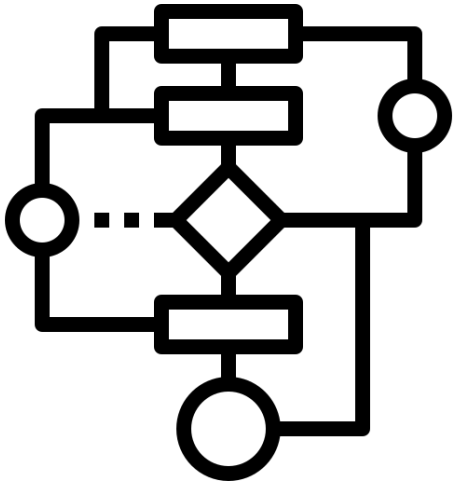
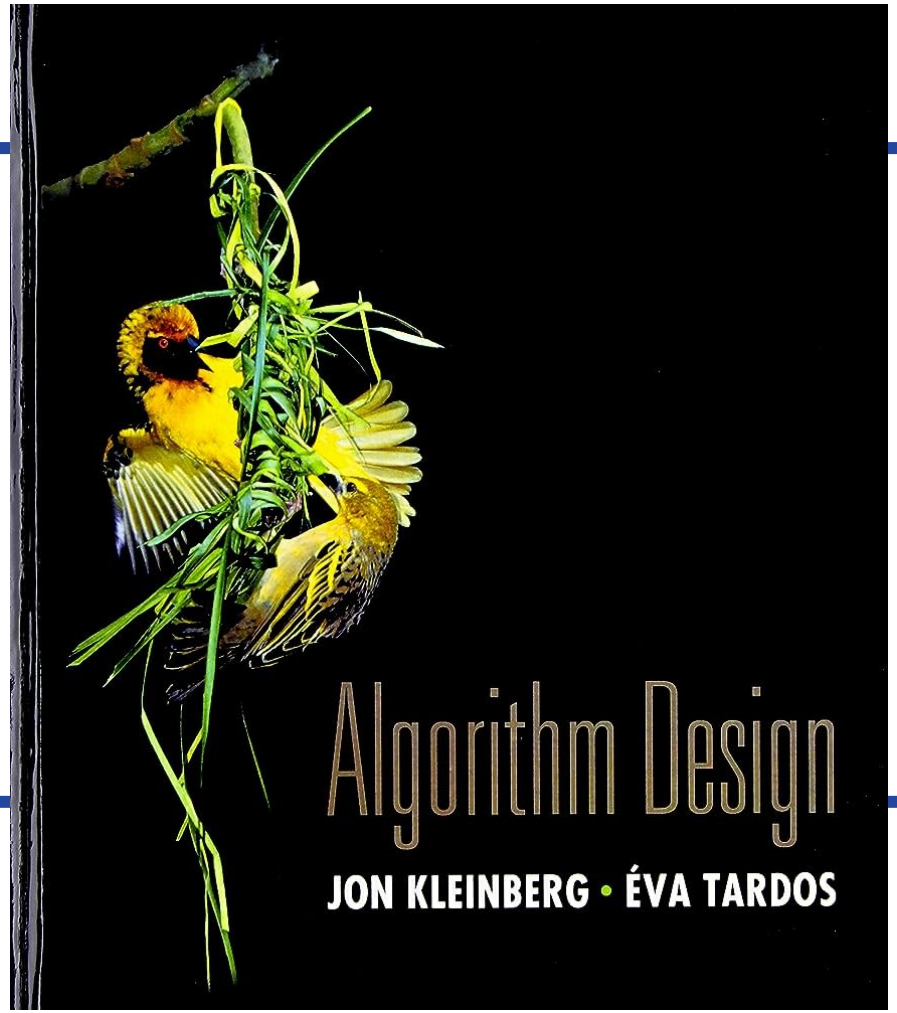


CS 310: Algorithms

Lecture 9

Instructor: Naveed Anwar Bhatti

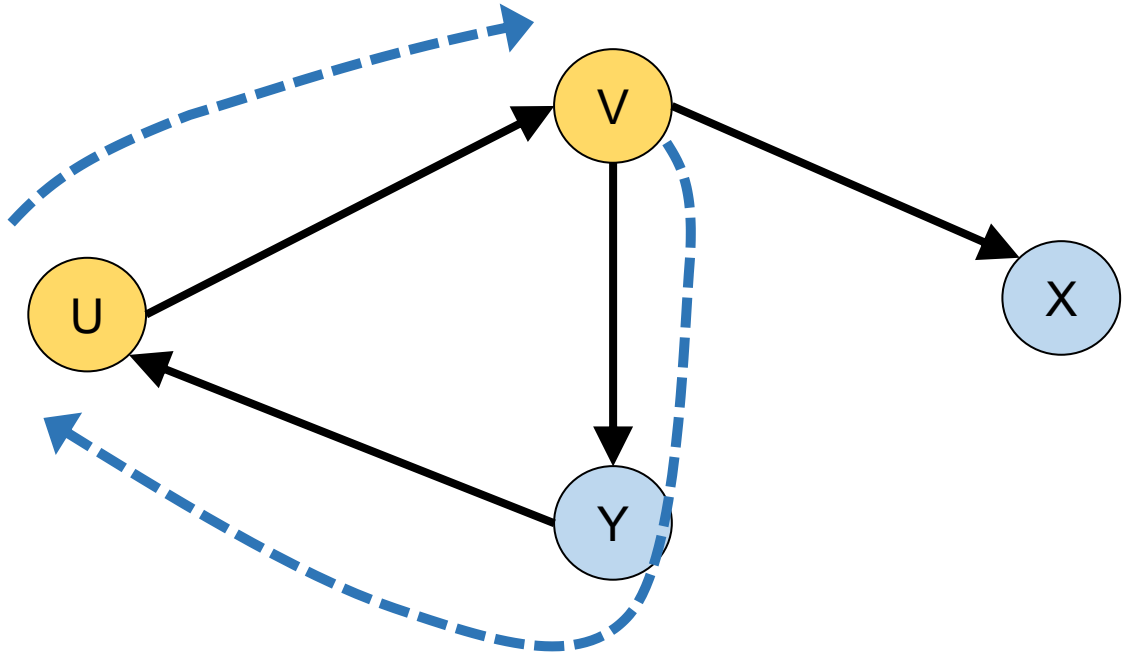


Chapter 3: **Graphs**

Section 3.5:
Connectivity in Directed Graphs

Strong Connectivity

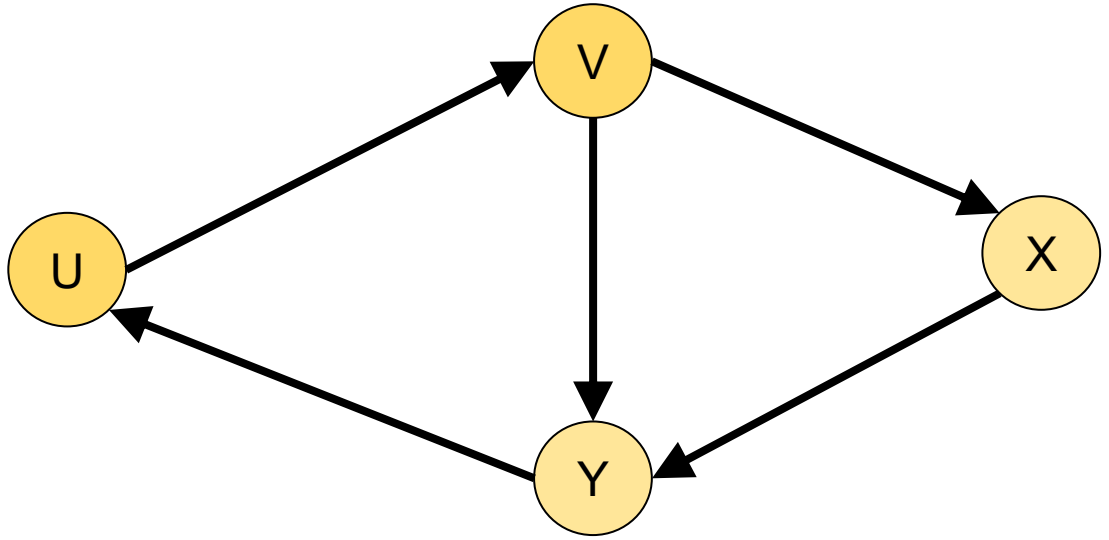
- **Def:** Node ***u*** and ***v*** are mutually reachable if there is a path from ***u*** to ***v*** and also a path from ***v*** to ***u***.



Strong Connectivity

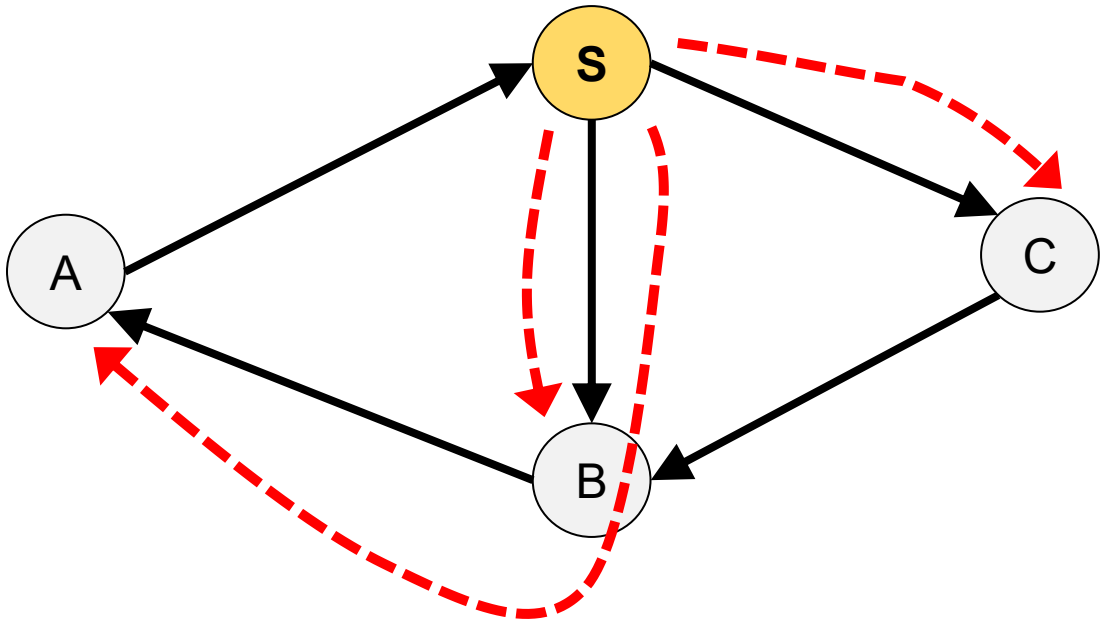
- **Def:** Node ***u*** and ***v*** are mutually reachable if there is a path from ***u*** to ***v*** and also a path from ***v*** to ***u***.

- **Def:** A graph is strongly connected if every pair of nodes is mutually reachable.



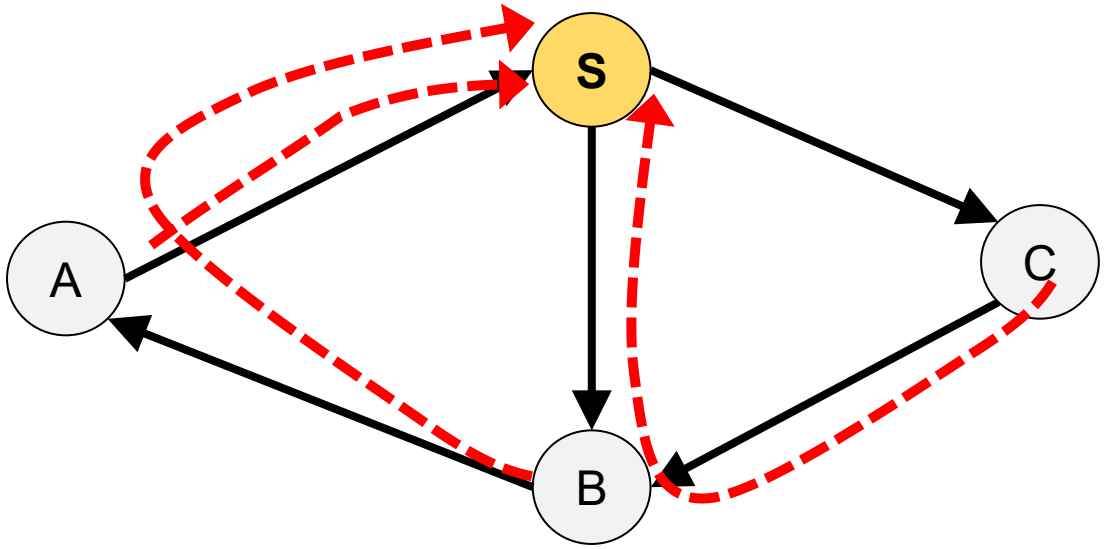
Strong Connectivity

- **Lemma.** Let ***s*** be any node in graph ***G***. ***G*** is ***strongly connected*** *iff* every node is reachable from ***s***, and ***s*** is reachable from every node**.**



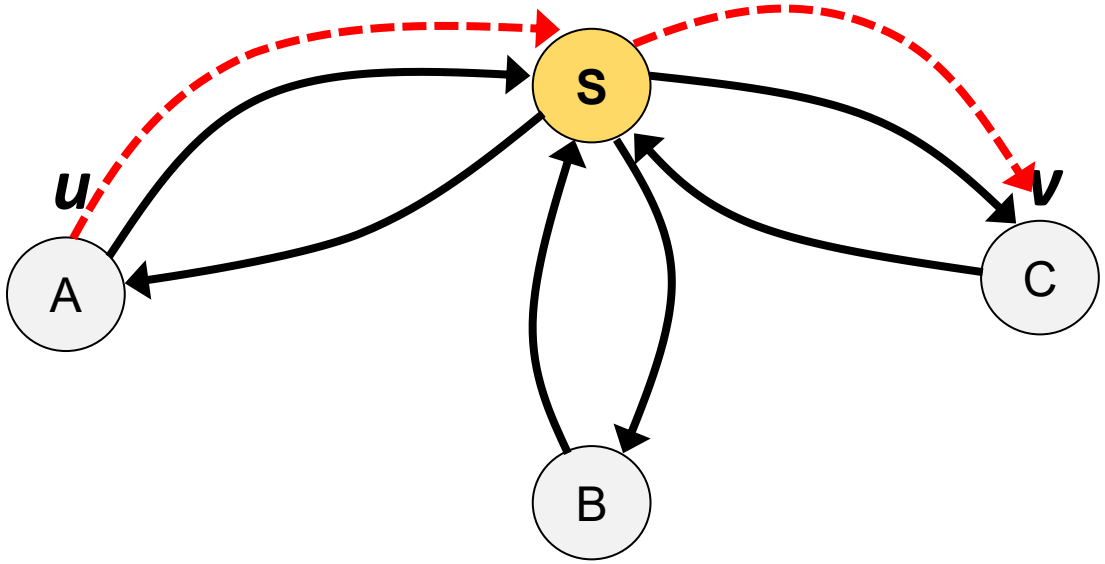
Strong Connectivity

- **Lemma.** Let ***s*** be any node in graph ***G***. ***G*** is ***strongly connected*** *iff* every node is reachable from ***s***, and ***s*** is reachable from every node**.**



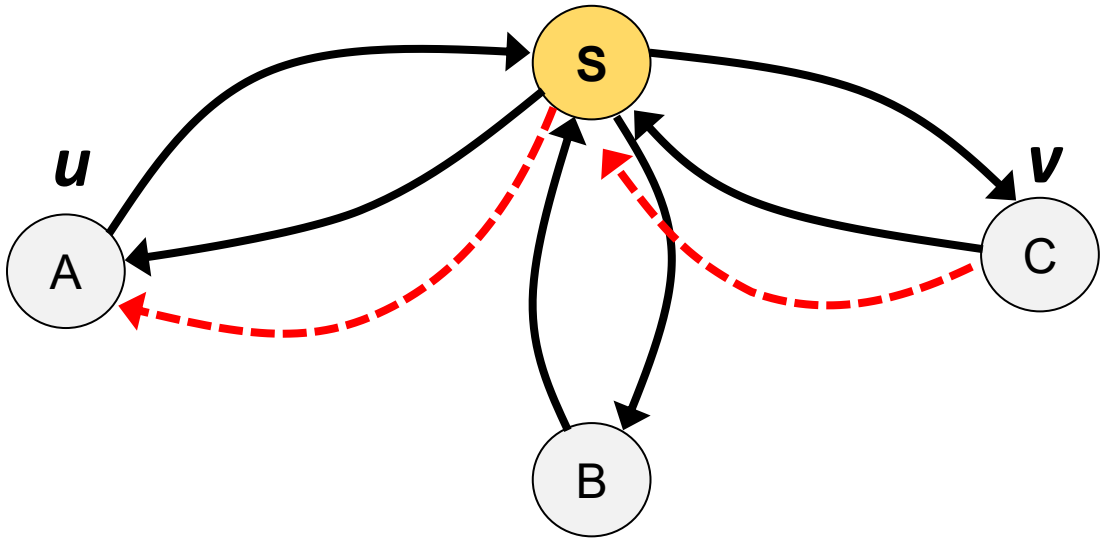
Strong Connectivity

- **Proof.** If every node is reachable from ***s*** and ***s*** is reachable from every node, then for any two nodes ***u*** and ***v*** in ***G***:
  - There is a path from **u** to **s**
  - And another from **s** to **v**
  - Combining these, ***u*** can reach ***v*** through **s**



Strong Connectivity

- **Proof.** If every node is reachable from ***s*** and ***s*** is reachable from every node, then for any two nodes ***u*** and ***v*** in ***G***:
  - There is a path from **u** to **s**
  - And another from **s** to **v**
  - Combining these, ***u*** can reach ***v*** through **s**
  - Similarly, ***v*** can reach ***u*** through ***s***

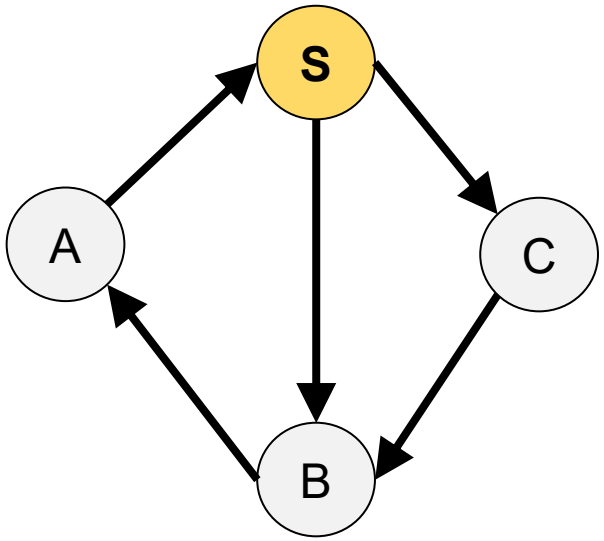


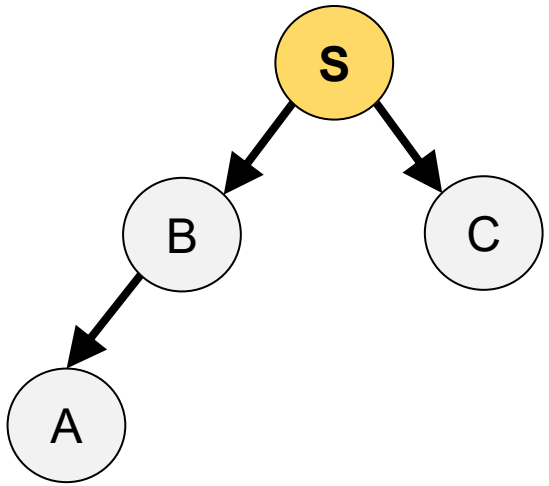
Strong Connectivity: Algorithm

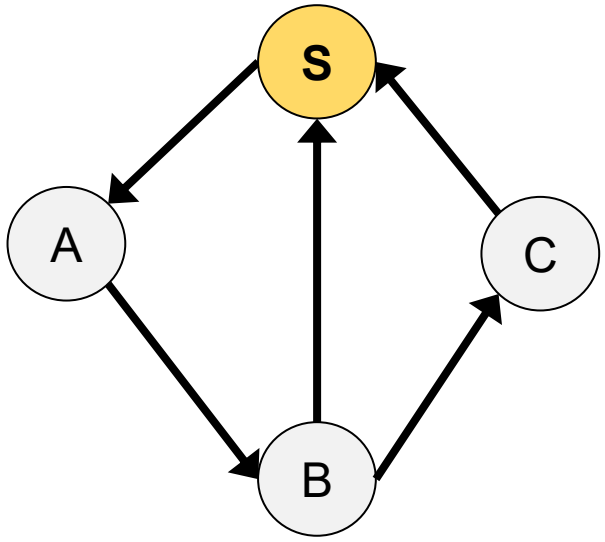
- Pick any node s.
- Run BFS from s in G.
- Run BFS from s in G^{rev}

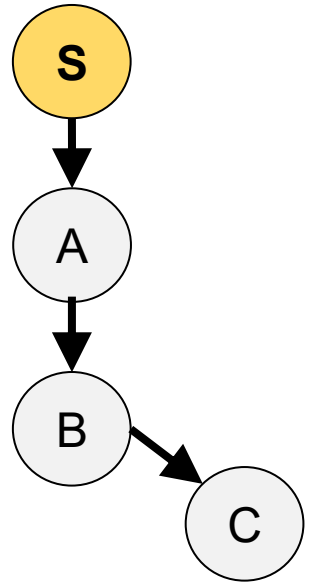
reverse orientation of every edge in **G**

- Return true ***iff*** all nodes reached in both BFS executions.
- Correctness follows immediately from previous lemma.









Strong Connectivity: Live Poll 1

What is the complexity of this algo?

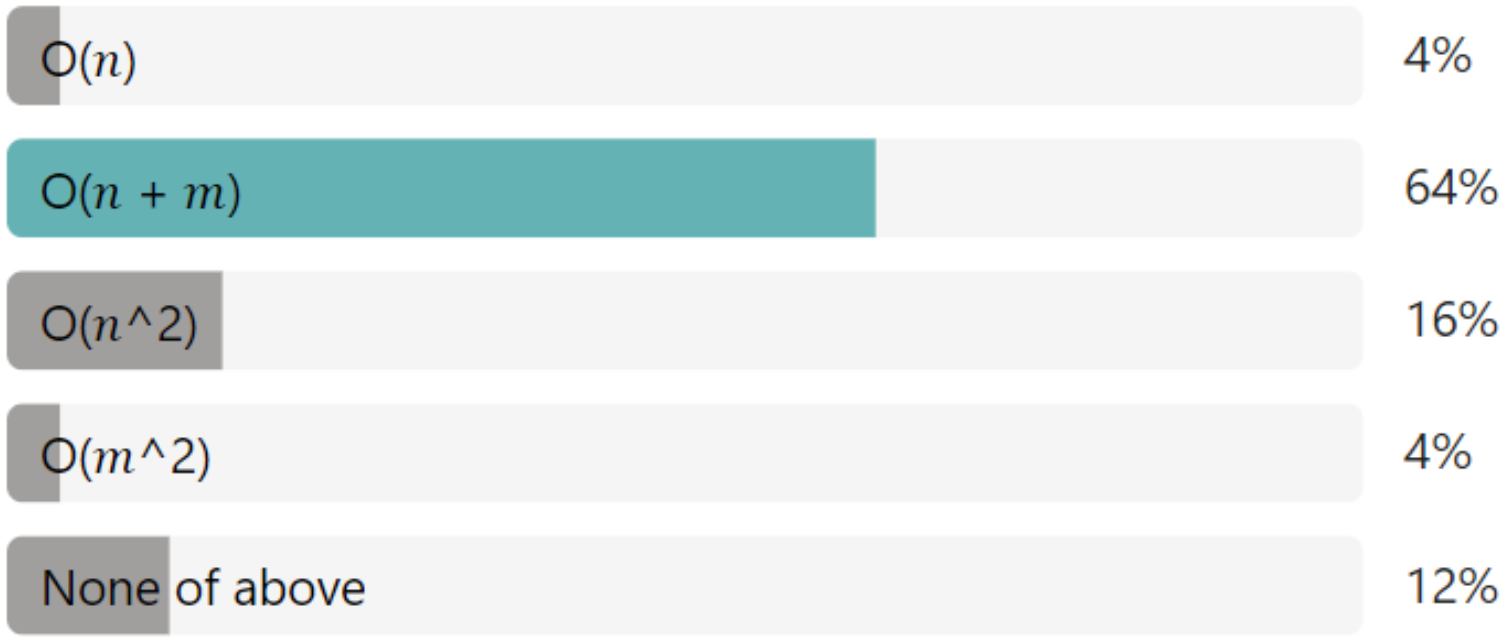
A. $O(n)$
B. $O(n + m)$
C. $O(n^2)$
D. $O(m^2)$
E. None of above

Scan the QR code to vote or go to https://forms.office.com/r/Ne2tEuqWXa

Strong Connectivity: Live Poll 1

Only people in my organization can respond, Record name

1. What is the complexity of this algo?



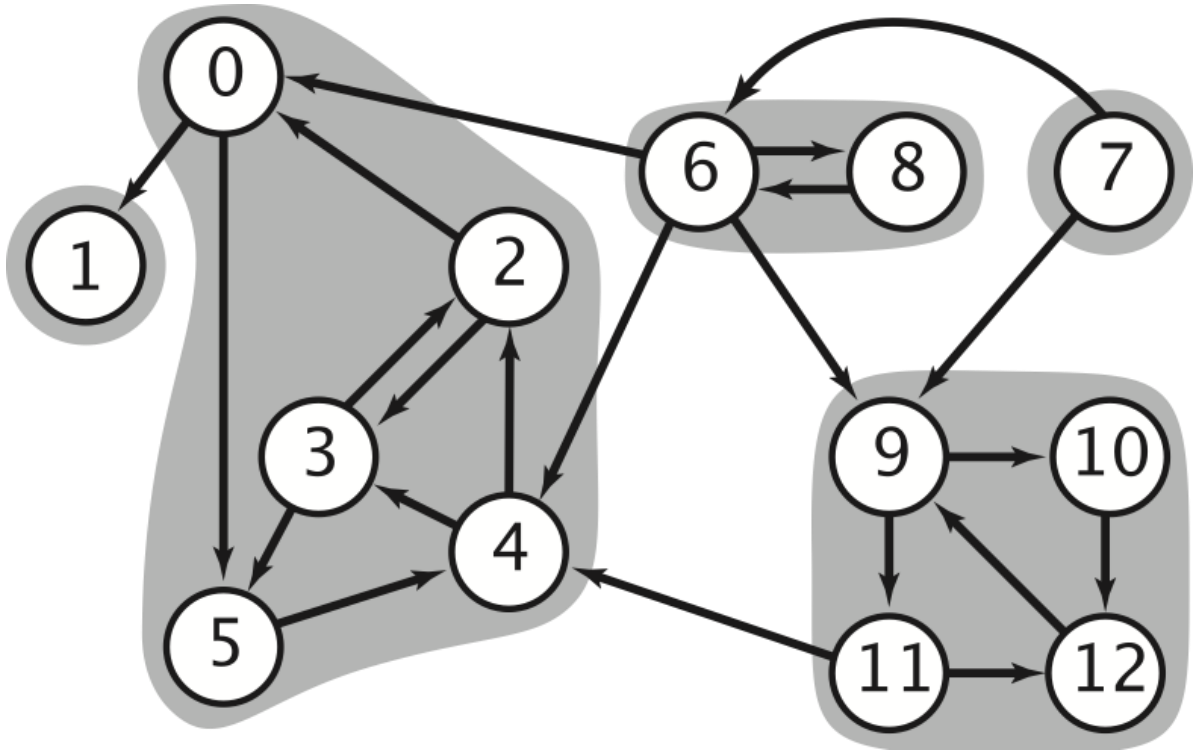
Scan the QR code to vote or go to https://forms.office.com/r/Ne2tEuqWXa

25 responses

1/1

Strong components

Def: A strong component is a maximal subset of mutually reachable nodes.



Algorithm for finding strong components in a directed graph

STRONG-COMPONENTS(G)

1 Call DFS(**G**) to compute finishing times $u.f$ for each vertex u

2 Compute $G^{reverse}$

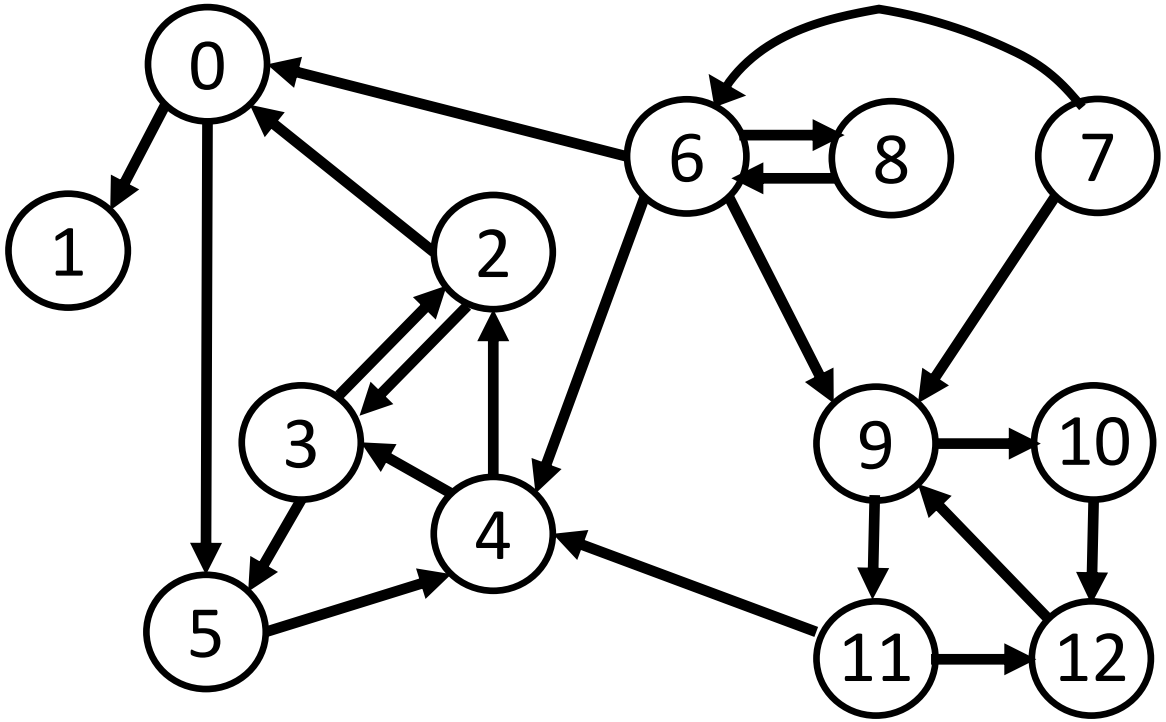
3 Call DFS($G^{reverse}$), but in the main loop of DFS, consider the vertices in order of decreasing $u.f$

4 Output the vertices of each tree in the depth-first forest formed in line 3 as a separate strong component

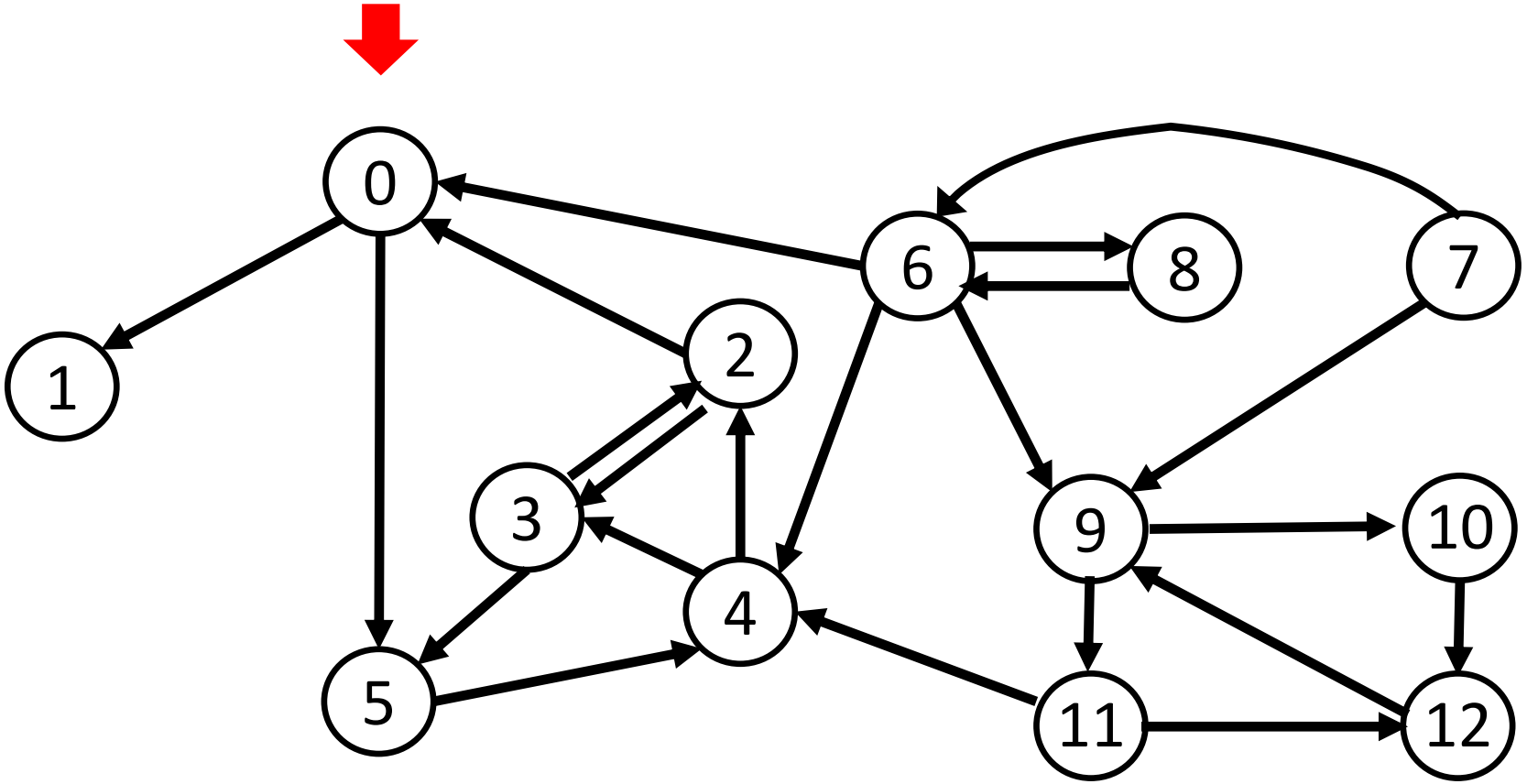
15

Strong components

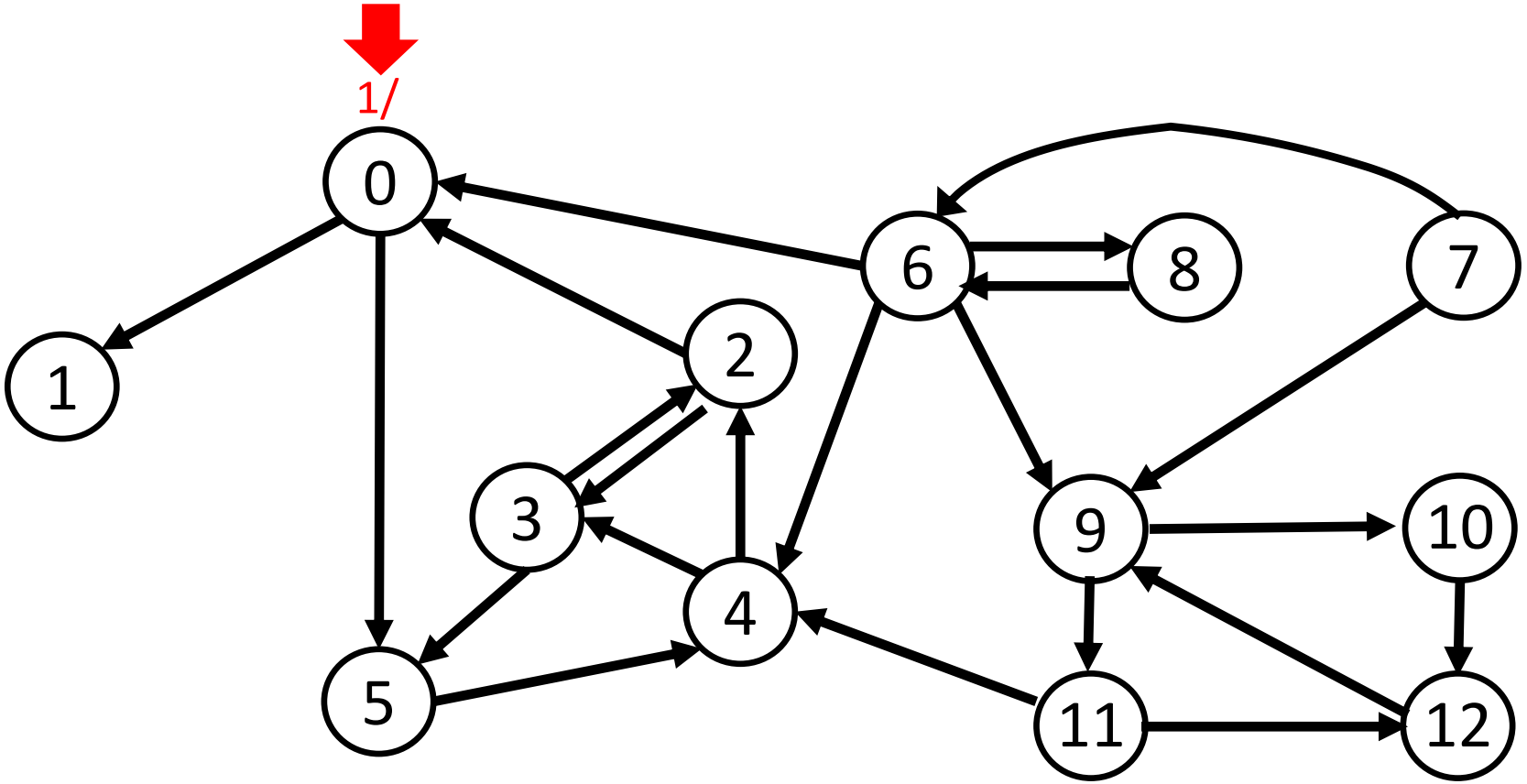
Def: A strong component is a maximal subset of mutually reachable nodes.



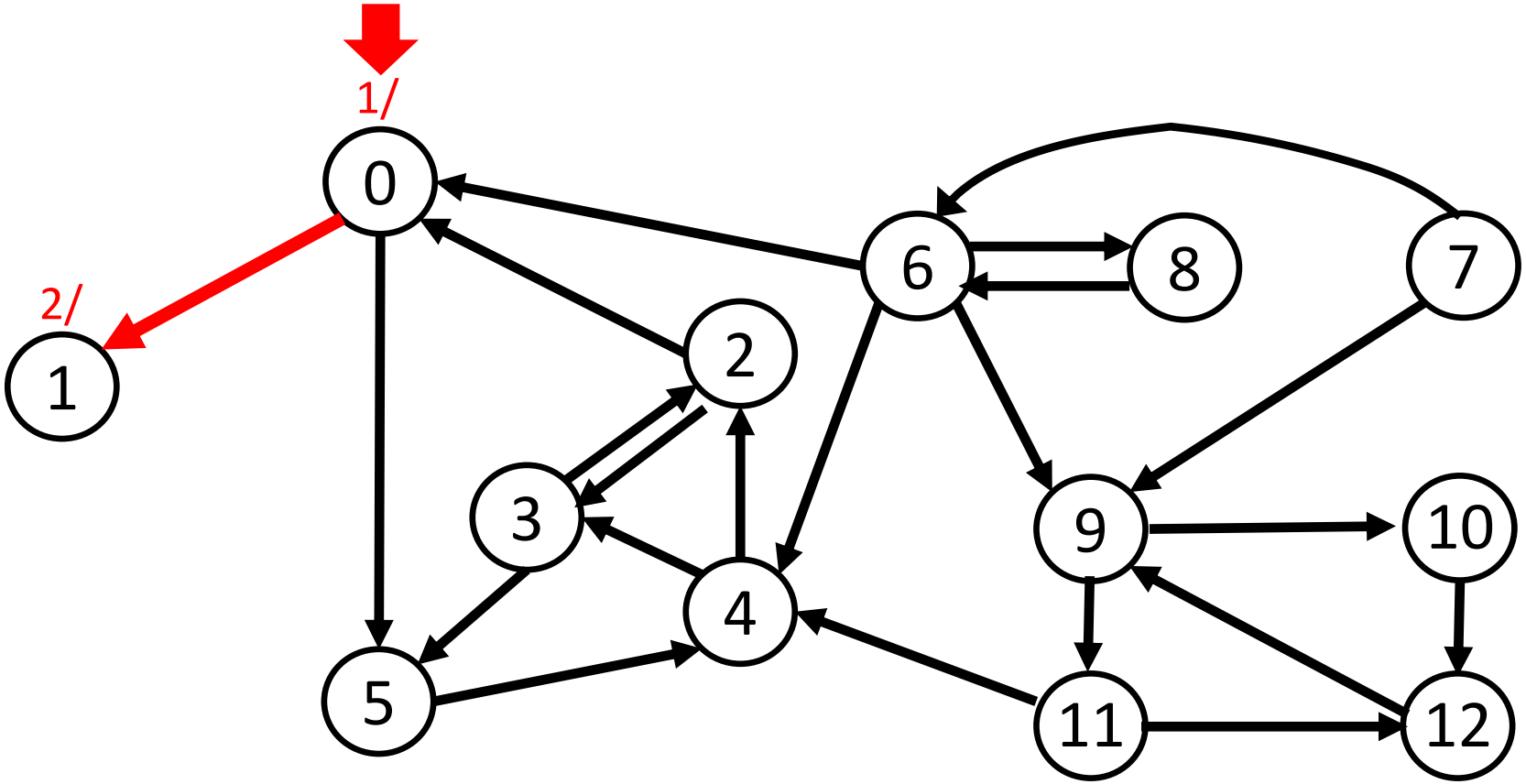
Strong components



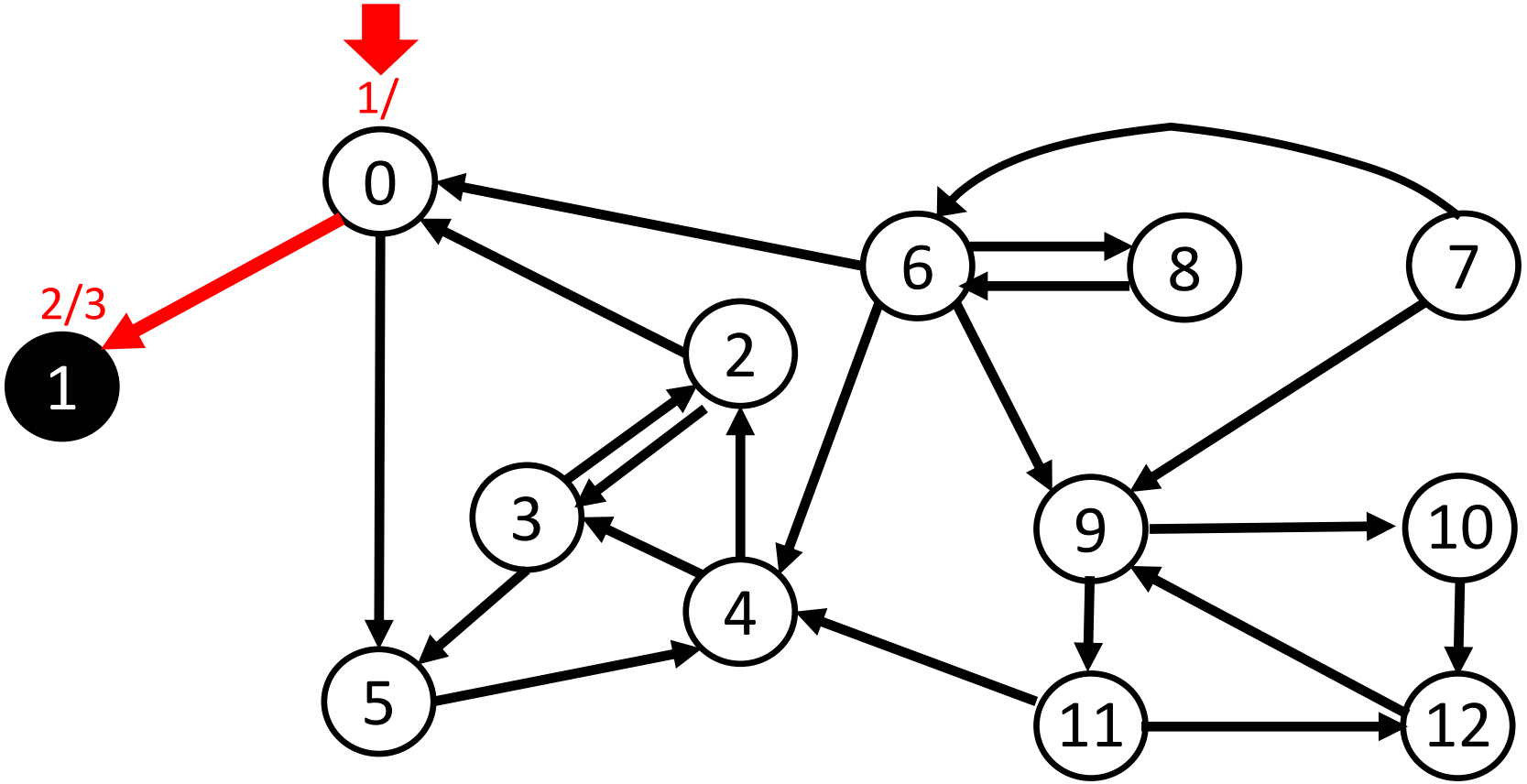
Strong components



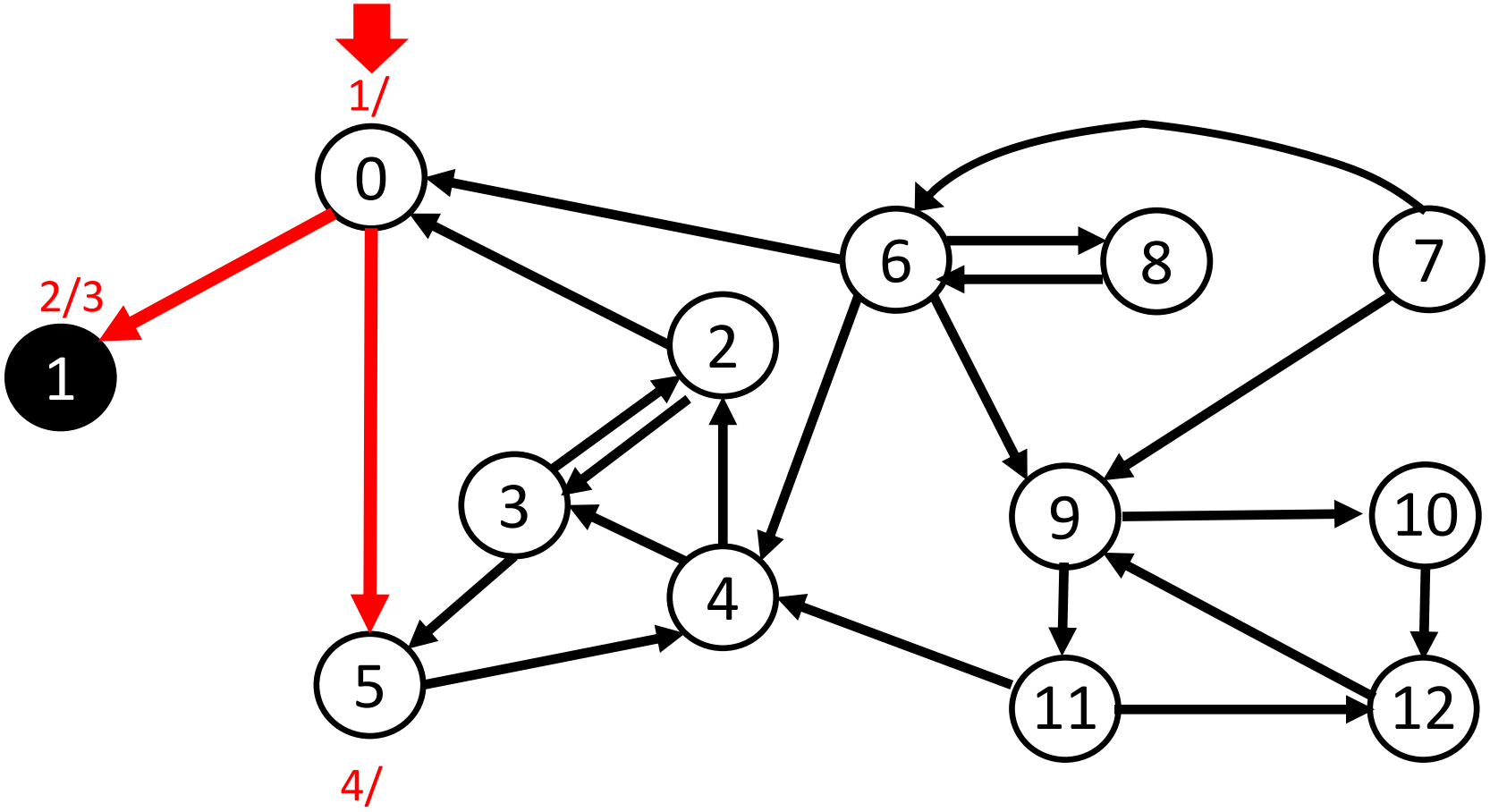
Strong components



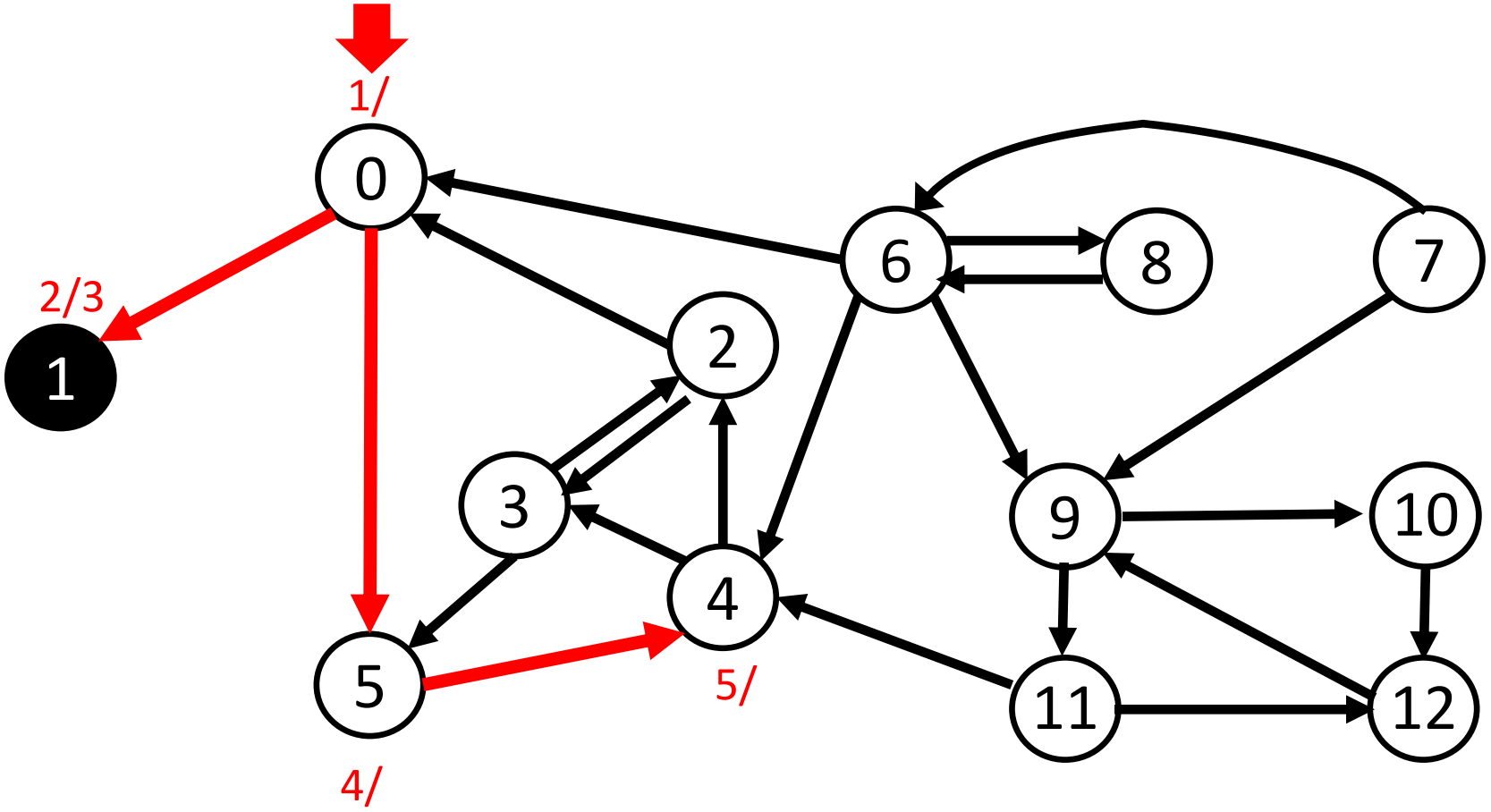
Strong components



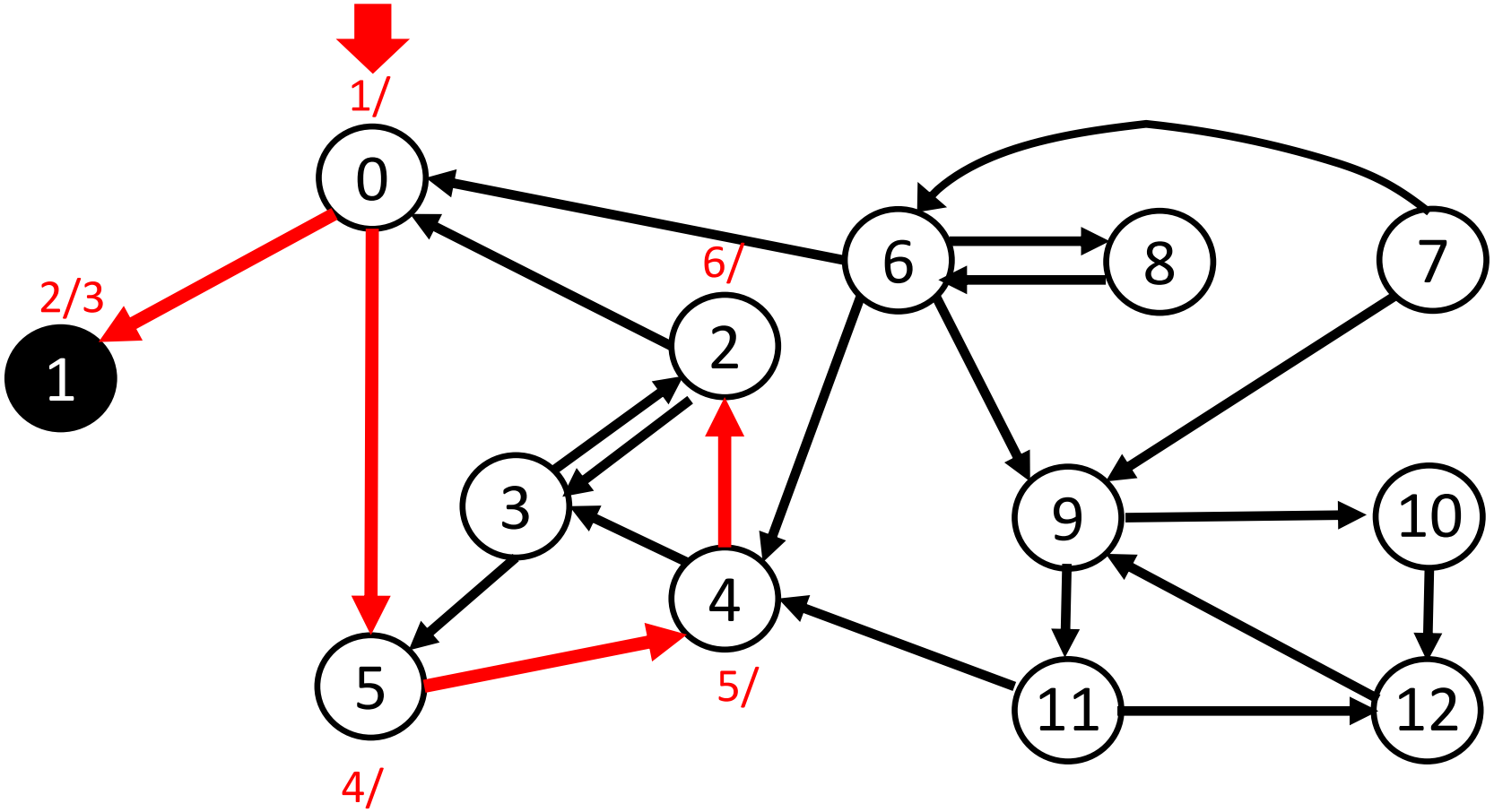
Strong components



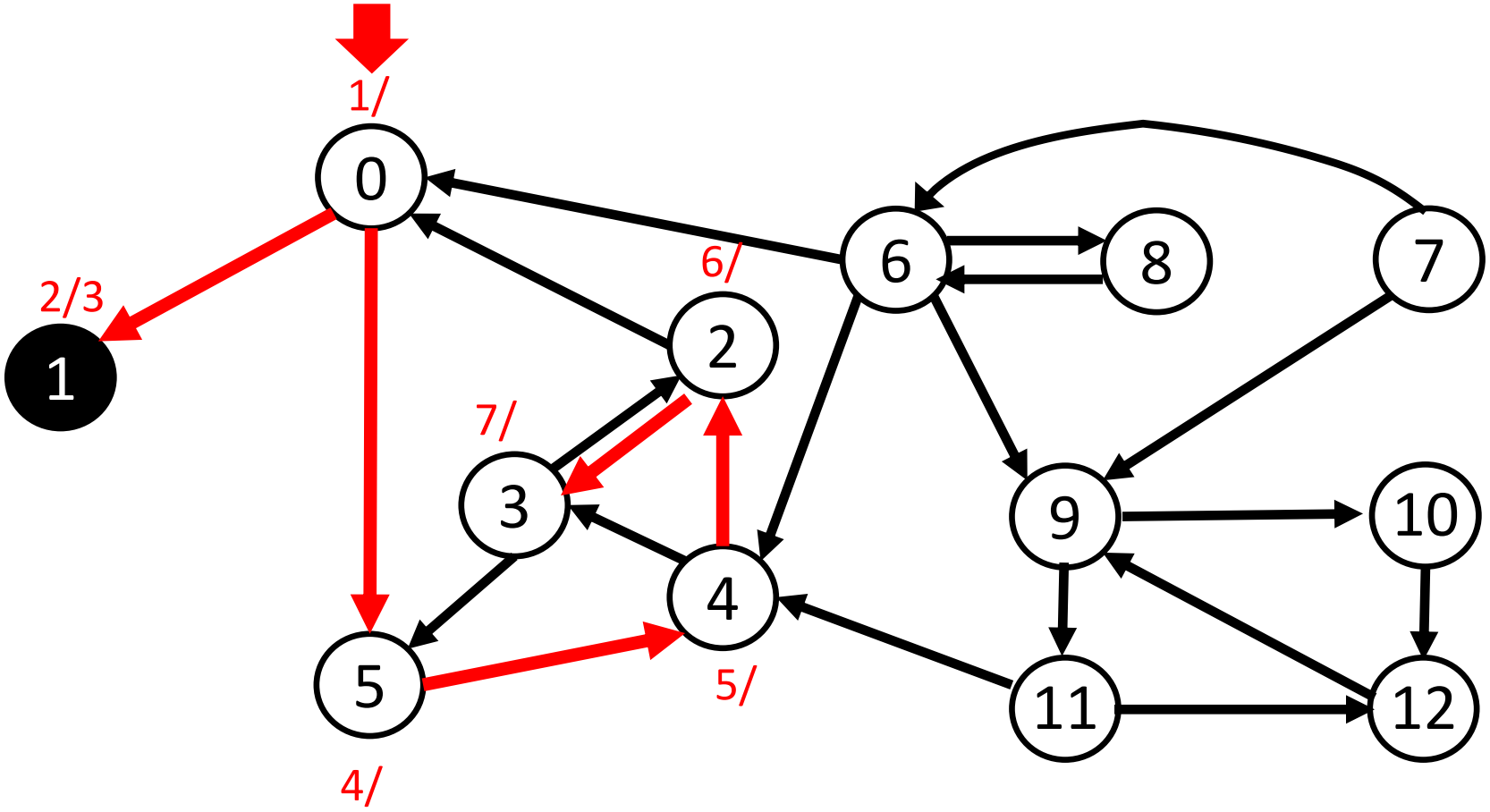
Strong components



Strong components

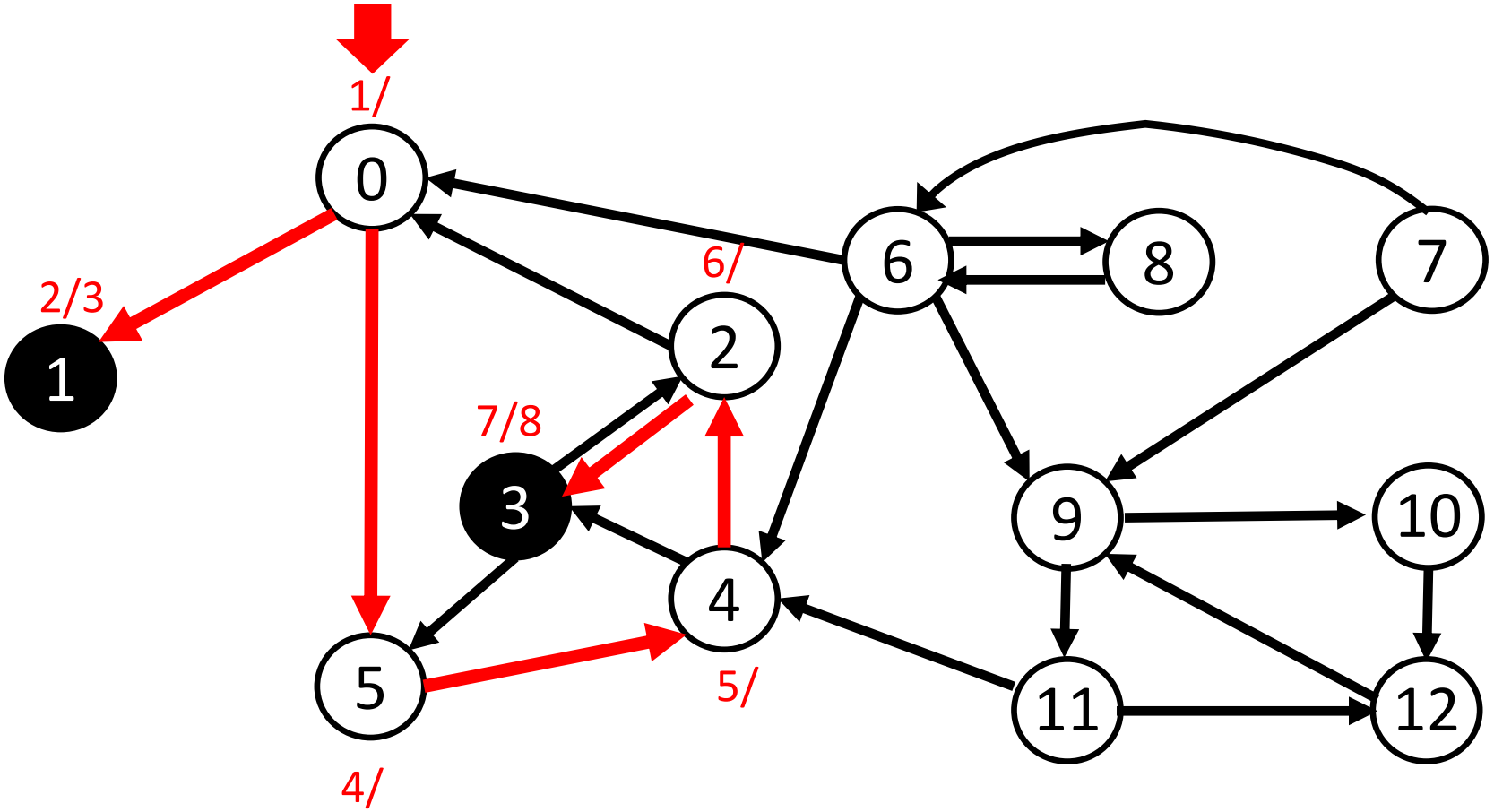


Strong components

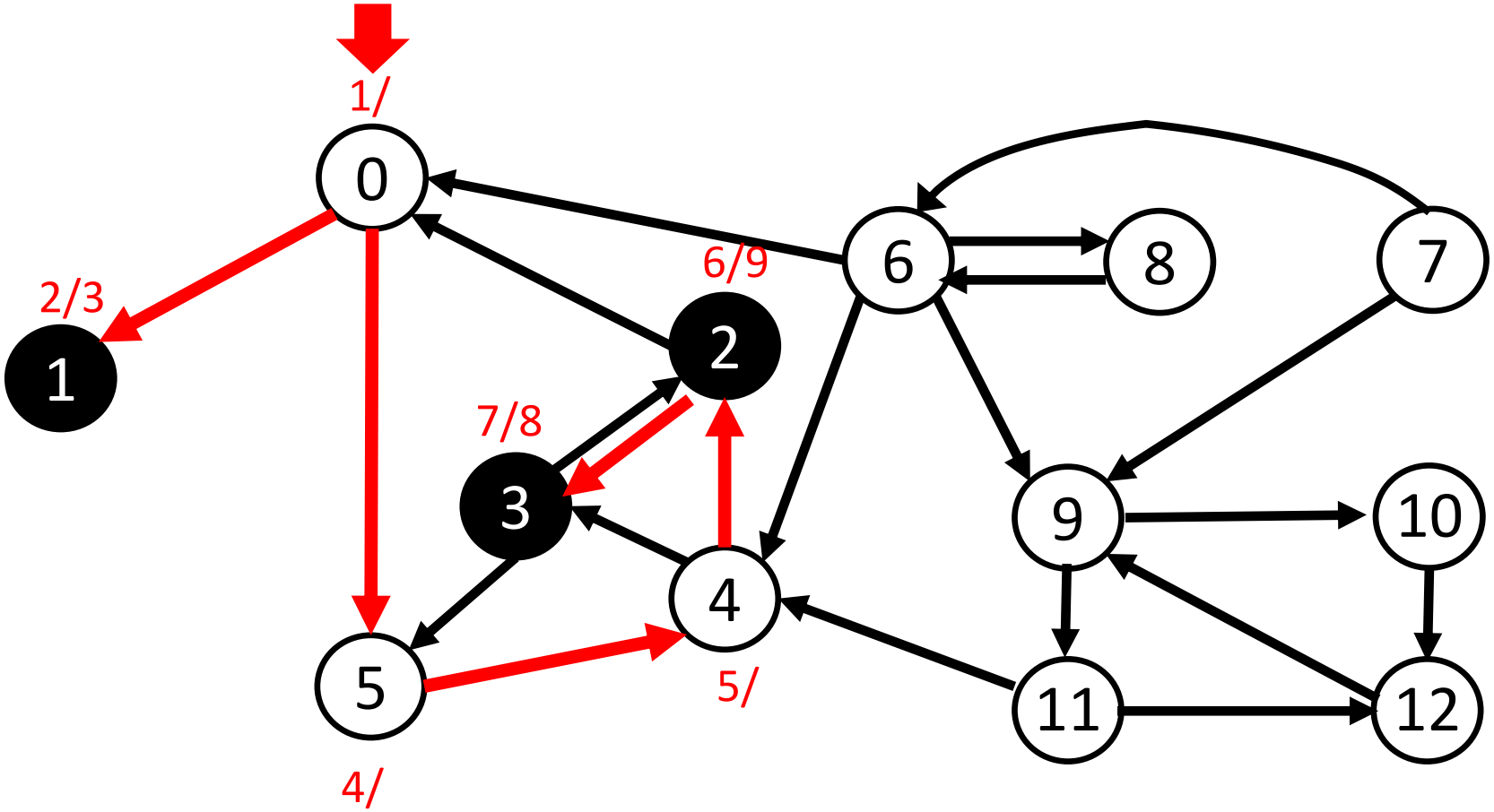


24

Strong components

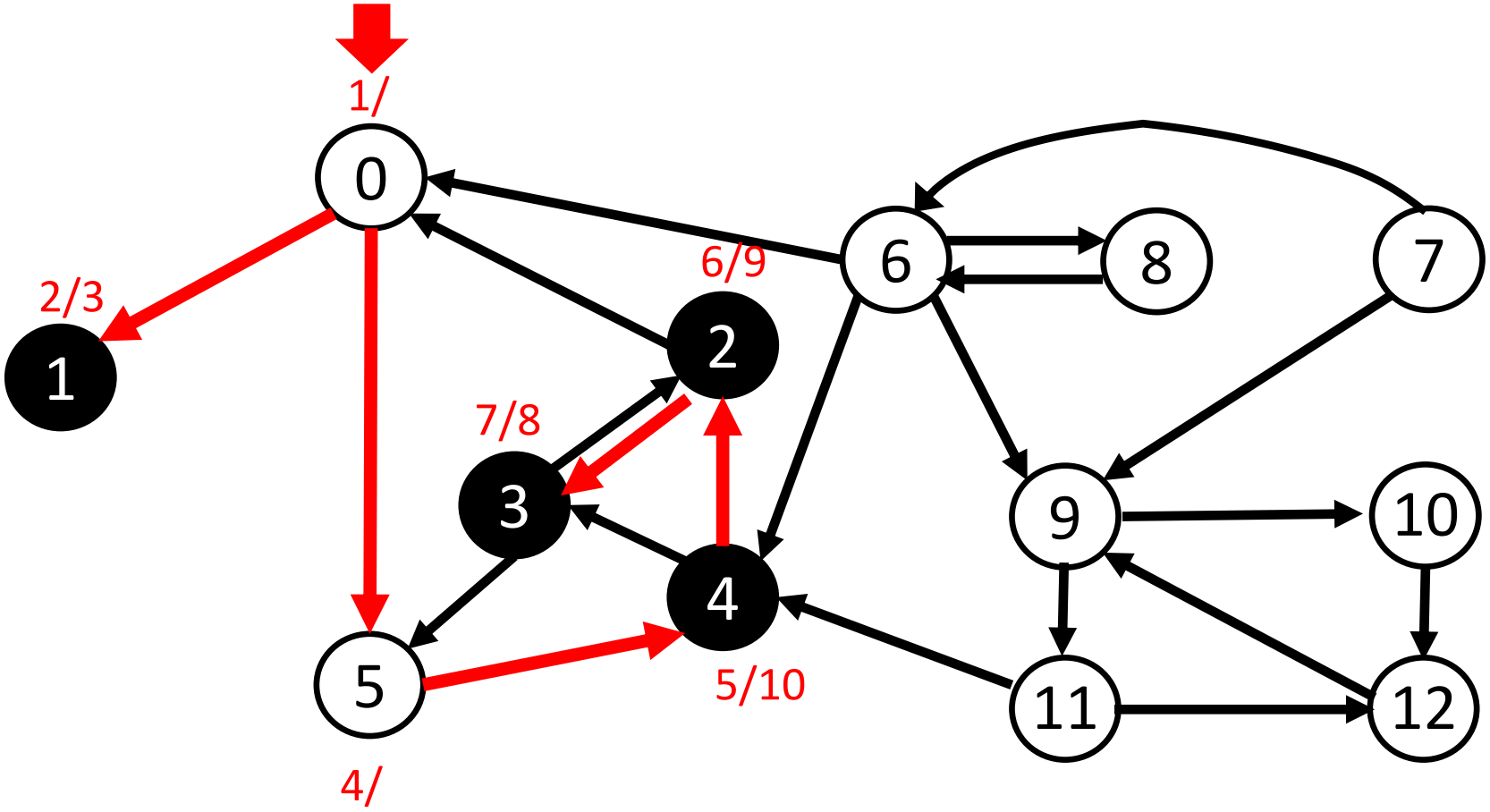


Strong components

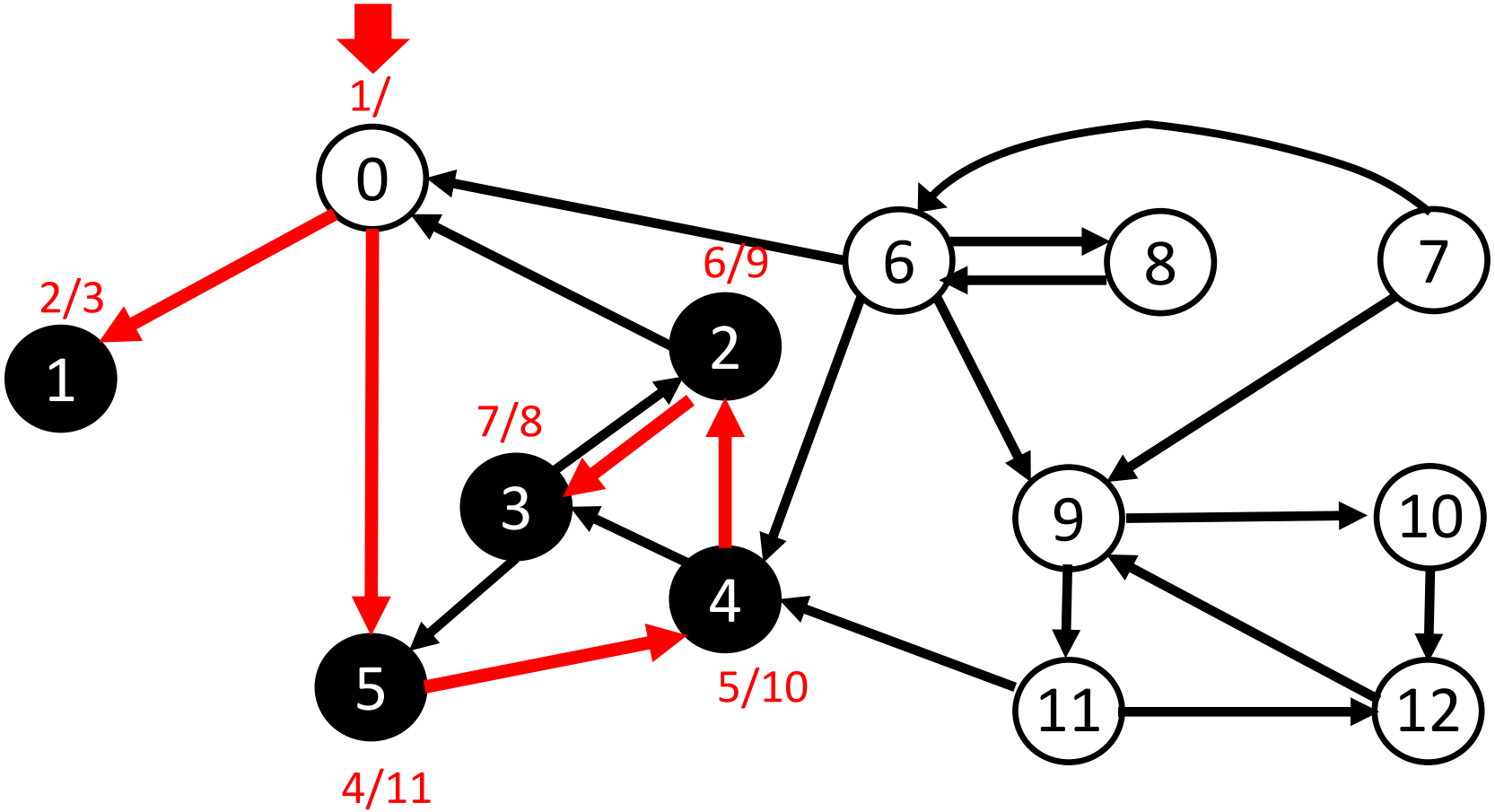


26

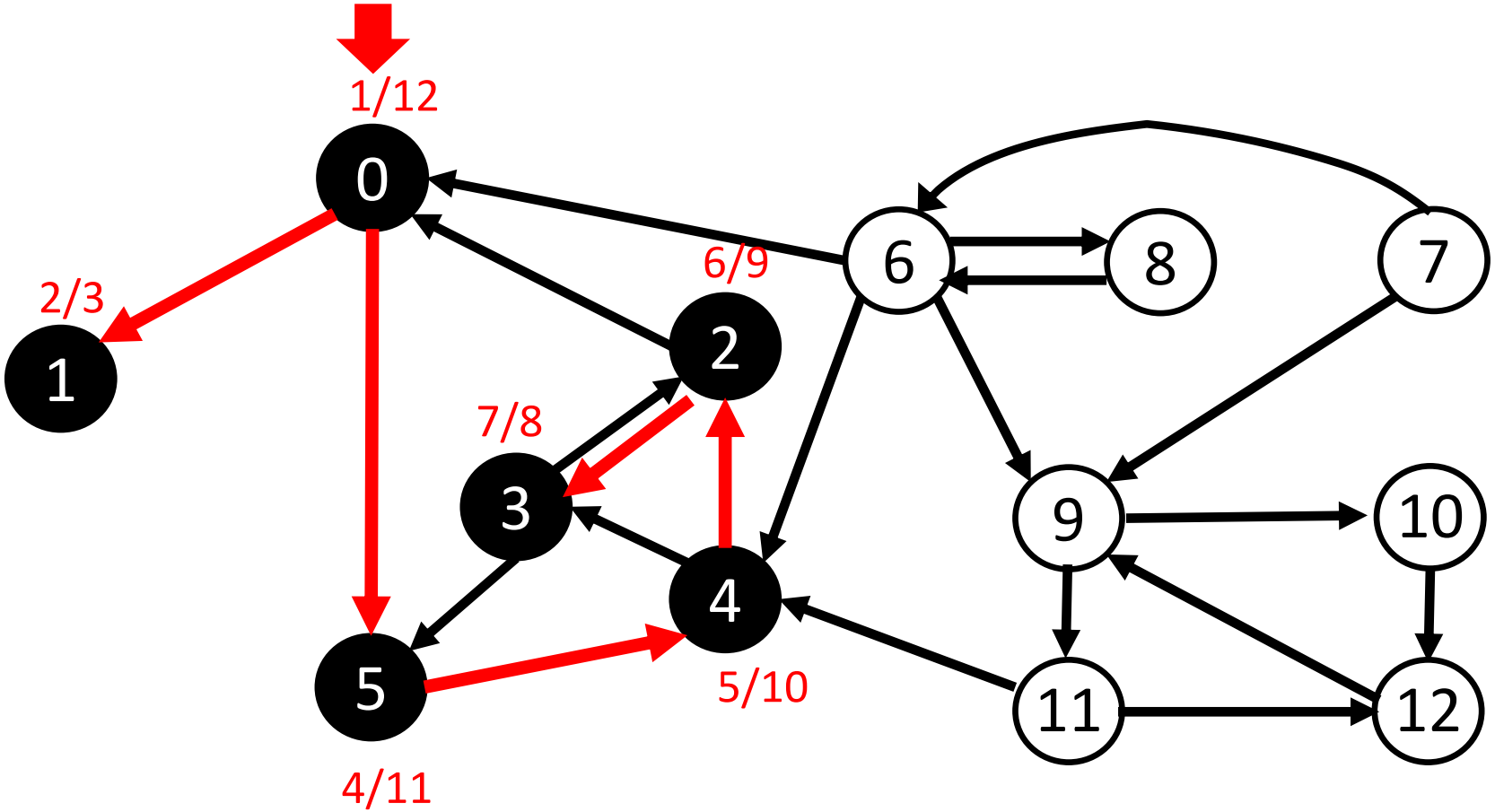
Strong components



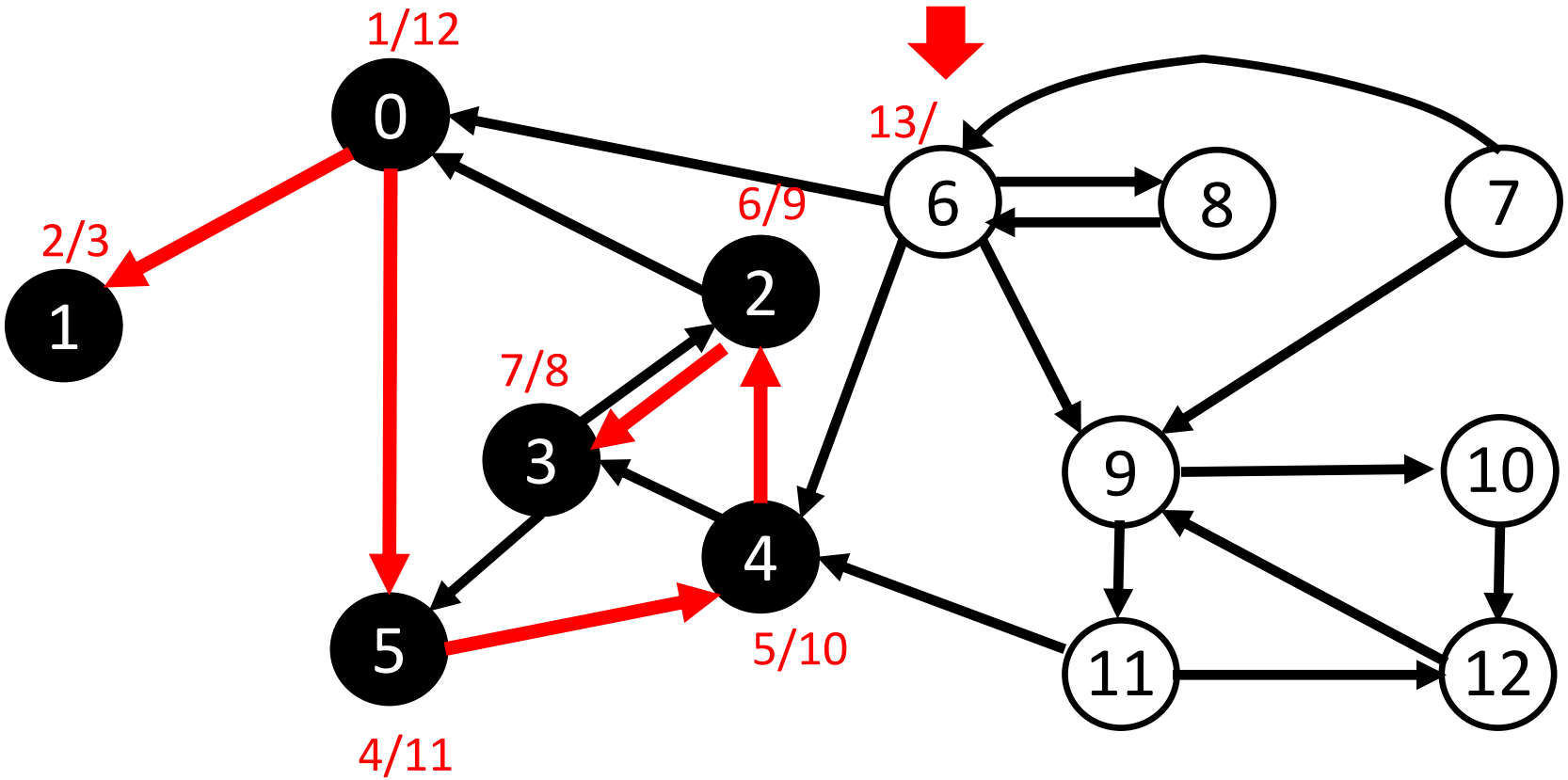
Strong components



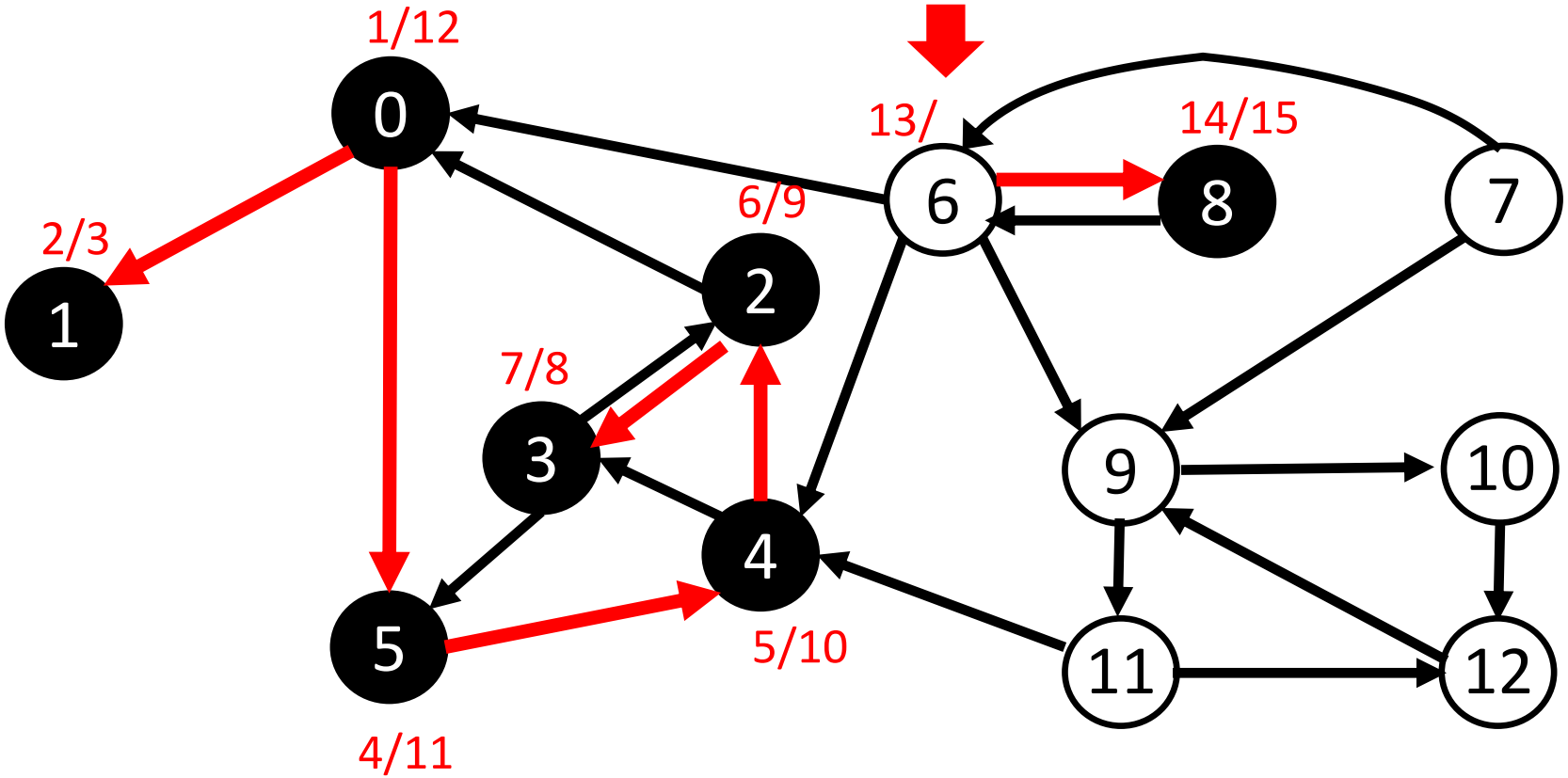
Strong components



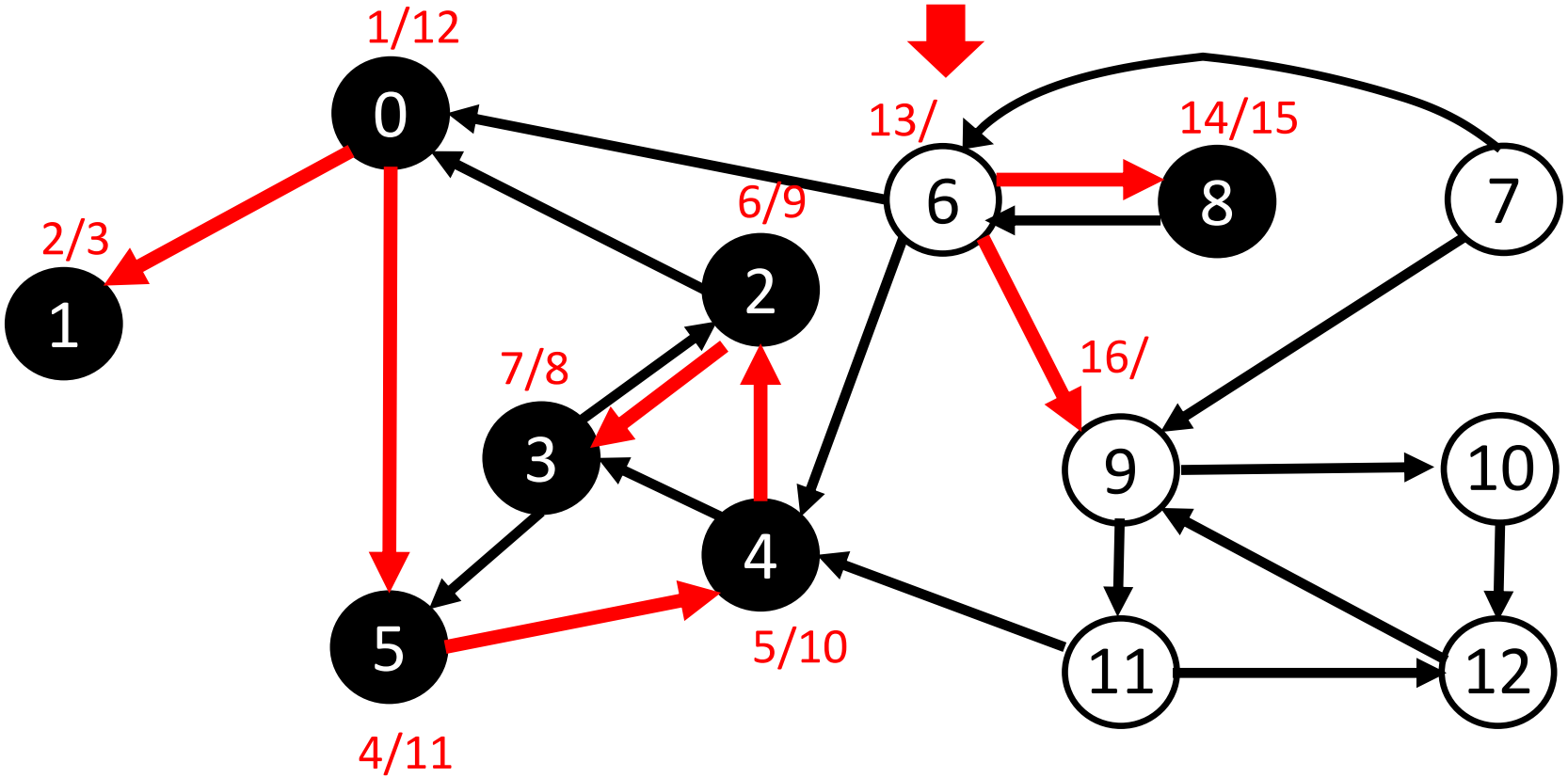
Strong components



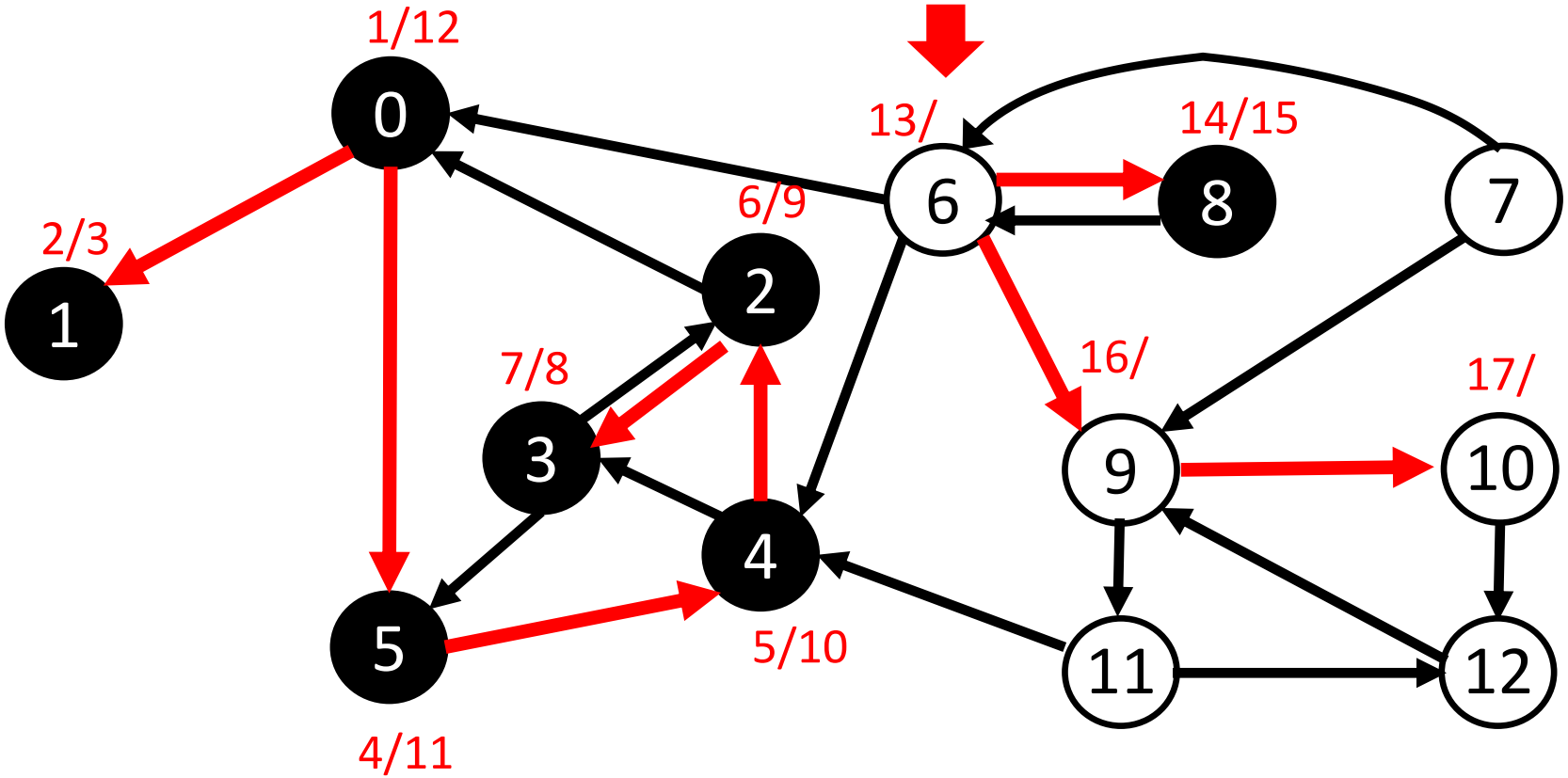
Strong components



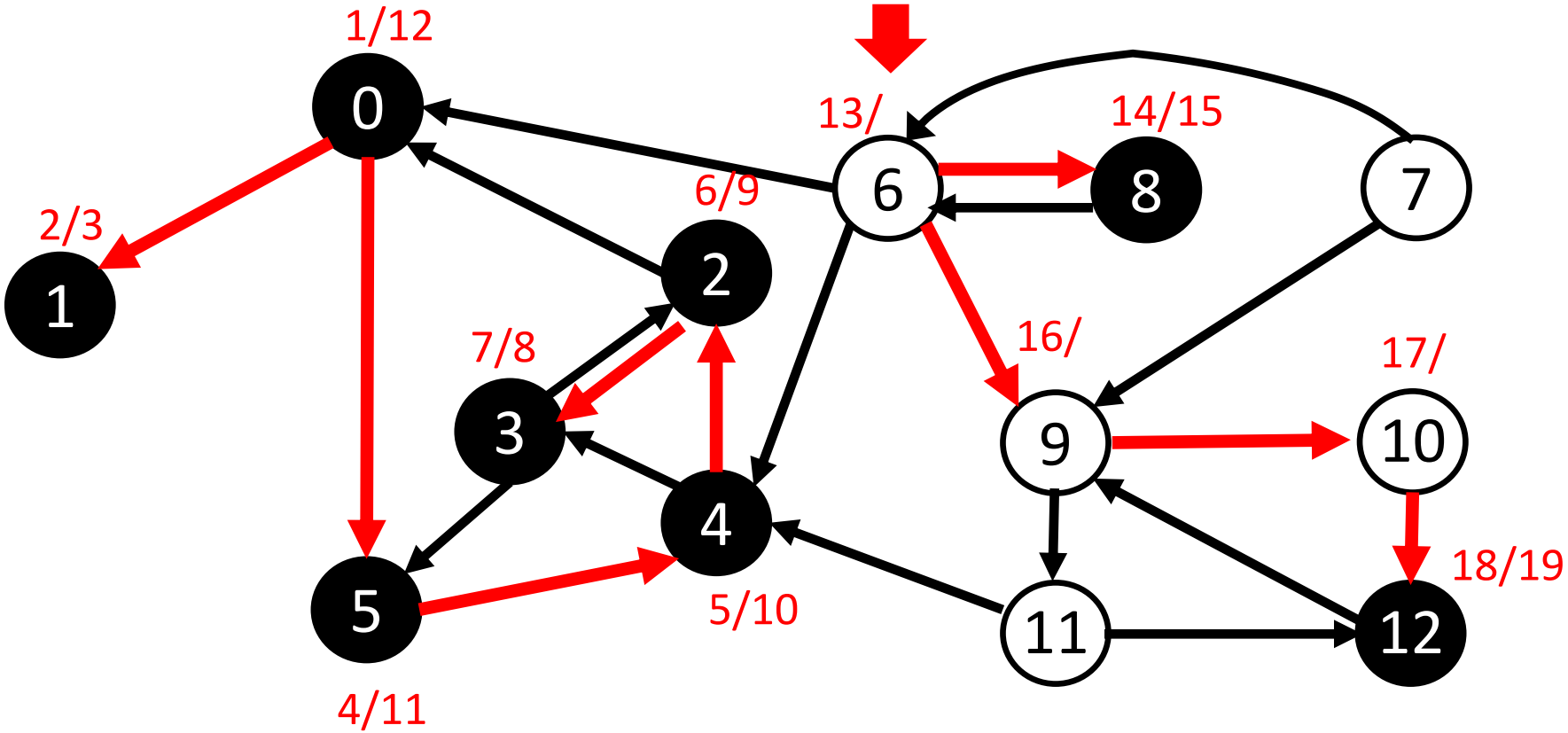
Strong components



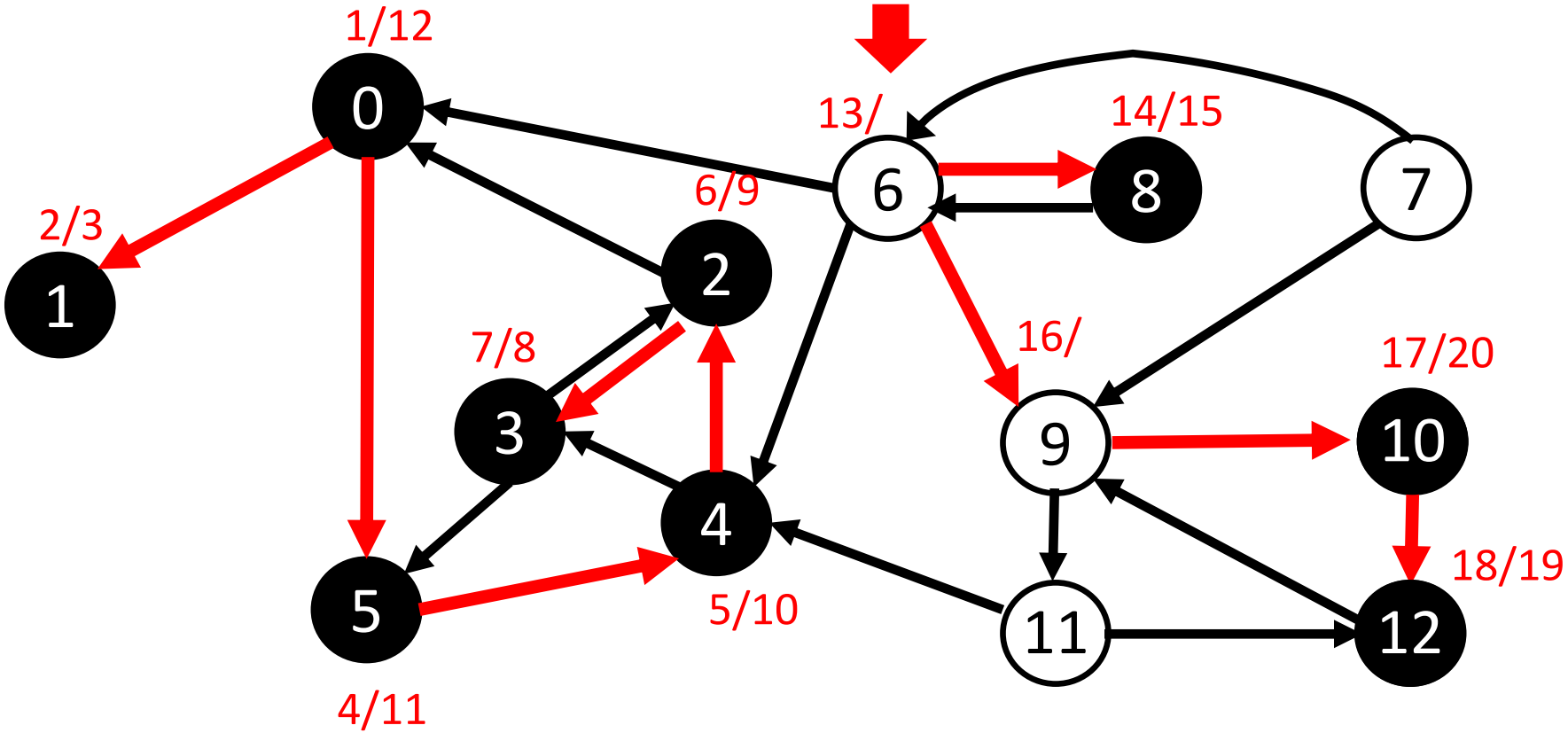
Strong components



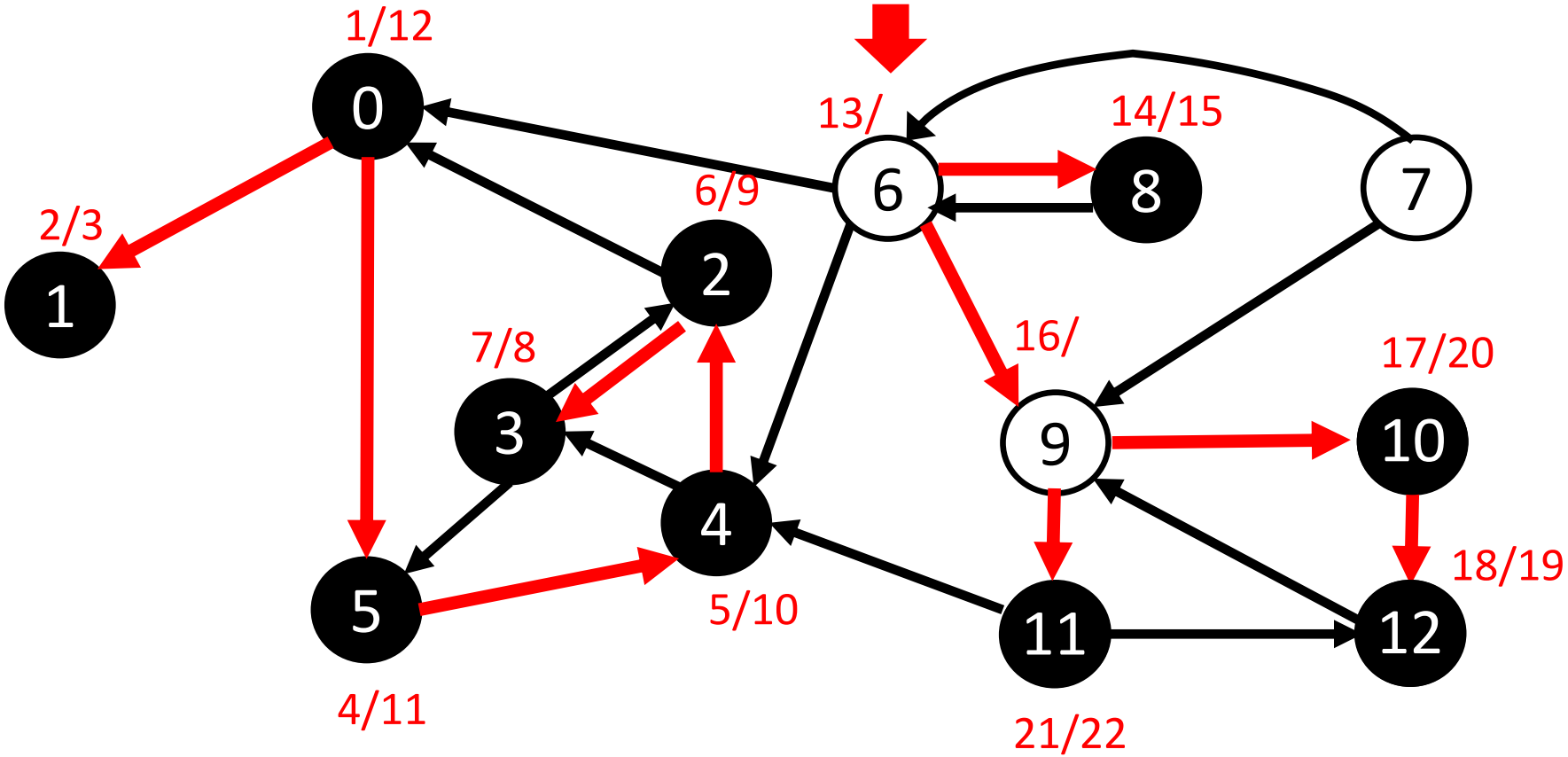
Strong components



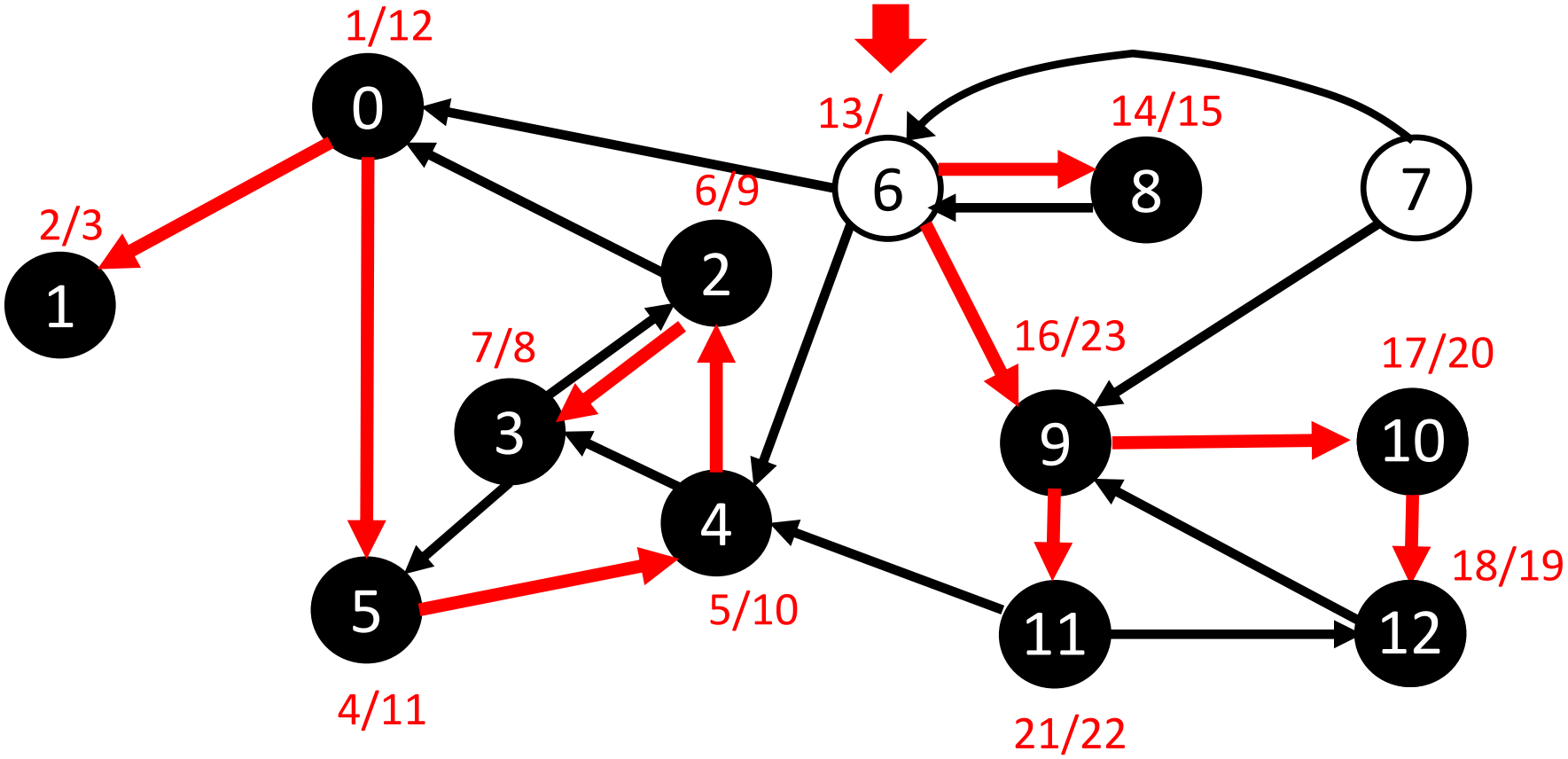
Strong components



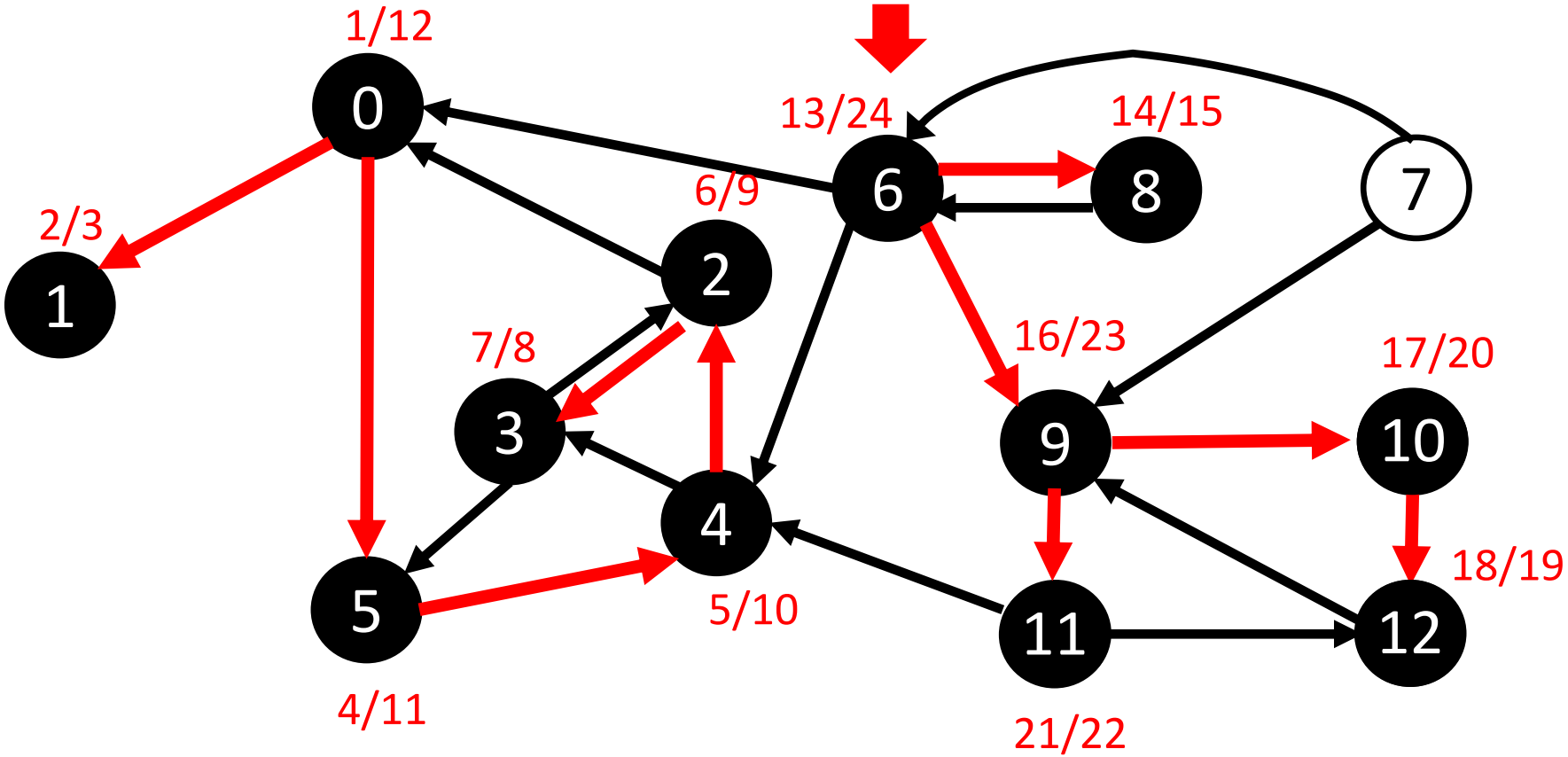
Strong components



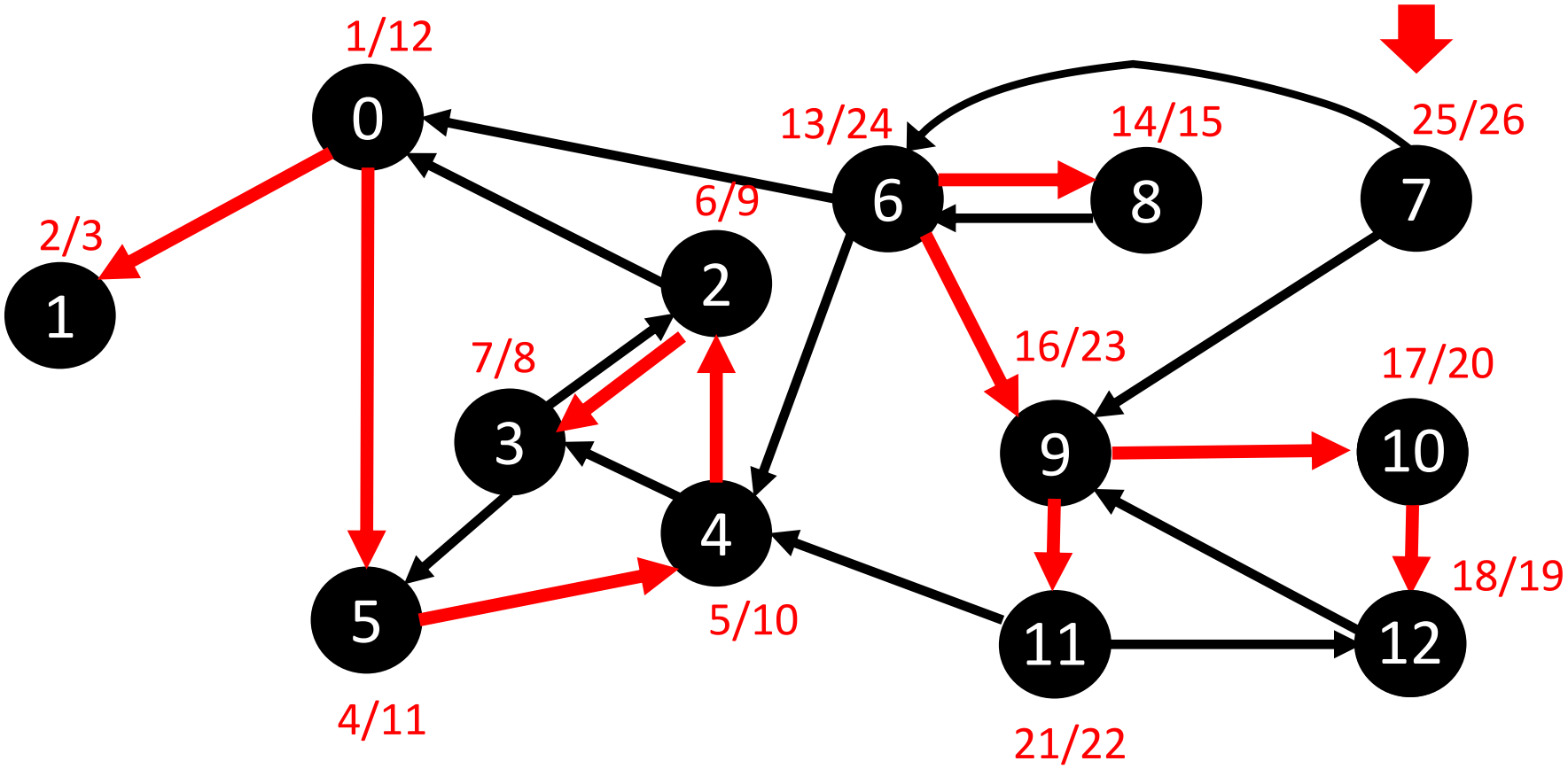
Strong components



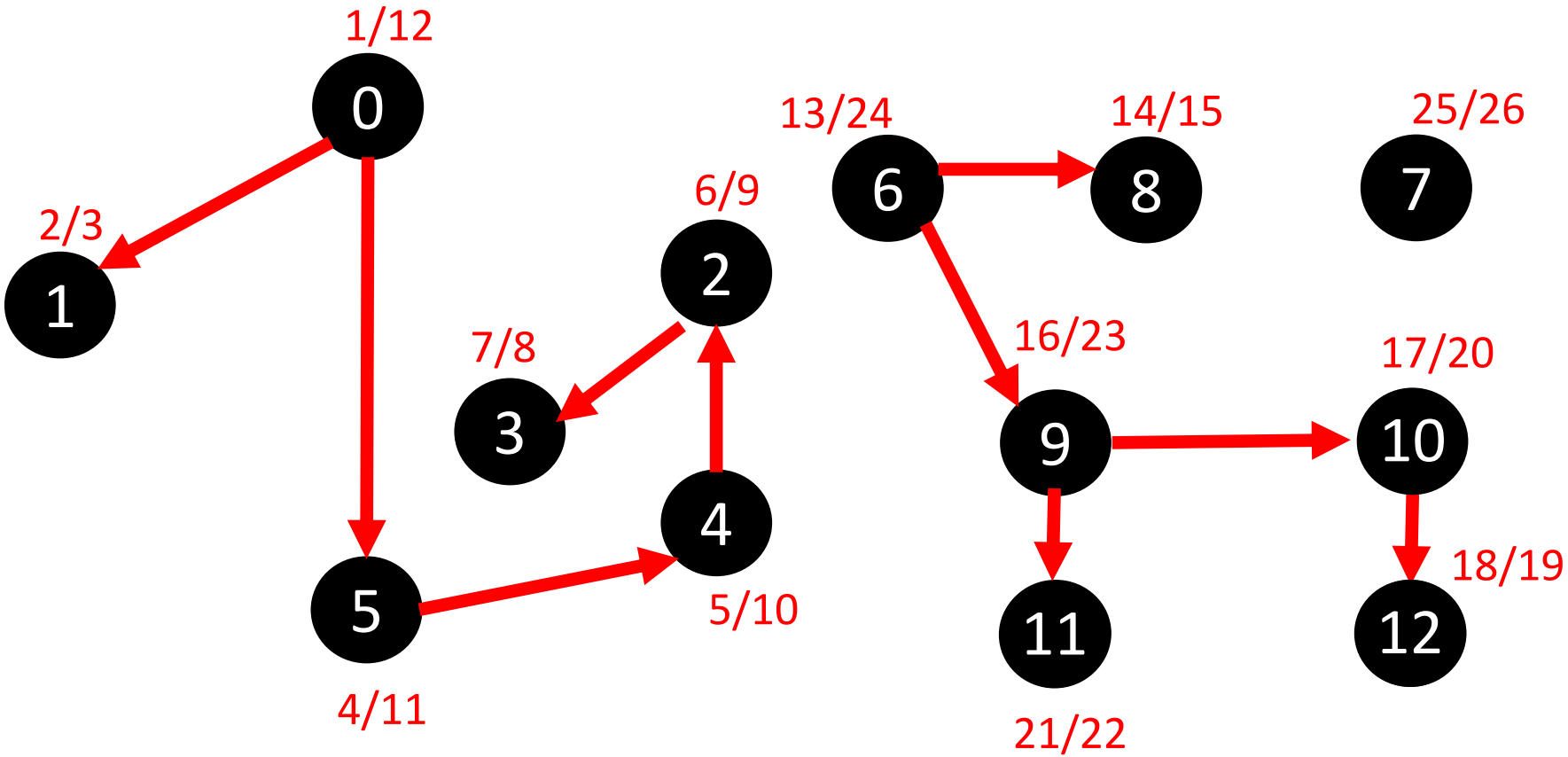
Strong components



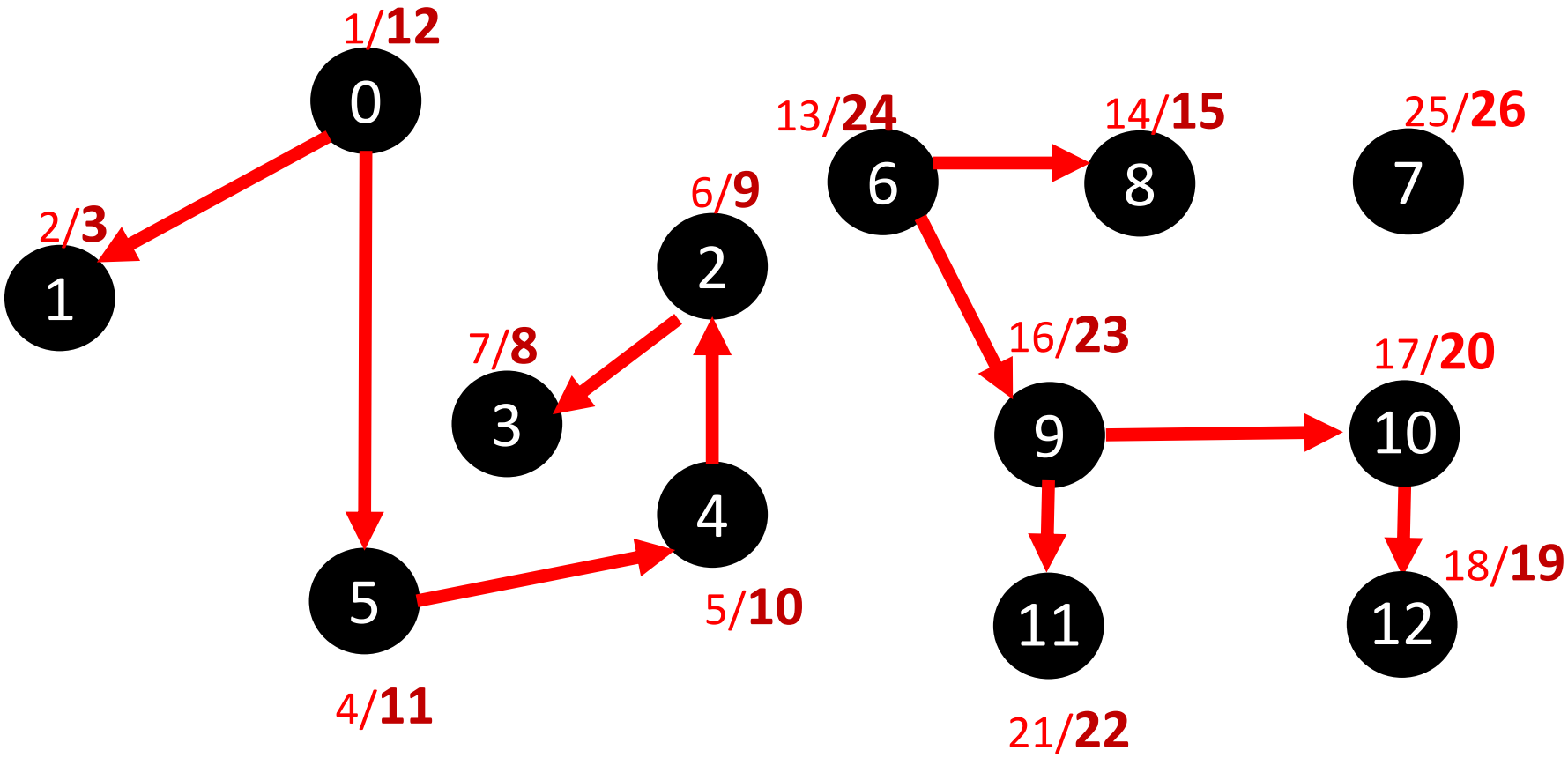
Strong components



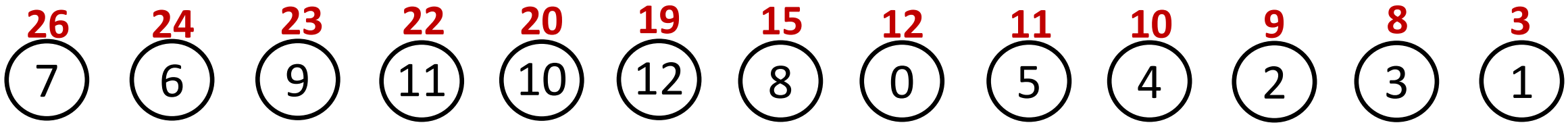
Strong components



Strong components

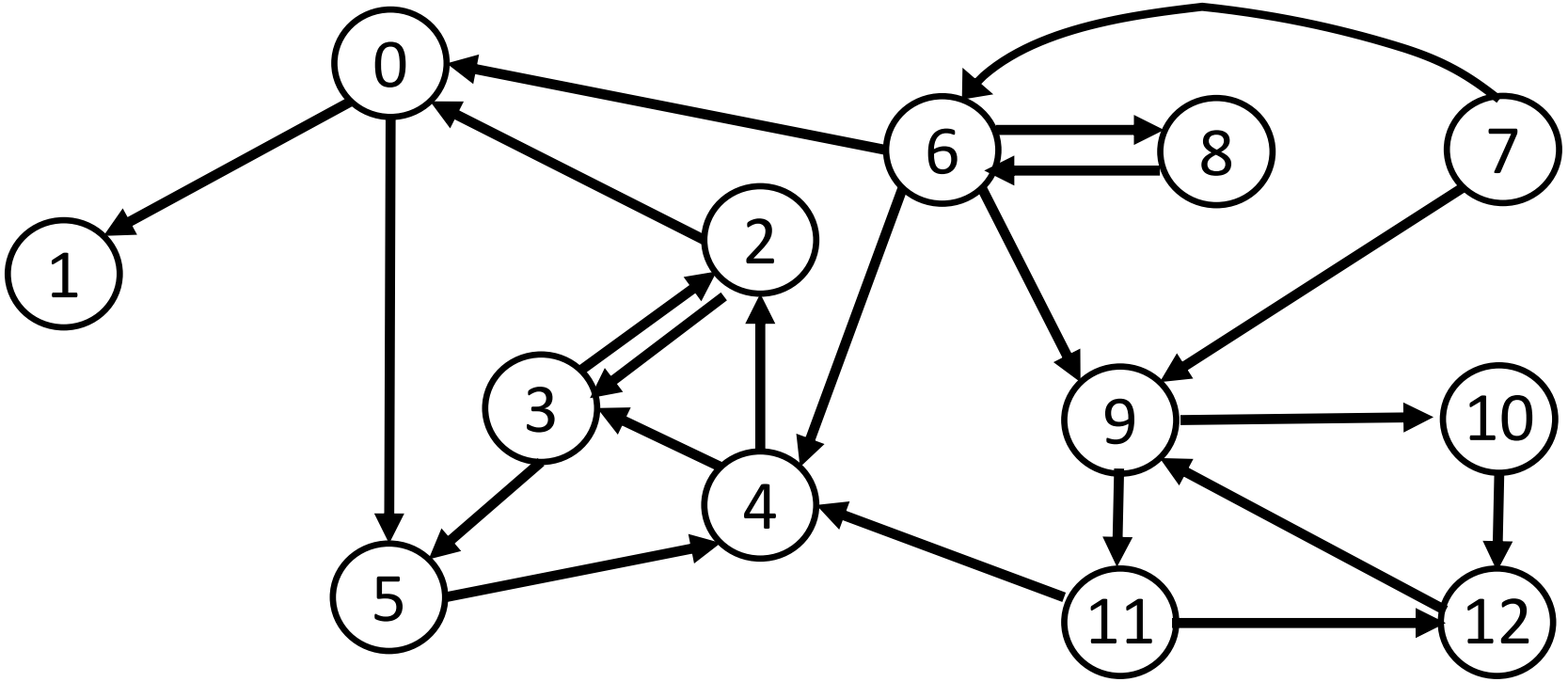


Strong components



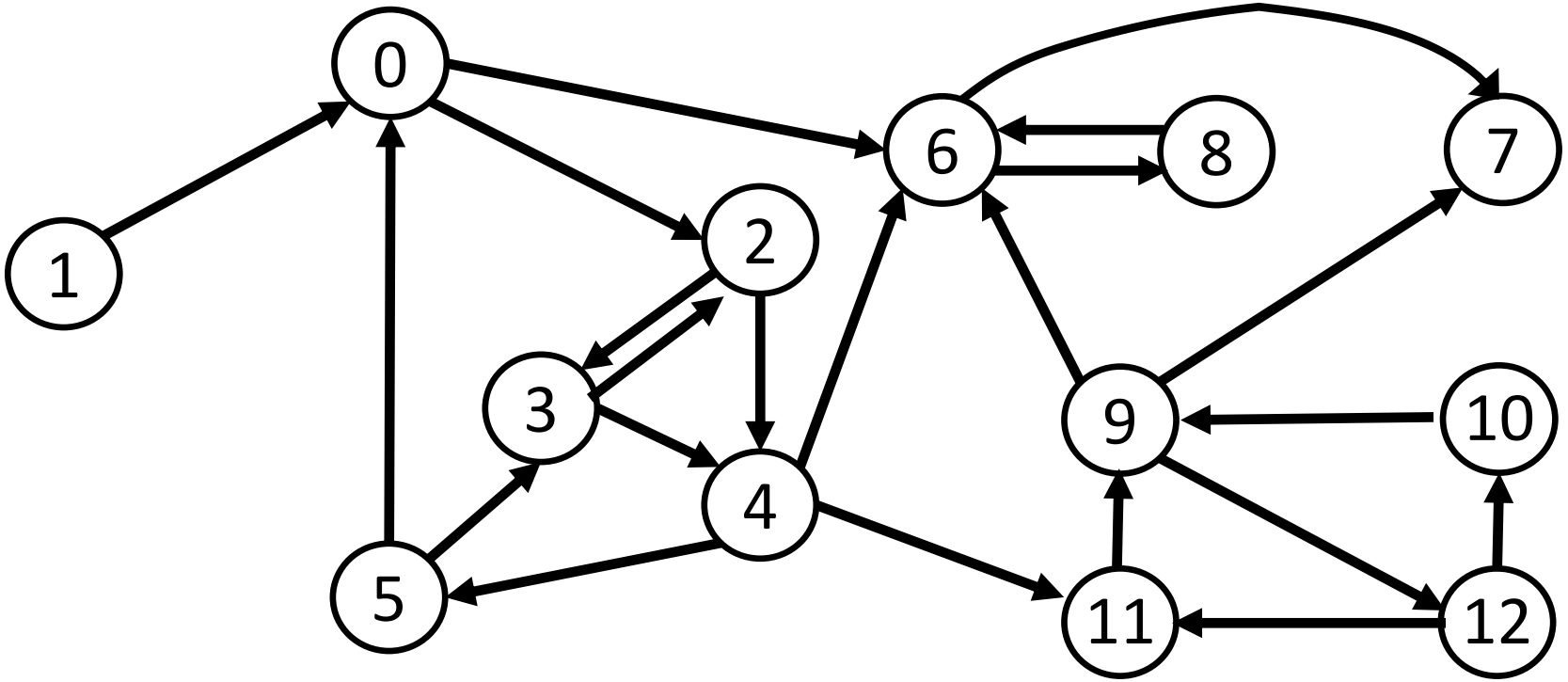
Strong components

26	24	23	22	20	19	15	12	11	10	9	8	3
7	6	9	11	10	12	8	0	5	4	2	3	1



Strong components

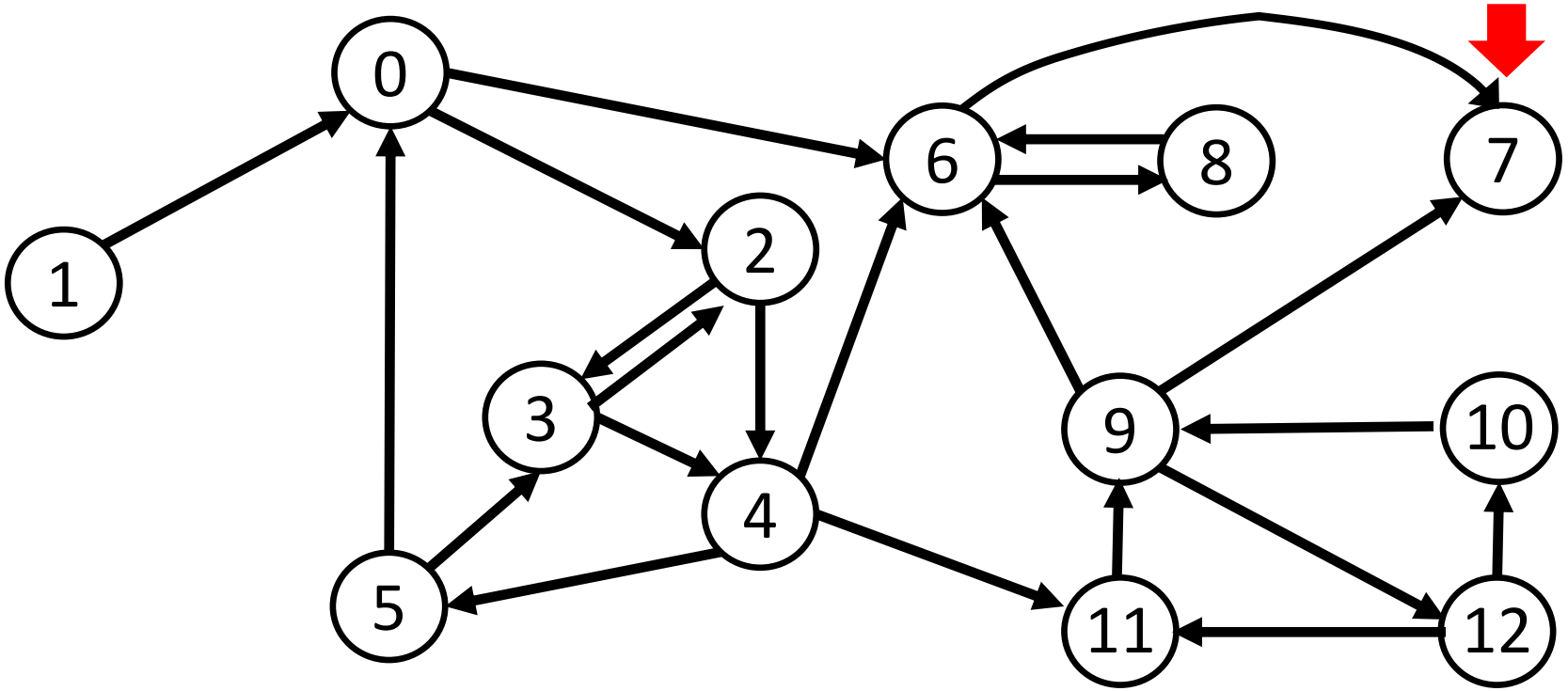
26	24	23	22	20	19	15	12	11	10	9	8	3
7	6	9	11	10	12	8	0	5	4	2	3	1



44

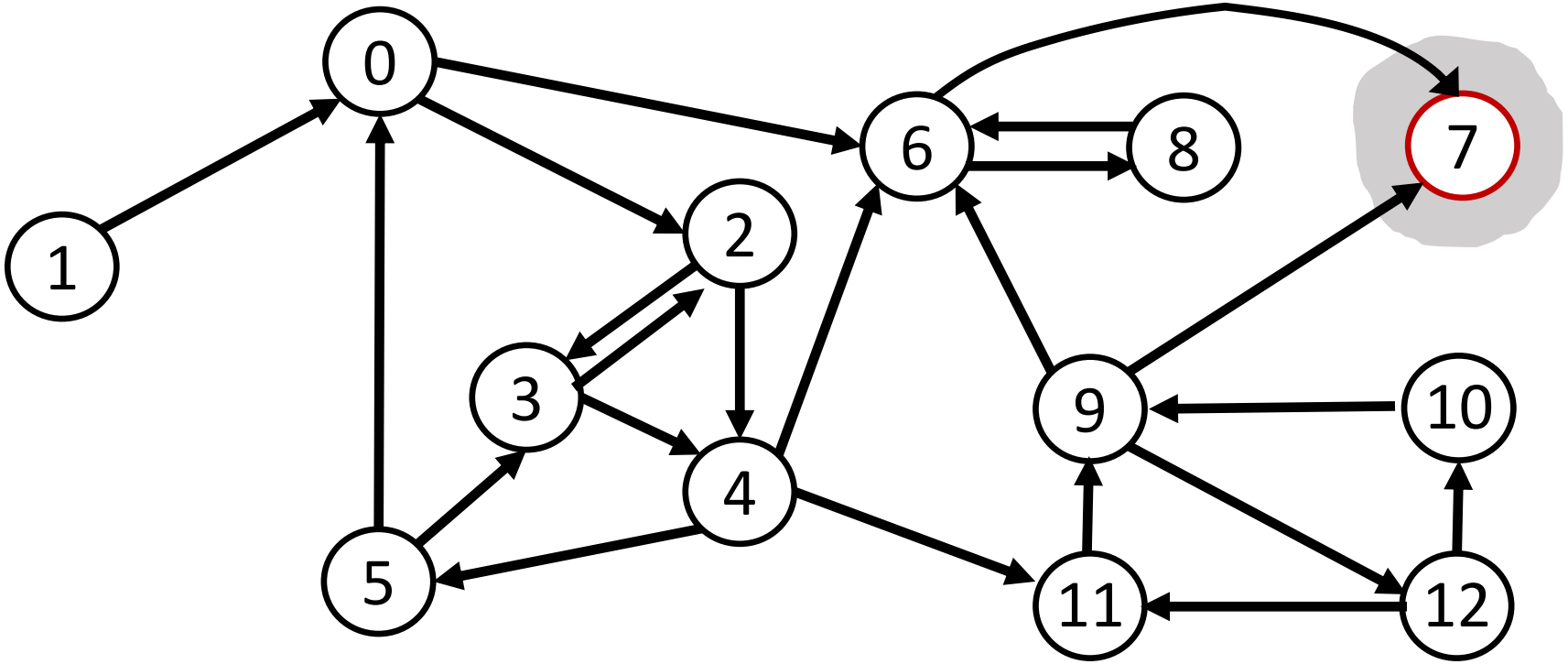
Strong components

26	24	23	22	20	19	15	12	11	10	9	8	3
~~X~~	6	9	11	10	12	8	0	5	4	2	3	1



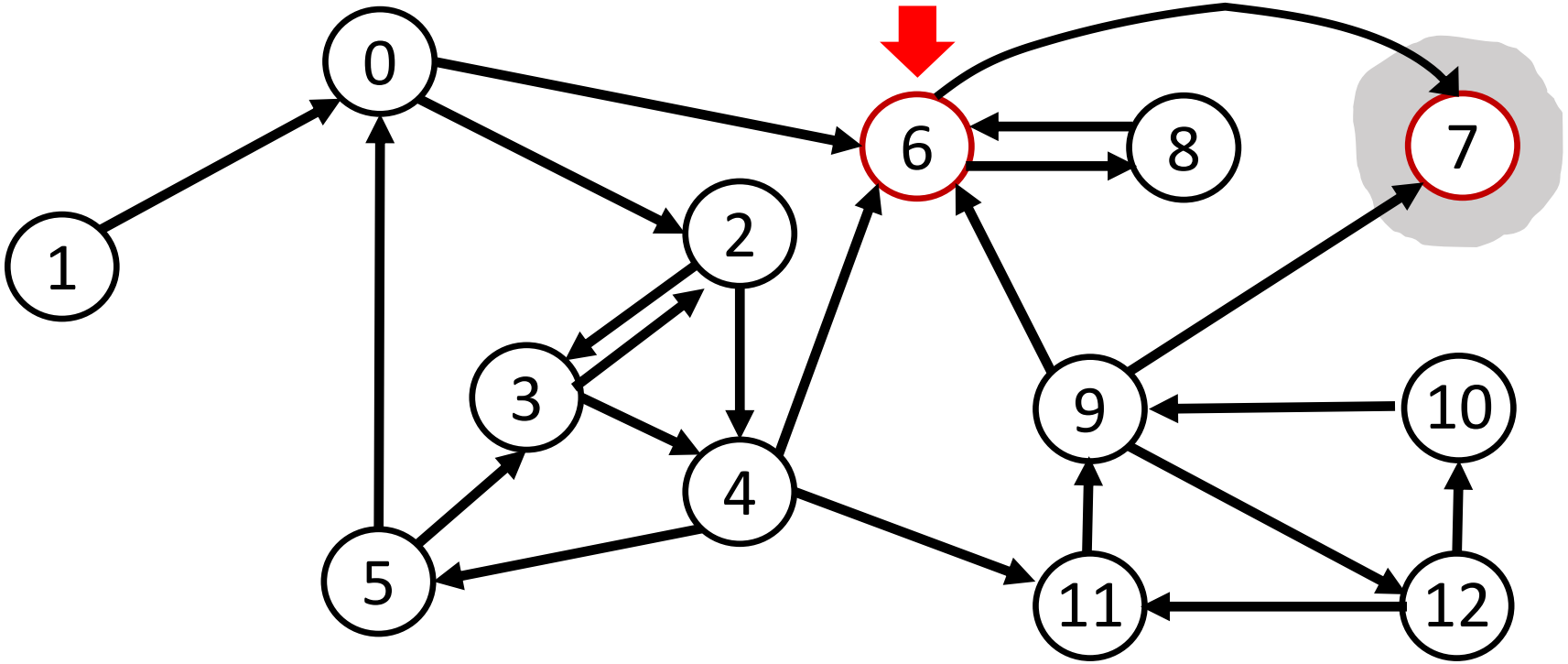
Strong components

26	24	23	22	20	19	15	12	11	10	9	8	3
~~7~~	6	9	11	10	12	8	0	5	4	2	3	1

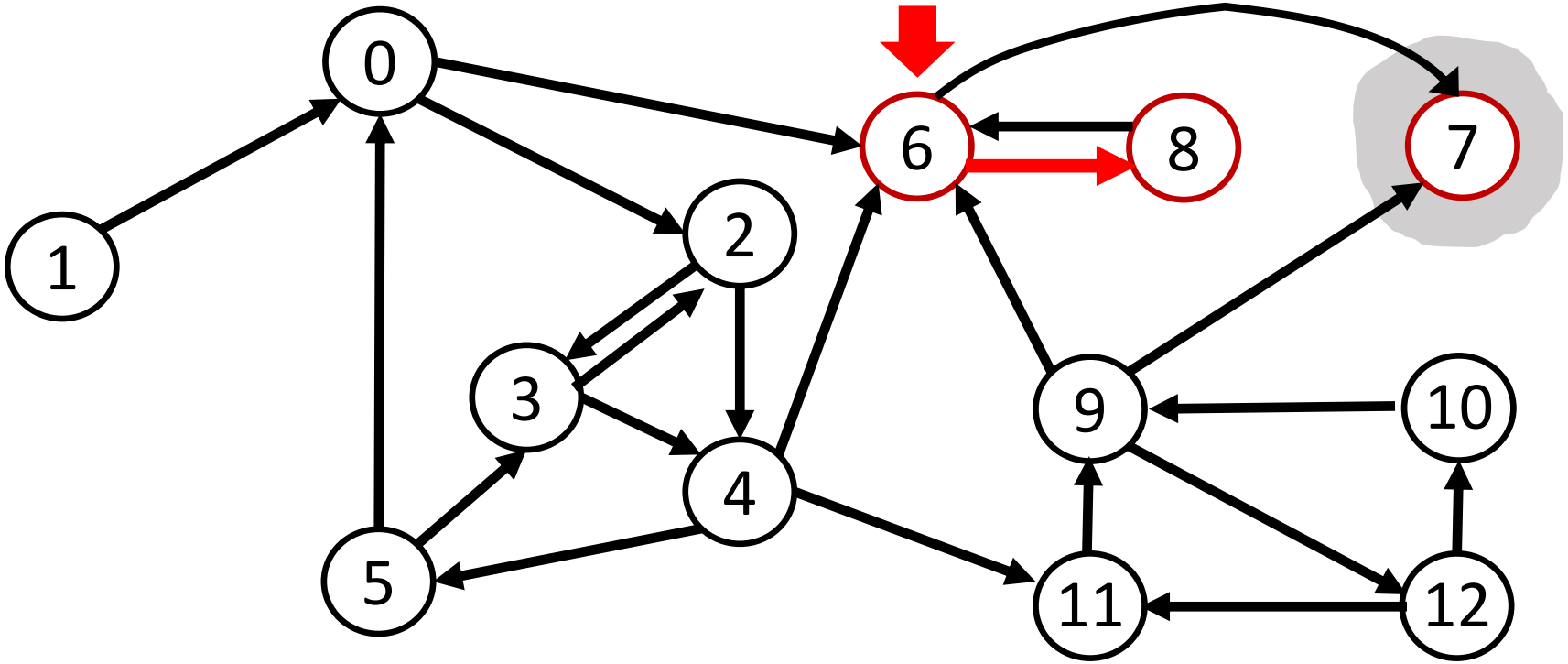


Strong components

26	24	23	22	20	19	15	12	11	10	9	8	3
~~9~~	~~11~~	9	11	10	12	8	0	5	4	2	3	1

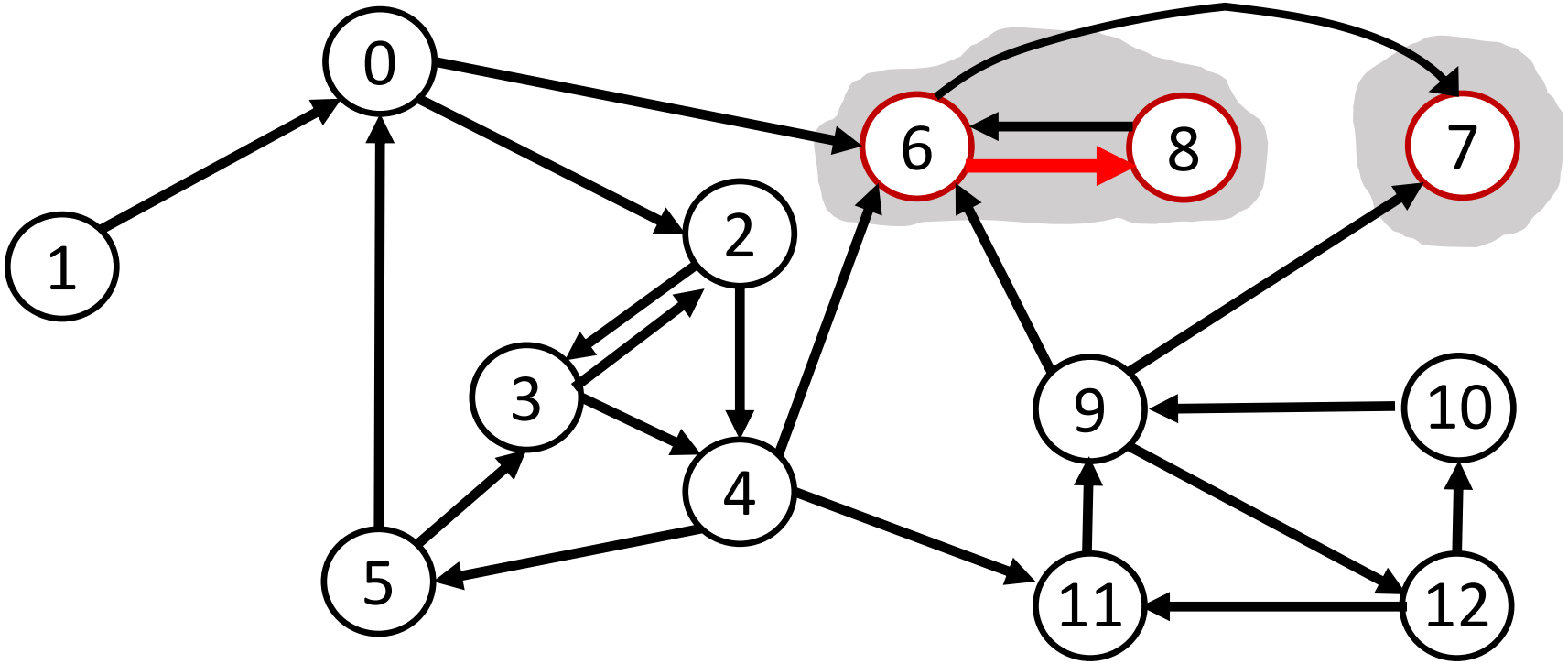


Strong components

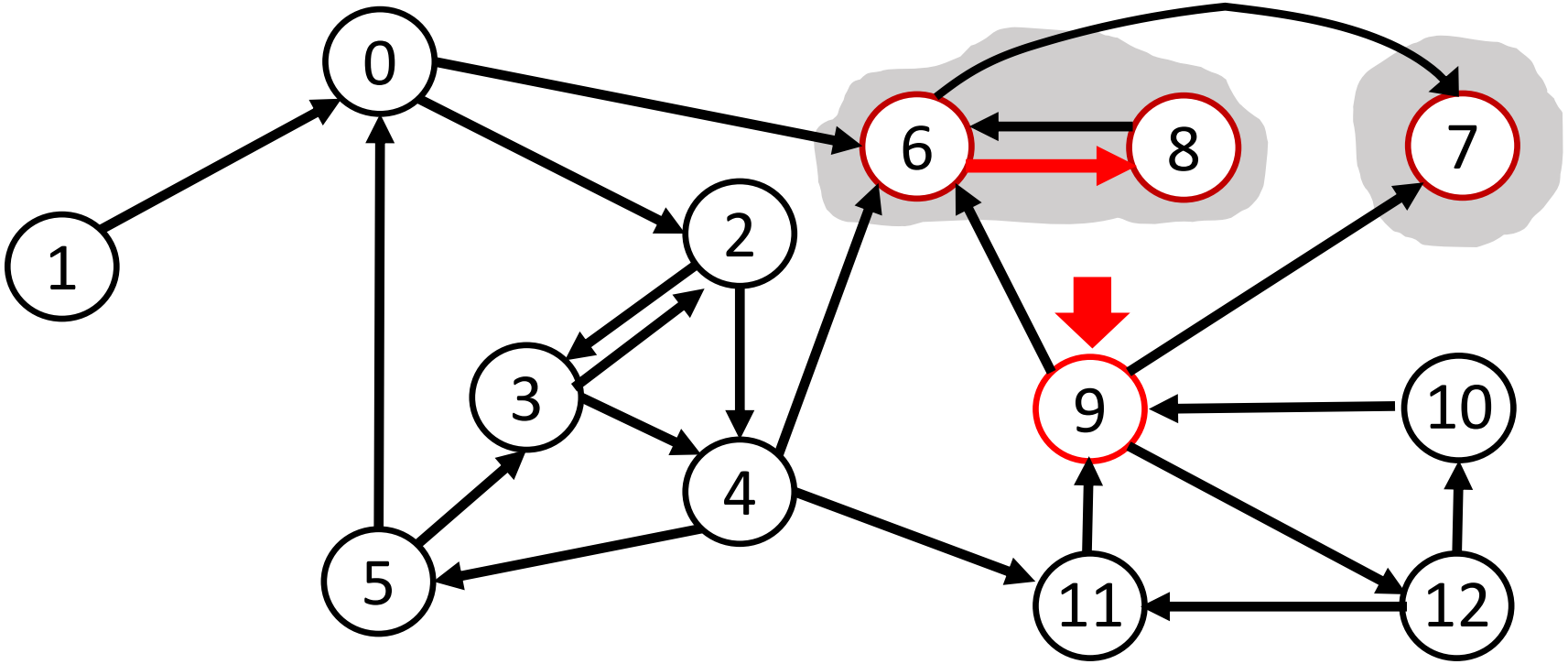


Strong components

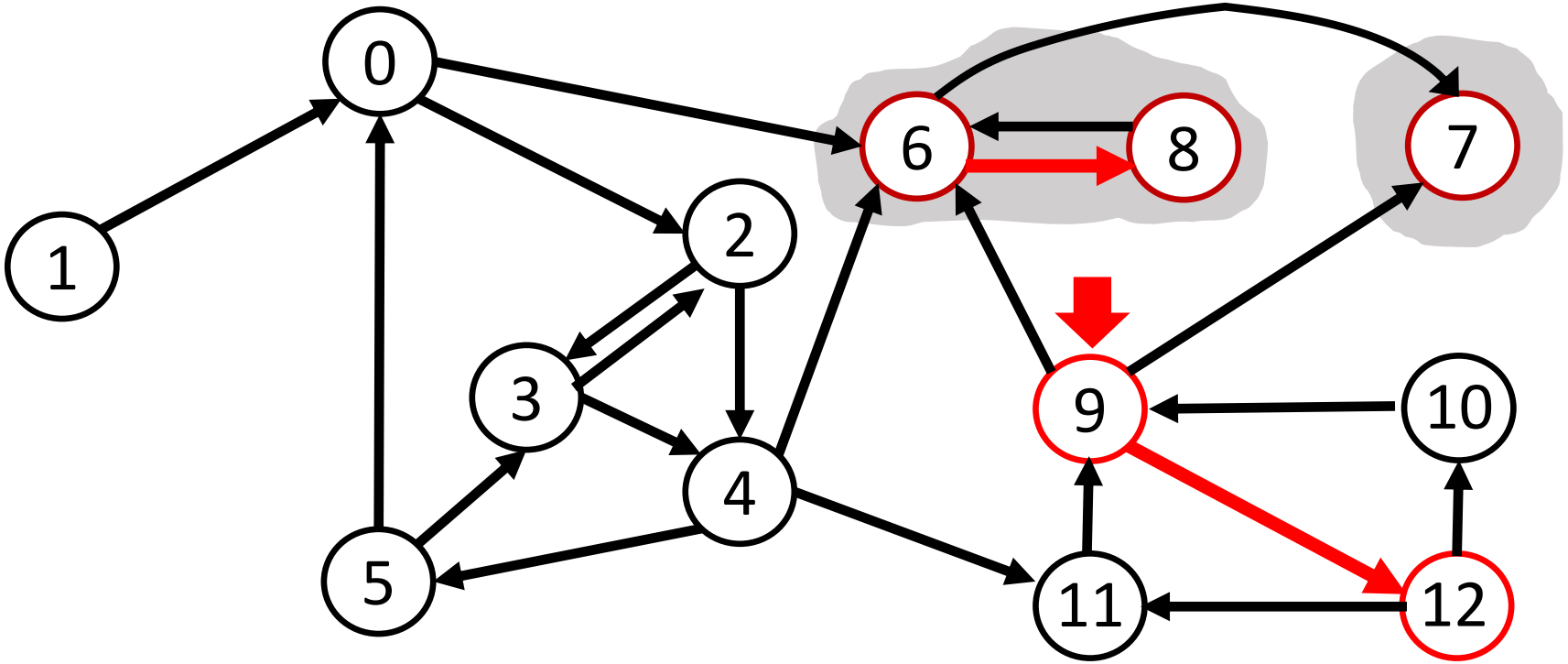
26	24	23	22	20	19	15	12	11	10	9	8	3
✗	✗	9	11	10	12	✗	0	5	4	2	3	1



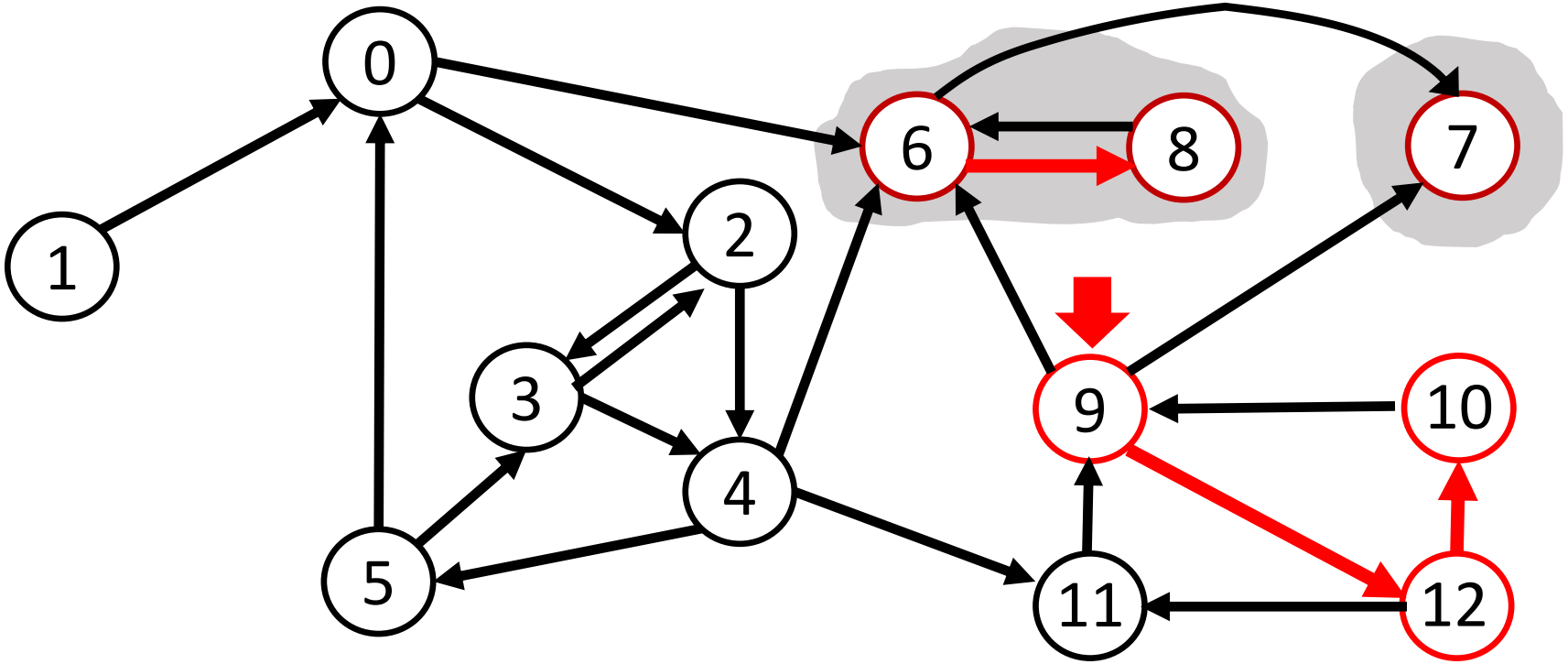
Strong components



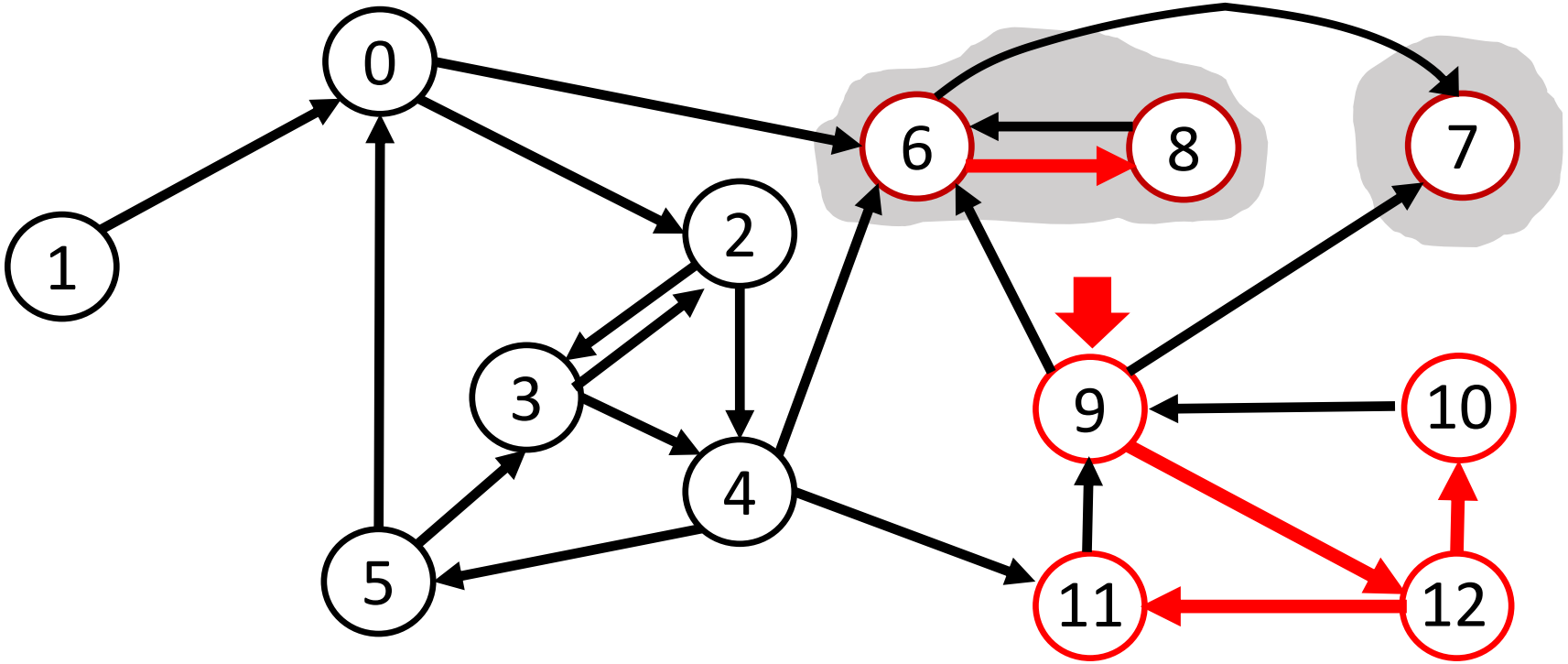
Strong components



Strong components

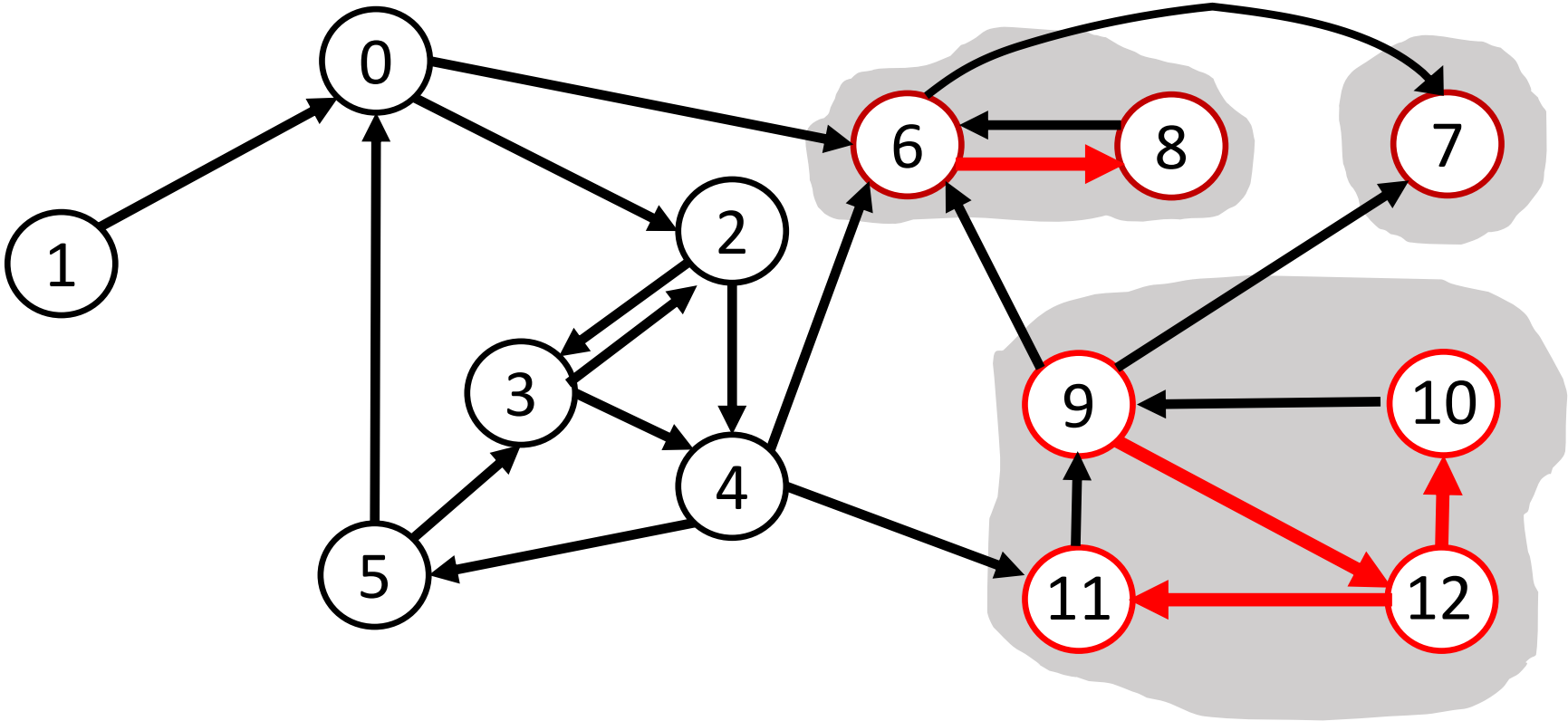


Strong components

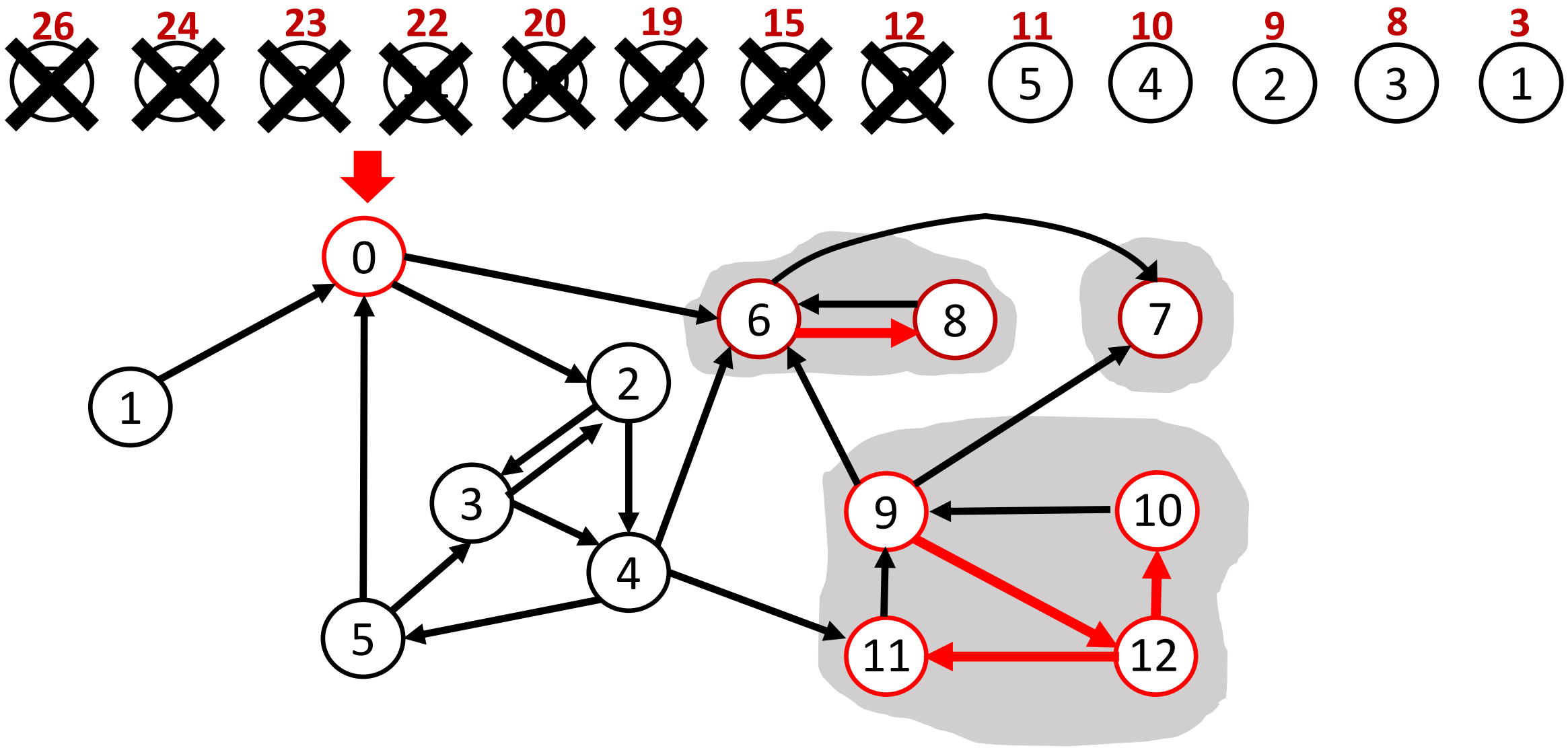


53

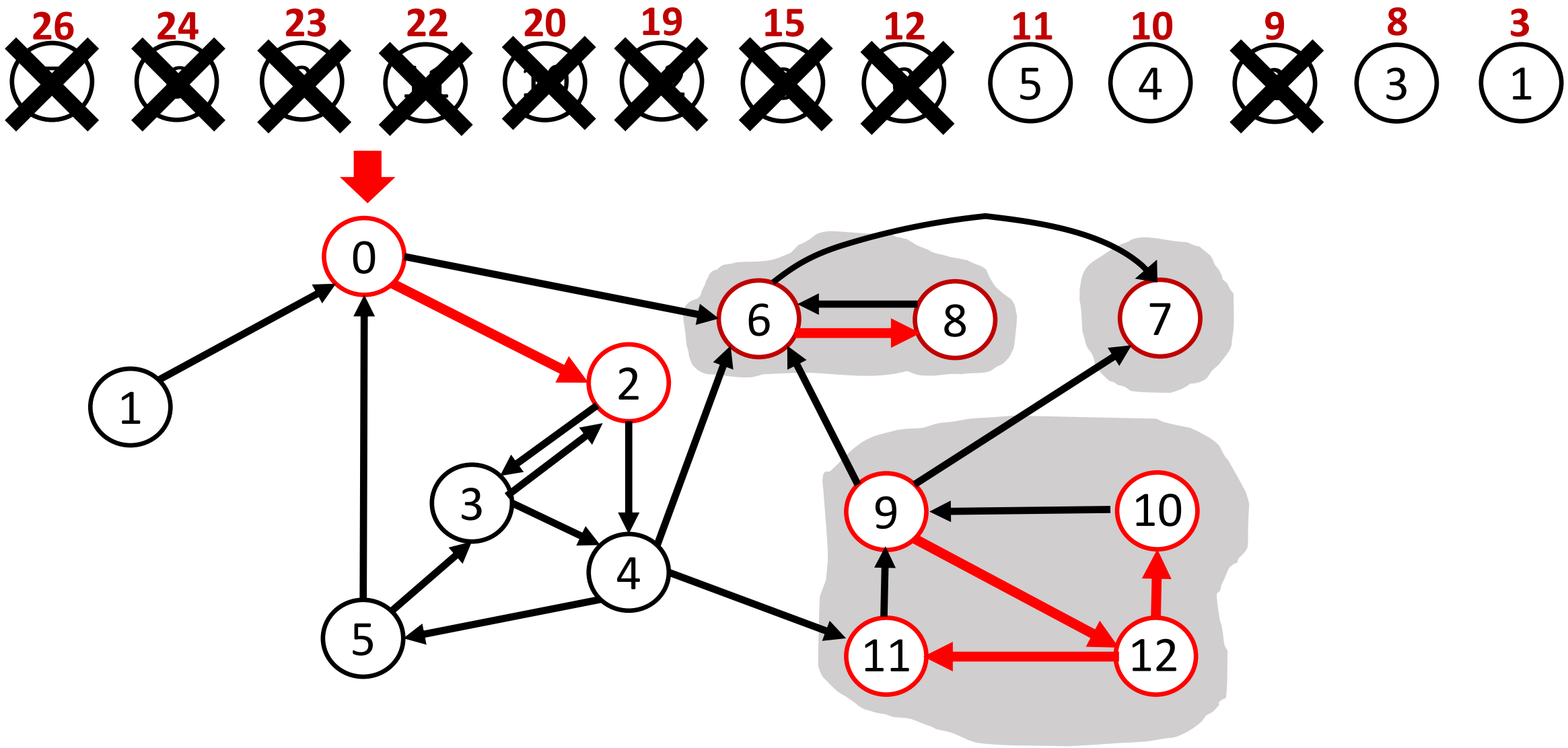
Strong components



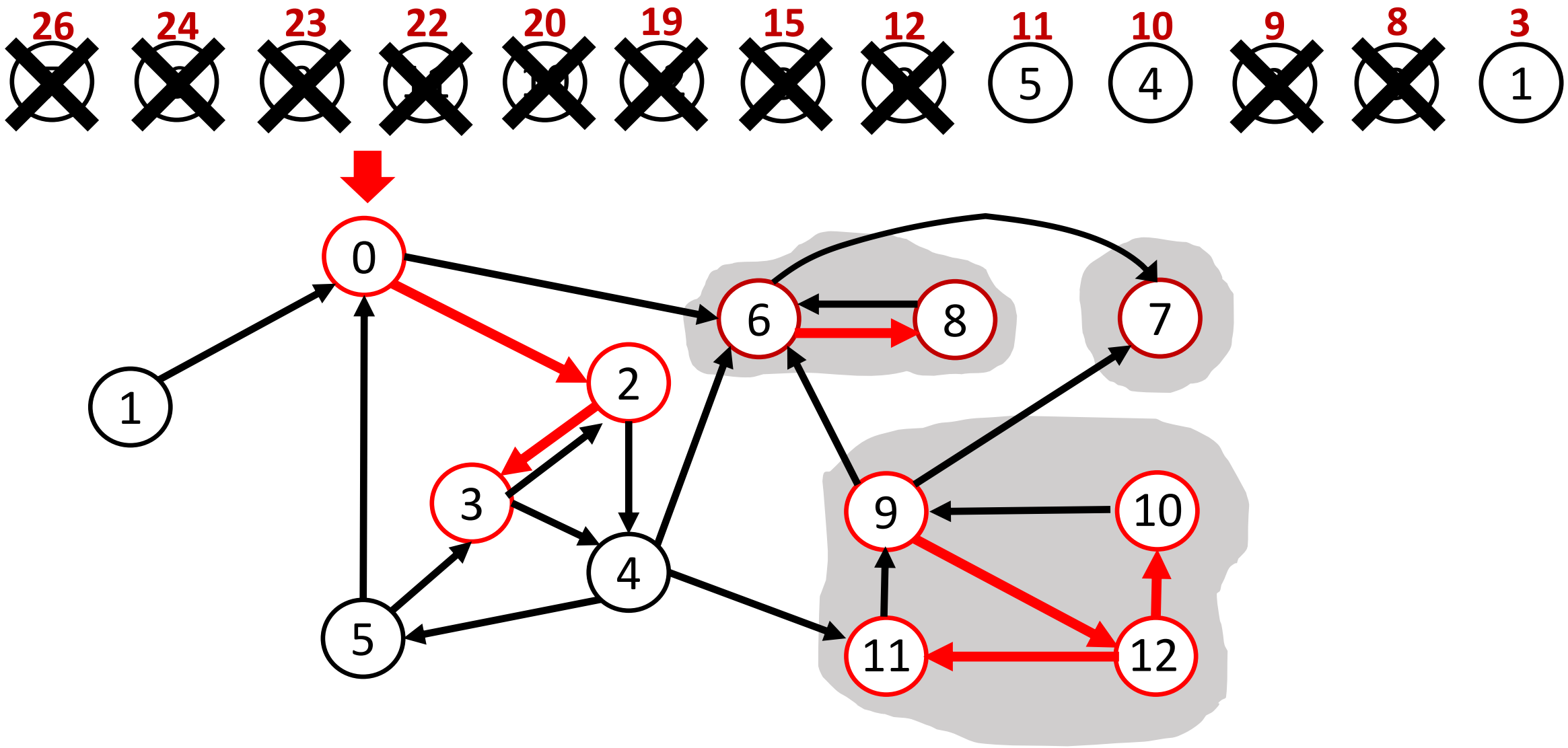
Strong components



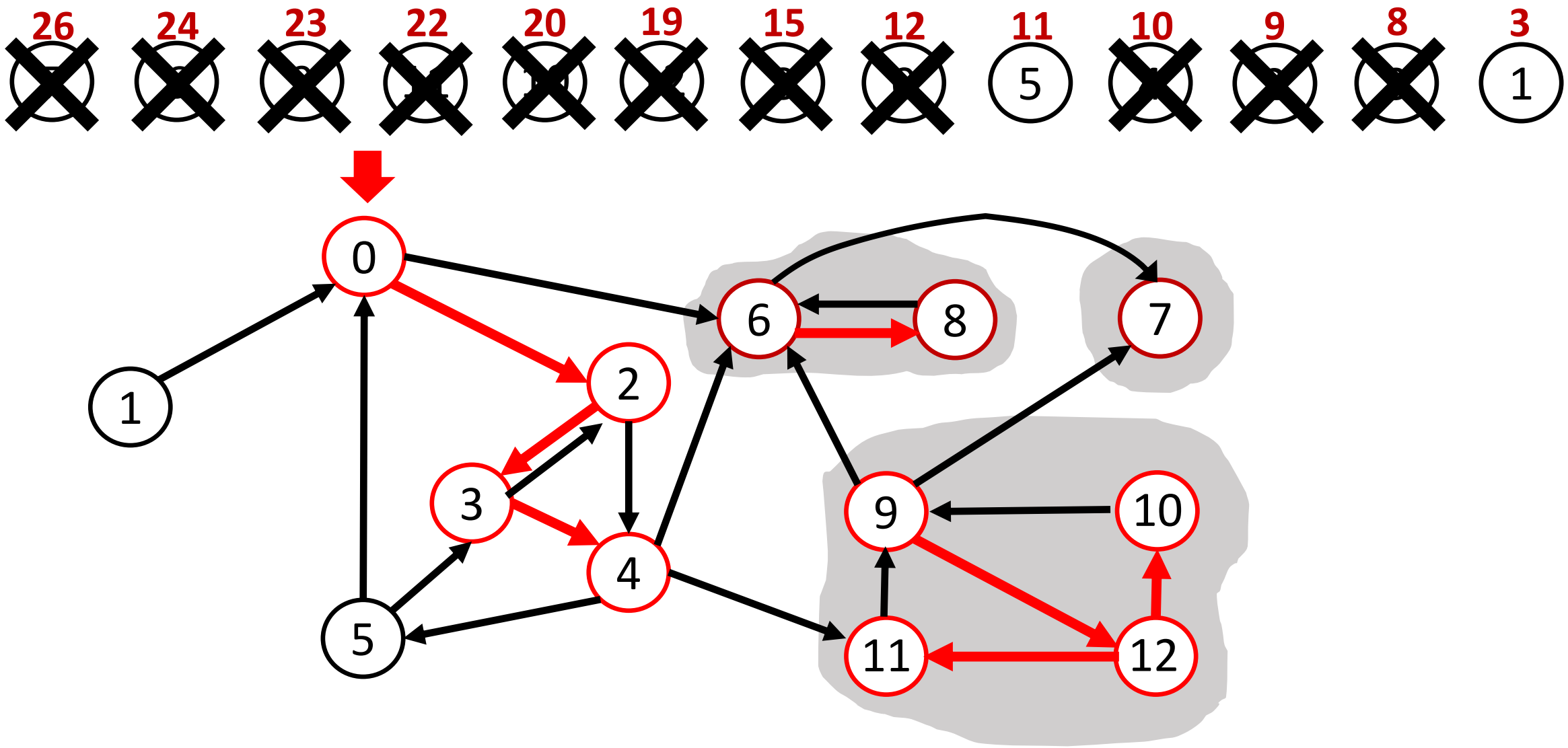
Strong components



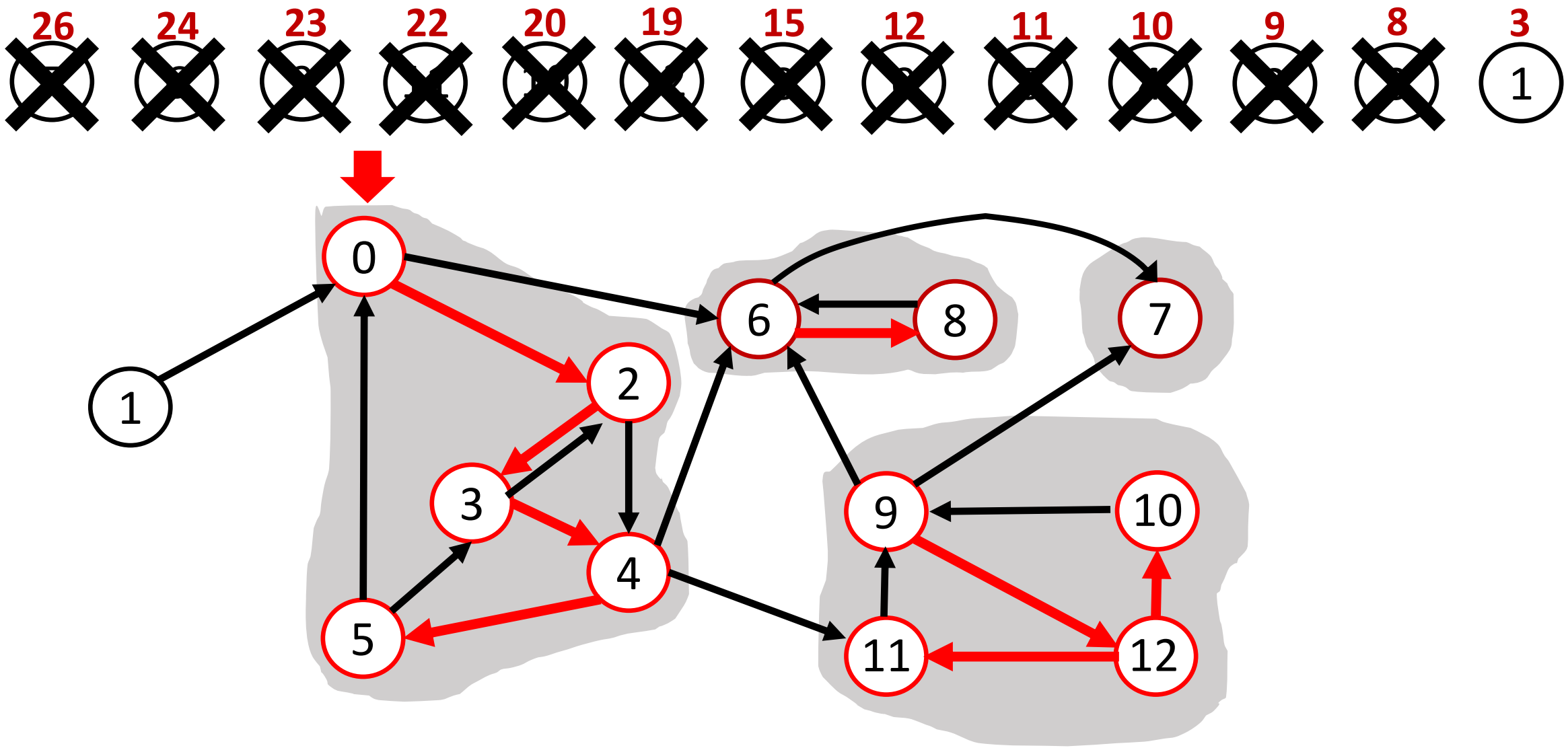
Strong components



Strong components

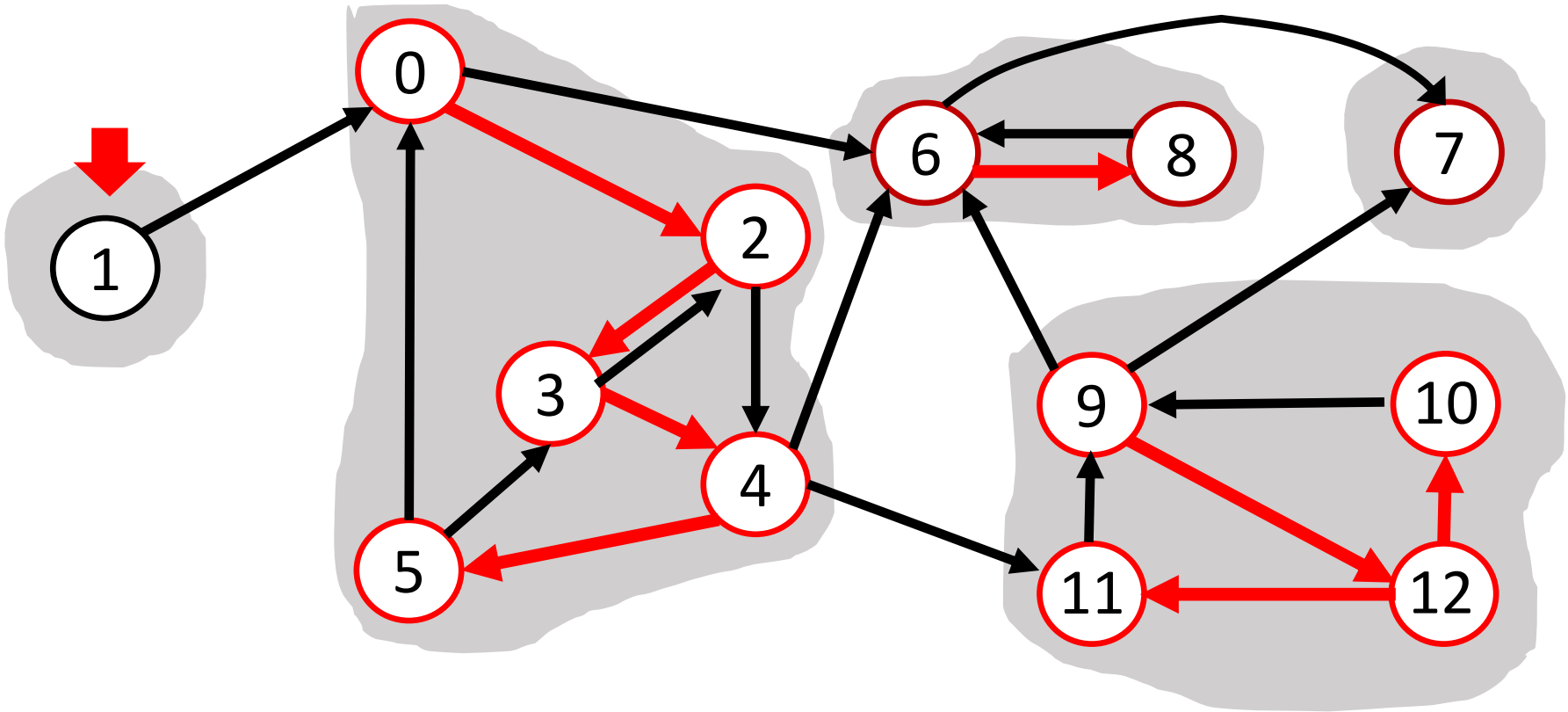


Strong components

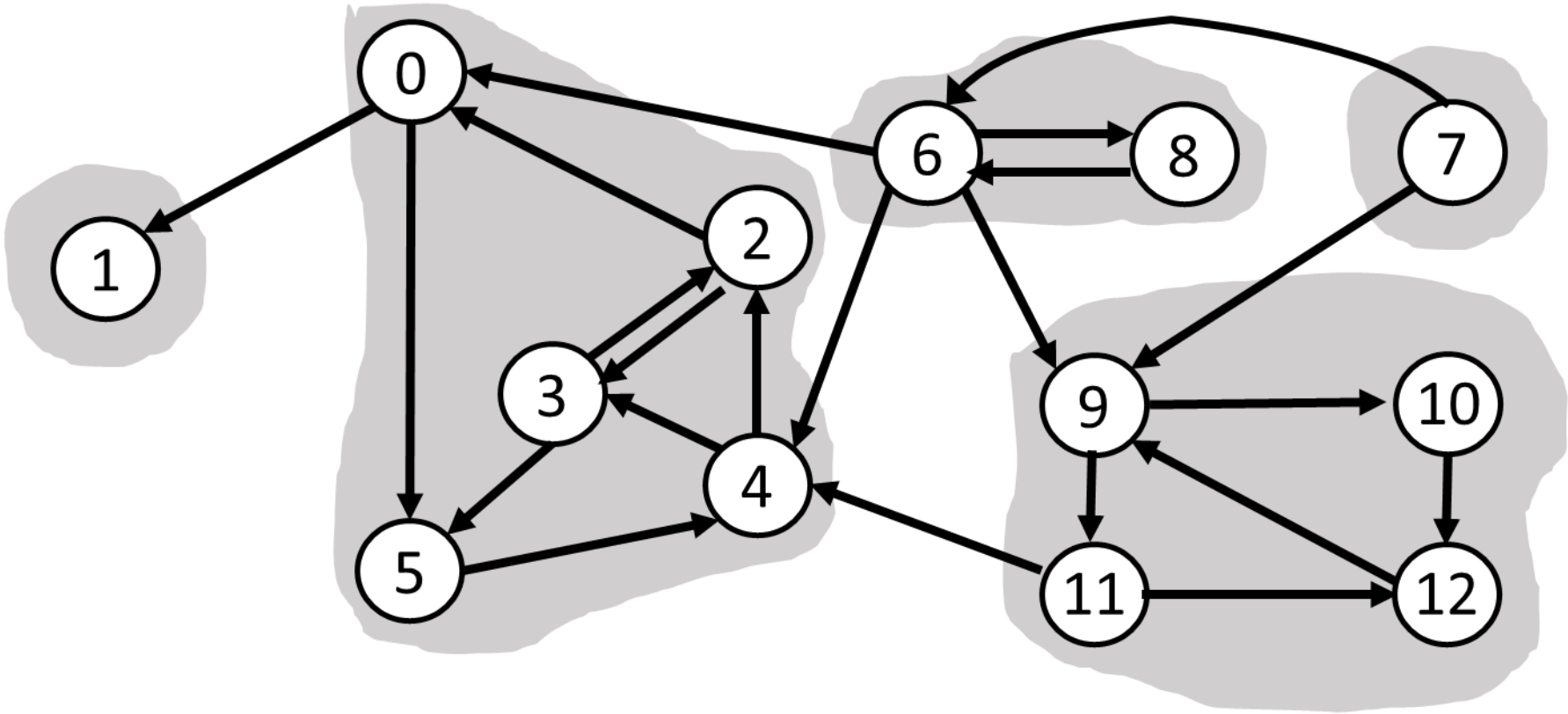


Strong components

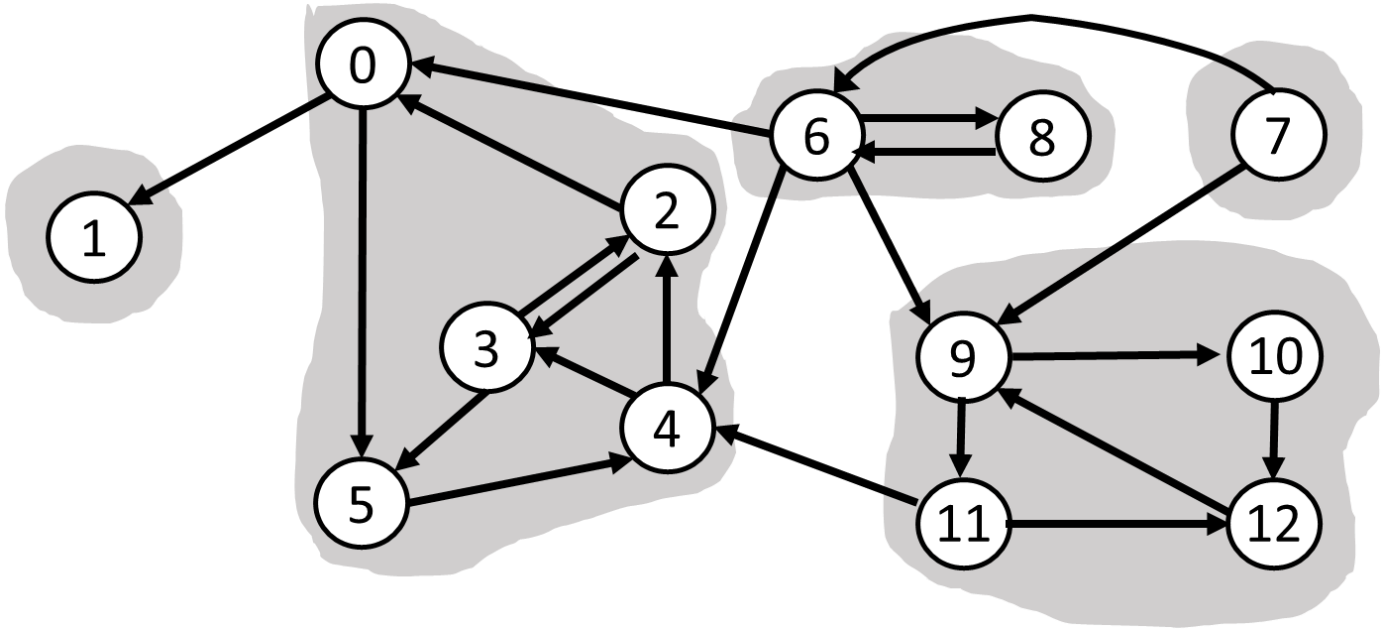
26 24 23 22 20 19 15 12 11 10 9 8 3



Strong components



Strong components



Theorem: [Tarjan 1972] Can find all strong components in $O(m + n)$ time.

SIAM J. COMPUT.
Vol. 1, No. 2, June 1972

DEPTH-FIRST SEARCH AND LINEAR GRAPH ALGORITHMS*

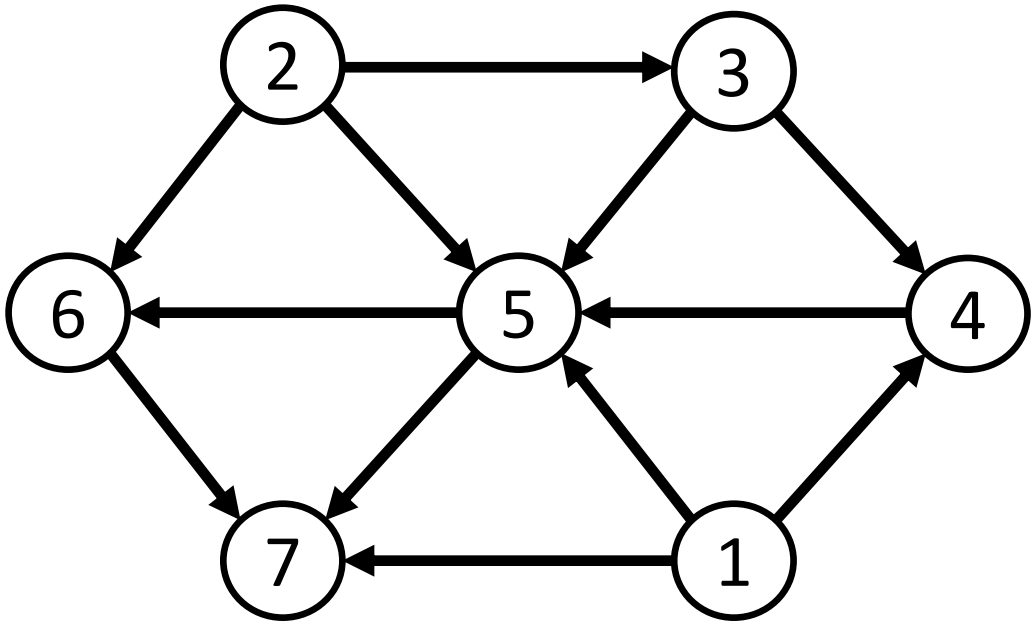
ROBERT TARJAN†

Abstract. The value of depth-first search or "backtracking" as a technique for solving problems is illustrated by two examples. An improved version of an algorithm for finding the strongly connected components of a directed graph and an algorithm for finding the biconnected components of an undirect graph are presented. The space and time requirements of both algorithms are bounded by $k_1V + k_2E + k_3$ for some constants k_1, k_2, and k_3, where V is the number of vertices and E is the number of edges of the graph being examined.

Section 3.6:
DAGs and Topological Ordering

Directed acyclic graphs

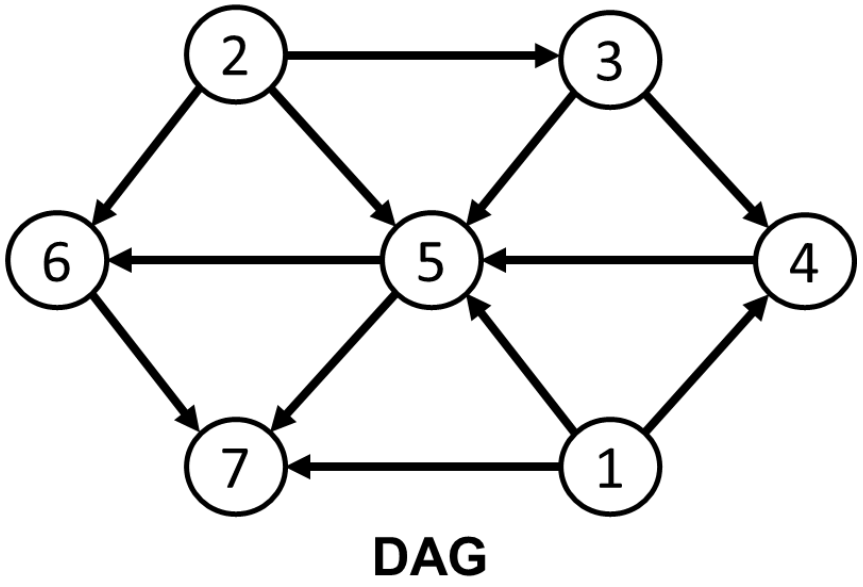
- **Def:** A DAG is a directed graph that contains no directed cycles.



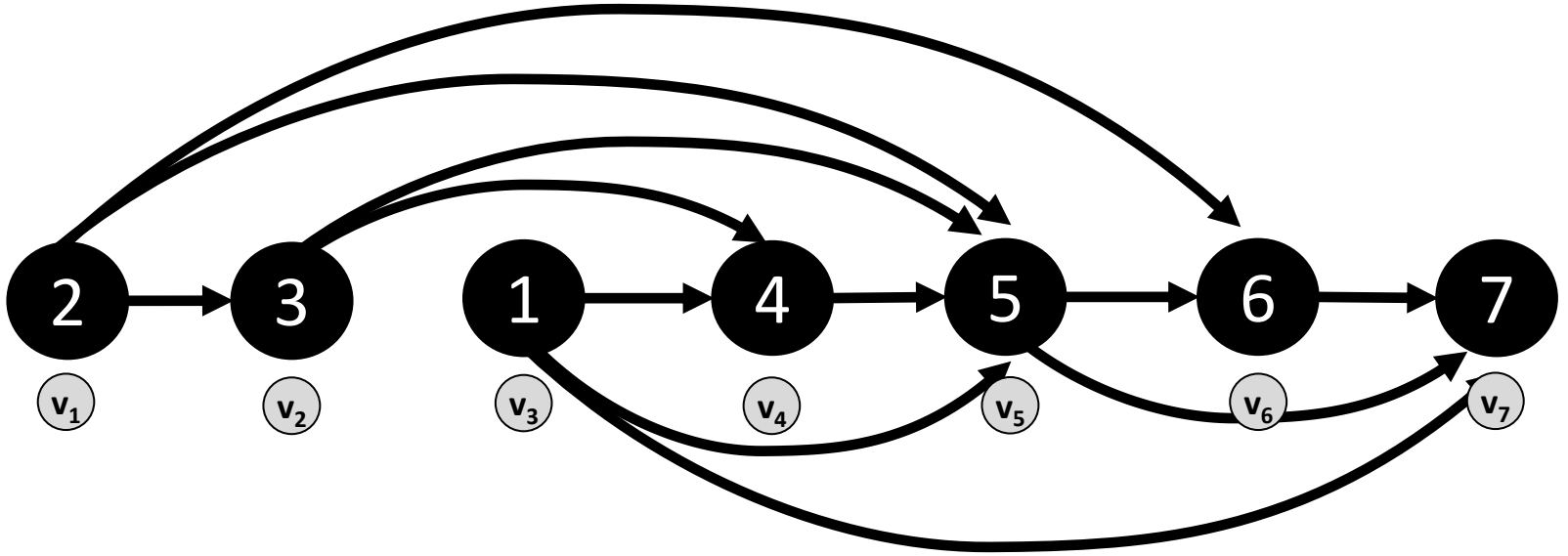
DAG

Directed acyclic graphs

- **Def:** A DAG is a directed graph that contains no directed cycles.
- **Def:** A topological order of a directed graph $G = (V, E)$ is an ordering of its nodes as $v_1, v_2, \ldots, v_n$ so that for every edge (v_i, v_j) we have $i < j$.



DAG



Topological Ordering

Precedence constraints

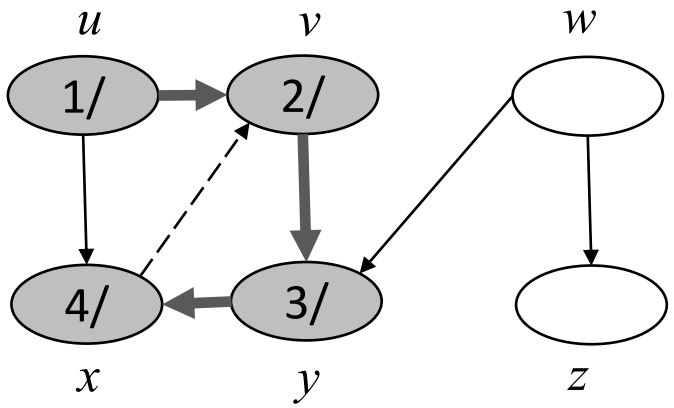
- **Precedence constraints:** Edge (v_i, v_j) means task v_i must occur before v_j.

- Applications.
  - **Course prerequisite graph**: course v_i must be taken before v_j.
  - **Pipeline of computing jobs**: output of job v_i needed to determine input of job v_j.

Topological sorting algorithm: DFS, *the boy savior*

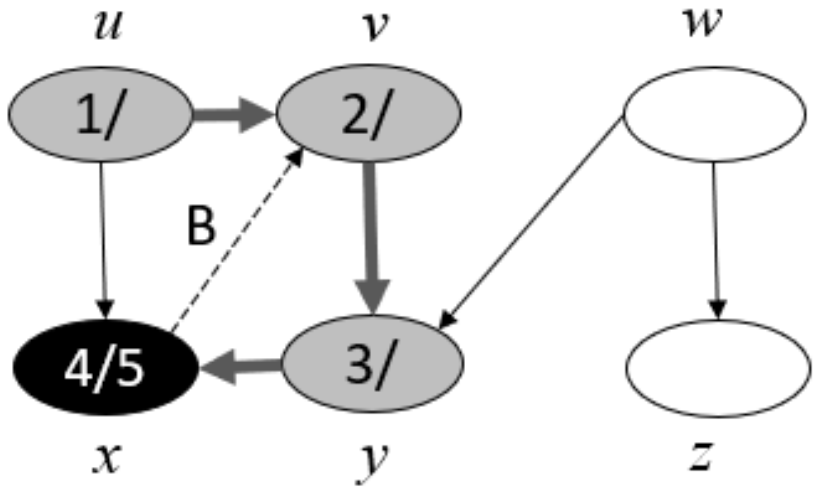
Remember, last time:

Back Edge



Back edges are those edges (u, v) connecting a vertex u to an ancestor vertex v in a depth-first tree. Self-loop (edge (u, u)), is also considered as Back edge

Three colors



Topological sorting algorithm: DFS, *the boy savior*

DFS(G)

1 **for** each vertex $u \in V$
2 $u.color = WHITE$
3 ~~$u.\pi = NIL$~~
4 ~~$time = 0$~~
5 **for** each vertex $u \in V$
6 **if** $u.color == WHITE$
7 DFS-VISIT(G, u)

DFS-VISIT(G, u)

1 ~~$time = time + 1$~~
2 ~~$u.d = time$~~
3 $u.color = GRAY$
4 **for** each $v \in Adj[u]$
5 **if** $v.color == WHITE$
6 ~~$v.\pi = u$~~
7 DFS-VISIT(G, v)
8 ~~$u.color = BLACK$~~
9 ~~$time = time + 1$~~
10 ~~$u.f = time$~~

New ***if*** block to check if it a **BACK Edge**

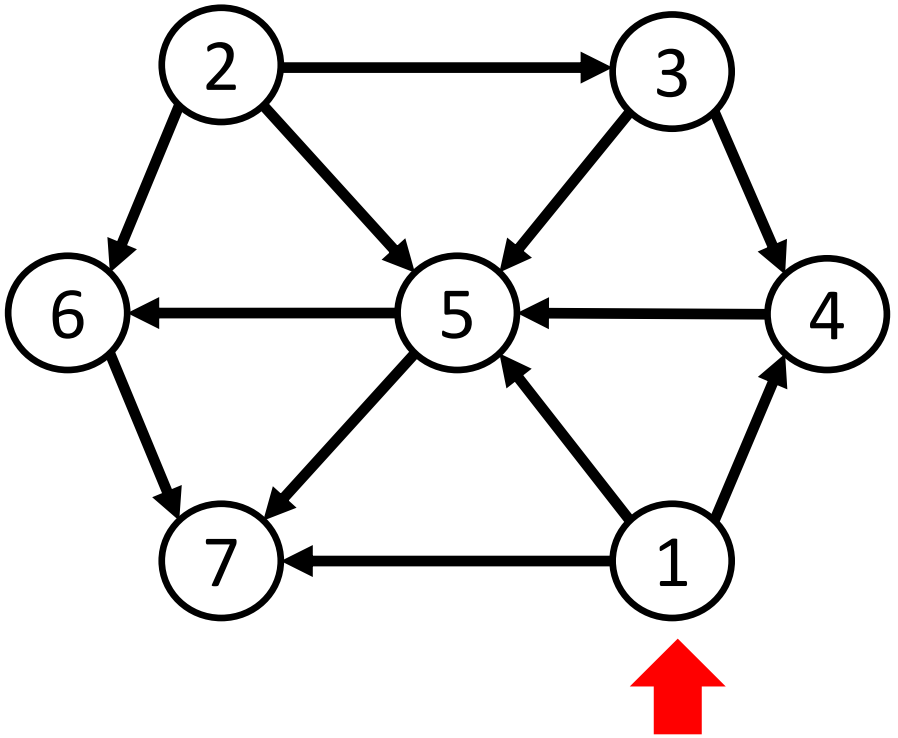
Topological sorting algorithm: DFS, *the boy savior*

DFS(G)

1 **for** each vertex $u \in V$
2 $u.color = WHITE$
3
4 L=[ ]
5 **for** each vertex $u \in V$
6 **if** $u.color == WHITE$
7 DFS-VISIT(G, u)

DFS-VISIT(G, u)

1
2
3 $u.color = GRAY$
4 **for** each $v \in Adj[u]$
5 **if** $v.color == WHITE$
6 DFS-VISIT(G, v)
7 **if** $v.color == GRAY$
8 Break
9 L-> add it to the front (u)
10



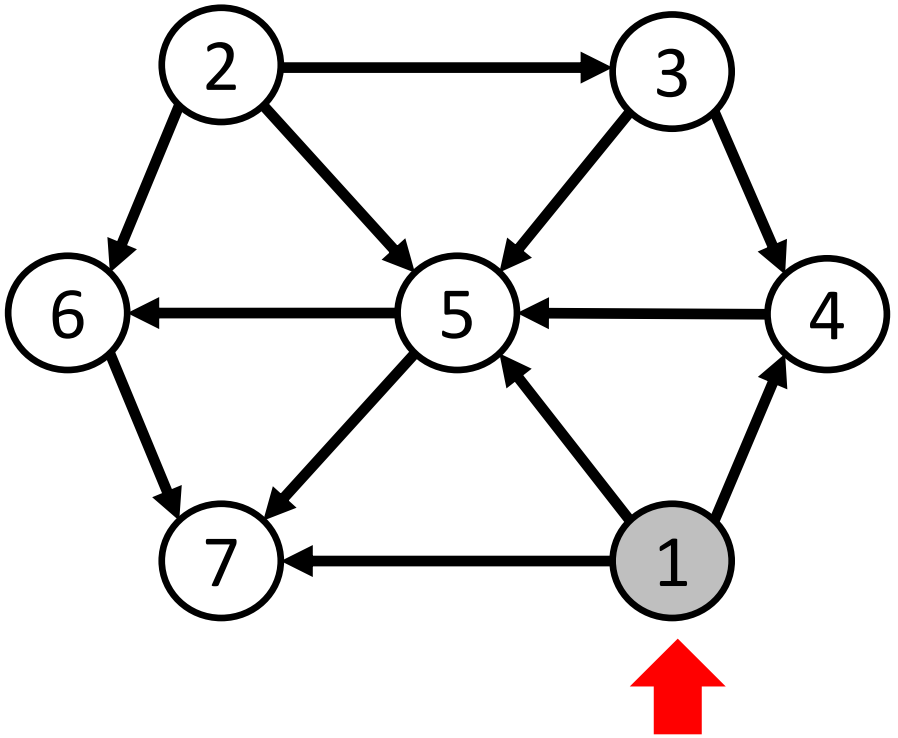
Topological sorting algorithm: DFS, *the boy savior*

DFS(G)

1 **for** each vertex $u \in V$
2 $u.color = WHITE$
3
4 L=[ ]
5 **for** each vertex $u \in V$
6 **if** $u.color == WHITE$
7 DFS-VISIT(G, u)

DFS-VISIT(G, u)

1
2
3 $u.color = GRAY$
4 **for** each $v \in Adj[u]$
5 **if** $v.color == WHITE$
6 DFS-VISIT(G, v)
7 **if** $v.color == GRAY$
8 Break
9 L-> add it to the front (u)
10



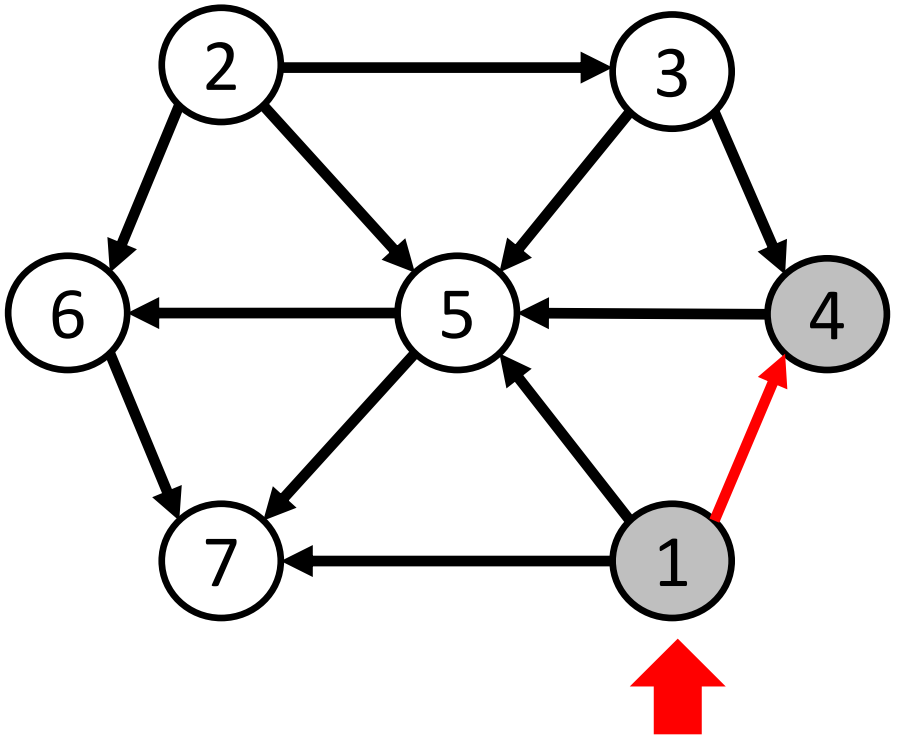
Topological sorting algorithm: DFS, *the boy savior*

DFS(G)

1 **for** each vertex $u \in V$
2 $u.color = WHITE$
3
4 L=[ ]
5 **for** each vertex $u \in V$
6 **if** $u.color == WHITE$
7 DFS-VISIT(G, u)

DFS-VISIT(G, u)

1
2
3 $u.color = GRAY$
4 **for** each $v \in Adj[u]$
5 **if** $v.color == WHITE$
6 DFS-VISIT(G, v)
7 **if** $v.color == GRAY$
8 Break
9 L-> add it to the front (u)
10



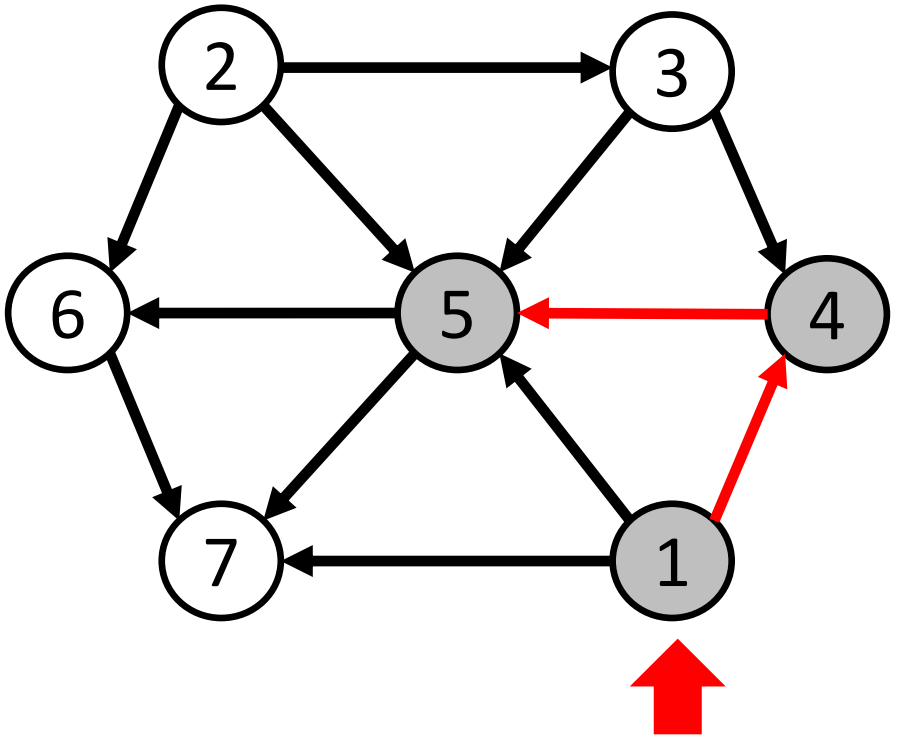
Topological sorting algorithm: DFS, *the boy savior*

DFS(G)

1 **for** each vertex $u \in V$
2 $u.color = WHITE$
3
4 L=[ ]
5 **for** each vertex $u \in V$
6 **if** $u.color == WHITE$
7 DFS-VISIT(G, u)

DFS-VISIT(G, u)

1
2
3 $u.color = GRAY$
4 **for** each $v \in Adj[u]$
5 **if** $v.color == WHITE$
6 DFS-VISIT(G, v)
7 **if** $v.color == GRAY$
8 Break
9 L-> add it to the front (u)
10



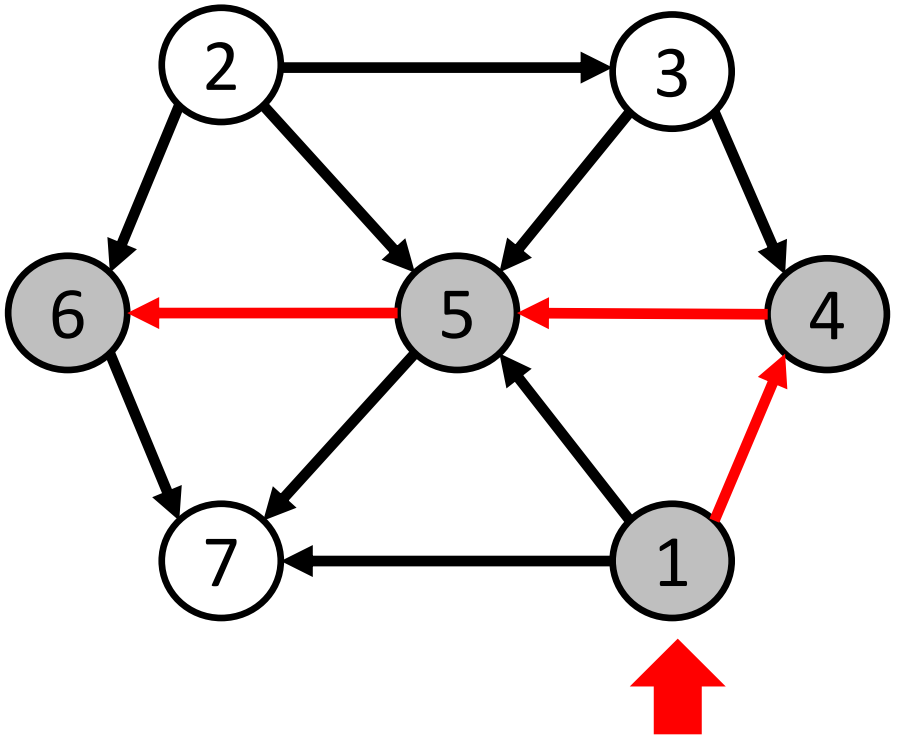
Topological sorting algorithm: DFS, *the boy savior*

DFS(G)

1 **for** each vertex $u \in V$
2 $u.color = WHITE$
3
4 L=[ ]
5 **for** each vertex $u \in V$
6 **if** $u.color == WHITE$
7 DFS-VISIT(G, u)

DFS-VISIT(G, u)

1
2
3 $u.color = GRAY$
4 **for** each $v \in Adj[u]$
5 **if** $v.color == WHITE$
6 DFS-VISIT(G, v)
7 **if** $v.color == GRAY$
8 Break
9 L-> add it to the front (u)
10



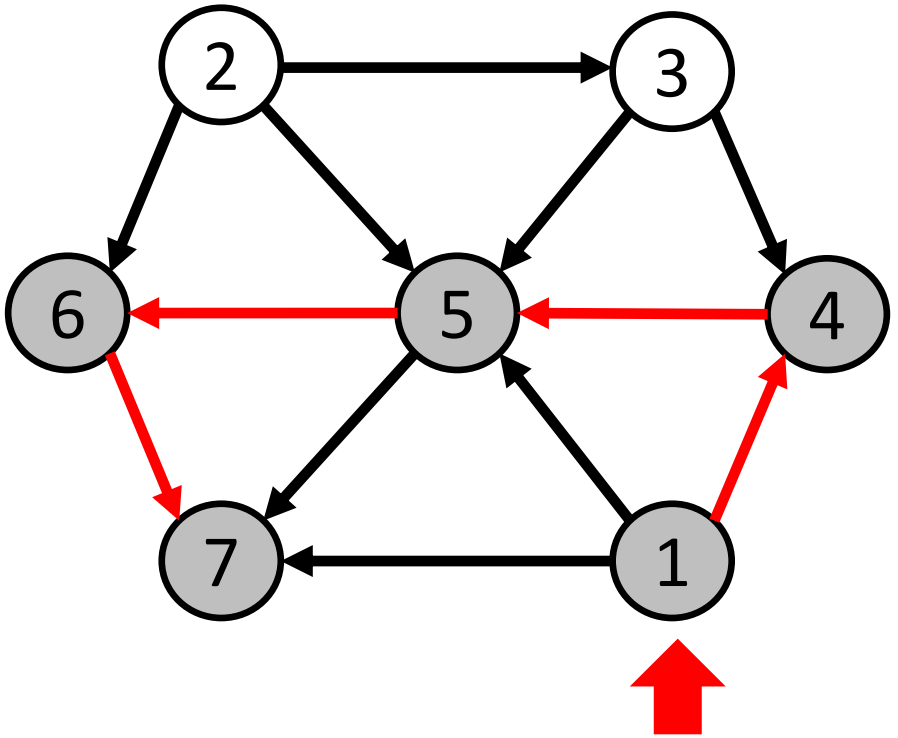
Topological sorting algorithm: DFS, *the boy savior*

DFS(G)

1 **for** each vertex $u \in V$
2 $u.color = WHITE$
3
4 L=[ ]
5 **for** each vertex $u \in V$
6 **if** $u.color == WHITE$
7 DFS-VISIT(G, u)

DFS-VISIT(G, u)

1
2
3 $u.color = GRAY$
4 **for** each $v \in Adj[u]$
5 **if** $v.color == WHITE$
6 DFS-VISIT(G, v)
7 **if** $v.color == GRAY$
8 Break
9 L-> add it to the front (u)
10



Topological sorting algorithm: DFS, *the boy savior*

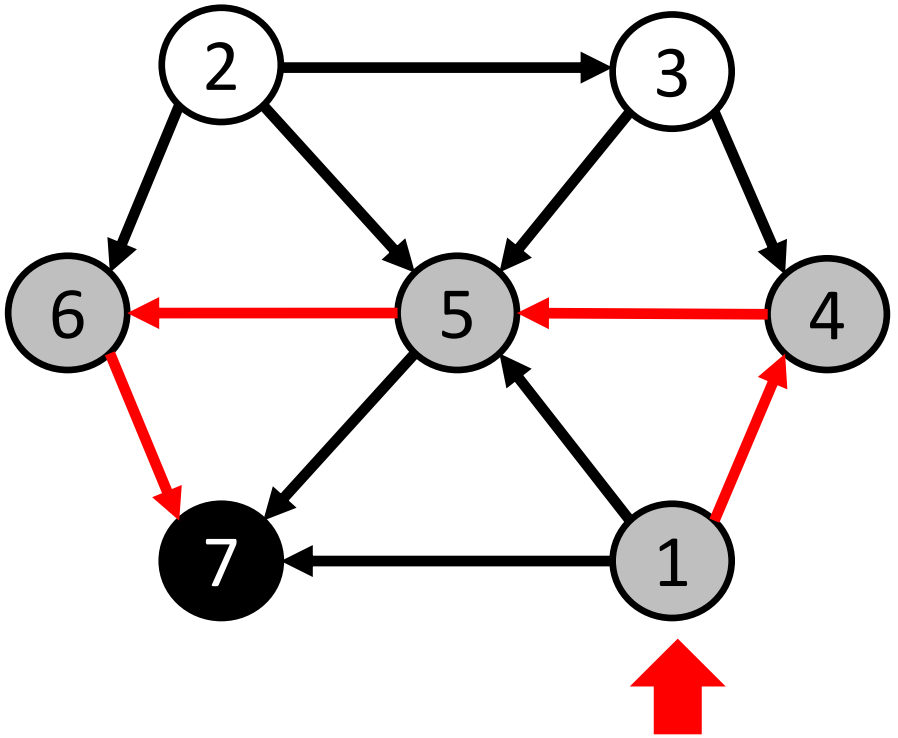
L 7

DFS(G)

```
1   for each vertex u ∈ V
2       u.color = WHITE
3
4   L=[ ]
5   for each vertex u ∈ V
6       if u.color == WHITE
7           DFS-VISIT(G, u)
```

DFS-VISIT(G, u)

```
1
2
3   u.color = GRAY
4   for each v ∈ Adj[u]
5       if v.color == WHITE
6           DFS-VISIT(G, v)
7       if v.color == GRAY
8           Break
9   L-> add it to the front (u)
10
```



Topological sorting algorithm: DFS, *the boy savior*

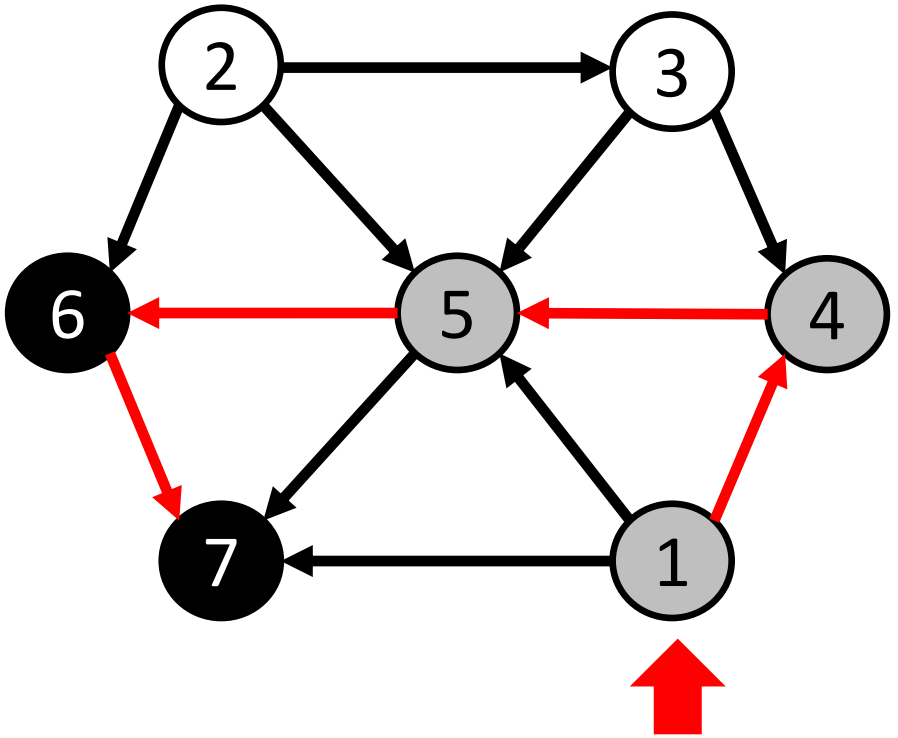
L 6 7

DFS(G)

```
1   for each vertex u ∈ V
2       u.color = WHITE
3
4   L=[ ]
5   for each vertex u ∈ V
6       if u.color == WHITE
7           DFS-VISIT(G, u)
```

DFS-VISIT(G, u)

```
1
2
3   u.color = GRAY
4   for each v ∈ Adj[u]
5       if v.color == WHITE
6           DFS-VISIT(G, v)
7       if v.color == GRAY
8           Break
9   L-> add it to the front (u)
10
```



Topological sorting algorithm: DFS, *the boy savior*

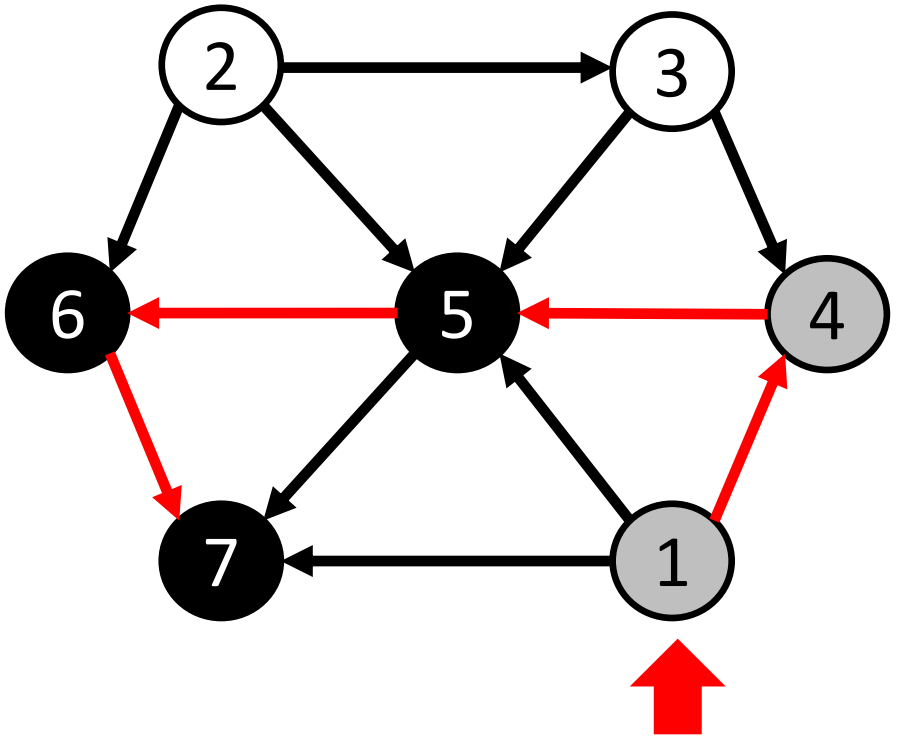
L 5 6 7

DFS(G)

```
1   for each vertex u ∈ V
2       u.color = WHITE
3
4   L=[ ]
5   for each vertex u ∈ V
6       if u.color == WHITE
7           DFS-VISIT(G, u)
```

DFS-VISIT(G, u)

```
1
2
3   u.color = GRAY
4   for each v ∈ Adj[u]
5       if v.color == WHITE
6           DFS-VISIT(G, v)
7       if v.color == GRAY
8           Break
9   L-> add it to the front (u)
10
```



Topological sorting algorithm: DFS, *the boy savior*

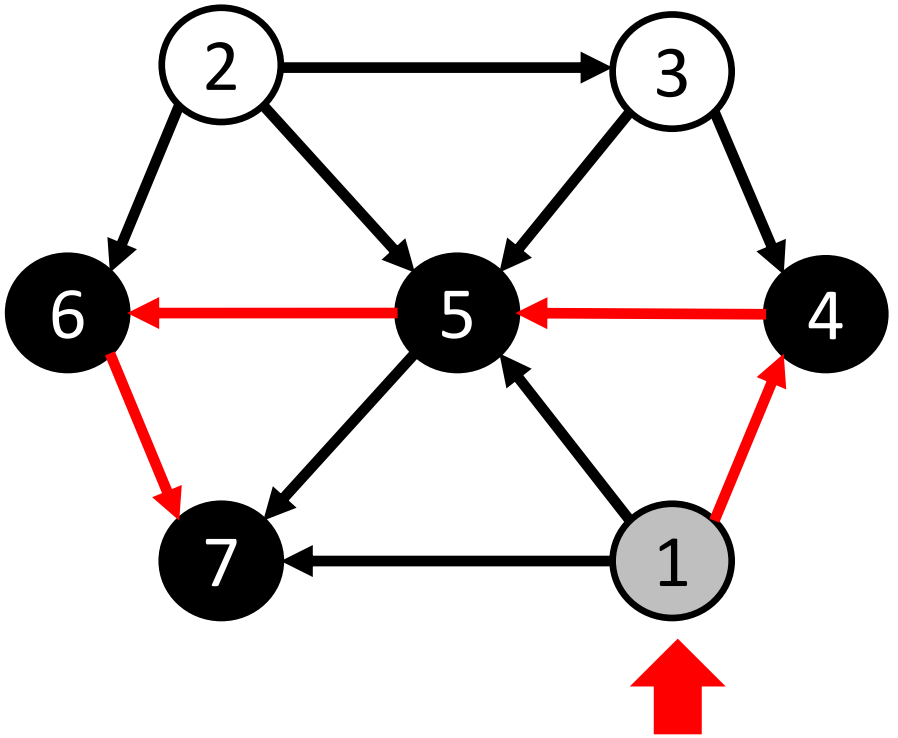
L 4 5 6 7

DFS(G)

```
1   for each vertex u ∈ V
2       u.color = WHITE
3
4   L=[ ]
5   for each vertex u ∈ V
6       if u.color == WHITE
7           DFS-VISIT(G, u)
```

DFS-VISIT(G, u)

```
1
2
3   u.color = GRAY
4   for each v ∈ Adj[u]
5       if v.color == WHITE
6           DFS-VISIT(G, v)
7       if v.color == GRAY
8           Break
9   L-> add it to the front (u)
10
```



Topological sorting algorithm: DFS, *the boy savior*

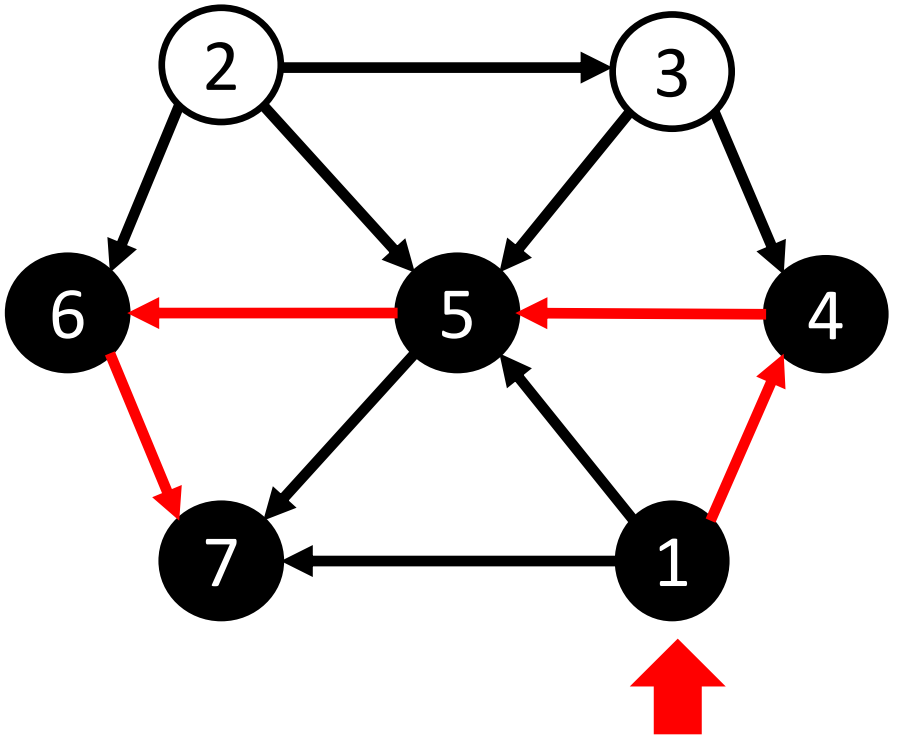
L 1 4 5 6 7

DFS(G)

```
1   for each vertex u ∈ V
2       u.color = WHITE
3
4   L=[ ]
5   for each vertex u ∈ V
6       if u.color == WHITE
7           DFS-VISIT(G, u)
```

DFS-VISIT(G, u)

```
1
2
3   u.color = GRAY
4   for each v ∈ Adj[u]
5       if v.color == WHITE
6           DFS-VISIT(G, v)
7       if v.color == GRAY
8           Break
9   L-> add it to the front (u)
10
```



Topological sorting algorithm: DFS, *the boy savior*

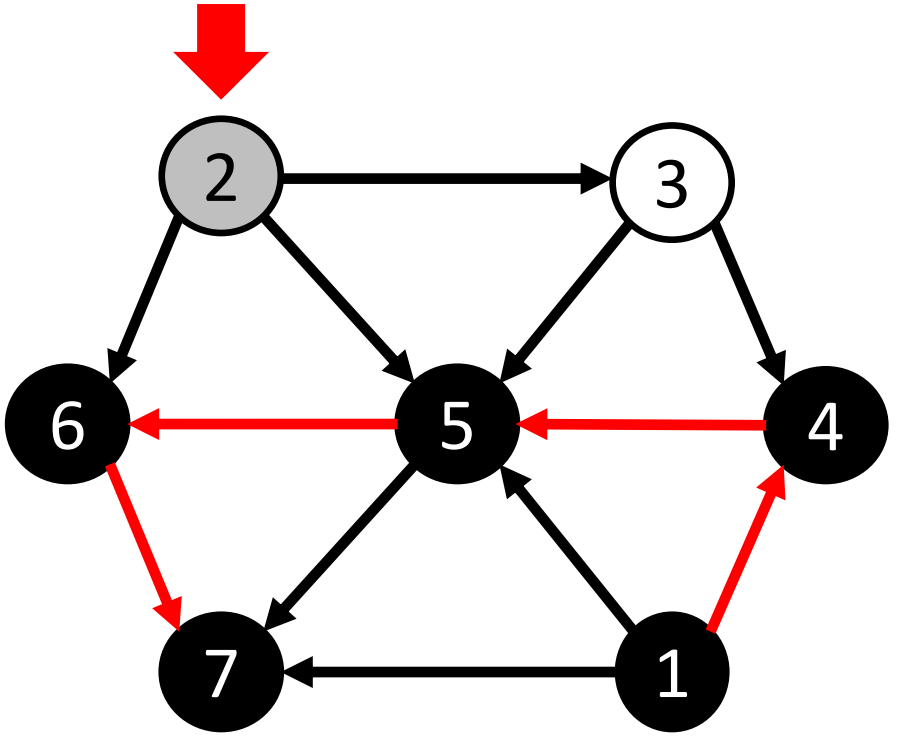
L 1 4 5 6 7

DFS(G)

```
1   for each vertex u ∈ V
2       u.color = WHITE
3
4   L=[ ]
5   for each vertex u ∈ V
6       if u.color == WHITE
7           DFS-VISIT(G, u)
```

DFS-VISIT(G, u)

```
1
2
3   u.color = GRAY
4   for each v ∈ Adj[u]
5       if v.color == WHITE
6           DFS-VISIT(G, v)
7       if v.color == GRAY
8           Break
9   L-> add it to the front (u)
10
```



Topological sorting algorithm: DFS, *the boy savior*

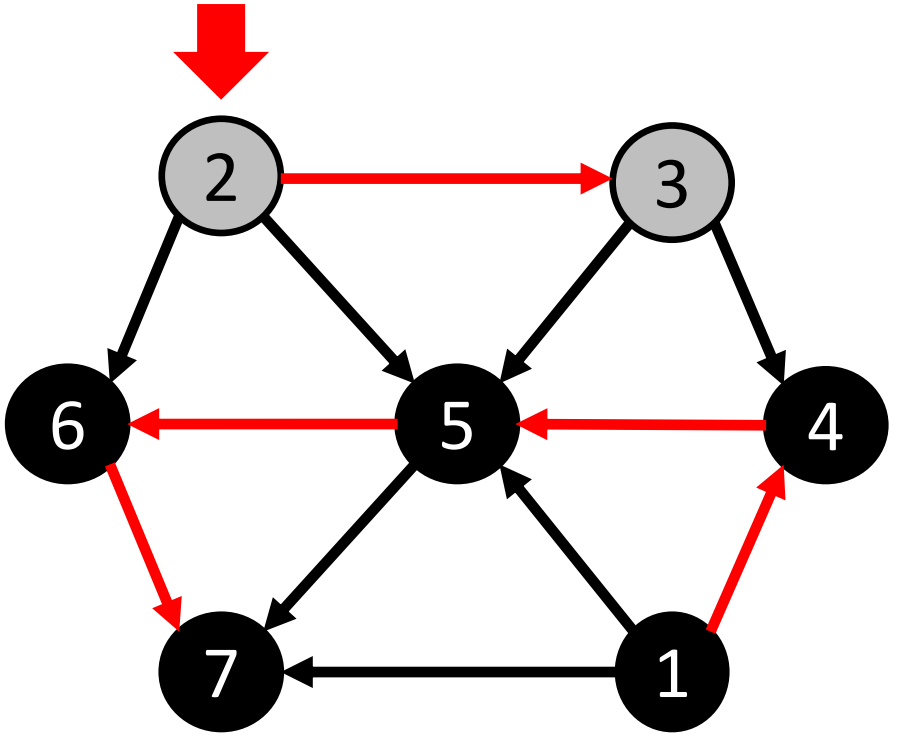
L 1 4 5 6 7

DFS(G)

```
1   for each vertex u ∈ V
2       u.color = WHITE
3
4   L=[ ]
5   for each vertex u ∈ V
6       if u.color == WHITE
7           DFS-VISIT(G, u)
```

DFS-VISIT(G, u)

```
1
2
3   u.color = GRAY
4   for each v ∈ Adj[u]
5       if v.color == WHITE
6           DFS-VISIT(G, v)
7       if v.color == GRAY
8           Break
9   L-> add it to the front (u)
10
```



Topological sorting algorithm: DFS, *the boy savior*

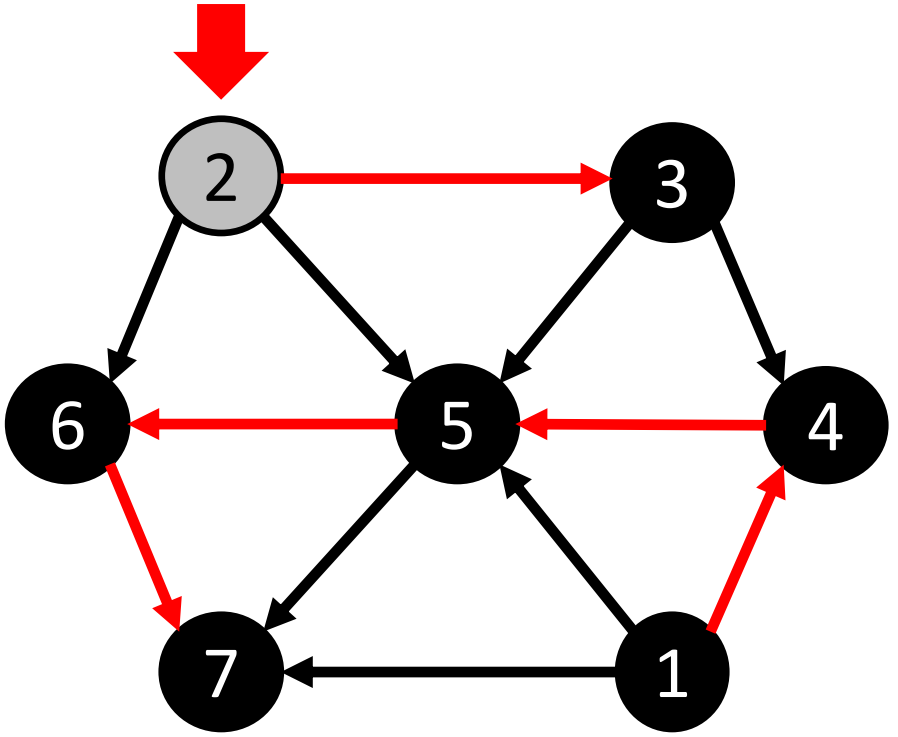
L 3 1 4 5 6 7

DFS(G)

```
1   for each vertex u ∈ V
2       u.color = WHITE
3
4   L=[ ]
5   for each vertex u ∈ V
6       if u.color == WHITE
7           DFS-VISIT(G, u)
```

DFS-VISIT(G, u)

```
1
2
3   u.color = GRAY
4   for each v ∈ Adj[u]
5       if v.color == WHITE
6           DFS-VISIT(G, v)
7       if v.color == GRAY
8           Break
9   L-> add it to the front (u)
10
```



Topological sorting algorithm: DFS, *the boy savior*

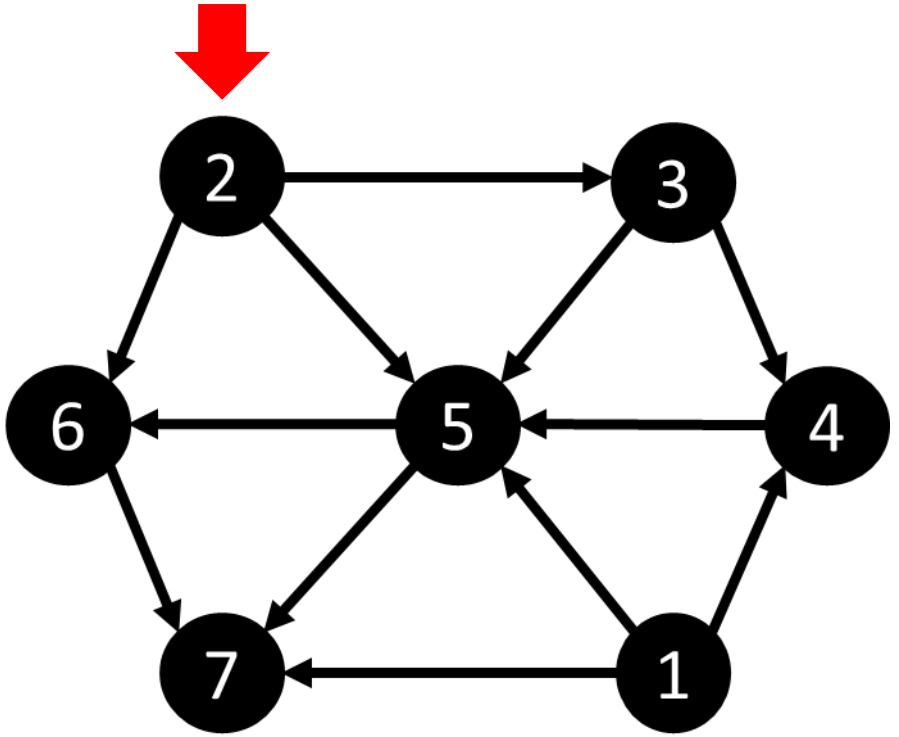
L 2 3 1 4 5 6 7

DFS(G)

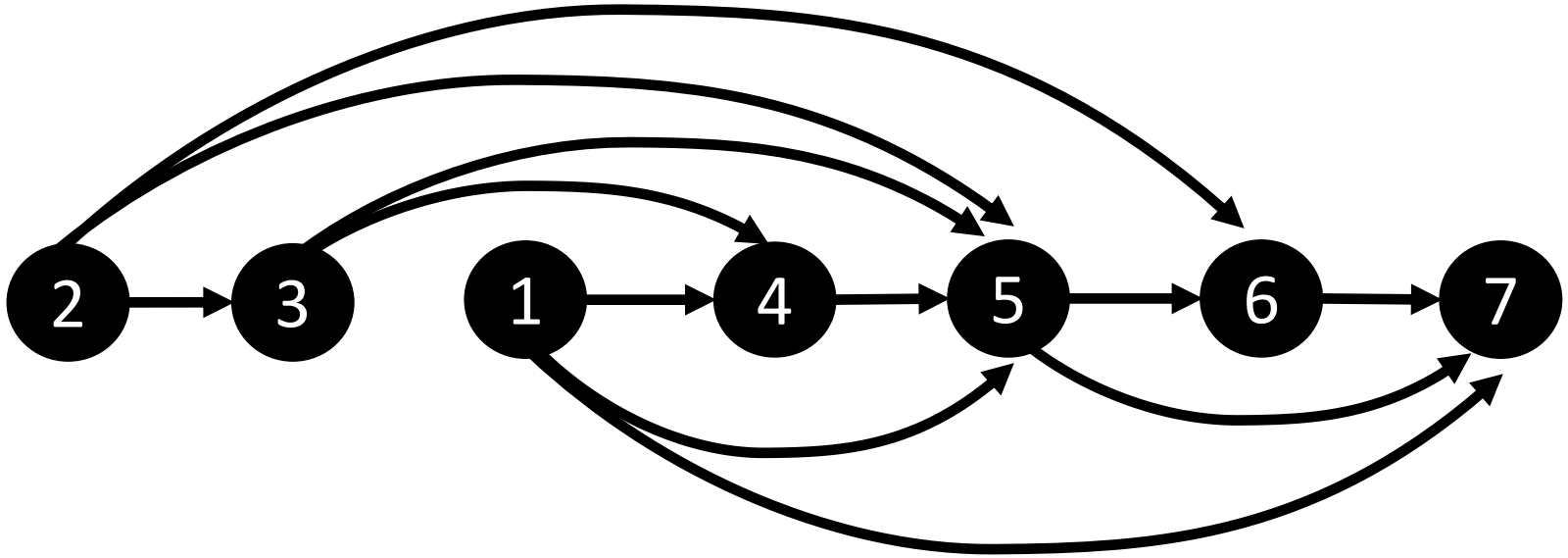
```
1   for each vertex u ∈ V
2       u.color = WHITE
3
4   L=[ ]
5   for each vertex u ∈ V
6       if u.color == WHITE
7           DFS-VISIT(G, u)
```

DFS-VISIT(G, u)

```
1
2
3   u.color = GRAY
4   for each v ∈ Adj[u]
5       if v.color == WHITE
6           DFS-VISIT(G, v)
7       if v.color == GRAY
8           Break
9   L-> add it to the front (u)
10
```

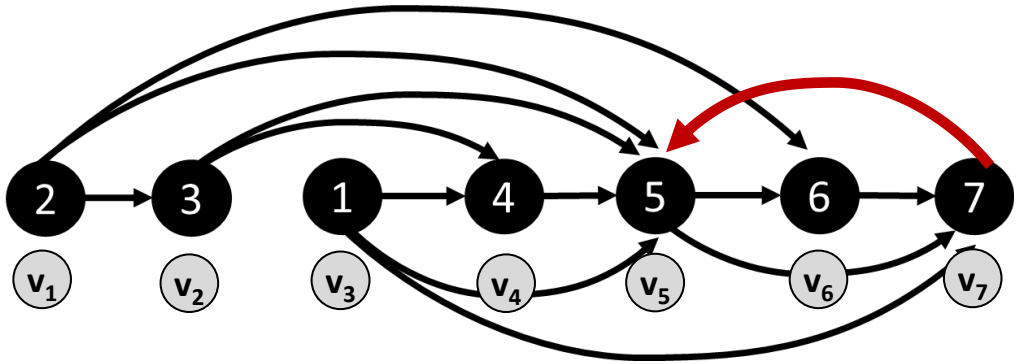


Topological sorting algorithm: DFS, *the boy savior*



Directed acyclic graphs

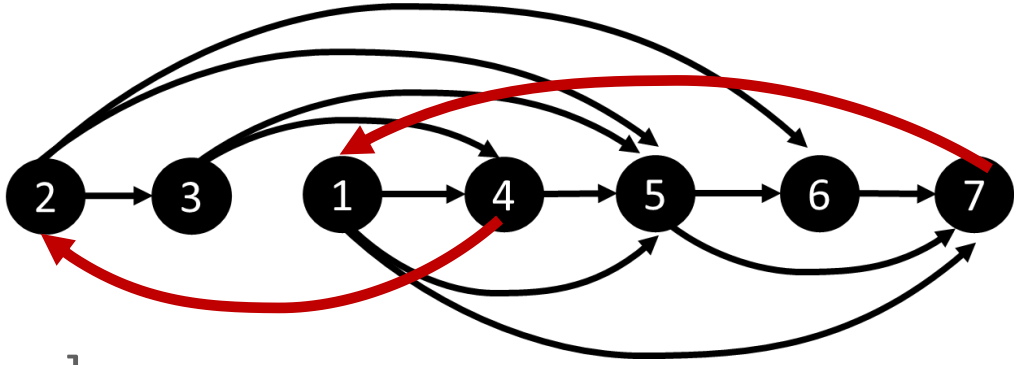
- **Lemma.** If G has a topological order, then G is a DAG.



- **Pf.** [by contradiction]
  - Suppose that G has a topological order, and that G also has a directed cycle C. Let's see what happens.
  - By definition, every edge (v_i, v_j) in topological order, i < j.
  - On the other hand, since (v_7, v_5) is an edge, we must have $j < i$, **a contradiction**.

Directed acyclic graphs

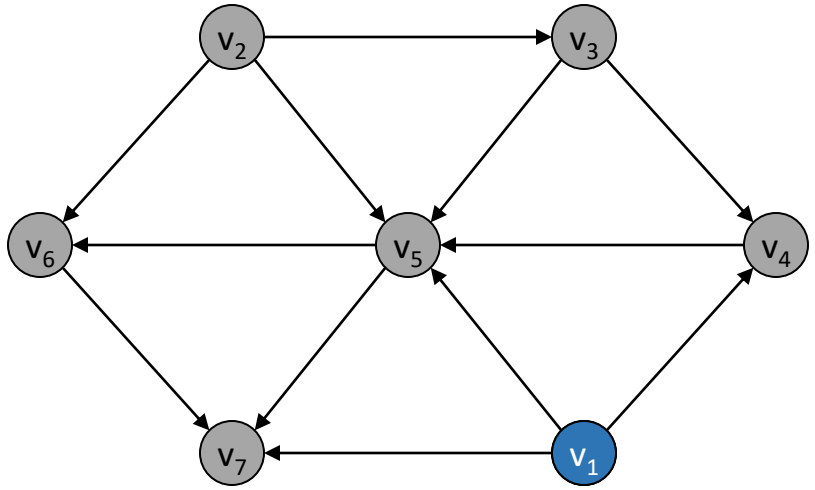
- Lemma. If G is a DAG, then G has a node with no entering edges.



- Pf. [by contradiction]
  - Suppose that G is a DAG and every node has at least one entering edge. Let's see what happens.
  - Graph **G** will have a **cycle**

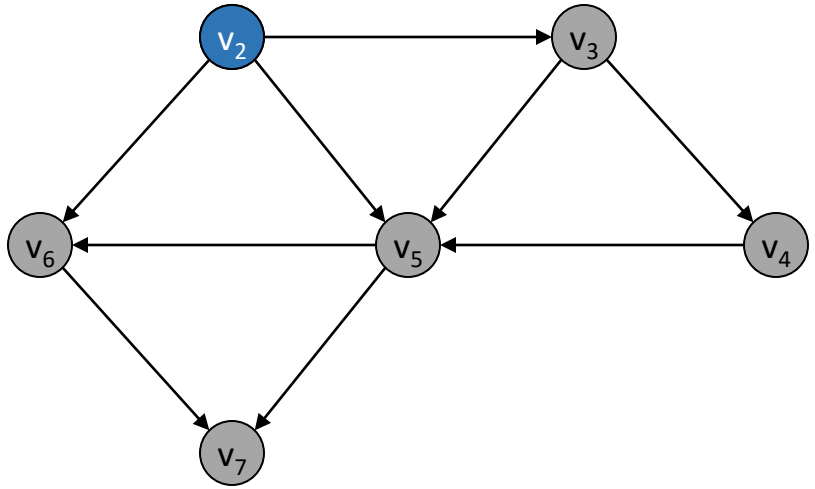
Another Topological Ordering Algorithm: Example

To compute a topological ordering of G:
Find a node v with no incoming edges and order it first
Delete v from G
Recursively compute a topological ordering of $G-\{v\}$
  and append this order after v



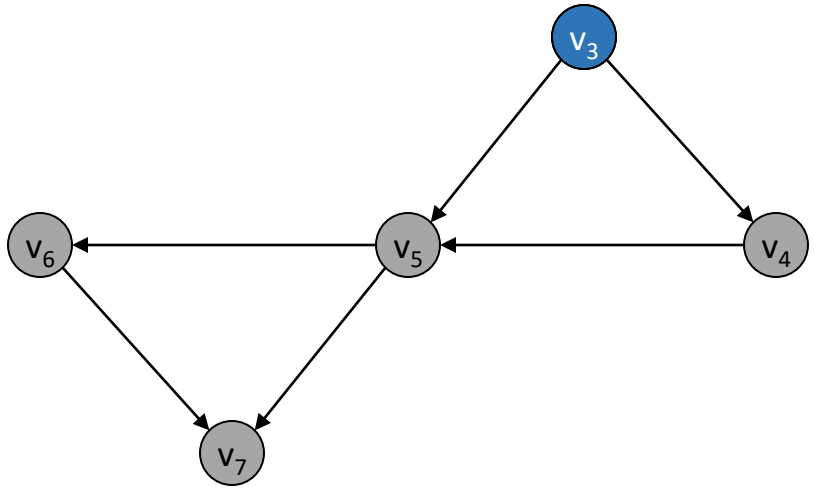
Topological order:

Topological Ordering Algorithm: Example



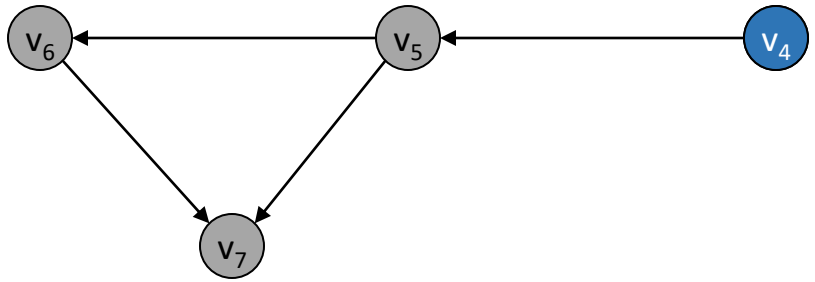
Topological order: v_1

Topological Ordering Algorithm: Example



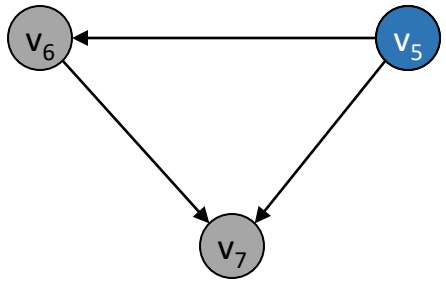
Topological order: v_1, v_2

Topological Ordering Algorithm: Example



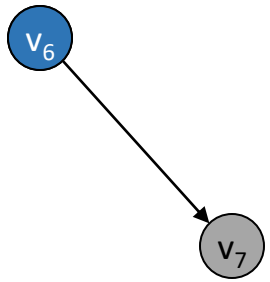
Topological order: v_1, v_2, v_3

Topological Ordering Algorithm: Example



Topological order: v_1, v_2, v_3, v_4

Topological Ordering Algorithm: Example

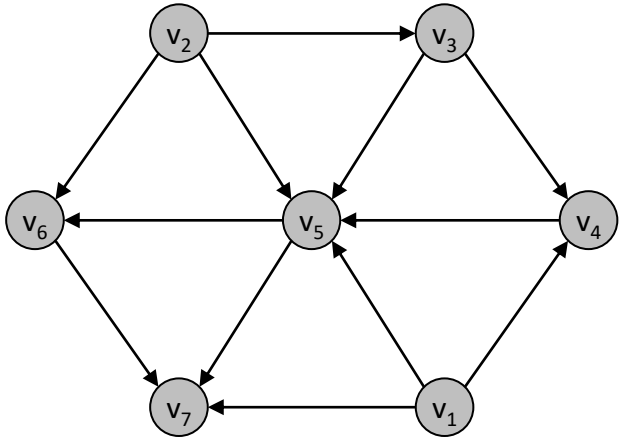


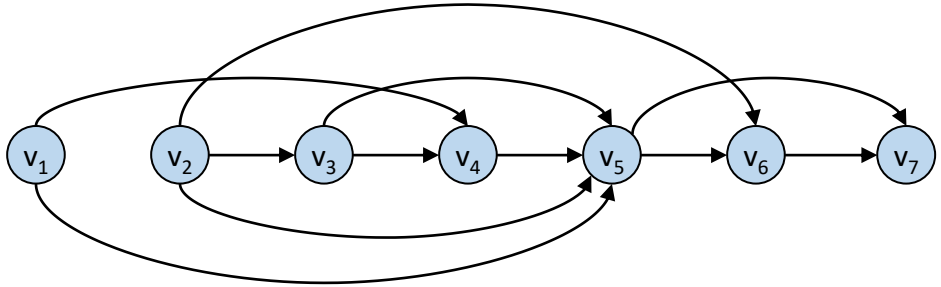
Topological order: v_1, v_2, v_3, v_4, v_5

Topological Ordering Algorithm: Example

Topological order: $v_1, v_2, v_3, v_4, v_5, v_6$

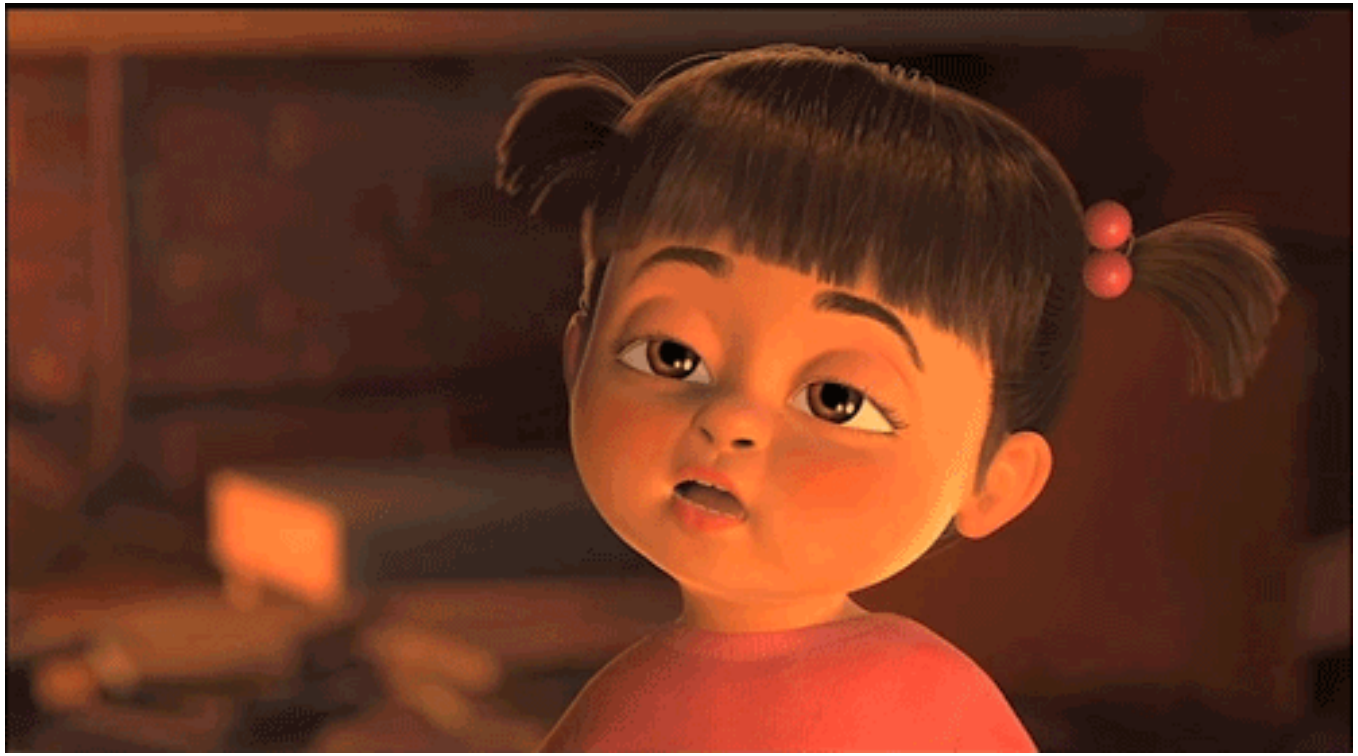
Topological Ordering Algorithm: Example





Topological order: $v_1, v_2, v_3, v_4, v_5, v_6, v_7$.

Thanks a lot



If you are taking a Nap, **wake up**.....Lecture Over