

# Connected Components & Biconnected Components

Joseph Chuang-Chieh Lin (林莊傑)

Department of Computer Science & Engineering,  
National Taiwan Ocean University

Fall 2025



# Outline

## 1 Connected Components

- Spanning Trees
- Articulation Points & Biconnected Graph
- Finding the articulation points



# Outline

## 1 Connected Components

- Spanning Trees
- Articulation Points & Biconnected Graph
- Finding the articulation points



# Connectivity

## Problem I

Determine if an (un)directed graph is connected.

# Connectivity

## Problem I

Determine if an (un)directed graph is connected.

We can solve this problem by calling either  $\text{dfs}(v)$  or  $\text{bfs}(v)$  for an arbitrary vertex  $v \in V(G)$ , and then determining if there are **any unvisited vertices**.

# Connectivity

## Problem I

Determine if an (un)directed graph is connected.

We can solve this problem by calling either  $\text{dfs}(v)$  or  $\text{bfs}(v)$  for an arbitrary vertex  $v \in V(G)$ , and then determining if there are **any unvisited vertices**.

## Problem II

List all connected components of an (un)directed graph.

# Connectivity

## Problem I

Determine if an (un)directed graph is connected.

We can solve this problem by calling either  $\text{dfs}(v)$  or  $\text{bfs}(v)$  for an arbitrary vertex  $v \in V(G)$ , and then determining if there are **any unvisited vertices**.

## Problem II

List all connected components of an (un)directed graph.

This can be done by making repeated calls to either  $\text{dfs}(v)$  or  $\text{bfs}(v)$  where  $v$  is an **unvisited vertex**.



```
void connected(void) { // dfs(0) or bfs(0)
/* determine the connected components of a graph */
    int i;
    for (i=0; i<n; i++) {
        if (!visited[i]) {
            dfs(i);
            printf("\n");
        }
    }
}
```

# Analysis of connected

- If  $G$  is represented by its adjacency lists, then the total time taken by DFS is  $O(e)$ .



# Analysis of connected

- If  $G$  is represented by its adjacency lists, then the total time taken by DFS is  $O(e)$ .
- Since the for loop takes  $O(n)$  time, the total time needed to generate all the connected components is

# Analysis of connected

- If  $G$  is represented by its adjacency lists, then the total time taken by DFS is  $O(e)$ .
- Since the for loop takes  $O(n)$  time, the total time needed to generate all the connected components is  $O(n + e)$ .

# Analysis of connected

- If  $G$  is represented by its adjacency lists, then the total time taken by DFS is  $O(e)$ .
- Since the for loop takes  $O(n)$  time, the total time needed to generate all the connected components is  $O(n + e)$ .
- If  $G$  is represented by an **adjacency matrix**, then the time needed to determine the connected components is  $O(n^2)$ .

# Spanning Trees

## Spanning Trees

A tree  $T$  is said to be a *spanning tree* of a connected graph  $G$  if  $T$  is a subgraph of  $G$  and  $T$  contains all vertices of  $G$ .

# Spanning Trees

## Spanning Trees

A tree  $T$  is said to be a *spanning tree* of a connected graph  $G$  if  $T$  is a subgraph of  $G$  and  $T$  contains all vertices of  $G$ .

- When graph  $G$  is connected, a DFS or BFS implicitly partitions the edges in  $G$  into two sets:

# Spanning Trees

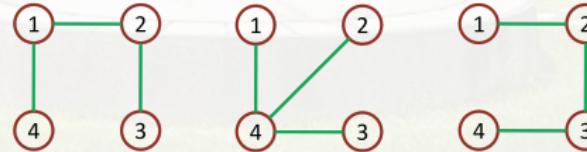
## Spanning Trees

A tree  $T$  is said to be a *spanning tree* of a connected graph  $G$  if  $T$  is a subgraph of  $G$  and  $T$  contains all vertices of  $G$ .

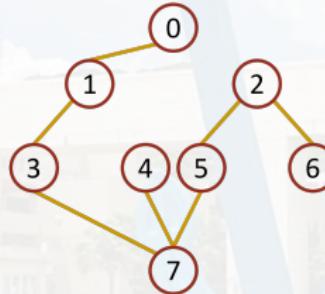
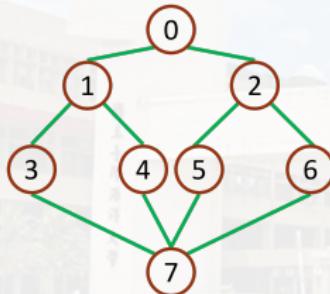
- When graph  $G$  is connected, a DFS or BFS implicitly partitions the edges in  $G$  into two sets:
  - Tree edges:** the set of edges used or traversed during the search.
  - Nontree edges:** the set of remaining edges.



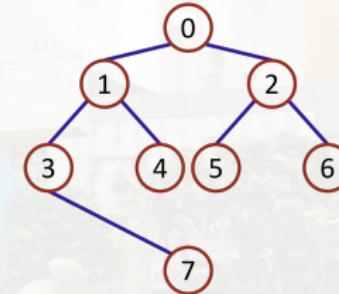
Three spanning trees of  $G_1$ .



# DFS Spanning Trees & BFS Spanning Trees



DFS (0)  
spanning tree

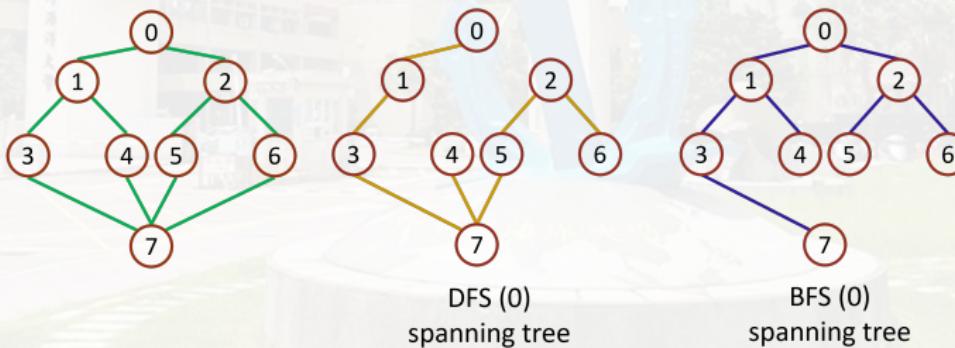


BFS (0)  
spanning tree

# Properties of (DFS or BFS) Spanning Trees

## Property I

Suppose we add a nontree edge,  $(v, w)$ , into any spanning tree,  $T$ . The result is a cycle that consists of the edge  $(v, w)$  and all the edges on the path from  $w$  to  $v$  in  $T$ .

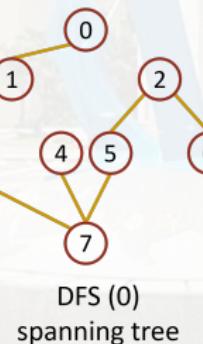
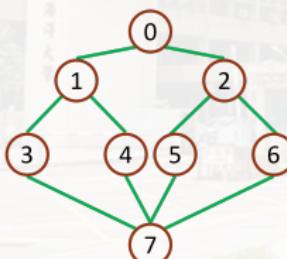


# Properties of (DFS or BFS) Spanning Trees

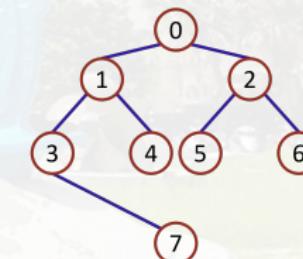
## Property II

A spanning tree is a **minimal** subgraph,  $G'$ , of  $G$  such that  $V(G') = V(G)$  and  $G'$  is connected.

- A spanning tree has  $n - 1$  edges.



DFS (0)  
spanning tree



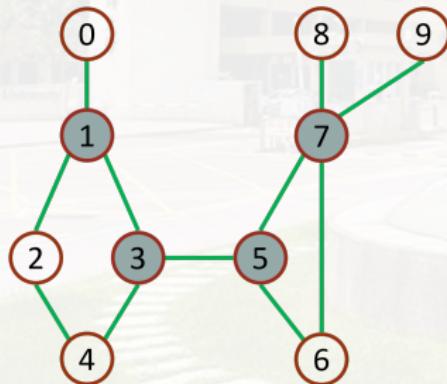
BFS (0)  
spanning tree

# Articulation Points

## Articulation Points

An **articulation point** is a vertex  $v$  of  $G$  such that the deletion of  $v$ , together with all edges incident on  $v$ , produces a graph,  $G'$ , that has  $\geq 2$  connected components.

a connected graph



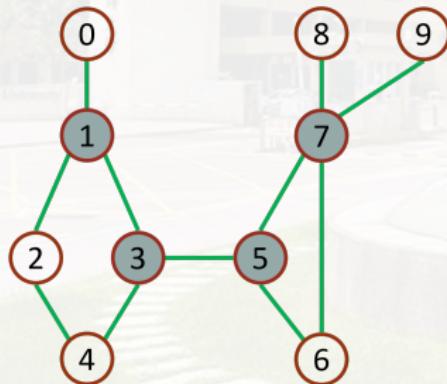
# Articulation Points

## Articulation Points

An **articulation point** is a vertex  $v$  of  $G$  such that the deletion of  $v$ , together with all edges incident on  $v$ , produces a graph,  $G'$ , that has  $\geq 2$  connected components.

a connected graph

- Four articulation points:

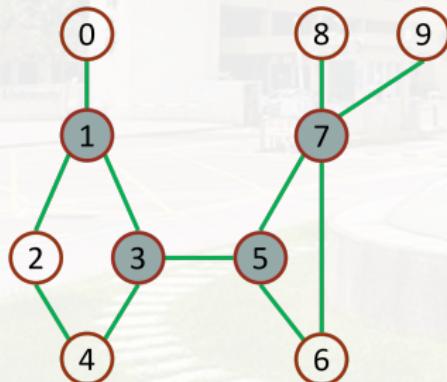


# Articulation Points

## Articulation Points

An **articulation point** is a vertex  $v$  of  $G$  such that the deletion of  $v$ , together with all edges incident on  $v$ , produces a graph,  $G'$ , that has  $\geq 2$  connected components.

a connected graph



- Four articulation points:  
1, 3, 5, 7.

# Biconnected Graph (雙連通圖)

## Biconnected Graph

A biconnected graph is a connected graph that has **NO** articulation points.

## Biconnected Component

A biconnected component of a connected graph  $G$  is a **maximal biconnected subgraph  $H$  of  $G$ .**

- $H$  is “maximal”: no other subgraph that is both biconnected and properly contains  $H$ .



# Biconnected Graph (雙連通圖)

## Biconnected Graph

A biconnected graph is a connected graph that has **NO** articulation points.

- A connected graph  $G$  which has an articulation point  $\Rightarrow G$  is NOT biconnected.

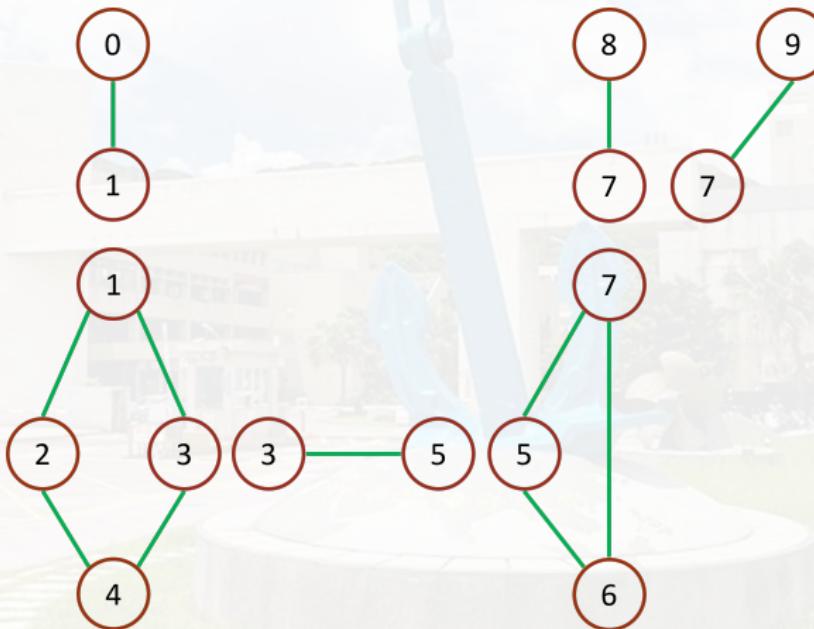
## Biconnected Component

A biconnected component of a connected graph  $G$  is a **maximal biconnected subgraph  $H$  of  $G$** .

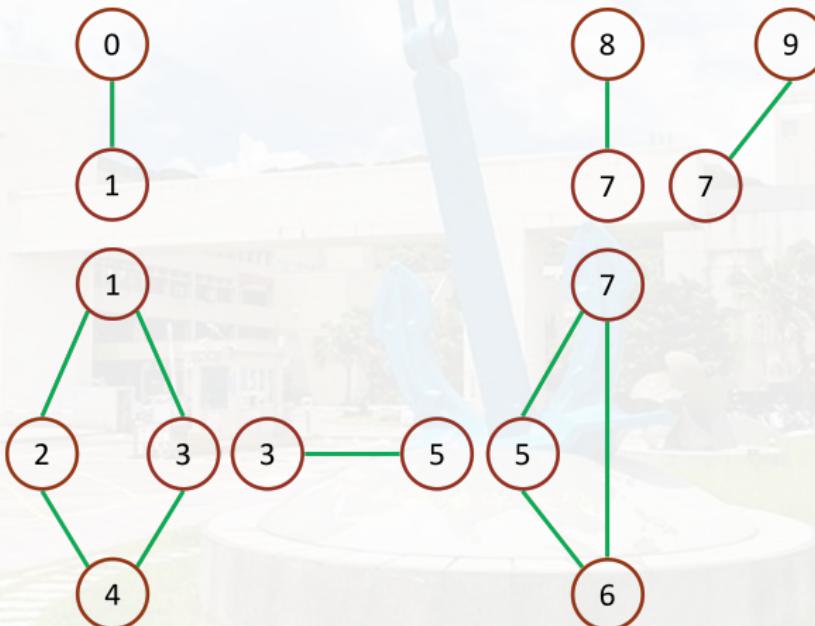
- $H$  is “maximal”: no other subgraph that is both biconnected and properly contains  $H$ .



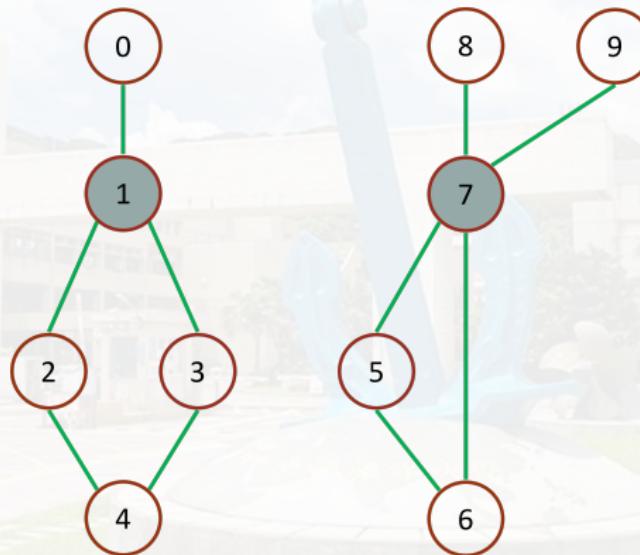
# Connected Components



# Biconnected Components (an Example)



# Biconnected Components (NOT an Example)

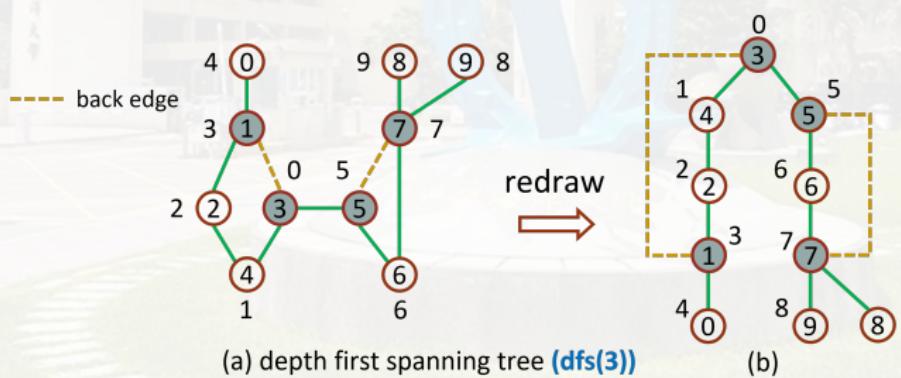


# Finding articulation points (1/3)

We can find biconnected components of a graph  $G$  using any depth-first spanning of  $G$ .

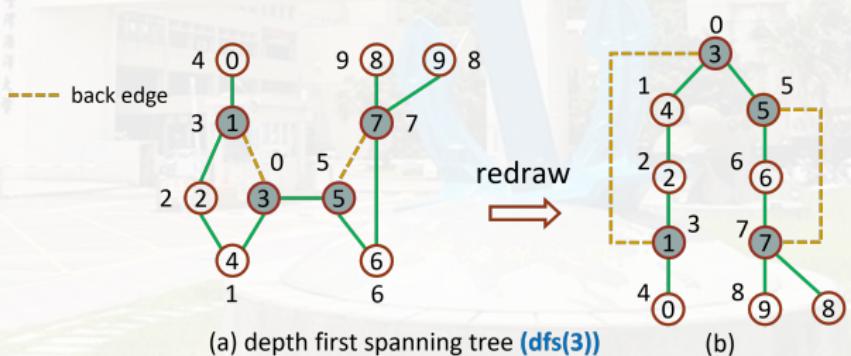
## Back edges

- Tree edges: DFS
- Nontree edges: we call them **back edges**



# Observations

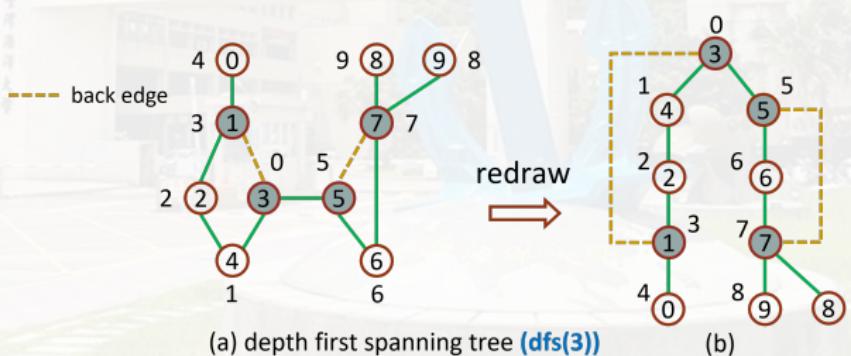
- The root of a depth first spanning tree is an articulation point if and only if it has  $\geq 2$  children.
- Any other vertex  $u$  is an articulation point if and only if it has  $\geq 1$  child  $w$  such that we cannot reach an ancestor of  $u$  using that consists of only  $w$ , descendants of  $w$ , and a single back edge.



- $v_5$  is an articulation point

# Observations

- The root of a depth first spanning tree is an articulation point if and only if it has  $\geq 2$  children.
- Any other vertex  $u$  is an articulation point if and only if it has  $\geq 1$  child  $w$  such that we cannot reach an ancestor of  $u$  using that consists of only  $w$ , descendants of  $w$ , and a single back edge.



- $v_5$  is an articulation point, but  $v_6$  is NOT.

## Finding articulation points (2/3)

$\text{dfn}(v)$

The depth first numbers, or  $\text{dfn}$ , of the vertices give the sequence in which the vertices are visited during the depth first search.

## Finding articulation points (2/3)

$\text{dfn}(v)$

The depth first numbers, or  $\text{dfn}$ , of the vertices give the sequence in which the vertices are visited during the depth first search.

- If  $u$  is an ancestor of  $v$  in the depth first spanning tree, then  $\text{dfn}(u) < \text{dfn}(v)$ .

## Finding articulation points (2/3)

### $\text{dfn}(v)$

The depth first numbers, or  $\text{dfn}$ , of the vertices give the sequence in which the vertices are visited during the depth first search.

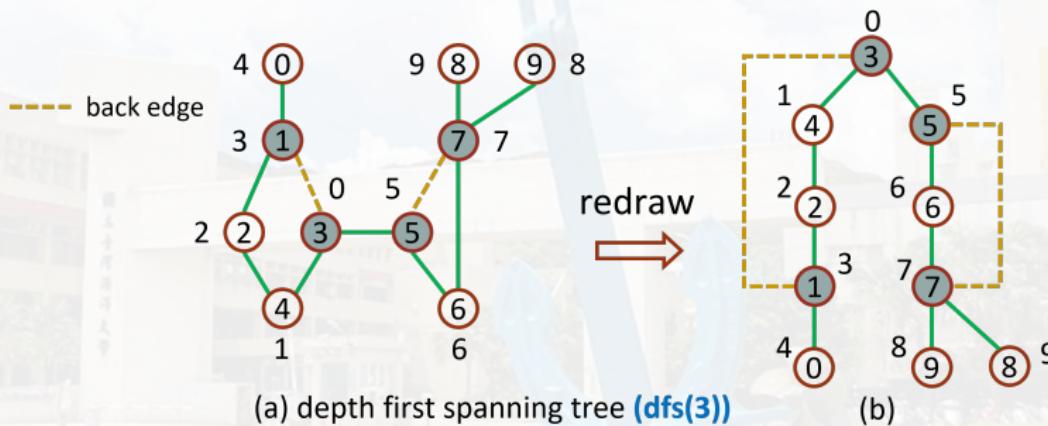
- If  $u$  is an ancestor of  $v$  in the depth first spanning tree, then  $\text{dfn}(u) < \text{dfn}(v)$ .

### $\text{low}(v)$

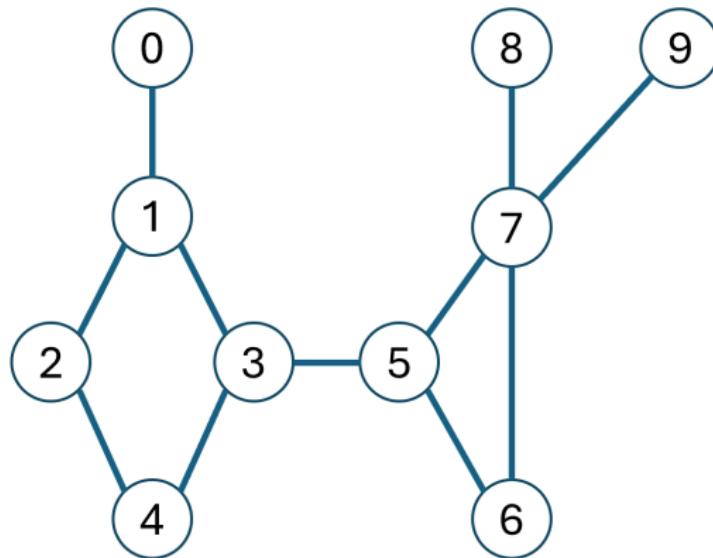
The  $\text{low}(u)$  value of vertex  $u$  is the lowest depth first number that we can reach from  $u$  using a path of descendants followed by at most 1 back edge.

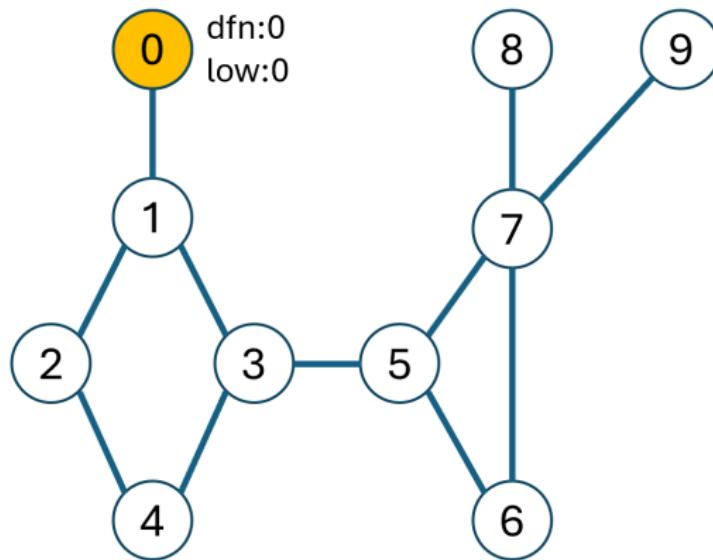
$$\text{low}(u) = \min \left\{ \begin{array}{l} \text{dfn}(u), \\ \min\{\text{low}(w) \mid w \text{ is a child of } u\}, \dots (*) \\ \min\{\text{dfn}(w) \mid (u, w) \text{ is a back edge}\} \dots (**) \end{array} \right.$$

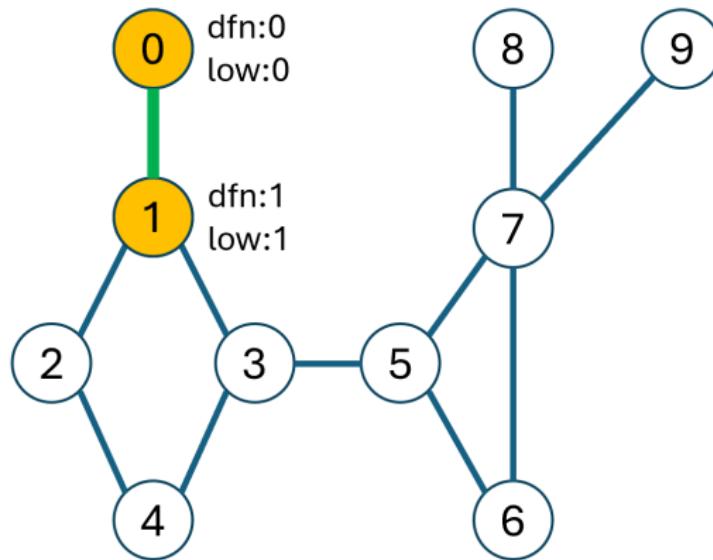
# Example of Computing dfn and low values

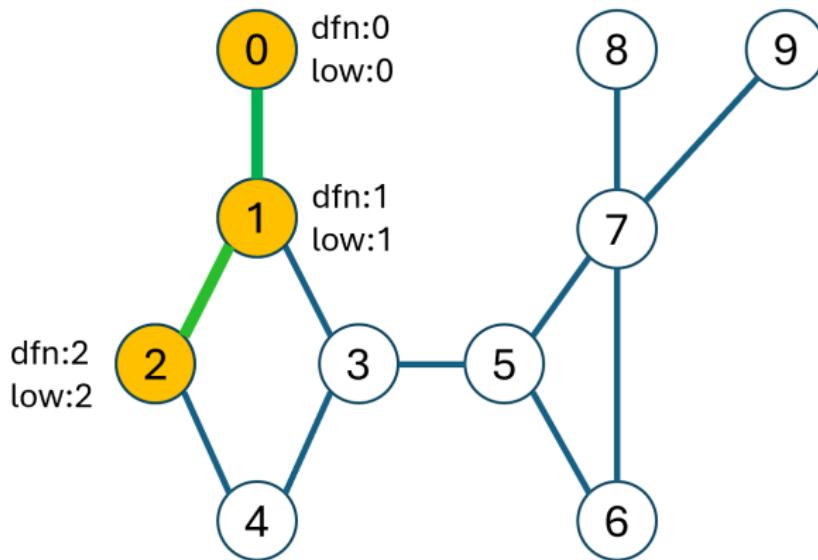


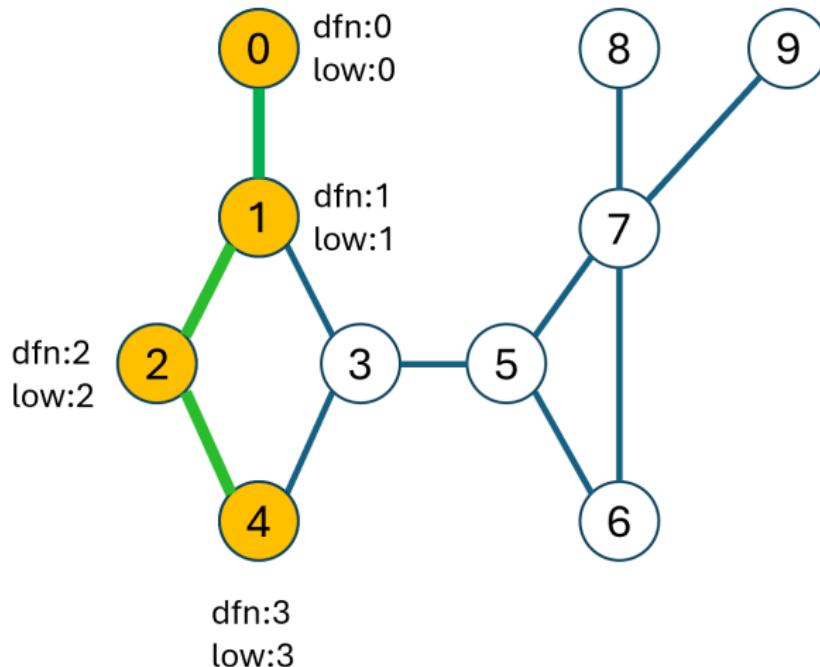
vertex	0	1	2	3	4	5	6	7	8	9
dfn	4	3	2	0	1	5	6	7	9	8
low	4	0	0	0	0	5	5	5	9	8

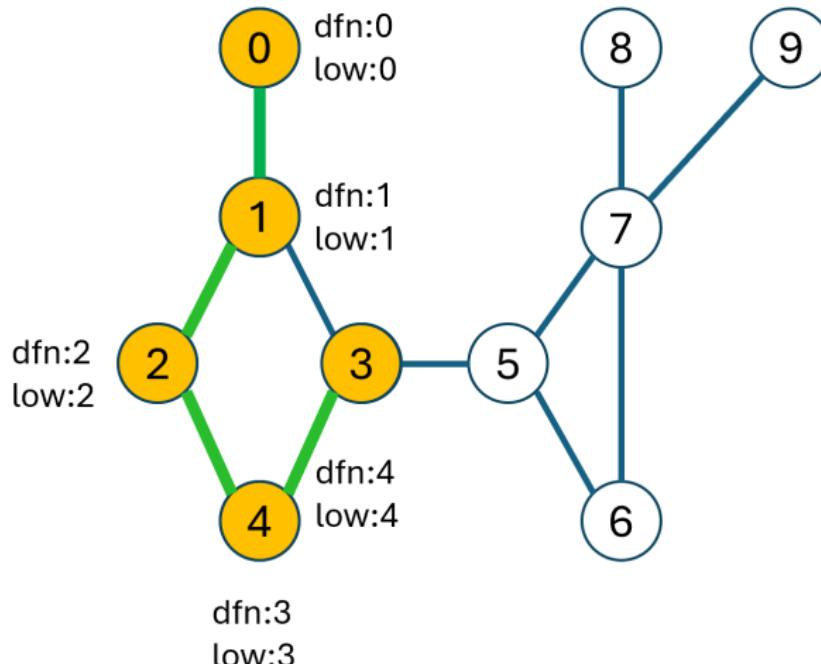


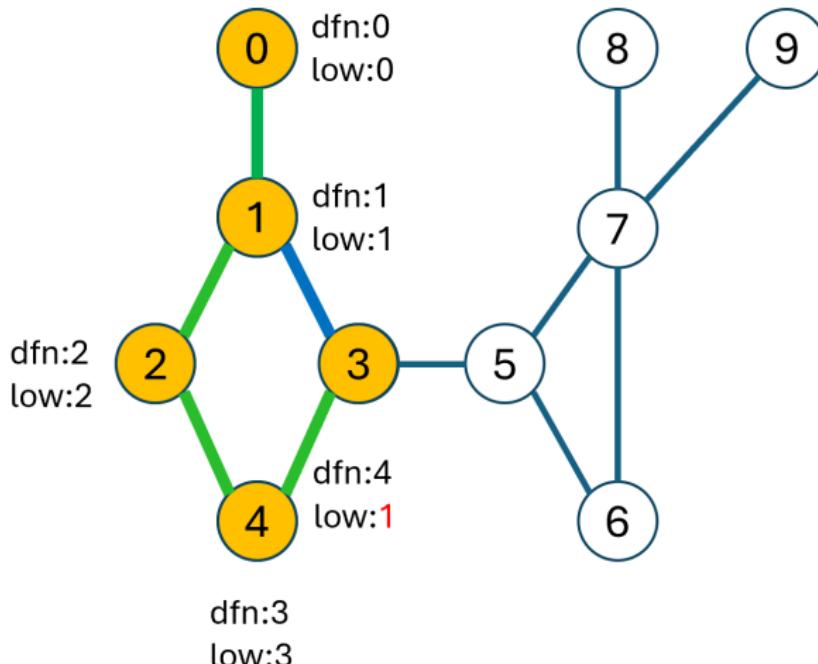


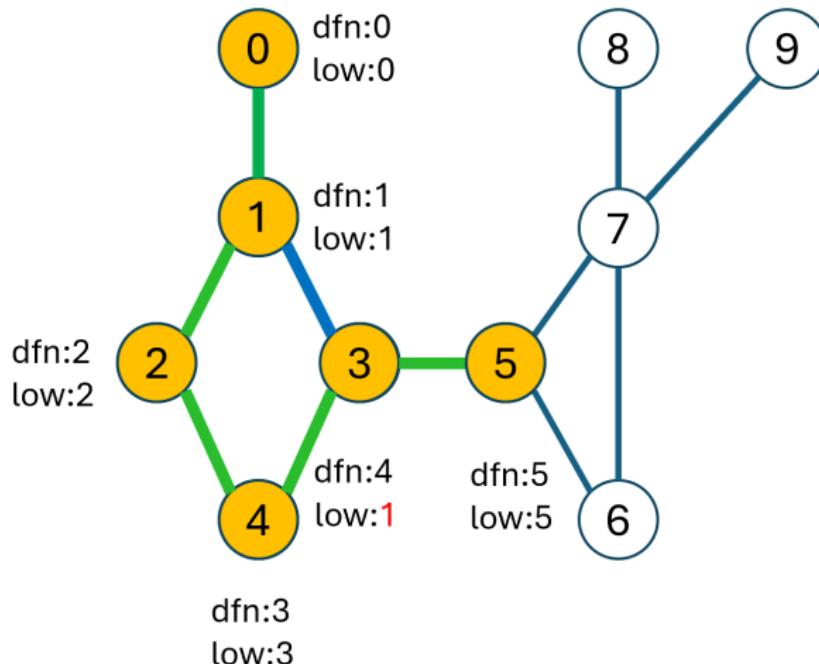


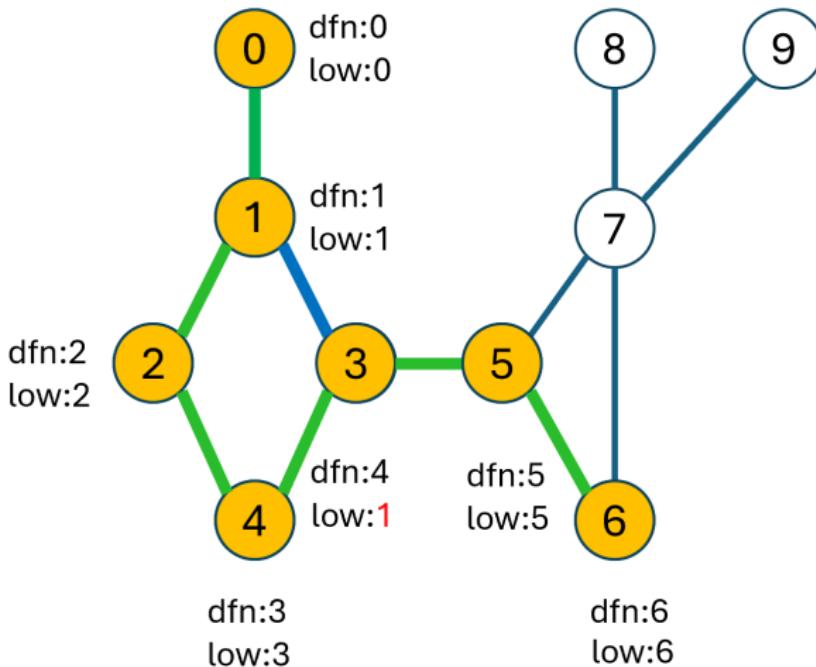


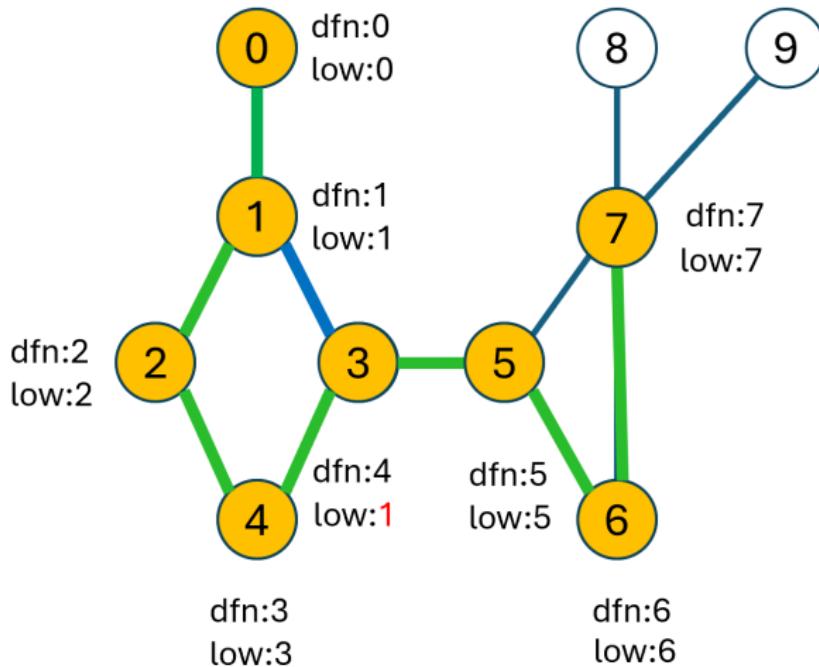


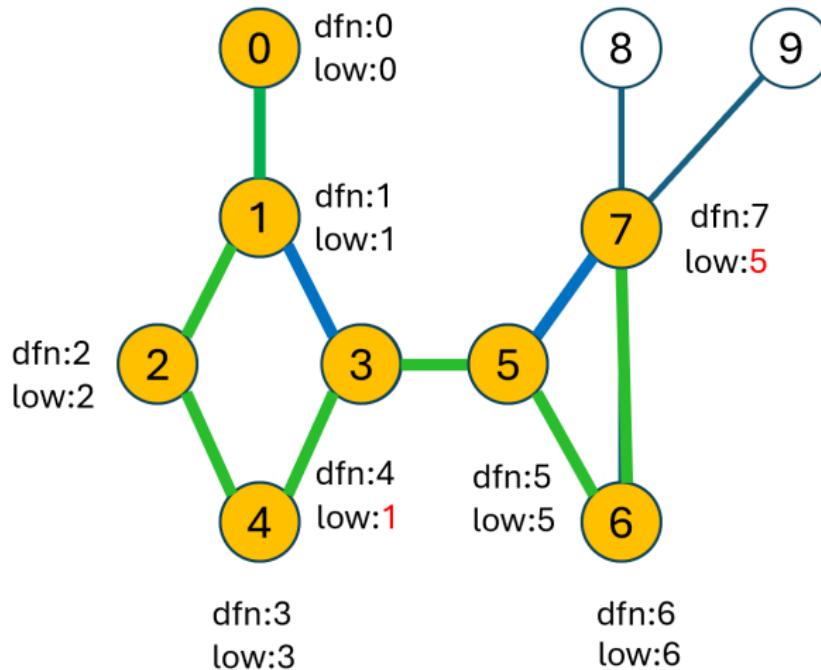


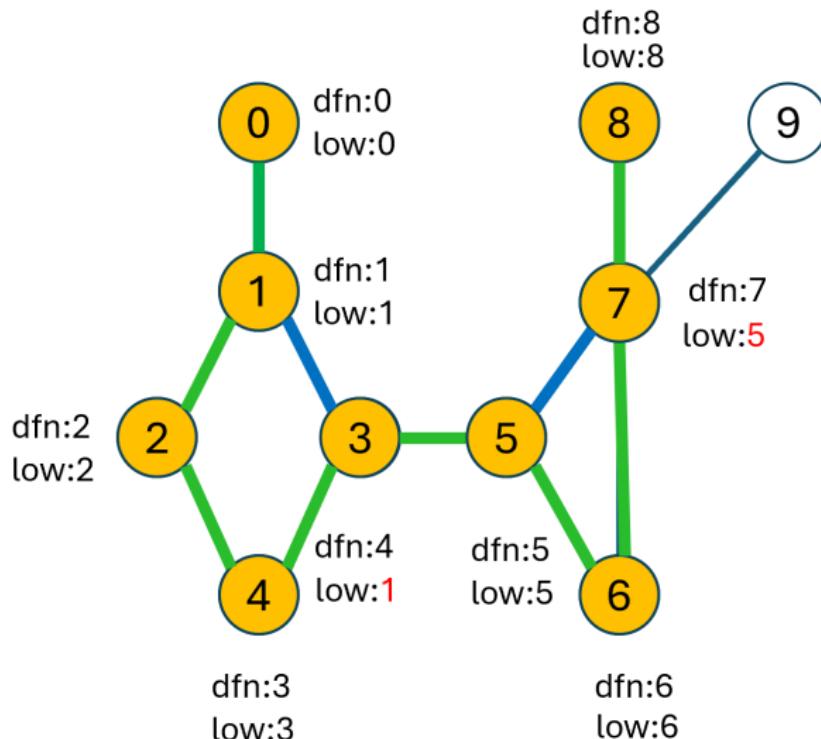


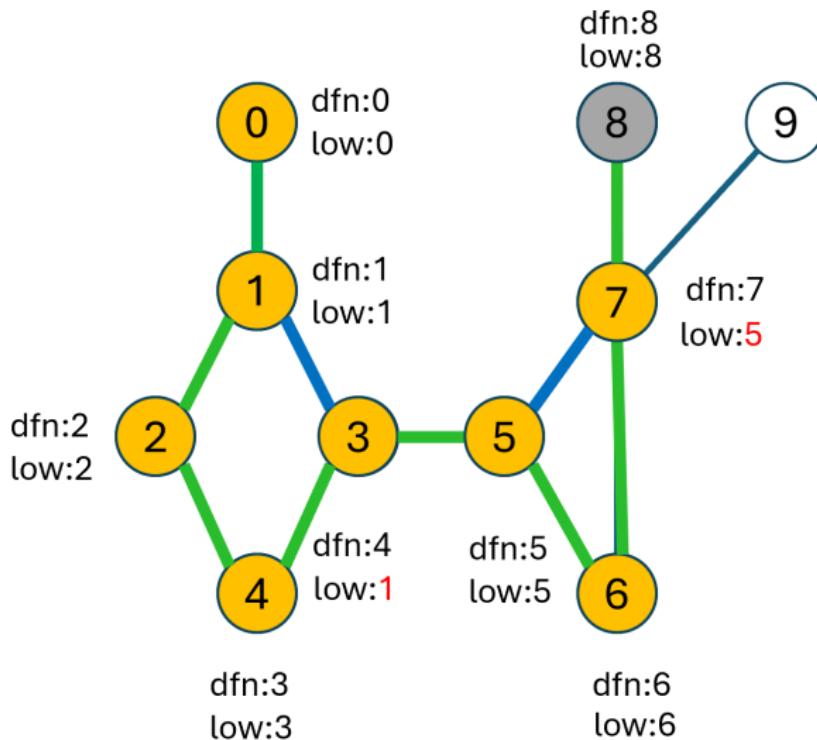


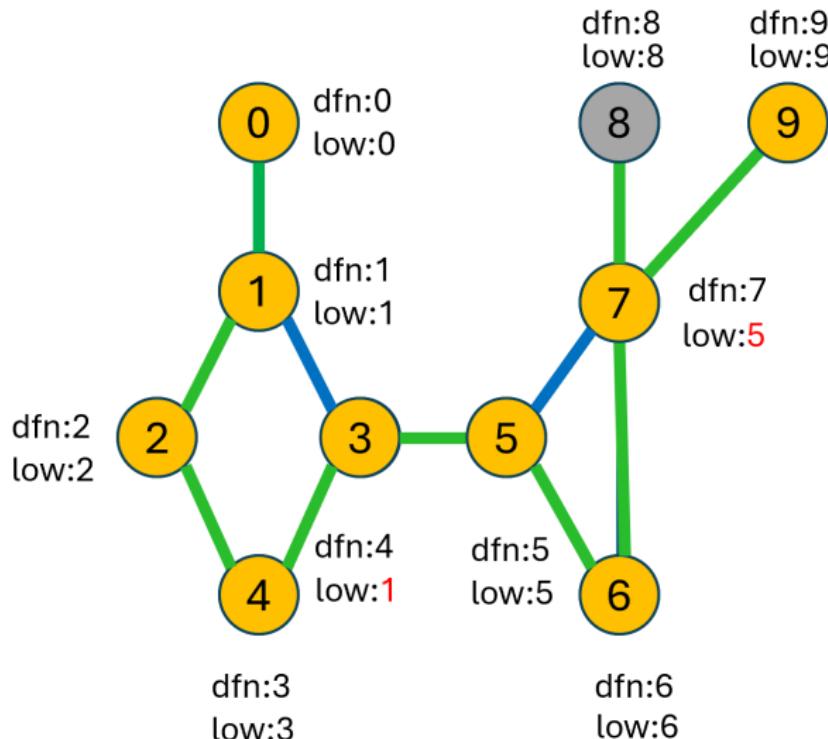


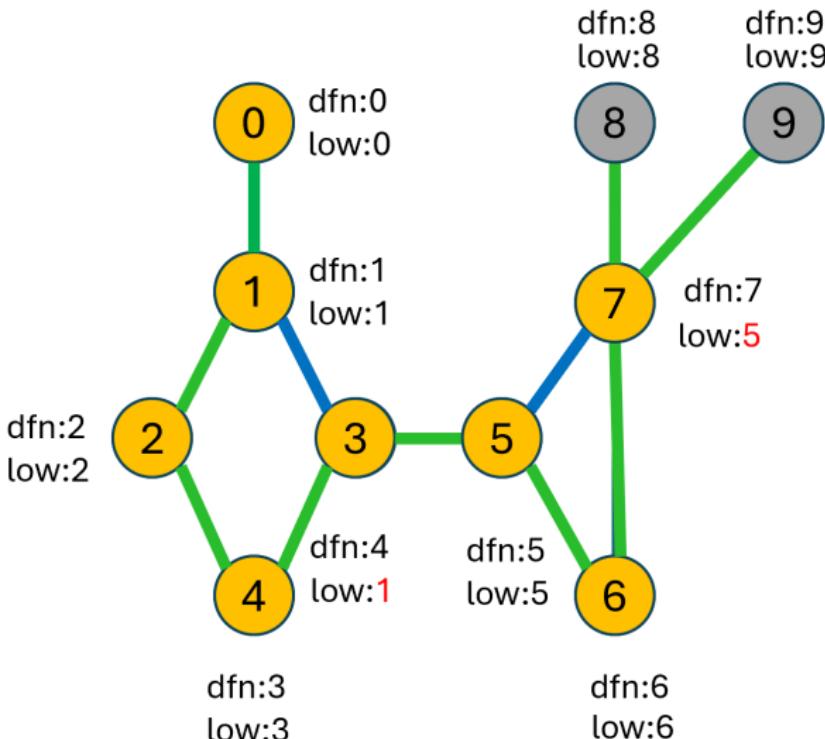


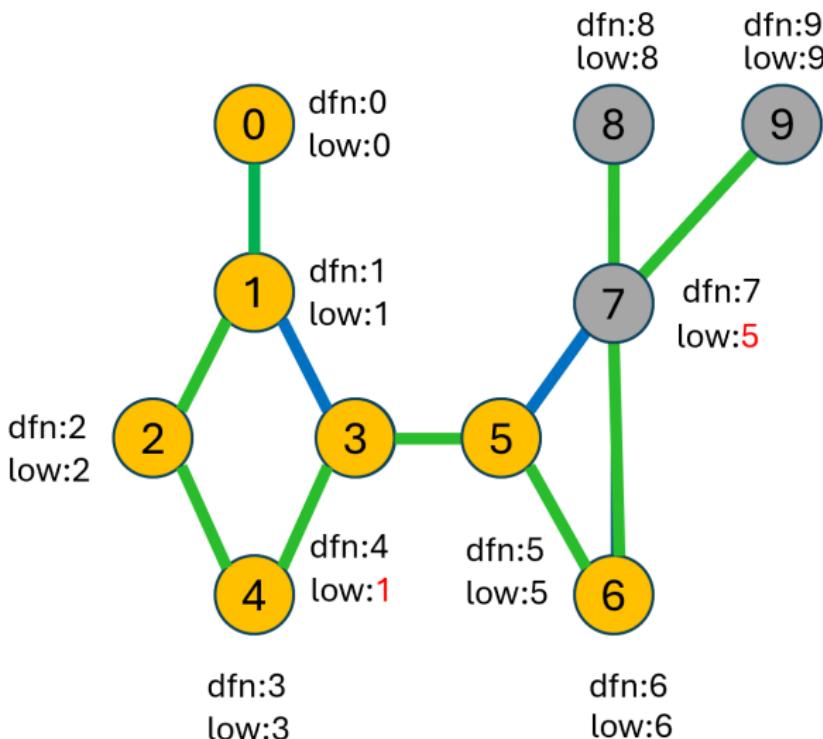


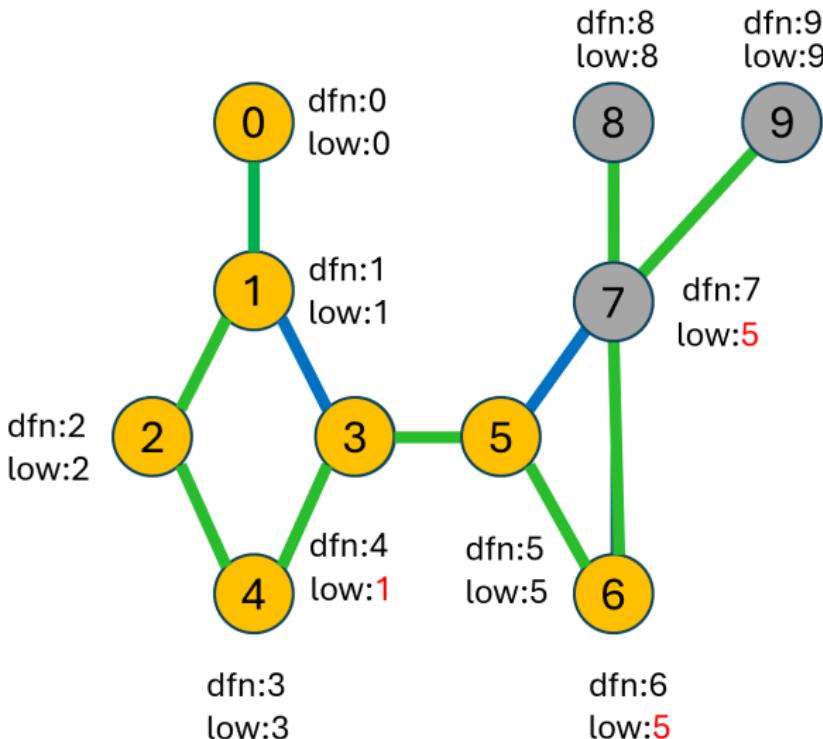


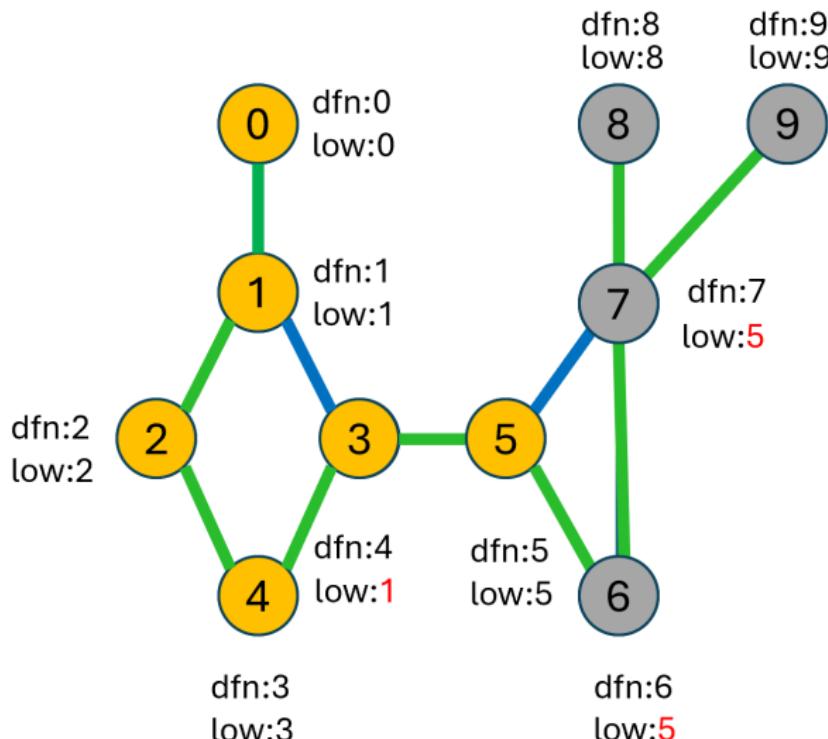


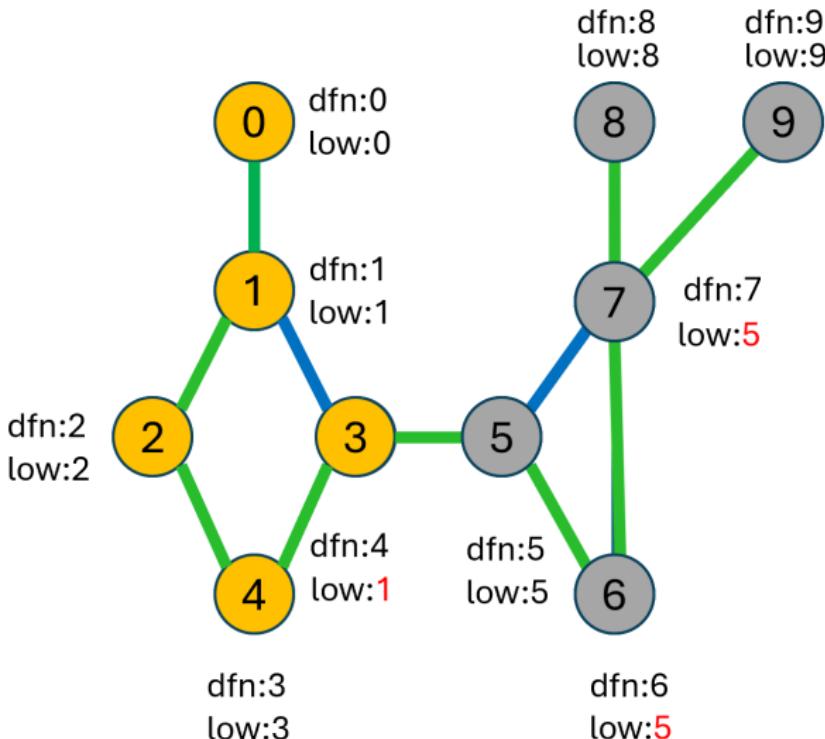


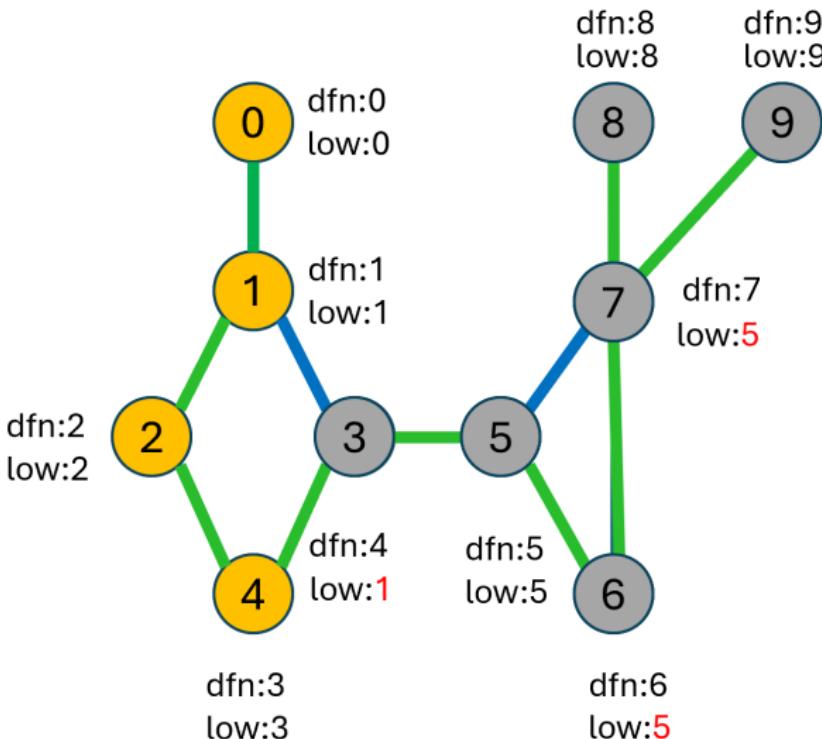


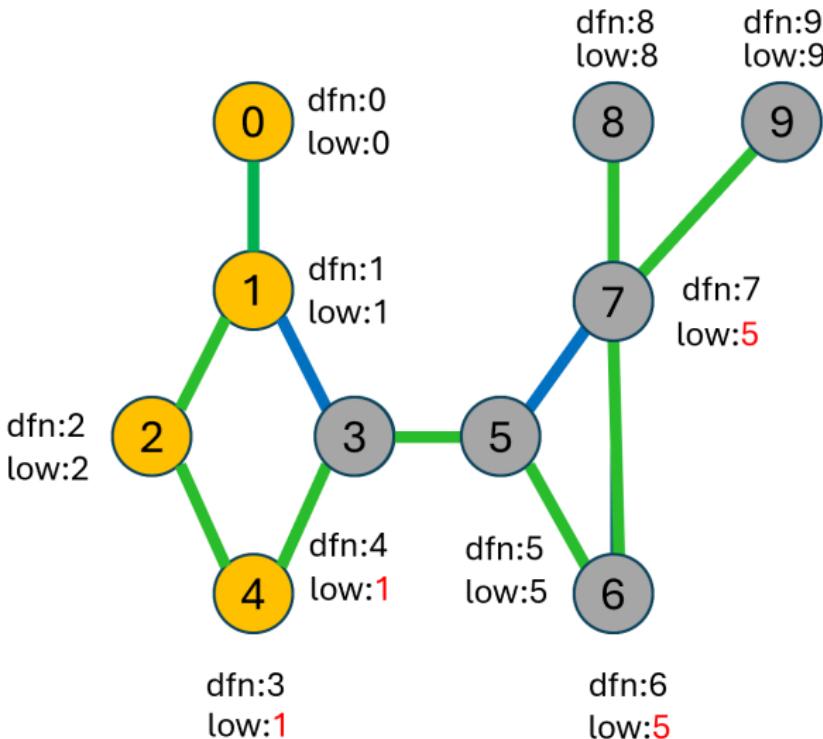


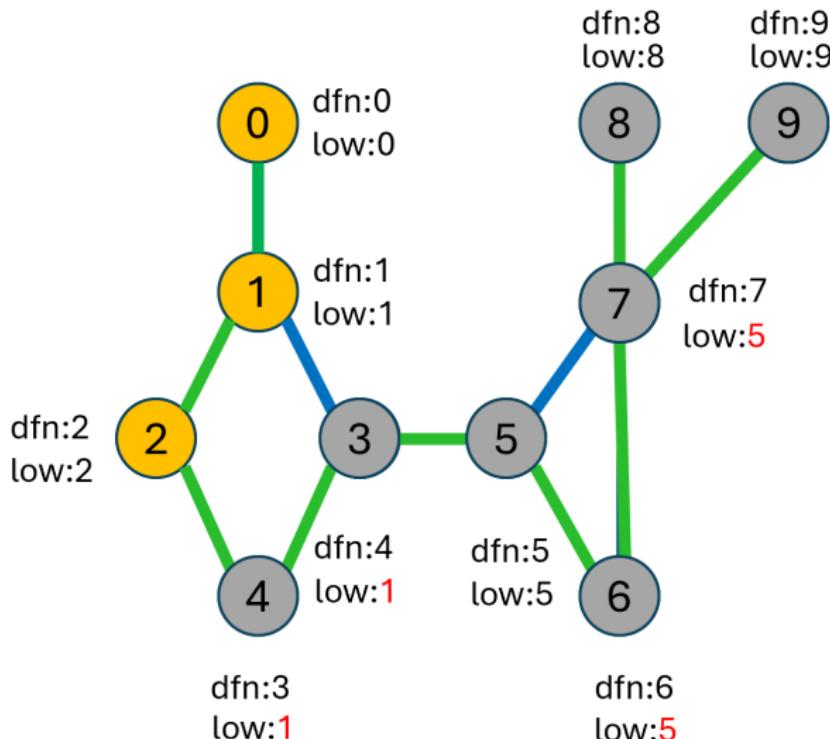


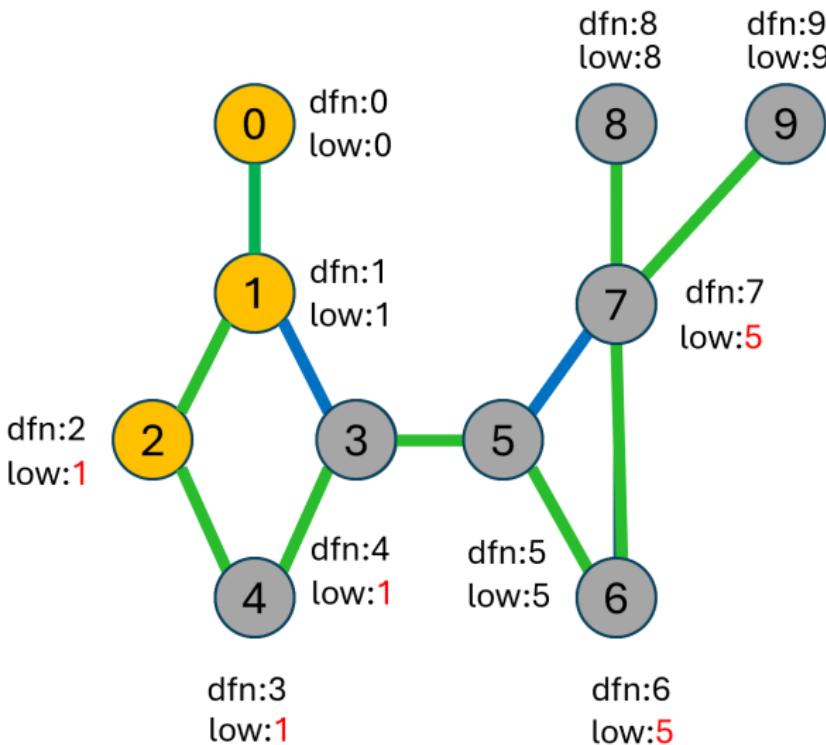


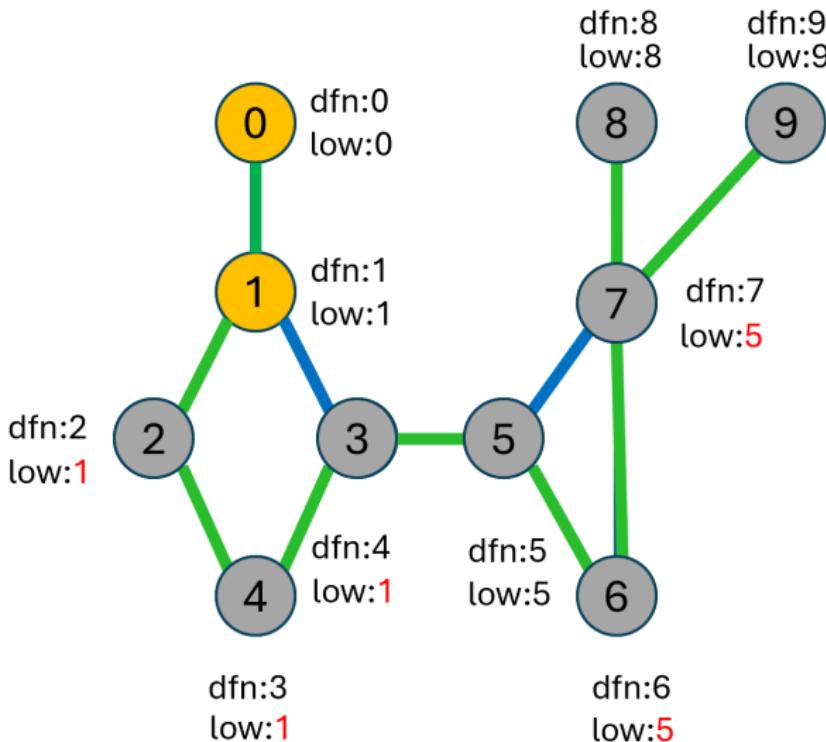


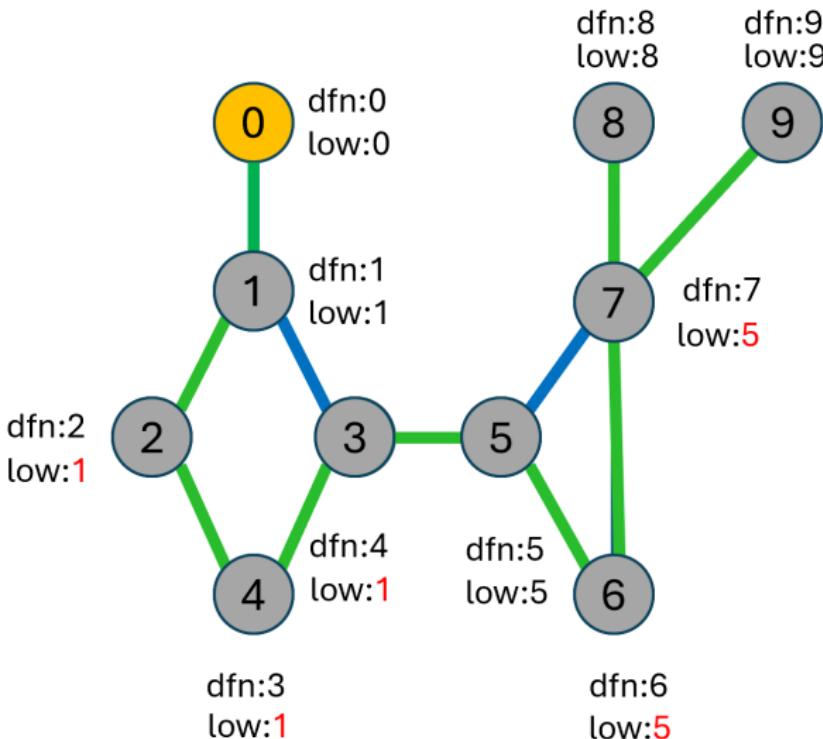


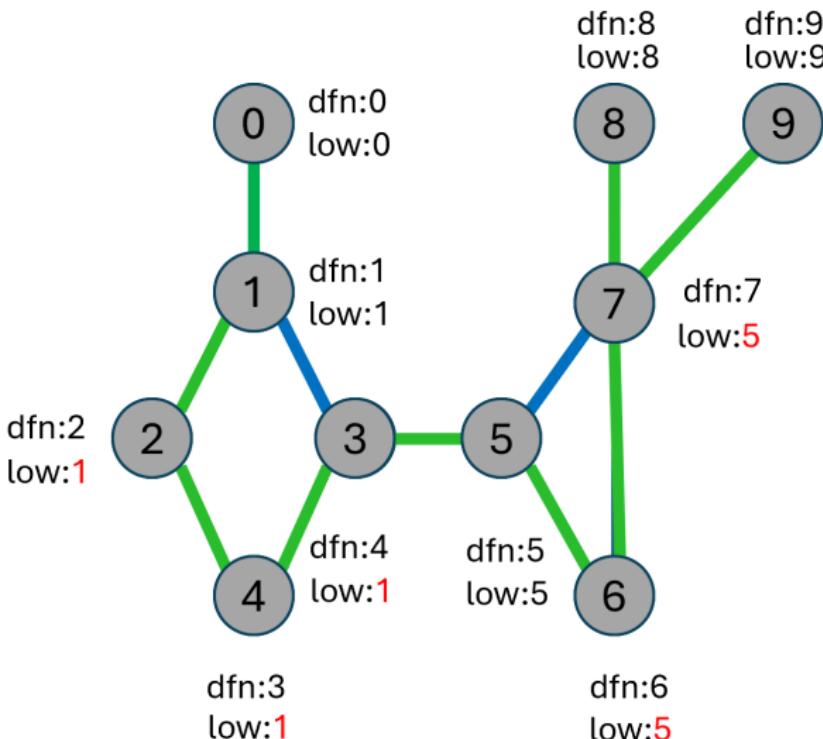












# The codes for computing dfn and low

Time complexity:  $O(e)$ .

```
void dfn_low(int u, int v) {
    /* compute dfn and low while performing a dfs
    search beginning at vertex u, v is the parent
    of u (if any) */
    node_pointer ptr;
    int w;
    dfn[u] = low[u] = num++;
    for (ptr = graph[u]; ptr; ptr = ptr->link) {
        w = ptr->vertex;
        if (dfn[w] < 0) {
            /* w is an unvisited vertex */
            dfn_low(w, u); // visit w from u
            low[u] = MIN2(low[u], low[w]); // (*)
        } else if (w != v)
            low[u] = MIN2(low[u], dfn[w]); // (**)
    }
}
```

```
short int dfn [MAX_VERTICES];
short int low[MAX_VERTICES];
int num = 0;

void init(void) {
    int i;
    for(i = 0; i < n; i++) {
        visited[i] = false;
        dfn[i] = low[i] = -1;
    }
    num = 0;
}
```

bootstrapping by

`dfn_low(x, -1)`



# Finding articulation points (3/3)

## articulation points

$u$  is an articulation point iff one of the following conditions are satisfied:

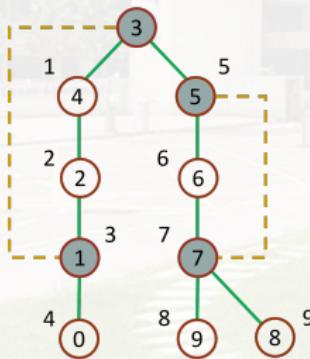
- $u$  is the root of the spanning tree and has two or more children.
- $u$  is not the root of the spanning tree and  $u$  has a child  $w$  such that  $\text{low}(w) \geq \text{dfn}(u)$ .

# Finding articulation points (3/3)

## articulation points

$u$  is an articulation point iff one of the following conditions are satisfied:

- $u$  is the root of the spanning tree and has two or more children.
- $u$  is not the root of the spanning tree and  $u$  has a child  $w$  such that  $\text{low}(w) \geq \text{dfn}(u)$ .



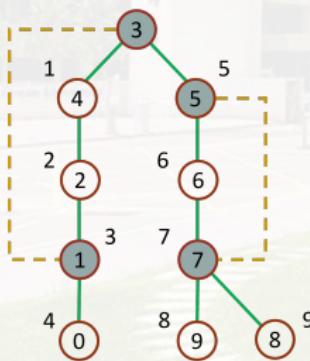
vertex	0	1	2	3	4	5	6	7	8	9
dfn	4	3	2	0	1	5	6	7	9	8
low	4	0	0	0	0	5	5	5	9	8

# Finding articulation points (3/3)

## articulation points

$u$  is an articulation point iff one of the following conditions are satisfied:

- $u$  is the root of the spanning tree and has two or more children.
- $u$  is not the root of the spanning tree and  $u$  has a child  $w$  such that  $\text{low}(w) \geq \text{dfn}(u)$ .



vertex	0	1	2	3	4	5	6	7	8	9
dfn	4	3	2	0	1	5	6	7	9	8
low	4	0	0	0	0	5	5	5	9	8

- articulation points: 1, 3, 5, 7.

## Connected Components

Finding the articulation points

Code for Biconnected Components ( $O(n + e)$  time)

```

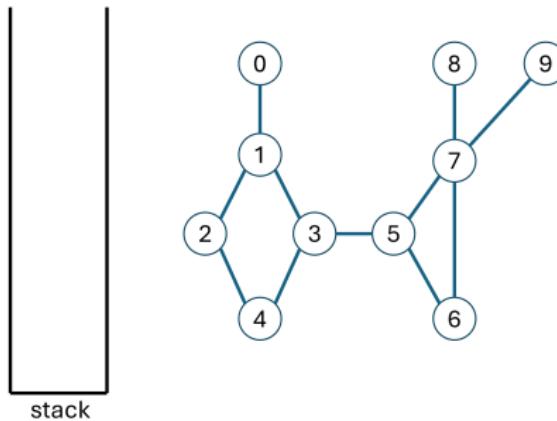
void bicon(int u, int v) { /* dfn[] = -1, num = 0, s is an empty stack initially*/
/* v is the parent node of u in the DFS spanning tree */
    nodePointer ptr;
    int w, x, y;
    dfn[u] = low[u] = num++;
    for (ptr = graph[u]; ptr; ptr = ptr->link) {
        w = ptr->vertex;
        if (v != w && dfn[w] < dfn[u]) {
            push(u,w); /* add edge (u,w) into stack s */
            if (dfn[w] < 0) { /* w is not visited yet */
                bicon(w, u);
                low[u] = MIN2(low[u],low[w]);
                if (low[w] >= dfn[u]) {
                    printf("New biconnected component:");
                    do { /* pop an edge from stack s */
                        pop(&x, &y);
                        printf("<%d,%d>", x, y);
                    } while (!(x == u) && (y == w));
                    printf("\n");
                }
            } else if (w != v)
                low[u] = MIN2(low[u],dfn[w]);
        }
    }
}

```

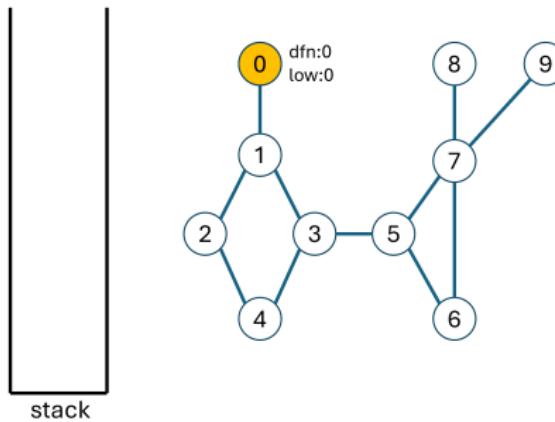
If we have  $\text{low}(w) \geq \text{dfn}(v)$  whenever  $\text{dfn\_low}(u, w)$  returns, then a new biconnected component is found!



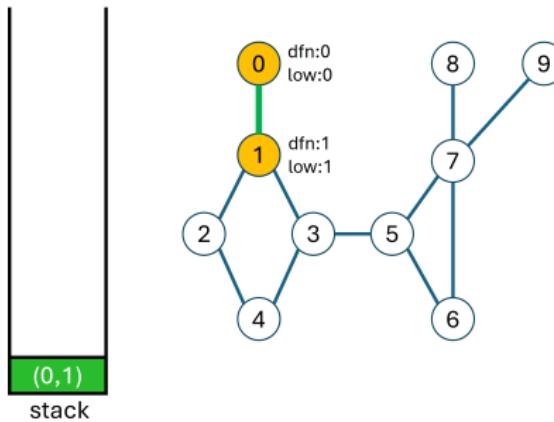
# Illustration



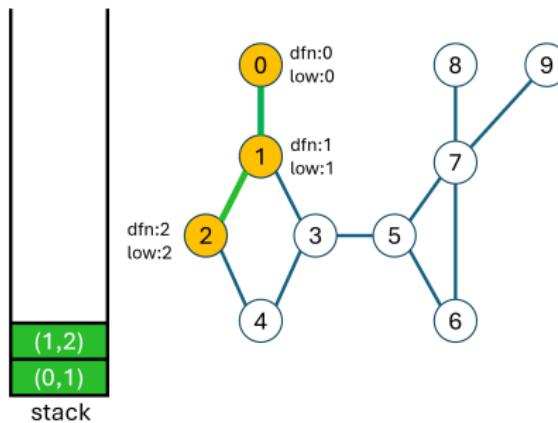
# Illustration



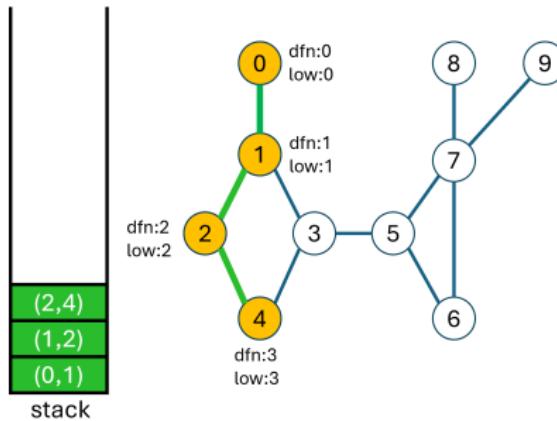
# Illustration



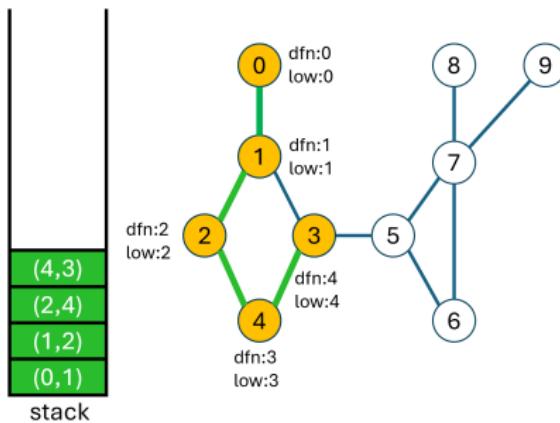
# Illustration



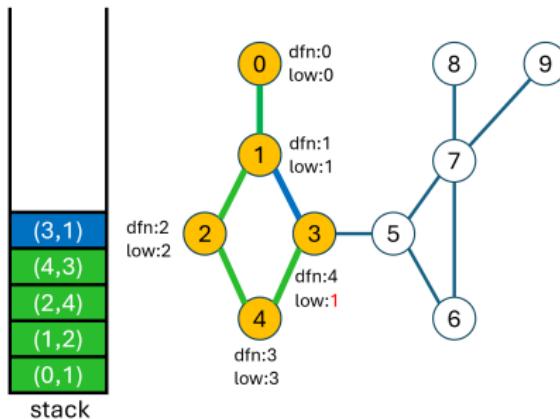
# Illustration



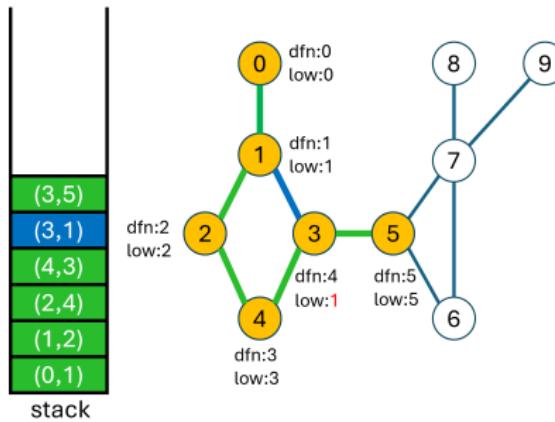
# Illustration



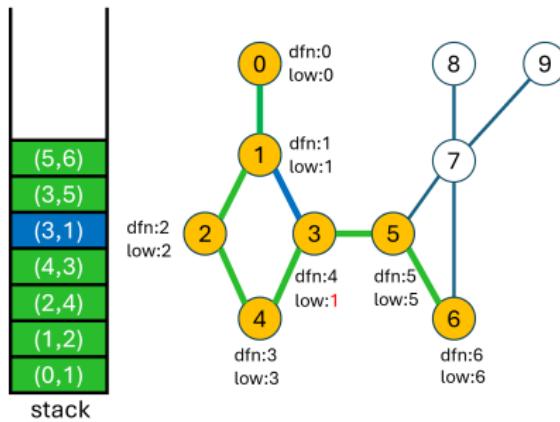
# Illustration



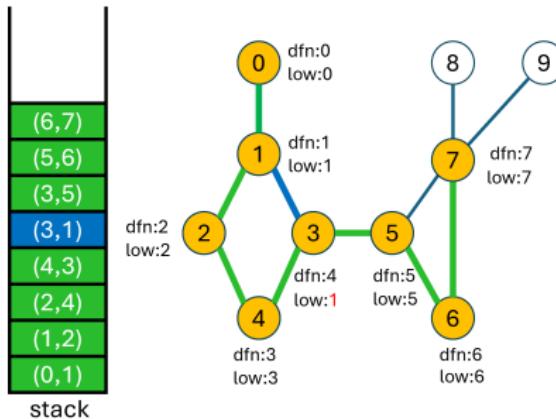
# Illustration



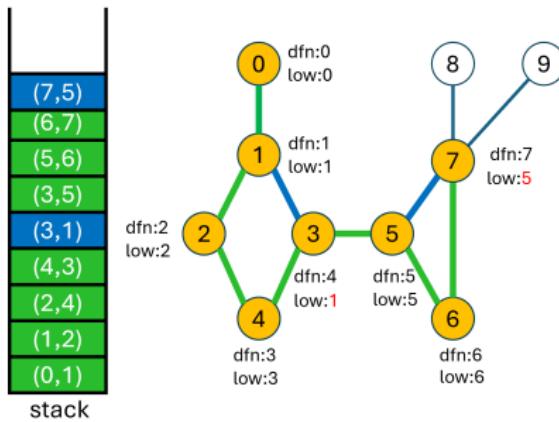
# Illustration



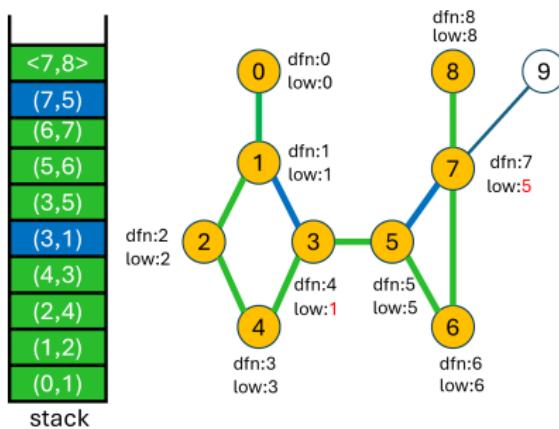
# Illustration



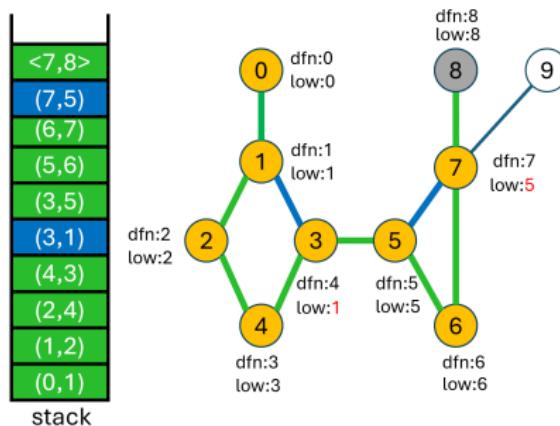
# Illustration



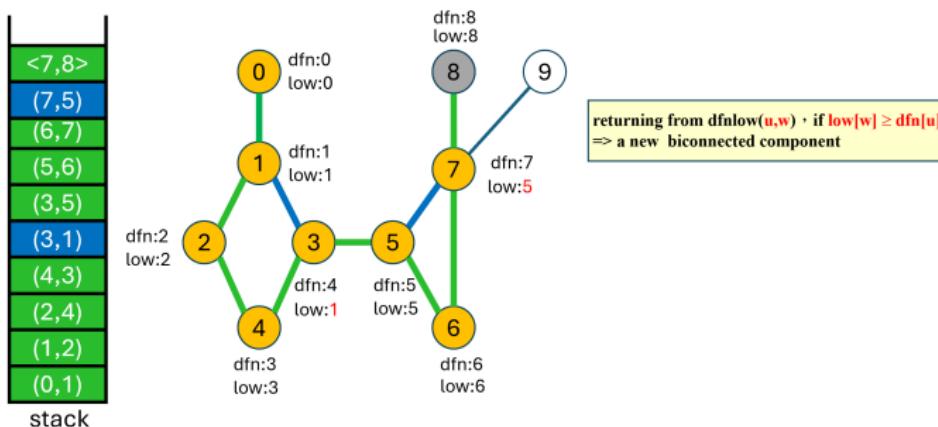
# Illustration



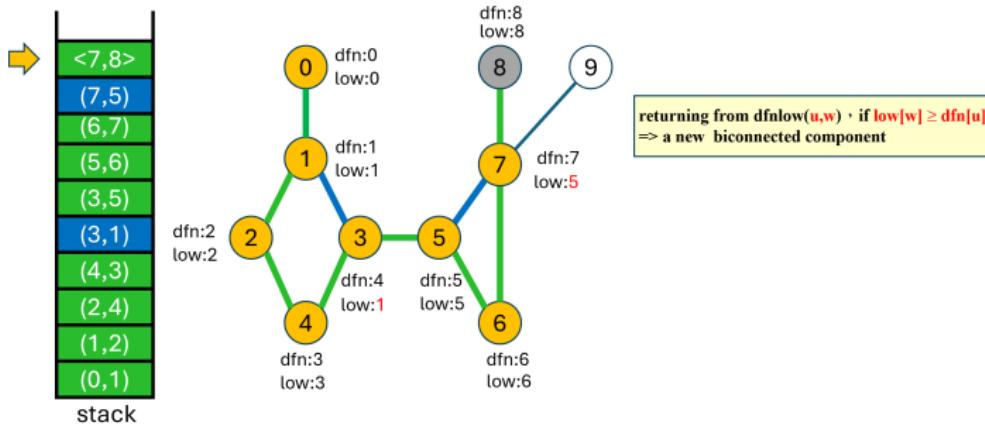
# Illustration



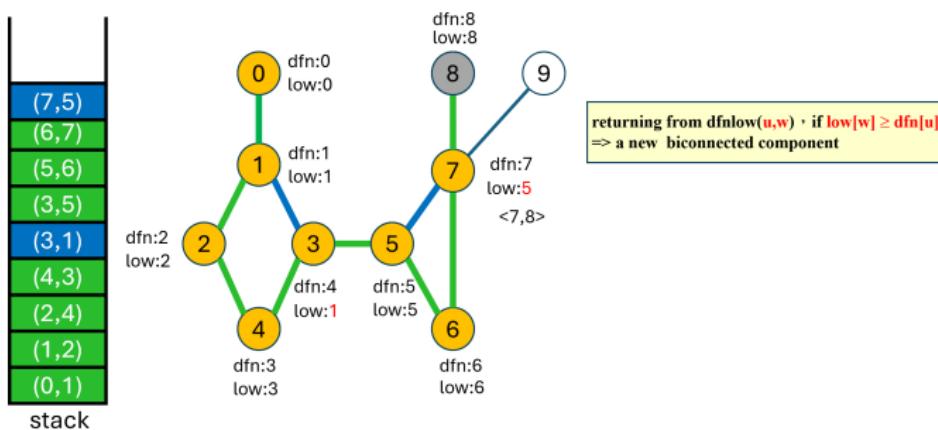
# Illustration



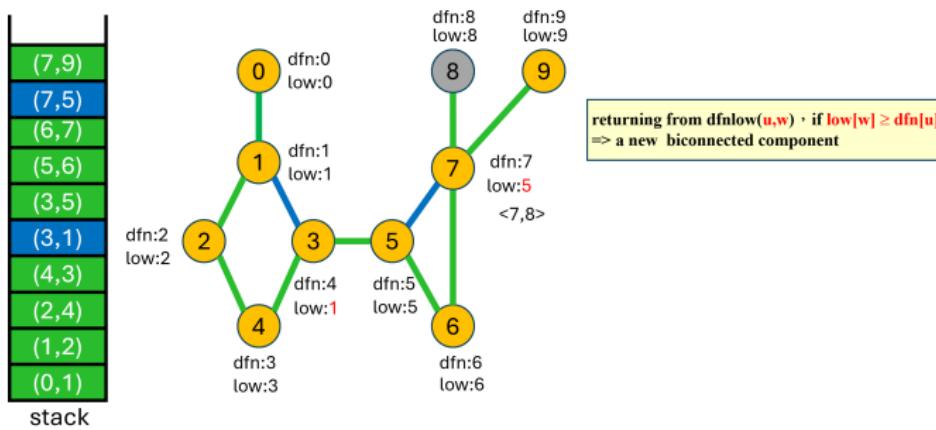
# Illustration



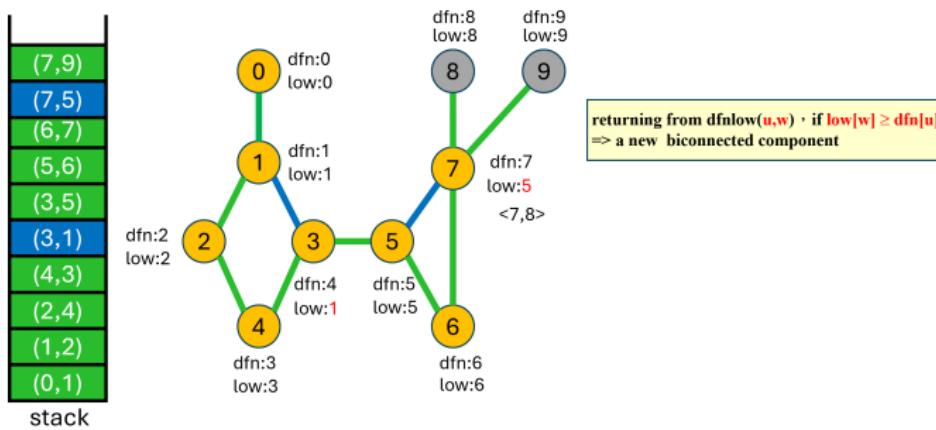
# Illustration



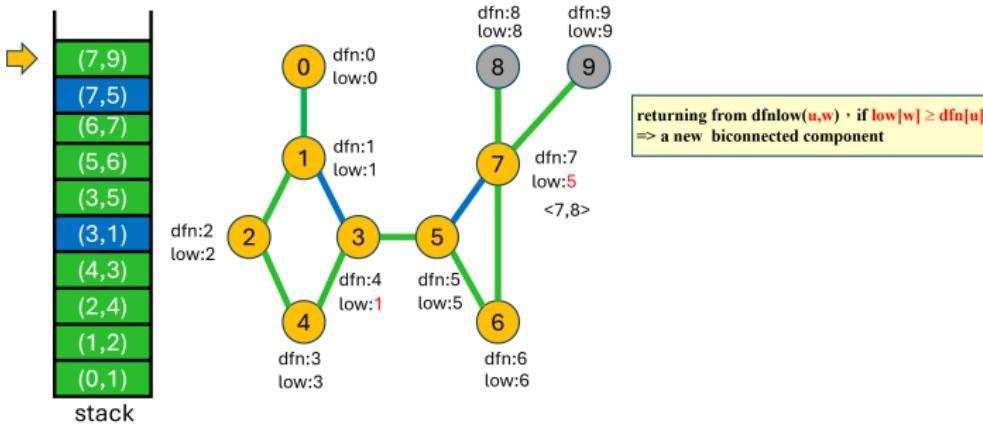
# Illustration



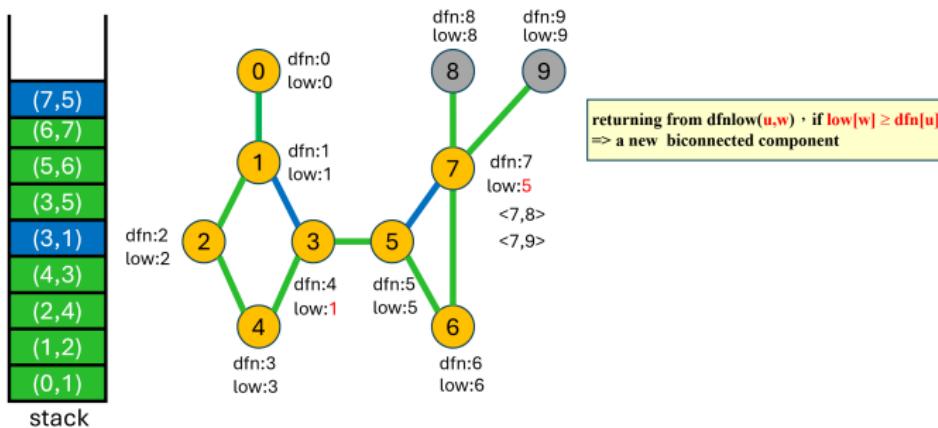
# Illustration



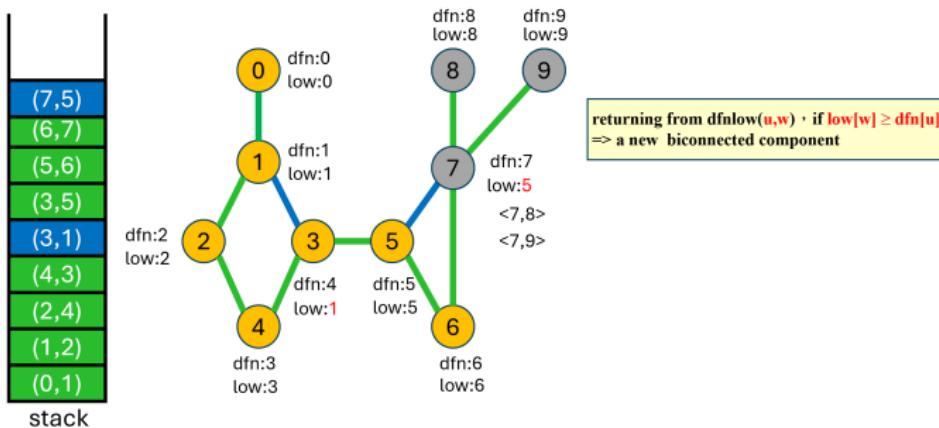
# Illustration



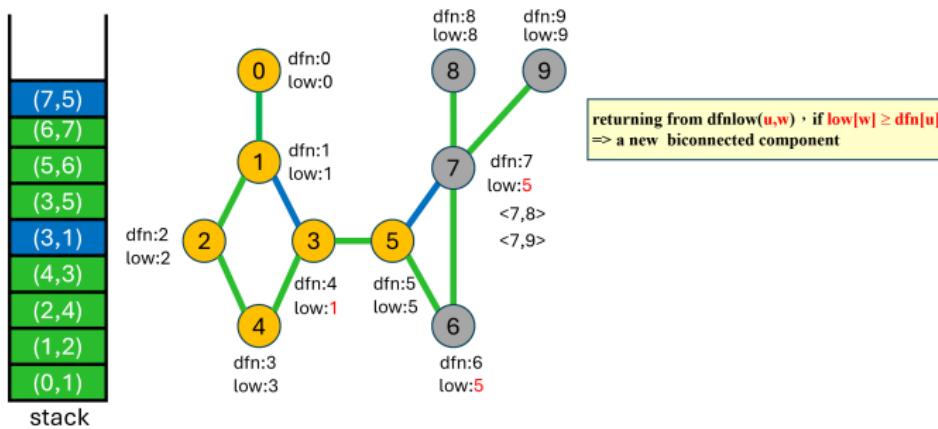
# Illustration



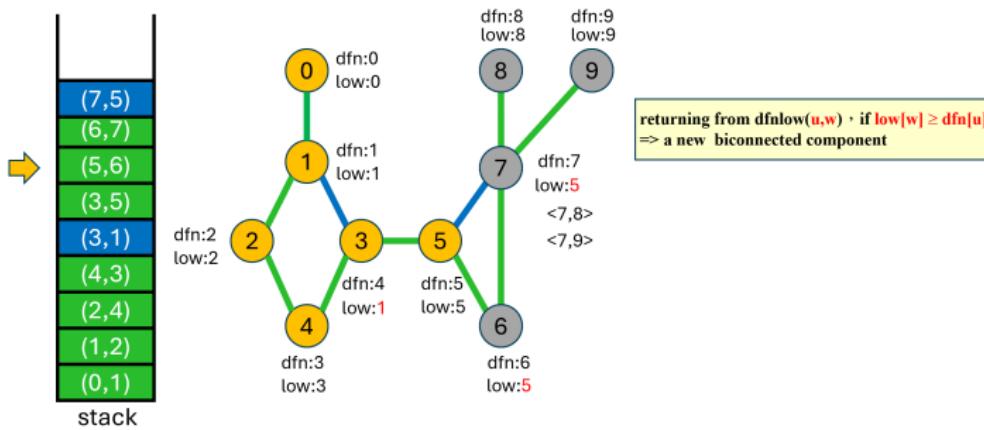
# Illustration



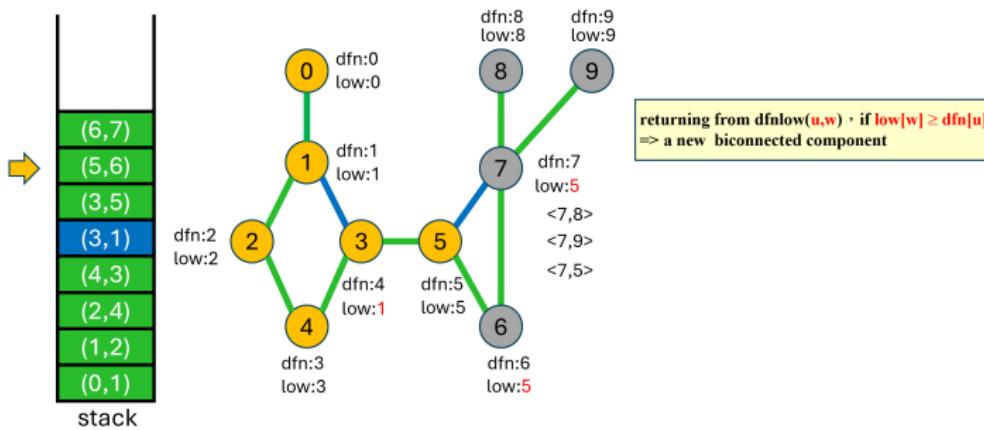
# Illustration



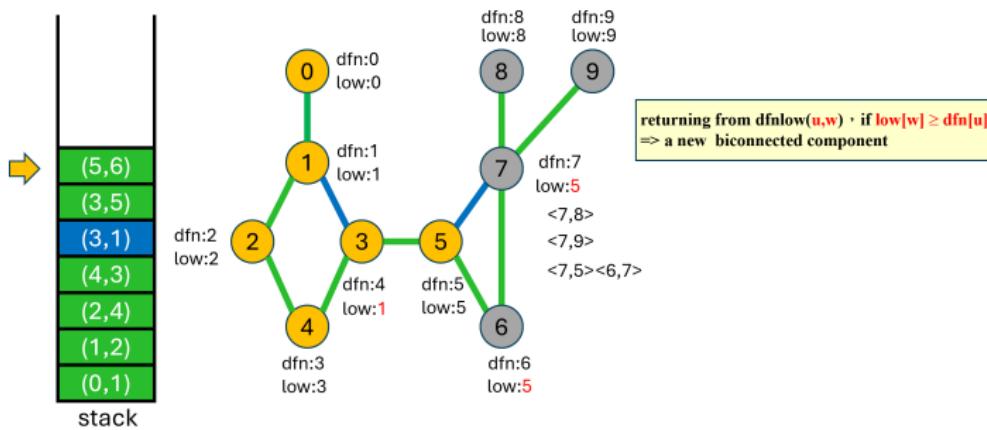
# Illustration



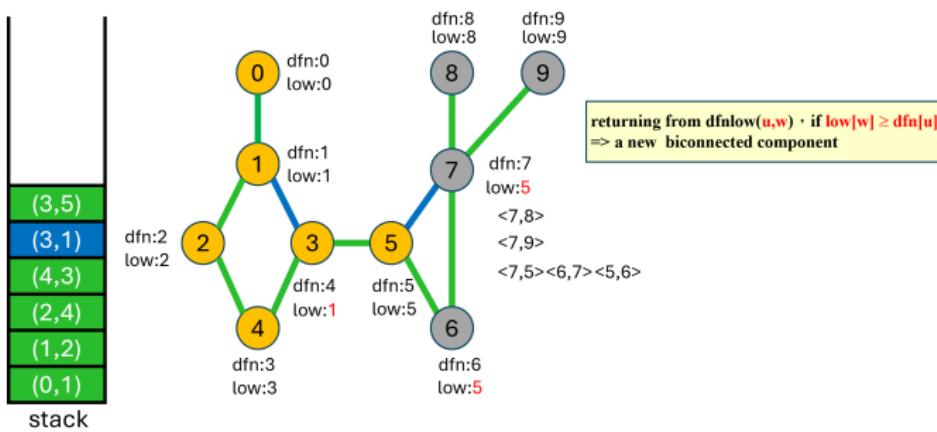
# Illustration



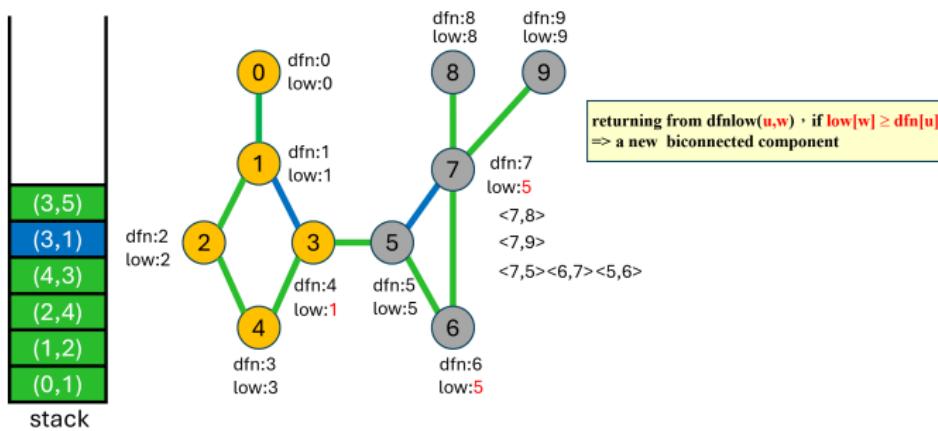
# Illustration



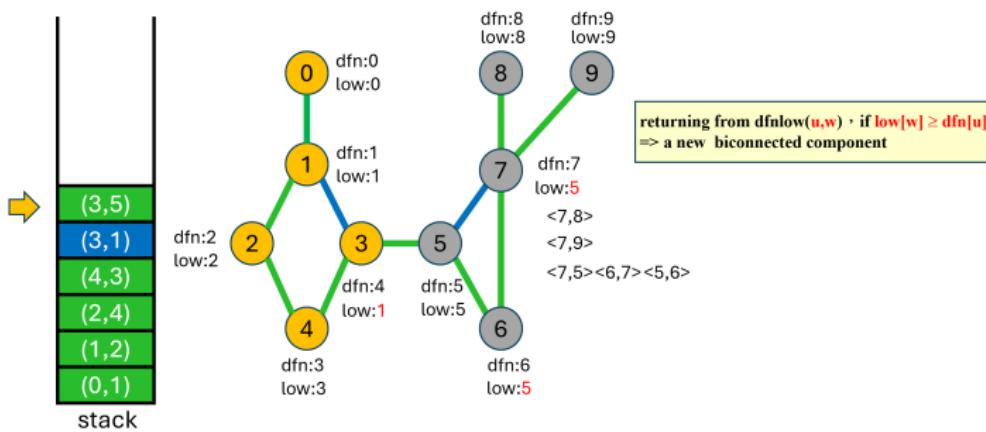
# Illustration



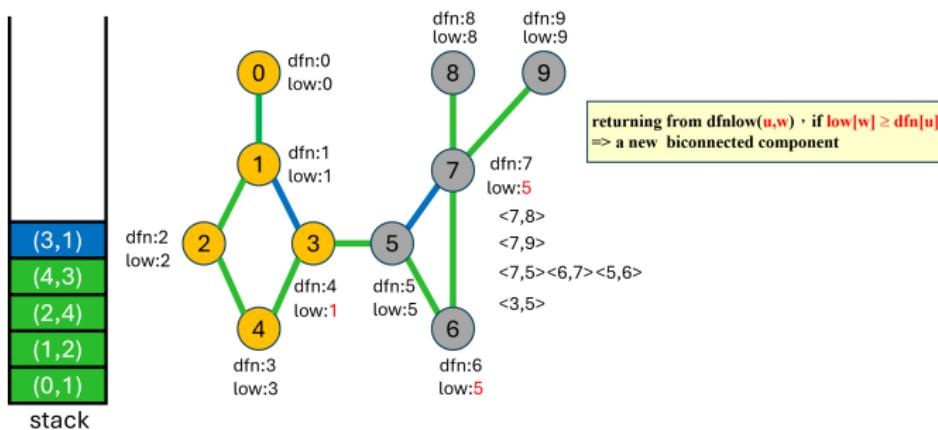
# Illustration



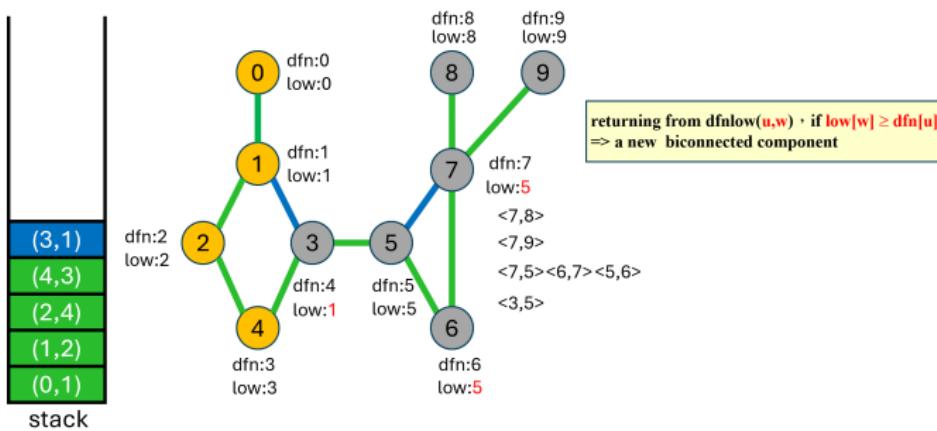
# Illustration



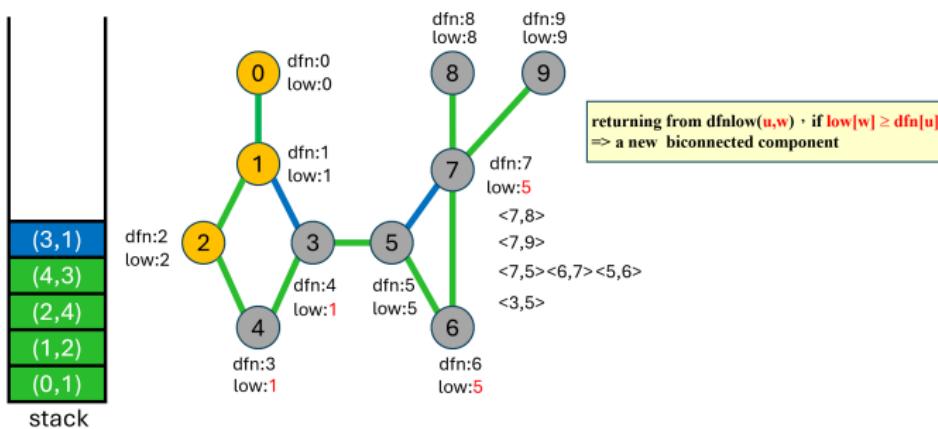
# Illustration



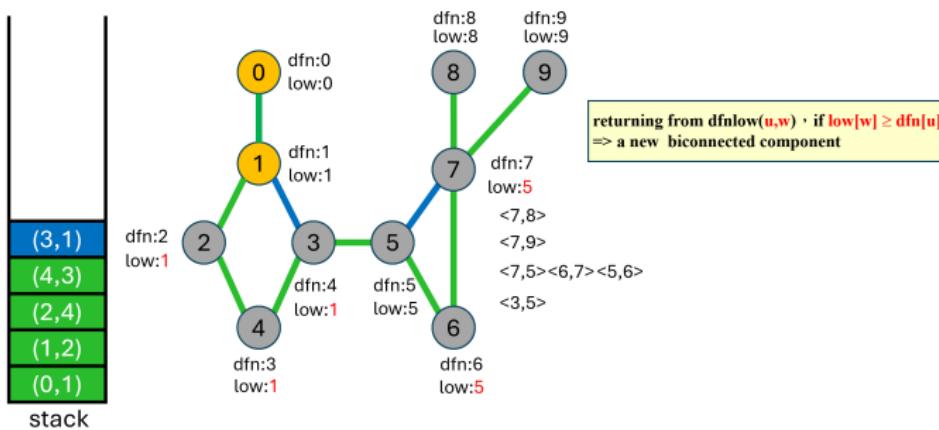
# Illustration



# Illustration



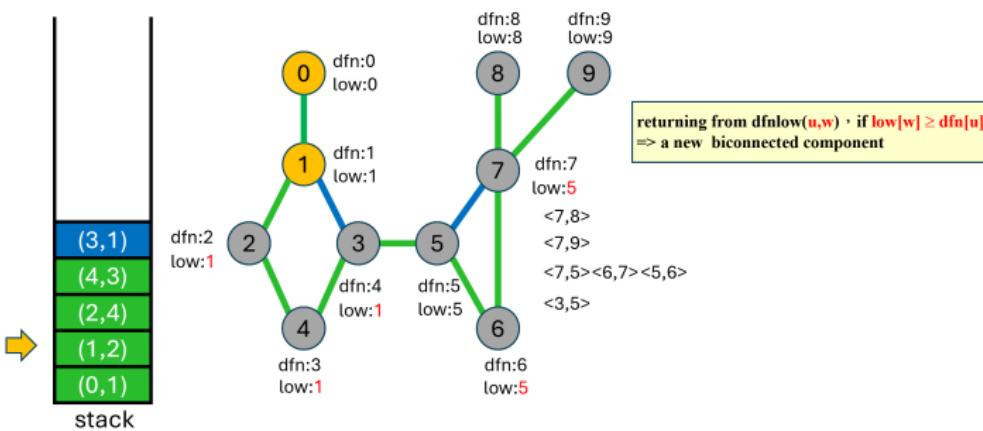
# Illustration



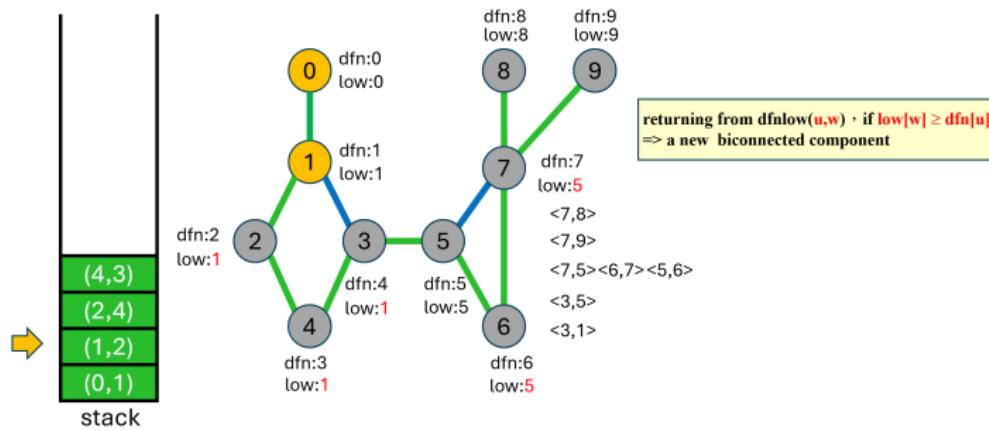
## Connected Components

Finding the articulation points

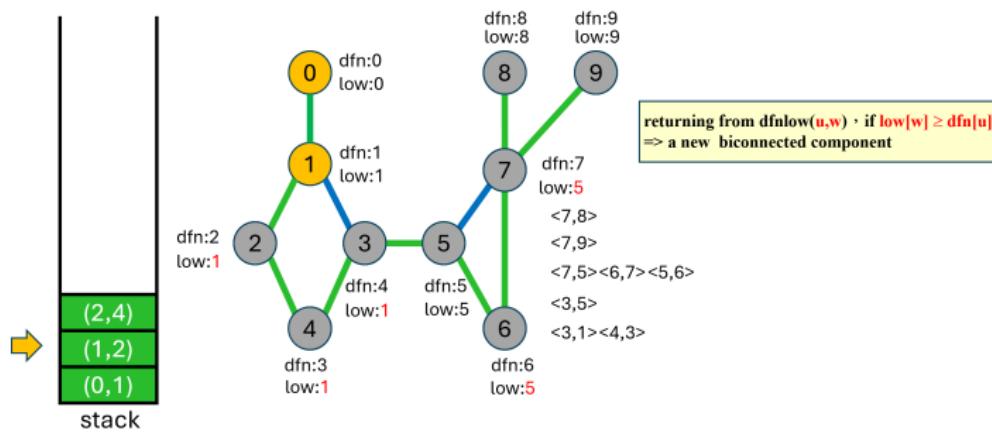
## Illustration



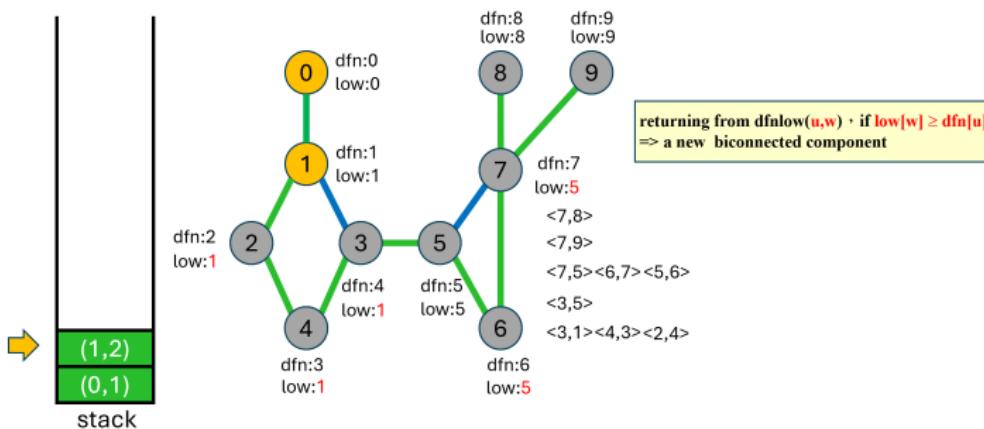
# Illustration



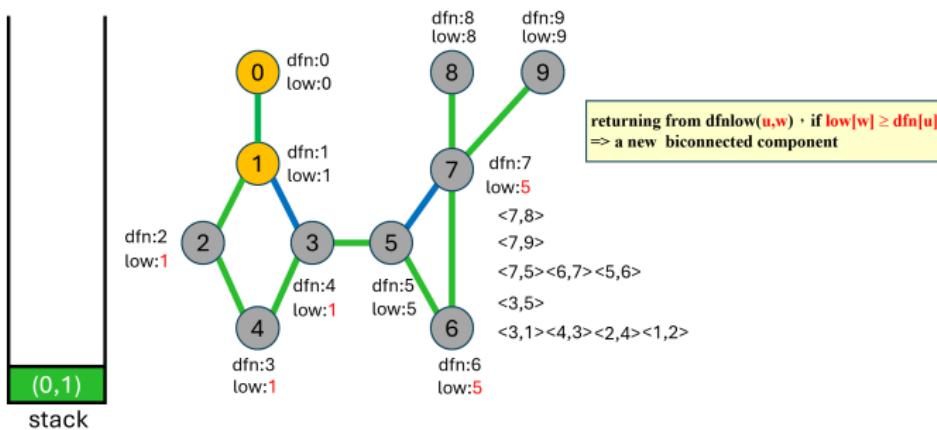
# Illustration



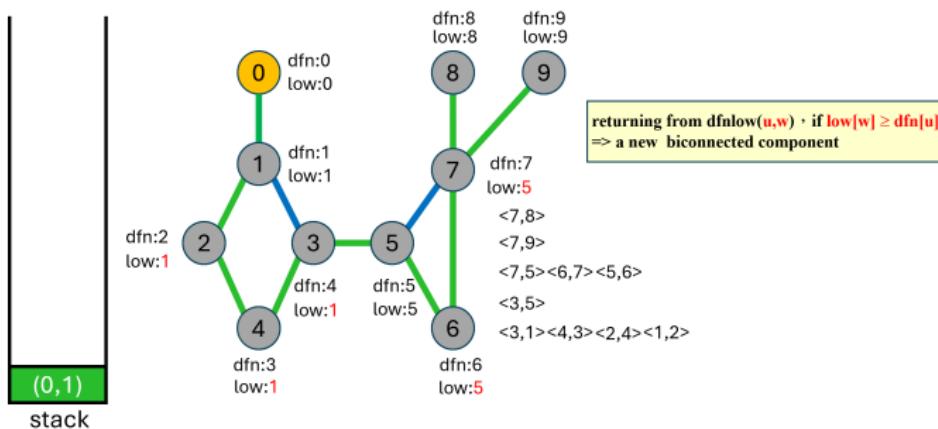
# Illustration



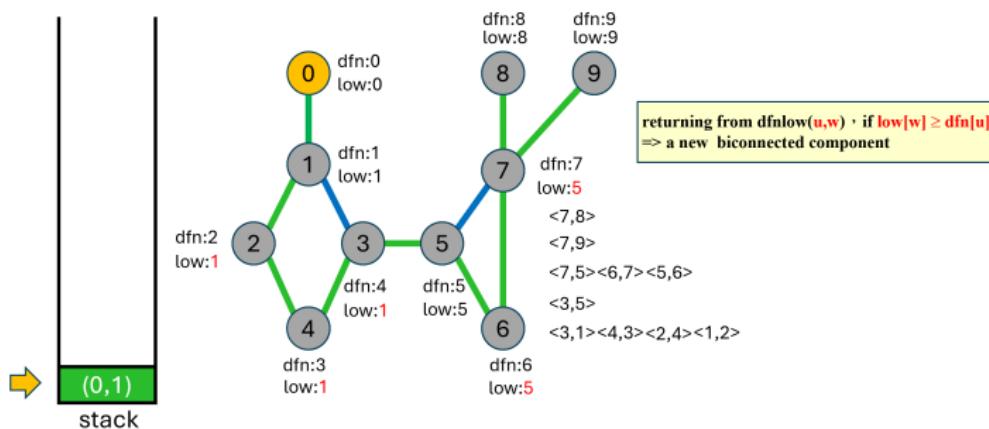
# Illustration



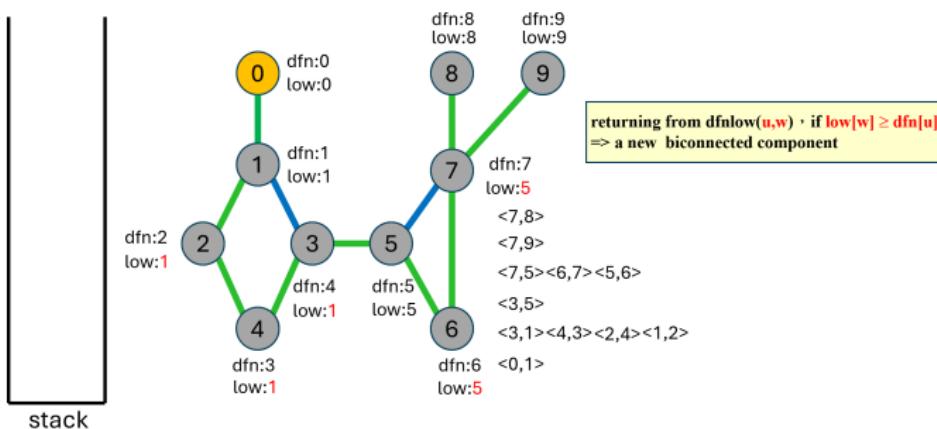
# Illustration



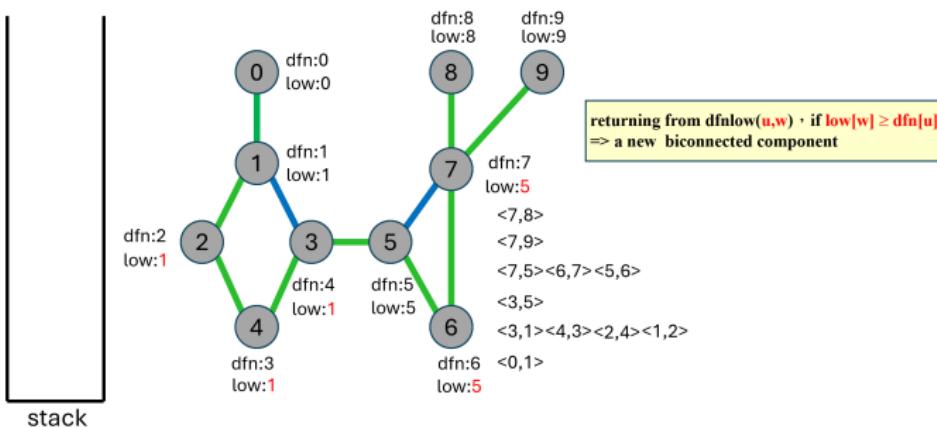
# Illustration



# Illustration



# Illustration



# Discussions

