Connected Components & Spanning Trees

Joseph Chuang-Chieh Lin (林莊傑)

Department of Computer Science & Engineering,
National Taiwan Ocean University

Fall 2024

Outline

Outline

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 `dfs(v)` or `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 `dfs(`v`)` or `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 `dfs(`v`)` or `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.

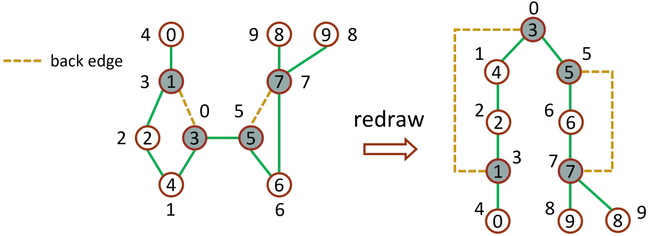
This can be done by making repeated calls to either `dfs(`v`)` or `bfs(`v`)` where v is an unvisited vertex.

Finding articulation points (1/3)

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

Back edges

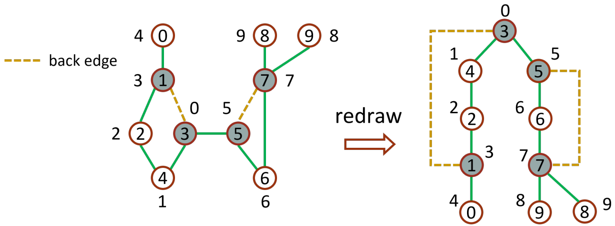
- Tree edges: DFS
- Nontree edges: we call them back edges



(a) depth first spanning tree **(dfs(3))** (b)

Observations

- The root of a depth first spanning tree is an articulation point if and only if it has ≥ 2 children.
- Any other vertex u is an articulation point if and only if it has ≥ 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.

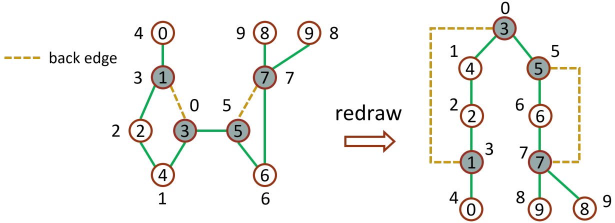


(a) depth first spanning tree **(dfs(3))** (b)

- 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 ≥ 2 children.
- Any other vertex u is an articulation point if and only if it has ≥ 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.



(a) depth first spanning tree **(dfs(3))** (b)

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

Finding articulation points (2/3)

dfn(v)

The depth first numbers, or dfn, of the vertices give the sequence in which the vertices are visited during the depth first search.

Finding articulation points (2/3)

dfn(v)

The depth first numbers, or dfn, of the vertices give the sequence in which the vertices are visited during the depth first search.

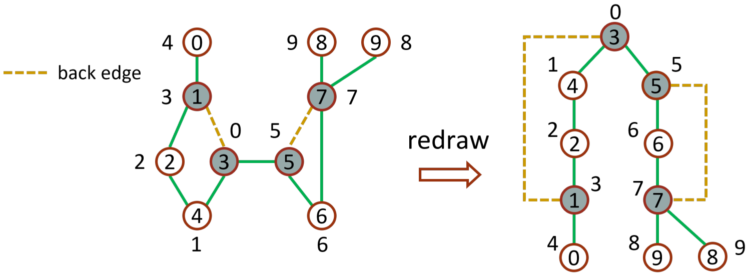
- If u is an ancestor of v in the depth first spanning tree, then $\text{dfn}(u) < \text{dfn}(v)$.

low(v)

The 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:

$$\text{low}(u) = \min \begin{cases} \text{dfn}(u), \\ \min\{\text{low}(w) \mid w \text{ is a child of } u\}, \\ \min\{\text{dfn}(w) \mid (u, w) \text{ is a back edge}\} \end{cases}$$

Example of Computing dfn and low values



(a) depth first spanning tree **(dfs(3))** (b)

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);
            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:

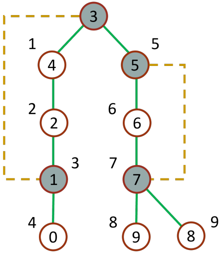
- u is the root of the spanning tree and has two or more children.
- u is not the root of the spanning tree and u has a child w such that $\mathrm{low}(w) \geq \mathrm{dfn}(u)$.

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 $\text{low}(w) \geq \text{dfn}(u)$.



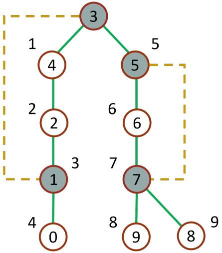
vertex	0	1	2	3	4	5	6	7	8	9
dfn	4	3	2	0	1	5	6	7	9	8
low	4	0	0	0	0	5	5	5	9	8

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.

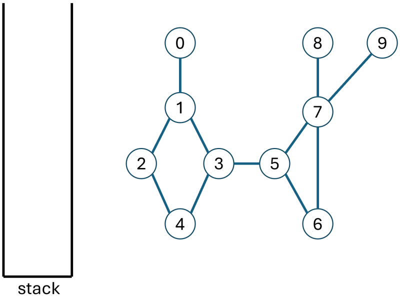
Algorithm for Finding Biconnected Components

If we have $\mathrm{low}[w] \geq \mathrm{dfn}(v)$ when $\texttt{dfn\_low}(u, w)$ returns.

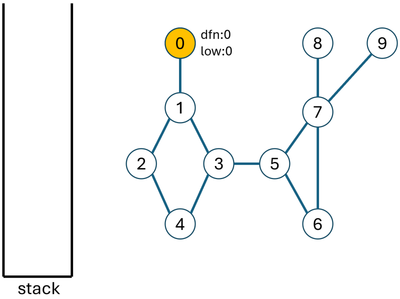
Code for Biconnected Components ($O(n + e)$ time)

```c
void bicon(int u, int v) { /* dfn[] = -1, num = 0, s is an empty stack initially*/
    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]);
        }
    }
}
```

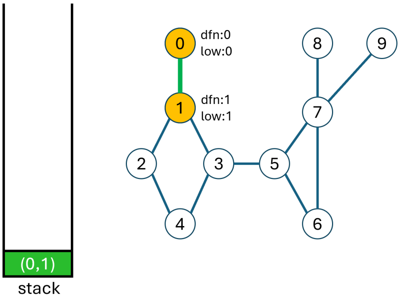
Illustration



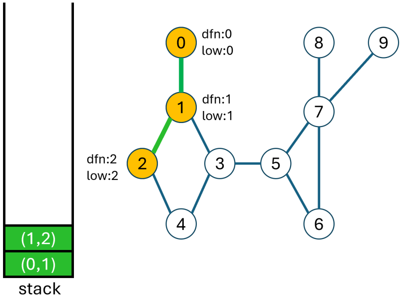
Illustration



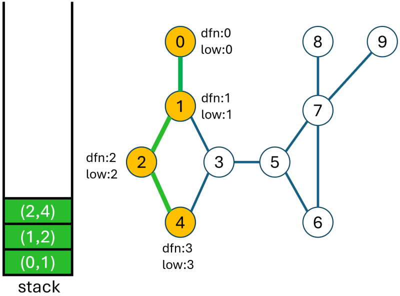
Illustration



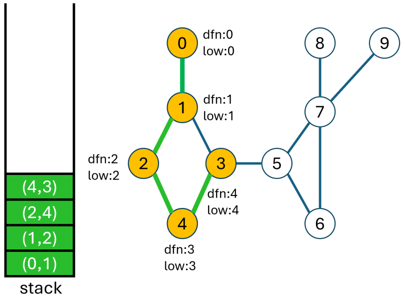
Illustration



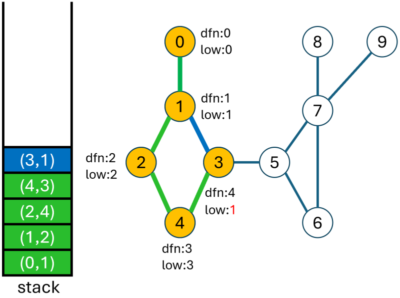
Illustration



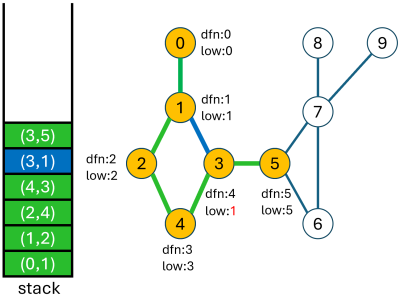
Illustration



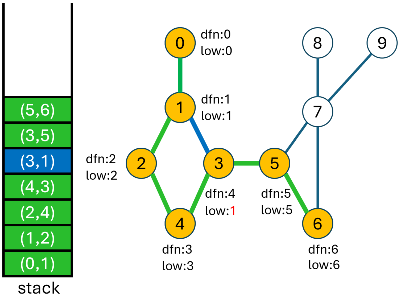
Illustration



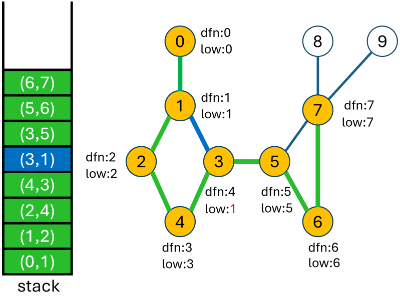
Illustration



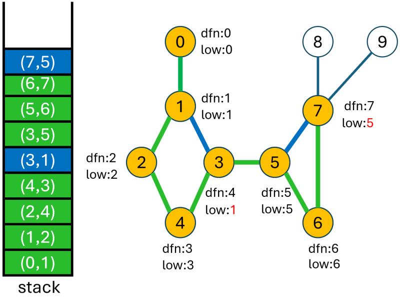
Illustration



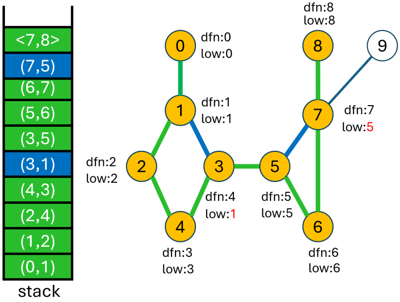
Illustration



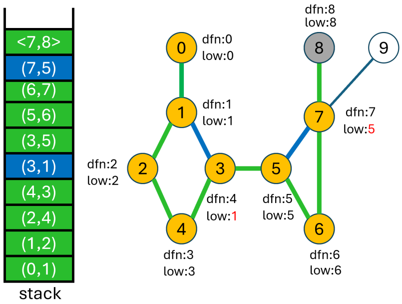
Illustration



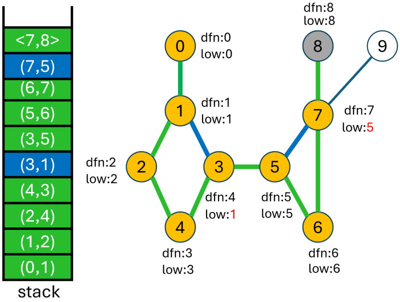
Illustration



Illustration

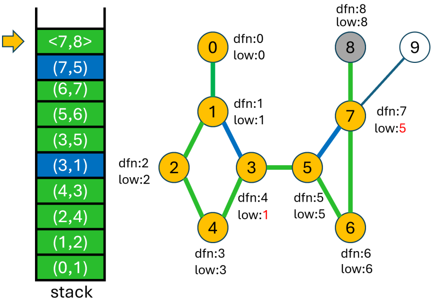


Illustration



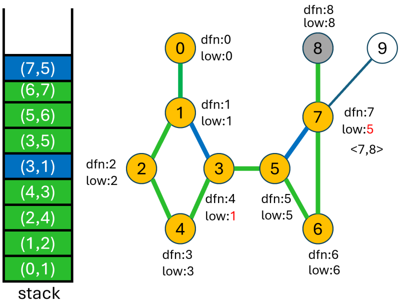
returning from dfnlow(u,w) , if low[w] ≥ dfn[u]
=> a new biconnected component

Illustration



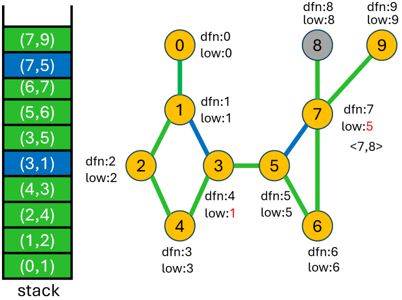
returning from dfnlow(u,w), if low[w] ≥ dfn[u]
=> a new biconnected component

Illustration



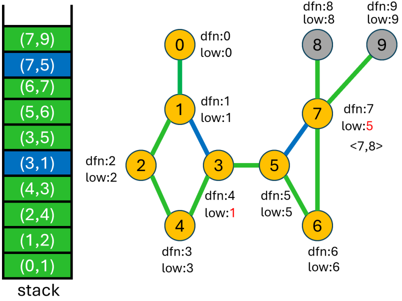
returning from dfnlow(u,w), if low[w] ≥ dfn[u]
=> a new biconnected component

Illustration



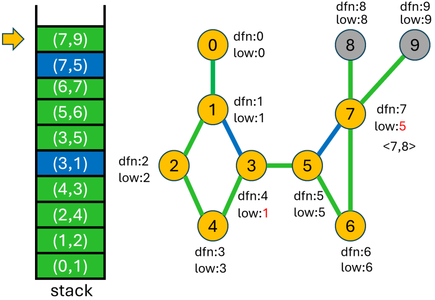
returning from dfnlow(u,w), if low[w] ≥ dfn[u]
=> a new biconnected component

Illustration



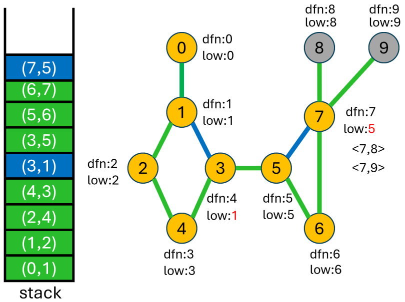
returning from dfnlow(u,w), if low[w] ≥ dfn[u]
=> a new biconnected component

Illustration



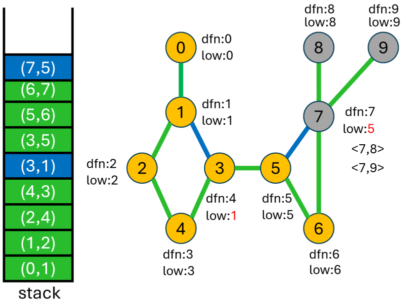
returning from dfnlow(u,w), if low[w] ≥ dfn[u]
=> a new biconnected component

Illustration



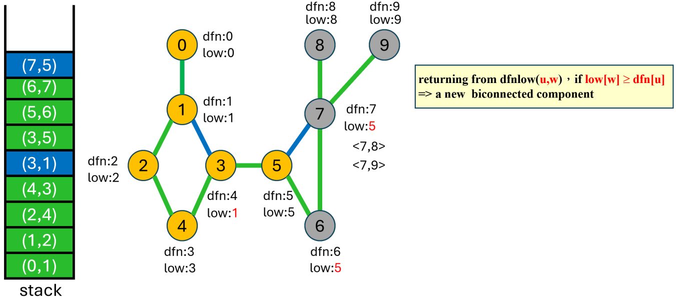
returning from dfnlow(u,w) , if low[w] ≥ dfn[u]
=> a new biconnected component

Illustration

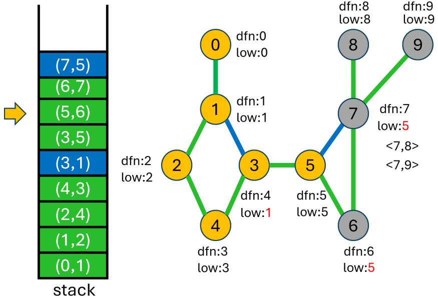


returning from dfnlow(u,w), if low[w] ≥ dfn[u]
=> a new biconnected component

Illustration

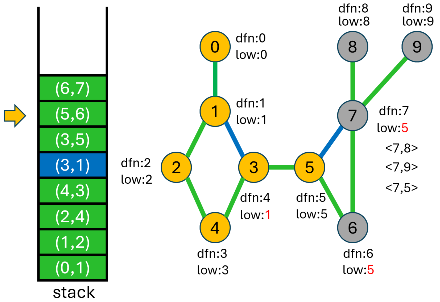


Illustration



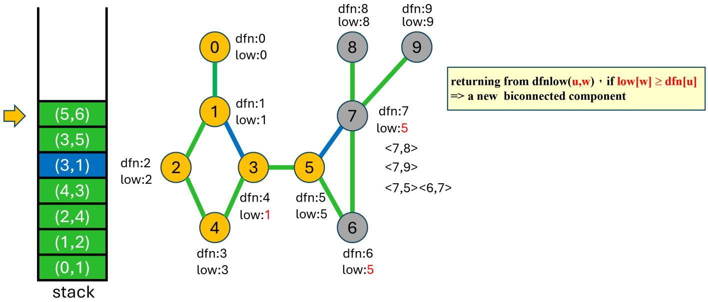
returning from dfnlow(u,w)，if low[w] ≥ dfn[u]
=> a new biconnected component

Illustration

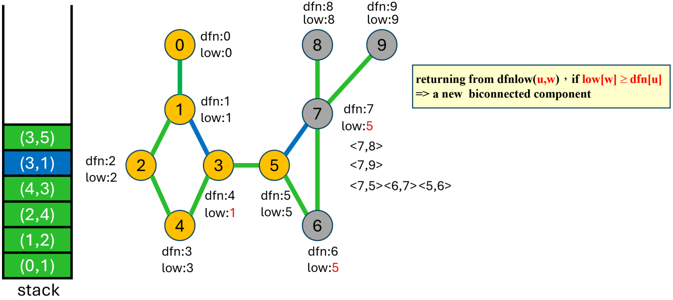


returning from dfnlow(u,w) , if low[w] ≥ dfn[u]
=> a new biconnected component

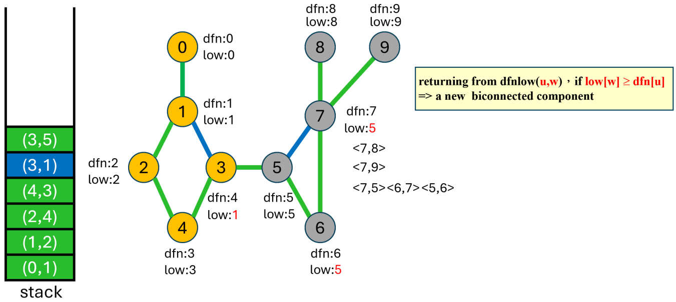
Illustration



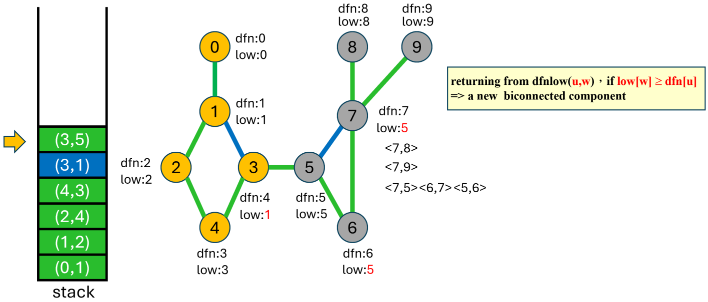
Illustration



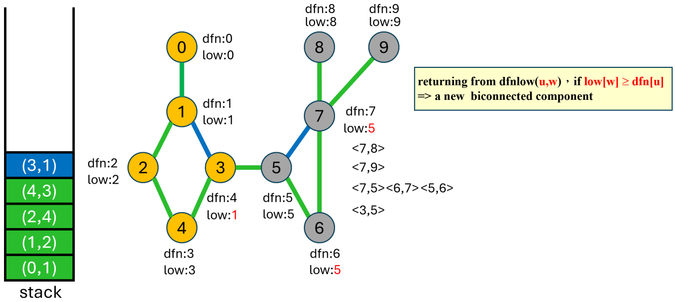
Illustration



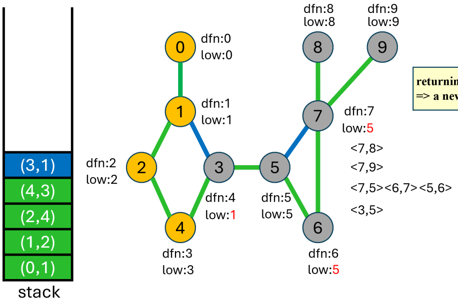
Illustration



Illustration

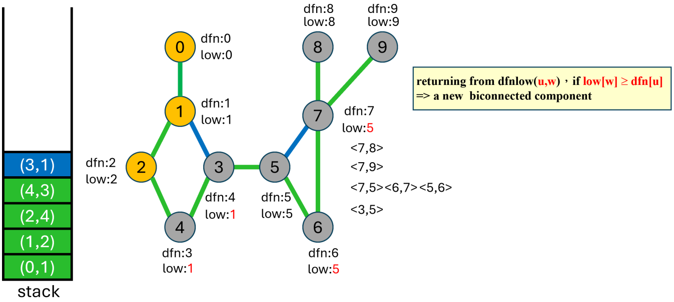


Illustration

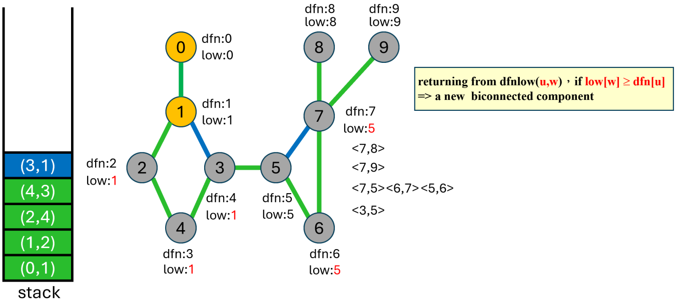


returning from dfnlow(u,w)，if low[w] ≥ dfn[u]
=> a new biconnected component

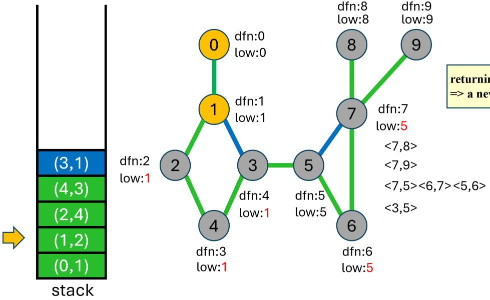
Illustration



Illustration

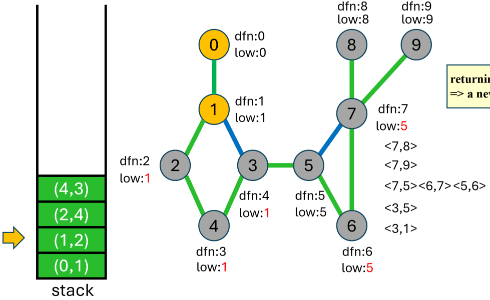


Illustration



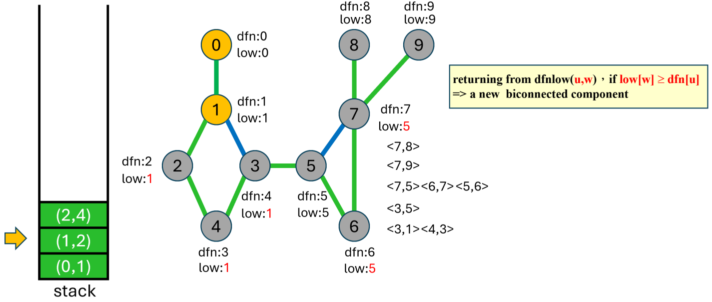
returning from dfnlow(u,w) , if low[w] ≥ dfn[u]
=> a new biconnected component

Illustration

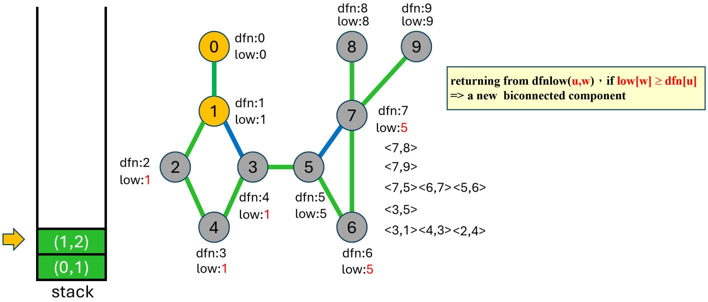


returning from dfnlow(u,w)，if low[w] ≥ dfn[u]
=> a new biconnected component

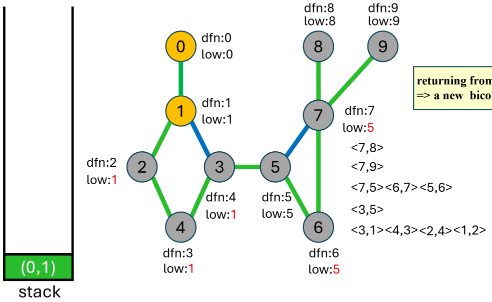
Illustration



Illustration

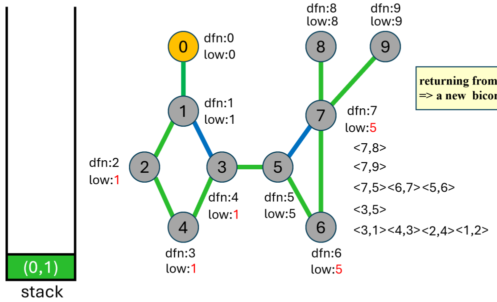


Illustration



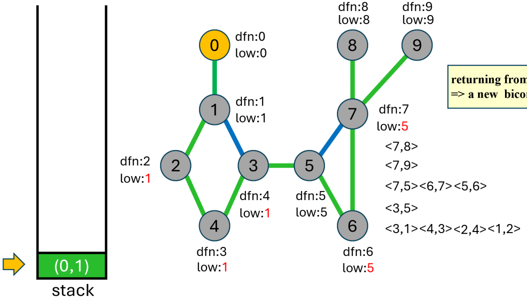
returning from dfnlow(u,w)，if low[w] ≥ dfn[u]
=> a new biconnected component

Illustration



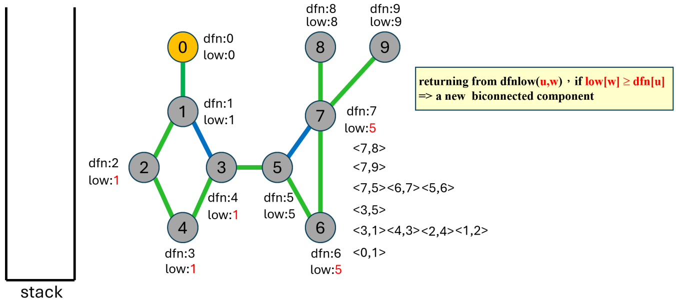
returning from dfnlow(u,w)，if low[w] ≥ dfn[u]
=> a new biconnected component

Illustration

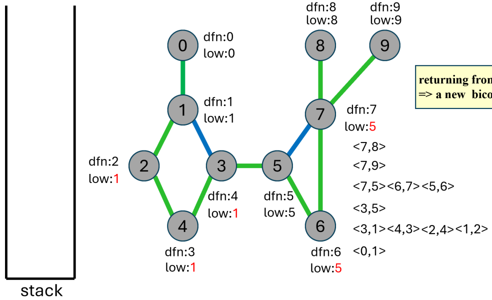


returning from dfnlow(u,w), if low[w] ≥ dfn[u]
=> a new biconnected component

Illustration



Illustration



returning from dfnlow(u,w) , if low[w] ≥ dfn[u]
=> a new biconnected component

Discussions