



# Advanced algorithms

strongly connected components algorithms,  
Euler trail,  
Hamiltonian path

Jiří Vyskočil, Radek Mařík

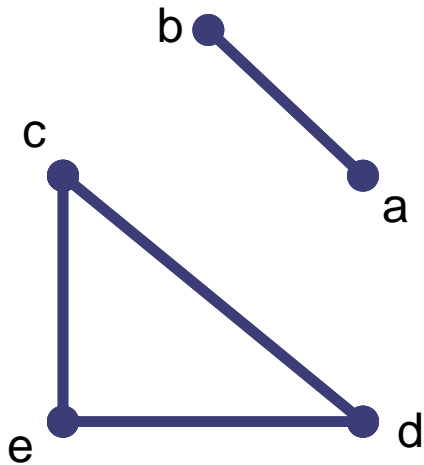
2012

# Connected component

- A connected component of graph  $G = (V, E)$  with regard to vertex  $v$  is a set

$\mathcal{C}(v) = \{u \in V \mid \text{there exists a path in } G \text{ from } u \text{ to } v\}$ .

- In other words: If a graph is disconnected, then parts from which is composed from and that are themselves connected, are called *connected components*.



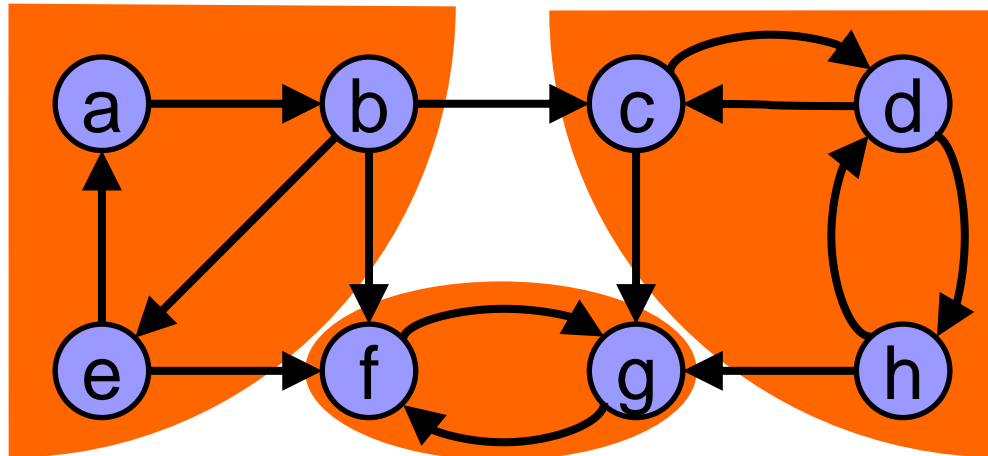
$$\mathcal{C}(a) = \mathcal{C}(b) = \{a, b\}$$

$$\mathcal{C}(c) = \mathcal{C}(d) = \mathcal{C}(e) = \{c, d, e\}$$

# Strongly Connected Components

- A directed graph  $G = (V, E)$  is called **strongly connected** if there is a path in each direction between every couple of vertices in the graph.
- The **strongly connected components** of a directed graph  $G$  are its maximal strongly connected subgraphs.

$SCC(v) = \{u \in V \mid \text{there exists a path in } G \text{ from } u \text{ to } v \text{ and a path in } G \text{ from } v \text{ to } u\}$



# Kosaraju-Sharir Algorithm

**input:** graph  $G = (V, E)$

**output:** set of strongly connected components (sets of vertices)

1.  $S =$  empty stack;
2. **while**  $S$  does not contain all vertices **do**  
    Choose an arbitrary vertex  $v$  not in  $S$ ;  
    DFS-Walk'( $v$ ) and each time that DFS finishes expanding a vertex  $u$ , push  $u$  onto  $S$ ;
3. Reverse the directions of all arcs to obtain the transpose graph;
4. **while**  $S$  is nonempty **do**  
     $v = \text{pop}(S)$ ;  
    **if**  $v$  is UNVISITED **then** DFS-Walk( $v$ );  
    The set of visited vertices will give the strongly connected component containing  $v$ ;

■ **input:** Graph  $G$ .

```
1) procedure DFS-Walk(Vertex  $u$ ) {  
2)   state[ $u$ ] = OPEN;  $d[u] = ++time$ ;  
3)   for each Vertex  $v$  in succ( $u$ )  
4)     if (state[ $v$ ] == UNVISITED) then { $p[v] = u$ ; DFS-Walk( $v$ ); }  
5)   state[ $u$ ] = CLOSED;  $f[u] = ++time$ ;  
6) }
```

```
7) procedure DFS-Walk'(Vertex  $u$ ) {  
8)   state[ $u$ ] = OPEN;  $d[u] = ++time$ ;  
9)   for each Vertex  $v$  in succ( $u$ )  
10)    if (state[ $v$ ] == UNVISITED) then { $p[v] = u$ ; DFS-Walk'( $v$ ); }  
11)   state[ $u$ ] = CLOSED;  $f[u] = ++time$ ; push  $u$  to  $S$ ;  
12) }
```

■ **output:** array  $p$  pointing to predecessor vertex, array  $d$  with times of vertex opening and array  $f$  with time of vertex closing.

# DFS-Walk - optimized

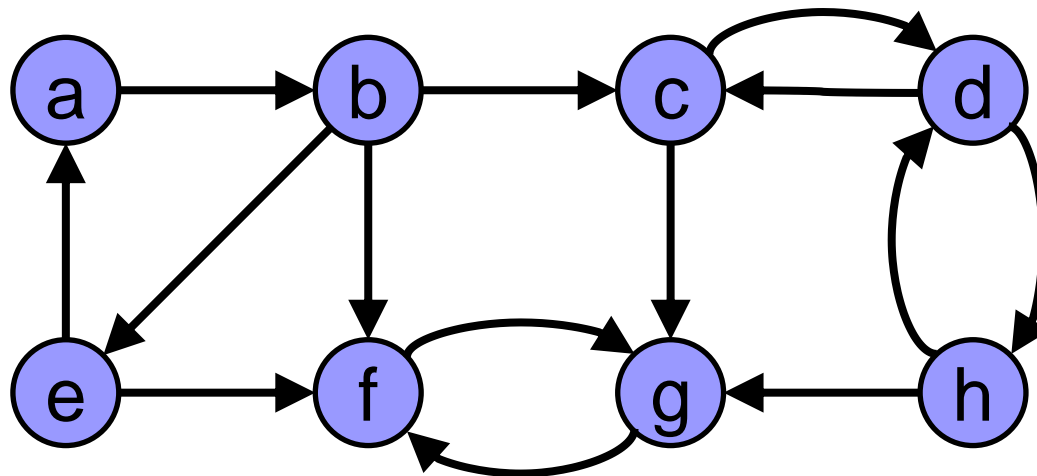
■ **input:** Graph  $G$ .

```
1) procedure DFS-Walk(Vertex  $u$ ) {
2)   state[ $u$ ] = OPEN;
3)   for each Vertex  $v$  in succ( $u$ )
4)     if (state[ $v$ ] == UNVISITED) then DFS-Walk( $v$ );
5)   state[ $u$ ] = CLOSED;
6) }
```

```
7) procedure DFS-Walk'(Vertex  $u$ ) {
8)   state[ $u$ ] = OPEN;
9)   for each Vertex  $v$  in succ( $u$ )
10)    if (state[ $v$ ] == UNVISITED) then DFS-Walk'( $v$ );
11)   state[ $u$ ] = CLOSED; push  $u$  to  $S$ ;
12) }
```

# Kosaraju-Sharir Algorithm



OPEN

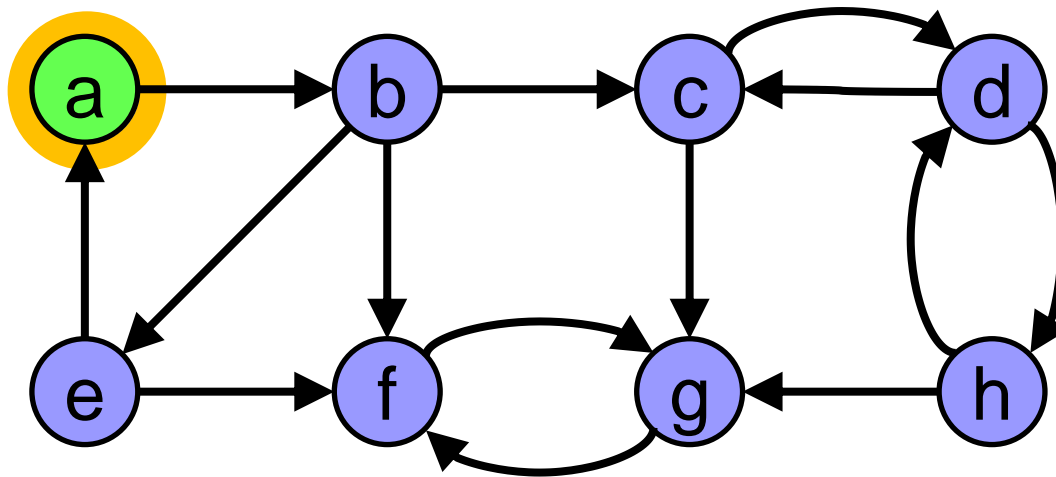


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN



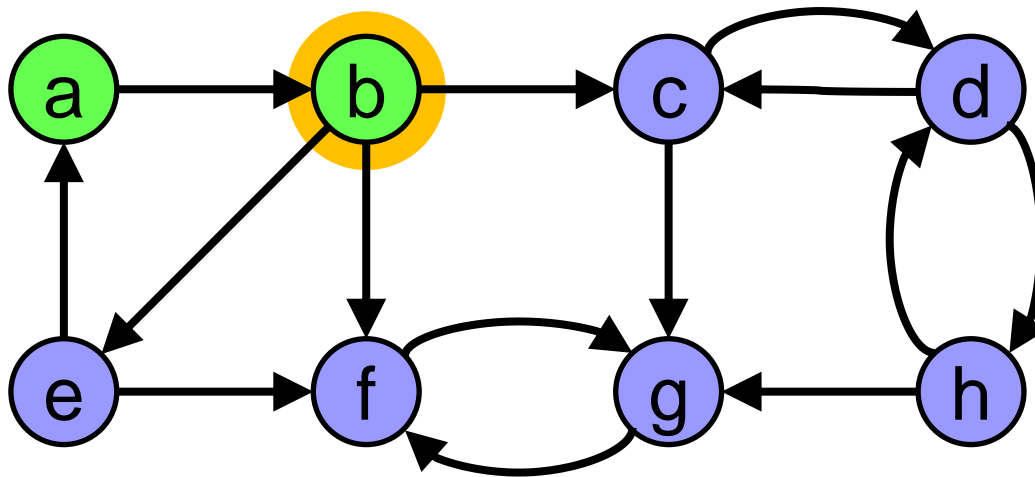
CLOSED



UNVISITED



# Kosaraju-Sharir Algorithm



OPEN

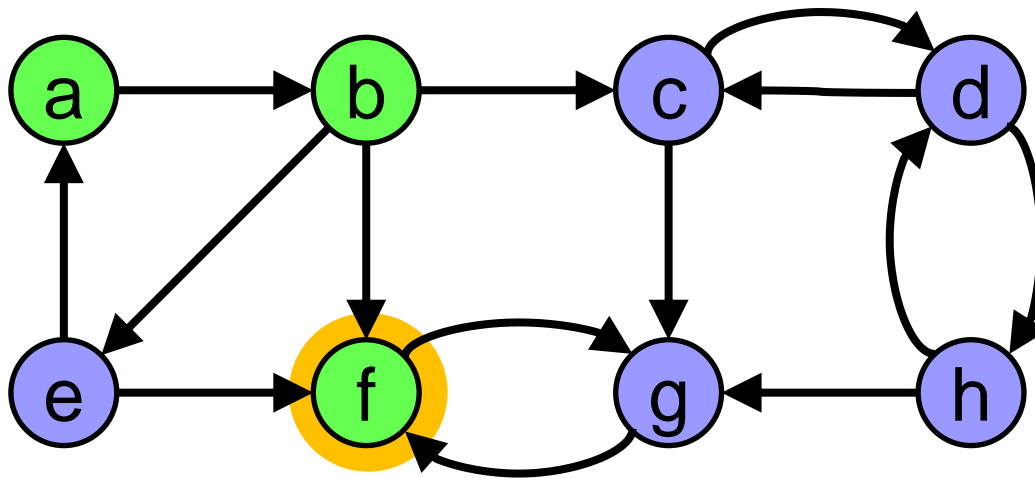


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN

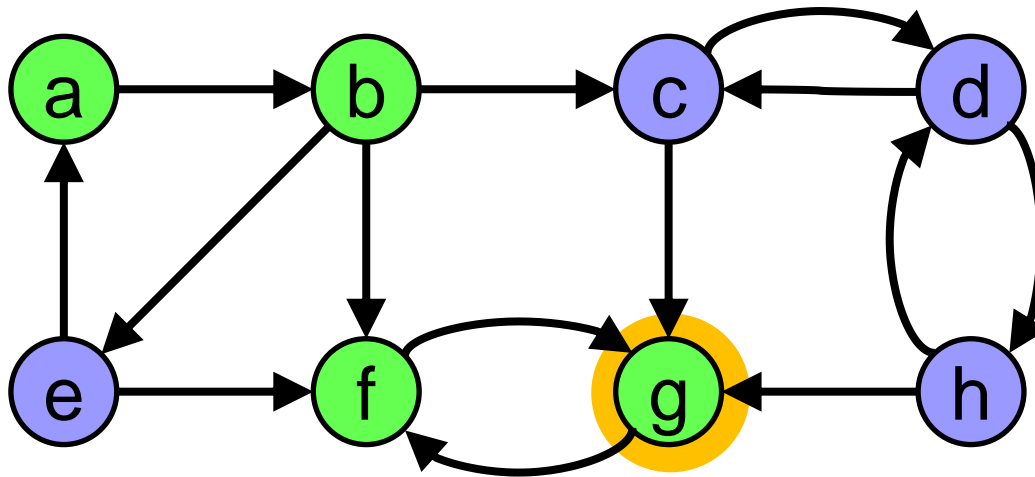


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN

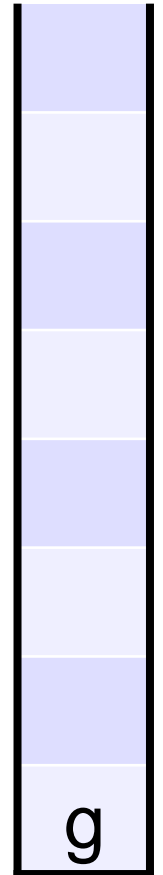
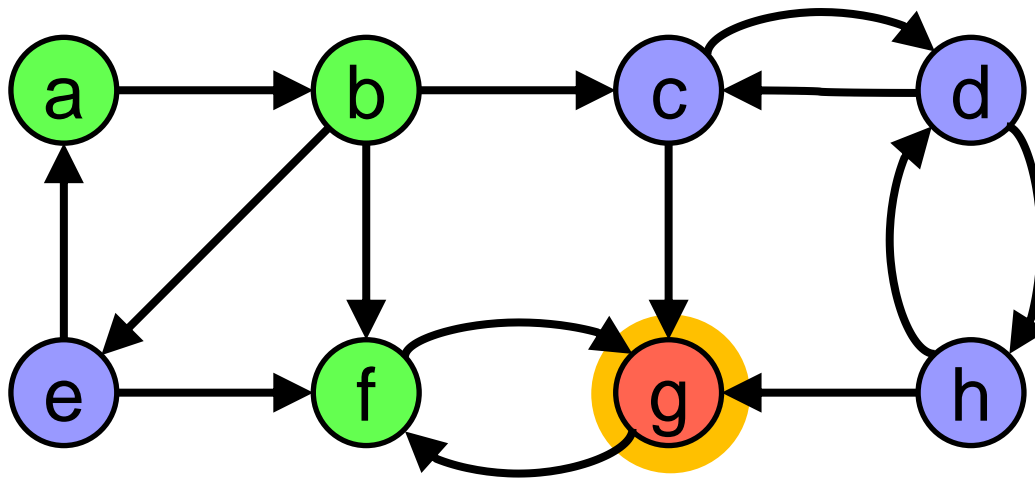


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN

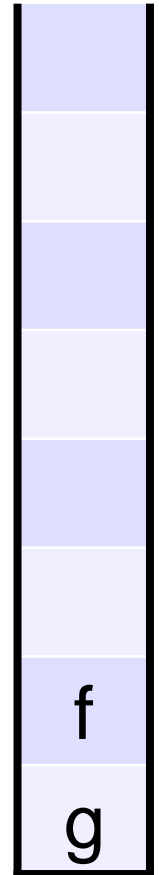
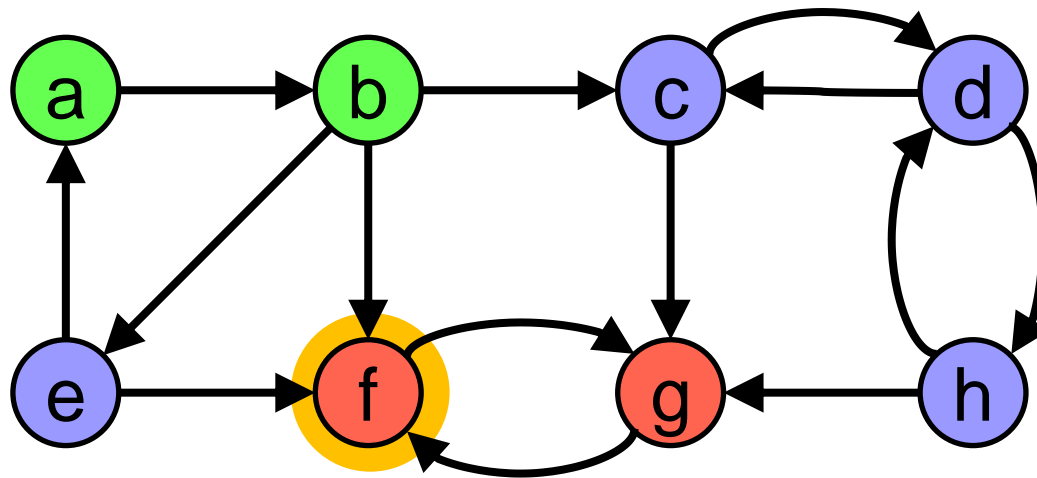


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN

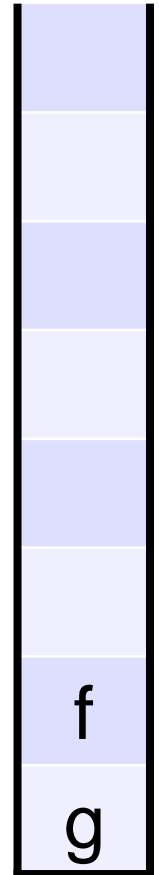
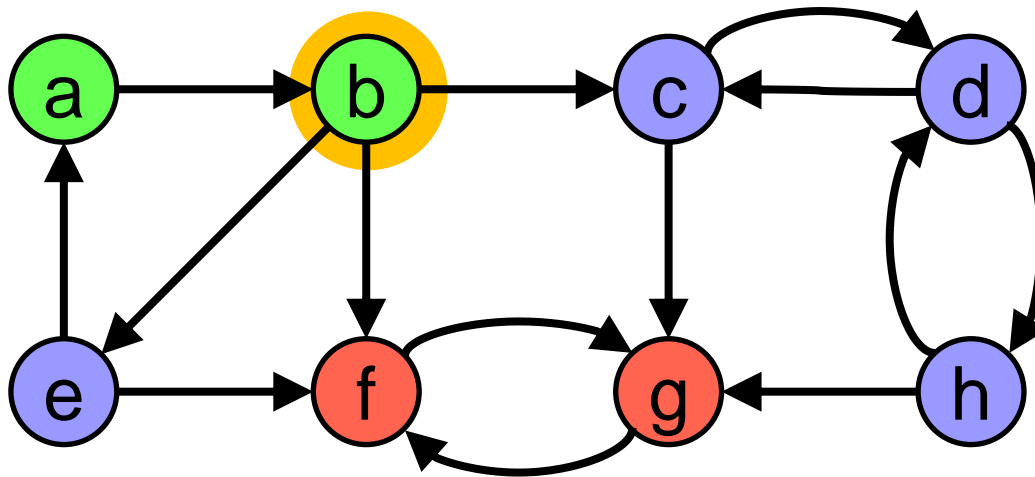


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN

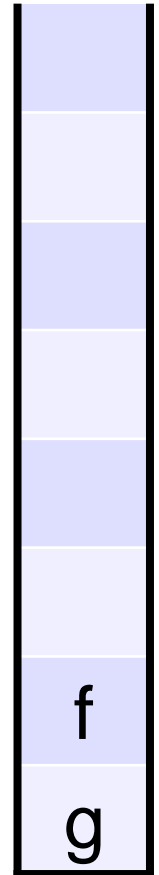
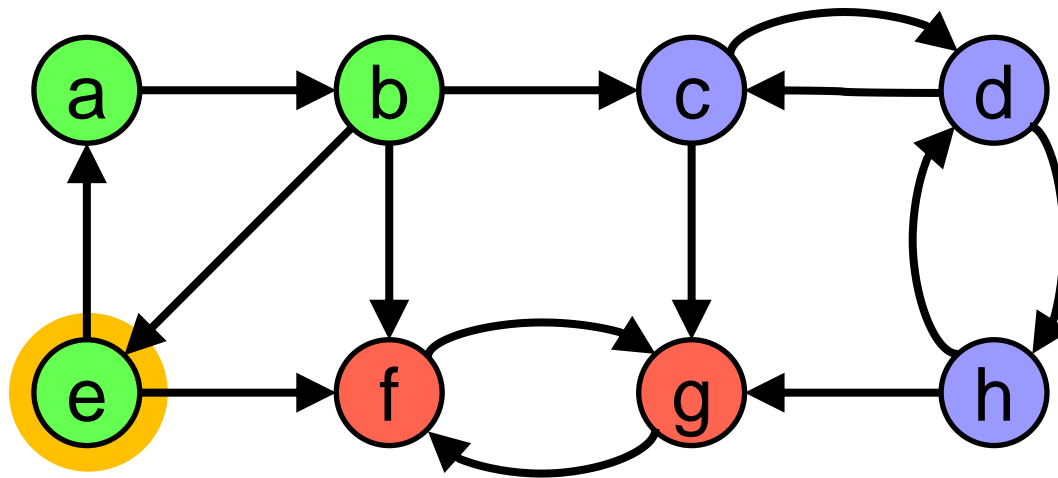


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN

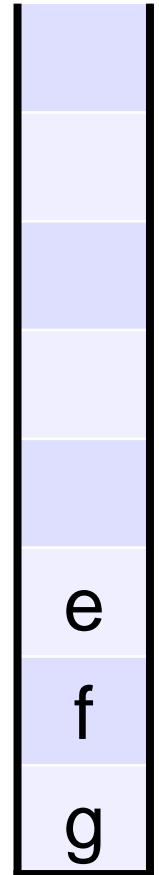
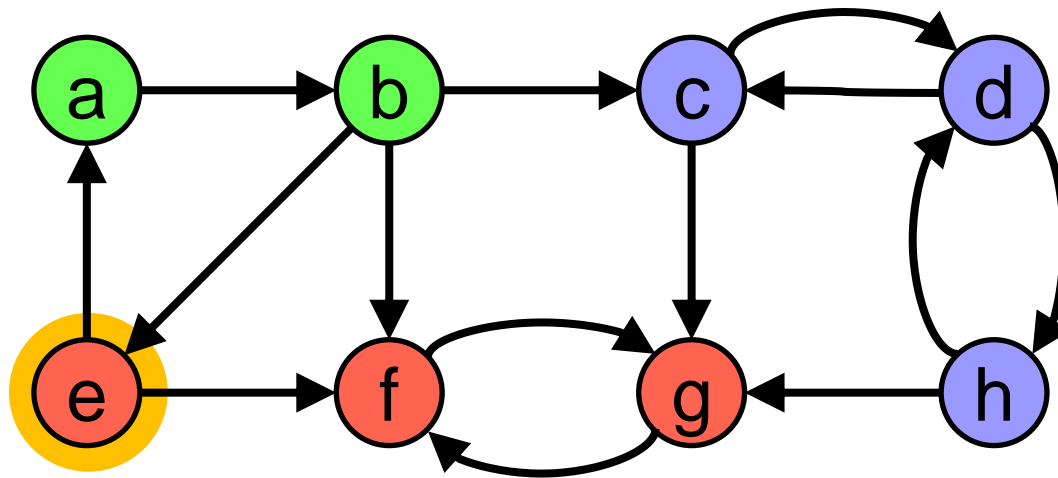


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN



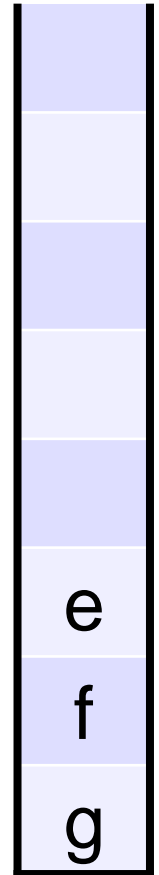
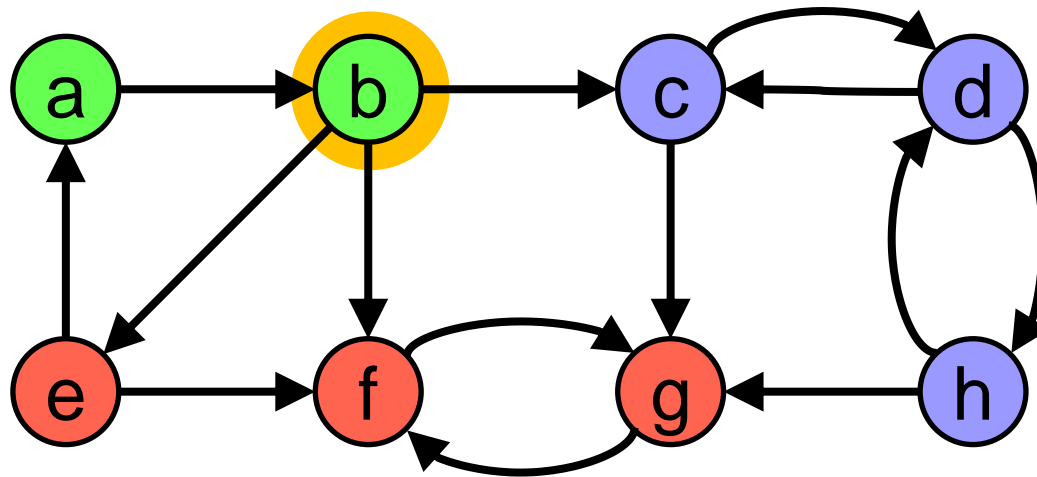
CLOSED



UNVISITED



# Kosaraju-Sharir Algorithm



OPEN

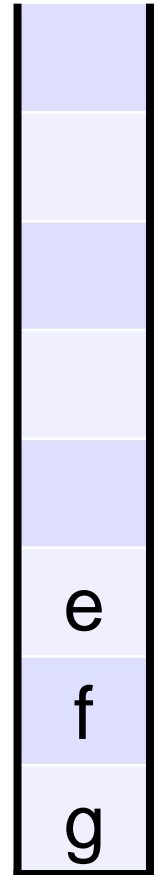
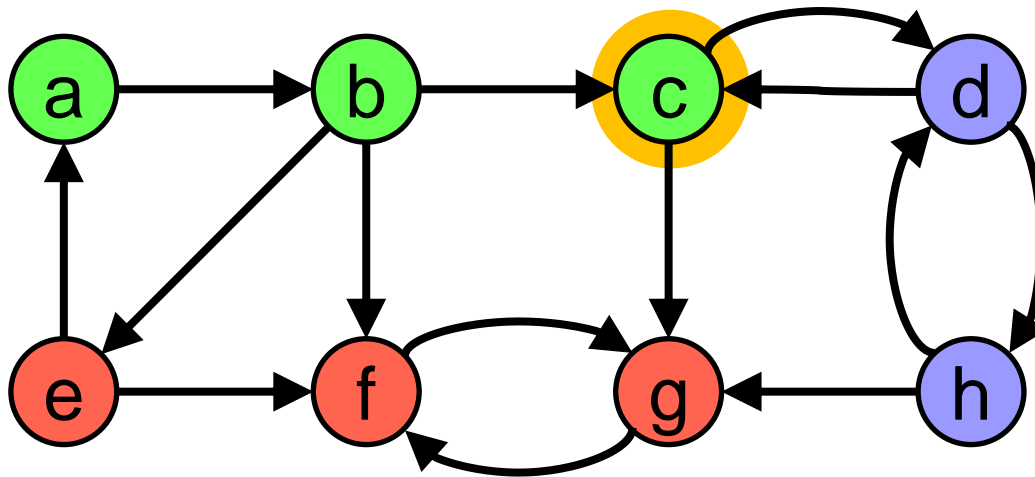


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN

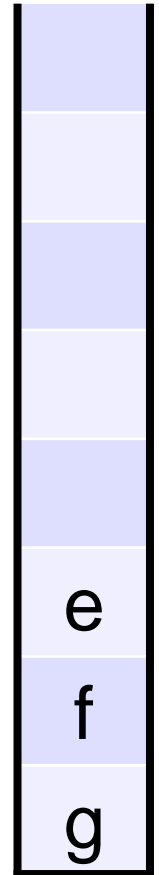
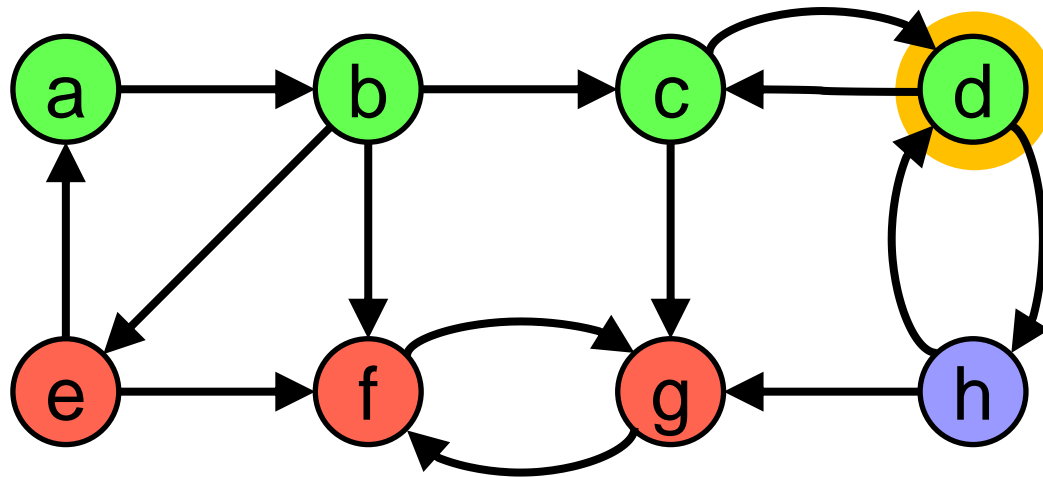


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN

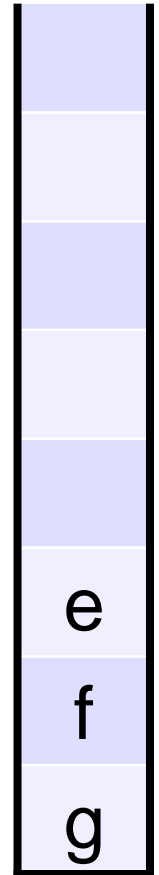
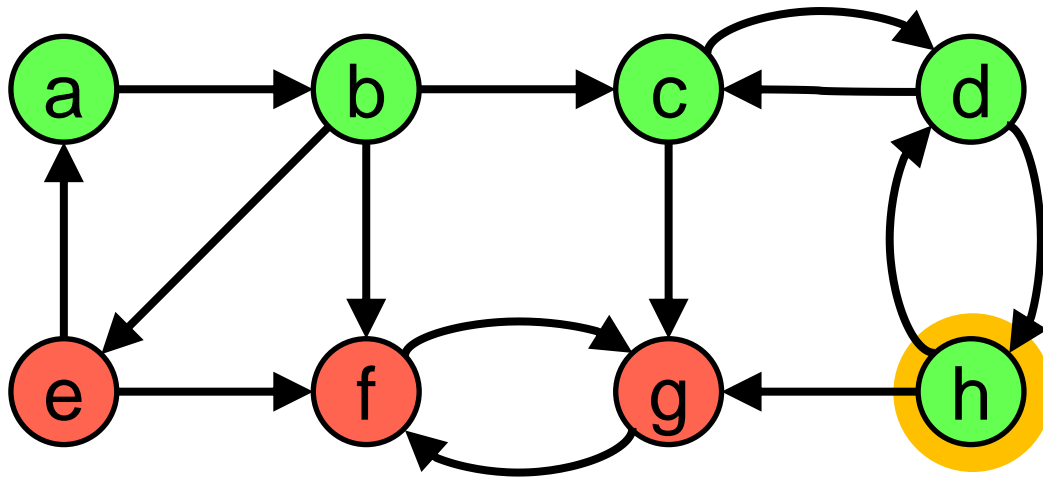


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN

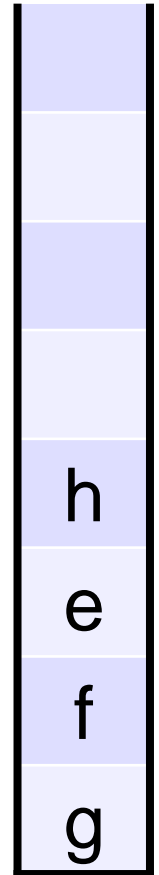
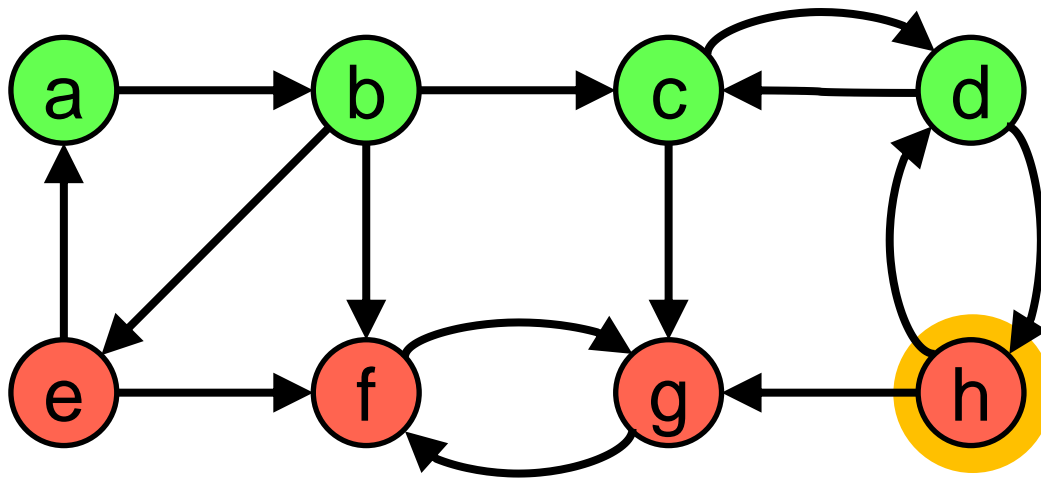


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN

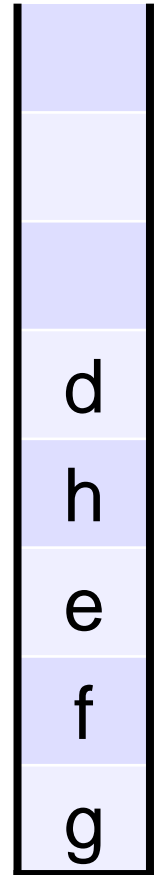
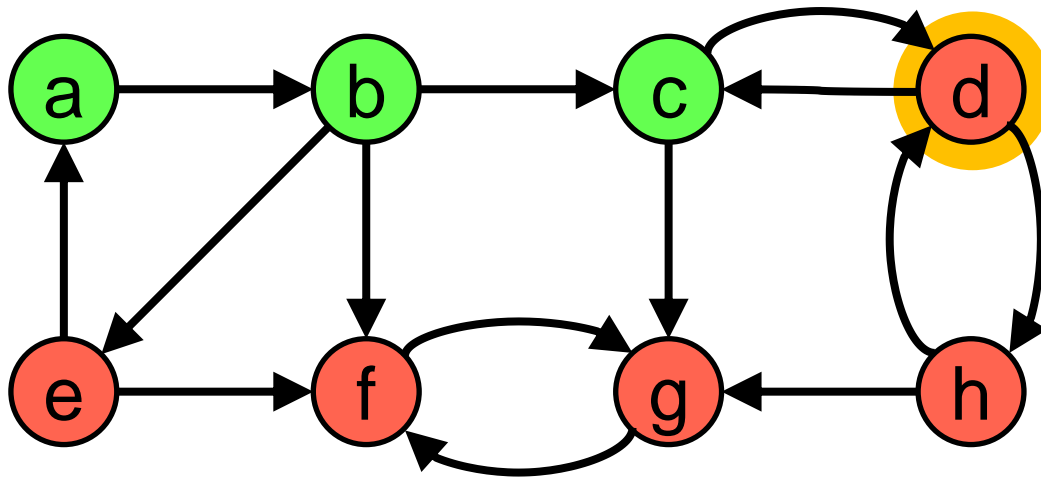


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN

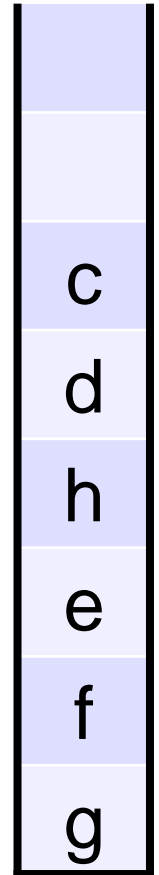
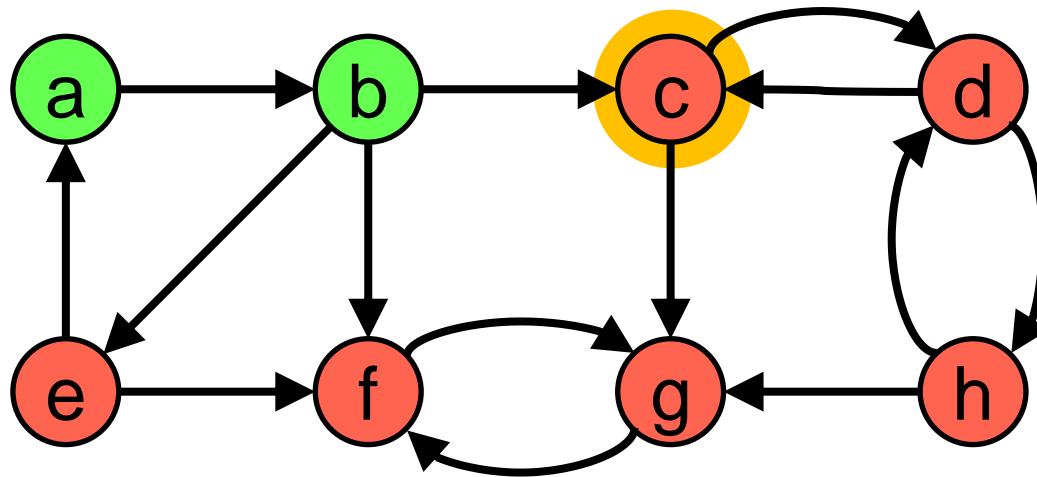


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN

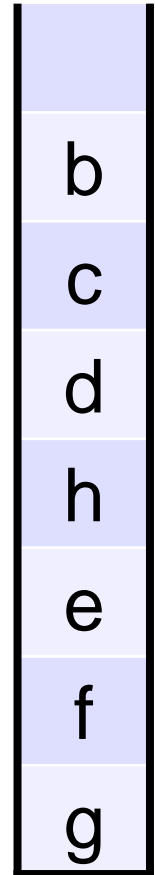
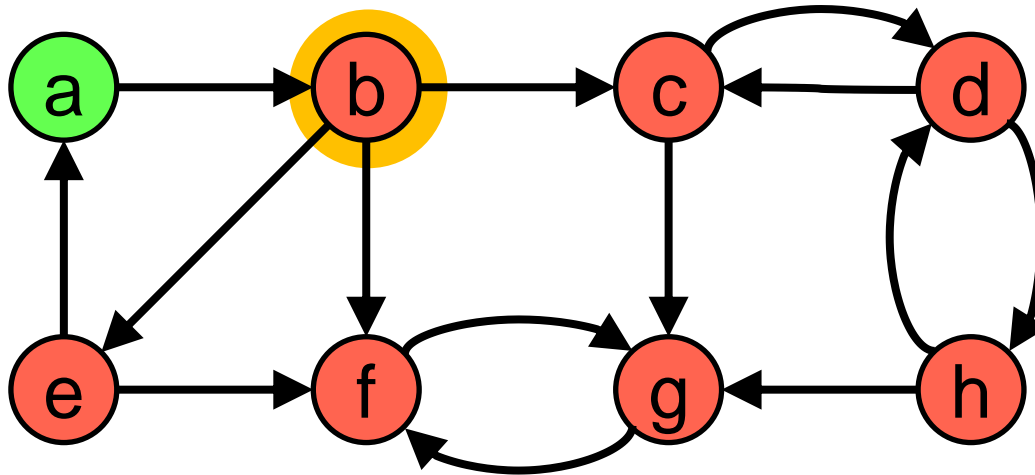


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN



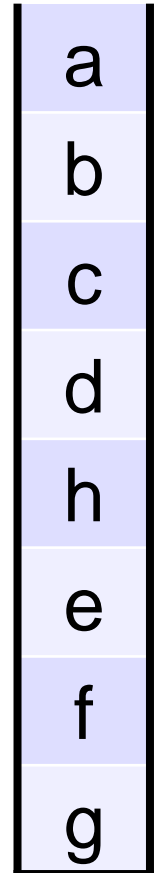
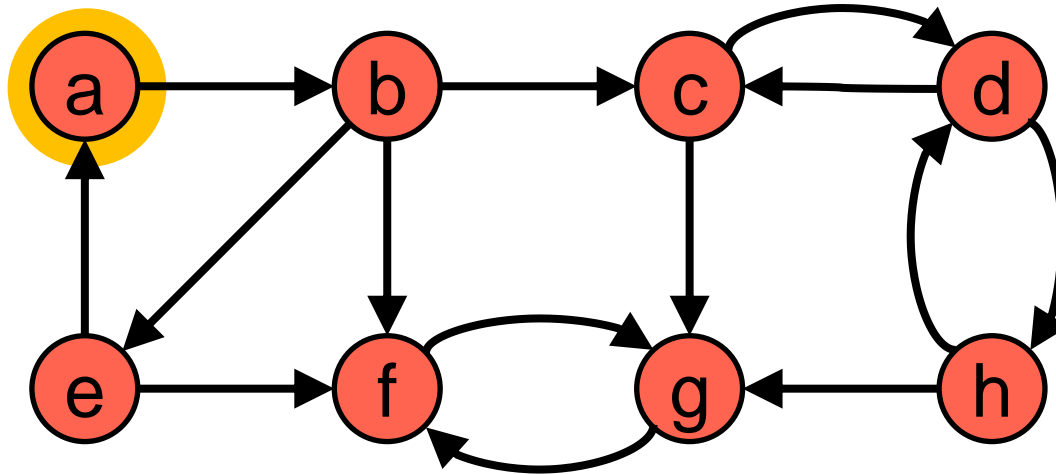
CLOSED



UNVISITED



# Kosaraju-Sharir Algorithm



OPEN

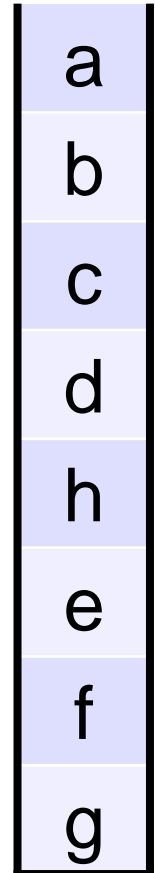
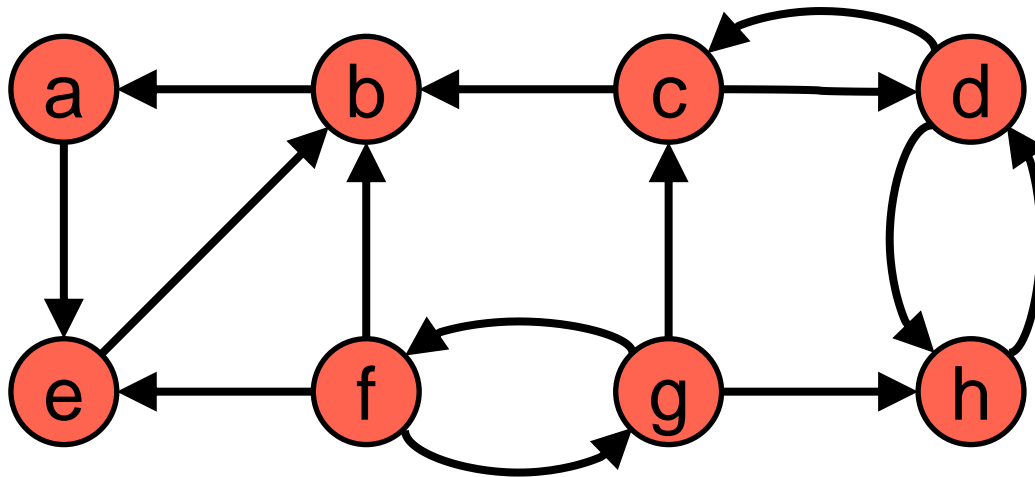


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN

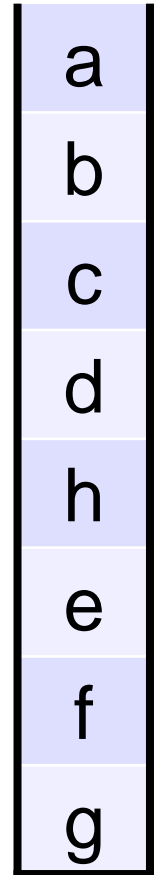
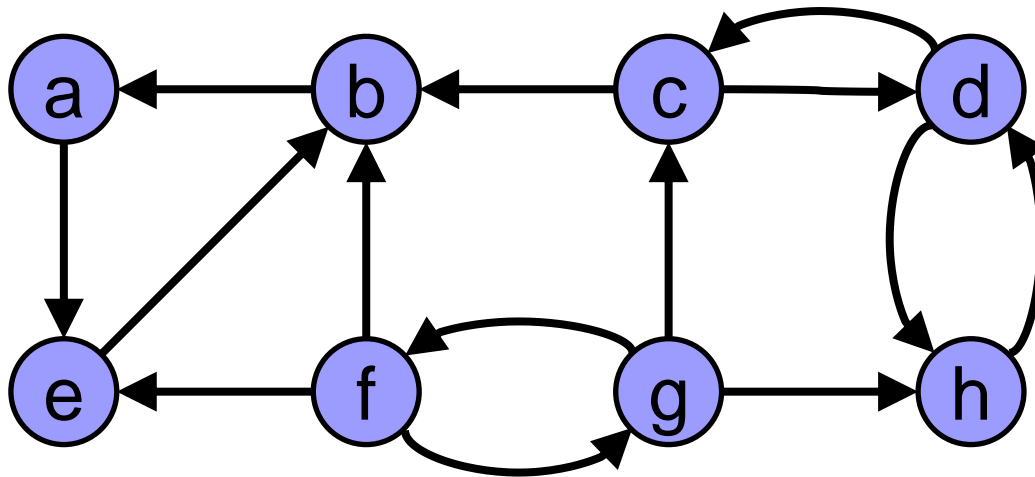


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN

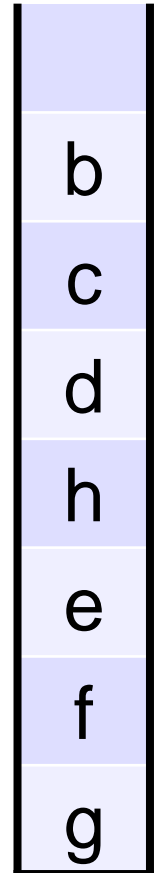
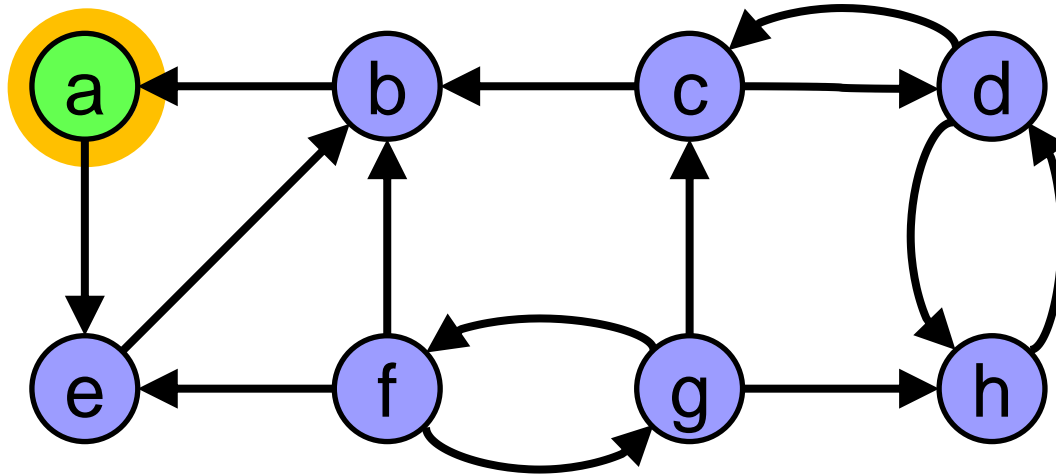


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN

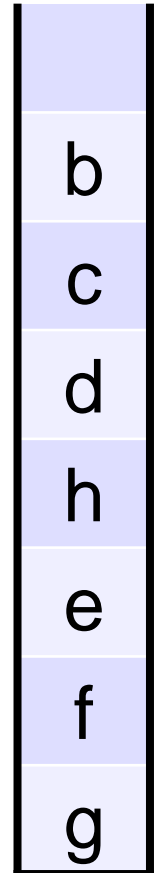
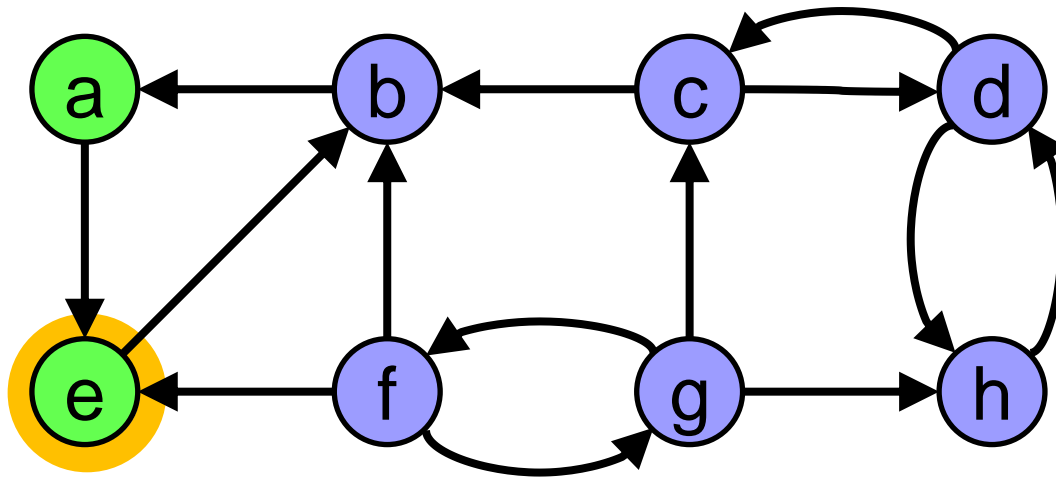


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN

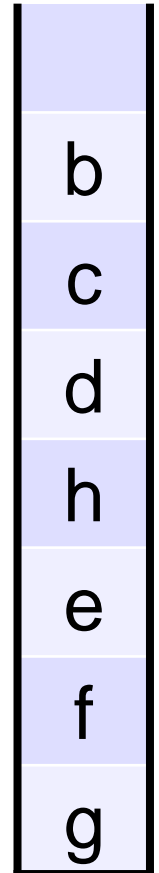
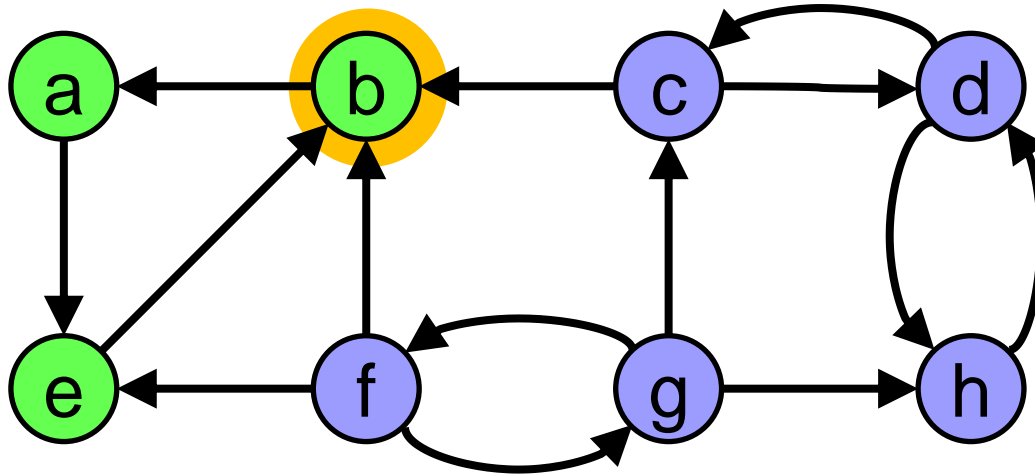


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN

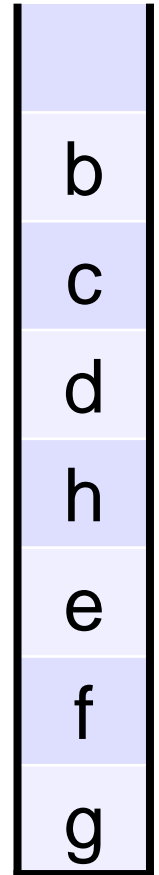
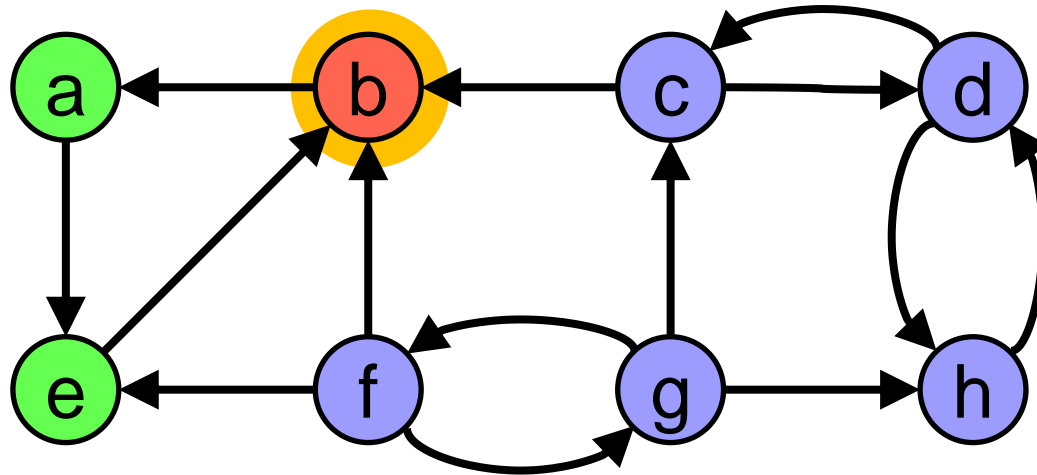


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN

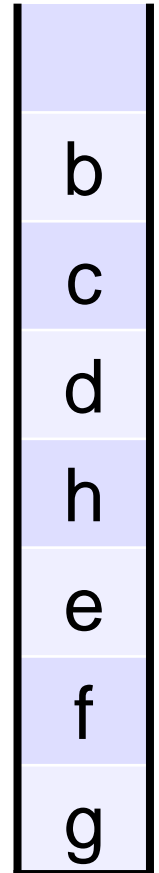
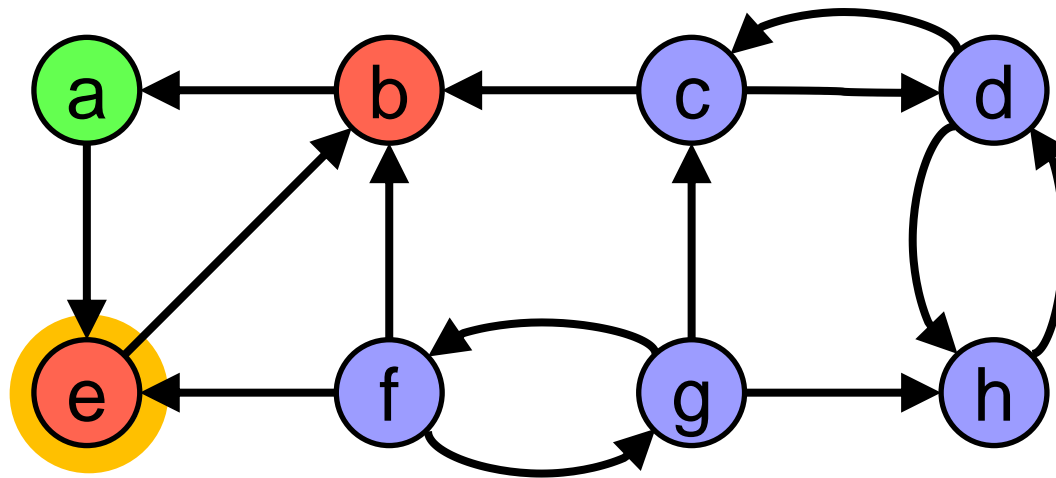


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN



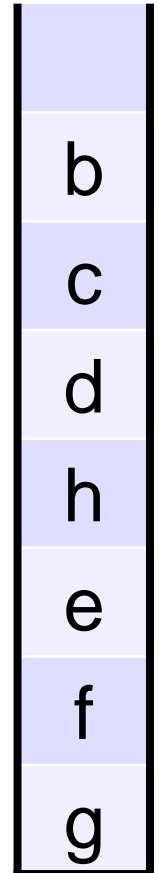
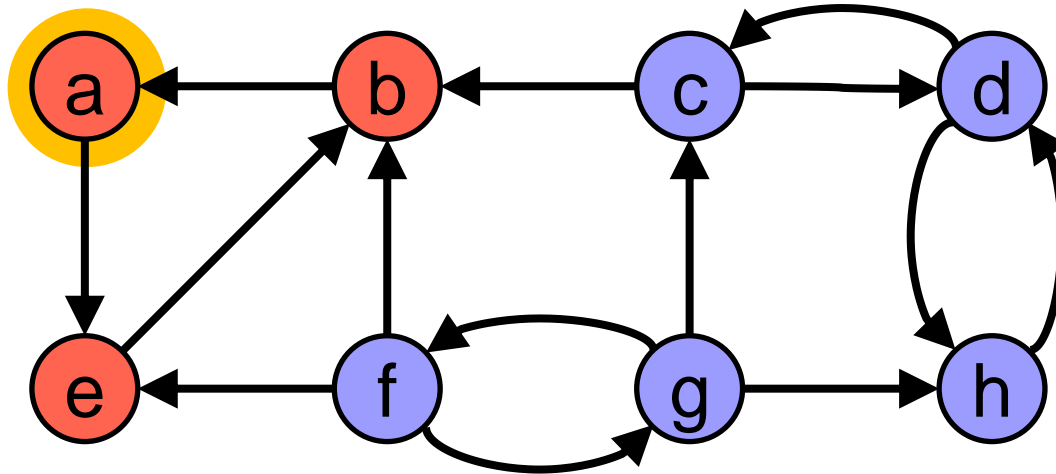
CLOSED



UNVISITED



# Kosaraju-Sharir Algorithm



OPEN

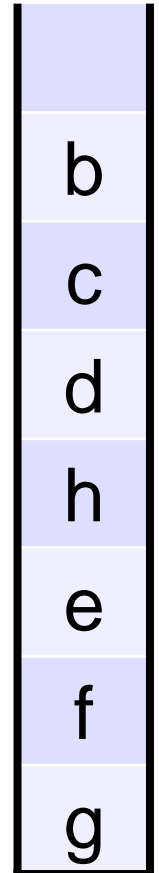
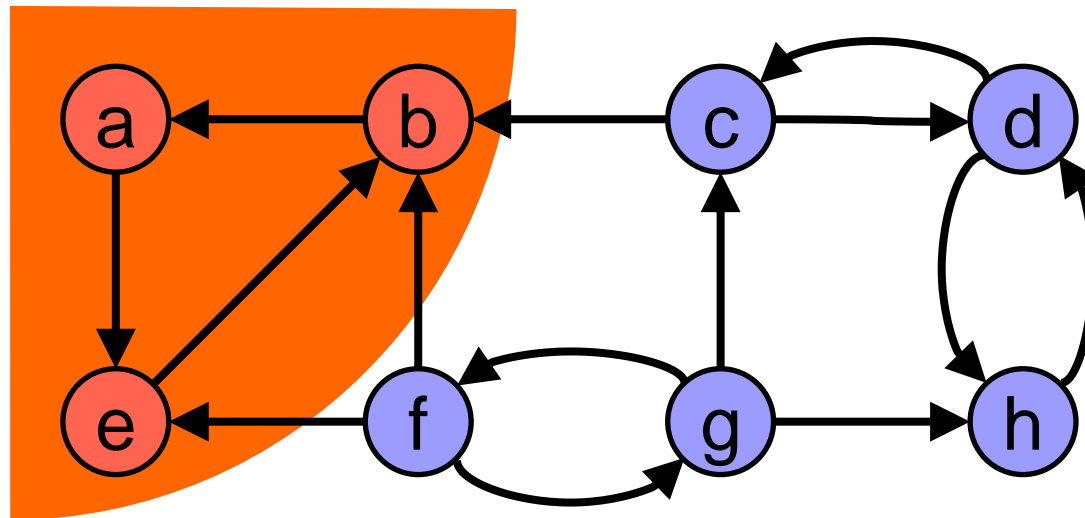


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN

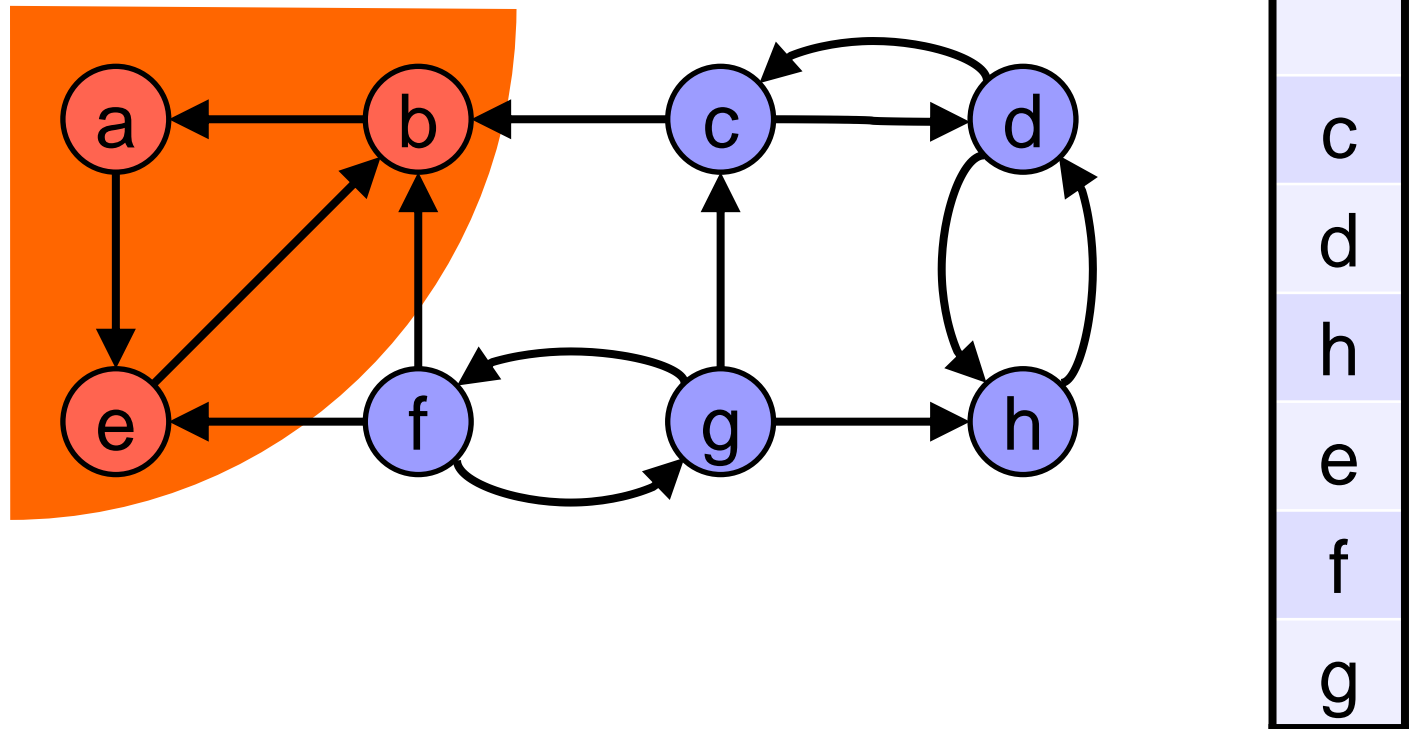


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN

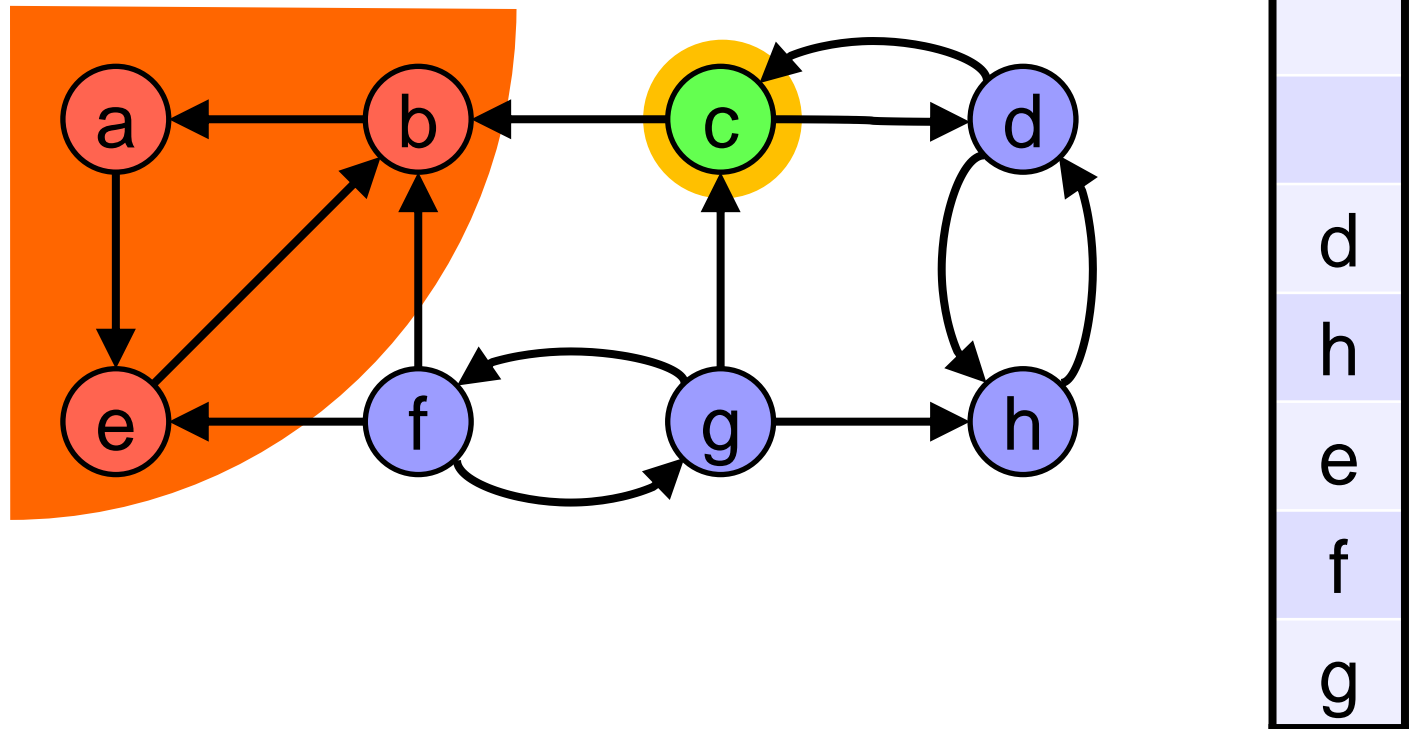


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN

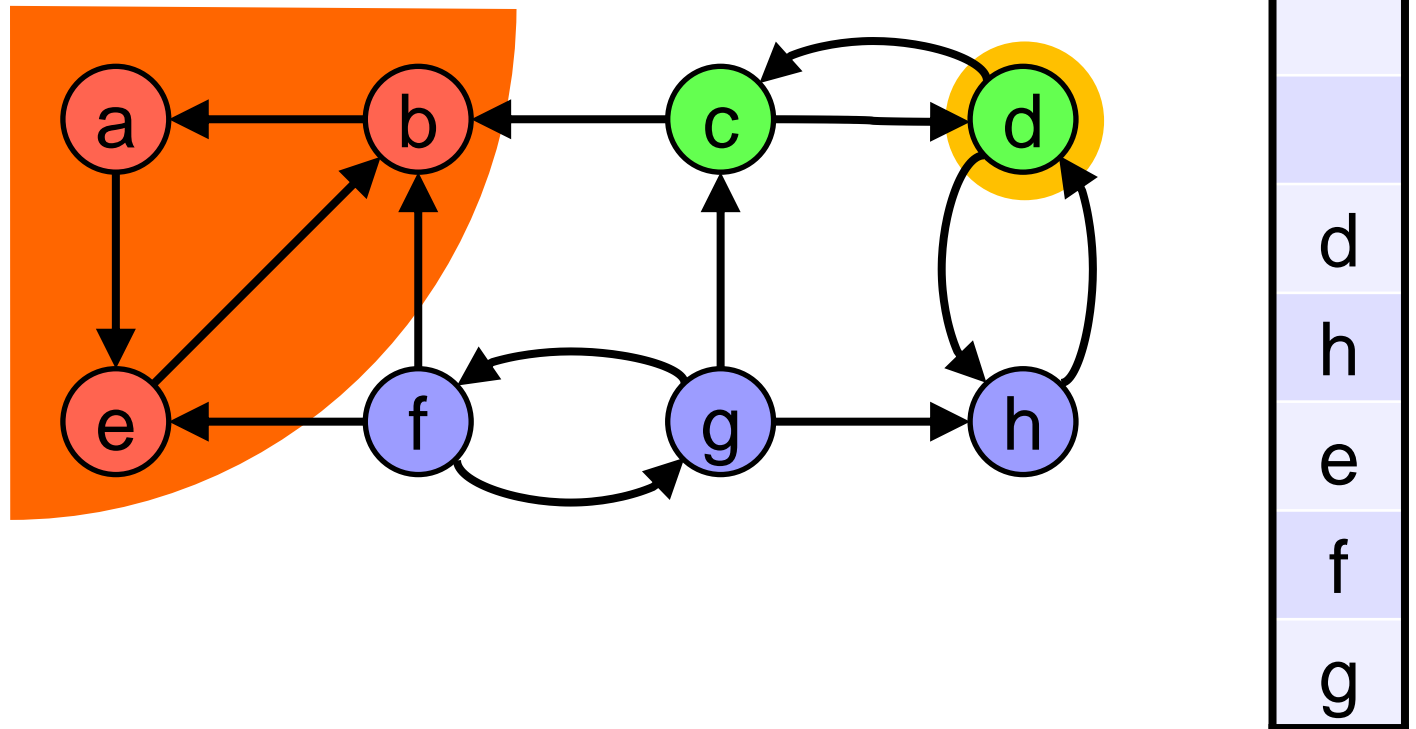


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN

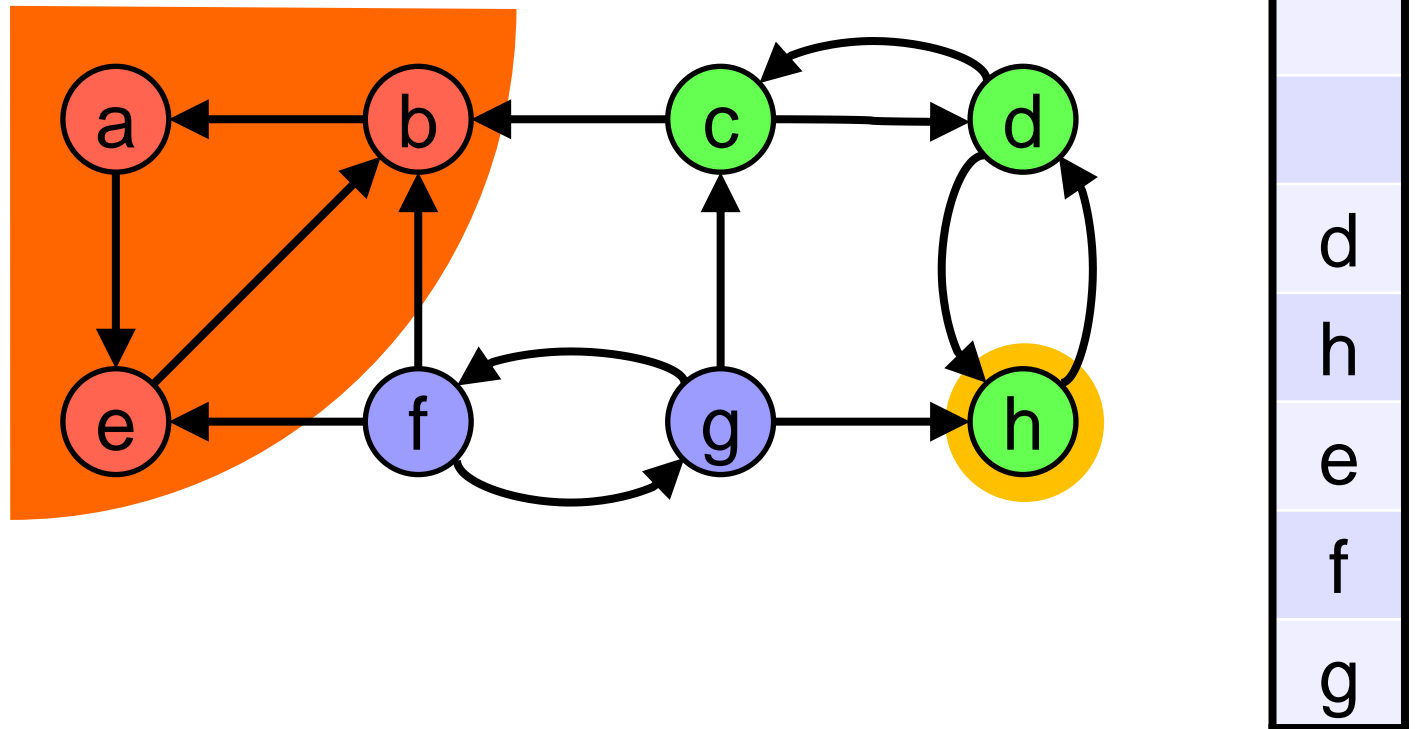


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN

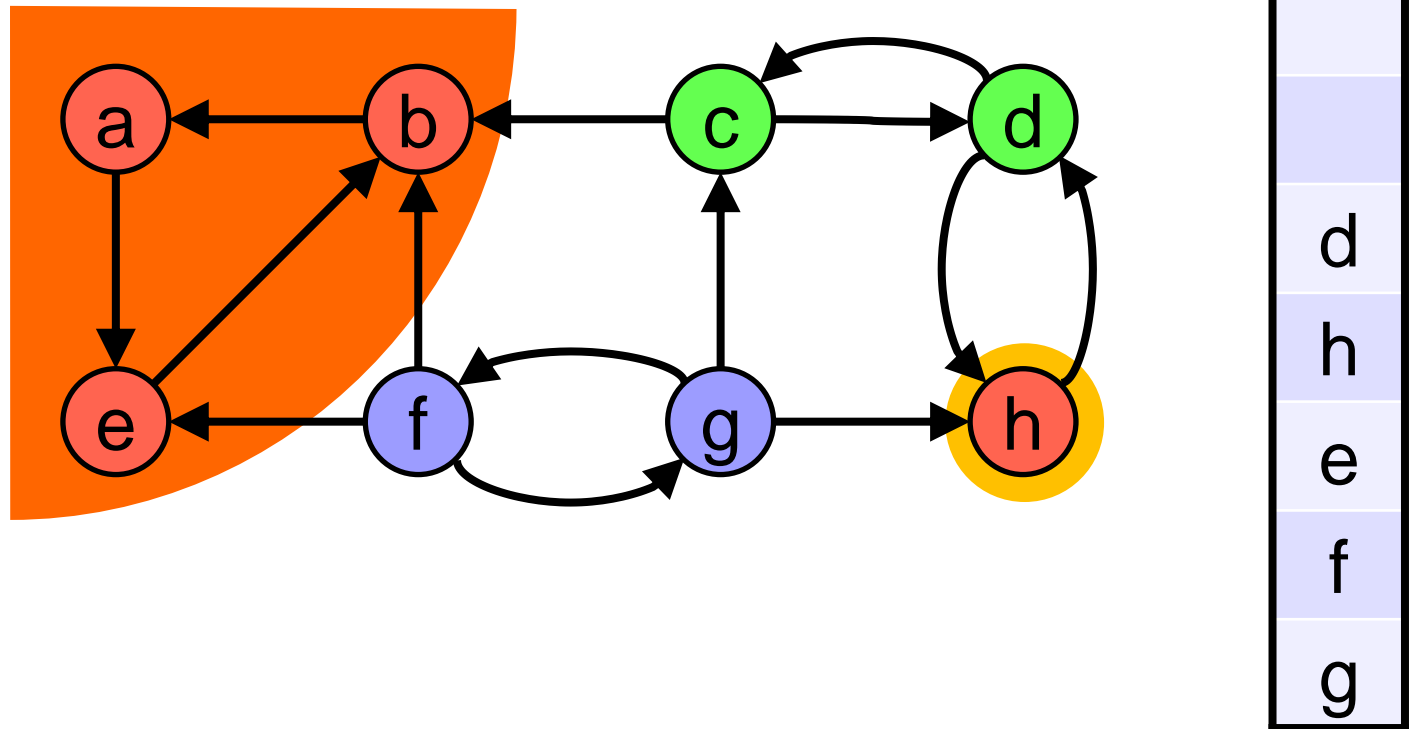


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN

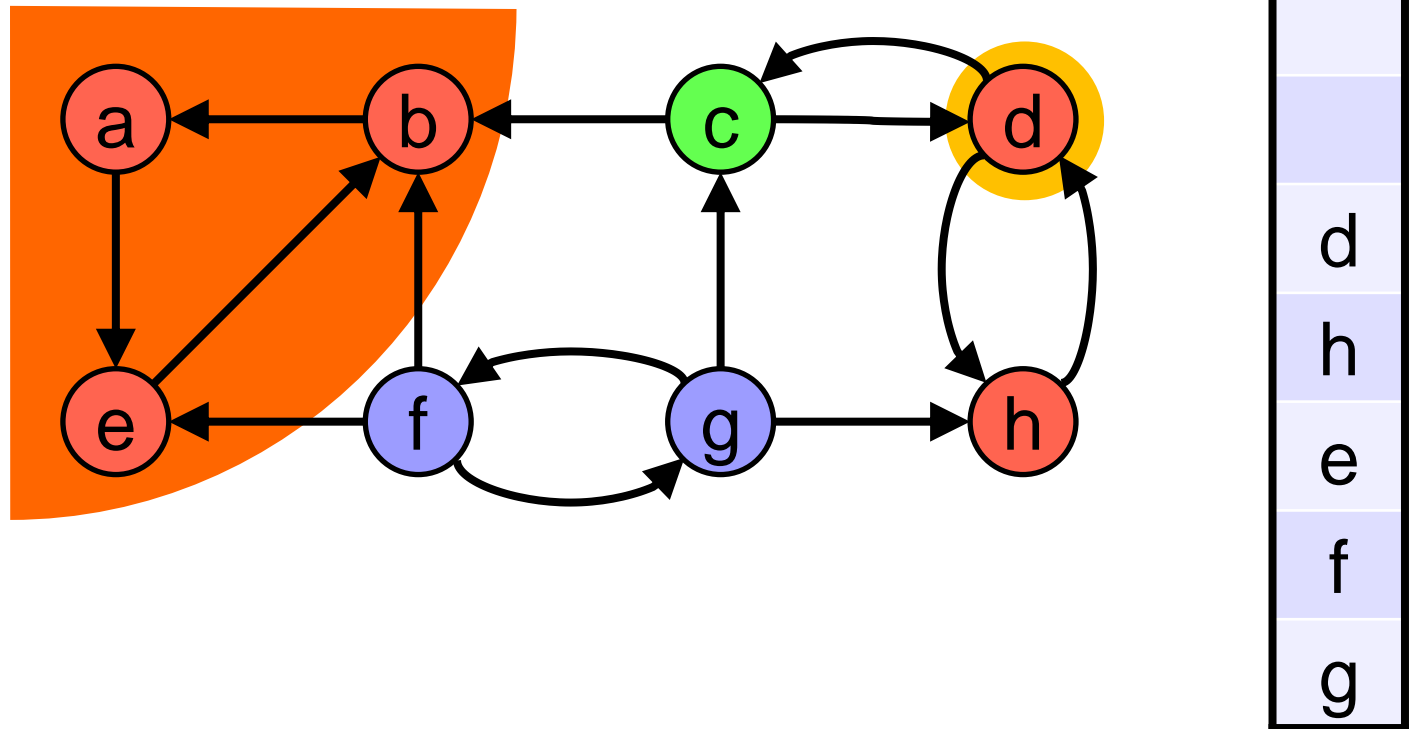


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN



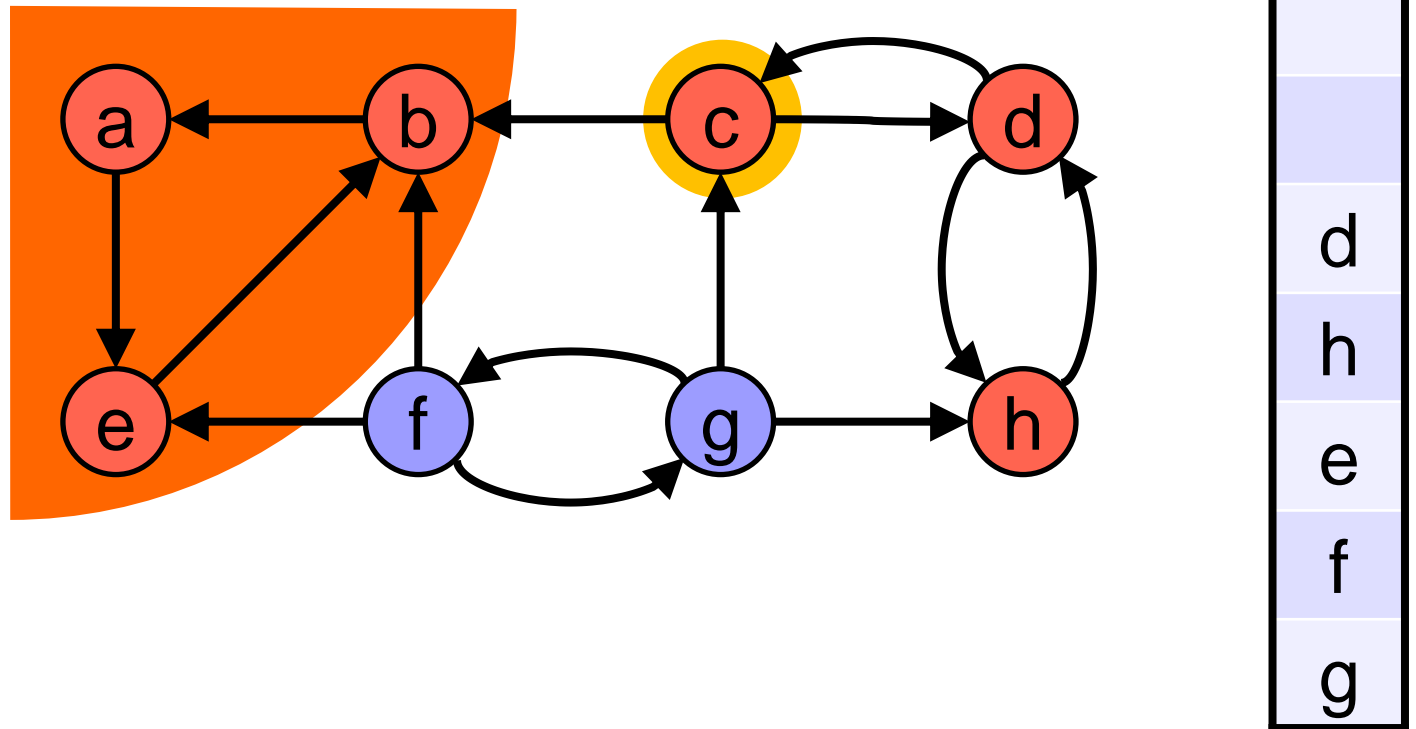
CLOSED



UNVISITED



# Kosaraju-Sharir Algorithm



OPEN

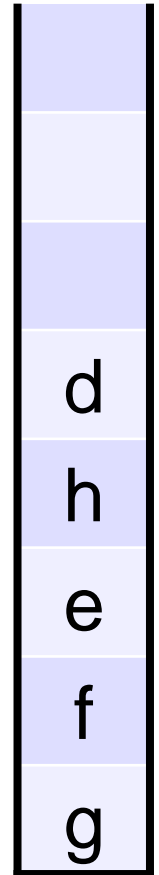
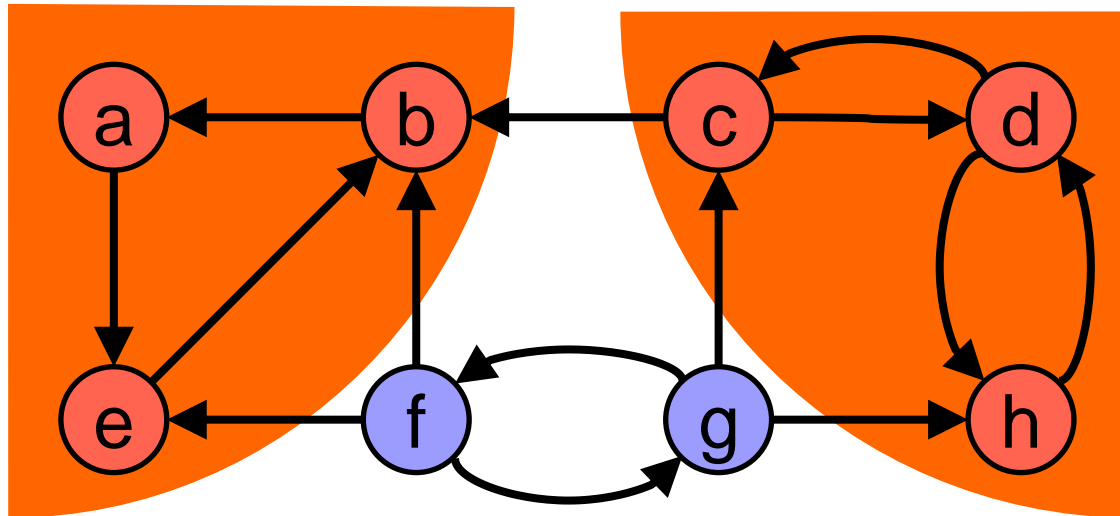


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN

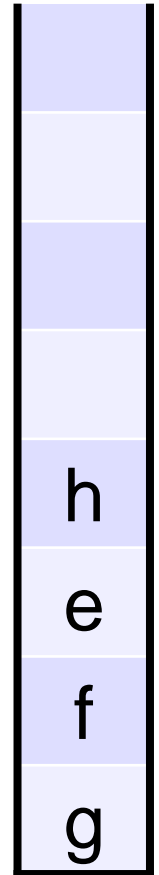
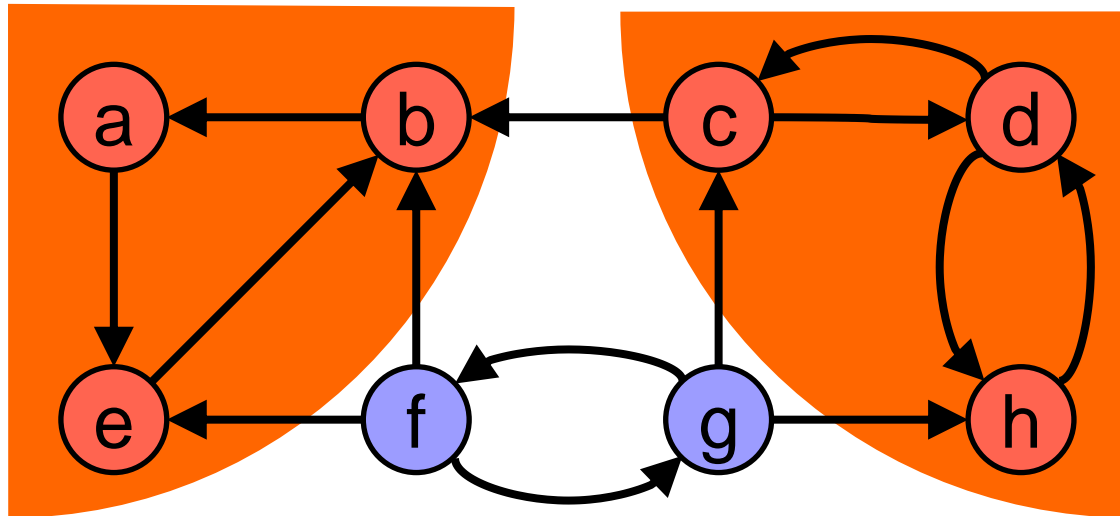


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN

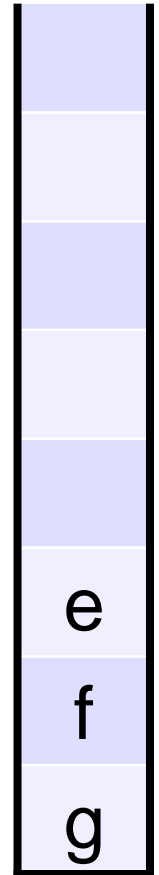
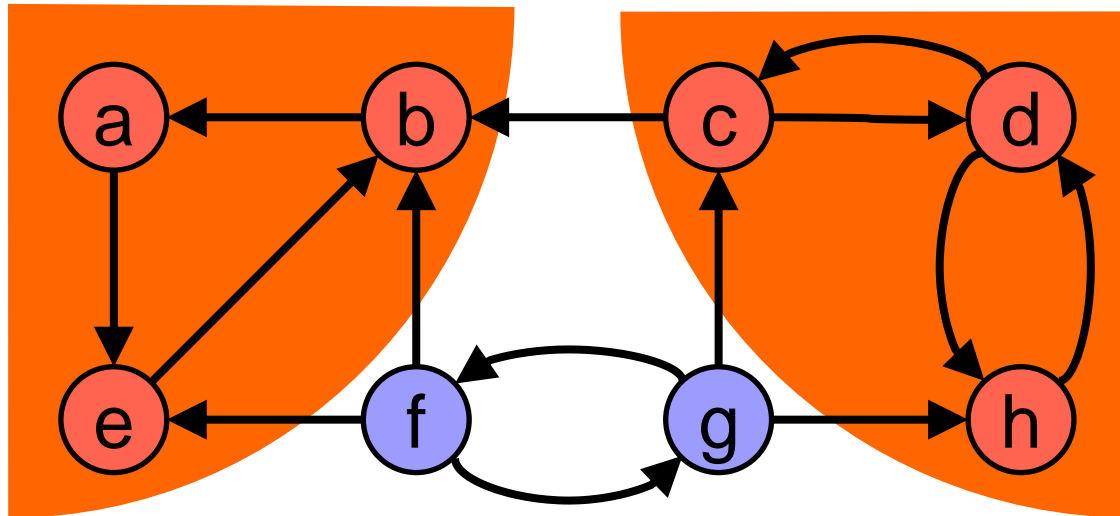


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN

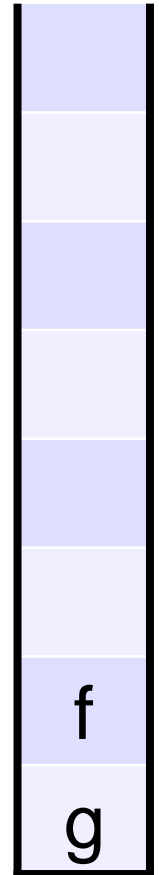
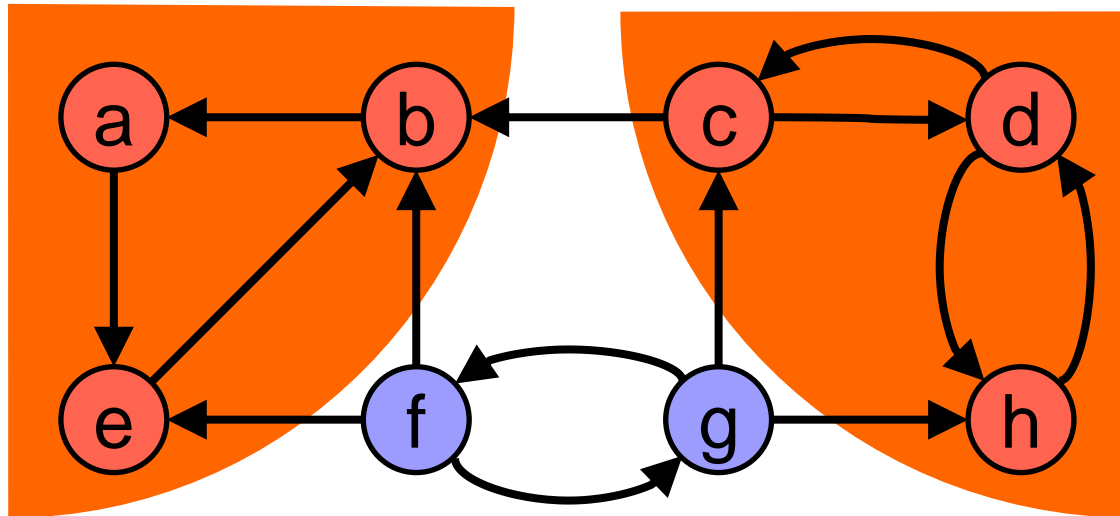


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN

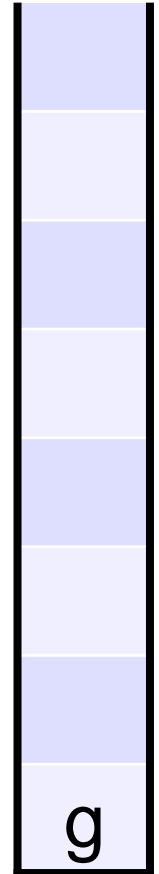
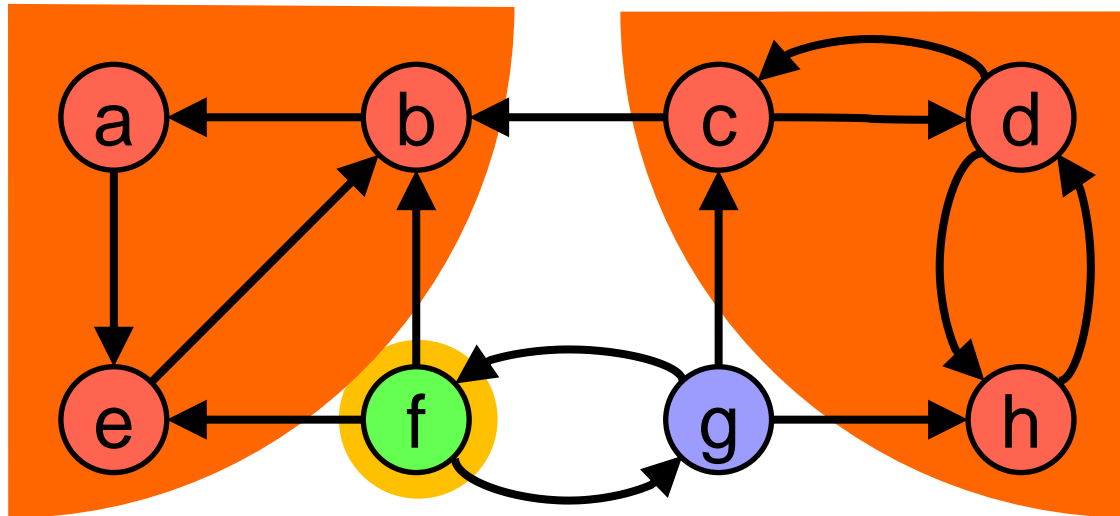


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN

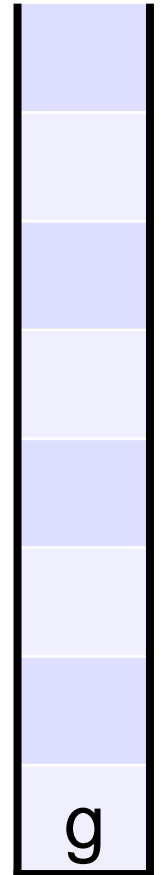
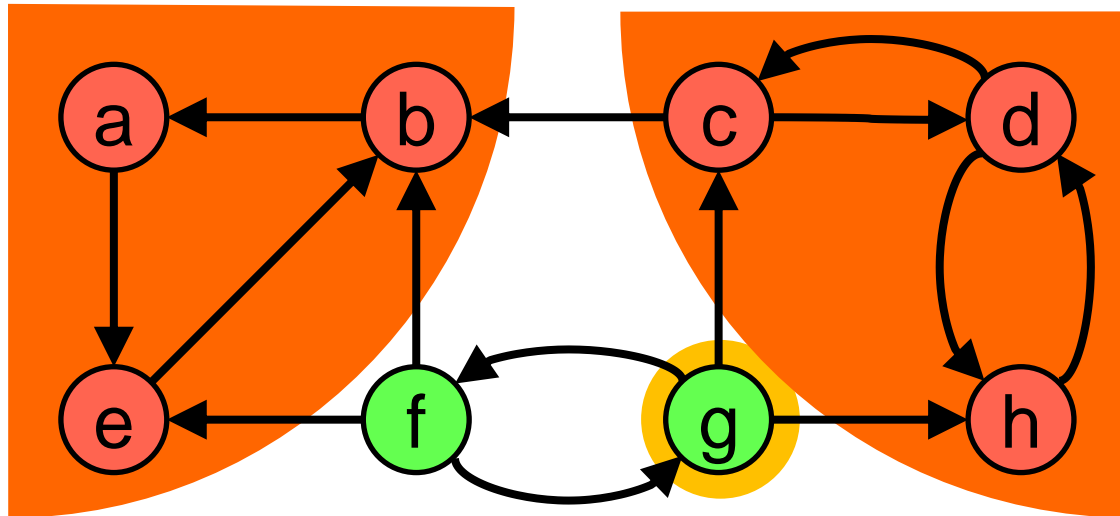


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN

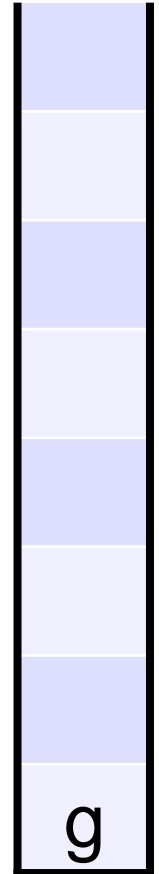
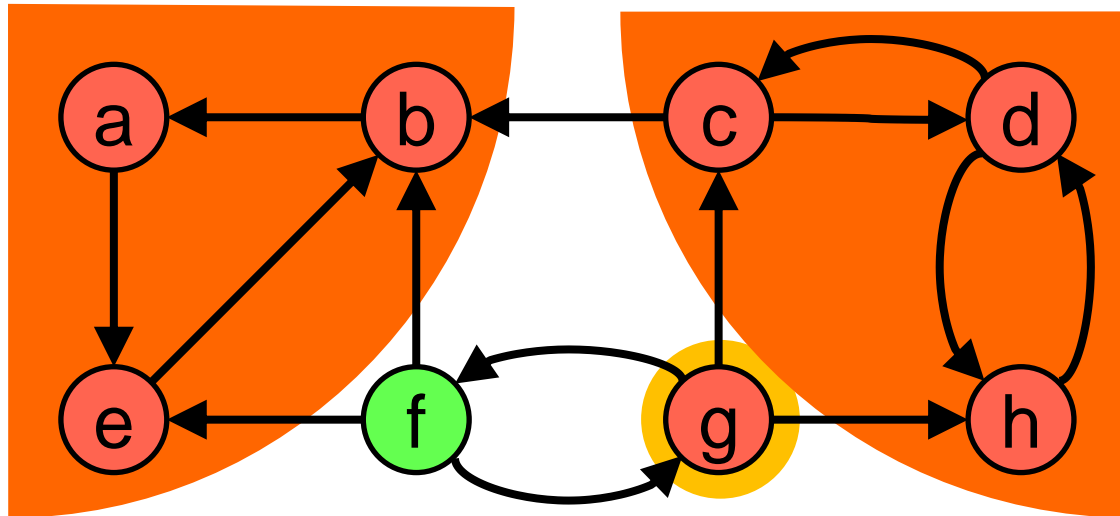


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN



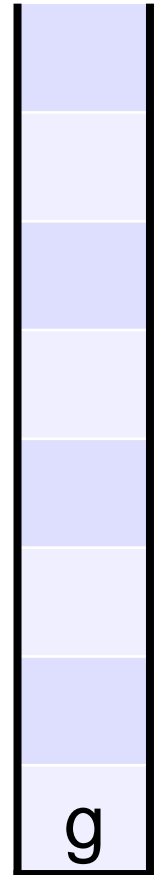
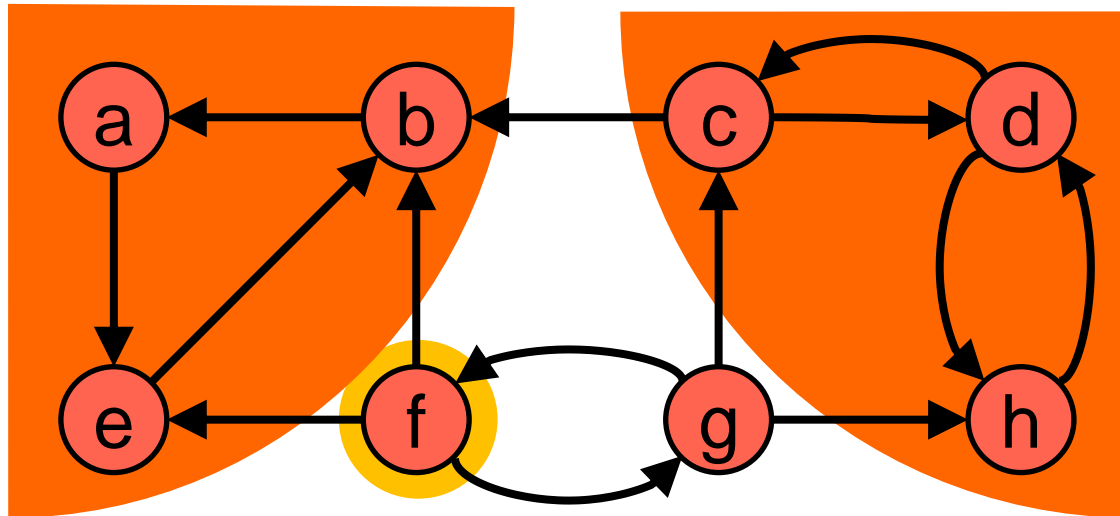
CLOSED



UNVISITED



# Kosaraju-Sharir Algorithm



OPEN

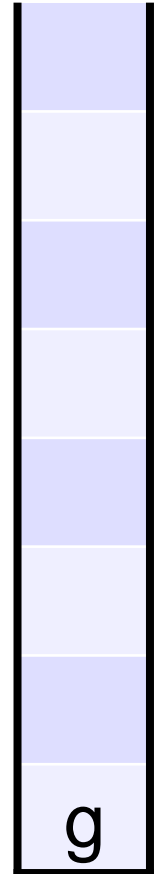
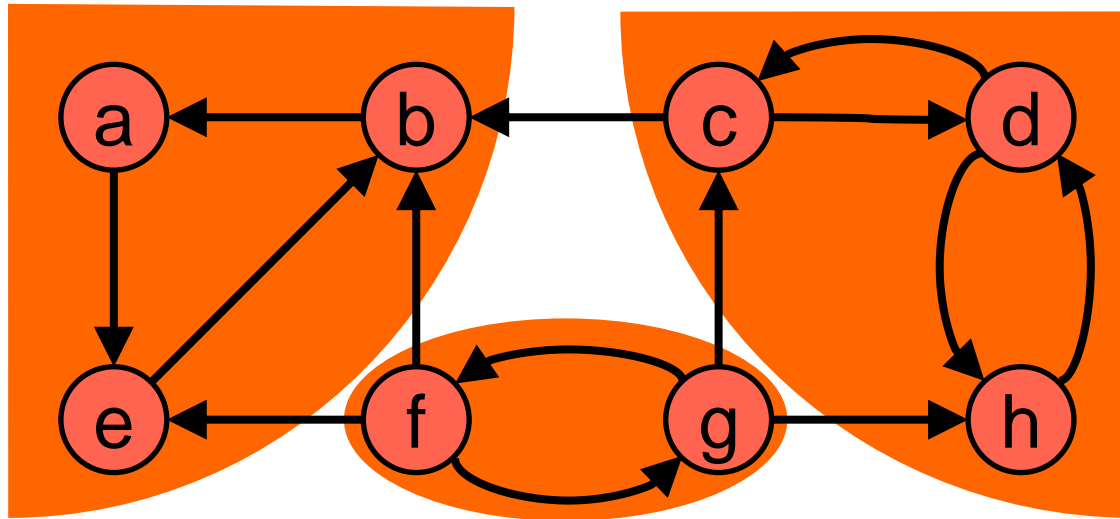


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN

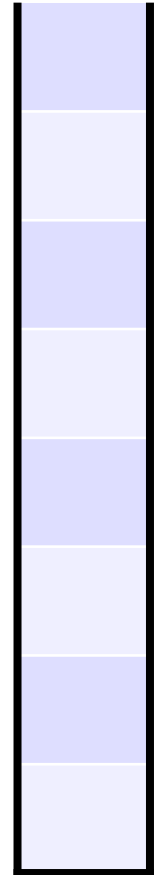
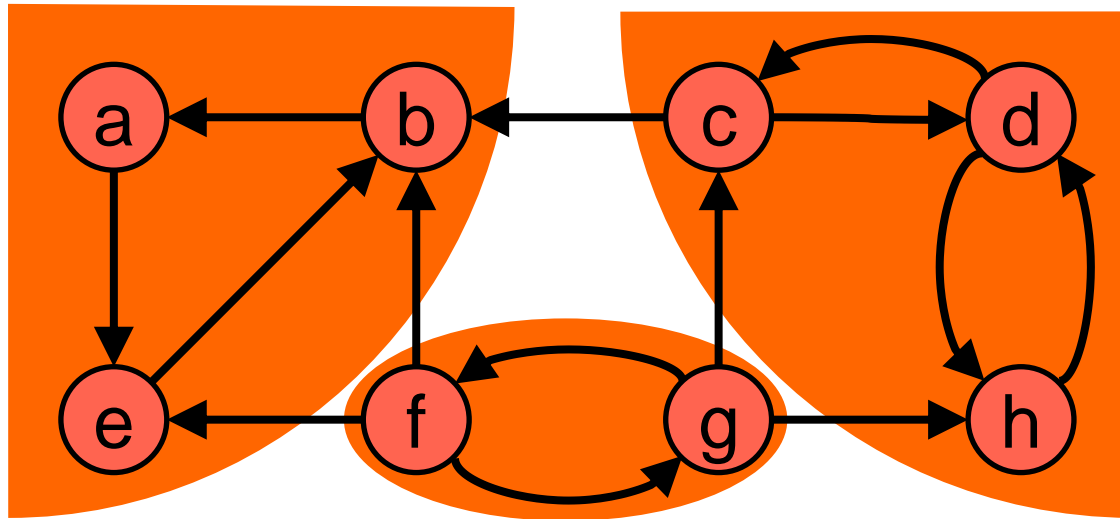


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN

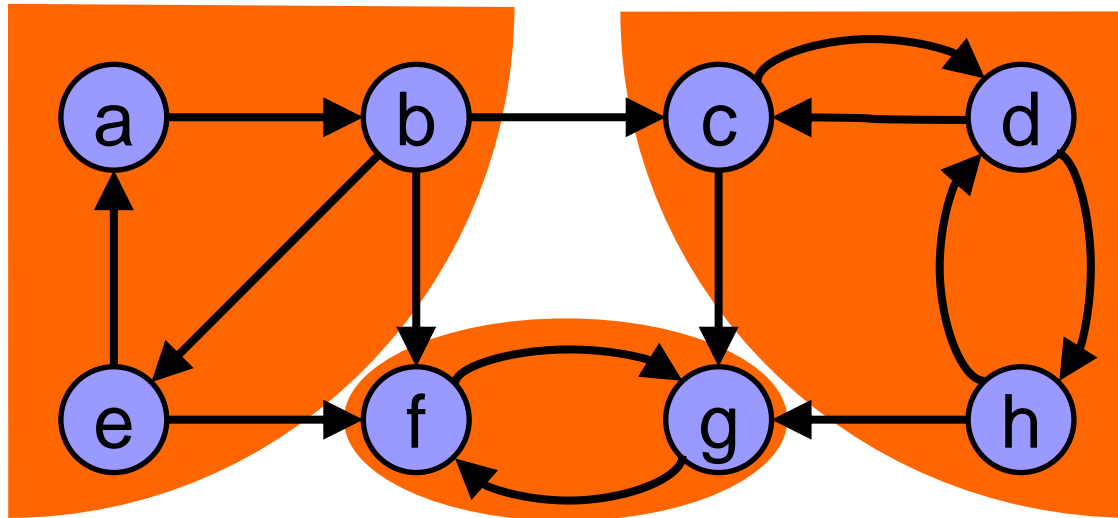


CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm



OPEN



CLOSED



UNVISITED

# Kosaraju-Sharir Algorithm

## ■ Complexity:

- The Kosaraju-Sharir algorithm performs two complete traversals of the graph.
- If the graph is represented as an *adjacency list* then the algorithm runs in  $\Theta(|V|+|E|)$  time (linear time).
- If the graph is represented as an *adjacency matrix* then the algorithm runs in  $\mathbf{O}(|V|^2)$  time.

# Tarjan's Algorithm

**input:** graph  $G = (V, E)$   
**output:** set of strongly connected components

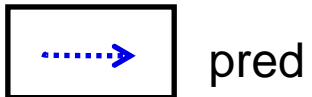
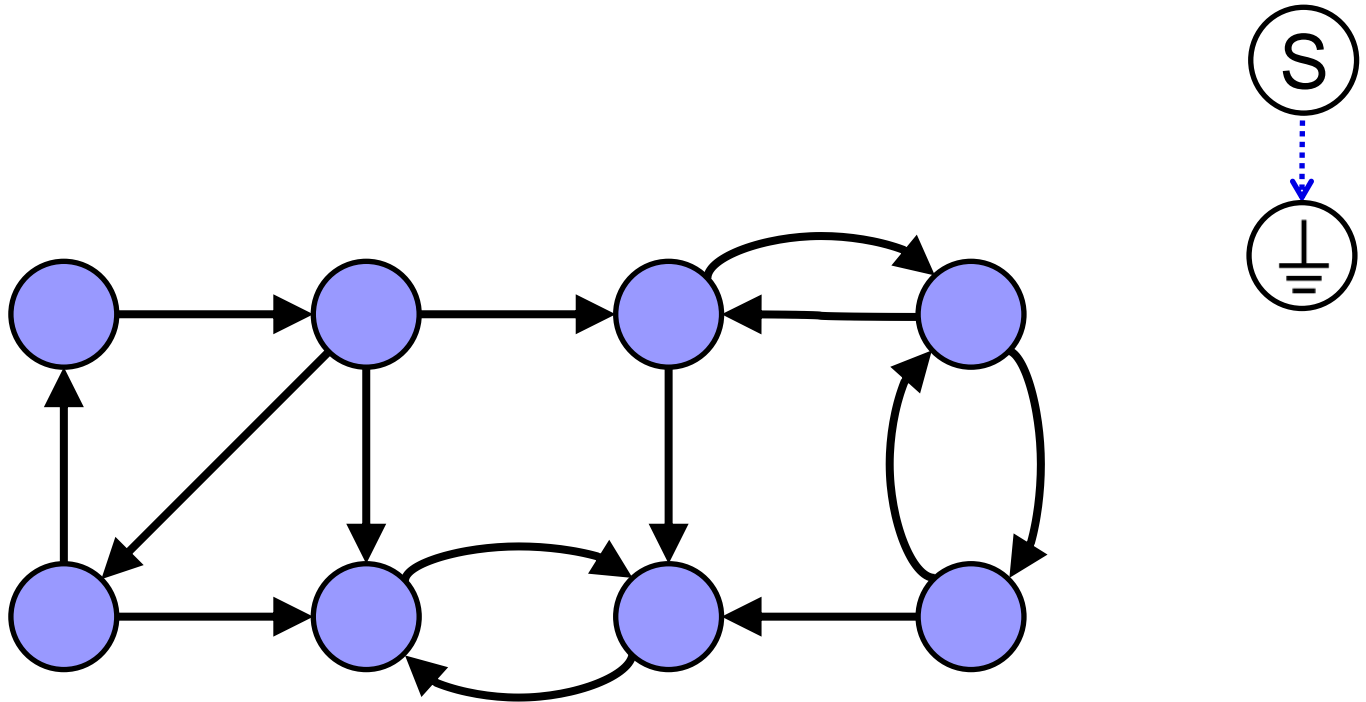
```
// every node has following fields:  
// index: a unique number to ID node  
// lowlink: ties node to others in SCC  
// pred: pointer to stack predecessor  
// instack: true if node is in stack
```

```
procedure push( v )  
// stack may be null  
  v.pred    = S;  
  v.instack = true;  
  S         = v;  
end push;
```

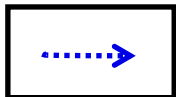
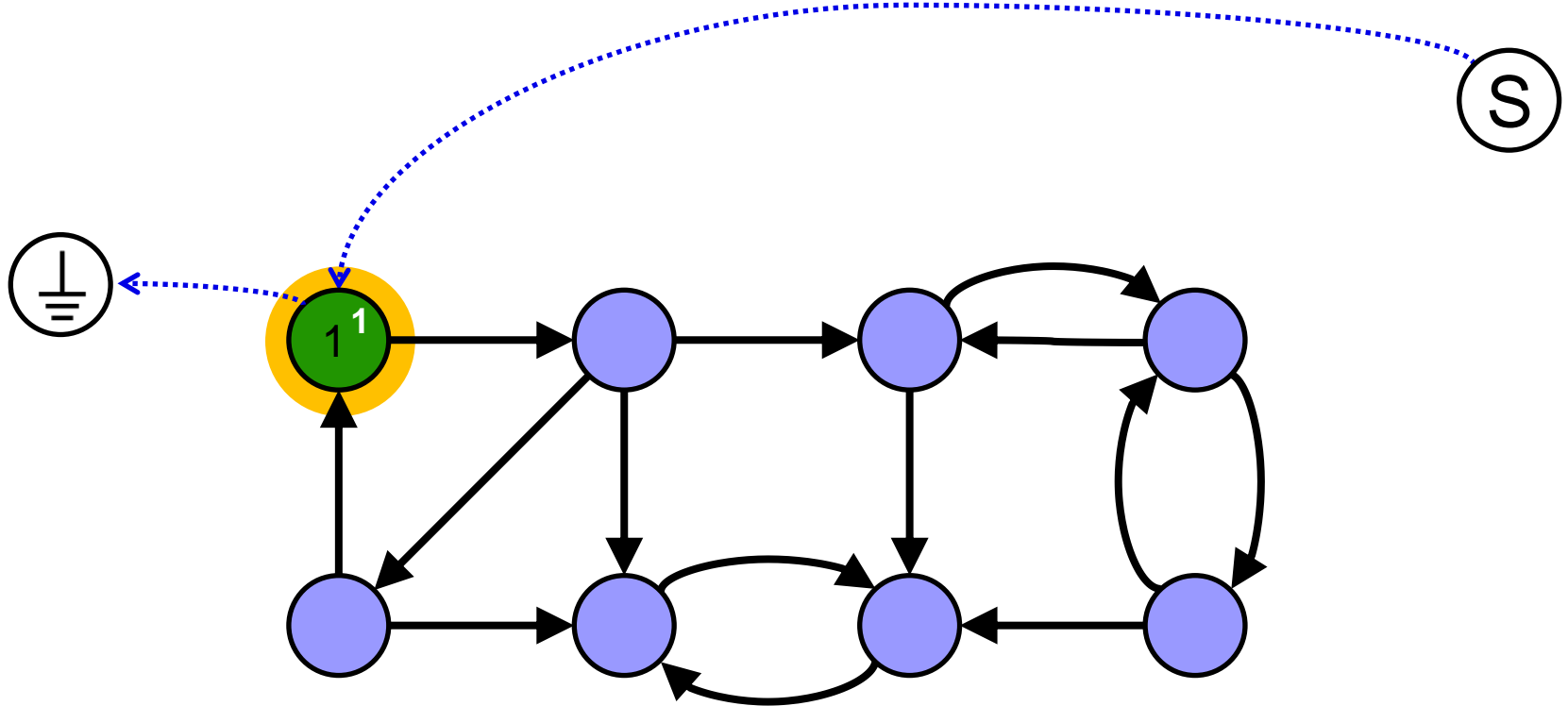
```
function pop( v )  
// val param v is stack copy  
  S      = v.pred;  
  v.pred = null;  
  v.instack = false;  
  return v;  
end pop;
```

```
procedure find_scc( v )  
  v.index = v.lowlink = ++index;  
  push( v );  
  foreach node w in succ( v ) do  
    if w.index = 0 then // not yet visited  
      find_scc( w );  
      v.lowlink = min( v.lowlink, w.lowlink );  
    elseif w.instack then  
      v.lowlink = min( v.lowlink, w.index );  
    end if  
  end foreach  
  
  if v.lowlink = v.index then // v: head of SCC  
    SCC++ // track how many SCCs found  
    repeat  
      x = pop( S );  
      add x to current strongly connected component;  
    until x = v;  
    output the current strongly connected component;  
  end if  
end find_scc;  
  
index = 0; // unique node number > 0  
S = null; // pointer to node stack  
SCC = 0; // number of SCCs in G  
foreach node v in V do  
  if v.index = 0 then // yet unvisited  
    find_scc( v );  
  end if  
end foreach;
```

# Tarjan's Algorithm



# Tarjan's Algorithm



pred



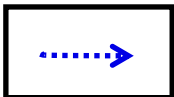
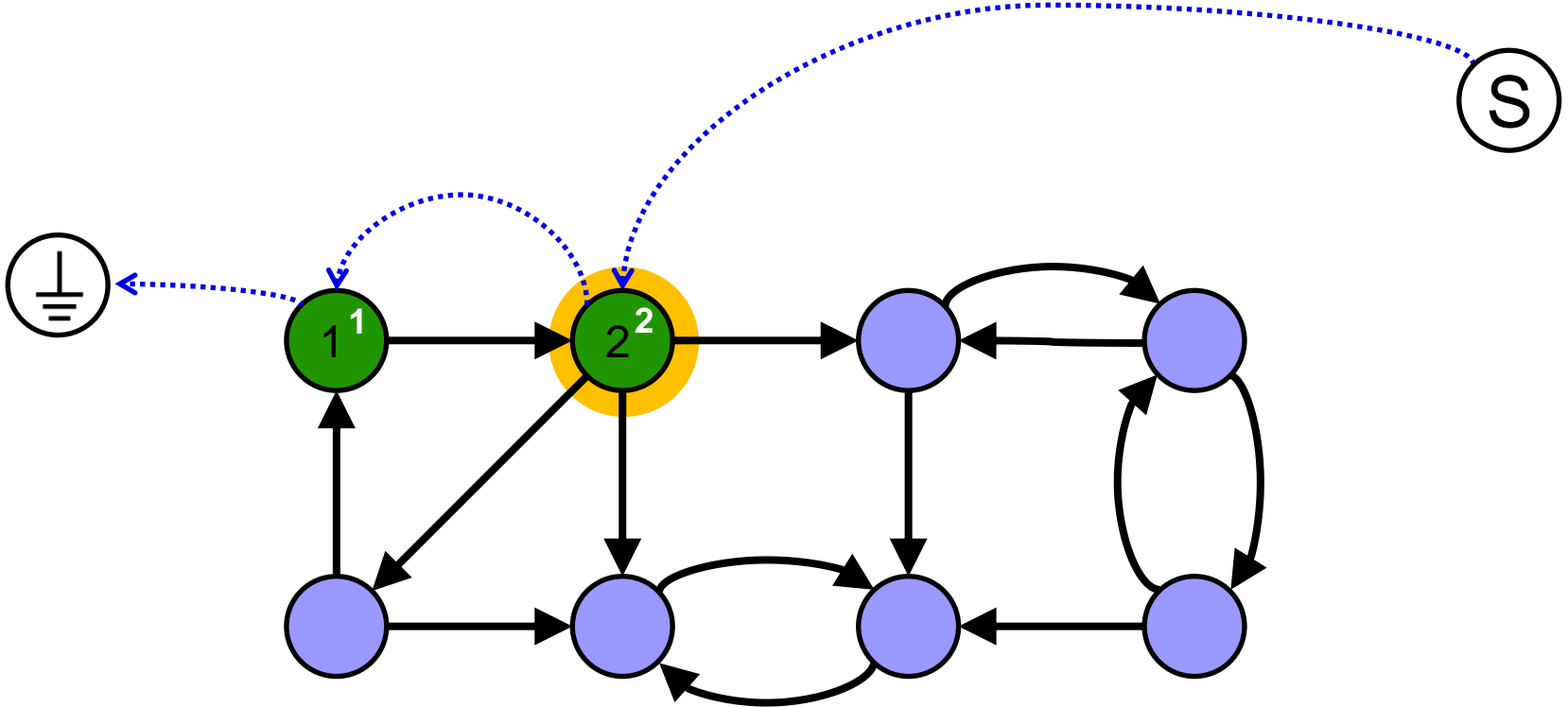
instack = true



instack = false



# Tarjan's Algorithm



pred

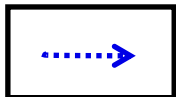
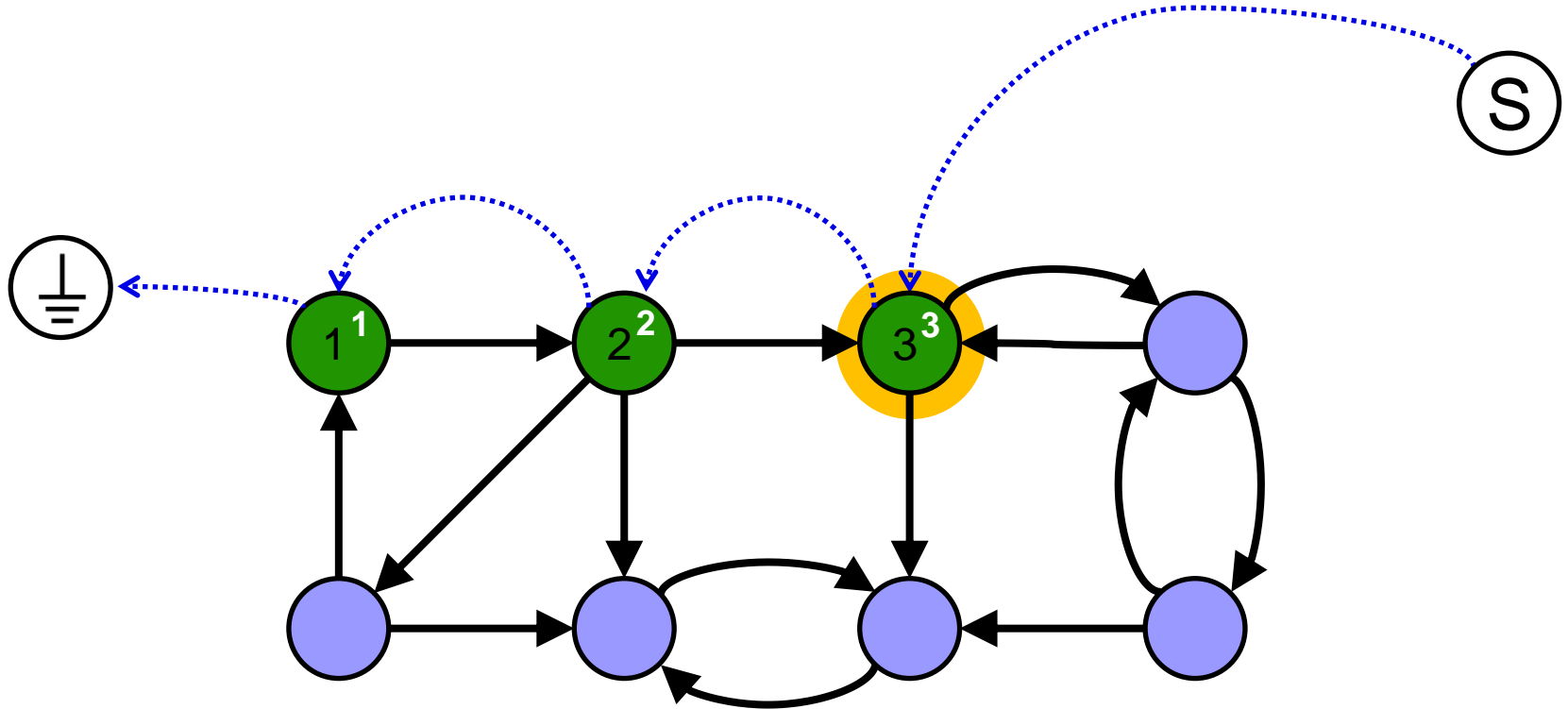


instack = true



instack = false

# Tarjan's Algorithm



pred

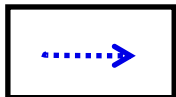
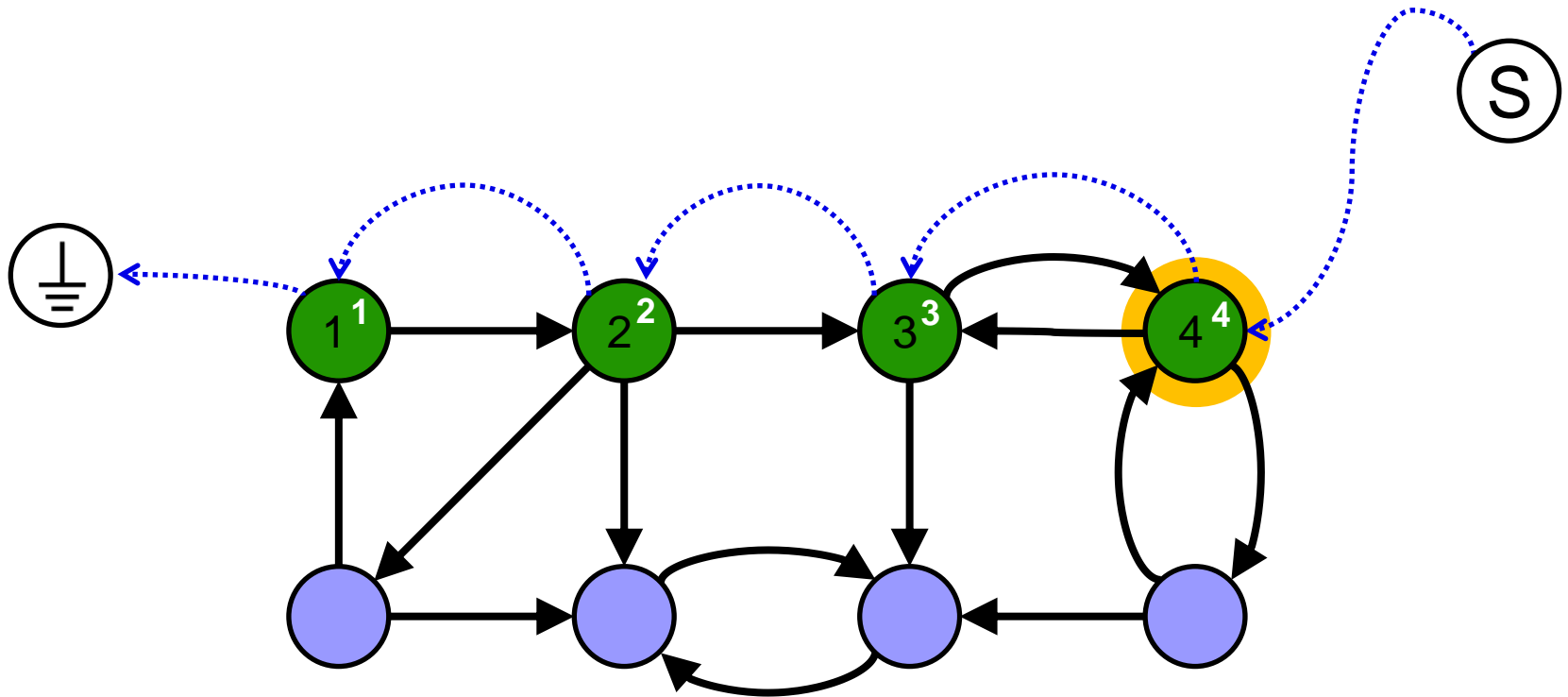


instack = true



instack = false

# Tarjan's Algorithm



pred

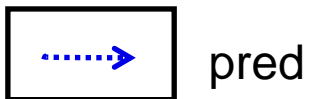
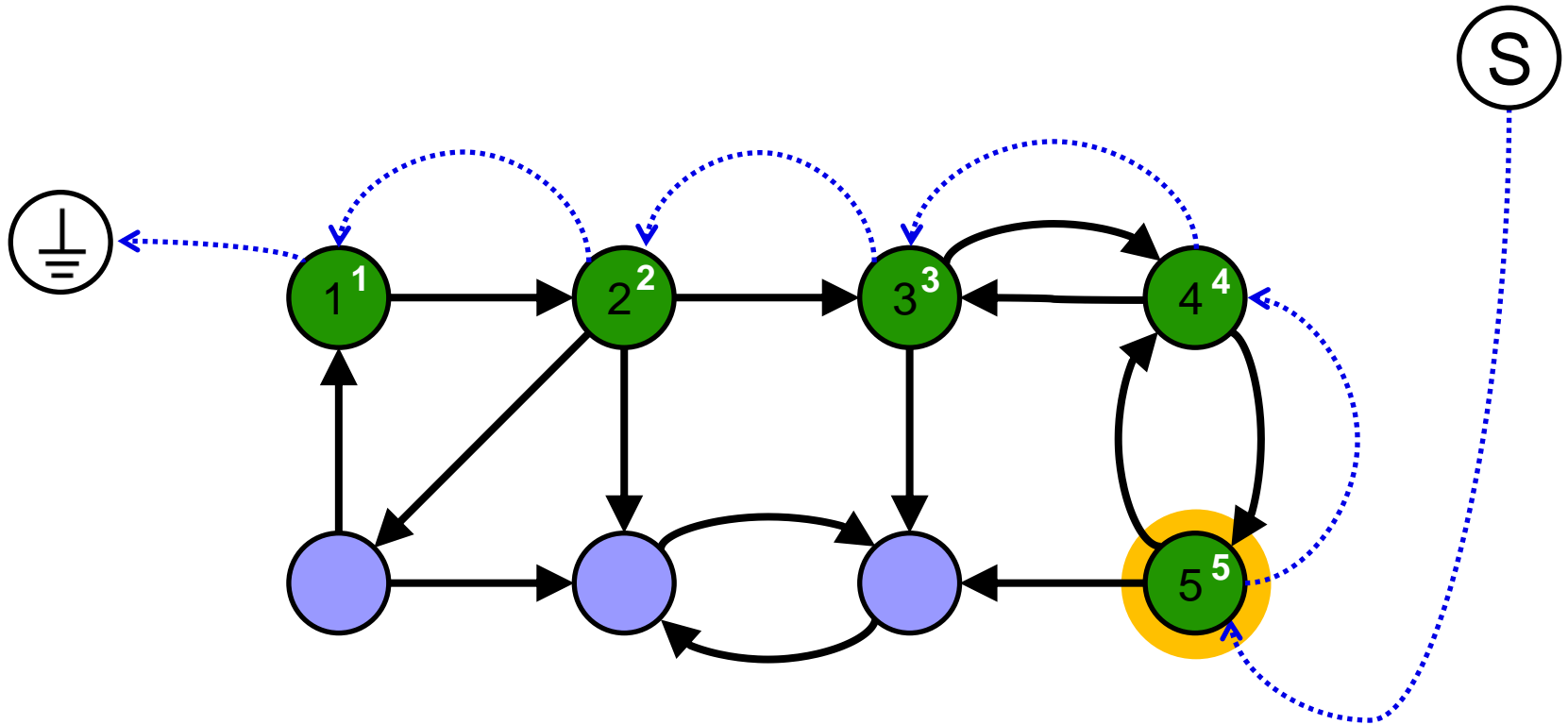


instack = true

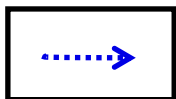
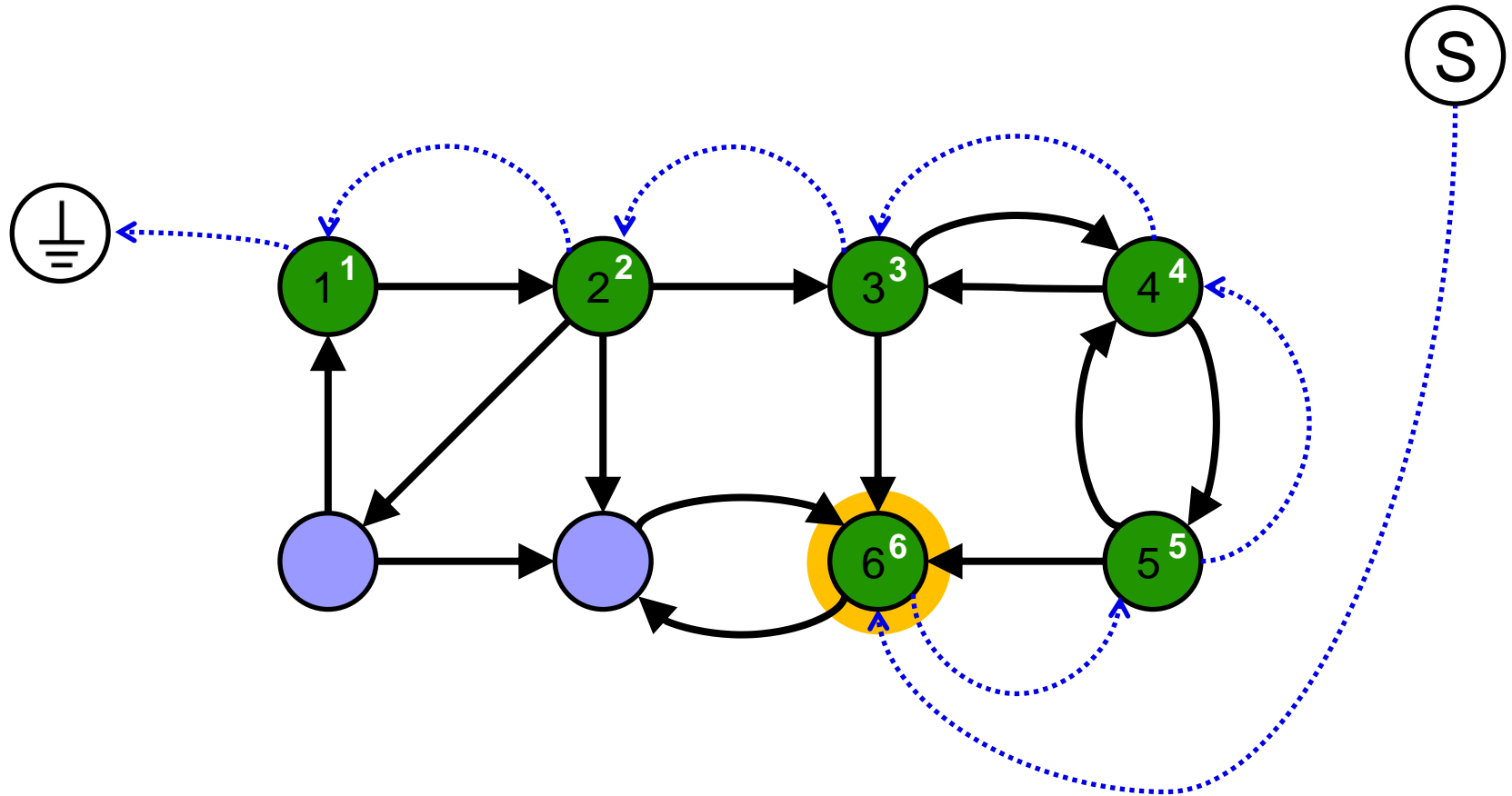


instack = false

# Tarjan's Algorithm



# Tarjan's Algorithm



pred

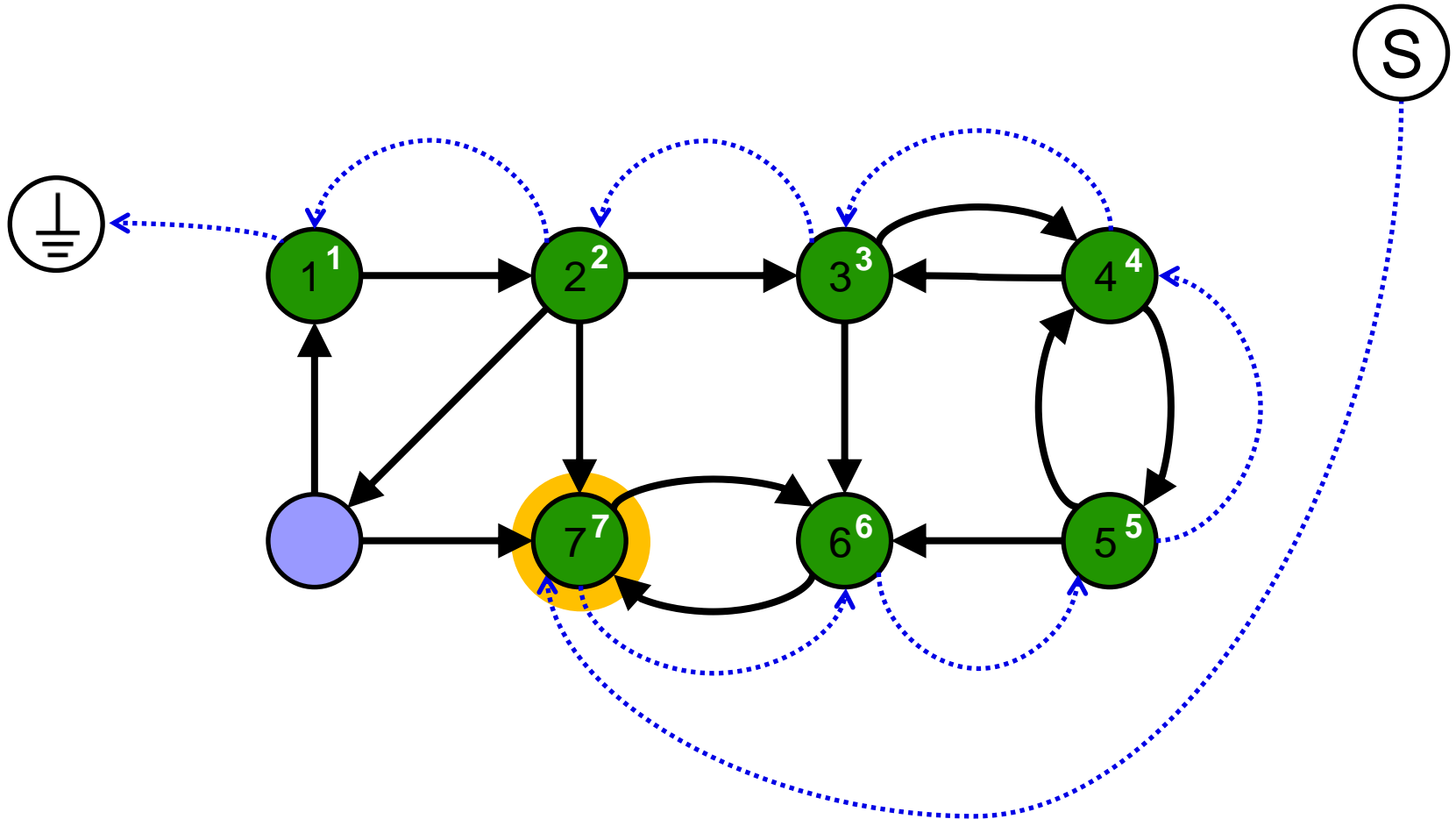


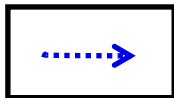
instack = true



instack = false

# Tarjan's Algorithm

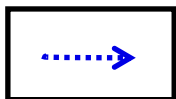
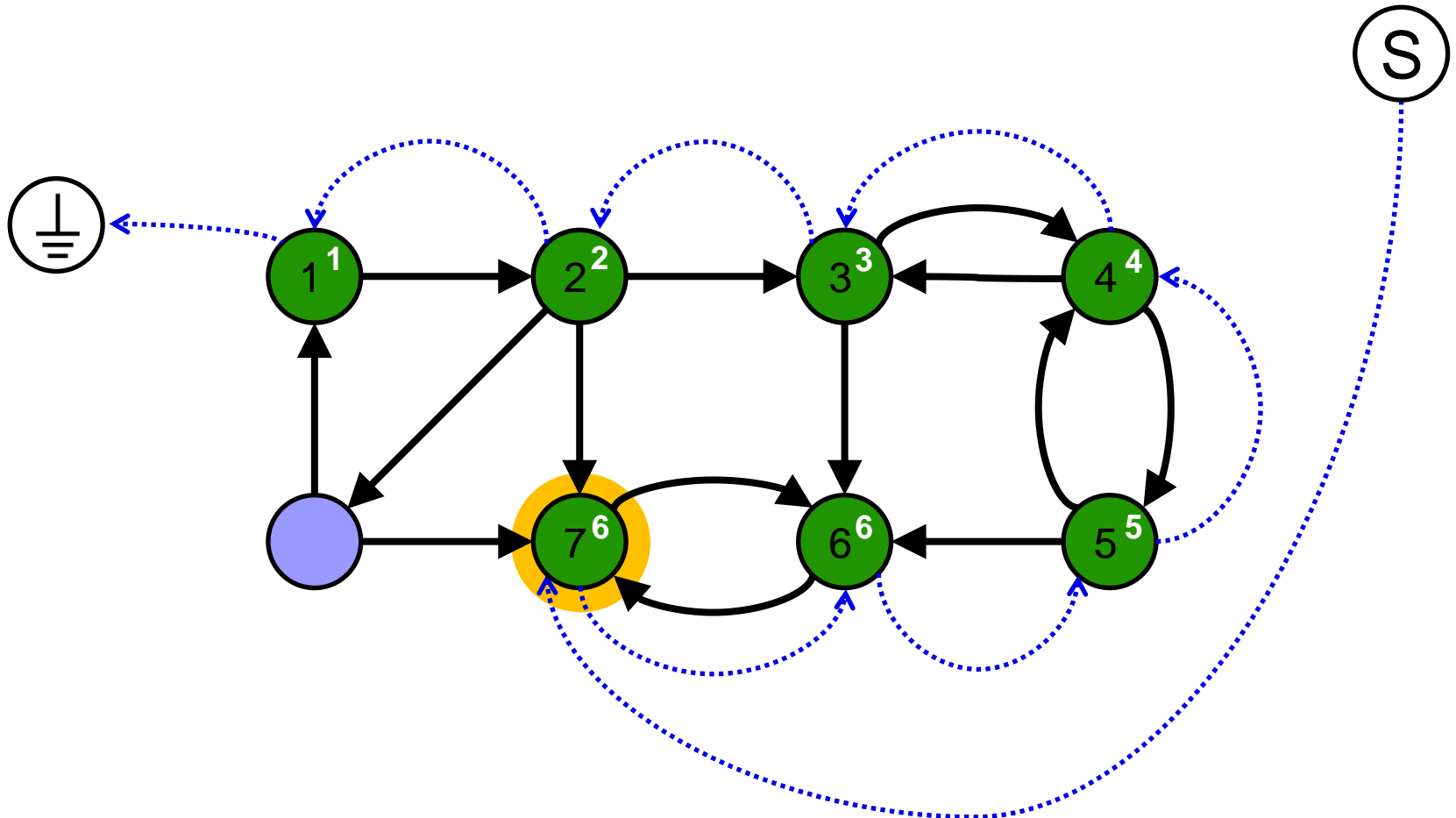


 pred

 instack = true

 instack = false

# Tarjan's Algorithm



pred

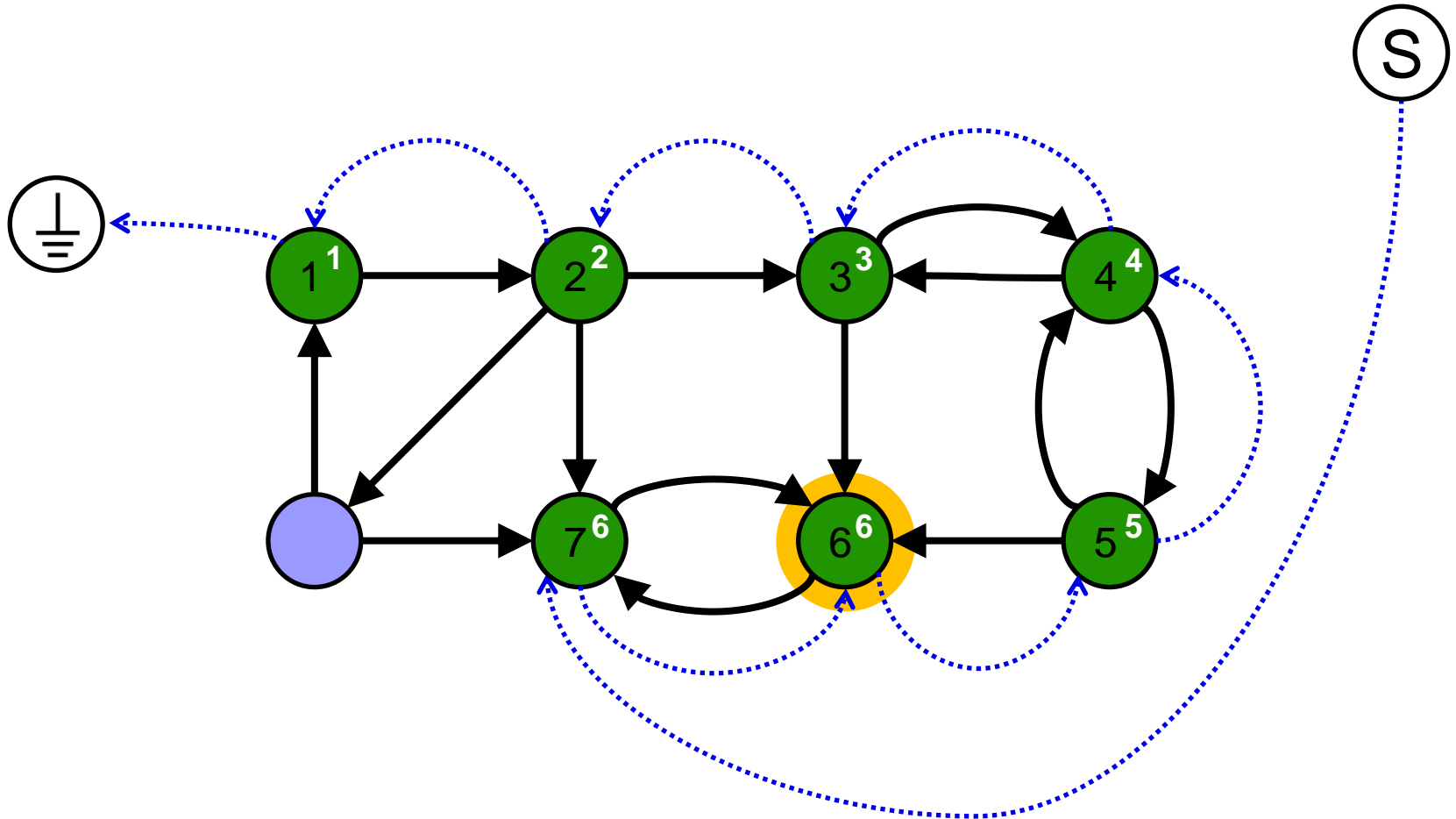


instack = true



instack = false

# Tarjan's Algorithm



pred



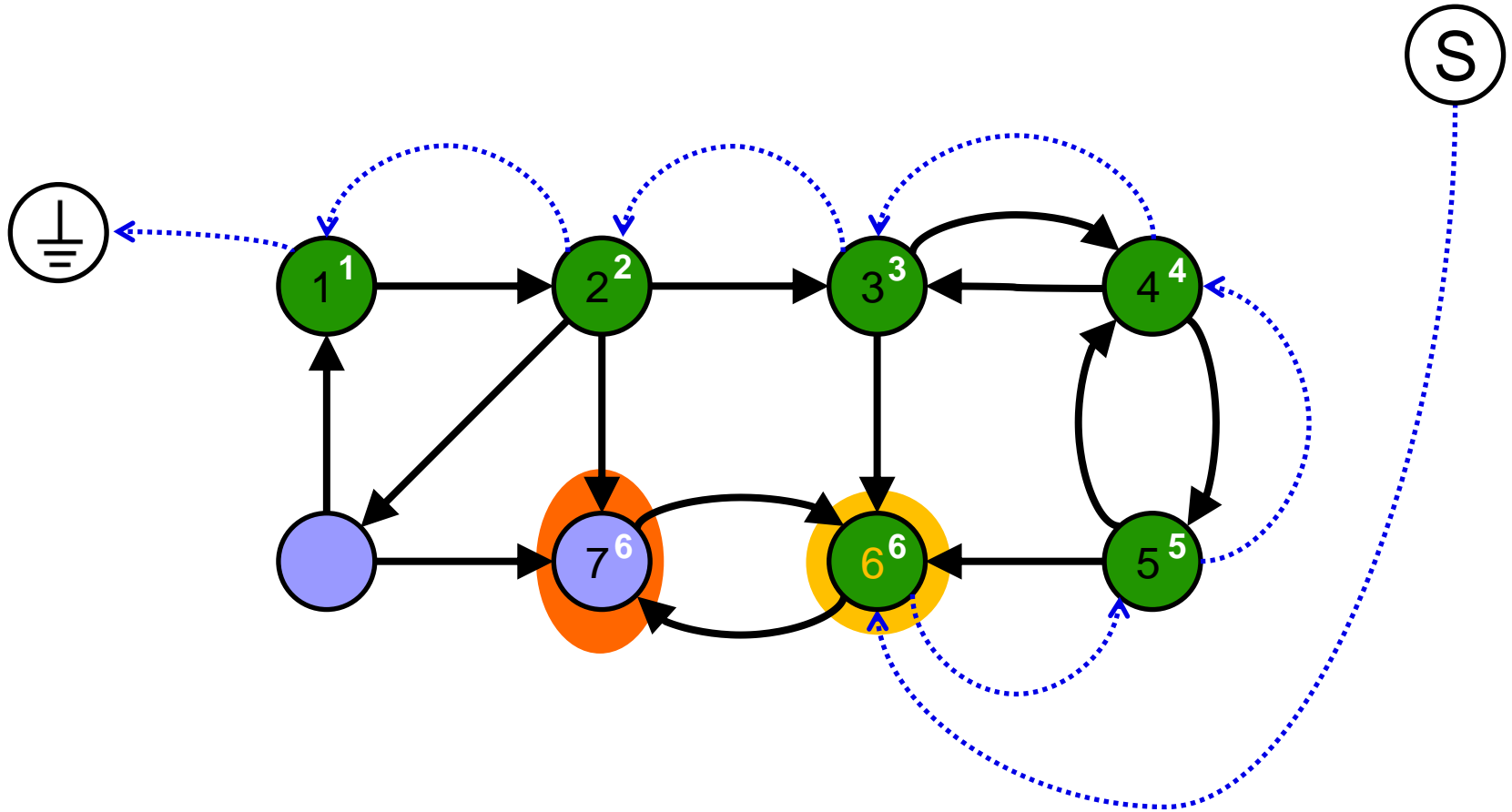
instack = true

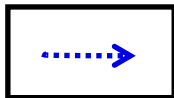


instack = false



# Tarjan's Algorithm

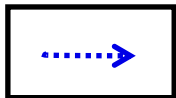
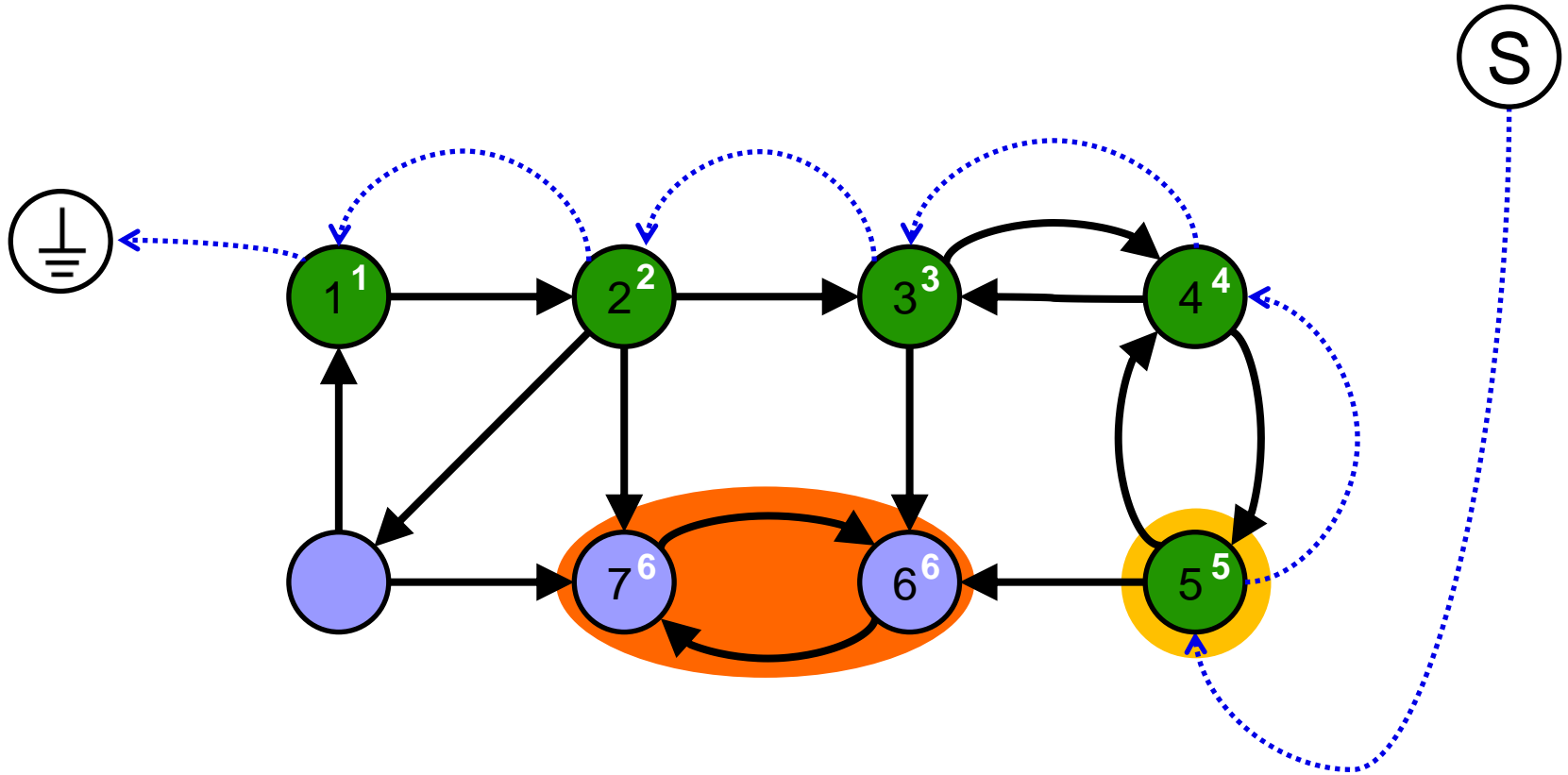


 pred

 instack = true

 instack = false

# Tarjan's Algorithm



pred

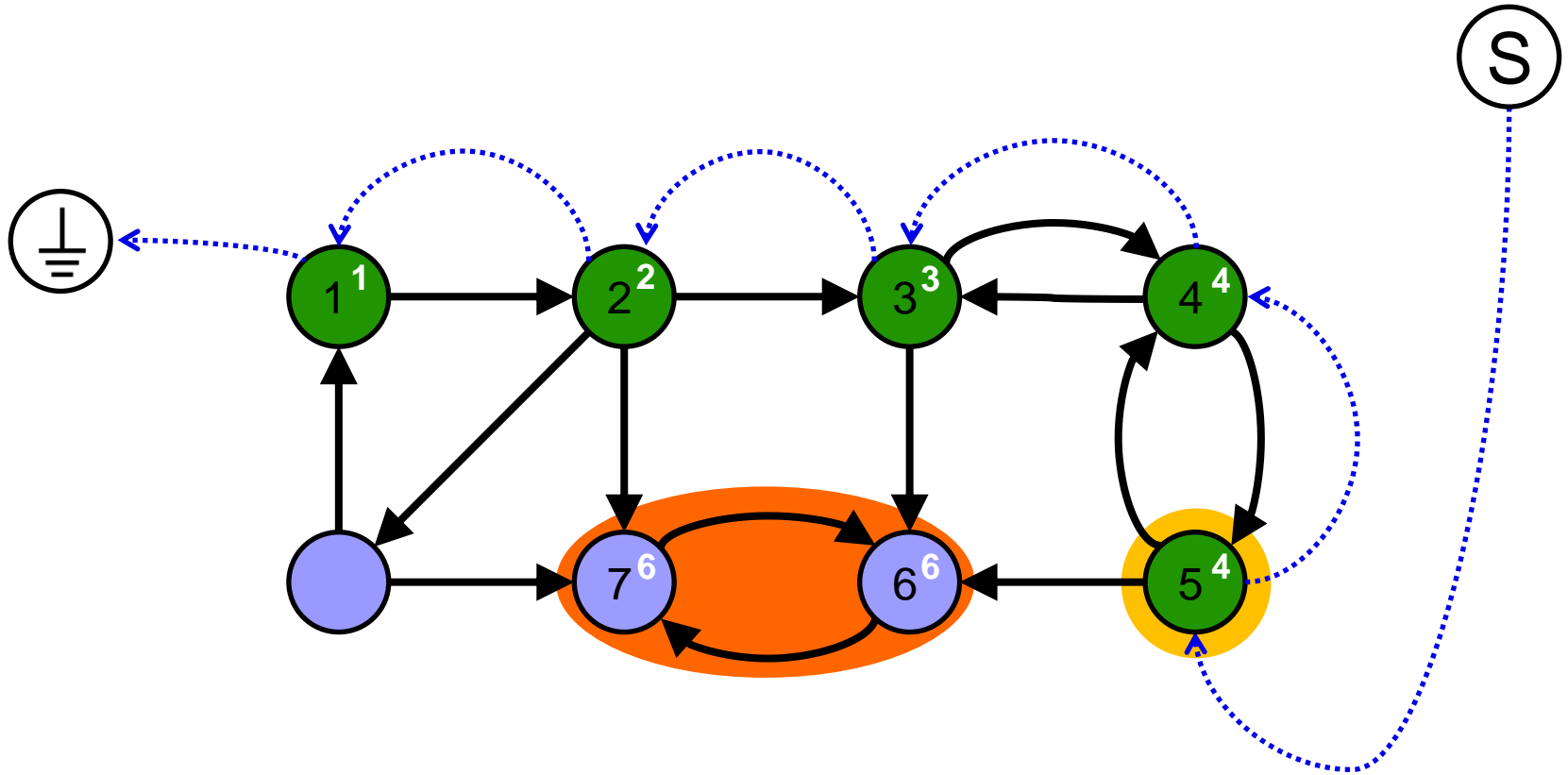


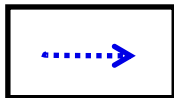
instack = true



instack = false

# Tarjan's Algorithm

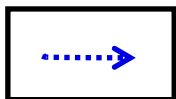
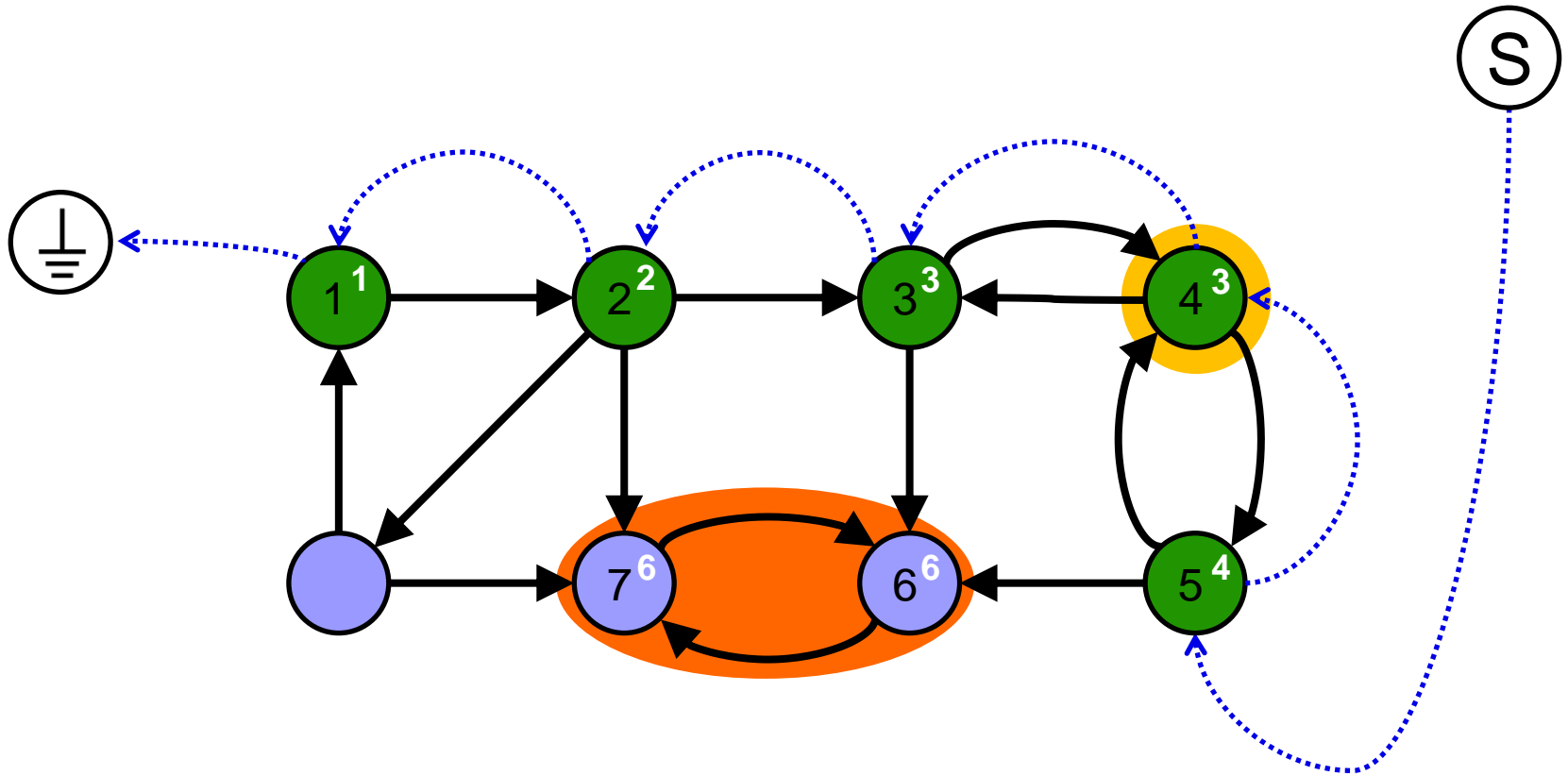


 pred

 instack = true

 instack = false

# Tarjan's Algorithm



pred

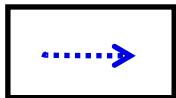
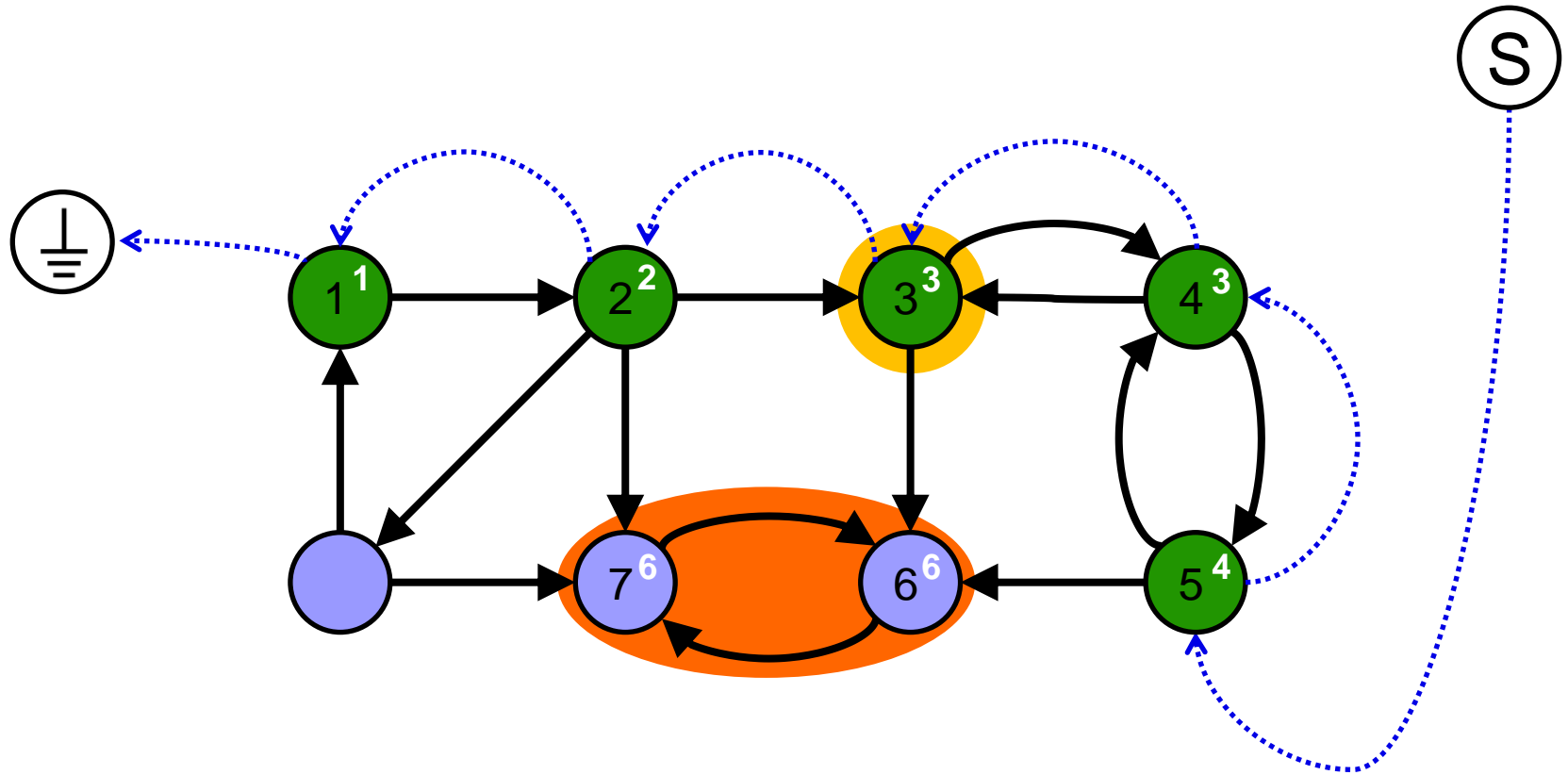


instack = true



instack = false

# Tarjan's Algorithm



pred

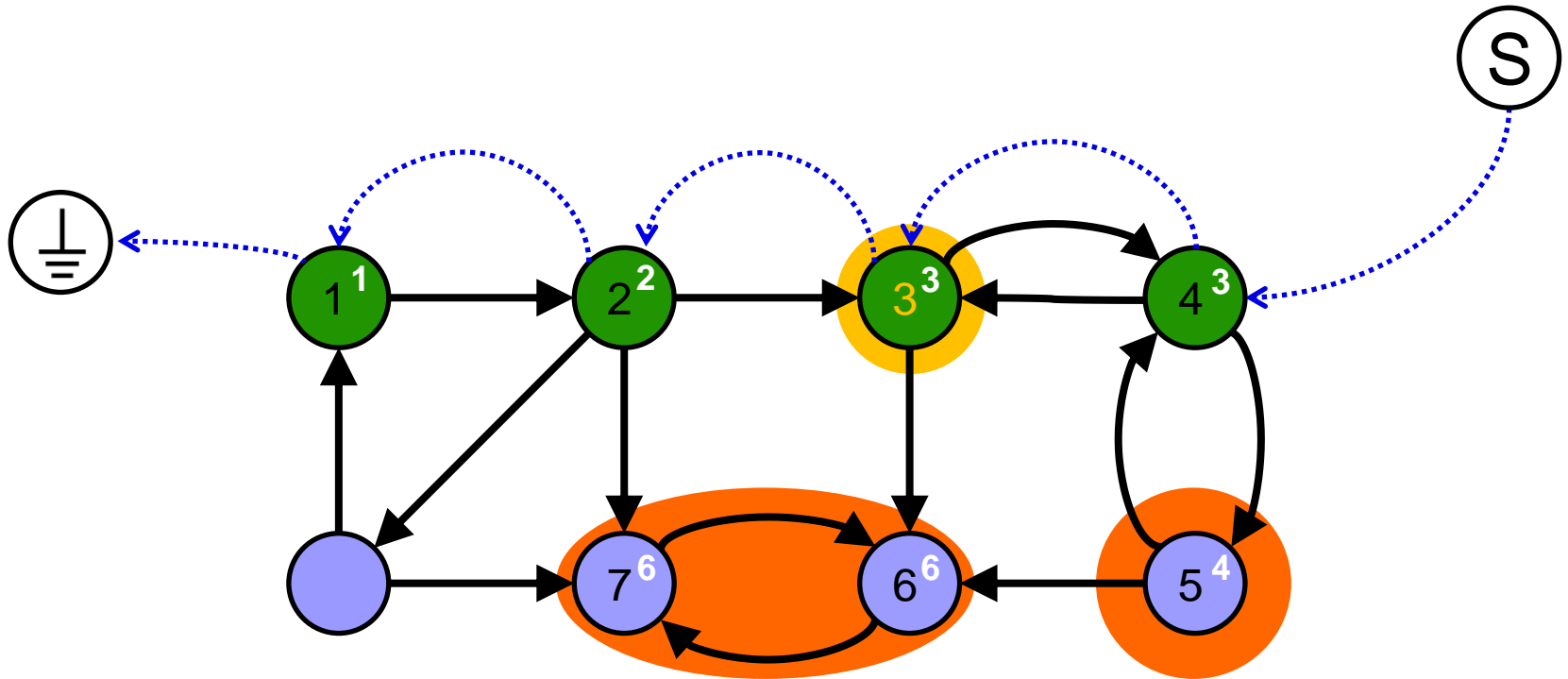


instack = true



instack = false

# Tarjan's Algorithm



pred

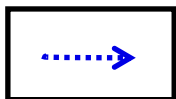
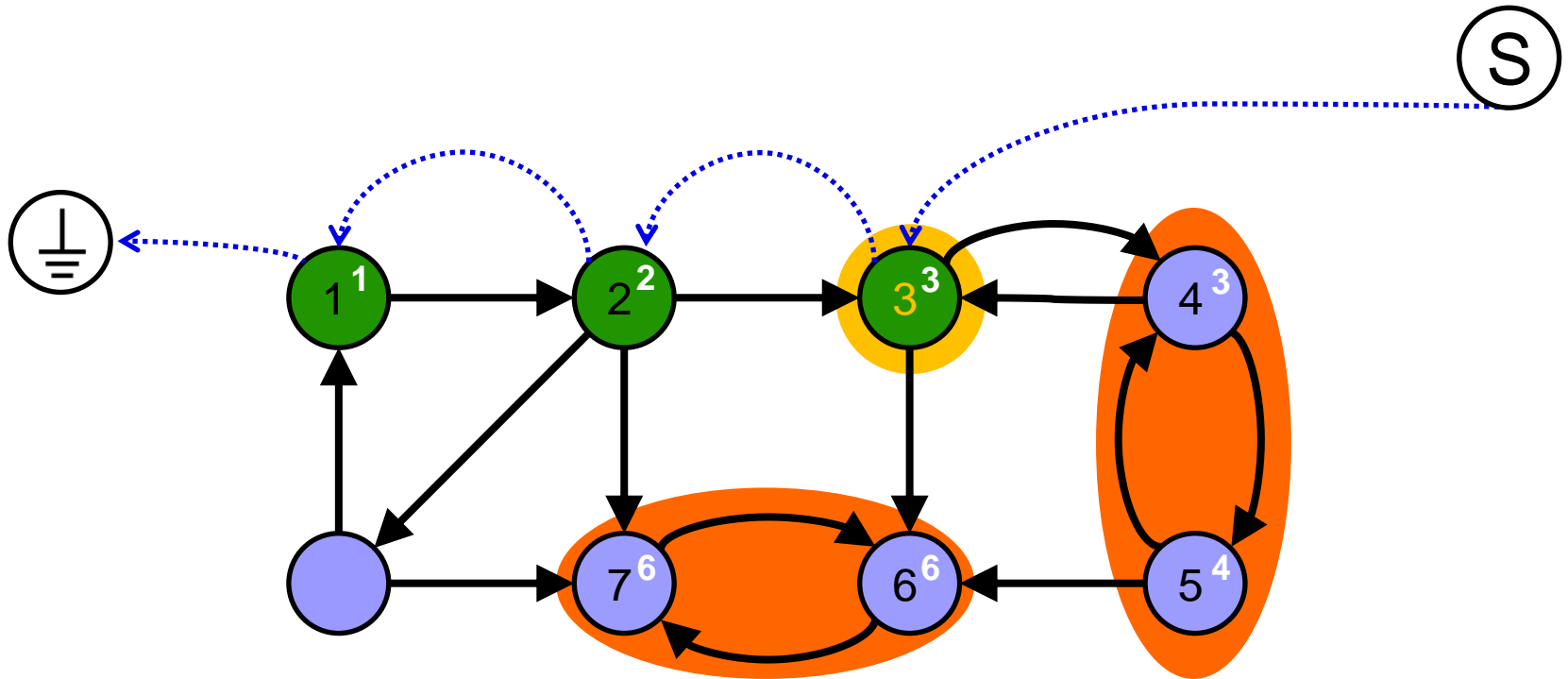


instack = true



instack = false

# Tarjan's Algorithm



pred

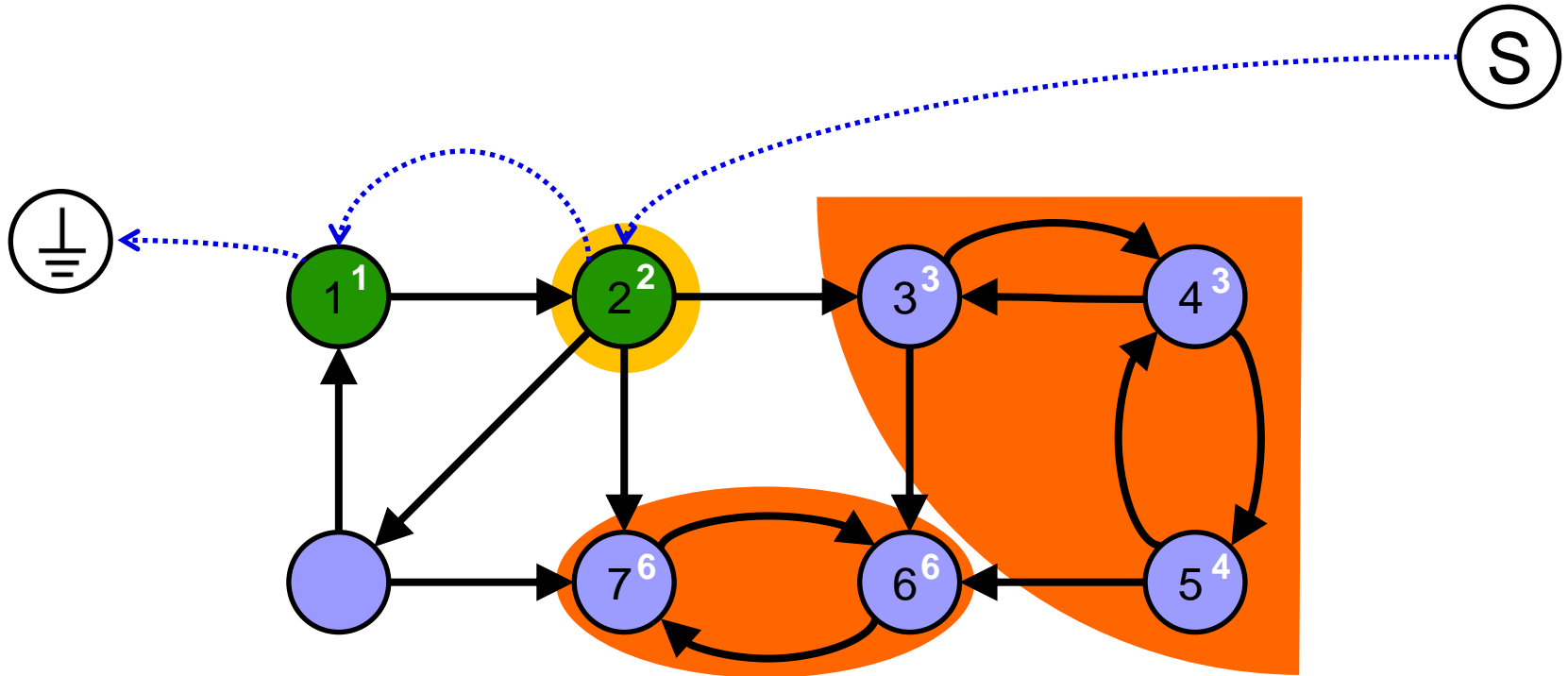


instack = true



instack = false

# Tarjan's Algorithm



pred



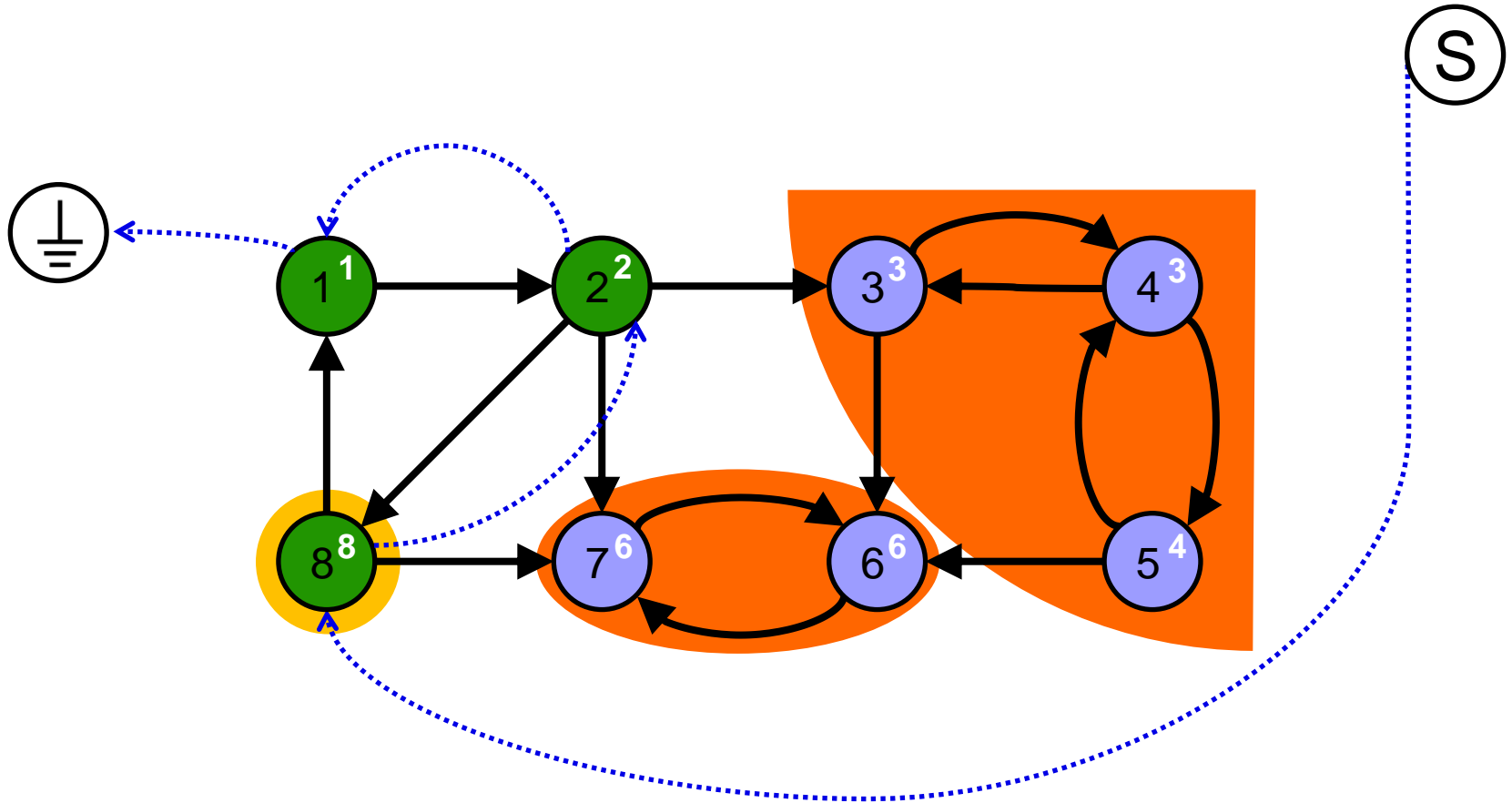
instack = true

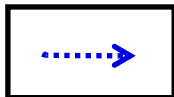


instack = false



# Tarjan's Algorithm

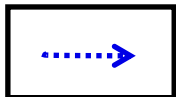
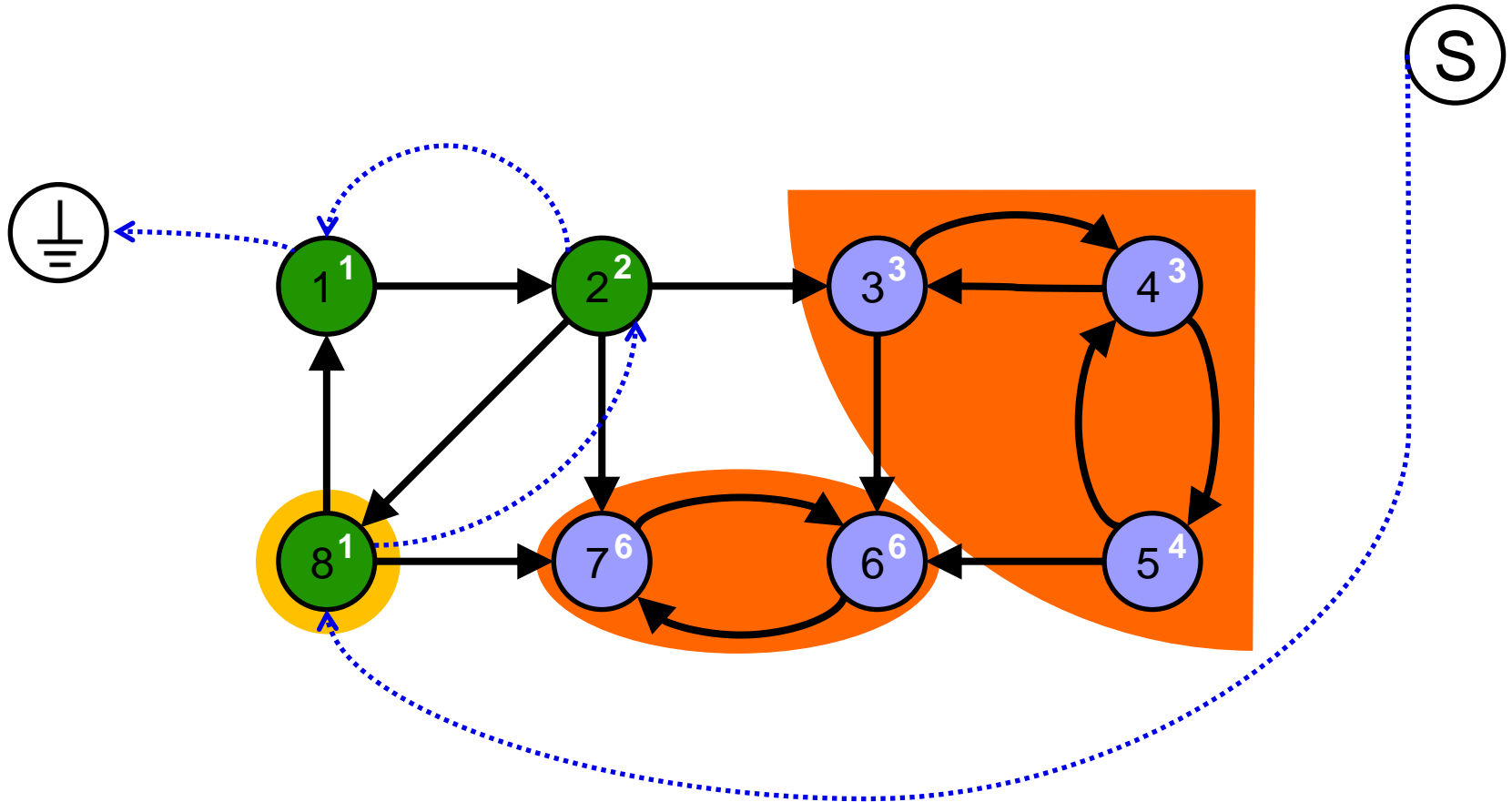


 pred

 instack = true

 instack = false

# Tarjan's Algorithm



pred

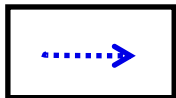
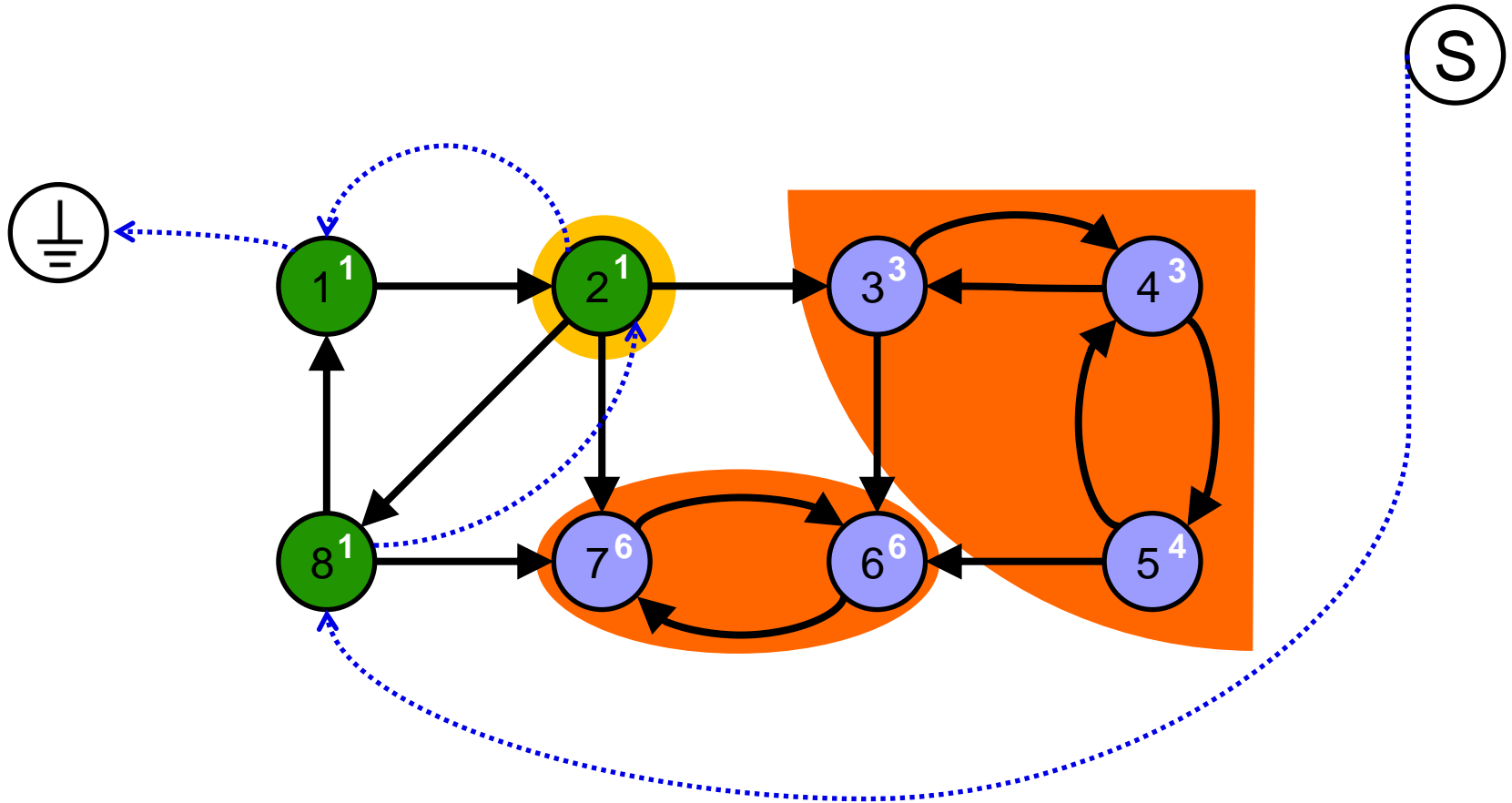


instack = true



instack = false

# Tarjan's Algorithm



pred

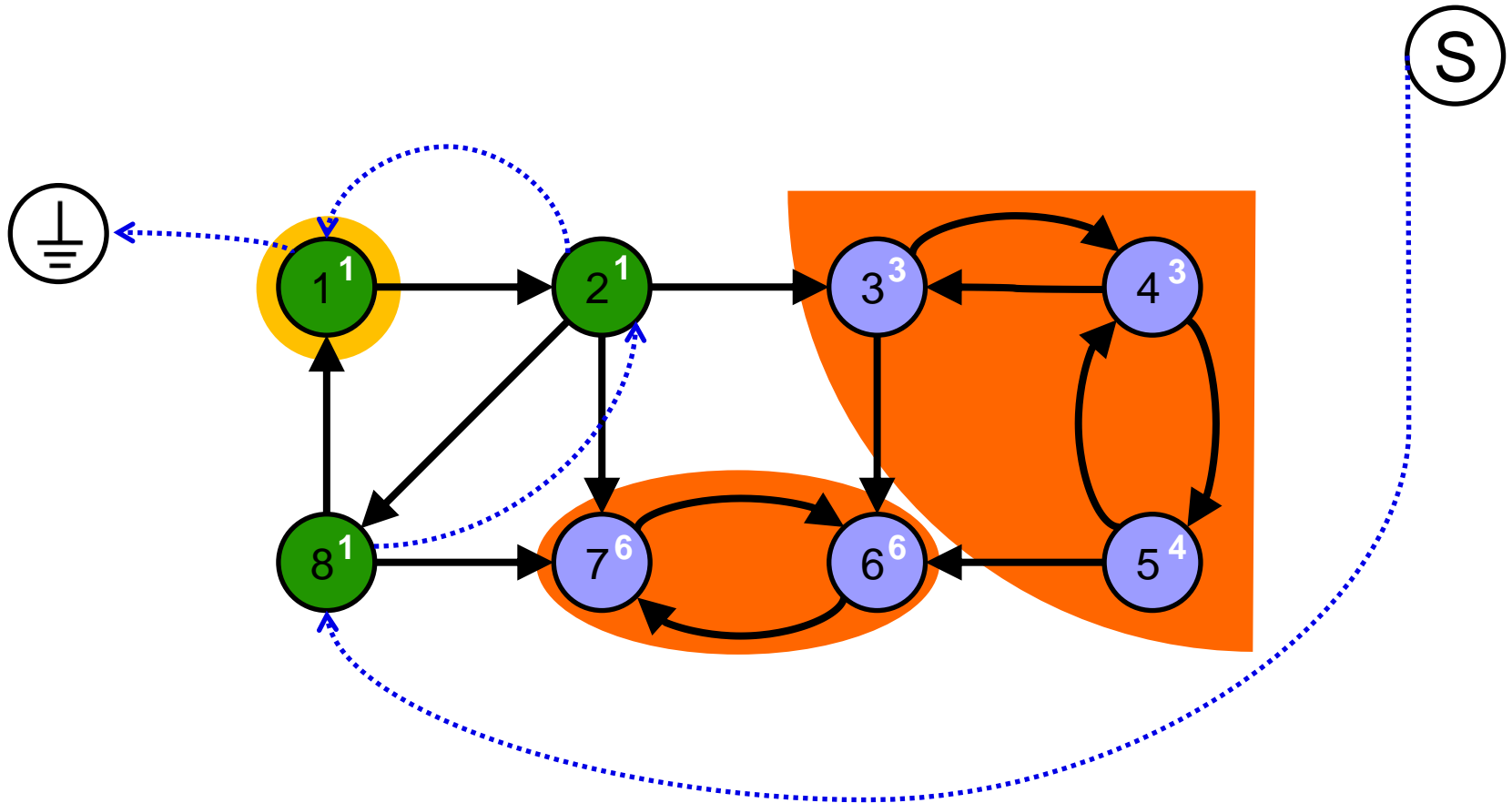


instack = true



instack = false

# Tarjan's Algorithm



pred

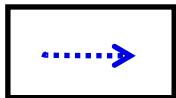
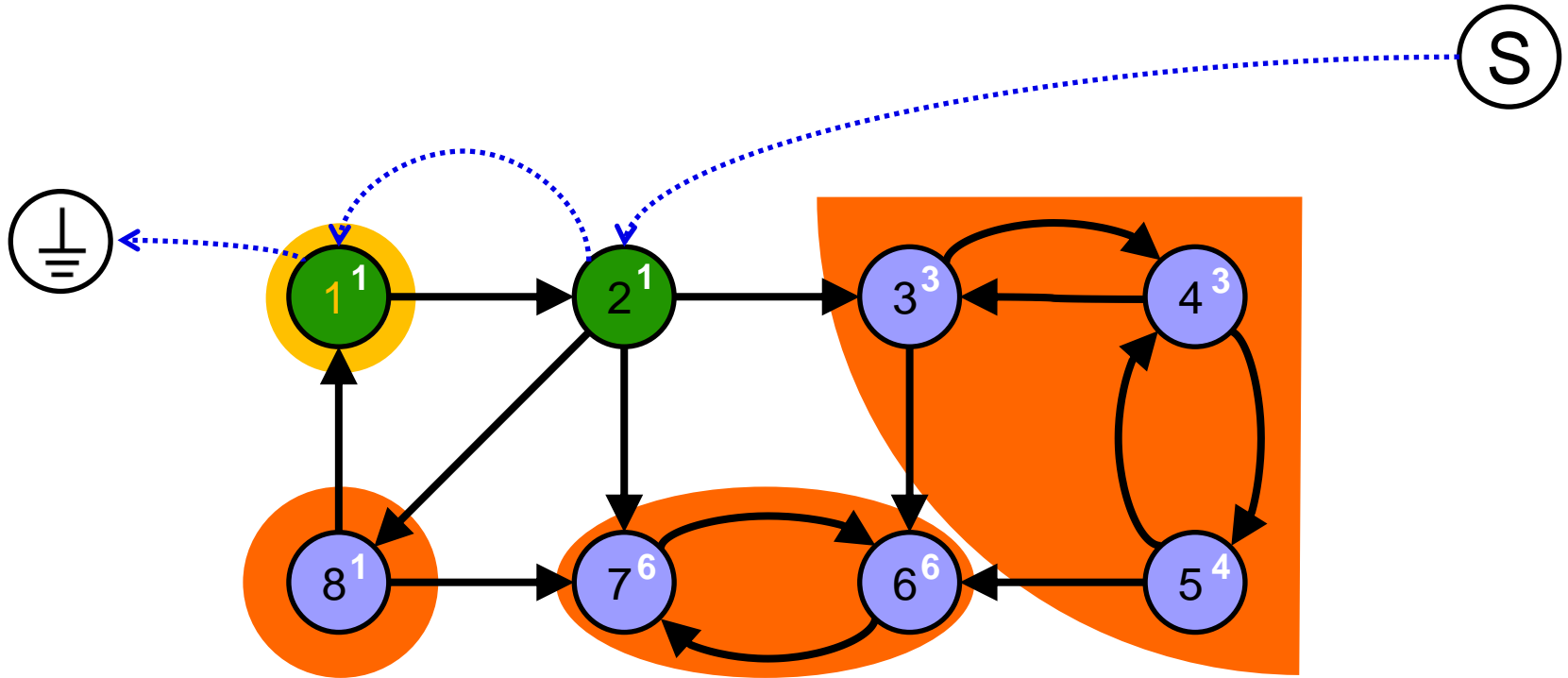


instack = true



instack = false

# Tarjan's Algorithm



pred

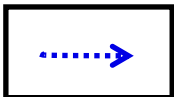
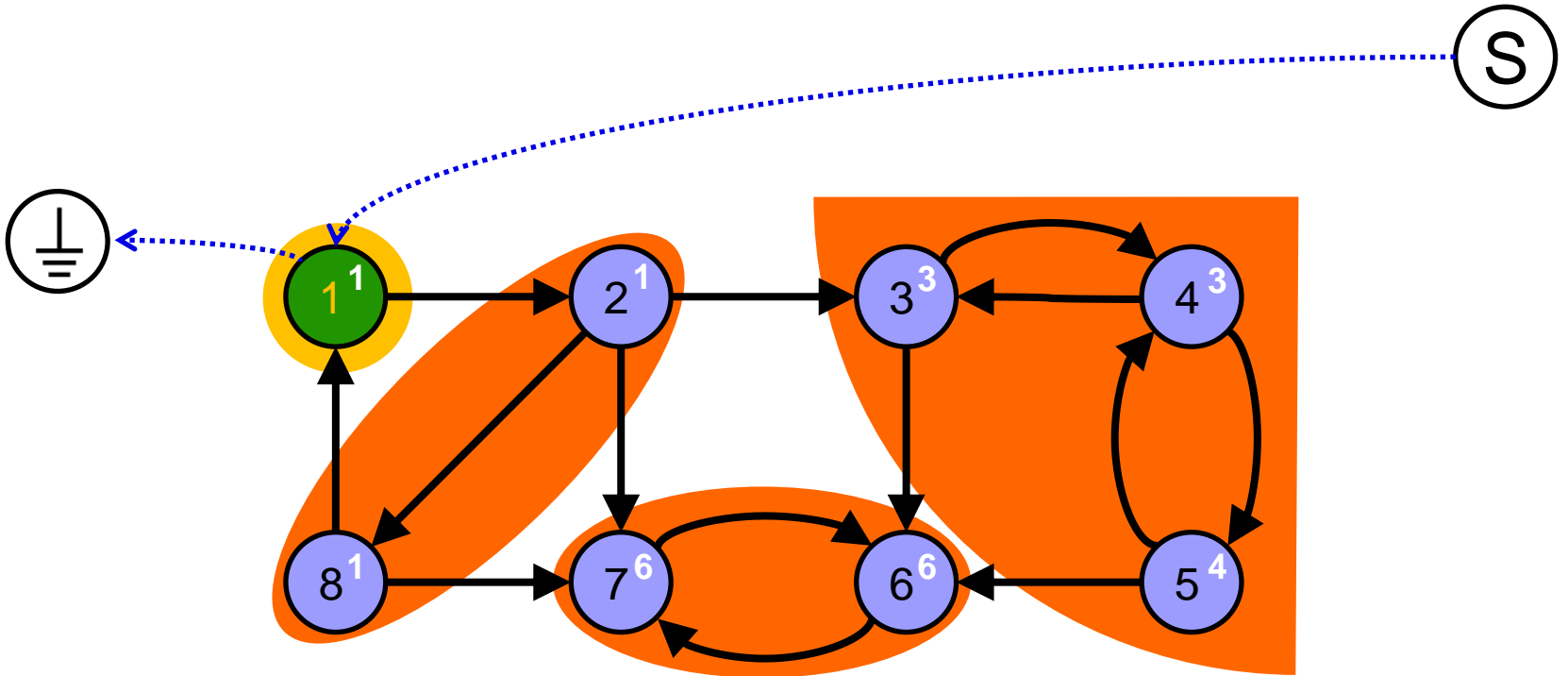


instack = true



instack = false

# Tarjan's Algorithm



pred

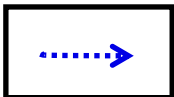
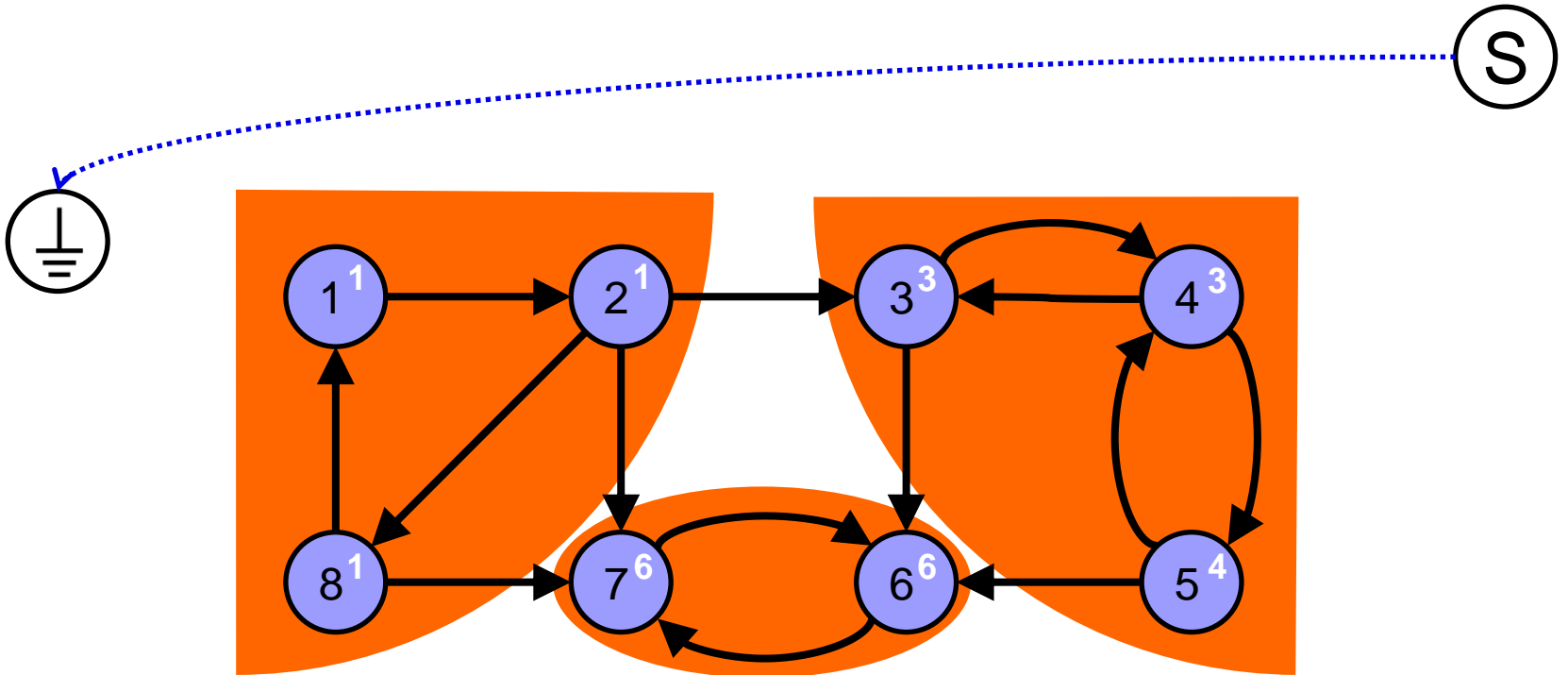


instack = true



instack = false

# Tarjan's Algorithm



pred



instack = true



instack = false

# Tarjan's Algorithm

## ■ Complexity:

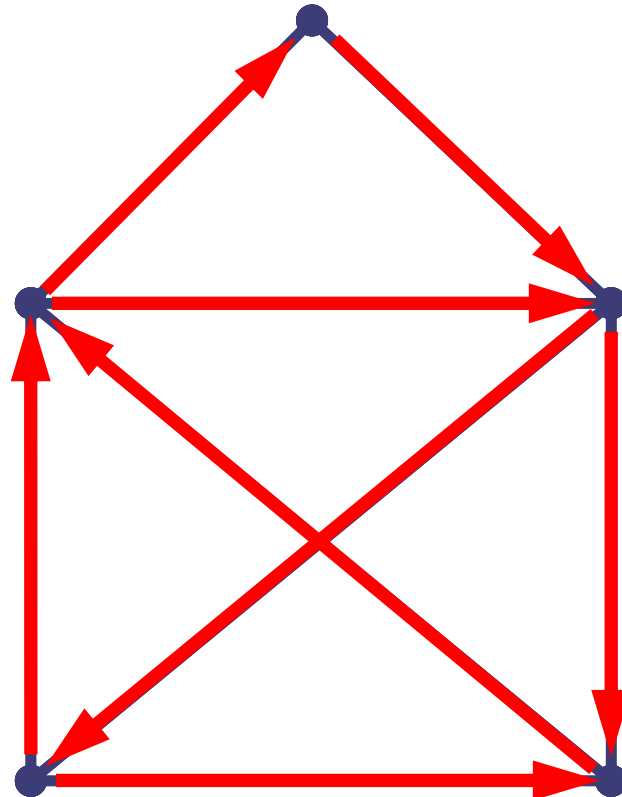
- The Tarjan's algorithm performs only one complete traversal of the graph.
- If the graph is represented as an *adjacency list* then the algorithm runs in  $\Theta(|V|+|E|)$  time (linear time).
- If the graph is represented as an *adjacency matrix* then the algorithm runs in  $O(|V|^2)$  time.
- The Tarjan's algorithm runs faster than the Kosaraju-Sharir algorithm.



# Euler Trail

## ■ Euler Trail Problem:

Does a (directed or undirected) graph  $G$  contain a trail (trail is similar to path but vertices can repeat and edges cannot repeat) that visits every edge exactly once?



# Euler Trail - Properties

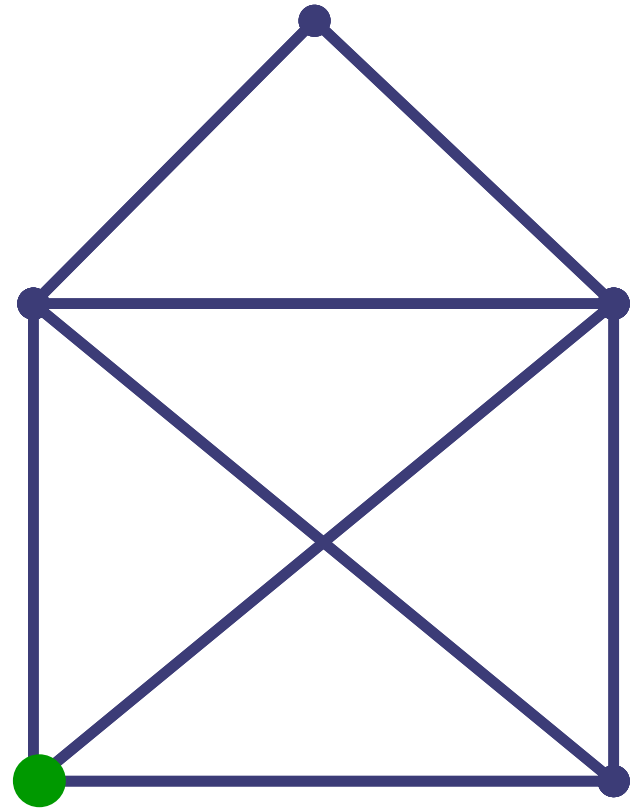
- **Theorem:** A graph  $G$  has an Euler trail if and only if it is connected and has 0 or 2 vertices of odd degree.
- We can distinguish two cases:
  1. Euler trail starts and ends in the same vertex.  
(Eulerian Tour)  
→ Every vertex must have even degree.
  2. Euler trail starts and ends in the different vertices.  
→ The starting and ending vertex must have odd degree and the others have even degree.

# Euler Trail

**input:** graph  $G = (V, E)$

**output:** trail (as a stack with edges)

```
procedure euler-trail(vertex  $v$ );  
{  
  foreach vertex  $u$  in succ( $v$ ) do {  
    remove edge( $v,u$ ) from graph;  
    euler-trail( $u$ );  
    push(edge( $v,u$ ));  
  }  
}
```

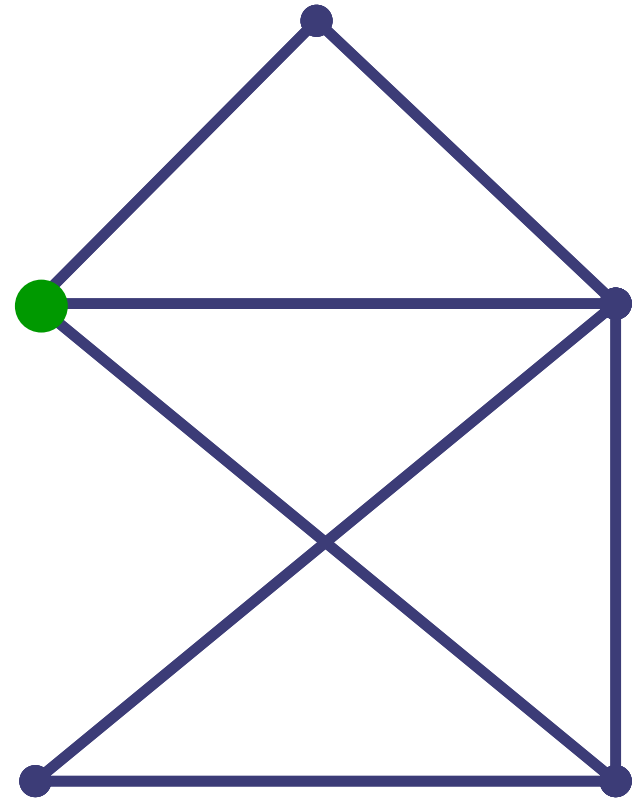


# Euler Trail

**input:** graph  $G = (V, E)$

**output:** trail (as a stack with edges)

```
procedure euler-trail(vertex  $v$ );  
{  
  foreach vertex  $u$  in succ( $v$ ) do {  
    remove edge( $v, u$ ) from graph;  
    euler-trail( $u$ );  
    push(edge( $v, u$ ));  
  }  
}
```

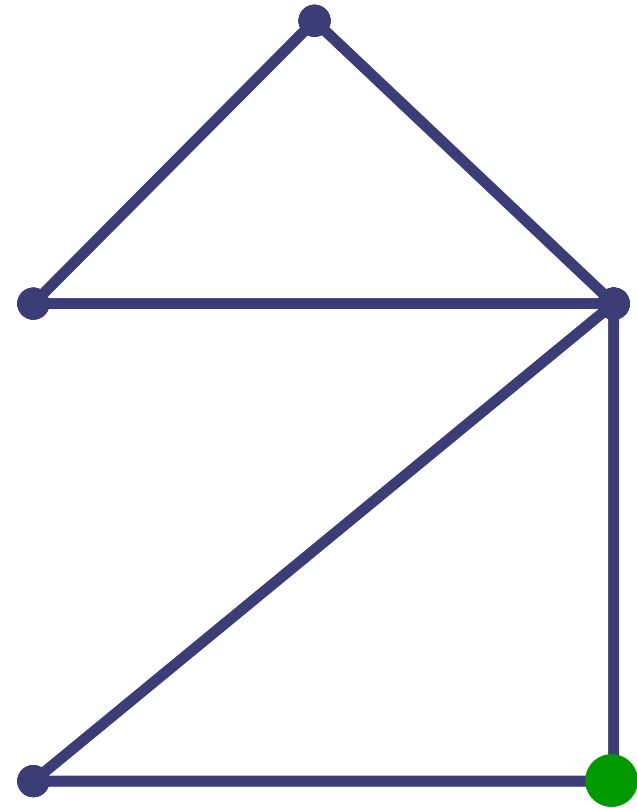


# Euler Trail

**input:** graph  $G = (V, E)$

**output:** trail (as a stack with edges)

```
procedure euler-trail(vertex  $v$ );  
{  
  foreach vertex  $u$  in succ( $v$ ) do {  
    remove edge( $v, u$ ) from graph;  
    euler-trail( $u$ );  
    push(edge( $v, u$ ));  
  }  
}
```

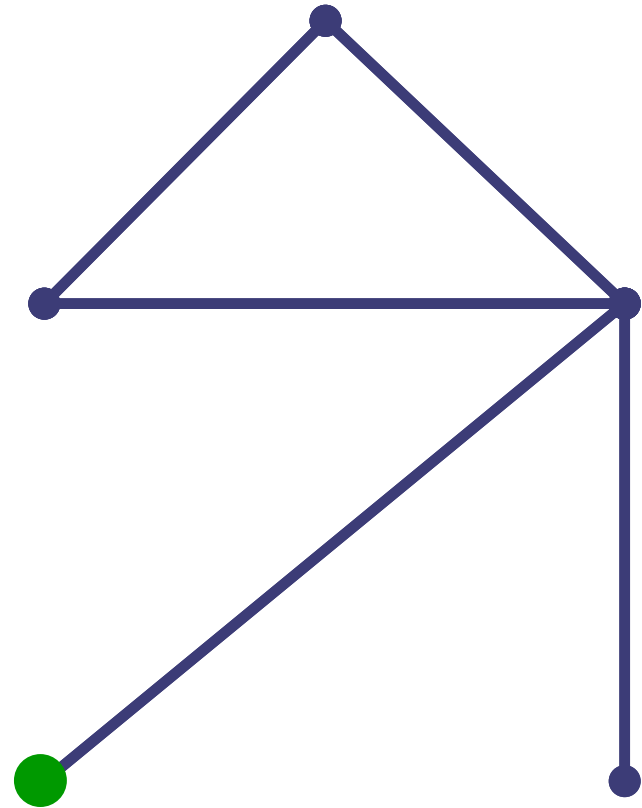


# Euler Trail

**input:** graph  $G = (V, E)$

**output:** trail (as a stack with edges)

```
procedure euler-trail(vertex  $v$ );  
{  
  foreach vertex  $u$  in succ( $v$ ) do {  
    remove edge( $v, u$ ) from graph;  
    euler-trail( $u$ );  
    push(edge( $v, u$ ));  
  }  
}
```

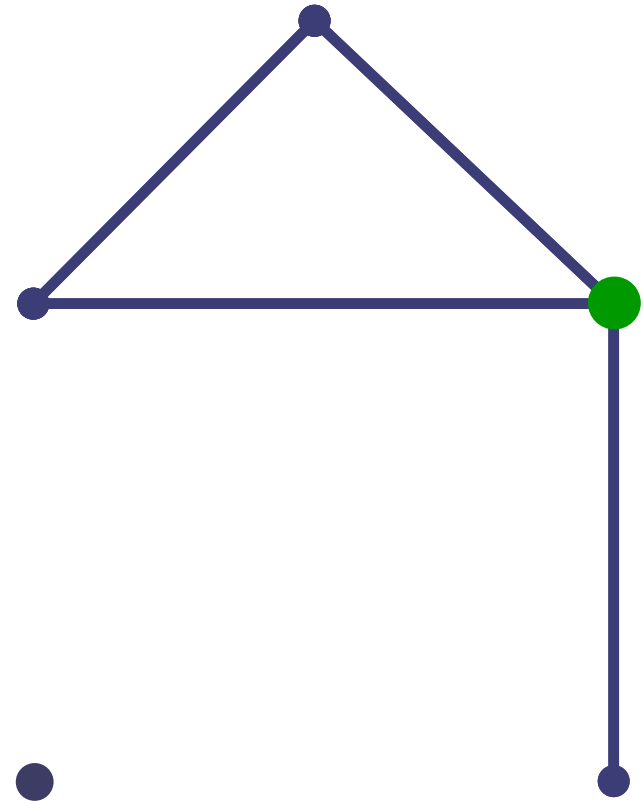


# Euler Trail

**input:** graph  $G = (V, E)$

**output:** trail (as a stack with edges)

```
procedure euler-trail(vertex  $v$ );  
{  
  foreach vertex  $u$  in succ( $v$ ) do {  
    remove edge( $v, u$ ) from graph;  
    euler-trail( $u$ );  
    push(edge( $v, u$ ));  
  }  
}
```

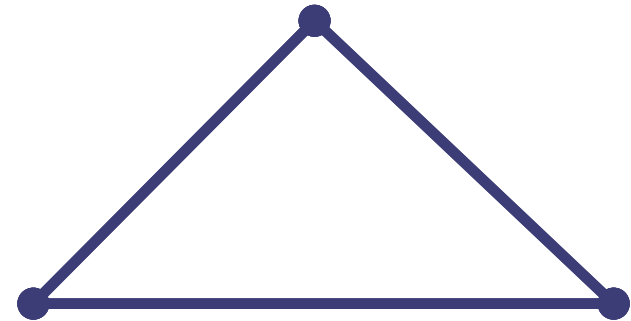


# Euler Trail

**input:** graph  $G = (V, E)$

**output:** trail (as a stack with edges)

```
procedure euler-trail(vertex  $v$ );  
{  
  foreach vertex  $u$  in succ( $v$ ) do {  
    remove edge( $v, u$ ) from graph;  
    euler-trail( $u$ );  
    push(edge( $v, u$ ));  
  }  
}
```



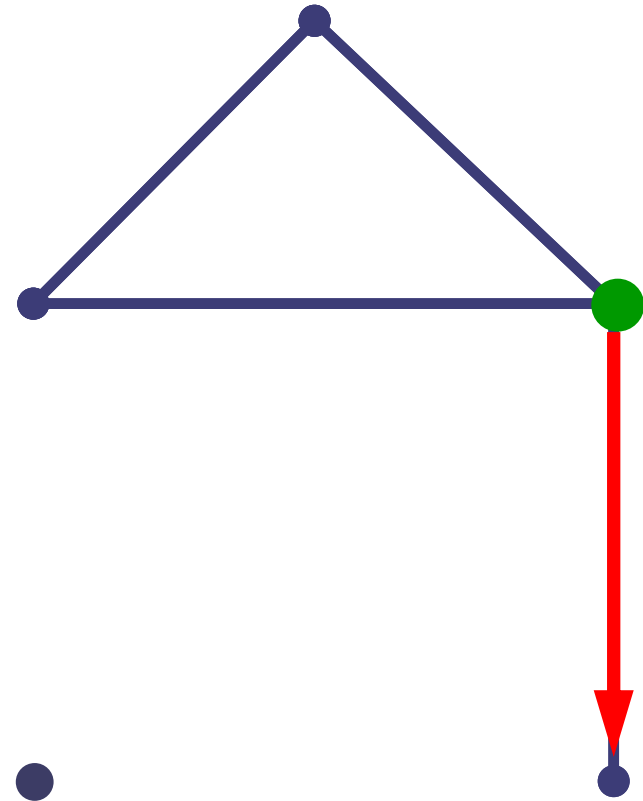


# Euler Trail

input: graph  $G = (V, E)$

output: trail (as a stack with edges)

```
procedure euler-trail(vertex  $v$ );  
{  
  foreach vertex  $u$  in succ( $v$ ) do {  
    remove edge( $v, u$ ) from graph;  
    euler-trail( $u$ );  
    push(edge( $v, u$ ));  
  }  
}
```

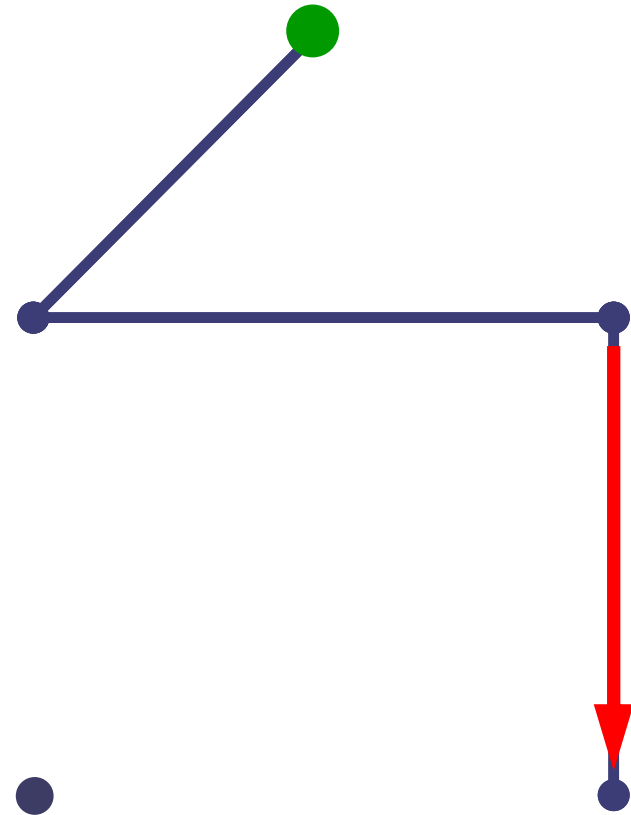


# Euler Trail

**input:** graph  $G = (V, E)$

**output:** trail (as a stack with edges)

```
procedure euler-trail(vertex  $v$ );  
{  
  foreach vertex  $u$  in succ( $v$ ) do {  
    remove edge( $v, u$ ) from graph;  
    euler-trail( $u$ );  
    push(edge( $v, u$ ));  
  }  
}
```

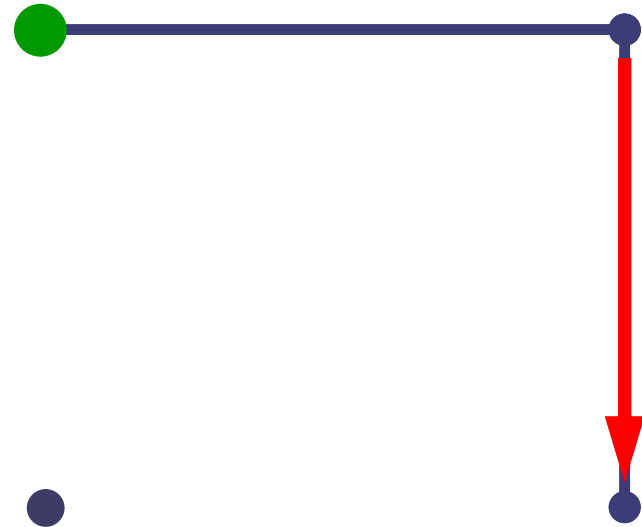


# Euler Trail

**input:** graph  $G = (V, E)$

**output:** trail (as a stack with edges)

```
procedure euler-trail(vertex  $v$ );  
{  
  foreach vertex  $u$  in succ( $v$ ) do {  
    remove edge( $v, u$ ) from graph;  
    euler-trail( $u$ );  
    push(edge( $v, u$ ));  
  }  
}
```

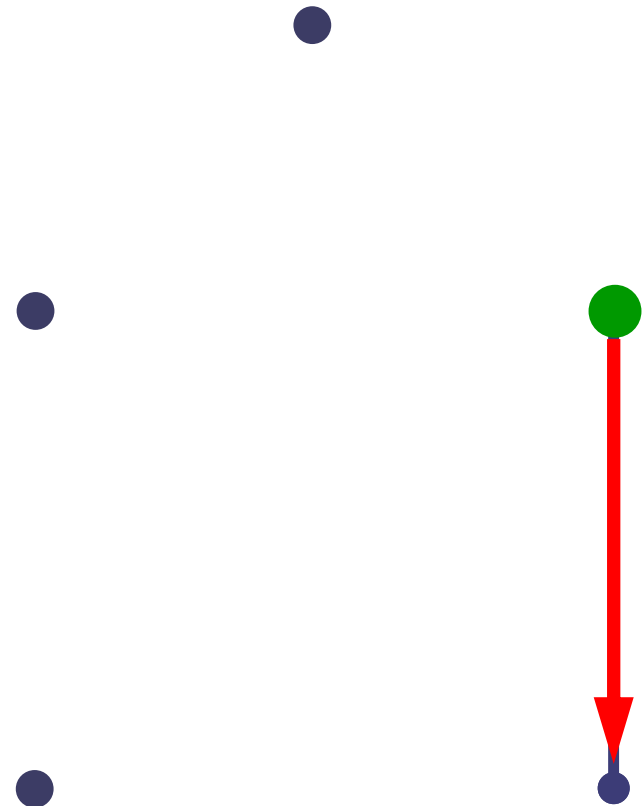


# Euler Trail

input: graph  $G = (V, E)$

output: trail (as a stack with edges)

```
procedure euler-trail(vertex  $v$ );  
{  
  foreach vertex  $u$  in succ( $v$ ) do {  
    remove edge( $v, u$ ) from graph;  
    euler-trail( $u$ );  
    push(edge( $v, u$ ));  
  }  
}
```

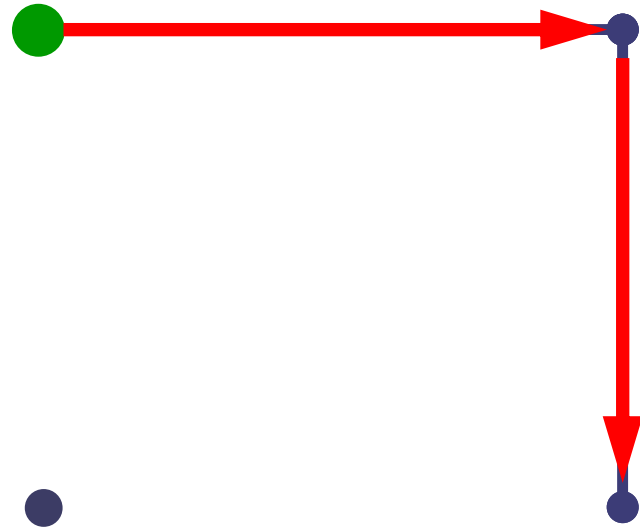


# Euler Trail

input: graph  $G = (V, E)$

output: trail (as a stack with edges)

```
procedure euler-trail(vertex  $v$ );  
{  
  foreach vertex  $u$  in succ( $v$ ) do {  
    remove edge( $v, u$ ) from graph;  
    euler-trail( $u$ );  
    push(edge( $v, u$ ));  
  }  
}
```

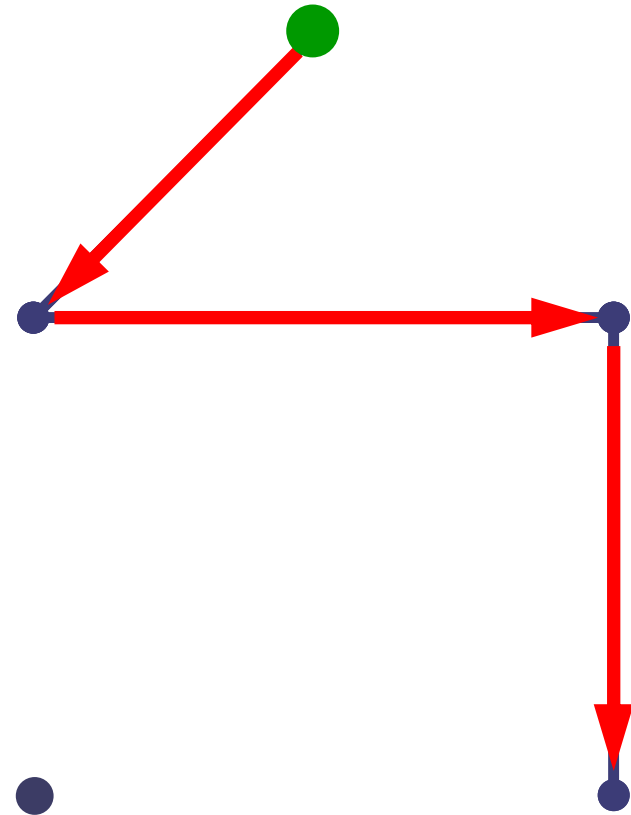


# Euler Trail

**input:** graph  $G = (V, E)$

**output:** trail (as a stack with edges)

```
procedure euler-trail(vertex  $v$ );  
{  
  foreach vertex  $u$  in succ( $v$ ) do {  
    remove edge( $v, u$ ) from graph;  
    euler-trail( $u$ );  
    push(edge( $v, u$ ));  
  }  
}
```

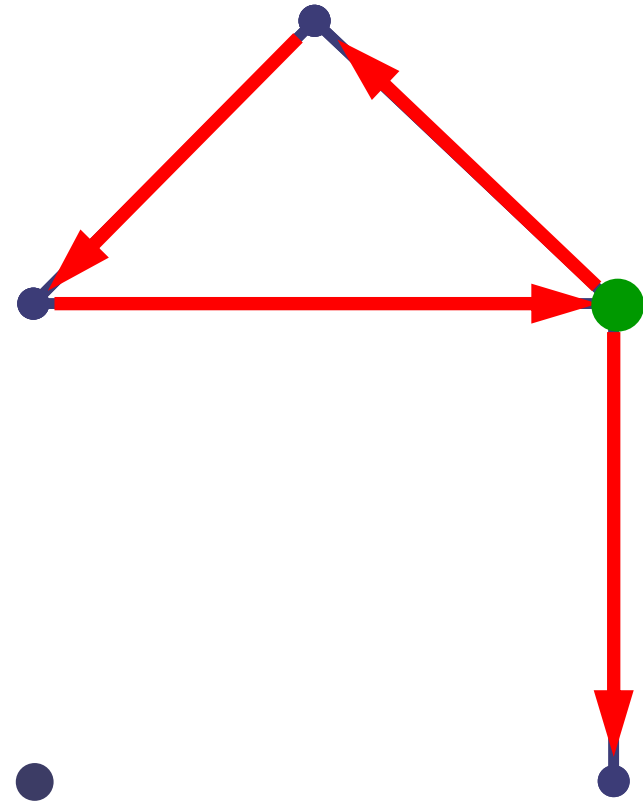


# Euler Trail

input: graph  $G = (V, E)$

output: trail (as a stack with edges)

```
procedure euler-trail(vertex  $v$ );  
{  
  foreach vertex  $u$  in succ( $v$ ) do {  
    remove edge( $v, u$ ) from graph;  
    euler-trail( $u$ );  
    push(edge( $v, u$ ));  
  }  
}
```

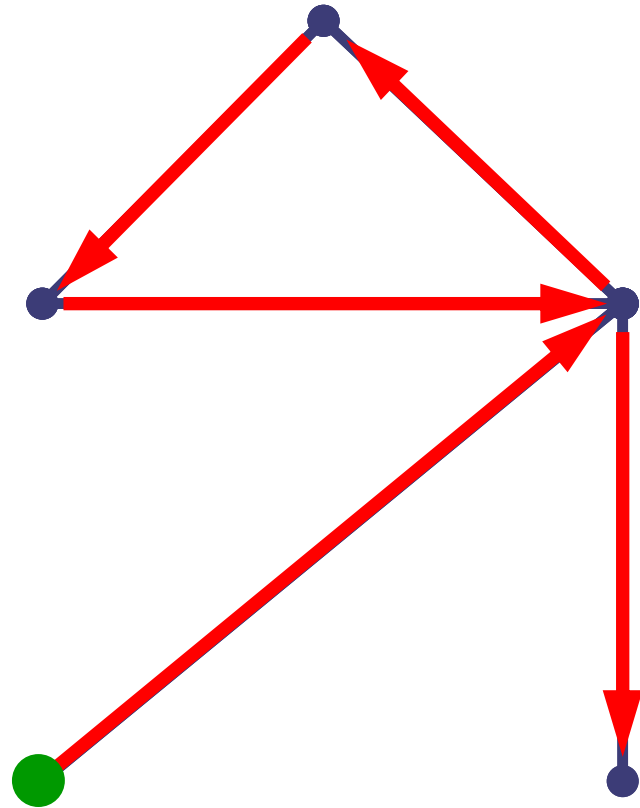


# Euler Trail

input: graph  $G = (V, E)$

output: trail (as a stack with edges)

```
procedure euler-trail(vertex  $v$ );  
{  
  foreach vertex  $u$  in succ( $v$ ) do {  
    remove edge( $v, u$ ) from graph;  
    euler-trail( $u$ );  
    push(edge( $v, u$ ));  
  }  
}
```



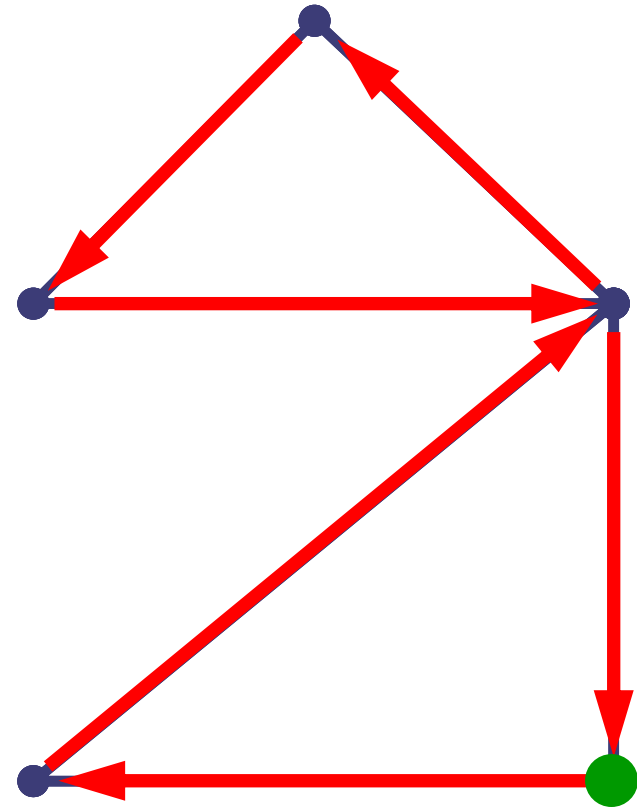


# Euler Trail

input: graph  $G = (V, E)$

output: trail (as a stack with edges)

```
procedure euler-trail(vertex  $v$ );  
{  
  foreach vertex  $u$  in succ( $v$ ) do {  
    remove edge( $v, u$ ) from graph;  
    euler-trail( $u$ );  
    push(edge( $v, u$ ));  
  }  
}
```

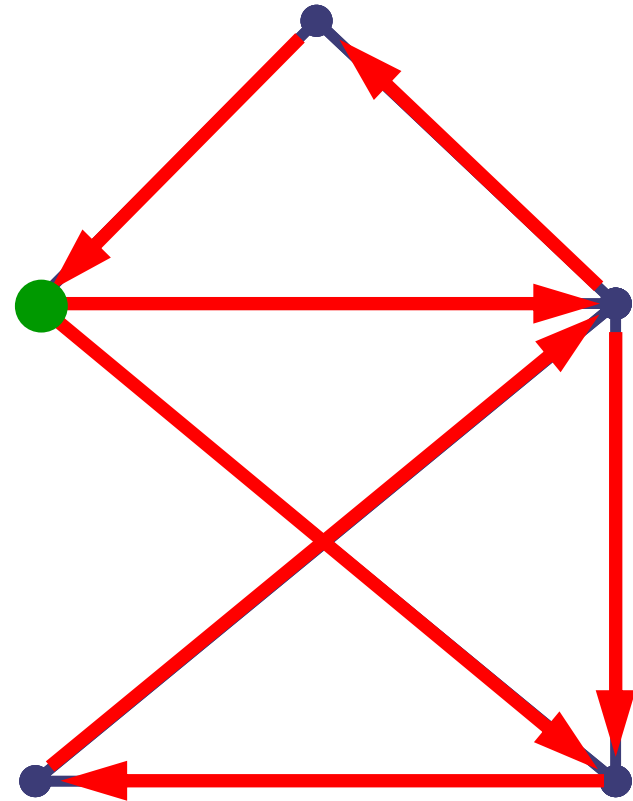


# Euler Trail

input: graph  $G = (V, E)$

output: trail (as a stack with edges)

```
procedure euler-trail(vertex  $v$ );  
{  
  foreach vertex  $u$  in succ( $v$ ) do {  
    remove edge( $v, u$ ) from graph;  
    euler-trail( $u$ );  
    push(edge( $v, u$ ));  
  }  
}
```

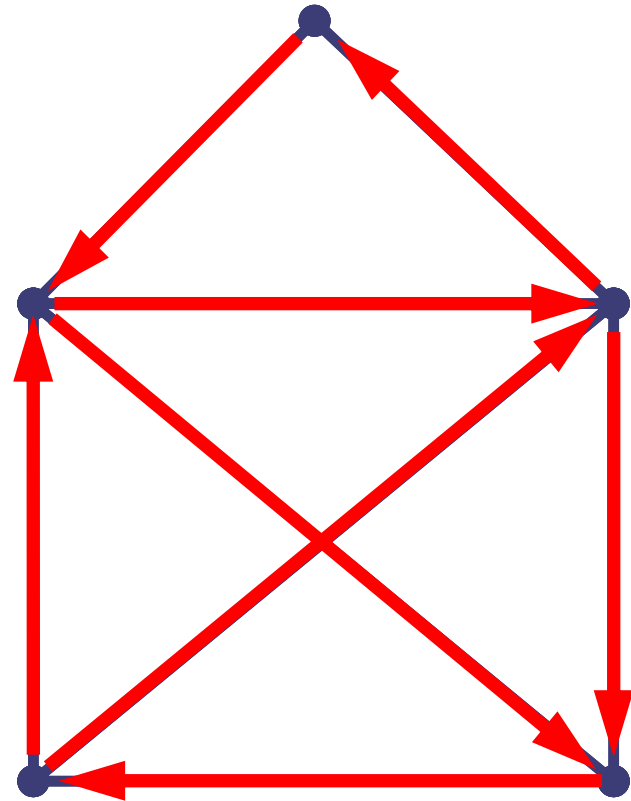


# Euler Trail

**input:** graph  $G = (V, E)$

**output:** trail (as a stack with edges)

```
procedure euler-trail(vertex  $v$ );  
{  
  foreach vertex  $u$  in succ( $v$ ) do {  
    remove edge( $v, u$ ) from graph;  
    euler-trail( $u$ );  
    push(edge( $v, u$ ));  
  }  
}
```



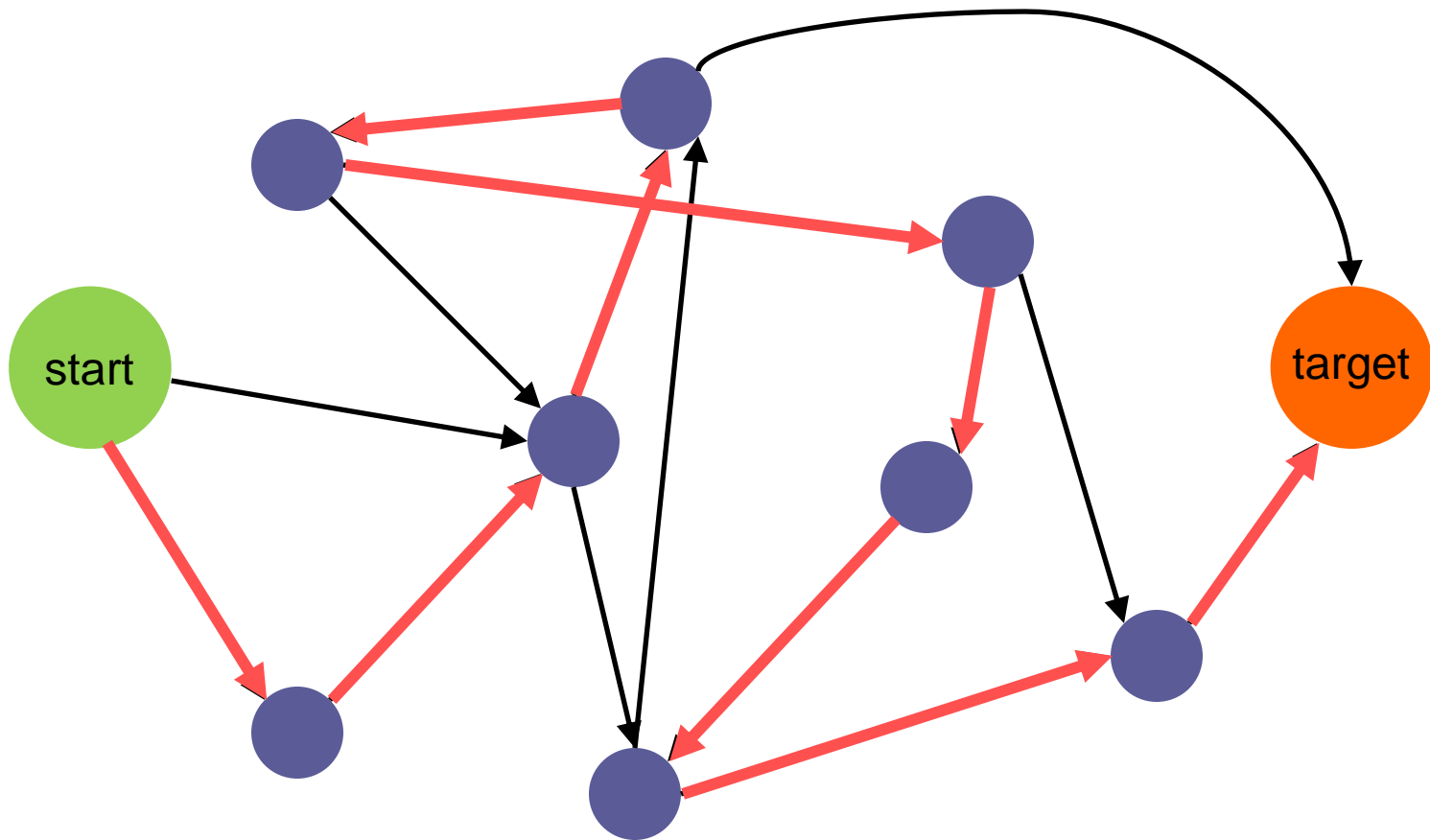
## ■ Complexity:

- The Euler trail algorithm performs only one complete traversal of the graph.
- If the graph is represented as an *adjacency list* then the algorithm runs in  $\Theta(|V|+|E|)$  time (linear time).
- If the graph is represented as an *adjacency matrix* then the algorithm runs in  $\mathbf{O}(|V|^2)$  time.

# Hamiltonian Path

## ■ Hamiltonian Path Problem:

Does a (directed or undirected) graph  $G$  contain a path that visits every node exactly once?

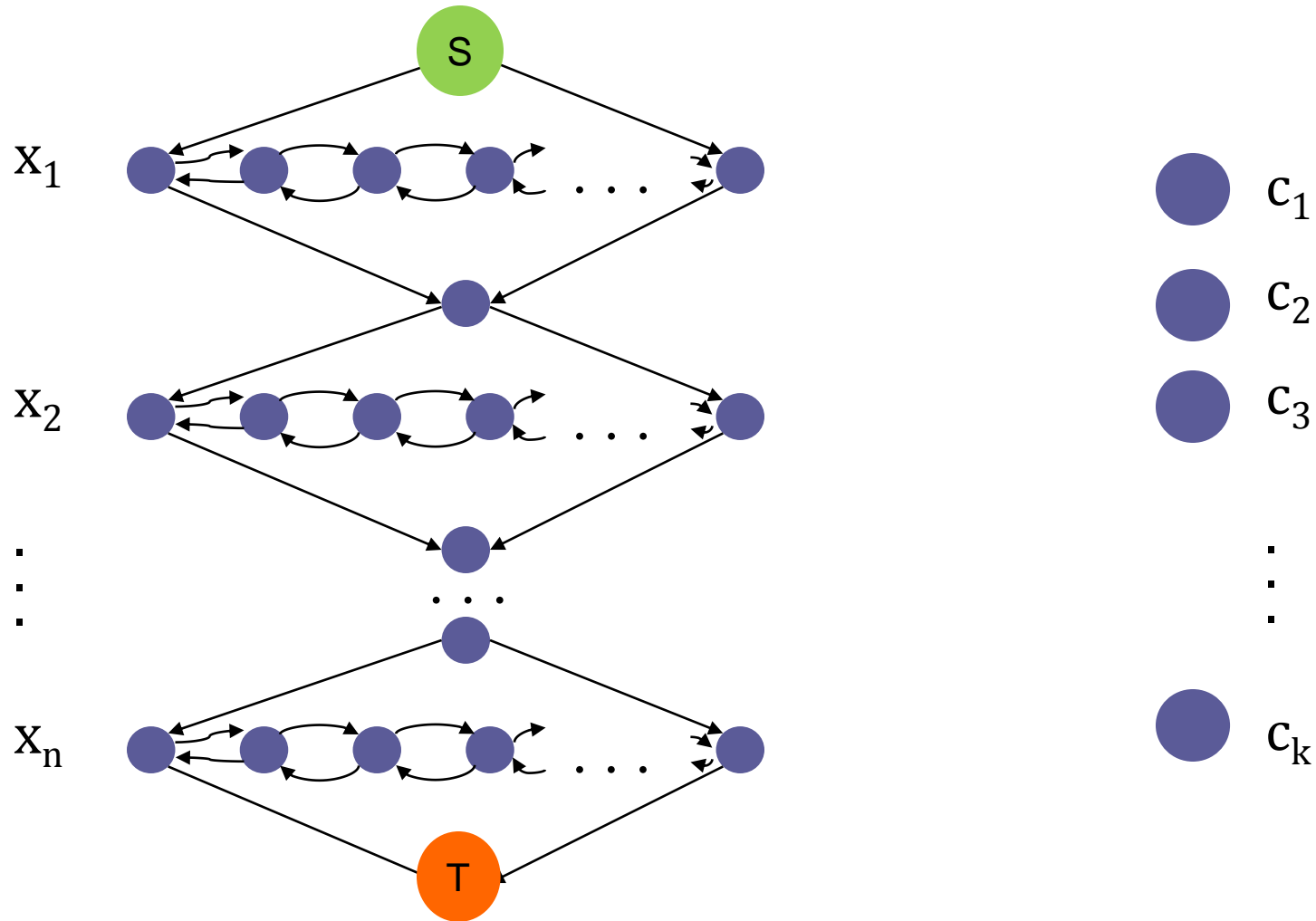


# Hamiltonian Path

- Why is the Hamiltonian Path problem so hard (NPC)?
- Reduction Idea:
  - Suppose we have a black box to solve Hamiltonian Path.
  - We already know that SAT is hard – NP-Complete (Cook 1971).
  - If we can do a polynomial time transformation of an arbitrary input SAT instance to some instance for our black box in such a way, that our black box solution will directly represent SAT solution for the input, then If we solve our black box in polynomial time then we can solve even SAT in polynomial time.

# Hamiltonian Path

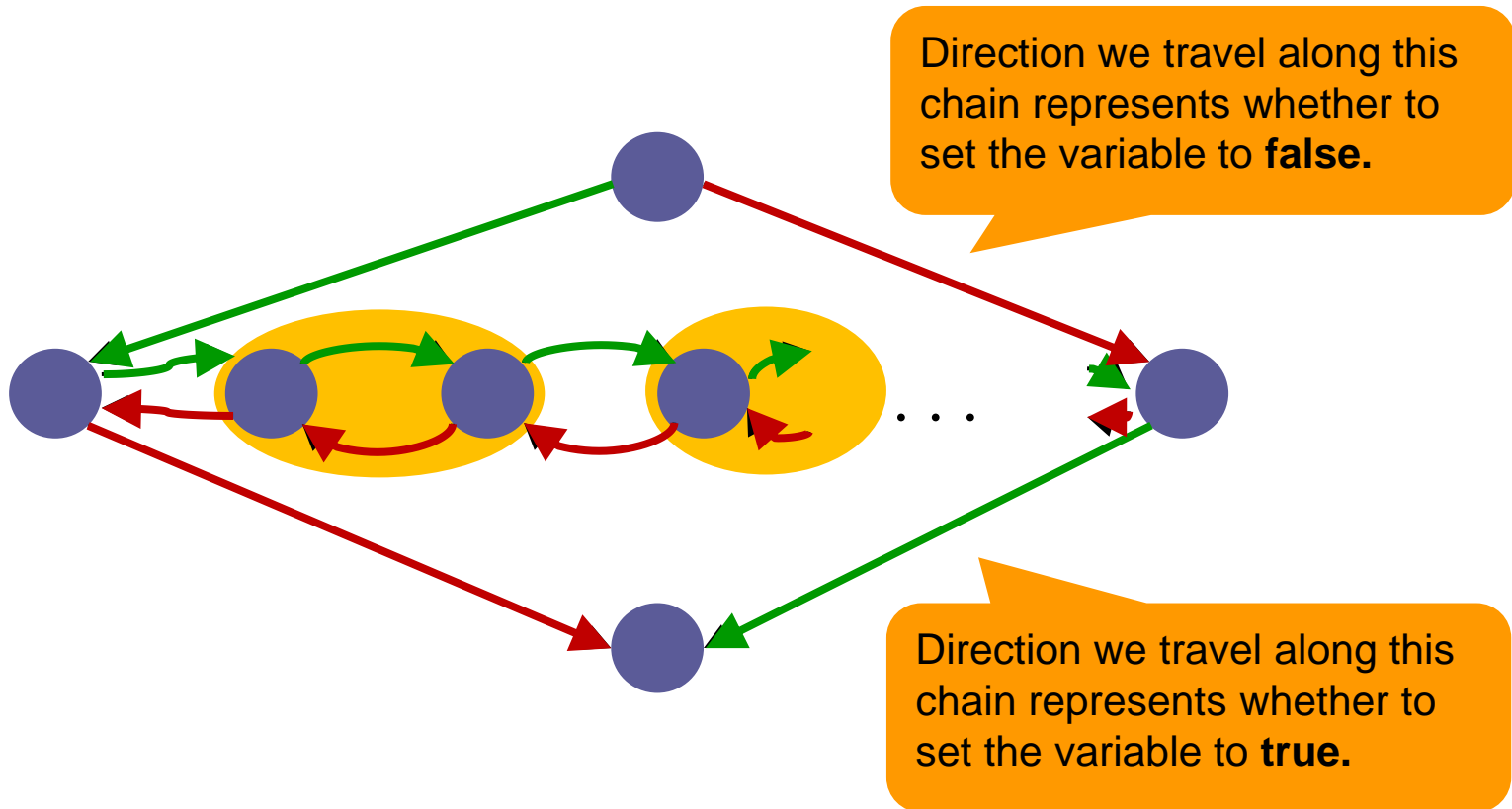
- High level structure:



# Hamiltonian Path

- Internal structure of variable  $x_i$ :
  - A number of occurrences of variable  $x_i$  in the whole SAT exactly corresponds to the number of pairs in yellow ovals.

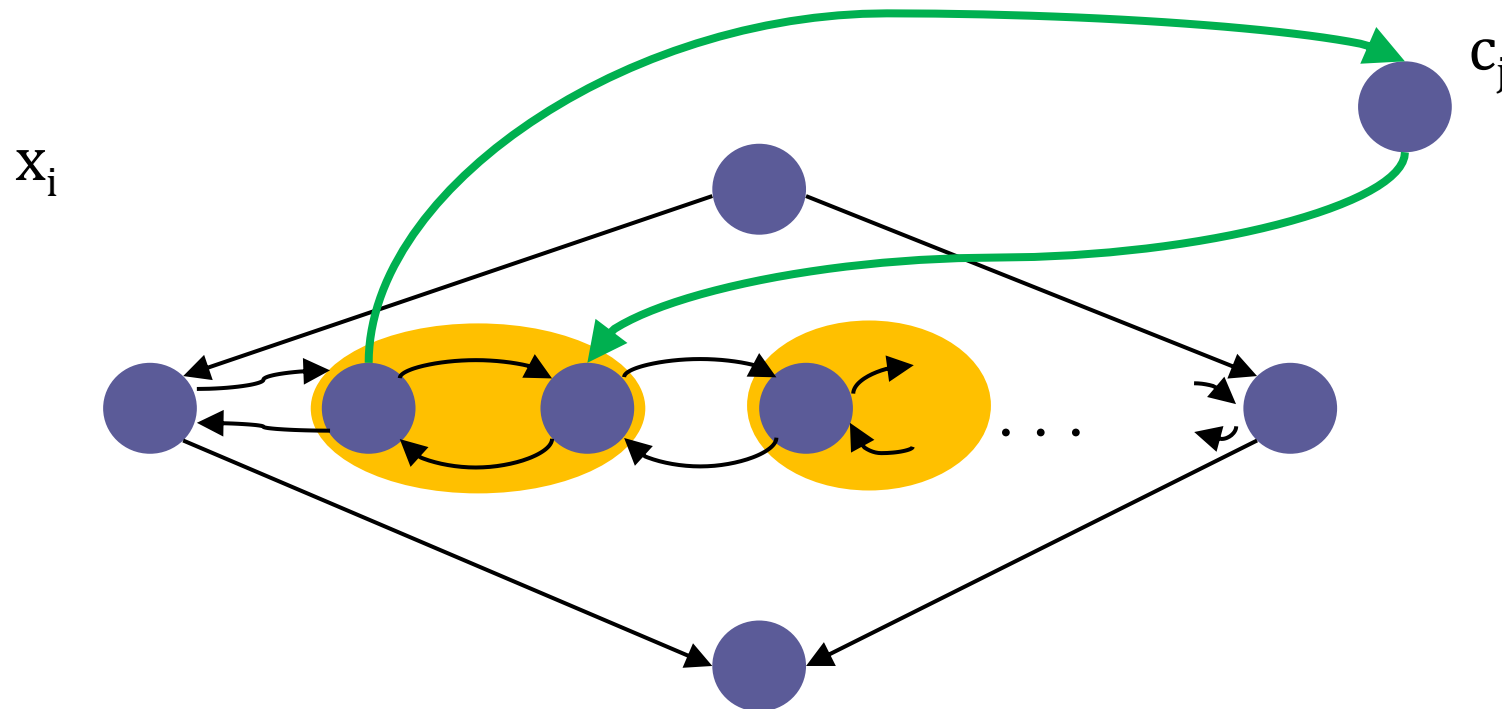
$x_i$





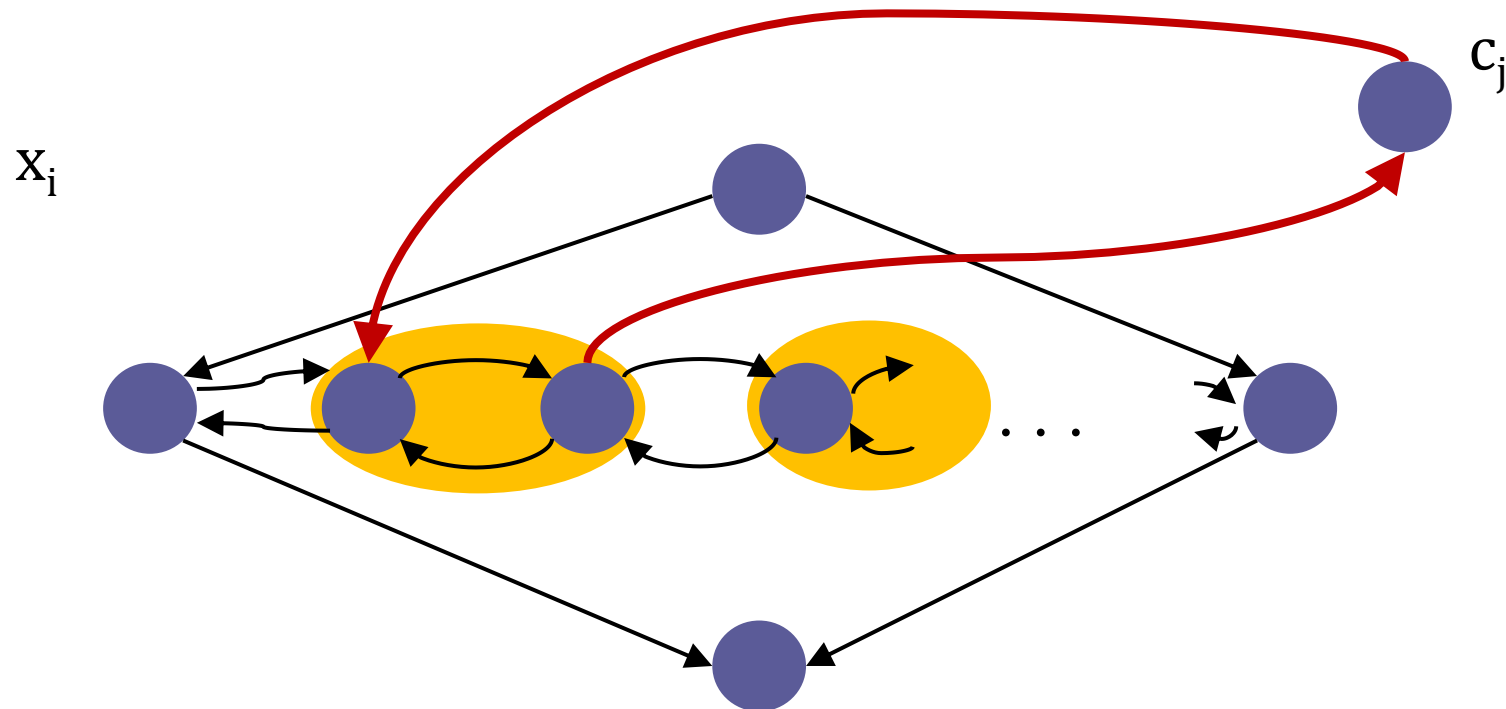
# Hamiltonian Path

- Internal structure of variable  $x_i$ :
  - If the clause  $c_j$  contains the **positive** literal:  $x_i$



# Hamiltonian Path

- Internal structure of variable  $x_i$ :
  - If the clause  $c_j$  contains the **negative** literal:  $\neg x_i$



# References

- Matoušek, J.; Nešetřil, J. *Kapitoly z diskretní matematiky*. Karolinum. Praha 2002. ISBN 978-80-246-1411-3.
- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). *Introduction to Algorithms (2nd ed.)*. MIT Press and McGraw-Hill. ISBN 0-262-53196-8.
- Tarjan, R. E. (1972). *Depth-first search and linear graph algorithms*, SIAM Journal on Computing 1 (2): 146–160, doi:10.1137/0201010