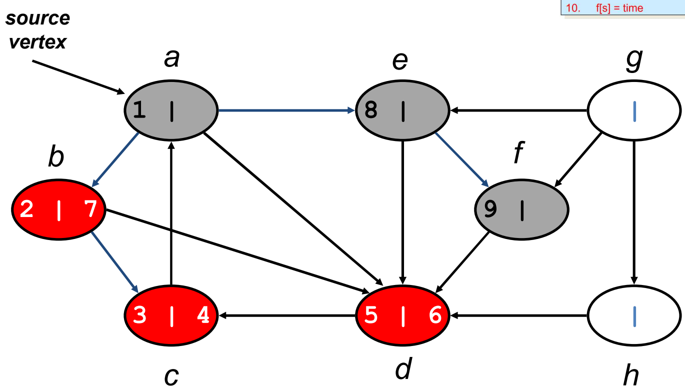
```
DFS(s)
    visited[s]= true; //Visit s
    time = time +1
    d[s] = time
    for each v ∈ Adj[s]
        if (visited[v] == false) {
            pred[v] ← s;
            DFS(v);
        }
    time = time +1
    f[s] = time
```



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);
```

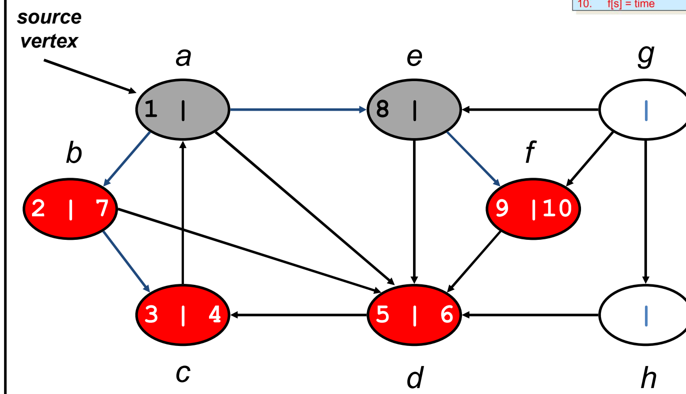
```
DFS(s)
visited[s]= true; //Visit s
time = time +1
d[s] = time
for each v ∈ Adj[s]
        if (visited[v] == false) {
            pred[v] ← s;
            DFS(v);
        }
time = time +1
f[s] = time
```



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);
```

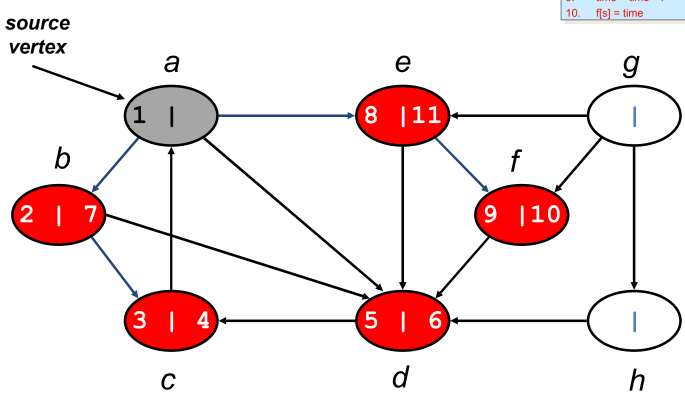
```
DFS(s)
visited[s]= true; //Visit s
time = time +1
d[s] = time
for each v ∈ Adj[s]
        if (visited[v] == false) {
            pred[v] ← s;
            DFS(v);
        }
time = time +1
f[s] = time
```

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);
```

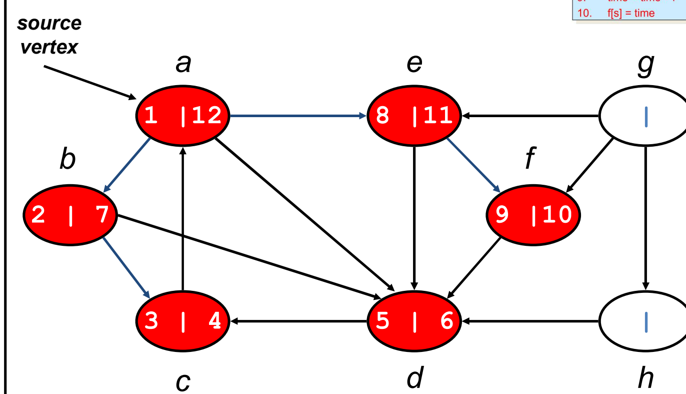
```
DFS(s)
visited[s]= true; //Visit s
time = time +1
d[s] = time
for each v ∈ Adj[s]
        if (visited[v] == false) {
            pred[v] ← s;
            DFS(v);
        }
time = time +1
f[s] = time
```

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);
```

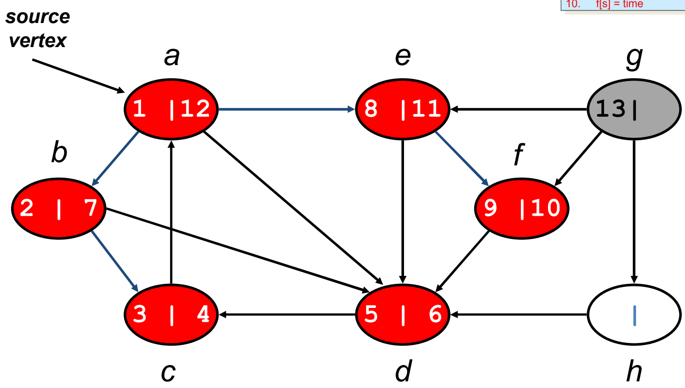
```
DFS(s)
visited[s]= true; //Visit s
time = time +1
d[s] = time
for each v ∈ Adj[s]
        if (visited[v] == false) {
            pred[v] ← s;
            DFS(v);
        }
time = time +1
f[s] = time
```

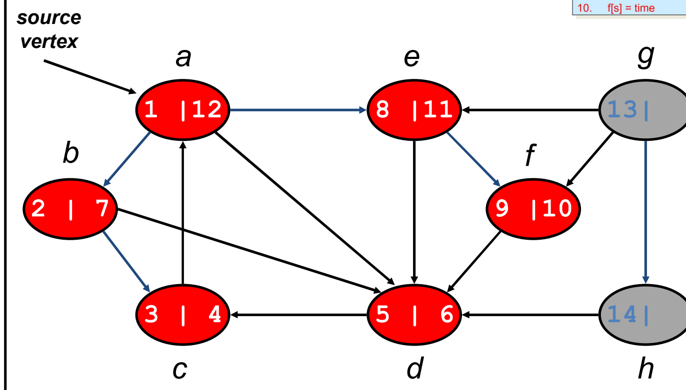
Example: DFS

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

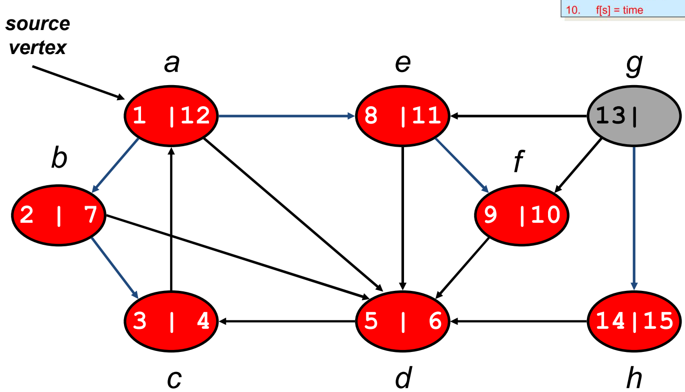
```
DFS(s)
    visited[s]= true; //Visit s
    time = time +1
    d[s] = time
    for each v ∈ Adj[s]
        if (visited[v] == false) {
            pred[v] ← s;
            DFS(v);
        }
    time = time +1
    f[s] = time
```



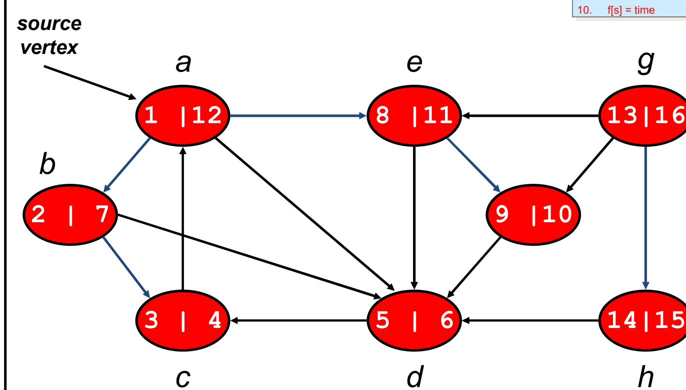
Example: DFS
(*Main program *)
for each s ∈ V
pred[s] = NULL;
visited[s] = false;
time = 0
for each s ∈ V
if (visited[s] == false) DFS(s);
DFS(s)
visited[s]= true; //Visit s
time = time +1
d[s] = time
for each v ∈ Adj[s]
if (visited[v] == false) {
pred[v] ← s;
DFS(v);
}
time = time +1
f[s] = time



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



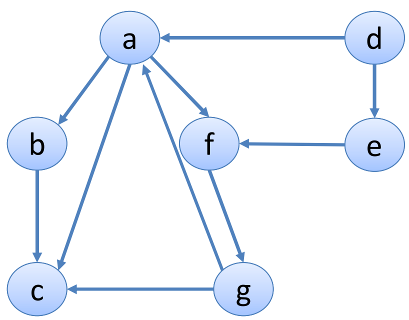
Example: DFS
(*Main program *)
for each s ∈ V
pred[s] = NULL;
visited[s] = false;
time = 0
for each s ∈ V
if (visited[s] == false) DFS(s);
DFS(s)
visited[s]= true; //Visit s
time = time +1
d[s] = time
for each v ∈ Adj[s]
if (visited[v] == false) {
pred[v] ← s;
DFS(v);
}
time = time +1
f[s] = time



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.



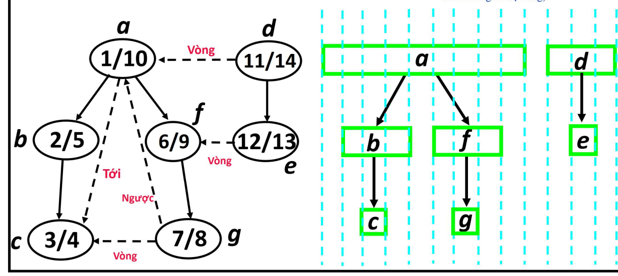
- *Tree edge (cạnh cây)*: the edge whereby from a vertex visits a new vertex (cạnh theo đó từ một đỉnh đến thăm đỉnh mới)
- *Back edge (cạnh ngược)*: going from descendants to ancestors (đi từ con cháu đến tổ tiên)
- *Forward edge (cạnh tới)*: going from ancestor to descendant (đi từ tổ tiên đến con 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.

- *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)

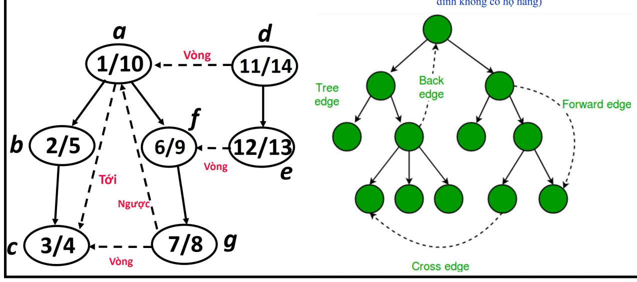


Lemma of nested intervals

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.

- *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)



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

Example

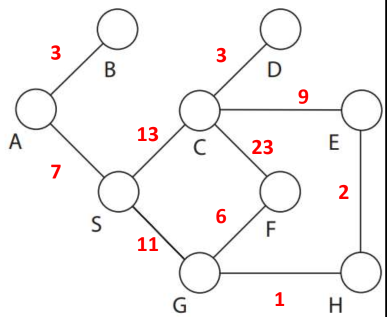
Given graph G as shown in the figure below

Question 1: Represent G by using

1. Adjacency matrix
2. Weight matrix
3. Adjacency list

Question 2: Given source vertex S

Draw trees BFS(S) and DFS(S)



Some applications of DFS

1. Connectedness of graph

2. Find the path from s to t

3. Cycle detection

4. Check strongly connectedness

5. Graph orientation

6. Topo order

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

DFS to solve the problem of connectivity

```
(* Main Program*)
for s ∈ V
    visited[s] ← false
for s ∈ V
    if (visited[s] == false)
        DFS(s)
```

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

```
(* Main Program*)
for u ∈ V
    id[u] ← 0;
cnt ← 0;  //cnt – number of
    connected components
for u ∈ V
    if (id[u] == 0){
        cnt ← cnt +1;
        DFS(u) ;
    }
```

```
DFS(u)
    id[u] ← cnt;
    for each v ∈ Adj[u]
        if (id[v] == 0)
            DFS(v);
```

BFS to solve the problem of connectivity

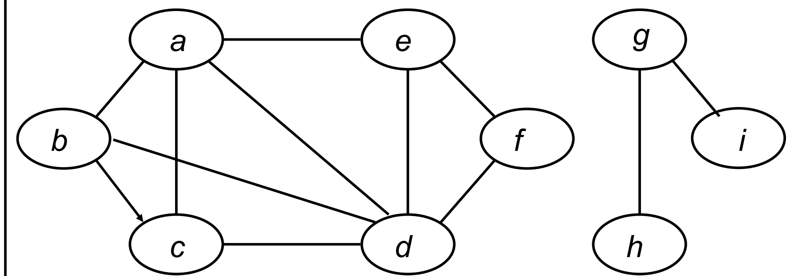
```
BFS(s)
// Breadth first search starts from vertex s
    visited[s] ← 1; //visited
    Q ← ∅; enqueue(Q,s); // insert s into Q
    while (Q ≠ ∅)
    {
        u ← dequeue(Q);  // Remove u from Q
        for  v ∈Adj[u]
            if (visited[v] == 0) //not visited yet
            {
              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

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

```
(* Main Program*)
for s ∈ V
    id[s] ← 0
cnt ← 0  //cnt – number of connected components
for s ∈ V
    if (id[s] == 0){
        cnt ← cnt +1
        BFS(s)
    }
```

Example

Using the DFS algorithm determine the number of connected components of the following graph, and show which vertices each connected component consists of



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.

Some applications of DFS

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

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

DFS to solve the problem of finding the path

```
(* Main Program*)
for u ∈ V {
    visited[u] ← false
    pred[u] ← NULL }
DFS(s)
```

```
DFS(s)
visited[s] ← true //visit s
for each v ∈ Adj[s]
    if (visited[v] == false) {
        pred[v] ← s
        DFS(v)
    }
```

BFS to solve the problem of finding the path

```
(* Main Program*)
for u ∈ V {
    visited[u] ← false
    pred[u] ← NULL }
BFS(s)
```

```
BFS(s)
visited[s] ← true;  //visit s
Q ← ∅; enqueue(Q,s); // Insert s into Q
while (Q ≠ ∅)
{
    u ← dequeue(Q);  // Remove u from Q
    for  v ∈Adj[u]
        if (visited[v] == false)   //not visited yet
        {
            visited[v] ← true;  //visited
            pred[v] ← u;
            enqueue(Q,v)   //Insert v into Q
        }
}
```

Some applications of DFS

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

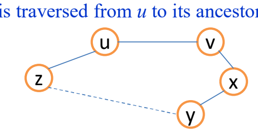
3. Cycle detection: using DFS

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 $u \to v$ 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.

3. Cycle detection: using DFS

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

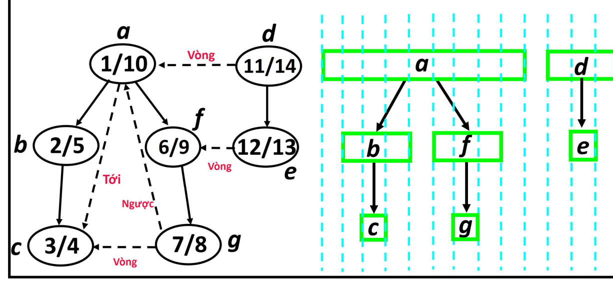
- ***Theorem:*** *Graph G does not contain cycle if and only if during the DFS execution, we don't not detect the back edge.*
  - **The way to detect the existence of back edge:**
    - 1st method: use lemma of nested intervals
    - 2nd method: mark the state for vertices

DFS: Lemma of nested intervals (Method 1)

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)*: cạnh theo đó từ một đỉnh đến thăm đỉnh mới
- *Back edge (cạnh ngược)*: đi từ con cháu đến tổ tiên
- *Forward edge (cạnh tới)*: đi từ tổ tiên đến con cháu
- *Cross edge (cạnh vòng)*: giữa hai đỉnh không có họ hàng



DFS and cycle detection: Method 2

- ***How to modify so we can detect the back edge → the cycle***

(* Main Program*)	DFS(u)
1. **for** $u \in V$	1. *visited*[*u*] ← true //visit *u*
2. visited[u] ← false	**2. for** each $v \in Adj[u]$
3. **for** $u \in V$	3. **if** (*visited*[*v*] == false)
4. **if** (*visited*[*u*] == false)	4. DFS(v)
5. DFS(u)	

Currently, each vertex has 2 states: `visited =false` or `true`

DFS and cycle detection: Method 2

- Instead of using only two states: not visited/visited (`visited = 0/1`) for each vertex → Need to modify: each vertex has one of 3 states:
  - Not visited: `visited = 0`
  - Visited (but not finish traversal yet): `visited = 1`
  - Finish traversal: `visited = 2`
- The state of vertex is changed according to the following rule:
  - Initialize: each vertex `v` is not visited `visited[v] = 0`
  - Visit `v`: `visited[v] = 1` (become visited but not finish traversal yet).
  - When all adjacency vertices of `v` are visited, vertex `v` is called as finish traversal `visited[v] = 2`

(* Main Program*)	DFS(u)
1. **for** $u \in V$	1. *visited*[*u*] ← true //visit *u*
2. visited[u] ← false	**2. for** each $v \in Adj[u]$
3. **for** $u \in V$	3. **if** (*visited*[*v*] == false)
4. **if** (*visited*[*u*] == false)	4. DFS(v)
5. DFS(u)	

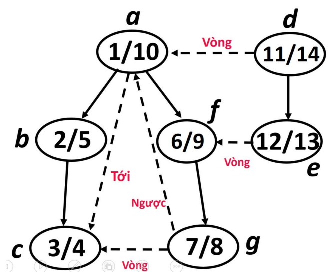
DFS: Edges classification

DFS yields edges classification of the graph:

- When we traverse edge $e = (u, v)$ from vertex u, based on the value of `visited[v]`, we could know the type of edge e :
  1) `visited[v] = 0`: so e is tree edge
  2) `visited[v] = 1`: so e is back edge
  3) `visited[v] = 2`: so e is either forward edge or cross edge

– *Tree edge (cạnh cây)*: cạnh theo đó từ một đỉnh đến thăm đỉnh mới
– *Back edge (cạnh ngược)*: đi từ con cháu đến tổ tiên
– *Forward edge (cạnh tới)*: đi từ tổ tiên đến con cháu
– *Cross edge (cạnh vòng)*: giữa hai đỉnh không có họ hàng

```
DFS(u)
visited[u] ← true  //visit u
for each v ∈ Adj[u]
      if (visited[v] == false)
            DFS(v)
```



DFS and cycle detection: Method 2

- ***How to modify so we can detect the back edge → the cycle***

```
(* Main Program*)
for u ∈ V
    visited[u] ← false
for u ∈ V
    if (visited[u] == false)
        DFS(u)
```

```
DFS(u)
visited[u] ← true  //visit u
for each v ∈ Adj[u]
      if (visited[v] == false)
            DFS(v)
```

When we traverse edge $e = (u, v)$ from vertex u, based on the value of `visited[v]`, we could know the type of edge e :
1) `visited[v] = 0`: so e is tree edge
2) `visited[v] = 1`: so e is back edge

```
(* Main Program*)
for u ∈ V
    visited[u] ← 0
for u ∈ V
    if (visited[u] == 0)
        DFS(u)
```

```
DFS(u)
visited[u] ← 1  //visit u
for each v ∈ Adj[u]
      if (visited[v] == 1)
      {
         print ("Graph has cycle");
         exit();
      }
      else if(visited[v]==0)   DFS(v)
visited[u] ← 2 //u is at state of finish traversal
```

DFS and cycle detection

- **Question:** What is the computation time?

Answer: It is the computation time of *DFS*: $O(|V|+|E|)$.

- **Question:** If G is undirected graph, can we evaluate the computation time more precisely?

Answer: Computation time is $O(|V|)$, because:

– In the forest (graph does not contain cycle): $|E| \leq |V| - 1$

– Thus, if graph consists of $|V|$ edges, then it for sure contain the cycle, and algorithm finishes.

Proposition. An undirected simple graph with *n* vertices and *n* edges certainly contains a cycle.

Some applications of DFS

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

Check strongly connectedness of directed graph

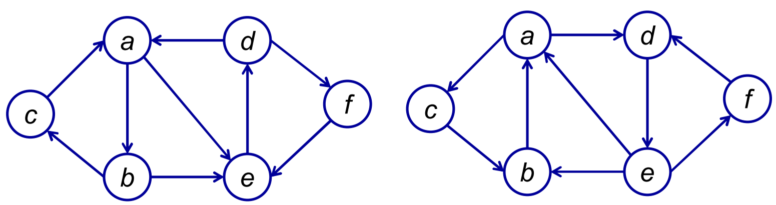
Problem: Given directed graph $G=(V,E)$. Check if the graph G is strongly connected or not?

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

- 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: $O(|V|+|E|)$

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.

Question: If graph G is represented by adjacency matrix A

➔ How to obtain graph G^T from matrix A ?

Reversal graph (transpose graph)

- Given the directed graph $G=(V,E)$. We call reversal graph (transpose graph) of graph G is the directed graph $G^T = (V, E^T)$, where $E^T = \{(u, v): (v, u) \in E\}$, it means edge set E^T is obtained from edge set E by reversing direction of all edges.
- It is easy to see if A is the adjacency matrix of graph G, then the transpose matrix A^T is the adjacency matrix of G^T (this explains the name of the transpose graph).

Some applications of DFS

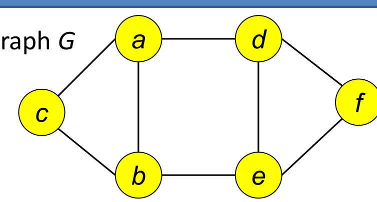
1. Connectedness of graph
2. Find the path from s to t
3. Cycle detection
4. Check strongly connectedness
5. **Graph orientation**
6. Topo order

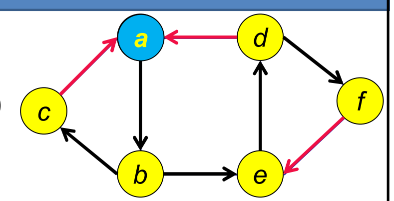
Graph orientation (Định hướng đồ thị)

- **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

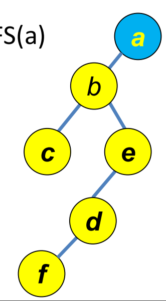
Graph G





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 $G(\delta)$ is strongly connected

Some applications of DFS

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

6.Topological order (Sắp xếp topo)

A typical application of topological order is to plan a sequence of jobs:

- Each vertex of the graph represents one job to be performed.
- The jobs are interdependent. So some jobs cannot be done before others are completed.

Example:

- IT3302 (C basic) is a prerequisite course when registering IT3312 (C advanced)

Given a directed graph with no cycle (DAG - directed acyclic graph), find an order of jobs need to be done so that the constraints on the prerequisite (shown by the direction of edges on the graph) are preserved.

Note: there might exist more than one executable schedule

70

6.Topological order (Sắp xếp topo)

The topological order of a directed graph is an order of vertices such that for all directed edges (u, v) in the graph, then u is always previous v in this order.

The algorithm for finding topological order is called **topological sorting algorithm**.

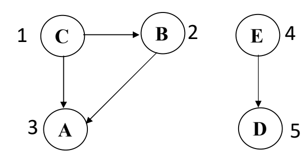
Topological order exists when and only when the directed graph has no cycle (DAG - directed acyclic graph). A DAG always has at least one topological order.

71

6.Topological order (Sắp xếp topo)

- **Problem:** Given directed acyclic graph $G=(V, E)$. Find the way to order vertices of G such that if there is a directed edge (u,v) in G, then u is always previous v in this order (in other words, find the way to index vertices such that edges of graph is always directed from vertex with smaller index to vertex with larger index).

Example:



2 methods:

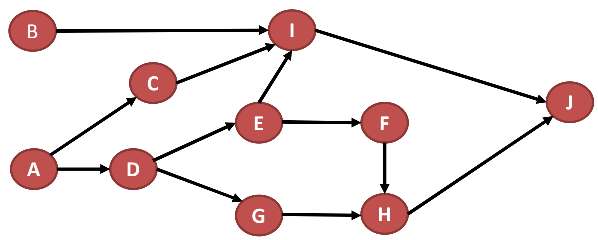
- DFS algorithm
- BFS with modification

72

Example: Renovating office for rent

Công đoạn	Các công đoạn trước	Thời gian(Tuần)
A: Chuẩn bị thiết kế ban đầu	-	3
B: Tìm kiếm khách hàng	-	5
C: Thiết kế tờ rơi	A	3
D: Chuẩn bị bản thiết kế cuối cùng	A	8
E: Xin giấy phép sửa chữa	D	2
F: Nhận tiền đầu tư từ ngân hàng	E	3
G: Lựa chọn khách hàng	D	4
H: Xây dựng sửa chữa	G, F	17
I: Hoàn thiện hợp đồng thuê mặt bằng	B, C, E	13
J: Khách hàng chuyển vào	I, H	2

Give a sequence of tasks to be executed



(1) Apply DFS

The algorithm can be briefly described as follows:

- Performing DFS (G), when each vertex is visited finish, we put it at the top of the linked list (which means that the later the vertices that finish visit will be closer to the top of the list).
- When DFS(G) terminates, the obtained linked list gives us the order of tasks to be executed.

Topological sorting algorithm: apply DFS

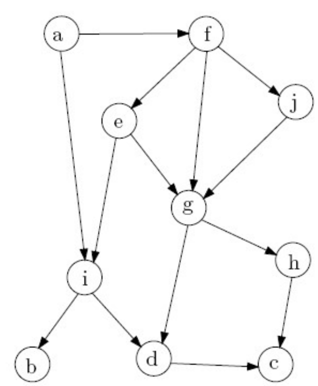
```
(* Main Program*)
1. for s ∈ V
2.     visited[s] ← false
3. for s ∈ V
4.     if (visited[s] == false)
5.         DFS(s)
```

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

```
TopoSort(G)
   1. for s∈V visited[s] = false;    // initialize
   2. L = new(linked_list);          // initialize the empty linked list L
   3. for s∈V
   4.     if (visited[s] == false) DFS(s);
   5. return L;                      // L gives the order
DFS (s) {                            // starting searching from s
   1. visited[s] = true;             // mark s as visited
   2. for v∈Adj(s)
   3.     if (visited[v] == FALSE) DFS(v);
   4. Insert s at the beginning of list L    // s is visited finish
}
```

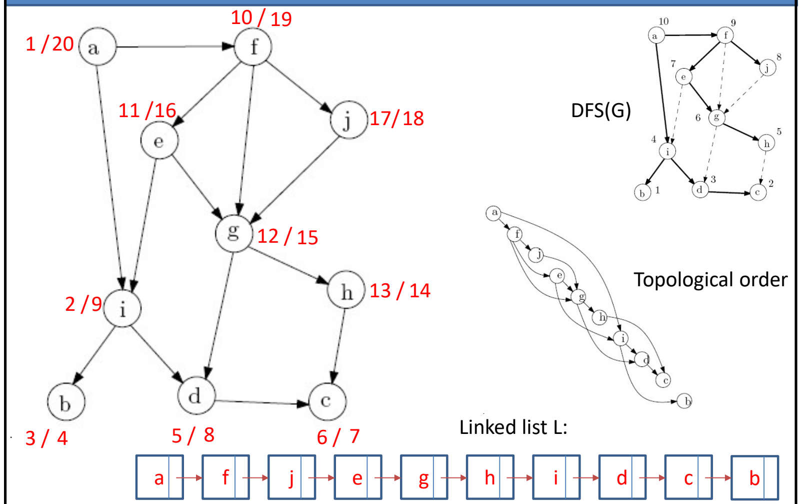
Computation time of TopoSort(G) is $O(|V|+|E|)$.

Example 1: Topological order by using DFS

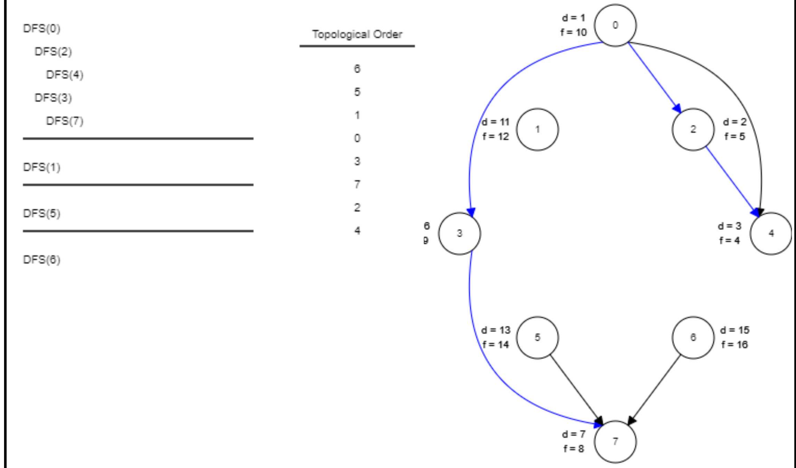


Graph G

Example 1: Topological order by using DFS



Example 2: Topological order by using DFS



(2) BFS with modification

Another algorithm for topological sorting is based on the following clause:

Proposition. Suppose G is a directed acyclic graph. Then

1) All subgraphs of H of G are directed acyclic graph.

2) It is always possible to find a vertex with in-order degree of zero.

Topological order: BFS with modification

From the proposition, we have the algorithm:

- First, find vertices with in-order degree of zero. Obviously, we can number them in an arbitrary order starting with 1.
- Next, delete all vertices that are numbered from the graph and all edges going out of them, we then obtain a new graph with no cycles. And the process is repeated with this new graph.
- That process will be continued until all vertices of the graph are numbered.

Topological order: BFS with modification

for $v \in V$
    Calculate InDegree[v] – indegree of vertex v;
Q = queue contains all vertices of indegree = 0;
num=0;
while $Q \neq \varnothing$
    v = Dequeue(Q); num=num+1;
    Number vertex v by num;
    for $u \in$ Adj(v)
        InDegree[u]=InDegree[u] -1;
        if (InDegree[u]==0)
            Enqueue(Q,u);

Computation time: O(|V+E|)

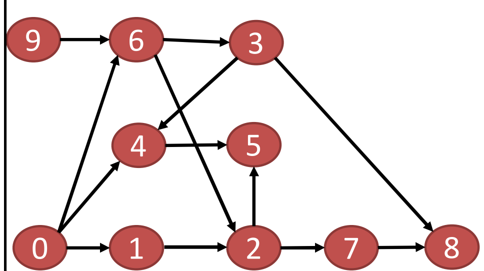
81

Topological sorting algorithm

- Obviously, in the initial step we must traverse through all the edges of the graph when calculating the in-dgree of the vertices, so that we take $O(|E|)$ operations. Next, each time indexing a vertex, in order to perform the removal of this indexed vertex along with the arcs going out of it, we traverse through all these edges. In order to index all the vertices of the graph we will have to traverse through all the edges of the graph again.
- Therefore, the running time: $O(|E|)$.

82

Topological order: BFS with modification



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

Indegree

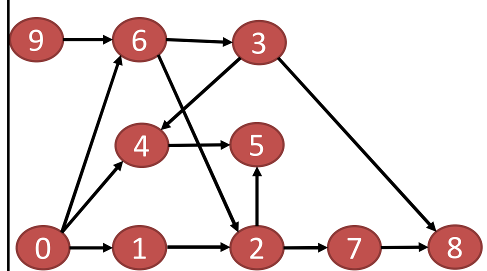
0	0
1	1
2	2
3	1
4	2
5	2
6	2
7	1
8	2
9	0

Q={0, 9}

Output:

83

Topological order: BFS with modification



1

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

Indegree

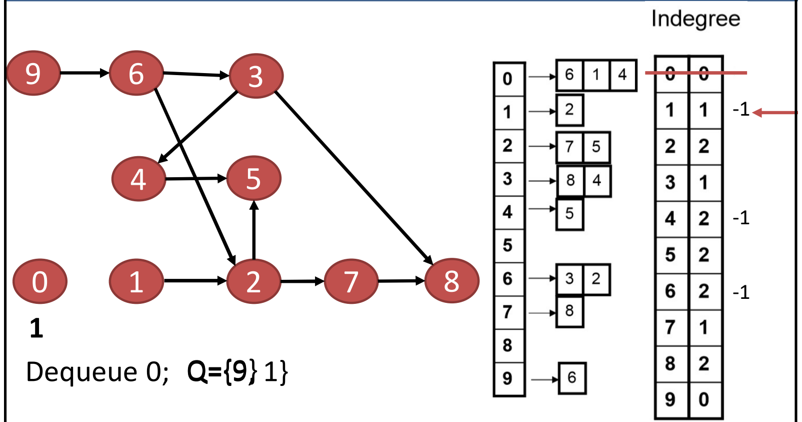
~~0~~	~~0~~	
1	1	-1
2	2	
3	1	
4	2	-1
5	2	
6	2	-1
7	1	
8	2	
9	0	

Dequeue 0; Q={9}

Output: 0

84

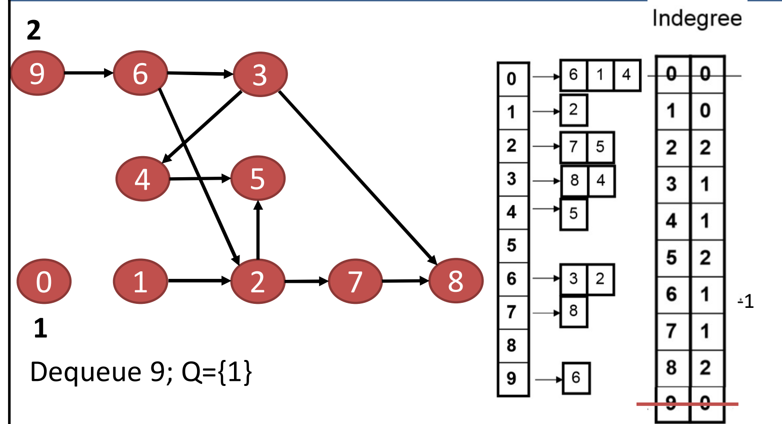
Topological order: BFS with modification



Dequeue 0;   Q={9} 1}

Output: 0

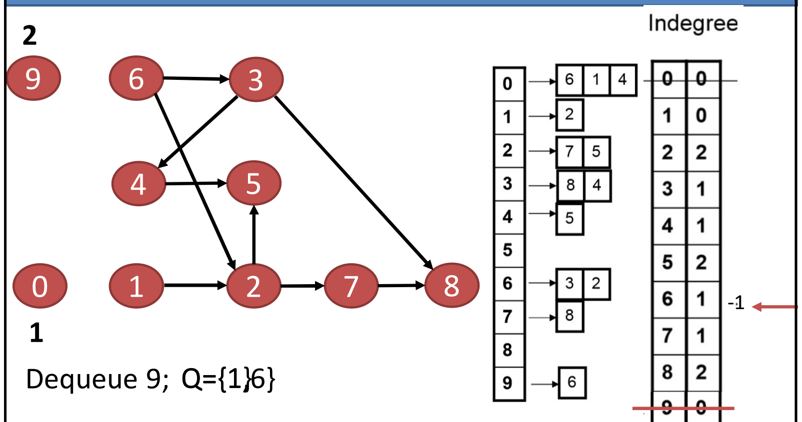
Topological order: BFS with modification



Dequeue 9; Q={1}

Output: 0 9

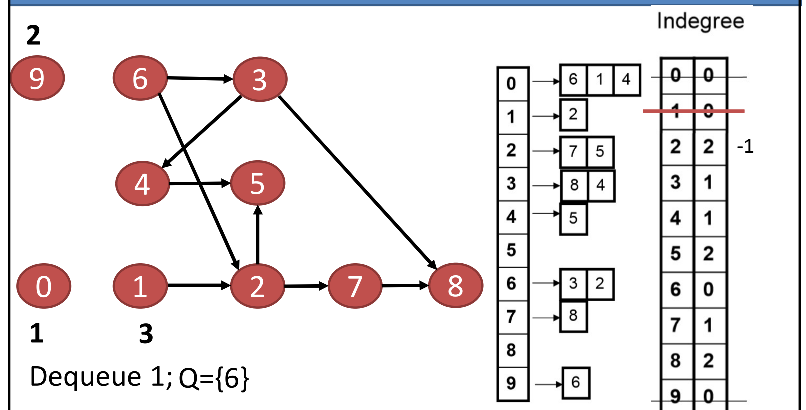
Topological order: BFS with modification



Dequeue 9; Q={1}6}

Output: 0 9

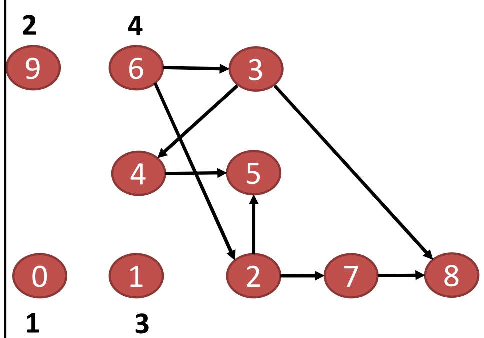
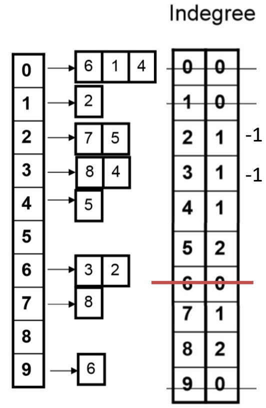
Topological order: BFS with modification



Dequeue 1; Q={6}

Output: 0 9 1

Topological order: BFS with modification

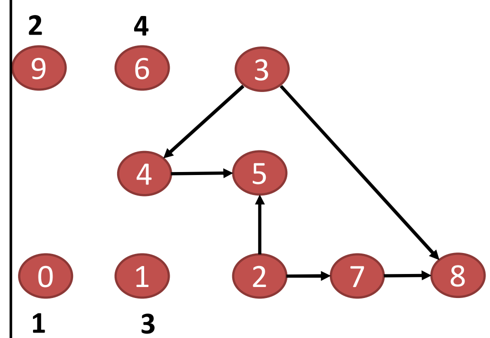
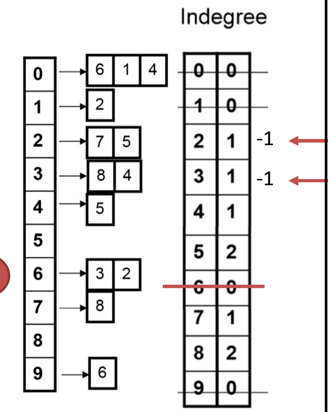



Dequeue 6; Q={}

Output: 0 9 1 6

89

Topological order: BFS with modification

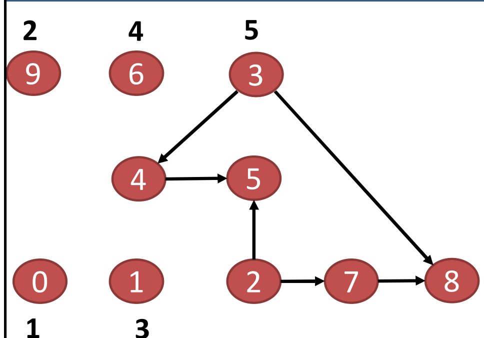
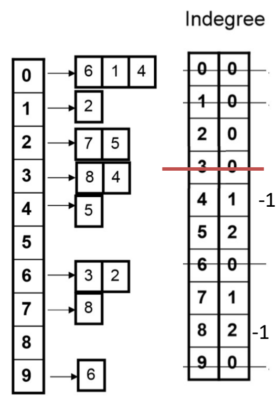



Dequeue 6; **Q={**3,2}

Output: 0 9 1 6

90

Topological order: BFS with modification

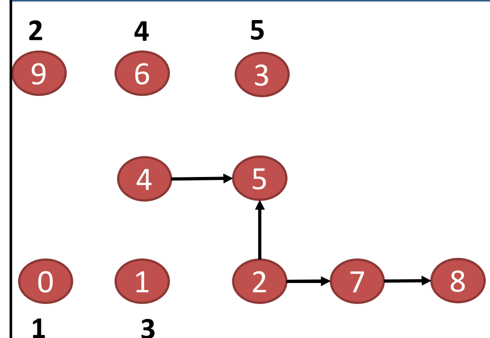
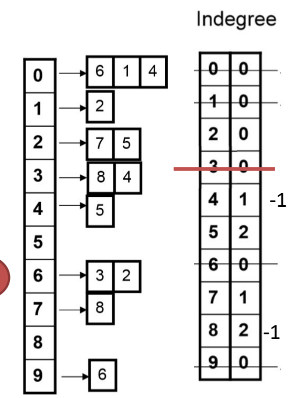



Dequeue 3; Q={2}

Output: 0 9 1 6 3

91

Topological order: BFS with modification

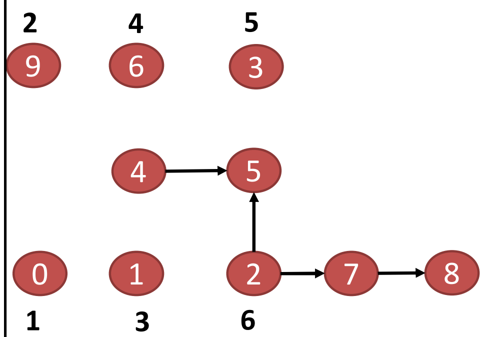
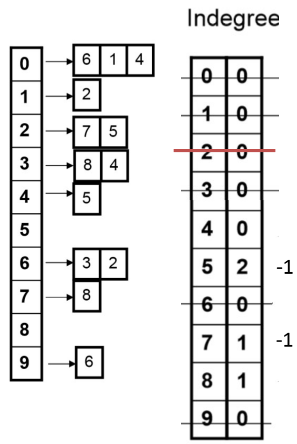



Dequeue 3; **Q={2**} 4}

Output: 0 9 1 6 3

92

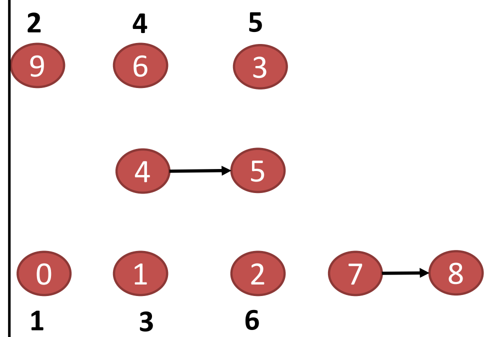
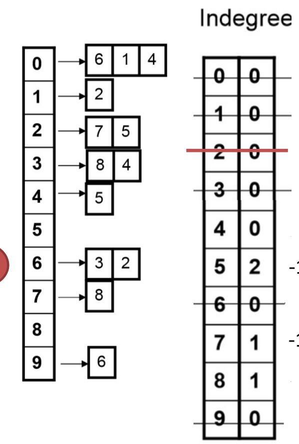
Topological order: BFS with modification

Dequeue 2; Q={4}

Output: 0 9 1 6 3 2

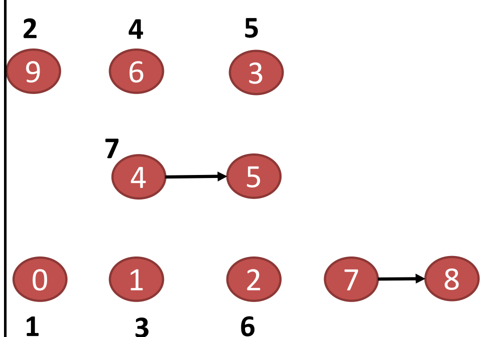
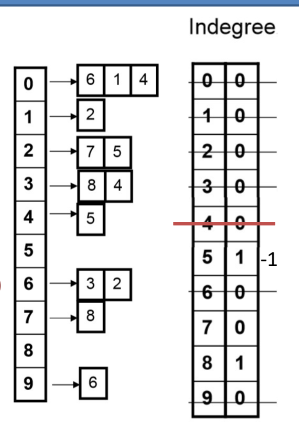
Topological order: BFS with modification

Dequeue 2; Q={4} 7}

Output: 0 9 1 6 3 2

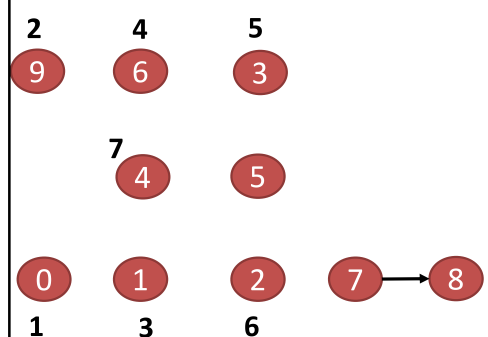
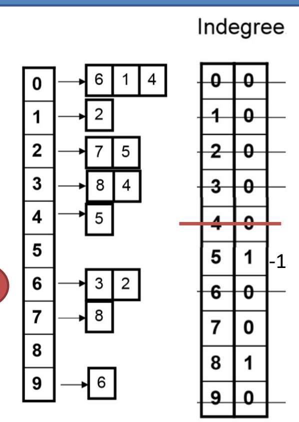
Topological order: BFS with modification

Dequeue 4; Q={7}

Output: 0 9 1 6 3 2 4

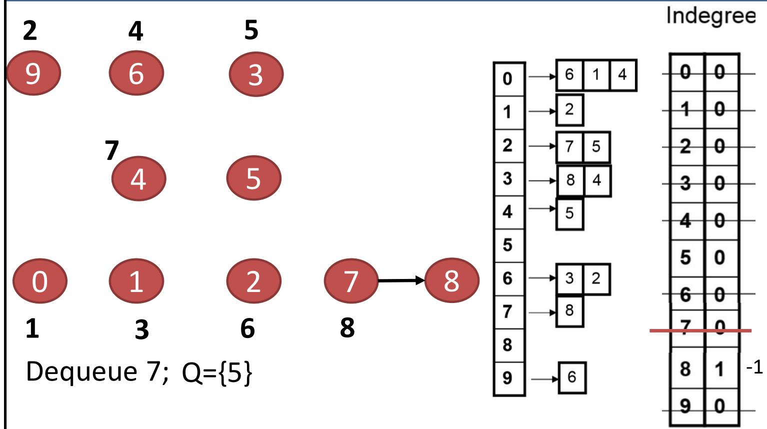
Topological order: BFS with modification

Dequeue 4; Q={7} 5}

Output: 0 9 1 6 3 2 4

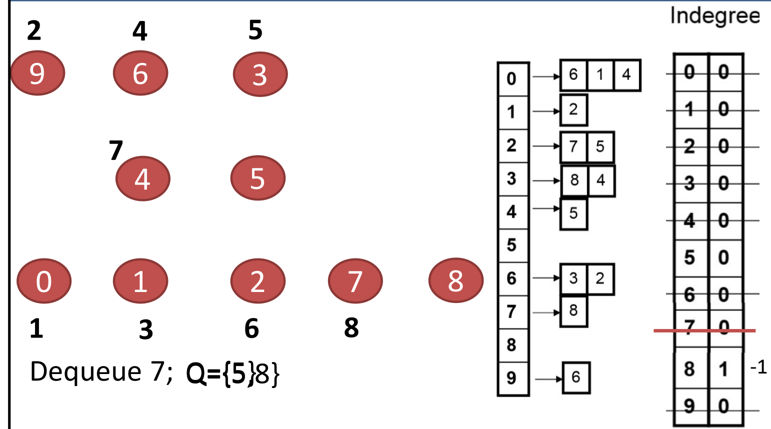
Topological order: BFS with modification



Dequeue 7; Q={5}

Output: 0 9 1 6 3 2 4 7

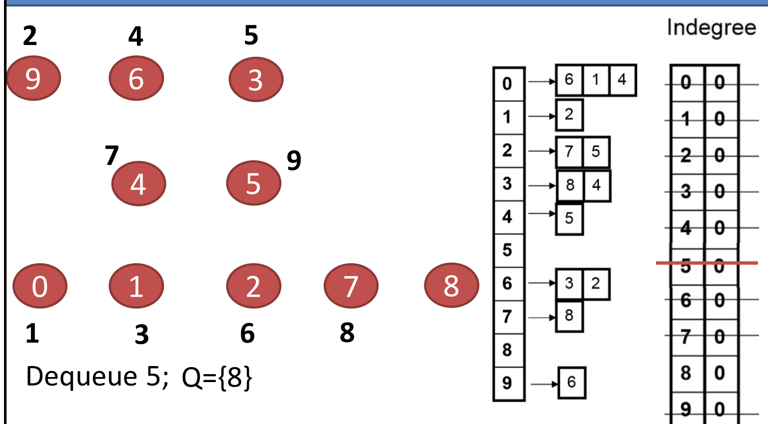
Topological order: BFS with modification



Dequeue 7; Q={5}8}

Output: 0 9 1 6 3 2 4 7

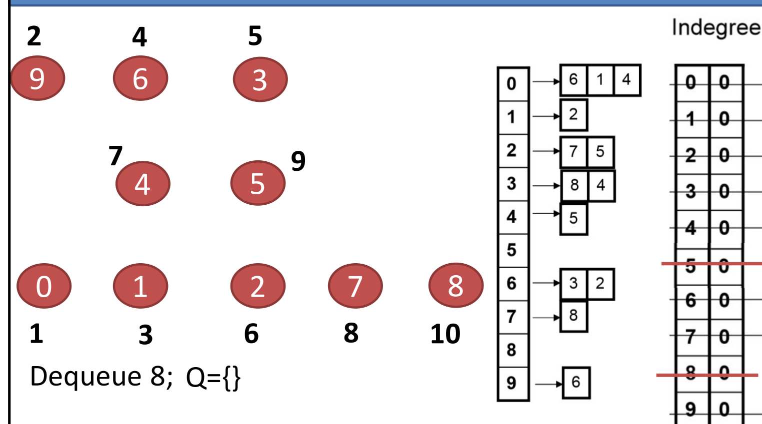
Topological order: BFS with modification



Dequeue 5; Q={8}

Output: 0 9 1 6 3 2 4 7 5

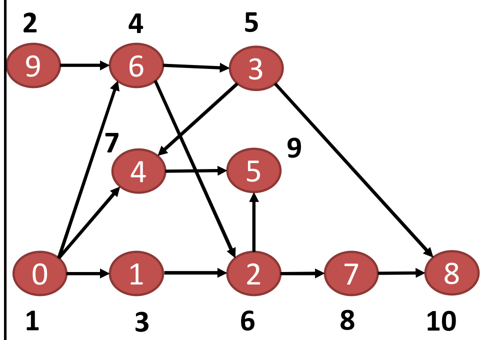
Topological order: BFS with modification



Dequeue 8; Q={}

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

Topological order: BFS with modification



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

101