



ĐỒ ÁN 1

Lý thuyết



Thông tin nhóm sinh viên

Mã nhóm: **30**

Nội dung đã hoàn thành:

Sinh viên 1

- Đã hoàn thành hết

- MSSV: 22850034
- Họ và tên: Cao Hoài Việt

Sinh viên 2

- MSSV: 22850026
- Họ và tên: Chương Hương Quý

1. Phát biểu khái niệm thành phần liên thông mạnh (strongly connected component) trên đồ thị có hướng

Thành phần liên thông mạnh của một đồ thị có hướng là những đồ thị con nằm trong đồ thị có hướng mà bản thân nó chính là một đồ thị liên thông mạnh.

2. Giải thuật tìm thành phần liên thông mạnh

Thuật toán sử dụng: Tarjan

a. Mô tả bằng lời ý tưởng tổng quát của giải thuật

Ý tưởng của thuật toán này là tìm kiếm theo chiều sâu (**DFS**), bắt đầu từ một đỉnh tùy chọn và tìm kiếm sâu dần tới bất kì đỉnh kề nào chưa được viếng thăm. Các thành phần liên thông mạnh trong đồ thị sẽ tạo ra các cây con của cây tìm kiếm mà gốc (**root**) của những cây con đó chính là gốc hay điểm bắt đầu của các thành phần liên thông mạnh.

2. Giải thuật tìm thành phần liên thông mạnh

b. Các bước thực hiện của giải thuật

Các đỉnh được đưa vào một ngăn xếp (**stack**) theo thứ tự của chúng đã được viếng thăm. Khi việc tìm kiếm trả về một cây con, các đỉnh đó sẽ được lấy ra khỏi ngăn xếp và được xác định xem liệu mỗi đỉnh được lấy ra có phải là gốc của một thành phần liên thông mạnh hay không. Nếu một đỉnh đã được xác định là gốc (**root**) của một thành phần liên thông mạnh thì nó và tất cả các đỉnh được lấy ra trước đó hình thành nên thành phần liên thông mạnh.

2. Giải thuật tìm thành phần liên thông mạnh

Pseudocode

```
set UNVISITED = -1
n = numver of vertices in the graph
g = directed adjacency matrix
stack = empty stack
ids[n] = {-1, -1, ... , -1}
root[n] = {-1, -1, ... , -1}
on_stack[n] = {false, false, ... , false}
cps = empty vector contains the list of connected components
for (i = 0; i < n, i++)
    if (g[i] is UNVISITED)
        dfs_find(v)
return root
```

Phần này chúng em viết ra IDE rồi chụp hình cho dễ nhìn vì có màu.
Vì nếu copy/paste ra thì không có màu, rất khó đọc

2. Giải thuật tìm thành phần liên thông mạnh

```
function dfs_find(v)
    stack.push(v)
    ids[v] = root[v] = index++
    on_stack[v] = true

    for (i = 0; i < n, i++)
        if g[u][i]
            if ids[i] is UNVISITED
                dfs_find(i)
                root[v] = minimum(root[v], root[i])
            else if on_stack[i] then
                root[v] = minimum(root[v], ids[i])

    if root[v] == ids[v]
        cp = a new strongly connected component (cp)
        while (stack.top() ≠ v)
            w = stack.pop()
            on_stack[w] = false
            cp.push[w]
        w = stack.pop()
        on_stack[w] = false
        cp.push[w]
        cps.push[cp]

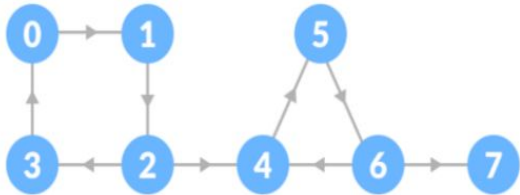
print connected components CPS to console
```

Phần này chúng em viết ra IDE rồi chụp hình cho dễ nhìn vì có màu.
Vì nếu copy/paste ra thì không có màu, rất khó đọc

2. Giải thuật tìm thành phần liên thông mạnh

c. Ví dụ trên đồ thị cụ thể để minh họa các bước của giải thuật

Chúng em sẽ sử dụng ví dụ số 2 để minh họa.

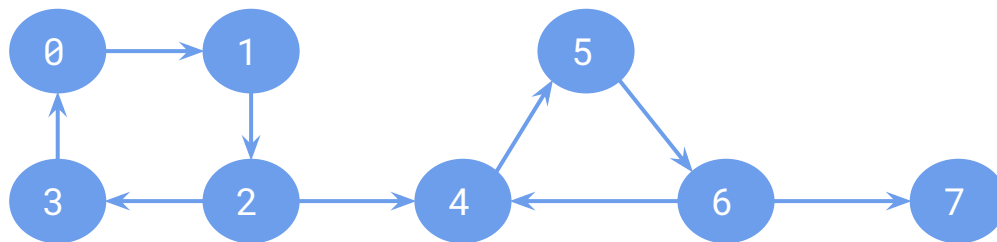
	<p>Đồ thị liên thông từng phần</p> <p>Thành phần liên thông mạnh 1: 0, 1, 2, 3</p> <p>Thành phần liên thông mạnh 2: 4, 5, 6</p> <p>Thành phần liên thông mạnh 3: 7</p>
---	--

ids[]

stack

root[]

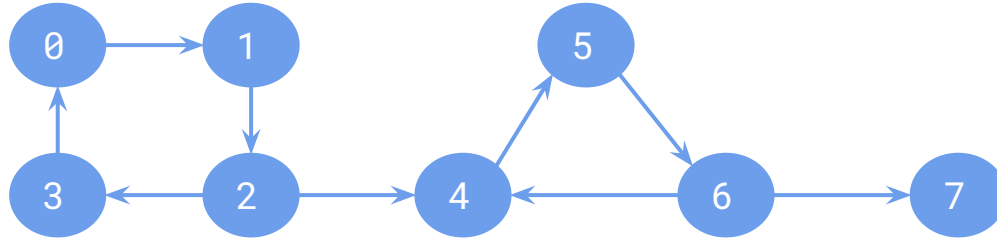
on_stack[]



Number of vertices **gNumVertices** = 8;

constant **UNVISITED** = -1;

ids[]	-1, -1, -1, -1, -1, -1, -1, -1	stack	(empty)
root[]	-1, -1, -1, -1, -1, -1, -1, -1	on_stack[]	f, f, f, f, f, f, f, f



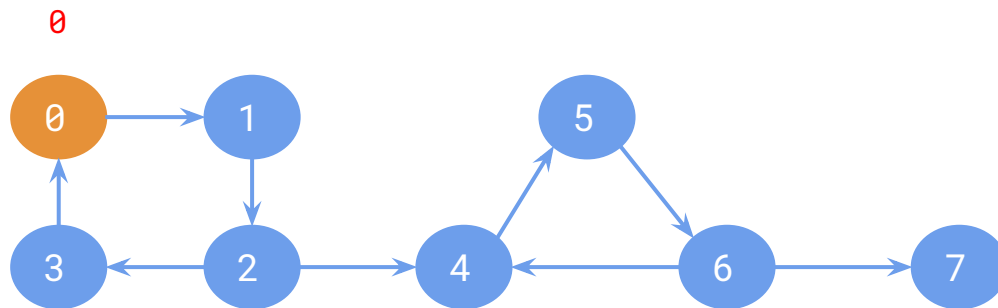
Initiation array ids, root with value = **UNVISITED**.
 Initiation an empty stack: **stack**
 Initiation an array on_stack with **false**

ids[] 0, -1, -1, -1, -1, -1, -1, -1

stack 0

root[] 0, -1, -1, -1, -1, -1, -1, -1

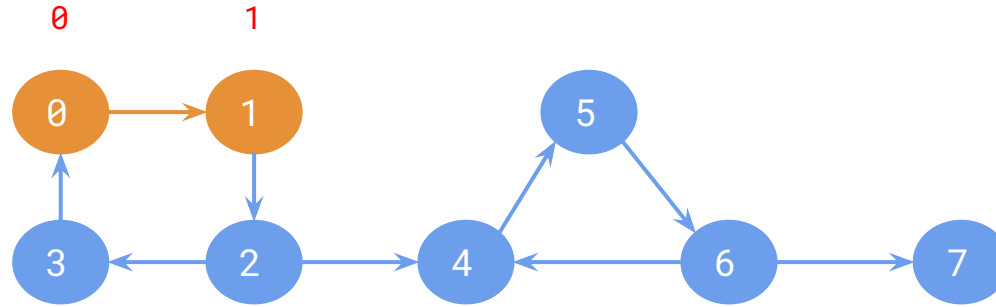
on_stack[] t, f, f, f, f, f, f, f



Start from vertex: 0

ids[]	0, 1, -1, -1, -1, -1, -1, -1
root[]	0, 1, -1, -1, -1, -1, -1, -1

stack	1, 0
on_stack[]	t, t, f, f, f, f, f, f



Vertex 1 is **UNVISITED** so we update the **root** and **id** of 1 is itself, 1.

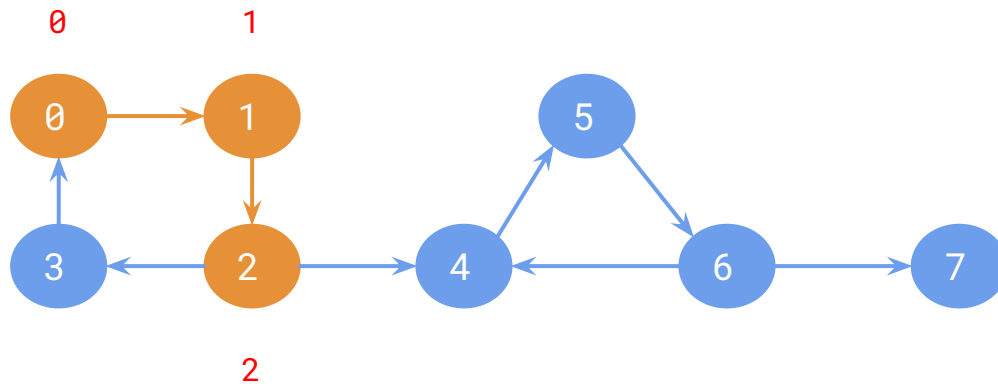
```
Ids[1] = 1;  
Root[1] = 1;
```

ids[] 0, 1, 2, -1, -1, -1, -1, -1

stack 2, 1, 0

root[] 0, 1, 2, -1, -1, -1, -1, -1

on_stack[] t, t, t, f, f, f, f, f

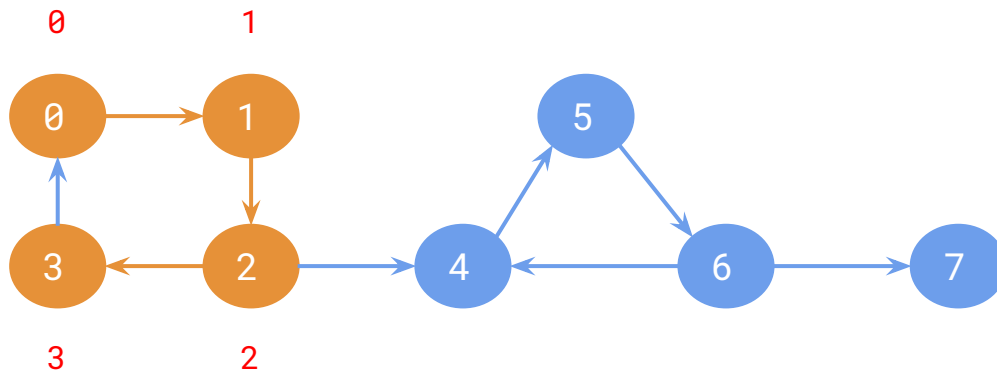


ids[] 0, 1, 2, 3, -1, -1, -1, -1

stack 3, 2, 1, 0

root[] 0, 1, 2, 3, -1, -1, -1, -1

on_stack[] t, t, t, t, f, f, f, f

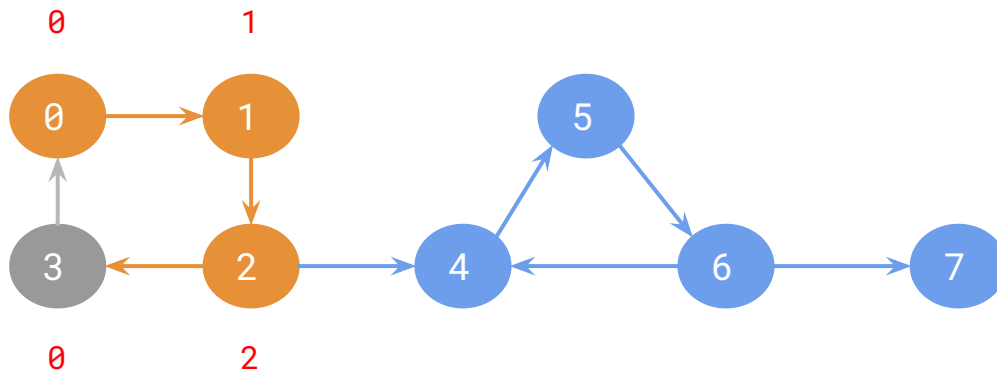


ids[]	0, 1, 2, 3, -1, -1, -1, -1
-------	----------------------------

stack	3, 2, 1, 0
-------	------------

root[]	0, 1, 2, 0, -1, -1, -1, -1
--------	----------------------------

on_stack[]	t, t, t, t, f, f, f, f
------------	------------------------



From vertex **3**, we found **0** and **on_stack[0] = true**.

Back edge

-> **minimum(root[3], root[0]) = 0**

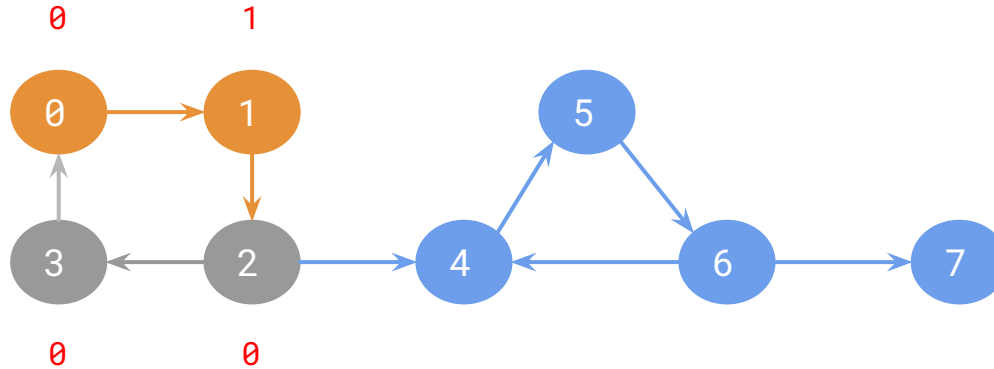
-> **update root[3] = 0**

ids[] 0, 1, 2, 3, -1, -1, -1, -1

stack 3, 2, 1, 0

root[] 0, 1, 0, 0, -1, -1, -1, -1

on_stack[] t, t, t, t, f, f, f, f



Back edge

-> `minimum(root[2], root[3]) = 0`

-> `update root[2] = 0`

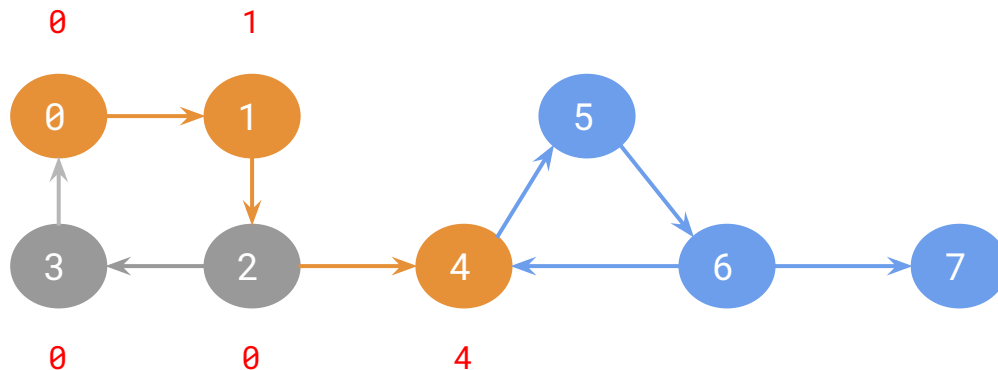
At vertex 2, DFS found the edge to vertex 4, so continue

ids[] 0, 1, 2, 3, 4, -1, -1, -1

root[] 0, 1, 0, 0, 4, -1, -1, -1

stack 4, 3, 2, 1, 0

on_stack[] t, t, t, t, t, f, f, f

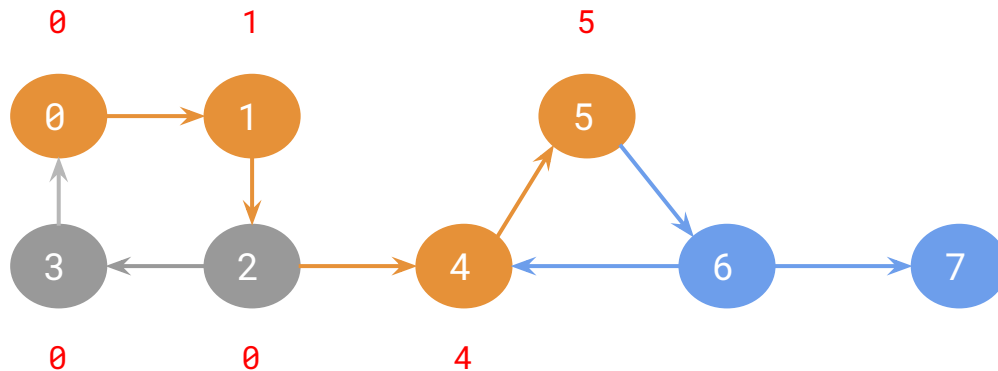


ids[] 0, 1, 2, 3, 4, 5, -1, -1

stack 5, 4, 3, 2, 1, 0

root[] 0, 1, 0, 0, 4, 5, -1, -1

on_stack[] t, t, t, t, t, t, f, f

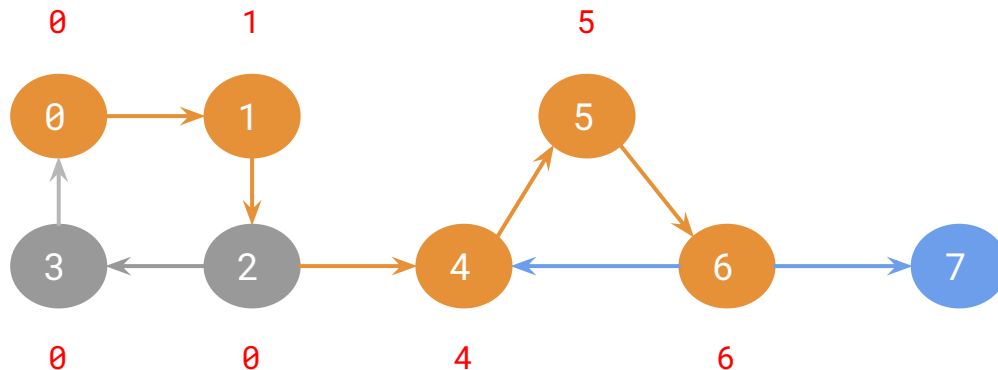


ids[] 0, 1, 2, 3, 4, 5, 6, -1

stack 6, 5, 4, 3, 2, 1, 0

root[] 0, 1, 0, 0, 4, 5, 6, -1

on_stack[] t, t, t, t, t, t, t, f

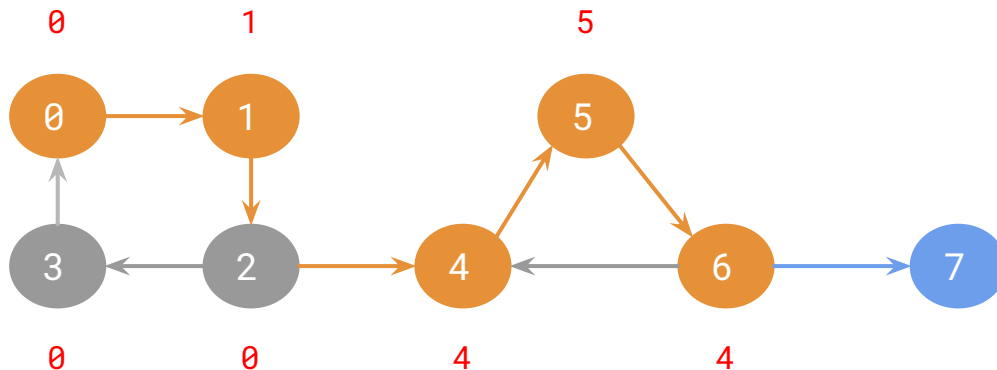


ids[] 0, 1, 2, 3, 4, 5, 6, -1

stack 6, 5, 4, 3, 2, 1, 0

root[] 0, 1, 0, 0, 4, 5, 4, -1

on_stack[] t, t, t, t, t, t, t, f



`on_stack[4] == true`

`-> minimum(root[6], root[4]) = 4;`

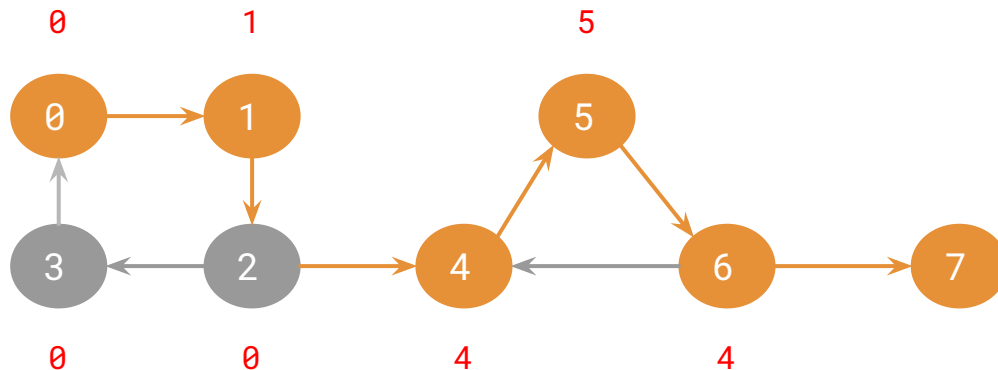
`-> update root[6] = 4;`

ids[] 0, 1, 2, 3, 4, 5, 6, 7

stack 7, 6, 5, 4, 3, 2, 1, 0

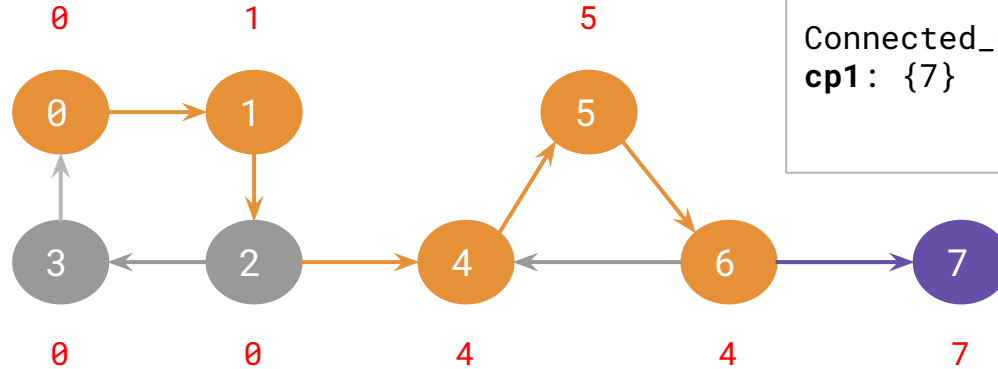
root[] 0, 1, 0, 0, 4, 5, 4, 7

on_stack[] t, t, t, t, t, t, t, t



ids[]	0, 1, 2, 3, 4, 5, 6, 7
root[]	0, 1, 0, 0, 4, 5, 4, 7

stack	7, 6, 5, 4, 3, 2, 1, 0
on_stack[]	t, t, t, t, t, t, t, t



Connected_components (CPS)
cp1: {7}

Backtracking.

root[7] == ids[7] -> We found a ROOT. Init a new connected component: **cp1**

7 == stack.top()

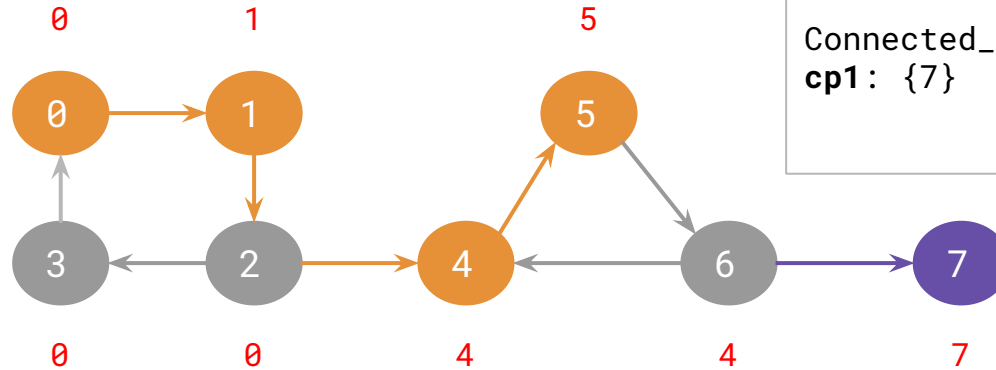
- > Add 7 to the connected component **cp1**;
- > Add **cp1** to the Connected Components (CPS);
- > update **on_stack[7] = false**;
- > remove 7 from **stack (stack.pop());**

ids[] 0, 1, 2, 3, 4, 5, 6, 7

stack 6, 5, 4, 3, 2, 1, 0

root[] 0, 1, 0, 0, 4, 4, 4, 7

on_stack[] t, t, t, t, t, t, f, f



Connected_components (CPS)
cp1: {7}

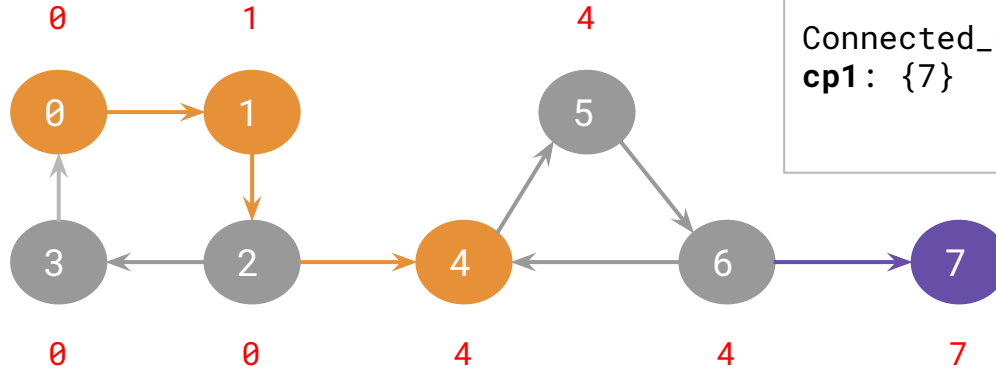
```
-> minimum(root[6], root[7]) = 4;  
-> update root[6] = 4;
```

ids[] 0, 1, 2, 3, 4, 5, 6, 7

stack 6, 5, 4, 3, 2, 1, 0

root[] 0, 1, 0, 0, 4, 4, 4, 7

on_stack[] t, t, t, t, t, f, f, f



Connected_components (CPS)
cp1: {7}

-> `minimum(root[5], root[6]) = 4;`

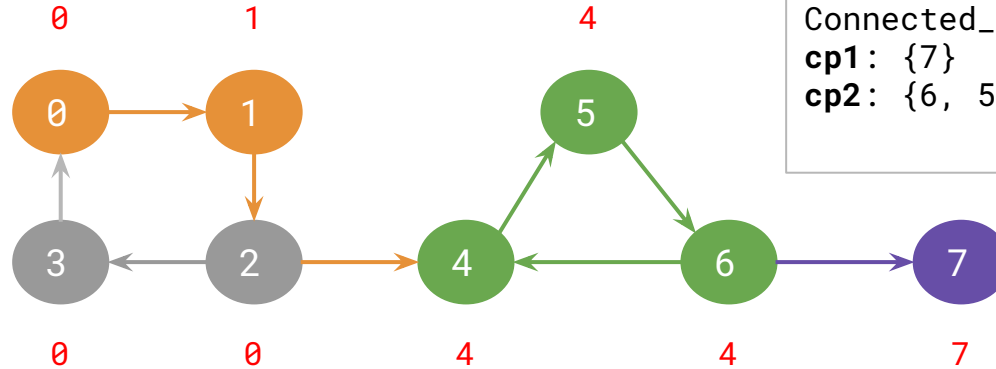
-> `update root[5] == 4;`

ids[] 0, 1, 2, 3, 4, 5, 6, 7

stack 3, 2, 1, 0

root[] 0, 1, 0, 0, 4, 4, 4, 7

on_stack[] t, t, t, t, f, f, f, f



Connected_components (CPS)

cp1: {7}

cp2: {6, 5, 4}

-> `minimum(root[4], root[5]) = 4;`

-> `update root[4] = 4;`

`root[4] == ids[4]` -> We found another **ROOT**. Init a new connected component: **cp2**

Add current `stack.top()` to the **cp2** then **remove** it from the stack, set `on_stack == false` until `stack.top() == 4;`

When `stack.top() == 4;`

-> Also remove 4 from stack.

-> Add 4 to the component **cp2**

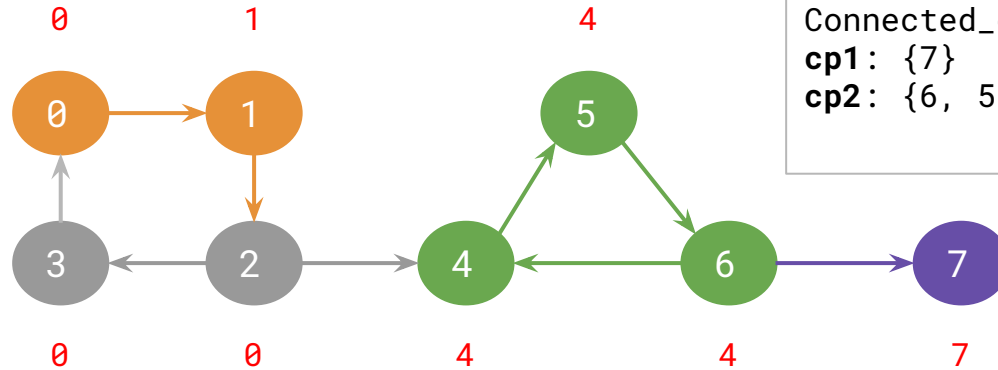
-> Push **cp2** to the list of connected components **CPS**

ids[] 0, 1, 2, 3, 4, 5, 6, 7

stack 3, 2, 1, 0

root[] 0, 1, 0, 0, 4, 4, 4, 7

on_stack[] t, t, t, t, f, f, f, f



Connected_components (CPS)

cp1: {7}

cp2: {6, 5, 4}

-> minimum(root[2], root[4]) = 0;

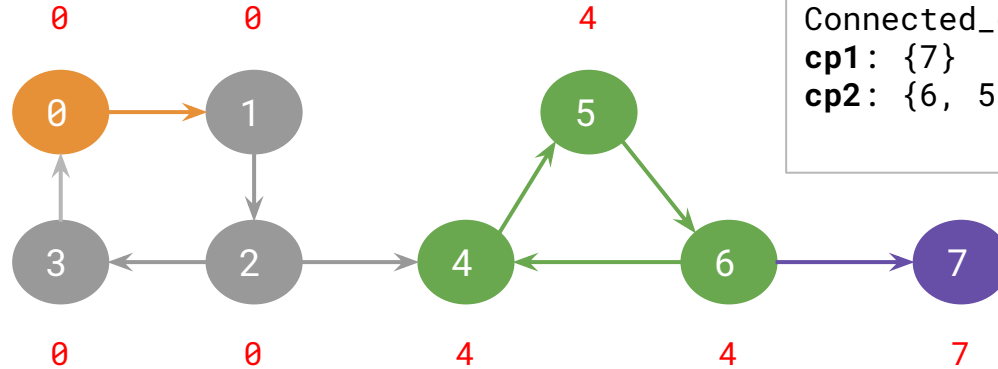
-> update root[2] = 0;

ids[] 0, 1, 2, 3, 4, 5, 6, 7

stack 3, 2, 1, 0

root[] 0, 0, 0, 0, 4, 4, 4, 7

on_stack[] t, t, t, t, f, f, f, f



Connected_components (CPS)

cp1: {7}

cp2: {6, 5, 4}

-> minimum(root[1], root[2]) = 0;

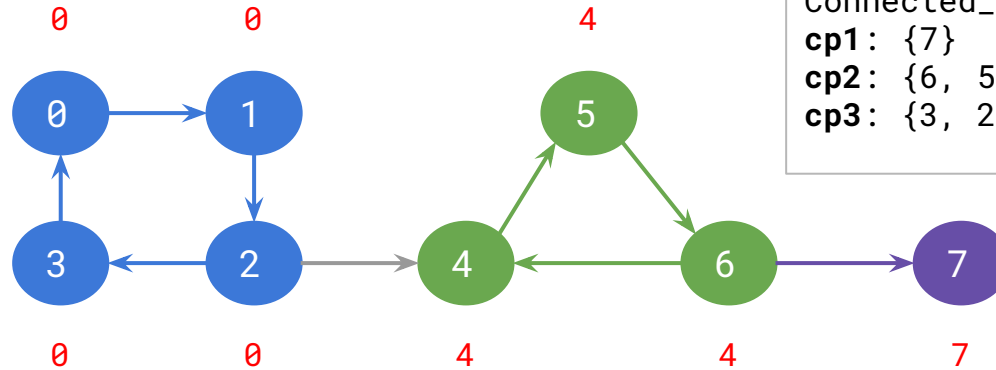
-> update root[1] = 0;

ids[] 0, 1, 2, 3, 4, 5, 6, 7

stack (empty)

root[] 0, 0, 0, 0, 4, 4, 4, 7

on_stack[] f, f, f, f, f, f, f, f



Connected_components (CPS)

cp1: {7}

cp2: {6, 5, 4}

cp3: {3, 2, 1, 0}

`root[0] == ids[0]` -> Found another **ROOT**. Init a new connected component: **cp3**

Add current `stack.top()` to the **cp3** then remove it from the stack, set `on_stack == false` until `stack.top() == 0`;

When `stack.top() == 0`;

-> Also remove **0** from stack.

-> Add **0** to the component **cp3**

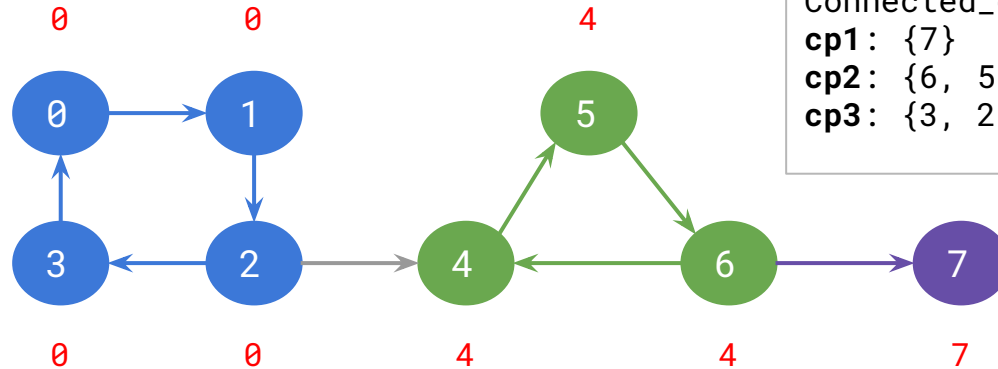
-> Push **cp3** to the list of connected components **CPS**

ids[] 0, 1, 2, 3, 4, 5, 6, 7

root[] 0, 0, 0, 0, 4, 4, 4, 7

stack (empty)

on_stack[] f, f, f, f, f, f, f, f



Connected_components (CPS)

cp1: {7}

cp2: {6, 5, 4}

cp3: {3, 2, 1, 0}

All vertices have been visited and stack is empty.

-> **Print the connected components (CPS) to console.**

2. Giải thuật tìm thành phần liên thông mạnh

d. So sánh ưu và nhược điểm của các thuật toán

Thuật toán	Ưu điểm	Nhược điểm
Tarjan	Chỉ cần DFS một lần duy nhất	Để in được thành phần liên thông mạnh, thuật toán này cần phải đợi tìm thấy gốc (root) của nó.
Kosarajus	Để in được thành phần liên thông mạnh, thuật toán này có thể in ngay khi thực hiện DFS lần thứ 2 (Chiều ngược lại).	Cần tới hai lần thực hiện DFS, nên thời gian thực hiện sẽ cao gấp đôi Tarjan.