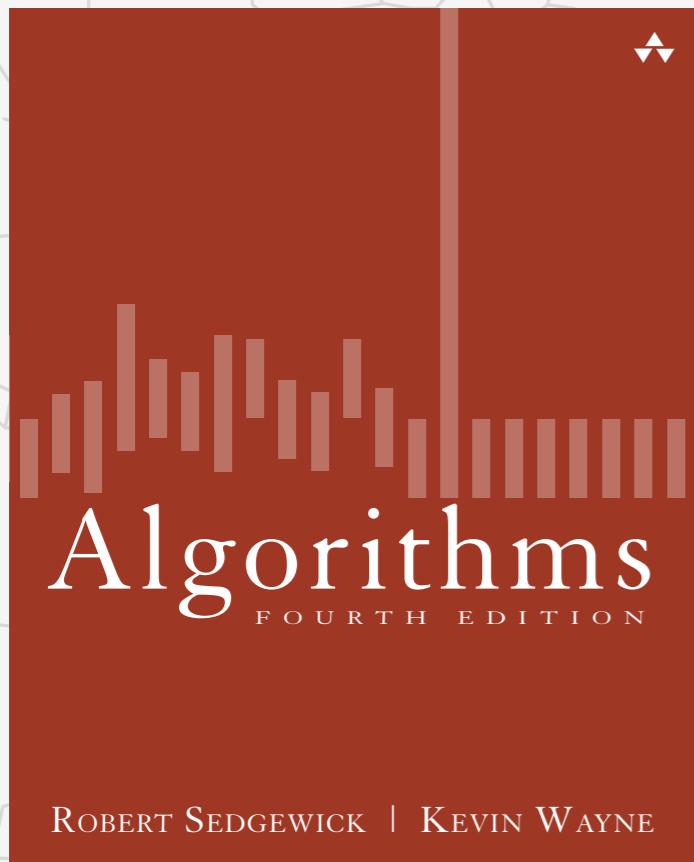


# Algorithms

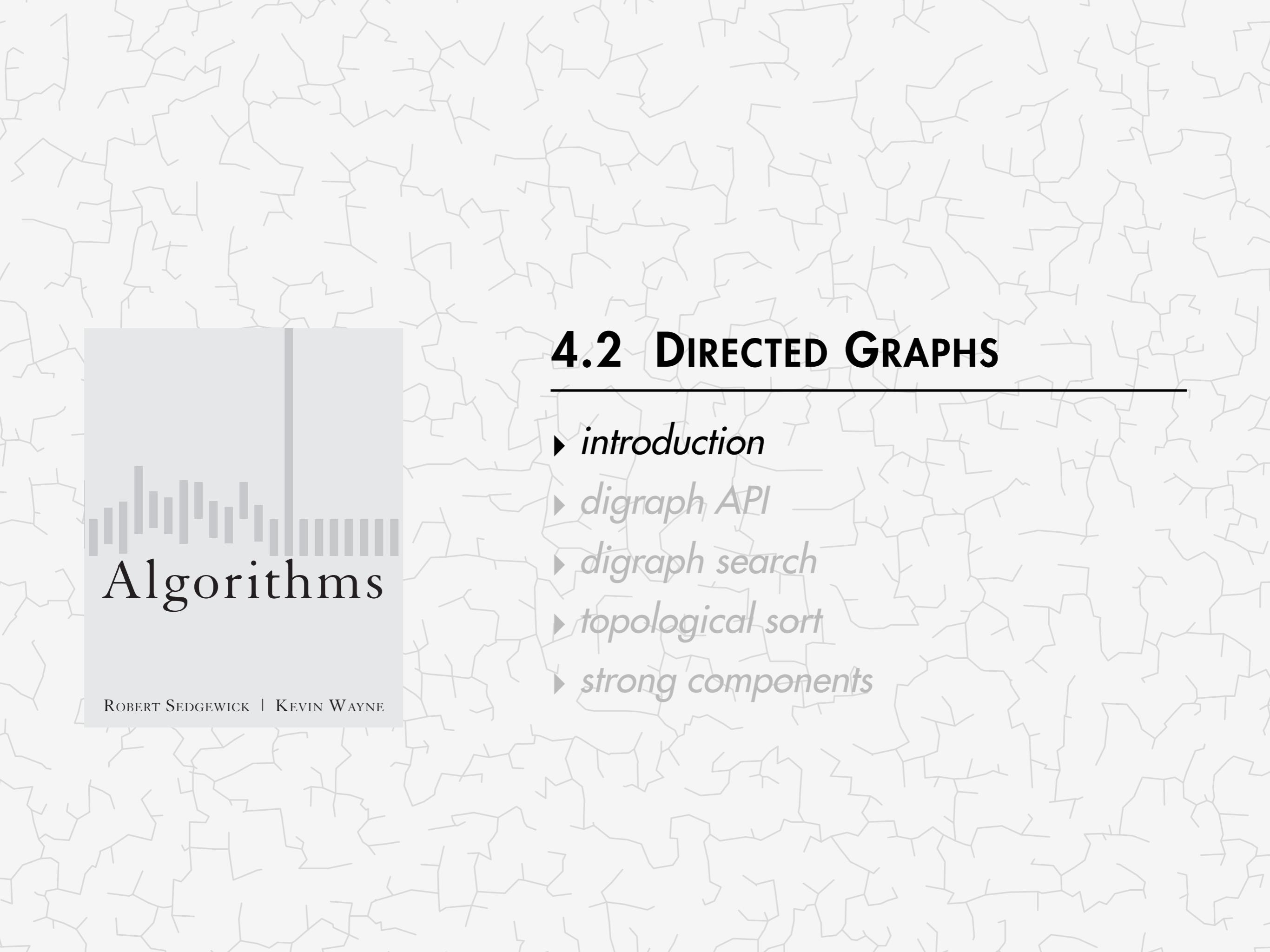
ROBERT SEDGEWICK | KEVIN WAYNE



## 4.2 DIRECTED GRAPHS

---

- ▶ *introduction*
- ▶ *digraph API*
- ▶ *digraph search*
- ▶ *topological sort*
- ▶ *strong components*



# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

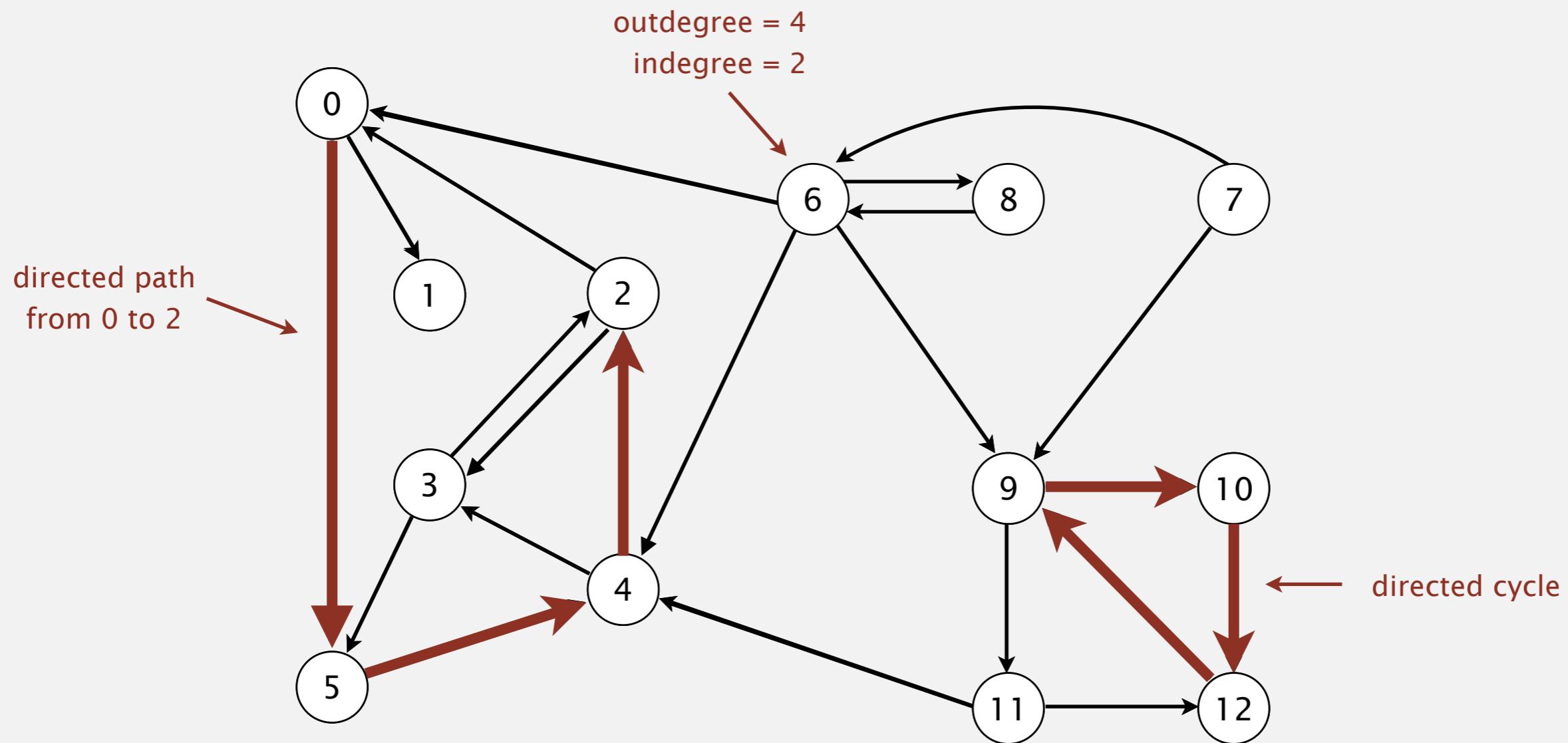
## 4.2 DIRECTED GRAPHS

---

- ▶ *introduction*
- ▶ *digraph API*
- ▶ *digraph search*
- ▶ *topological sort*
- ▶ *strong components*

# Directed graphs

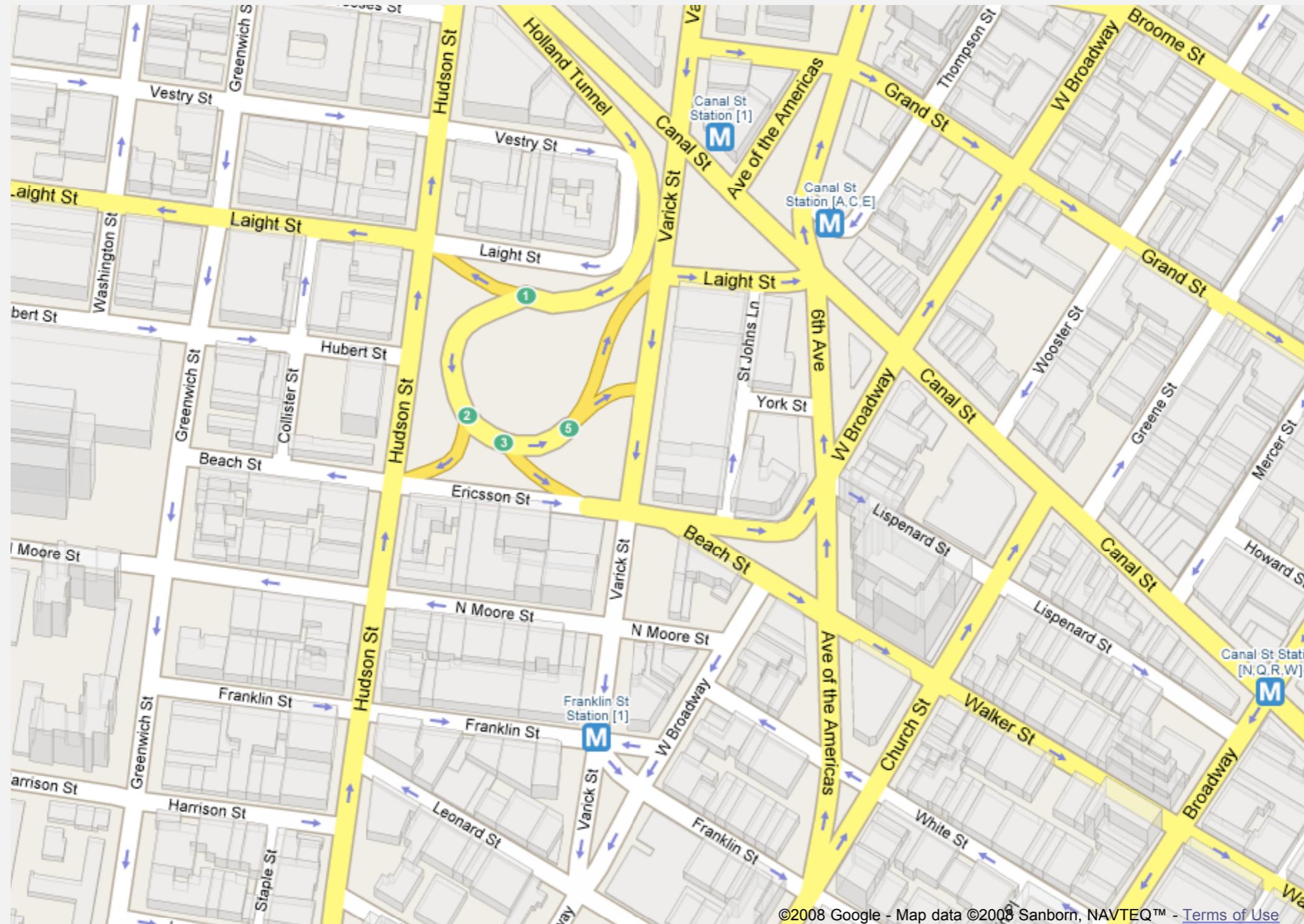
Digraph. Set of vertices connected pairwise by **directed** edges.



# Road network

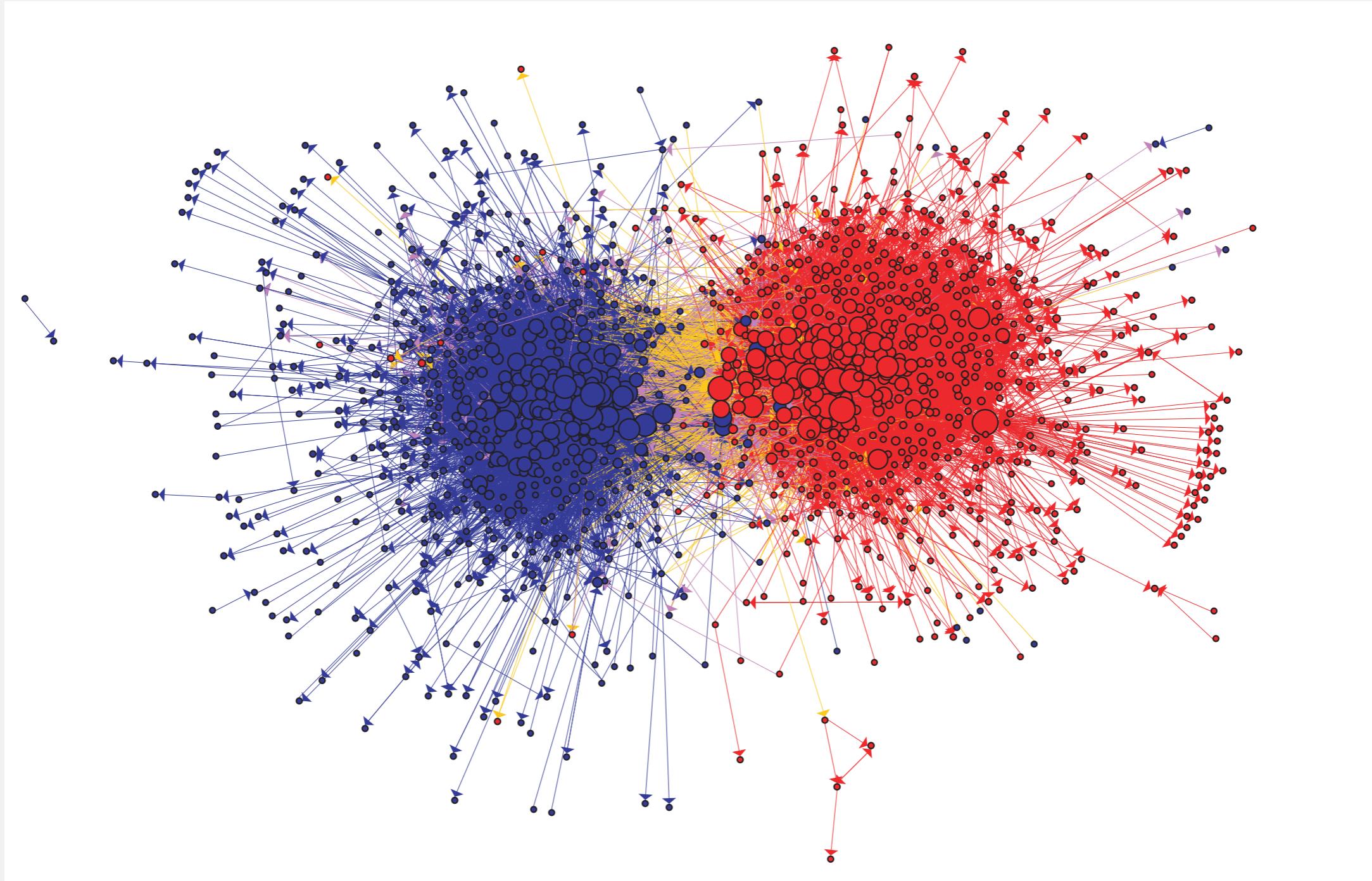
---

Vertex = intersection; edge = one-way street.



# Political blogosphere graph

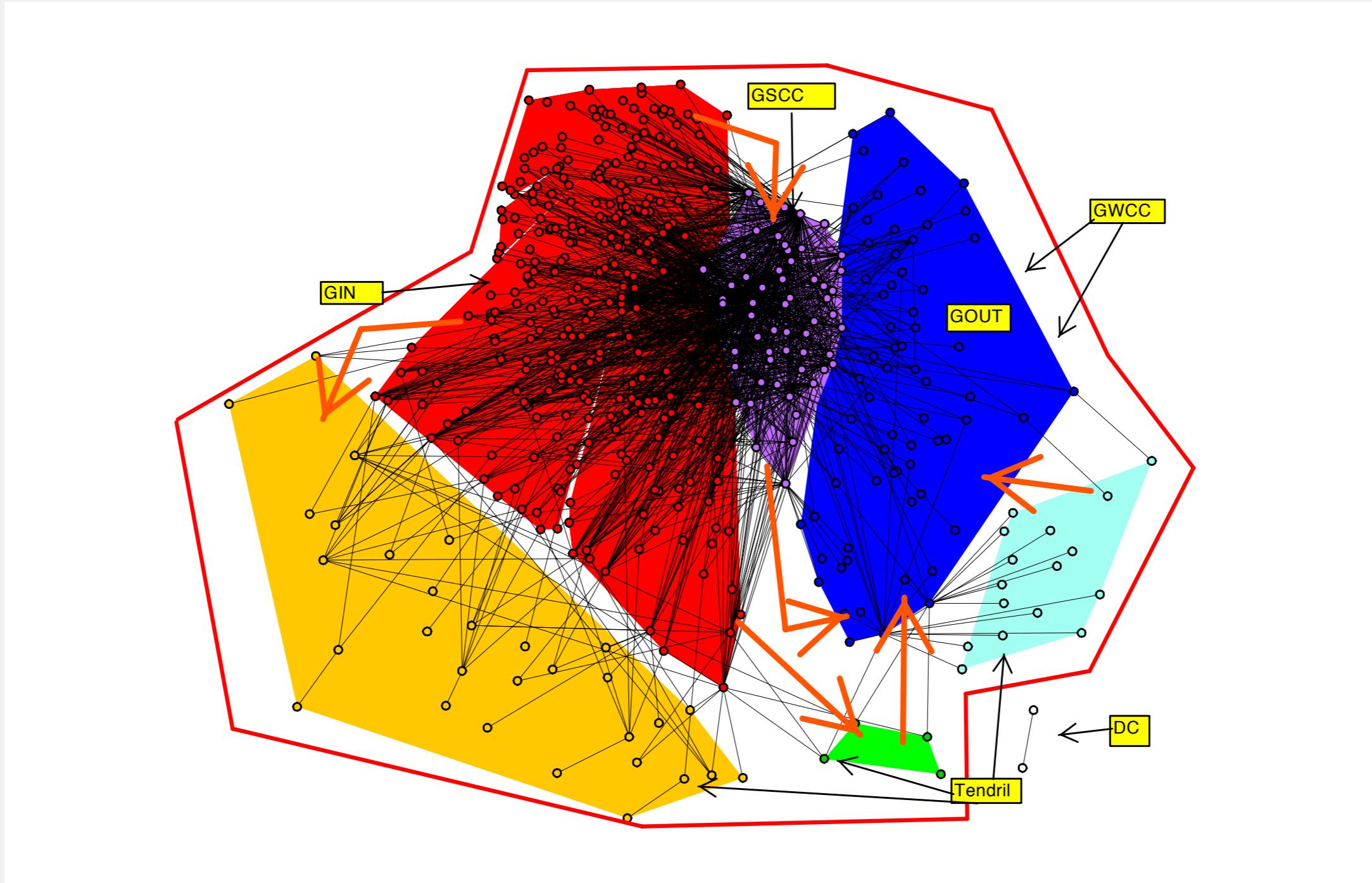
Vertex = political blog; edge = link.



The Political Blogosphere and the 2004 U.S. Election: Divided They Blog, Adamic and Glance, 2005

# Overnight interbank loan graph

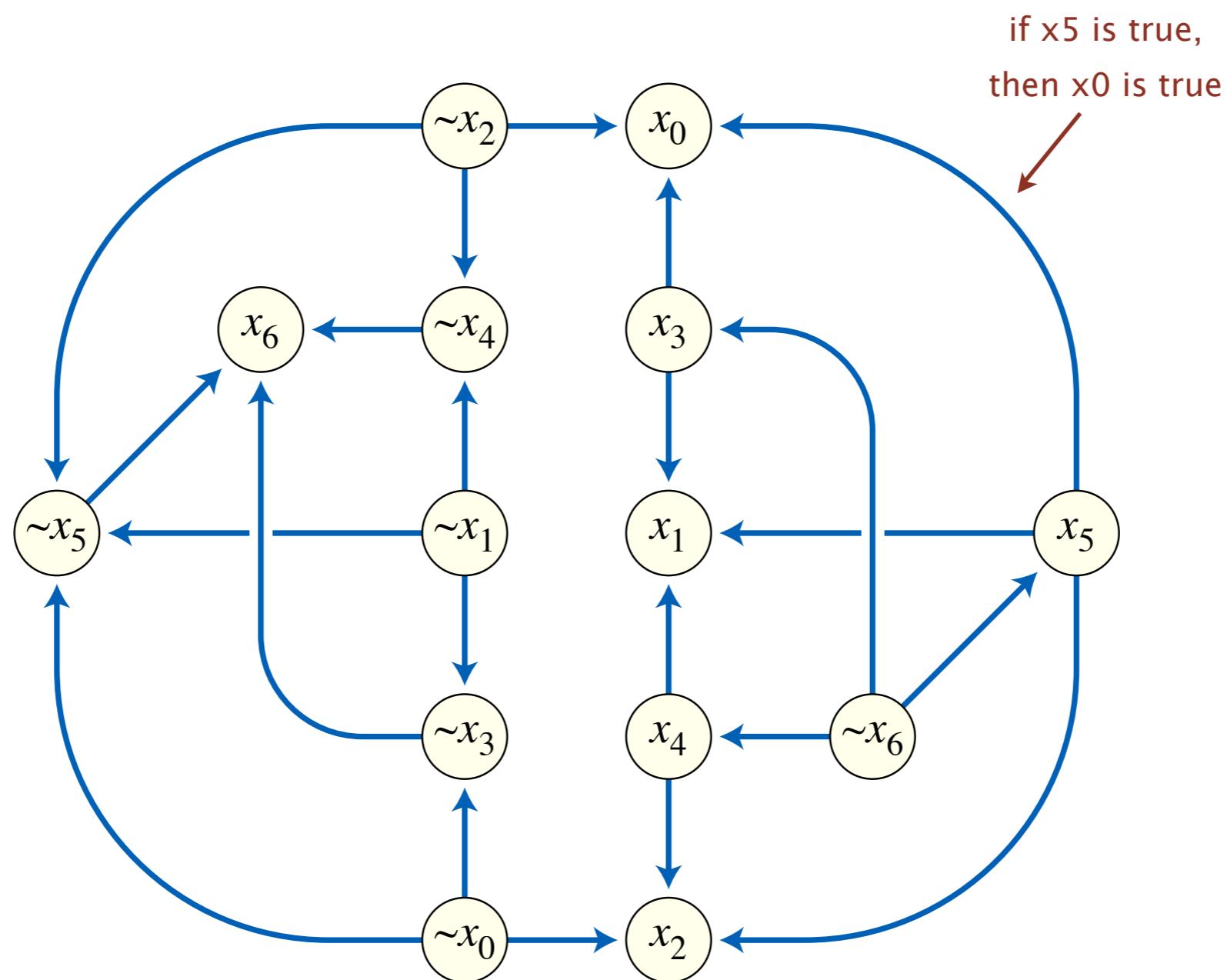
Vertex = bank; edge = overnight loan.



The Topology of the Federal Funds Market, Bech and Atalay, 2008

# Implication graph

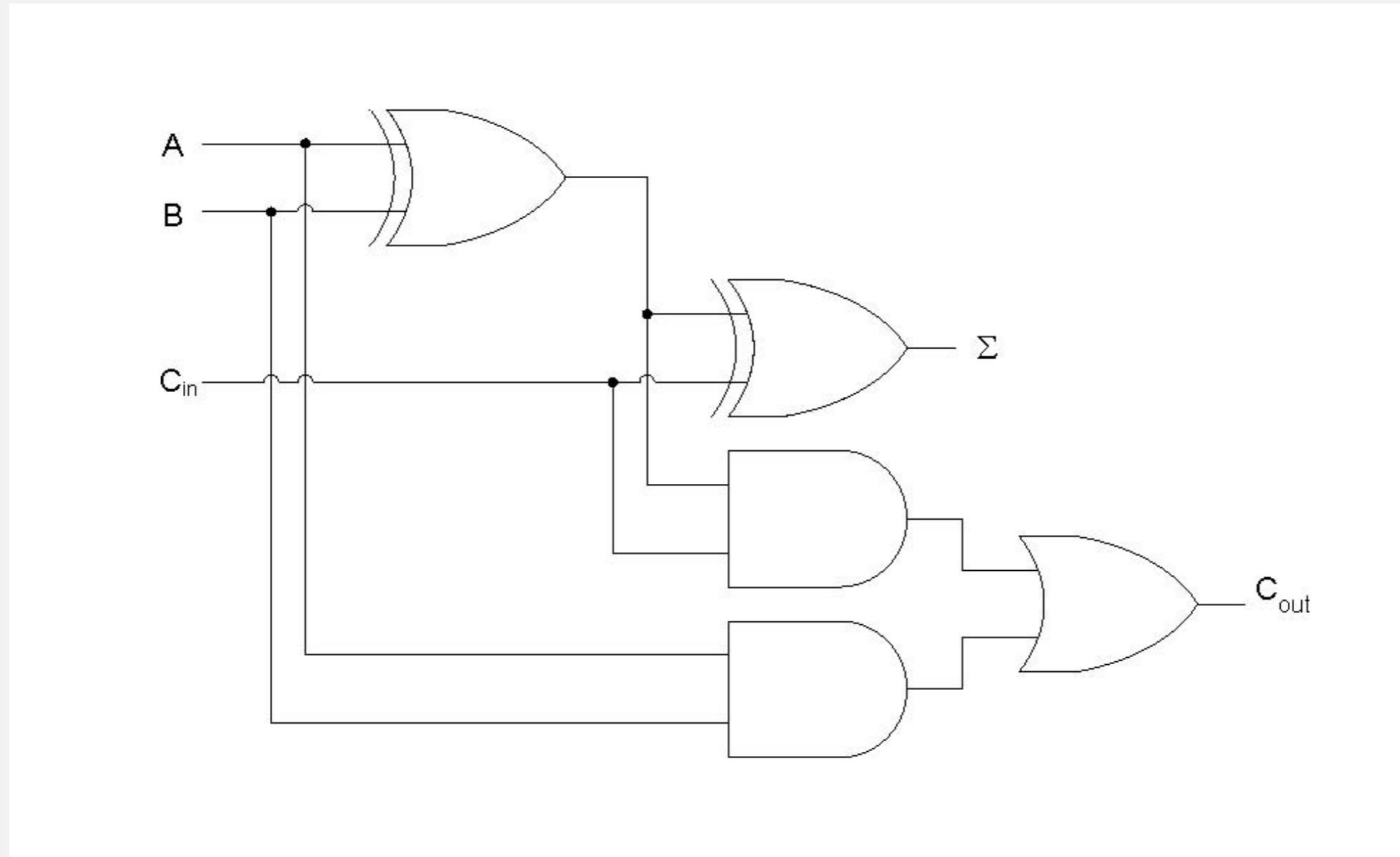
Vertex = variable; edge = logical implication.



# Combinational circuit

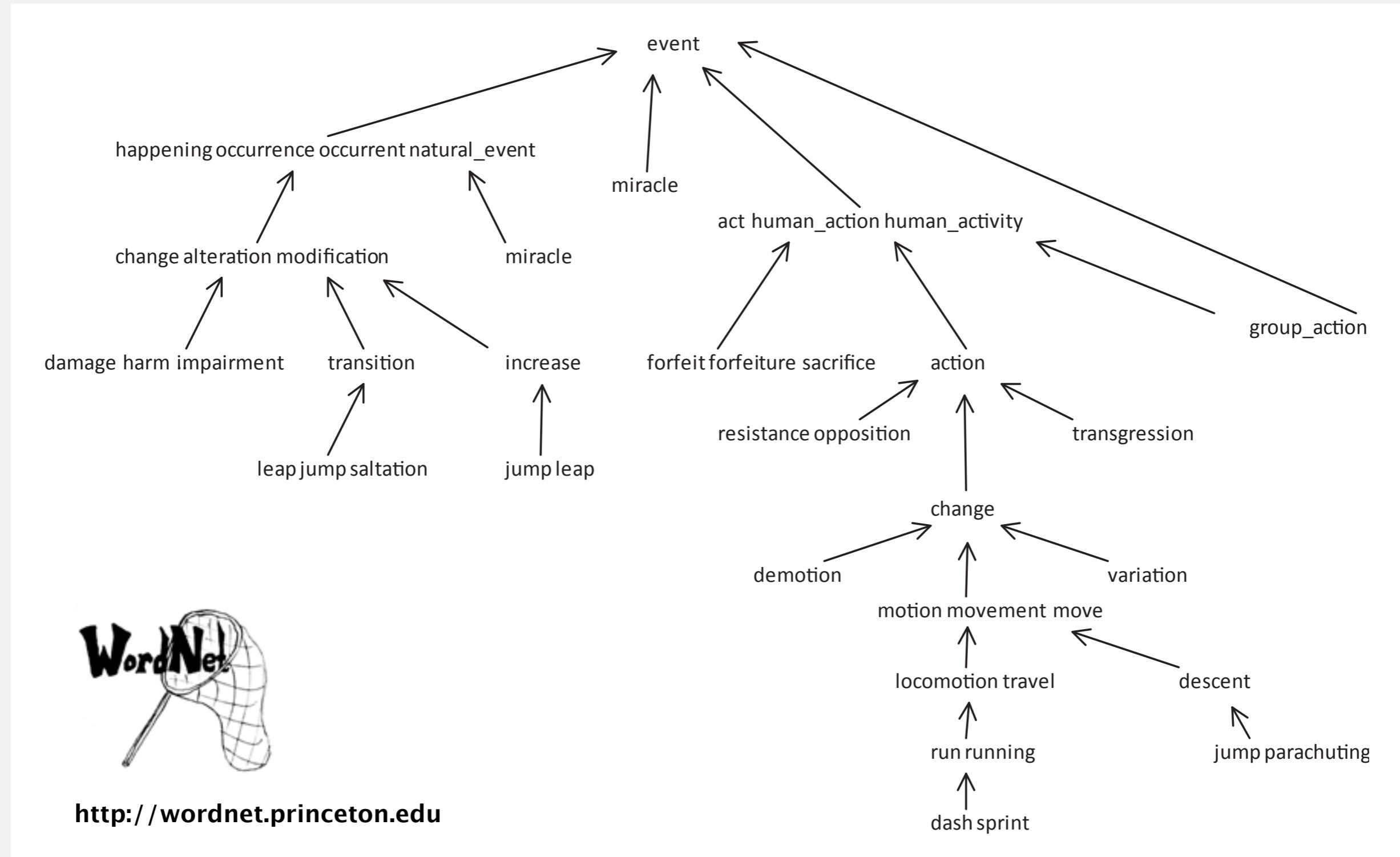
---

Vertex = logical gate; edge = wire.



# WordNet graph

Vertex = synset; edge = hypernym relationship.



# Digraph applications

---

digraph	vertex	directed edge
<b>transportation</b>	street intersection	one-way street
<b>web</b>	web page	hyperlink
<b>food web</b>	species	predator-prey relationship
<b>WordNet</b>	synset	hyponym
<b>scheduling</b>	task	precedence constraint
<b>financial</b>	bank	transaction
<b>cell phone</b>	person	placed call
<b>infectious disease</b>	person	infection
<b>game</b>	board position	legal move
<b>citation</b>	journal article	citation
<b>object graph</b>	object	pointer
<b>inheritance hierarchy</b>	class	inherits from
<b>control flow</b>	code block	jump

# Some digraph problems

---

problem	description
<b>s→t path</b>	<i>Is there a path from s to t ?</i>
<b>shortest s→t path</b>	<i>What is the shortest path from s to t ?</i>
<b>directed cycle</b>	<i>Is there a directed cycle in the graph ?</i>
<b>topological sort</b>	<i>Can the digraph be drawn so that all edges point upwards?</i>
<b>strong connectivity</b>	<i>Is there a directed path between all pairs of vertices ?</i>
<b>transitive closure</b>	<i>For which vertices v and w is there a directed path from v to w ?</i>
<b>PageRank</b>	<i>What is the importance of a web page ?</i>

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

## 4.2 DIRECTED GRAPHS

---

- ▶ *introduction*
- ▶ ***digraph API***
- ▶ *digraph search*
- ▶ *topological sort*
- ▶ *strong components*

# Digraph API

```
public class Digraph
```

```
    Digraph(int V)
```

*create an empty digraph with V vertices*

```
    Digraph(In in)
```

*create a digraph from input stream*

```
    void addEdge(int v, int w)
```

*add a directed edge  $v \rightarrow w$*

```
    Iterable<Integer> adj(int v)
```

*vertices pointing from v*

```
    int V()
```

*number of vertices*

```
    int E()
```

*number of edges*

```
    Digraph reverse()
```

*reverse of this digraph*

```
    String toString()
```

*string representation*

```
In in = new In(args[0]);
Digraph G = new Digraph(in);

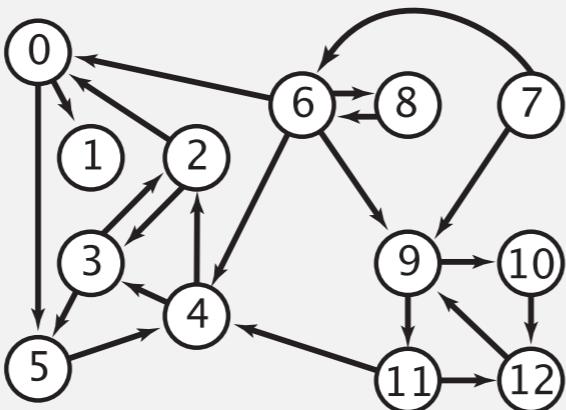
for (int v = 0; v < G.V(); v++)
    for (int w : G.adj(v))
        StdOut.println(v + "->" + w);
```

# Digraph API

**tinyDG.txt**

*V* → 13  
22 ← *E*

4 2  
2 3  
3 2  
6 0  
0 1  
2 0  
11 12  
12 9  
9 10  
9 11  
7 9  
10 12  
11 4  
4 3  
3 5  
6 8  
8 6  
:



```
% java Digraph tinyDG.txt
0->5
0->1
2->0
2->3
3->5
3->2
4->3
4->2
5->4
:
11->4
11->12
12->9
```

```
In in = new In(args[0]);
Digraph G = new Digraph(in);
```

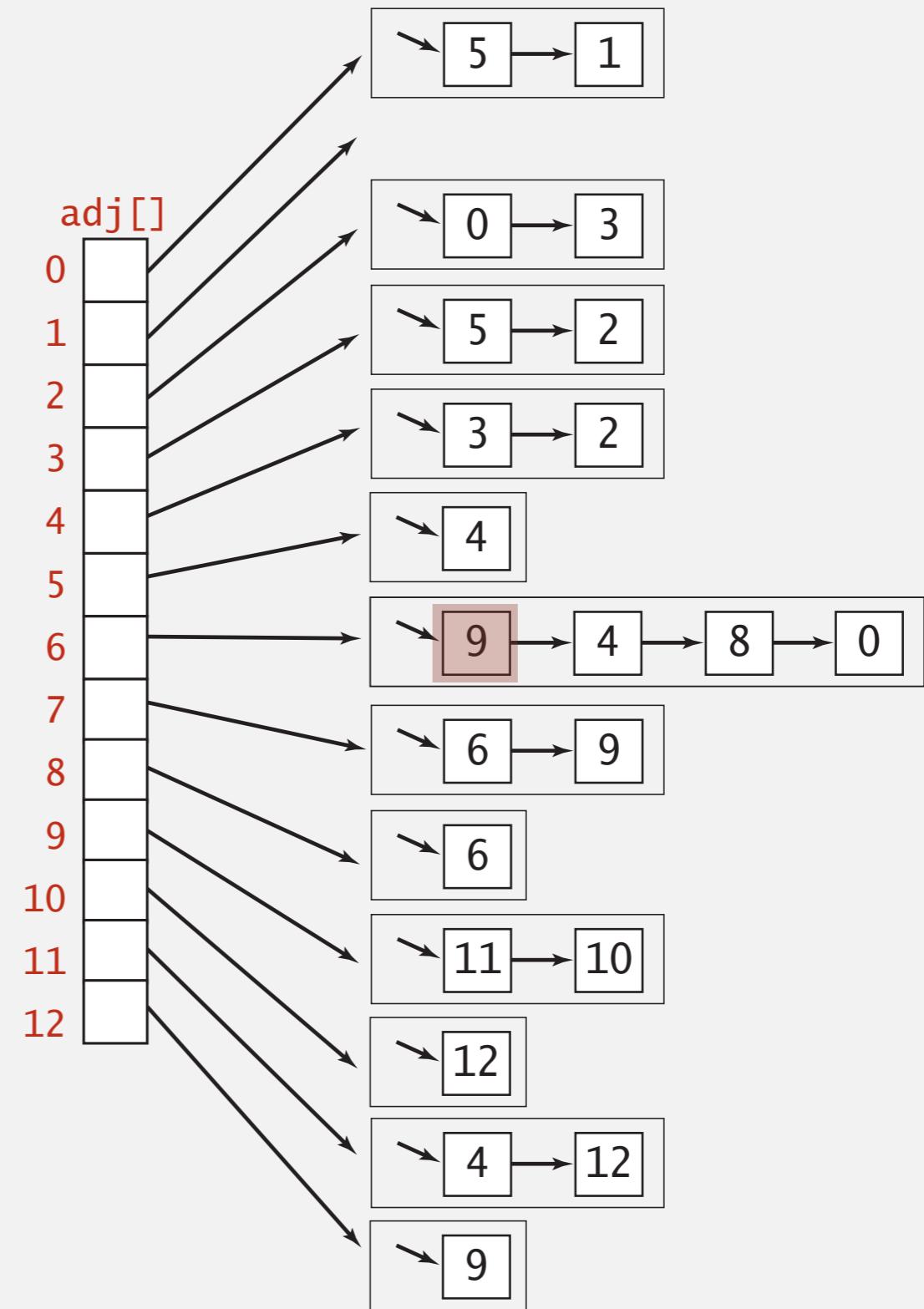
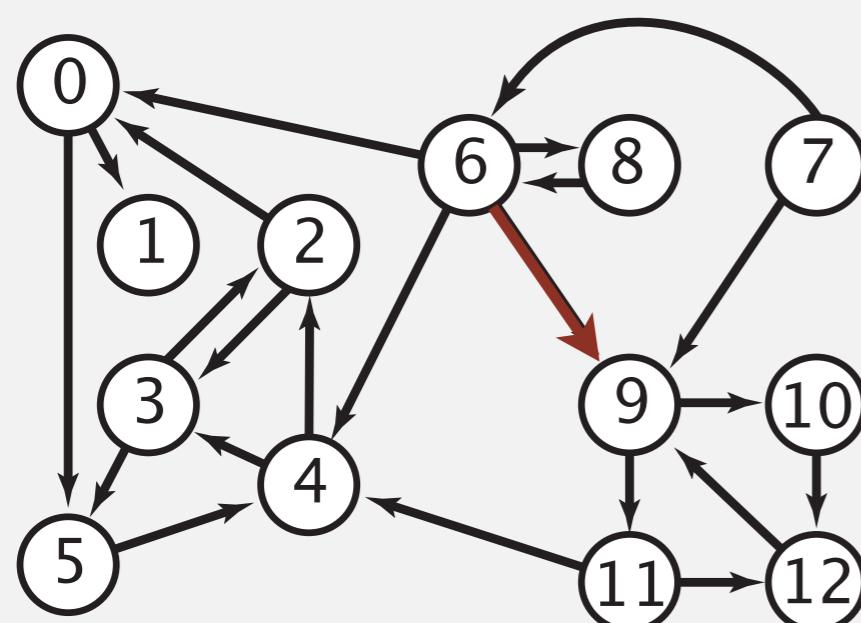
read digraph from  
input stream

```
for (int v = 0; v < G.V(); v++)
    for (int w : G.adj(v))
        StdOut.println(v + "->" + w);
```

print out each  
edge (once)

# Digraph representation: adjacency lists

Maintain vertex-indexed array of lists.



# Adjacency-lists graph representation (review): Java implementation

```
public class Graph
{
    private final int V;
    private final Bag<Integer>[] adj; ← adjacency lists

    public Graph(int V)
    {
        this.V = V;
        adj = (Bag<Integer>[]) new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<Integer>();
    }

    public void addEdge(int v, int w) ← add edge v-w
    {
        adj[v].add(w);
        adj[w].add(v);
    }

    public Iterable<Integer> adj(int v) ← iterator for vertices
    { return adj[v]; } adjacent to v
}
```

# Adjacency-lists digraph representation: Java implementation

```
public class Digraph
{
    private final int V;
    private final Bag<Integer>[] adj; ← adjacency lists

    public Digraph(int V)
    {
        this.V = V;
        adj = (Bag<Integer>[]) new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<Integer>();
    }

    public void addEdge(int v, int w) ← add edge v→w
    {
        adj[v].add(w);
    }

    public Iterable<Integer> adj(int v) ← iterator for vertices
    { return adj[v]; }                      pointing from v
}
```

# Digraph representations

In practice. Use adjacency-lists representation.

- Algorithms based on iterating over vertices pointing from  $v$ .
- Real-world digraphs tend to be sparse.

huge number of vertices,  
small average vertex degree

representation	space	insert edge from $v$ to $w$	edge from $v$ to $w$ ?	iterate over vertices pointing from $v$ ?
list of edges	$E$	1	$E$	$E$
adjacency matrix	$V^2$	$1^\dagger$	1	$V$
adjacency lists	$E + V$	1	$outdegree(v)$	$outdegree(v)$

<sup>†</sup> disallows parallel edges

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

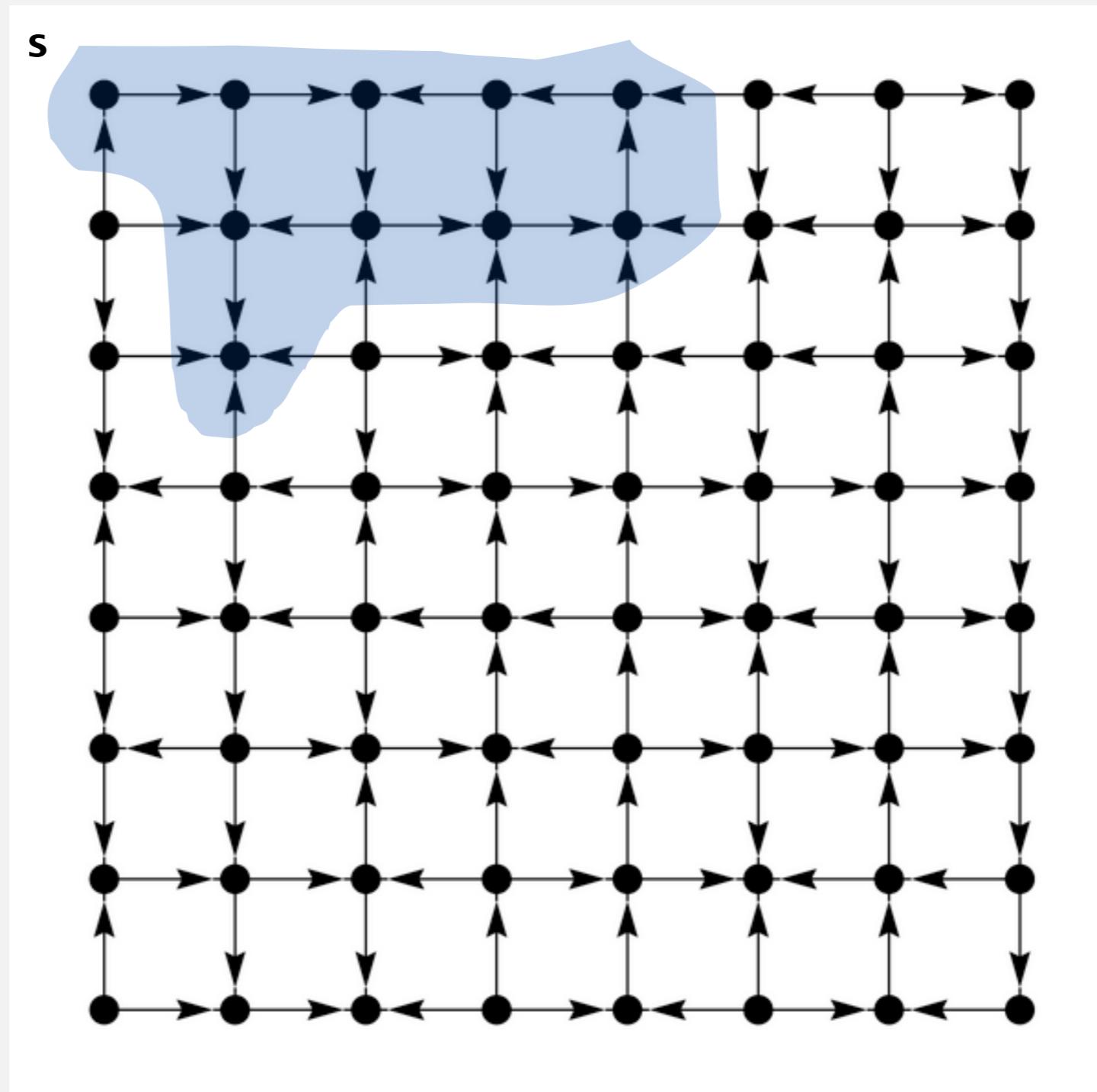
## 4.2 DIRECTED GRAPHS

---

- ▶ *introduction*
- ▶ *digraph API*
- ▶ ***digraph search***
- ▶ *topological sort*
- ▶ *strong components*

# Reachability

**Problem.** Find all vertices reachable from  $s$  along a directed path.



# Depth-first search in digraphs

---

Same method as for undirected graphs.

- Every undirected graph is a digraph (with edges in both directions).
- DFS is a **digraph** algorithm.

**DFS (to visit a vertex v)**

---

Mark v as visited.

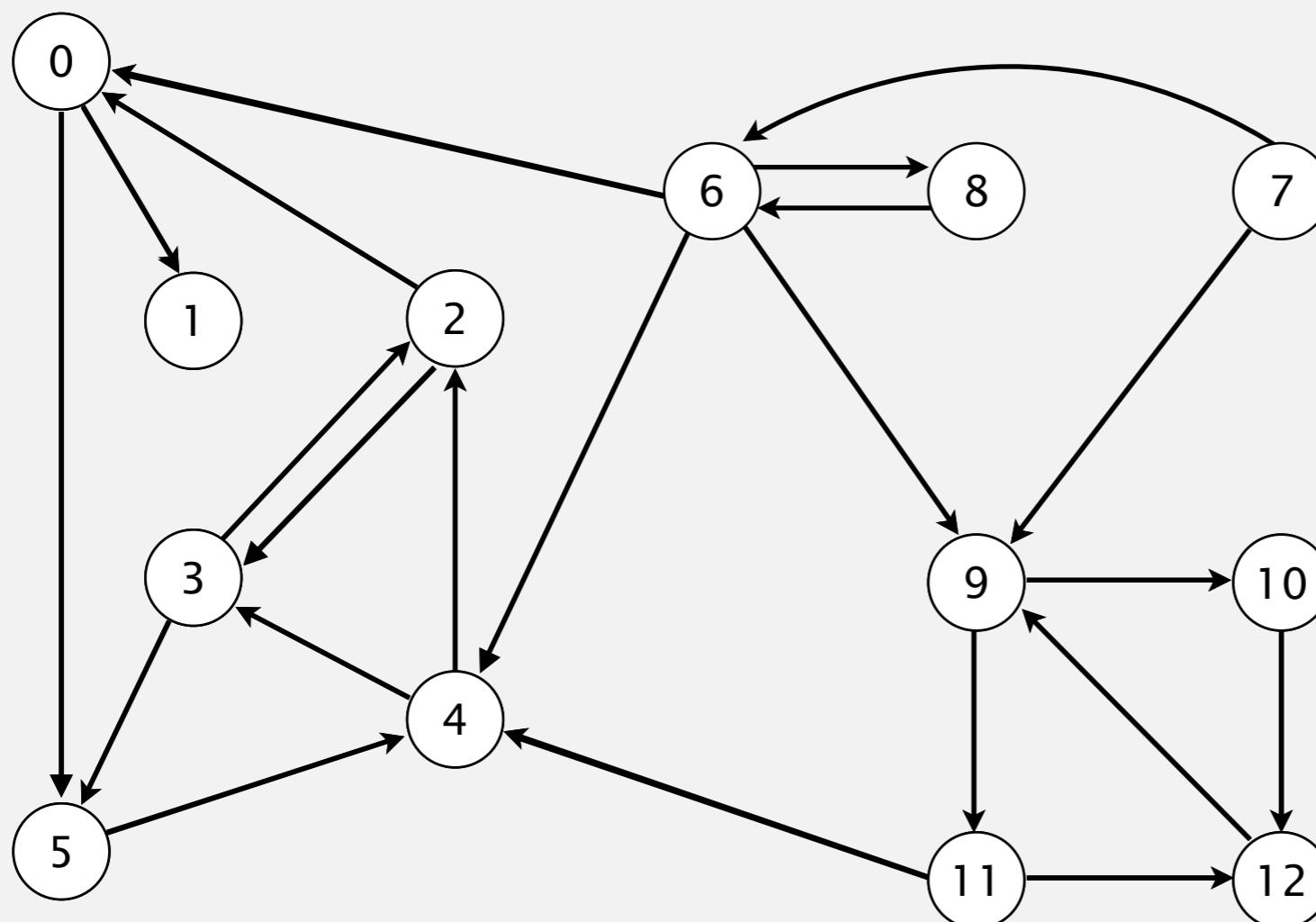
Recursively visit all unmarked  
vertices w pointing from v.

---

# Directed depth-first search demo

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices pointing from  $v$ .



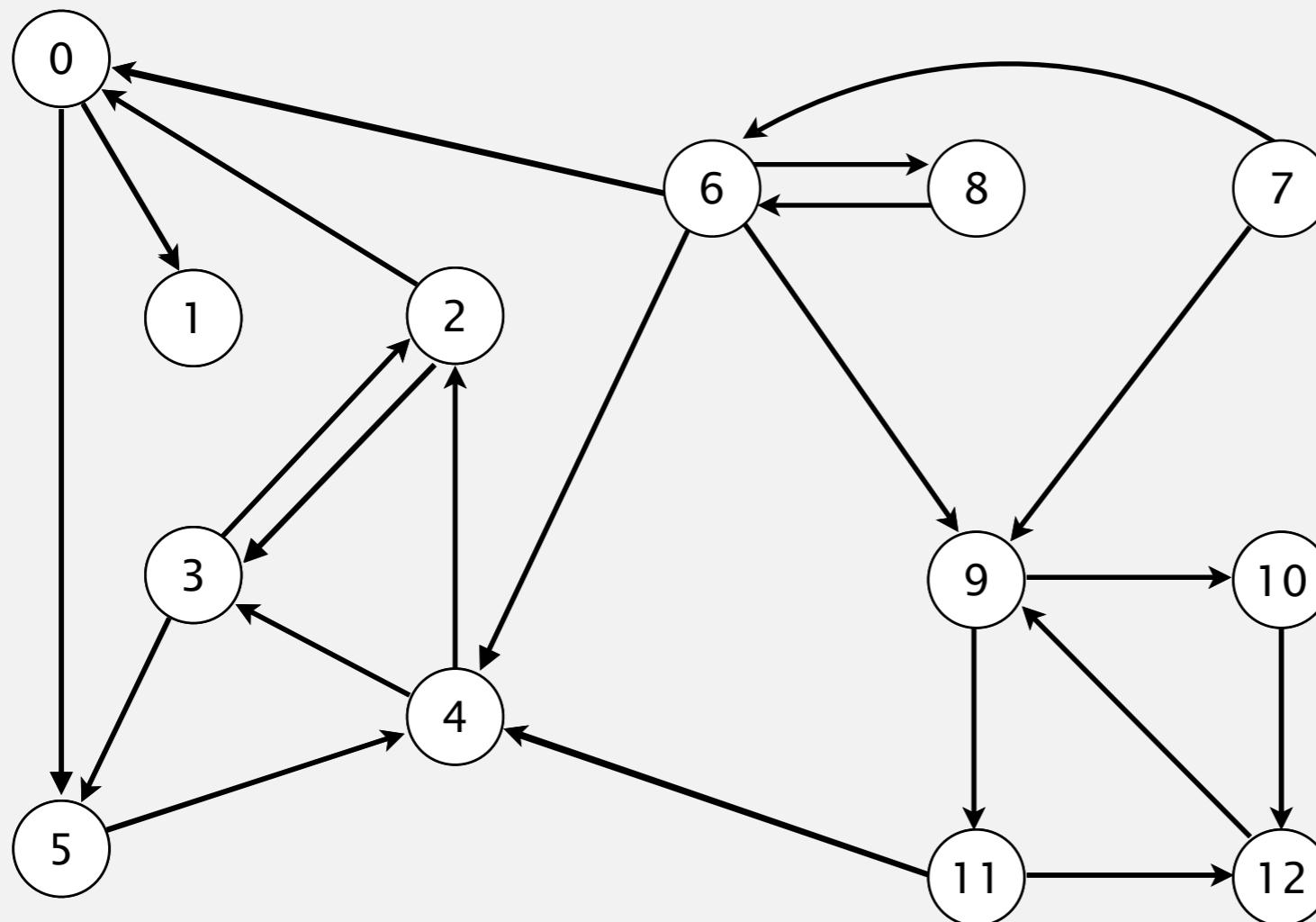
a directed graph

4→2  
2→3  
3→2  
6→0  
0→1  
2→0  
11→12  
12→9  
9→10  
9→11  
7→9  
10→12  
11→4  
4→3  
3→5  
6→8  
8→6  
5→4  
0→5

# Directed depth-first search demo

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices pointing from  $v$ .



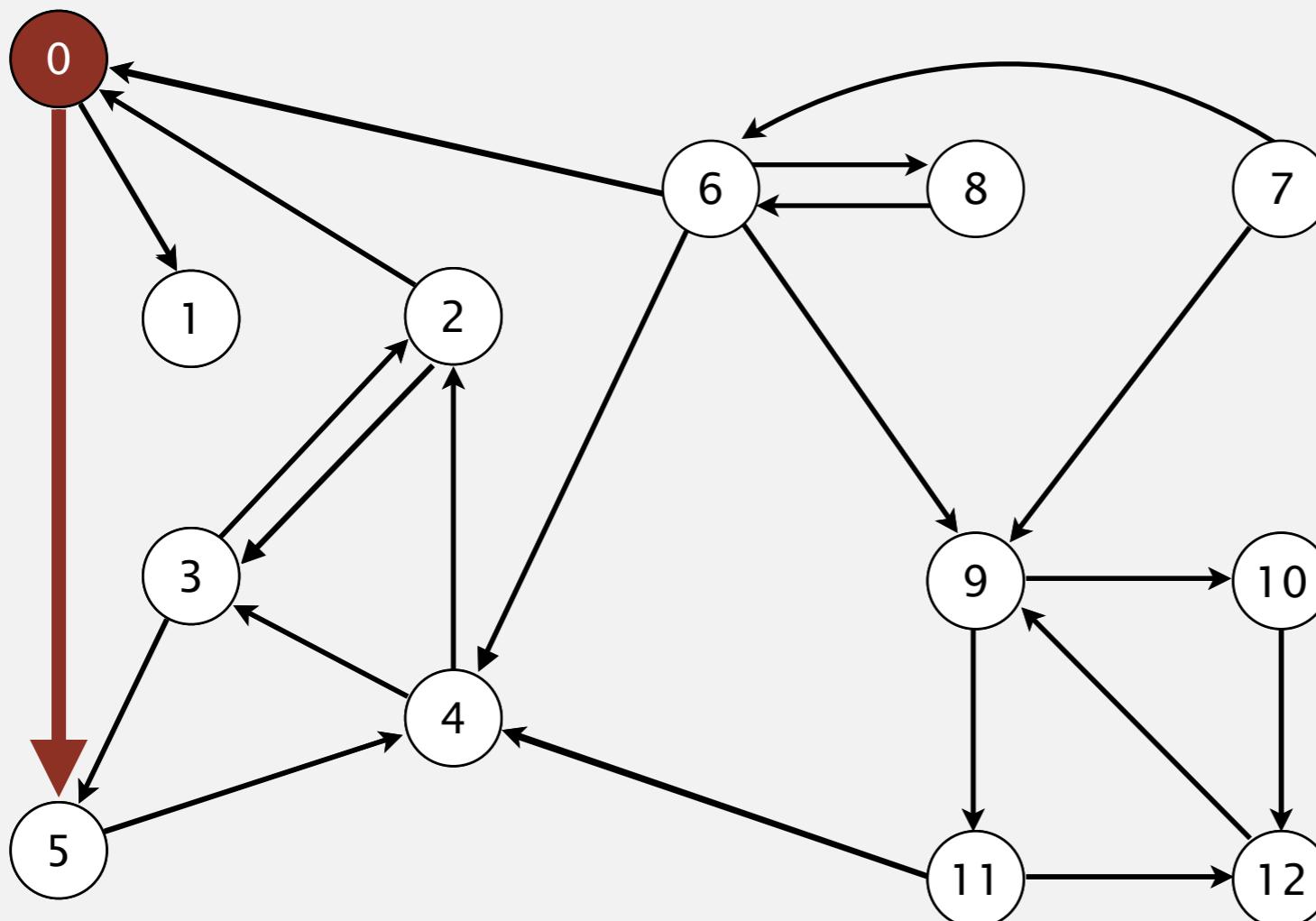
a directed graph

$v$	marked[]	edgeTo[]
0	F	-
1	F	-
2	F	-
3	F	-
4	F	-
5	F	-
6	F	-
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

# Directed depth-first search demo

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices pointing from  $v$ .



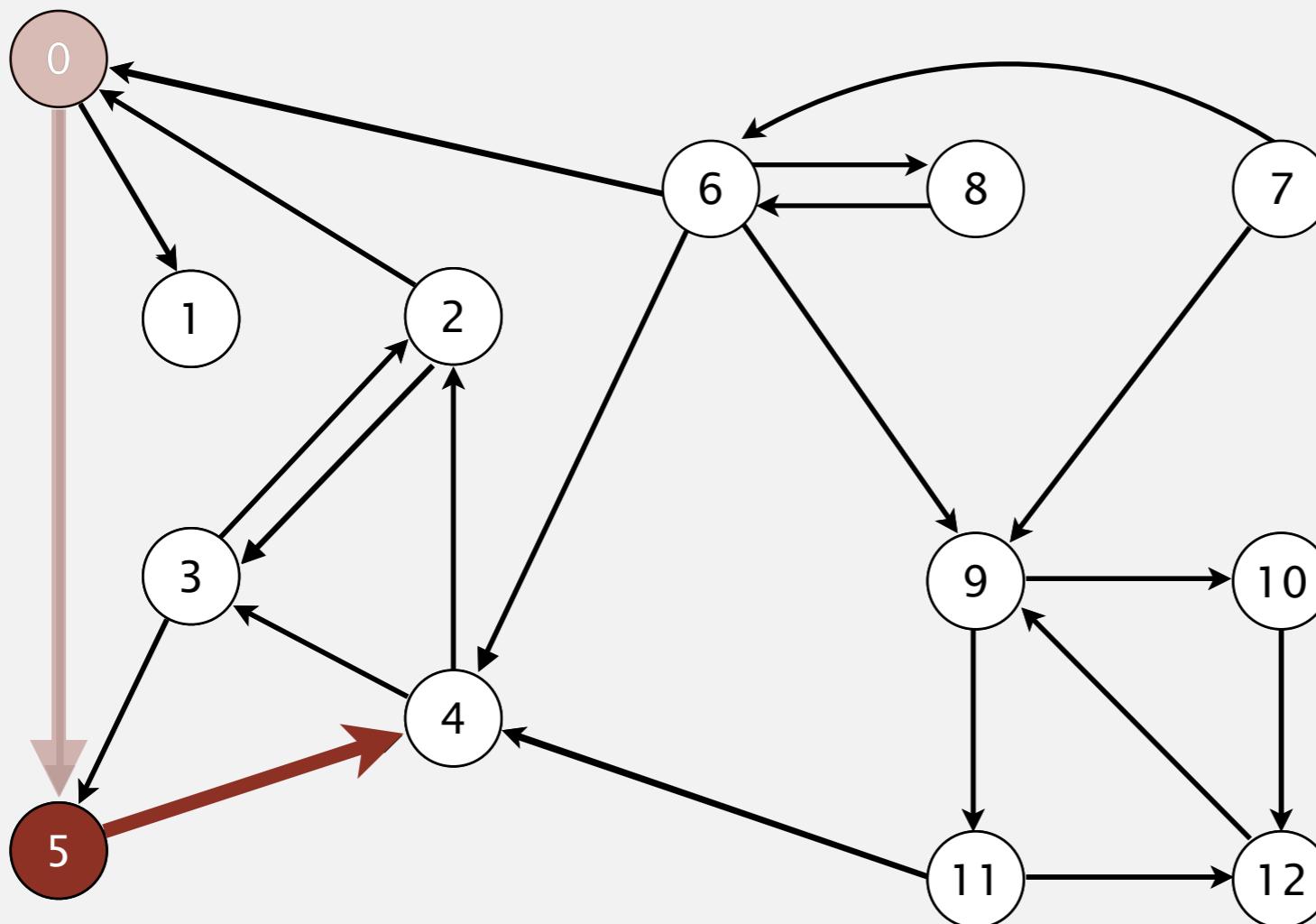
visit 0: check 5 and check 1

$v$	marked[]	edgeTo[]
0	T	-
1	F	-
2	F	-
3	F	-
4	F	-
5	F	-
6	F	-
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

# Directed depth-first search demo

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices pointing from  $v$ .

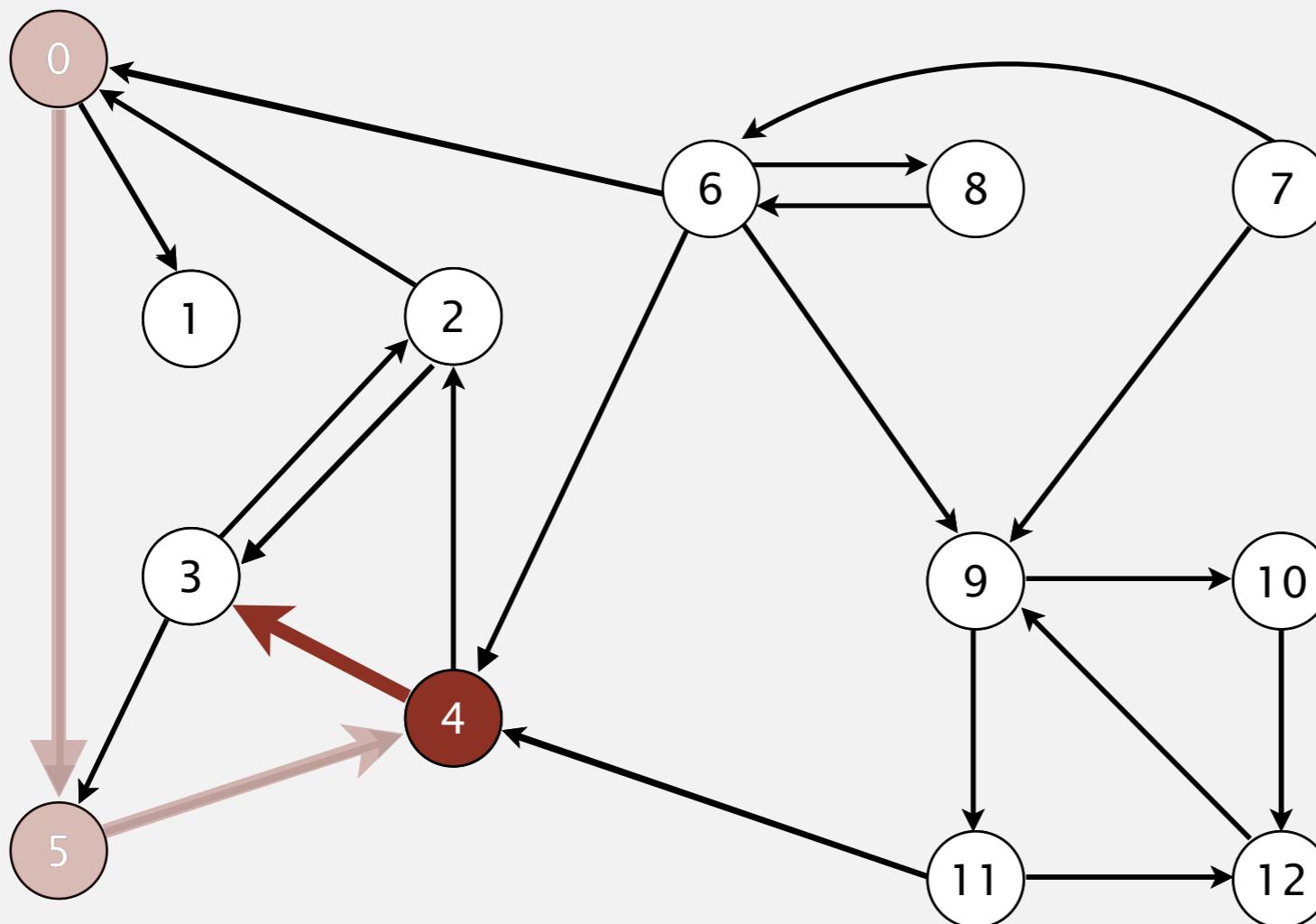


$v$	marked[]	edgeTo[]
0	T	-
1	F	-
2	F	-
3	F	-
4	F	-
5	T	0
6	F	-
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

# Directed depth-first search demo

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices pointing from  $v$ .



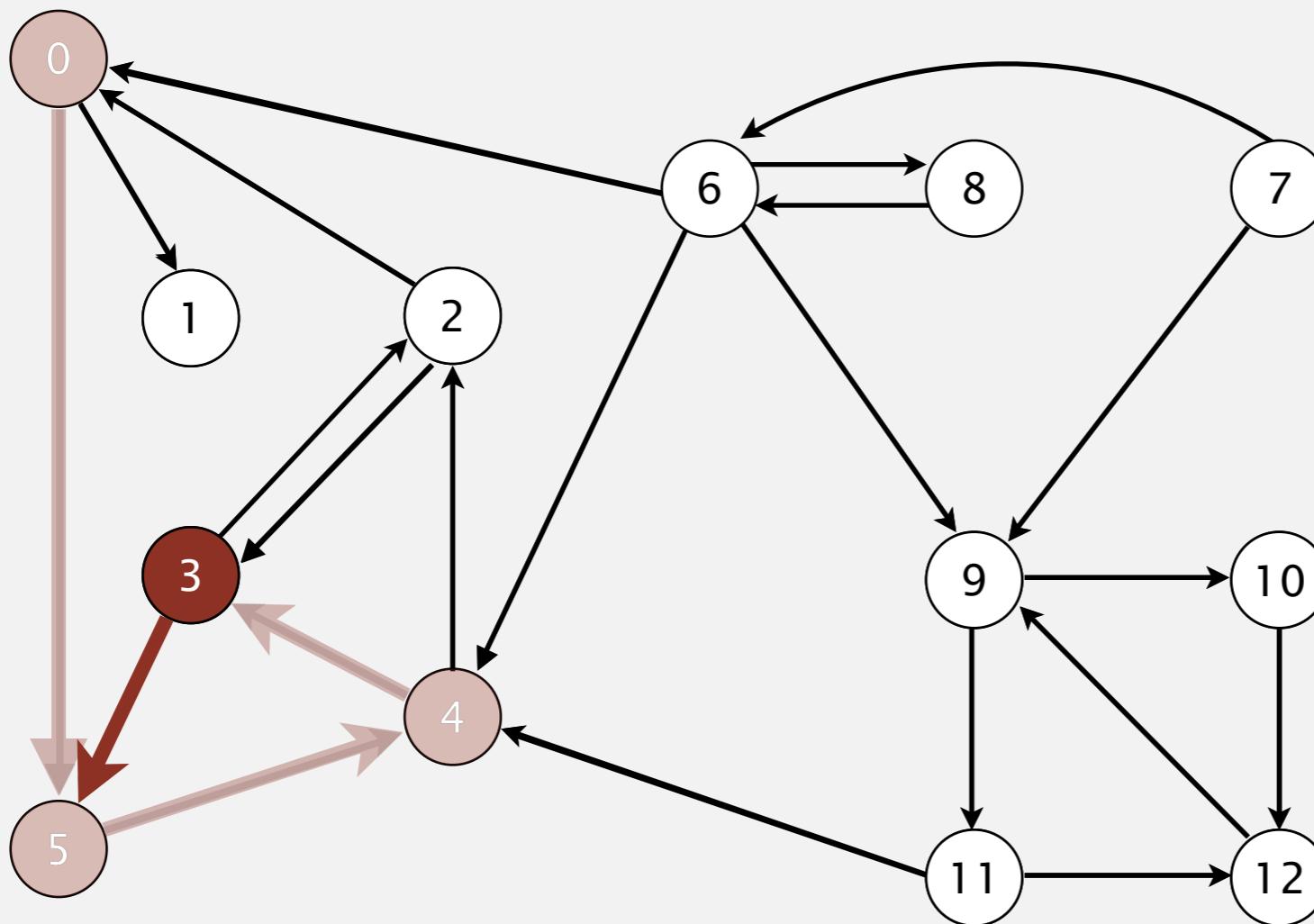
visit 4: check 3 and check 2

$v$	marked[]	edgeTo[]
0	T	-
1	F	-
2	F	-
3	F	-
4	T	5
5	T	0
6	F	-
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

# Directed depth-first search demo

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices pointing from  $v$ .



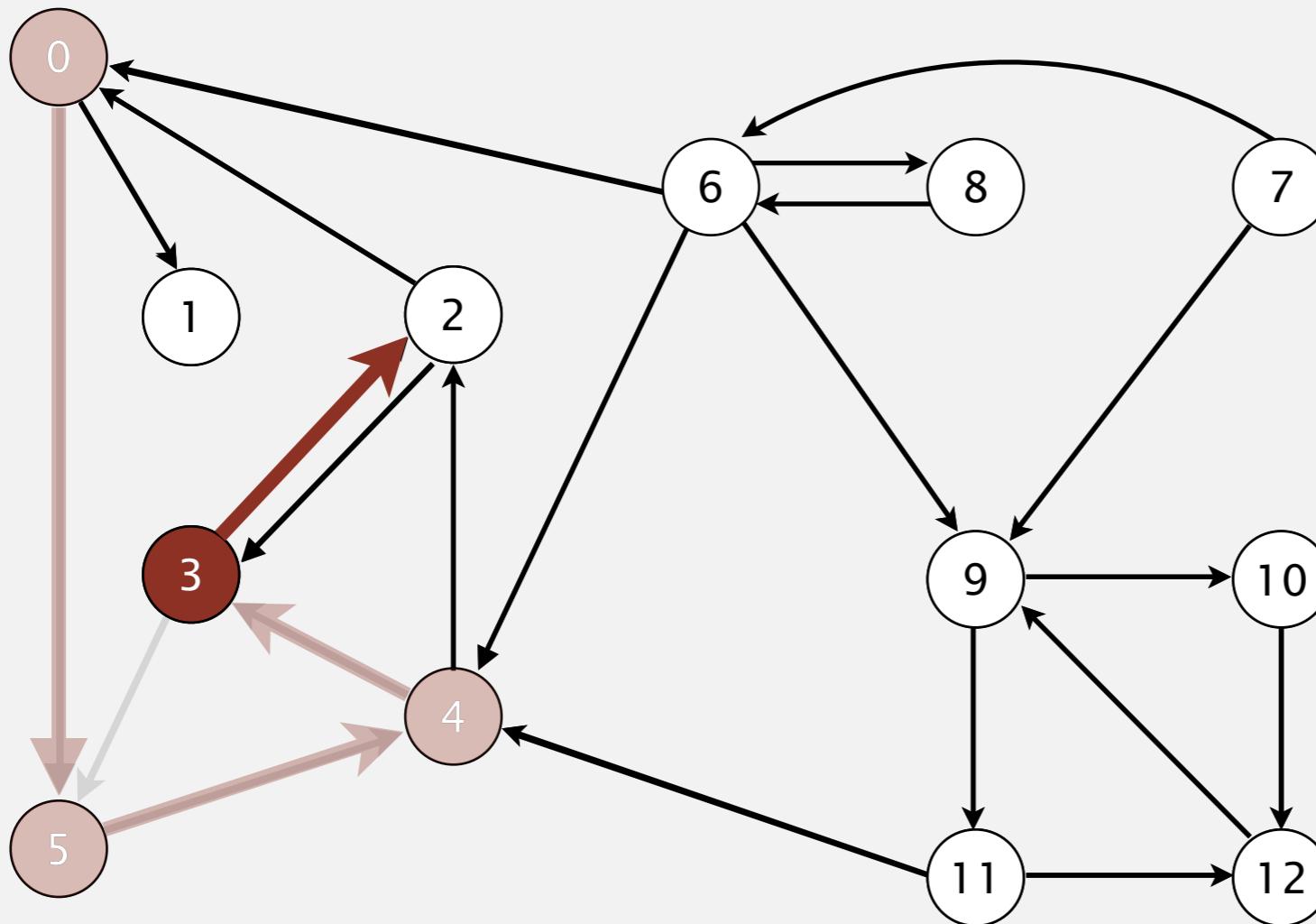
visit 3: check 5 and check 2

$v$	marked[]	edgeTo[]
0	T	-
1	F	-
2	F	-
3	T	4
4	T	5
5	T	0
6	F	-
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

# Directed depth-first search demo

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices pointing from  $v$ .



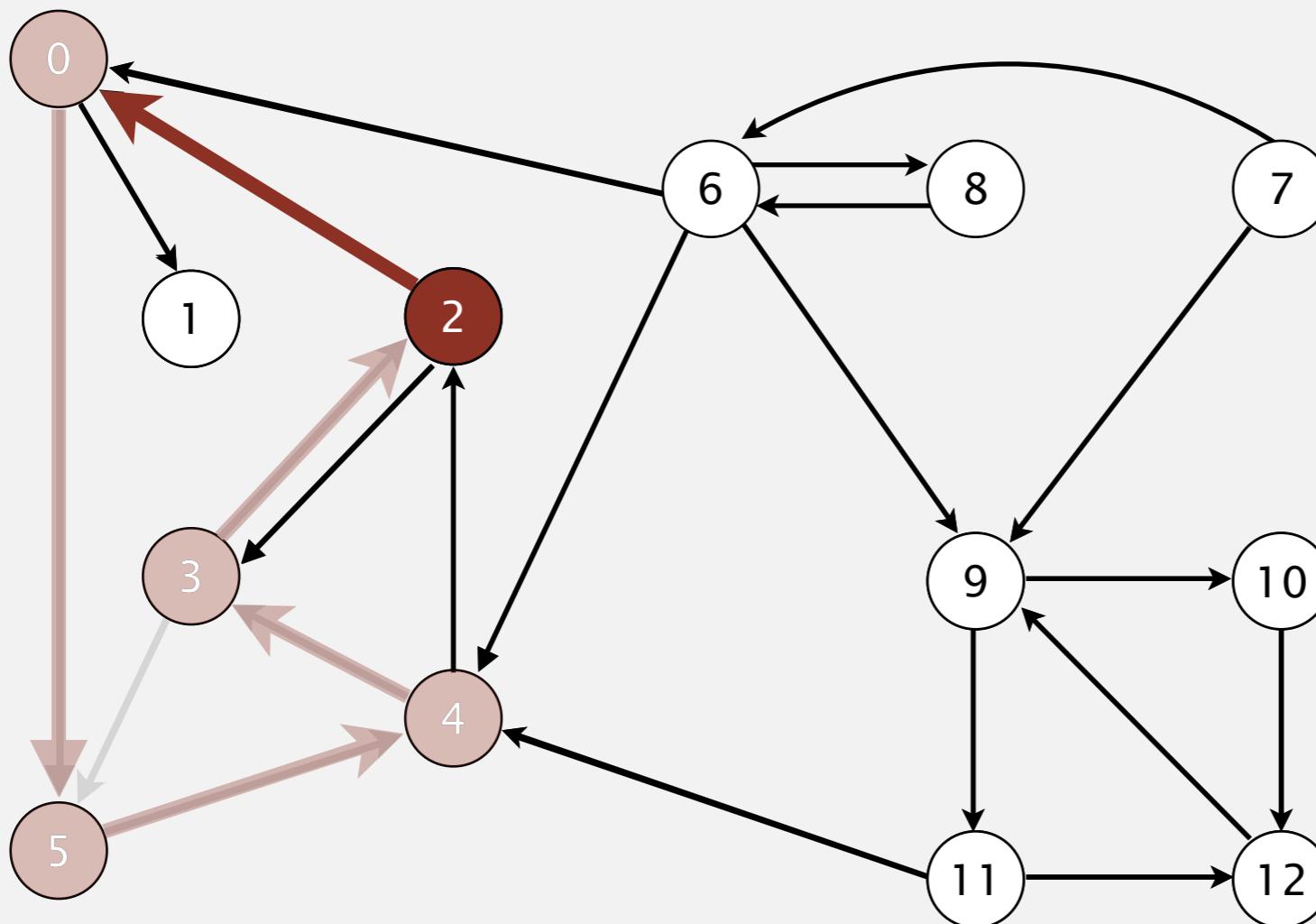
visit 3: check 5 and check 2

$v$	marked[]	edgeTo[]
0	T	-
1	F	-
2	F	-
3	T	4
4	T	5
5	T	0
6	F	-
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

# Directed depth-first search demo

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices pointing from  $v$ .



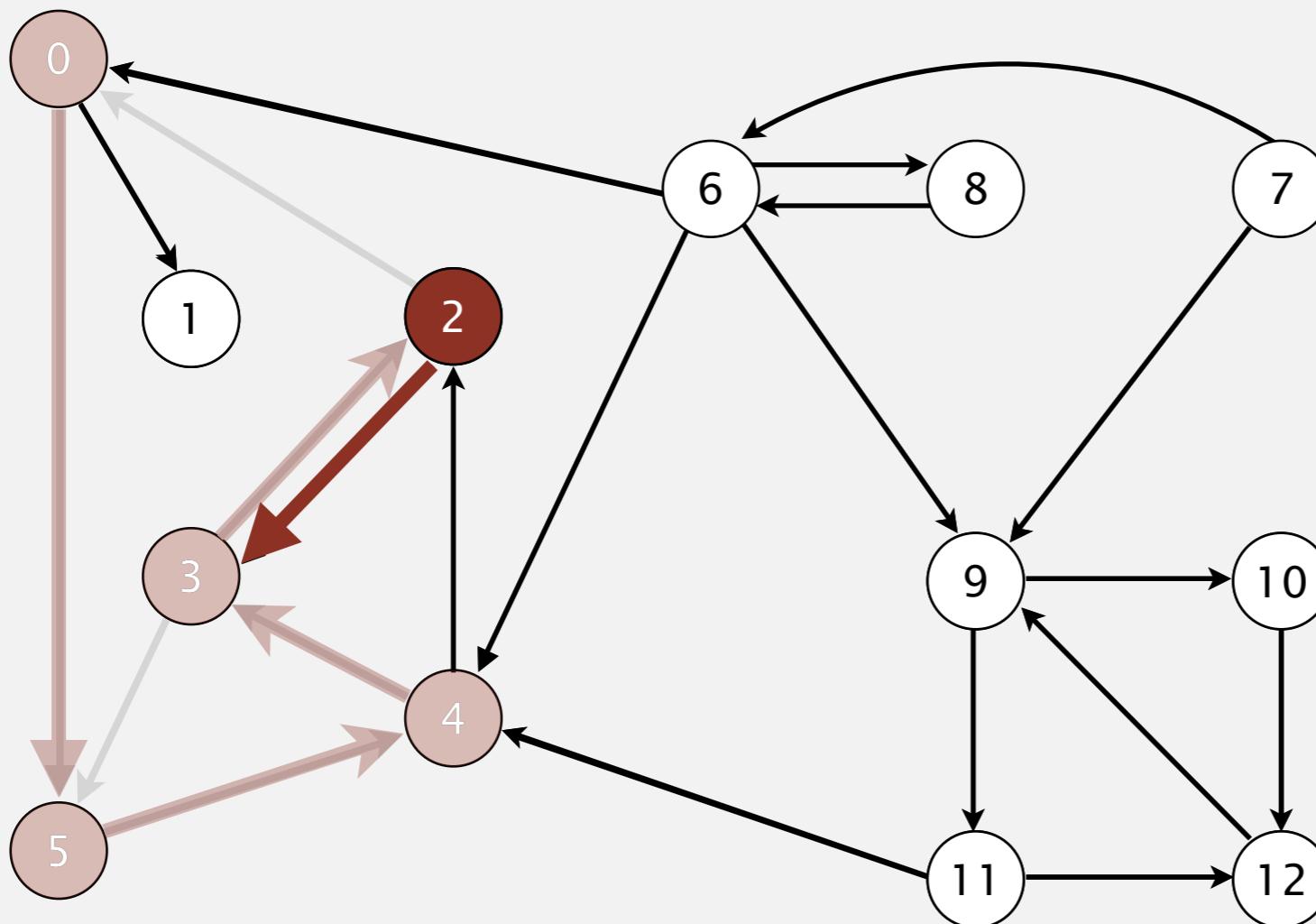
visit 2: check 0 and check 3

$v$	marked[]	edgeTo[]
0	T	-
1	F	-
2	T	3
3	T	4
4	T	5
5	T	0
6	F	-
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

# Directed depth-first search demo

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices pointing from  $v$ .



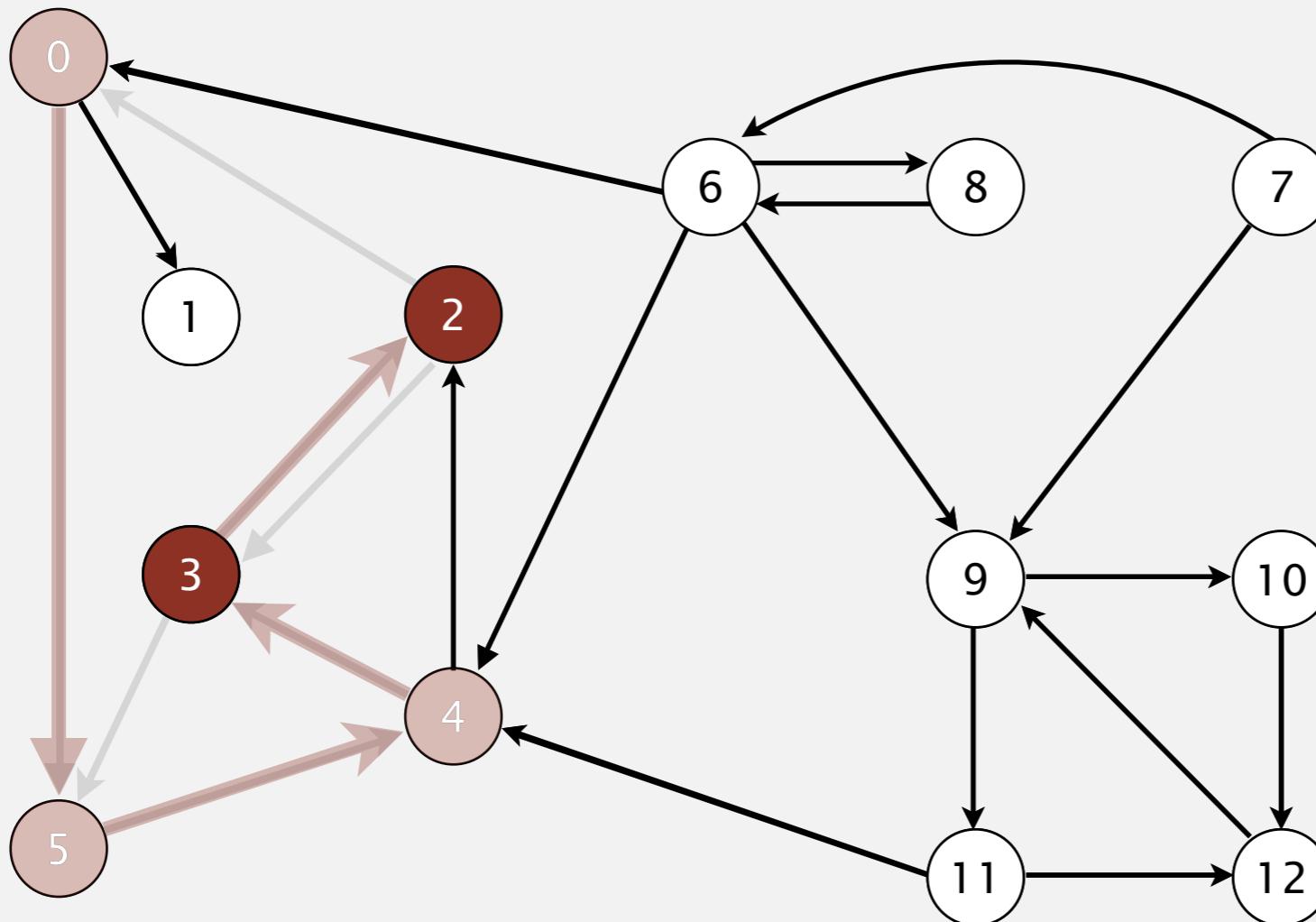
visit 2: check 0 and check 3

$v$	marked[]	edgeTo[]
0	T	-
1	F	-
2	T	3
3	T	4
4	T	5
5	T	0
6	F	-
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

# Directed depth-first search demo

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices pointing from  $v$ .



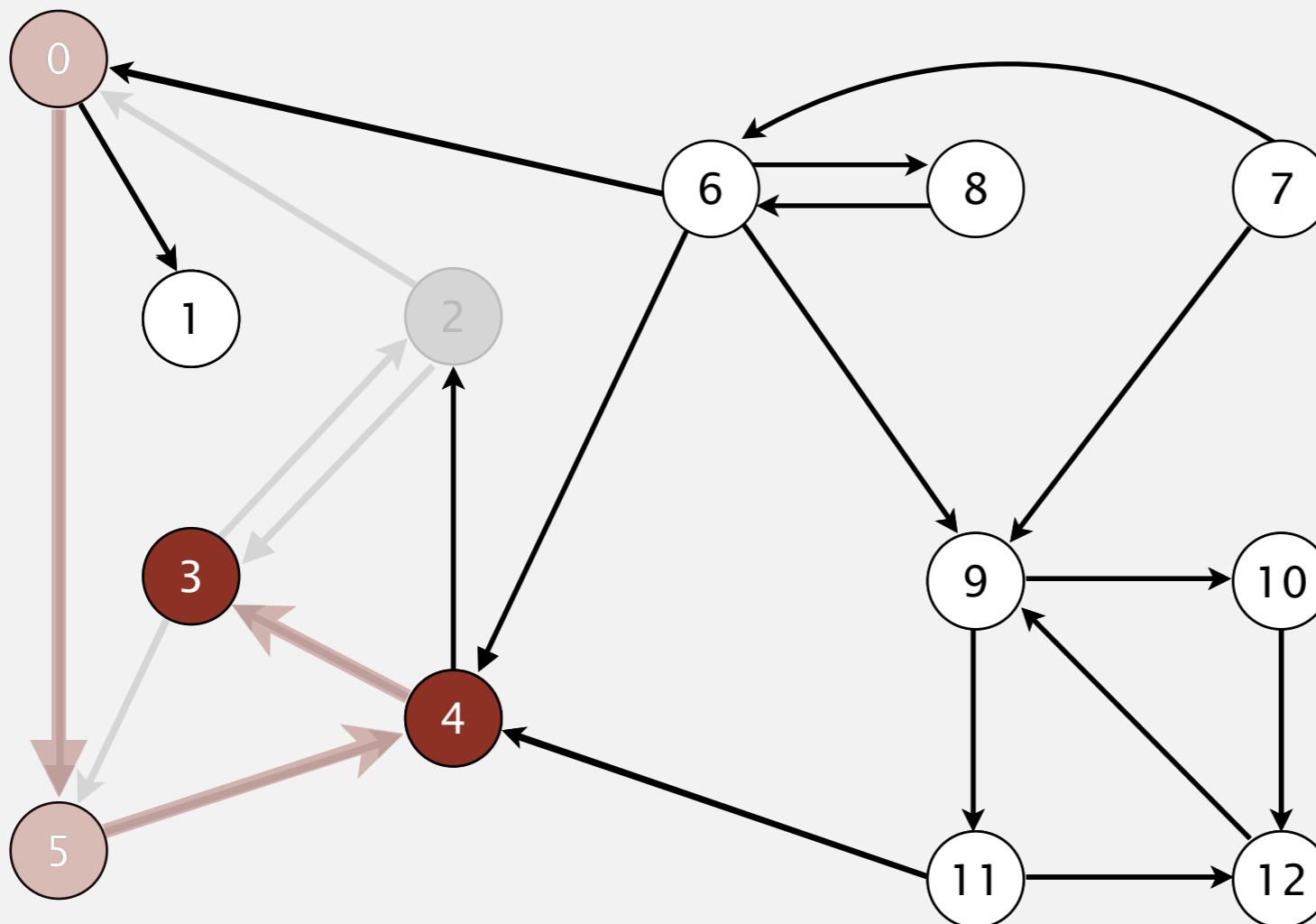
done 2

$v$	marked[]	edgeTo[]
0	T	-
1	F	-
2	T	3
3	T	4
4	T	5
5	T	0
6	F	-
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

# Directed depth-first search demo

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices pointing from  $v$ .

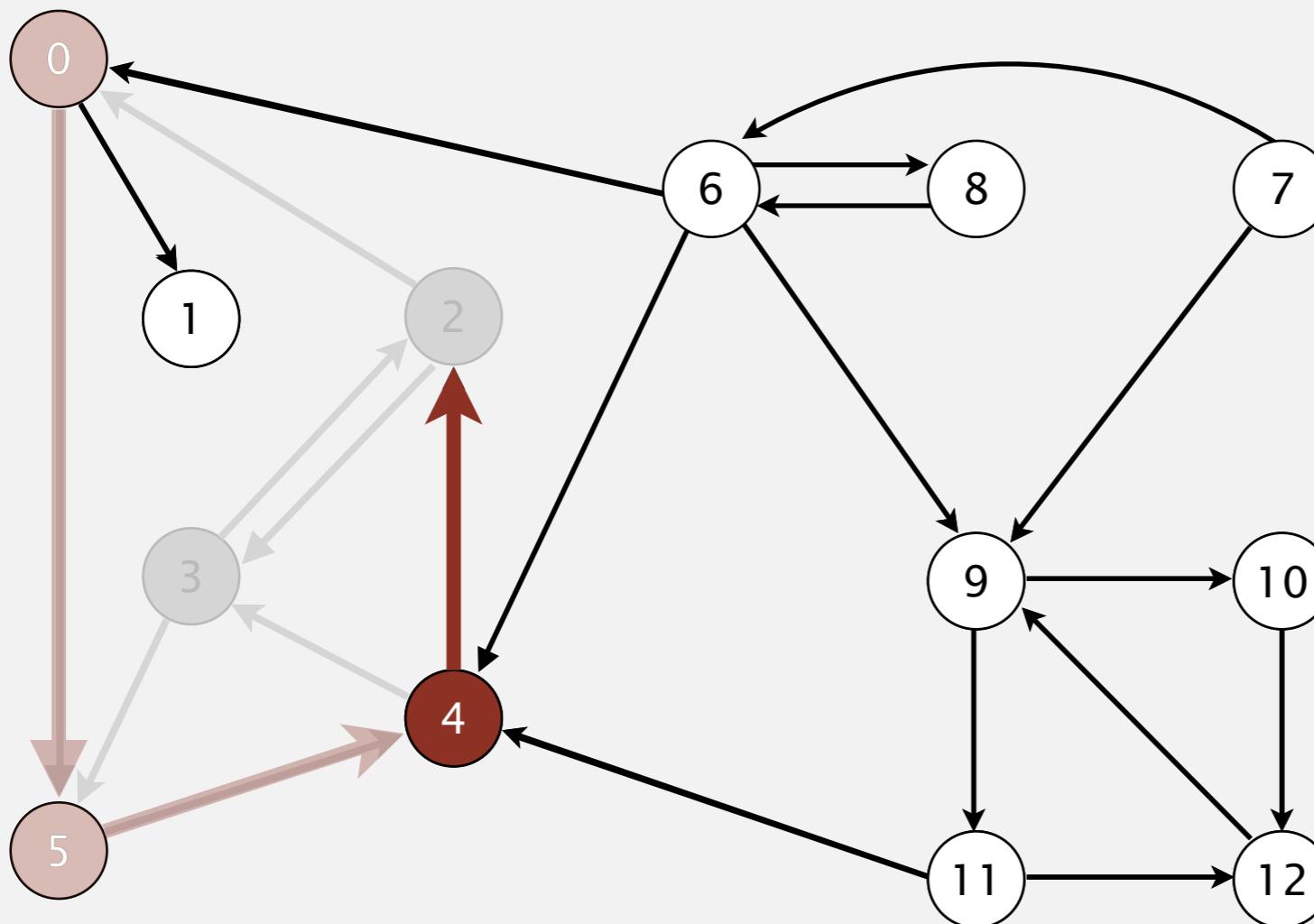


$v$	marked[]	edgeTo[]
0	T	-
1	F	-
2	T	3
3	T	4
4	T	5
5	T	0
6	F	-
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

# Directed depth-first search demo

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices pointing from  $v$ .



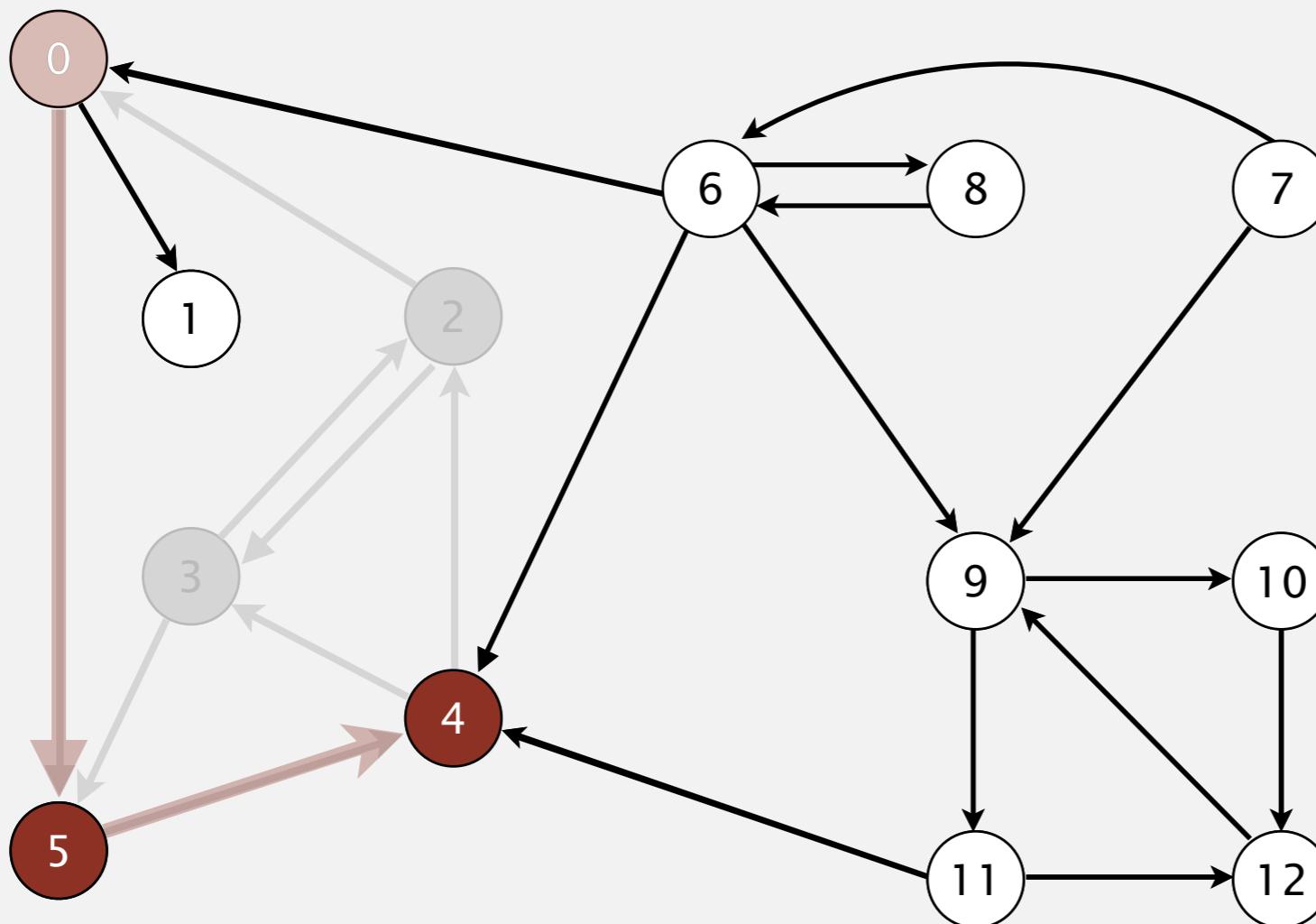
visit 4: check 3 and check 2

$v$	marked[]	edgeTo[]
0	T	-
1	F	-
2	T	3
3	T	4
4	T	5
5	T	0
6	F	-
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

# Directed depth-first search demo

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices pointing from  $v$ .



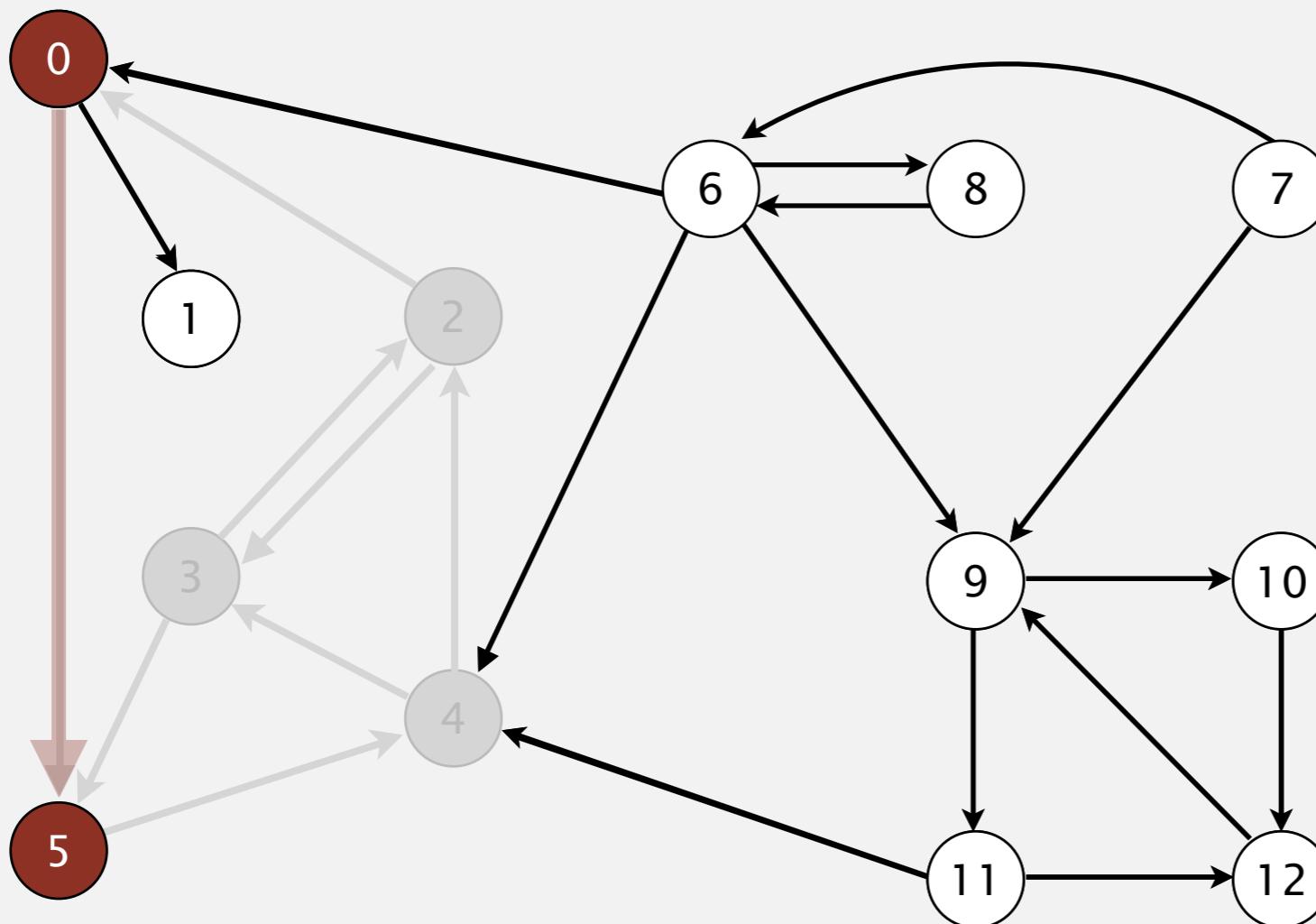
done 4

$v$	marked[]	edgeTo[]
0	T	-
1	F	-
2	T	3
3	T	4
4	T	5
5	T	0
6	F	-
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

# Directed depth-first search demo

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices pointing from  $v$ .

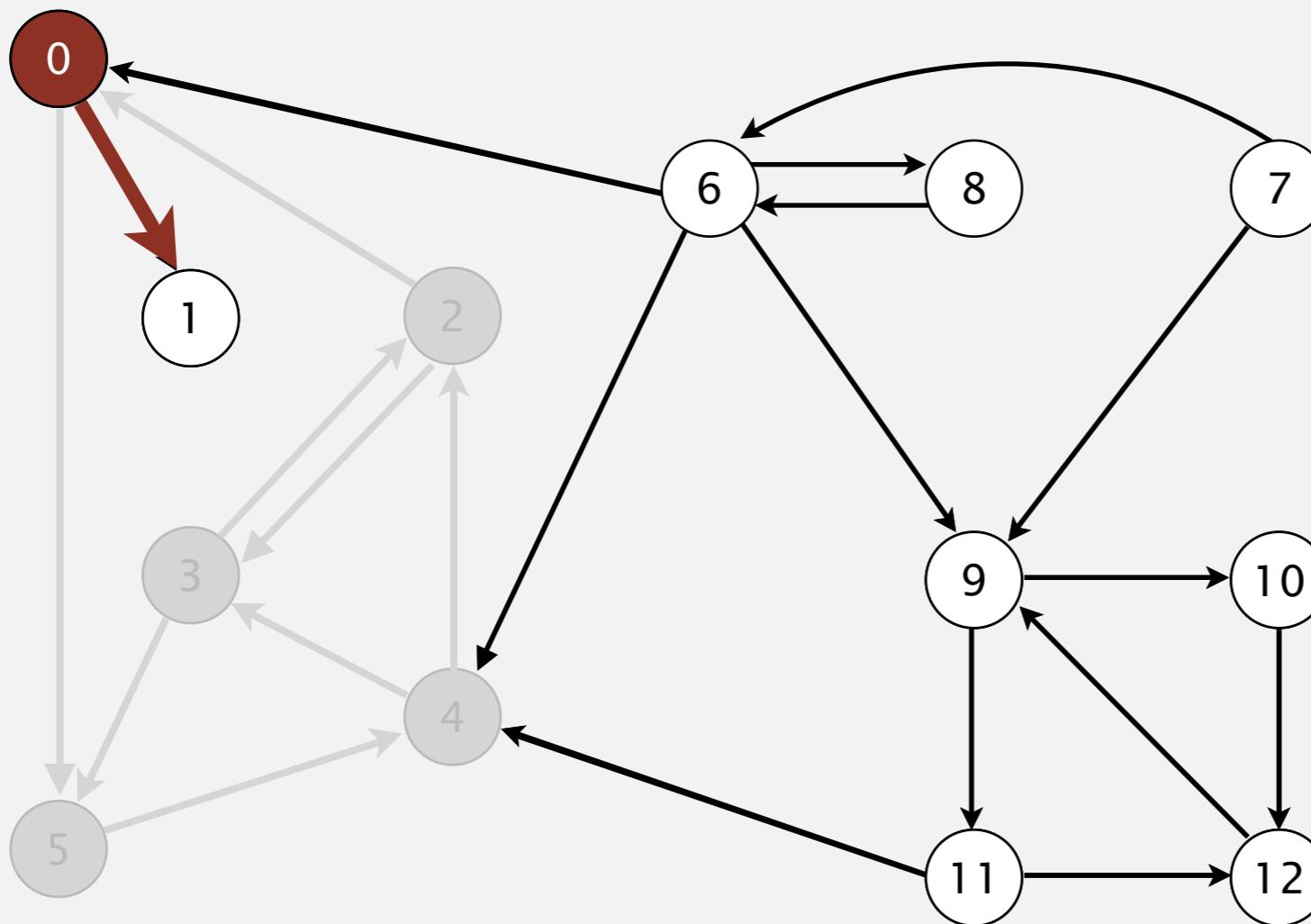


$v$	marked[]	edgeTo[]
0	T	-
1	F	-
2	T	3
3	T	4
4	T	5
5	T	0
6	F	-
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

# Directed depth-first search demo

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices pointing from  $v$ .



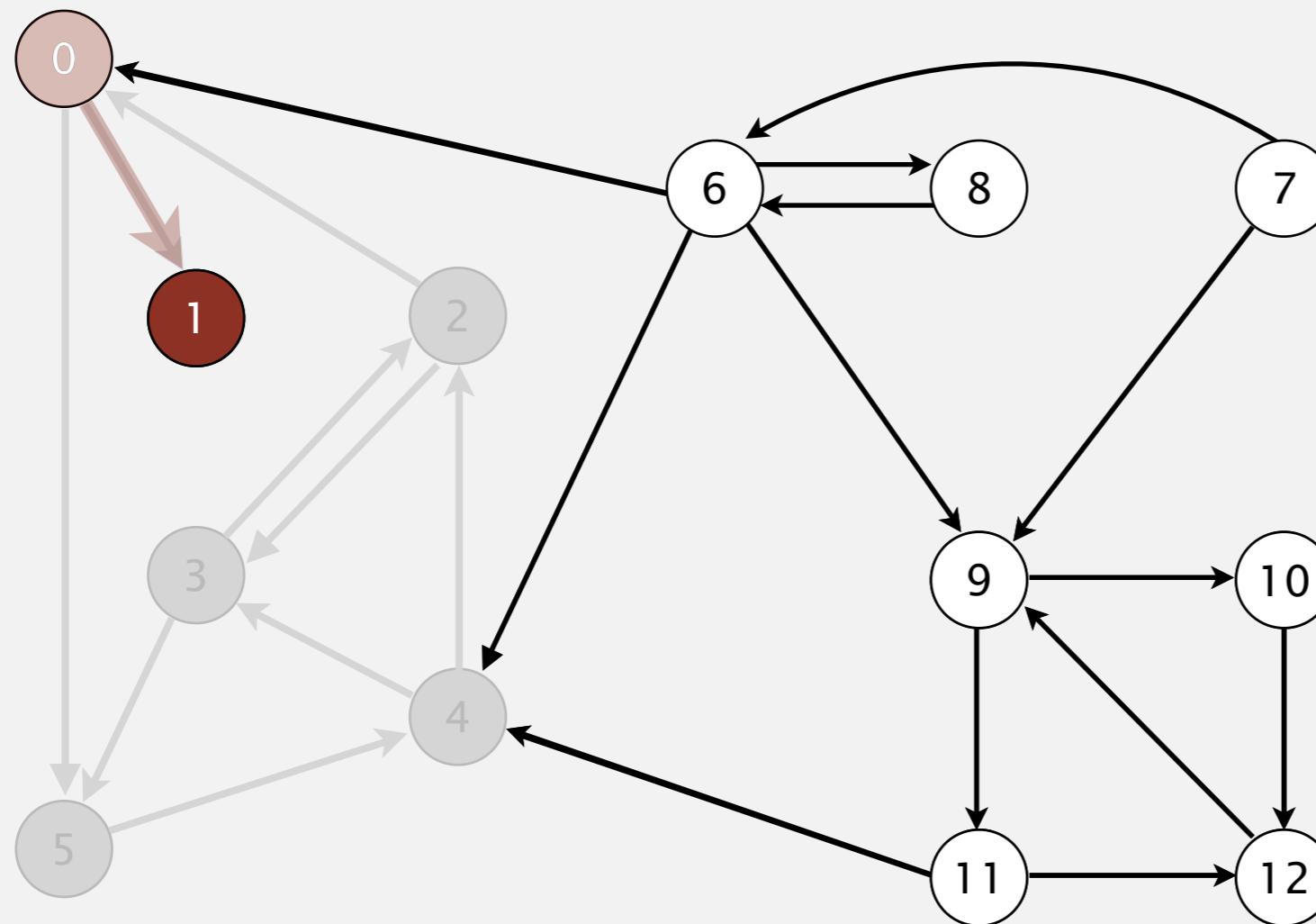
visit 0: check 5 and check 1

$v$	marked[]	edgeTo[]
0	T	-
1	F	-
2	T	3
3	T	4
4	T	5
5	T	0
6	F	-
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

# Directed depth-first search demo

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices pointing from  $v$ .

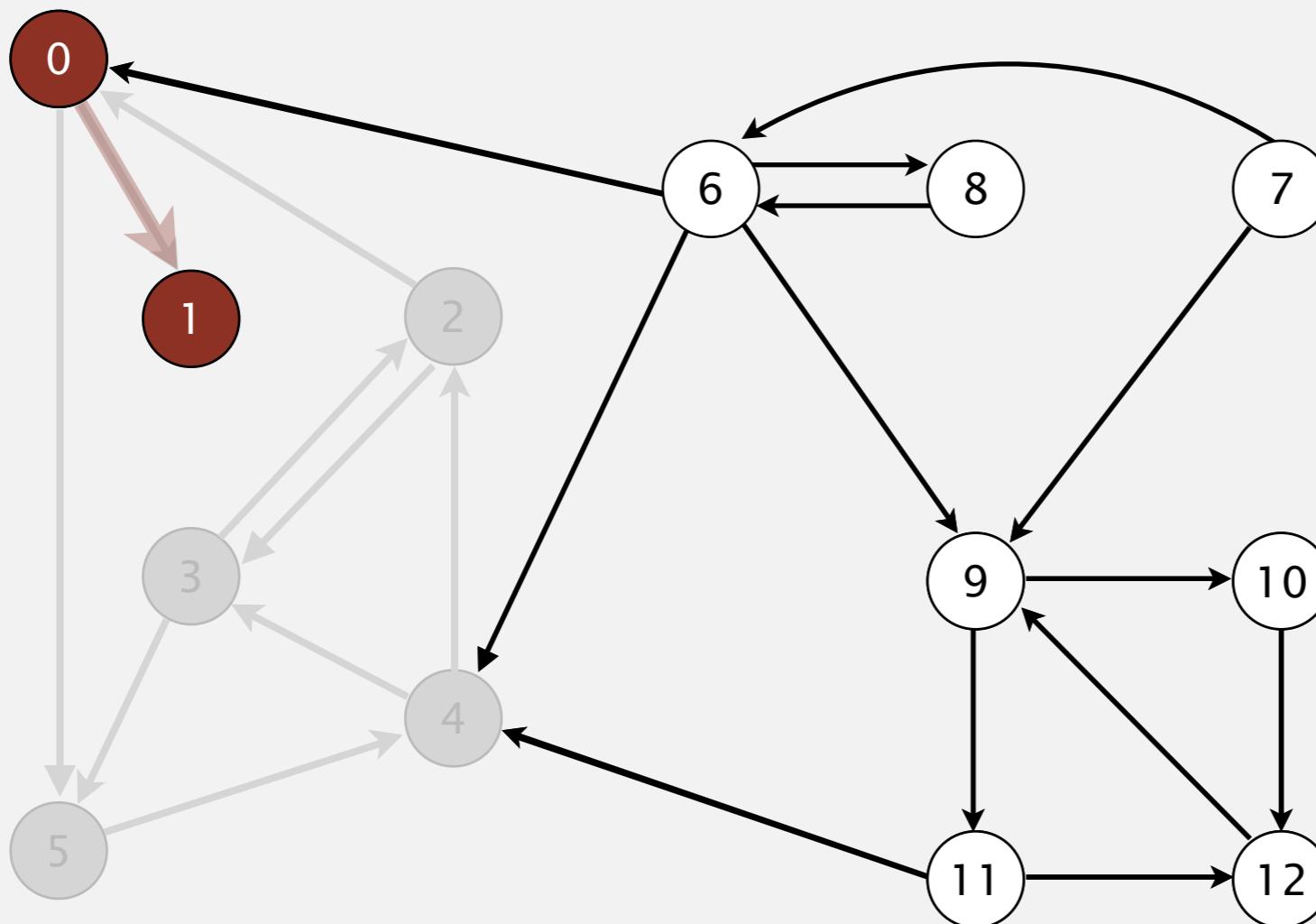


$v$	marked[]	edgeTo[]
0	T	-
1	T	0
2	T	3
3	T	4
4	T	5
5	T	0
6	F	-
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

# Directed depth-first search demo

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
  - Recursively visit all unmarked vertices pointing from  $v$ .



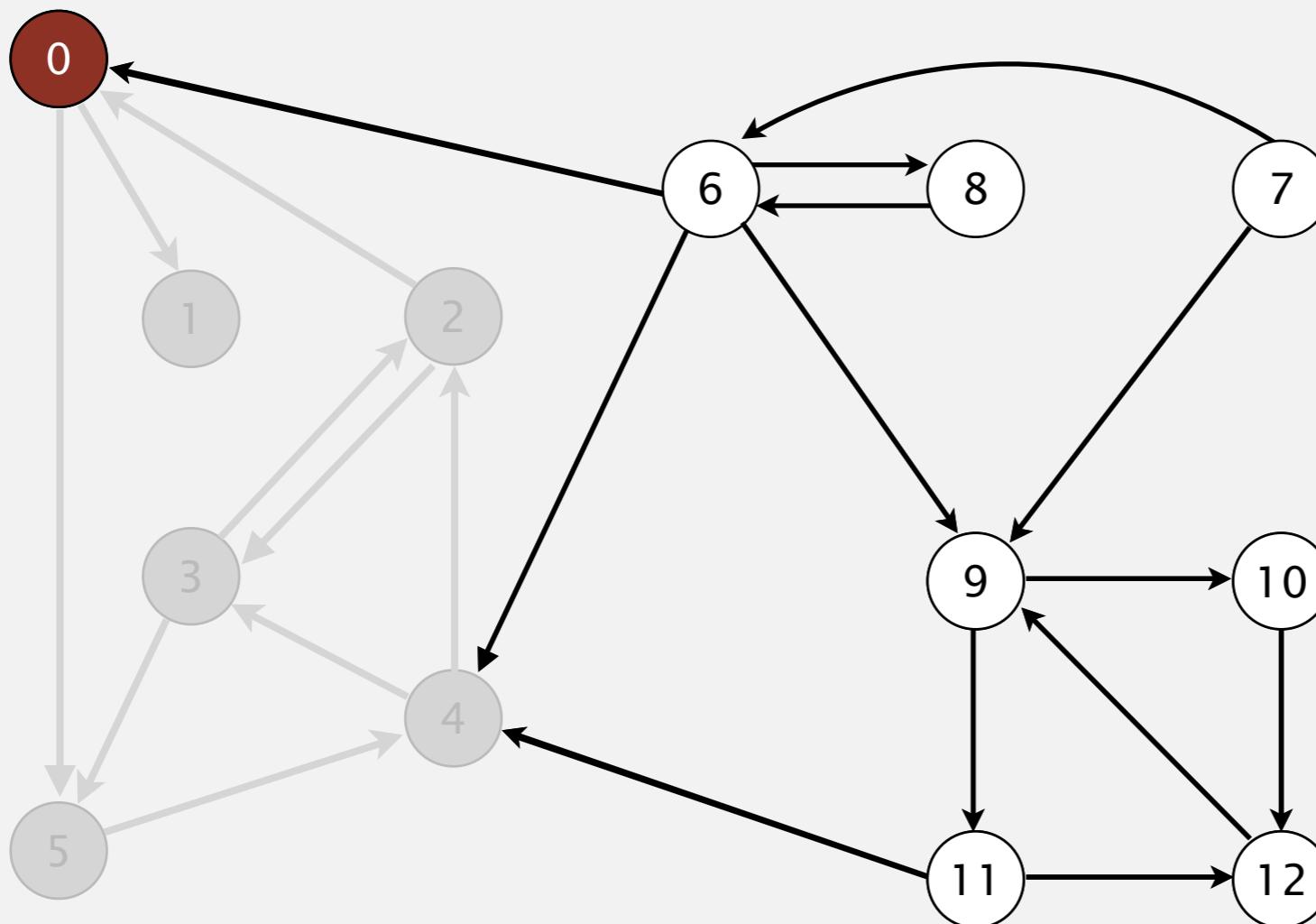
**done 1**

v	marked[]	edgeTo[]
0	T	-
1	T	0
2	T	3
3	T	4
4	T	5
5	T	0
6	F	-
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

# Directed depth-first search demo

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices pointing from  $v$ .

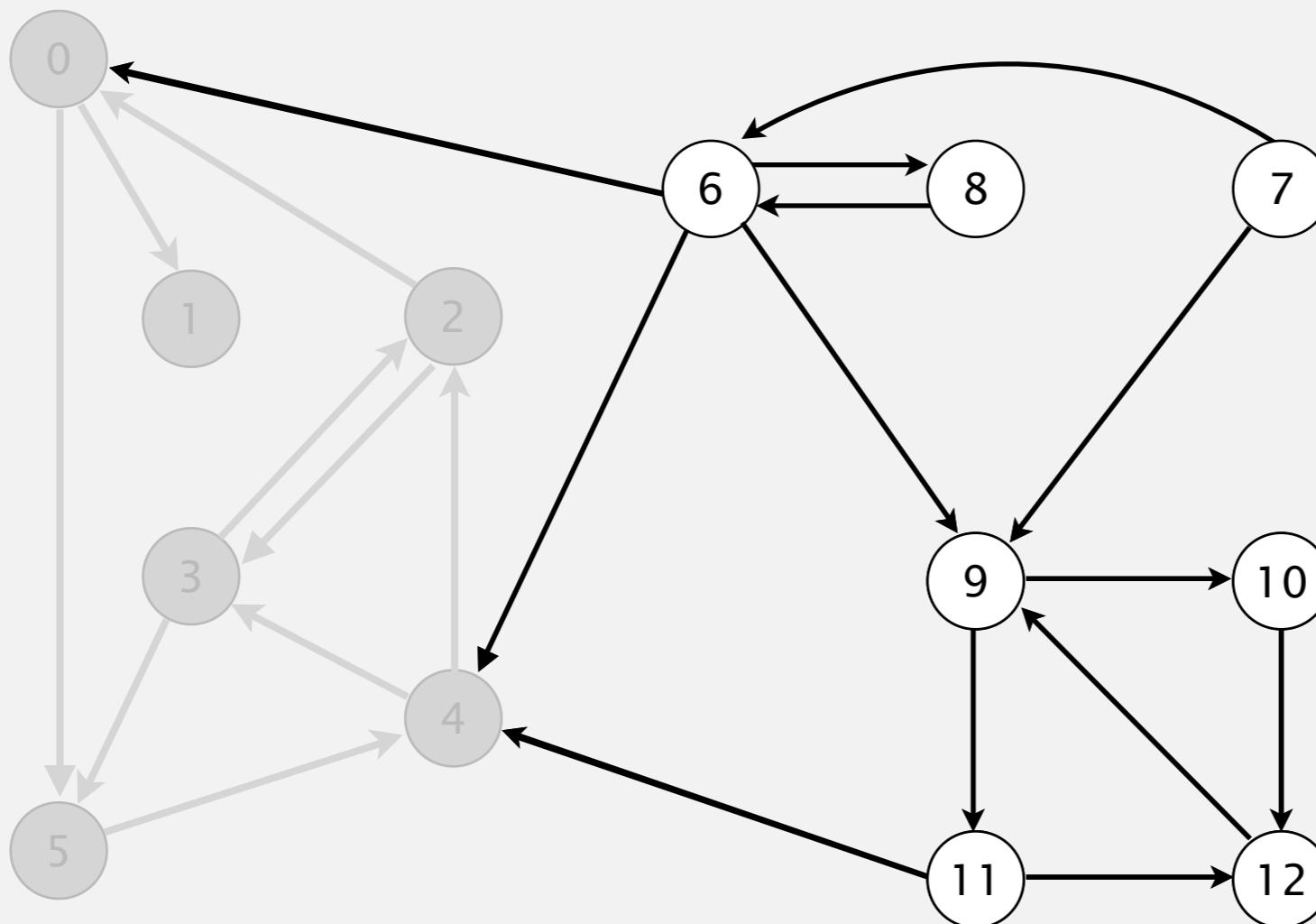


$v$	marked[]	edgeTo[]
0	T	-
1	T	0
2	T	3
3	T	4
4	T	5
5	T	0
6	F	-
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

# Directed depth-first search demo

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices pointing from  $v$ .



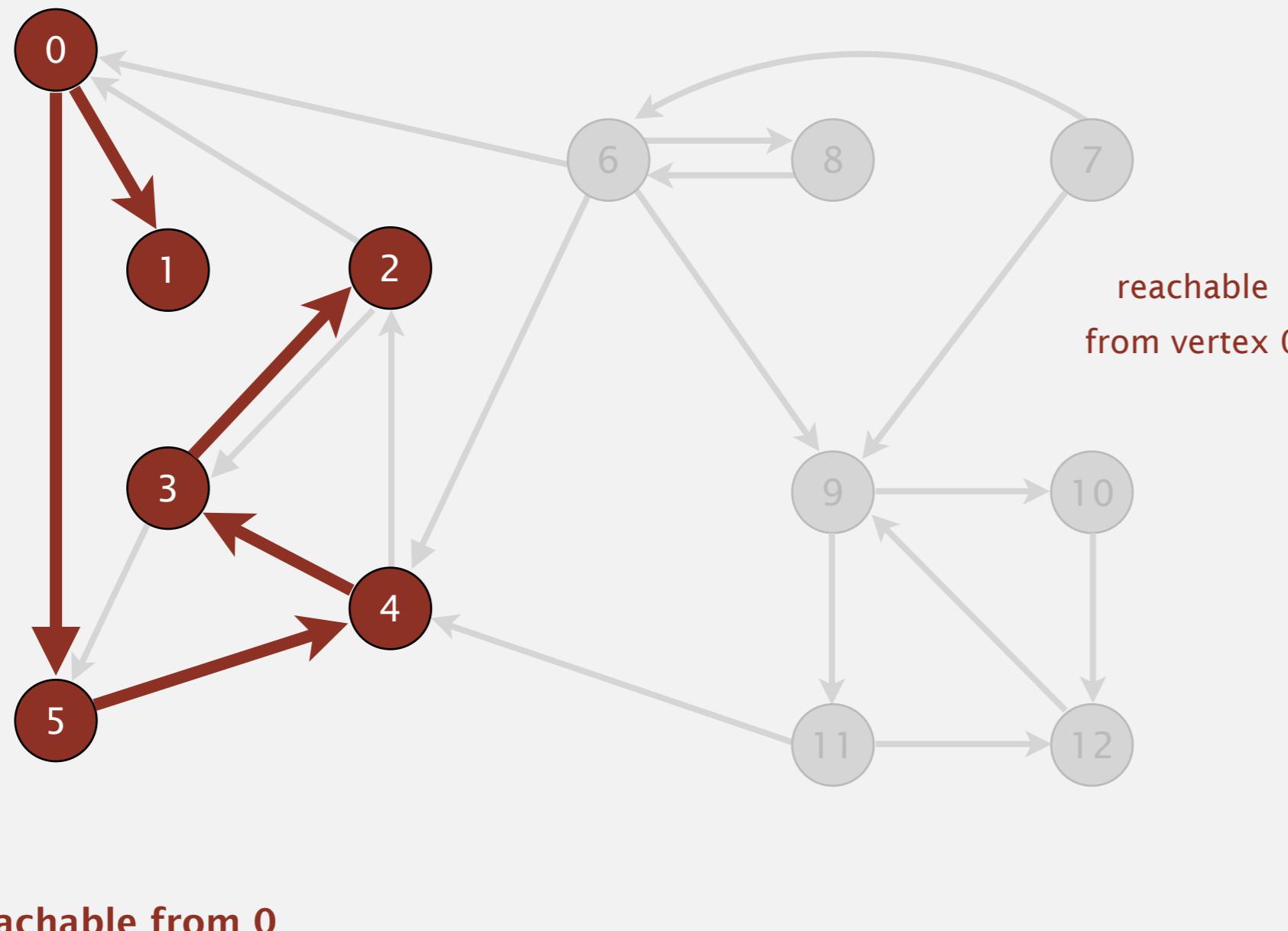
done

$v$	marked[]	edgeTo[]
0	T	-
1	T	0
2	T	3
3	T	4
4	T	5
5	T	0
6	F	-
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

# Directed depth-first search demo

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices pointing from  $v$ .

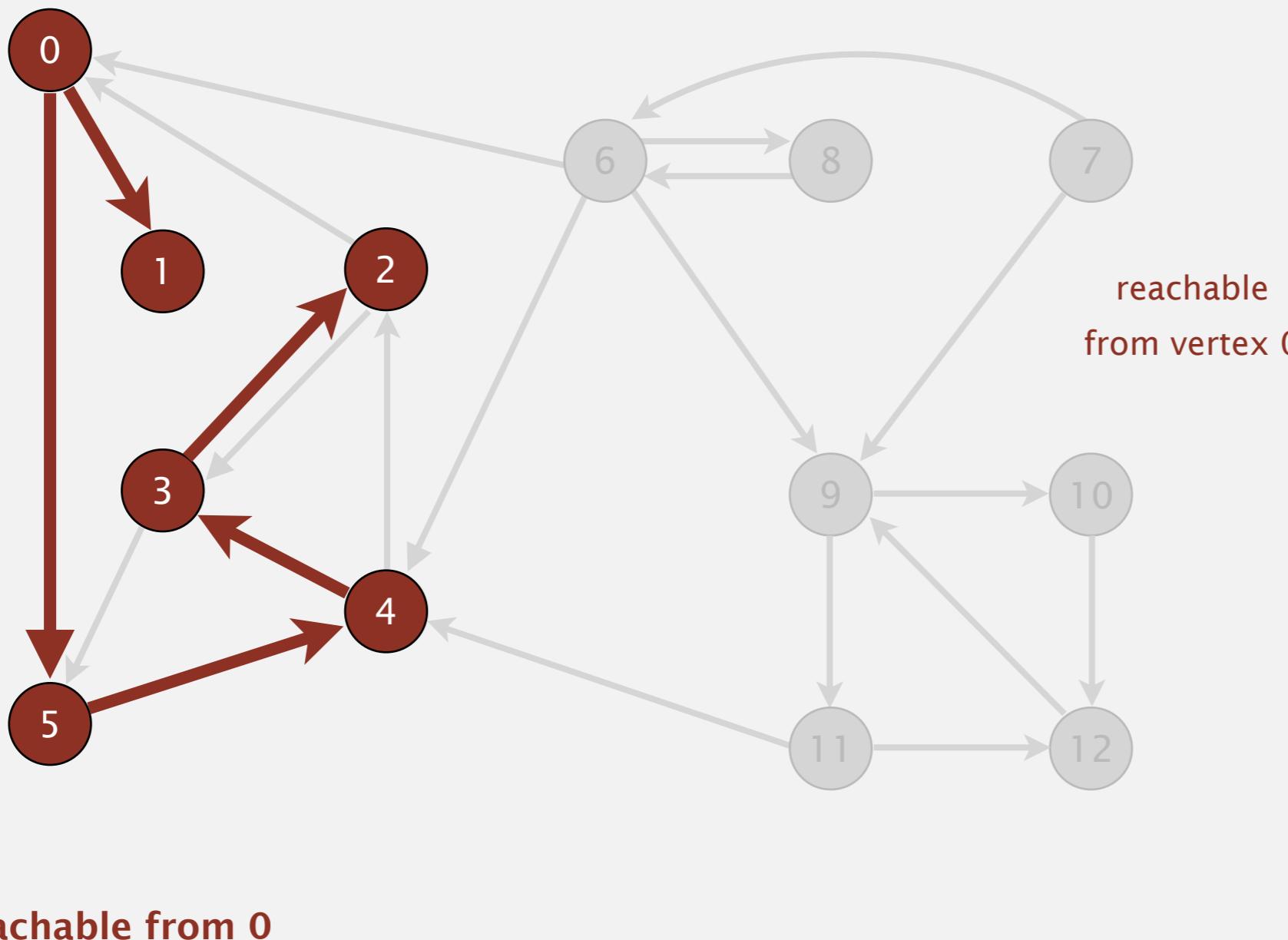


$v$	marked[]	edgeTo[]
0	T	-
1	T	0
2	T	3
3	T	4
4	T	5
5	T	0
6	F	-
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

# Depth-first search demo

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices pointing from  $v$ .



$v$	marked[]	edgeTo[]
0	T	-
1	T	0
2	T	3
3	T	4
4	T	5
5	T	0
6	F	-
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

# Depth-first search (in undirected graphs)

Recall code for undirected graphs.

```
public class DepthFirstSearch
{
    private boolean[] marked; ← true if connected to s

    public DepthFirstSearch(Graph G, int s)
    {
        marked = new boolean[G.V()];
        dfs(G, s); ← constructor marks
        vertices connected to s
    }

    private void dfs(Graph G, int v) ← recursive DFS does the work
    {
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w]) dfs(G, w);
    }

    public boolean visited(int v) ← client can ask whether any
    {   return marked[v]; }   vertex is connected to s
}
```

# Depth-first search (in directed graphs)

Code for **directed** graphs identical to undirected one.

[substitute Digraph for Graph]

```
public class DirectedDFS
{
    private boolean[] marked; ← true if path from s

    public DirectedDFS(Digraph G, int s)
    {
        marked = new boolean[G.V()]; ← constructor marks
        dfs(G, s);               vertices reachable from s
    }

    private void dfs(Digraph G, int v) ← recursive DFS does the work
    {
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w]) dfs(G, w);
    }

    public boolean visited(int v) ← client can ask whether any
    {   return marked[v]; }           vertex is reachable from s
}
```

# Reachability application: program control-flow analysis

Every program is a digraph.

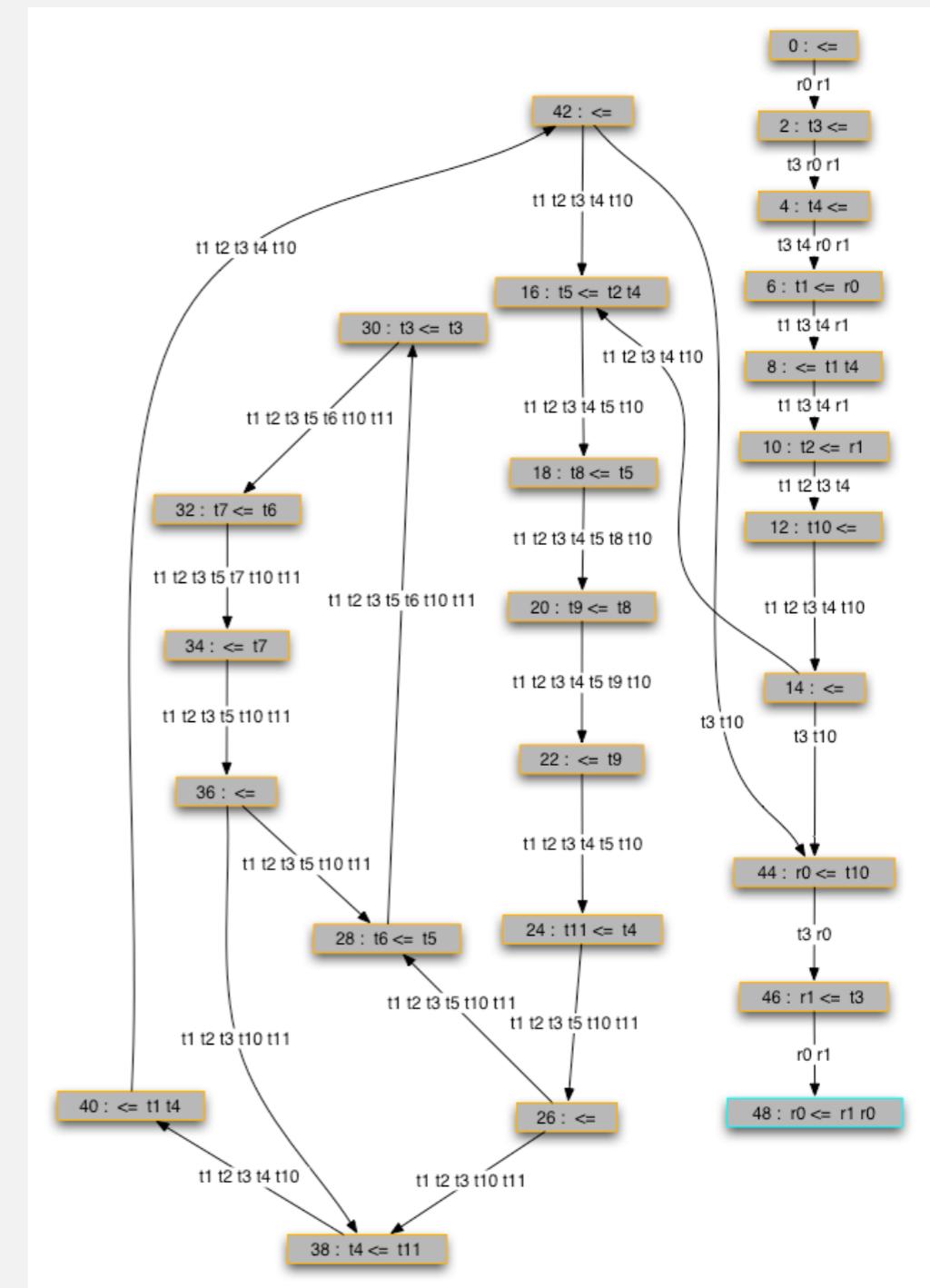
- Vertex = basic block of instructions (straight-line program).
- Edge = jump.

Dead-code elimination.

Find (and remove) unreachable code.

Infinite-loop detection.

Determine whether exit is unreachable.



# Reachability application: mark-sweep garbage collector

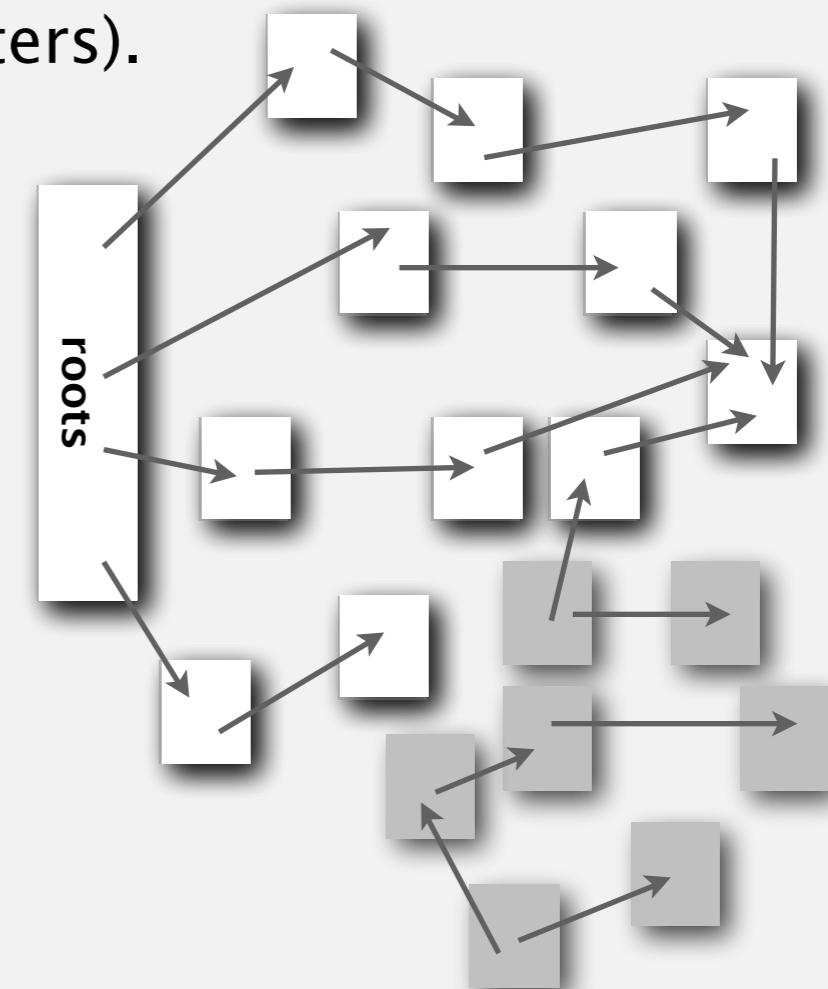
---

Every data structure is a digraph.

- Vertex = object.
- Edge = reference.

**Roots.** Objects known to be directly accessible by program (e.g., stack).

**Reachable objects.** Objects indirectly accessible by program  
(starting at a root and following a chain of pointers).

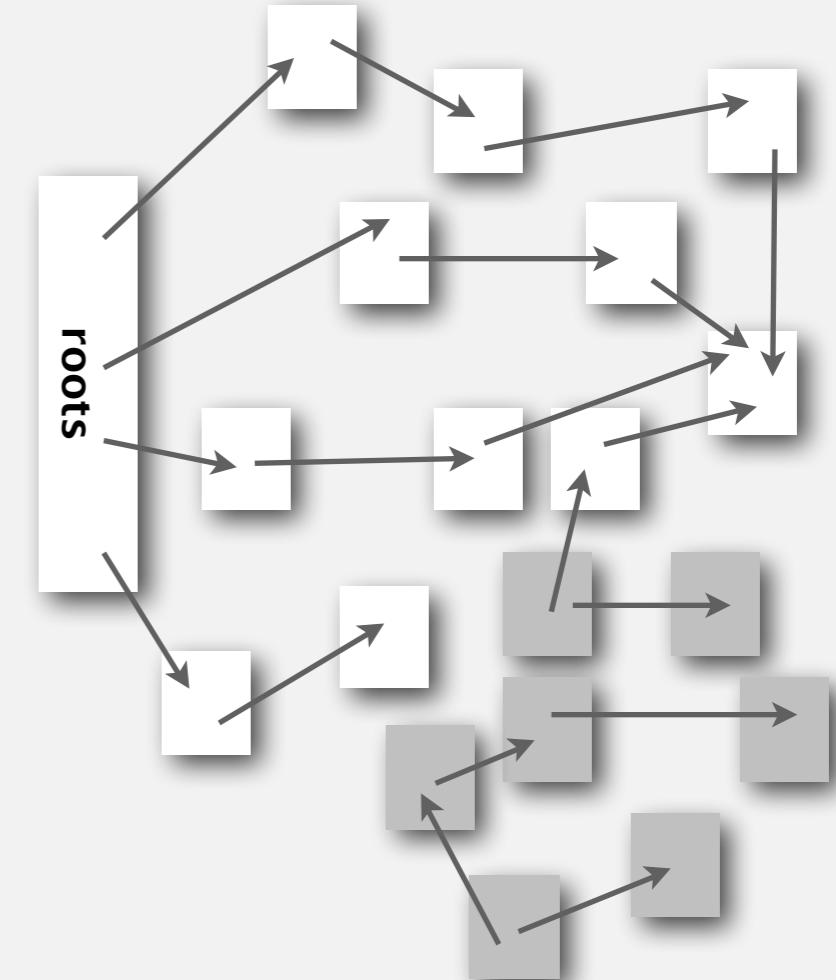


# Reachability application: mark-sweep garbage collector

Mark-sweep algorithm. [McCarthy, 1960]

- Mark: mark all reachable objects.
- Sweep: if object is unmarked, it is garbage (so add to free list).

Memory cost. Uses 1 extra mark bit per object (plus DFS stack).



# Depth-first search in digraphs summary

---

DFS enables direct solution of simple digraph problems.

- ✓ • Reachability.
- Path finding.
- Topological sort.
- Directed cycle detection.

Basis for solving difficult digraph problems.

- 2-satisfiability.
- Directed Euler path.
- Strongly-connected components.

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 undirected graph are presented. The space and time requirements of both algorithms are bounded by  $k_1 V + k_2 E + k_3$  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.

# Breadth-first search in digraphs

---

Same method as for undirected graphs.

- Every undirected graph is a digraph (with edges in both directions).
- BFS is a **digraph** algorithm.

## **BFS (from source vertex $s$ )**

---

**Put  $s$  onto a FIFO queue, and mark  $s$  as visited.**

**Repeat until the queue is empty:**

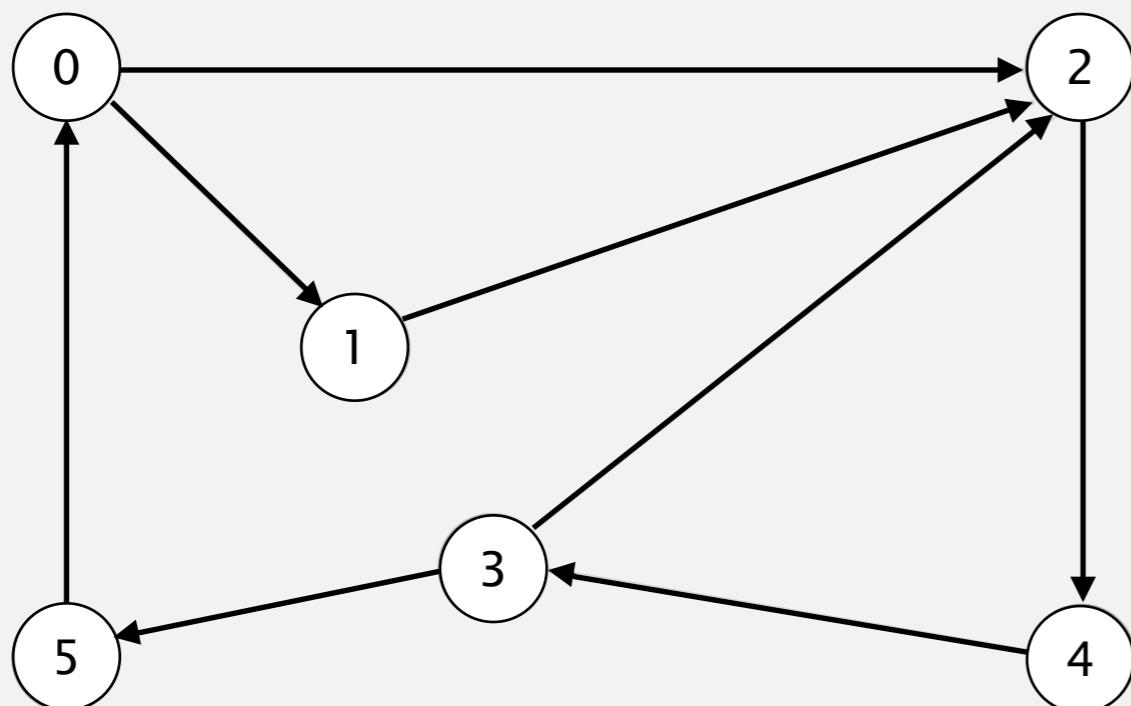
- **remove the least recently added vertex  $v$**
  - **for each unmarked vertex pointing from  $v$ :**
    - add to queue and mark as visited.**
- 

**Proposition.** BFS computes shortest paths (fewest number of edges) from  $s$  to all other vertices in a digraph in time proportional to  $E + V$ .

# Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices pointing from  $v$  and mark them.



**tinyDG2.txt**

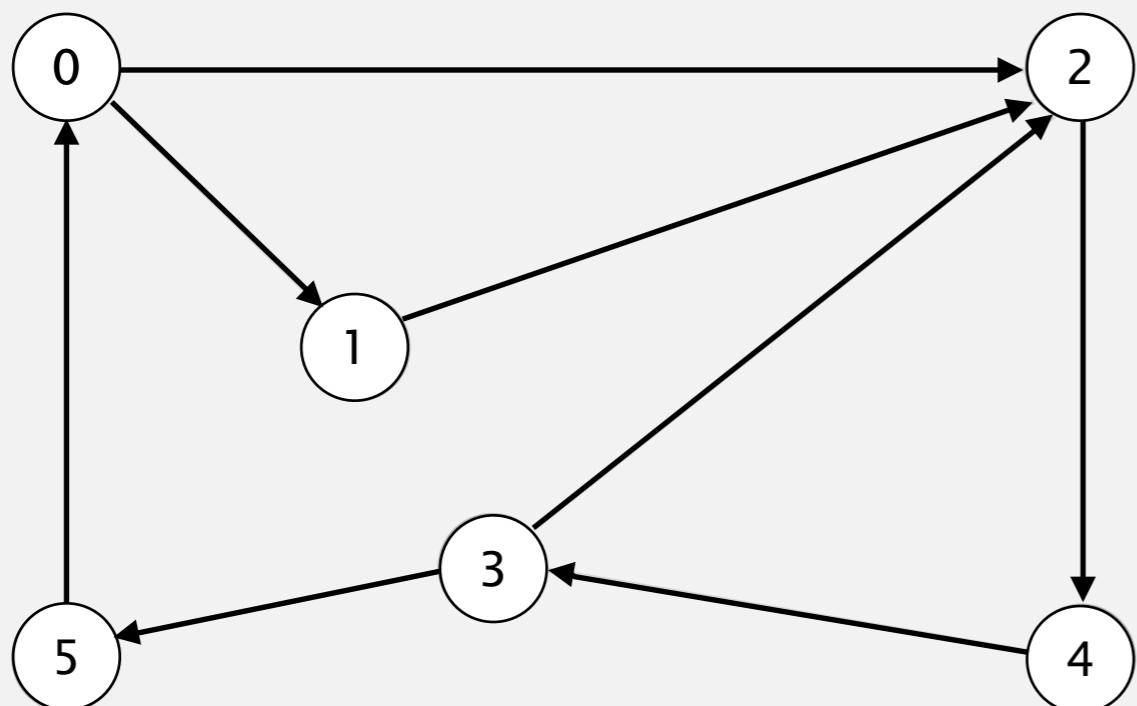
V → 6  
E → 8  
5 0  
2 4  
3 2  
1 2  
0 1  
4 3  
3 5  
0 2

**graph G**

# Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices pointing from  $v$  and mark them.



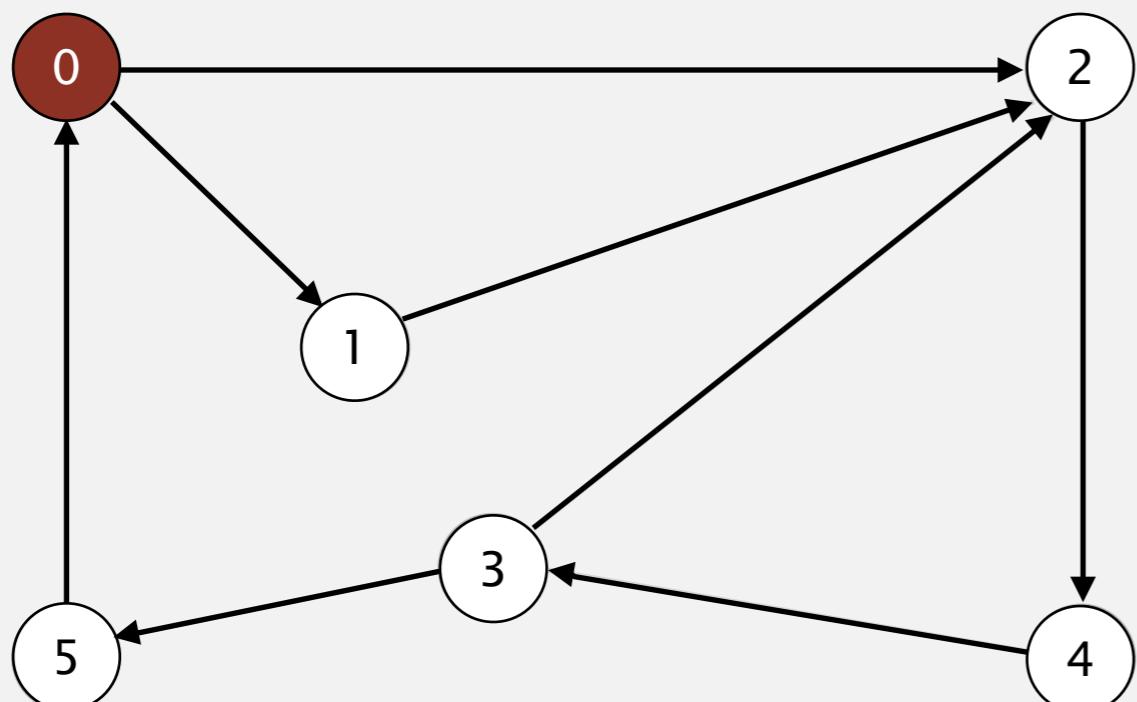
queue	v	edgeTo[]	distTo[]
	0	-	0
	1	-	-
	2	-	-
	3	-	-
	4	-	-
	5	-	-

**add 0 to queue**

# Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices pointing from  $v$  and mark them.



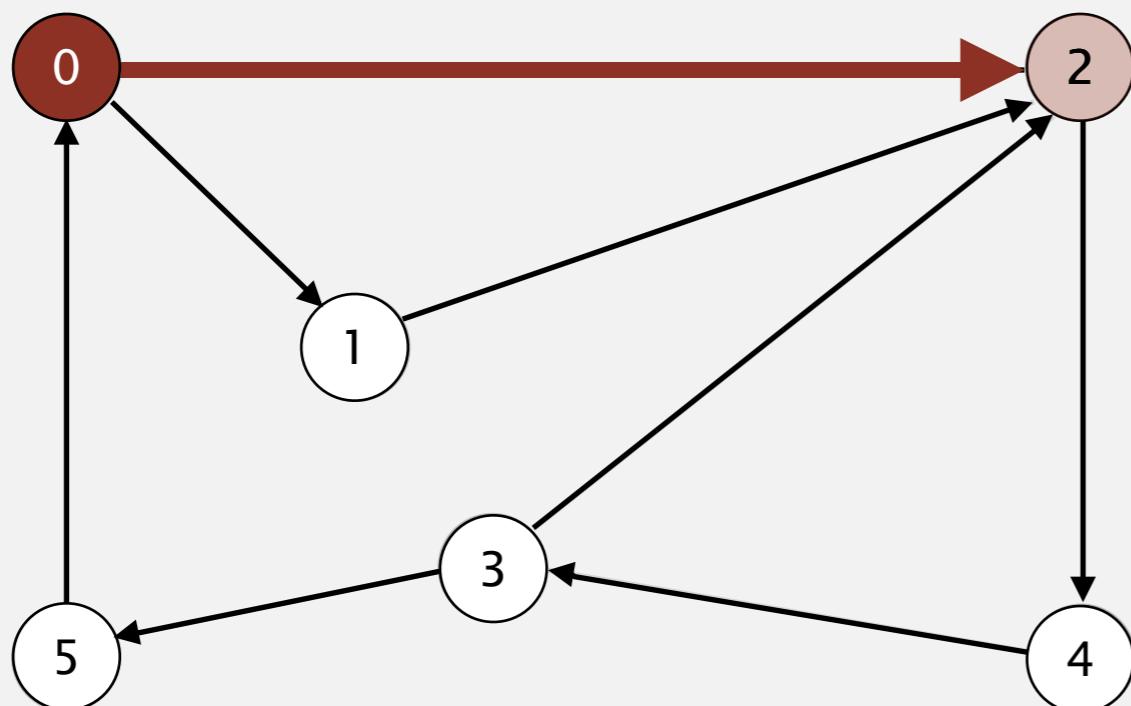
**dequeue 0**

queue	v	edgeTo[]	distTo[]
	0	-	0
	1	-	-
	2	-	-
	3	-	-
	4	-	-
	5	-	-
0			

# Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices pointing from  $v$  and mark them.



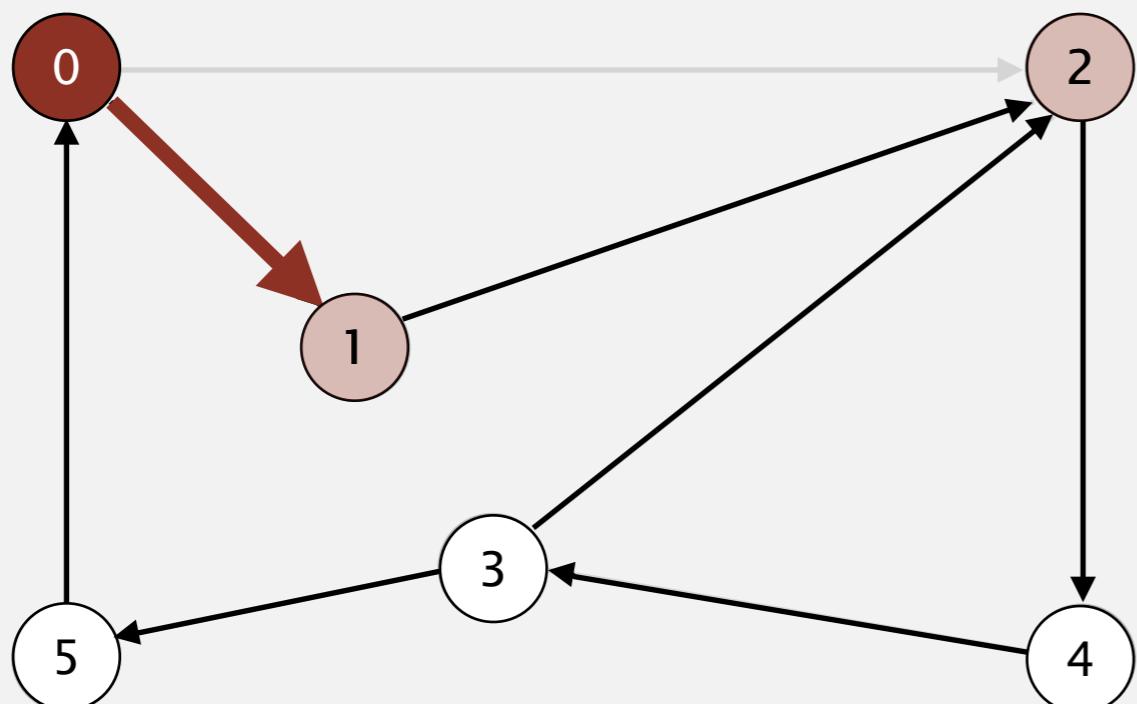
queue	v	edgeTo[]	distTo[]
	0	-	0
	1	-	-
	2	0	1
	3	-	-
	4	-	-
	5	-	-

**dequeue 0: check 2 and check 1**

# Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices pointing from  $v$  and mark them.



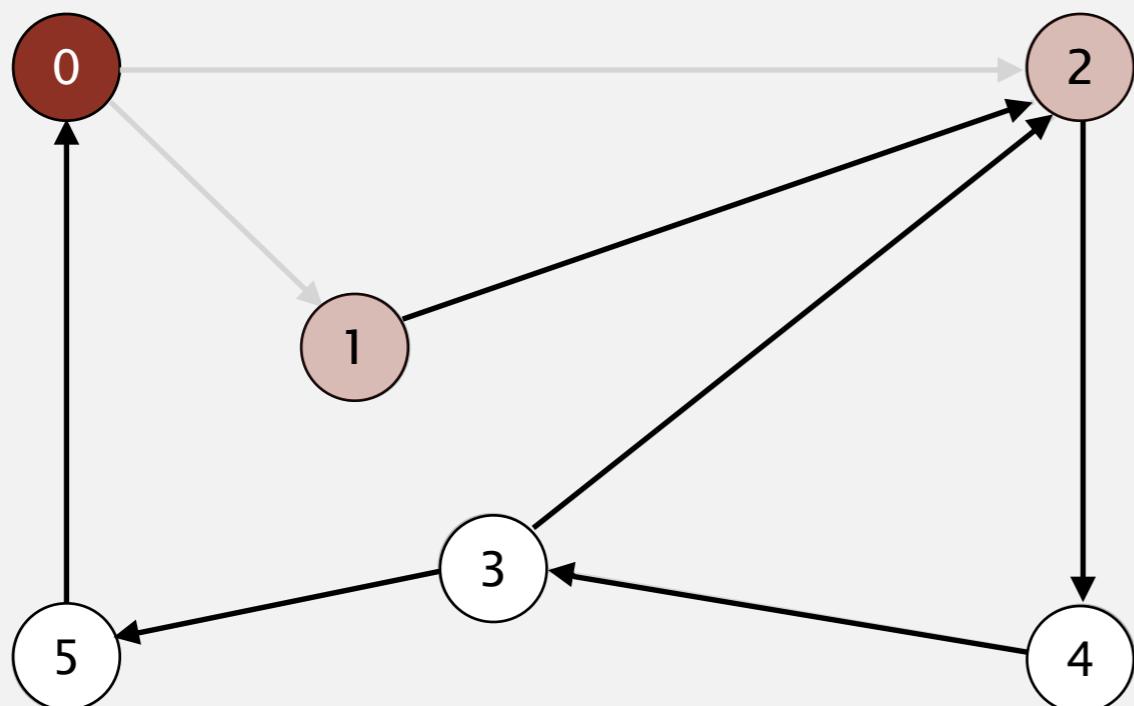
queue	v	edgeTo[]	distTo[]
	0	-	0
	1	0	1
	2	0	1
	3	-	-
	4	-	-
	5	-	-
2			

**dequeue 0:** check 2 and **check 1**

# Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices pointing from  $v$  and mark them.



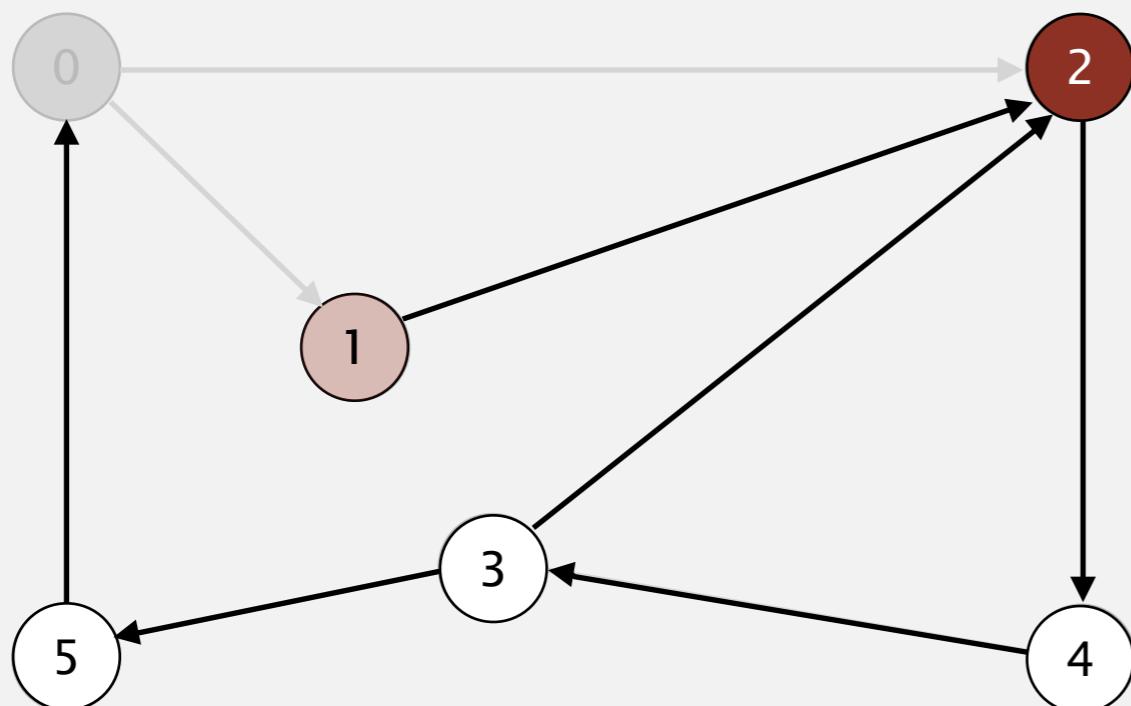
queue	v	edgeTo[]	distTo[]
	0	-	0
	1	0	1
	2	0	1
	3	-	-
	4	-	-
1	5	-	-
2			

0 done

# Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices pointing from  $v$  and mark them.



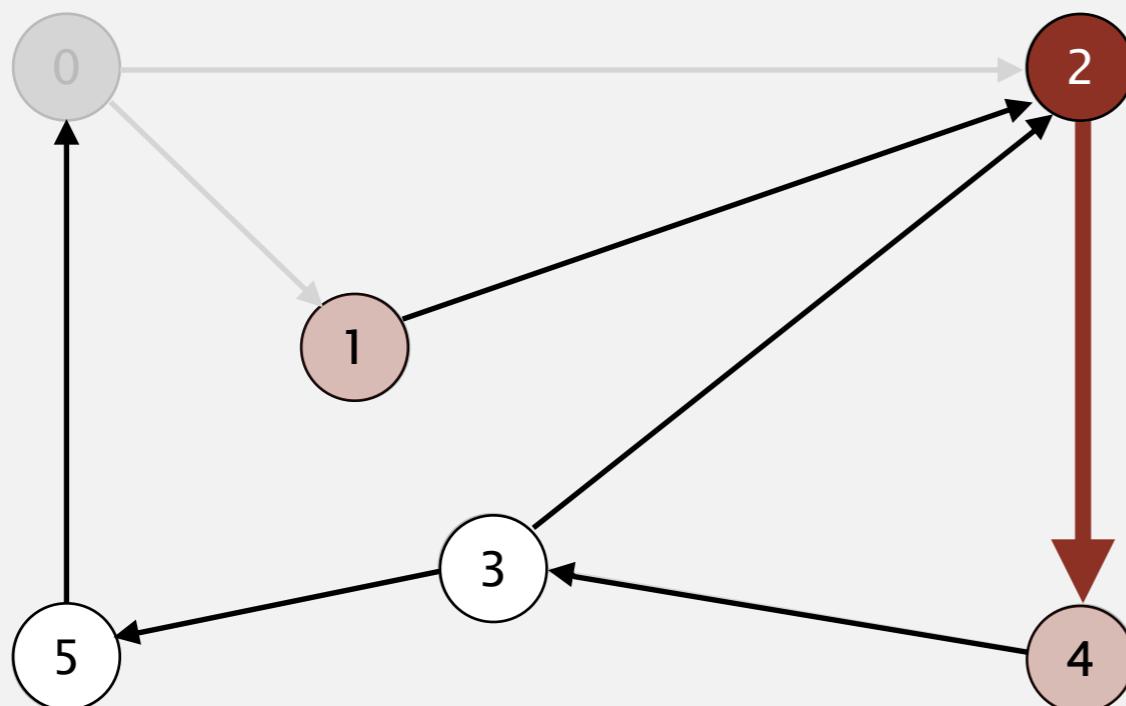
queue	v	edgeTo[]	distTo[]
	0	-	0
	1	0	1
	2	0	1
	3	-	-
	4	-	-
1	5	-	-
2			

**dequeue 2**

# Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices pointing from  $v$  and mark them.



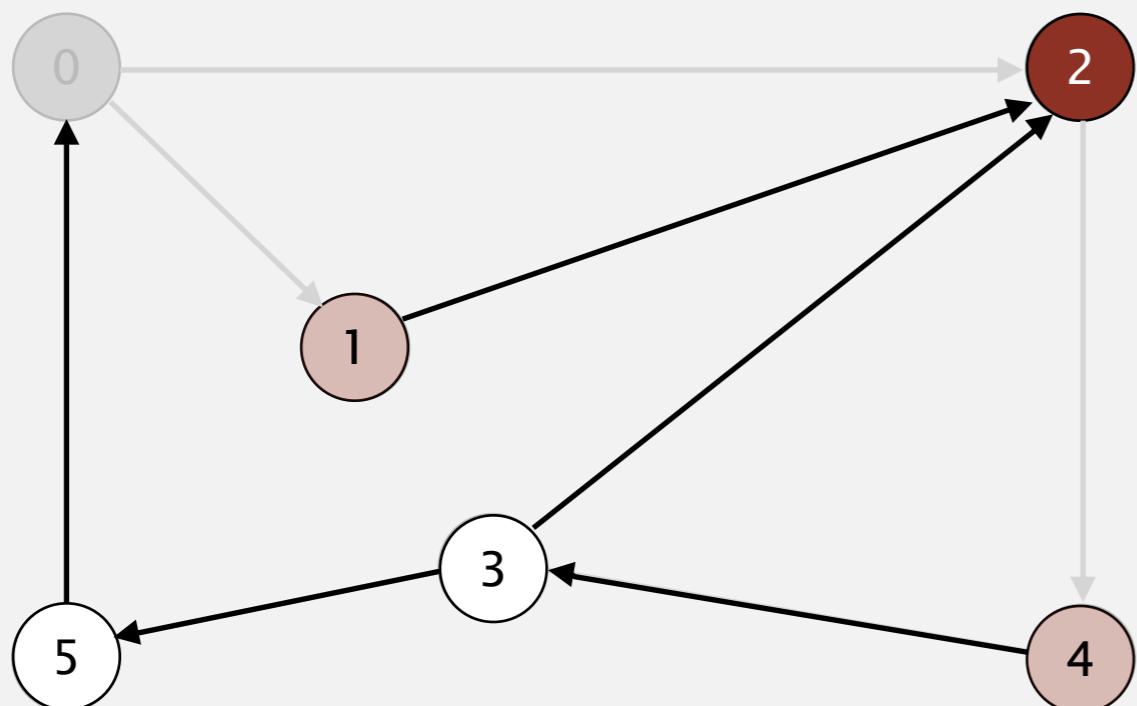
queue	v	edgeTo[]	distTo[]
	0	-	0
	1	0	1
	2	0	1
	3	2	2
	4	-	-
	5	-	-
1			

**dequeue 2: check 4**

# Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices pointing from  $v$  and mark them.



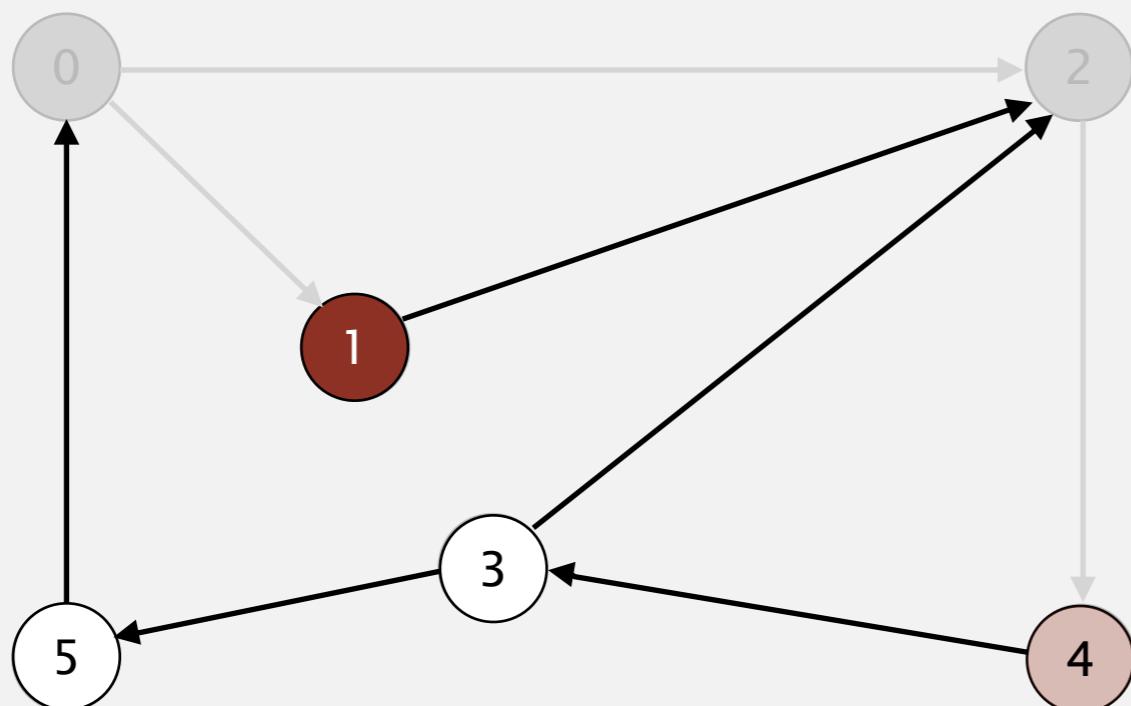
queue	v	edgeTo[]	distTo[]
	0	-	0
	1	0	1
	2	0	1
	3	-	-
	4	2	2
	5	-	-
4			
1			

2 done

# Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices pointing from  $v$  and mark them.



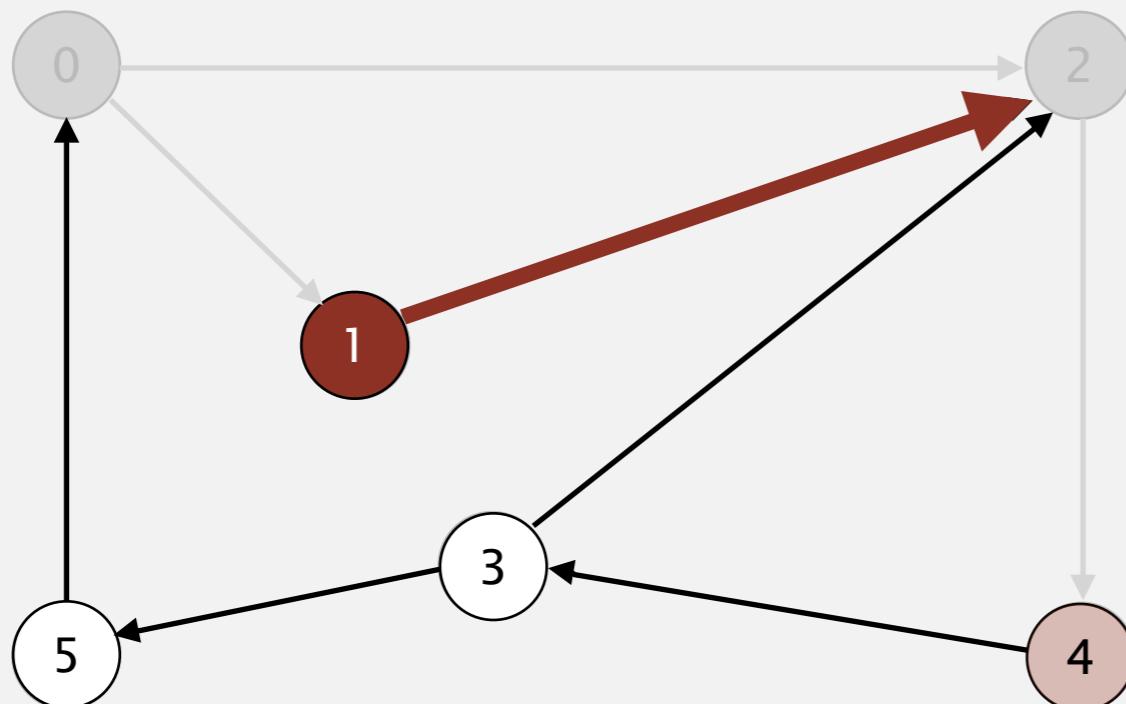
queue	v	edgeTo[]	distTo[]
	0	-	0
	1	0	1
	2	0	1
	3	-	-
	4	2	2
	5	-	-
4			
1			

**dequeue 1**

# Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices pointing from  $v$  and mark them.



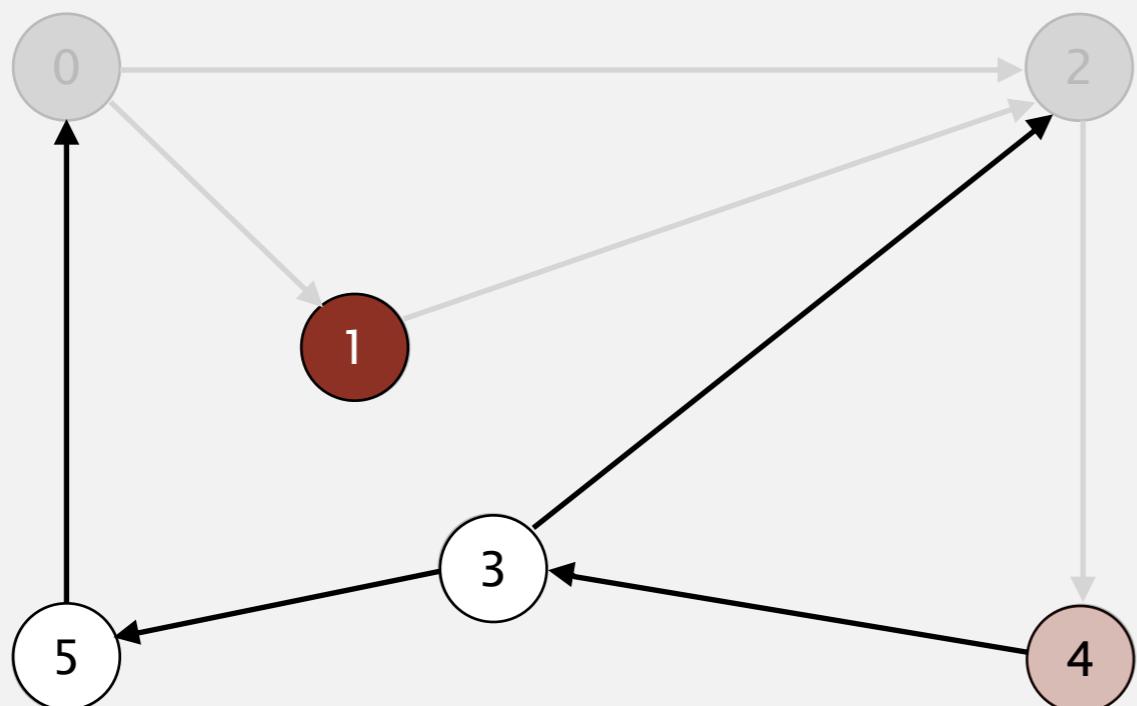
queue	v	edgeTo[]	distTo[]
	0	-	0
	1	0	1
	2	0	1
	3	-	-
	4	2	2
	5	-	-
4			

**dequeue 1; check 2**

# Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices pointing from  $v$  and mark them.



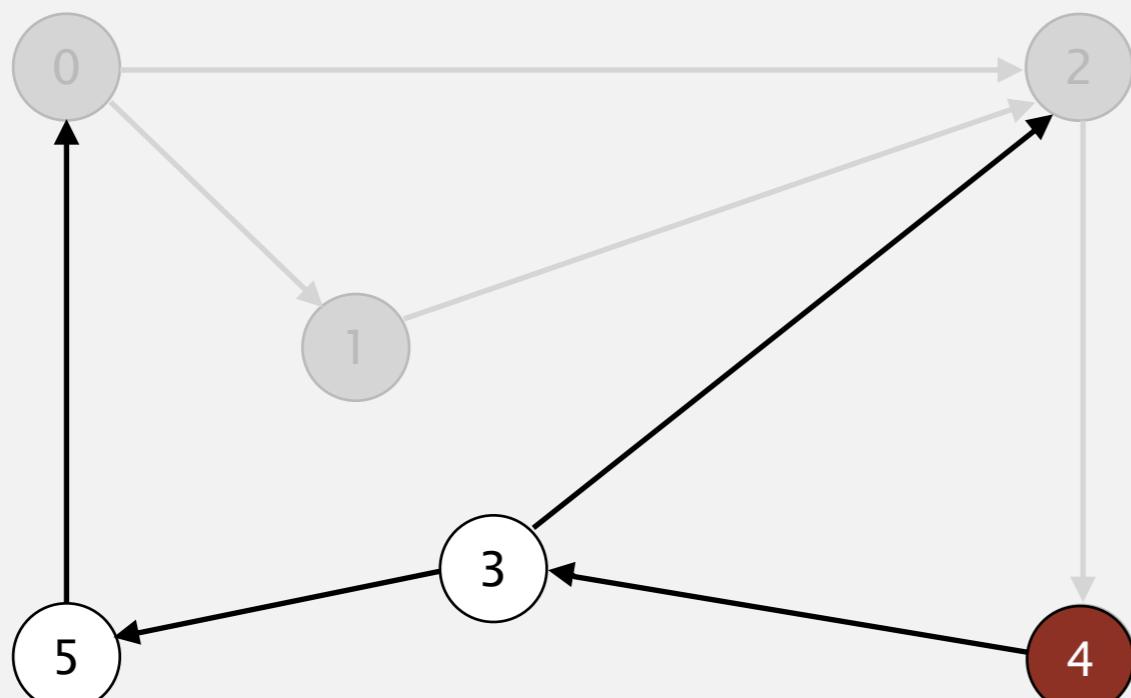
queue	v	edgeTo[]	distTo[]
	0	-	0
	1	0	1
	2	0	1
	3	-	-
	4	2	2
	5	-	-
4			

1 done

# Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices pointing from  $v$  and mark them.



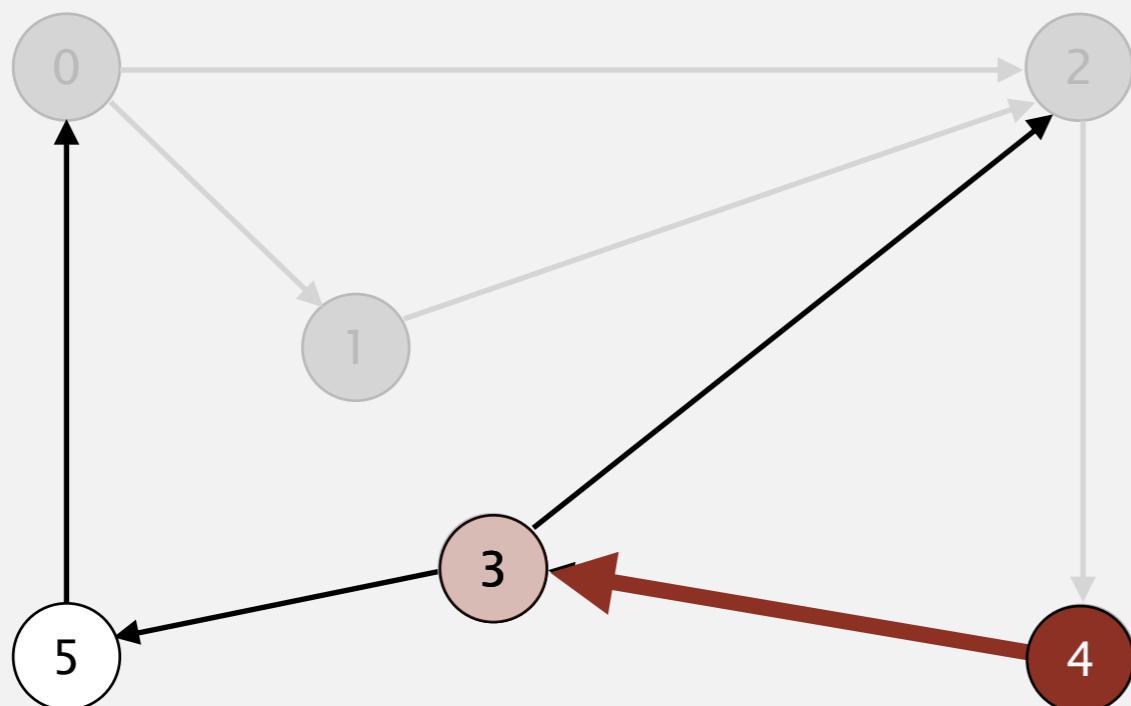
queue	v	edgeTo[]	distTo[]
	0	-	0
	1	0	1
	2	0	1
	3	-	-
	4	2	2
	5	-	-
4			

**dequeue 4**

# Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices pointing from  $v$  and mark them.



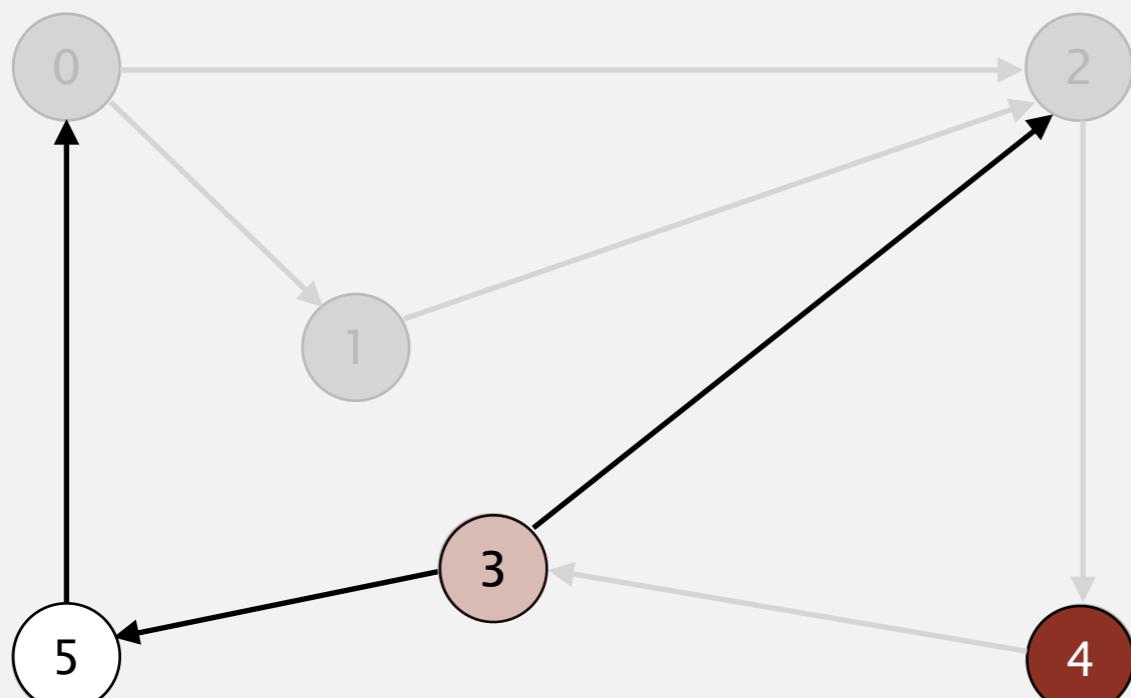
queue	v	edgeTo[]	distTo[]
	0	-	0
	1	0	1
	2	0	1
	3	4	3
	4	2	2
	5	-	-

dequeue 4: check 3

# Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices pointing from  $v$  and mark them.



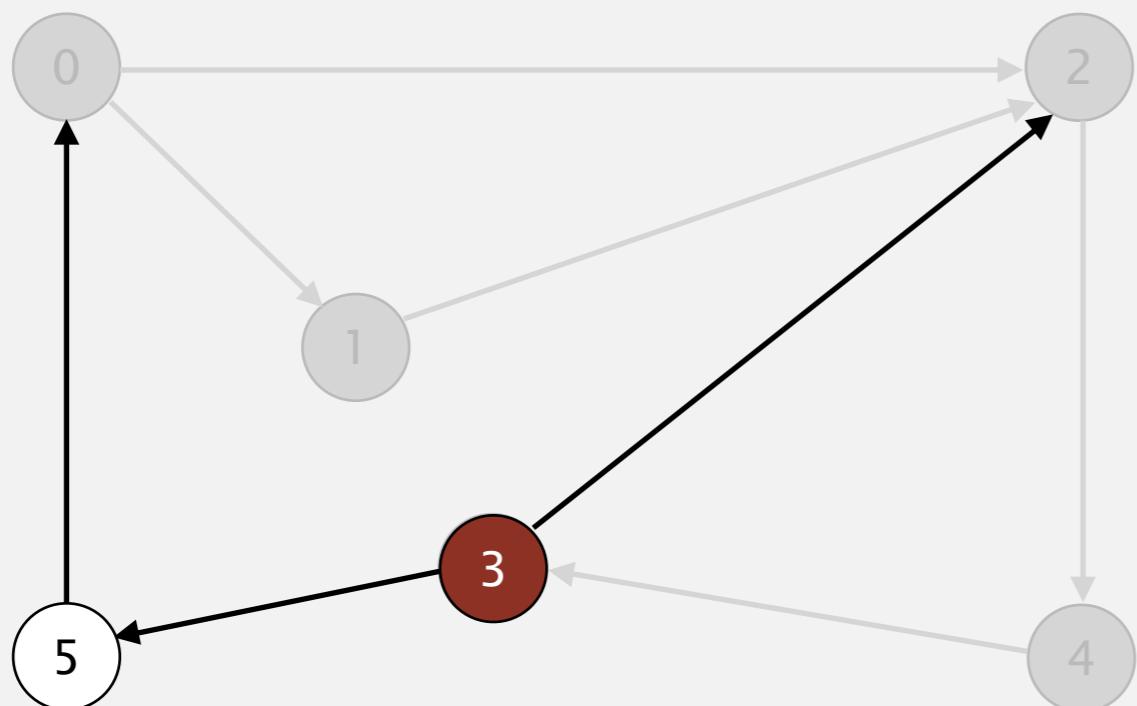
queue	v	edgeTo[]	distTo[]
	0	-	0
	1	0	1
	2	0	1
	3	4	3
	4	2	2
3	5	-	-

4 done

# Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices pointing from  $v$  and mark them.



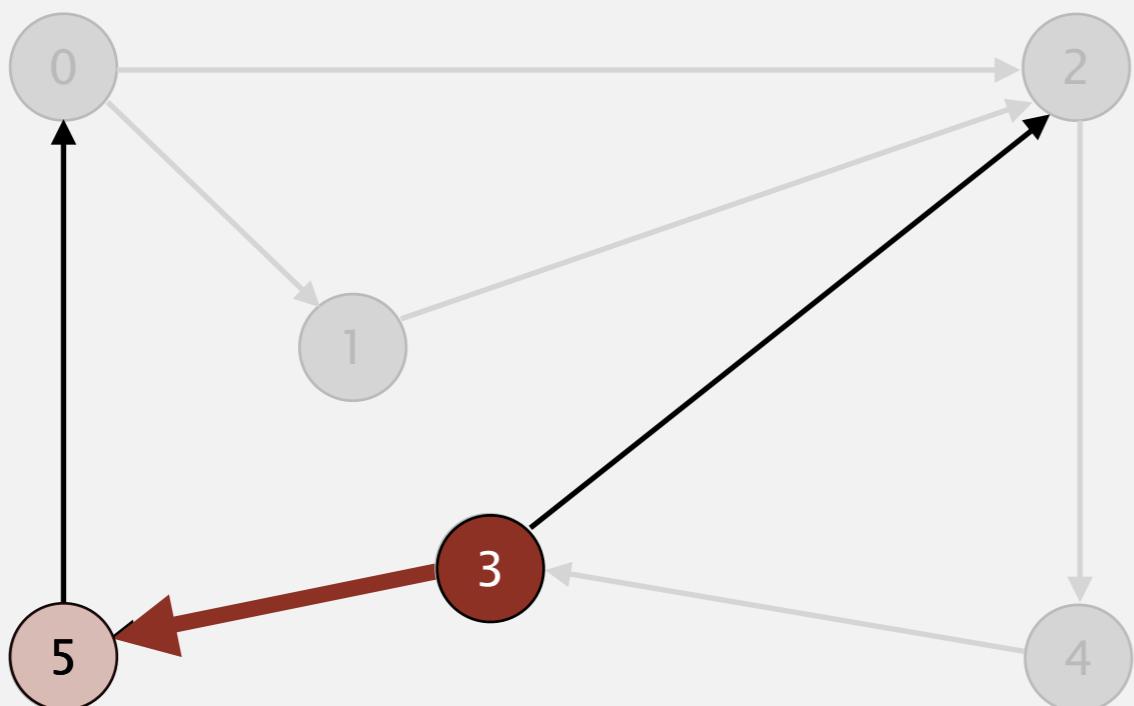
queue	v	edgeTo[]	distTo[]
	0	-	0
	1	0	1
	2	0	1
	3	4	3
	4	2	2
3	5	-	-

**dequeue 3**

# Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices pointing from  $v$  and mark them.



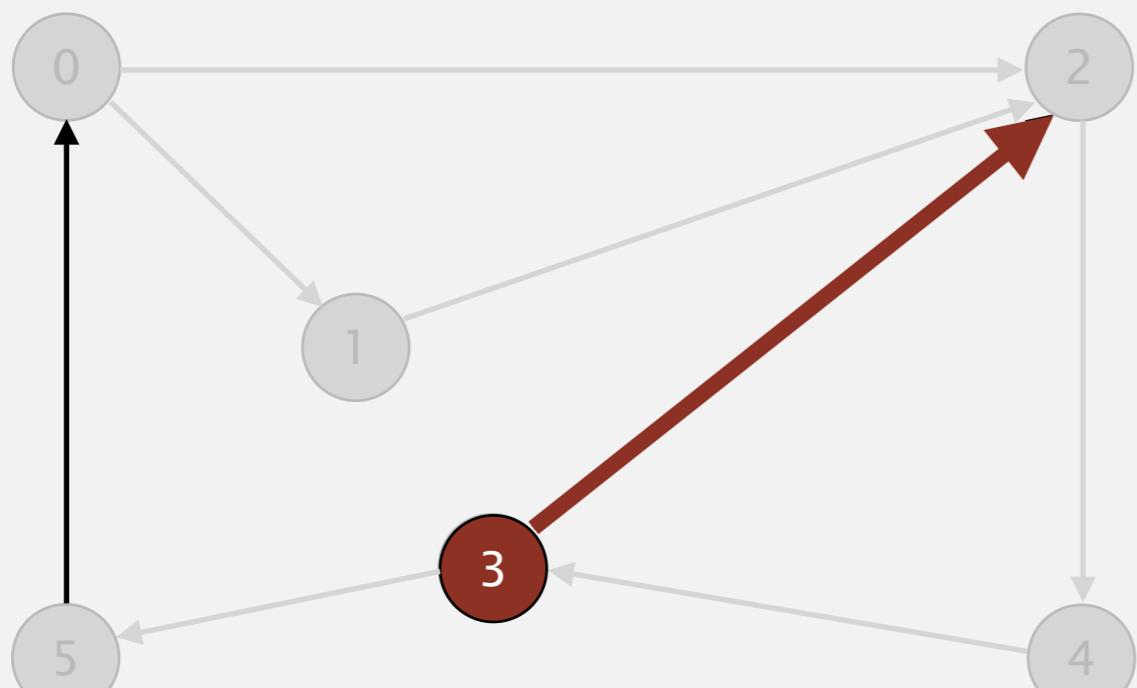
queue	v	edgeTo[]	distTo[]
	0	-	0
	1	0	1
	2	0	1
	3	4	3
	4	3	4
	5	-	-

**dequeue 3: check 5 and check 2**

# Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices pointing from  $v$  and mark them.



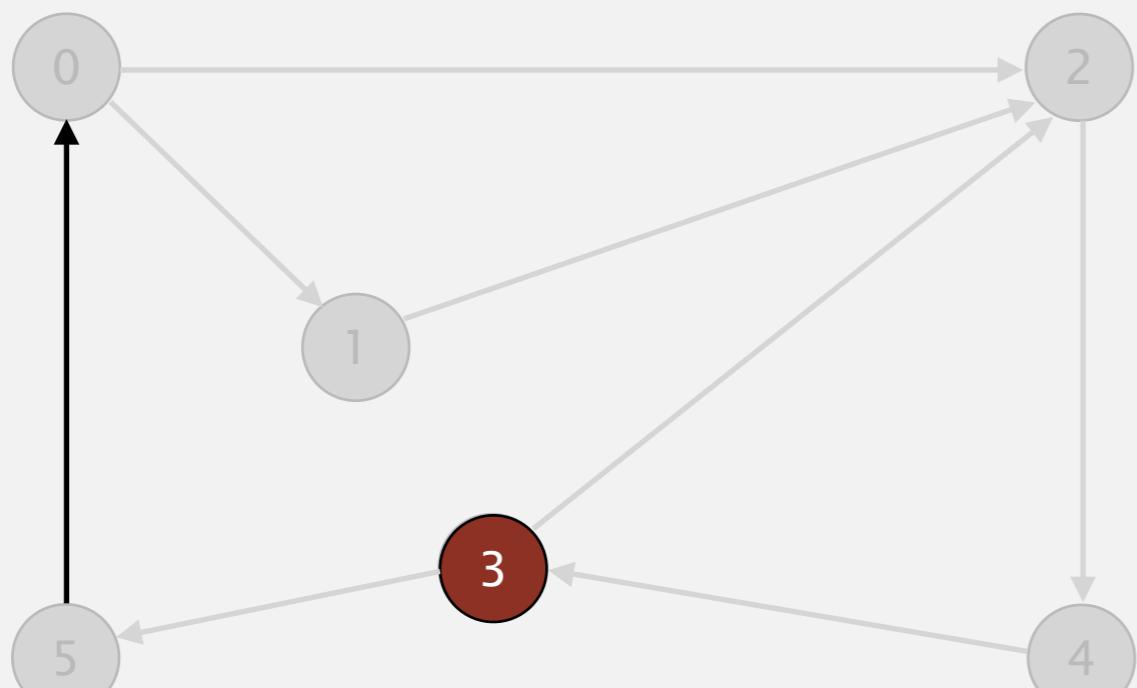
queue	v	edgeTo[]	distTo[]
	0	-	0
	1	0	1
	2	0	1
	3	4	3
	4	2	2
5	5	3	4

**dequeue 3: check 5 and check 2**

# Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices pointing from  $v$  and mark them.



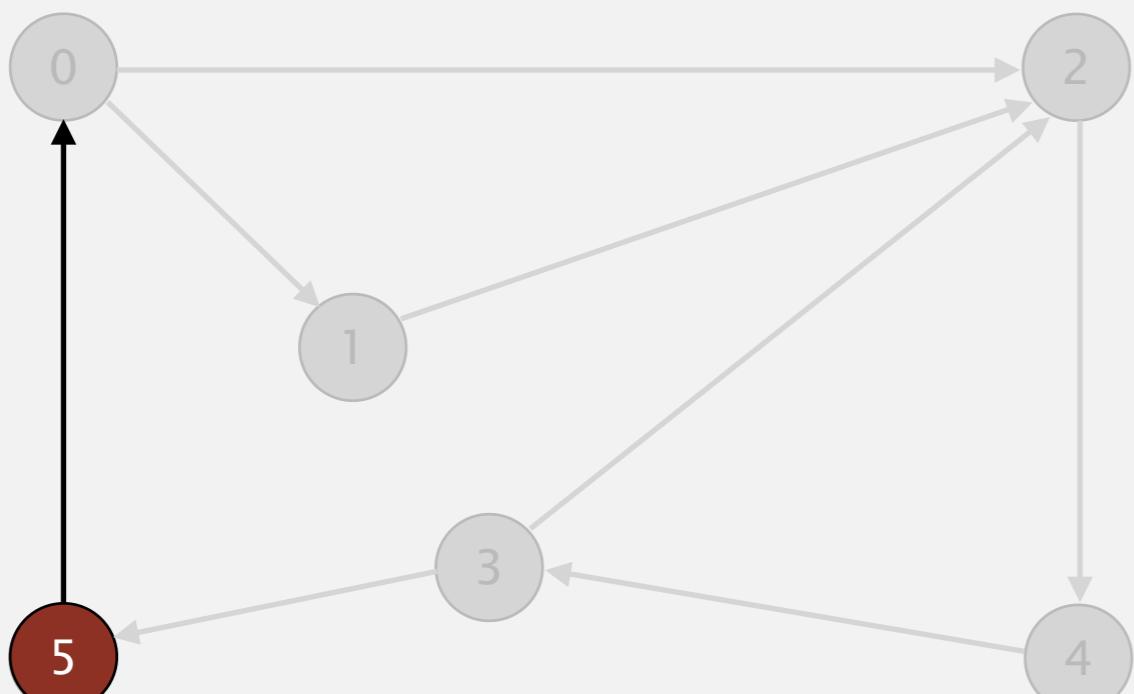
queue	v	edgeTo[]	distTo[]
	0	-	0
	1	0	1
	2	0	1
	3	4	3
	4	2	2
	5	3	4

3 done

# Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices pointing from  $v$  and mark them.



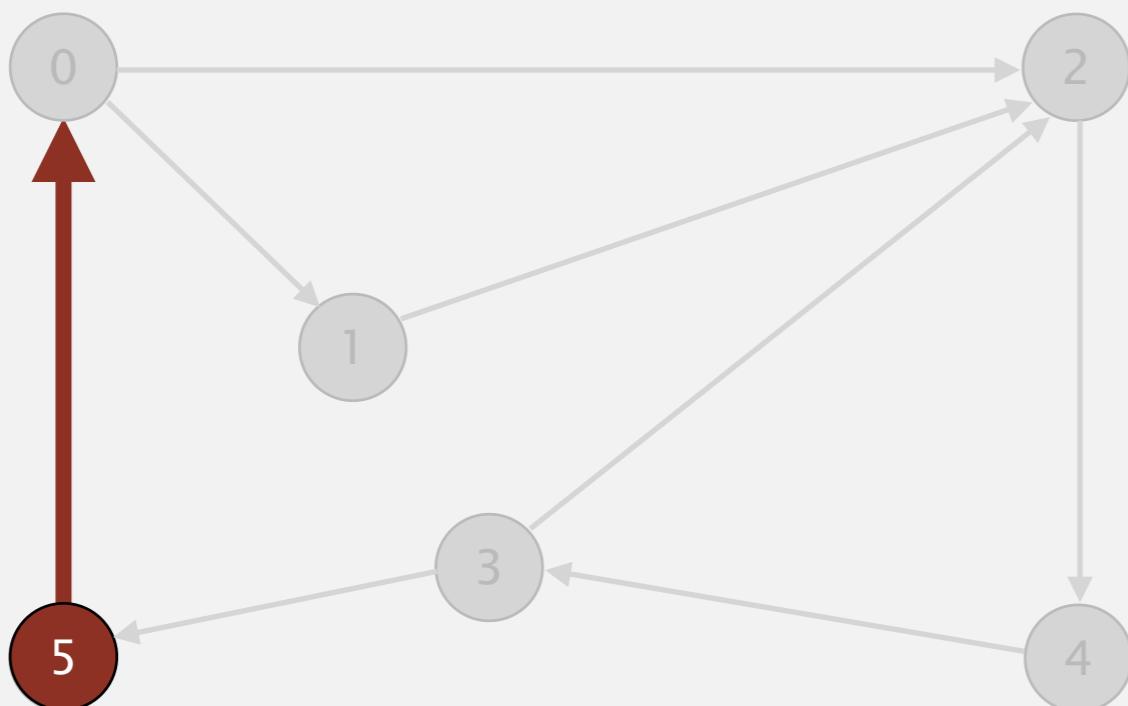
queue	v	edgeTo[]	distTo[]
	0	-	0
	1	0	1
	2	0	1
	3	4	3
	4	2	2
5	5	3	4

**dequeue 5**

# Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices pointing from  $v$  and mark them.



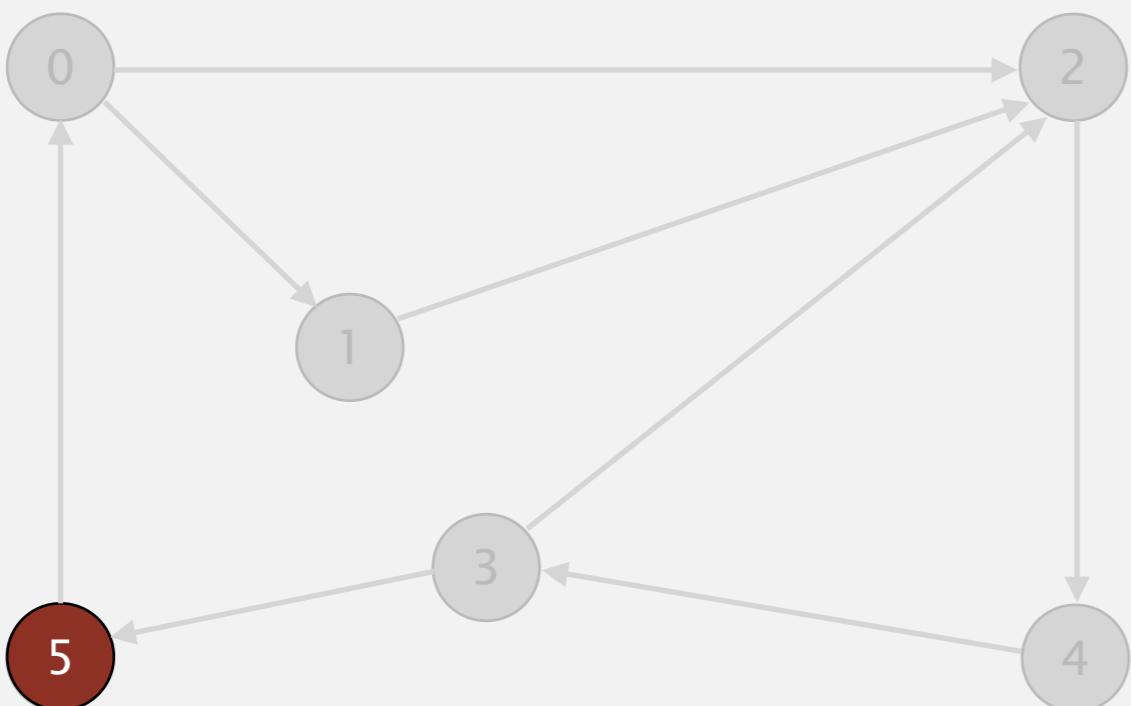
queue	v	edgeTo[]	distTo[]
	0	-	0
	1	0	1
	2	0	1
	3	4	3
	4	2	2
	5	3	4

dequeue 5: check 0

# Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices pointing from  $v$  and mark them.



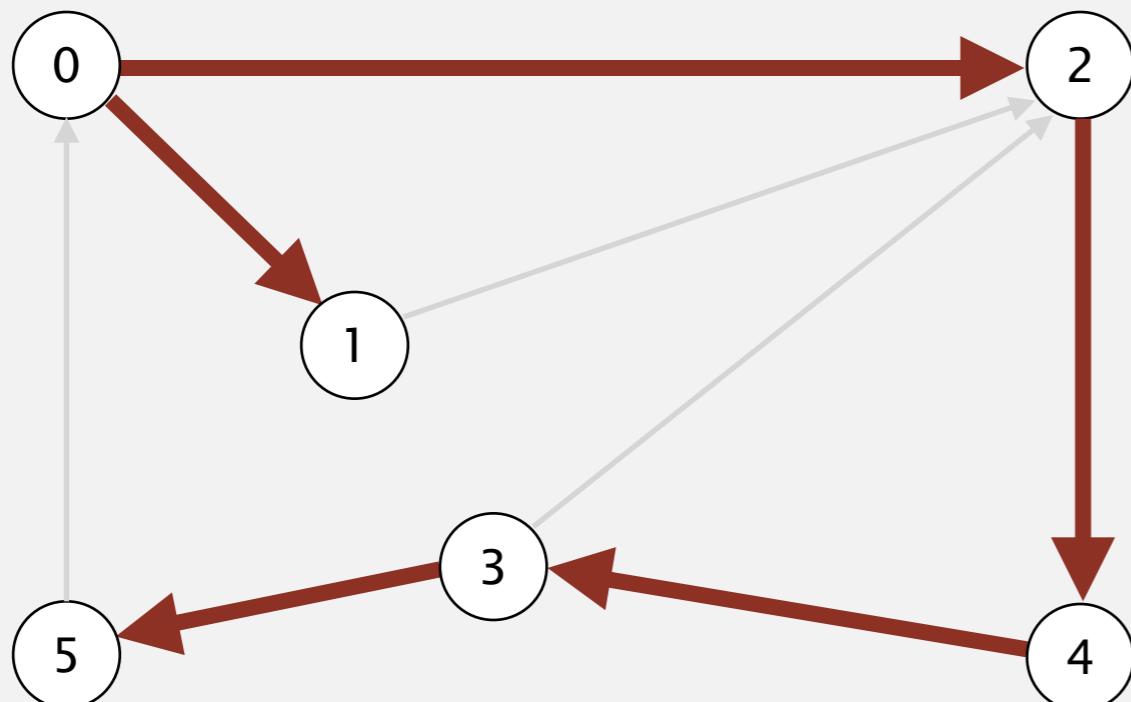
queue	v	edgeTo[]	distTo[]
	0	-	0
	1	0	1
	2	0	1
	3	4	3
	4	2	2
	5	3	4

5 done

# Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices pointing from  $v$  and mark them.



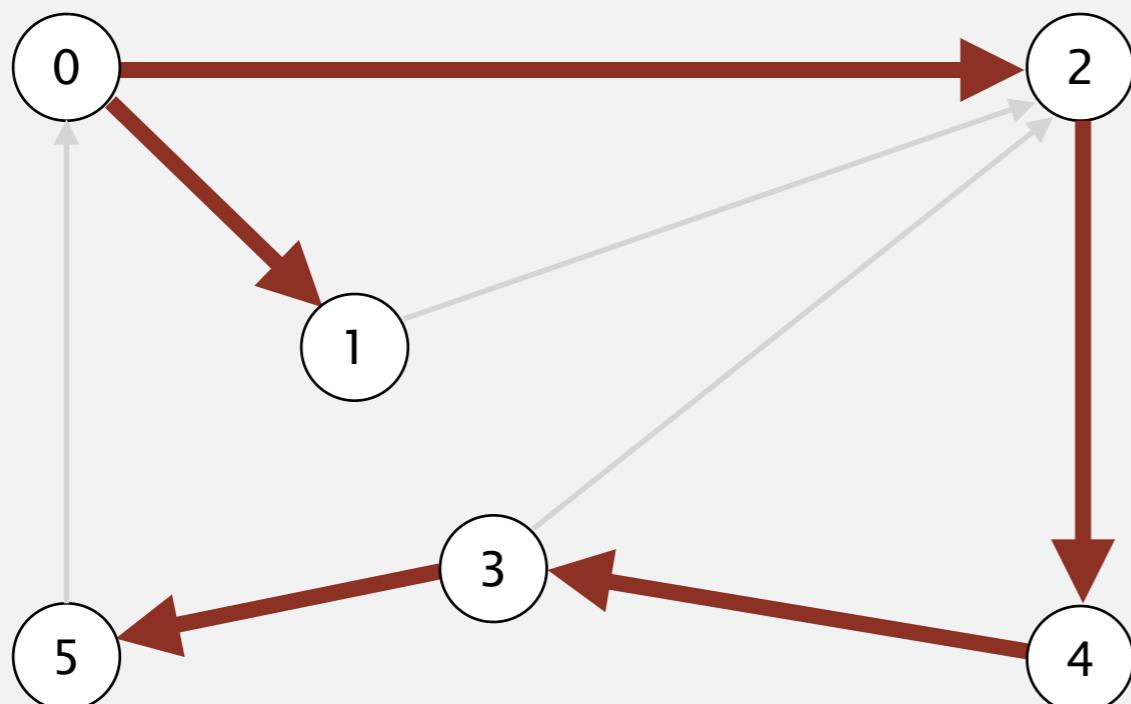
$v$	edgeTo[]	distTo[]
0	-	0
1	0	1
2	0	1
3	4	3
4	2	2
5	3	4

done

# Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices pointing from  $v$  and mark them.



$v$	edgeTo[]	distTo[]
0	-	0
1	0	1
2	0	1
3	4	3
4	2	2
5	3	4

done

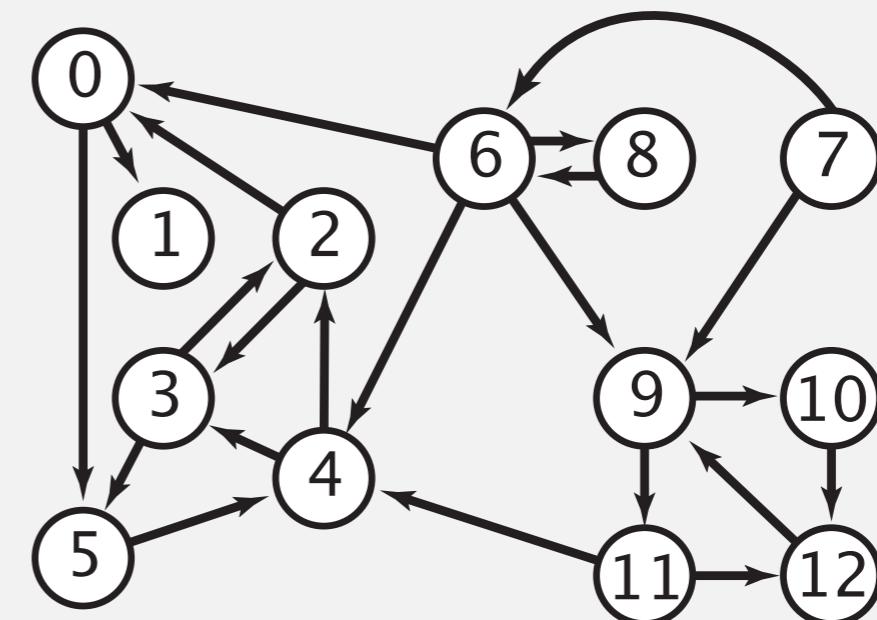
## Multiple-source shortest paths

---

**Multiple-source shortest paths.** Given a digraph and a **set** of source vertices, find shortest path from any vertex in the set to each other vertex.

**Ex.**  $S = \{ 1, 7, 10 \}$ .

- Shortest path to 4 is  $7 \rightarrow 6 \rightarrow 4$ .
- Shortest path to 5 is  $7 \rightarrow 6 \rightarrow 0 \rightarrow 5$ .
- Shortest path to 12 is  $10 \rightarrow 12$ .
- ...



**Q.** How to implement multi-source shortest paths algorithm?

**A.** Use BFS, but initialize by enqueueing all source vertices.

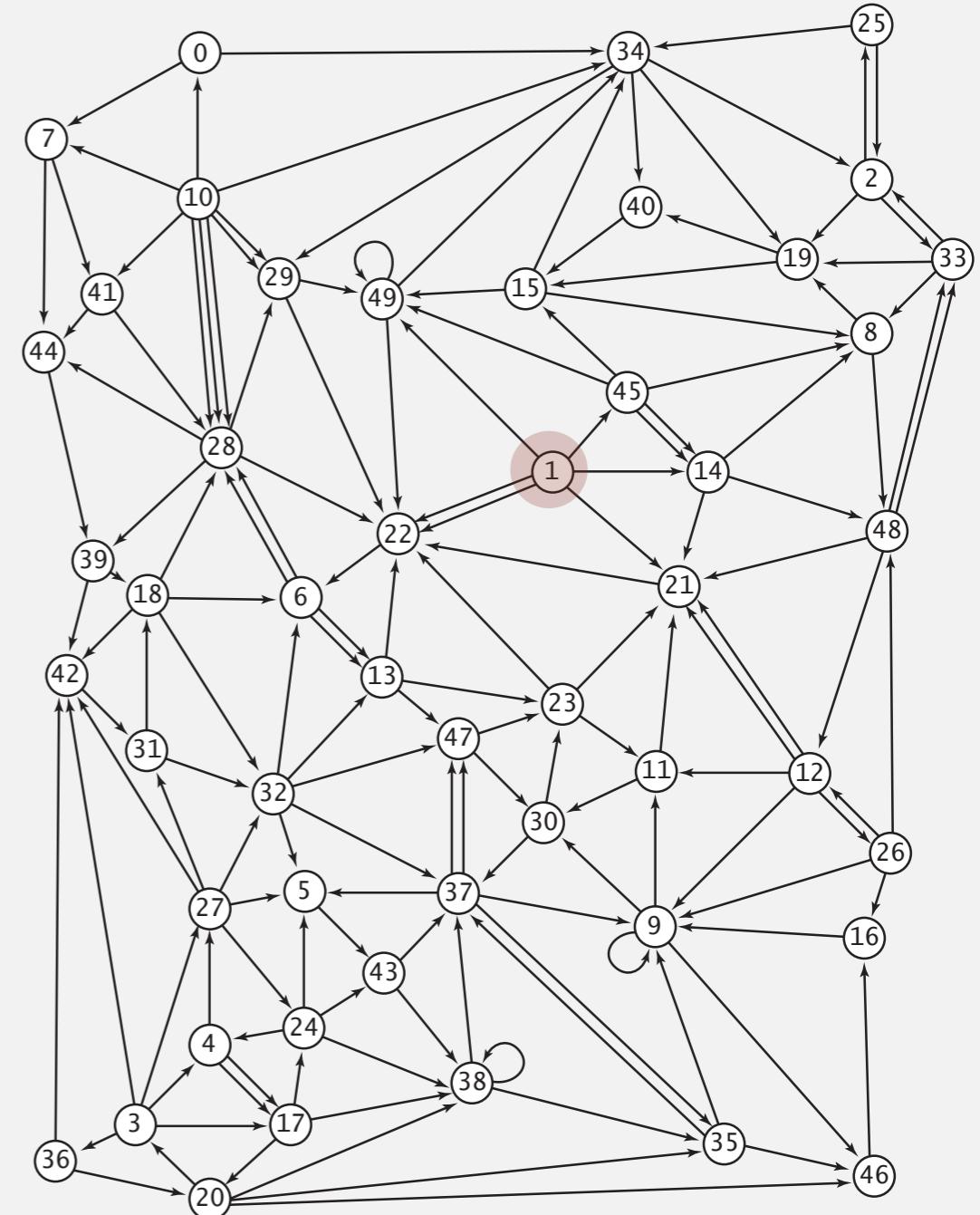
# Breadth-first search in digraphs application: web crawler

**Goal.** Crawl web, starting from some root web page, say [www.princeton.edu](http://www.princeton.edu).

**Solution.** [BFS with implicit digraph]

- Choose root web page as source  $s$ .
- Maintain a Queue of websites to explore.
- Maintain a SET of discovered websites.
- Dequeue the next website and enqueue websites to which it links  
(provided you haven't done so before).

**Q.** Why not use DFS?



# Bare-bones web crawler: Java implementation

```
Queue<String> queue = new Queue<String>();           ← queue of websites to crawl
SET<String> marked = new SET<String>();             ← set of marked websites

String root = "http://www.princeton.edu";
queue.enqueue(root);                                ← start crawling from root website
marked.add(root);

while (!queue.isEmpty())
{
    String v = queue.dequeue();
    StdOut.println(v);
    In in = new In(v);
    String input = in.readAll();

    String regexp = "http://(\w+\.\w+)+(\w+)";
    Pattern pattern = Pattern.compile(regexp);
    Matcher matcher = pattern.matcher(input);          ← use regular expression to find all URLs
                                                       in website of form http://xxx.yyy.zzz
                                                       [crude pattern misses relative URLs]
    while (matcher.find())
    {
        String w = matcher.group();
        if (!marked.contains(w))
        {
            marked.add(w);
            queue.enqueue(w);
        }
    }
}
```

if unmarked, mark it and put on the queue

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

## 4.2 DIRECTED GRAPHS

---

- ▶ *introduction*
- ▶ *digraph API*
- ▶ *digraph search*
- ▶ *topological sort*
- ▶ ***strong components***

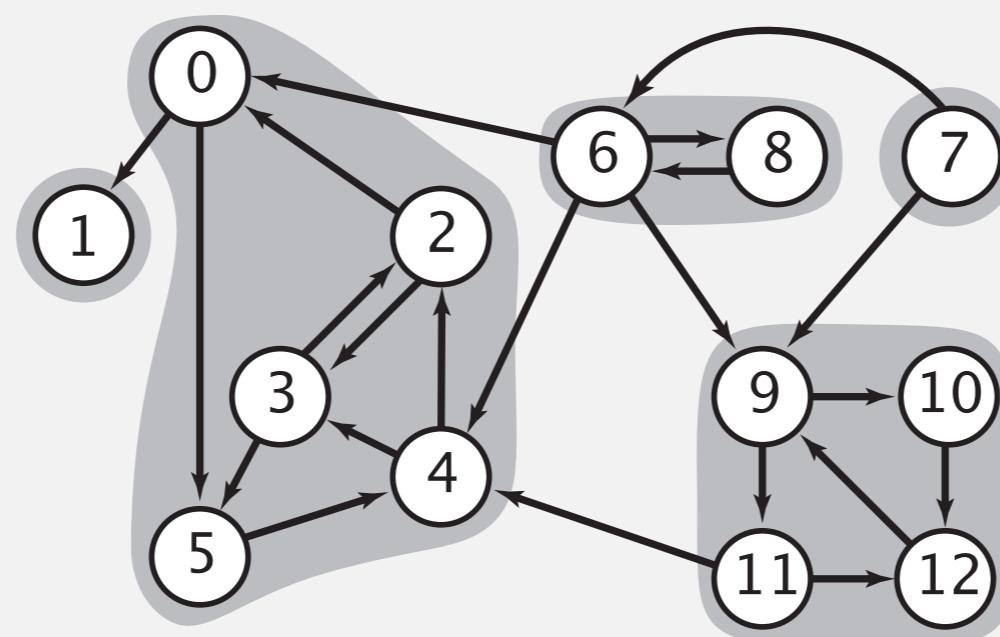
# Strongly-connected components

Def. Vertices  $v$  and  $w$  are **strongly connected** if there is both a directed path from  $v$  to  $w$  **and** a directed path from  $w$  to  $v$ .

Key property. Strong connectivity is an **equivalence relation**:

- $v$  is strongly connected to  $v$ .
- If  $v$  is strongly connected to  $w$ , then  $w$  is strongly connected to  $v$ .
- If  $v$  is strongly connected to  $w$  and  $w$  to  $x$ , then  $v$  is strongly connected to  $x$ .

Def. A **strong component** is a maximal subset of strongly-connected vertices.

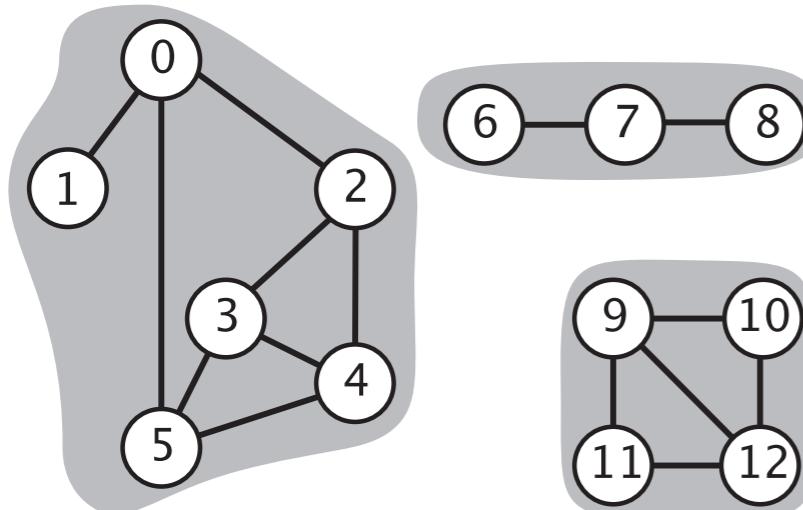


5 strongly-connected components

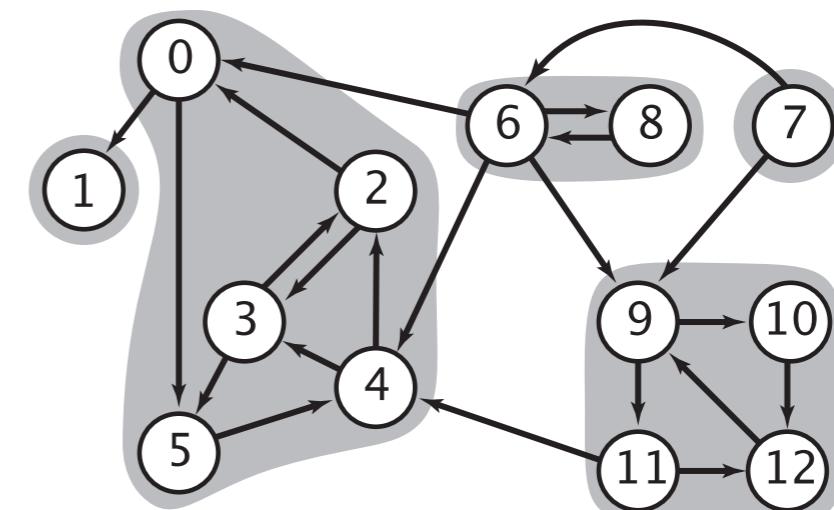
# Connected components vs. strongly-connected components

v and w are **connected** if there is a path between v and w

v and w are **strongly connected** if there is both a directed path from v to w and a directed path from w to v



3 connected components



5 strongly-connected components

connected component id (easy to compute with DFS)

0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	1	1	1	2	2	2	2

strongly-connected component id (how to compute?)

0	1	2	3	4	5	6	7	8	9	10	11	12
1	0	1	1	1	1	3	4	3	2	2	2	2

```
public boolean connected(int v, int w)  
{ return id[v] == id[w]; }
```

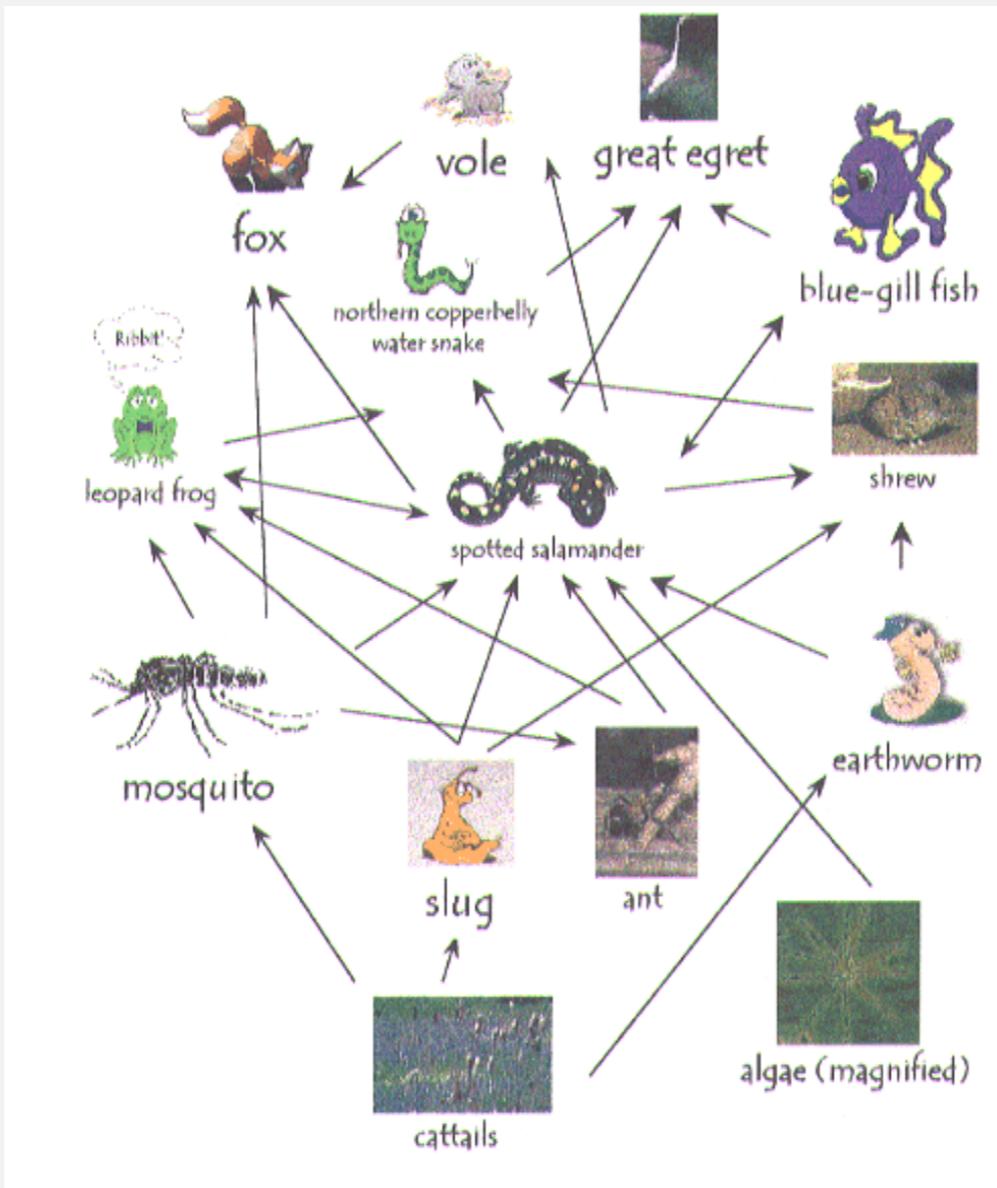
```
public boolean stronglyConnected(int v, int w)  
{ return id[v] == id[w]; }
```

constant-time client connectivity query

constant-time client strong-connectivity query

# Strong component application: ecological food webs

Food web graph. Vertex = species; edge = from producer to consumer.



<http://www.twingroves.district96.k12.il.us/Wetlands/Salamander/SalGraphics/salfoodweb.gif>

Strong component. Subset of species with common energy flow.

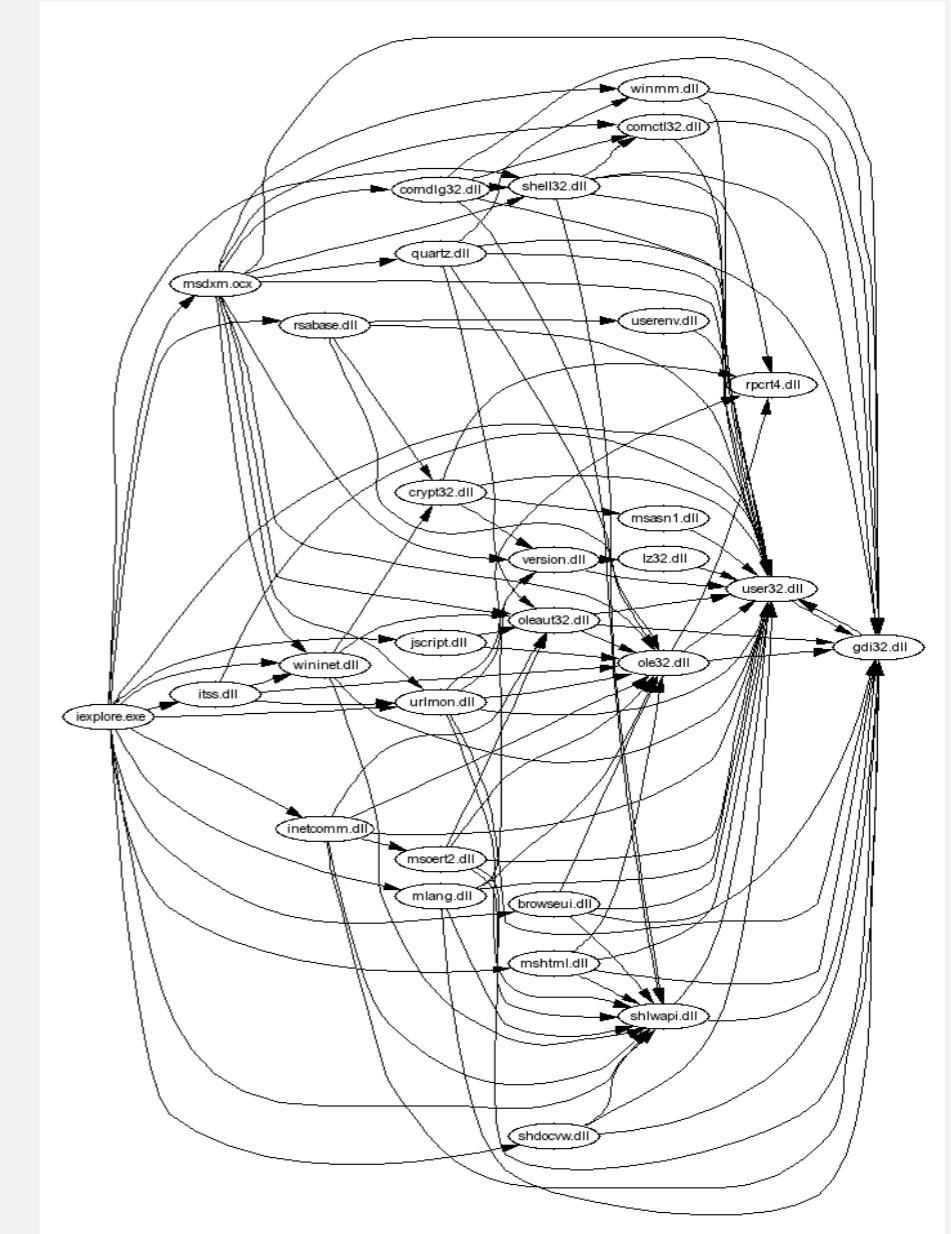
# Strong component application: software modules

## Software module dependency graph.

- Vertex = software module.
- Edge: from module to dependency.



Firefox



Internet Explorer

Strong component. Subset of mutually interacting modules.

Approach 1. Package strong components together.

Approach 2. Use to improve design!

# Strong components algorithms: brief history

---

## 1960s: Core OR problem.

- Widely studied; some practical algorithms.
- Complexity not understood.

## 1972: linear-time DFS algorithm (Tarjan).

- Classic algorithm.
- Level of difficulty: Algs4++.
- Demonstrated broad applicability and importance of DFS.

## 1980s: easy two-pass linear-time algorithm (Kosaraju-Sharir).

- Forgot notes for lecture; developed algorithm in order to teach it!
- Later found in Russian scientific literature (1972).

## 1990s: more easy linear-time algorithms.

- Gabow: fixed old OR algorithm.
- Cheriyan-Mehlhorn: needed one-pass algorithm for LEDA.

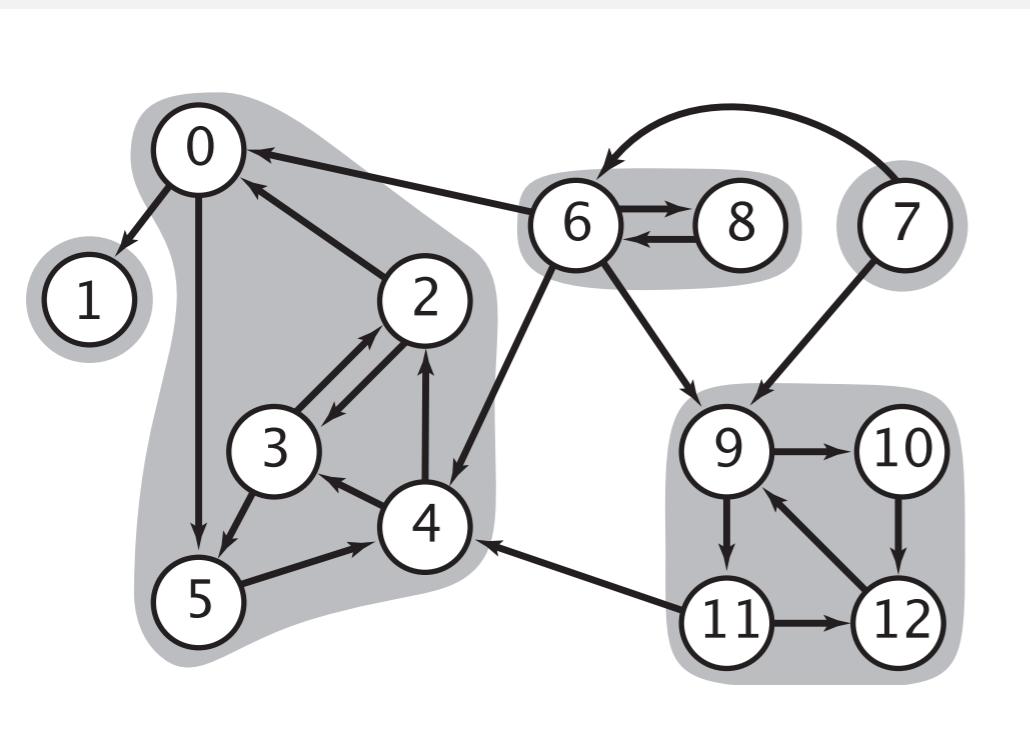
# Kosaraju-Sharir algorithm: intuition

Reverse graph. Strong components in  $G$  are same as in  $G^R$ .

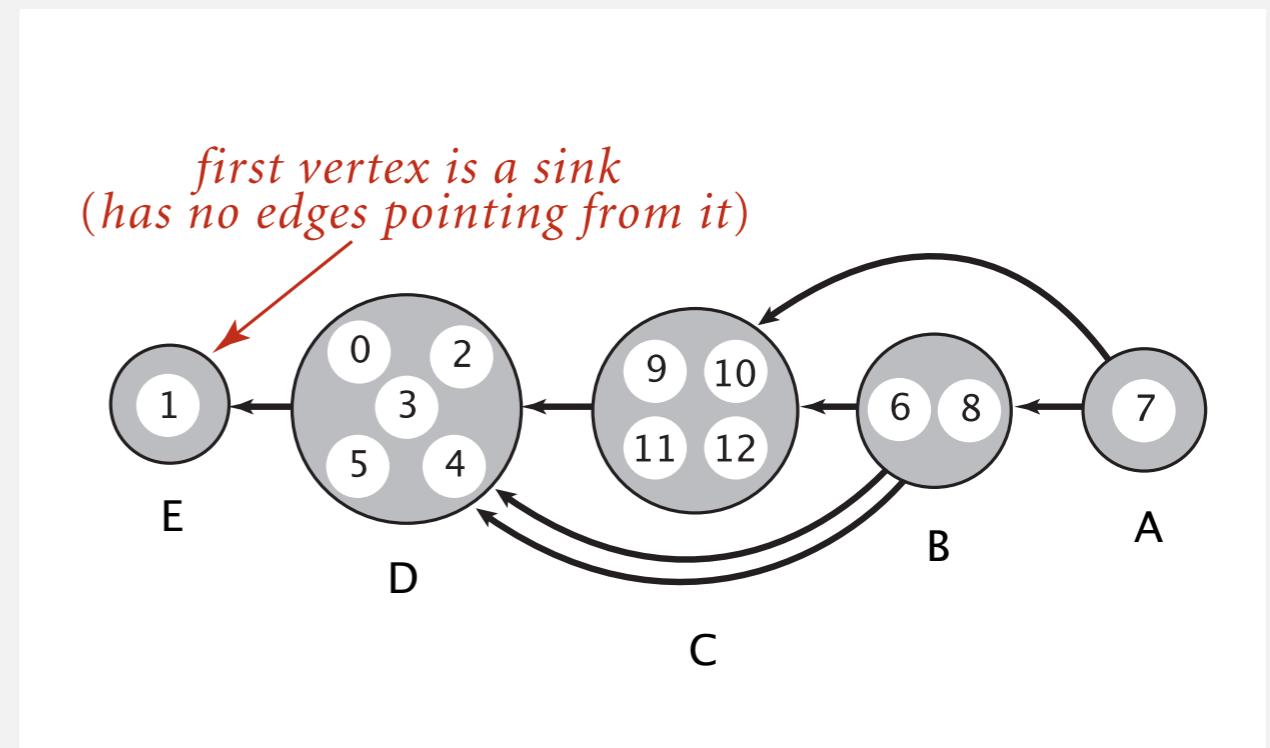
Kernel DAG. Contract each strong component into a single vertex.

Idea.

- Compute topological order (reverse postorder) in kernel DAG.
- Run DFS, considering vertices in reverse topological order.



digraph G and its strong components

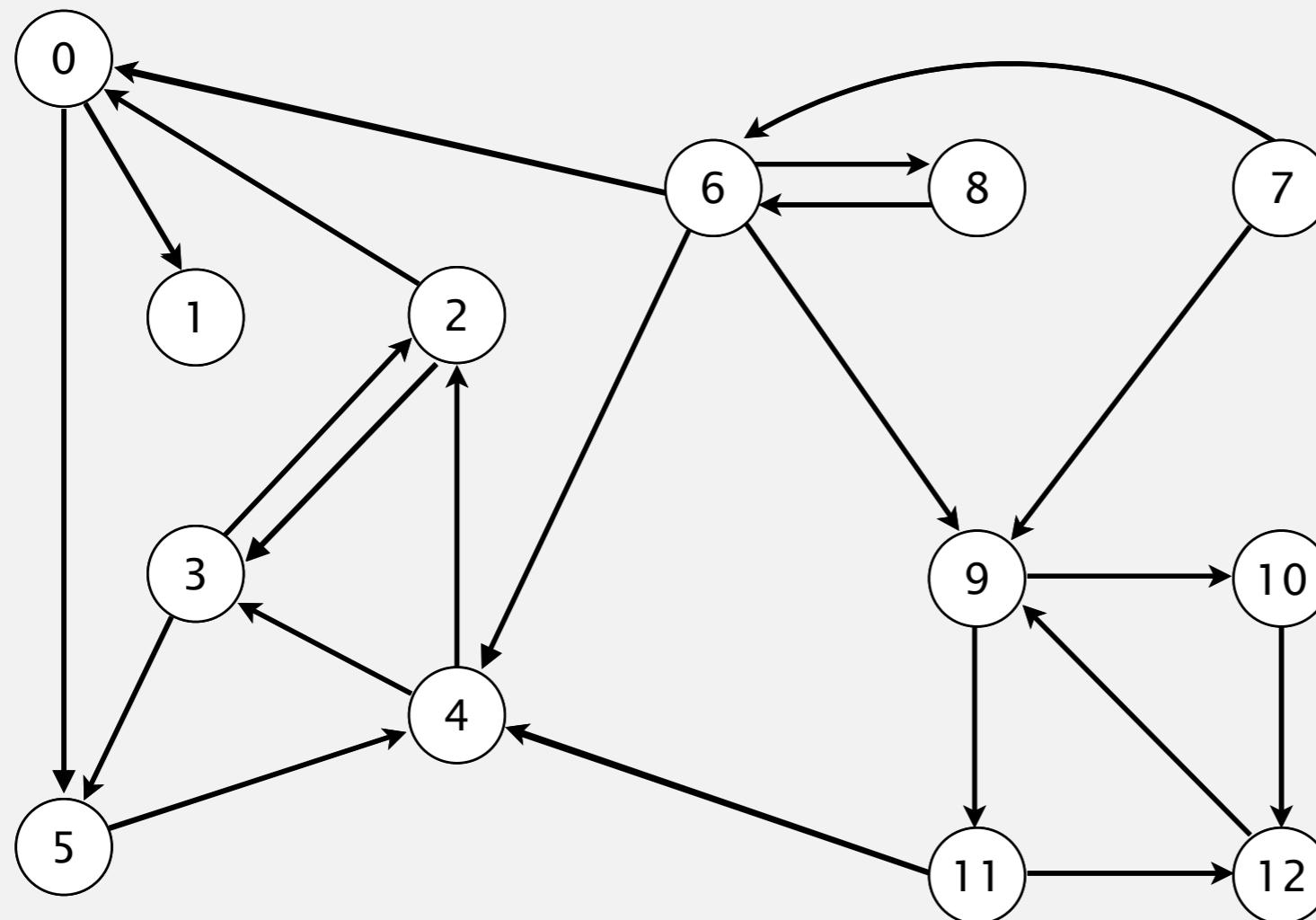


kernel DAG of G (topological order: A B C D E)

# Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in  $G^R$ .

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .



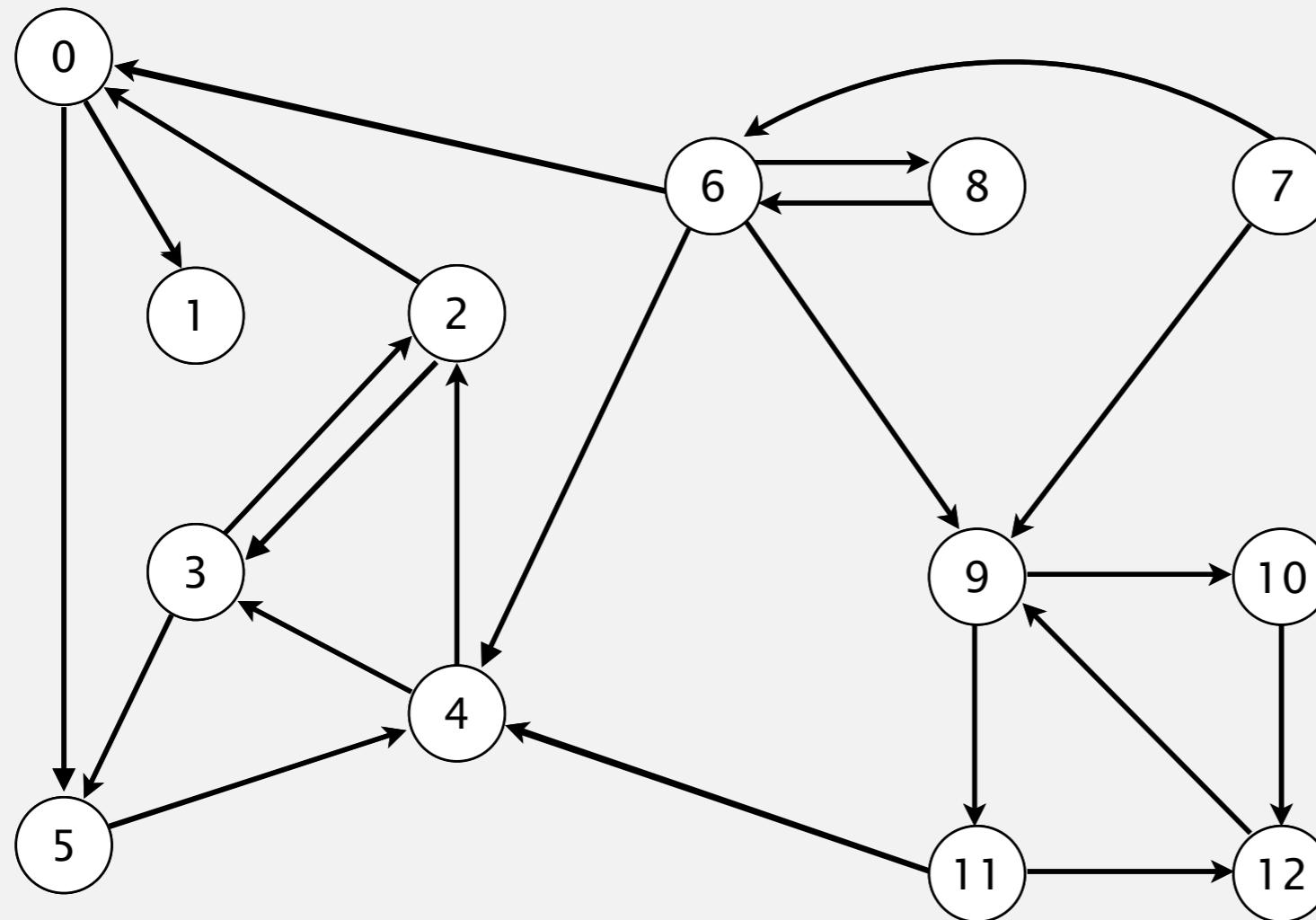
**digraph G**

4→2  
2→3  
3→2  
6→0  
0→1  
2→0  
11→12  
12→9  
9→10  
9→11  
7→9  
10→12  
11→4  
4→3  
3→5  
6→8  
8→6  
5→4  
0→5

# Kosaraju-Sharir algorithm demo

---

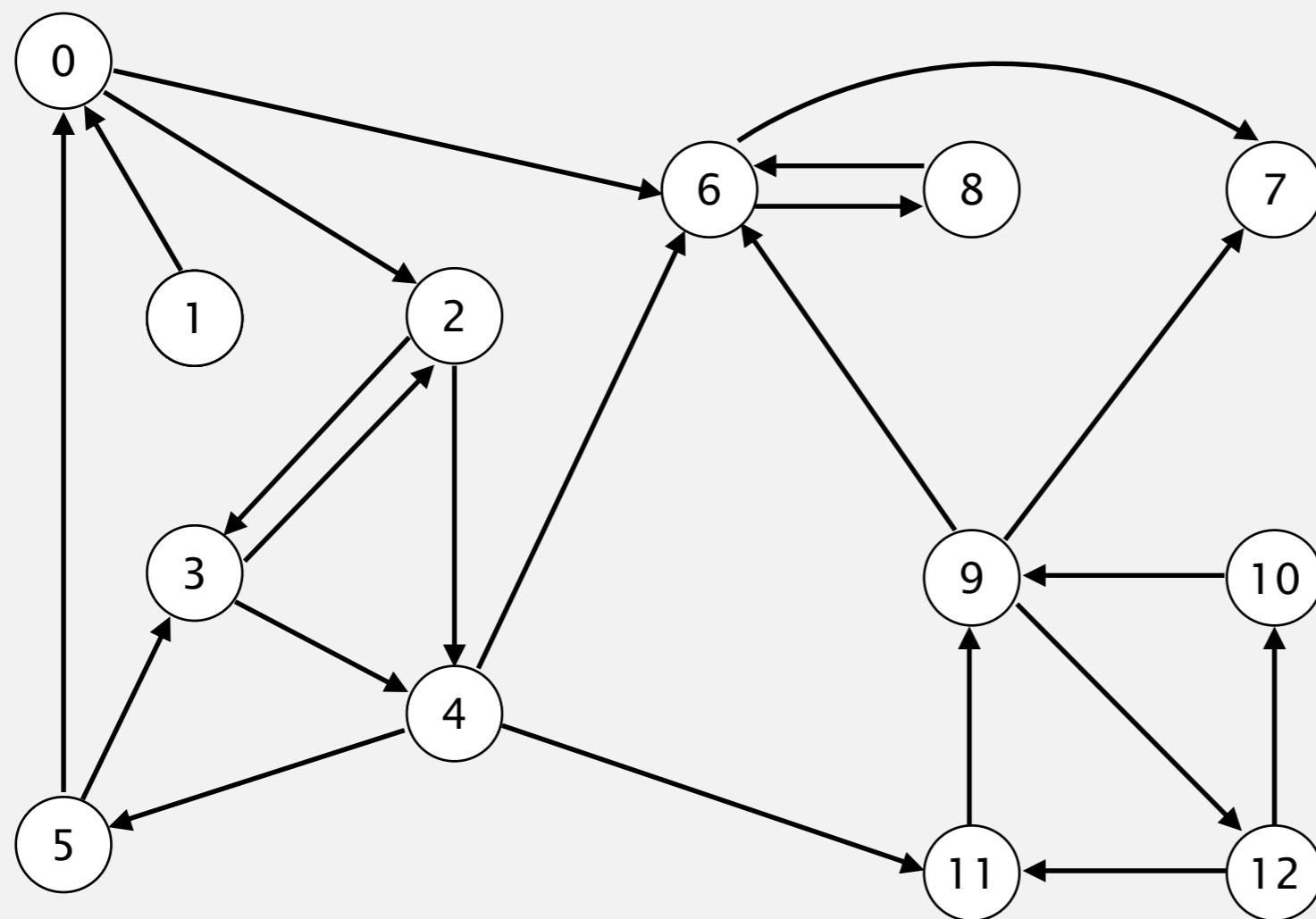
Phase 1. Compute reverse postorder in  $G^R$ .



**digraph G**

# Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in  $G^R$ .

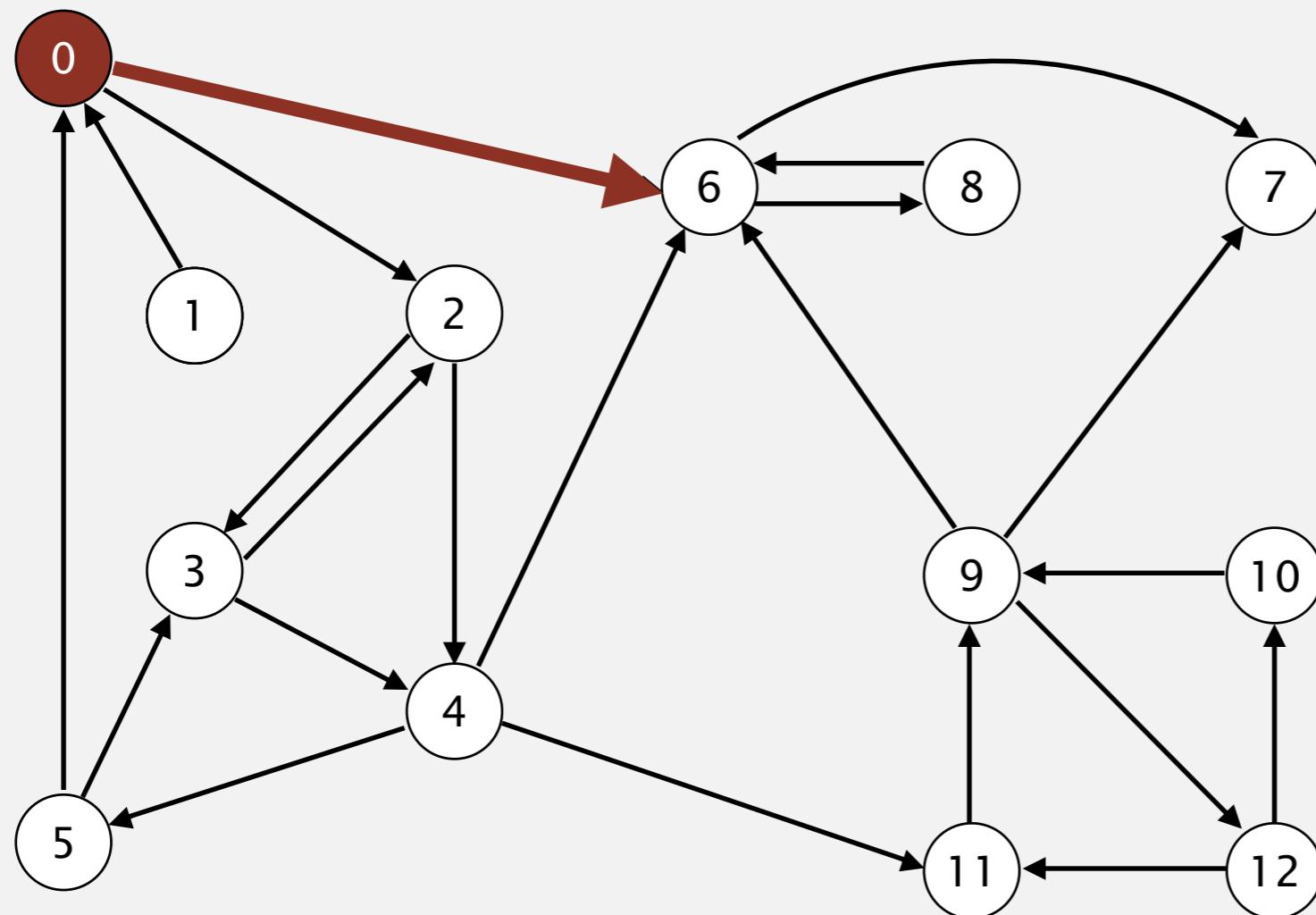


reverse digraph  $G^R$

v	marked[]
0	-
1	-
2	-
3	-
4	-
5	-
6	-
7	-
8	-
9	-
10	-
11	-
12	-

# Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in  $G^R$ .

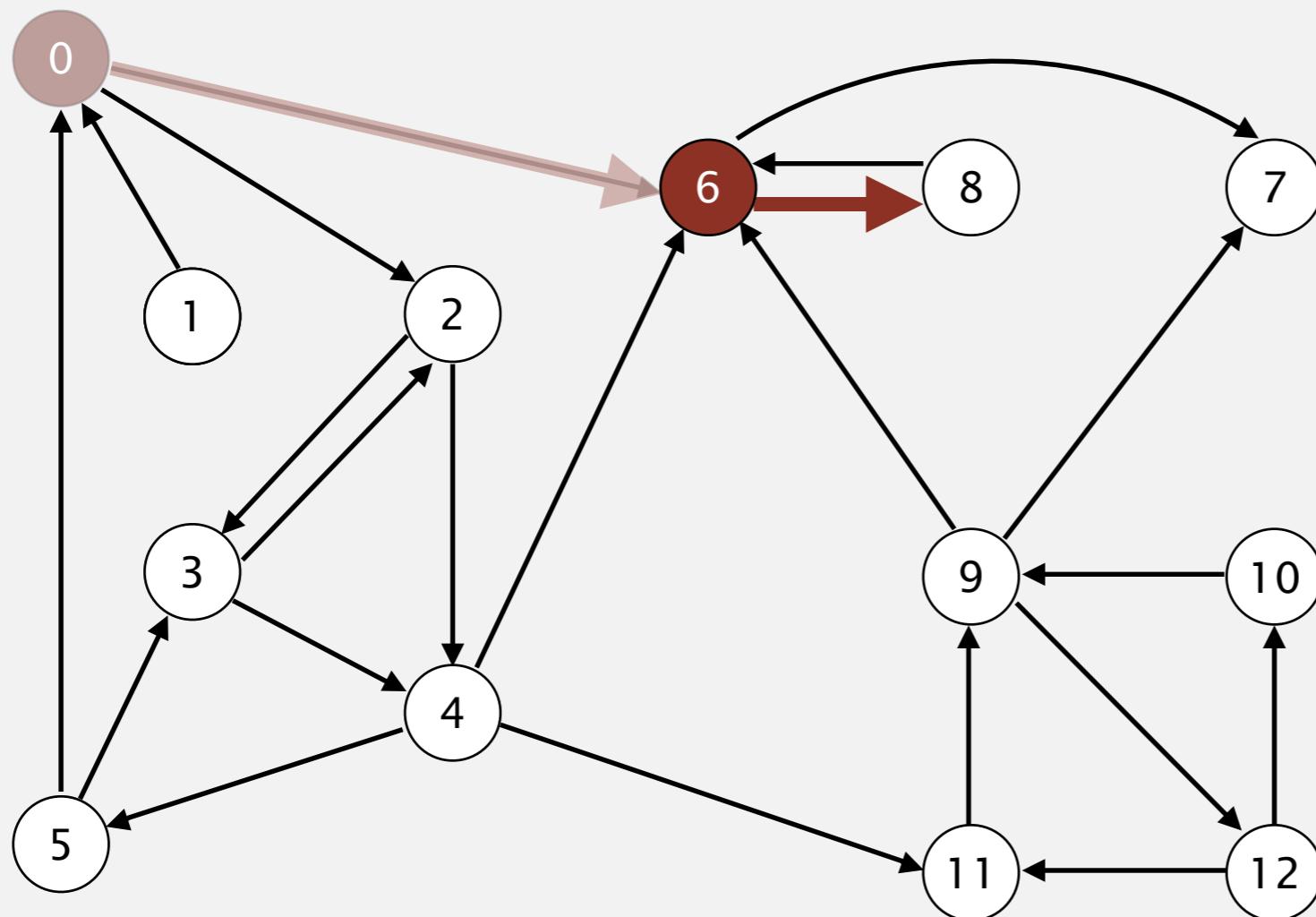


v	marked[]
0	T
1	F
2	F
3	F
4	F
5	F
6	F
7	F
8	F
9	F
10	F
11	F
12	F

visit 0: check 6 and check 2

# Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in  $G^R$ .

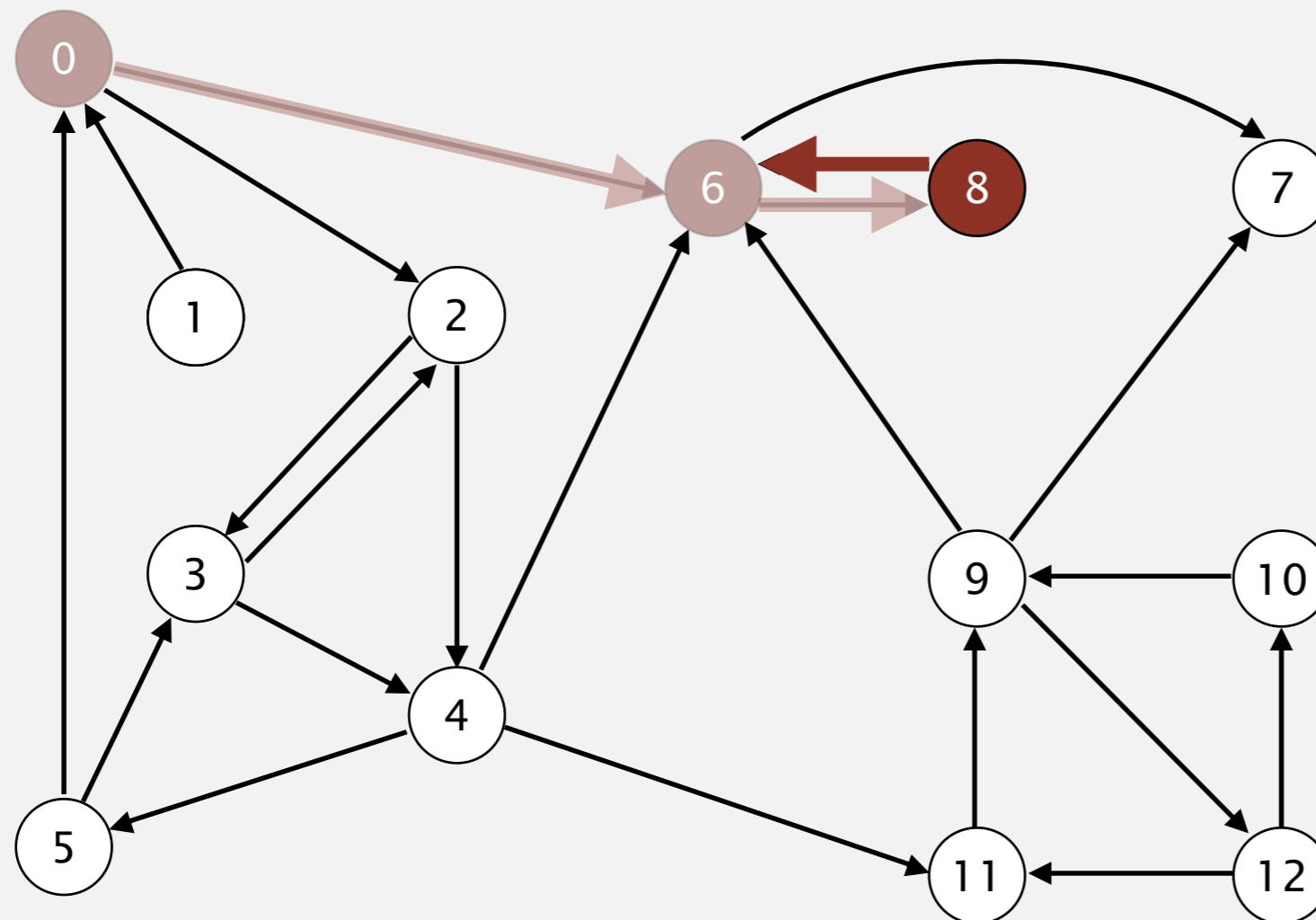


v	marked[]
0	T
1	F
2	F
3	F
4	F
5	F
6	T
7	F
8	F
9	F
10	F
11	F
12	F

visit 6: check 8 and check 7

# Kosaraju-Sharir algorithm demo

**Phase 1.** Compute reverse postorder in  $G^R$ .

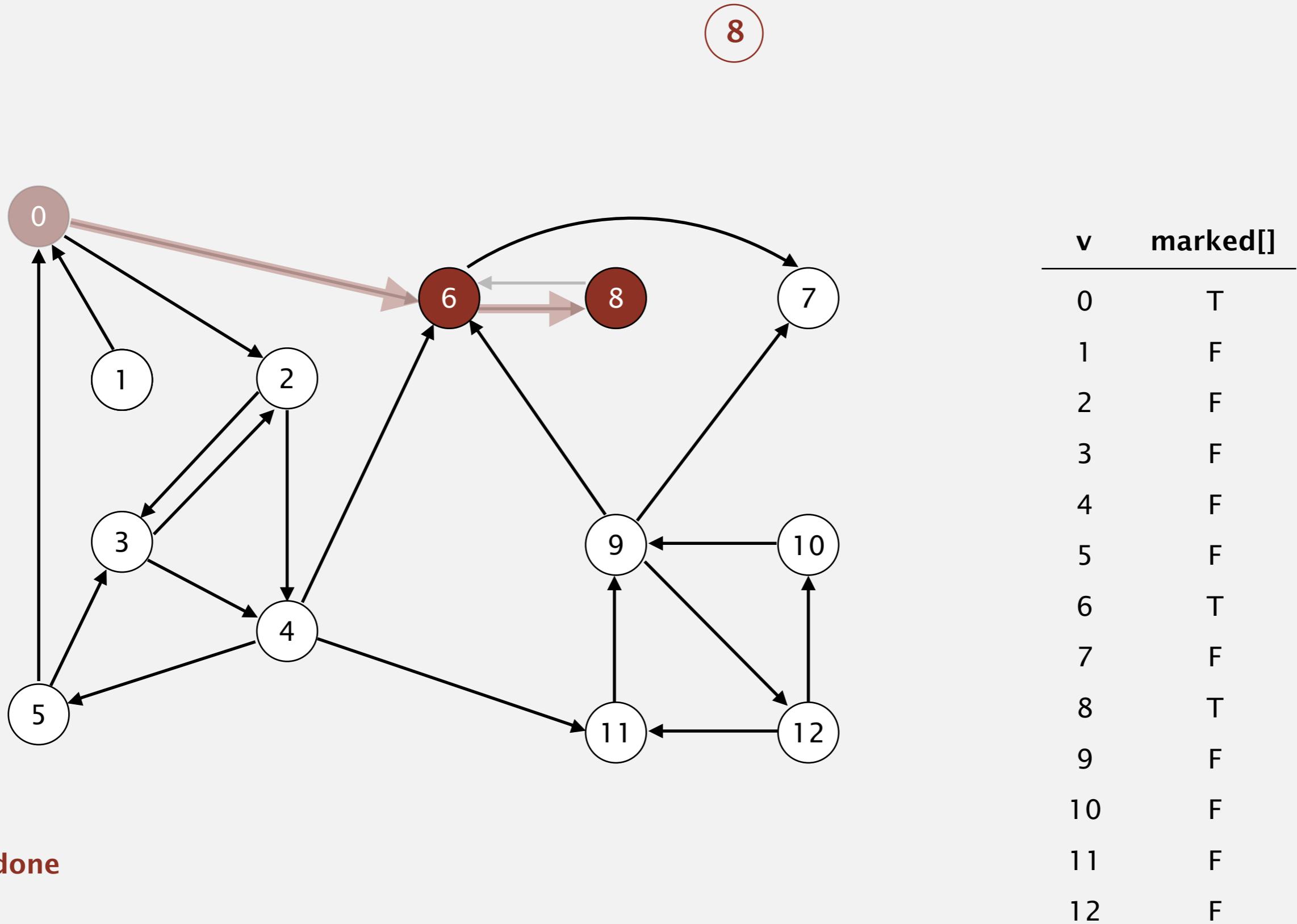


v	marked[]
0	T
1	F
2	F
3	F
4	F
5	F
6	T
7	F
8	T
9	F
10	F
11	F
12	F

## visit 8: check 6

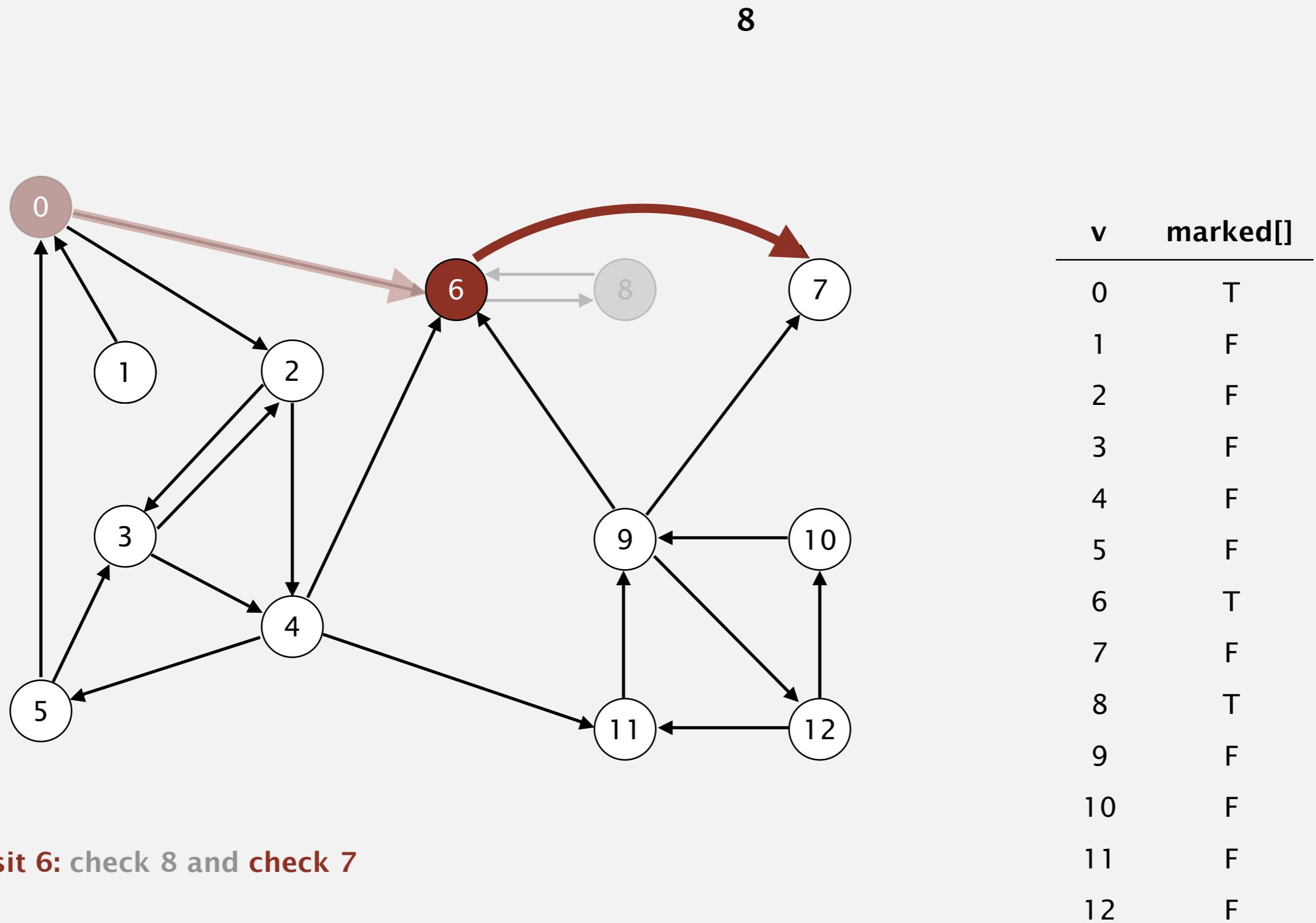
# Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in  $G^R$ .



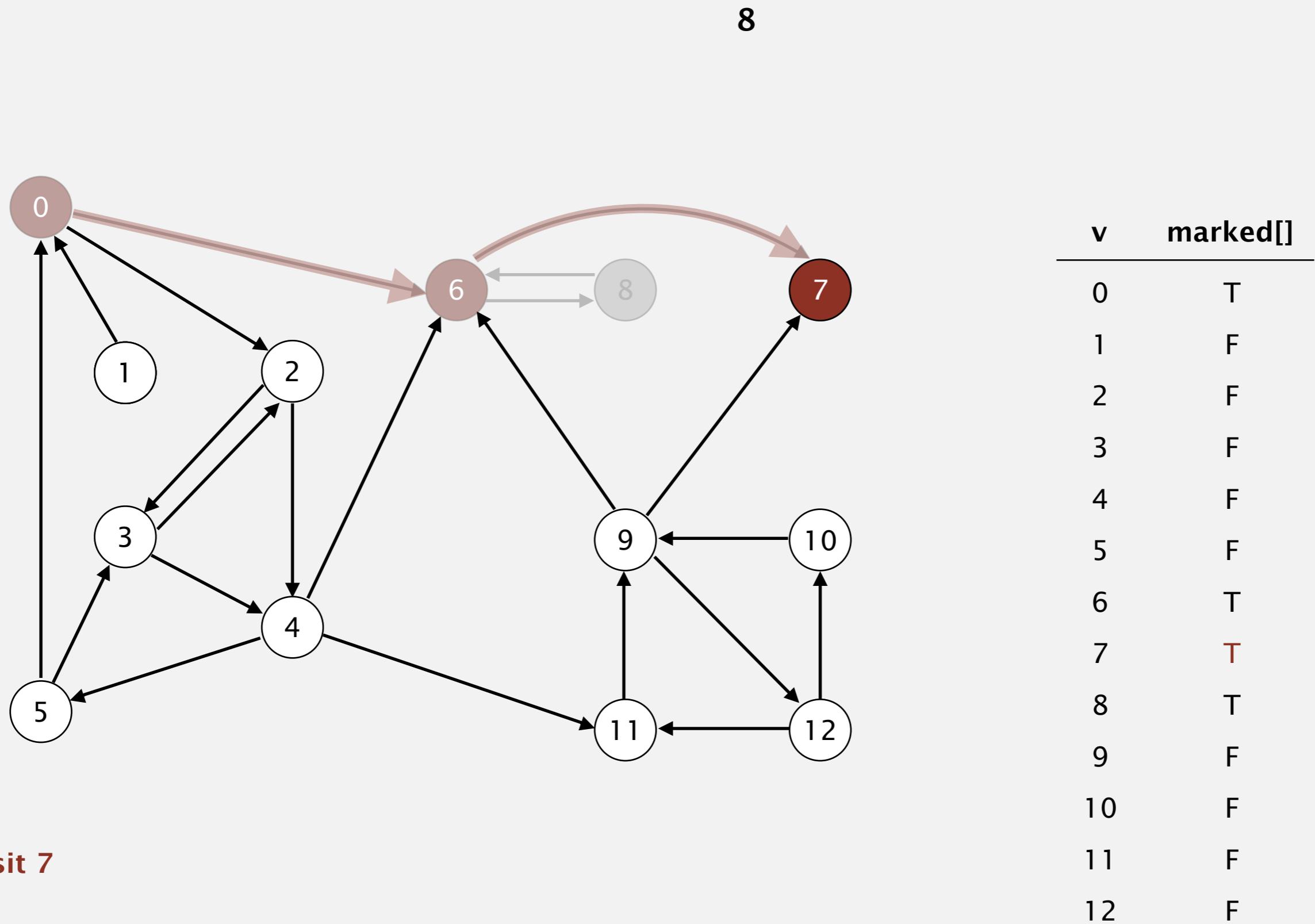
# Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in  $G^R$ .



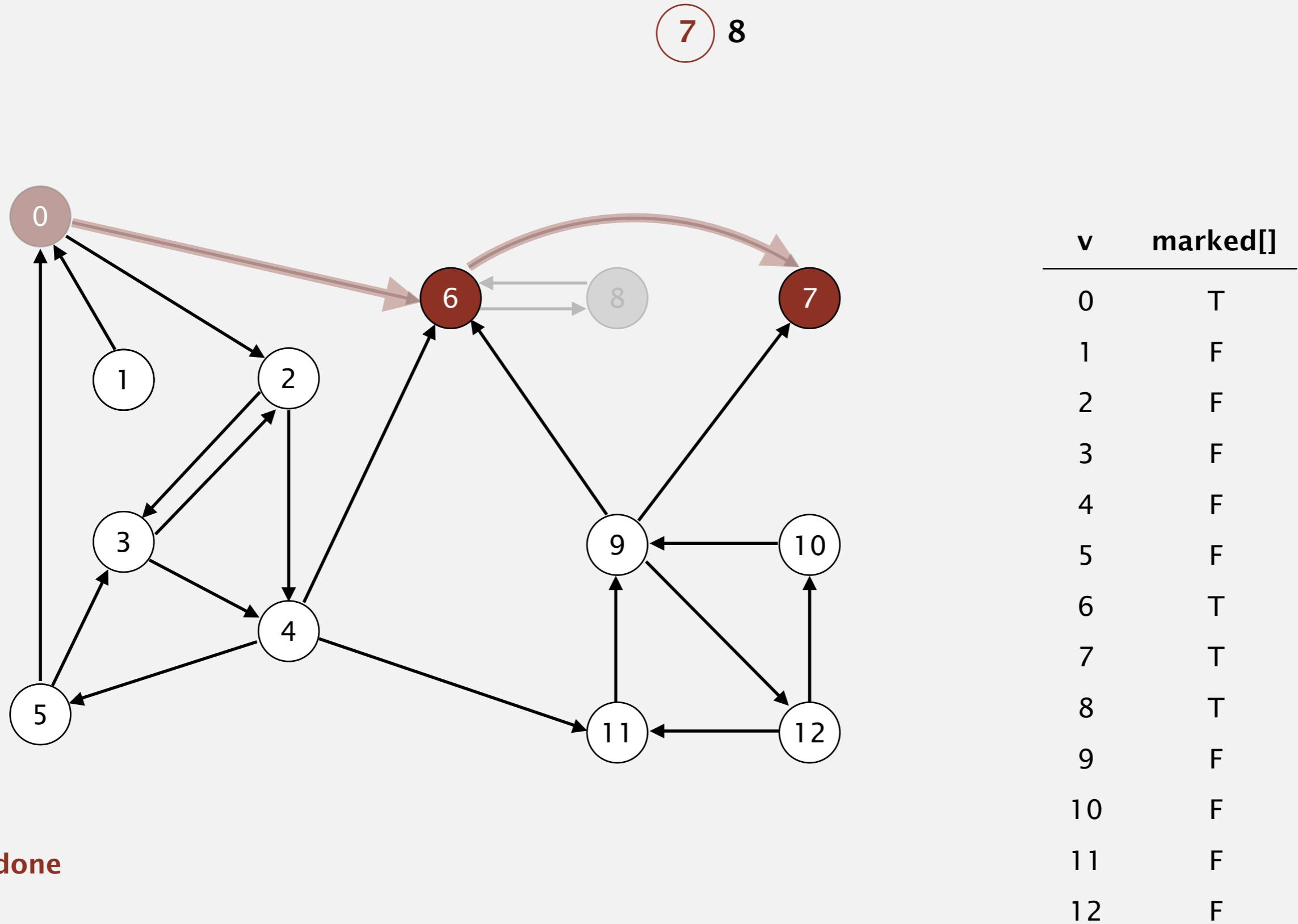
# Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in  $G^R$ .



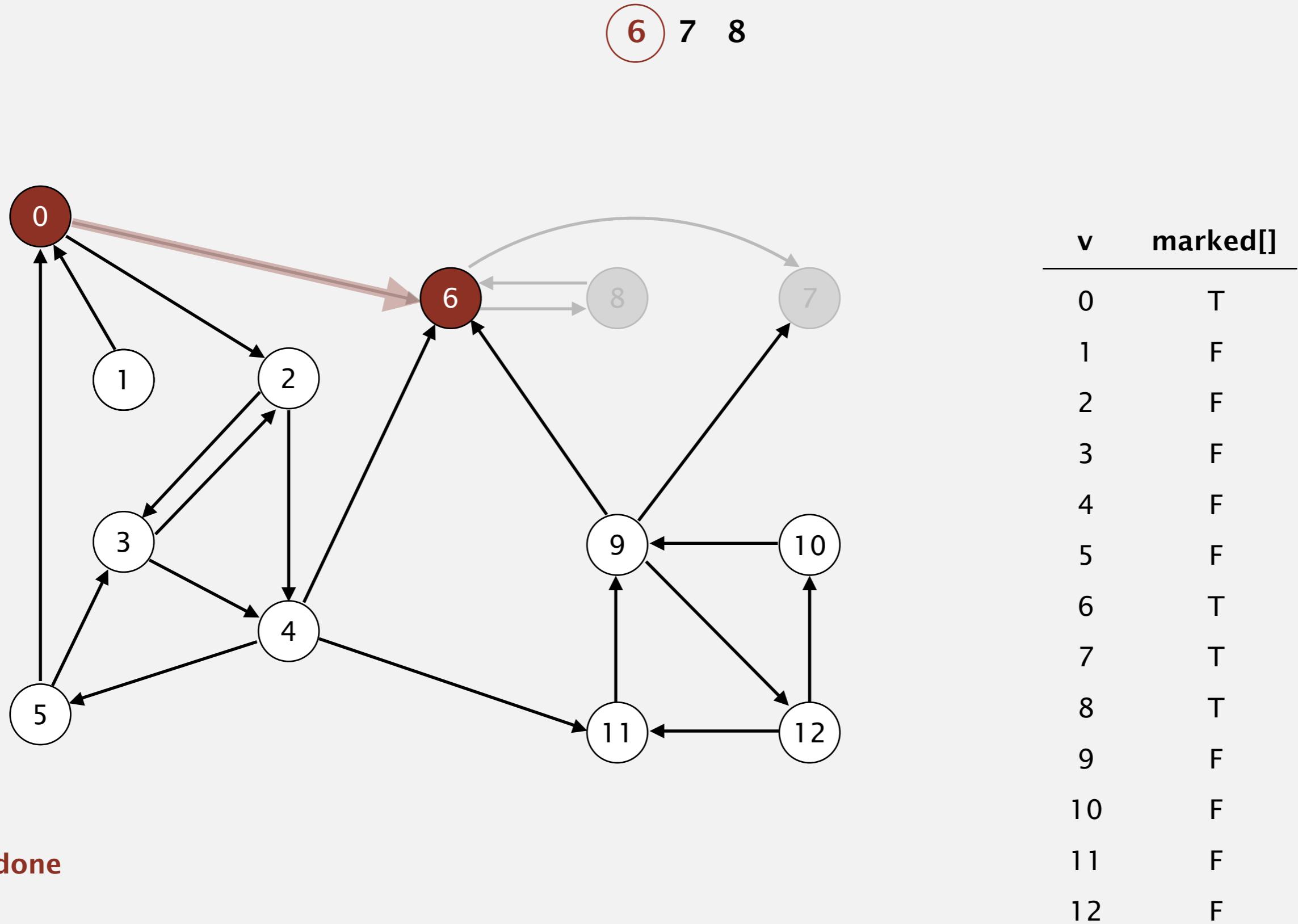
# Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in  $G^R$ .



# Kosaraju-Sharir algorithm demo

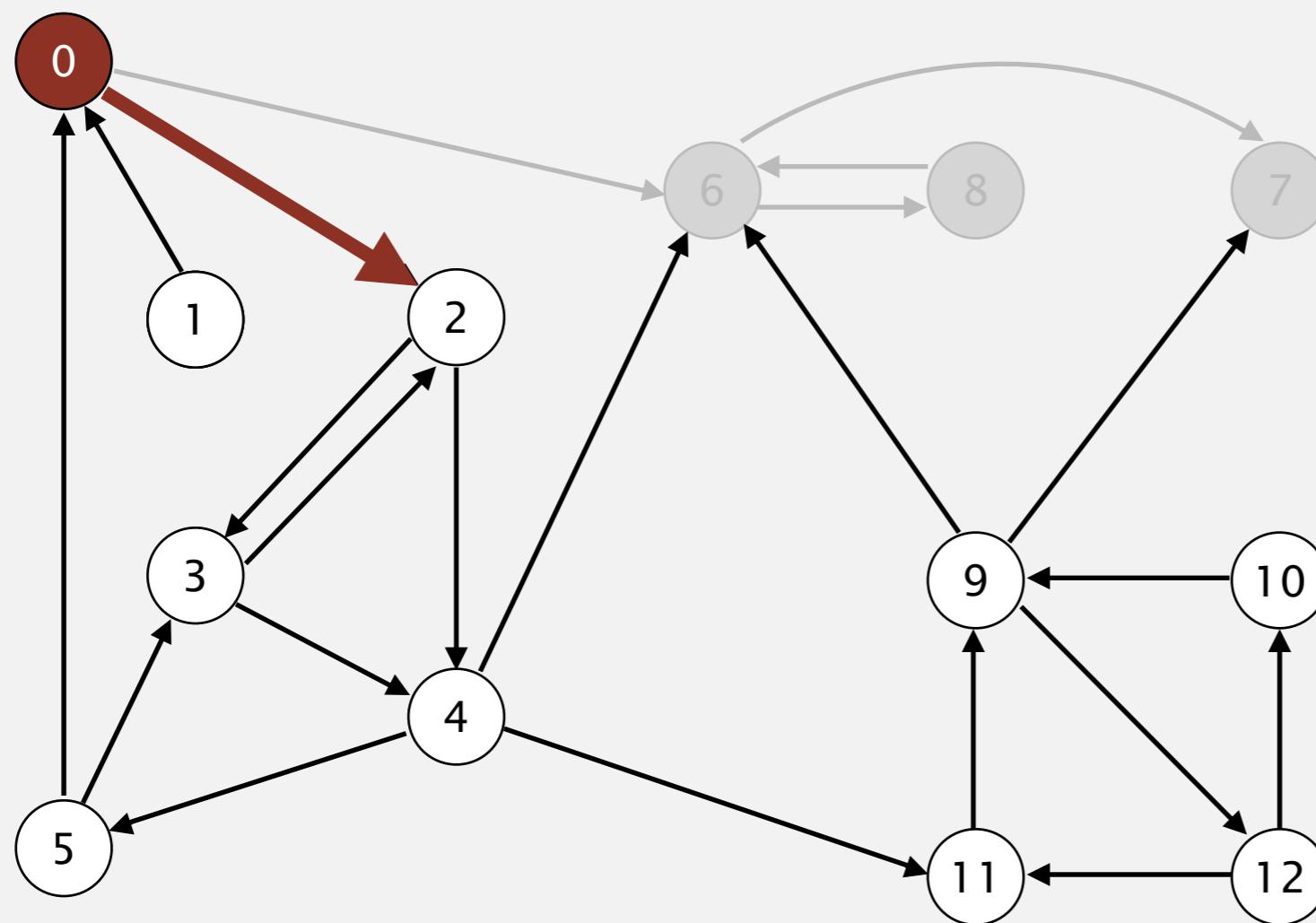
Phase 1. Compute reverse postorder in  $G^R$ .



# Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in  $G^R$ .

6 7 8



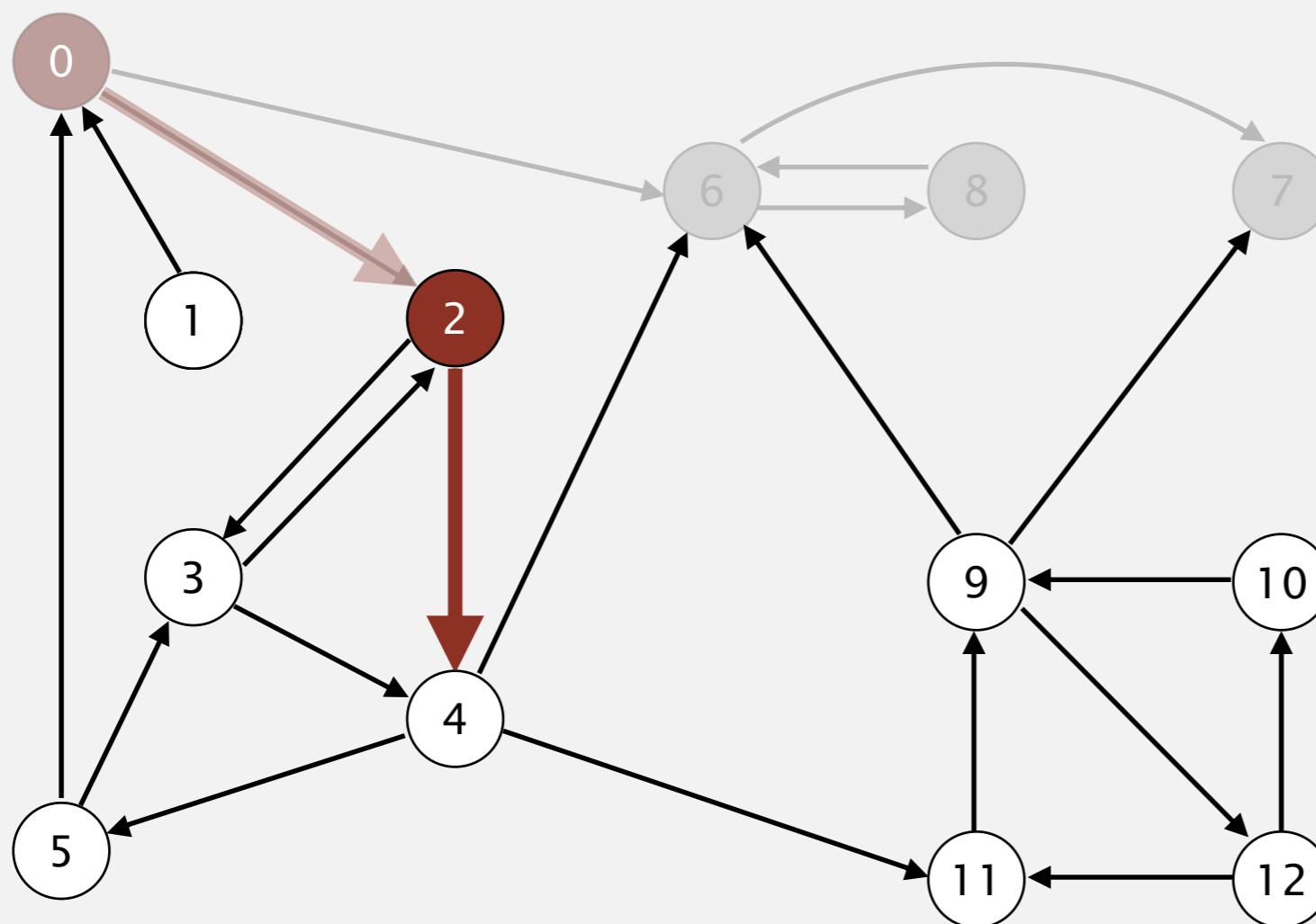
v	marked[]
0	T
1	F
2	F
3	F
4	F
5	F
6	T
7	T
8	T
9	F
10	F
11	F
12	F

visit 0: check 6 and check 2

# Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in  $G^R$ .

6 7 8



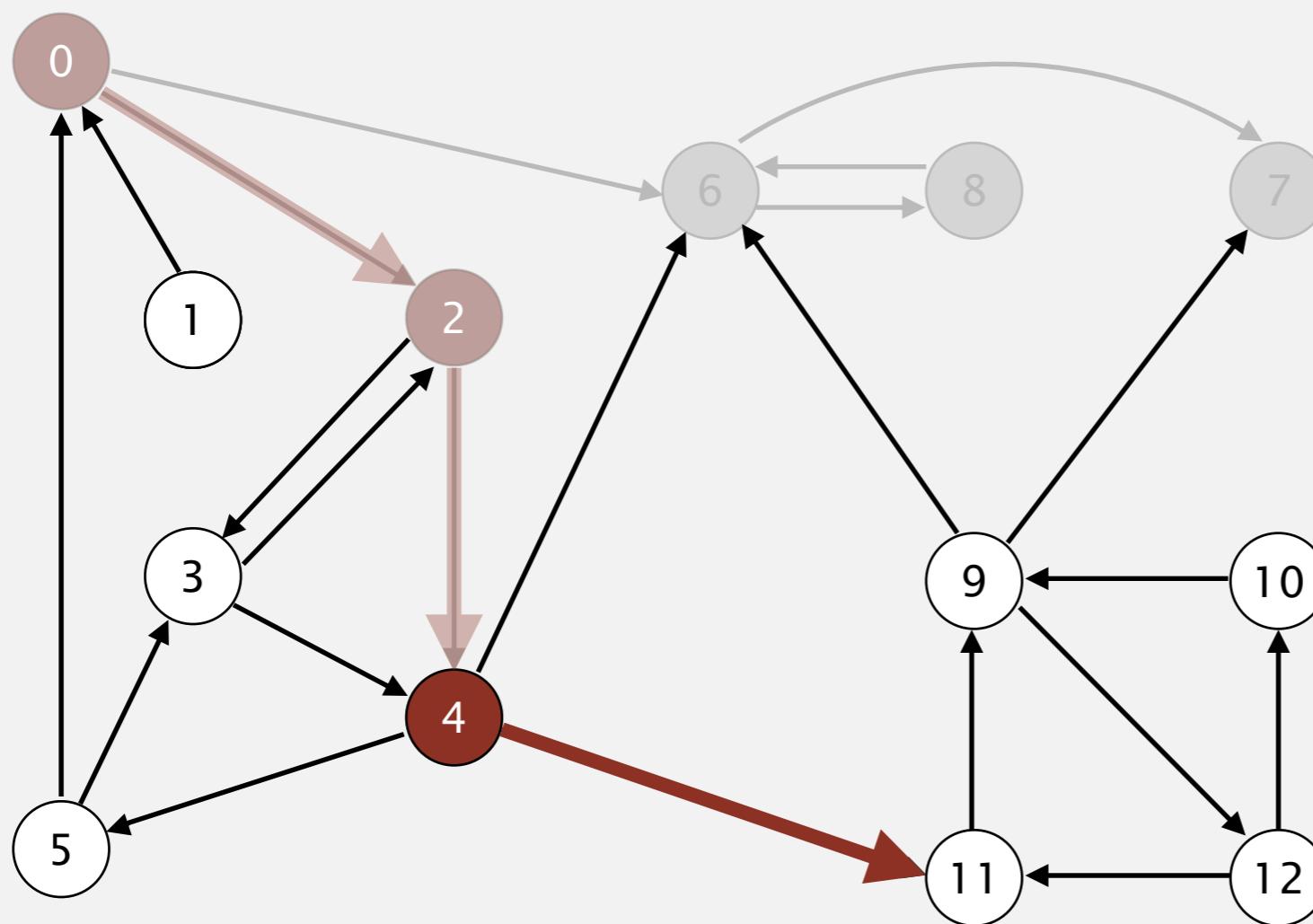
v	marked[]
0	T
1	F
2	T
3	F
4	F
5	F
6	T
7	T
8	T
9	F
10	F
11	F
12	F

visit 2: check 4 and check 3

# Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in  $G^R$ .

6 7 8



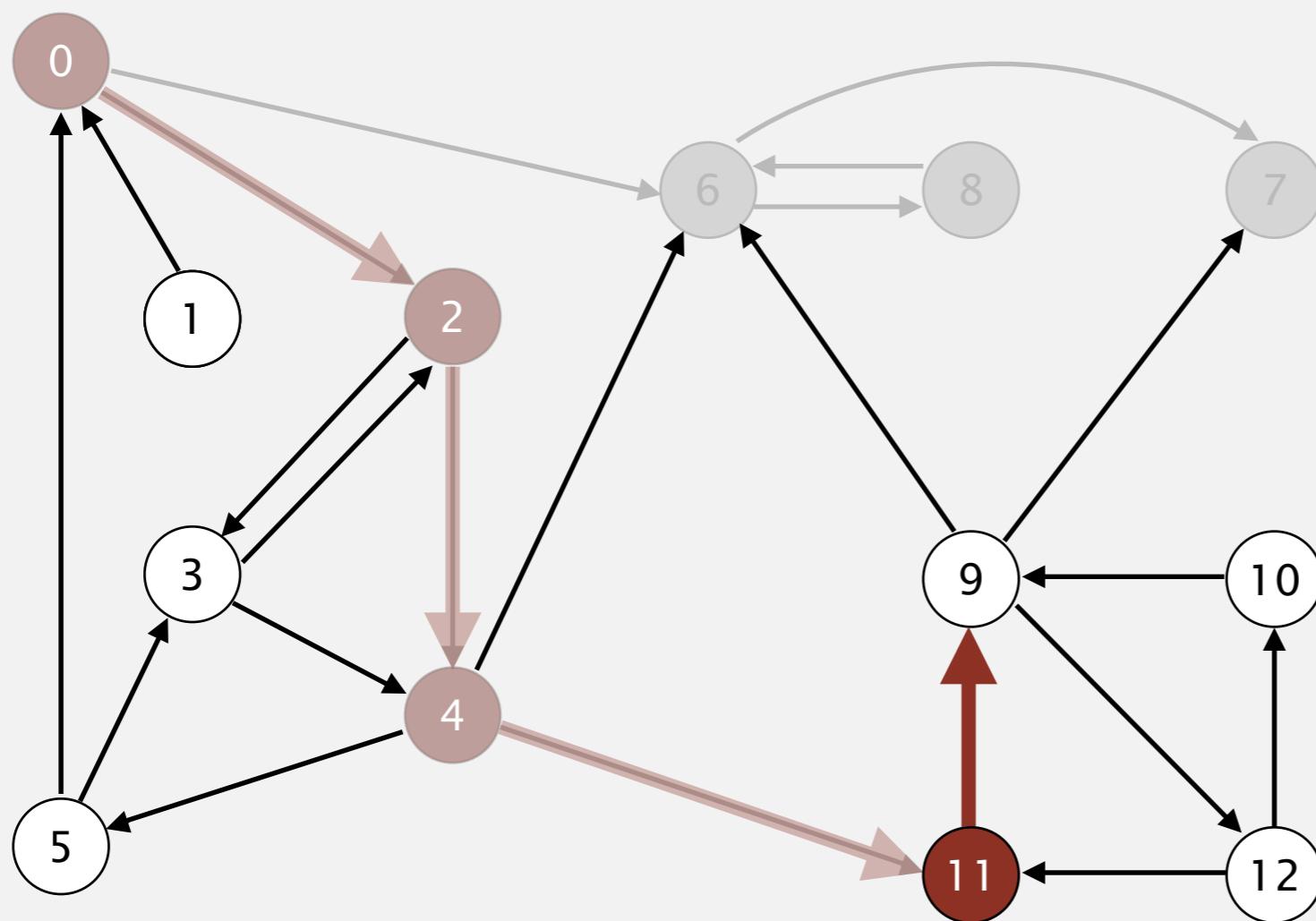
v	marked[]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	F
10	F
11	F
12	F

visit 4: check 11, check 6, and check 5

# Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in  $G^R$ .

6 7 8



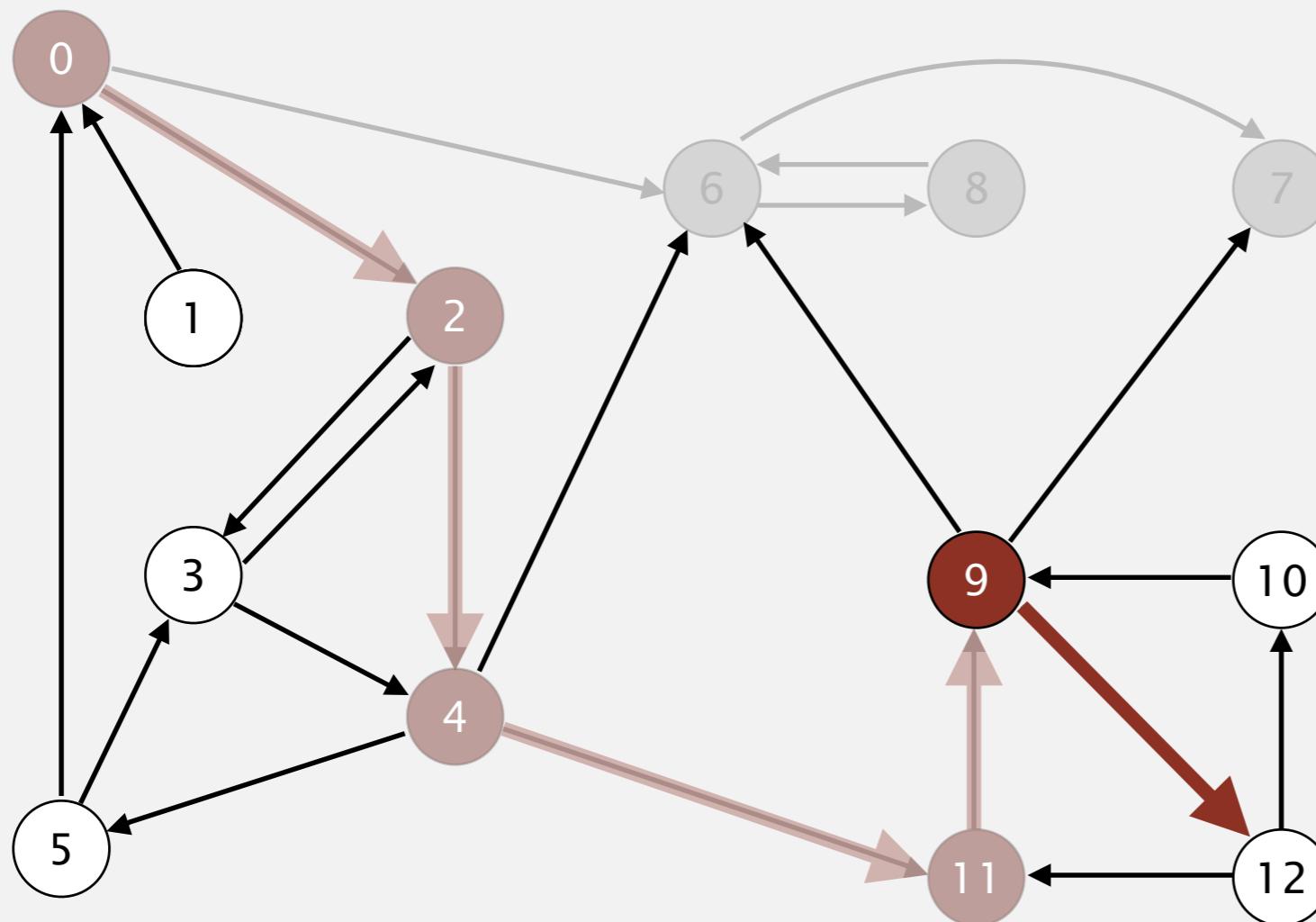
v	marked[]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	F
10	F
11	T
12	F

visit 11: check 9

# Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in  $G^R$ .

6 7 8



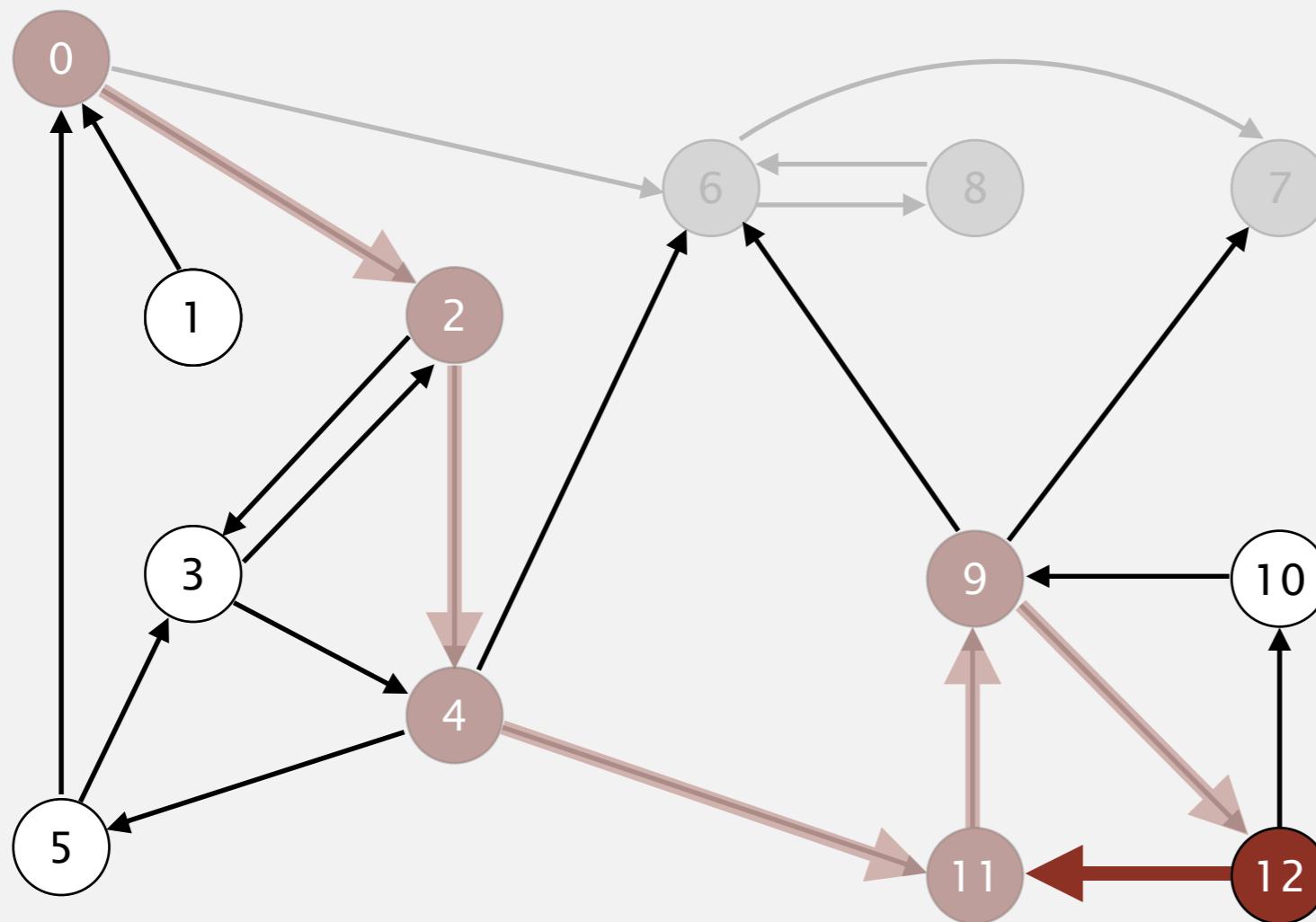
v	marked[]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	T
10	F
11	T
12	F

visit 9: check 12, check 7, and check 6

# Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in  $G^R$ .

6 7 8



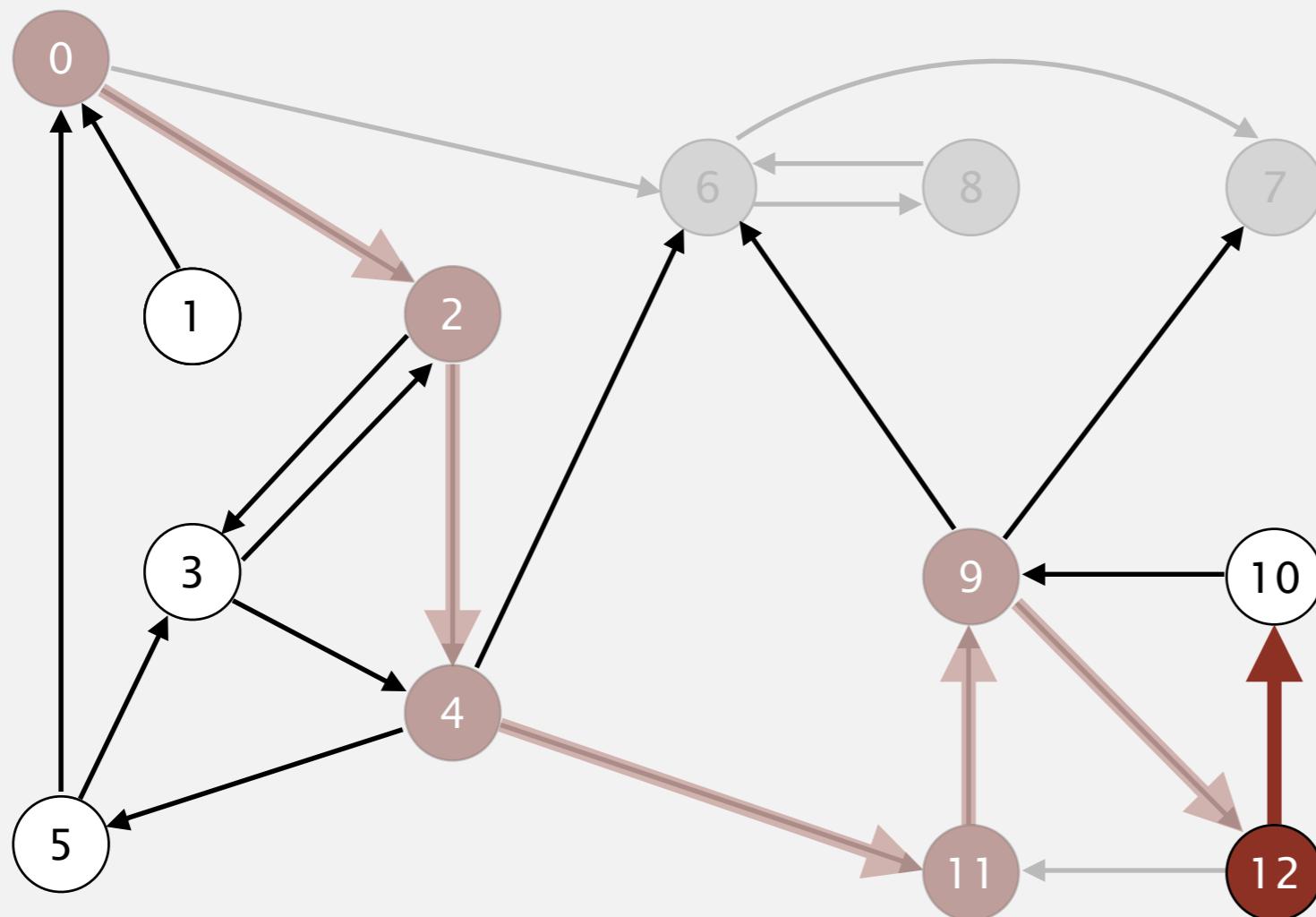
v	marked[]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	T
10	F
11	T
12	T

visit 12: check 11 and check 10

# Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in  $G^R$ .

6 7 8



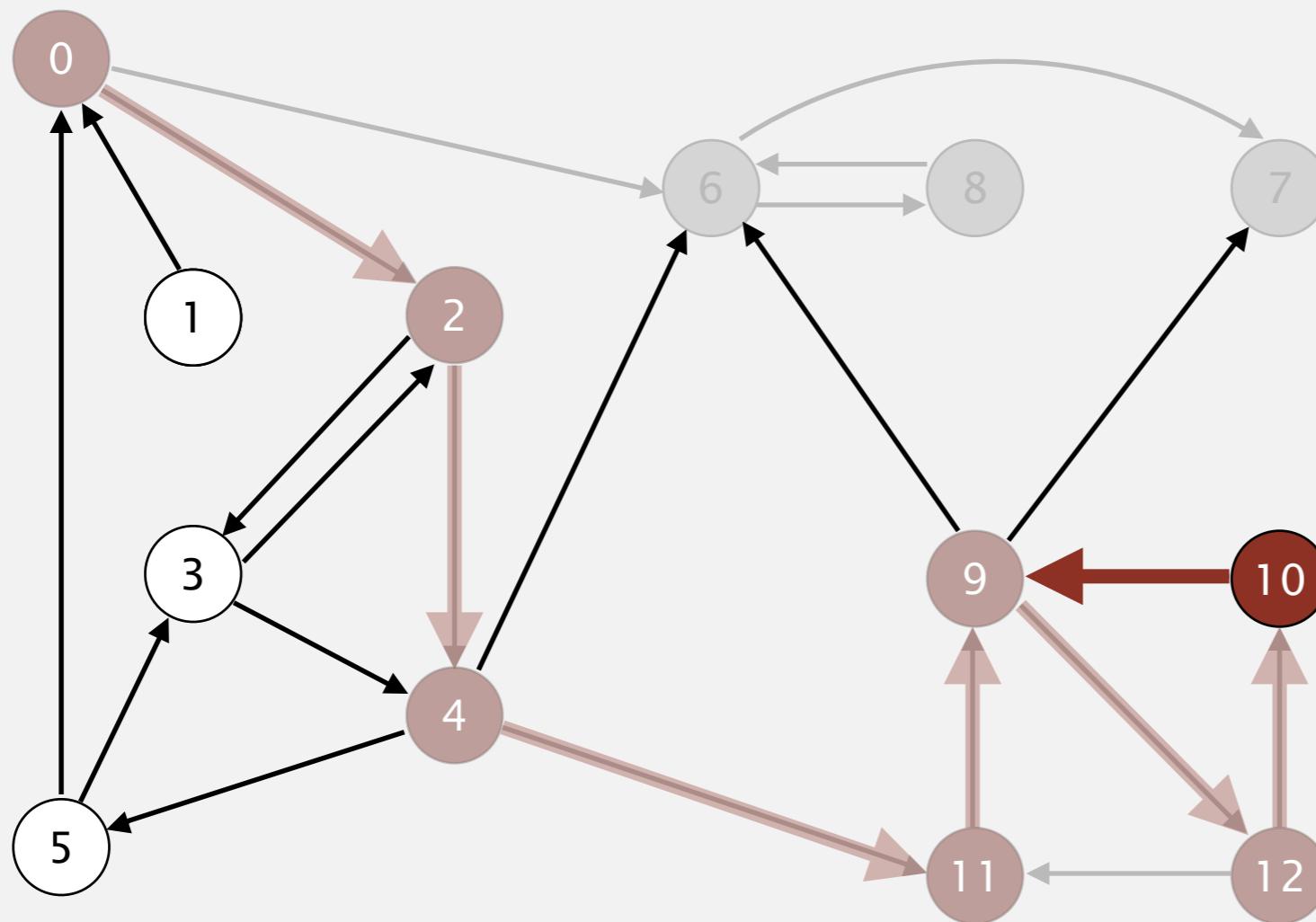
v	marked[]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	T
10	F
11	T
12	T

visit 12: check 11 and check 10

# Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in  $G^R$ .

6 7 8

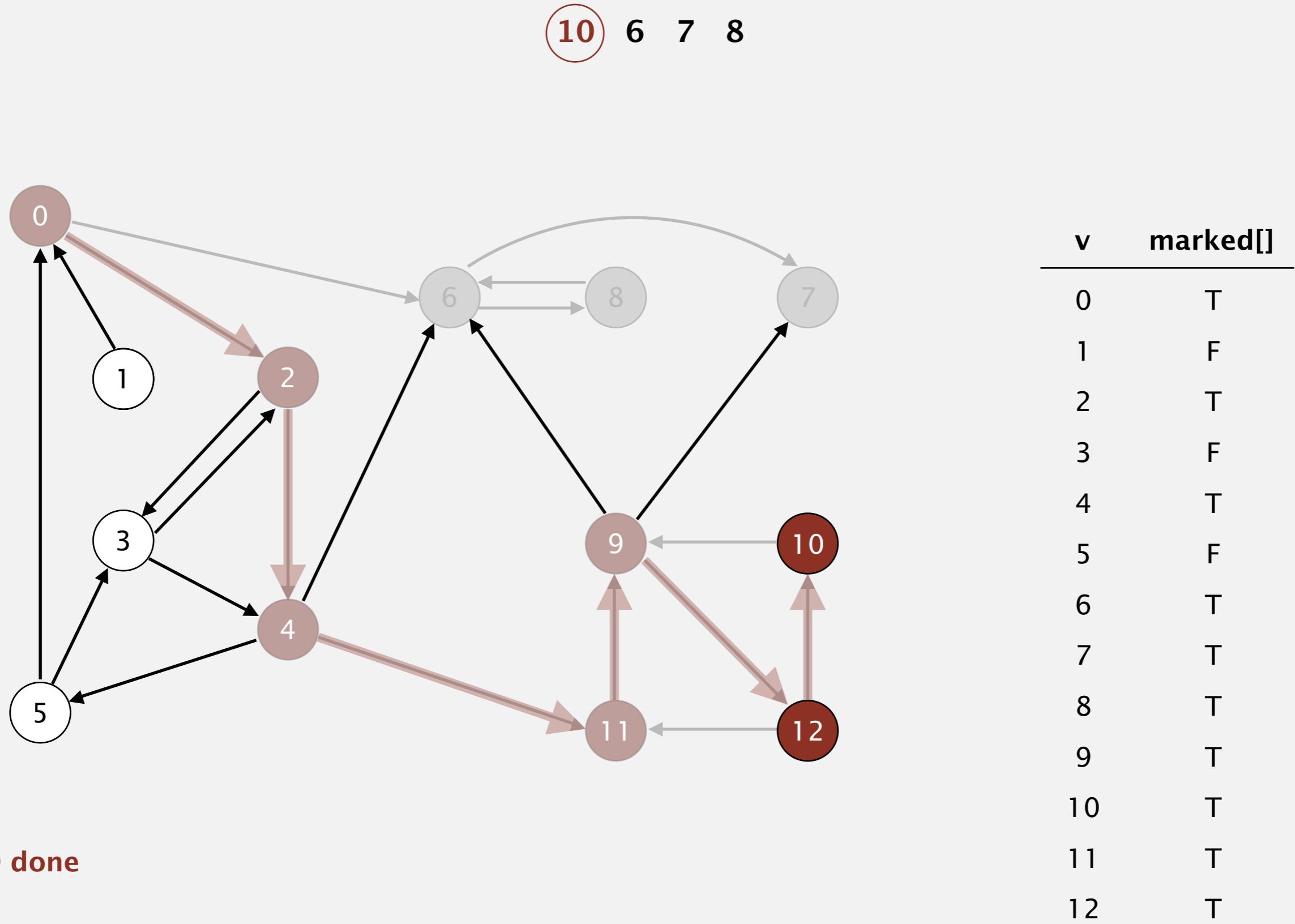


v	marked[]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	T
10	T
11	T
12	T

visit 10: check 9

# Kosaraju-Sharir algorithm demo

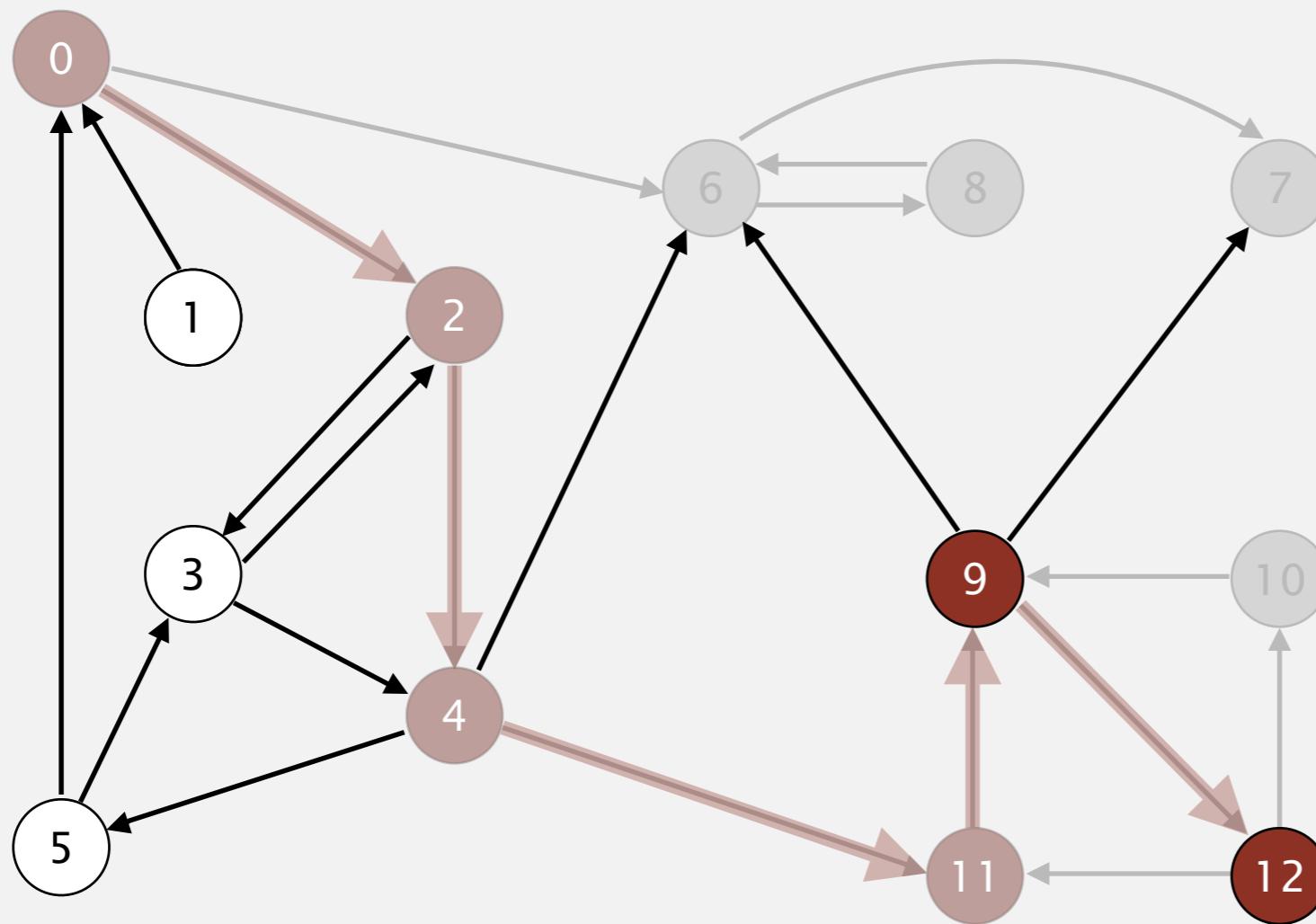
Phase 1. Compute reverse postorder in  $G^R$ .



# Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in  $G^R$ .

12 10 6 7 8



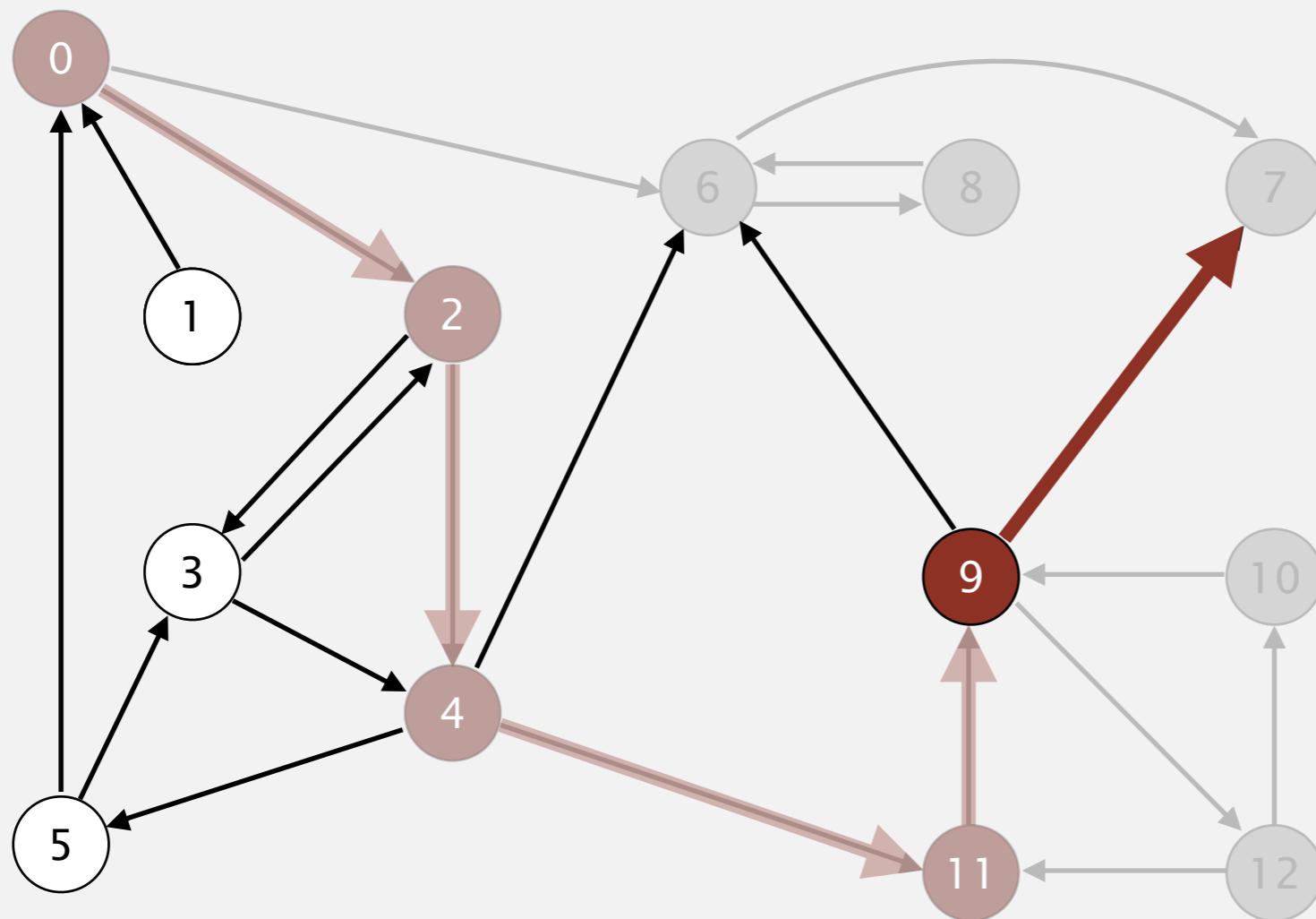
v	marked[]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	T
10	T
11	T
12	T

12 done

# Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in  $G^R$ .

12 10 6 7 8



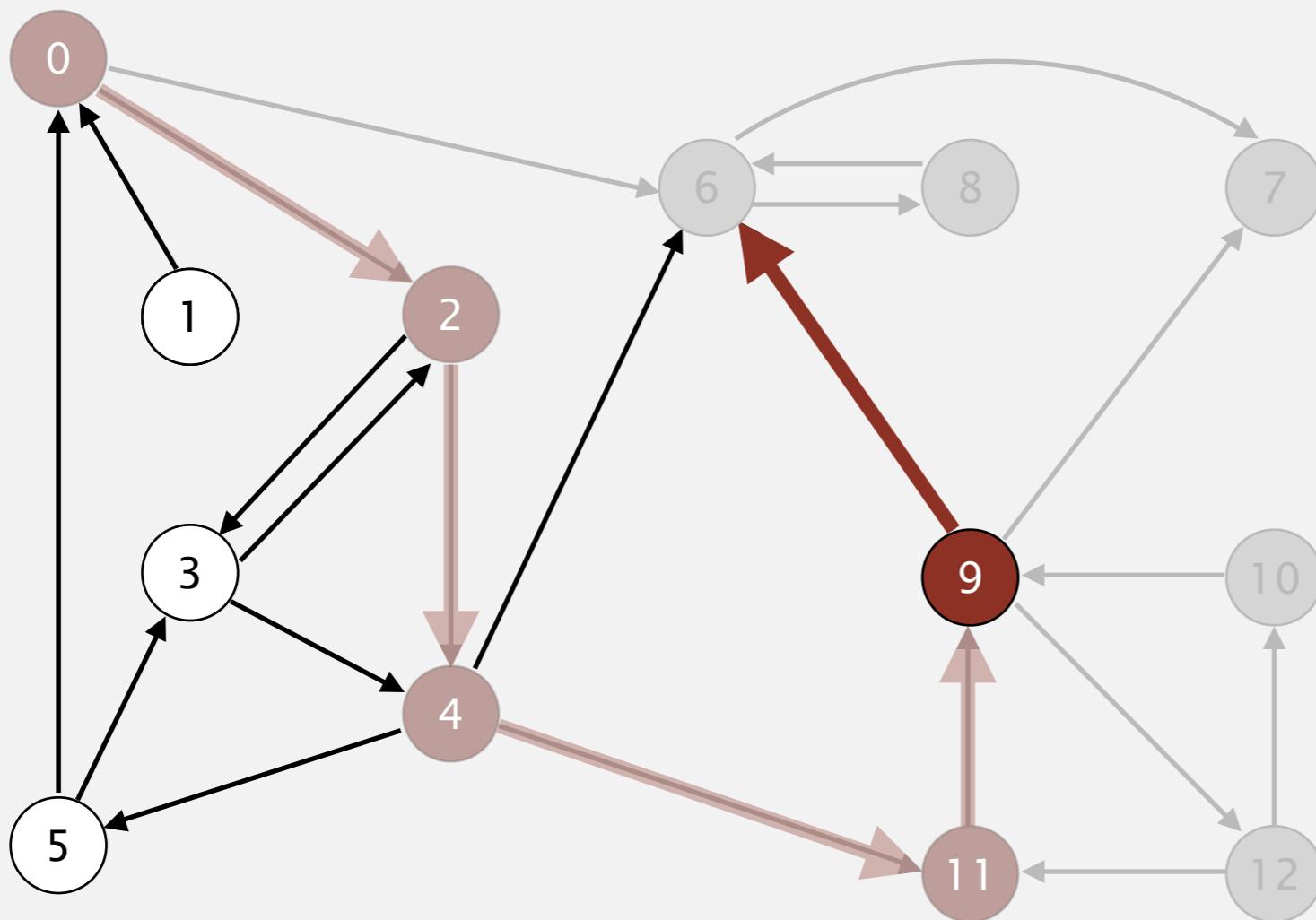
v	marked[]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	T
10	T
11	T
12	T

visit 9: check 12, check 7 and check 6

# Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in  $G^R$ .

12 10 6 7 8



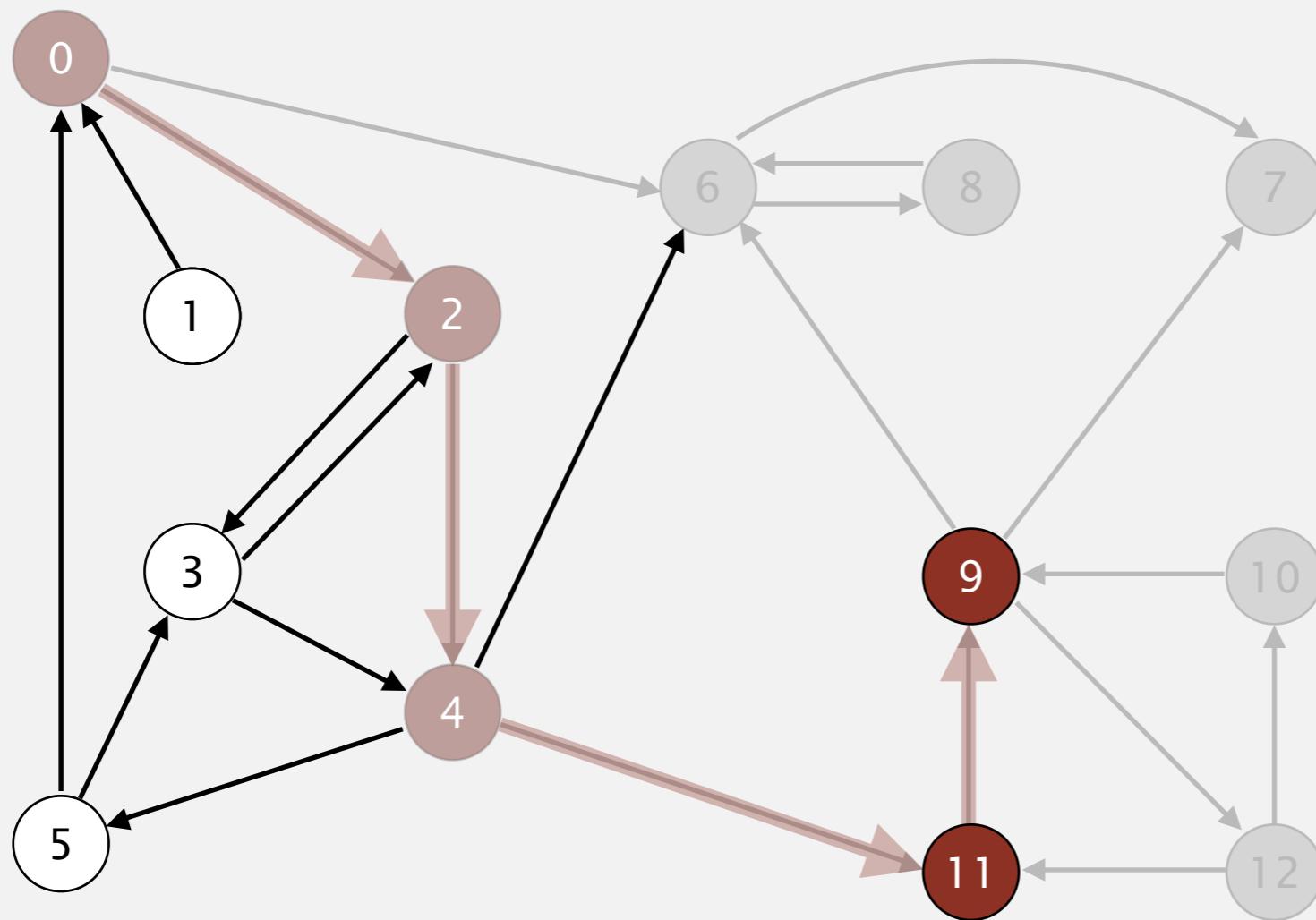
v	marked[]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	T
10	T
11	T
12	T

visit 9: check 12, check 7, and check 6

# Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in  $G^R$ .

9 12 10 6 7 8



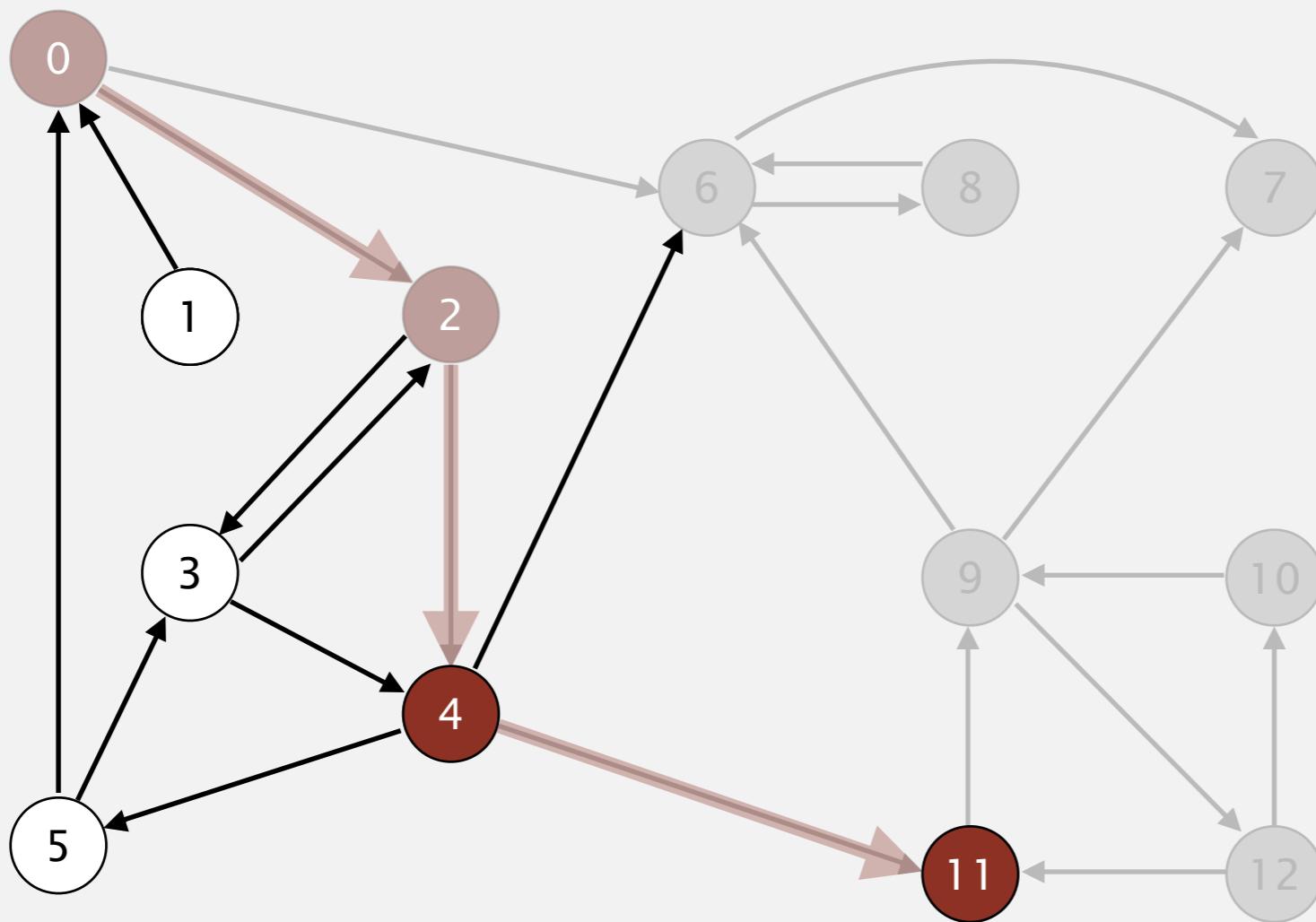
v	marked[]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	T
10	T
11	T
12	T

9 done

# Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in  $G^R$ .

11 9 12 10 6 7 8



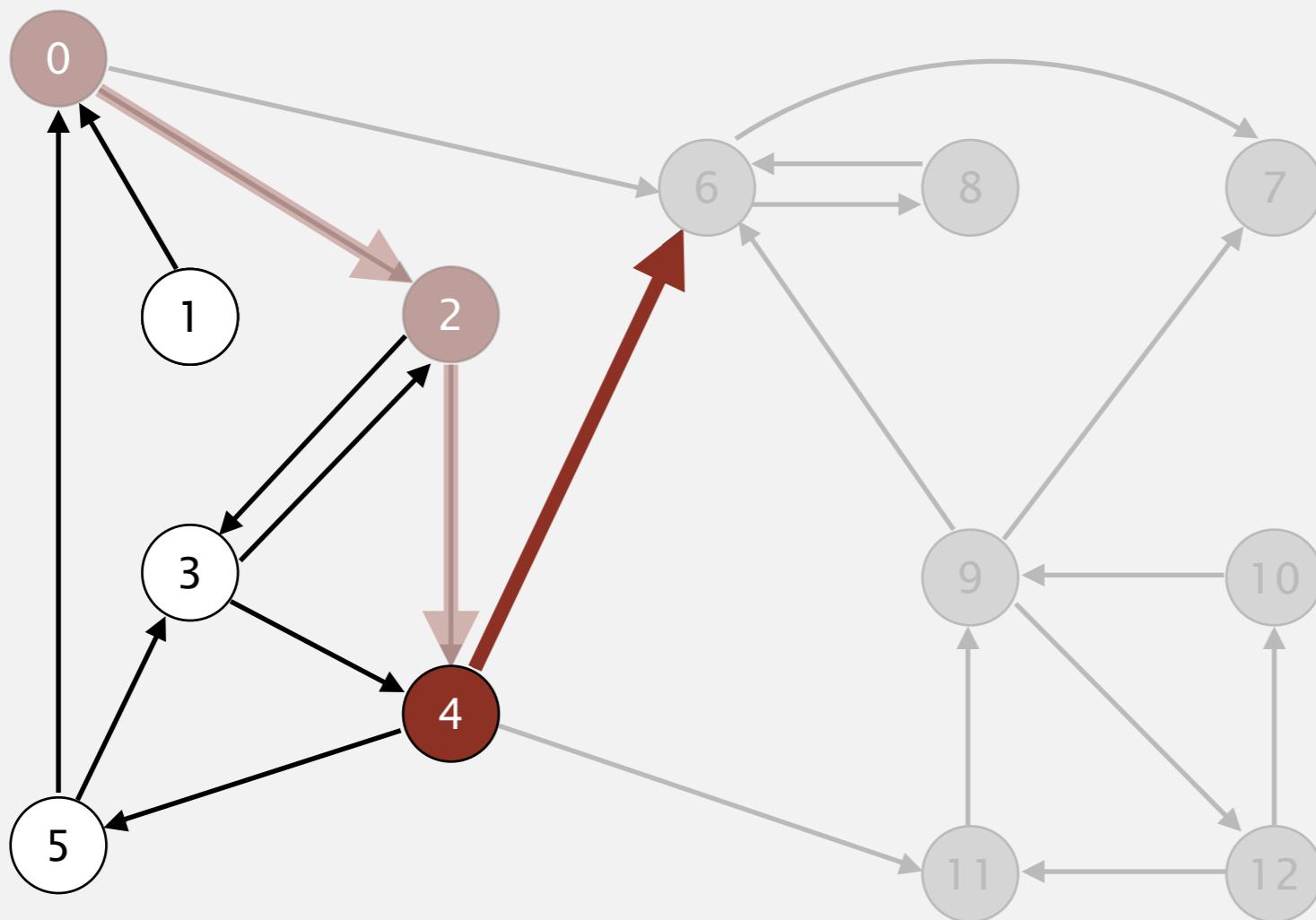
11 done

v	marked[]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	T
10	T
11	T
12	T

# Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in  $G^R$ .

11 9 12 10 6 7 8



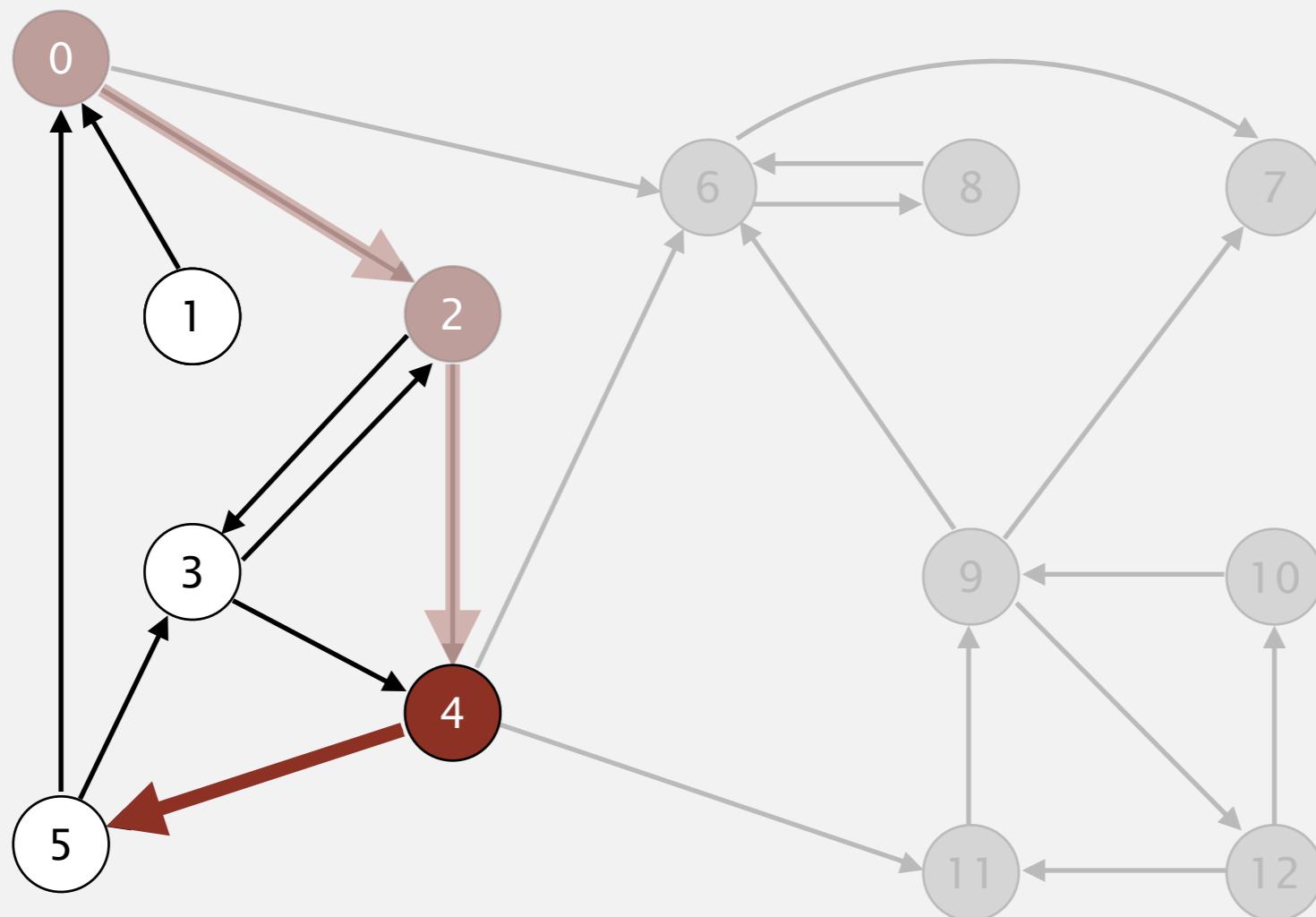
visit 4: check 11, check 6, and check 5

v	marked[]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	T
10	T
11	T
12	T

# Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in  $G^R$ .

11 9 12 10 6 7 8



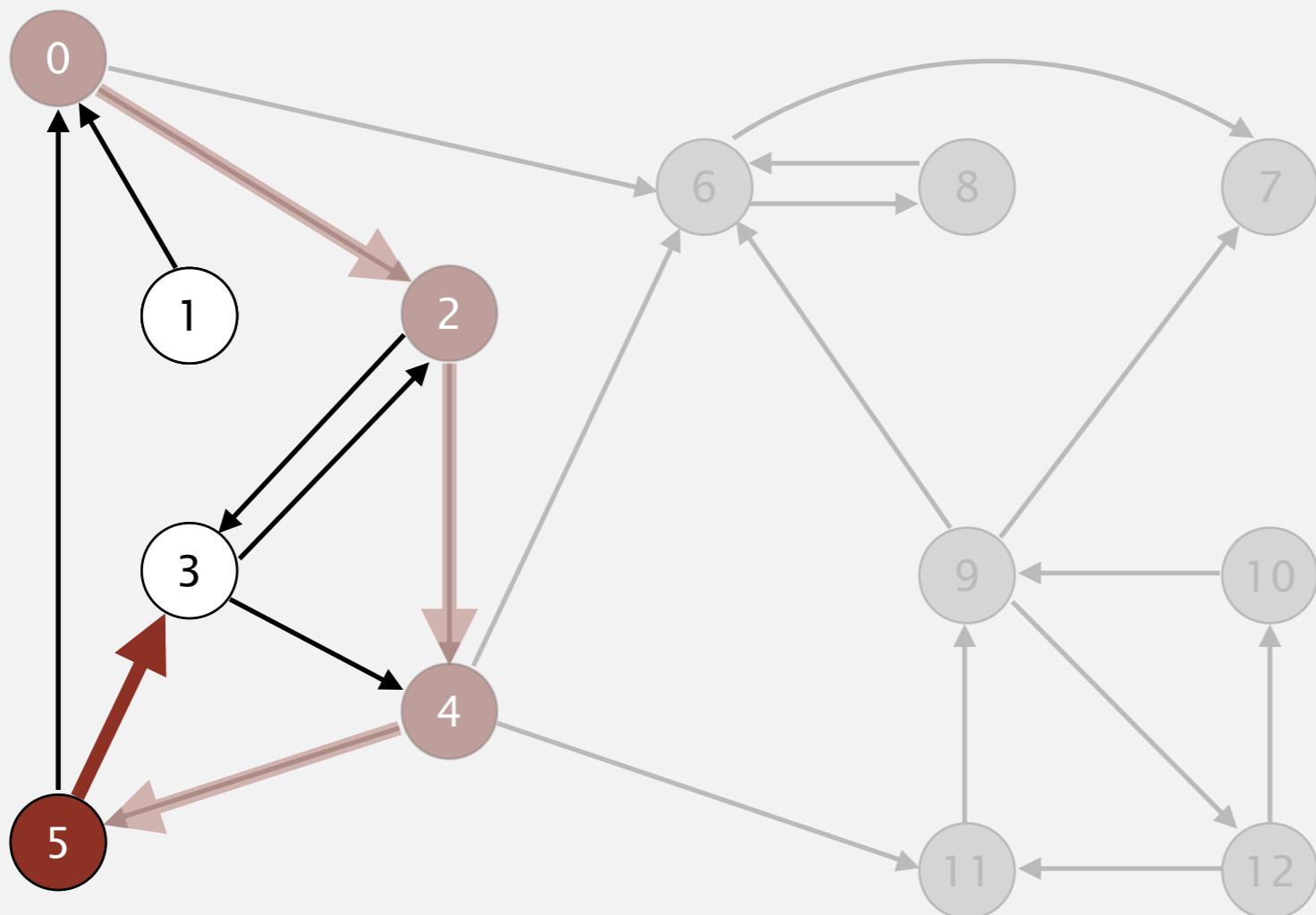
v	marked[]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	T
10	T
11	T
12	T

visit 4: check 11, check 6, and check 5

# Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in  $G^R$ .

11 9 12 10 6 7 8



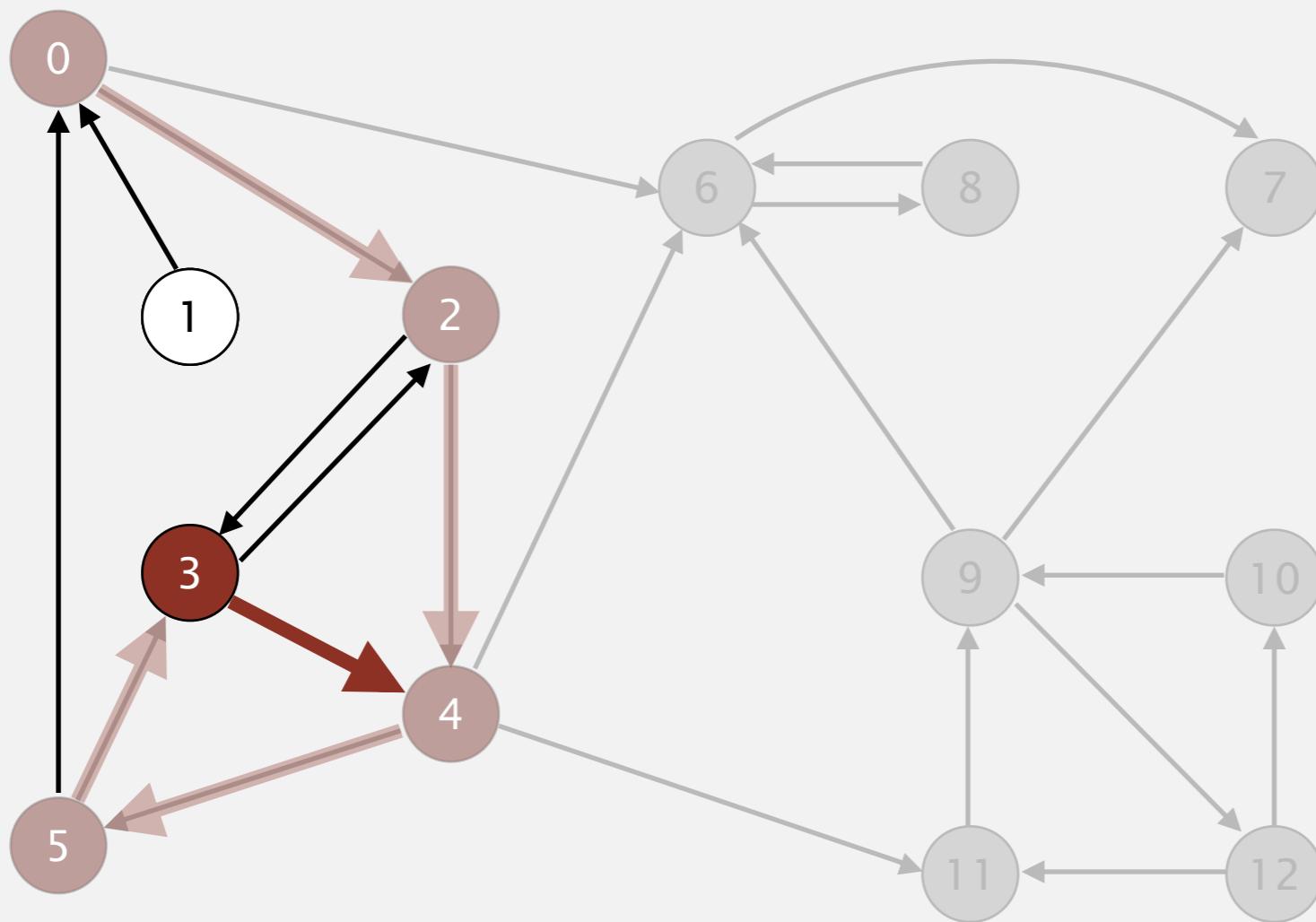
v	marked[]
0	T
1	F
2	T
3	F
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

visit 5: check 3 and check 0

# Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in  $G^R$ .

11 9 12 10 6 7 8



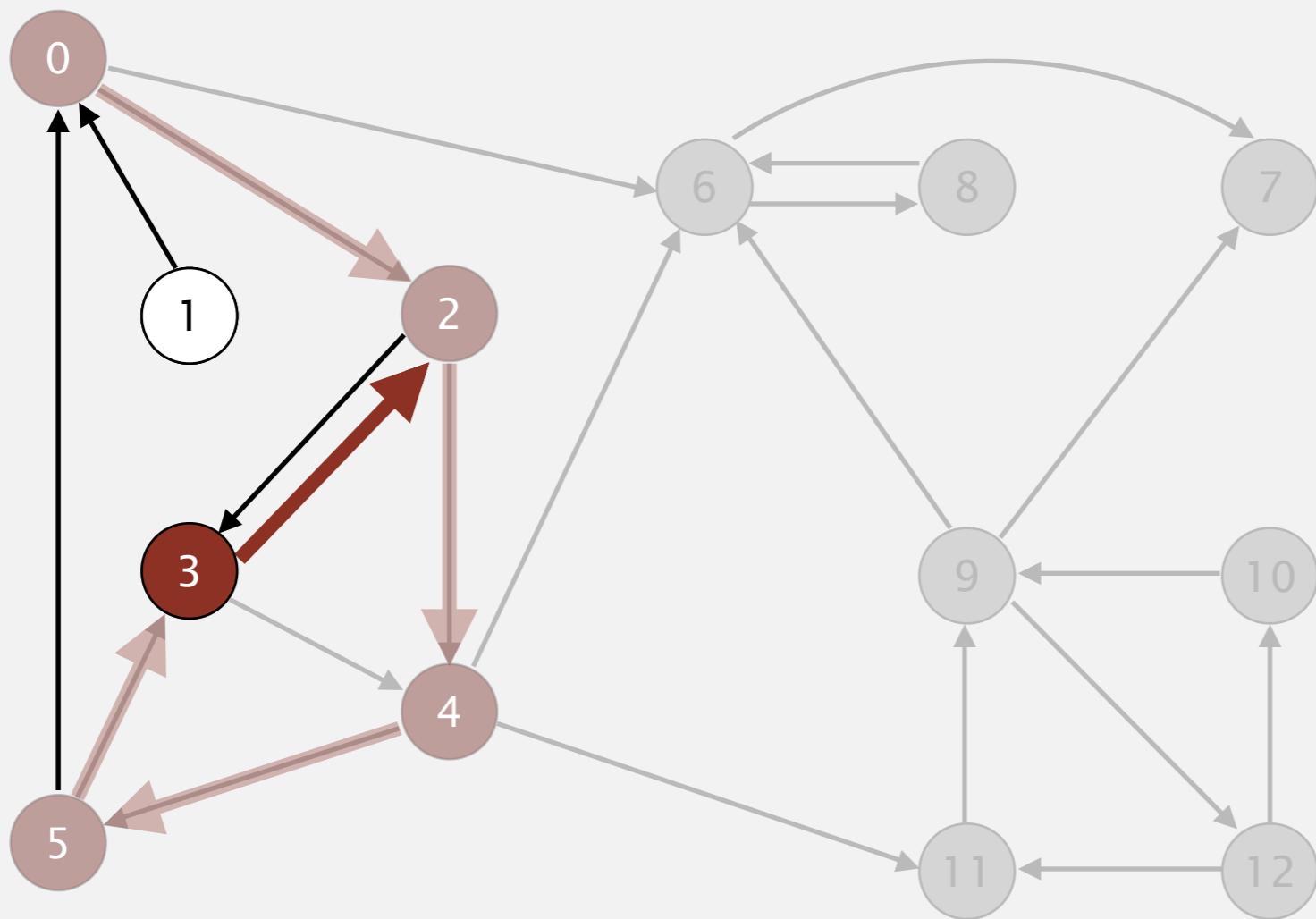
v	marked[]
0	T
1	F
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

visit 3: check 4 and check 2

# Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in  $G^R$ .

11 9 12 10 6 7 8

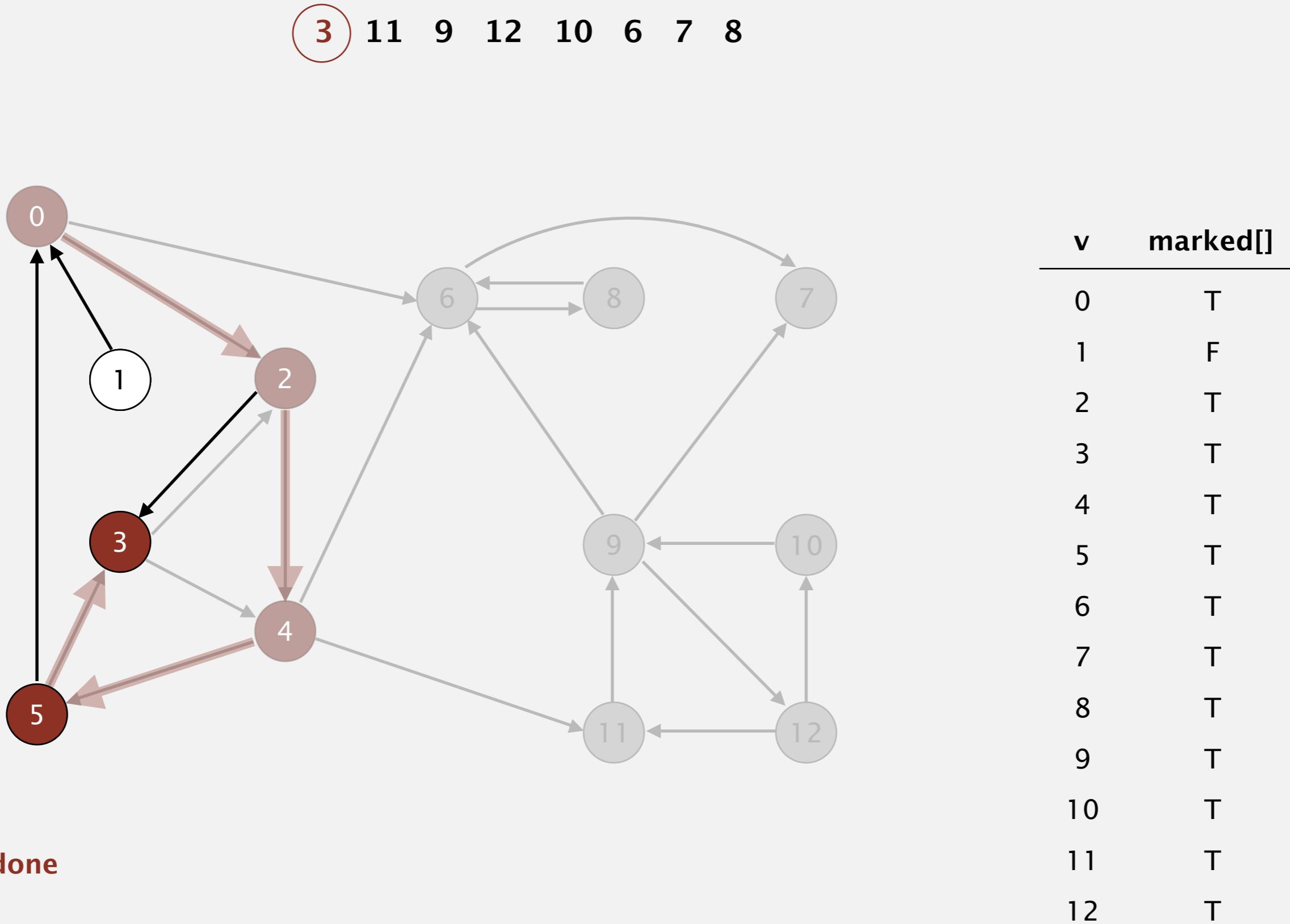


v	marked[]
0	T
1	F
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

visit 3: check 4 and check 2

# Kosaraju-Sharir algorithm demo

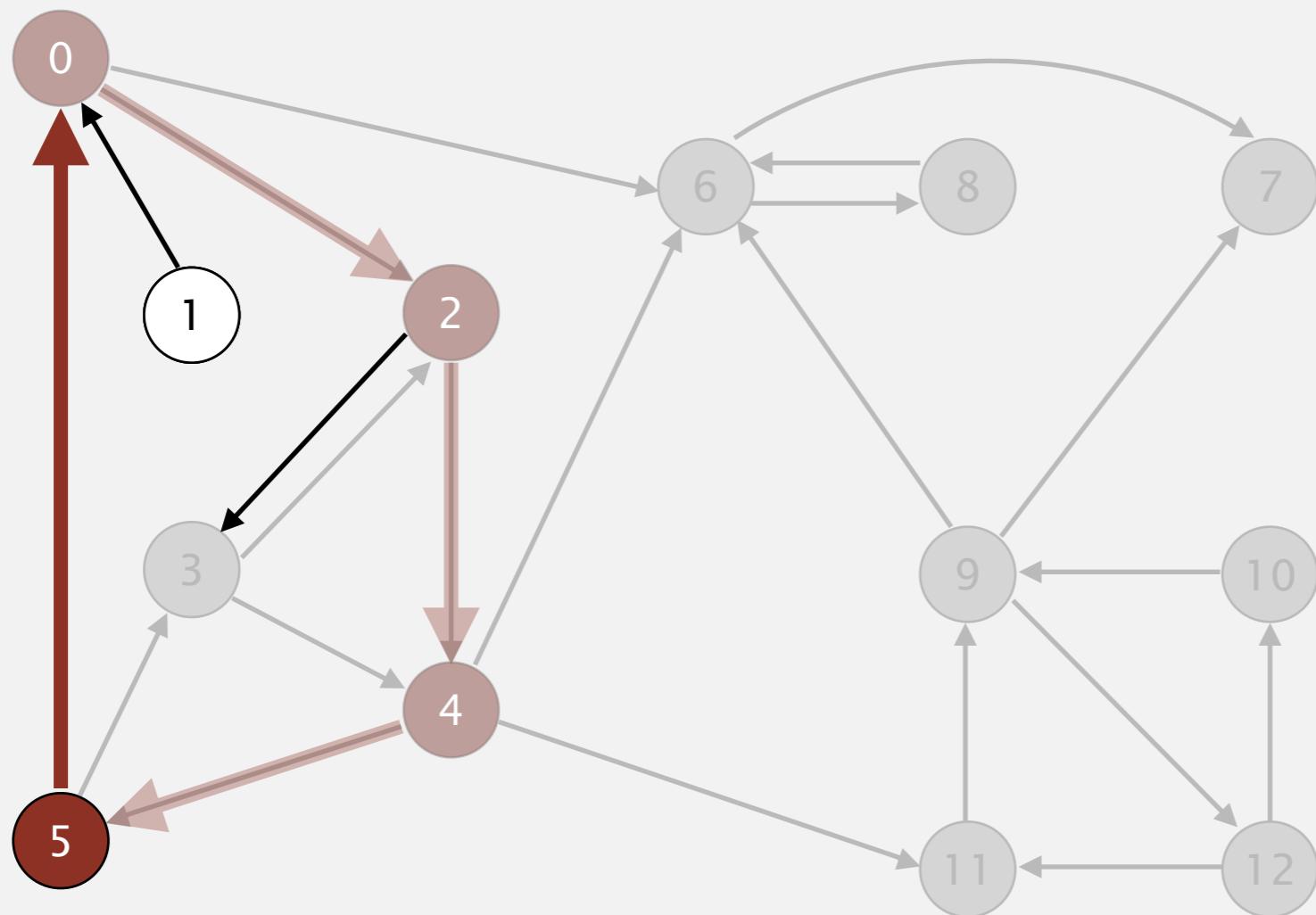
Phase 1. Compute reverse postorder in  $G^R$ .



# Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in  $G^R$ .

3 11 9 12 10 6 7 8



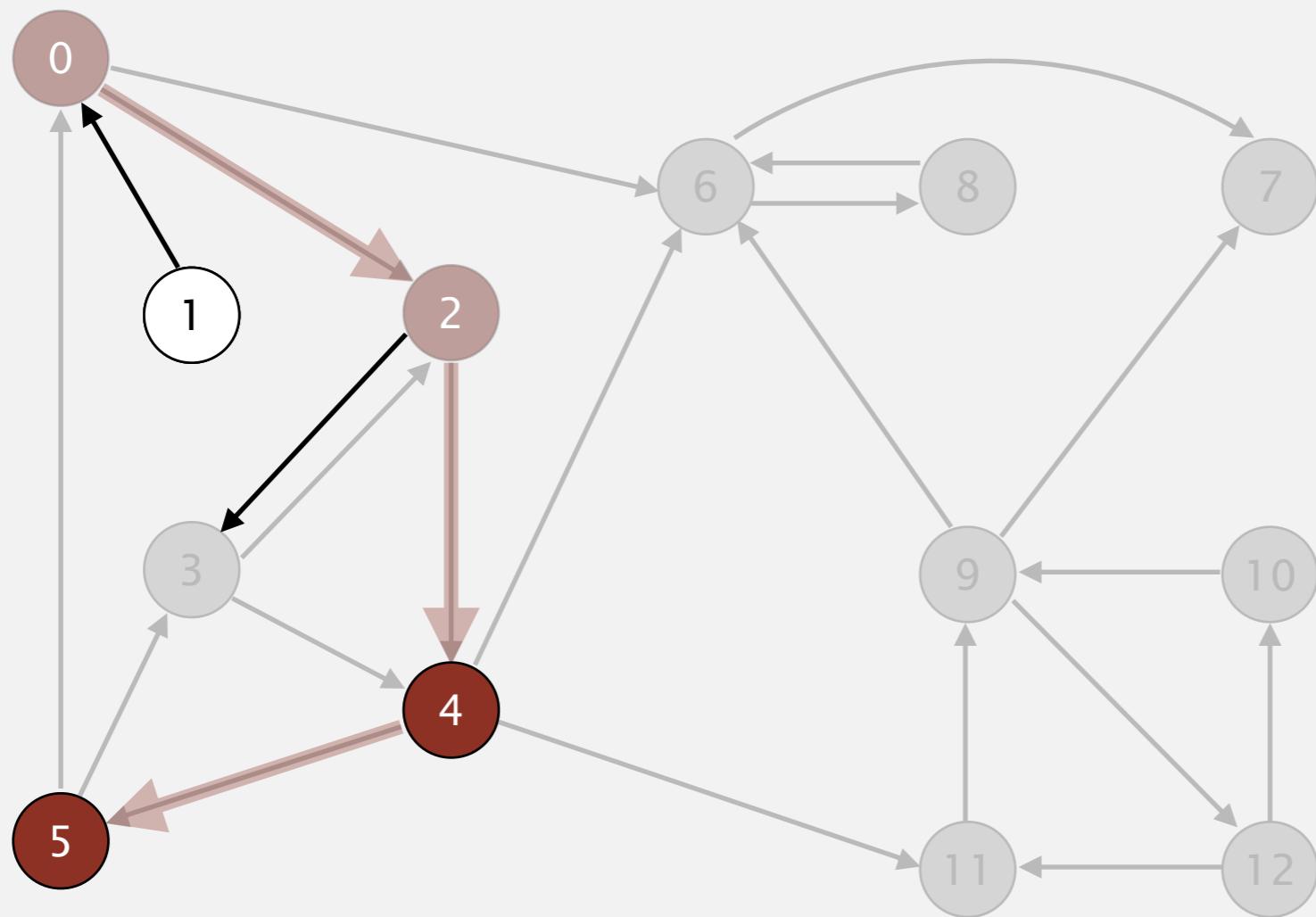
v	marked[]
0	T
1	F
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

visit 5: check 3 and check 0

# Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in  $G^R$ .

5 3 11 9 12 10 6 7 8

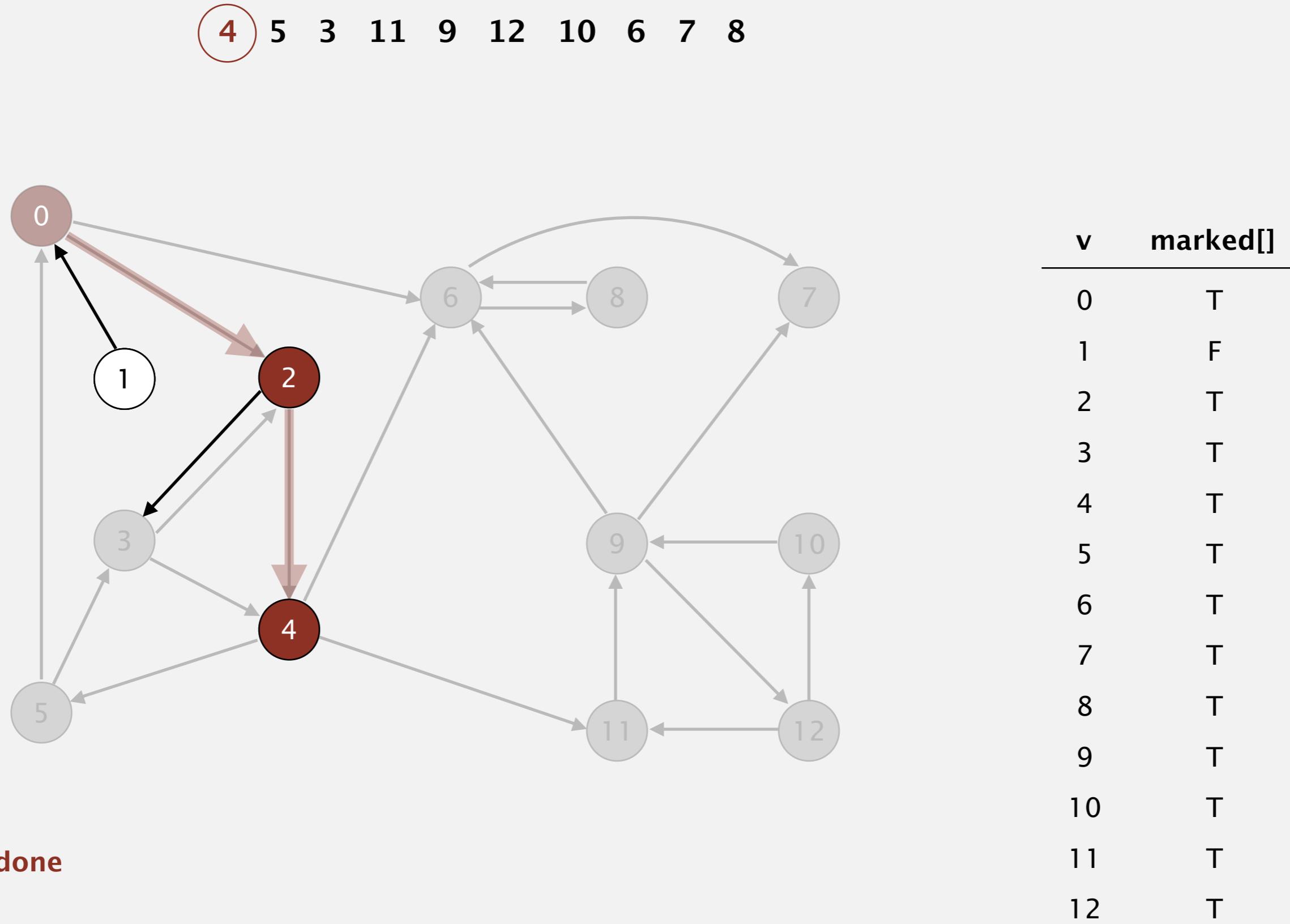


v	marked[]
0	T
1	F
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

5 done

# Kosaraju-Sharir algorithm demo

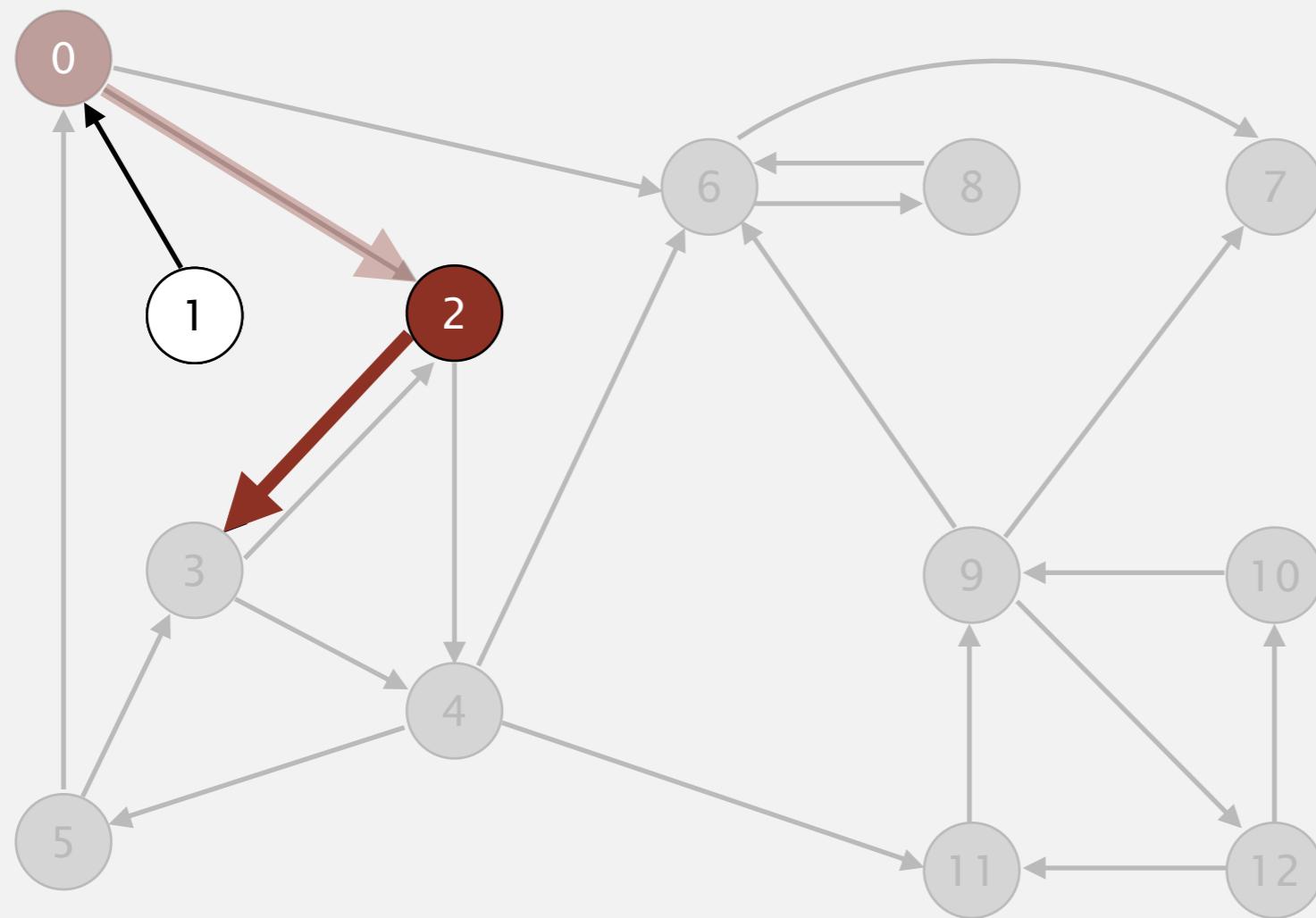
Phase 1. Compute reverse postorder in  $G^R$ .



# Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in  $G^R$ .

4 5 3 11 9 12 10 6 7 8



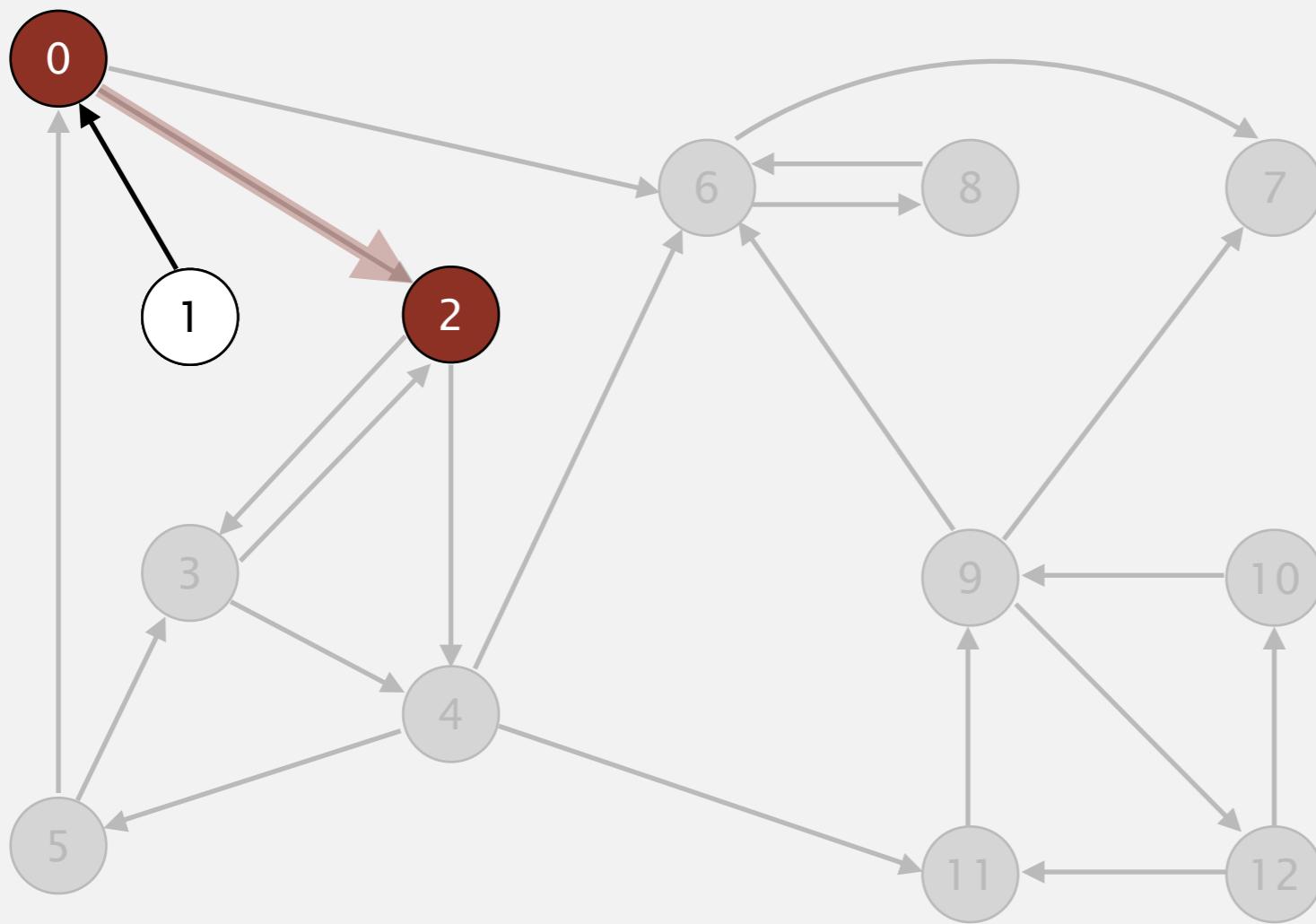
v	marked[]
0	T
1	F
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

visit 2: check 4 and check 3

# Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in  $G^R$ .

2 4 5 3 11 9 12 10 6 7 8

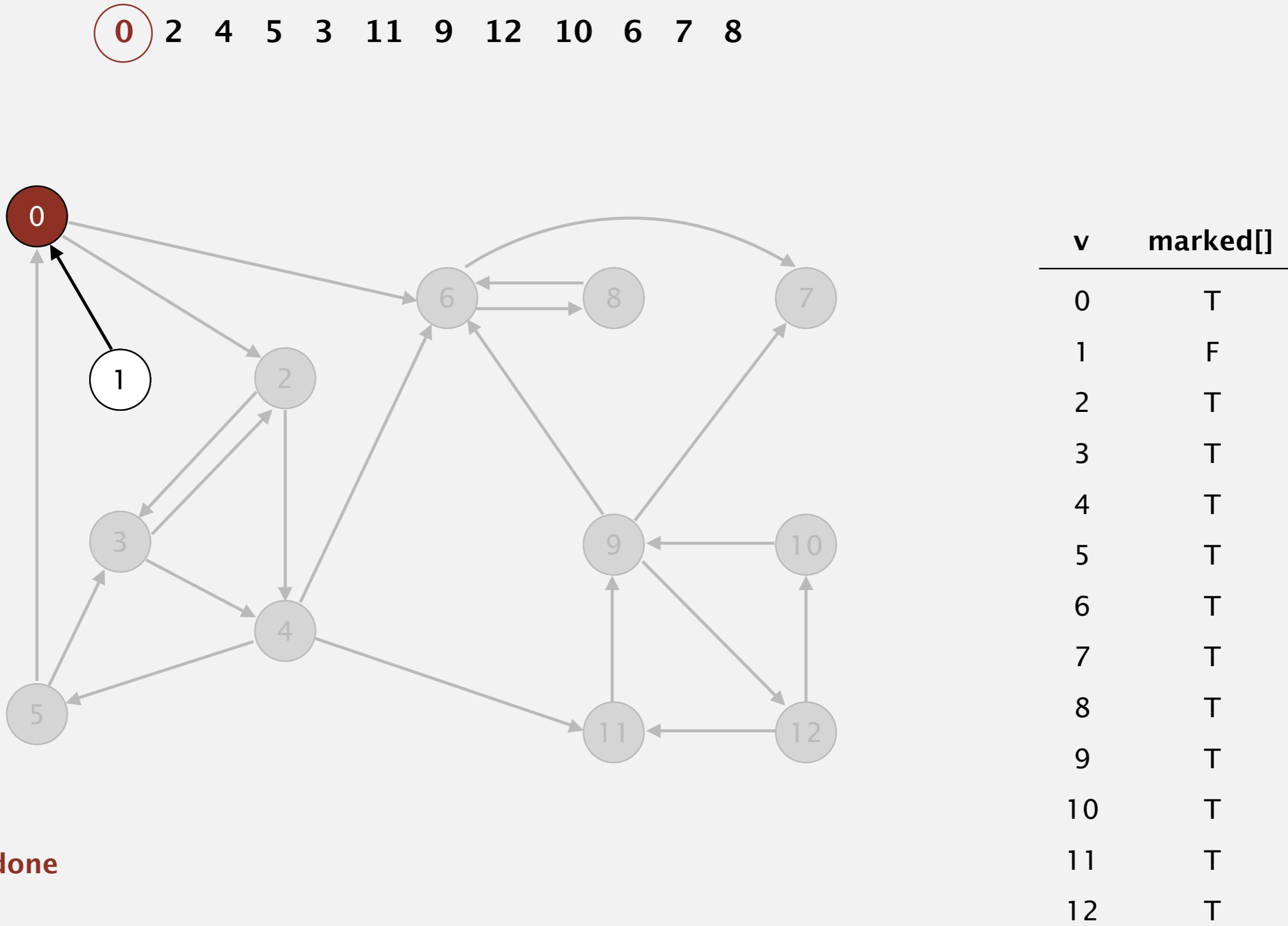


v	marked[]
0	T
1	F
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

2 done

# Kosaraju-Sharir algorithm demo

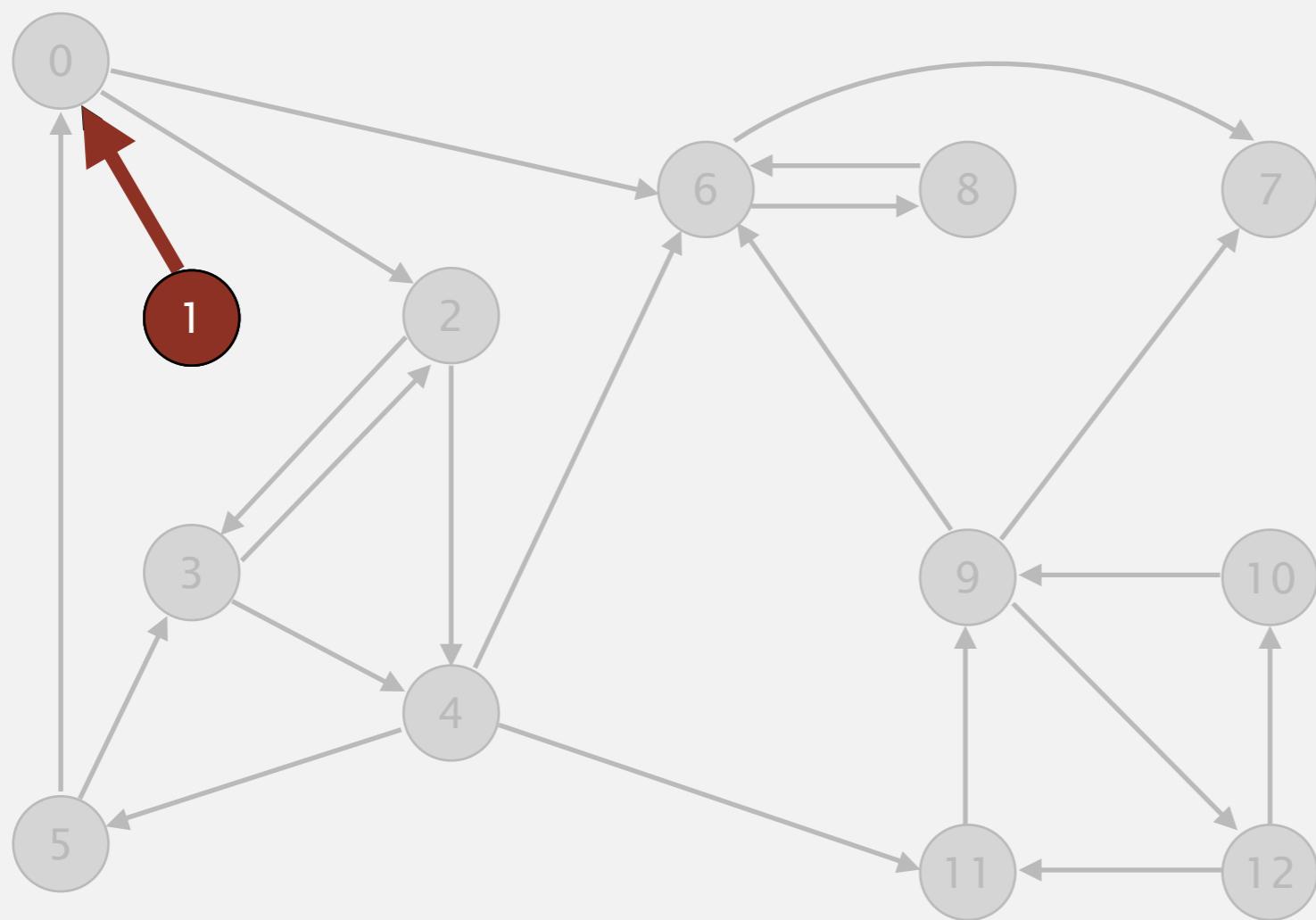
Phase 1. Compute reverse postorder in  $G^R$ .



# Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in  $G^R$ .

0 2 4 5 3 11 9 12 10 6 7 8



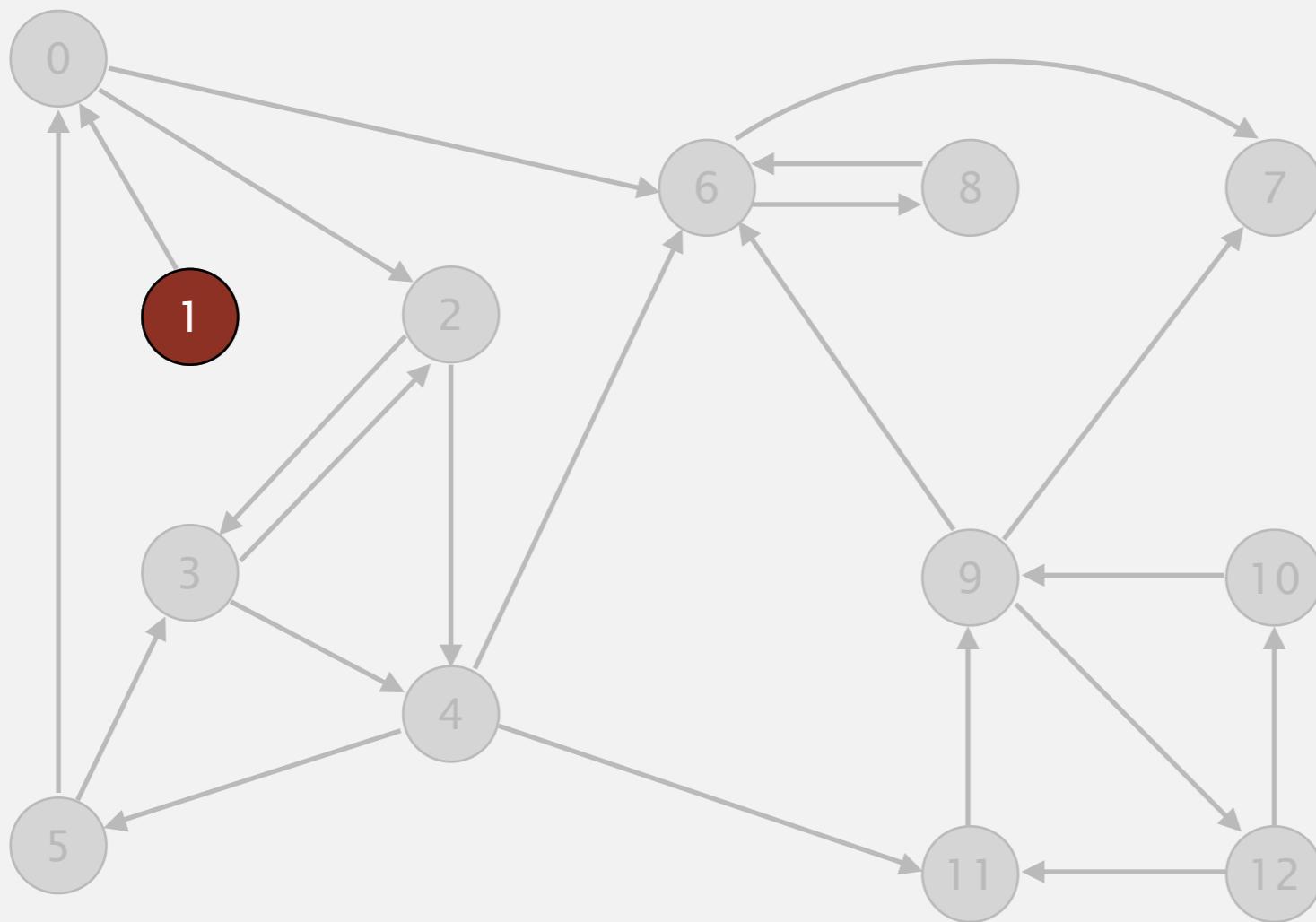
visit 1: check 0

v	marked[]
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

# Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in  $G^R$ .

① 0 2 4 5 3 11 9 12 10 6 7 8



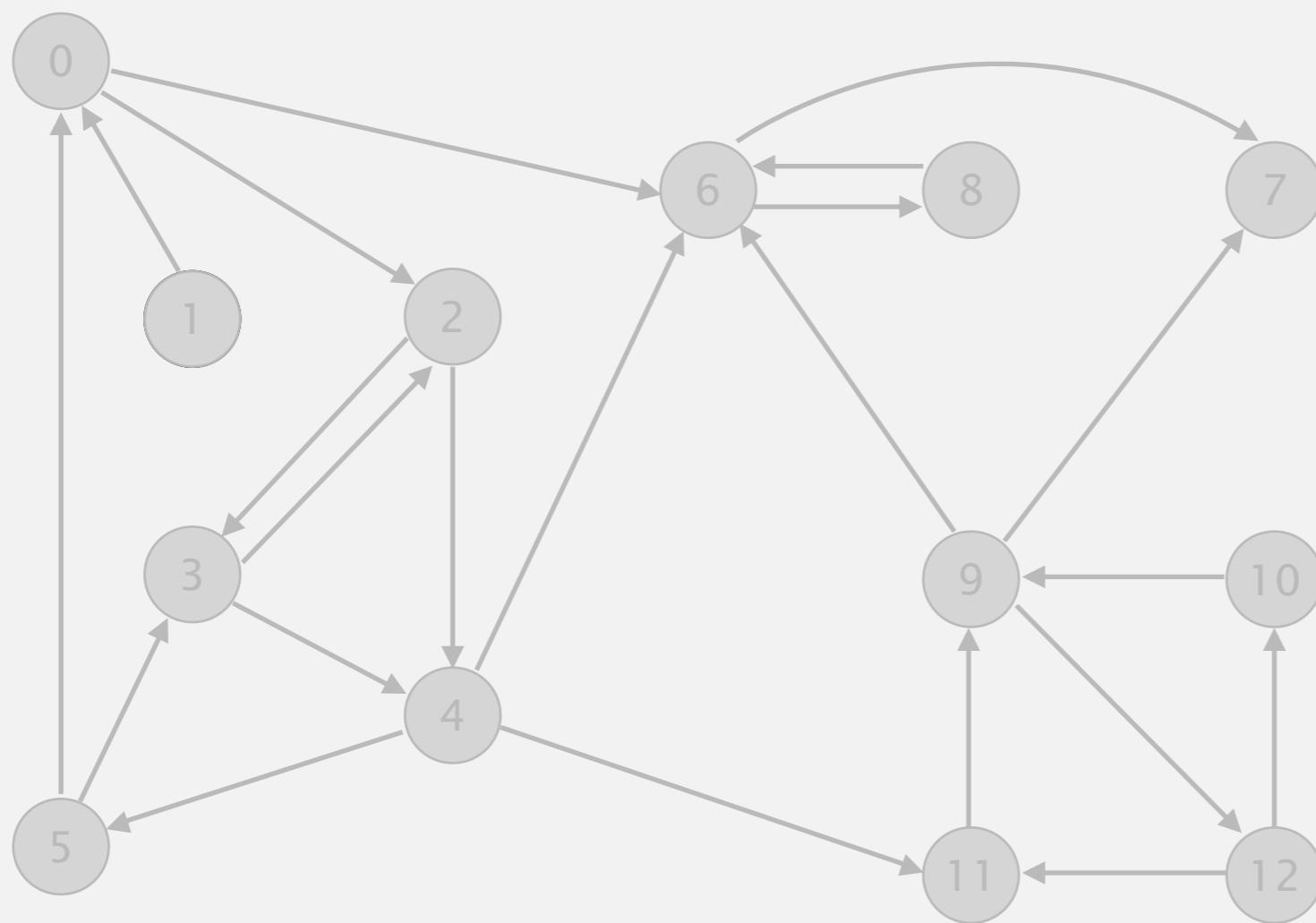
v	marked[]
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

1 done

# Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in  $G^R$ .

1 0 2 4 5 3 11 9 12 10 6 7 8



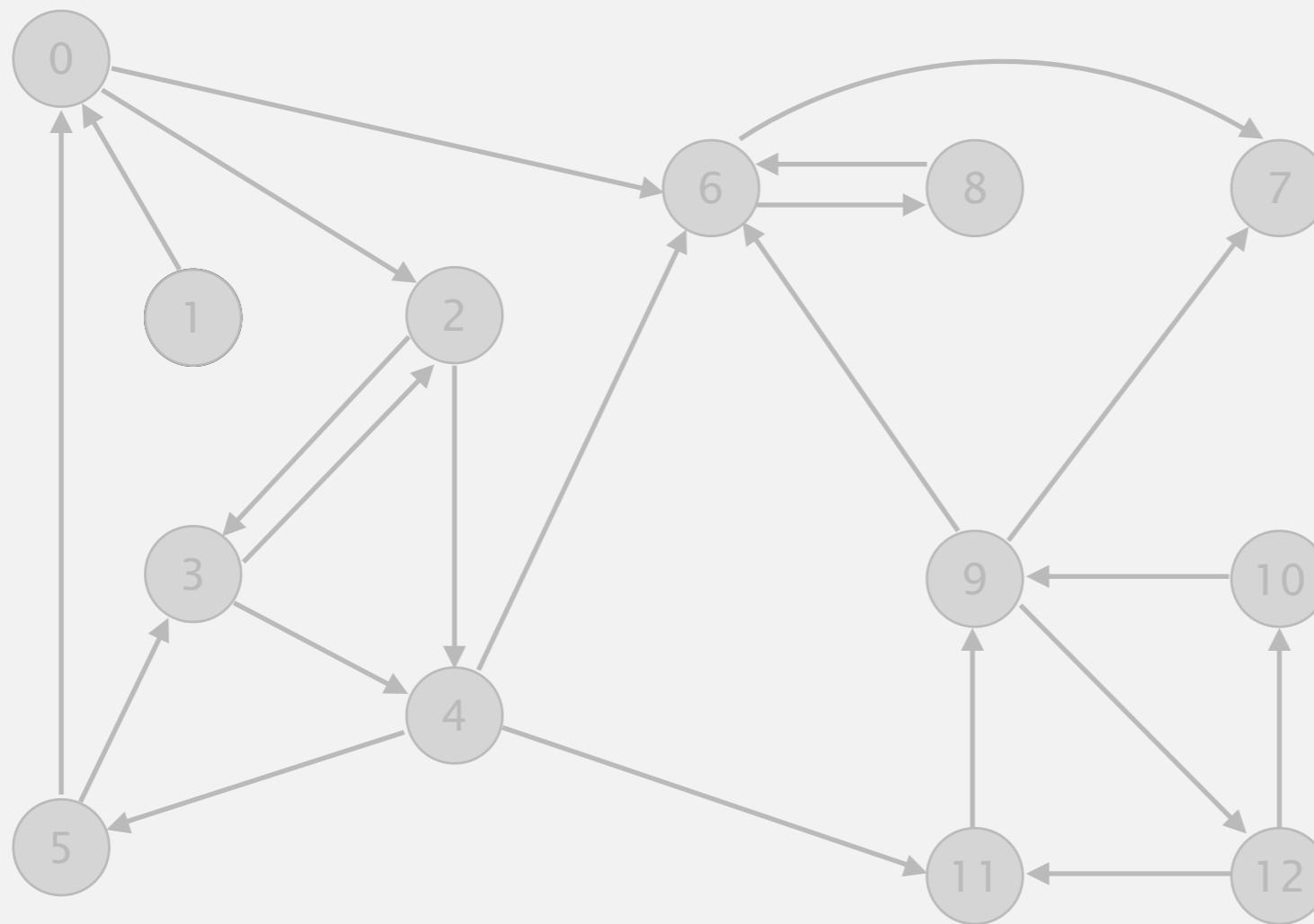
v	marked[]
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

check 2 3 4 5 6 7 8 9 10 11 12

# Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in  $G^R$ .

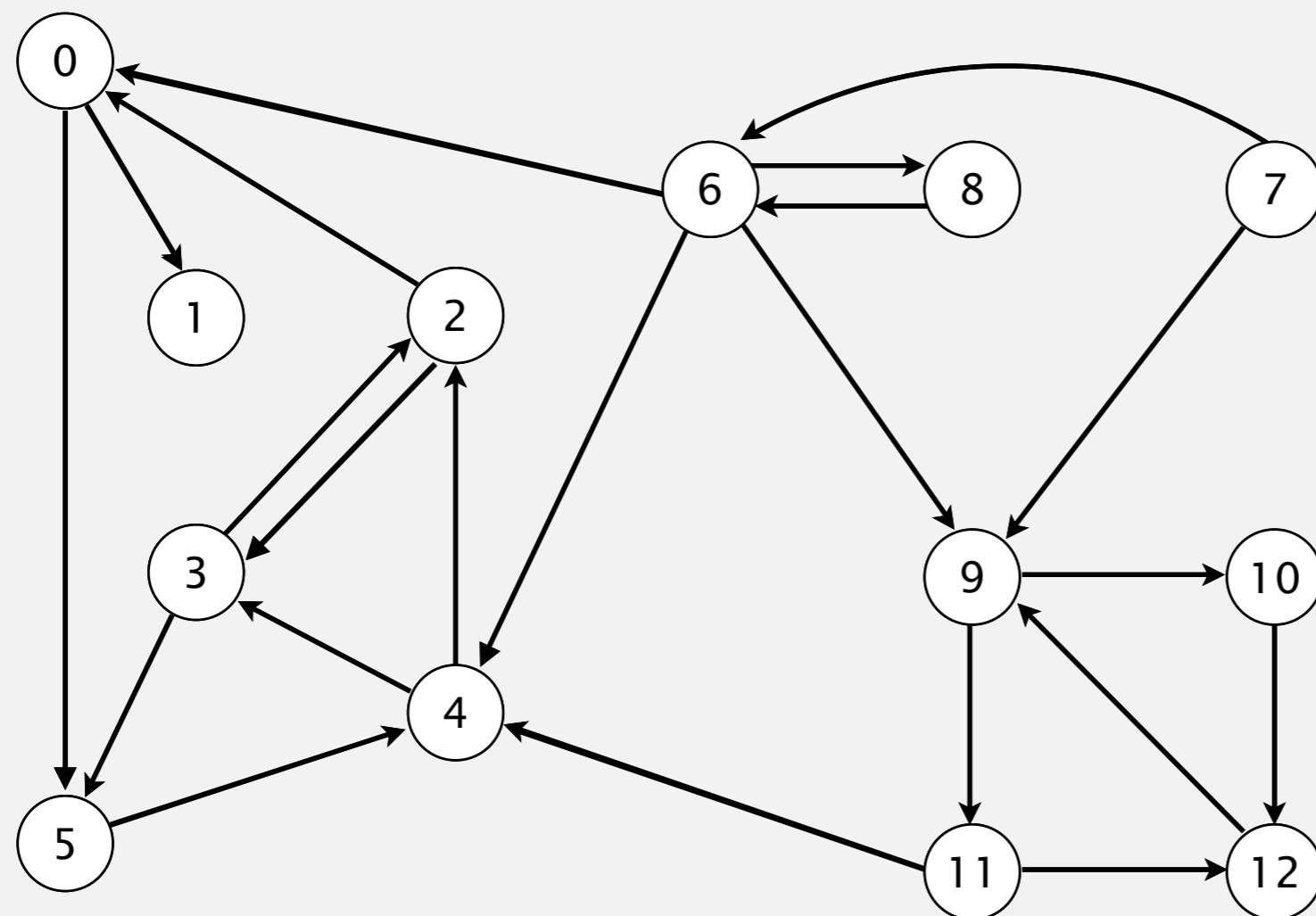
1 0 2 4 5 3 11 9 12 10 6 7 8



reverse digraph  $G^R$

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

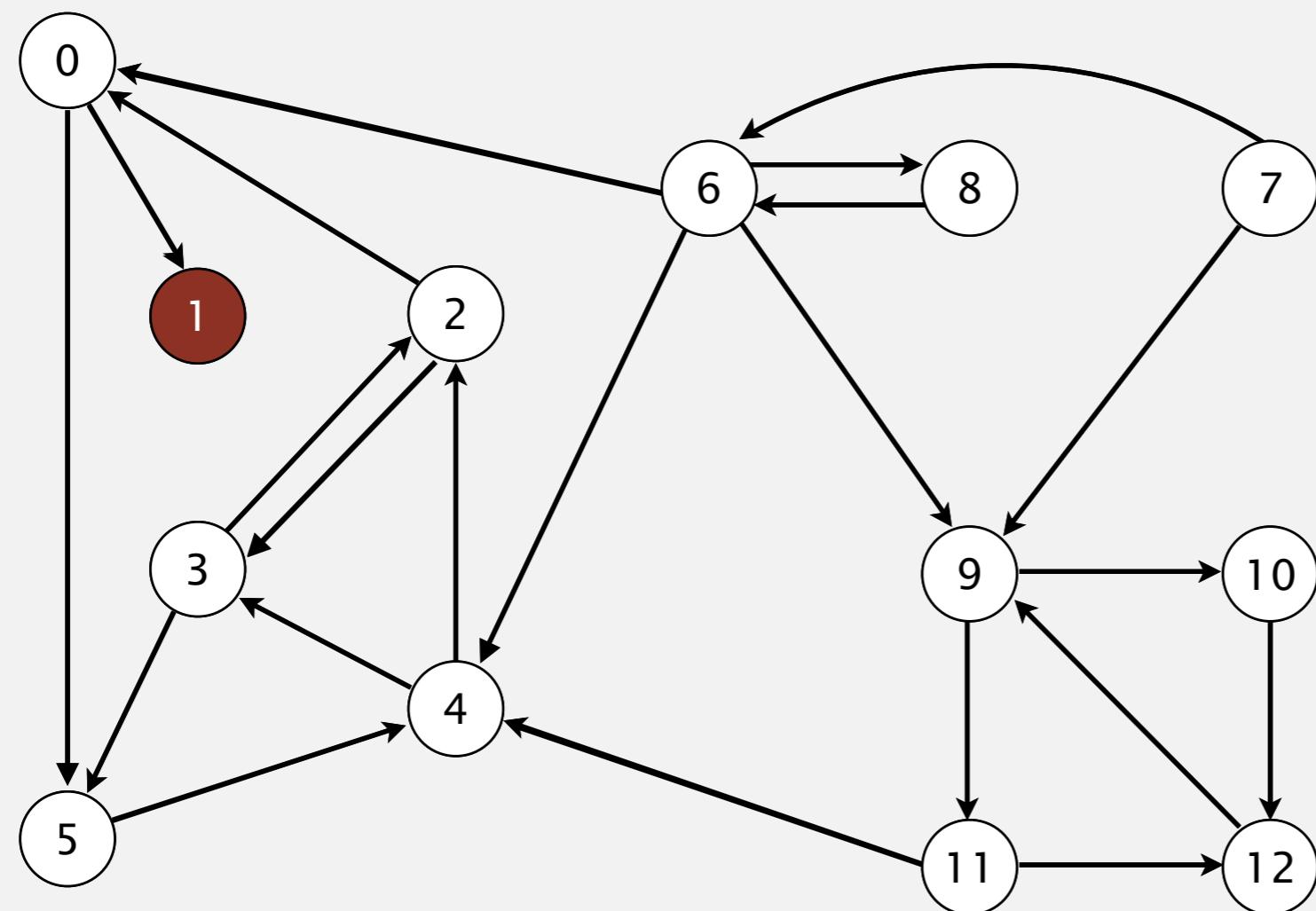


original digraph  $G$

v	id[]
0	-
1	-
2	-
3	-
4	-
5	-
6	-
7	-
8	-
9	-
10	-
11	-
12	-

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

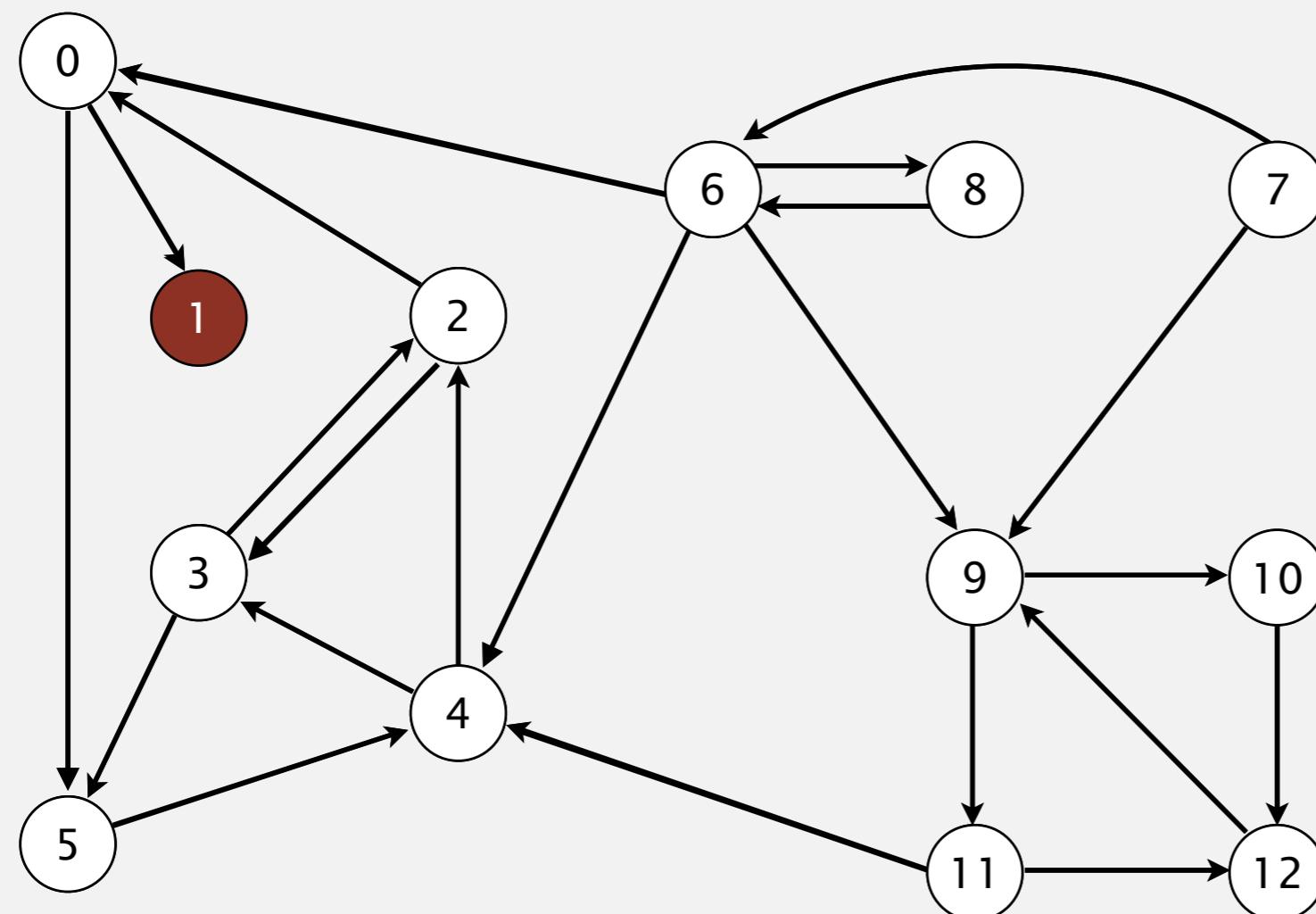


visit 1

v	id[]
0	-
1	0
2	-
3	-
4	-
5	-
6	-
7	-
8	-
9	-
10	-
11	-
12	-

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .



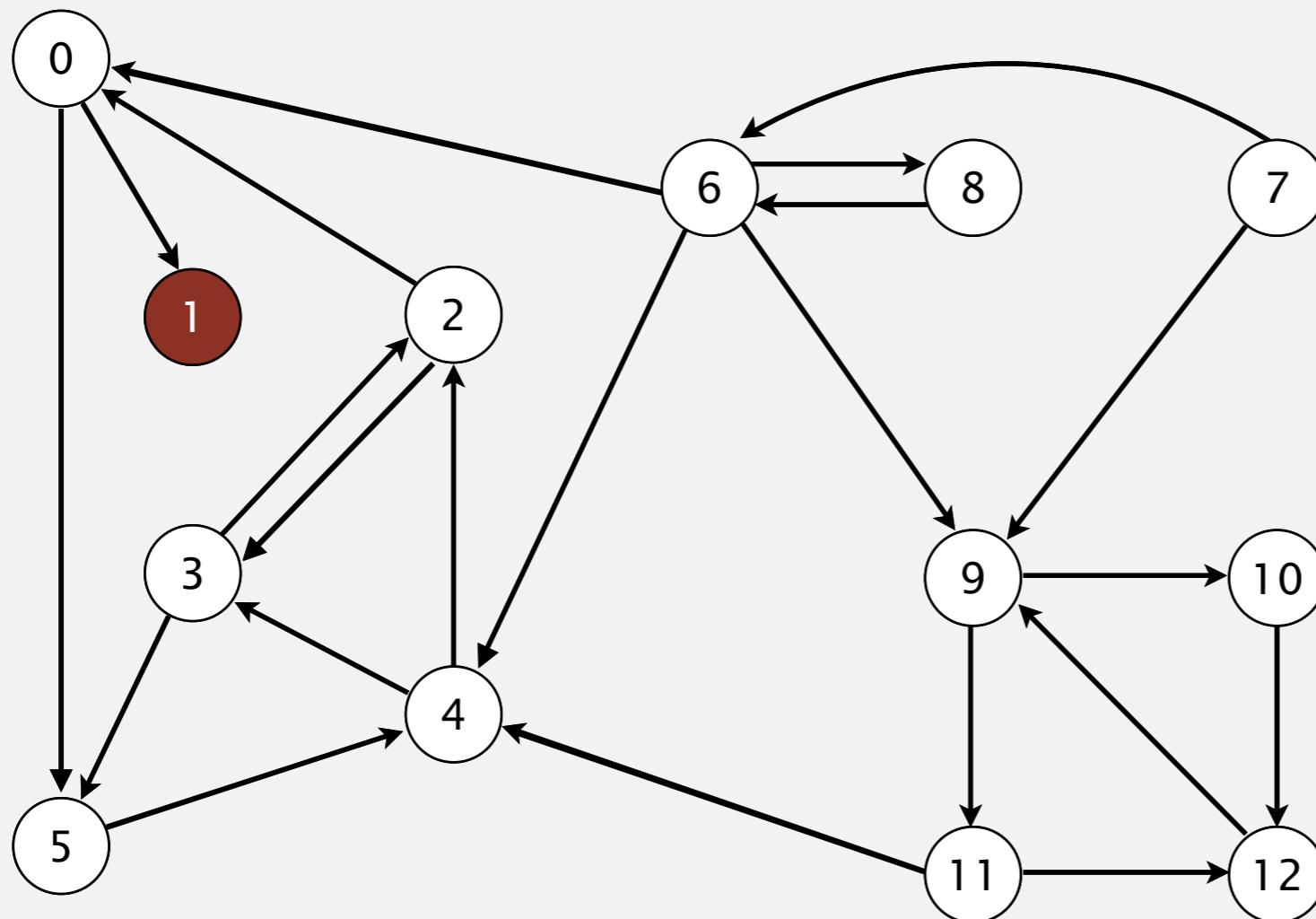
v	id[]
0	-
1	0
2	-
3	-
4	-
5	-
6	-
7	-
8	-
9	-
10	-
11	-
12	-

1 done

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

1 0 2 4 5 3 11 9 12 10 6 7 8



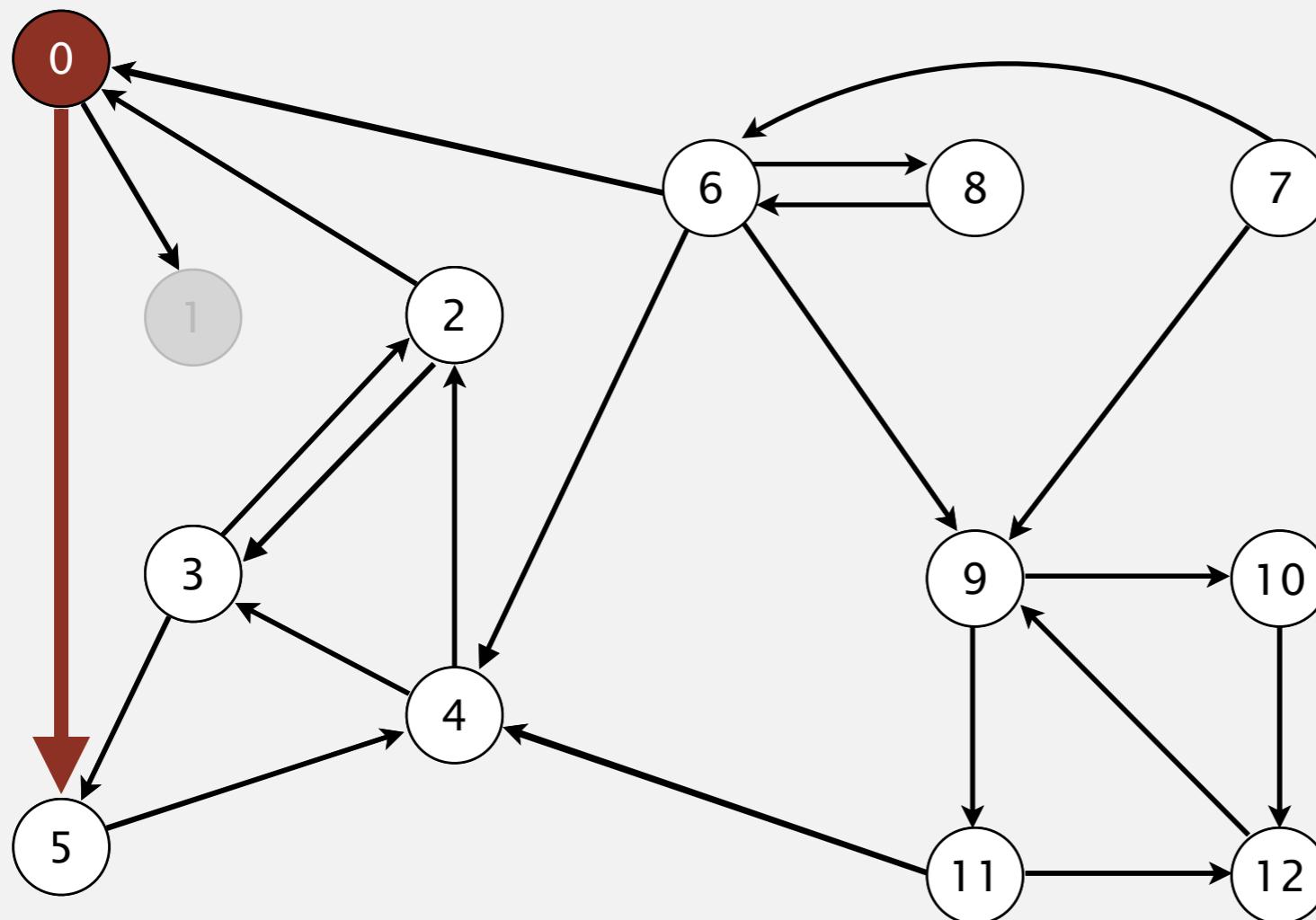
v	id[]
0	-
1	0
2	-
3	-
4	-
5	-
6	-
7	-
8	-
9	-
10	-
11	-
12	-

strong component: 1

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

1 **0** 2 4 5 3 11 9 12 10 6 7 8



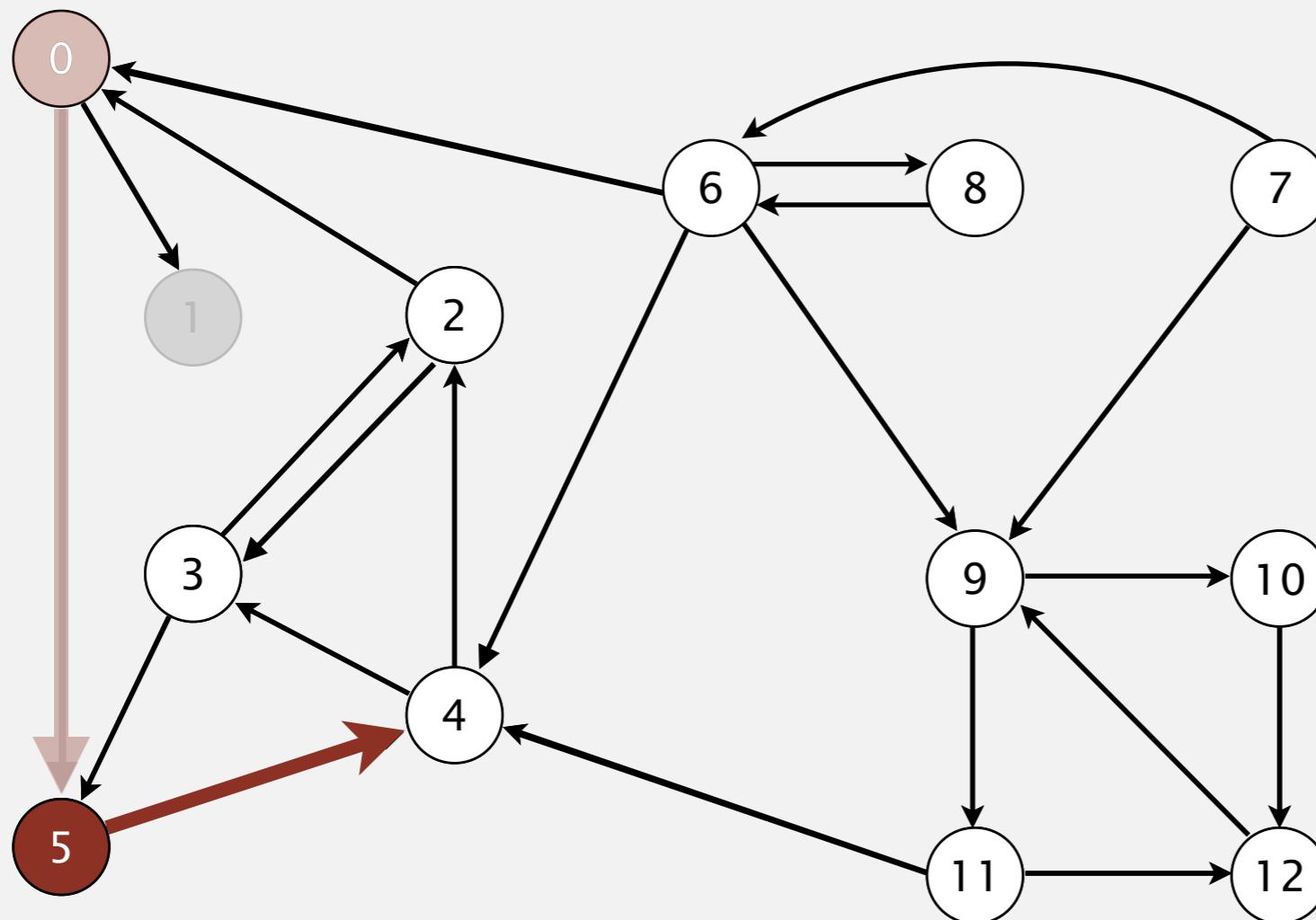
v	id[]
0	1
1	0
2	-
3	-
4	-
5	-
6	-
7	-
8	-
9	-
10	-
11	-
12	-

visit 0: check 5 and check 1

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

1 **0** 2 4 5 3 11 9 12 10 6 7 8

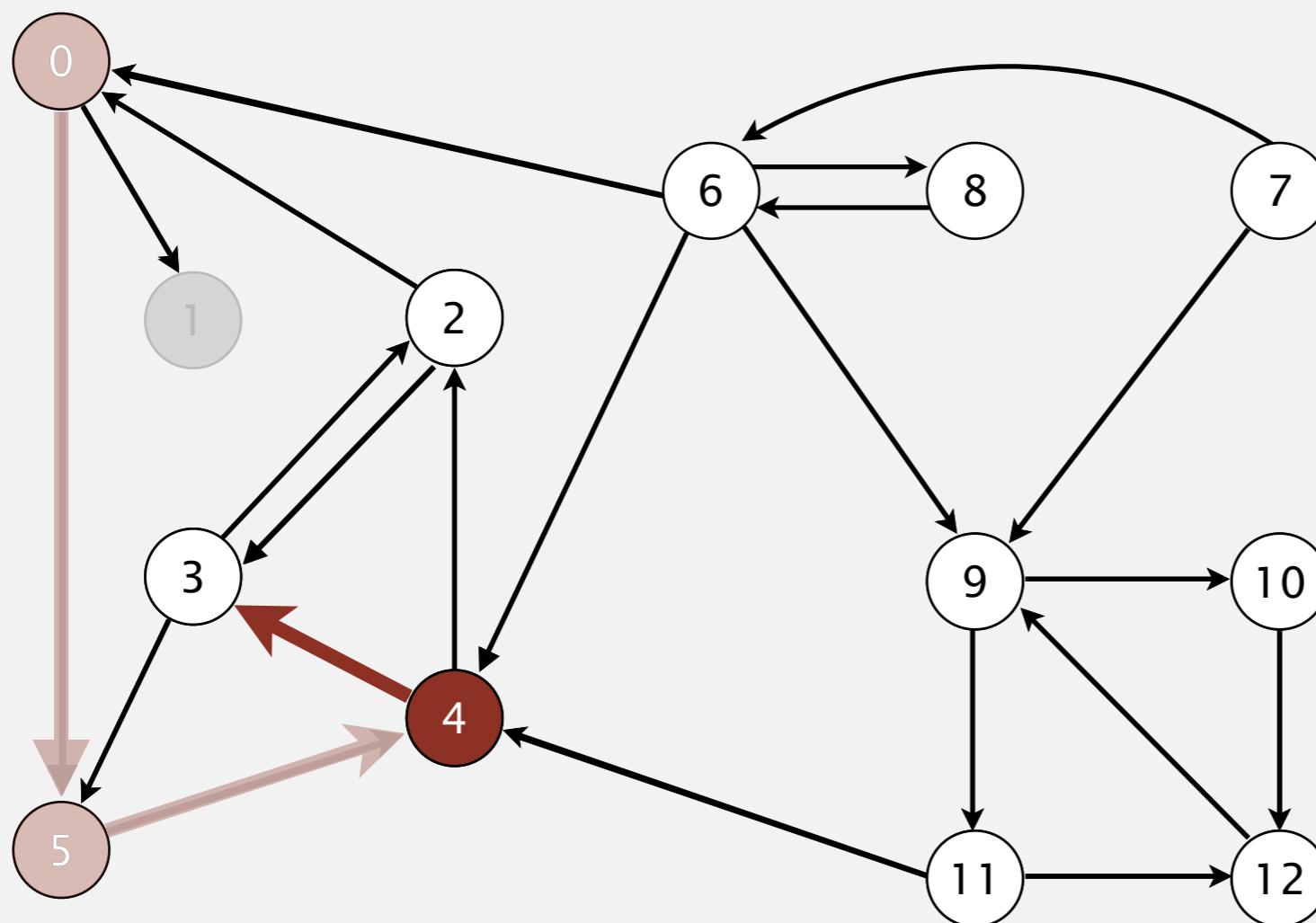


v	id[]
0	1
1	0
2	-
3	-
4	-
5	1
6	-
7	-
8	-
9	-
10	-
11	-
12	-

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

1 **0** 2 4 5 3 11 9 12 10 6 7 8



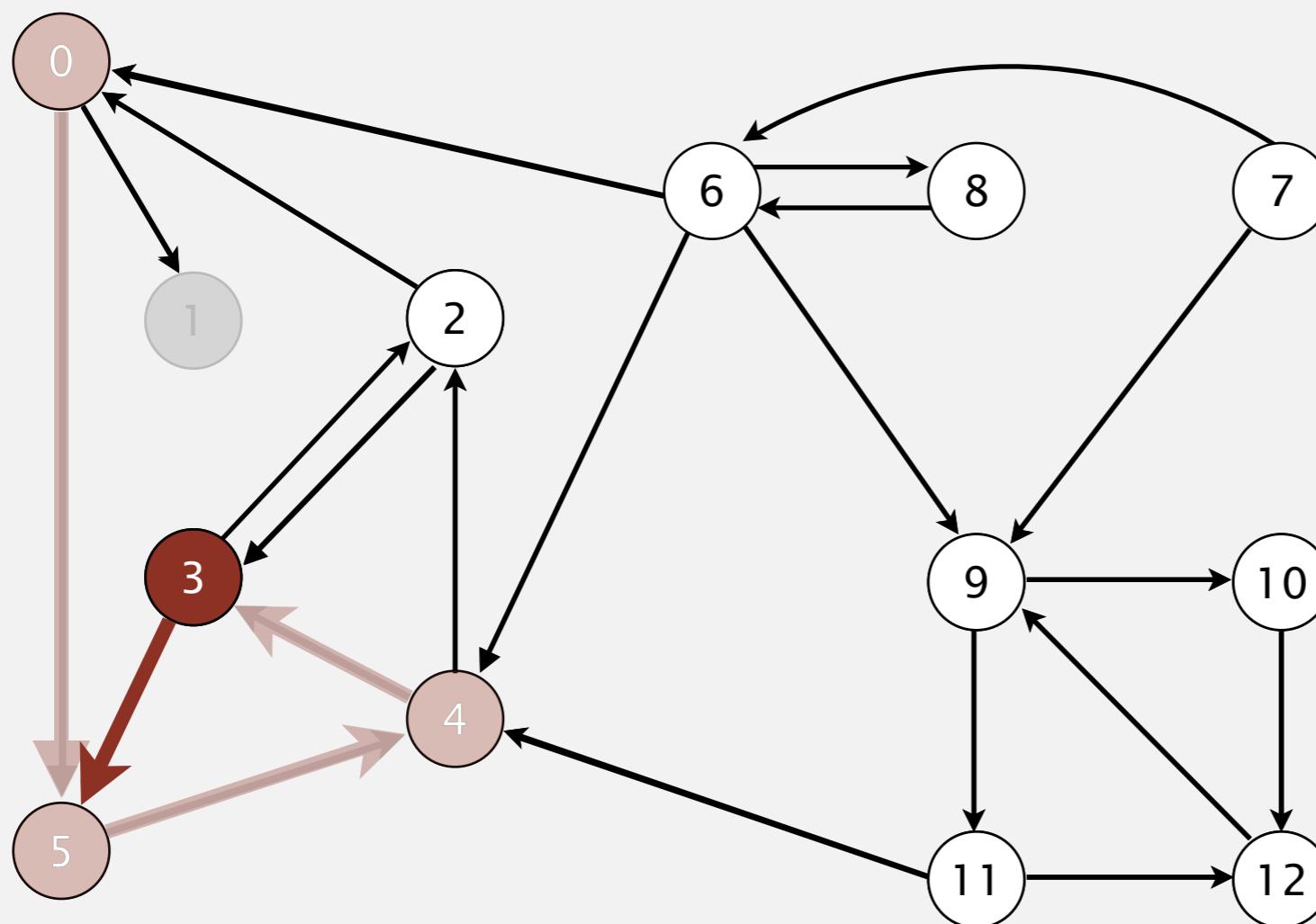
v	id[]
0	1
1	0
2	-
3	-
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	-
12	-

visit 4: check 3 and check 2

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

1 **0** 2 4 5 3 11 9 12 10 6 7 8



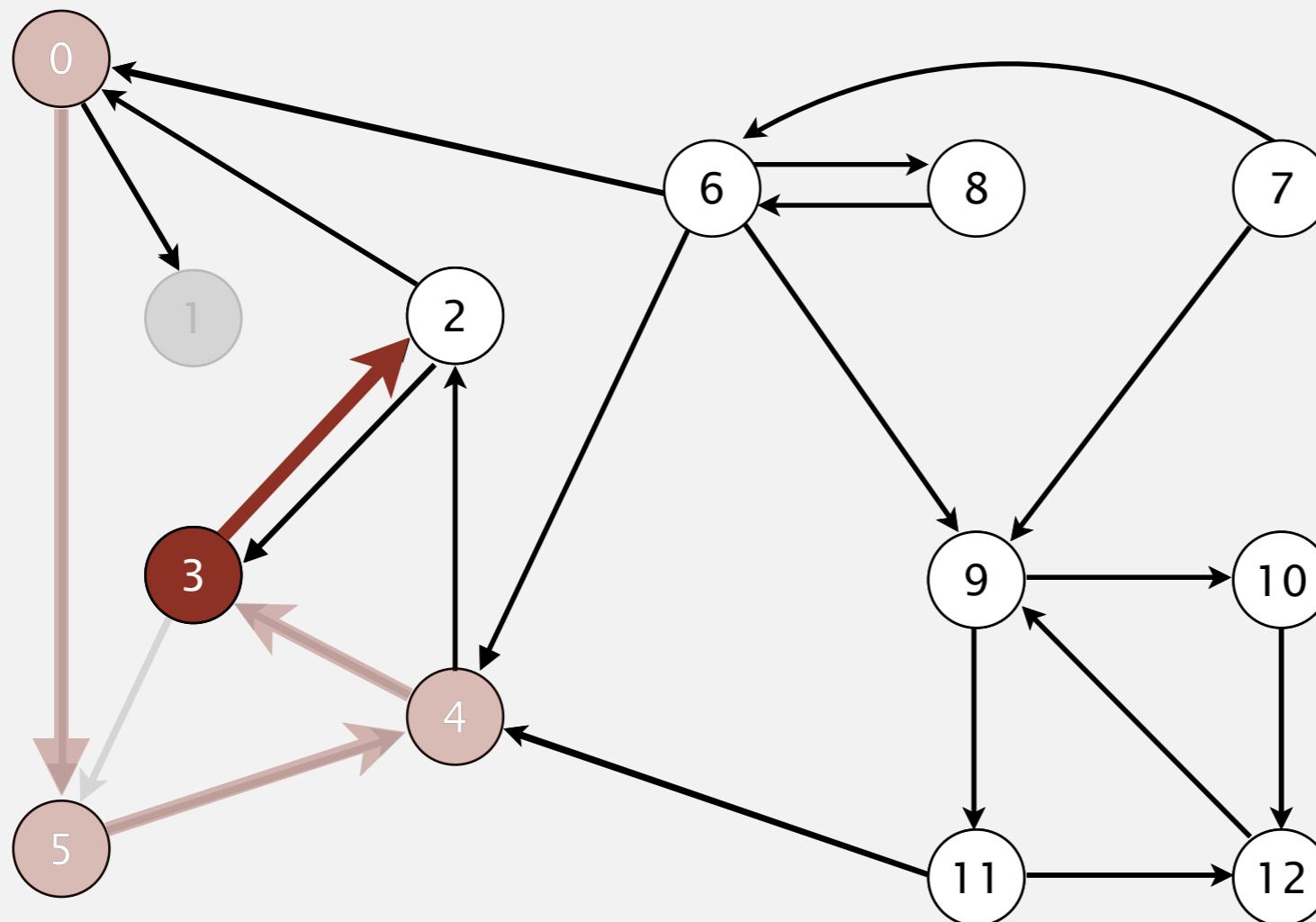
v	id[]
0	1
1	0
2	-
3	1
4	1
5	-
6	-
7	-
8	-
9	-
10	-
11	-
12	-

visit 3: check 5 and check 2

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

1 **0** 2 4 5 3 11 9 12 10 6 7 8



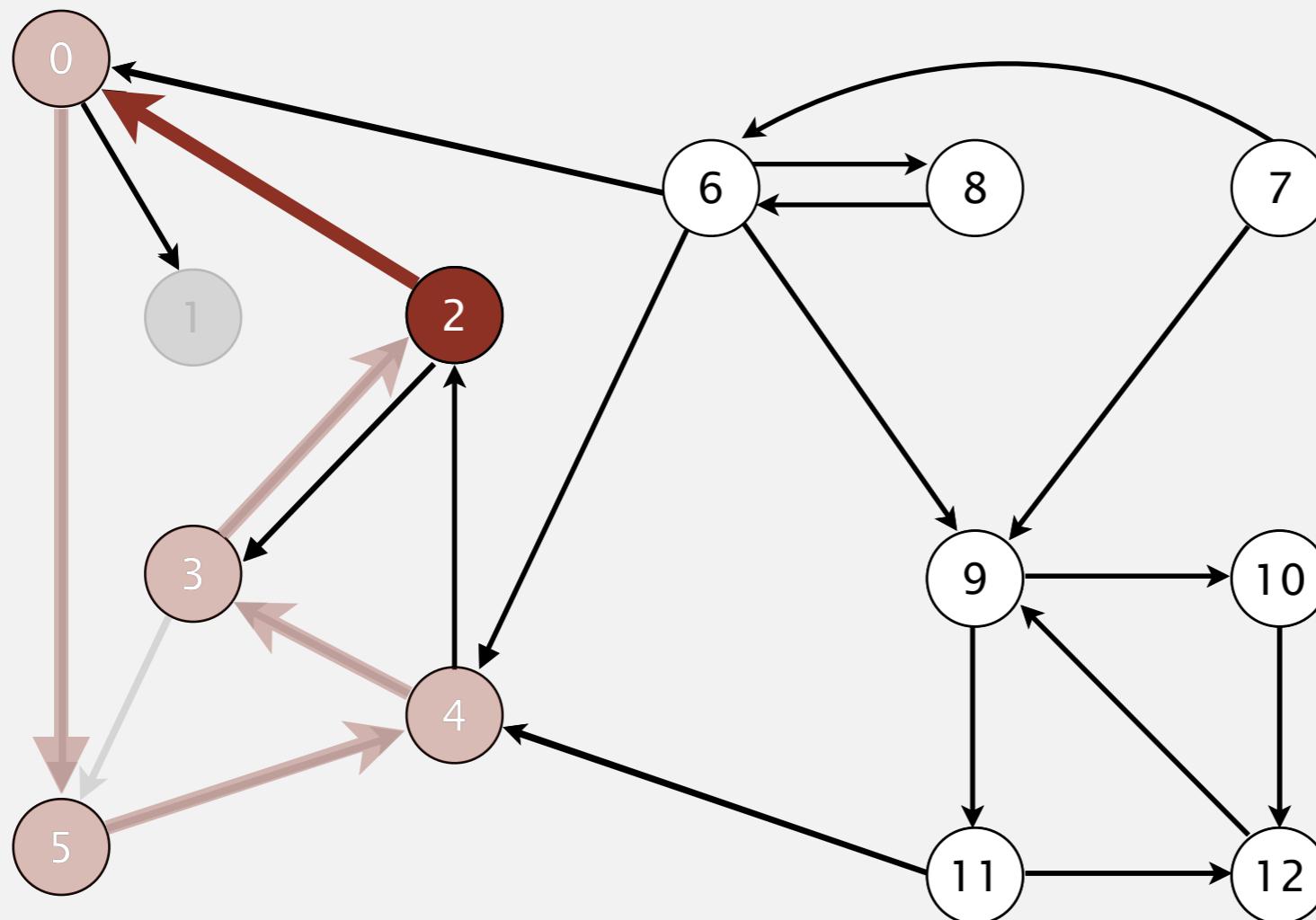
v	id[]
0	1
1	0
2	-
3	1
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	-
12	-

visit 3: check 5 and check 2

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

1 **0** 2 4 5 3 11 9 12 10 6 7 8



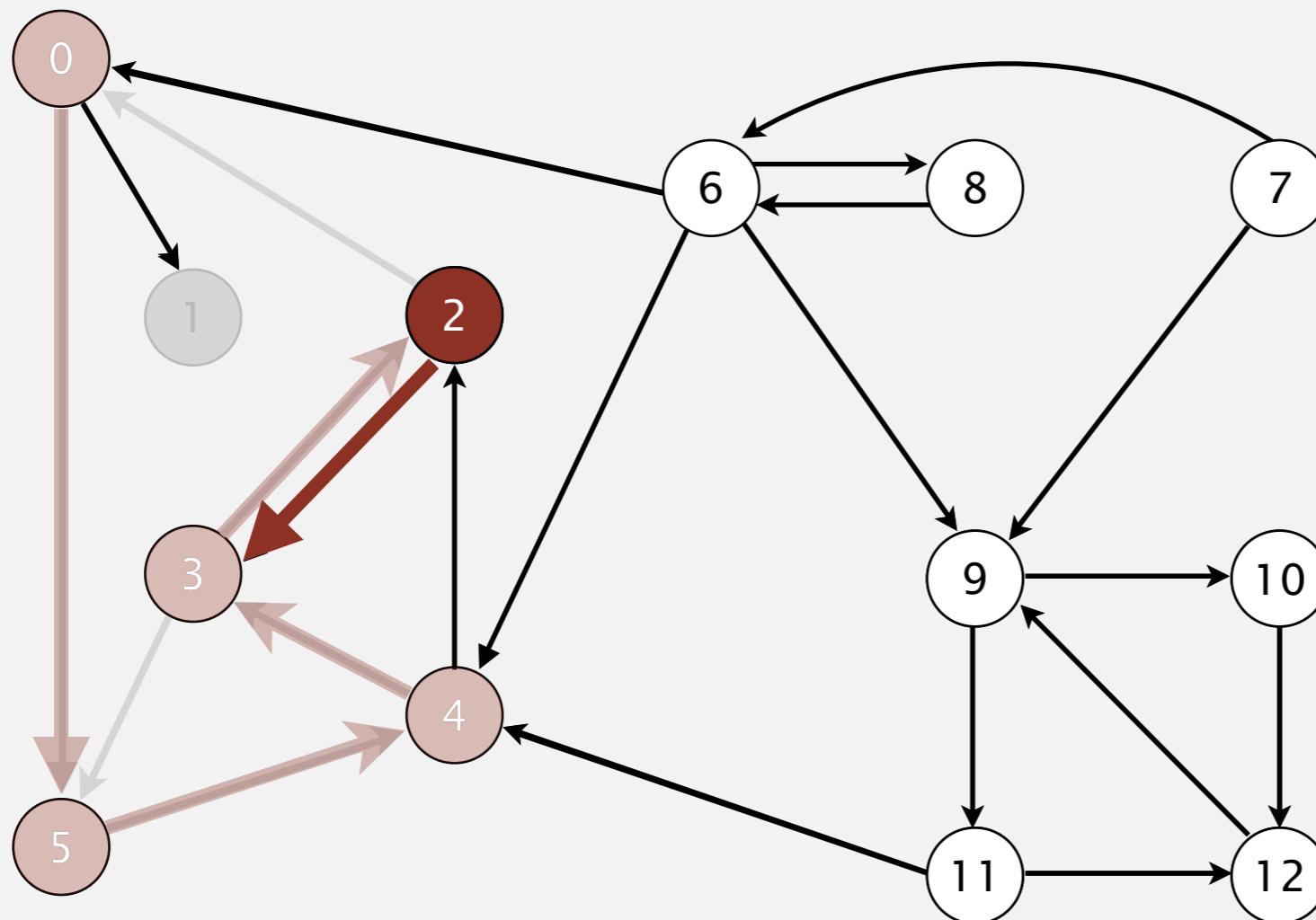
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	-
12	-

visit 2: check 0 and check 3

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

1 **0** 2 4 5 3 11 9 12 10 6 7 8



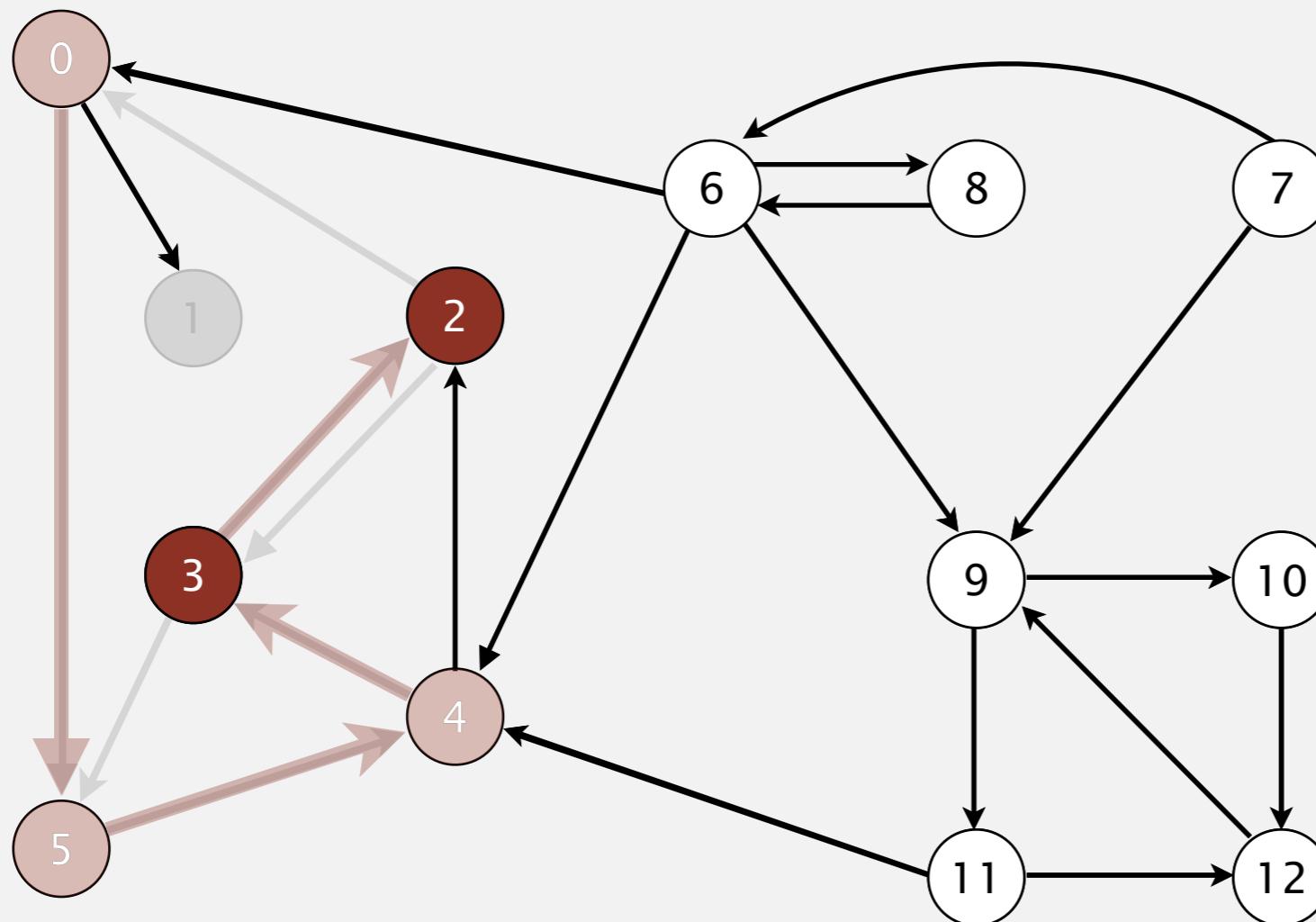
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	-
12	-

visit 2: check 0 and check 3

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

1 **0** 2 4 5 3 11 9 12 10 6 7 8



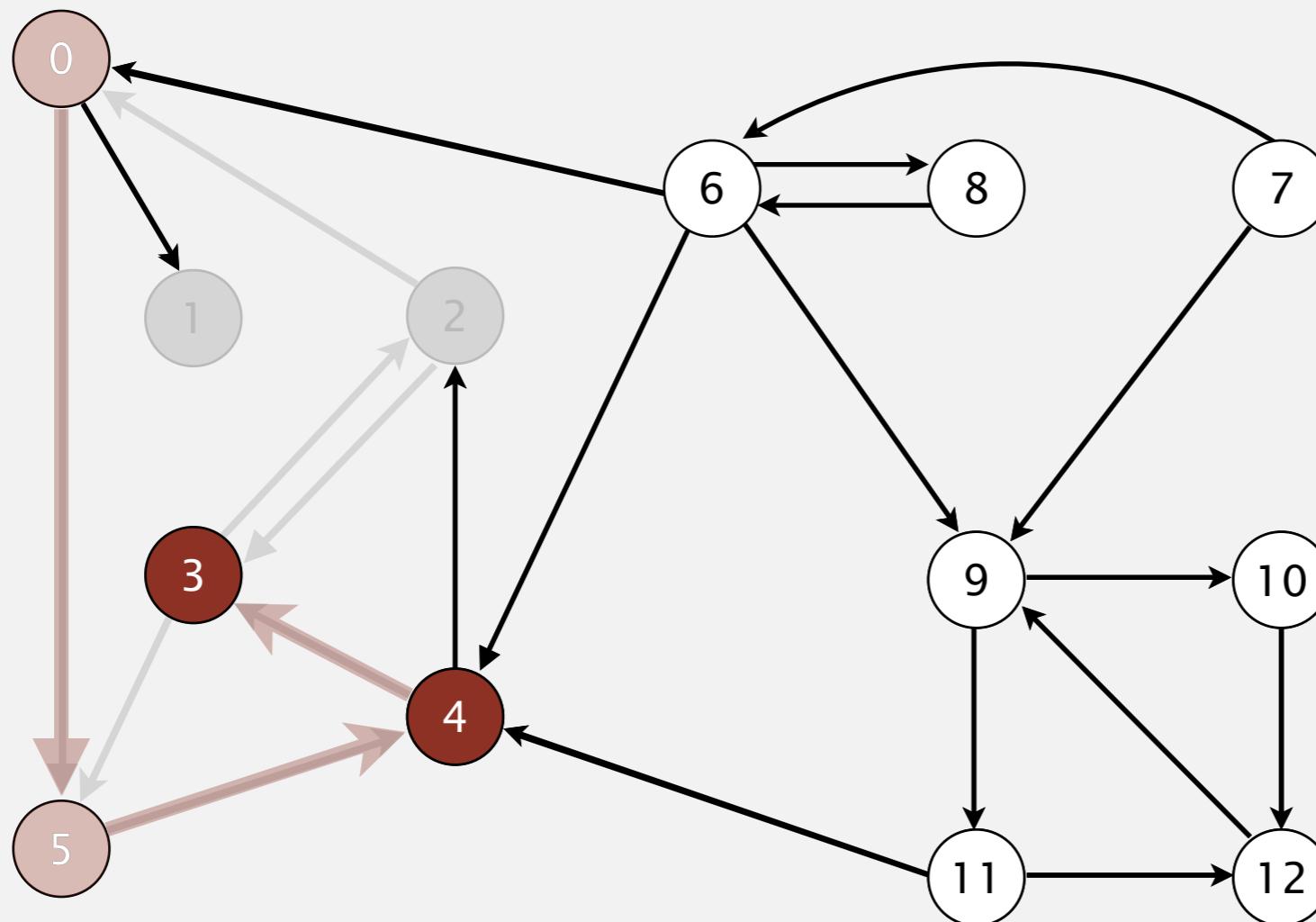
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	-
12	-

2 done

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

1 **0** 2 4 5 3 11 9 12 10 6 7 8



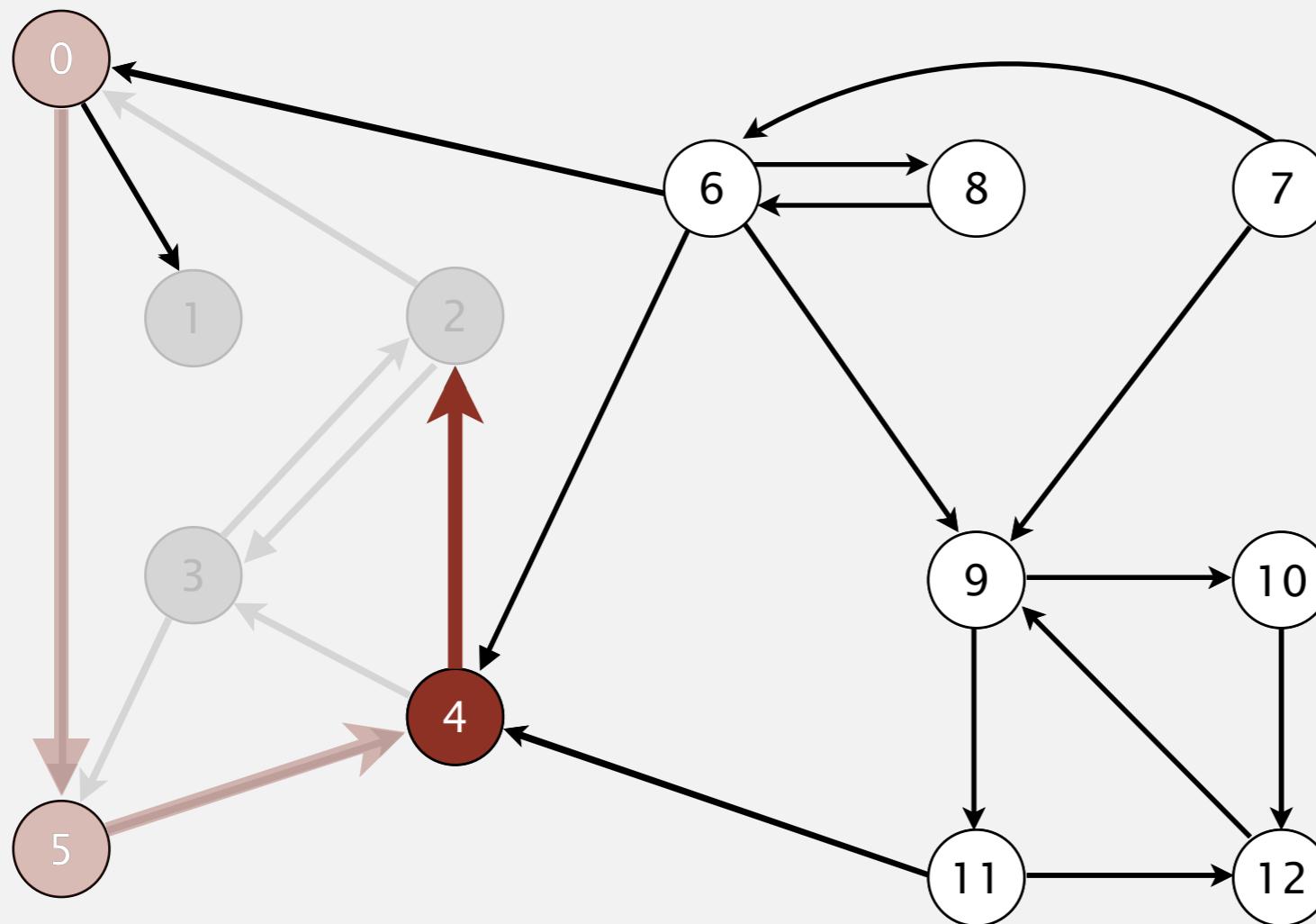
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	-
12	-

3 done

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

1 **0** 2 4 5 3 11 9 12 10 6 7 8



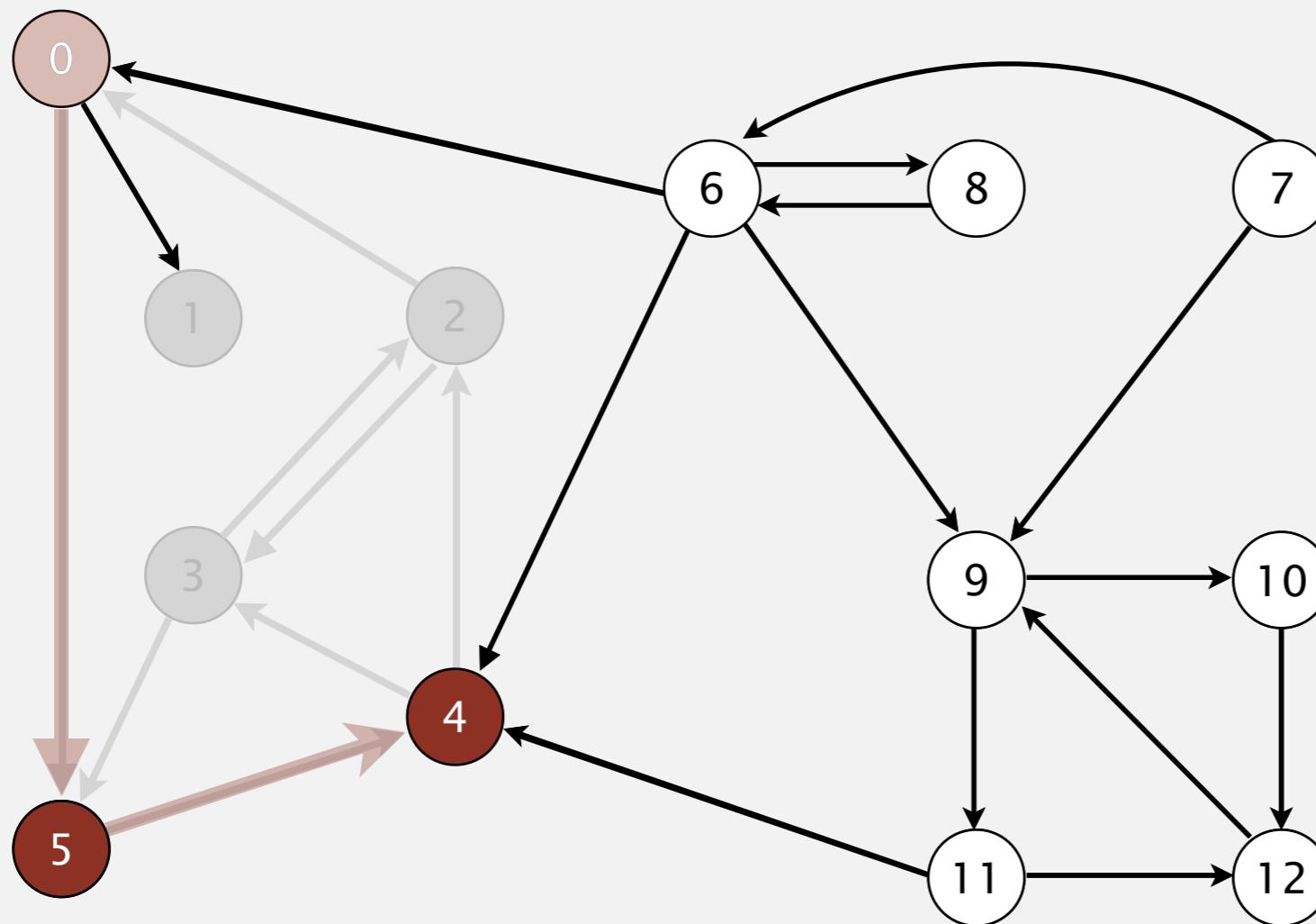
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	-
12	-

visit 4: check 3 and check 2

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

1 **0** 2 4 5 3 11 9 12 10 6 7 8



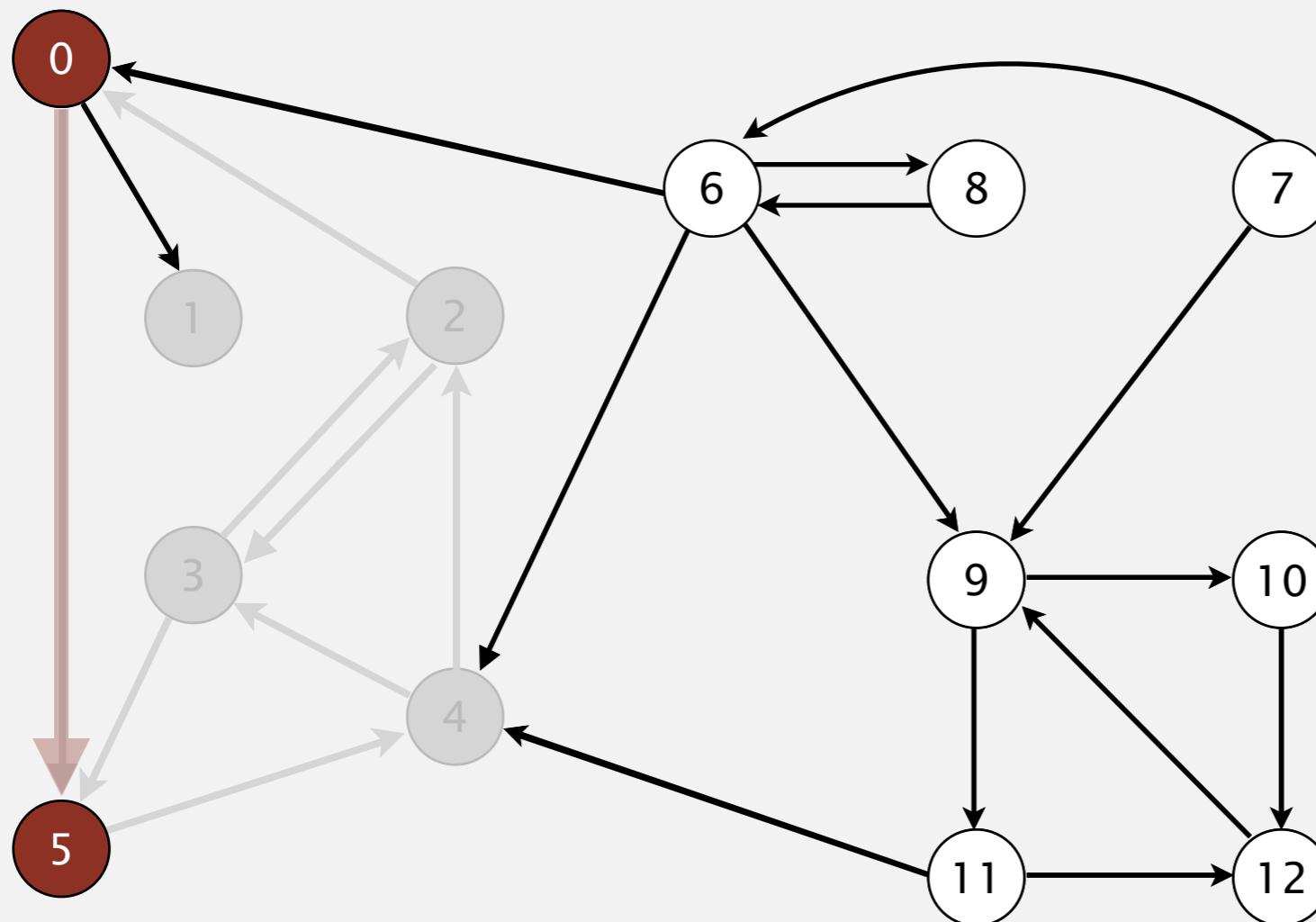
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	-
12	-

4 done

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

1 **0** 2 4 5 3 11 9 12 10 6 7 8

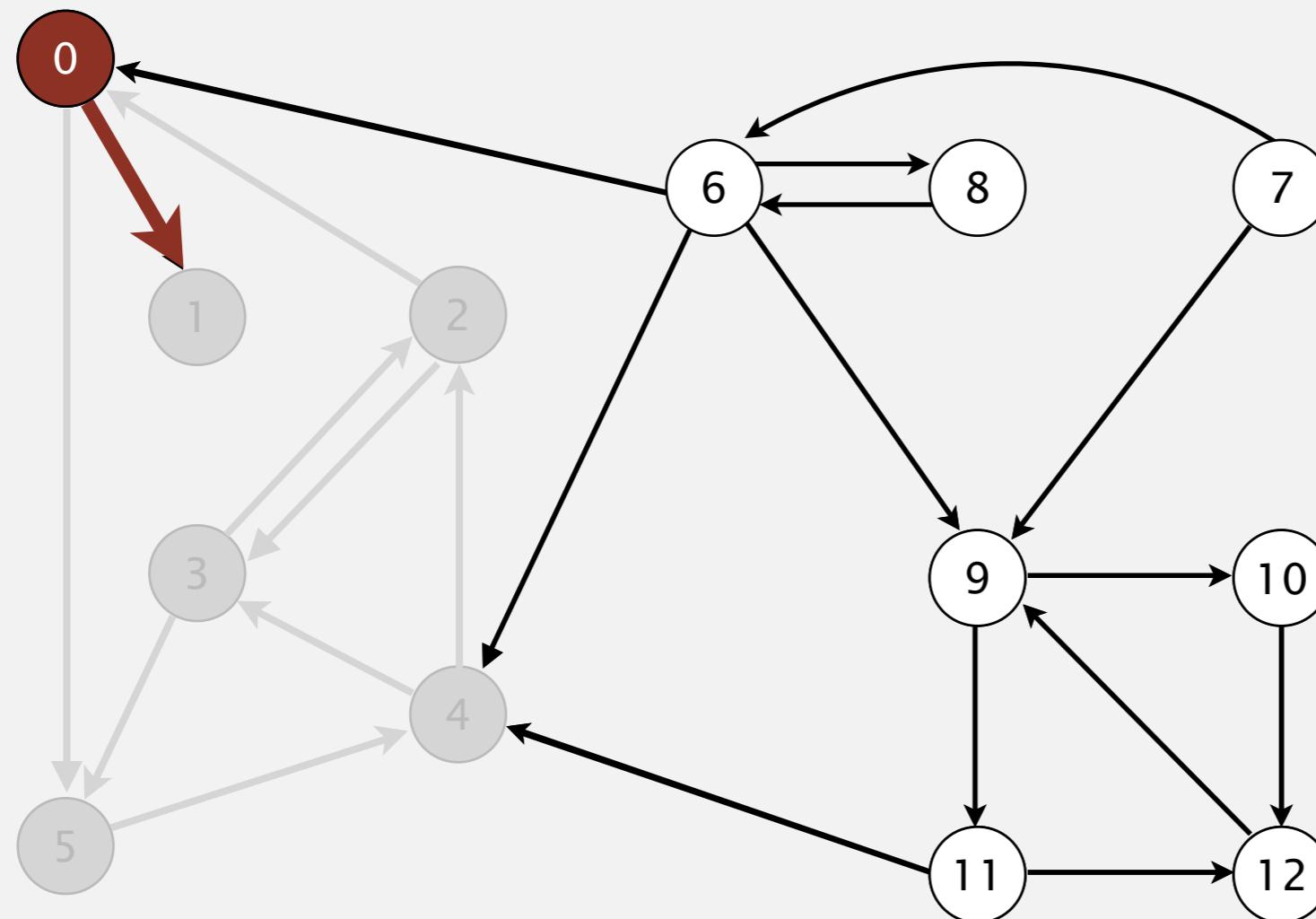


5 done

v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	-
12	-

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .      1 (0) 2 4 5 3 11 9 12 10 6 7 8

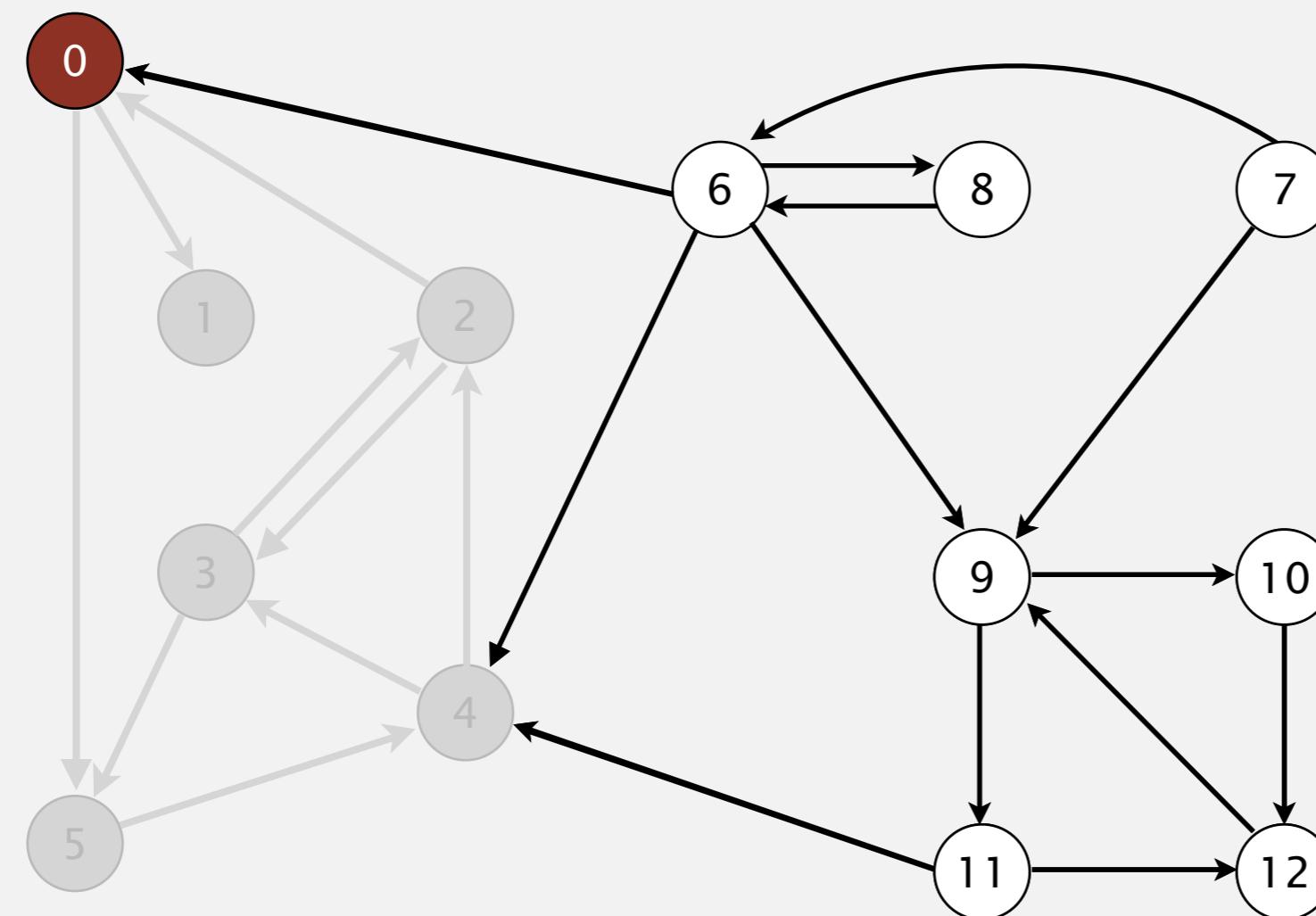


v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	-
12	-

## visit 0: check 5 and check 1

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

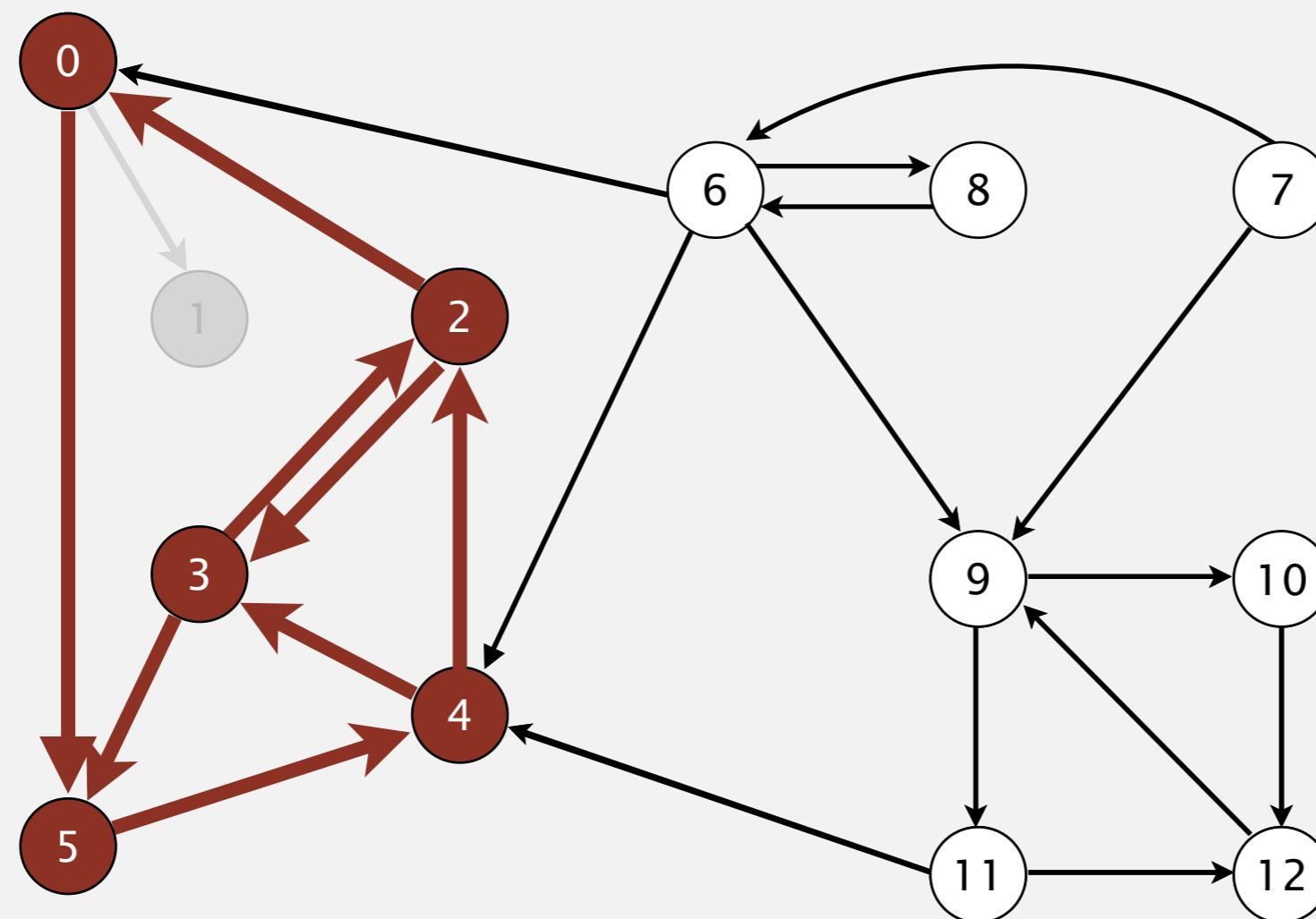


0 done

v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	-
12	-

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .



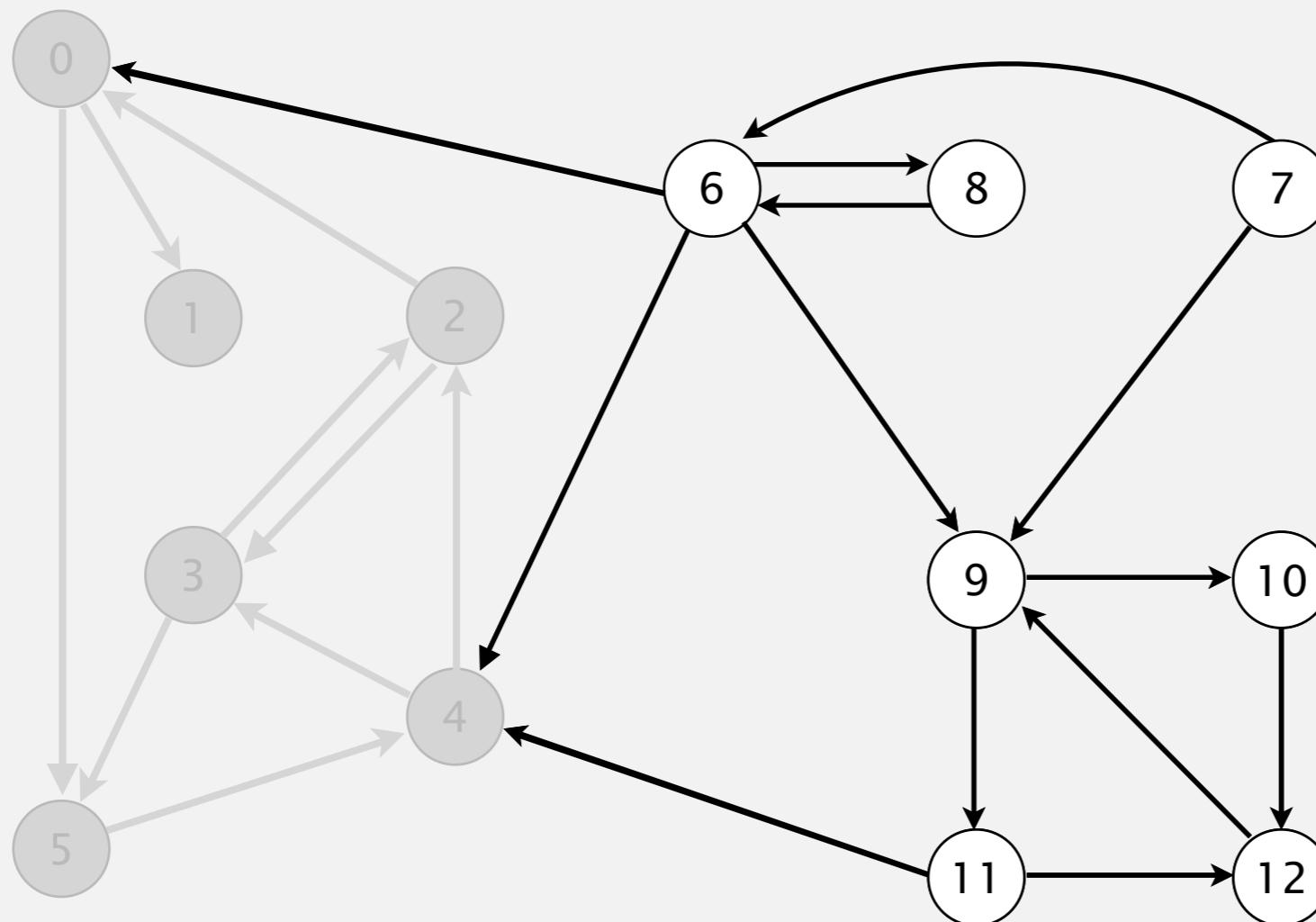
v	id[]
0	1
1	0
2	1
3	1
4	1
5	-
6	-
7	-
8	-
9	-
10	-
11	-
12	-

strong component: 0 2 3 4 5

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

1 0 **2** 4 5 3 11 9 12 10 6 7 8



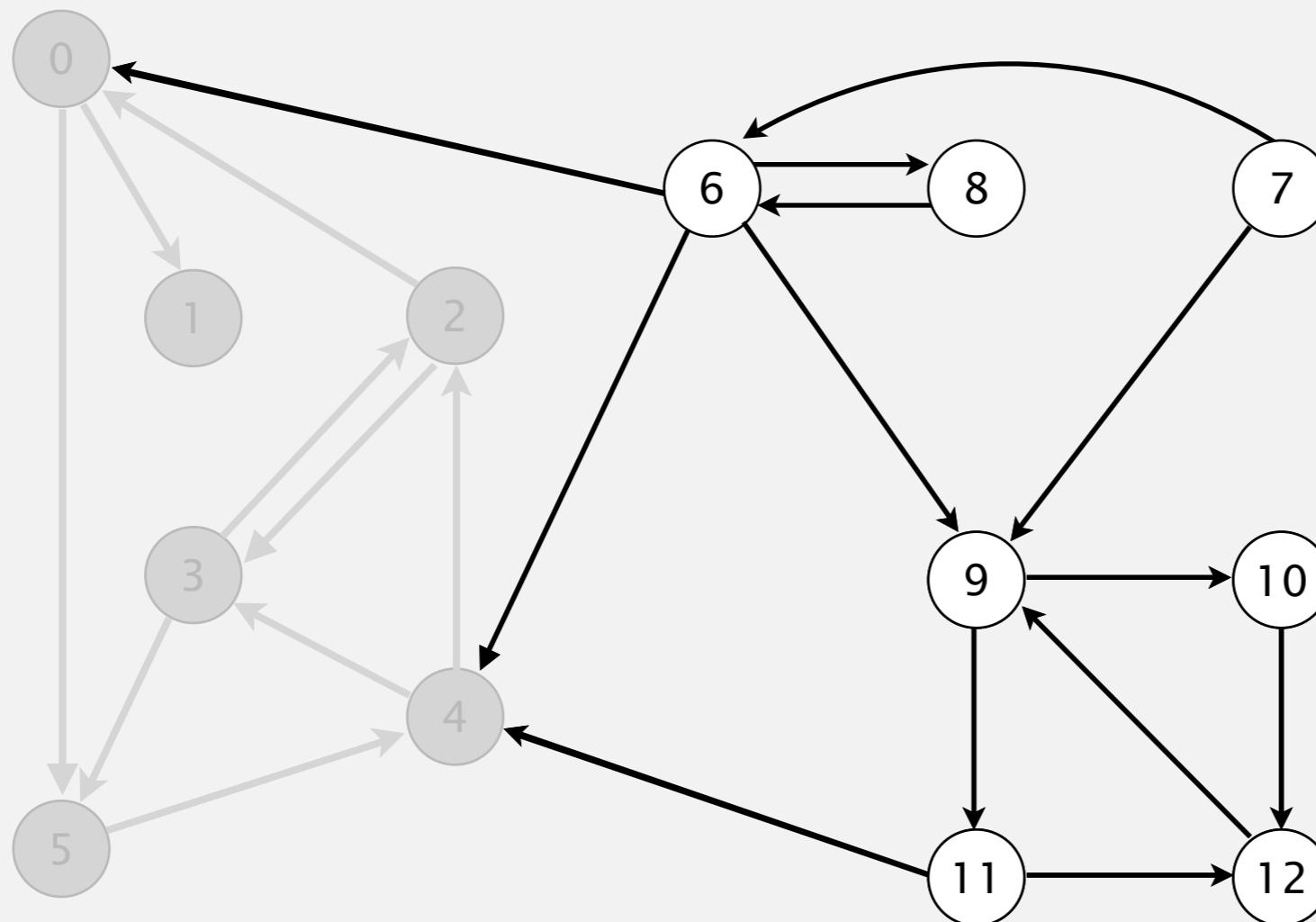
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	-
12	-

check 2

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

1 0 2 4 5 3 11 9 12 10 6 7 8



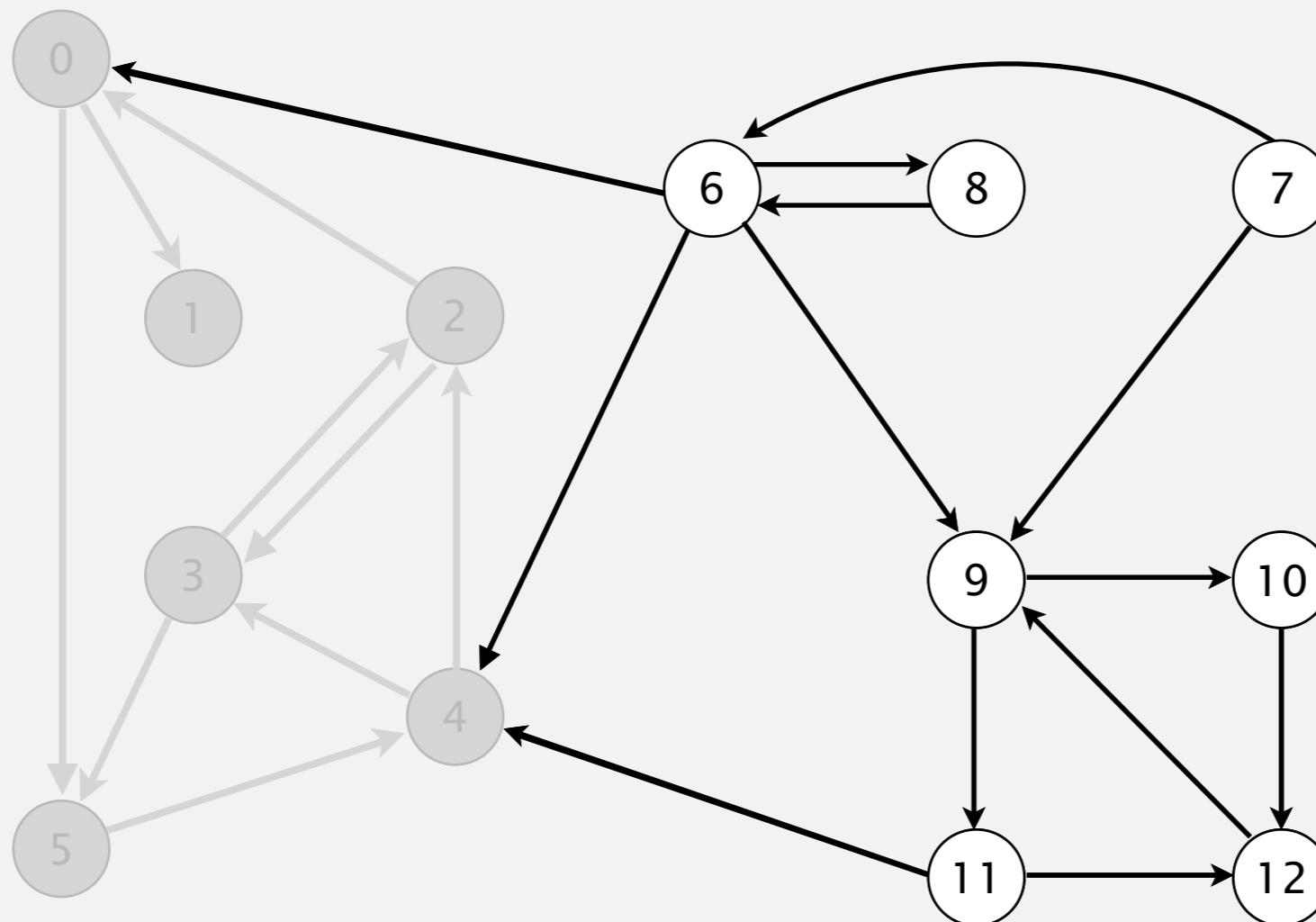
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	-
12	-

check 4

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

1 0 2 4 5 3 11 9 12 10 6 7 8

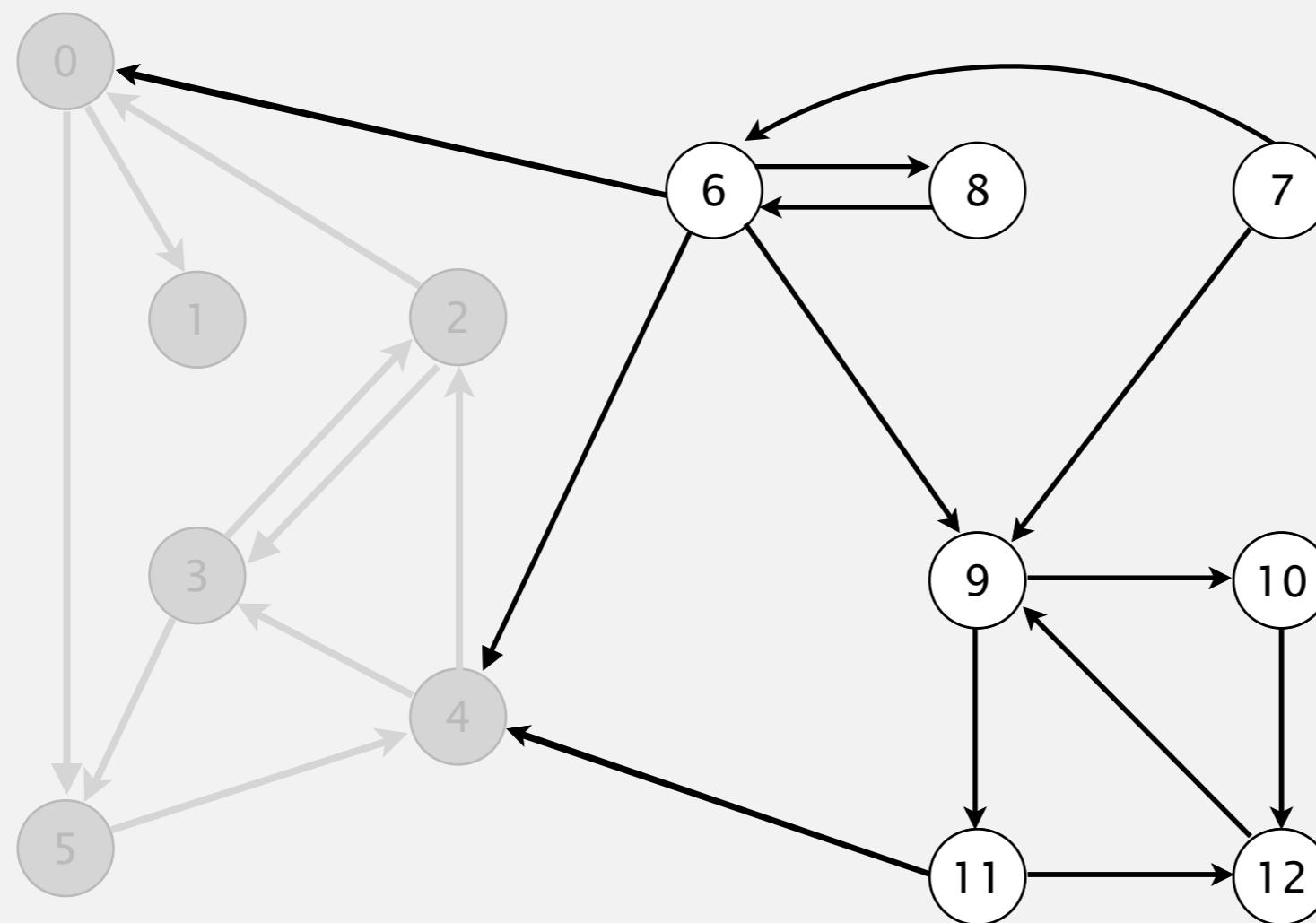


v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	-
12	-

check 5

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

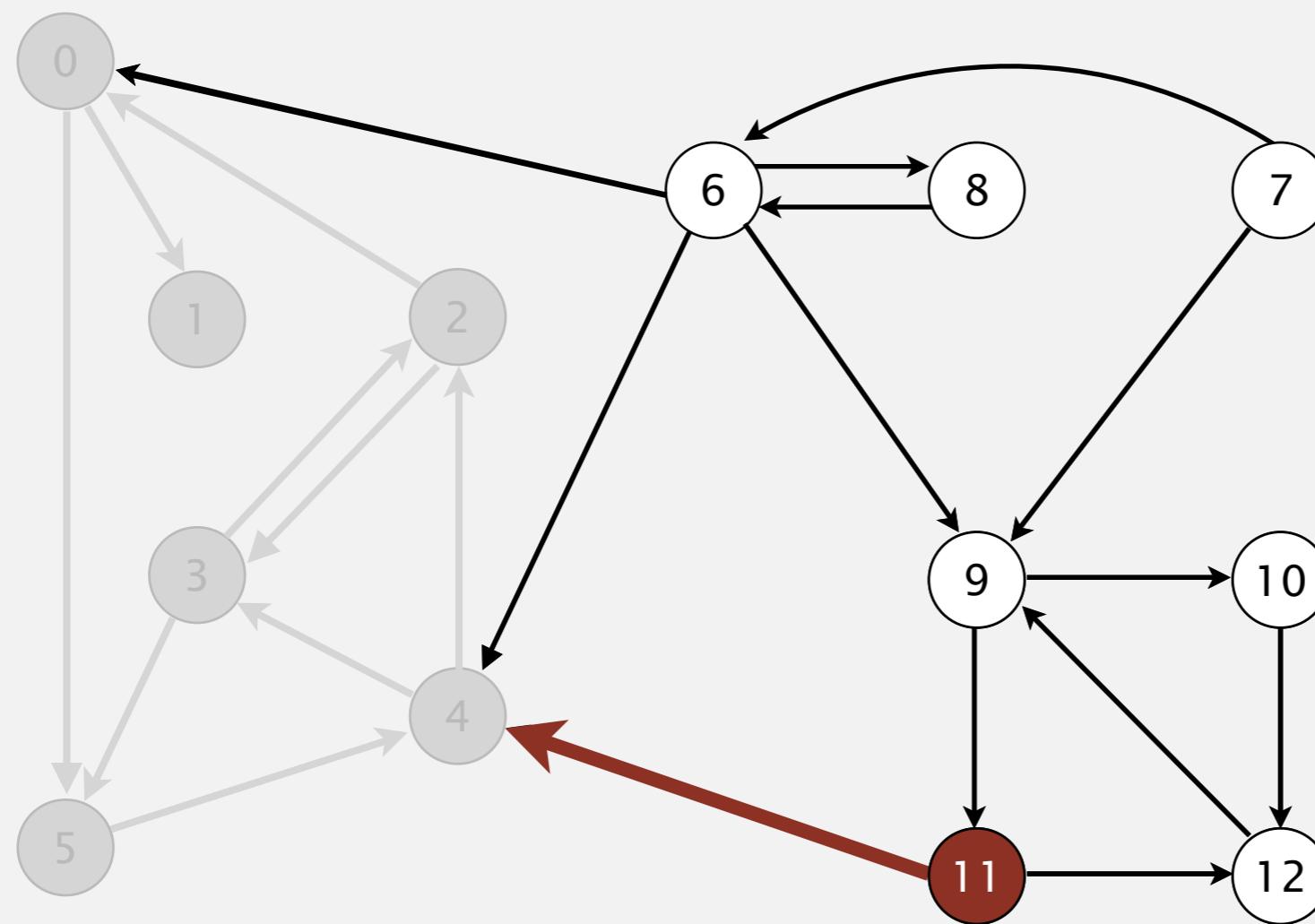


v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	-
12	-

check 3

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

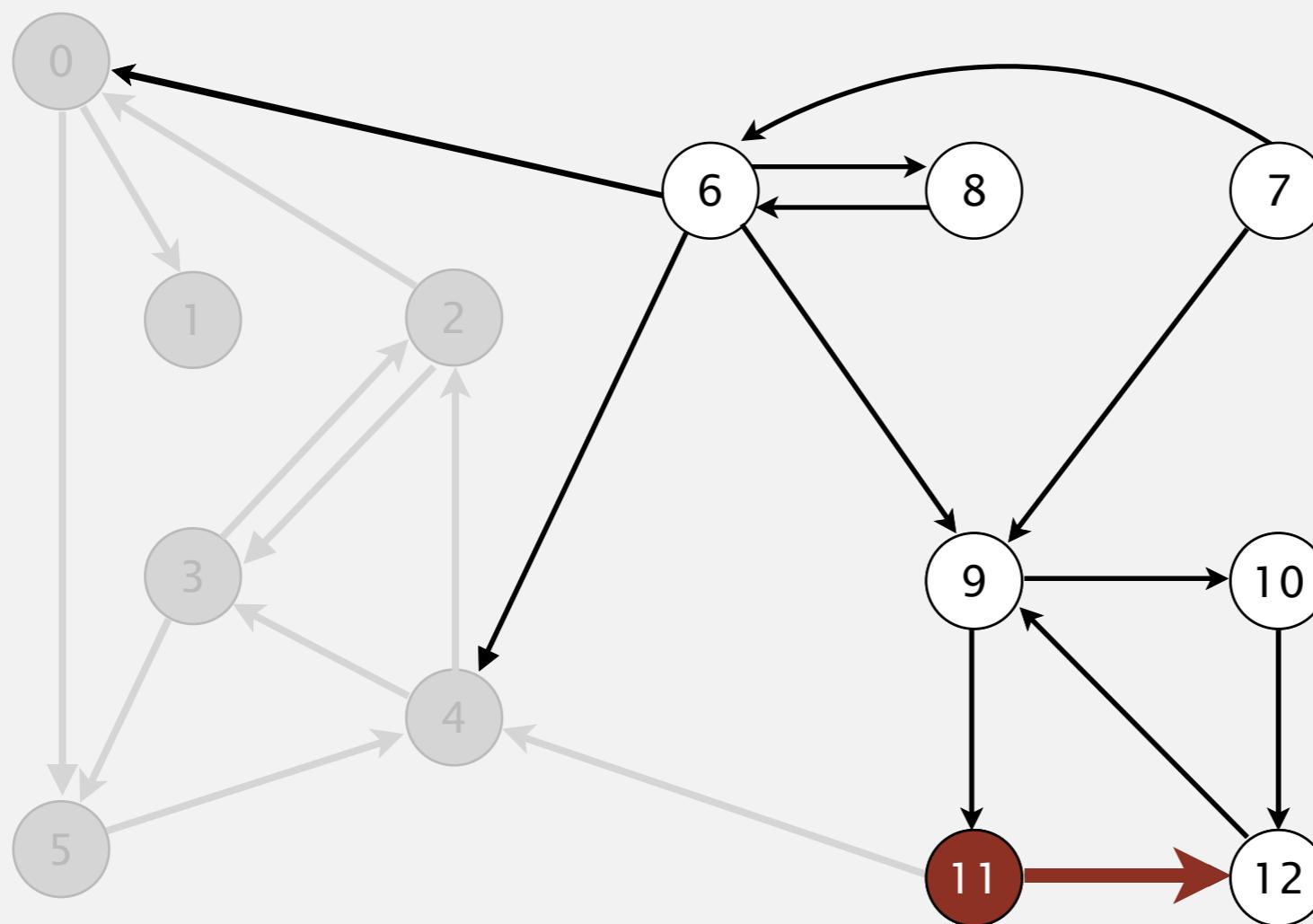


visit 11: check 4 and check 12

v	id[]
0	1
1	0
2	1
3	1
4	1
5	-
6	-
7	-
8	-
9	-
10	-
11	2
12	-

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

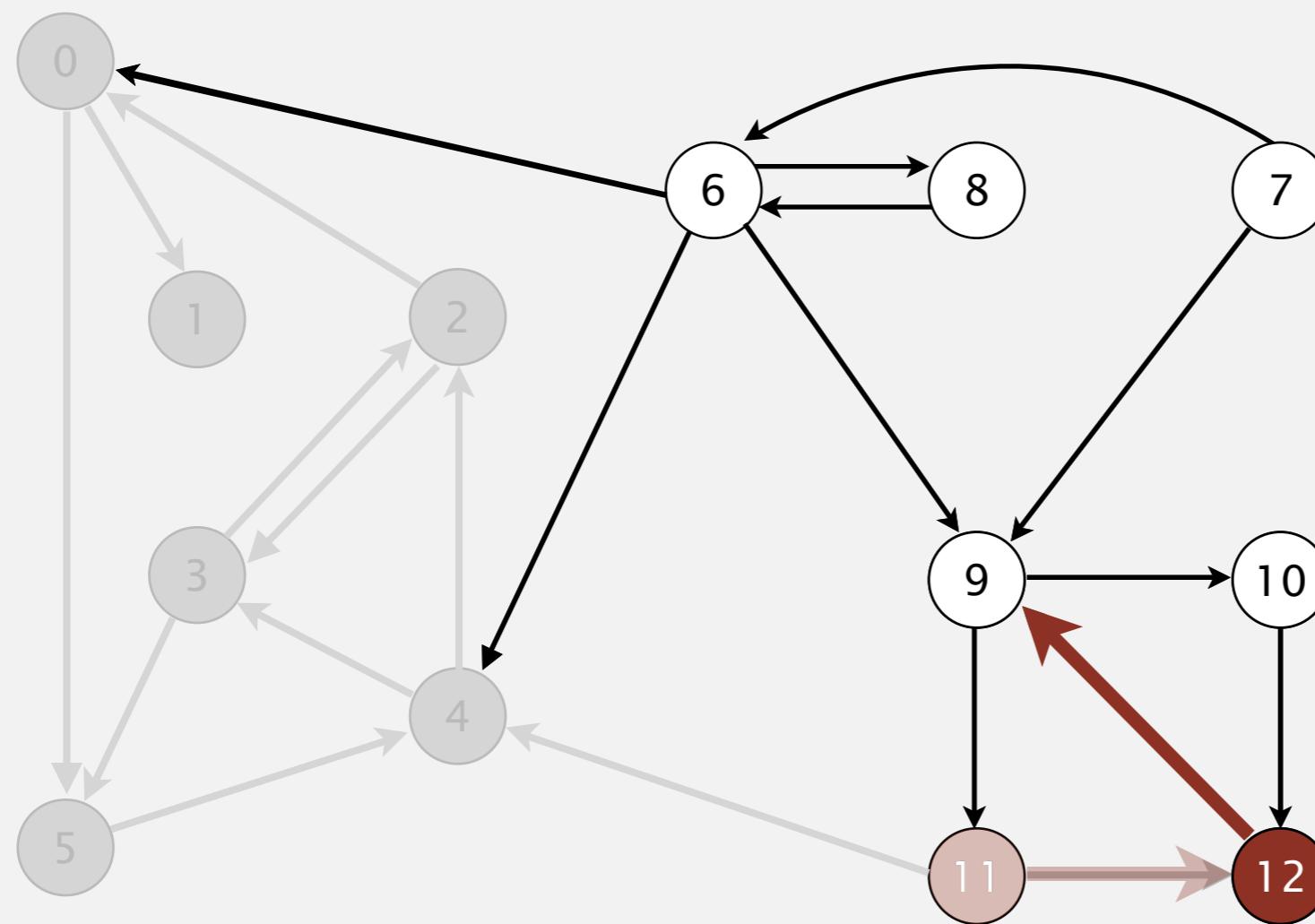


visit 11: check 4 and check 12

v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	2
12	-

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

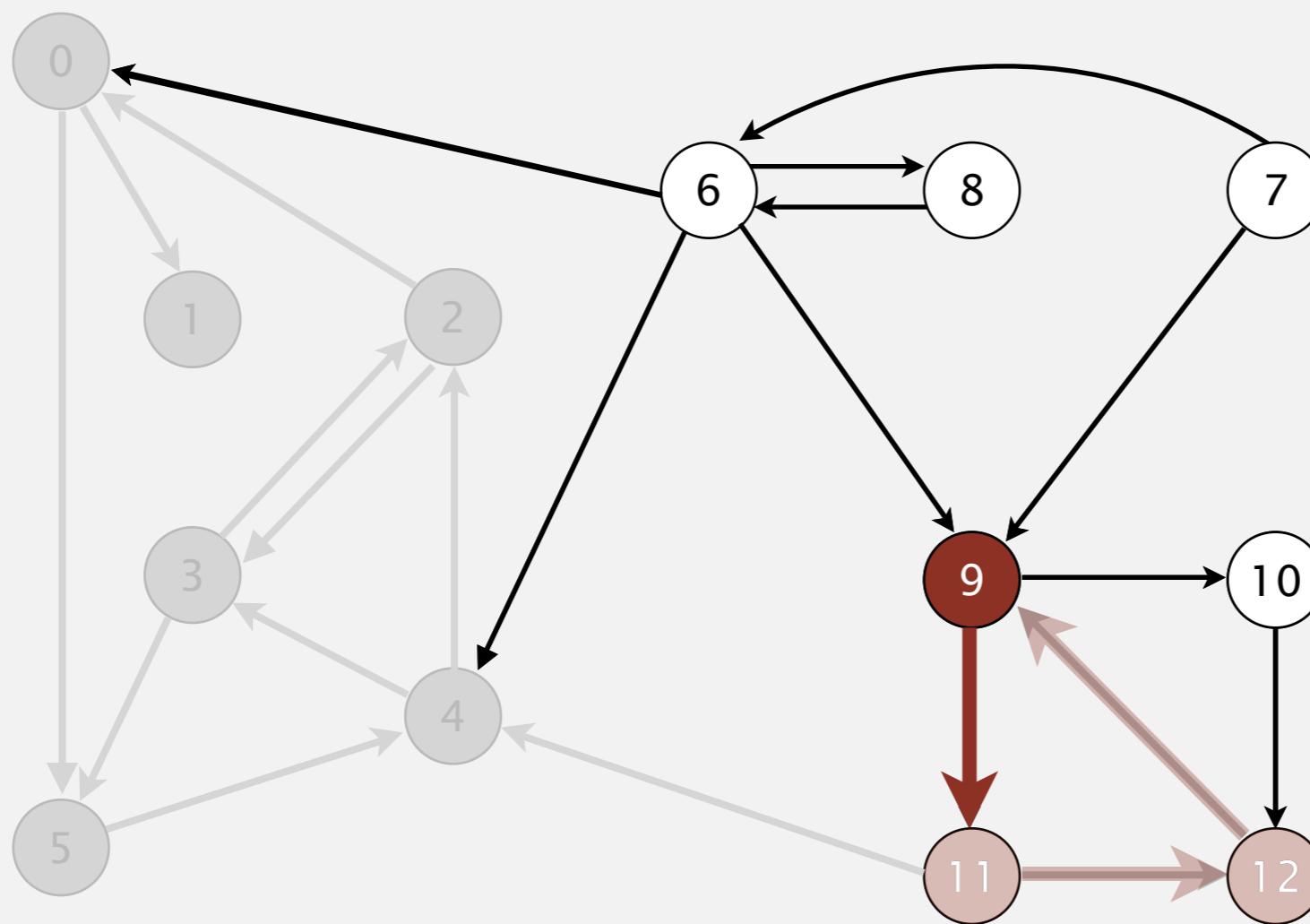


visit 12: check 9

v	$\text{id}[v]$
0	1
1	0
2	1
3	1
4	1
5	-
6	-
7	-
8	-
9	-
10	-
11	2
12	2

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

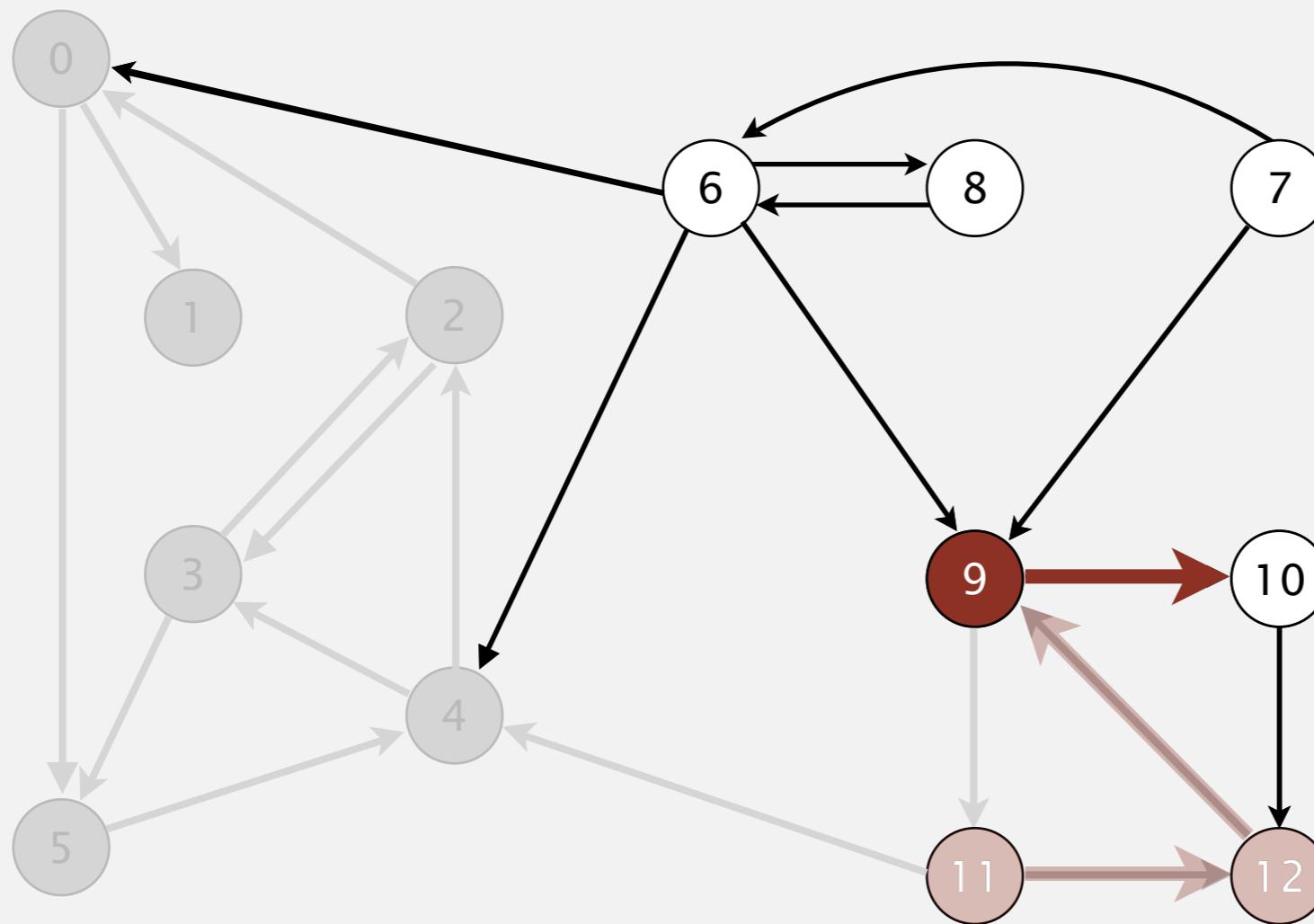


visit 9: check 11 and check 10

v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	2
10	-
11	2
12	2

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

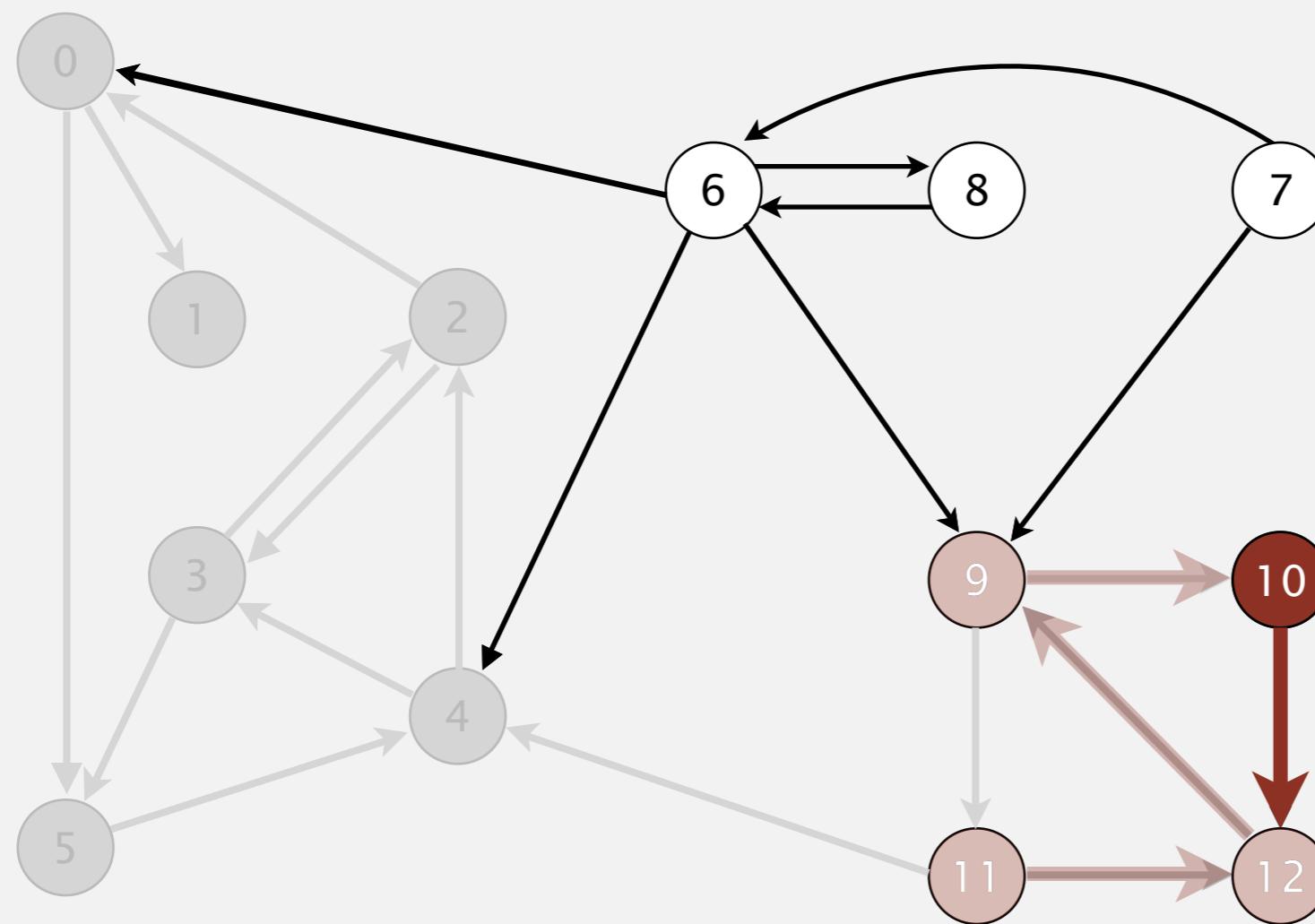


v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	2
10	-
11	2
12	2

visit 9: check 11 and check 10

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

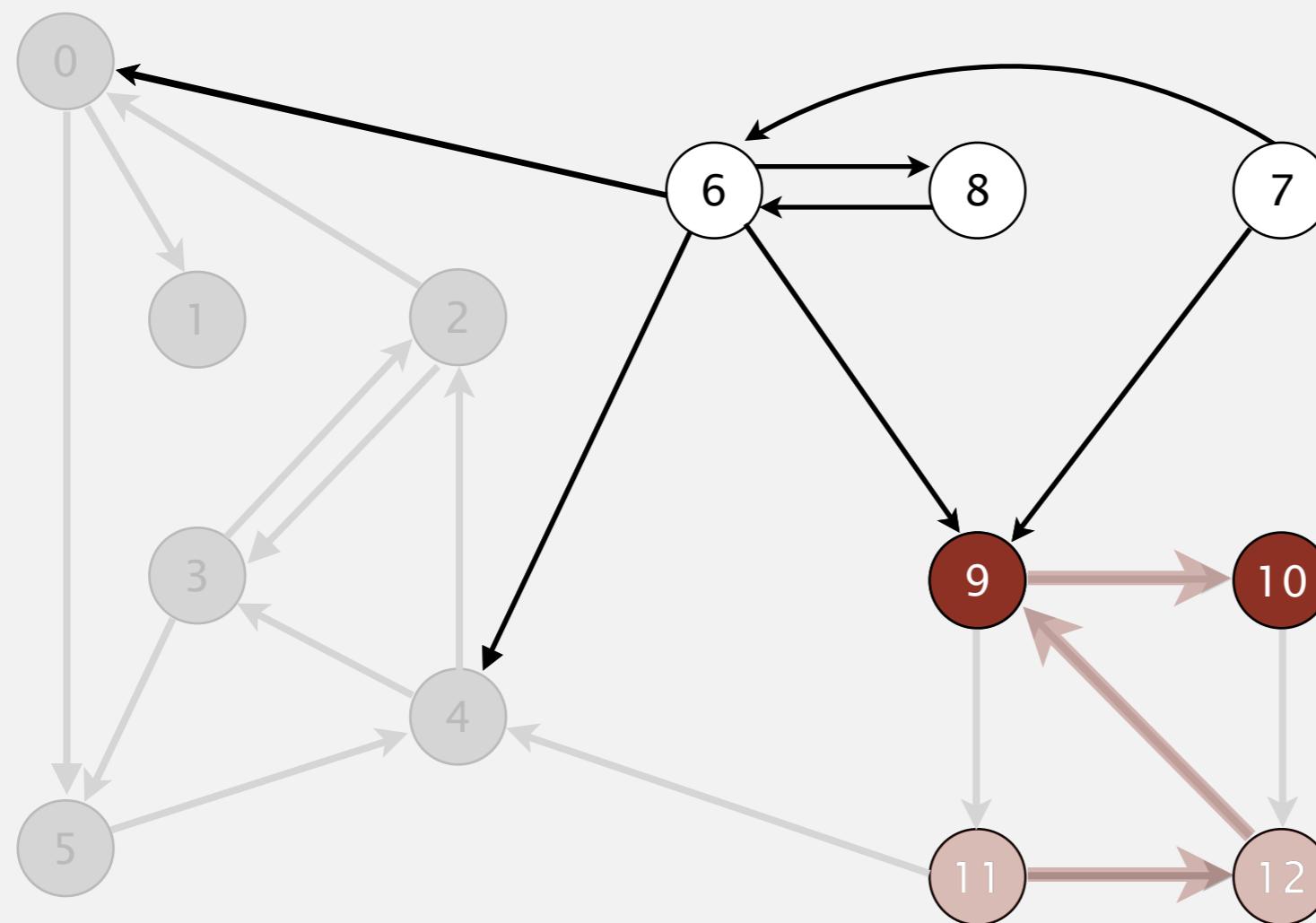


visit 10: check 12

v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	2
10	2
11	2
12	2

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

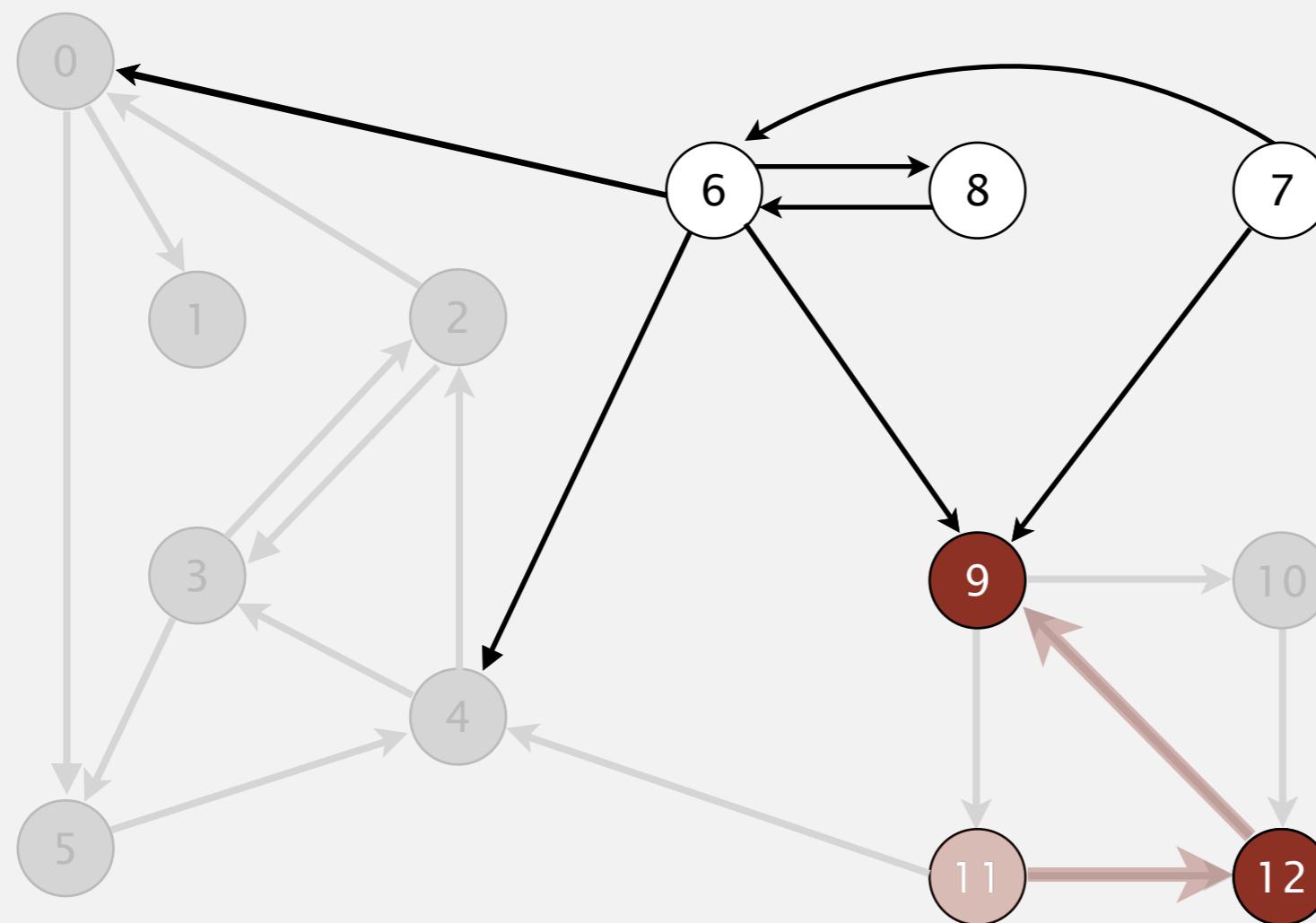


v	id[]
0	1
1	0
2	1
3	1
4	1
5	-
6	-
7	-
8	-
9	2
10	2
11	2
12	2

10 done

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

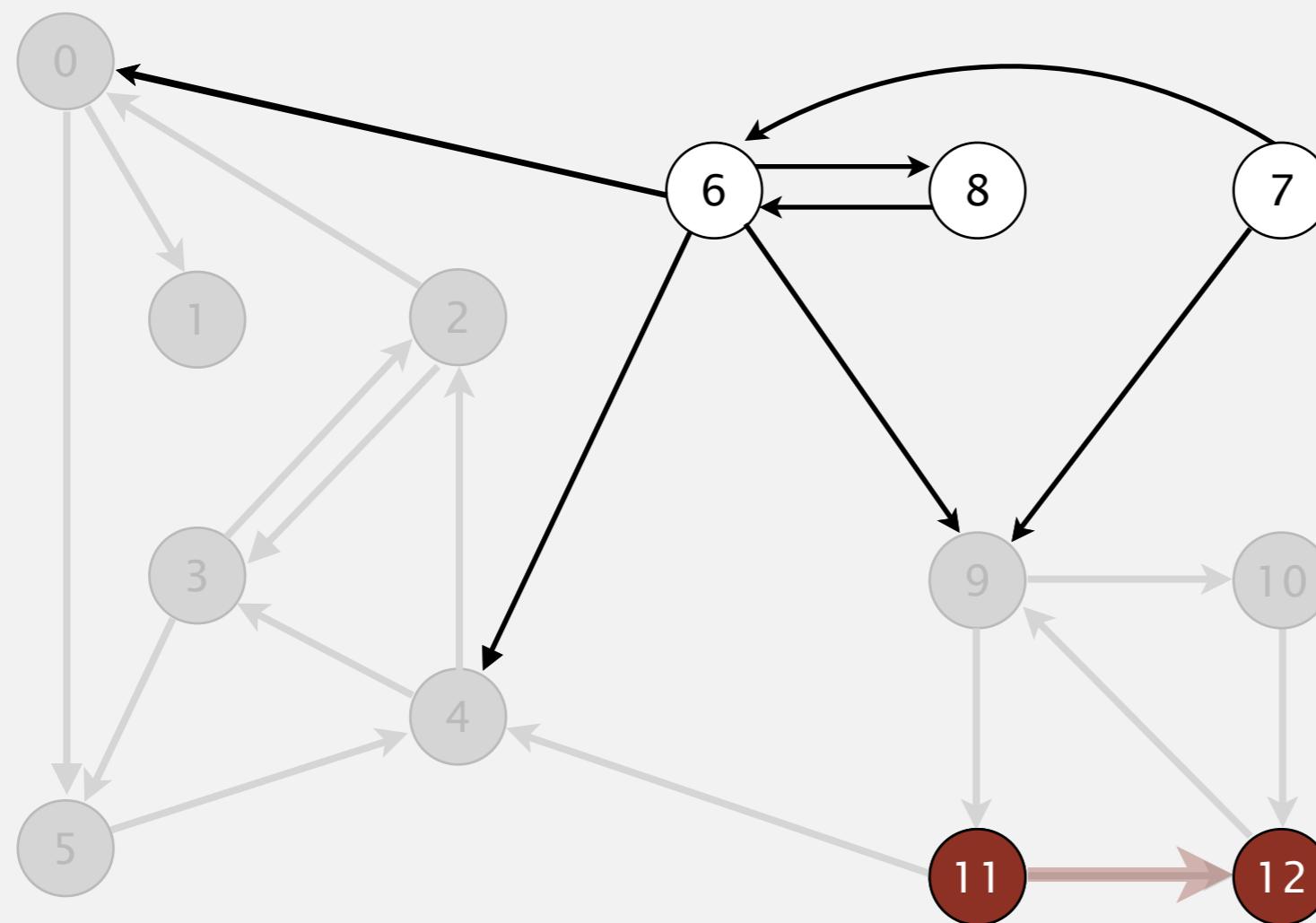


v	id[]
0	1
1	0
2	1
3	1
4	1
5	-
6	-
7	-
8	-
9	2
10	2
11	2
12	2

9 done

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

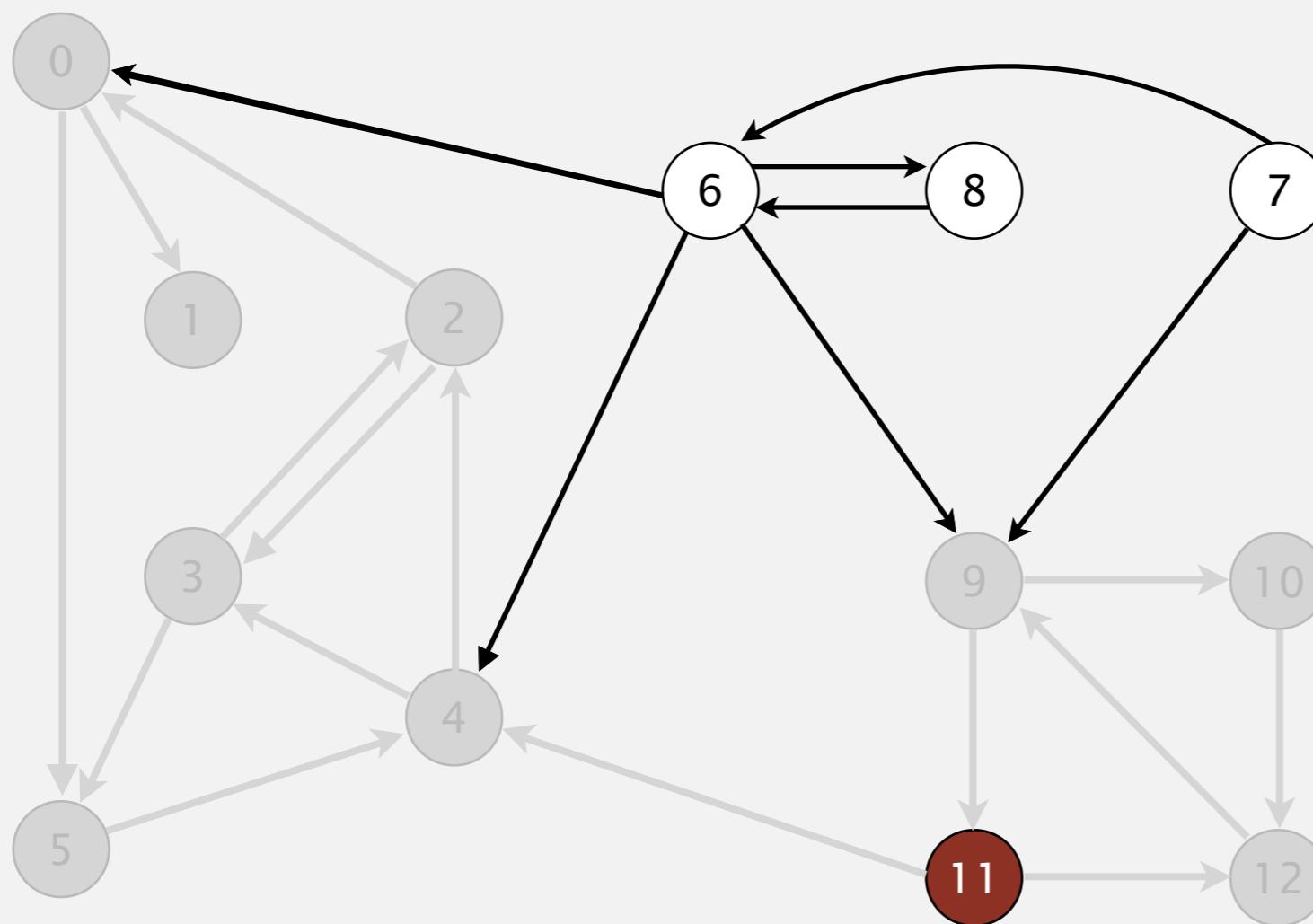


12 done

v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	2
10	2
11	2
12	2

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

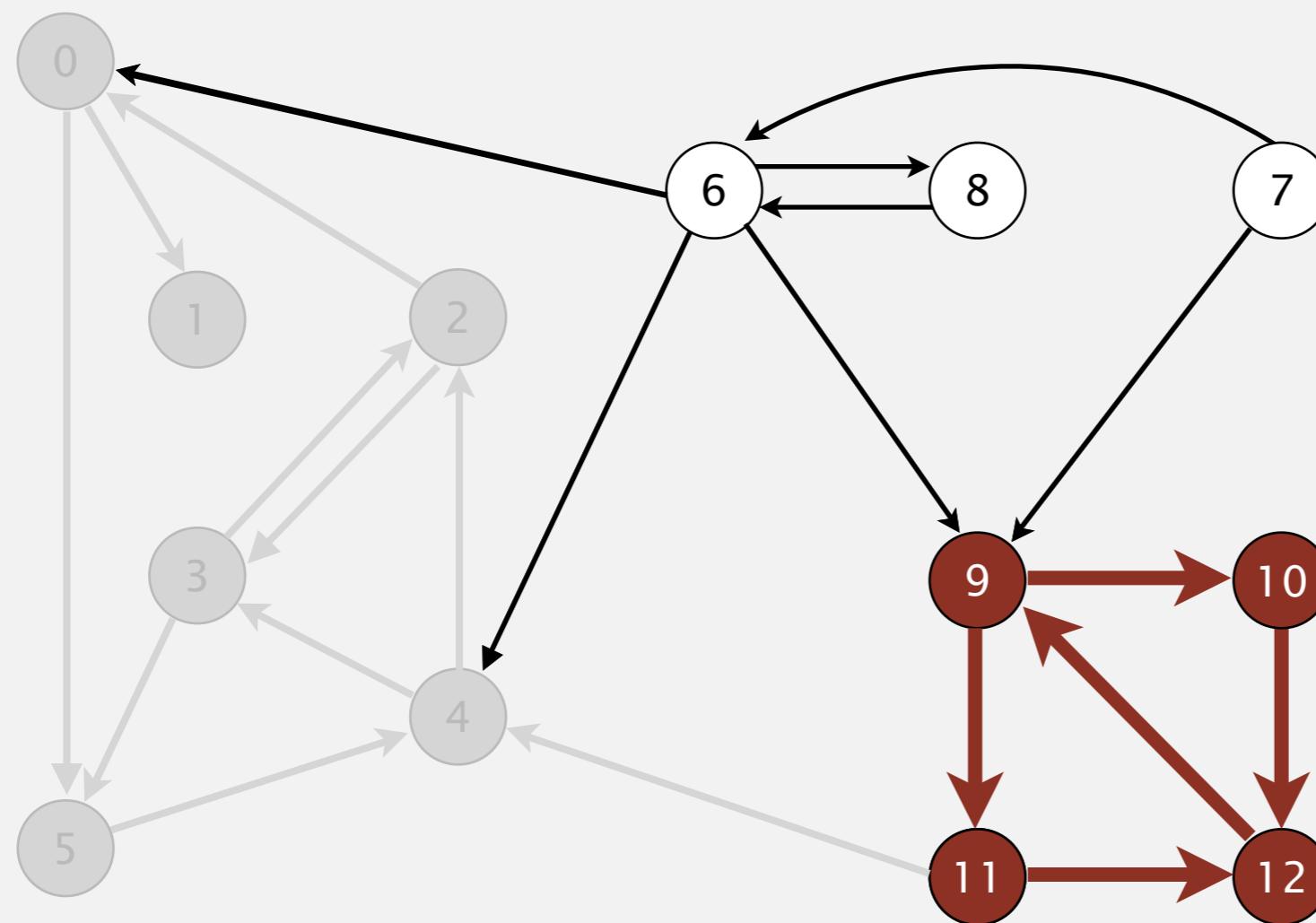


v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	2
10	2
11	2
12	2

11 done

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

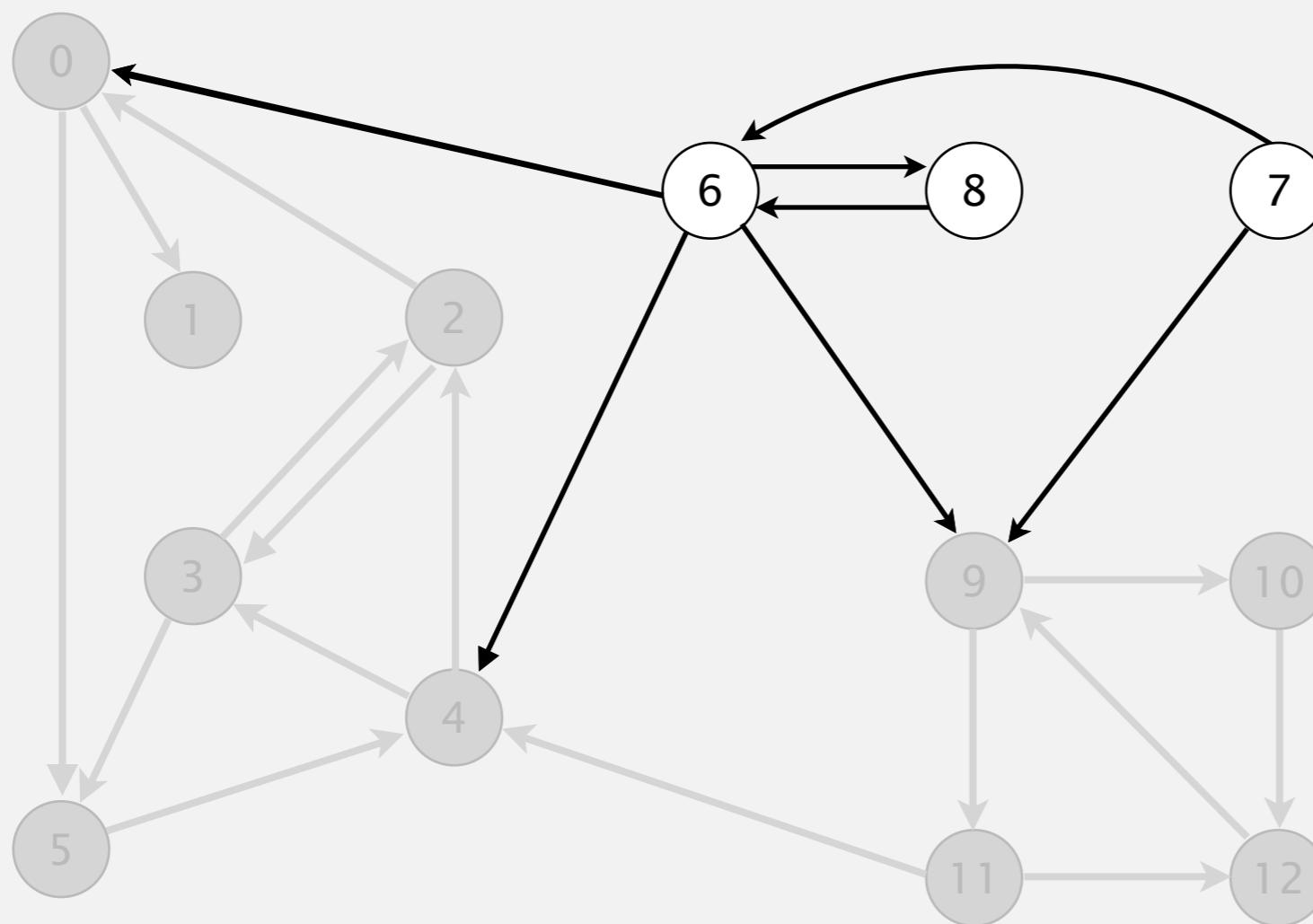


v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	2
10	2
11	2
12	2

**strong component:** 9 10 11 12

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

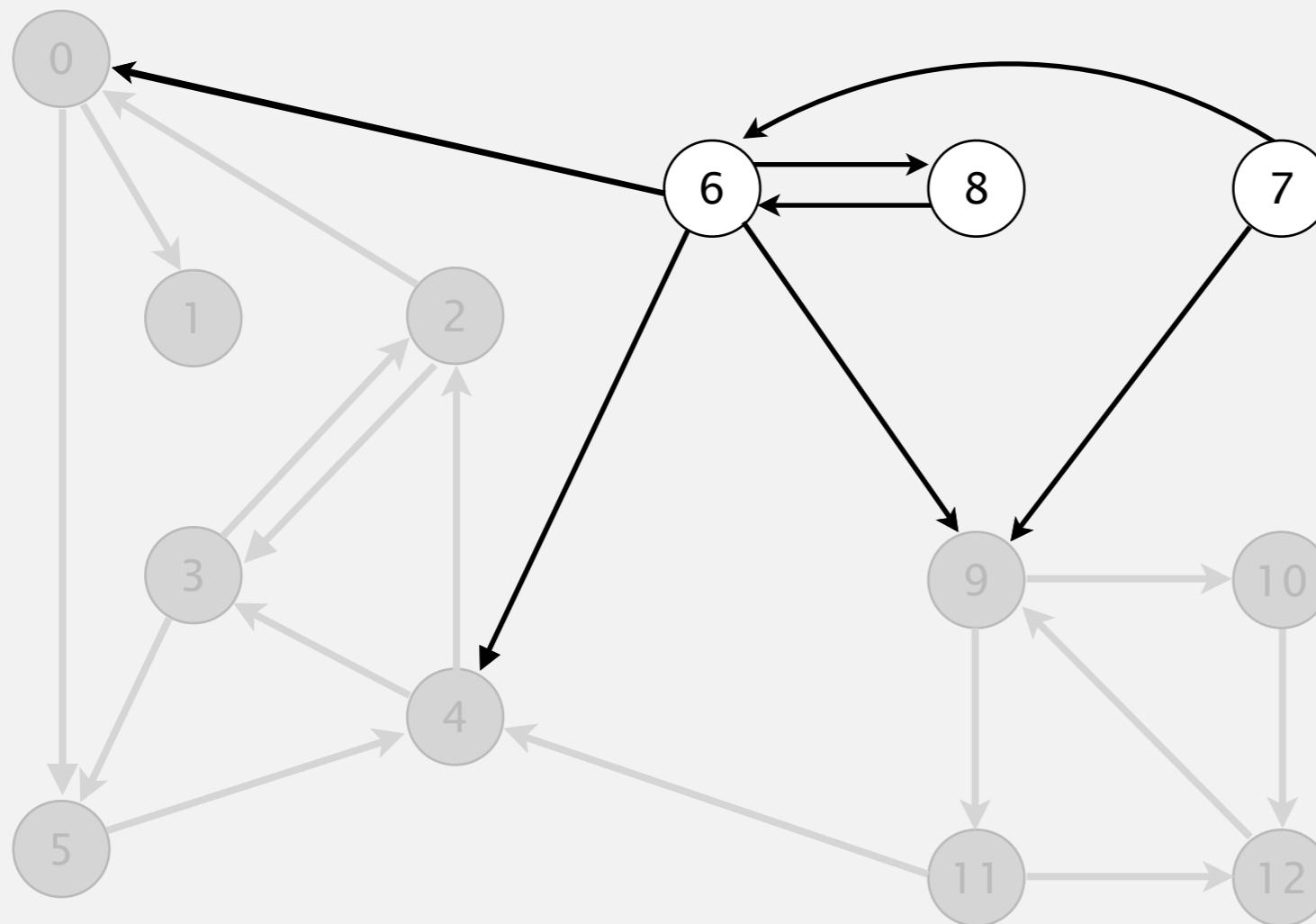


check 9

v	id[]
0	1
1	0
2	1
3	1
4	1
5	-
6	-
7	-
8	-
9	2
10	2
11	2
12	2

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

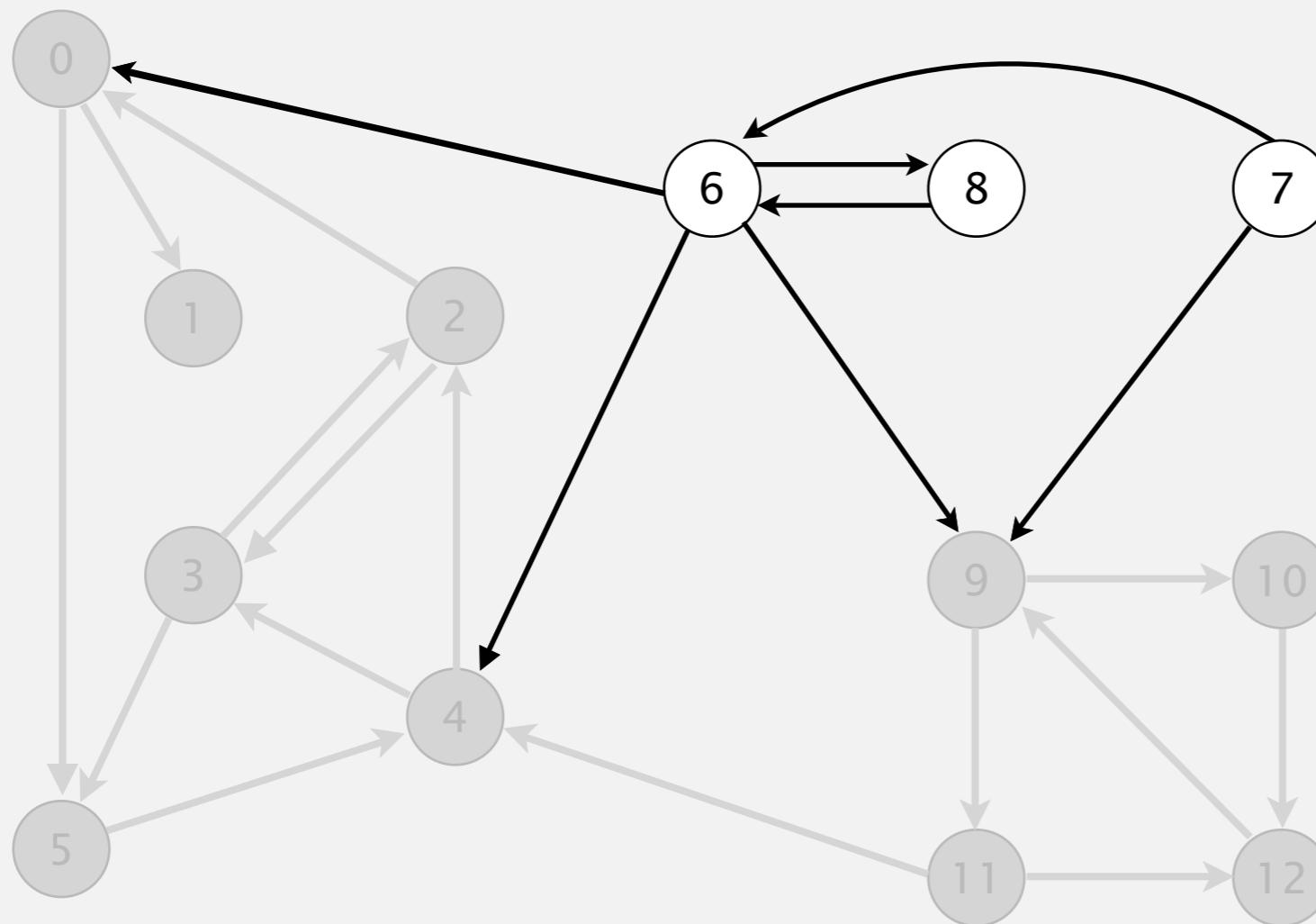


v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	2
10	2
11	2
12	2

check 12

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

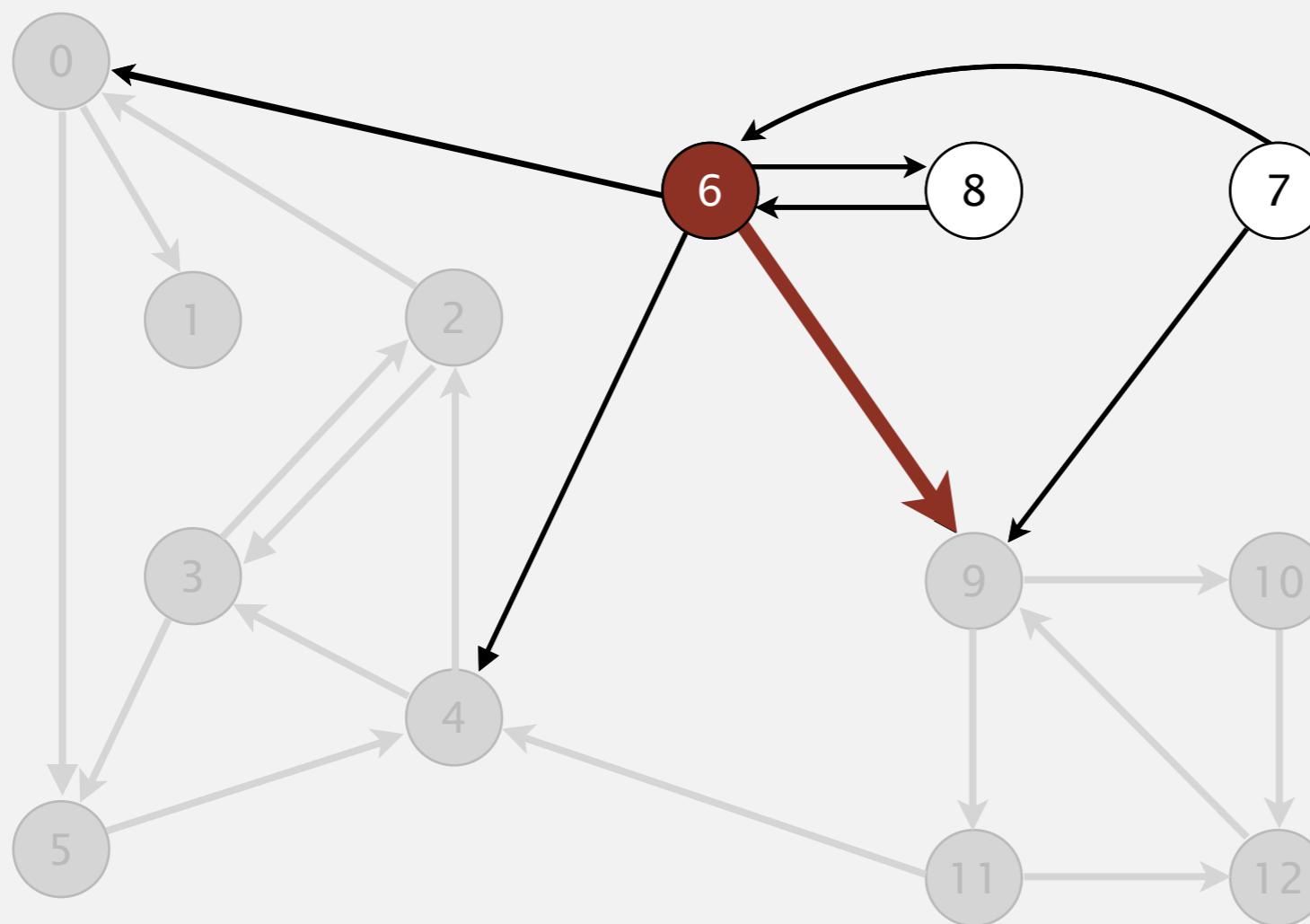


v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	2
10	2
11	2
12	2

check 10

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

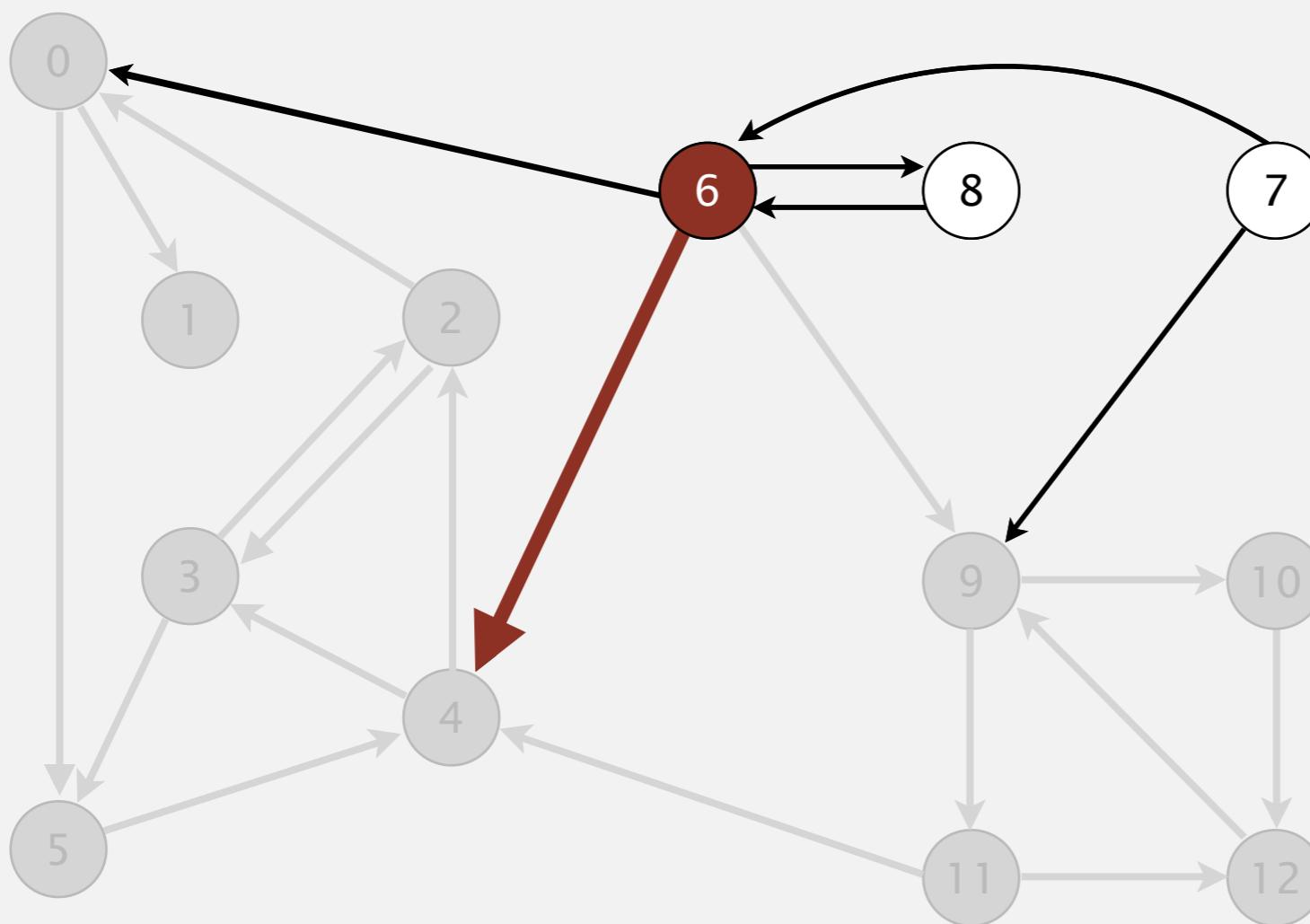


v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	-
8	-
9	2
10	2
11	2
12	2

visit 6: check 9, check 4, check 8, and check 0

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

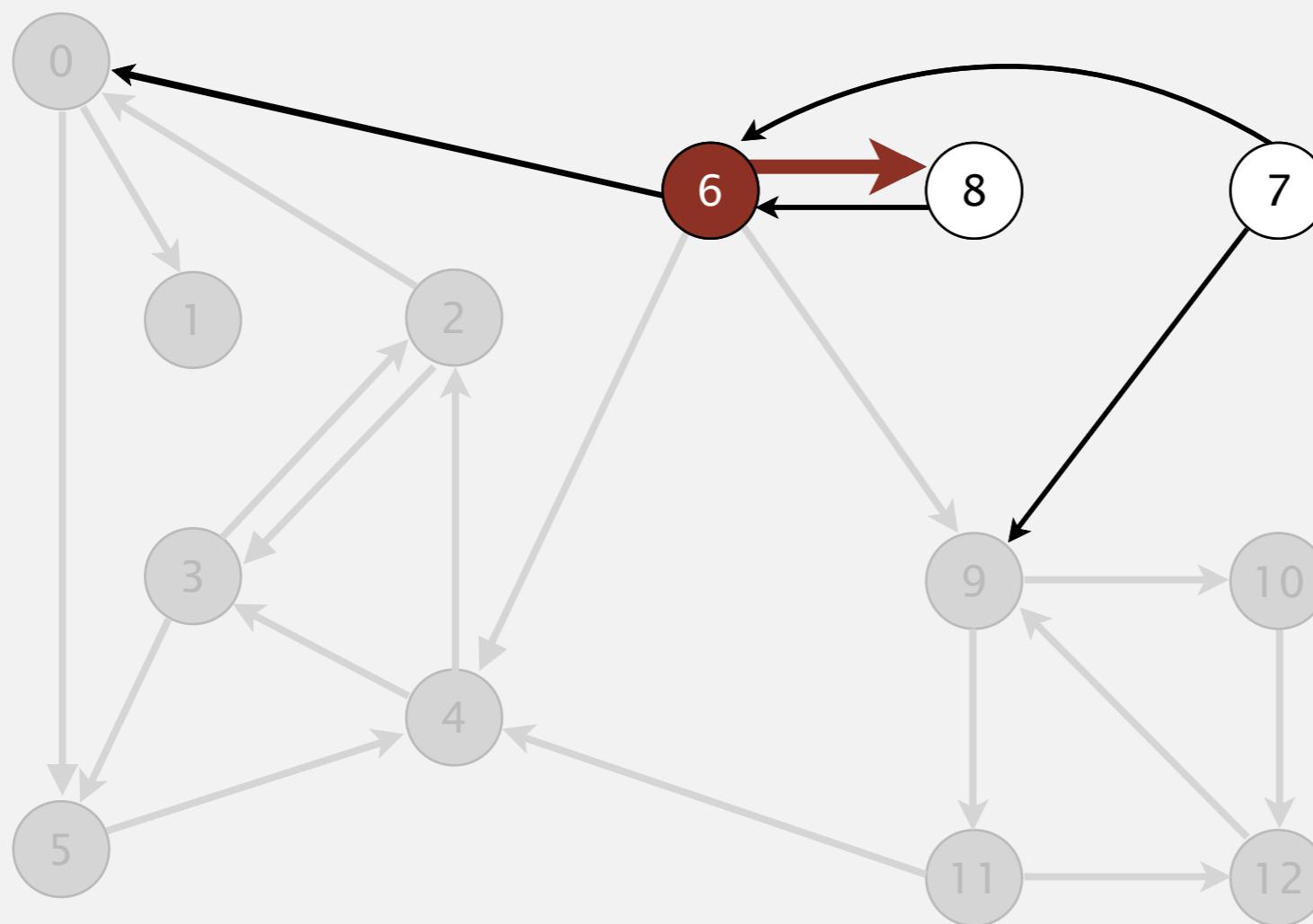


v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	-
8	-
9	2
10	2
11	2
12	2

visit 6: check 9, check 4, check 8, and check 0

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

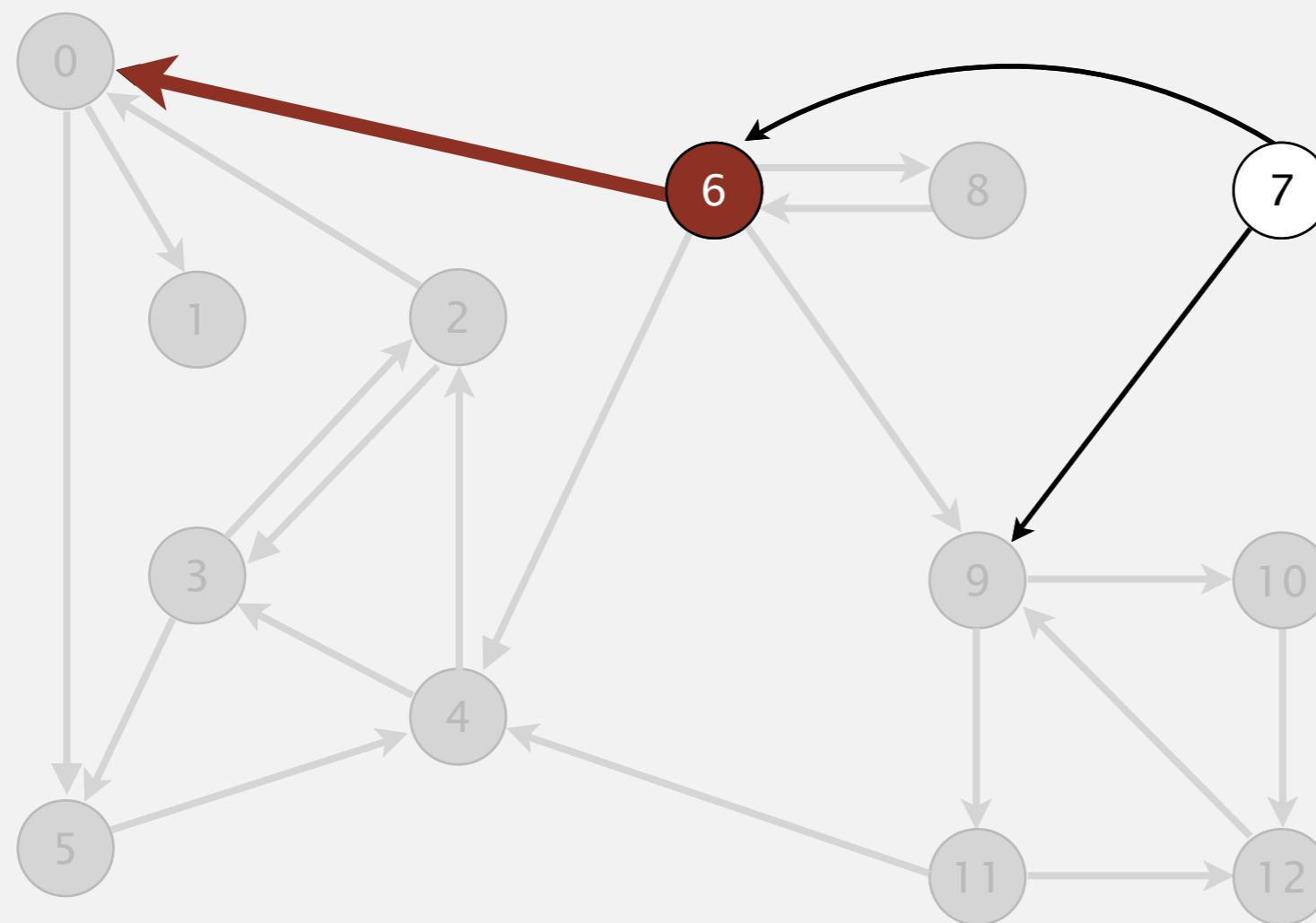


v	id[ ]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	-
8	-
9	2
10	2
11	2
12	2

visit 6: check 9, check 4, check 8, and check 0

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

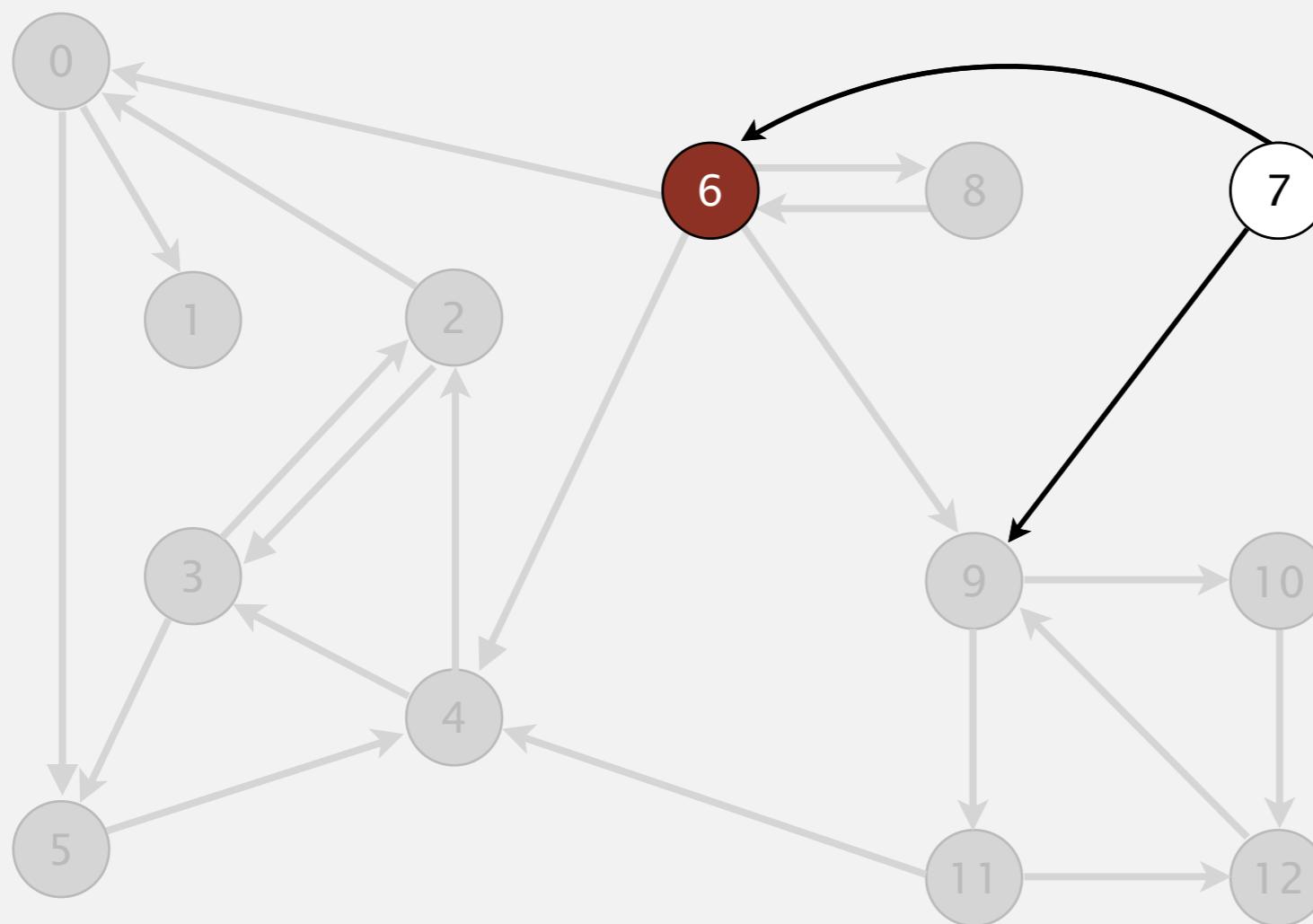


v	id[ ]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	-
8	3
9	2
10	2
11	2
12	2

visit 6: check 9, check 4, check 8, and check 0

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

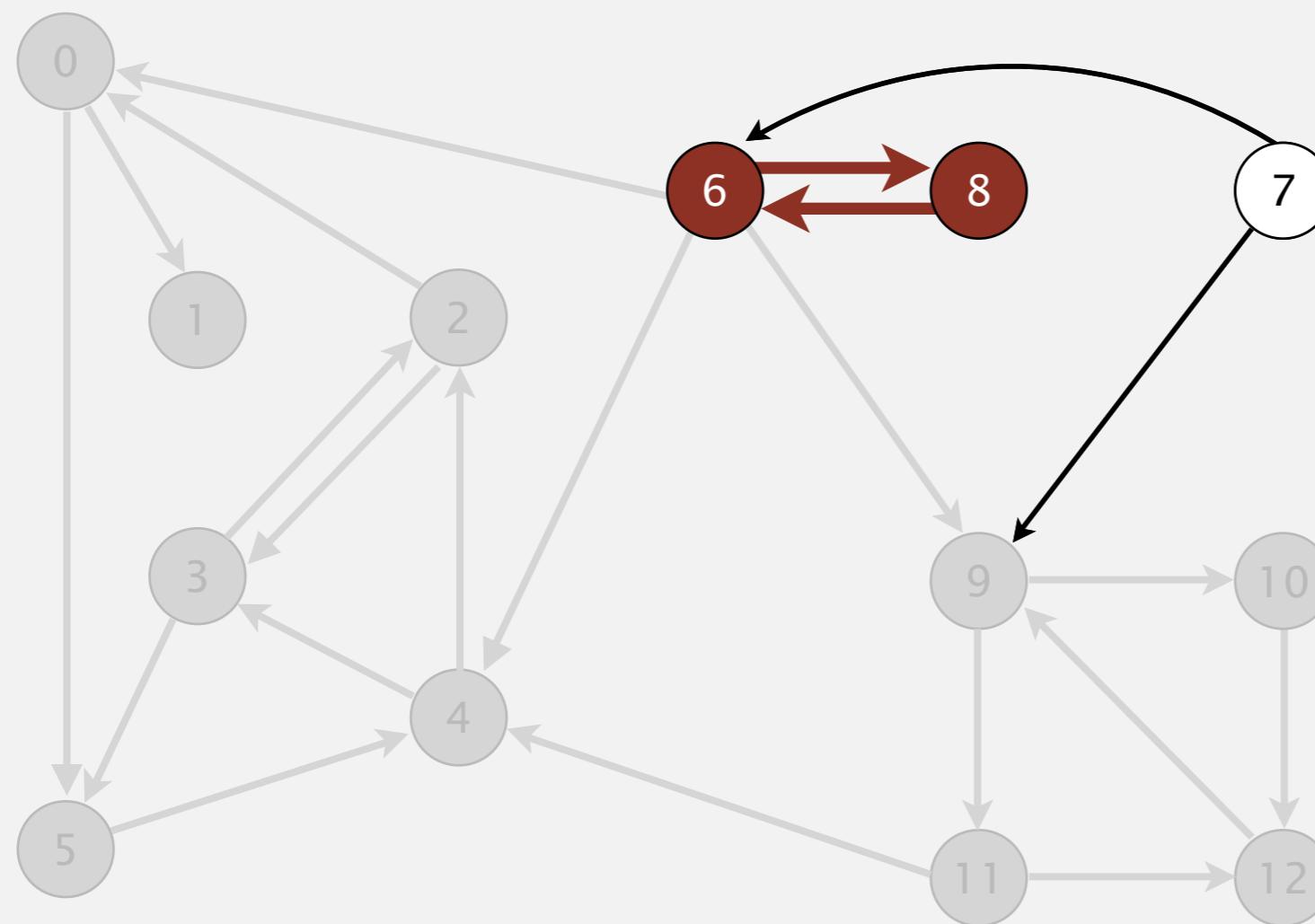


v	id[ ]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	-
8	3
9	2
10	2
11	2
12	2

6 done

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

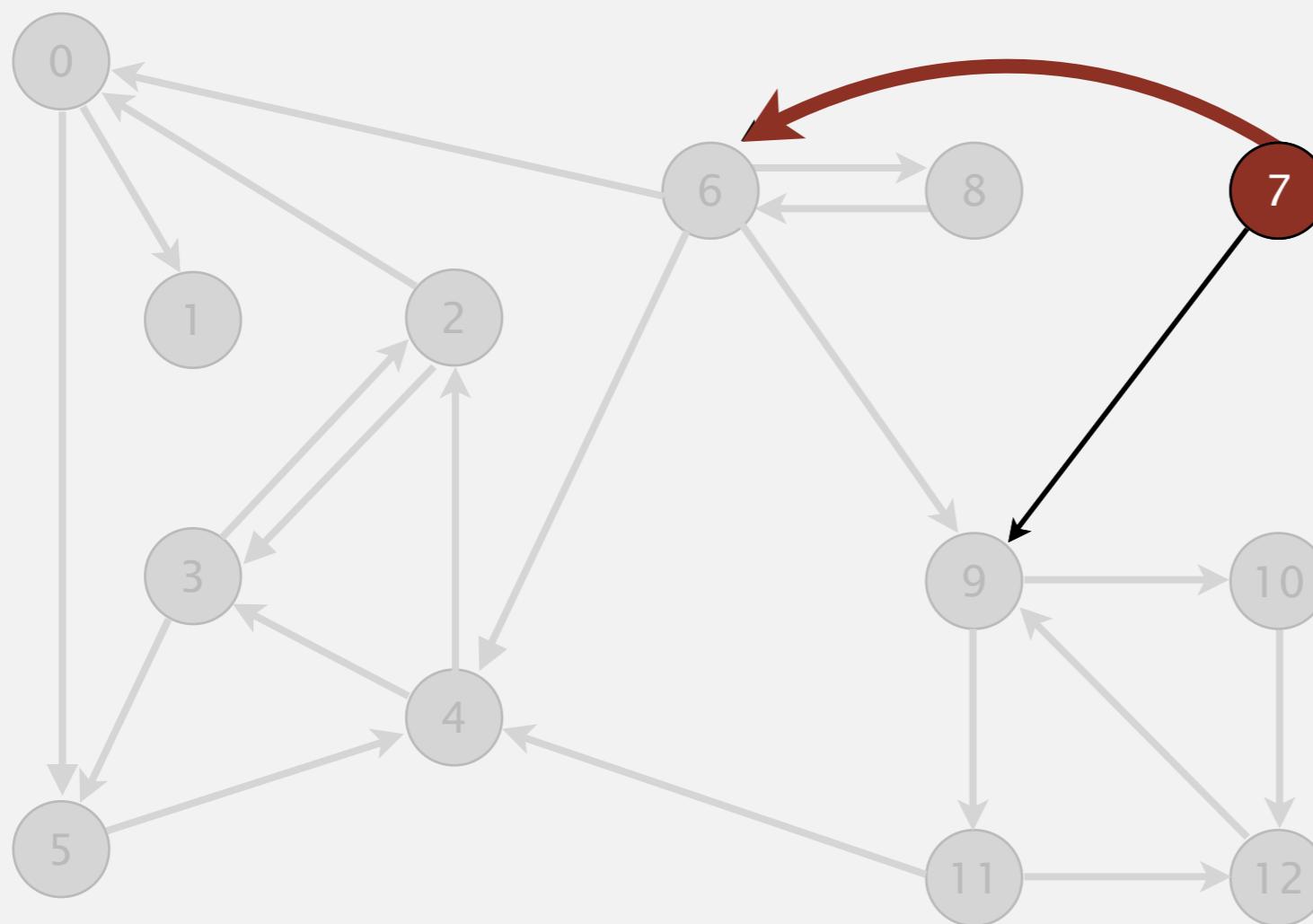


v	id[ ]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	3
8	3
9	2
10	2
11	2
12	2

**strong component:** 6 8

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

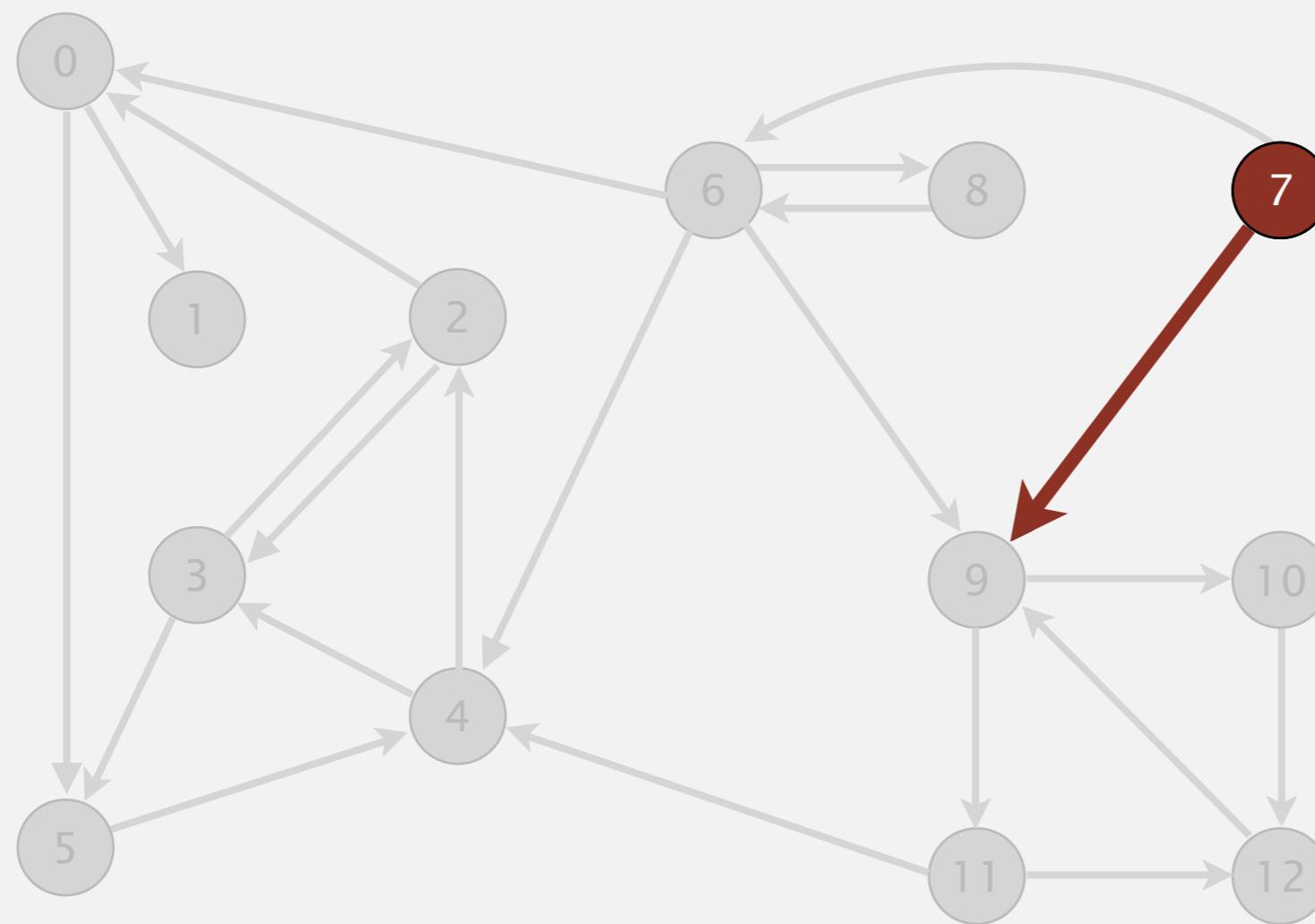


visit 7: check 6 and check 9

v	id[ ]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	4
8	3
9	2
10	2
11	2
12	2

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

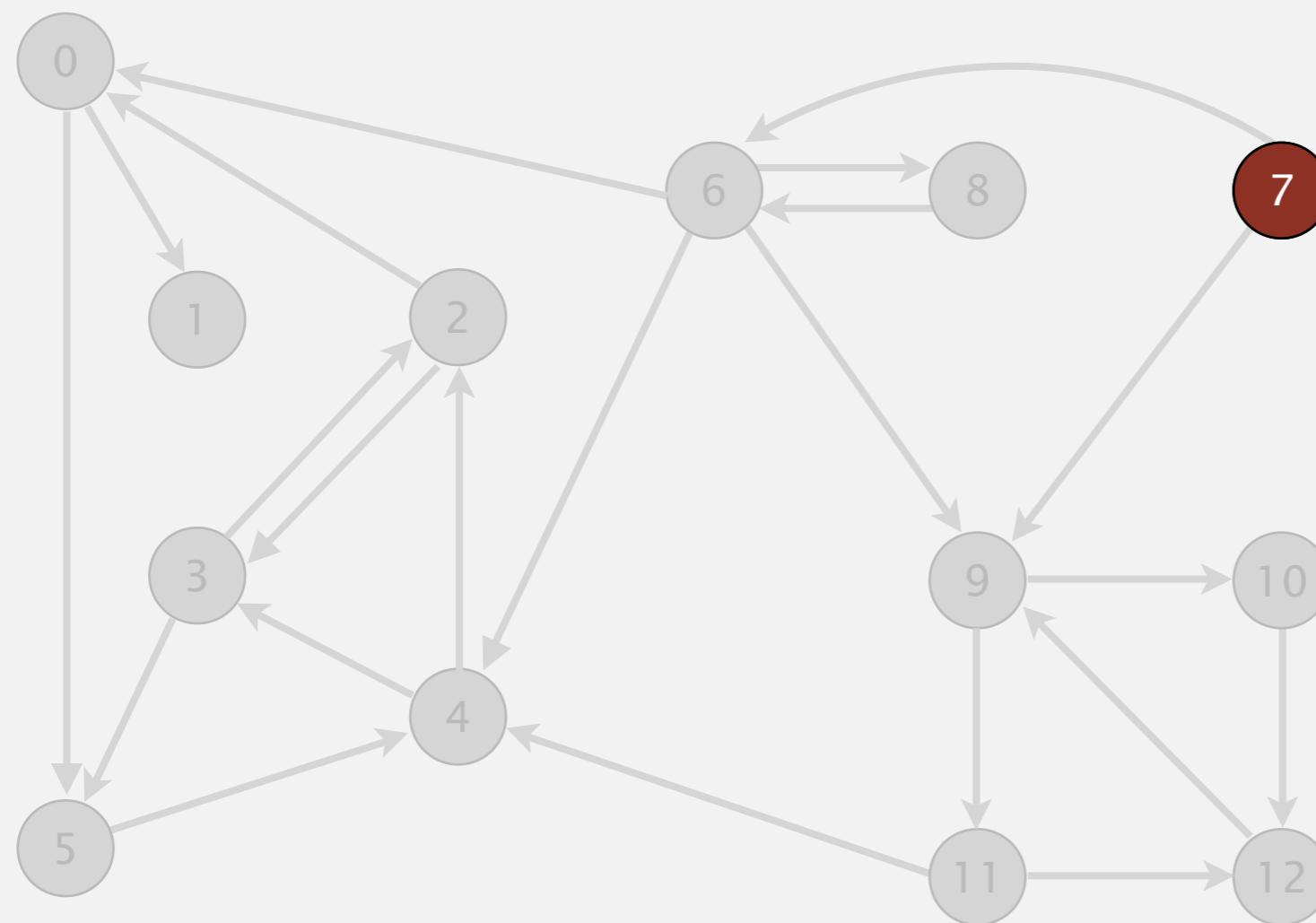


v	id[ ]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	4
8	3
9	2
10	2
11	2
12	2

visit 7: check 6 and check 9

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

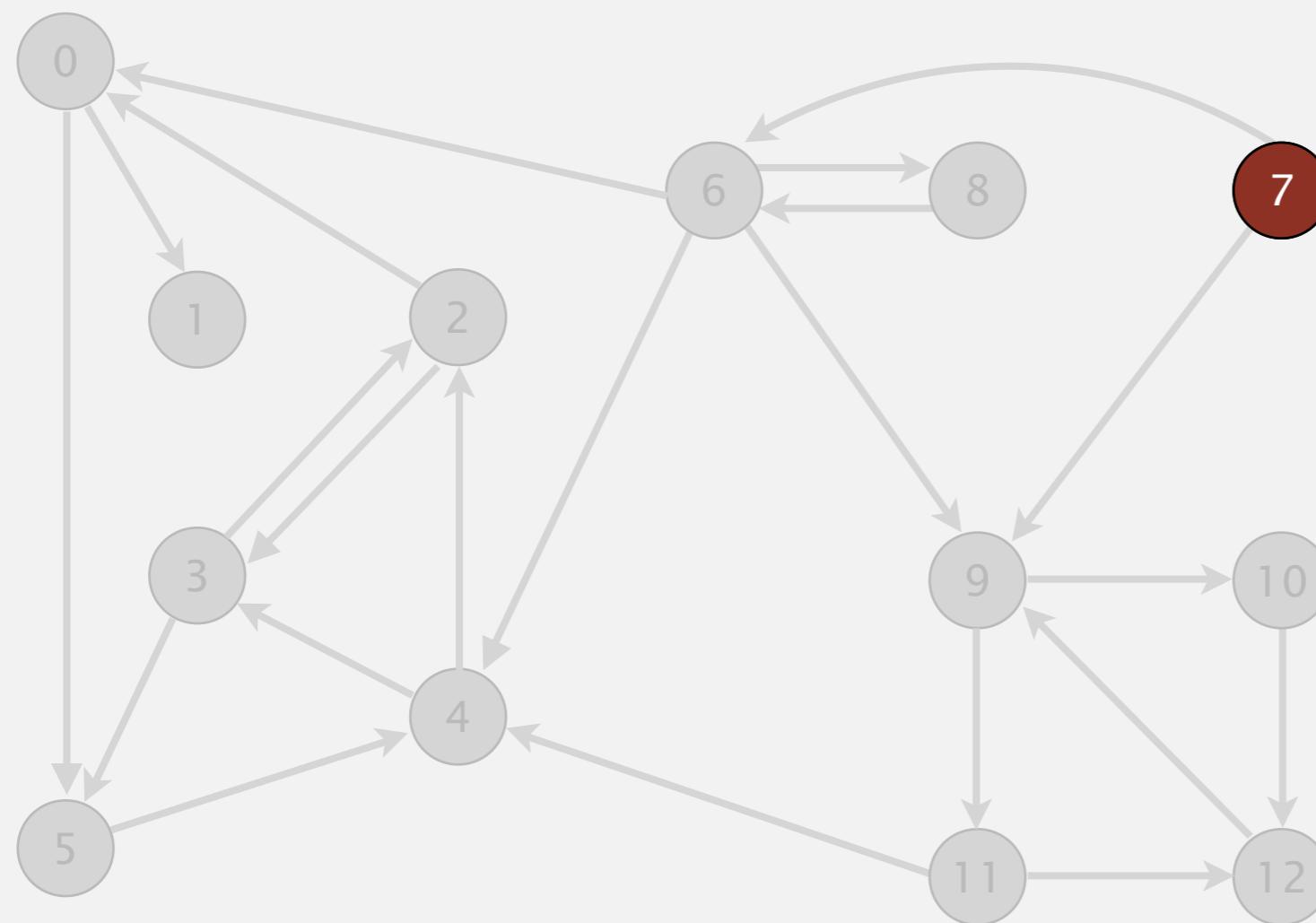


v	id[ ]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	4
8	3
9	2
10	2
11	2
12	2

7 done

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

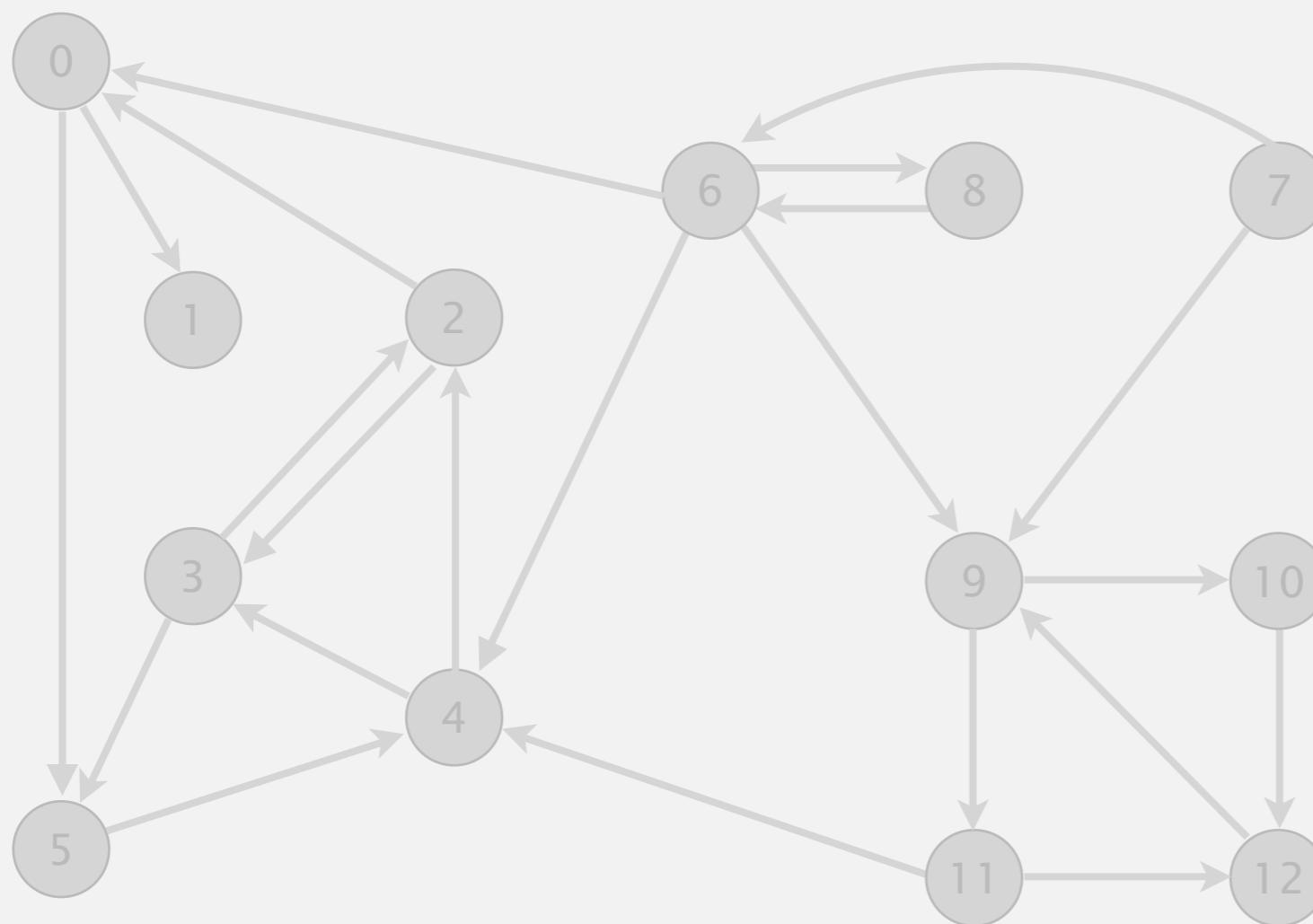


v	id[ ]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	4
8	3
9	2
10	2
11	2
12	2

**strong component:** 7

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .

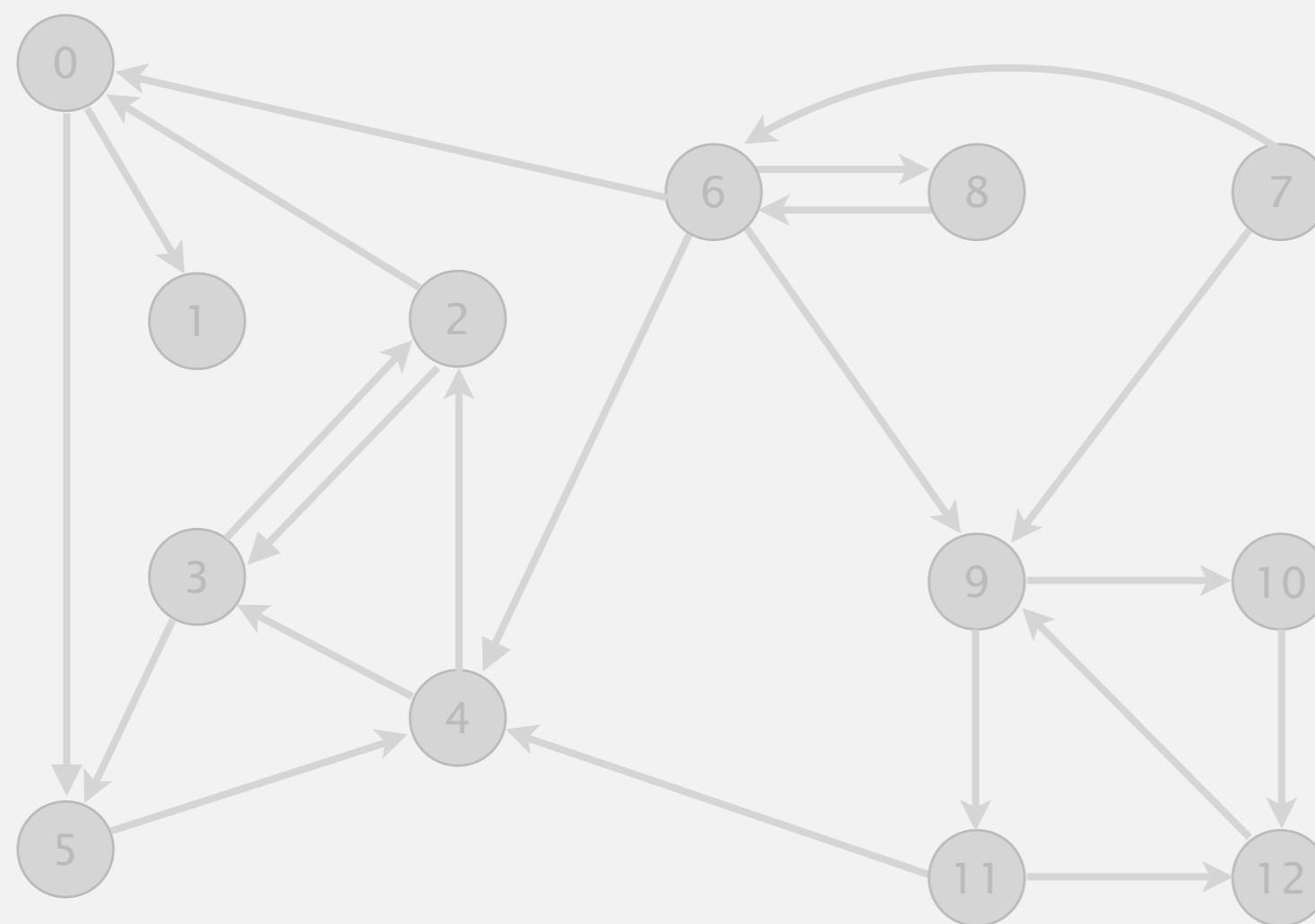


v	id[ ]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	4
8	3
9	2
10	2
11	2
12	2

check 8

# Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in  $G$ , visiting unmarked vertices in reverse postorder of  $G^R$ .



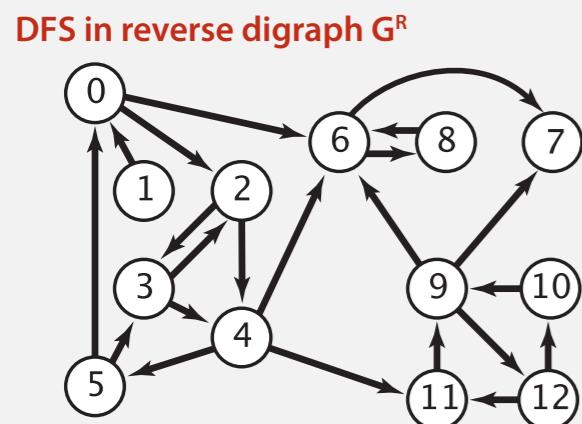
done

v	id[ ]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	4
8	3
9	2
10	2
11	2
12	2

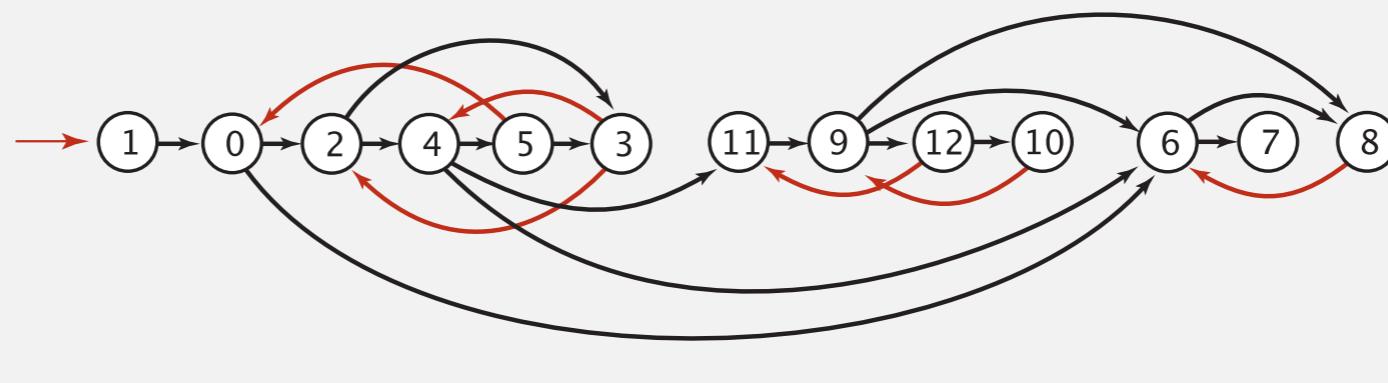
# Kosaraju-Sharir algorithm

Simple (but mysterious) algorithm for computing strong components.

- Phase 1: run DFS on  $G^R$  to compute reverse postorder.
- Phase 2: run DFS on  $G$ , considering vertices in order given by first DFS.



check unmarked vertices in the order  
0 1 2 3 4 5 6 7 8 9 10 11 12



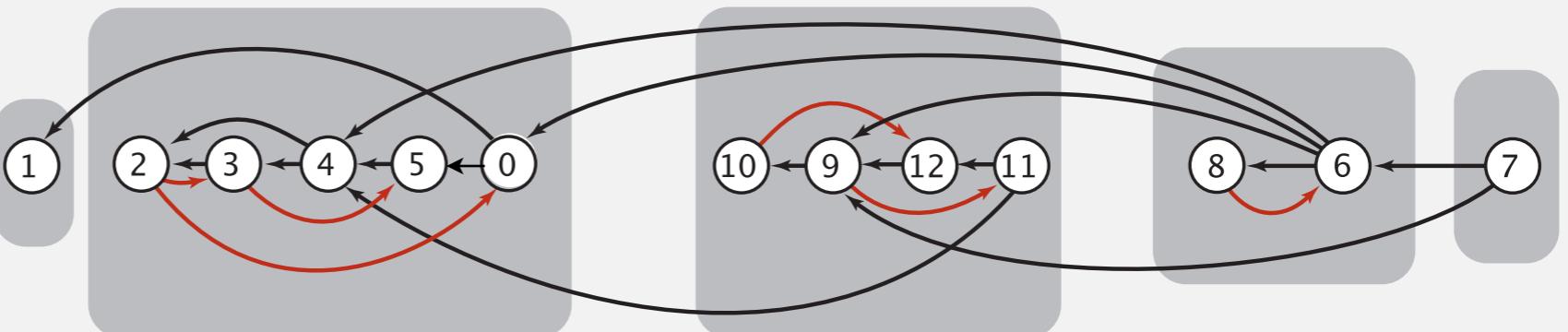
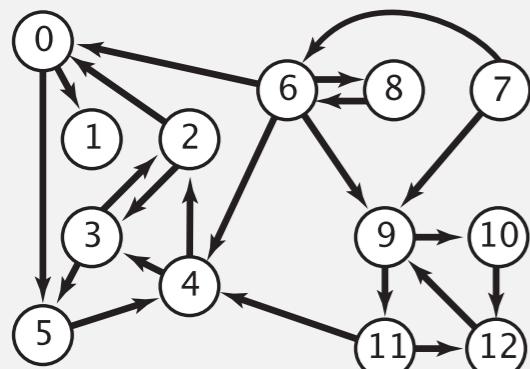
dfs(0)  
  dfs(6)  
    dfs(8)  
      | check 6  
      8 done  
      dfs(7)  
      7 done  
    6 done  
    dfs(2)  
      dfs(4)  
        dfs(11)  
          dfs(9)  
          dfs(12)  
          | check 11  
          dfs(10)  
          | check 9  
          10 done  
        12 done  
        check 7  
        check 6  
  ...

# Kosaraju-Sharir algorithm

Simple (but mysterious) algorithm for computing strong components.

- Phase 1: run DFS on  $G^R$  to compute reverse postorder.
- Phase 2: run DFS on  $G$ , considering vertices in order given by first DFS.

DFS in original digraph  $G$



check unmarked vertices in the order

1 0 2 4 5 3 11 9 12 10 6 7 8

↑↑ ↑ ↑↑

dfs(1)  
1 done

dfs(0)  
dfs(5)  
dfs(4)  
dfs(3)  
check 5  
dfs(2)  
check 0  
check 3  
2 done  
3 done  
check 2  
4 done  
5 done  
check 1  
0 done  
check 2  
check 4  
check 5  
check 3

dfs(11)  
check 4  
dfs(12)  
dfs(9)  
check 11  
dfs(10)  
check 12  
10 done  
9 done  
12 done  
11 done  
check 9  
check 12  
check 10

dfs(6)  
check 9  
check 4  
dfs(8)  
check 6  
8 done  
check 0  
6 done

dfs(7)  
check 6  
check 9  
7 done  
check 8

## Kosaraju-Sharir algorithm

---

**Proposition.** Kosaraju-Sharir algorithm computes the strong components of a digraph in time proportional to  $E + V$ .

Pf.

- Running time: bottleneck is running DFS twice (and computing  $G^R$ ).
- Correctness: tricky, see textbook (2<sup>nd</sup> printing).
- Implementation: easy!

# Connected components in an undirected graph (with DFS)

---

```
public class CC
{
    private boolean marked[];
    private int[] id;
    private int count;

    public CC(Graph G)
    {
        marked = new boolean[G.V()];
        id = new int[G.V()];

        for (int v = 0; v < G.V(); v++)
        {
            if (!marked[v])
            {
                dfs(G, v);
                count++;
            }
        }
    }

    private void dfs(Graph G, int v)
    {
        marked[v] = true;
        id[v] = count;
        for (int w : G.adj(v))
            if (!marked[w])
                dfs(G, w);
    }

    public boolean connected(int v, int w)
    { return id[v] == id[w]; }
}
```

# Strong components in a digraph (with two DFSs)

```
public class KosarajuSharirSCC
{
    private boolean marked[];
    private int[] id;
    private int count;

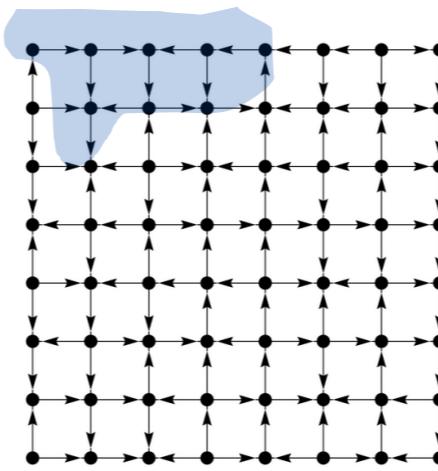
    public KosarajuSharirSCC(Digraph G)
    {
        marked = new boolean[G.V()];
        id = new int[G.V()];
        DepthFirstOrder dfs = new DepthFirstOrder(G.reverse());
        for (int v : dfs.reversePostorder())
        {
            if (!marked[v])
            {
                dfs(G, v);
                count++;
            }
        }
    }

    private void dfs(Digraph G, int v)
    {
        marked[v] = true;
        id[v] = count;
        for (int w : G.adj(v))
            if (!marked[w])
                dfs(G, w);
    }

    public boolean stronglyConnected(int v, int w)
    { return id[v] == id[w]; }
}
```

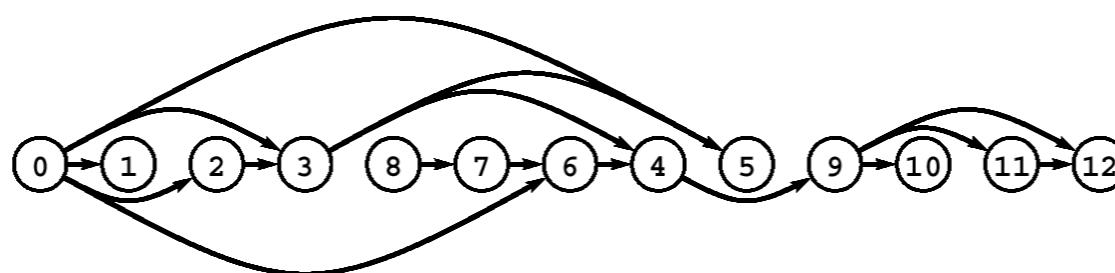
# Digraph-processing summary: algorithms of the day

**single-source  
reachability  
in a digraph**



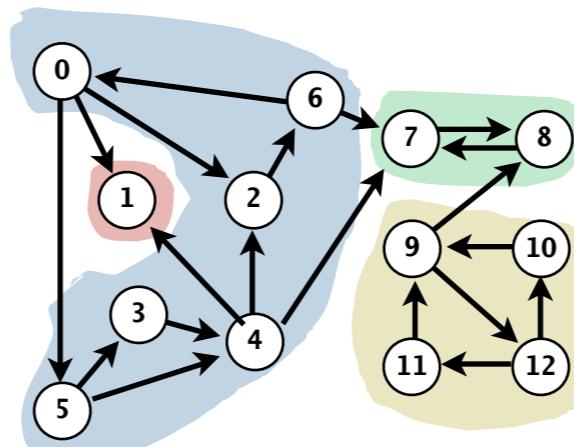
DFS

**topological sort  
in a DAG**



DFS

**strong  
components  
in a digraph**



Kosaraju-Sharir  
DFS (twice)