CMPSC 465: LECTURE XIII

Strongly Connected Components

Ke Chen

September 29, 2025

Recall from last week ...

On a directed graph:

Recall from last week ...

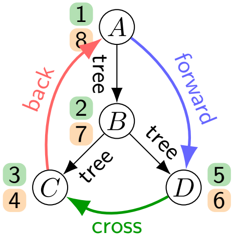
On a directed graph:

- The same DFS works.

Recall from last week ...

On a directed graph:

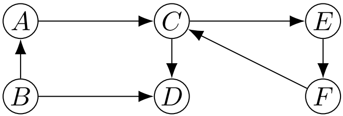
- The same DFS works.
- Four types of edges with respect to a particular DFS run.



Recall from last week ...

On a directed graph:

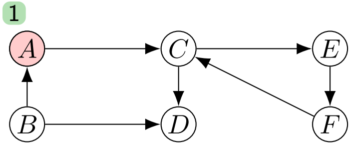
- The same DFS works.
- Four types of edges with respect to a particular DFS run.
- Can use the pre- and post-numbers to detect cycles or do topological sort if the input is a DAG, in $O(|V| + |E|)$ time.



Recall from last week ...

On a directed graph:

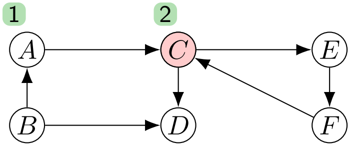
- The same DFS works.
- Four types of edges with respect to a particular DFS run.
- Can use the pre- and post-numbers to detect cycles or do topological sort if the input is a DAG, in $O(|V| + |E|)$ time.



Recall from last week ...

On a directed graph:

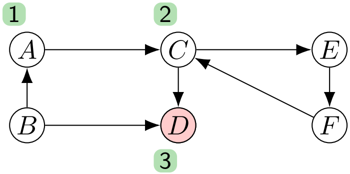
- The same DFS works.
- Four types of edges with respect to a particular DFS run.
- Can use the pre- and post-numbers to detect cycles or do topological sort if the input is a DAG, in $O(|V| + |E|)$ time.



Recall from last week ...

On a directed graph:

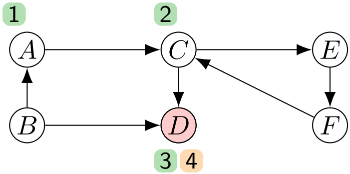
- The same DFS works.
- Four types of edges with respect to a particular DFS run.
- Can use the pre- and post-numbers to detect cycles or do topological sort if the input is a DAG, in $O(|V| + |E|)$ time.



Recall from last week ...

On a directed graph:

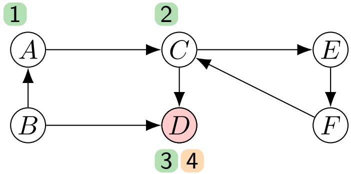
- The same DFS works.
- Four types of edges with respect to a particular DFS run.
- Can use the pre- and post-numbers to detect cycles or do topological sort if the input is a DAG, in $O(|V| + |E|)$ time.



Recall from last week ...

On a directed graph:

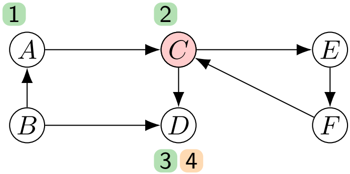
- The same DFS works.
- Four types of edges with respect to a particular DFS run.
- Can use the pre- and post-numbers to detect cycles or do topological sort if the input is a DAG, in $O(|V| + |E|)$ time.



Recall from last week ...

On a directed graph:

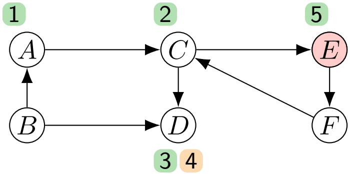
- The same DFS works.
- Four types of edges with respect to a particular DFS run.
- Can use the pre- and post-numbers to detect cycles or do topological sort if the input is a DAG, in $O(|V| + |E|)$ time.



Recall from last week ...

On a directed graph:

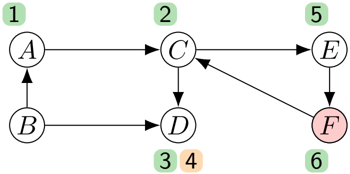
- The same DFS works.
- Four types of edges with respect to a particular DFS run.
- Can use the pre- and post-numbers to detect cycles or do topological sort if the input is a DAG, in $O(|V| + |E|)$ time.



Recall from last week ...

On a directed graph:

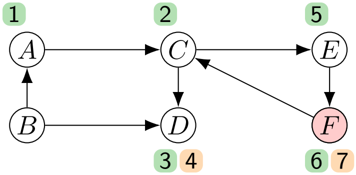
- The same DFS works.
- Four types of edges with respect to a particular DFS run.
- Can use the pre- and post-numbers to detect cycles or do topological sort if the input is a DAG, in $O(|V| + |E|)$ time.



Recall from last week ...

On a directed graph:

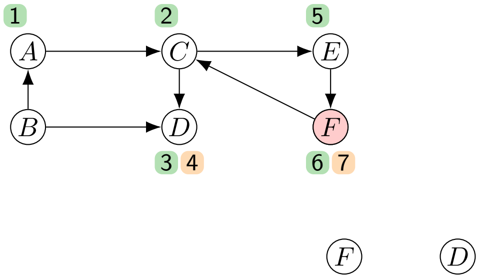
- The same DFS works.
- Four types of edges with respect to a particular DFS run.
- Can use the pre- and post-numbers to detect cycles or do topological sort if the input is a DAG, in $O(|V| + |E|)$ time.



Recall from last week ...

On a directed graph:

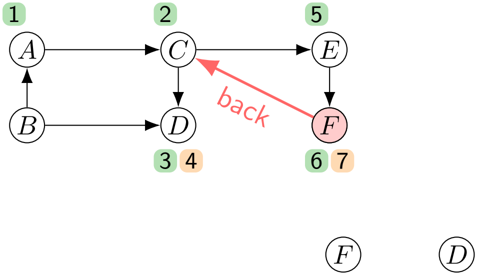
- The same DFS works.
- Four types of edges with respect to a particular DFS run.
- Can use the pre- and post-numbers to detect cycles or do topological sort if the input is a DAG, in $O(|V| + |E|)$ time.



Recall from last week ...

On a directed graph:

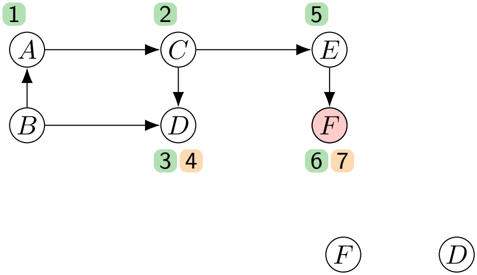
- The same DFS works.
- Four types of edges with respect to a particular DFS run.
- Can use the pre- and post-numbers to detect cycles or do topological sort if the input is a DAG, in $O(|V| + |E|)$ time.



Recall from last week ...

On a directed graph:

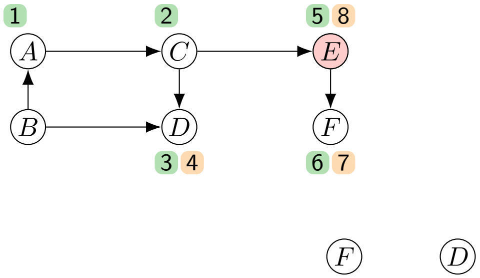
- The same DFS works.
- Four types of edges with respect to a particular DFS run.
- Can use the pre- and post-numbers to detect cycles or do topological sort if the input is a DAG, in $O(|V| + |E|)$ time.



Recall from last week ...

On a directed graph:

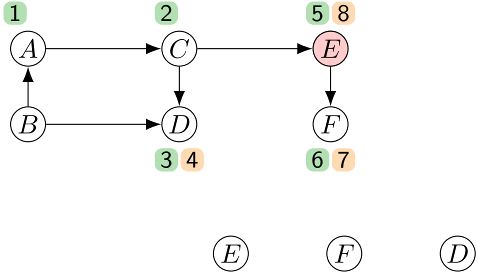
- The same DFS works.
- Four types of edges with respect to a particular DFS run.
- Can use the pre- and post-numbers to detect cycles or do topological sort if the input is a DAG, in $O(|V| + |E|)$ time.



Recall from last week ...

On a directed graph:

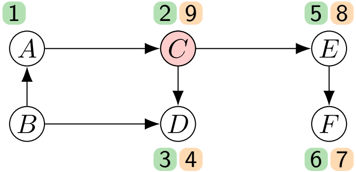
- The same DFS works.
- Four types of edges with respect to a particular DFS run.
- Can use the pre- and post-numbers to detect cycles or do topological sort if the input is a DAG, in $O(|V| + |E|)$ time.



Recall from last week ...

On a directed graph:

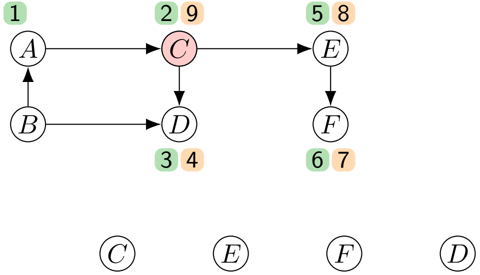
- The same DFS works.
- Four types of edges with respect to a particular DFS run.
- Can use the pre- and post-numbers to detect cycles or do topological sort if the input is a DAG, in $O(|V| + |E|)$ time.



Recall from last week ...

On a directed graph:

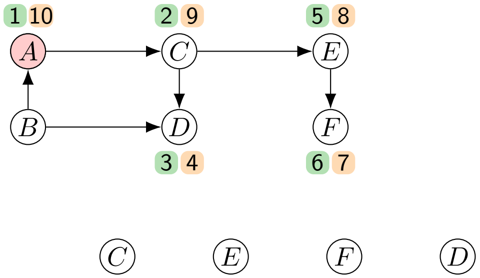
- The same DFS works.
- Four types of edges with respect to a particular DFS run.
- Can use the pre- and post-numbers to detect cycles or do topological sort if the input is a DAG, in $O(|V| + |E|)$ time.



Recall from last week ...

On a directed graph:

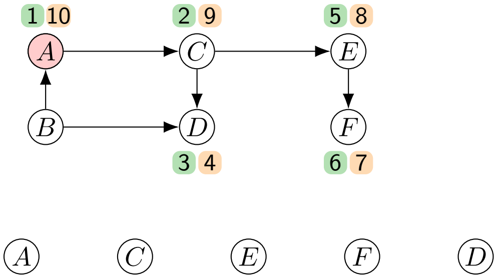
- The same DFS works.
- Four types of edges with respect to a particular DFS run.
- Can use the pre- and post-numbers to detect cycles or do topological sort if the input is a DAG, in $O(|V| + |E|)$ time.



Recall from last week ...

On a directed graph:

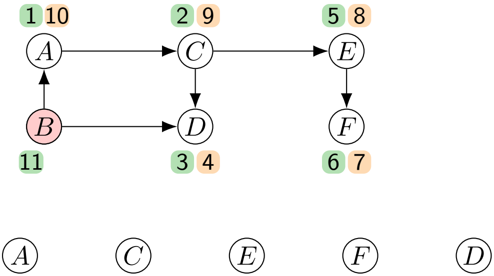
- The same DFS works.
- Four types of edges with respect to a particular DFS run.
- Can use the pre- and post-numbers to detect cycles or do topological sort if the input is a DAG, in $O(|V| + |E|)$ time.



Recall from last week ...

On a directed graph:

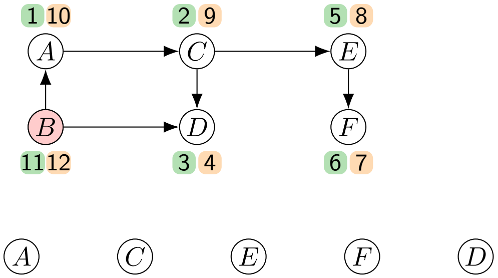
- The same DFS works.
- Four types of edges with respect to a particular DFS run.
- Can use the pre- and post-numbers to detect cycles or do topological sort if the input is a DAG, in $O(|V| + |E|)$ time.



Recall from last week ...

On a directed graph:

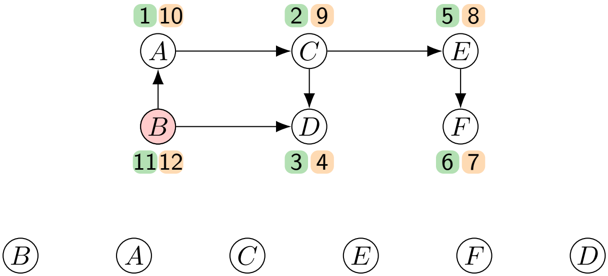
- The same DFS works.
- Four types of edges with respect to a particular DFS run.
- Can use the pre- and post-numbers to detect cycles or do topological sort if the input is a DAG, in $O(|V| + |E|)$ time.



Recall from last week ...

On a directed graph:

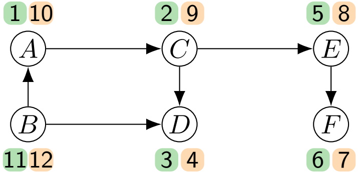
- The same DFS works.
- Four types of edges with respect to a particular DFS run.
- Can use the pre- and post-numbers to detect cycles or do topological sort if the input is a DAG, in $O(|V| + |E|)$ time.

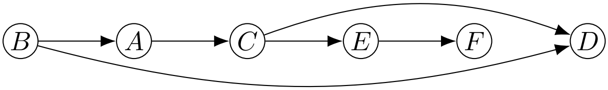


Recall from last week ...

On a directed graph:

- The same DFS works.
- Four types of edges with respect to a particular DFS run.
- Can use the pre- and post-numbers to detect cycles or do topological sort if the input is a DAG, in $O(|V| + |E|)$ time.

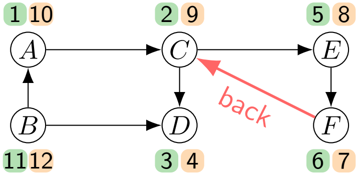


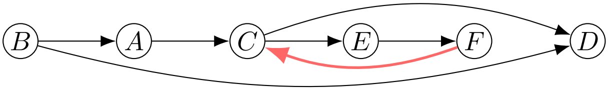


Recall from last week ...

On a directed graph:

- The same DFS works.
- Four types of edges with respect to a particular DFS run.
- Can use the pre- and post-numbers to detect cycles or do topological sort if the input is a DAG, in $O(|V| + |E|)$ time.





Connectivity in directed graphs

Consider $A \longrightarrow B \longrightarrow C$

Connectivity in directed graphs

Consider $A \rightarrow B \rightarrow C$

There is a path from A to C, but not the other way around.

Connectivity in directed graphs

Consider $\textcircled{A} \longrightarrow \textcircled{B} \longrightarrow \textcircled{C}$

There is a path from A to C, but not the other way around.

Definition In a directed graph G, we say vertices w and v are connected if there are paths from w to v and from v to w.

Connectivity in directed graphs

Consider $A \to B \to C$

There is a path from A to C, but not the other way around.

Definition In a directed graph G, we say vertices w and v are connected if there are paths from w to v and from v to w.

- This is an equivalance relation that divides the vertex set into disjoint subsets.

Connectivity in directed graphs

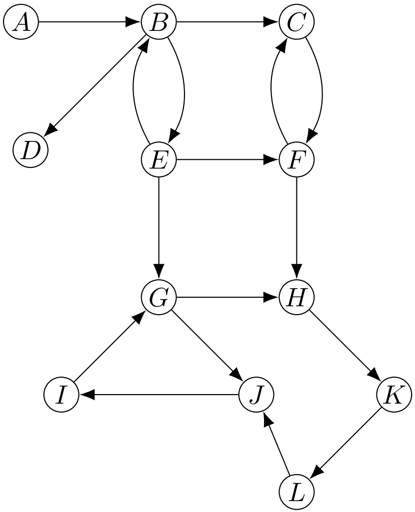
Consider $\textcircled{A} \to \textcircled{B} \to \textcircled{C}$

There is a path from A to C, but not the other way around.

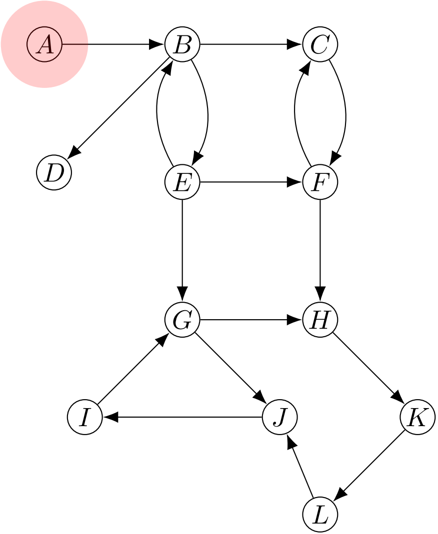
Definition In a directed graph G, we say vertices w and v are connected if there are paths from w to v and from v to w.

- This is an equivalance relation that divides the vertex set into disjoint subsets.
- Each subset is called a strongly connected component (SCC).

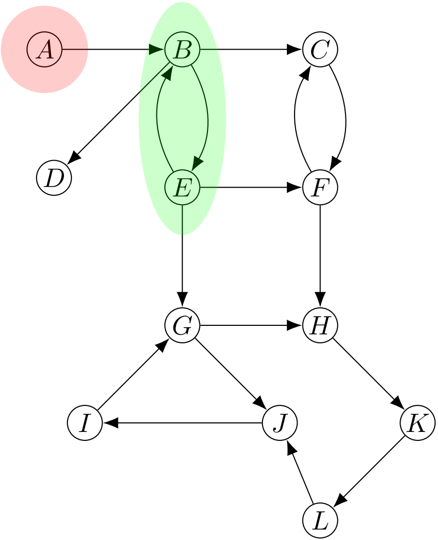
Strongly connected components (SCC's)



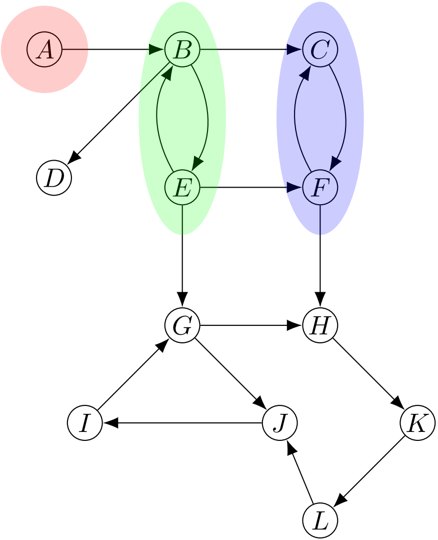
Strongly connected components (SCC's)



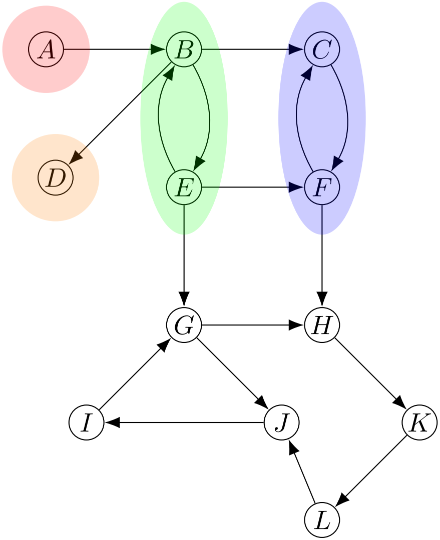
Strongly connected components (SCC's)



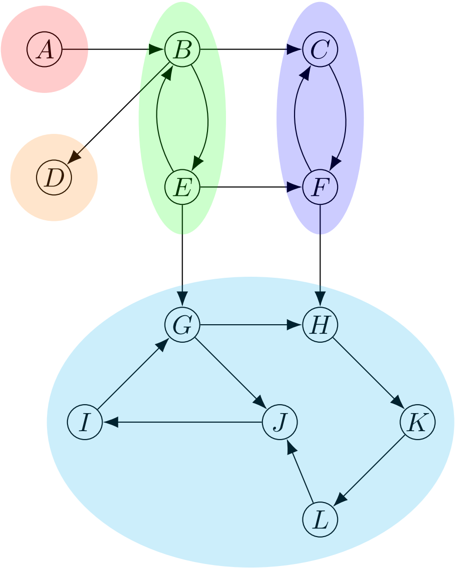
Strongly connected components (SCC's)



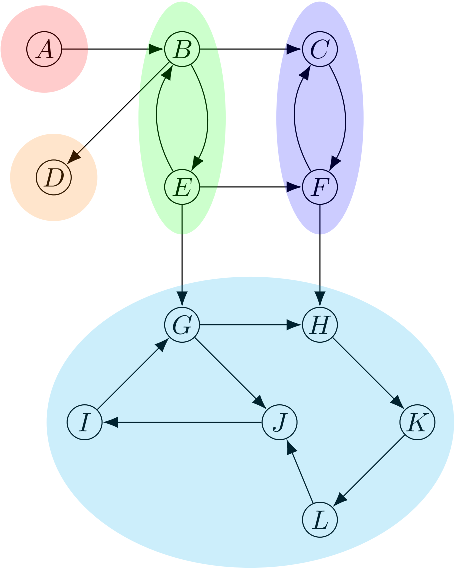
Strongly connected components (SCC's)



Strongly connected components (SCC's)

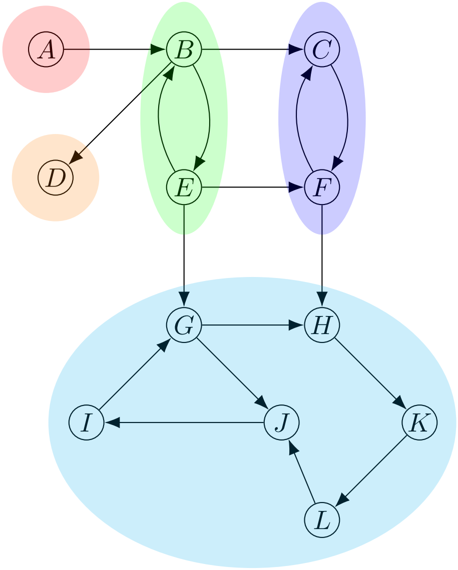


Strongly connected components (SCC's)



How to find all SCC's efficiently?

Strongly connected components (SCC's)



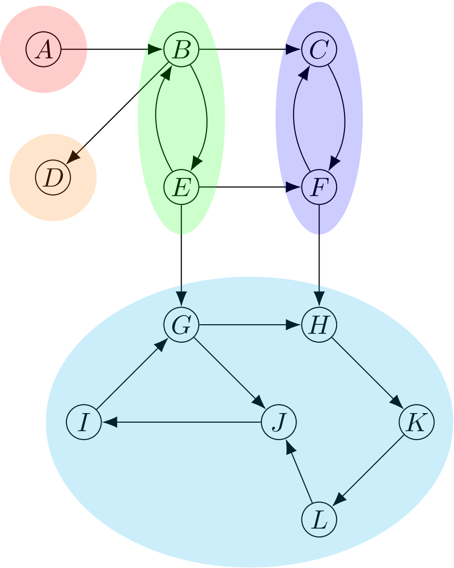
How to find all SCC's efficiently?

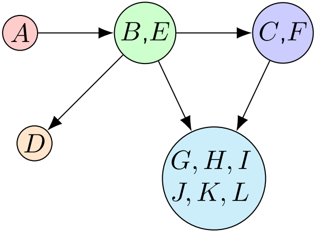
Let's try DFS!

Strongly connected components (SCC's)

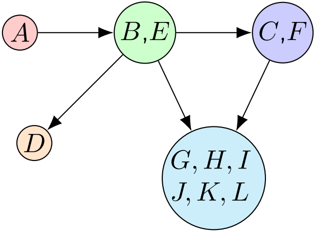
How to find all SCC's efficiently?

Let's try DFS!





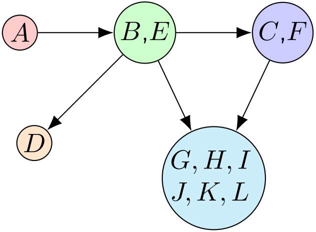
Meta graph



Each directed graph has a corresponding meta graph

- Vertices correspond to SCC's.
- Edges correspond to paths between SCC's.

Meta graph

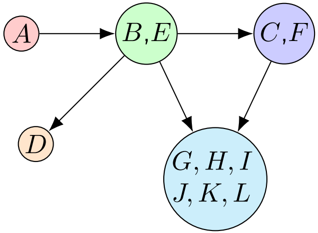


Each directed graph has a corresponding meta graph

- Vertices correspond to SCC's.
- Edges correspond to paths between SCC's.

Fact The meta graph is a DAG.

Meta graph



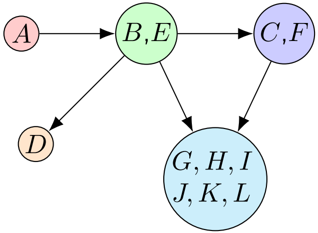
Each directed graph has a corresponding meta graph

- Vertices correspond to SCC's.
- Edges correspond to paths between SCC's.

Fact The meta graph is a DAG.

Proof. If there was a cycle in the meta graph, then the SCC's in the cycle would be merged together.

Meta graph

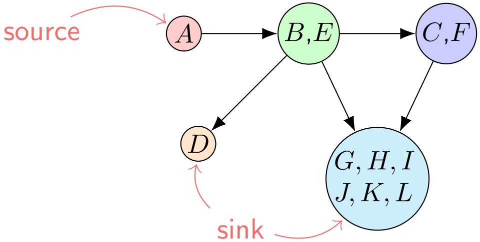


Each directed graph has a corresponding meta graph

- Vertices correspond to SCC's.
- Edges correspond to paths between SCC's.

Definition In a directed graph, a source is a vertex with no incoming edges; a sink is a vertex with no outgoing edges.

Meta graph

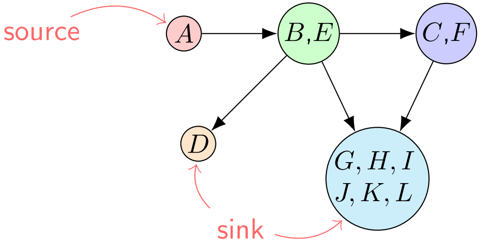


Each directed graph has a corresponding meta graph

- Vertices correspond to SCC's.
- Edges correspond to paths between SCC's.

Definition In a directed graph, a source is a vertex with no incoming edges; a sink is a vertex with no outgoing edges.

Meta graph

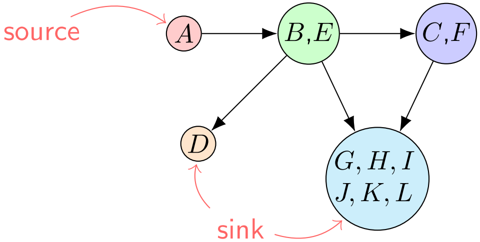


Each directed graph has a corresponding meta graph

- Vertices correspond to SCC's.
- Edges correspond to paths between SCC's.

Idea If we call Explore on a vertex from a sink SCC in the metagraph, it will discover exactly that SCC.

Meta graph



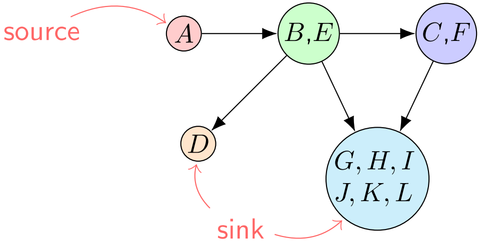
Each directed graph has a corresponding meta graph

- Vertices correspond to SCC's.
- Edges correspond to paths between SCC's.

Idea If we call Explore on a vertex from a sink SCC in the metagraph, it will discover exactly that SCC.

1. Find a vertex in a sink SCC.
2. Call Explore on it.
3. Remove the found SCC and repeat.

Meta graph



Each directed graph has a corresponding meta graph

- Vertices correspond to SCC's.
- Edges correspond to paths between SCC's.

Idea If we call Explore on a vertex from a sink SCC in the metagraph, it will discover exactly that SCC.

1. Find a vertex in a sink SCC. But how?
2. Call Explore on it.
3. Remove the found SCC and repeat.

Find a vertex in a sink SCC

Find a vertex in a sink SCC

Turns out to be not easy ...

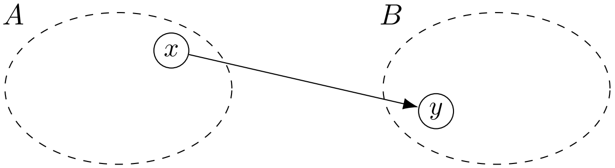
Find a vertex in a sink SCC

Turns out to be not easy ... How about a vertex in a source SCC?

Find a vertex in a sink SCC

Turns out to be not easy ... How about a vertex in a source SCC?

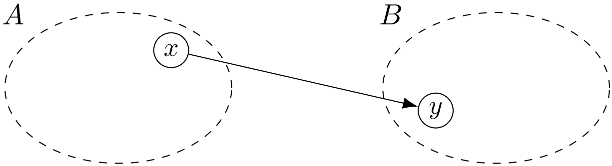
Fact Suppose A and B are two SCC's and there is an edge from a vertex in A to a vertex in B. Then, the vertex with the largest post number must be in ?.



Find a vertex in a sink SCC

Turns out to be not easy ... How about a vertex in a source SCC?

Fact Suppose A and B are two SCC's and there is an edge from a vertex in A to a vertex in B. Then, the vertex with the largest post number must be in ?.



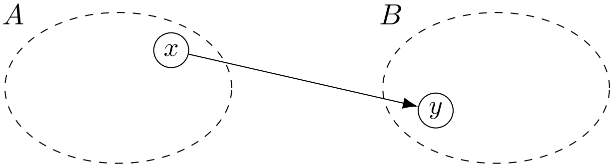
Proof. Among all vertices in $A \cup B$, if DFS first visits

- a node v in A,
- a node v in B.

Find a vertex in a sink SCC

Turns out to be not easy ... How about a vertex in a source SCC?

Fact Suppose A and B are two SCC's and there is an edge from a vertex in A to a vertex in B. Then, the vertex with the largest post number must be in ?.



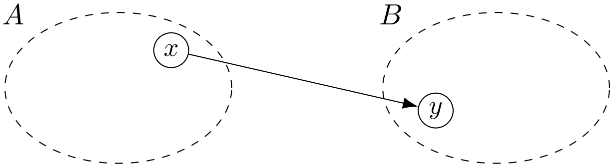
Proof. Among all vertices in $A \cup B$, if DFS first visits

- a node v in A, will finish all B before going back to v.
- a node v in B.

Find a vertex in a sink SCC

Turns out to be not easy ... How about a vertex in a source SCC?

Fact Suppose A and B are two SCC's and there is an edge from a vertex in A to a vertex in B. Then, the vertex with the largest post number must be in ?.



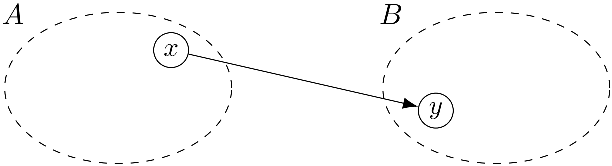
Proof. Among all vertices in $A \cup B$, if DFS first visits

- a node v in A, will finish all B before going back to v.
- a node v in B. will finish all B without visiting anything in A.

Find a vertex in a sink SCC

Turns out to be not easy ... How about a vertex in a source SCC?

Fact Suppose A and B are two SCC's and there is an edge from a vertex in A to a vertex in B. Then, the vertex with the largest post number must be in A.



Proof. Among all vertices in $A \cup B$, if DFS first visits

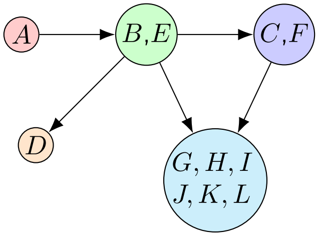
- a node v in A, will finish all B before going back to v.
- a node v in B. will finish all B without visiting anything in A.

Find a vertex in a sink SCC

Turns out to be not easy ... How about a vertex in a source SCC?

Fact Suppose A and B are two SCC's and there is an edge from a vertex in A to a vertex in B. Then, the vertex with the largest post number must be in A.

A key consequence of this fact is that the vertex with the largest post number must be in a source SCC.



Find a vertex in a sink SCC

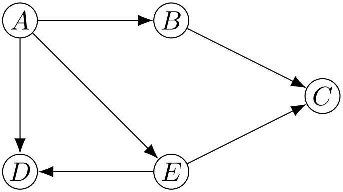
But we need a vertex from a sink SCC, not a source SCC.

Find a vertex in a sink SCC

But we need a vertex from a sink SCC, not a source SCC.

Idea Consider the reverse graph G^R.

G:

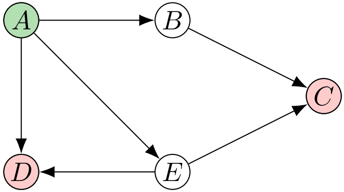


Find a vertex in a sink SCC

But we need a vertex from a sink SCC, not a source SCC.

Idea Consider the reverse graph G^R.

G:

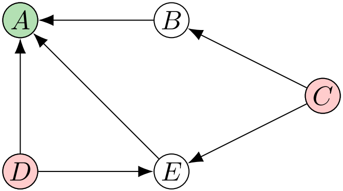


Find a vertex in a sink SCC

But we need a vertex from a sink SCC, not a source SCC.

Idea Consider the reverse graph G^R.

G^R:

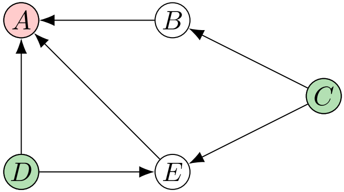


Find a vertex in a sink SCC

But we need a vertex from a sink SCC, not a source SCC.

Idea Consider the reverse graph G^R.

G^R:



Find a vertex in a sink SCC

But we need a vertex from a sink SCC, not a source SCC.

Facts about G^R

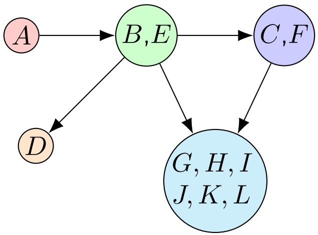
- G and G^R have the same SCC's.

Find a vertex in a sink SCC

But we need a vertex from a sink SCC, not a source SCC.

Facts about G^R

- G and G^R have the same SCC's.
- In the meta graph of G^R, every source SCC corresponds to a sink SCC in G.

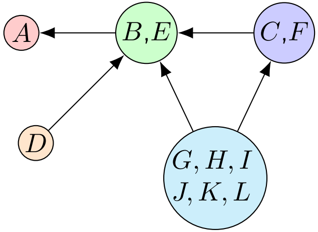


Find a vertex in a sink SCC

But we need a vertex from a sink SCC, not a source SCC.

Facts about G^R

- G and G^R have the same SCC's.
- In the meta graph of G^R, every source SCC corresponds to a sink SCC in G.

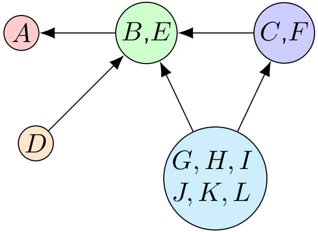


Find a vertex in a sink SCC

But we need a vertex from a sink SCC, not a source SCC.

Facts about G^R

- G and G^R have the same SCC's.
- In the meta graph of G^R, every source SCC corresponds to a sink SCC in G.

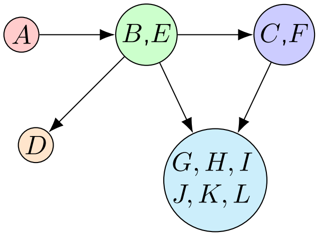


Therefore, if we run DFS on G^R and choose the node with the highest post number, it must be in a sink SCC of G!

Find SCC's

Recall our plan:

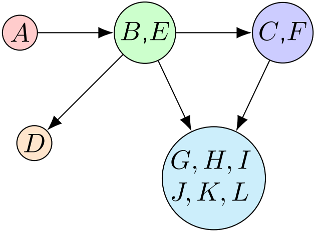
1. Find a vertex in a sink SCC.
2. Call Explore on it.
3. Remove the found SCC and repeat.



Find SCC's

Recall our plan:

1. Find a vertex in a sink SCC.
2. Call Explore on it.
3. Remove the found SCC and repeat.



Find SCC's

Input: Graph $G = (V, E)$ in adjacency list.

1. Build adjacency list for G^R.

2. Run DFS on G^R, assign pre/post numbers and output nodes sorted in descending order of post number. Let $v_1, v_2, \ldots, v_n$ be the ordering of the vertices.

3. Run DFS on G using the this ordering:

   // $visited$ is an array of size n filled with 0's
   $color = 1$
   for $i = 1$ **to** n **do**
       if $visited[i] == 0$ **then**
           Explore(G, v_i, $color$)
           $color = color + 1$

Find SCC's

Input: Graph $G = (V, E)$ in adjacency list.

1. Build adjacency list for G^R.

2. Run DFS on G^R, assign pre/post numbers and output nodes sorted in descending order of post number. Let $v_1, v_2, \ldots, v_n$ be the ordering of the vertices.

3. Run DFS on G using the this ordering:

   // $visited$ `is an array of size` n `filled with 0's`
   $color = 1$
   for $i = 1$ **to** n **do**
       if $visited[i] == 0$ **then**
           Explore(G, v_i, $color$)
           $color = color + 1$

Time complexity?

Find SCC's

Input: Graph $G = (V, E)$ in adjacency list.

1. Build adjacency list for G^R.
2. Run DFS on G^R, assign pre/post numbers and output nodes sorted in descending order of post number. Let $v_1, v_2, \ldots, v_n$ be the ordering of the vertices.
3. Run DFS on G using the this ordering:

```
// visited is an array of size n filled with 0's
color = 1
for i = 1 to n do
    if visited[i] == 0 then
        Explore(G, v_i, color)
        color = color + 1
```

Time complexity? Steps 1, 2, and 3 each takes $O(|V|+|E|)$, so overall $O(|V| + |E|)$.

Find SCC's

Input: Graph $G = (V, E)$ in adjacency list.

1. Build adjacency list for G^R.

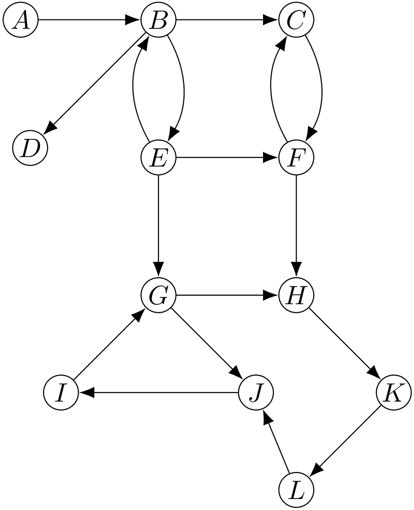
2. Run DFS on G^R, assign pre/post numbers and output nodes sorted in descending order of post number. Let $v_1, v_2, \ldots, v_n$ be the ordering of the vertices.

3. Run DFS on G using the this ordering:

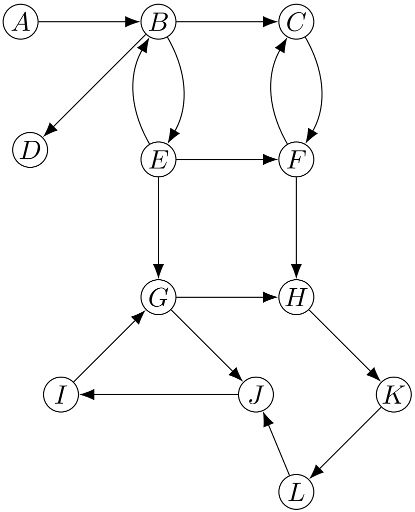
   `//` $visited$ `is an array of size` n `filled with 0's`
   $color = 1$
   for $i = 1$ **to** n **do**
       if $visited[i] == 0$ **then**
           Explore(G, v_i, $color$)
           $color = color + 1$

Final remark Note that we don't need to actually "remove" any SCC. Any node with a nonzero entry in $visited$ is no longer part of the graph for DFS.

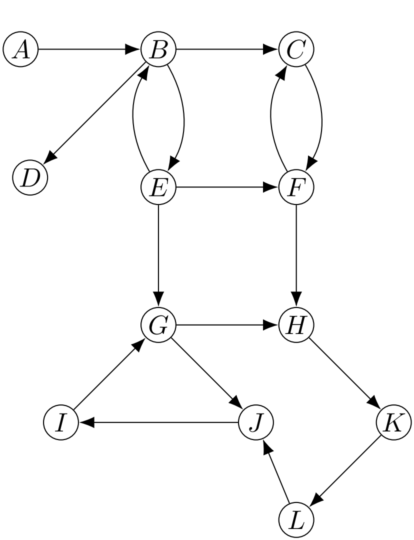
Find SCC's



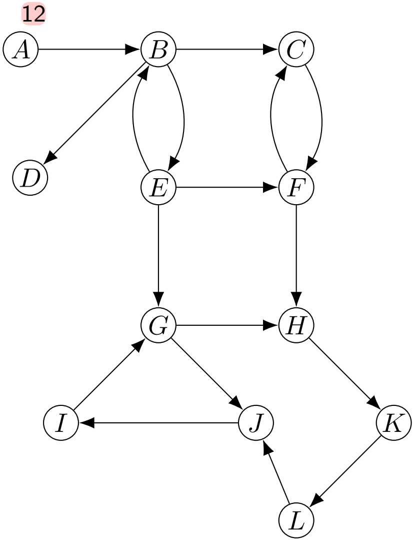
Find SCC's

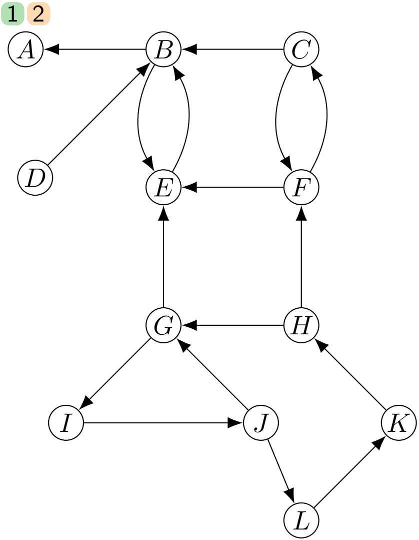


Find SCC's

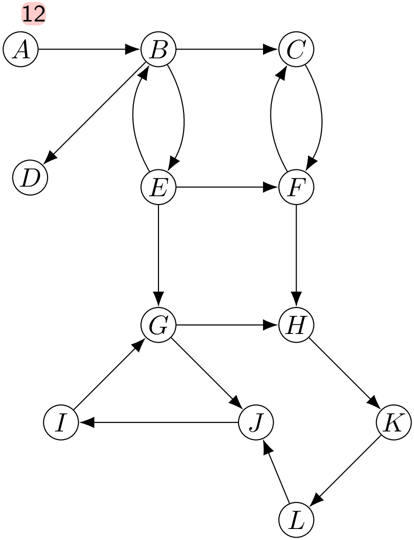


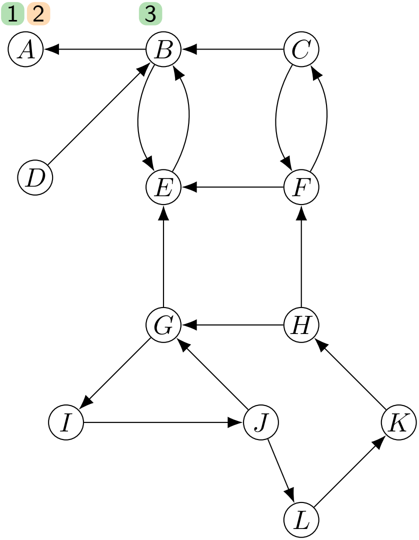
Find SCC's



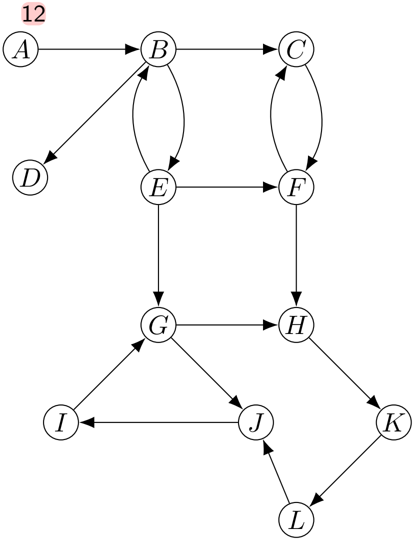


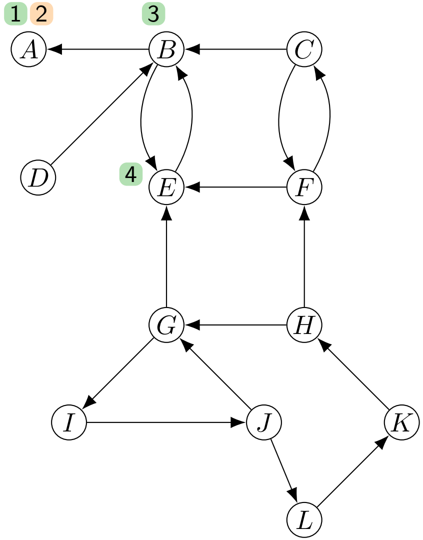
Find SCC's



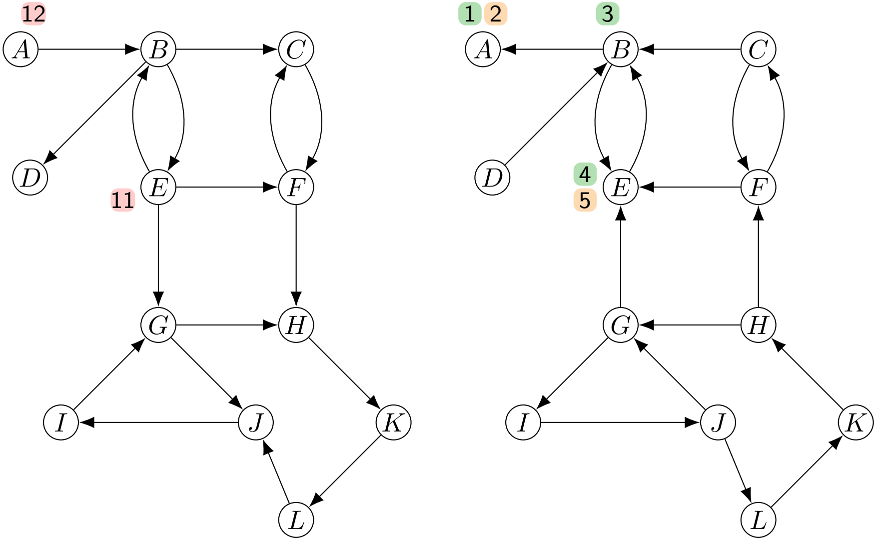


Find SCC's

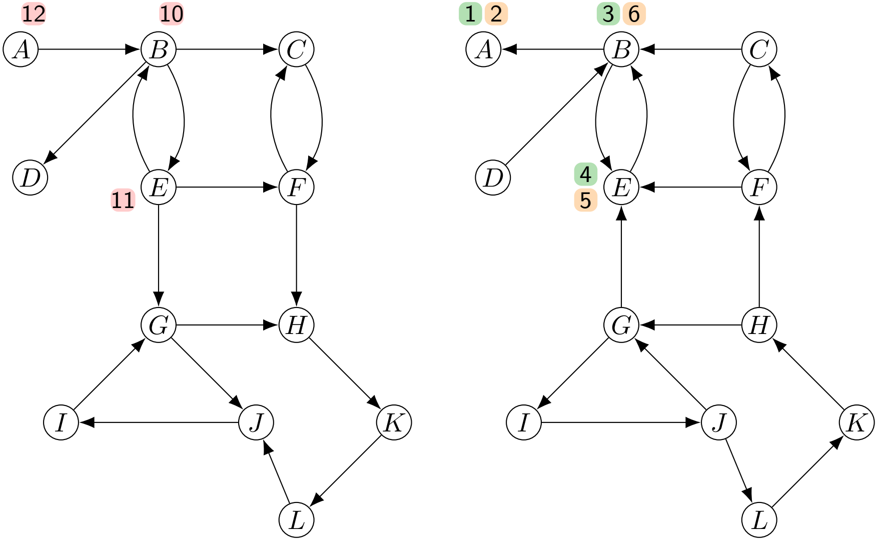




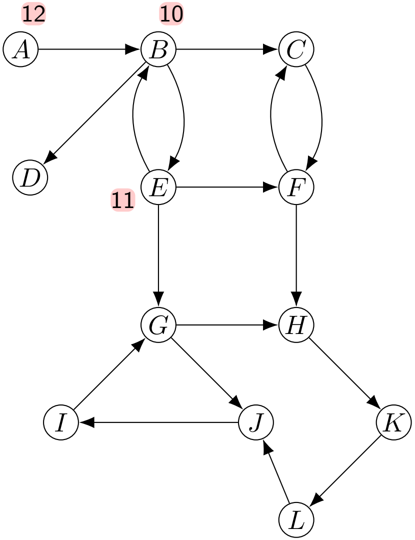
Find SCC's

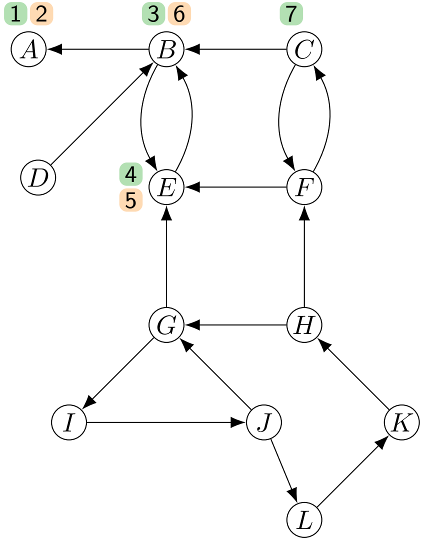


Find SCC's

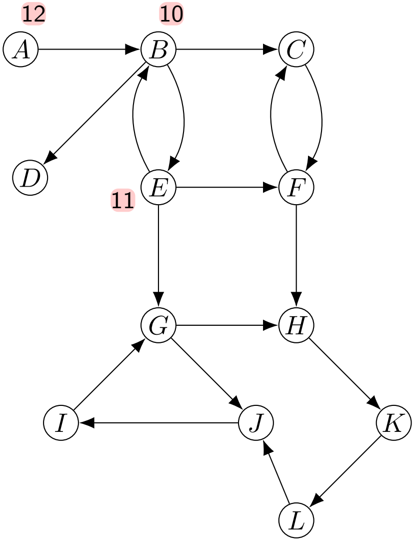


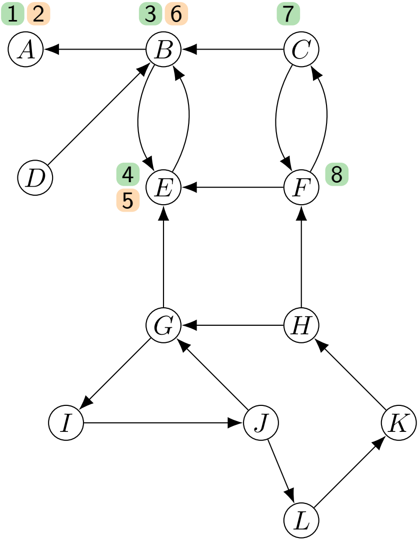
Find SCC's



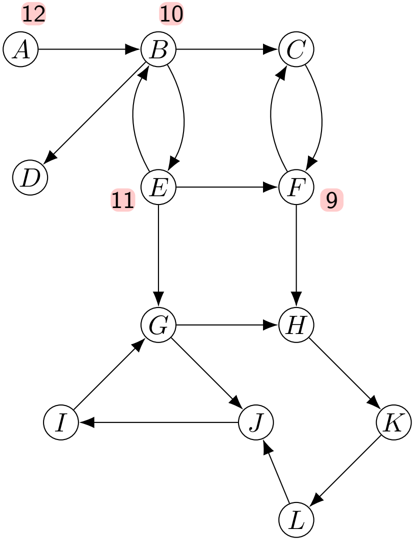


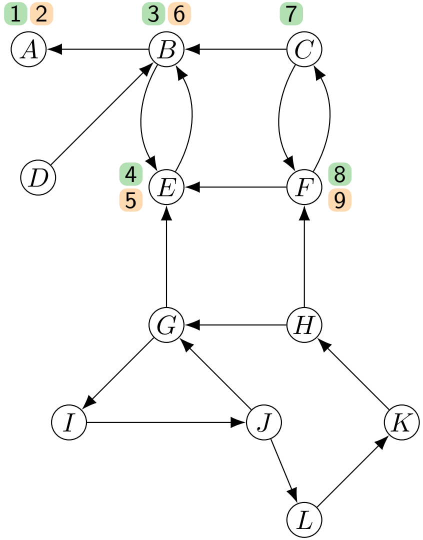
Find SCC's



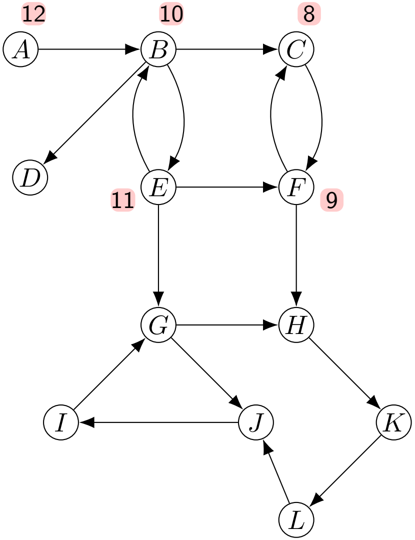


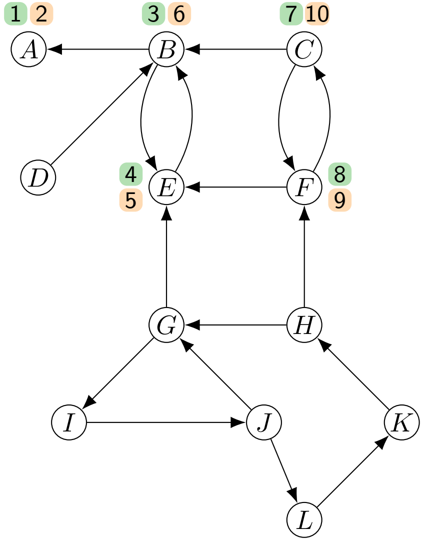
Find SCC's



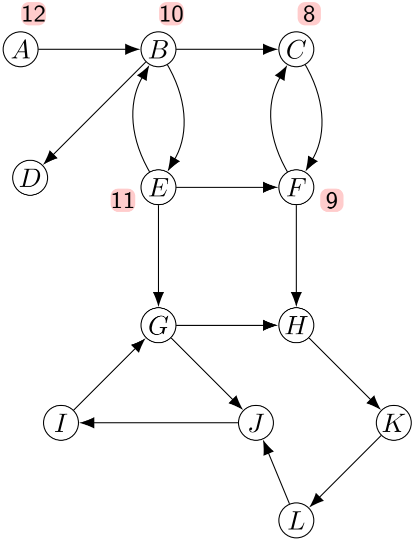


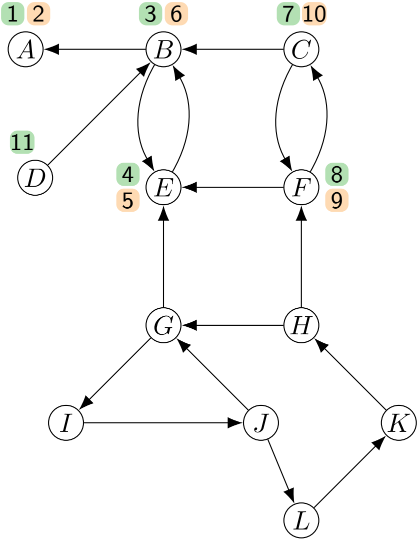
Find SCC's



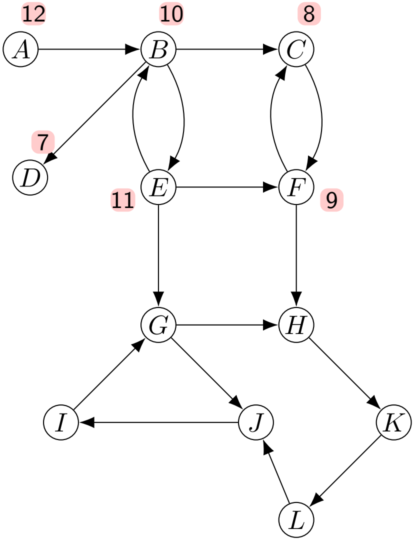


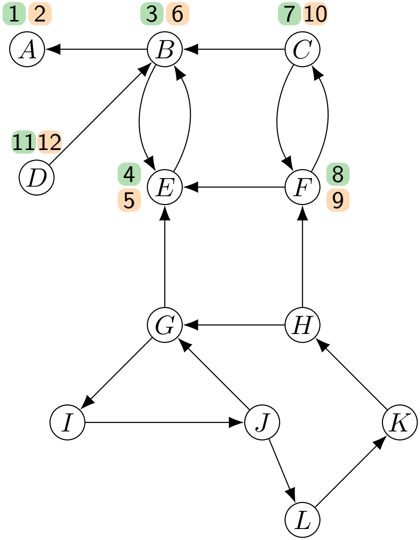
Find SCC's



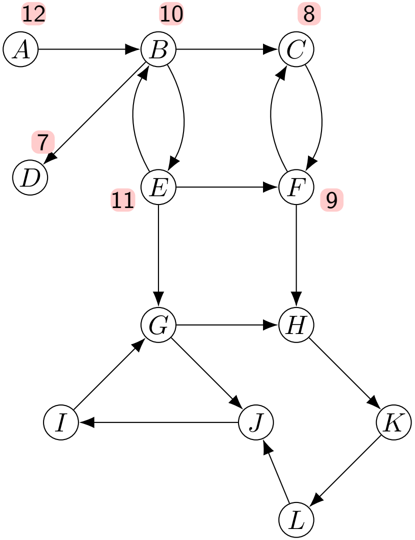


Find SCC's

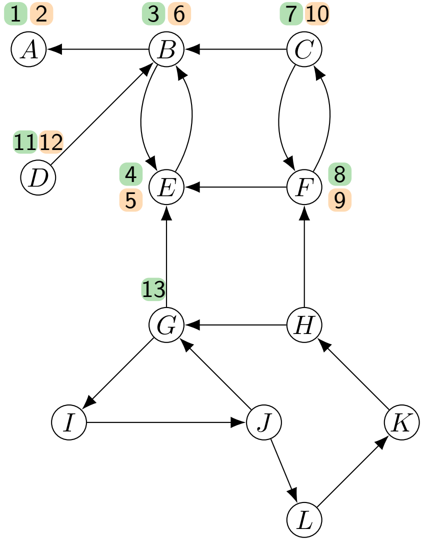




Find SCC's

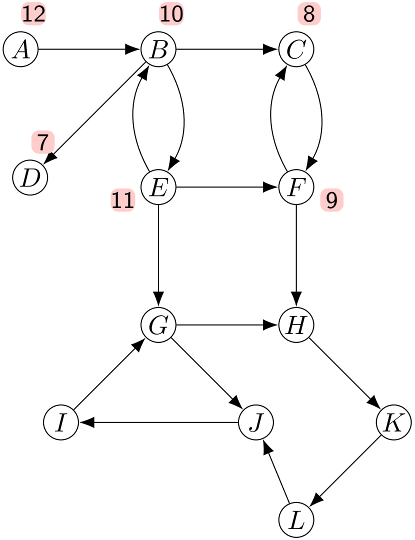


12
10
8
A
B
C
7
D
E
F
11
9
G
H
I
J
K
L

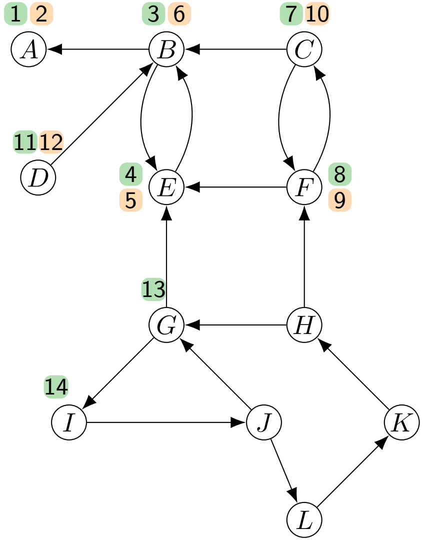


1 2
3 6
7 10
A
B
C
11 12
D
4
5
E
F
8
9
13
G
H
I
J
K
L

Find SCC's

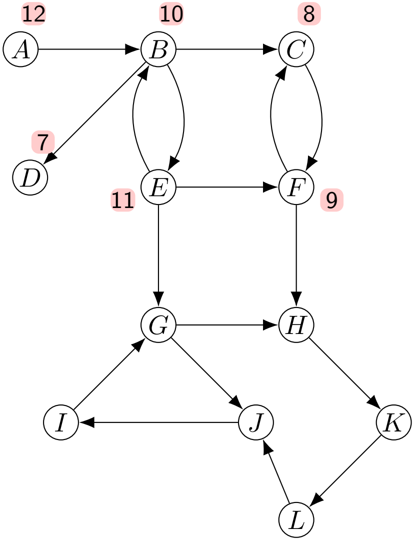


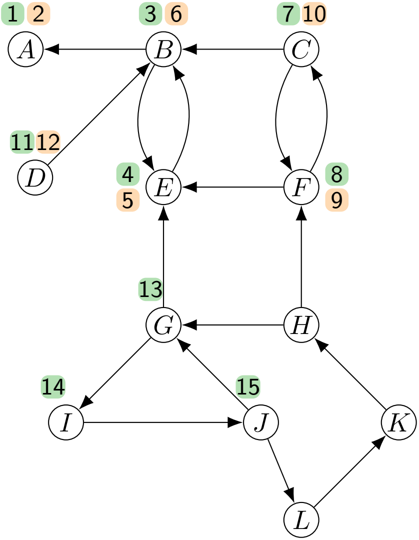
12
10
8
A
B
C
7
D
E
F
11
9
G
H
I
J
K
L



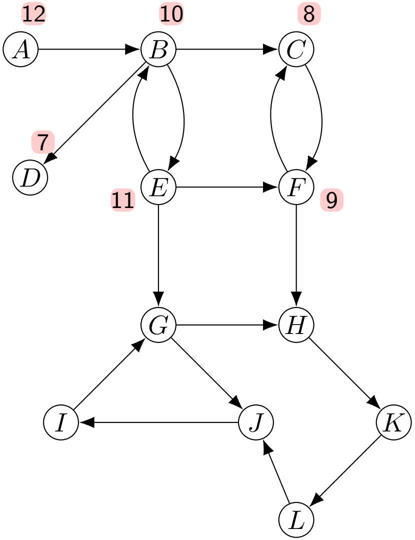
1 2
3 6
7 10
A
B
C
11 12
D
4
5
E
F
8
9
13
G
H
14
I
J
K
L

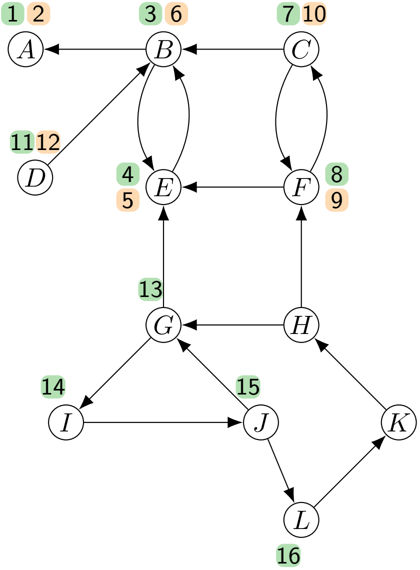
Find SCC's


12
10
8
A
B
C
7
D
E
F
11
9
G
H
I
J
K
L

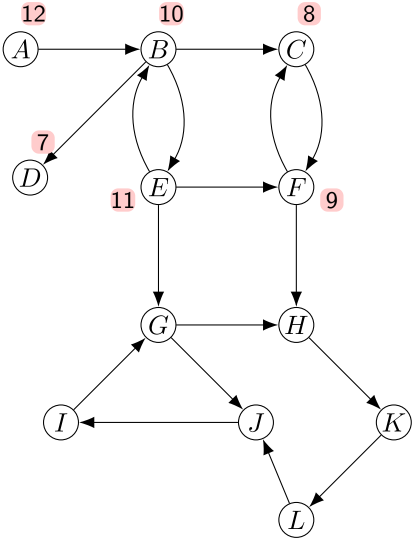

1 2
3 6
7 10
A
B
C
11 12
D
4
5
E
F
8
9
13
G
H
14
15
I
J
K
L

Find SCC's

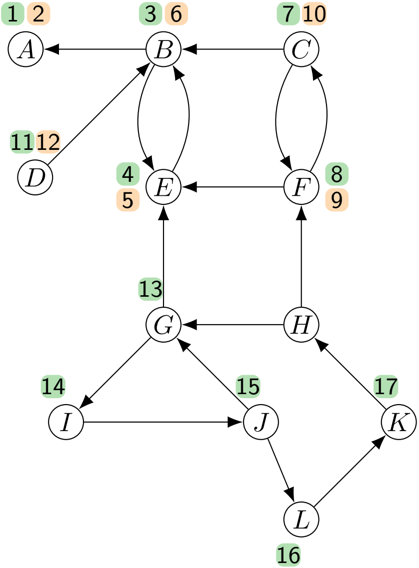

12
10
8
A
B
C
7
D
E
F
11
9
G
H
I
J
K
L


1 2
3 6
7 10
A
B
C
11 12
D
4
5
E
F
8
9
13
G
H
14
15
I
J
K
L
16

Find SCC's

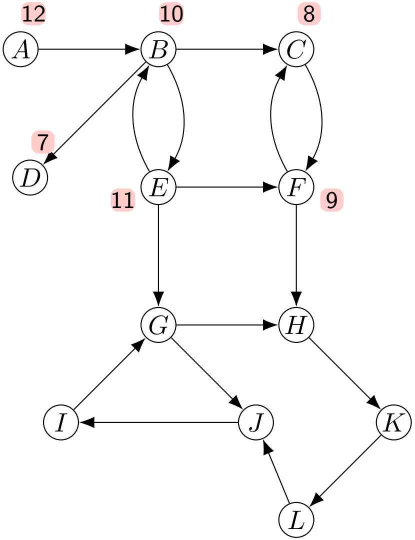


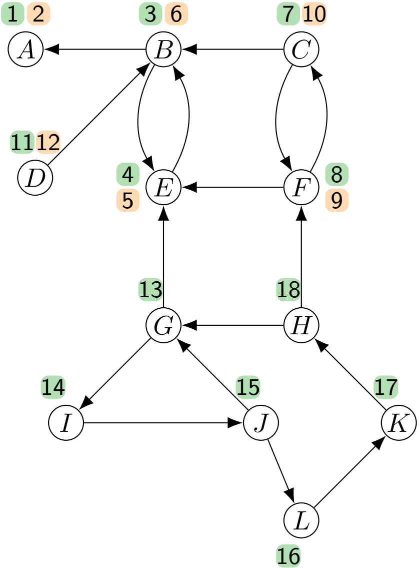
12
10
8
A
B
C
7
D
E
F
11
9
G
H
I
J
K
L



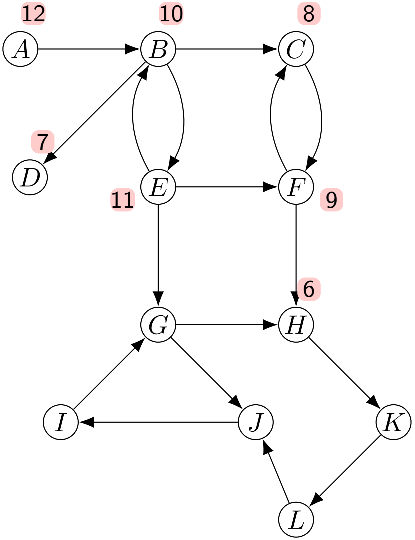
1
2
3
6
7
10
A
B
C
11
12
D
4
5
E
F
8
9
13
G
H
14
15
17
I
J
K
L
16

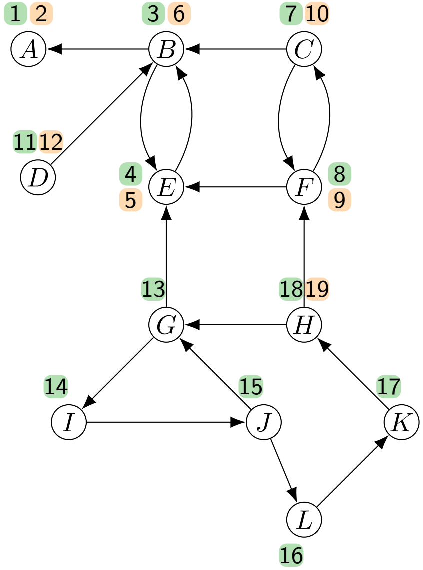
Find SCC's


A
B
C
D
E
F
G
H
I
J
K
L
12
10
8
7
11
9

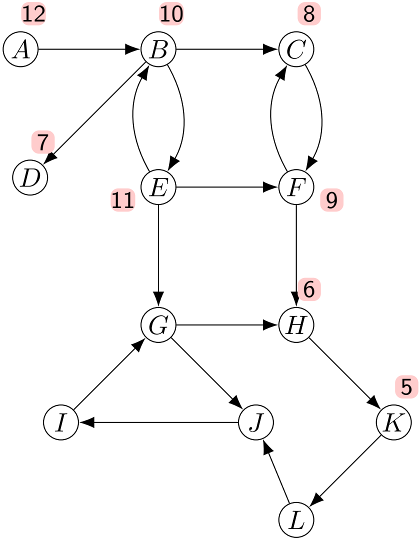

A
B
C
D
E
F
G
H
I
J
K
L
1 2
3 6
7 10
11 12
4
5
8
9
13
18
14
15
17
16

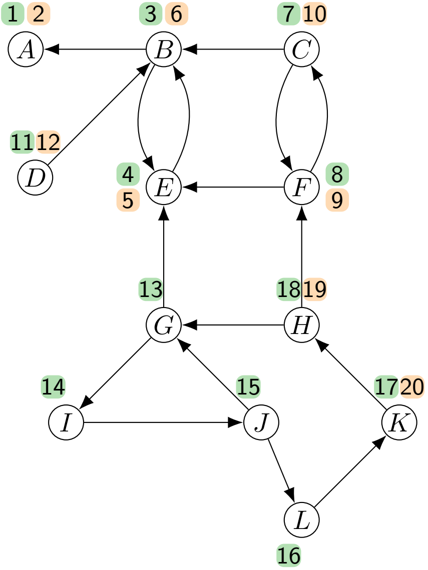
Find SCC's


12
10
8
A
B
C
7
D
E
F
11
9
6
G
H
I
J
K
L

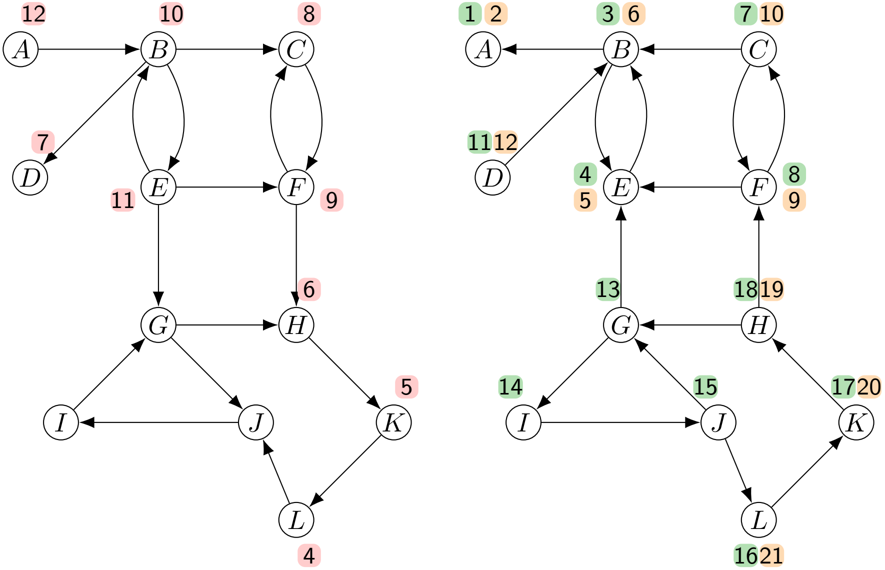

1 2
3 6
7 10
A
B
C
11 12
D
4
5
E
F
8
9
13
18 19
G
H
14
15
17
I
J
K
L
16

Find SCC's


12
10
8
7
11
9
6
5
A
B
C
D
E
F
G
H
I
J
K
L

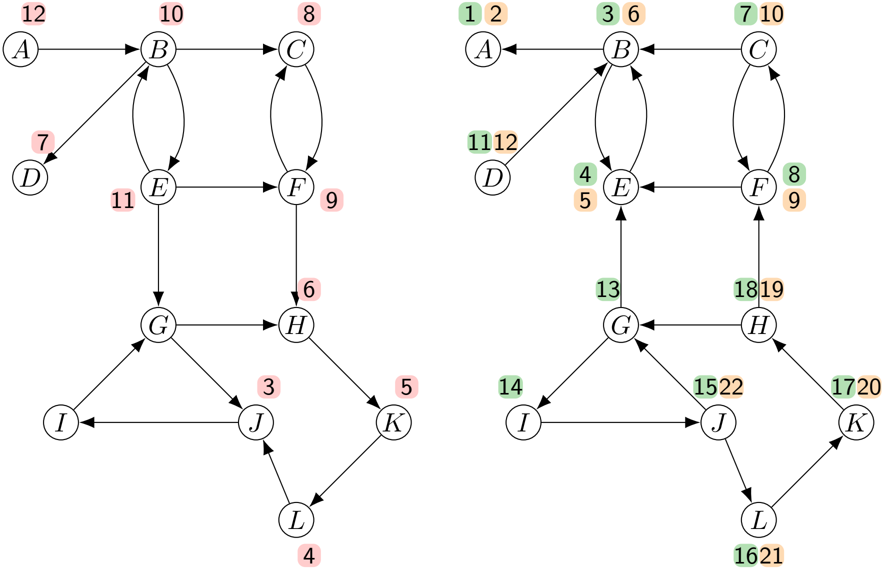

1 2
3 6
7 10
11 12
4
5
8
9
13
18 19
14
15
17 20
16
A
B
C
D
E
F
G
H
I
J
K
L

Find SCC's

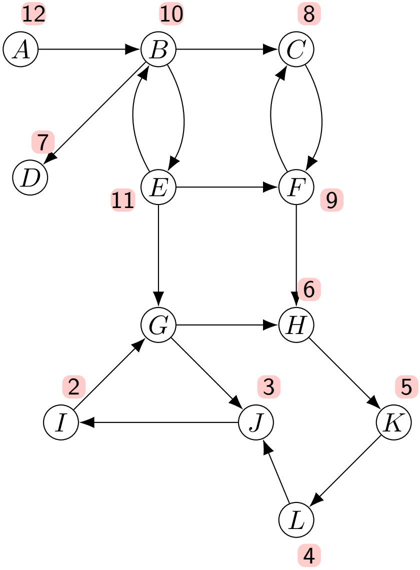


12
10
8
A
B
C
7
D
E
F
11
9
6
G
H
5
I
J
K
L
4
1 2
3 6
7 10
A
B
C
11 12
D
4
5
E
F
8
9
13
18 19
G
H
14
15
17 20
I
J
K
L
16 21

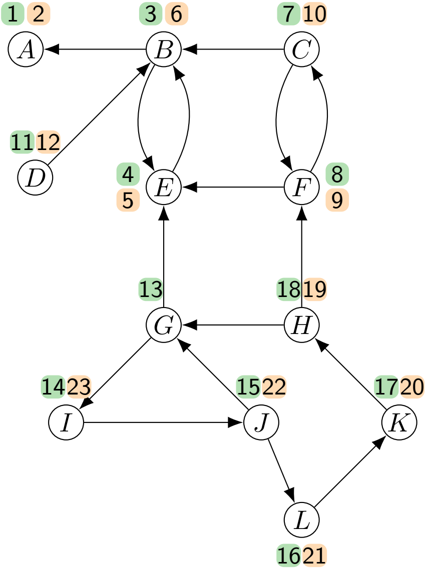
Find SCC's


A
B
C
D
E
F
G
H
I
J
K
L
12
10
8
7
11
9
6
3
5
4
1 2
3 6
7 10
11 12
4 5
8 9
13
18 19
14
15 22
17 20
16 21

Find SCC's

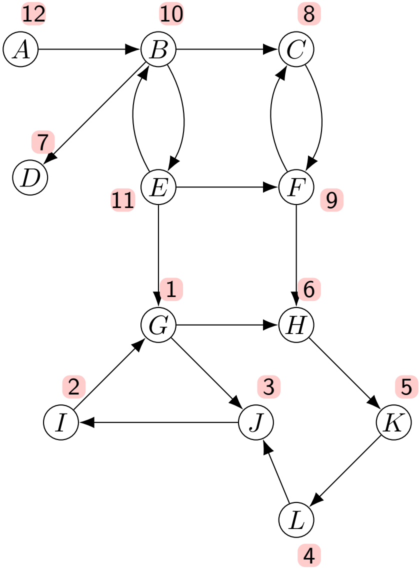


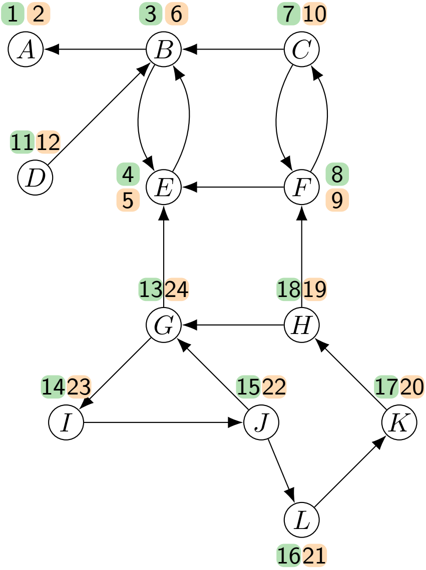
12
A
10
B
8
C
7
D
11
E
F
9
6
G
H
2
3
5
I
J
K
L
4



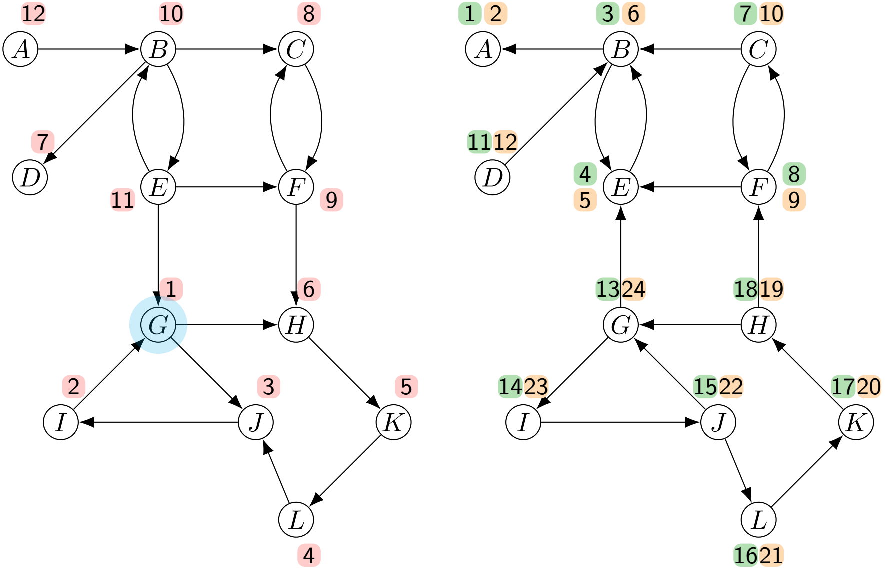
1 2
A
3 6
B
7 10
C
11 12
D
4
5
E
F
8
9
13
18 19
G
H
14 23
15 22
17 20
I
J
K
L
16 21

Find SCC's


12
10
8
A
B
C
7
D
E
F
11
9
1
6
G
H
2
3
5
I
J
K
L
4

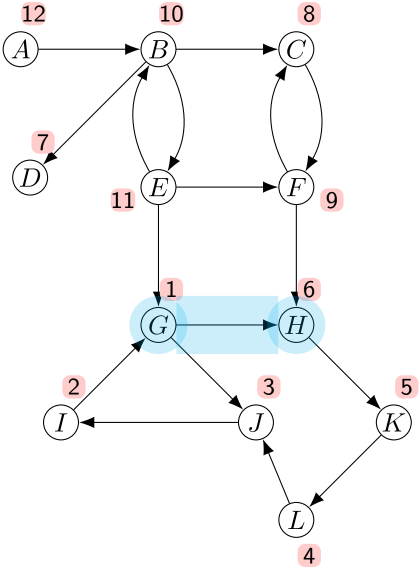

1 2
3 6
7 10
A
B
C
11 12
D
4
5
E
F
8
9
13 24
18 19
G
H
14 23
15 22
17 20
I
J
K
L
16 21

Find SCC's

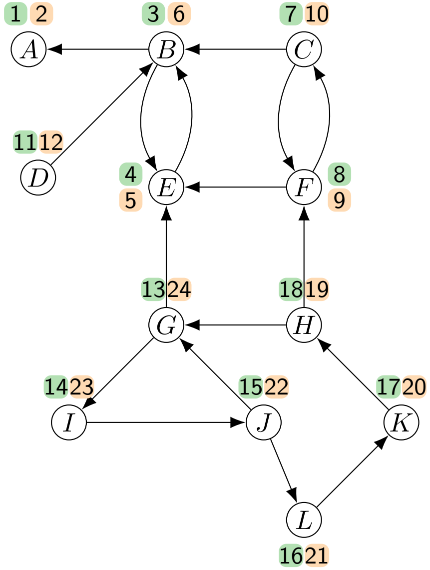


12
10
8
A
B
C
7
D
E
F
11
9
1
6
G
H
2
3
5
I
J
K
L
4
1 2
3 6
7 10
A
B
C
11 12
D
4
5
E
F
8
9
13 24
18 19
G
H
14 23
15 22
17 20
I
J
K
L
16 21

Find SCC's

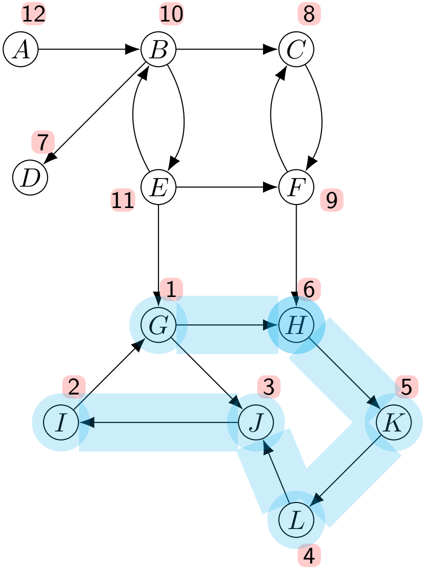


12
10
8
A
B
C
7
D
E
F
11
9
1
6
G
H
2
3
5
I
J
K
L
4

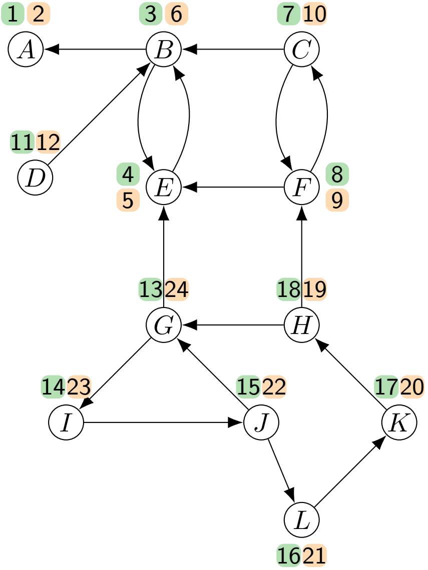


1 2
3 6
7 10
A
B
C
11 12
D
4
5
E
F
8
9
13 24
18 19
G
H
14 23
15 22
17 20
I
J
K
L
16 21

Find SCC's

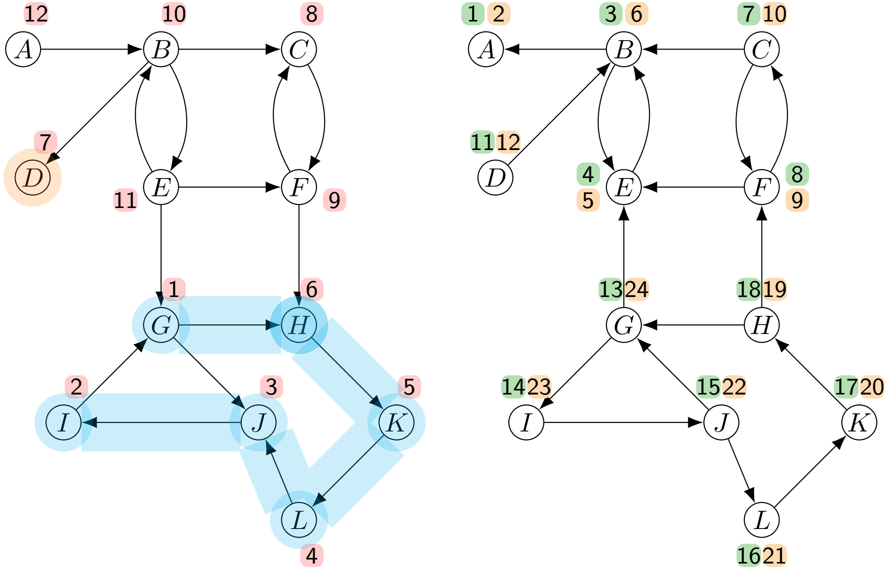


12
10
8
A
B
C
7
D
E
F
11
9
1
6
G
H
2
3
5
I
J
K
L
4



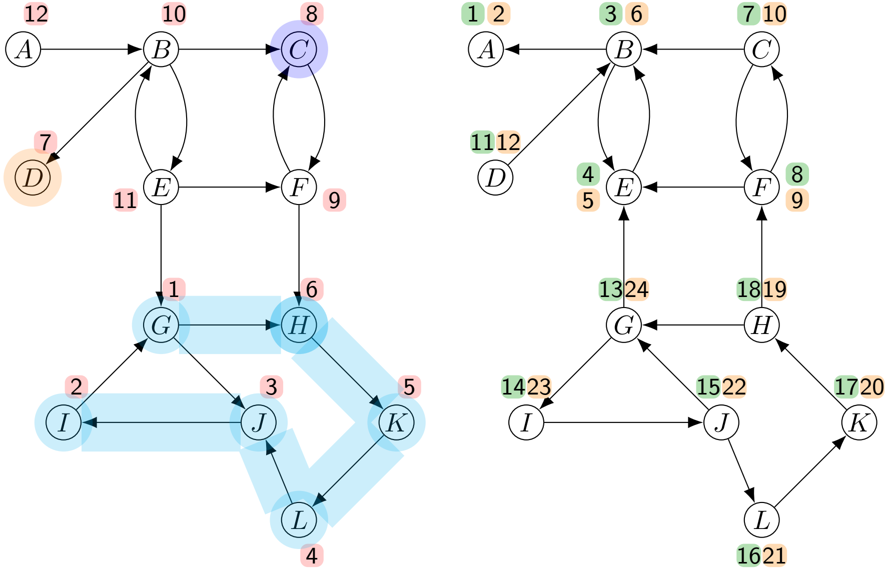
1 2
3 6
7 10
A
B
C
11 12
4
5
D
E
F
8
9
13 24
18 19
G
H
14 23
15 22
17 20
I
J
K
L
16 21

Find SCC's

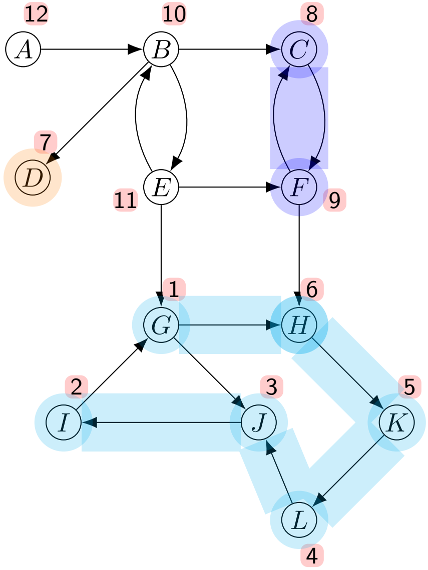


12
10
8
A
B
C
7
D
E
F
11
9
1
6
G
H
2
3
5
I
J
K
L
4
1 2
3 6
7 10
A
B
C
11 12
D
4
5
E
F
8
9
13 24
18 19
G
H
14 23
15 22
17 20
I
J
K
L
16 21

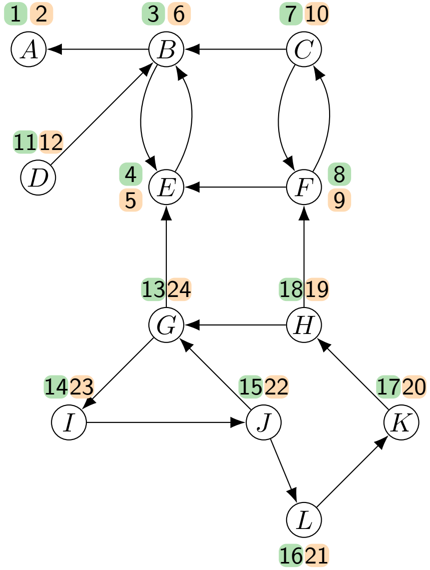
Find SCC's


12
A
10
B
8
C
7
D
11
E
F
9
1
G
6
H
2
I
3
J
5
K
L
4
1 2
A
3 6
B
7 10
C
11 12
D
4
5
E
F
8
9
13 24
G
18 19
H
14 23
I
15 22
J
17 20
K
L
16 21

Find SCC's

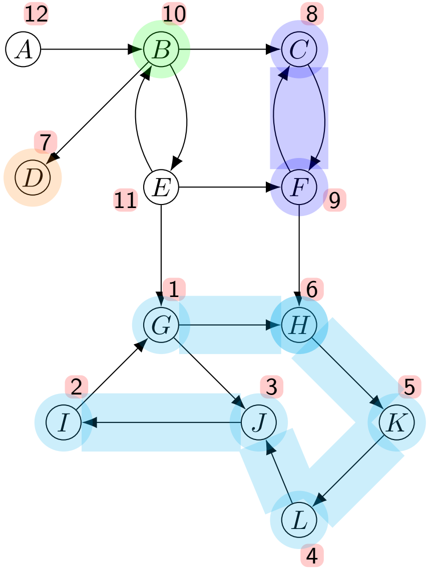


A B C D E F G H I J K L
12 10 8 7 11 9 1 6 2 3 5 4

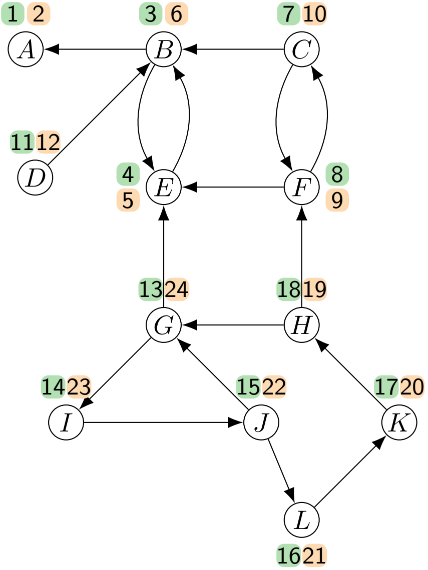


A B C D E F G H I J K L
1 2
3 6
7 10
11 12
4 5
8 9
13 24
18 19
14 23
15 22
17 20
16 21

Find SCC's

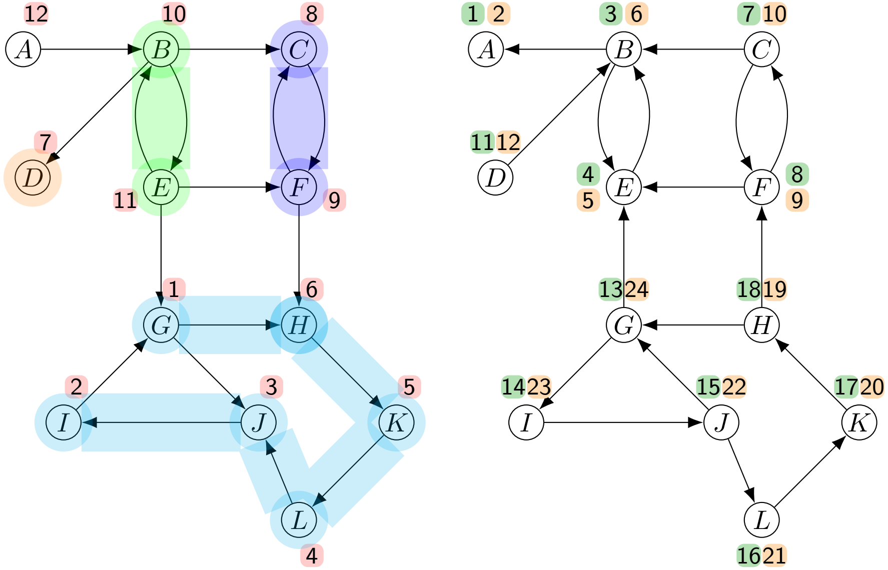


12
10
8
A
B
C
7
D
E
F
11
9
1
6
G
H
2
3
5
I
J
K
L
4



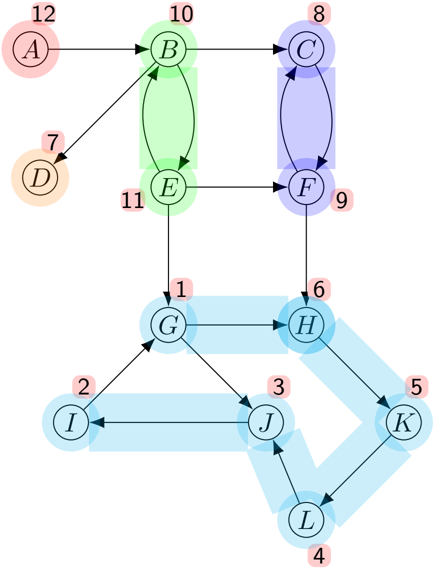
1 2
3 6
7 10
A
B
C
11 12
D
4
5
E
F
8
9
13 24
18 19
G
H
14 23
15 22
17 20
I
J
K
L
16 21

Find SCC's

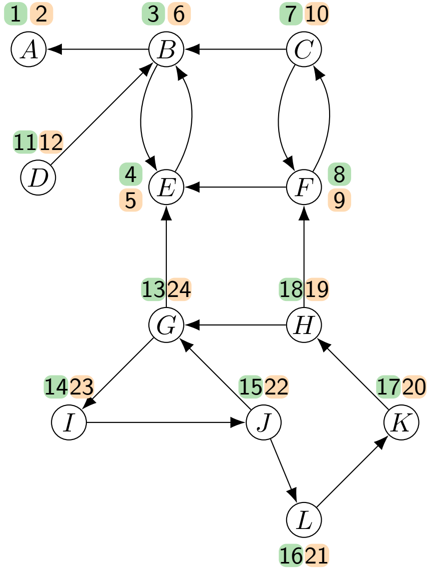


12
A
10
B
8
C
7
D
E
11
F
9
1
G
6
H
2
I
3
J
5
K
L
4
1 2
A
3 6
B
7 10
C
11 12
D
4
5
E
F
8
9
13 24
G
18 19
H
14 23
I
15 22
J
17 20
K
L
16 21

Find SCC's



12
A
10
B
8
C
7
D
E
11
F
9
1
G
6
H
2
I
3
J
5
K
L
4



1 2
A
3 6
B
7 10
C
11 12
D
4
5
E
F
8
9
13 24
G
18 19
H
14 23
I
15 22
J
17 20
K
L
16 21