Another DFS application: Strongly Connected Components

Lecture 04.03

By Marina Barsky

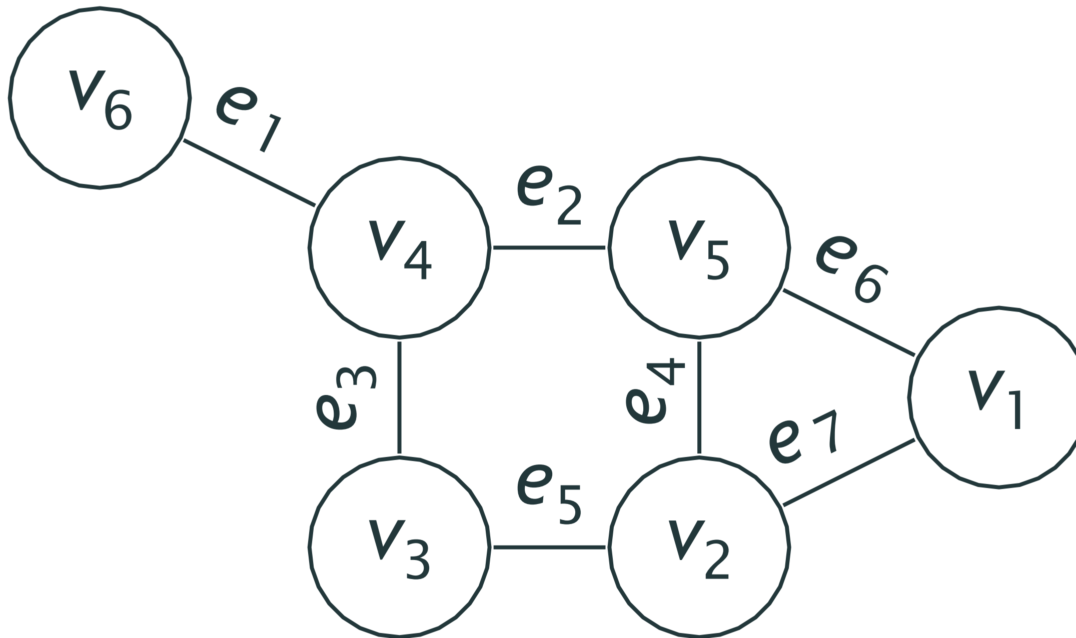
Walks and Paths

- A walk in a graph is a sequence of incident edges
- The length of a walk is the number of edges in it
- A path is a walk where all edges are distinct
- A simple path is a walk where all vertices are distinct

Example 1

A walk of length 6: (e_1, e_2, e_4, e_5, e_3, e_1)

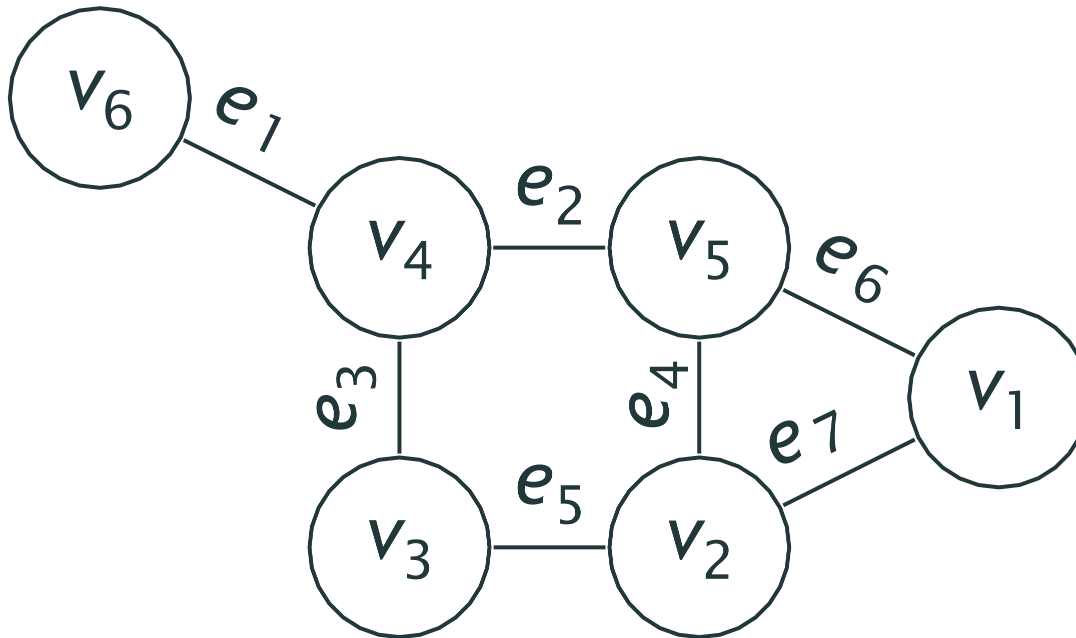
Is this walk also a path?



Example 1

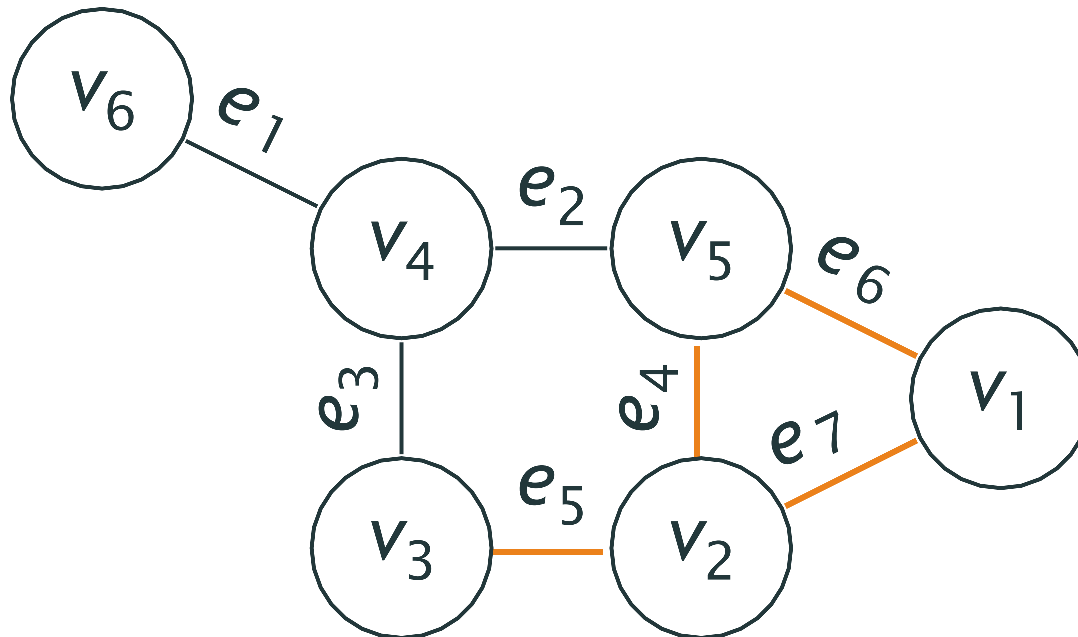
A walk of length 6: (e_1, e_2, e_4, e_5, e_3, e_1)

Not a path: uses e_1 twice



Example 2

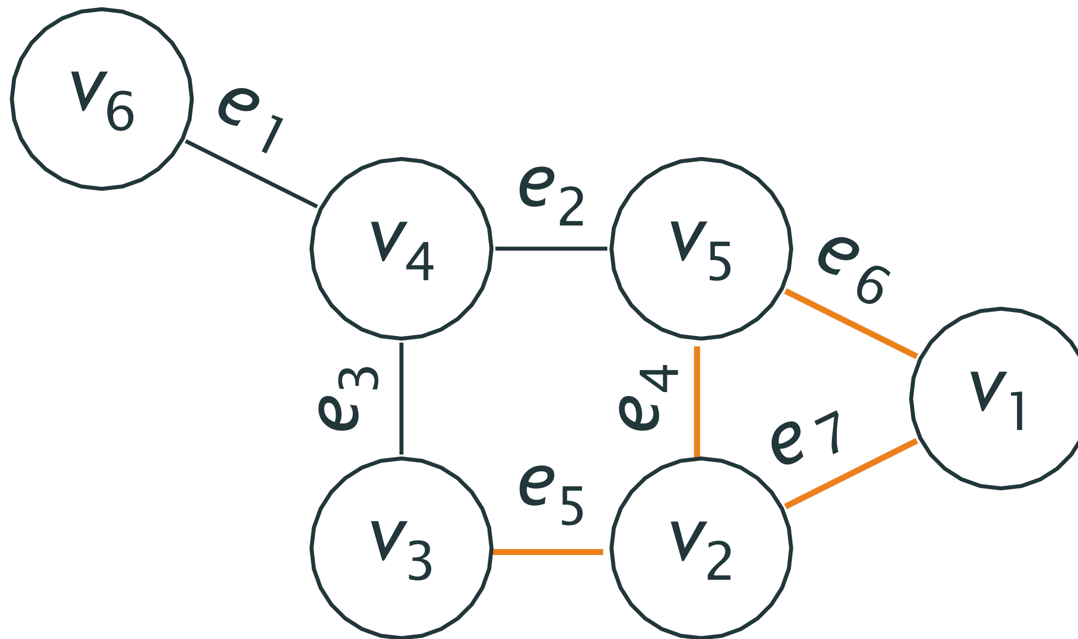
A path of length 4: (e_7, e_6, e_4, e_5)
Is it a simple path?



Example 2

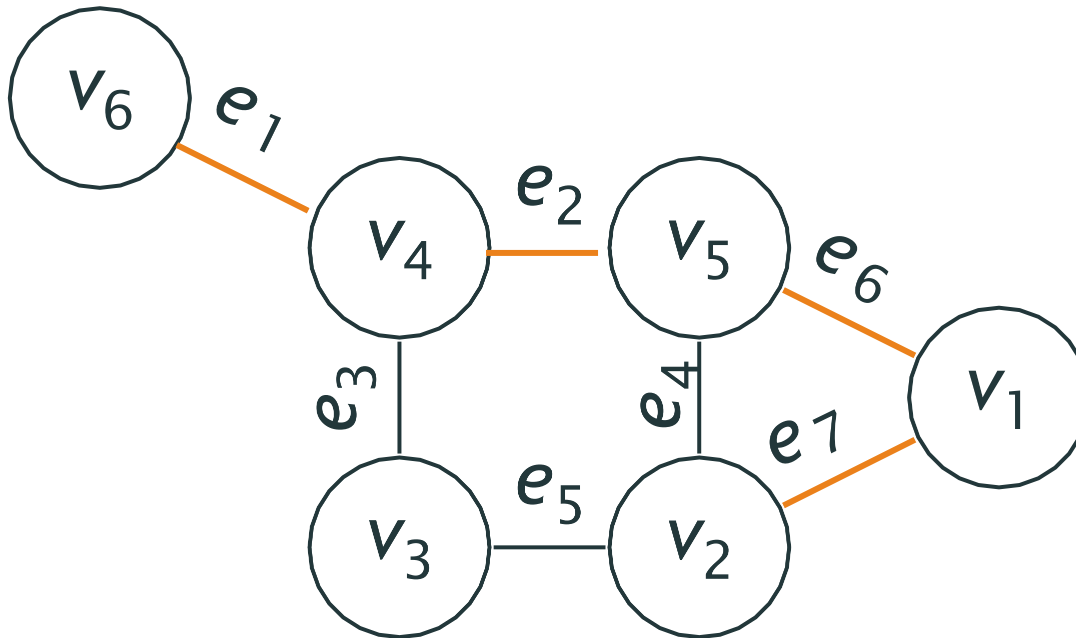
A path of length 4: (e_7, e_6, e_4, e_5)

Not a simple path: visits v_2 twice



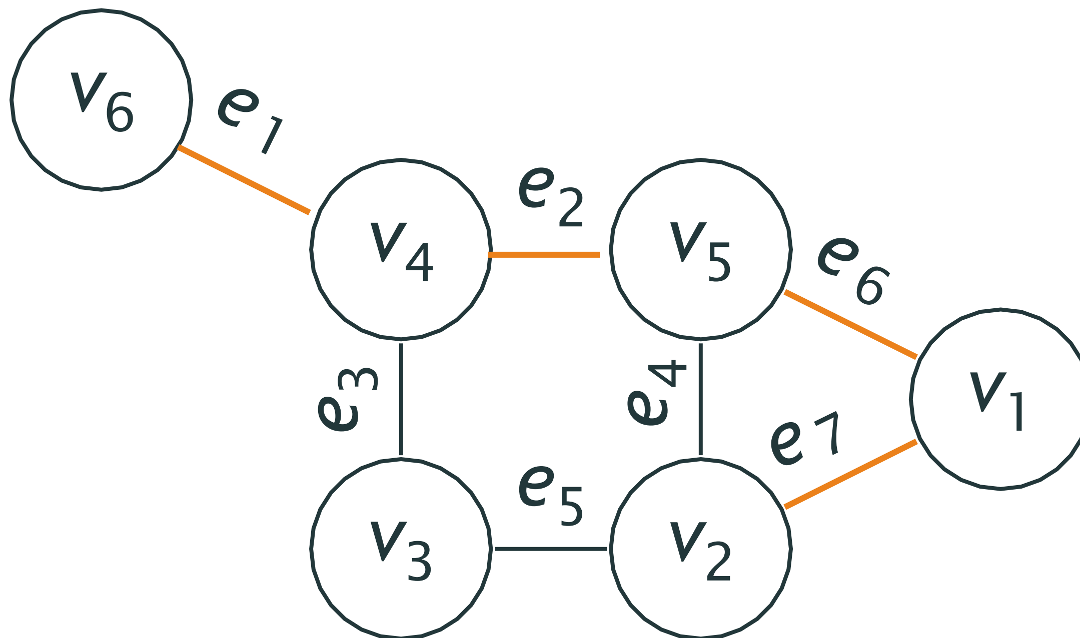
Example 3

A simple path of length 4: (e_7, e_6, e_2, e_1)

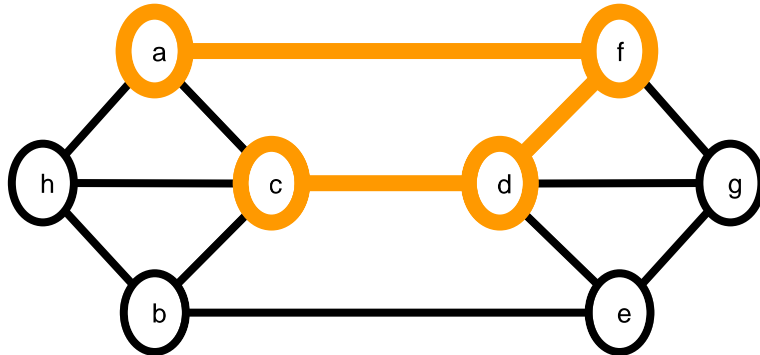


It is sometimes more convenient to specify a path (walk) by a list of vertices rather than edges

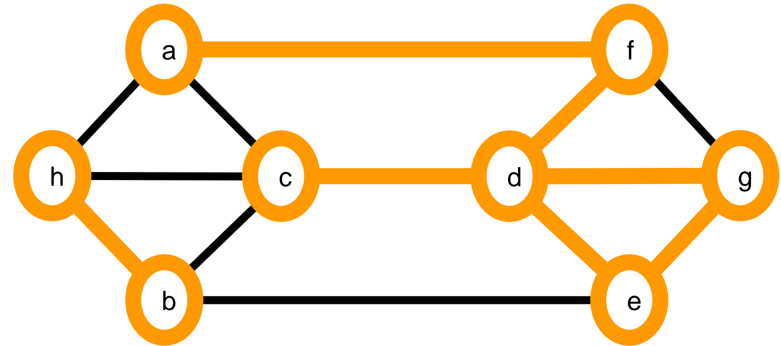
A path of length 4: $(v_2, v_1, v_5, v_4, v_6)$



The length of the path



A highlighted path
a, (a,f), f, (f,d), d, (d,c), c



This is not a path since it is disconnected and also d appears multiple times.

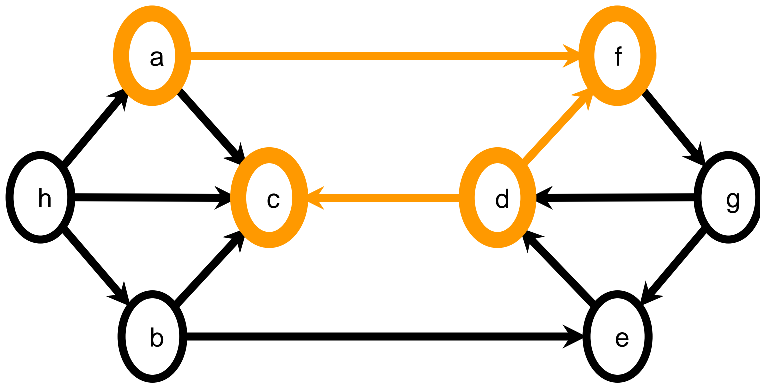
The ***length of a path*** is the number of traversed edges.

A path from u to v is **a** ***shortest path*** if there is no shorter path from u to v.
For example, there are two shortest paths from f to e above.

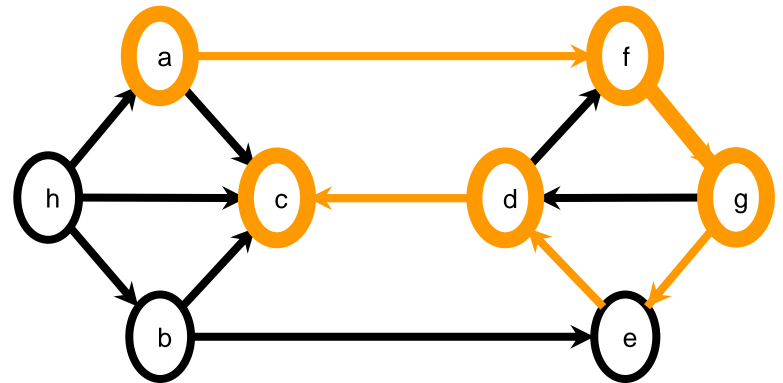
Directed Paths

In a directed graph each edge is oriented in one of two ways with respect to a path:

- The edge is *forward* if it has the form $v_i, (v_i, v_{i+1}), v_{i+1}$.
- The edge is *backward* if it has the form $v_i, (v_{i+1}, v_i), v_{i+1}$.



A highlighted path
a, (a,f), f, (f,d), d, (d,c), c
where (f,d) is the only backwards edge.



A directed path from a to c.

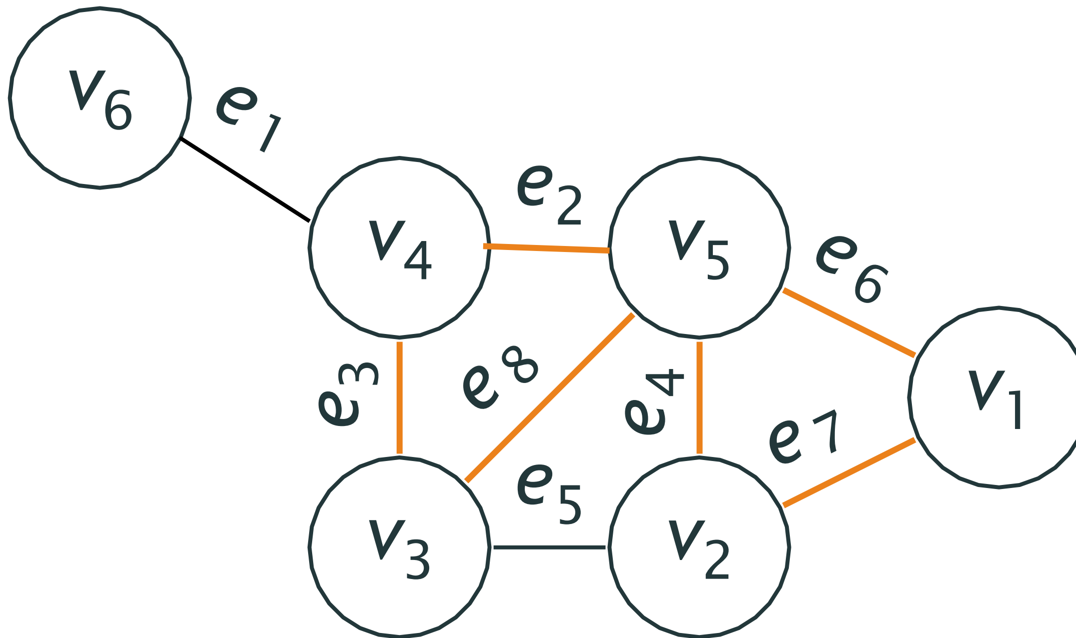
A path is a *directed path* if every edge is a forward edge.

Cycles

- A cycle (sometimes called a circuit) in a graph is a path where the first vertex is the same as the last one
- All the edges in a cycle are distinct
- A simple cycle is a cycle where all vertices except for the first=last are distinct

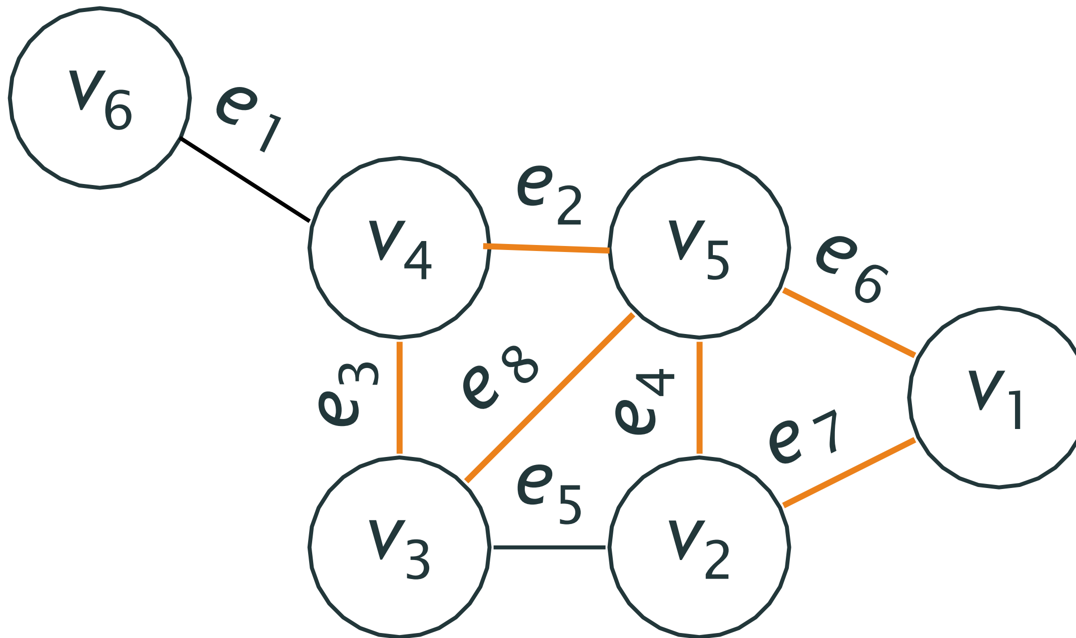
Example 4

A cycle of length 6: $(e_2, e_3, e_8, e_4, e_7, e_6)$



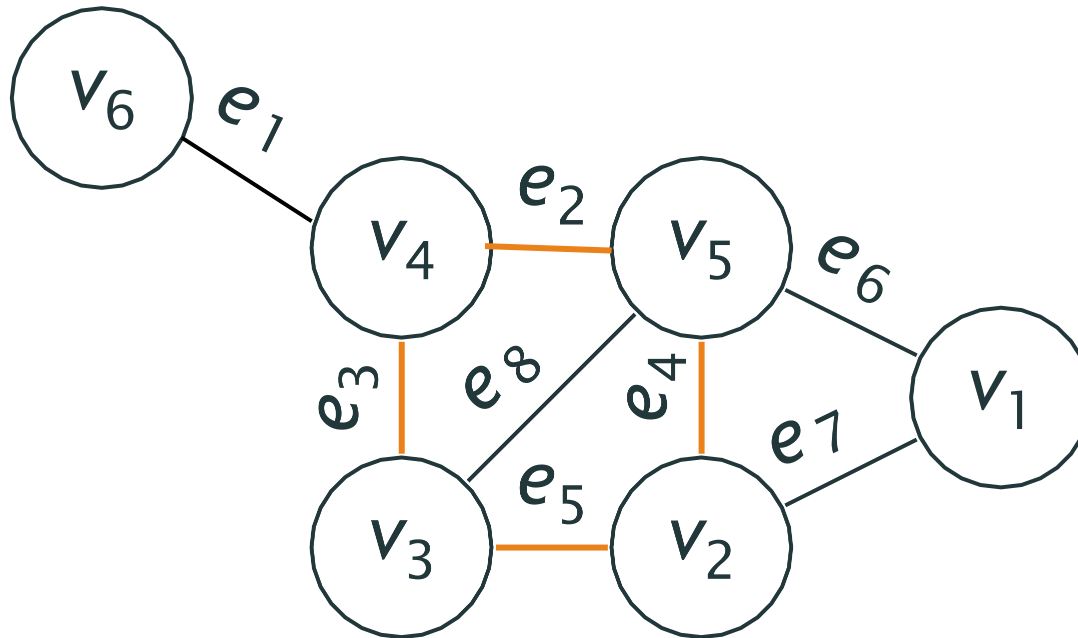
Example 4

A cycle of length 6: $(e_2, e_3, e_8, e_4, e_7, e_6)$
Not a simple cycle: visits v_5 three times



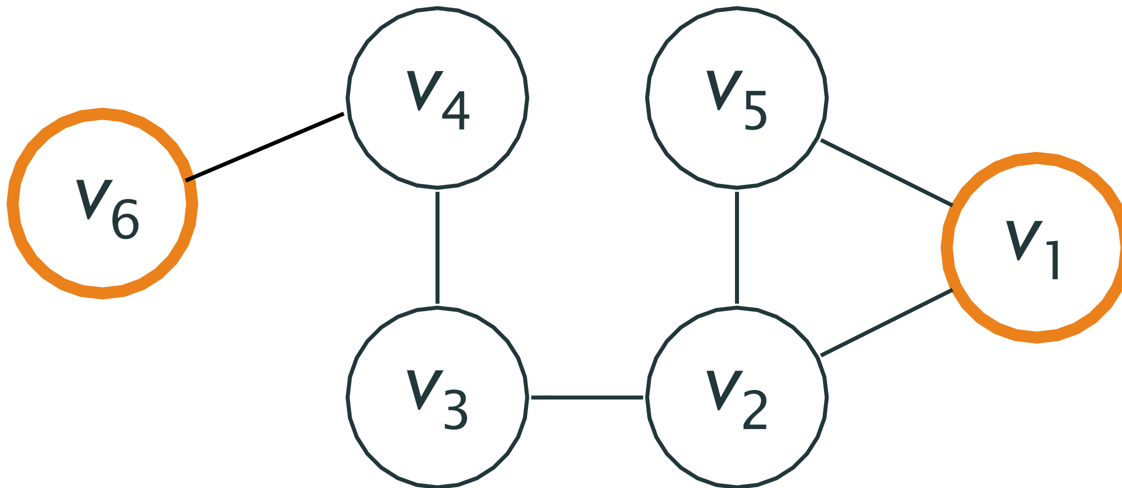
Example 5

A simple cycle of length 4: (e_5, e_4, e_2, e_3)



Connectivity in undirected graphs

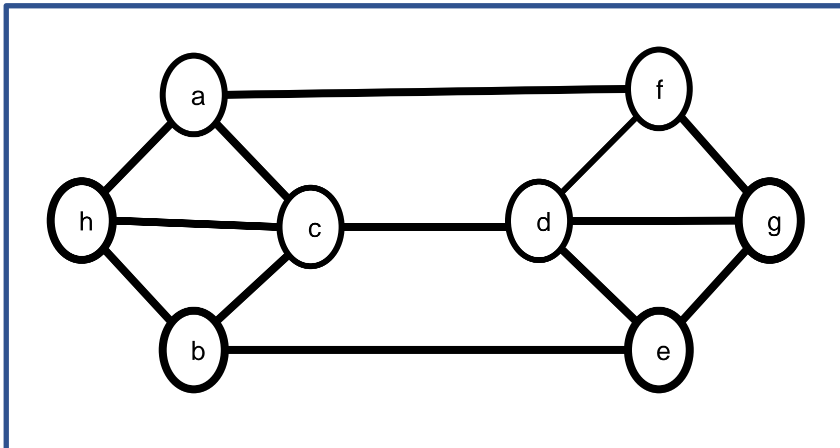
- Two vertices are connected, if there is a path between them
- The definition is transitive: if *u* and *v* are connected and *v* and *w* are connected, then *u* and *w* are connected as well



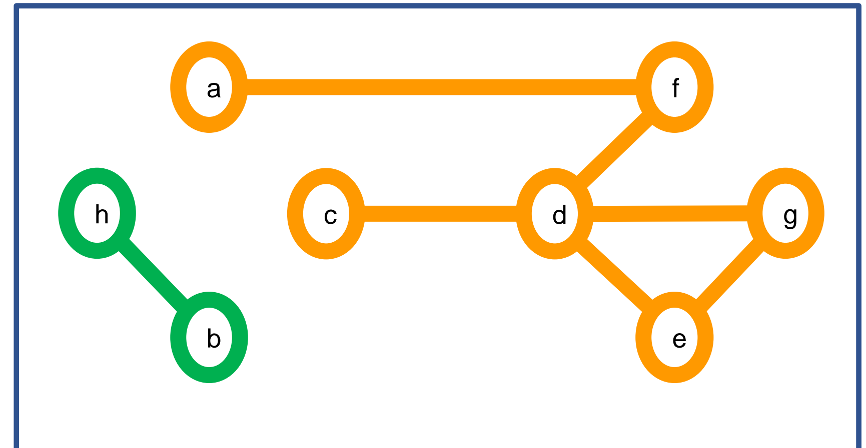
v_1 and v_6 are connected.

Connected graph

- A graph is connected, if any two of its nodes are connected. In other words, there is a path between any pair of nodes



This graph is connected.



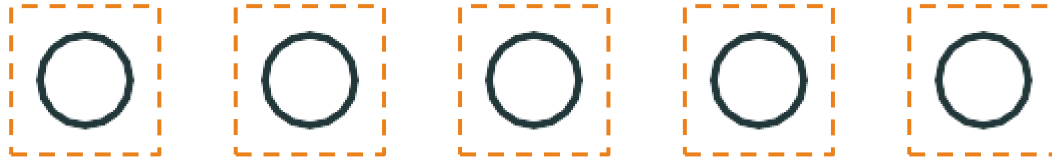
This graph is not connected.

Connected components

The nodes of any undirected graph can be partitioned into subgraphs called connected components:

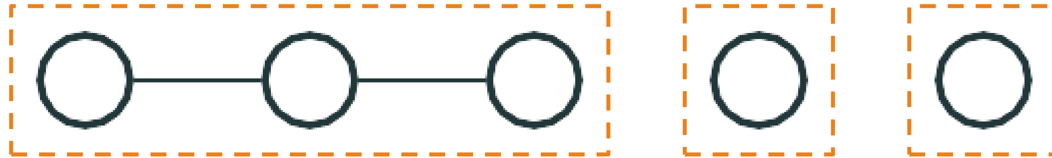
- Any node belongs to exactly one connected component
- Any two nodes from the same connected component are connected
- Any two nodes from different connected components are not connected

Example 6



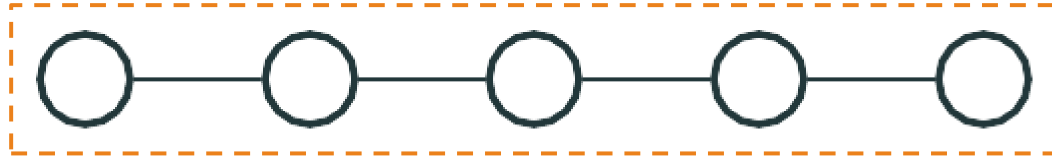
Graph with 5 connected components

Example 7



Graph with 3 connected components

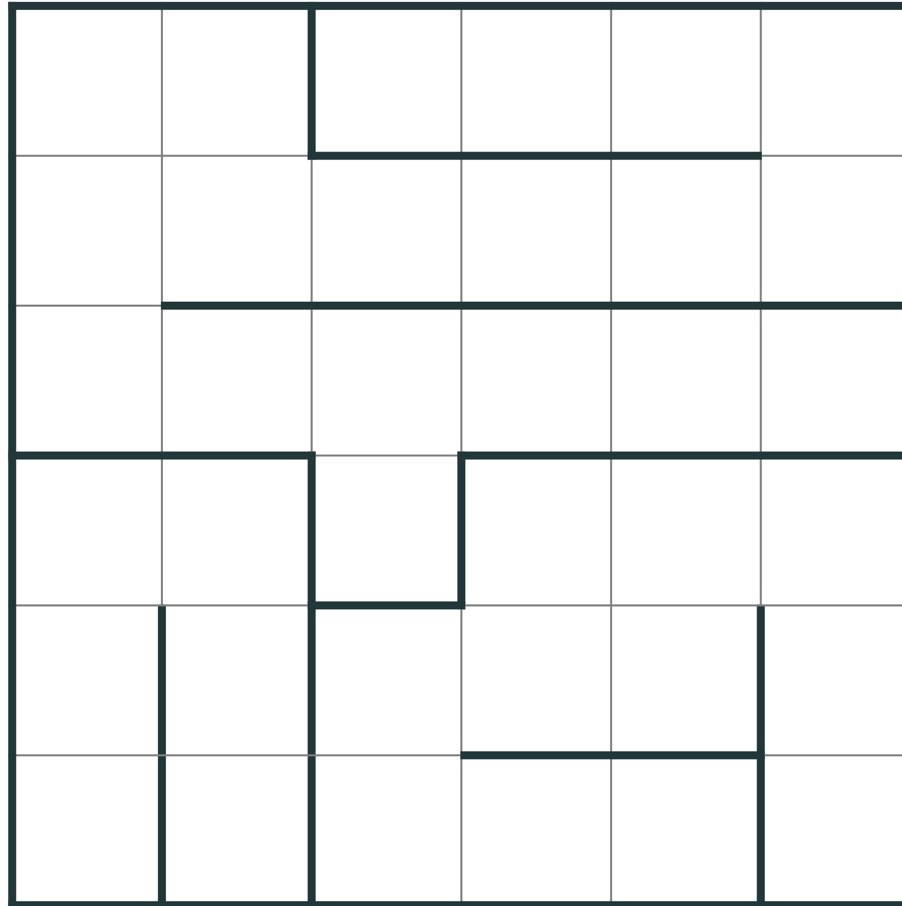
Example 8



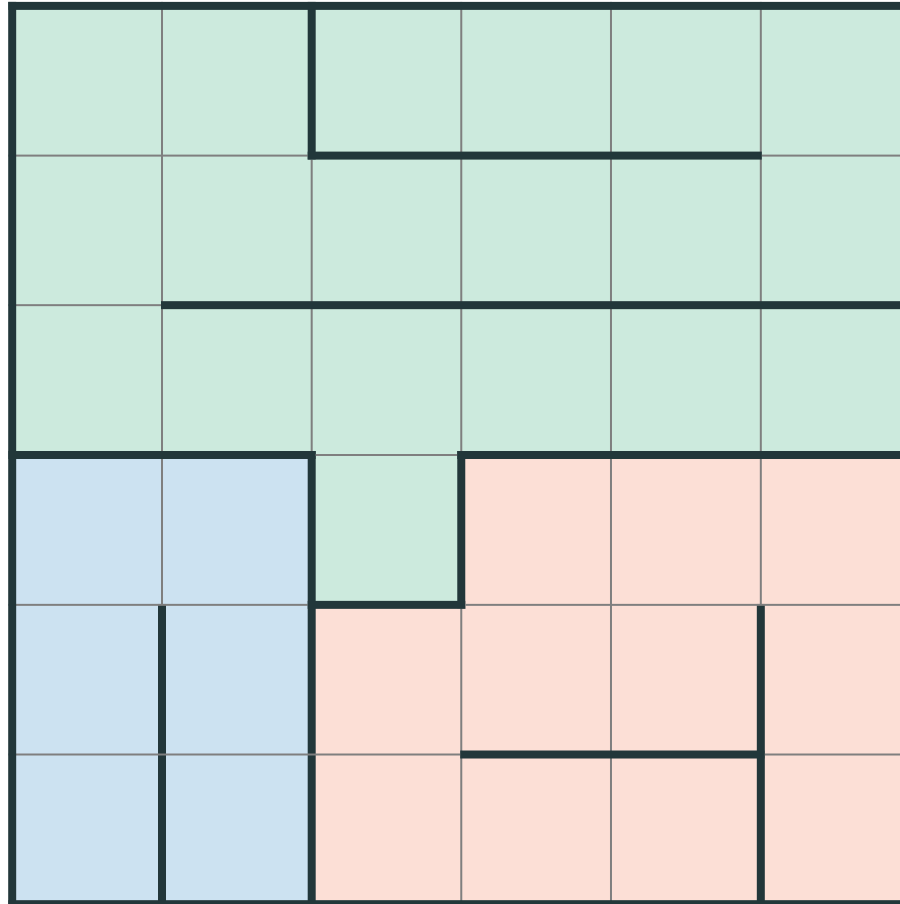
Graph with 1 connected component

This graph is *connected*

Connected Components in a Maze

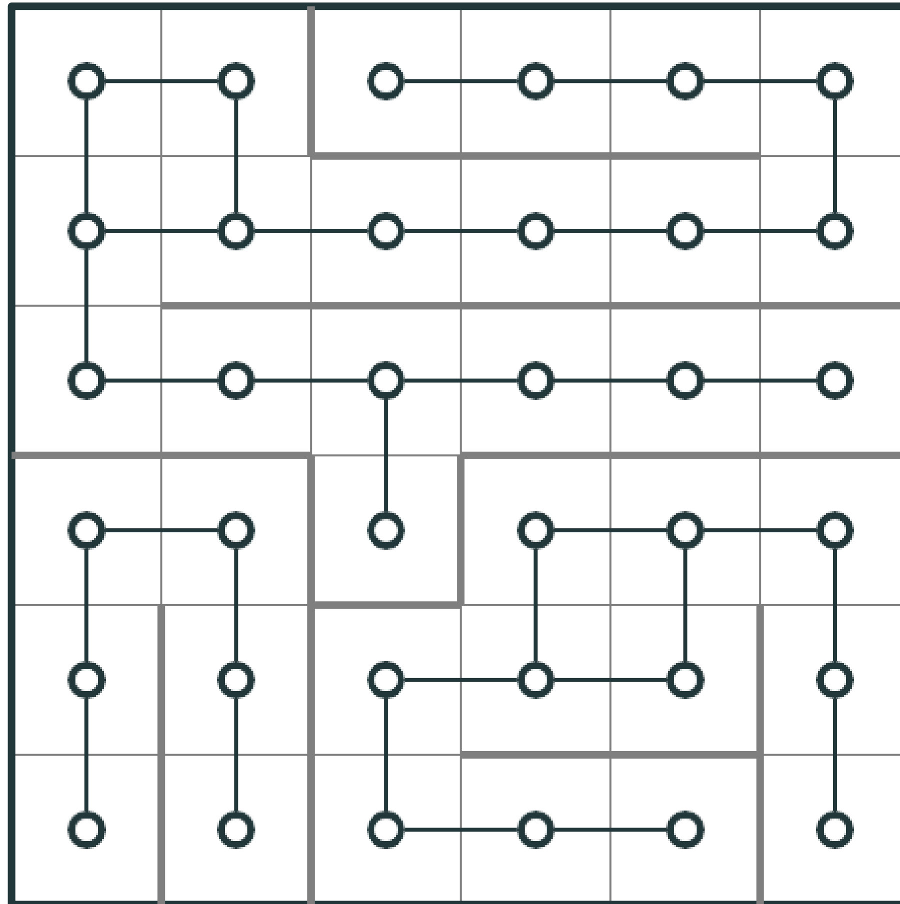


Connected Components in a Maze

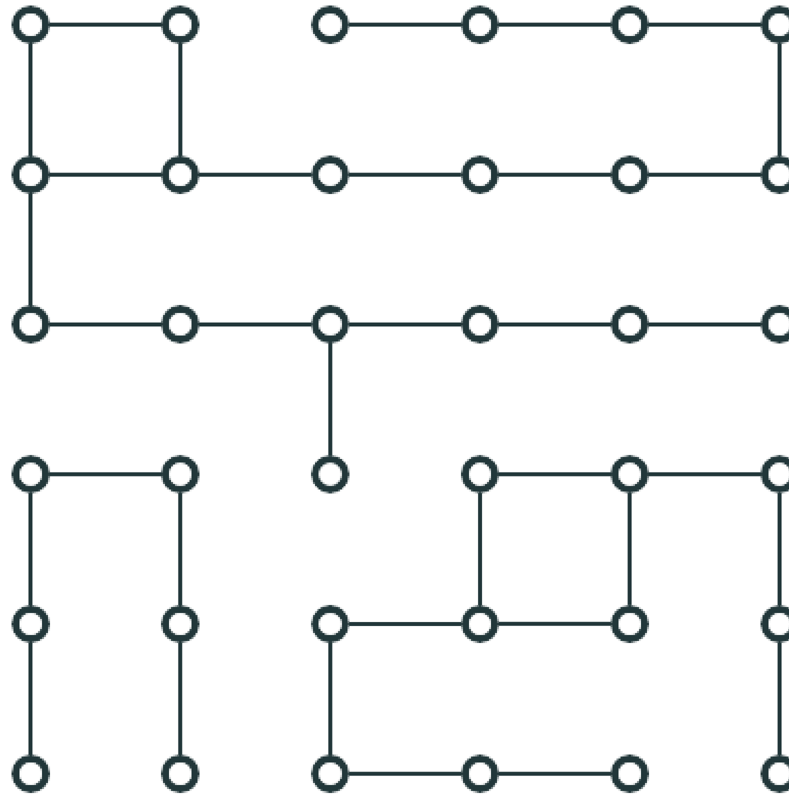


Connected Components in a Maze

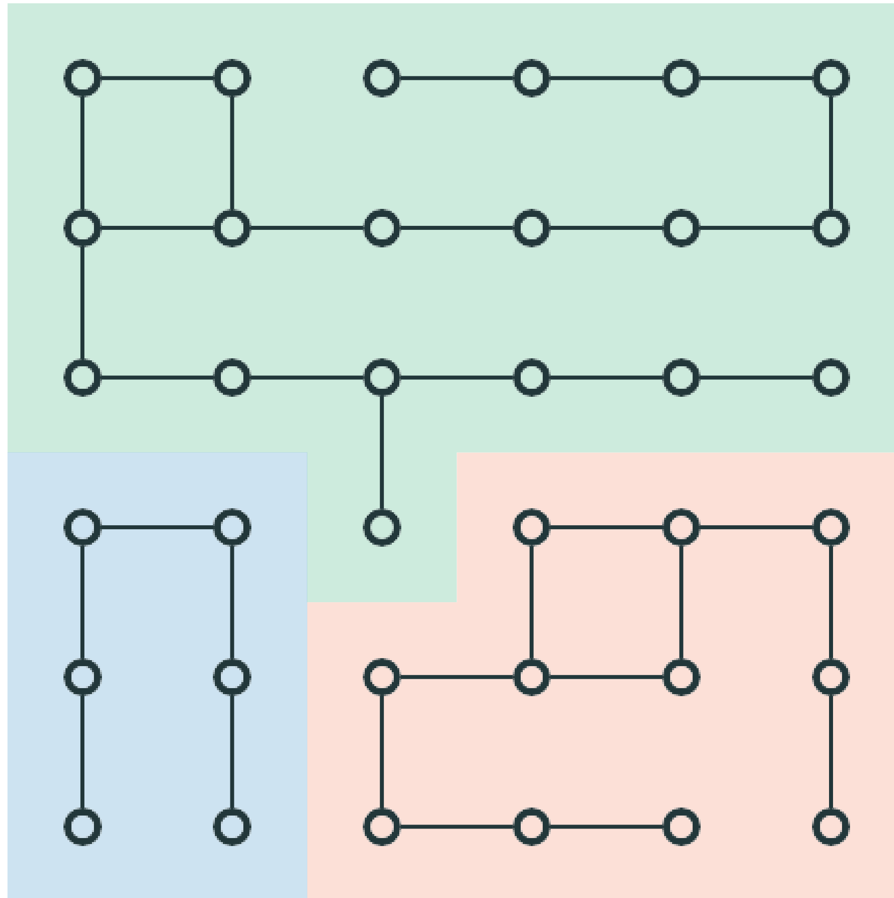
Connected Components in a Maze



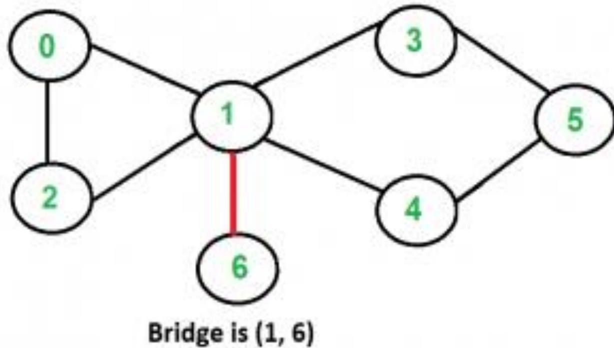
Connected Components in a Maze

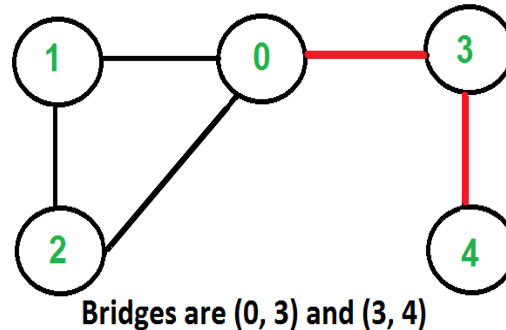


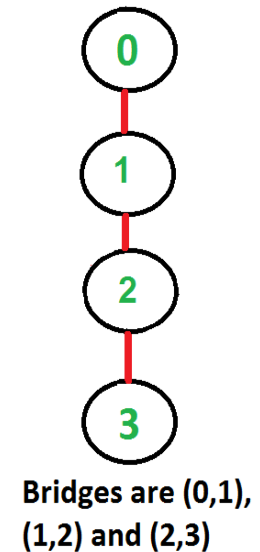
Connected Components in a Maze



Bridge







- An edge in an undirected connected graph is a **bridge** iff removing it increases number of connected components
- Bridges represent vulnerabilities in a connected network and are useful for designing reliable networks

  For example, in a wired computer network, a bridge indicates the critical wires or connections

Questions

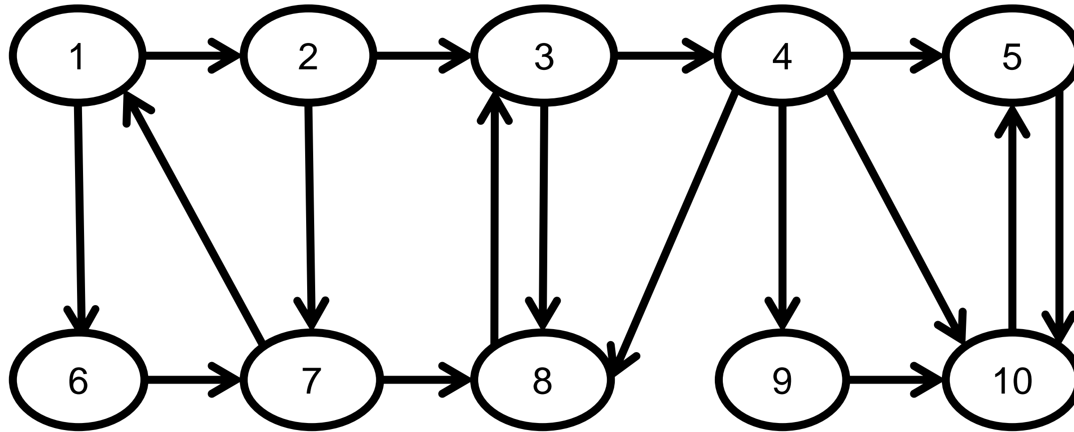
1. How to find out whether an undirected Graph is connected?
2. How to compute all connected components in the undirected Graph?

- Hint: traversals
- What is the running time of these algorithms?

Connectivity in Directed Graphs

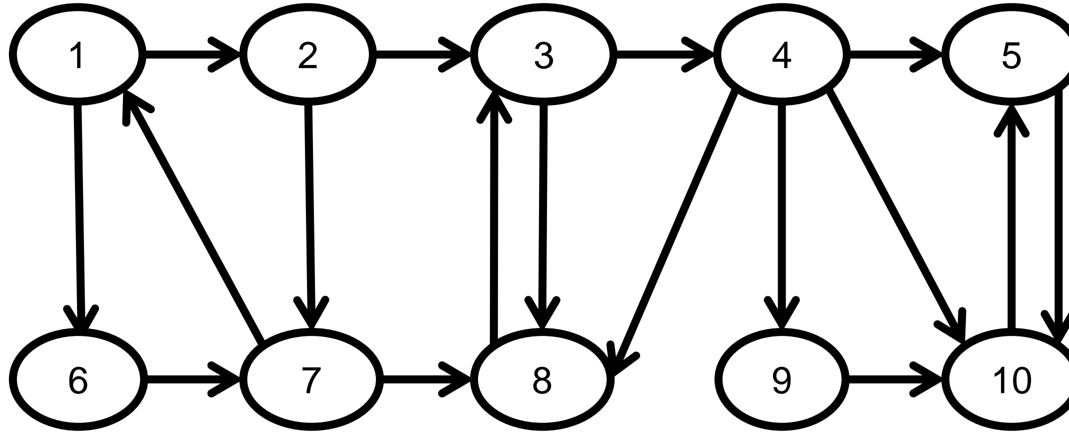
Strong connectivity

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 either path v~>u **OR** path u~>v
- This is called **weak connectivity**

Strongly connected directed graphs



This graph is NOT strongly-connected:
There is a path from 1 to 8, but there is no path from 8 to 1

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

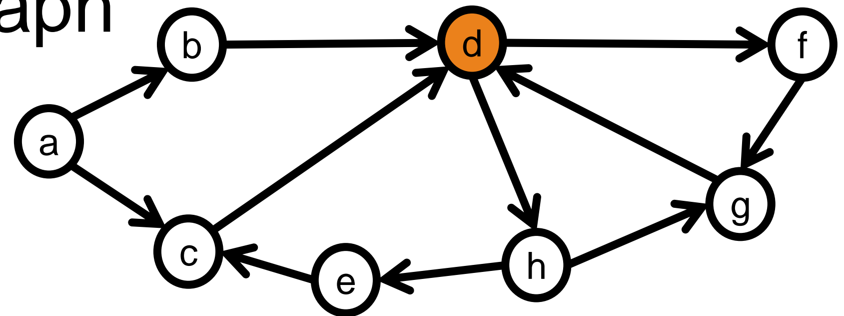
Connectivity vs. strong connectivity

How do we test whether an undirected graph G is strongly connected?

- Start DFS (or even BFS) loop from a random vertex s
- By the end of DFS:
  - All vertices are processed → G is connected: there is a path between s and any other vertex, and there is also a path between any pair of vertices (even if this path would need to go to s first)
  - Not all vertices are processed → G is not connected: there is no path from s to some vertices

- Can we apply this idea to test that directed graph G is strongly connected?

Recap: 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, s)   // s is a vertex in G
```

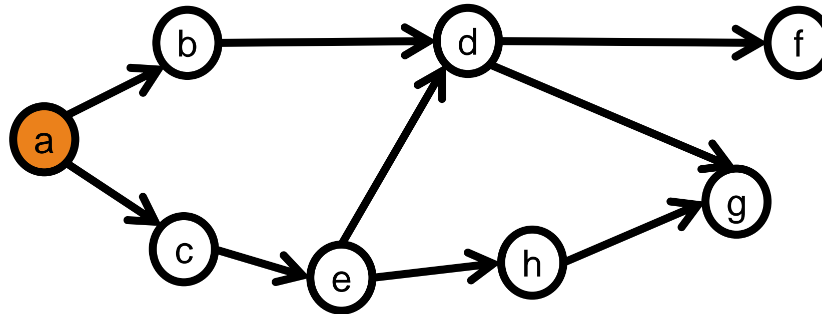
By the end we discover all the nodes in digraph G that are reachable from the source node *s*

Testing digraph G for strong connectivity

We can try running DFS:

DFS (G, s)

- Pick any node s - arbitrarily
- Perform DFS from s: every node which is reachable from s 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 s to any other node in G
- This however does not guarantee that there is a path from any node to s, or that there is a path between any pair of nodes



All nodes of G are reachable from a,
but there is not even a one-way path between g and f, or h and d

One DFS is not enough!

Testing digraph G for strong connectivity

We can do it using **two runs of DFS**:

DFS1 (G, v)

- Pick any node s - arbitrarily
- Perform DFS from s: every node which is reachable from s 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 s to any other node in G

DFS2 (G^T, v)

- Now we need to check if there is also a path from any vertex in G to s (this is required for strong connectivity)
- We want to know if there is a path from any vertex in G to s – should we do DFS from every vertex?
- To check if there is a path towards s from any other vertex, we perform DFS using backward edges from s. We create 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 from every vertex in G towards s

Testing digraph G for strong connectivity

We can do it using two runs of DFS:

DFS1 (G, v)

- Pick any node s - arbitrarily
- Perform DFS from s: every node which is reachable from s 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 s to any other node in G

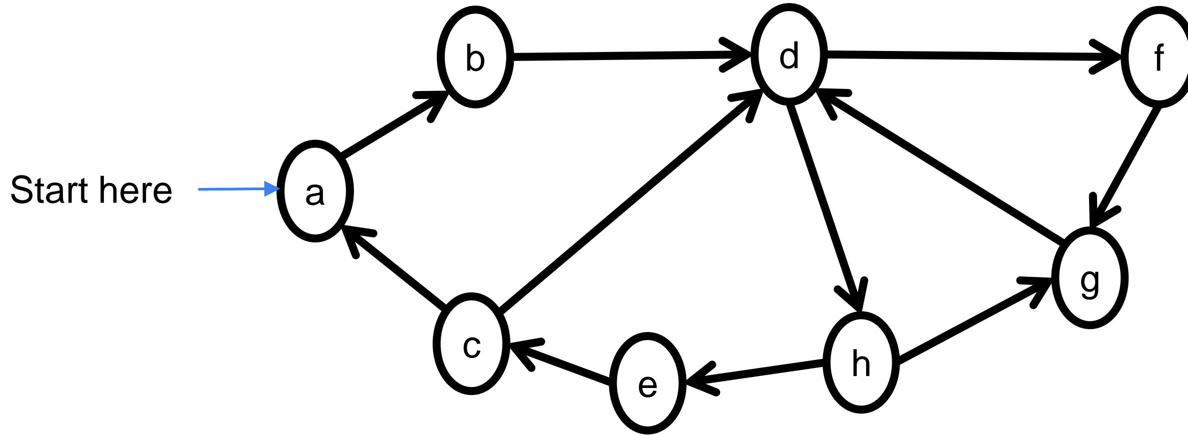
DFS2 (G^T, v)

- Now we need to check if there is also a path from any vertex in G to s (this is required for strong connectivity)
- We want to know if there is a path from any vertex in G to s – should we do DFS from every vertex?
- To check if there is a path towards s from any other vertex, 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 vertex in G with s in the opposite direction

This also guarantees that G is strongly connected

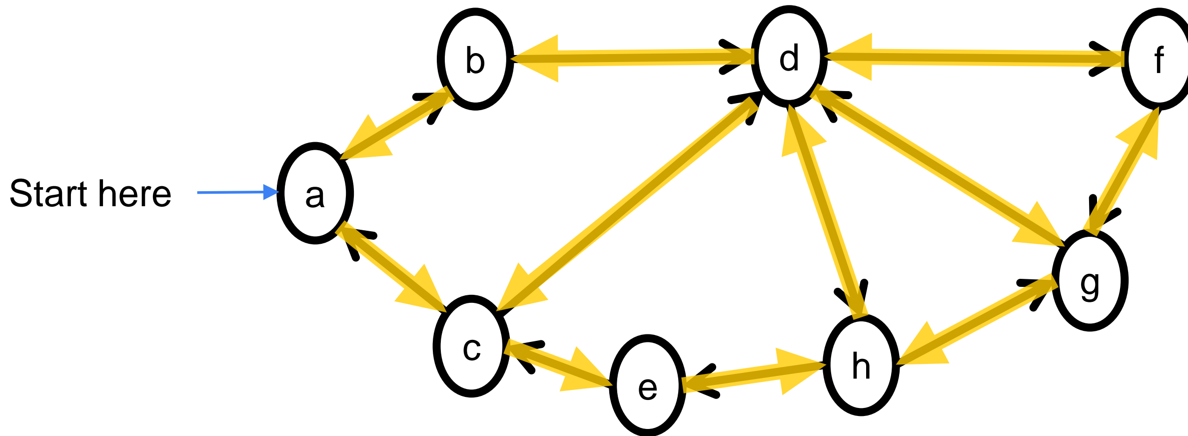
Can you see why?

Is this graph strongly-connected?



Graph G

- All the nodes are processed with DFS1 (G, a)

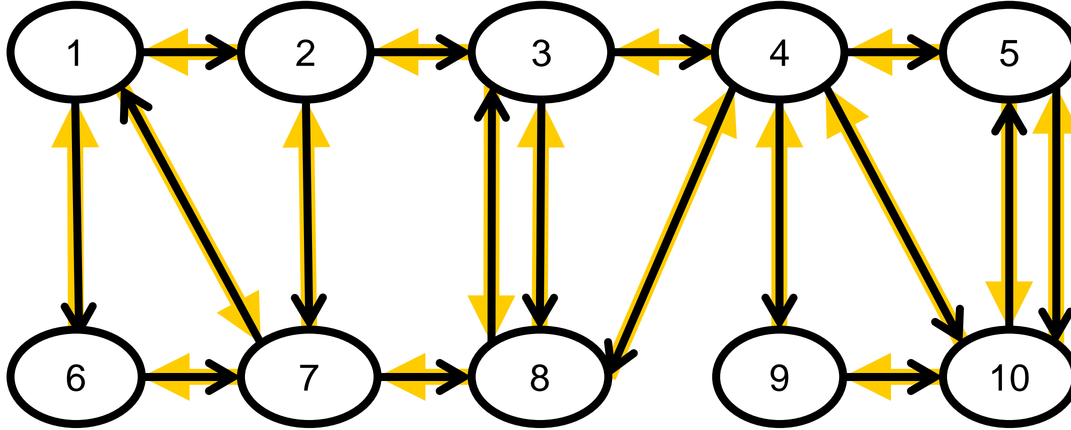


Graph G^T

- All the nodes are processed with DFS2 (G^T, a)

Conclusion: this graph is strongly-connected

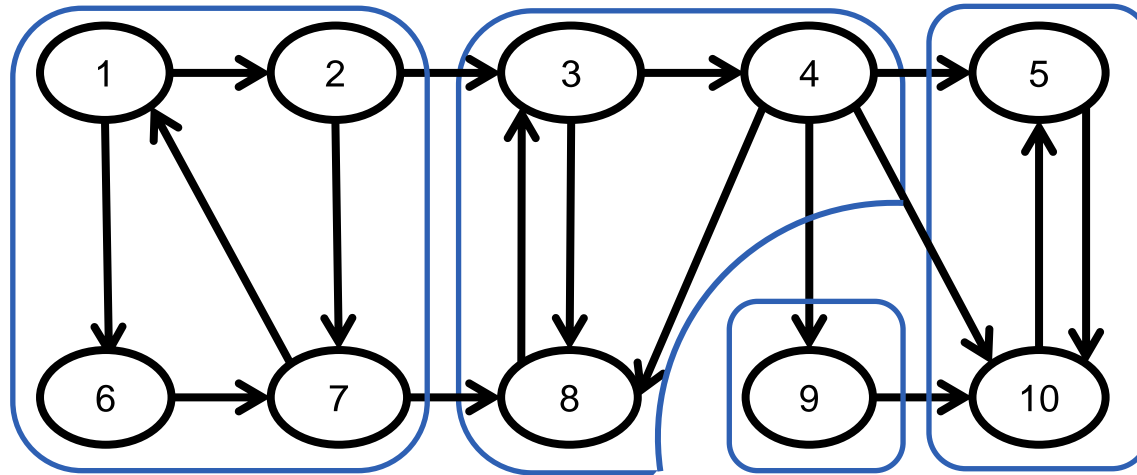
Is this graph strongly-connected?



- If we do DFS1(G, 1) we reach all the nodes
- DFS2 in G^T tells us if there is a return path in G from any node to node 1
- If we do DFS2(G^T, 1) we will only reach nodes 2, 7, 6

Conclusion: this graph is not strongly-connected

Strongly Connected Components (SCCs)

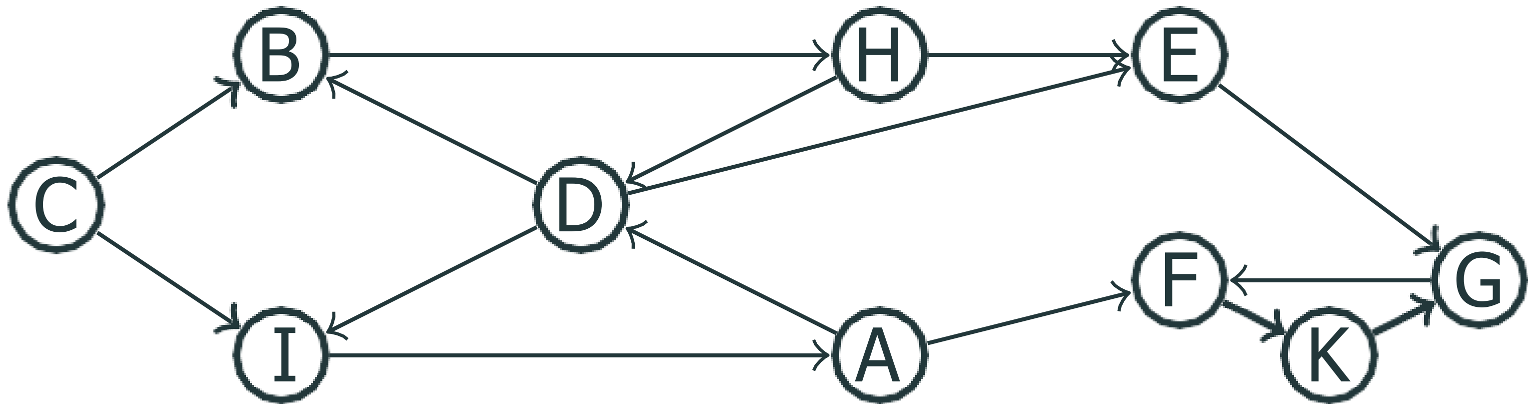


- The entire graph is not strongly-connected
- Is there any sub-graph that is strongly-connected?

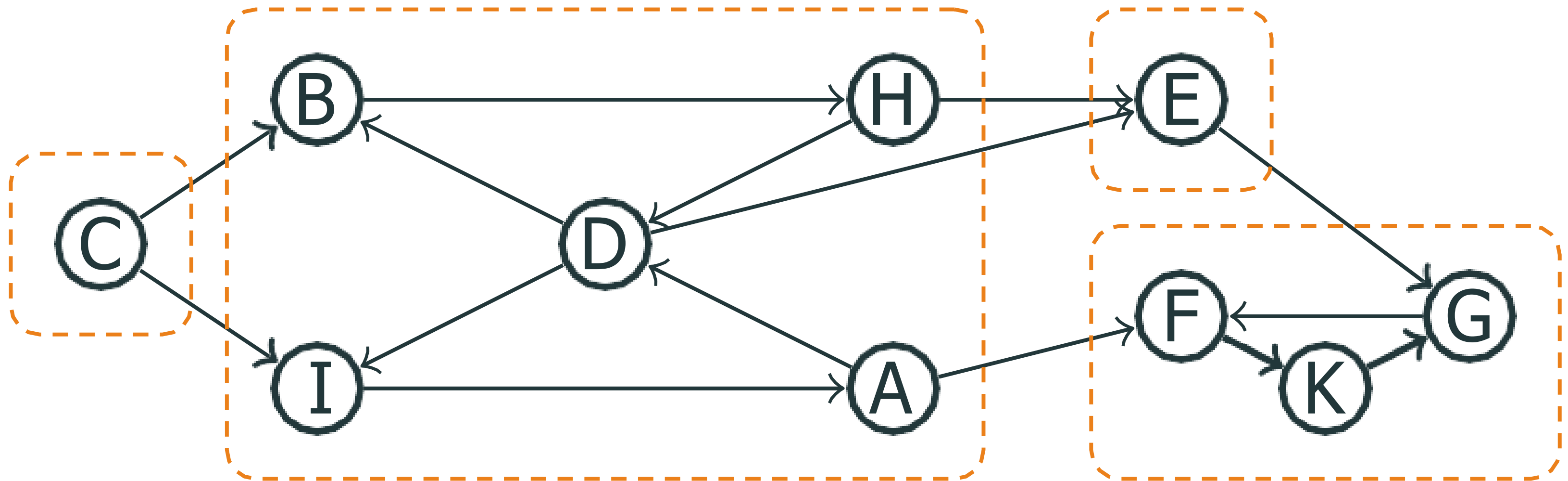
- Given digraph G = (V, E), we define ***a strongly connected component (SCC)*** of G to be a maximal subset C of vertices V, such that for all u, v in C, both u ~> v and v ~> u – that is, both u and v are reachable from each other. In other words, two vertices of directed graph are in the same SCC if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

- There are 4 strongly-connected components in this graph

What are SCCs in the following graph?

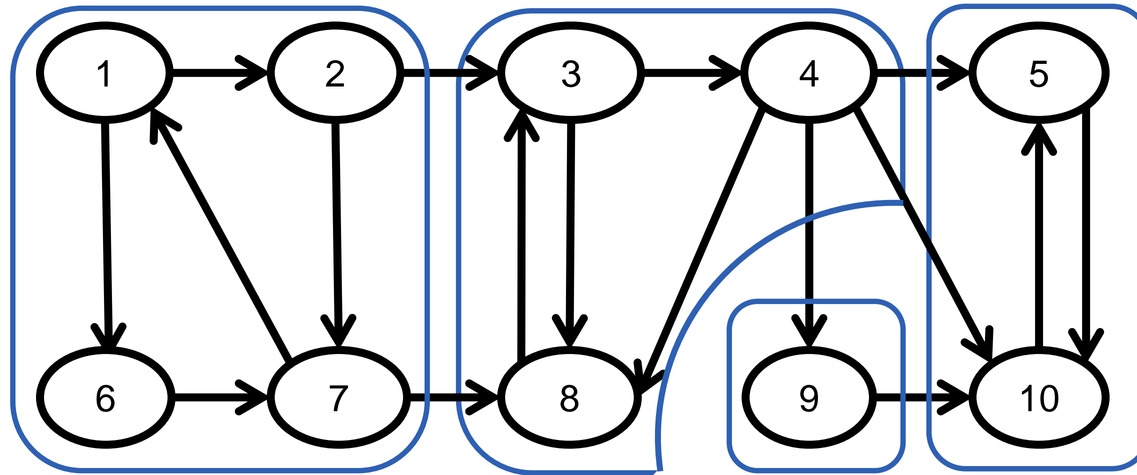


What are SCCs in the following graph?



This graph has 4 SCCs

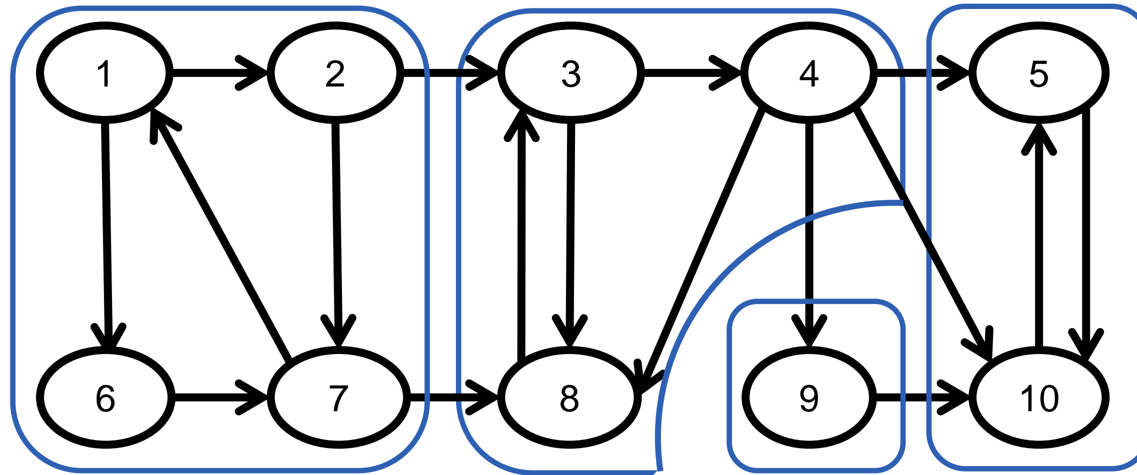
Discovering Strongly Connected Components



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

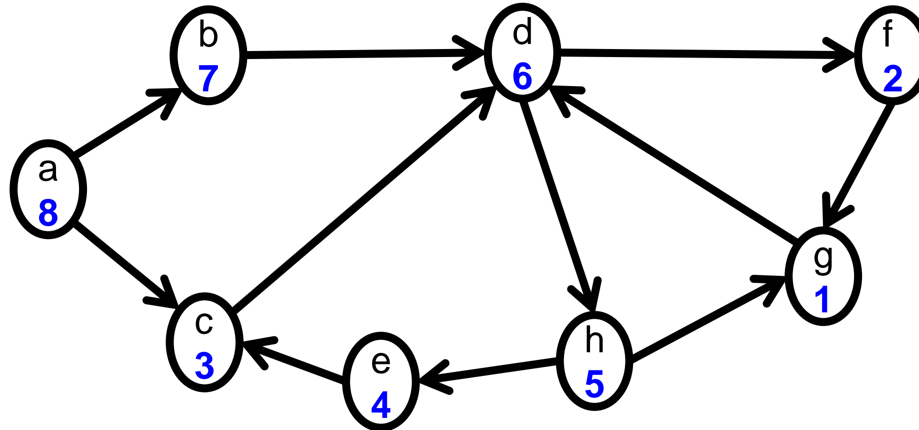
  For example, in social networks these are groups of people who can communicate with each other freely and have a limited connectivity to other groups.

Can we use DFS loop?



- We might think of applying the DFS loop (or 2 DFS loops) 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

Recap: finishing time



- The node is marked as processed only when none of its outgoing arcs lead to an undiscovered vertex
- The DFS loop 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

Recap: DFS loop with finishing time

```
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)
```

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

- General idea:

Step 1

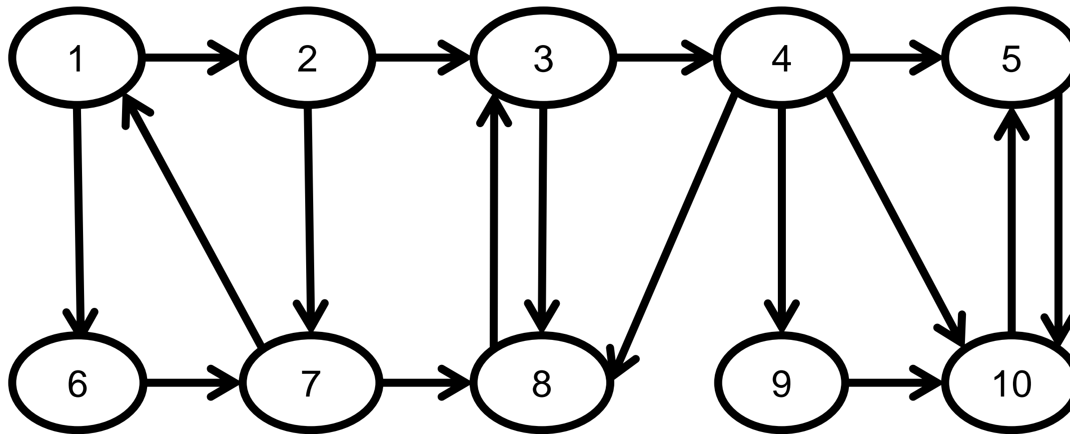
- Run DFS-loop on G^T and compute finishing time for each vertex

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

Step 2

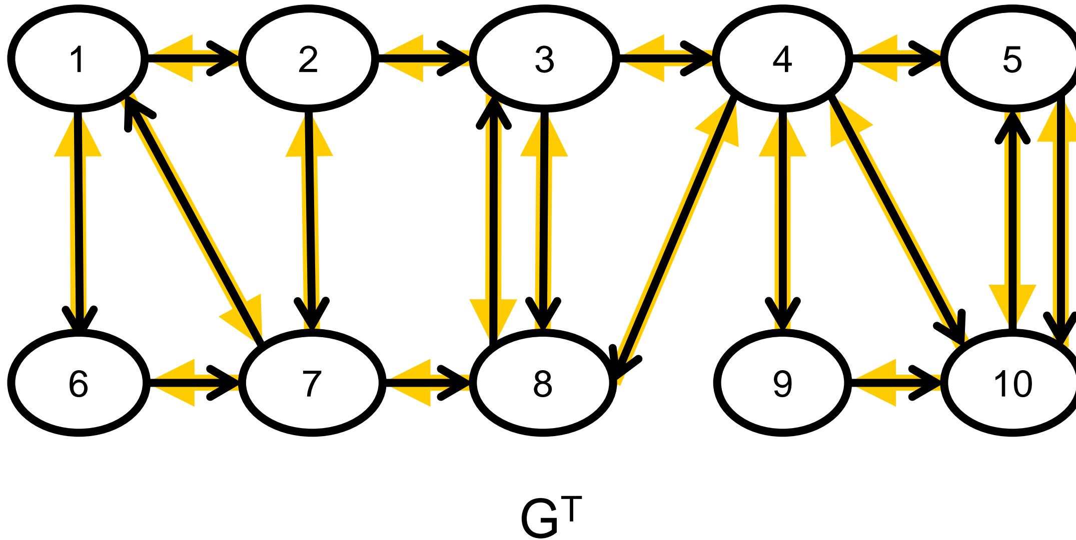
- Run DFS-loop on G, processing nodes in reverse order of their finishing times
- Each time we collected all nodes reachable from v, we discovered a new SCC

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



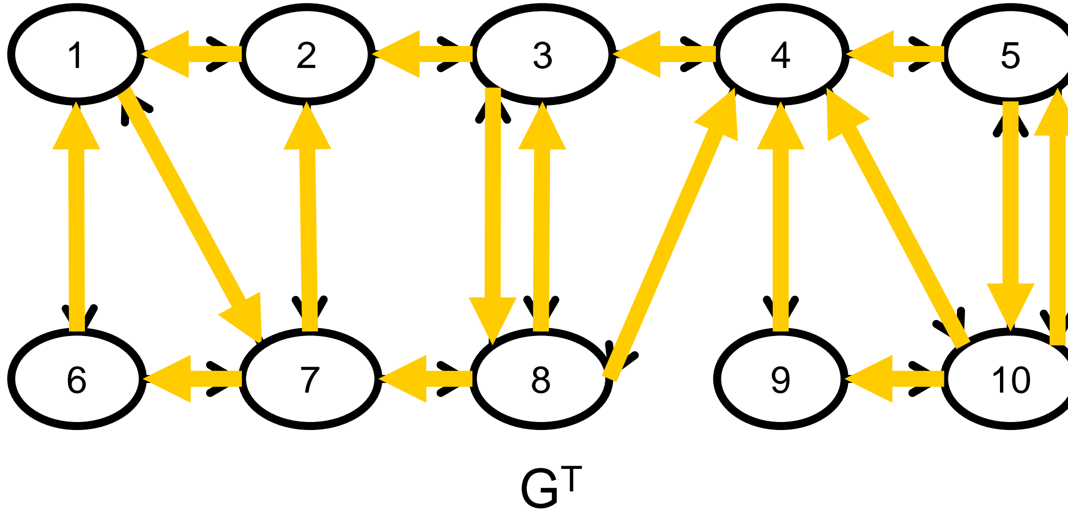
G

First, generate G^T



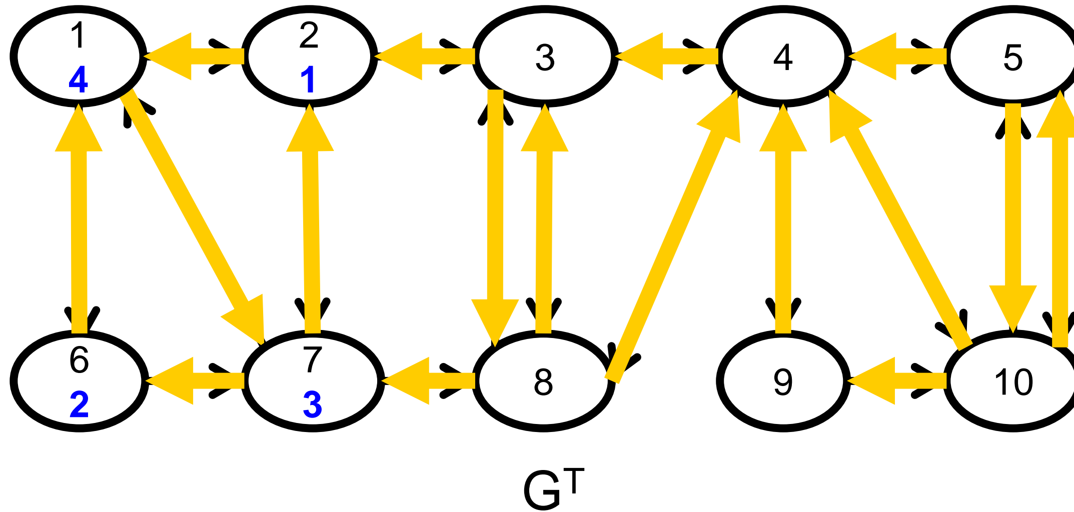
- Reverse every edge of G

Perform DFS loop 1 on G^T



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

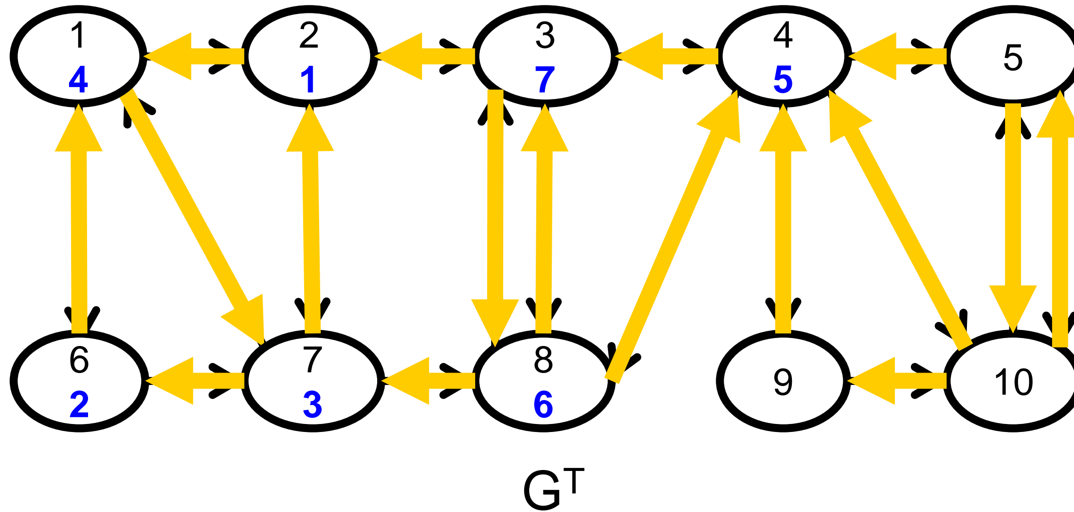
Computing finishing times in G^T



i	v
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

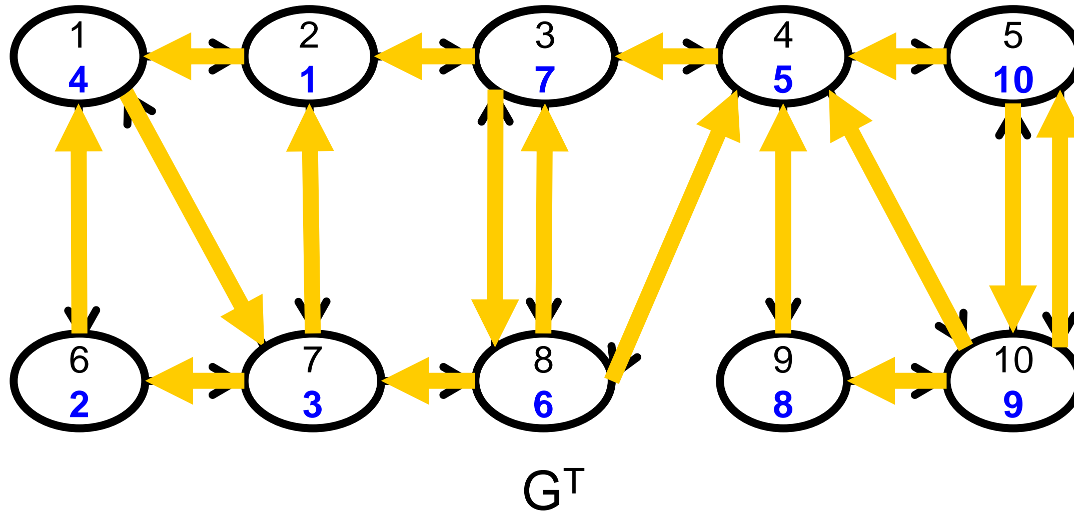
Computing finishing times in G^T



i	v
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

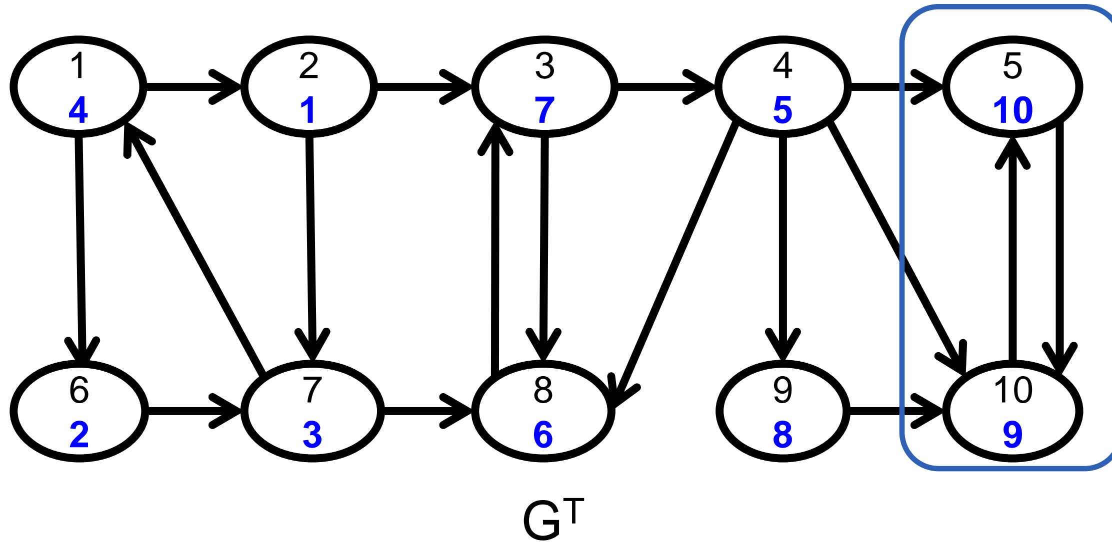
Computing finishing times in G^T



i	v
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

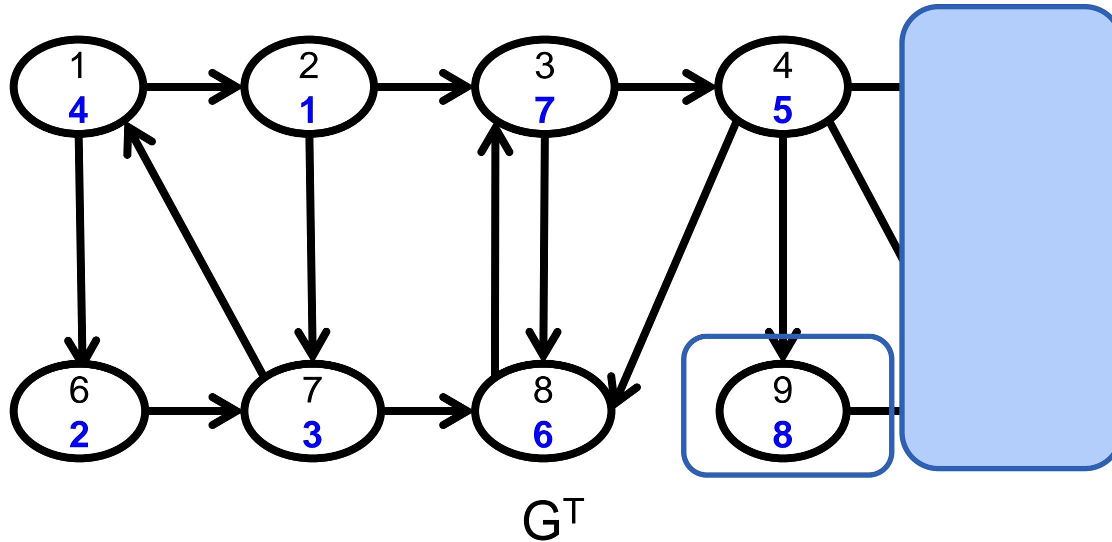


i	v
1	node 2
2	node 6
3	node 7
4	node 1
5	node 4
6	node 8
7	node 3
8	node 9
9	node 10
10	node 5

Leader: node 5

SCC1: {5,10}

DFS loop 2
Traversing G in reverse order of finishing times

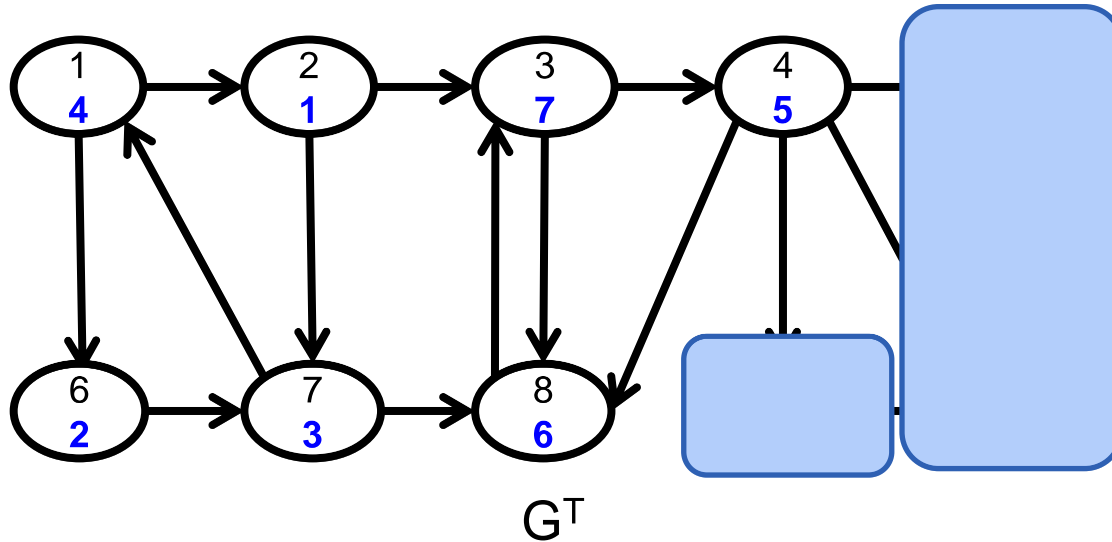


i	v
1	node 2
2	node 6
3	node 7
4	node 1
5	node 4
6	node 8
7	node 3
8	node 9
9	node 10
10	node 5

Leader: node 9

SCC2: {9}

DFS loop 2
Traversing G in reverse order of finishing times



i	v
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
```

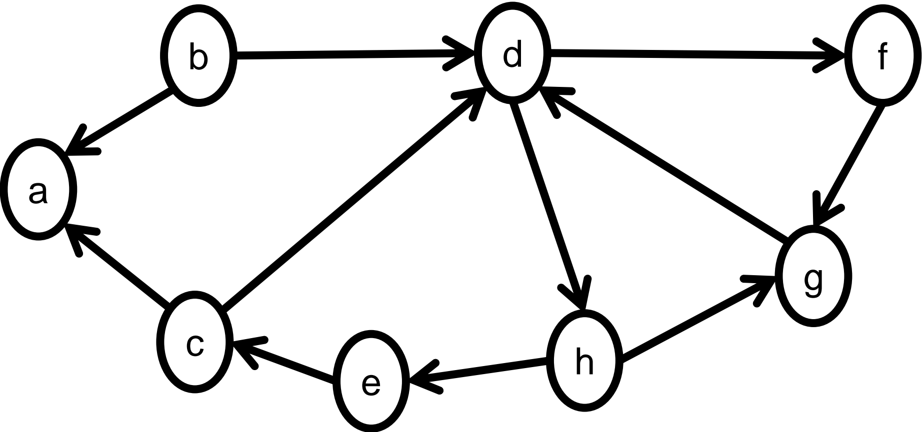
```
Algorithm DFS2(G, current)

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

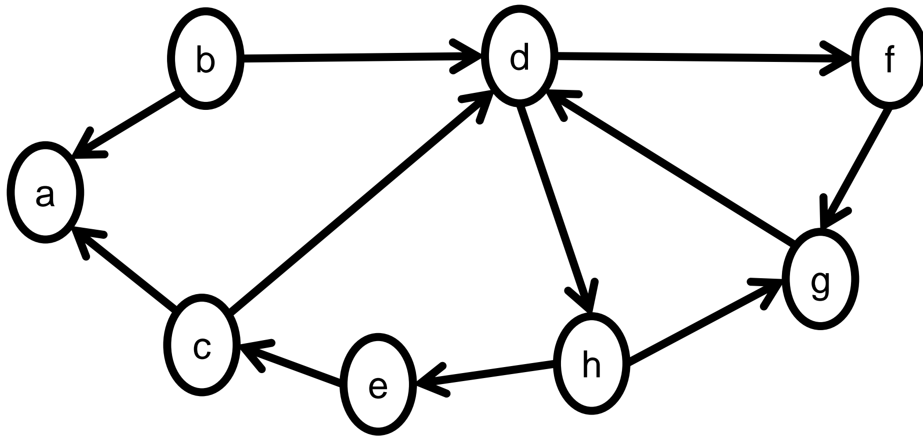
```
Algorithm DFS_loop2(G, F)

    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 10

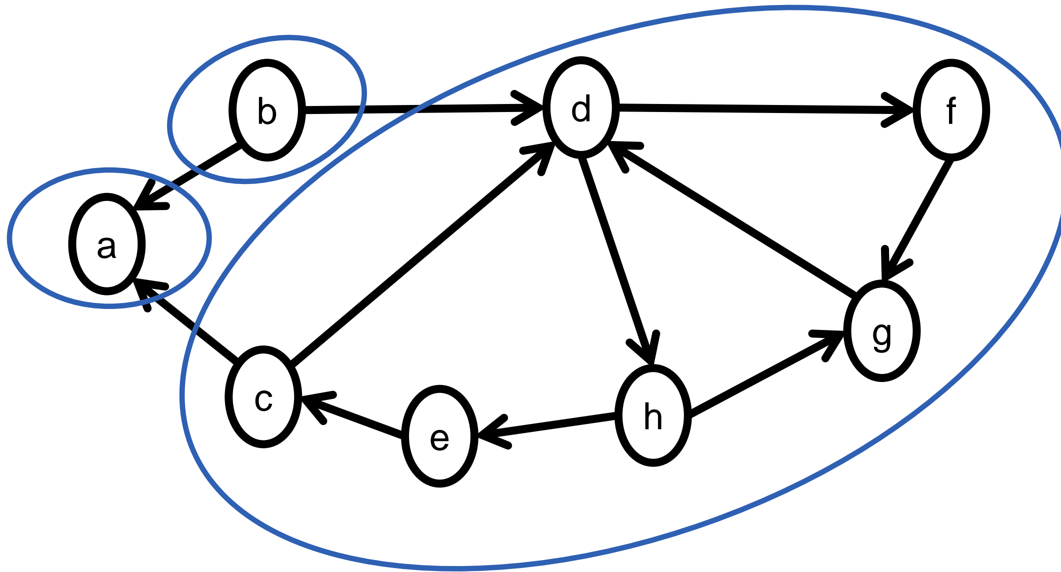


Finishing times in G^T after DFS loop 1



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}

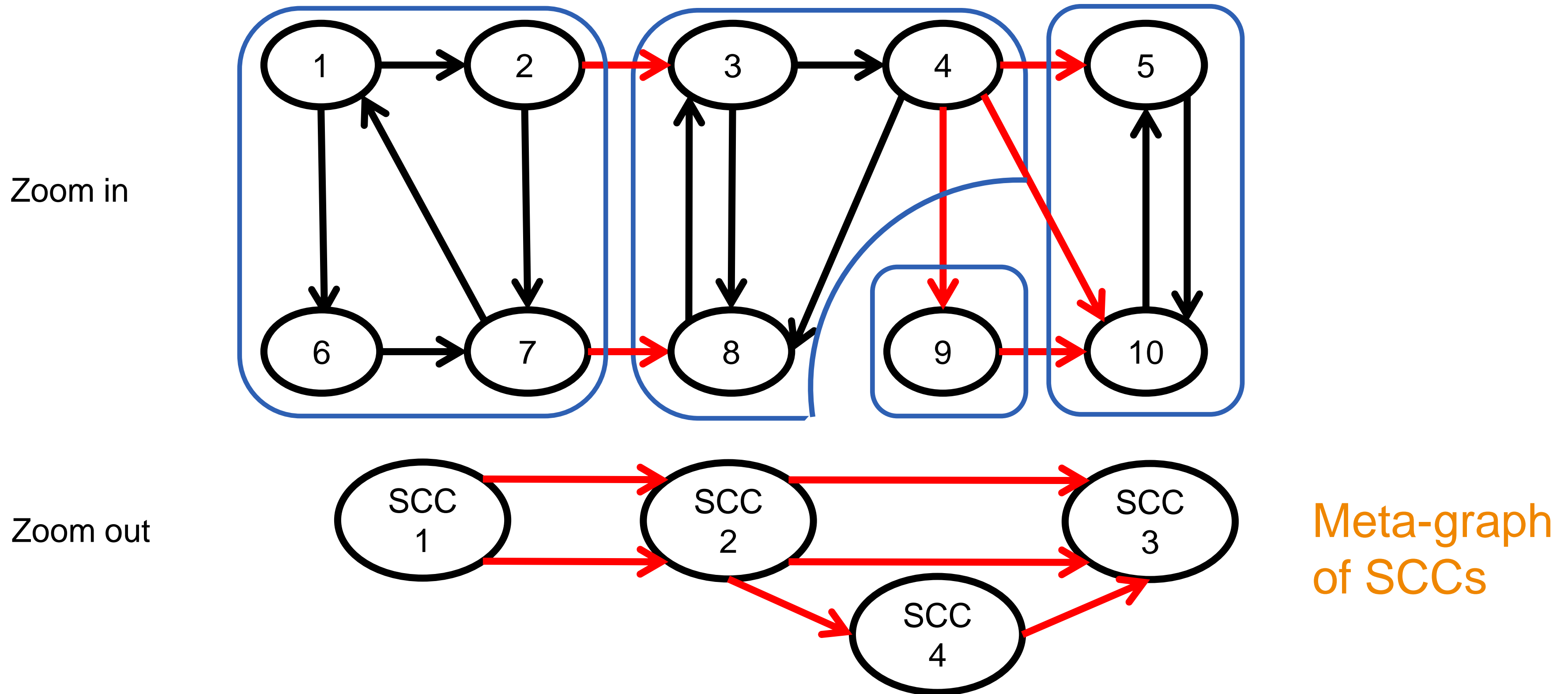
We run several examples of 2P-SCC and obtained the desired SCCs

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?

Why is the 2P-SCC algorithm correct?

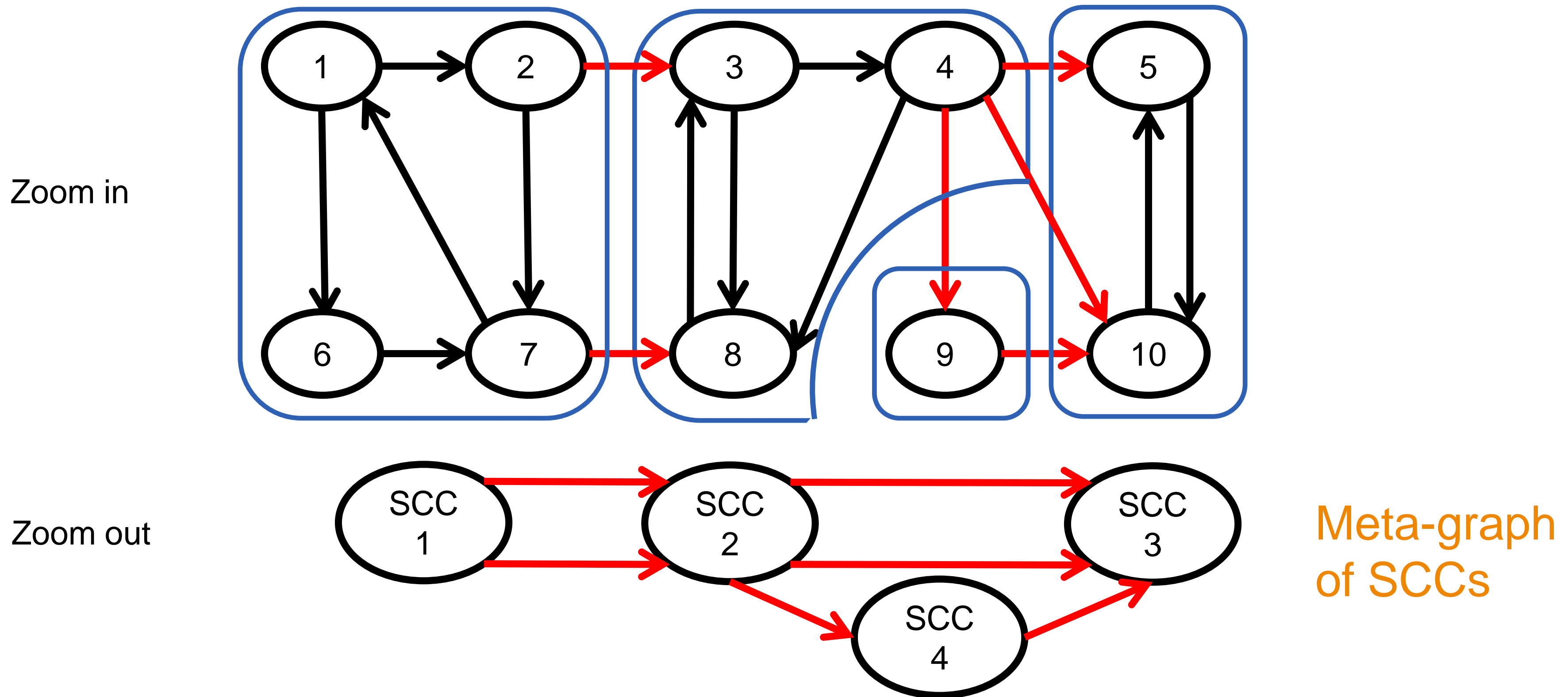
Meta-graph of SCCs



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

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

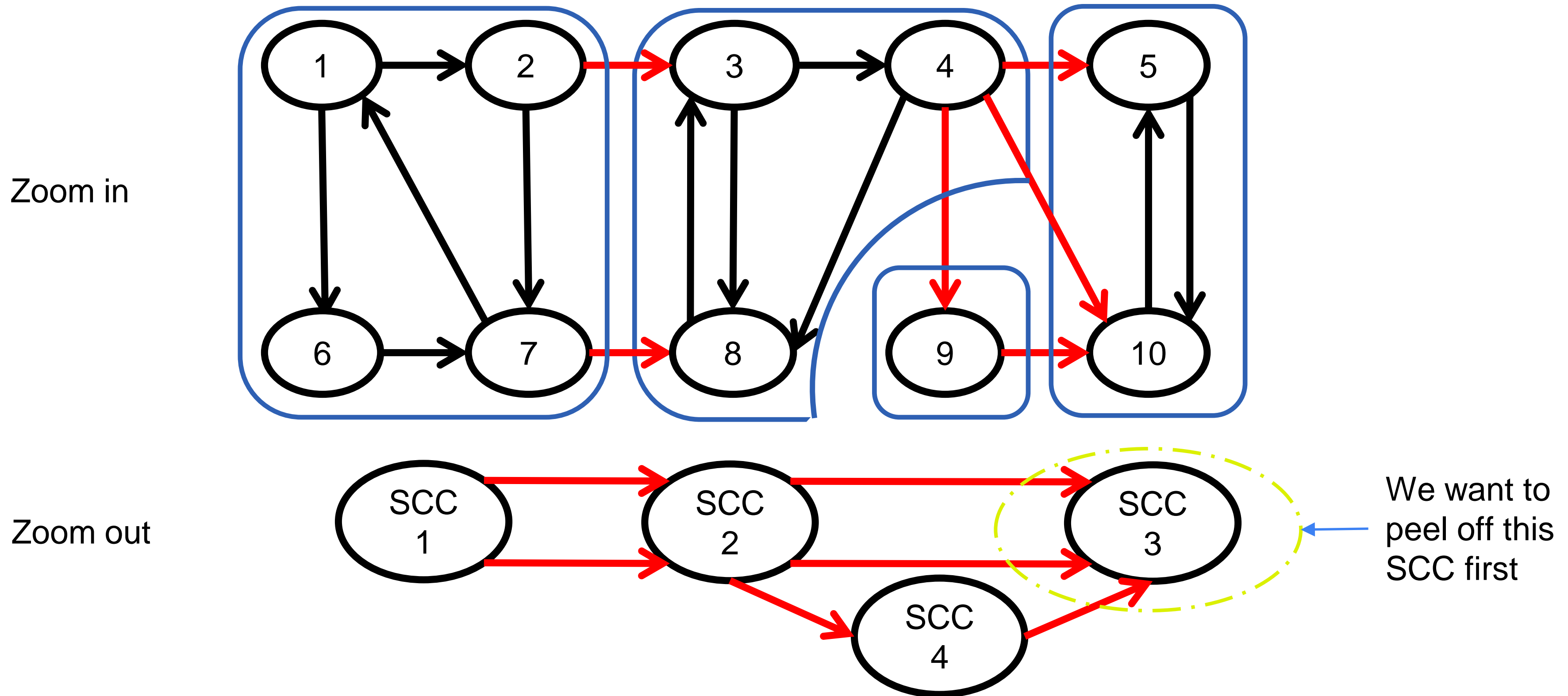
Meta-graph of SCCs



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

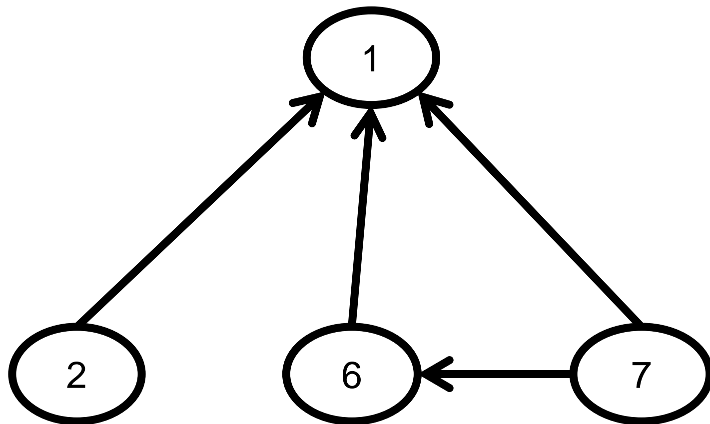
The desired order of SCC discovery



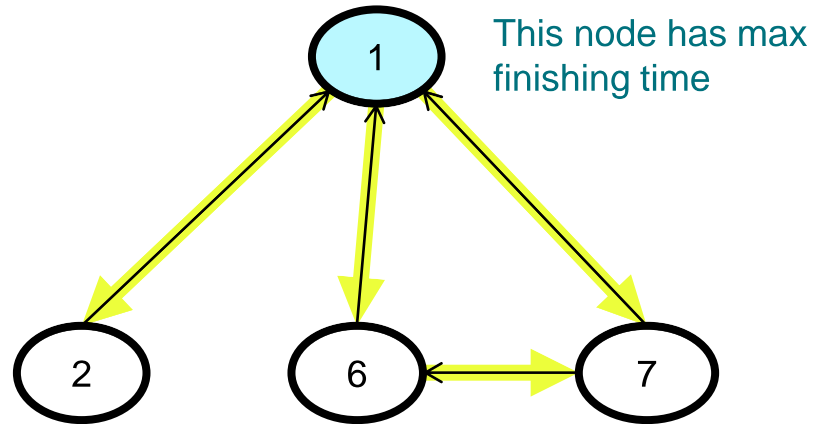
- Meta-graph is always a DAG → it has Topological Ordering. In particular, we have (at least one) sink SCC – an SCC which does not have any outgoing arcs in the meta-graph
- We should start DFS loop 2 with this sink SCC. After we mark all its nodes as processed, we essentially remove this SCC from further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

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

What does DFS loop 1 discover?



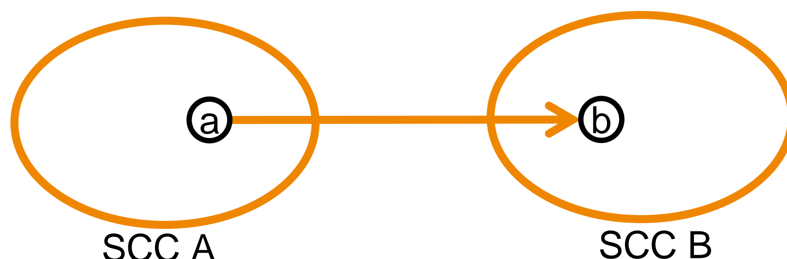
Graph G



Graph G^T

- When we do DFS on G^T with the edges reversed, the max finishing time will be always in the sink vertex – the vertex to which we found a path from all other vertices
- This is the vertex where we start DFS 1, and discover all two-way paths in a given SCC.

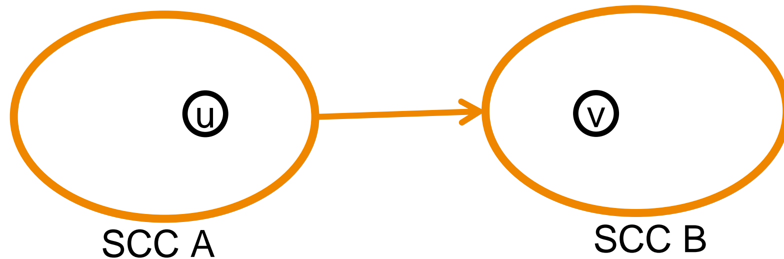
Max finishing time after DFS loop 1 is in the sink SCC

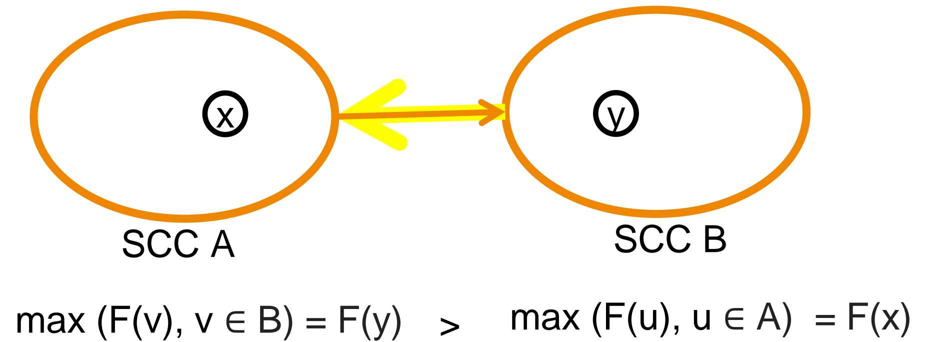


- Consider two adjacent SCC nodes A and B in a meta-graph.
- The nodes may have a one-way arc, from a to b, say
- Because there is a complete two-way reachability between all nodes inside each SCC, we have a path to any vertex in B from any vertex in A, but not in the opposite direction

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

Theorem

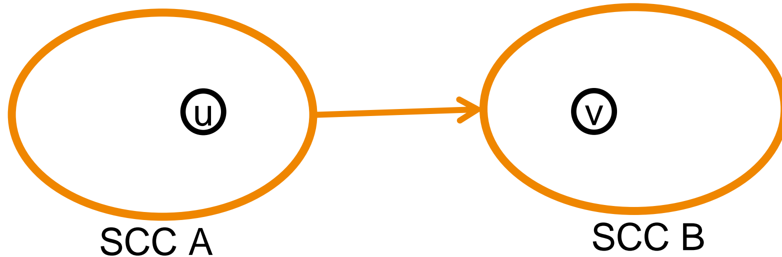


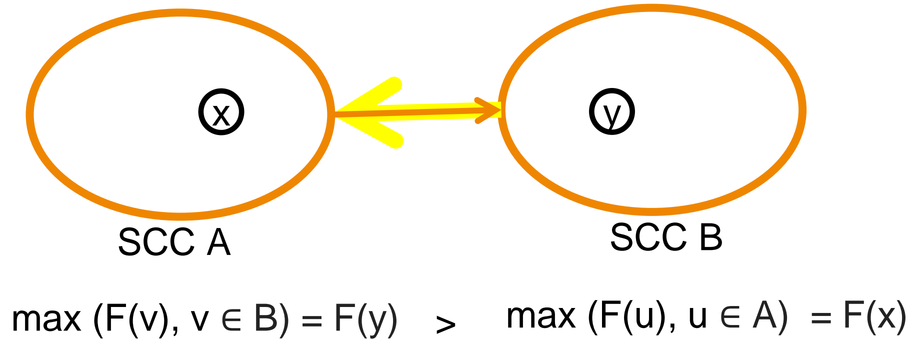


Given two adjacent SCC components in a meta-graph of all SCCs of graph G, and the finishing times F of all vertices obtained by DFS loop on G^T, the maximum finishing time among all nodes in the sink SCC will be greater than the maximum finishing time among all nodes in the source SCC:

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

Proof





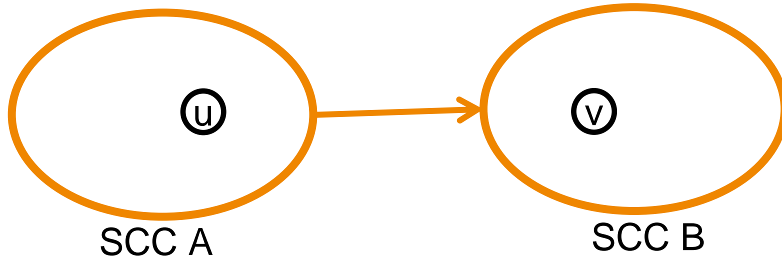
G

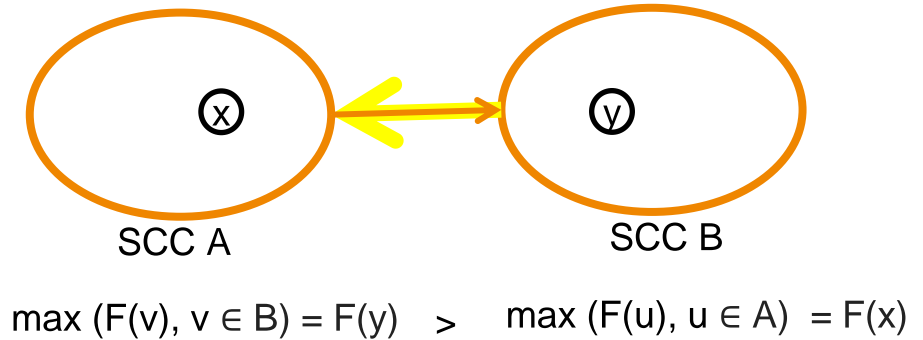
G^T

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

- 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(y) > F(x), no matter of the order in which we perform the DFS loop 1

Proof





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

G $\quad\quad$ G^T

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

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

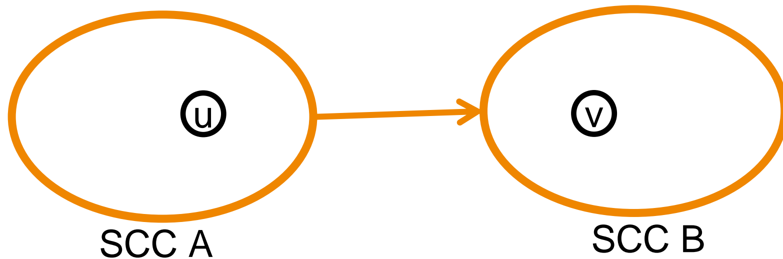
- Case 1. vertex y was picked before vertex x in DFS loop 1

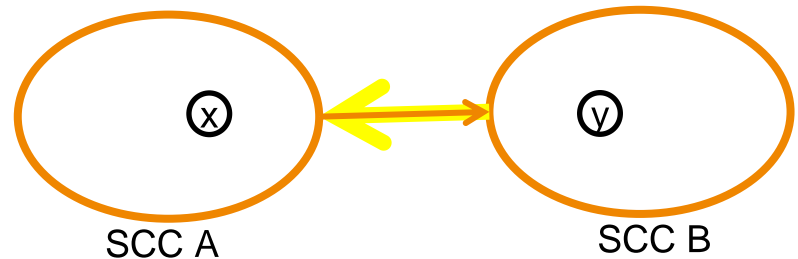
Because there is a path from y to x in G^T and because x has not been discovered yet, the DFS will discover and process node x before node y: y would need to wait until all undiscovered nodes reachable from it have been processed. Thus in this case F(y) > F(x)

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

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

Correctness of 2P-SCC algorithm (sketch)



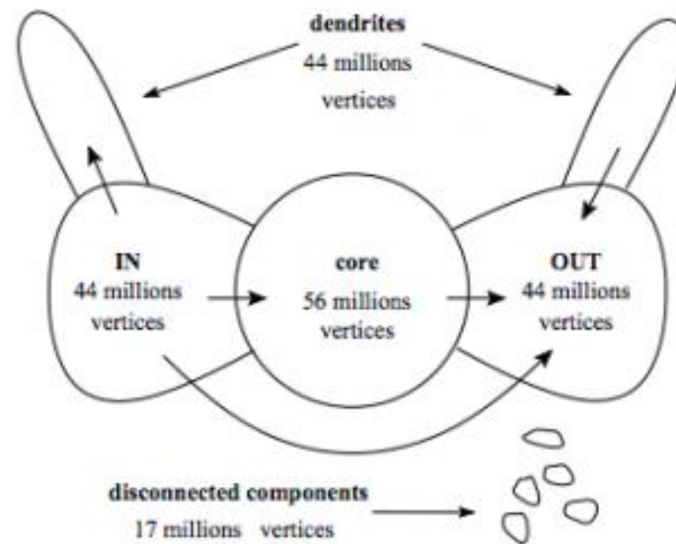


$\max(F(v), v \in B) = F(y) > \max(F(u), u \in A) = F(x)$

G $\qquad$ G^T

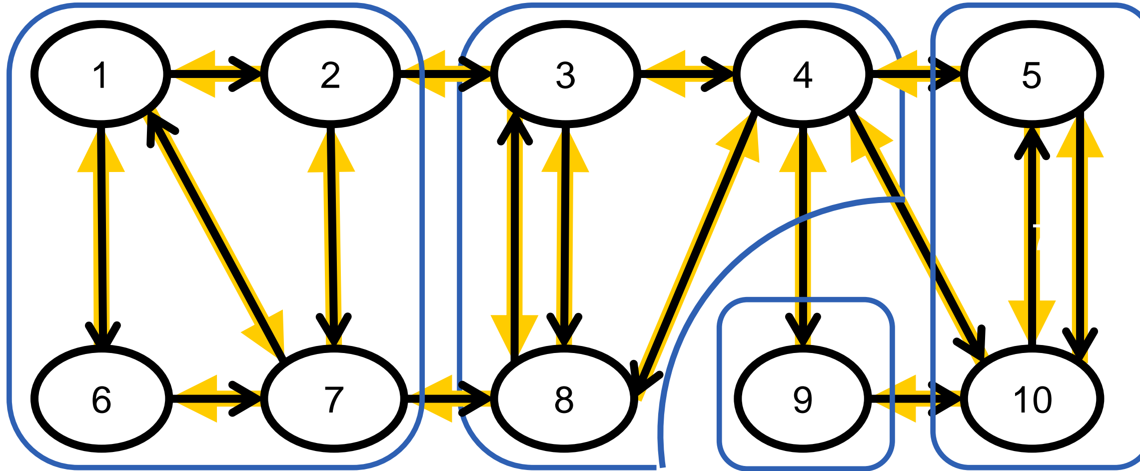
- Because for a pair of adjacent SCCs F(y) > F(x), if we apply the result of our theorem to all pairs of adjacent SCCs, the max finishing time will be in the sink SCC
- In DFS loop 2 we pick the node with max finishing time, and this node is guaranteed to be in the sink SCC. We collect all the nodes 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 and peel off each sink SCC of G: one-by-one

SCC application: Bowtie Structure of the Web



Read here

Question



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

Can you prove this for a general directed graph?