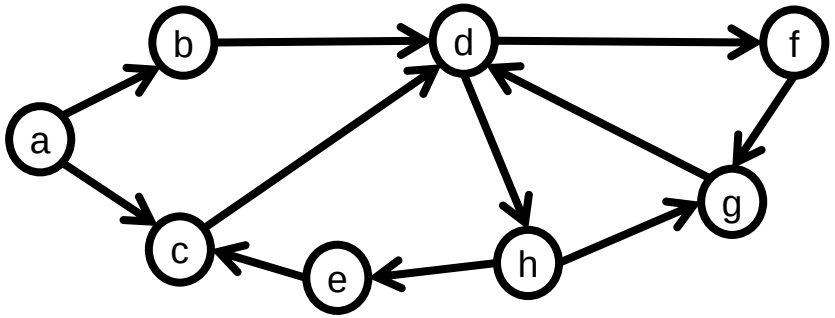
DFS applications: Strongly Connected Components

Lecture 04.04

By Marina Barsky

DFS on Directed Graph



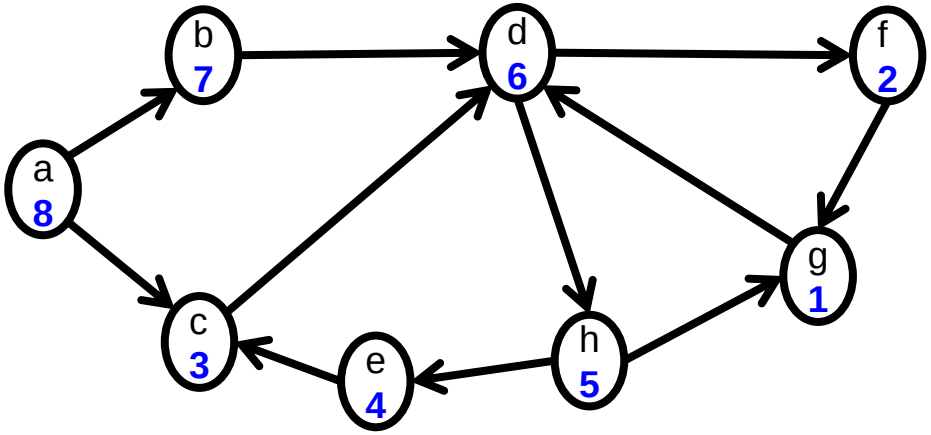
Running time O(n + m)

```
Algorithm DFS(digraph G, current)
    current.state:= "discovered"
    for each u in out_arcs(current)
        if u.state = "undiscovered" then
            DFS(G, u)
    current.state:="processed"
```

```
for each u in vertices of G
    u.state:= "undiscovered"
DFS(digraph G, start)  // start is a vertex in G
```

By the end we discover all the nodes in digraph G that are reachable from the source node *s*

Recap: finishing time



- The node is marked as processed only when none of its outgoing arcs lead to an undiscovered vertex
- The DFS will continue exploring undiscovered vertices until none left
- Finishing time ***f(v)*** of node v is defined to be the number of nodes that were marked as processed before v

To traverse all nodes in G: DFS loop

```
global clock: = 1
```

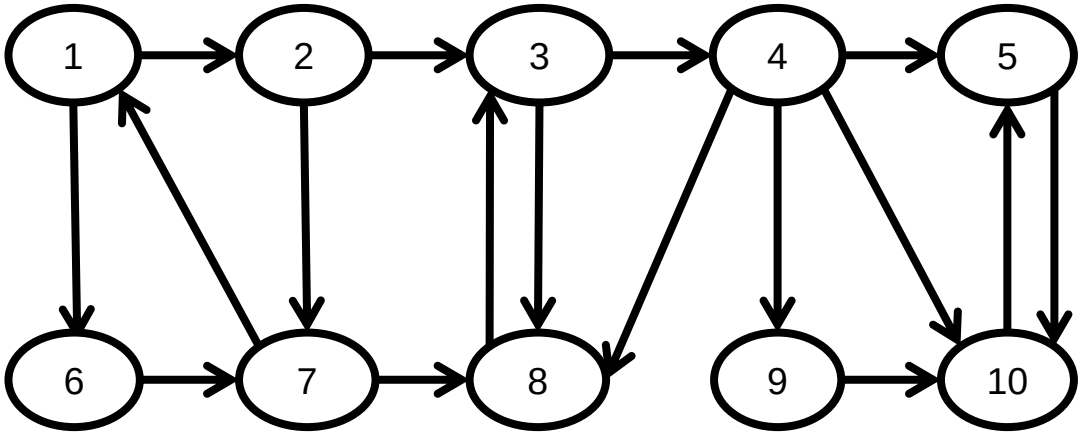
```
Algorithm DFS(DAG G, current)

    current.state:= "discovered"
    for each u in out_arcs(current)
        if u.state = "undiscovered" then
            DFS(G, u)
    current.state:="processed"
    current.f: = clock
    clock: = clock + 1
```

```
Algorithm DFS_loop(DAG G)

    mark all nodes of G as "undiscovered"
    for each u in vertices of G
        if u.state = "undiscovered"
            DFS(DAG G, u)
```

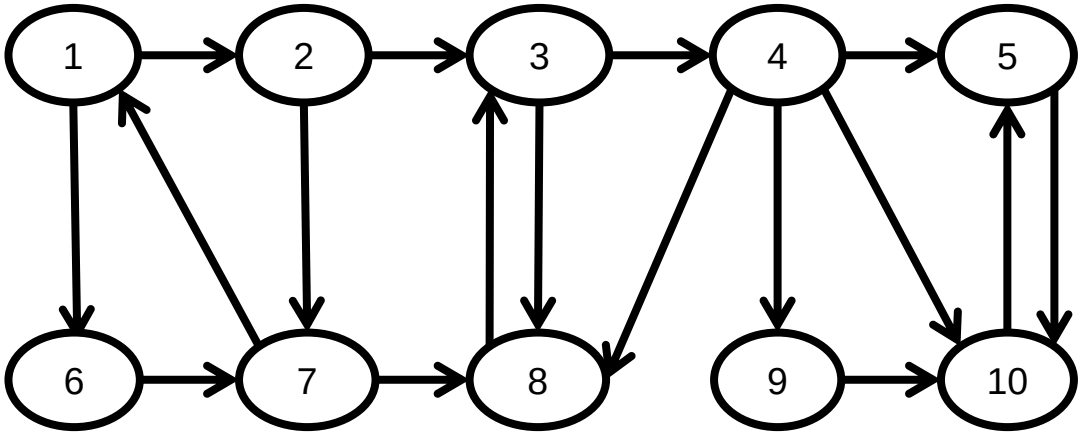
Is this graph connected?



- Yes, in a sense that it cannot be broken into completely isolated components: for any vertex pair v, u there is path v~>u **OR** path u~>v

- This is called **weak connectivity**

Strongly connected directed graphs



A directed graph G is ***strongly-connected*** if for any two vertices u and v there is a path u~>v, AND there is also a path v~u.

It means that in a strongly connected graphs communication flows through the network in both directions: there is a way to deliver information from any v to any u, and vice versa

Testing graph for strong connectivity

We can do it using two runs of DFS:

DFS1 (G, v)

- Pick any node v - arbitrarily
- Perform DFS from v: every node which is reachable from v will be discovered during this DFS, and marked as processed
- If there are no unprocessed nodes by the end of DFS1, then there is a one-way path from any node to any other node in G

Testing graph for strong connectivity

We can do it using two runs of DFS:

DFS1 (G, v)

- Pick any node v - arbitrarily
- Perform DFS from v: every node which is reachable from v will be discovered during this DFS, and marked as processed
- If there are no unprocessed nodes by the end of DFS1, then there is a one-way path from any node to any other node in G

DFS2 (G^T, v)

- Now we need to check if there is also a path from any vertex reachable from v to any other vertex in the graph in the opposite direction
- To do so we perform DFS on a ***transpose*** **of digraph G - graph G^T, which consists of the same vertices, but where each edge is reversed**
- During DFS on G^T we discover a path connecting every pair of vertices in the opposite direction

Testing graph for strong connectivity

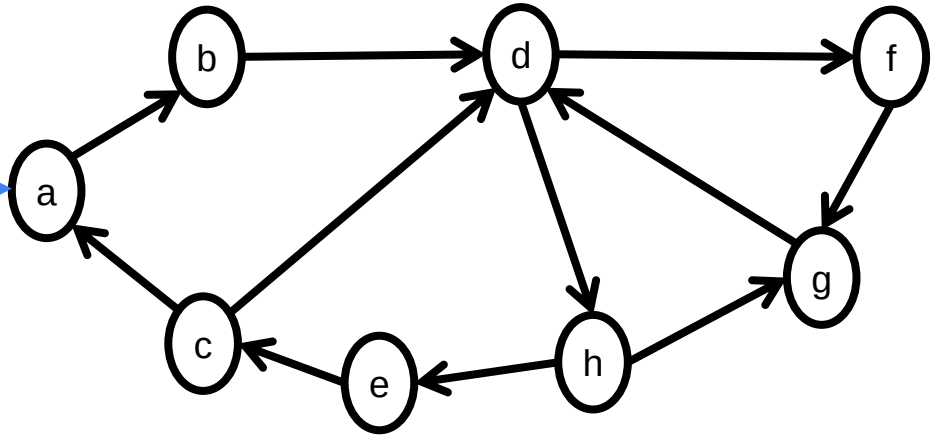
We can do it using two runs of DFS:
DFS1 (G, v)

DFS2 (G^T, v)

- If there is a path v ~> u AND the path u ~> v in graph G, then DFS1 will discover the first, and DFS2 (G^T) will discover the second

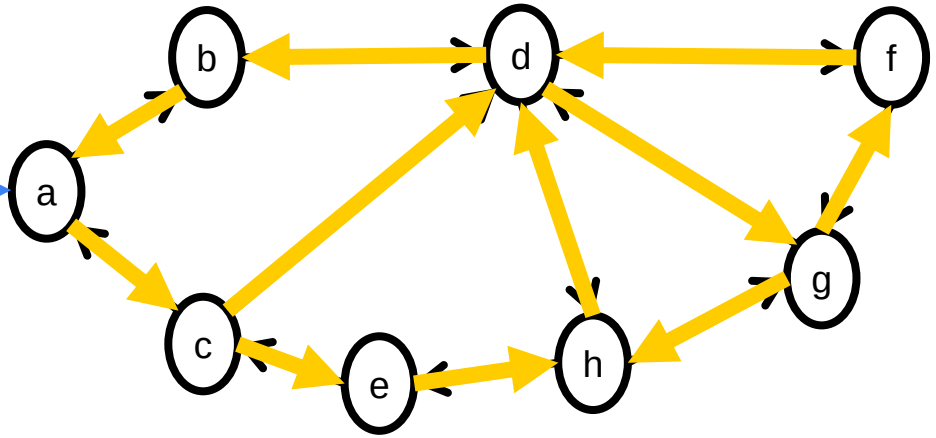
- Thus, if both DFS(G) and DFS(G^T) processed all vertices of the graph G, then graph G is strongly connected

Example



Graph G

- All the nodes are processed with DFS (G, a)

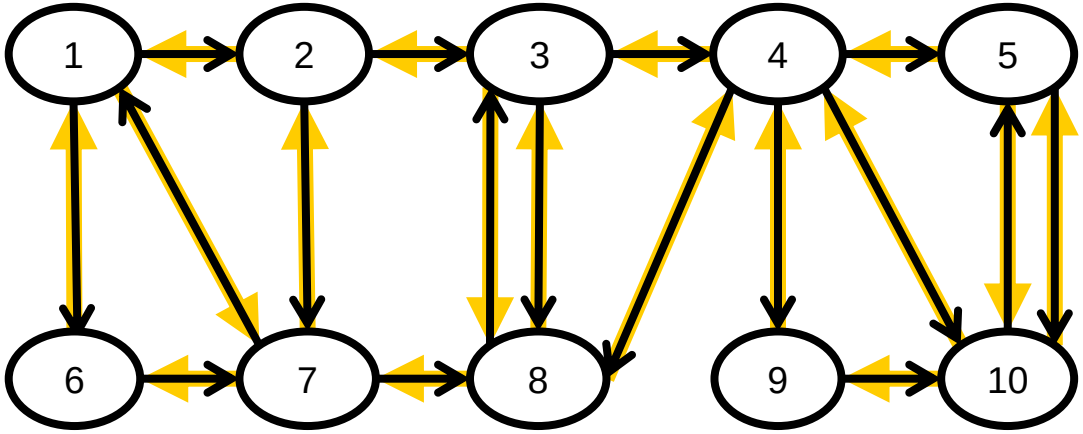


Graph G^T

- All the nodes are processed with DFS (G^T, a)

Conclusion: Graph G is strongly-connected

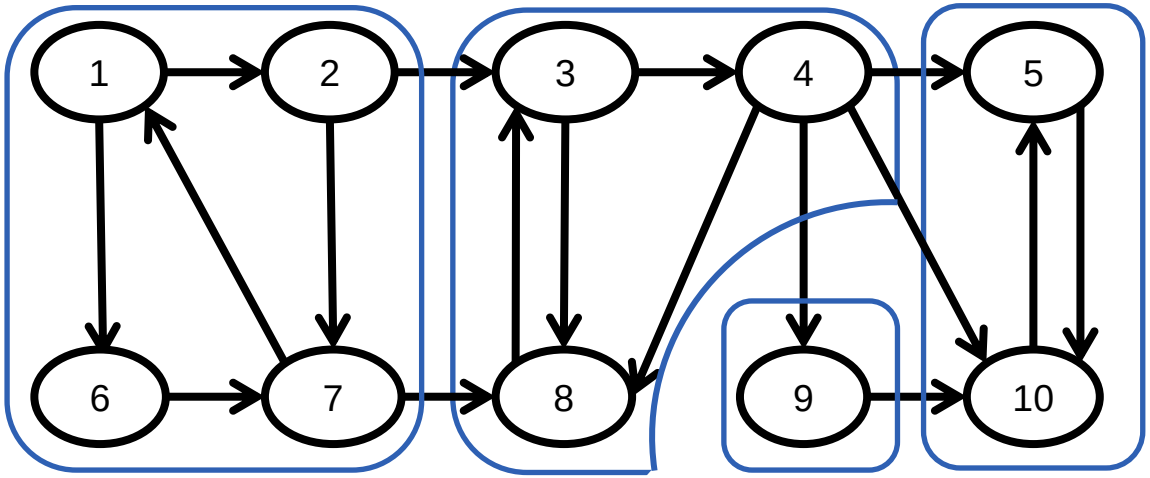
Is this graph strongly-connected?



- If we do DFS(G, 1) we reach all the nodes
- DFS in G^T tells us if there is a return path in G from any node to node 1
- If we do DFS(G^T, 1) we will only reach nodes 2, 7, 6

Conclusion: this graph is not strongly-connected

Strongly-Connected Components

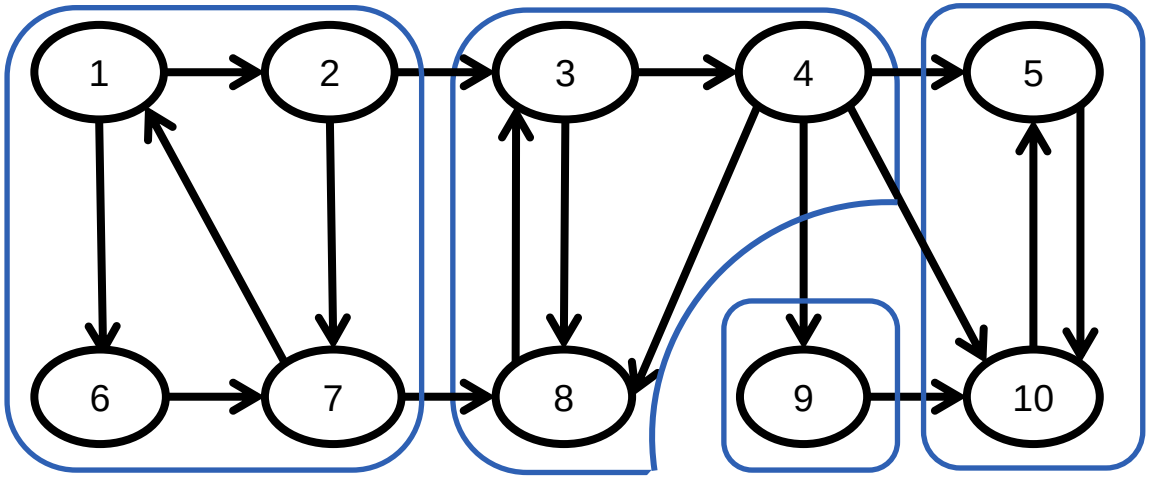


- The entire graph is not strongly-connected
- Is there any sub-graph that is strongly-connected?

- Given digraph G = (V, E), we define ***a strongly connected component (SCC)*** of G to be a maximal subset C of vertices V, such that for all u, v in C, both u ~> v and v ~> u; that is, both u and v are reachable from each other. In other words, two vertices of directed graph are in the same SCC if and only if they are reachable from each other.

- There are 4 strongly-connected components in this graph

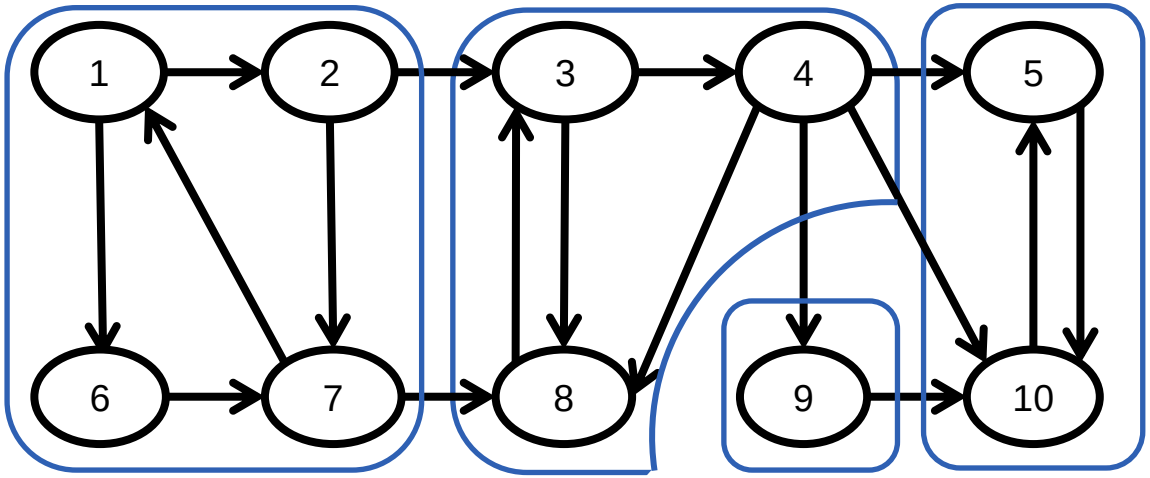
Discovering Strongly Connected Components



- The problem of finding connected components is at the heart of many graph application. Generally speaking, the connected components of the graph correspond to different classes of objects.

  For example, in social networks these are groups of people who can communicate with each other freely and have a limited connectivity to other groups

Can we use DFS loop?



- We might think of applying the DFS loop to test for reachability from any vertex of G
- However this time we will get different results depending which node do we use as a start
  - For example if we start DFS at vertex 5, we will indeed discover the SCC {5,10} and nothing else
  - If we start with vertex 8, we will discover {3,4,5,10,9} – which is not SCC
  - Moreover, if we start with vertex 1, then we will reach all the vertices of G and will not discover any SCCs

Two-pass algorithm for discovering SCCs (Kosaraju and Sharir, 1981)

- General idea:

Step 1

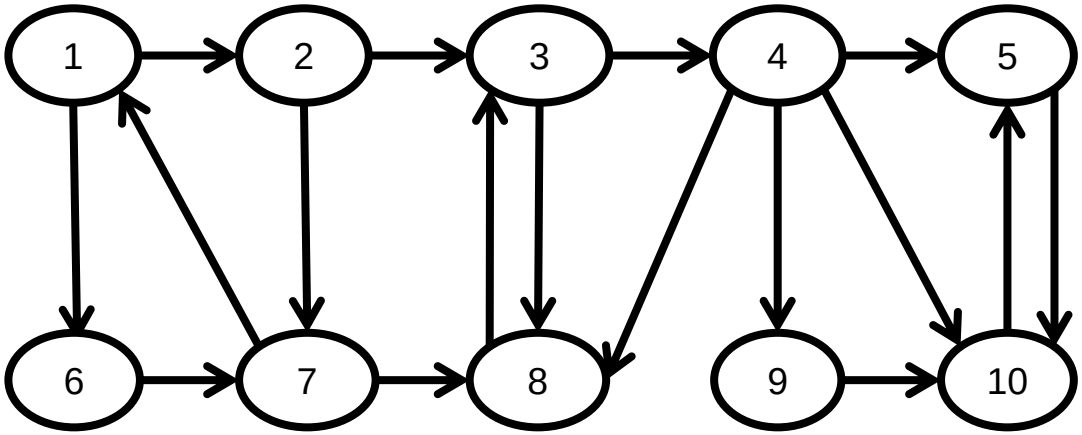
- Run DFS-loop on G^T and compute finishing time for each vertex

    Use this finishing time as a “magic” number to guide the order of the second DFS loop

Step 2

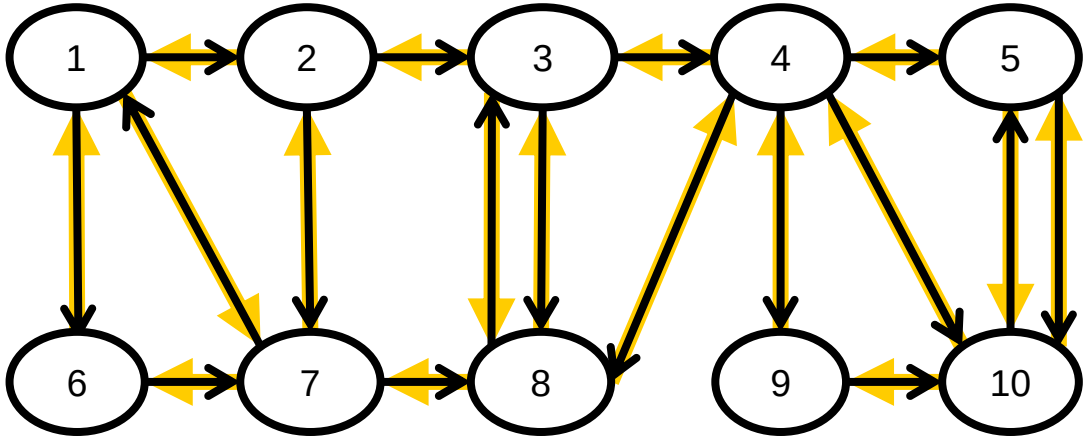
- Run DFS-loop on G, processing nodes in reverse order of their finishing times
- Each time we collected all nodes reachable from v, we discovered a new SCC

Step-by-step example of 2P-SCC algorithm



G

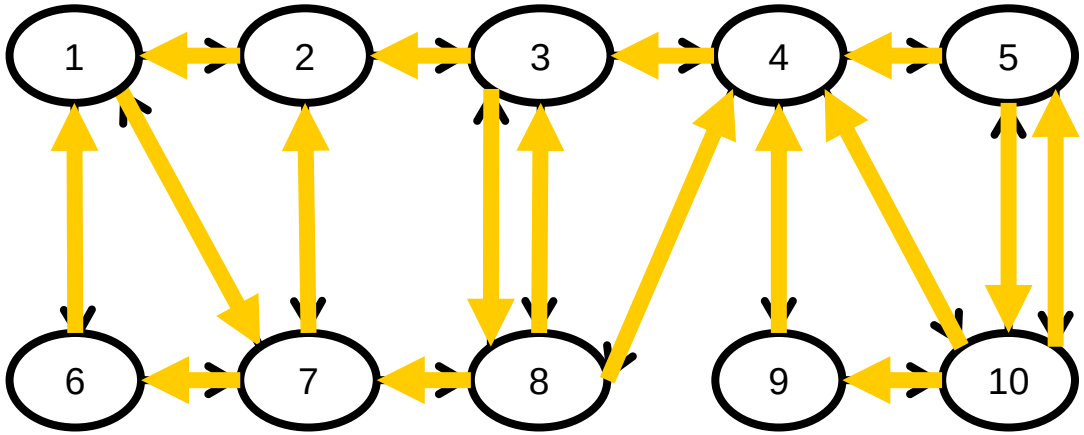
First, generate G^T



G^T

- Reverse every edge of G

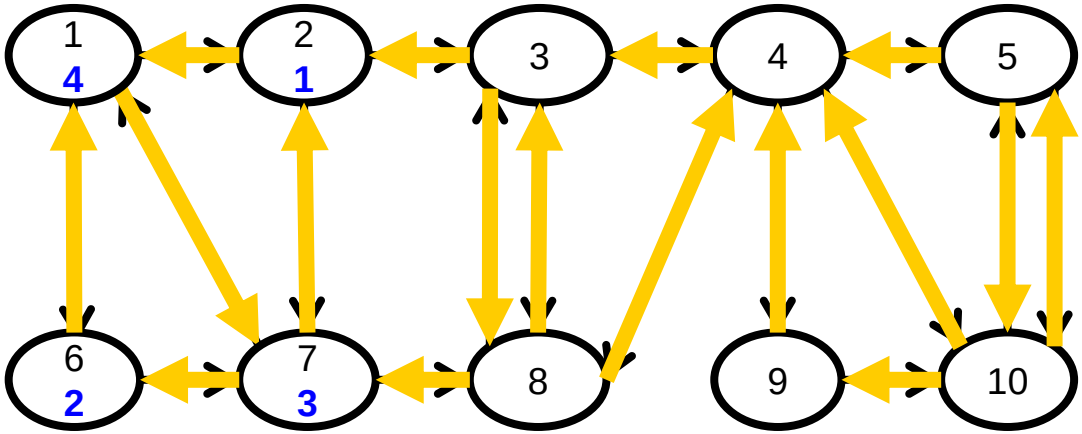
Perform DFS loop 1 on G^T



G^T

- Run DFS-loop on G^T
- Compute finishing time for each vertex

Computing finishing times in G^T

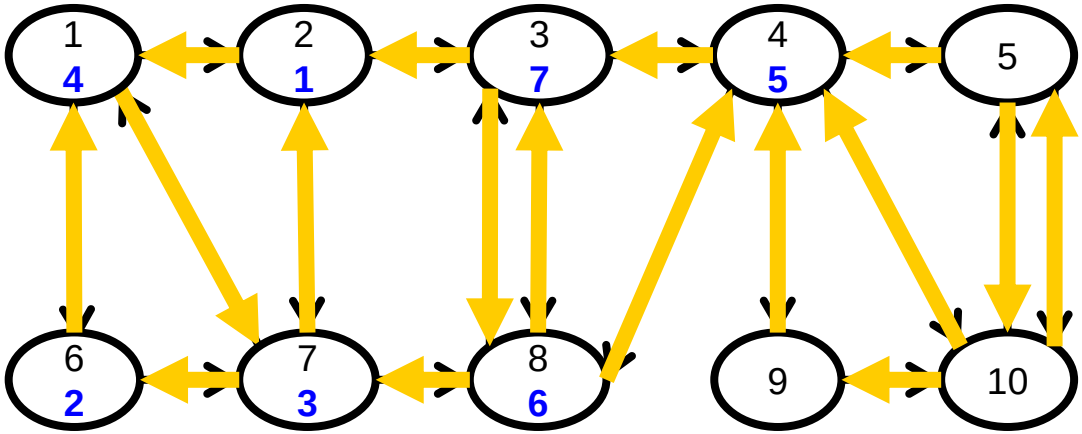


G^T

i	F[i]
1	node 2
2	node 6
3	node 7
4	node 1
5	
6	
7	
8	
9	
10	

Finishing times after calling DFS1 from node 1

Computing finishing times in G^T

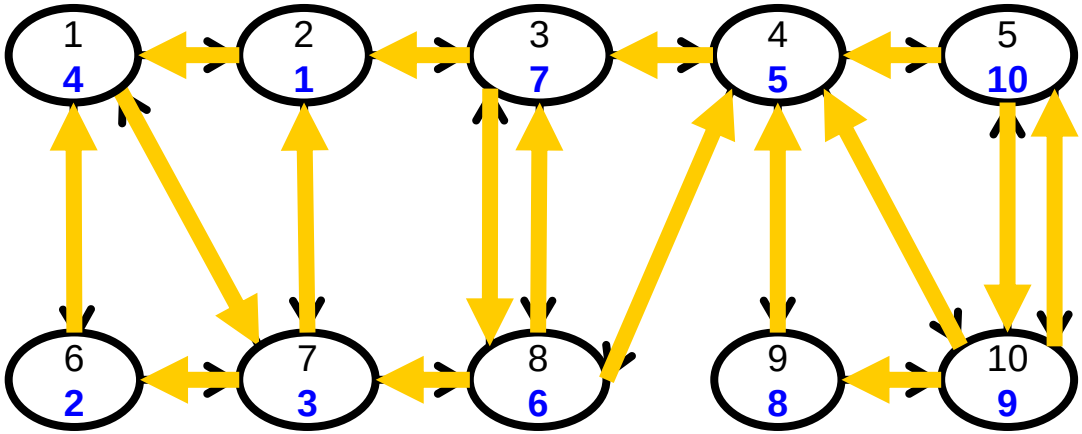


G^T

i	F[i]
1	node 2
2	node 6
3	node 7
4	node 1
5	node 4
6	node 8
7	node 3
8	
9	
10	

Finishing times after calling DFS1 from node 3

Computing finishing times in G^T

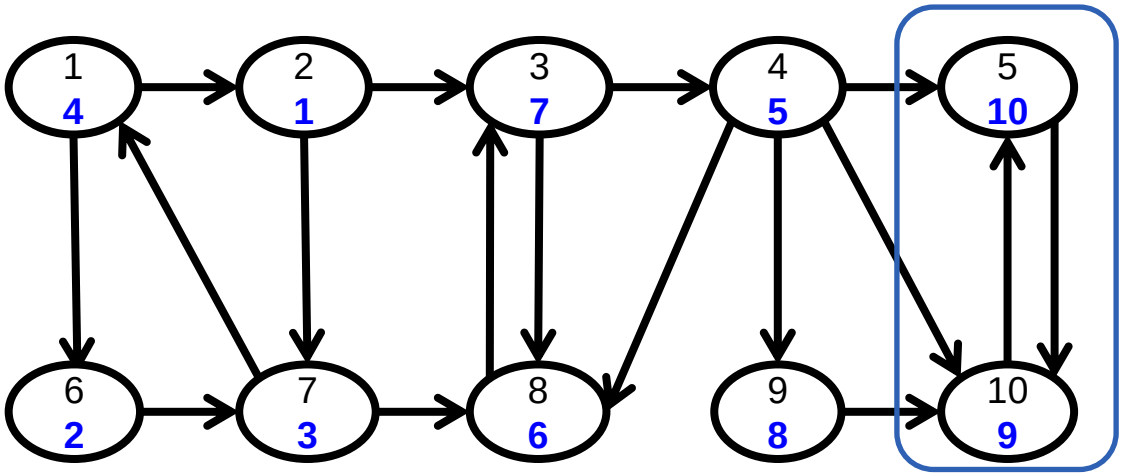


G^T

i	F[i]
1	node 2
2	node 6
3	node 7
4	node 1
5	node 4
6	node 8
7	node 3
8	node 9
9	node 10
10	node 5

Finishing times after calling DFS1 from node 5

DFS loop 2
Traversing G in reverse order of finishing times



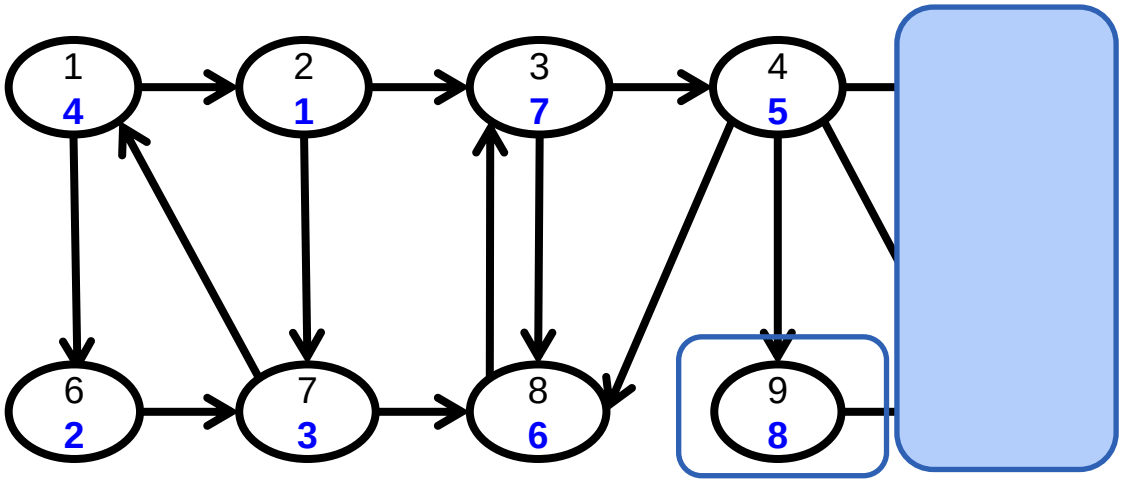
G^T

i	F[i]
1	node 2
2	node 6
3	node 7
4	node 1
5	node 4
6	node 8
7	node 3
8	node 9
9	node 10
10	node 5

Leader: node 5

SCC1: {5,10}

Example: DFS loop 2
Traversing G in reverse order of finishing times



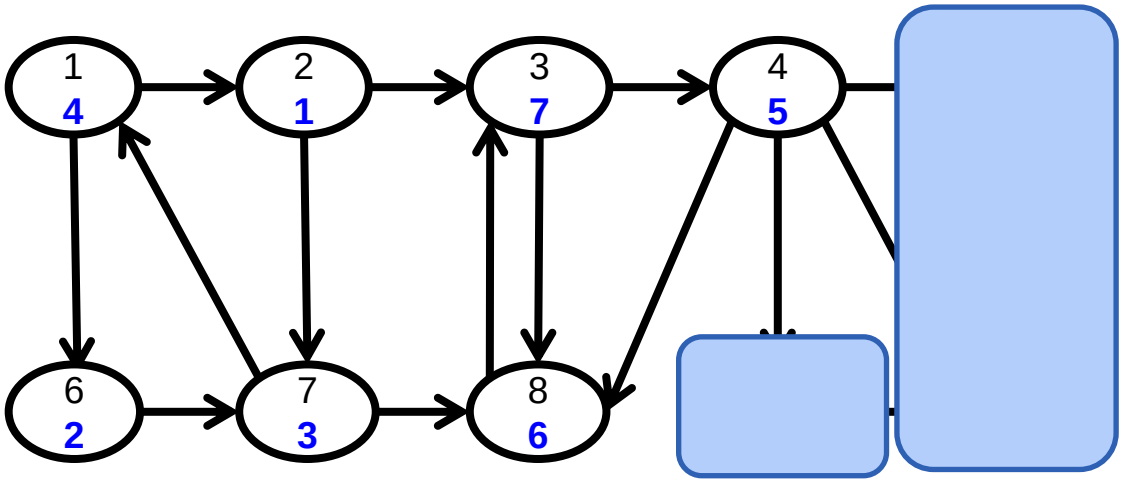
G^T

i	F[i]
1	node 2
2	node 6
3	node 7
4	node 1
5	node 4
6	node 8
7	node 3
8	node 9
9	node 10
10	node 5

Leader: node 9

SCC2: {9}

Example: DFS loop 2
Traversing G in reverse order of finishing times



G^T

i	F[i]
1	node 2
2	node 6
3	node 7
4	node 1
5	node 4
6	node 8
7	node 3
8	node 9
9	node 10
10	node 5

Leader: node 3

SCC3: {3, 4, 8} etc.

2P-SCC: pseudocode

```
Algorithm SCC(digraph G)

    call DFS_loop1(G^T)      # this will compute array F of finishing times
    call DFS_loop2(G, F)     # this will discover SCCs in G
```

First DFS loop performed on G^T

```
global clock: = 1

# nodes indexed by finishing time
global F: = array of size n
```

Algorithm ***DFS1***(G, current)

```
    current.state:= "discovered"
    for each u in out_arcs(current)
        if u.state = "undiscovered" then
            DFS1(G, u)
    current.state:="processed"
    F[clock]: = current
    clock: = clock + 1
```

Algorithm DFS_loop1(G)

```
    mark all nodes of G as "undiscovered"
    for each u in vertices of G
        if u.state = "undiscovered"
            DFS1(G^T, u)
```

Second loop performed on G

```
# the leading node with which we started SCC
global leader: = null

# nodes indexed by finishing time
global F: = array of size n
```

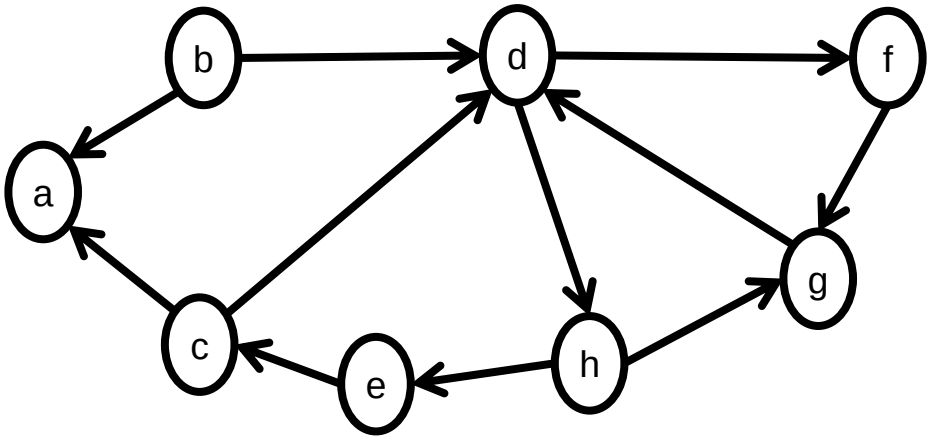
```
Algorithm DFS2(G, current)

    current.state:= "discovered"
    for each u in out_arcs(current)
        if u.state = "undiscovered" then
            DFS2(G, u)
    current.state:="processed"
    current.leader: = leader
```

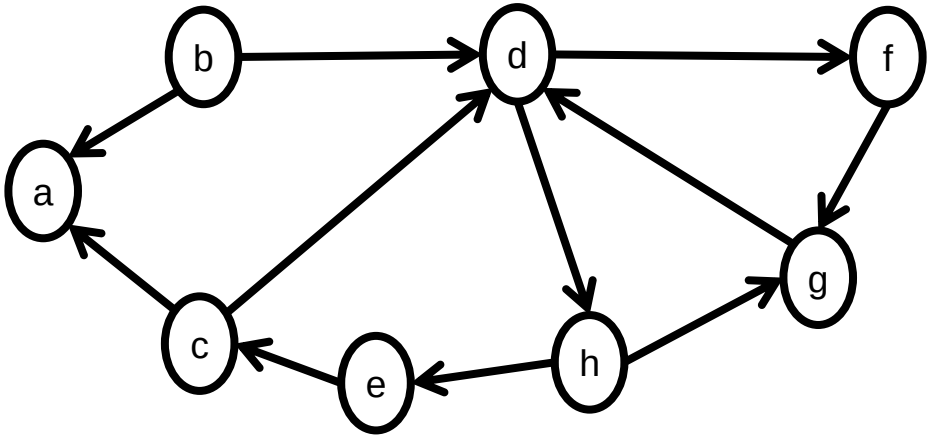
```
Algorithm DFS_loop2(G, F)

    mark all nodes of G as "undiscovered"
    for i from n downto 1:
        u = F[i]
        if u.state = "undiscovered"
            leader: = u
            DFS2(G, u)
```

Try it out: Activity 9

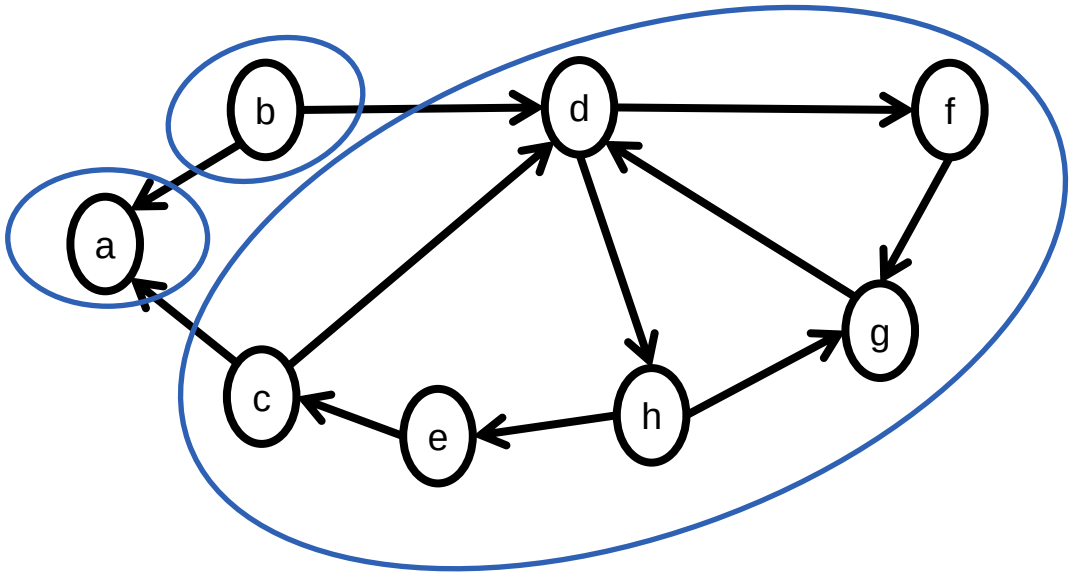


Finishing times in G^T after DFS loop 1



i	F[i]
1	node b
2	node f
3	node g
4	node d
5	node h
6	node e
7	node c
8	node a

DFS loop 2



i	F[i]
1	node b
2	node f
3	node g
4	node d
5	node h
6	node e
7	node c
8	node a

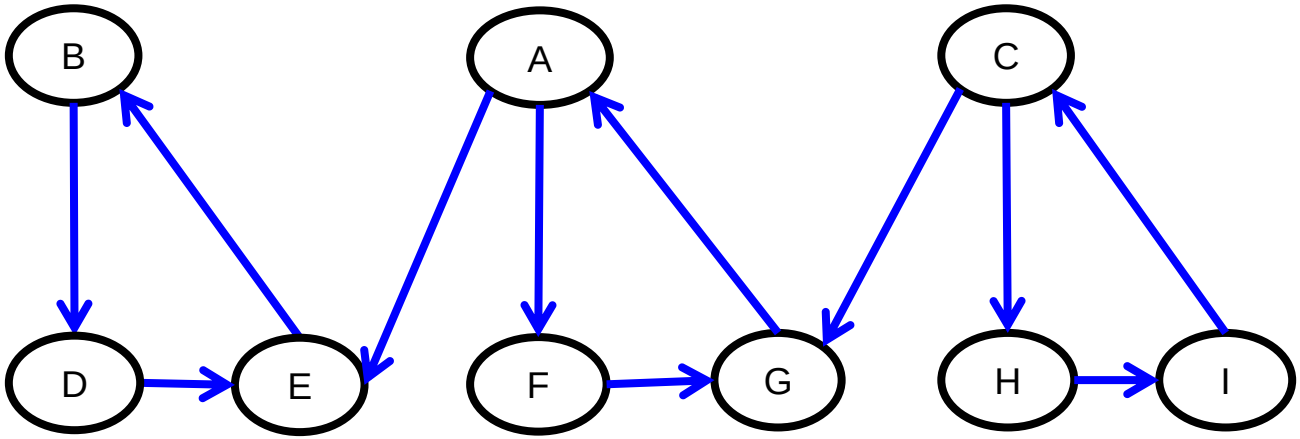
Leader: node a
SCC1: {a}

Leader: node c
SCC2: {c, d, f, g, h, e}

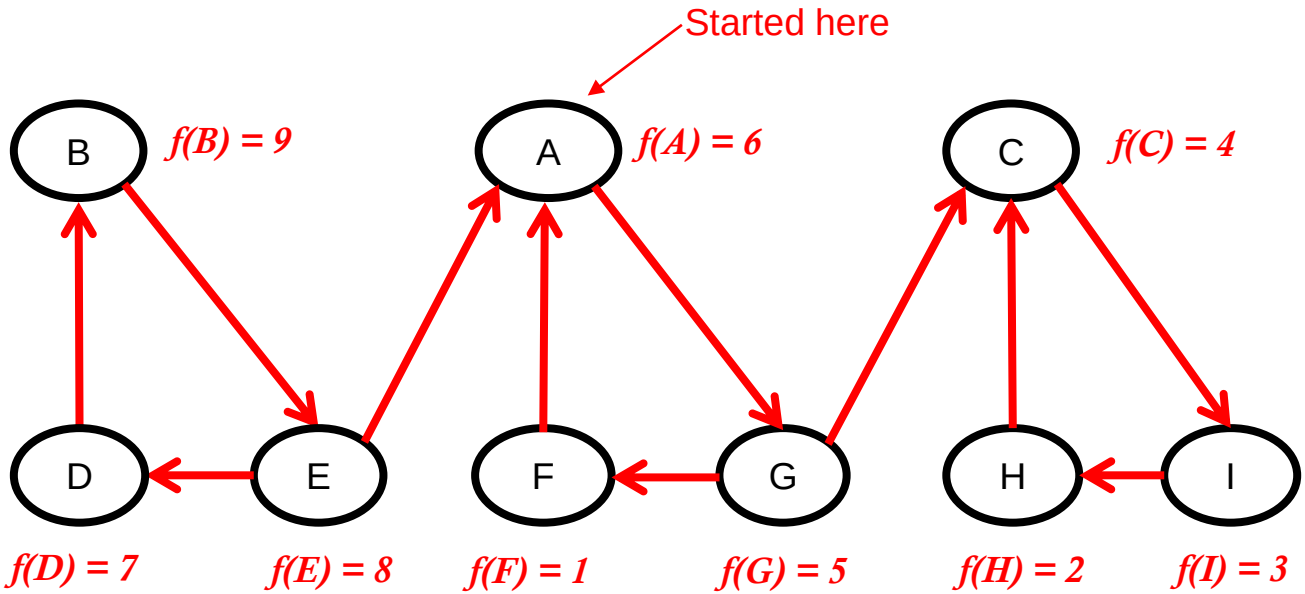
Leader: node b
SCC3: {b}

Board example

G

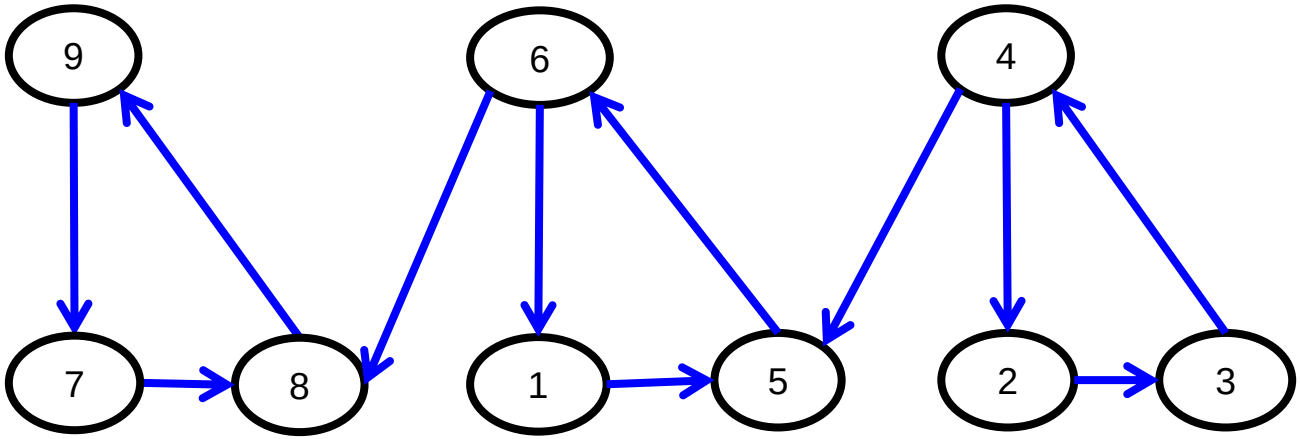


G^T

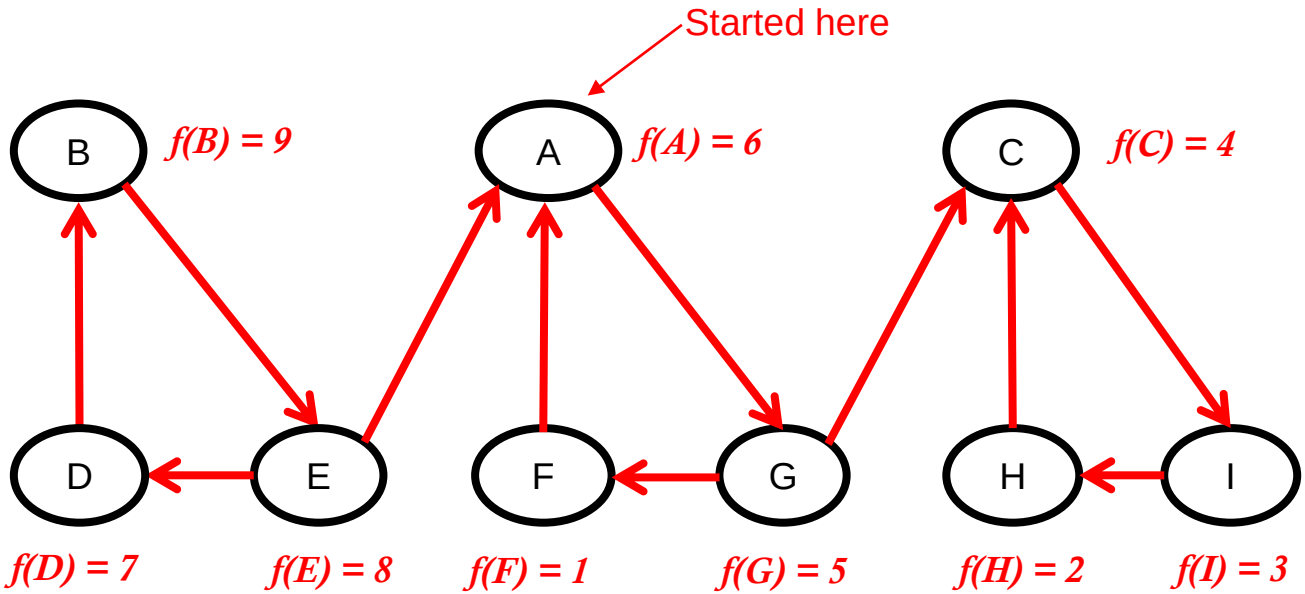


Board example

G

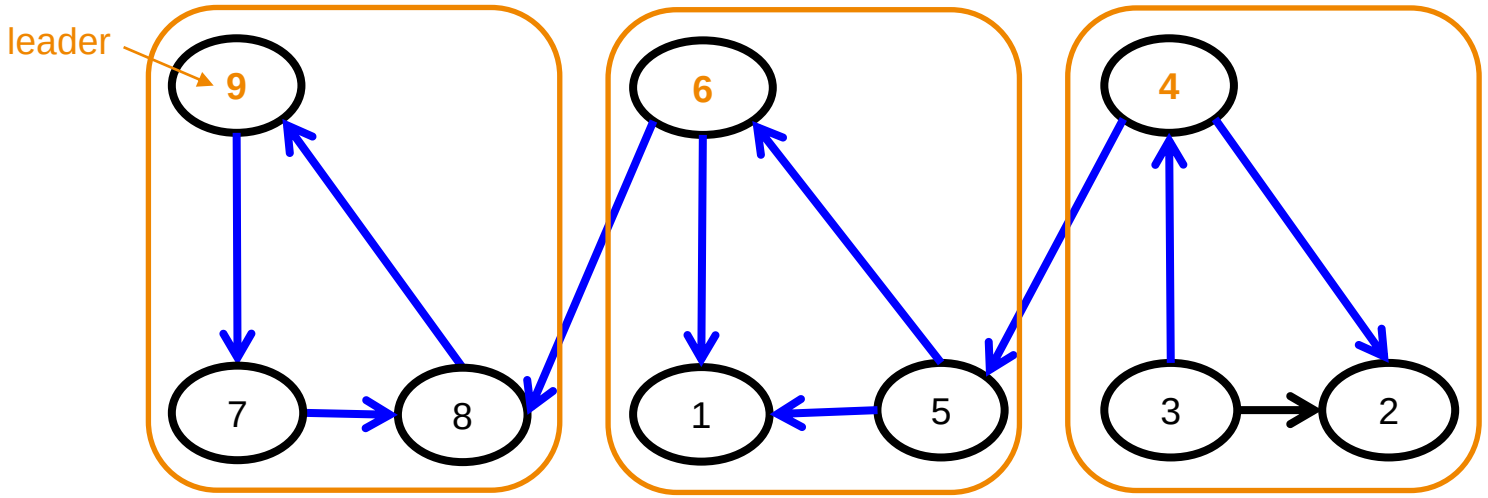


G^T



Board example: resulting SCCs

G



We run several examples of 2P-SCC and obtained the desired SCCs

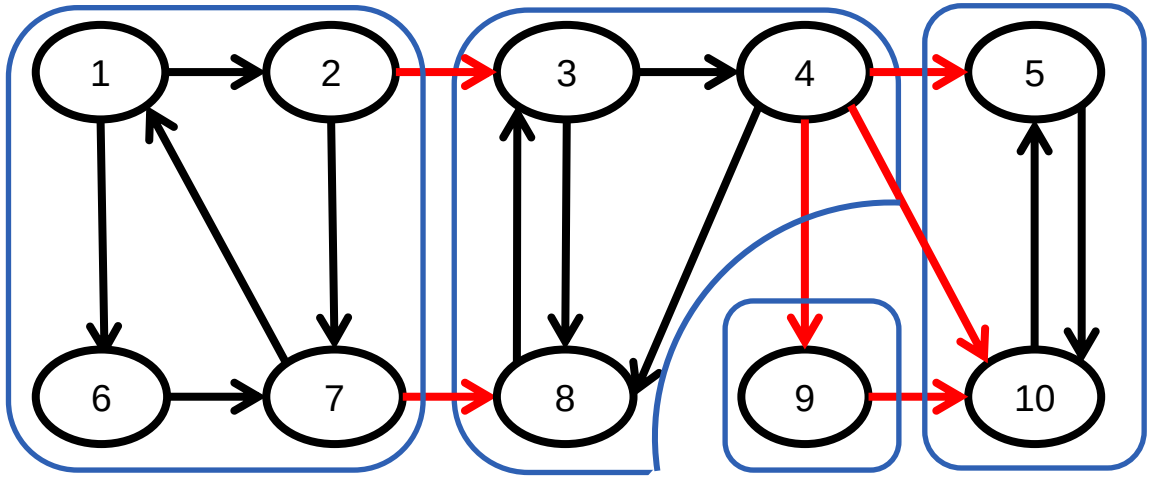
Now we want to **make sure** that a simple DFS loop 2 discovers all SCC, if it processes nodes according to the “magic” ordering discovered by the DFS loop 1

Would it work for any input graph G?

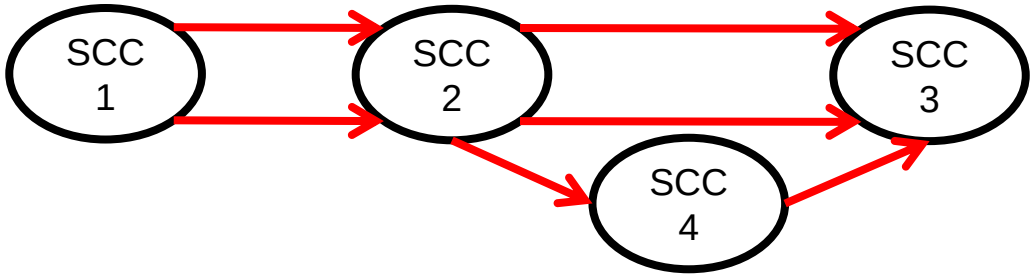
Is the 2P-SCC algorithm correct?

Meta-graph of SCCs

Zoom in



Zoom out



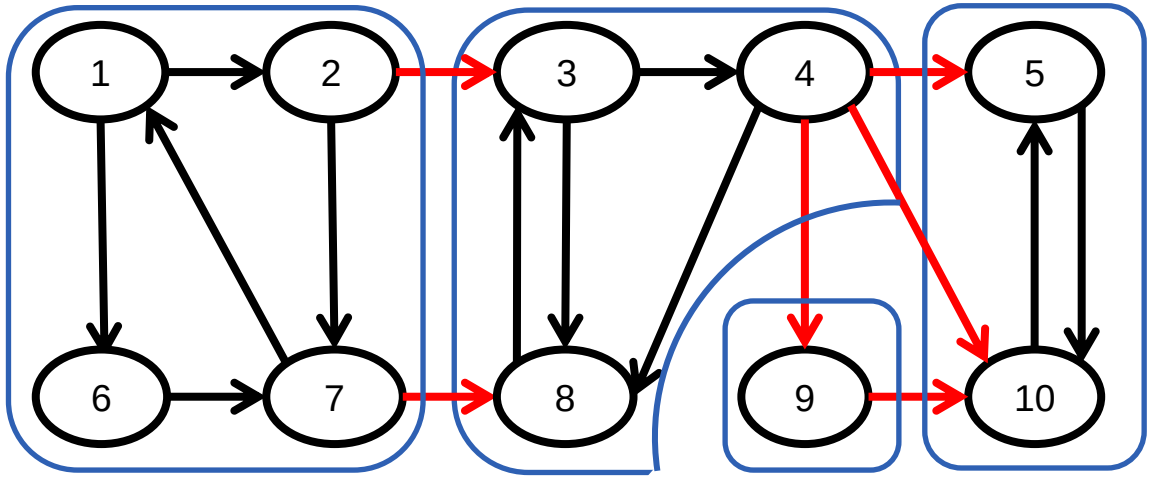
Meta-graph of SCCs

Each directed graph can be seen at two levels of granularity:

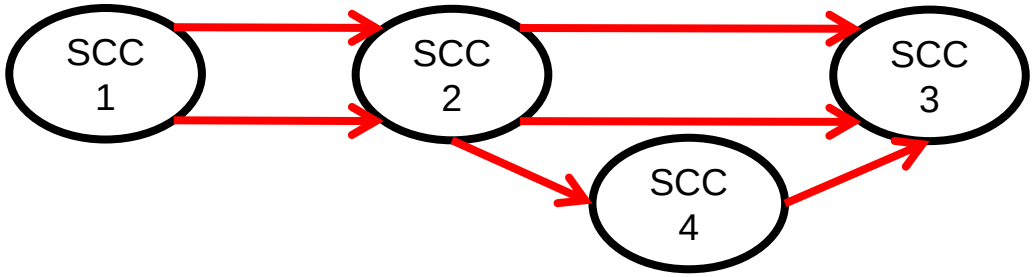
- Fine-grained: consists of all the original nodes and arcs
- Course-grained: nodes are SCCs, and we have only arcs from one SCC to another (this is a DAG – why?)

Meta-graph of SCCs

Zoom in



Zoom out



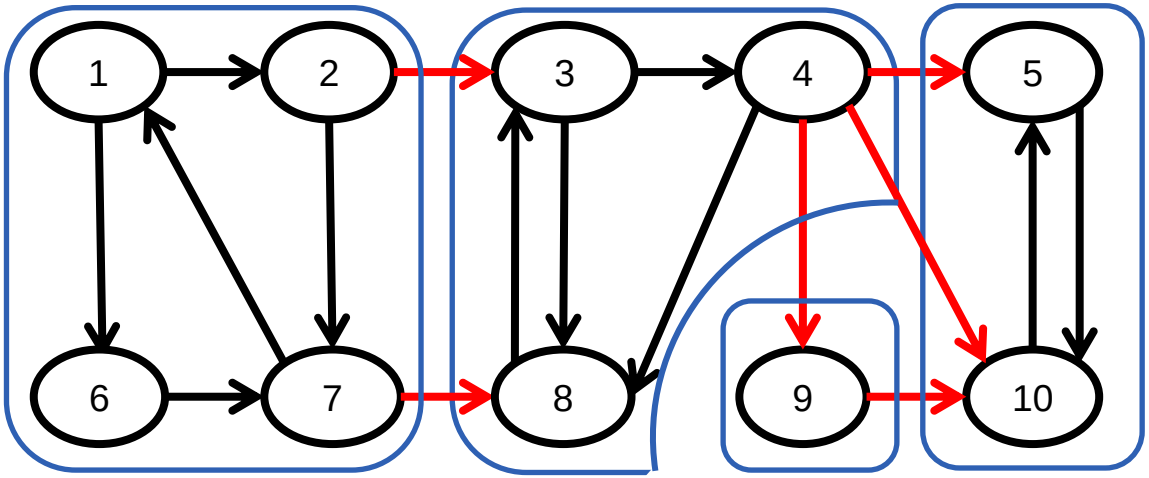
Meta-graph of SCCs

Meta-graph is always a DAG (Directed Acyclic Graph)

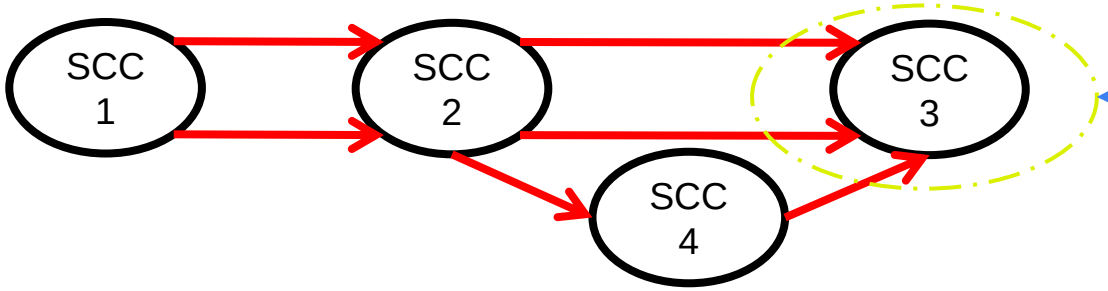
- Because there will be only one-directional edges from each meta-node to each other meta-node. For if there would be also back edges – then all nodes in 2 SCCs would be reachable from each other in both directions and 2 SCCs would collapse into one

Order of SCC discovery: intuition

Zoom in



Zoom out

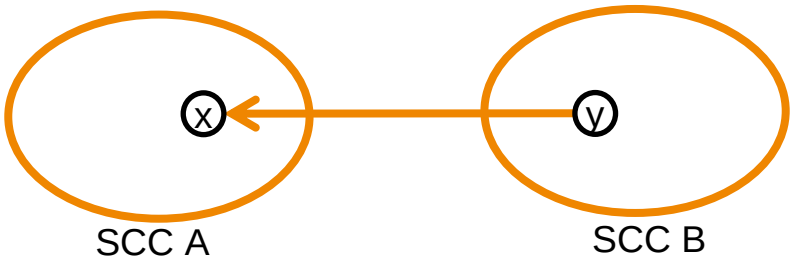


We want to peel off this SCC first

- Meta-graph is always a DAG: it has Topological Ordering. In particular, we have a sink SCC – an SCC which does not have any outgoing arcs in the meta-graph
- We should start DFS loop 2 with this sink SCC. After we mark all its nodes as processed, we essentially remove the SCC from further consideration. We then continue with the next sink SCC etc.

- But we don't know which nodes belong to the sink SCC yet. How to discover sink SCC?

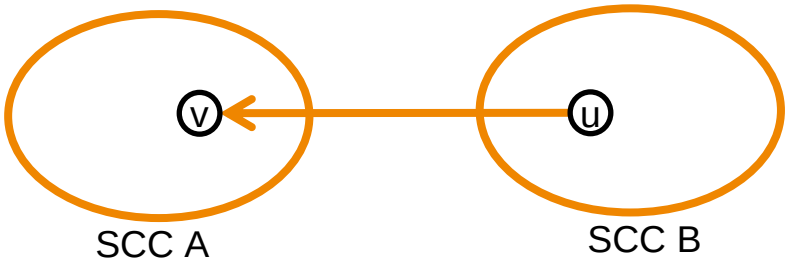
What we want to prove



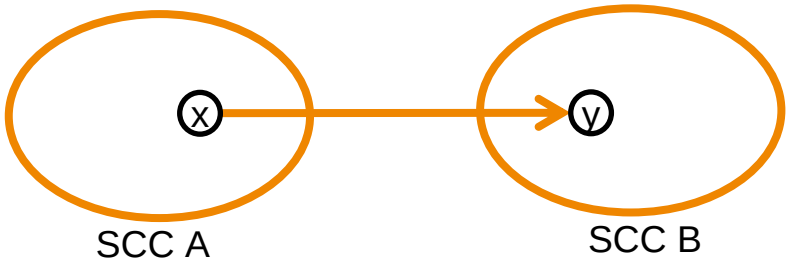
- Consider two adjacent SCC nodes A and B in a meta-graph.
- The nodes may have a one-way arc, from B to A, say
- Because there is a complete two-way reachability between all nodes inside each SCC, we have a path to any vertex in A from any vertex in B, but not in the opposite direction

- If we consider only one pair of adjacent SCCs, then one of them is a sink (SCC A is a sink in this example)
- We want to prove that the max finishing time in G^T of all vertices in A will be greater than the max finishing time of all vertices in B
- We will then use the max finishing time to identify and collect the first sink SCC

Theorem



G



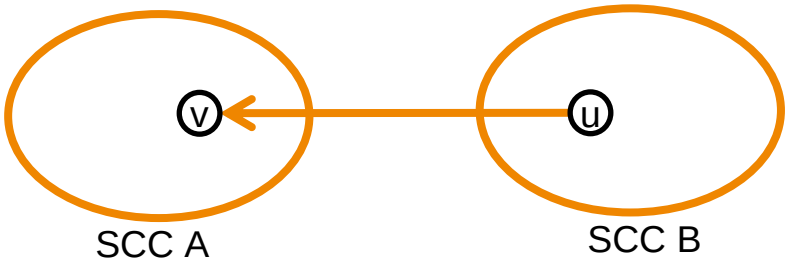
G^T

$\max (F(v), v \in A) = F(x) \; > \; \max (F(u), u \in B) = F(y)$

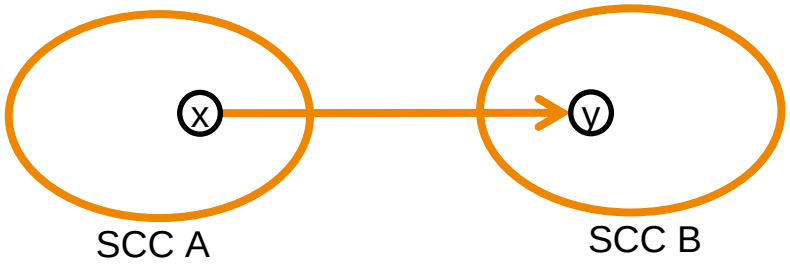
Given two adjacent SCC components in a meta-graph of all SCCs of graph G, and the finishing times F of all vertices obtained by DFS traversals in G^T, the maximum finishing time for all nodes in the sink SCC will be greater than the maximum finishing time of all nodes in the source SCC:

$$\max (F(v), v \in A) > \max (F(u), u \in B)$$

Proof



G



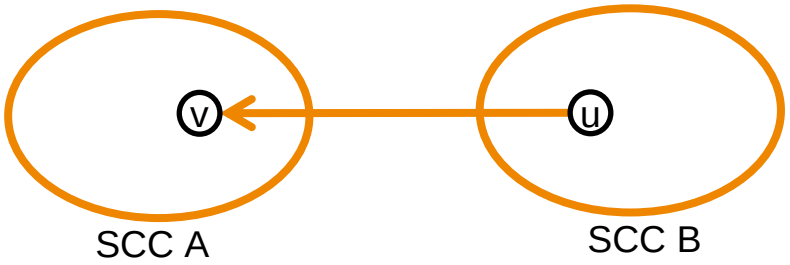
G^T

max (F(v), v ∈ A) = F(x) > max (F(u), u ∈ B) = F(y)

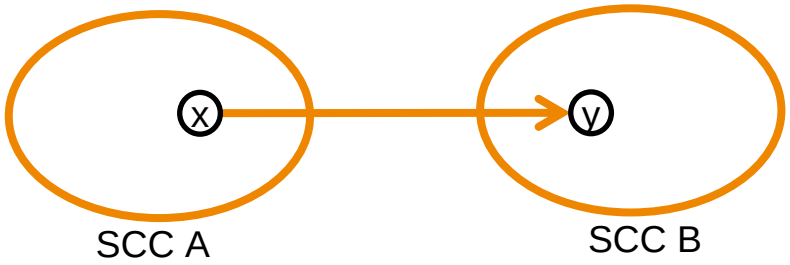
Prove that: max (F(v), v ∈ A) > max (F(u), u ∈ B)

- Let x be the vertex with the largest finishing time F among all vertices of A, and y be the vertex with the largest F among all vertices of B
- We show that F(x) > F(y), no matter of the order in which we perform the DFS loop 1

Proof



G



G^T

$\max (F(v), v \in A) = F(x) > \max (F(u), u \in B) = F(y)$

Prove that: $\max (F(v), v \in A) > \max (F(u), u \in B)$

There are only two possible cases for the order in which x and y are processed:

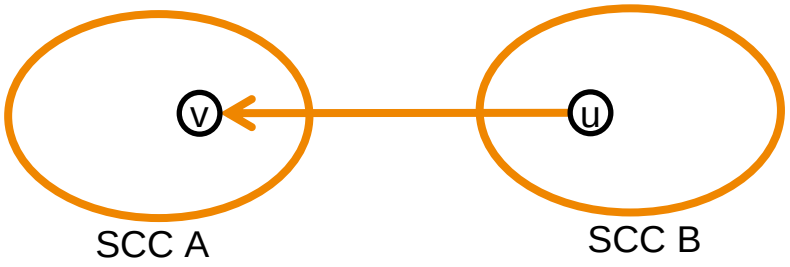
- Case 1. vertex x was picked before vertex y in DFS loop 1

Because there is a path from x to y in G^T and because y has not been discovered yet, the DFS will discover and process node y before node x: x would need to wait until all undiscovered nodes reachable from it have been processed. Thus in this case $F(x) > F(y)$

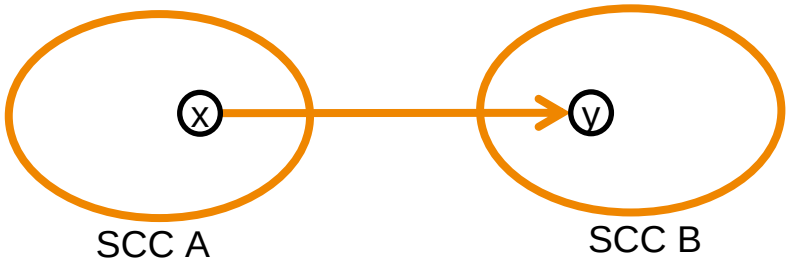
- Case 2. vertex y was picked first in DFS loop 1.

But in this case x cannot be reached from y in G^T, y will be processed once all vertices reachable from it have been processed, and only after that the DFS traversal will start from node x. Thus finishing time $F(y) < F(x)$

Correctness of 2P-SCC algorithm (sketch)



G



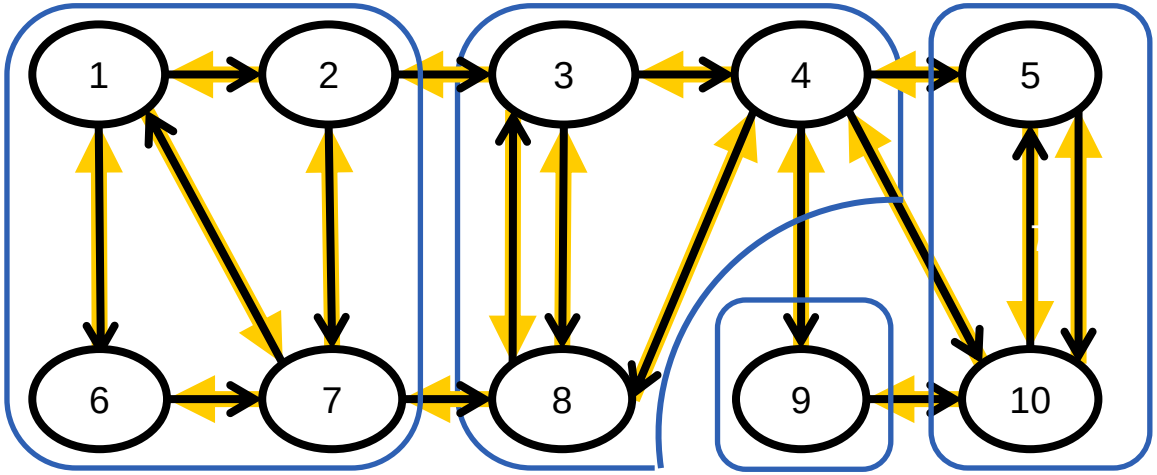
max (F(v), v ∈ A) = F(x) > max (F(u), u ∈ B) = F(y)

G^T

- Because for a pair of adjacent SCCs F(x) > F(y), if we apply the result of our theorem to all pairs of adjacent SCCs, the max finishing time will be in the sink SCC

- In DFS loop 2 we pick the node with max finishing time, and this node is guaranteed to be in the sink SCC. We collect all the nodes in the first sink, then remove all vertices in the first sink SCC from consideration, and move to the max F of all remaining vertices.
- Thus, performing the DFS loop in reverse order of these finishing times allows us to discover each SCC of G

Question to think about



Are the SCCs in G^T exactly the same as in G?

Can you prove this for a general directed graph?