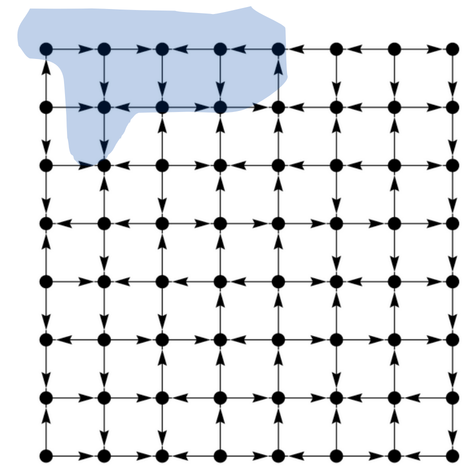
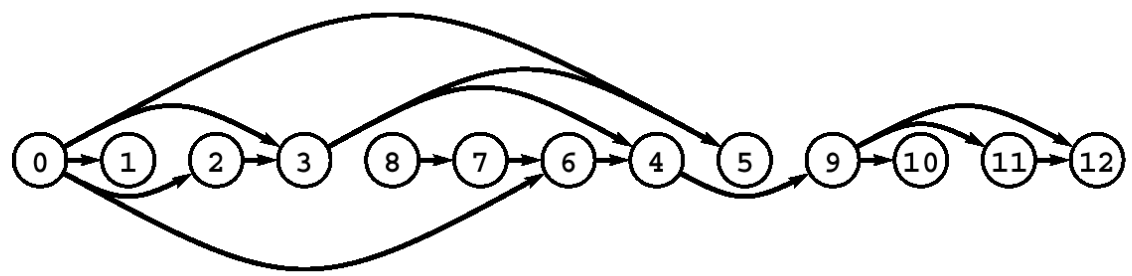
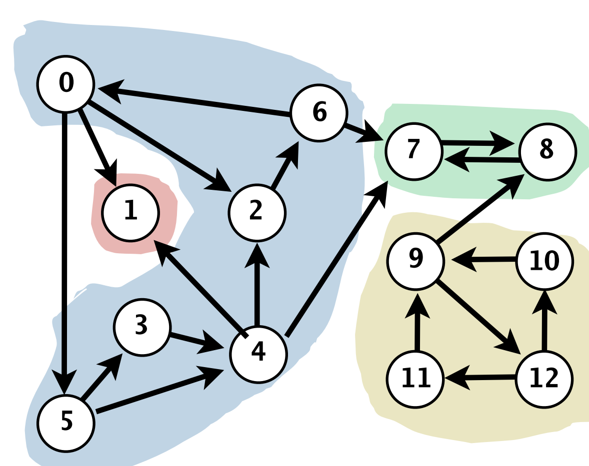
Digraph-processing summary: algorithms of the day

single-source reachability in a digraph		DFS
topological sort in a DAG		DFS
strong components in a digraph		Kosaraju-Sharir DFS (twice)

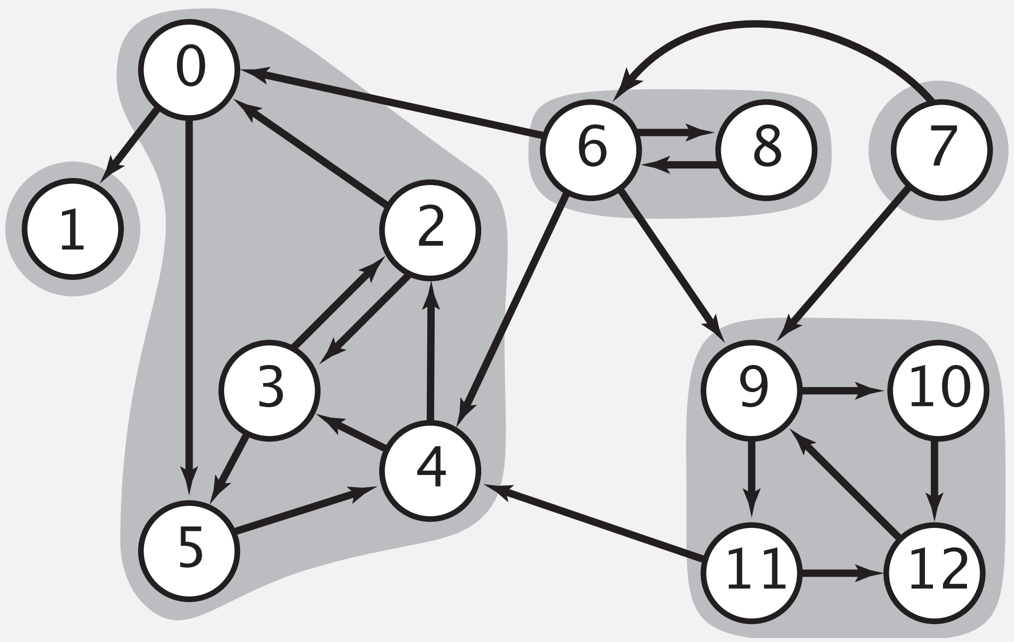
Strongly-connected components

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.



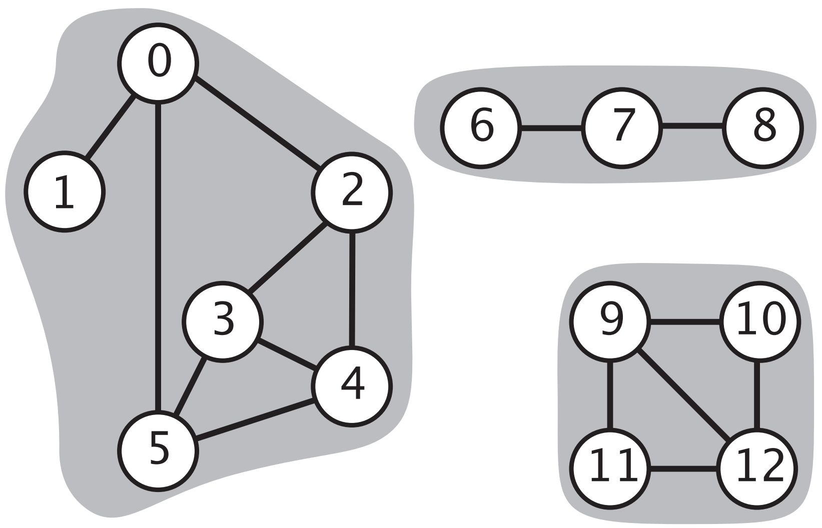
5 strongly-connected components

HOW IS THIS DIFFERENT FROM UNION FIND?

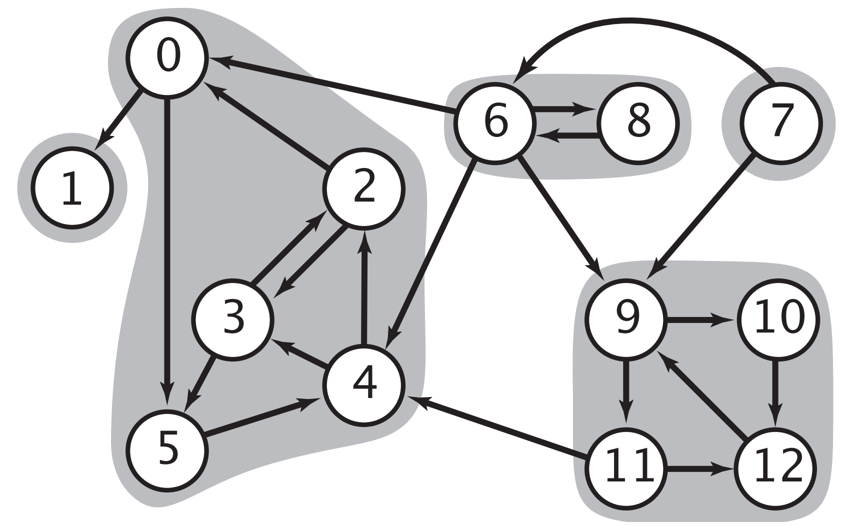
Connected components vs. strongly-connected components

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
id[]	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
id[]	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];  }
```

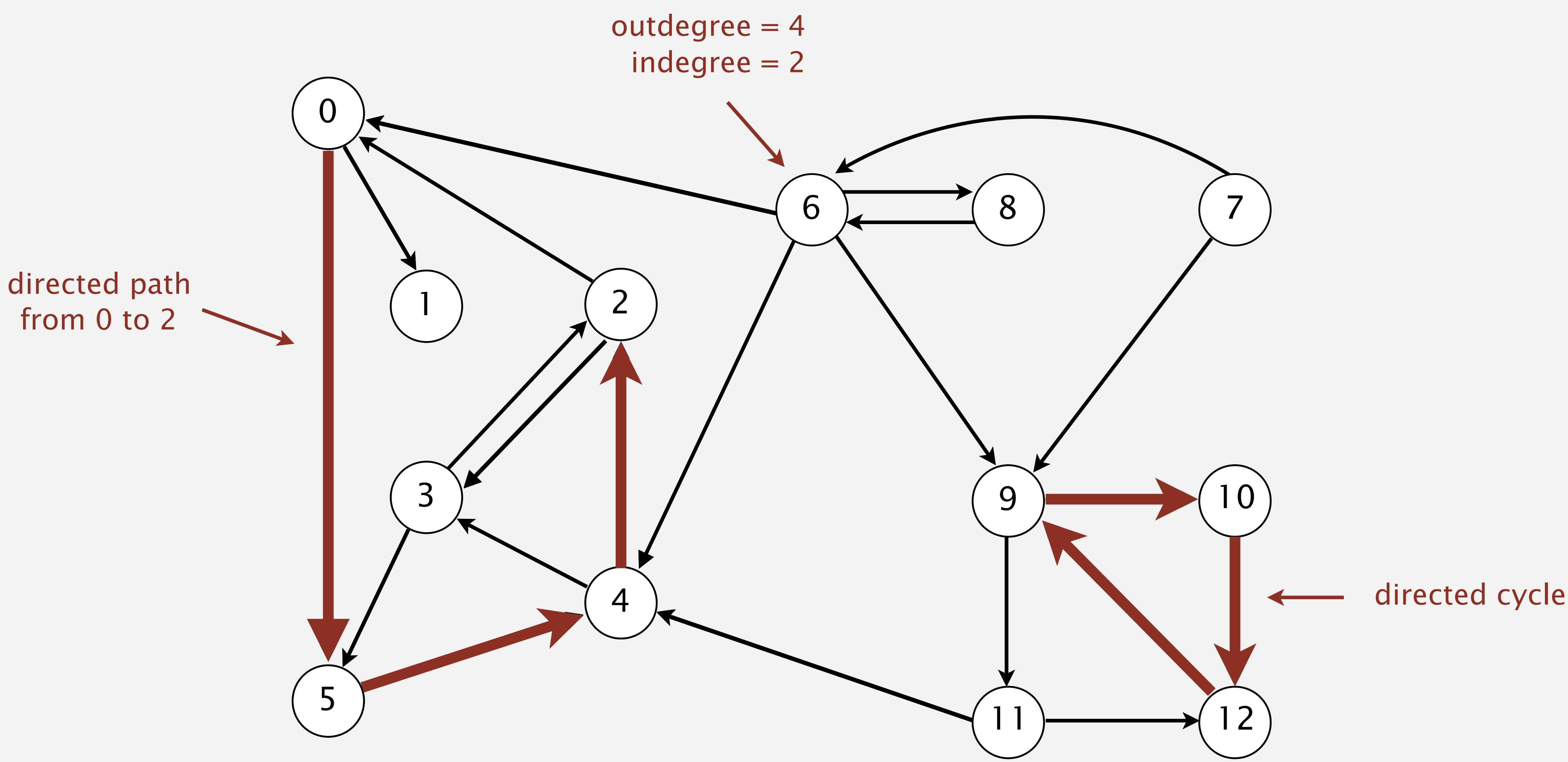
constant-time client connectivity query

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

constant-time client strong-connectivity query

Directed graphs

Digraph. Set of vertices connected pairwise by directed edges.



Some graph-processing problems

Path. Is there a path between s and t?

Shortest path. What is the shortest path between s and t?

Cycle. Is there a cycle in the graph?

Euler cycle. Is there a cycle that uses each edge exactly once?

Hamilton cycle. Is there a cycle that uses each vertex exactly once.

Connectivity. Is there a way to connect all of the vertices?

(Min Spanning Tree). What is the best way to connect all of the vertices?

Biconnectivity. Is there a vertex whose removal disconnects the graph?

Planarity. Can you draw the graph in the plane with no crossing edges

Graph isomorphism. Do two adjacency lists represent the same graph?

DIRECTED GRAPH API

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→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*

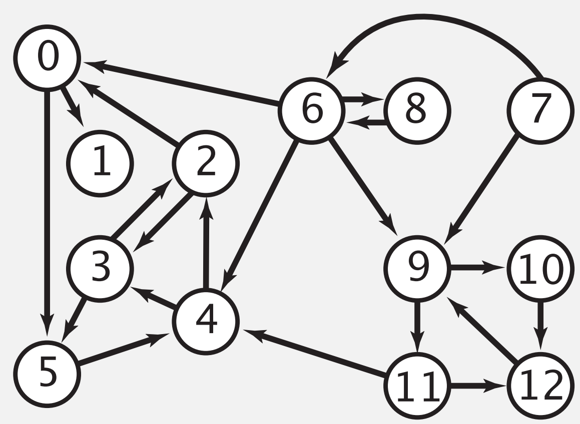
Digraph API

tinyDG.txt

$V \rightarrow$ 13

22 $\leftarrow E$

```
 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
```

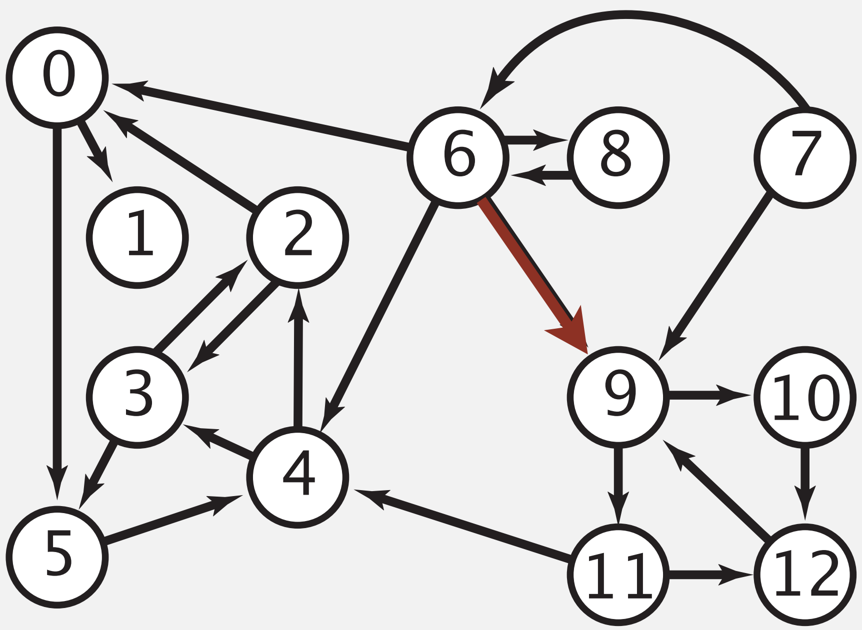
```
In in = new In(args[0]);
Digraph G = new Digraph(in);

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

read digraph from input stream

print out each edge (once)

Digraph representation: set of edges

Store a list of the edges (linked list or array).

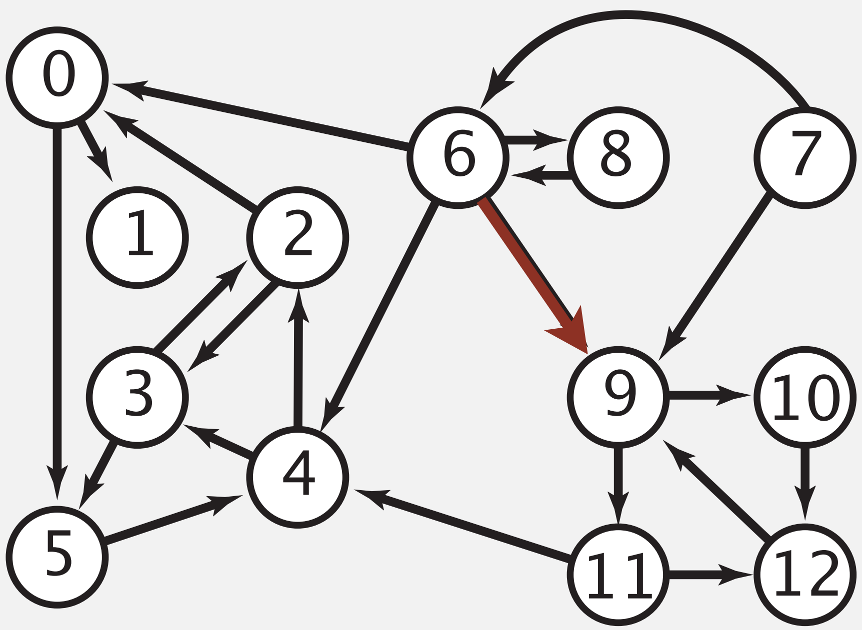


0	1
0	5
2	0
2	3
3	2
3	5
4	2
4	3
5	4
6	0
6	4
6	8
6	9
7	6
7	9
8	6
9	10
9	11
10	12
11	4
11	12
12	9

Digraph representation: adjacency matrix

Maintain a two-dimensional V-by-V boolean array;

for each edge $v \rightarrow w$ in the digraph: `adj[v][w] = true`.



to

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

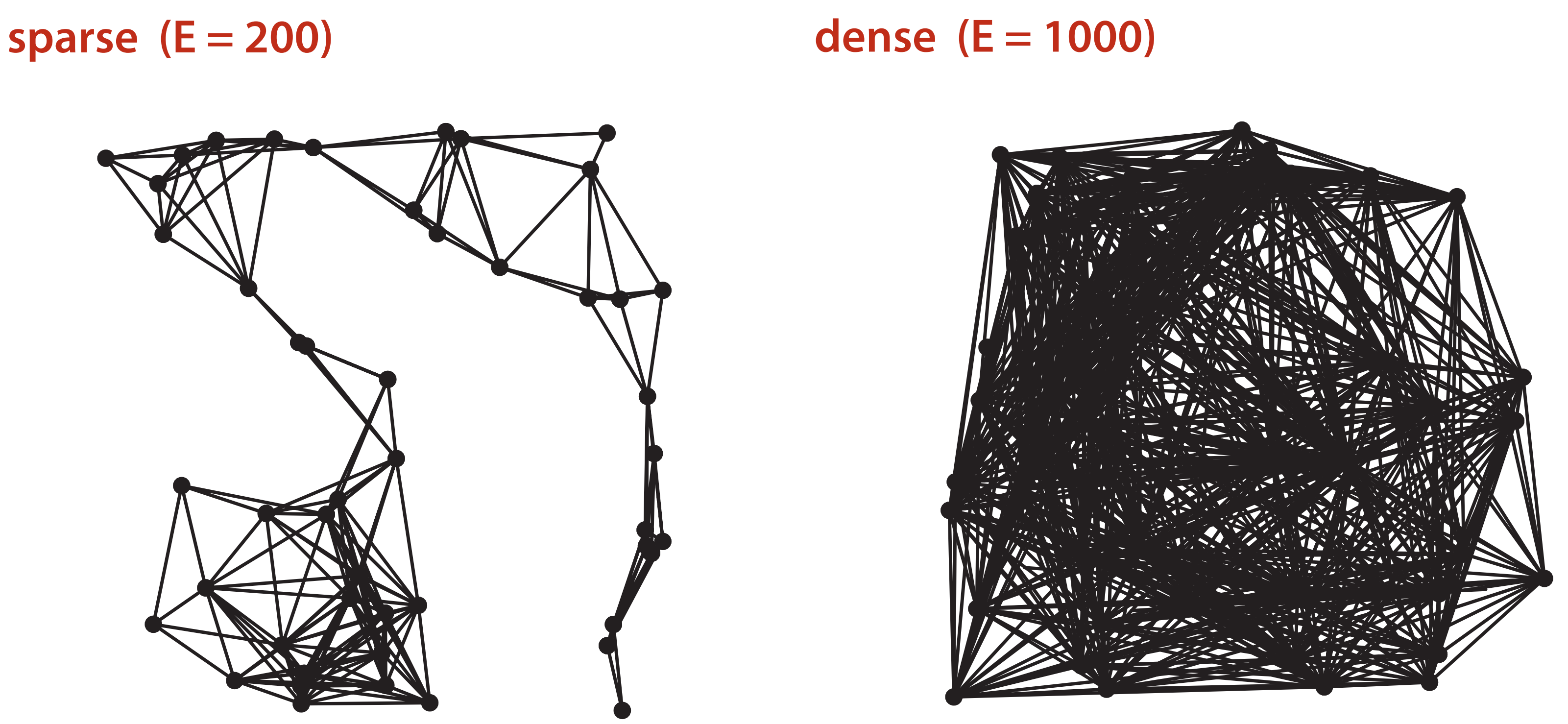
A-lot of empty space

Graph representations

In practice. Use adjacency-lists representation.

- Real-world graphs tend to be sparse.

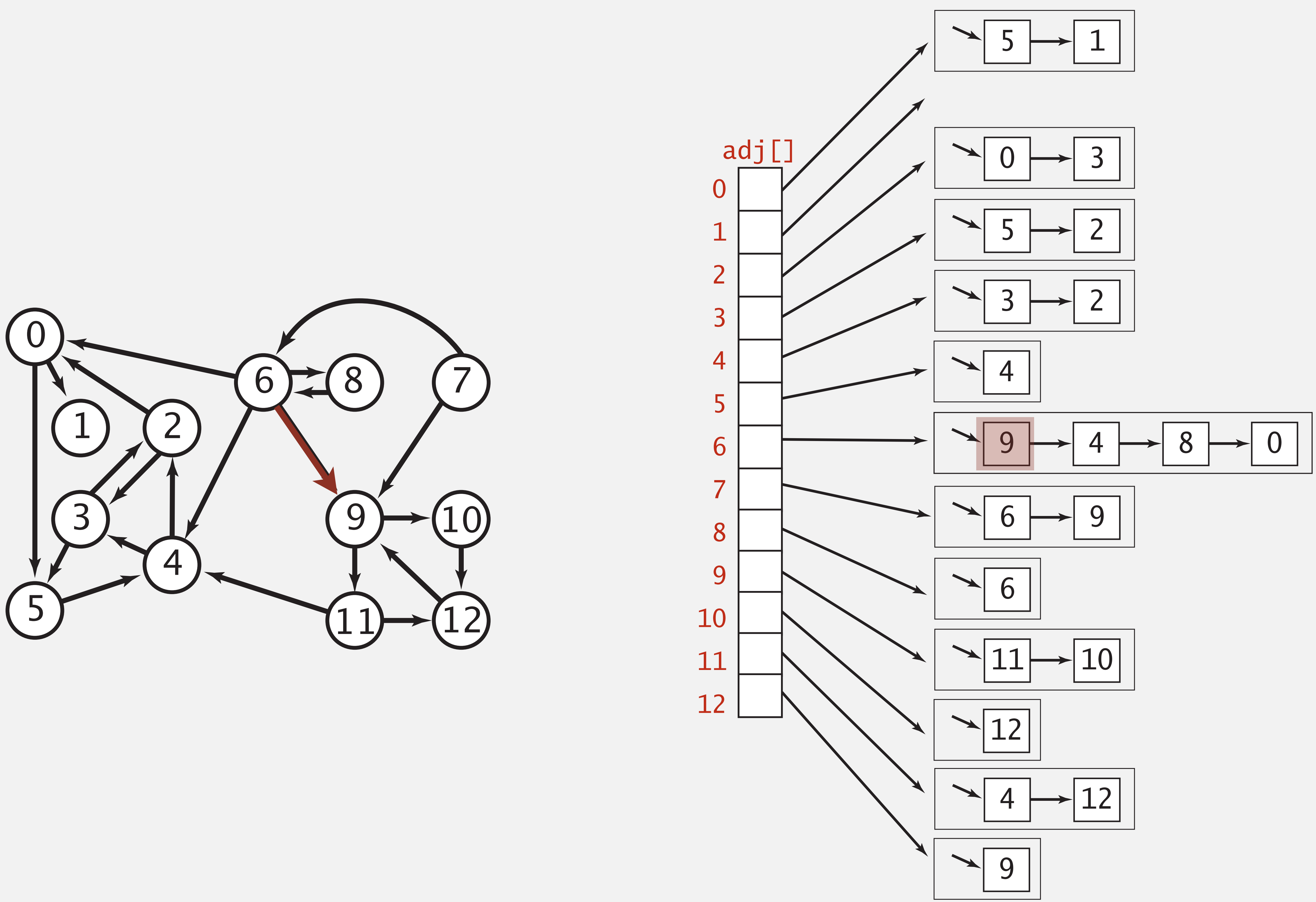
huge number of vertices,
small average vertex degree



Two graphs (V = 50)

Digraph representation: adjacency lists

Maintain vertex-indexed array of lists.



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	1	V
adjacency lists	$E + V$	1	$outdegree(v)$	$outdegree(v)$

Adjacency-lists graph representation (review): Java implementation

```
public class Digraph
{
   private final int V;
   private final int[] adj;

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

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

   public Iterable<Integer> adj(int v)
   {  return adj[v];  }
}
```

adjacency lists

create empty graph with V vertices

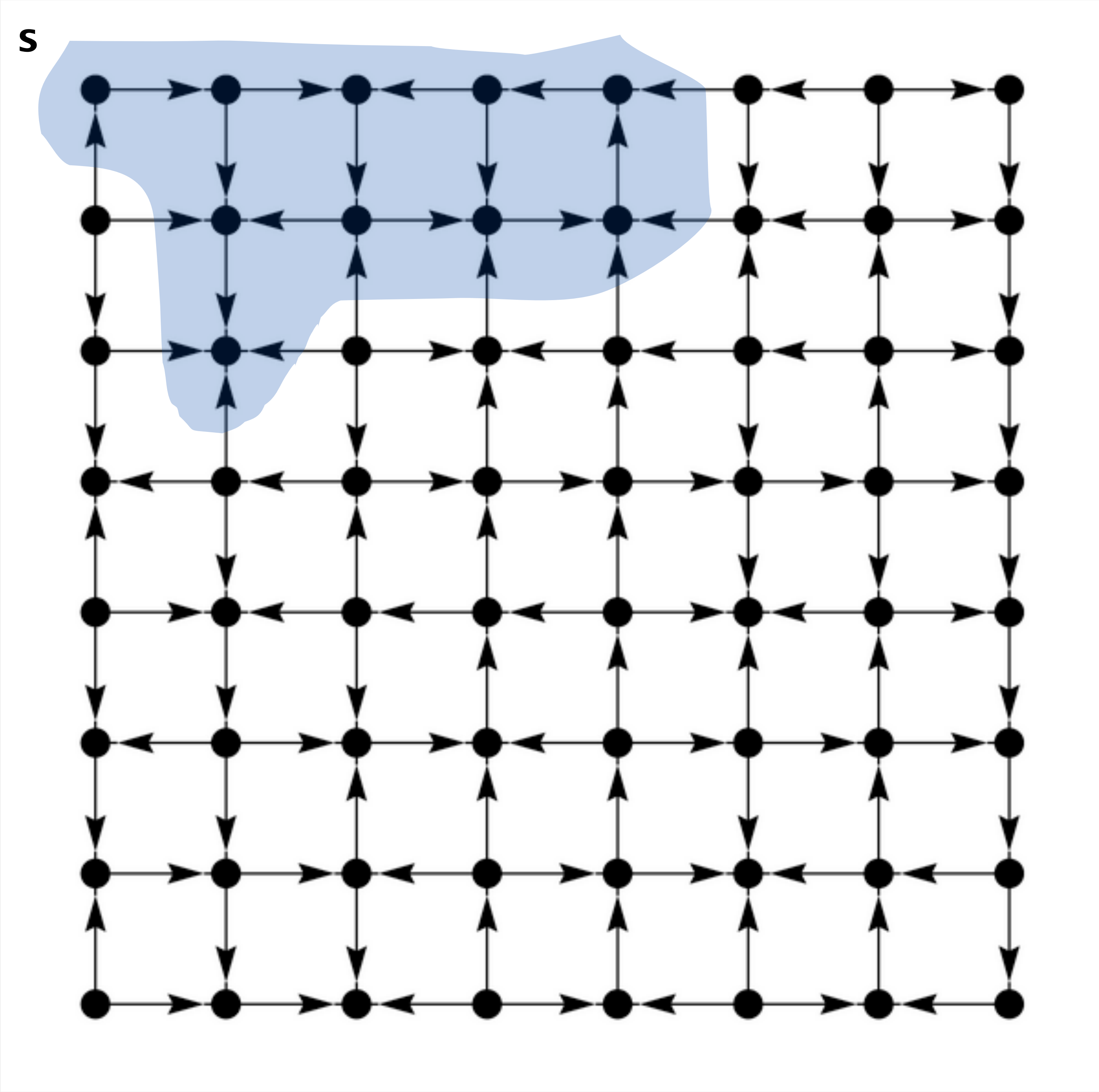
add edge v–w

iterator for vertices adjacent to v

SEARCHING A DIRECTED GRAPH

Reachability

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



Depth-first search in digraphs (review 2150)

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)

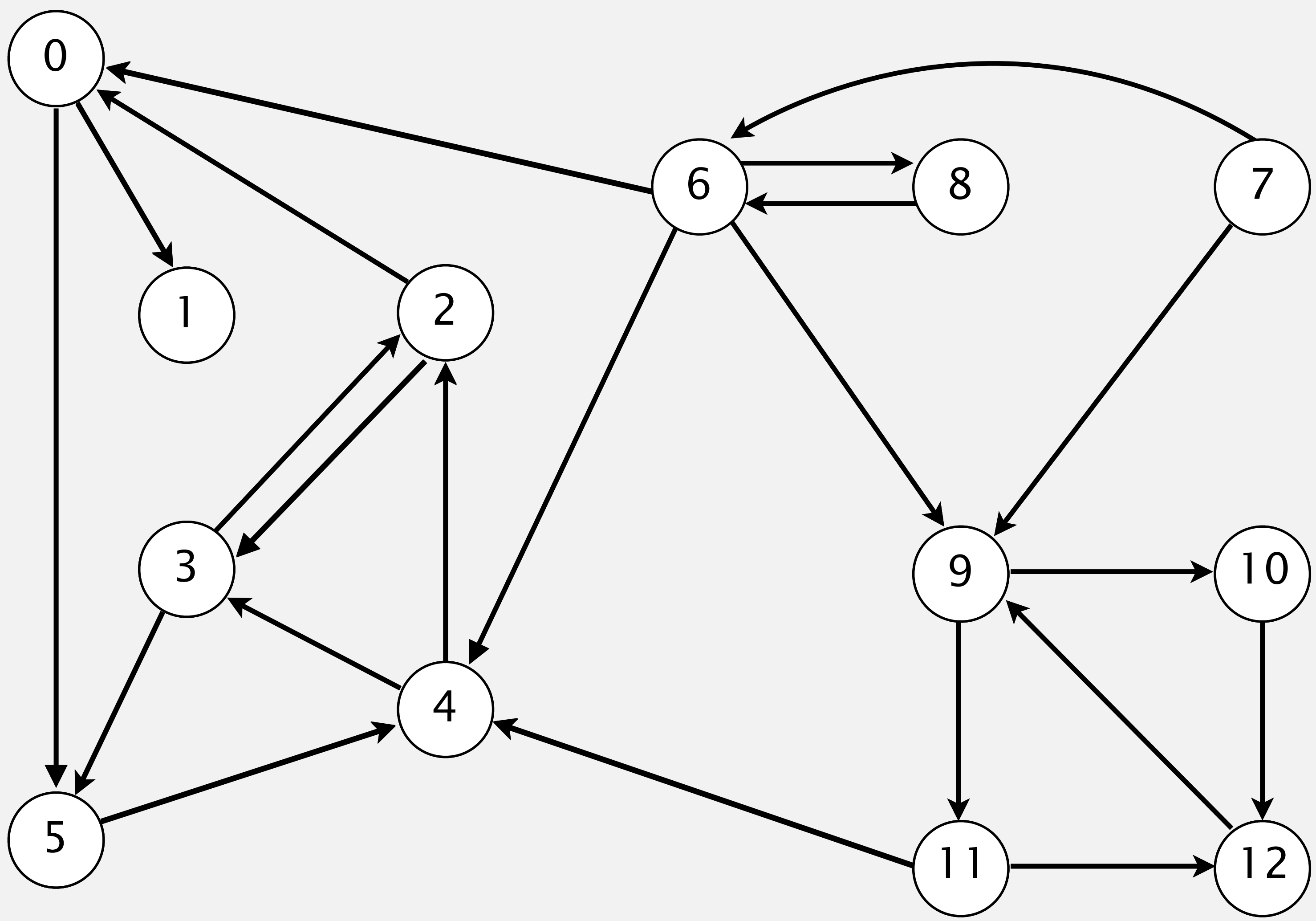
Mark v as visited.

Recursively visit all unmarked
vertices w pointing from v.

Depth-first search demo

To visit a vertex v :

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



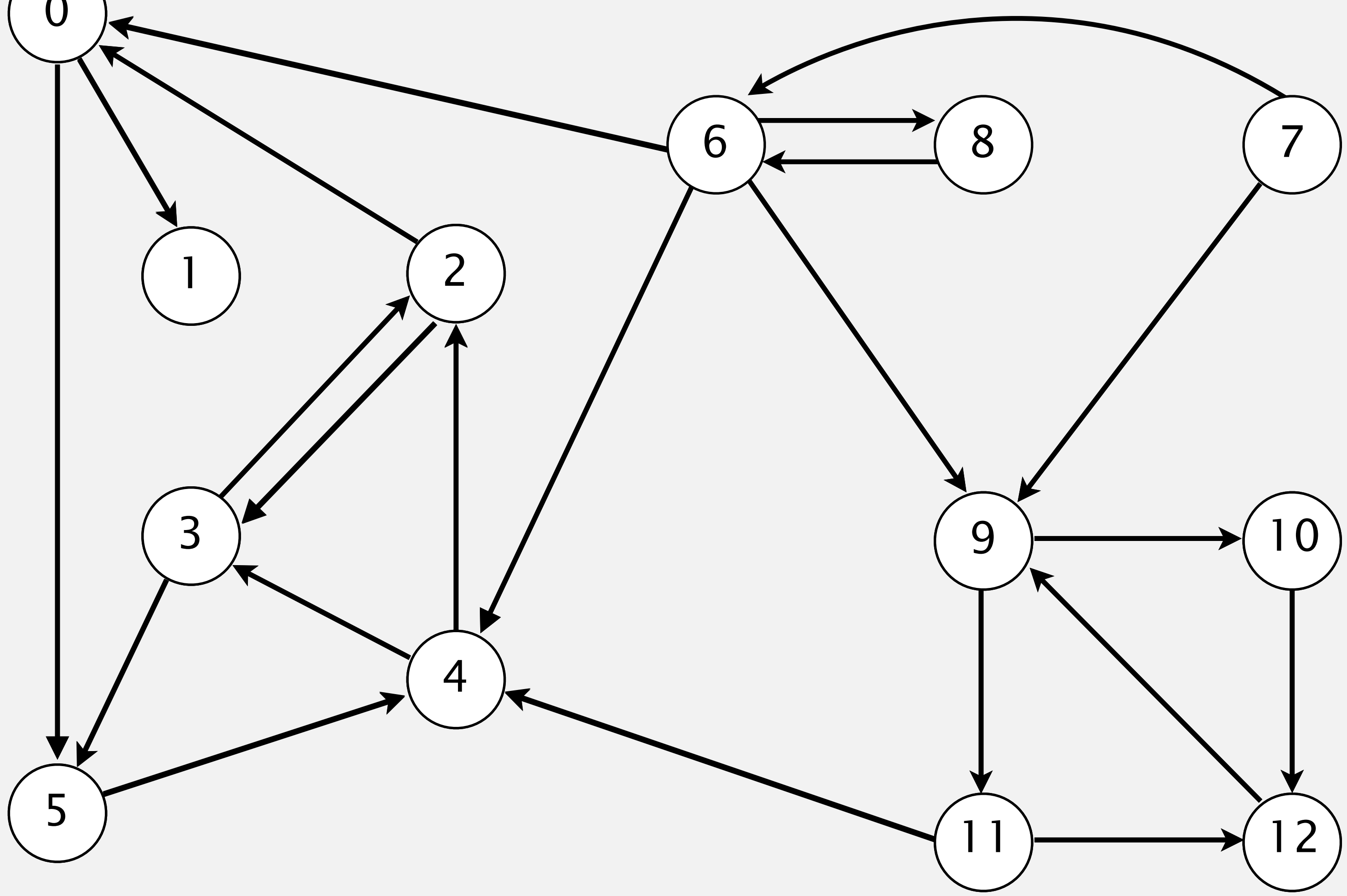
a directed graph

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

Directed depth-first search demo

To visit a vertex v :

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



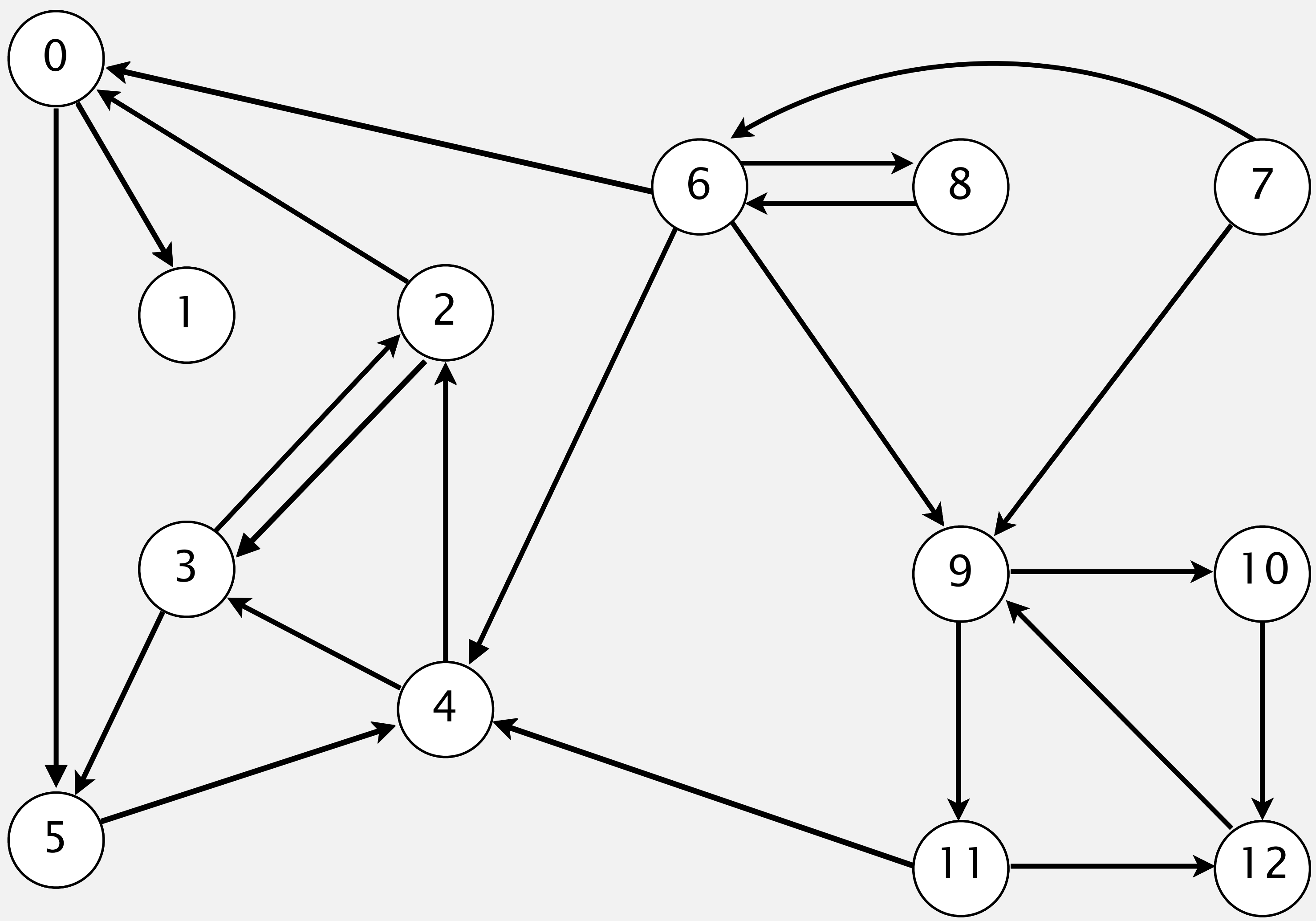
a directed graph

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
6→4
6→9
7→6

Directed depth-first search demo

To visit a vertex v :

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



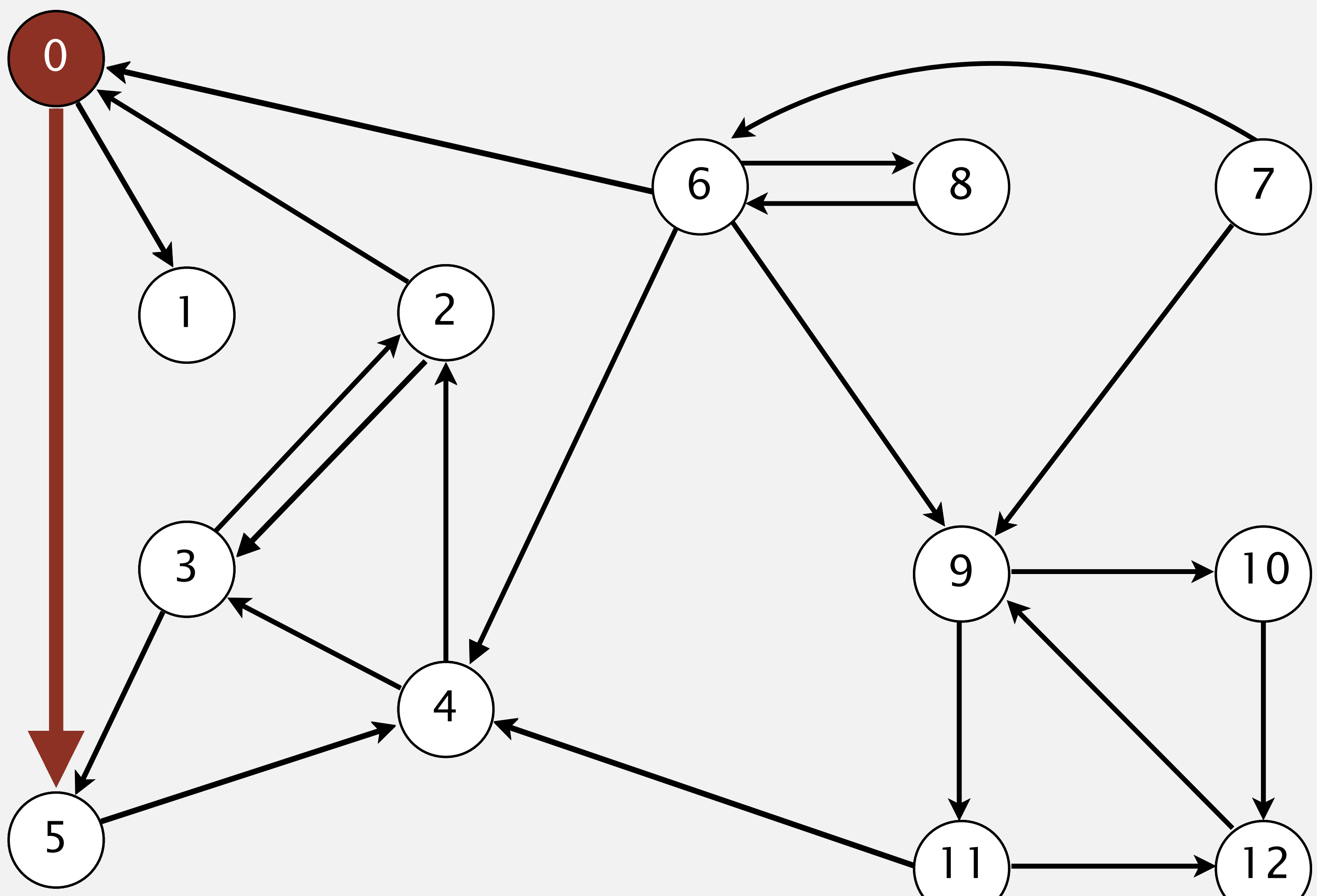
v	marked[]	edgeTo[]
0	F	–
1	F	–
2	F	–
3	F	–
4	F	–
5	F	–
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

a directed graph

Directed depth-first search demo

To visit a vertex v :

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



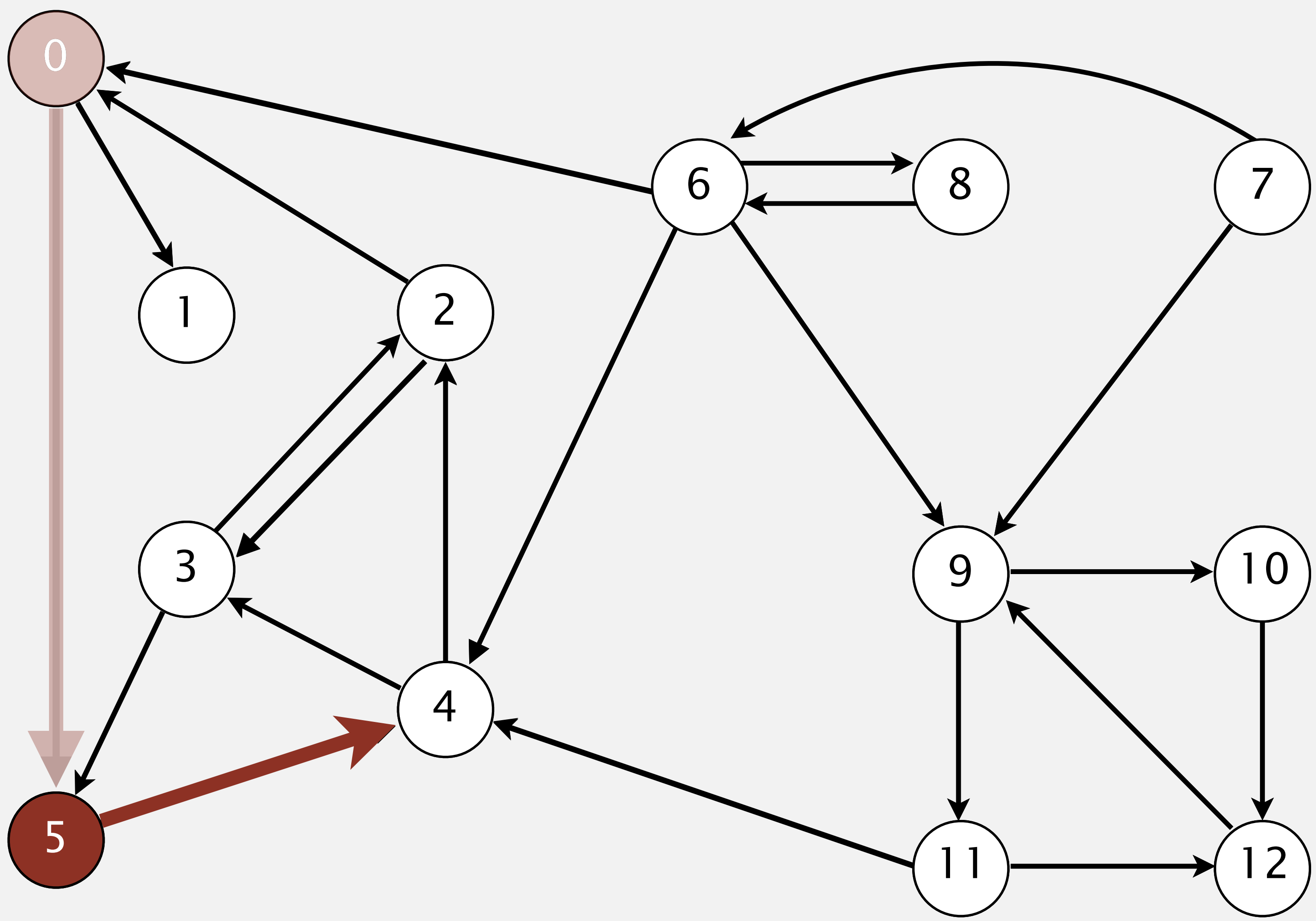
v	marked[]	edgeTo[]
0	T	–
1	F	–
2	F	–
3	F	–
4	F	–
5	F	–
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 0: check 5 and check 1

Directed depth-first search demo

To visit a vertex v :

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



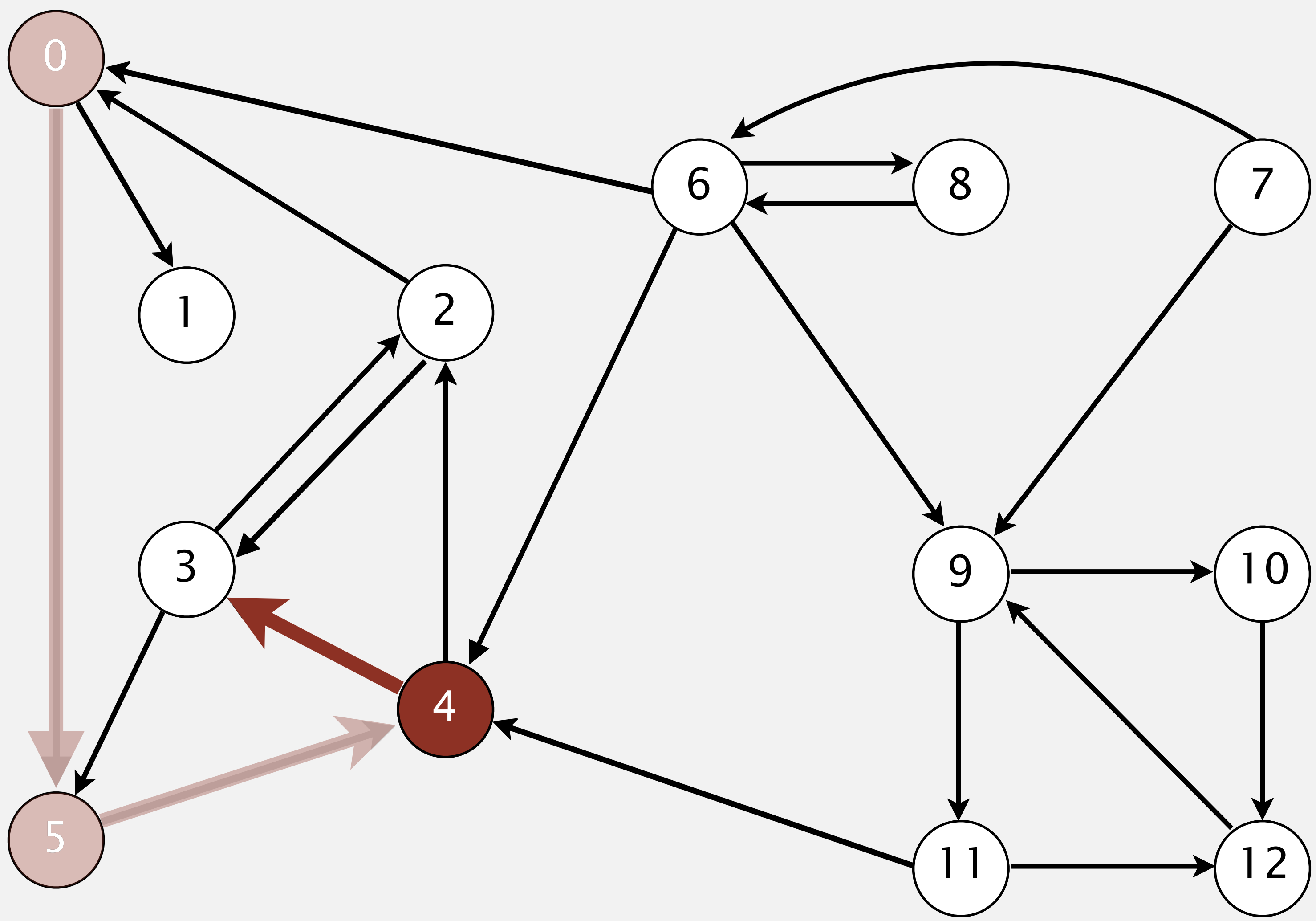
v	marked[]	edgeTo[]
0	T	–
1	F	–
2	F	–
3	F	–
4	F	–
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 5: check 4

Directed depth-first search demo

To visit a vertex v :

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



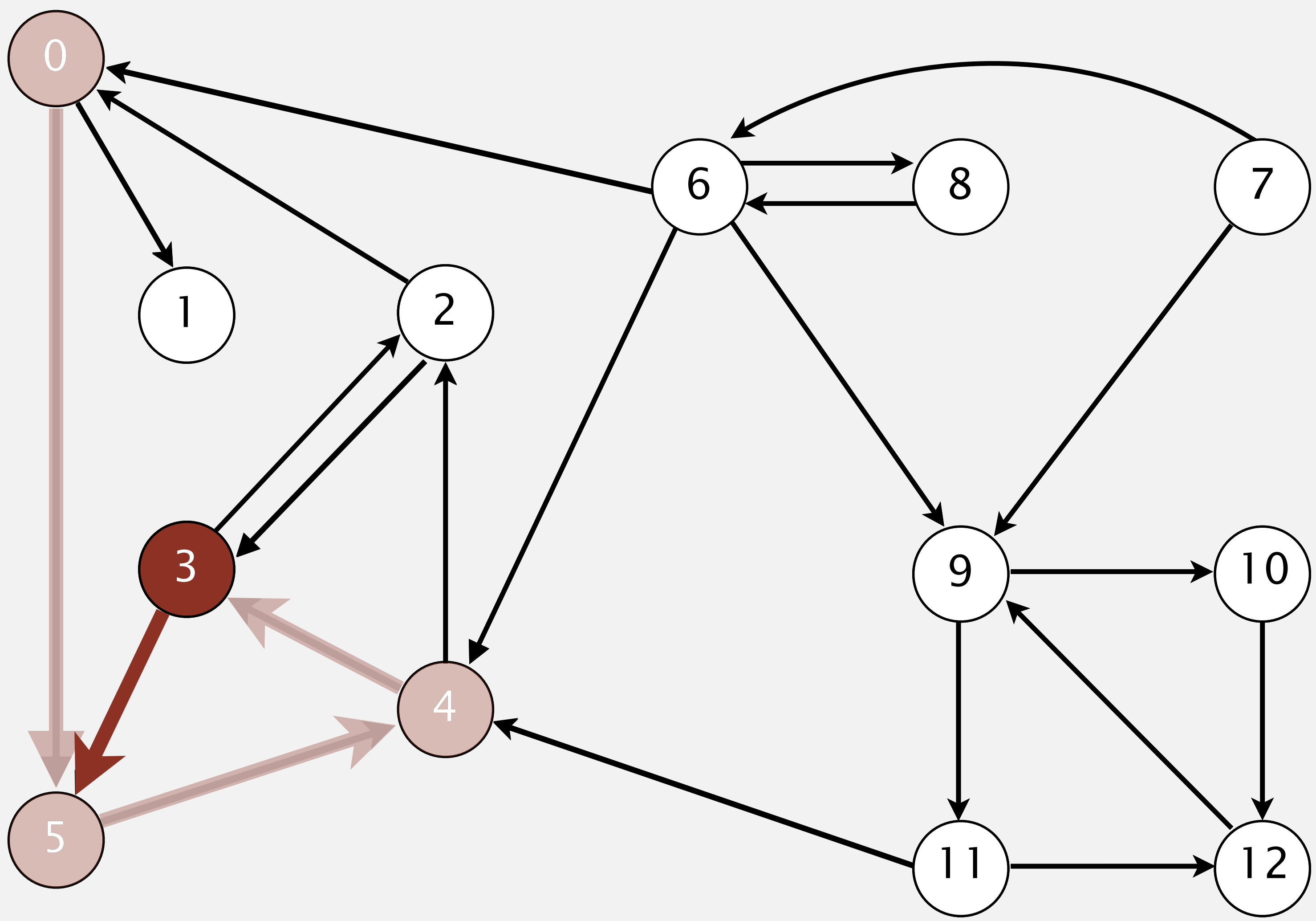
v	marked[]	edgeTo[]
0	T	–
1	F	–
2	F	–
3	F	–
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 4: check 3 and check 2

Directed depth-first search demo

To visit a vertex v :

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



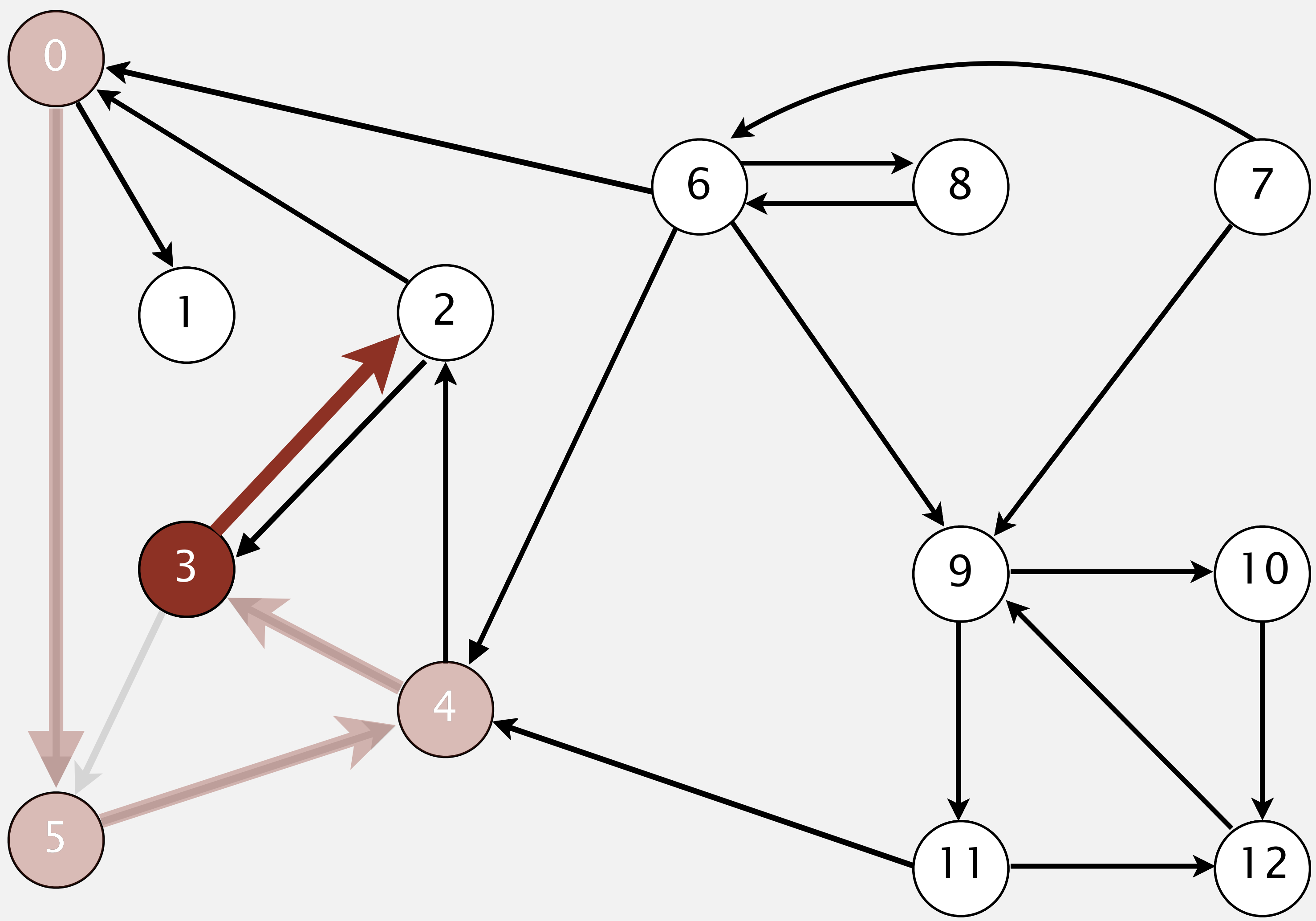
v	marked[]	edgeTo[]
0	T	–
1	F	–
2	F	–
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 3: check 5 and check 2

Directed depth-first search demo

To visit a vertex v :

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



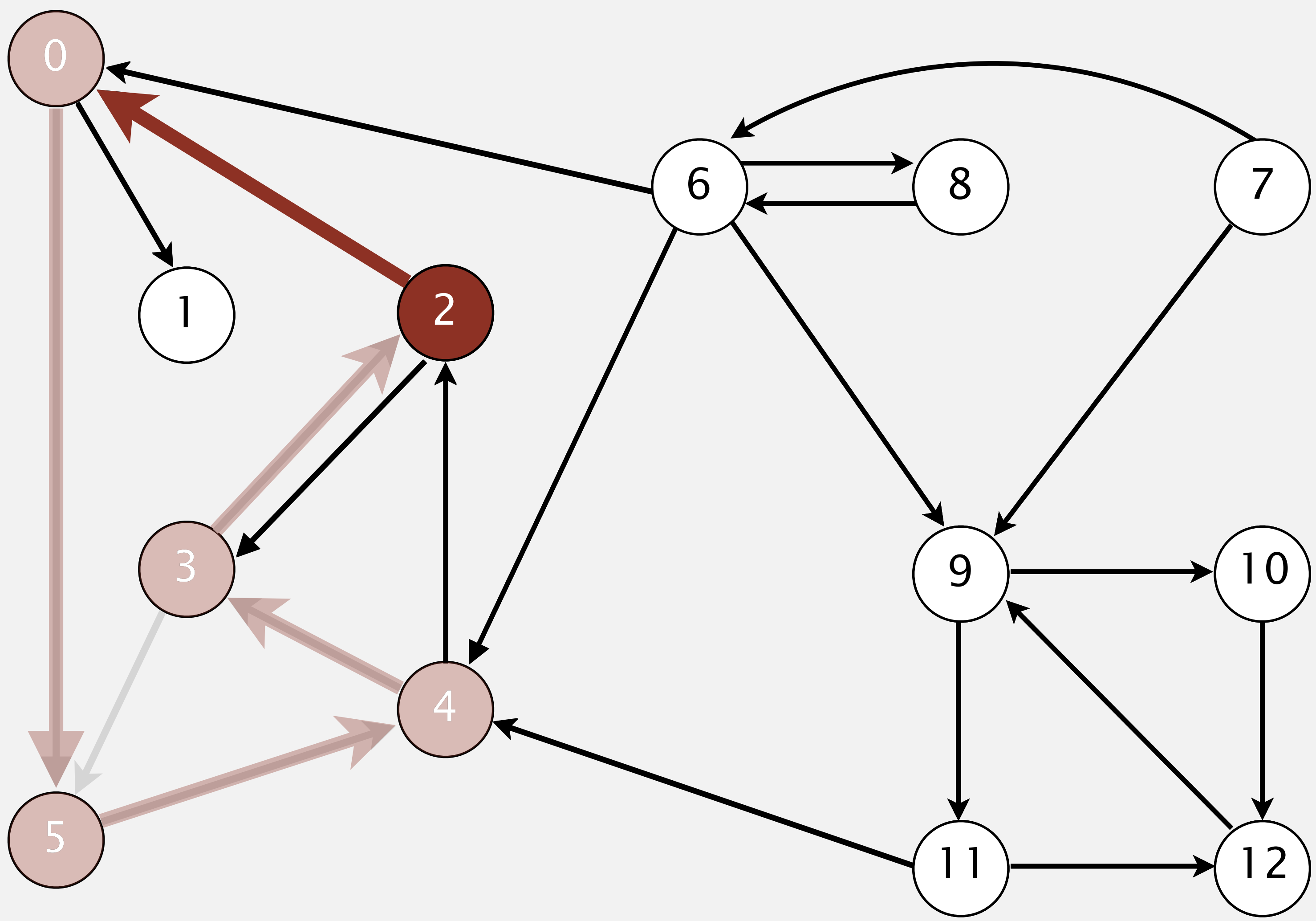
v	marked[]	edgeTo[]
0	T	–
1	F	–
2	F	–
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 3: check 5 and **check 2**

Directed depth-first search demo

To visit a vertex v :

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



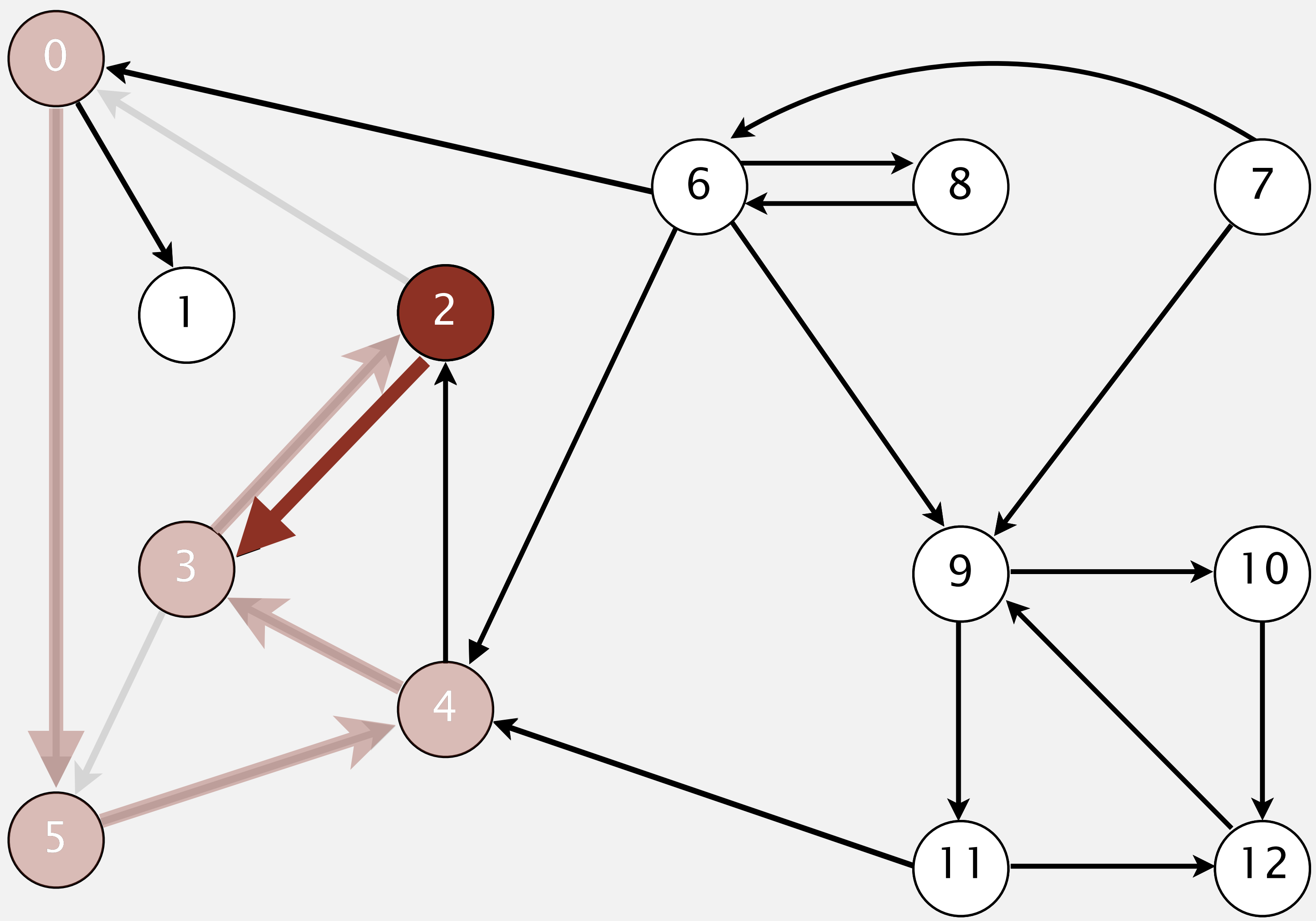
v	marked[]	edgeTo[]
0	T	–
1	F	–
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 2: check 0 and check 3

Directed depth-first search demo

To visit a vertex v :

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



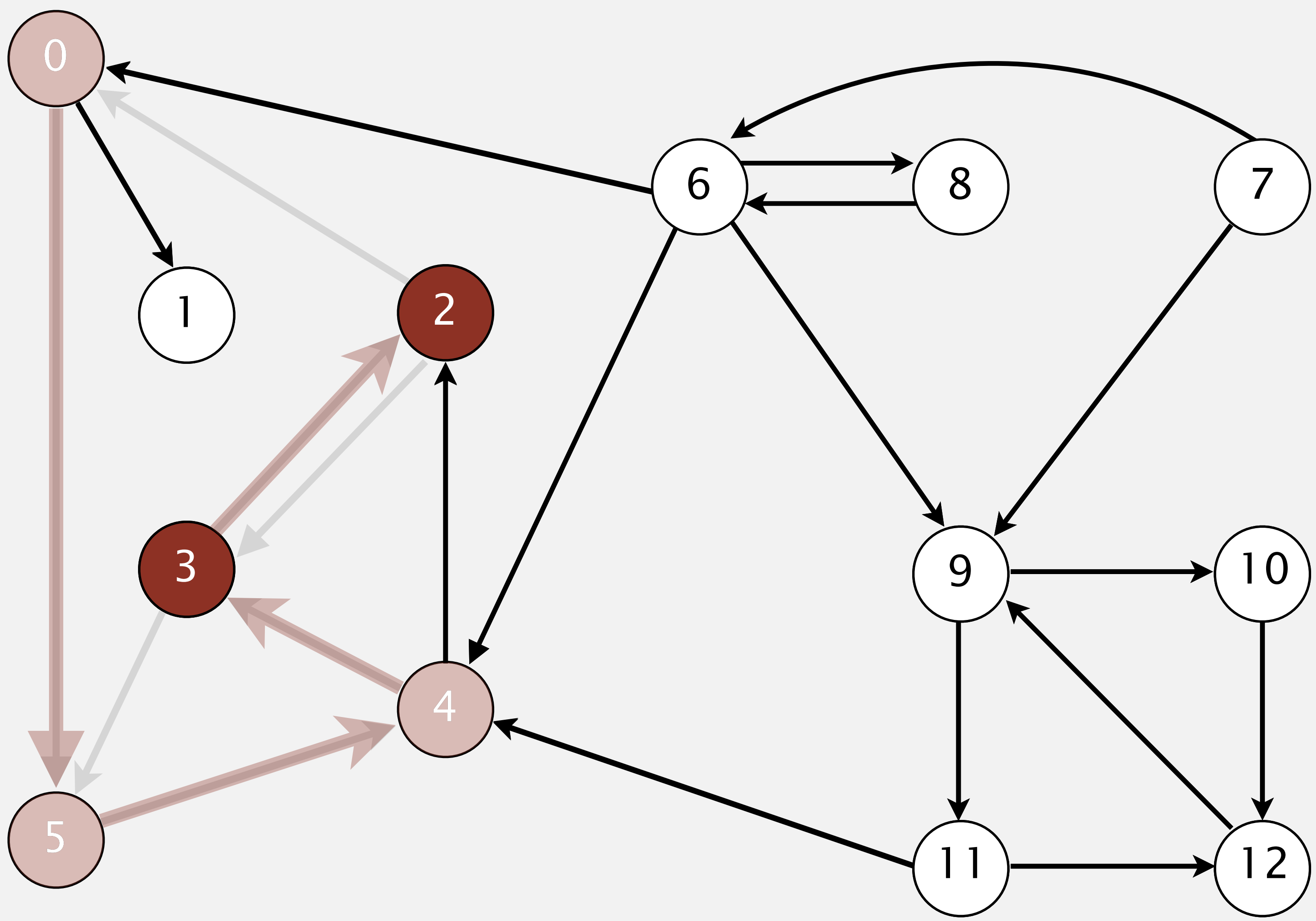
v	marked[]	edgeTo[]
0	T	–
1	F	–
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 2: check 0 and **check 3**

Directed depth-first search demo

To visit a vertex v :

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



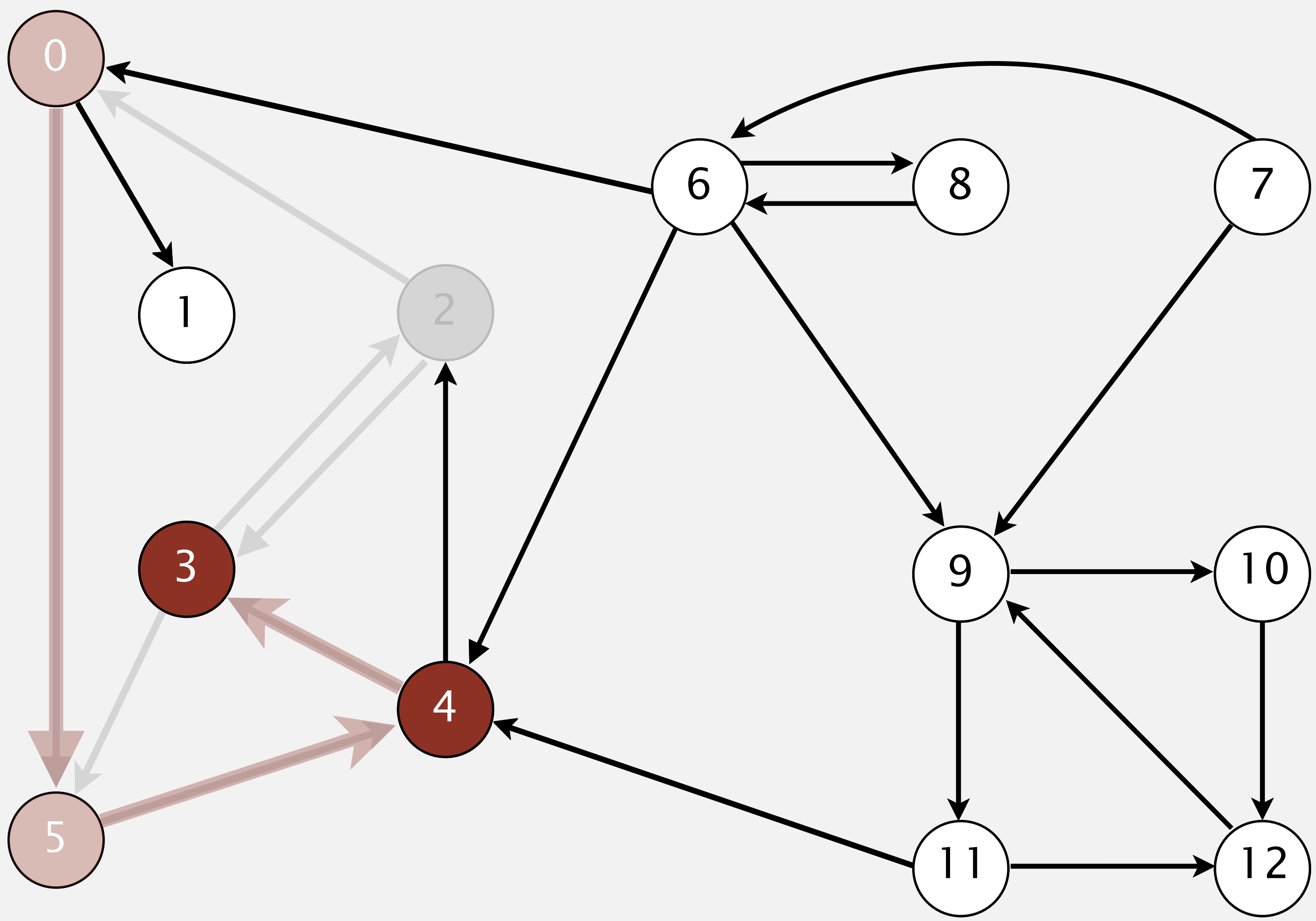
v	marked[]	edgeTo[]
0	T	–
1	F	–
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

done 2

Directed depth-first search demo

To visit a vertex v :

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



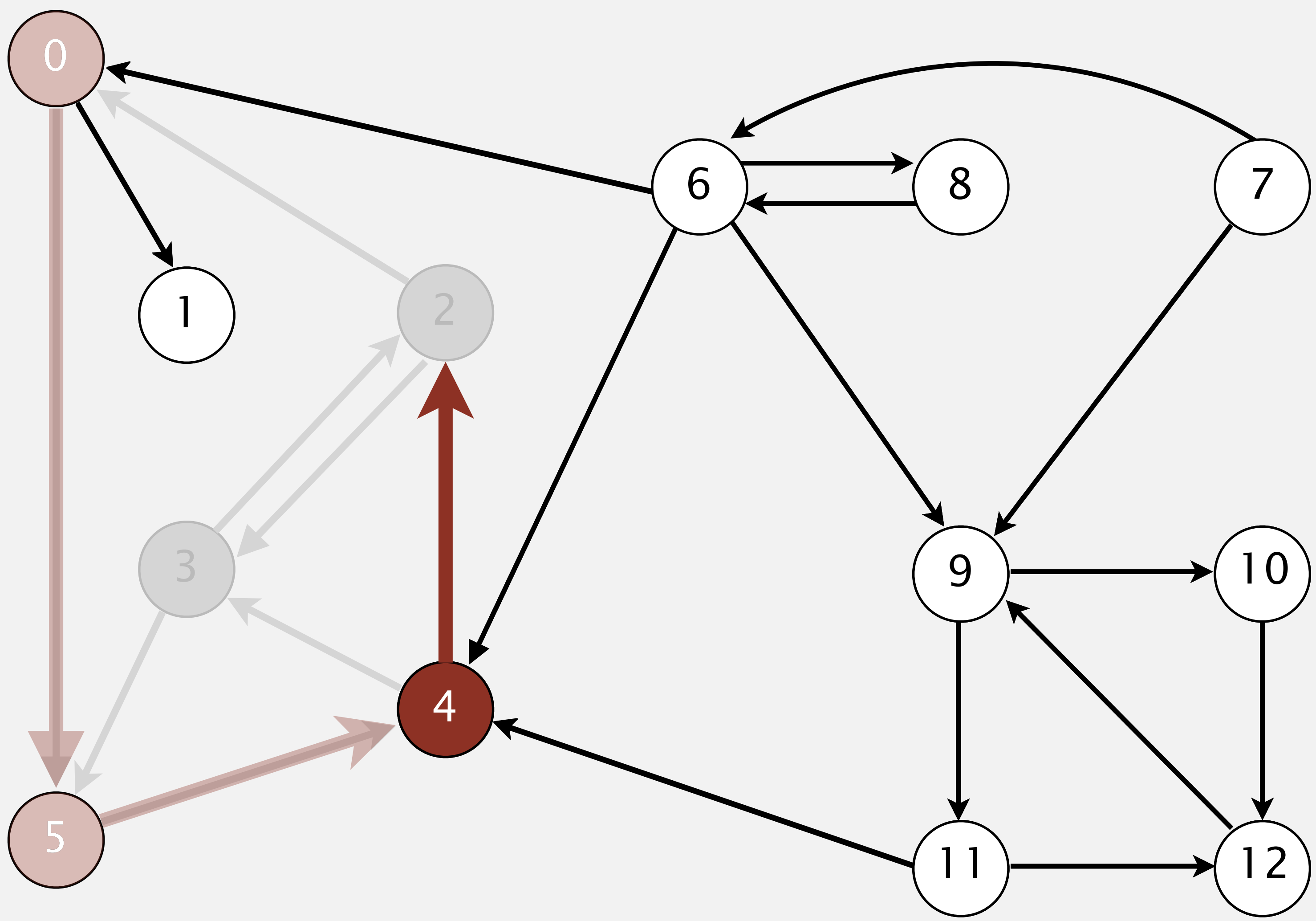
v	marked[]	edgeTo[]
0	T	–
1	F	–
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

done 3

Directed depth-first search demo

To visit a vertex v :

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



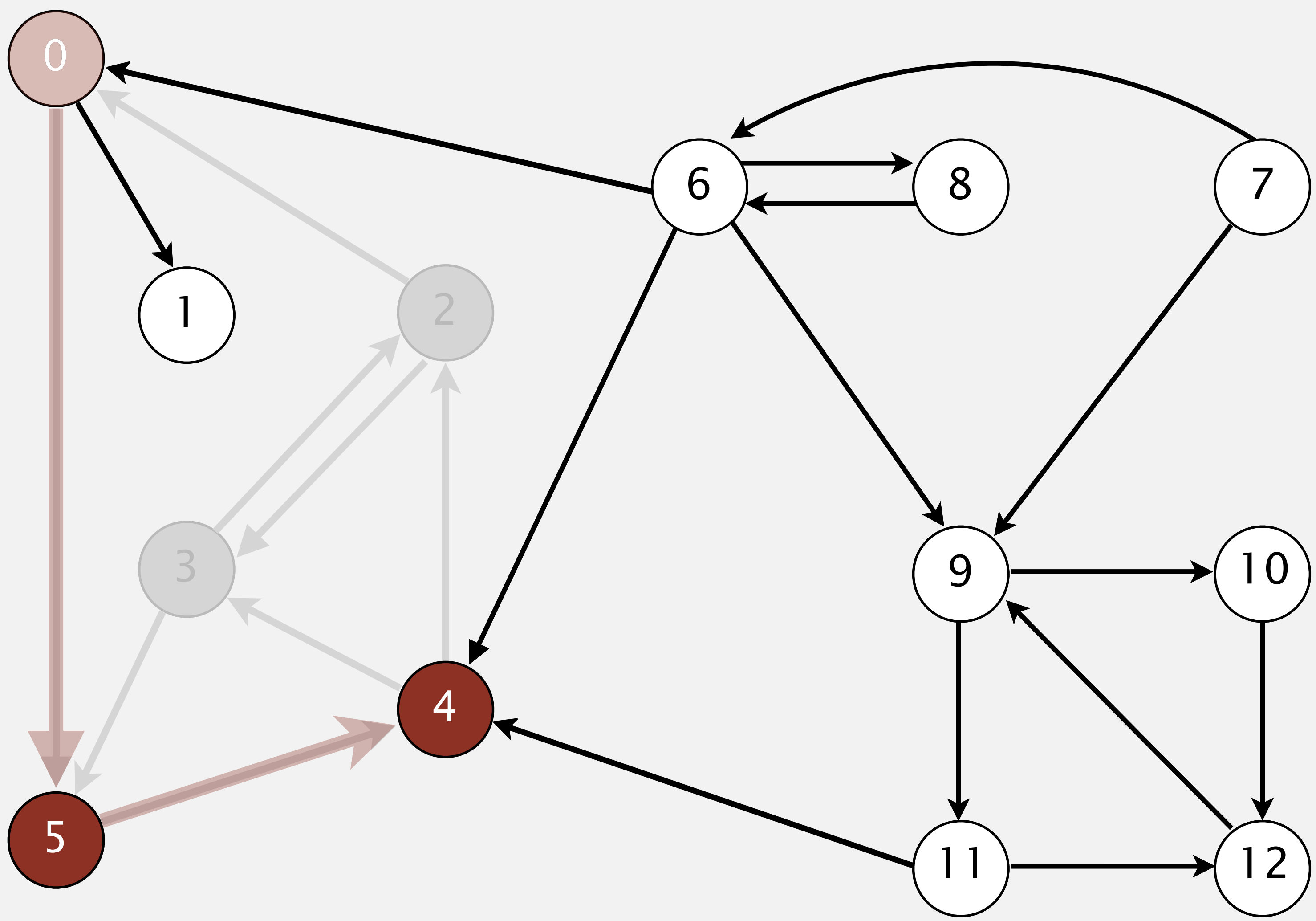
v	marked[]	edgeTo[]
0	T	–
1	F	–
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 4: check 3 and check 2

Directed depth-first search demo

To visit a vertex v :

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



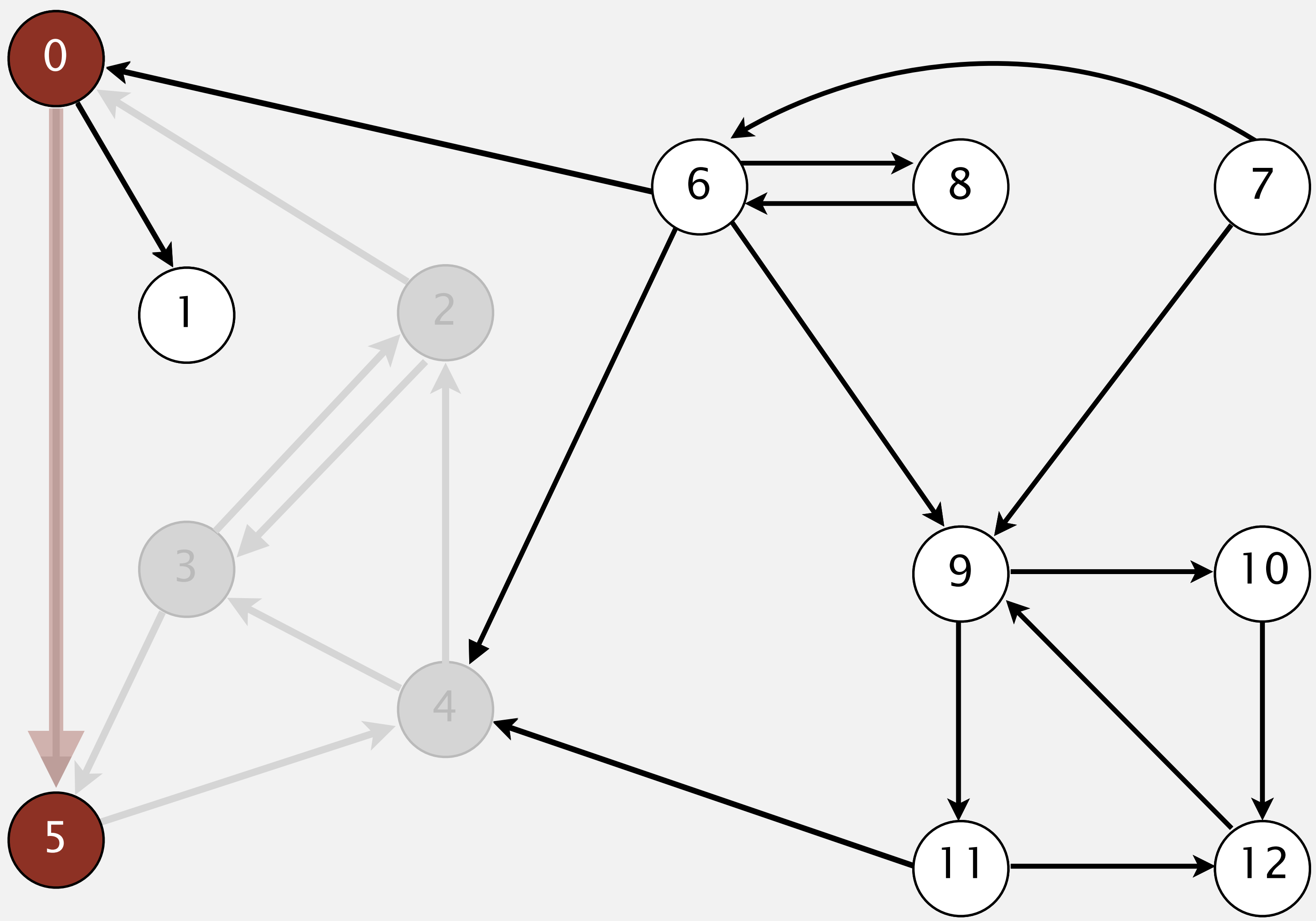
v	marked[]	edgeTo[]
0	T	–
1	F	–
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

done 4

Directed depth-first search demo

To visit a vertex v :

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



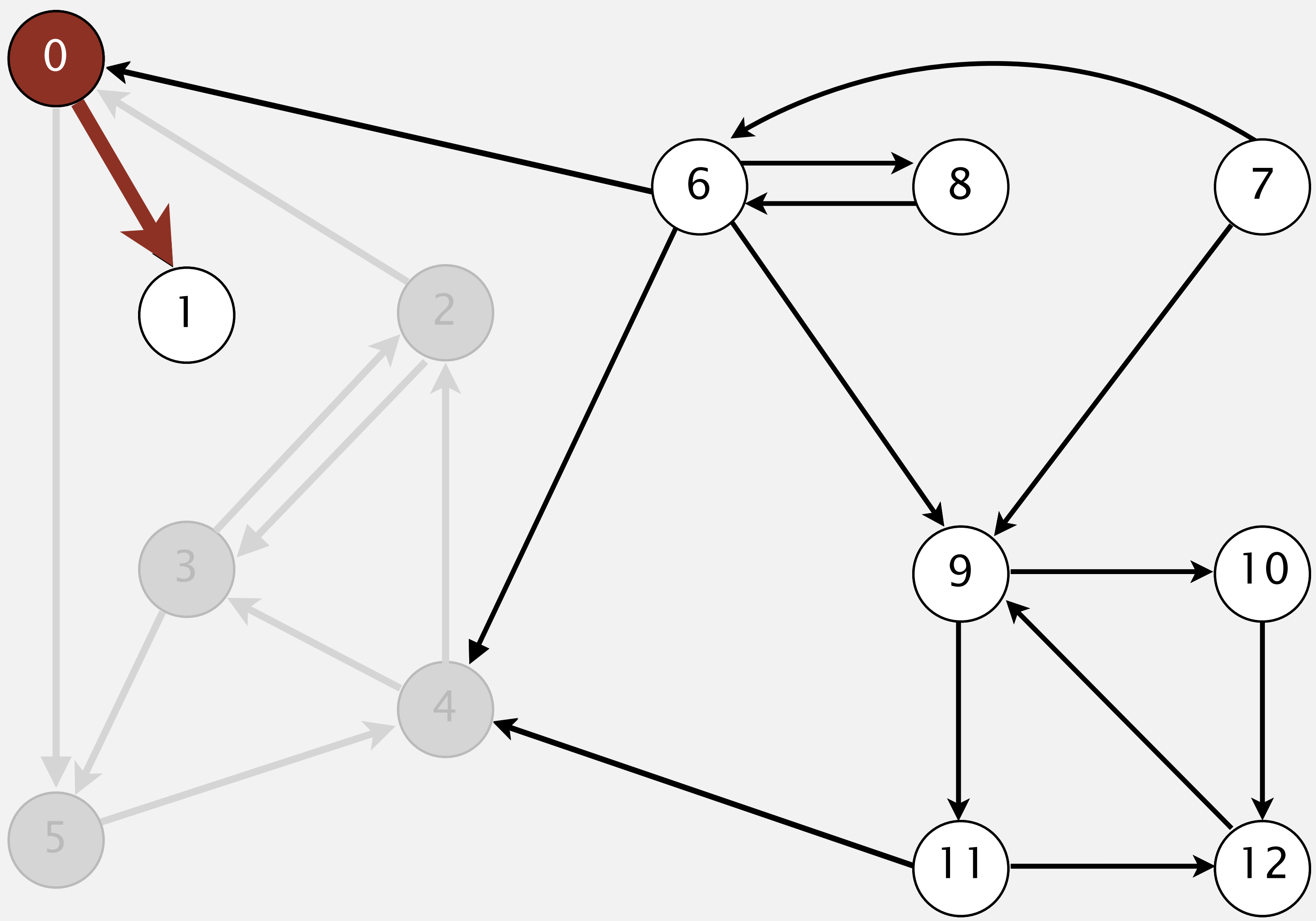
v	marked[]	edgeTo[]
0	T	–
1	F	–
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

done 5

Directed depth-first search demo

To visit a vertex v :

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



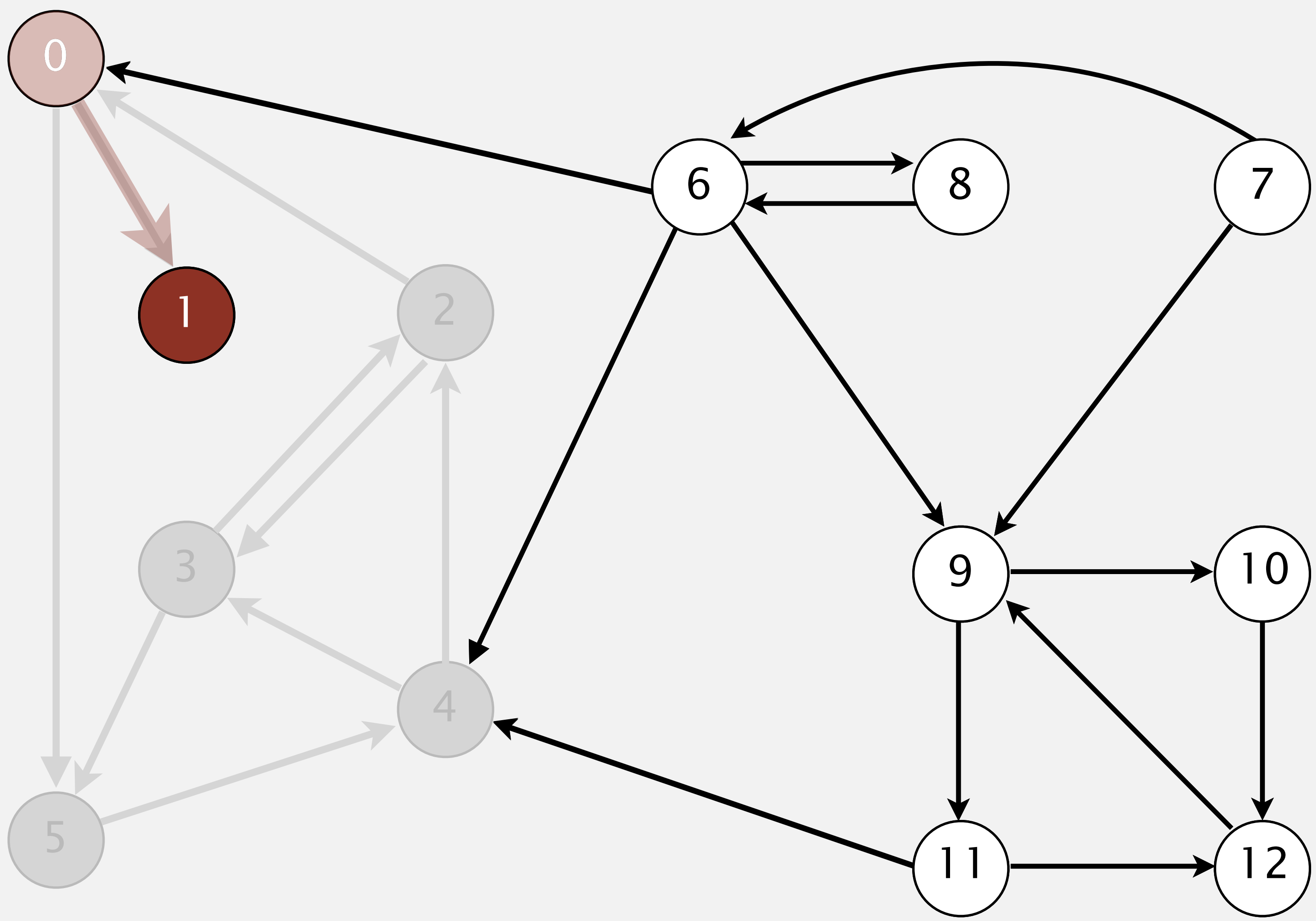
v	marked[]	edgeTo[]
0	T	–
1	F	–
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 0: check 5 and check 1

Directed depth-first search demo

To visit a vertex v :

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



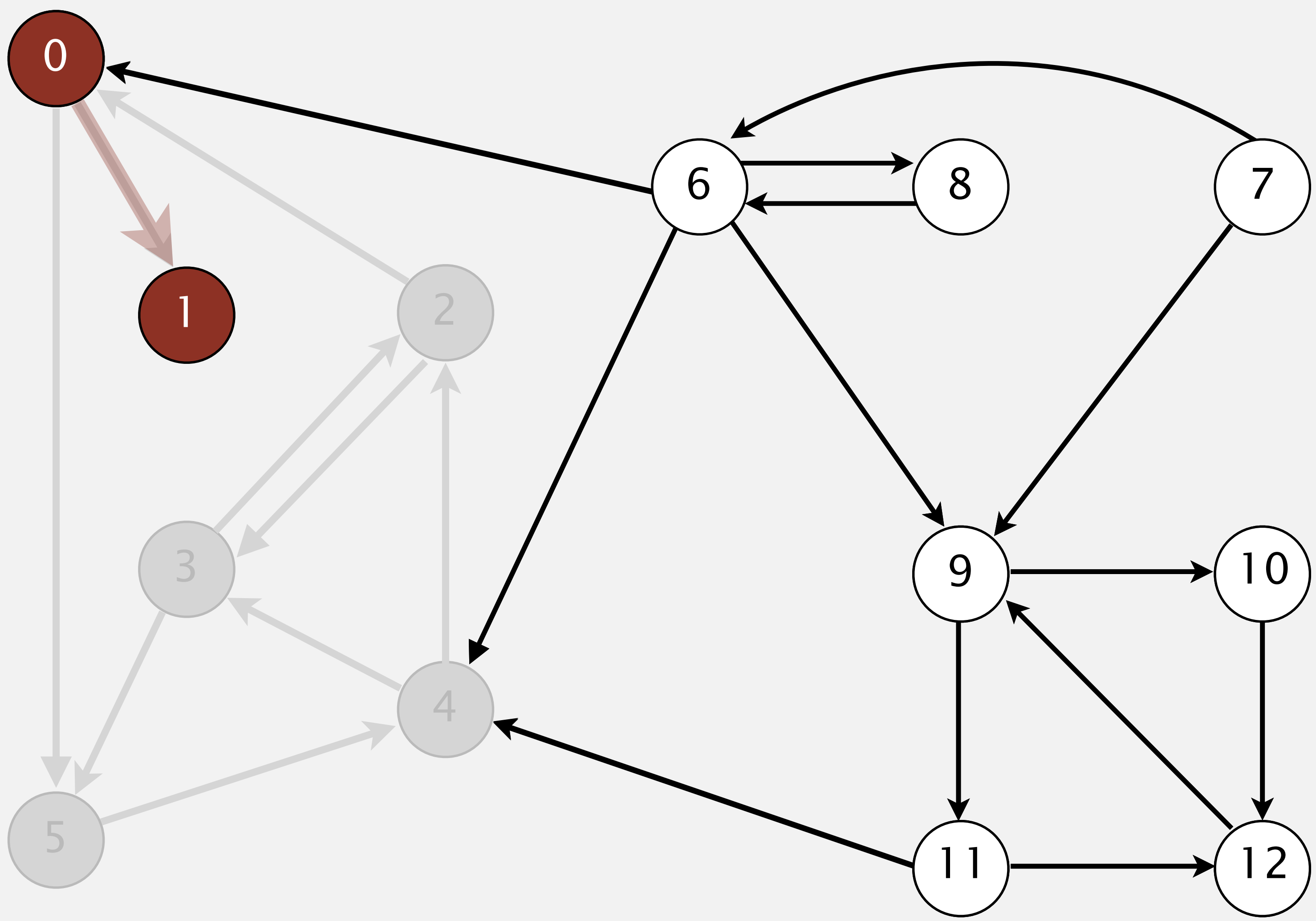
v	marked[]	edgeTo[]
0	T	–
1	T	0
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 1

Directed depth-first search demo

To visit a vertex v :

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



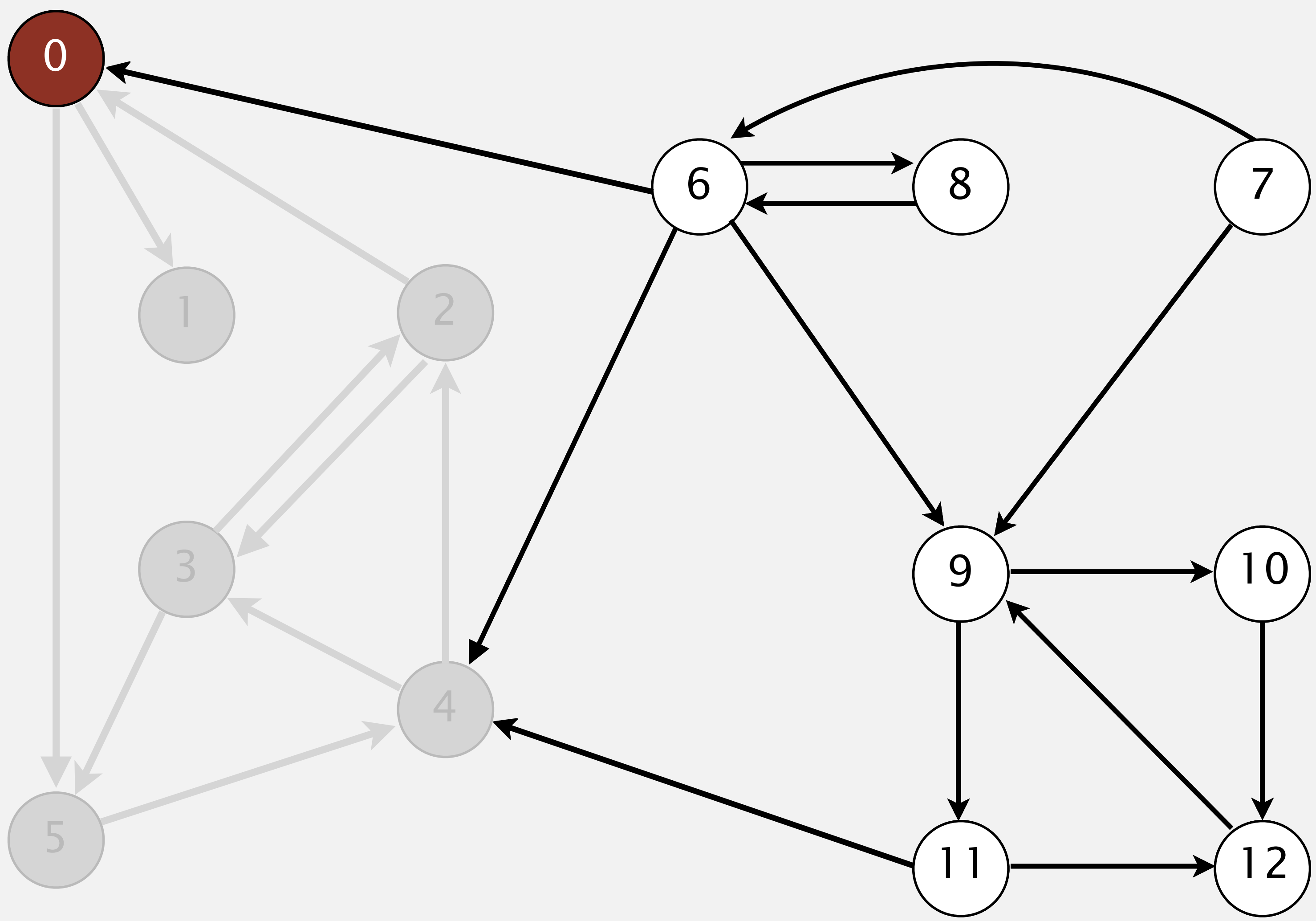
v	marked[]	edgeTo[]
0	T	–
1	T	0
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

done 1

Directed depth-first search demo

To visit a vertex v :

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



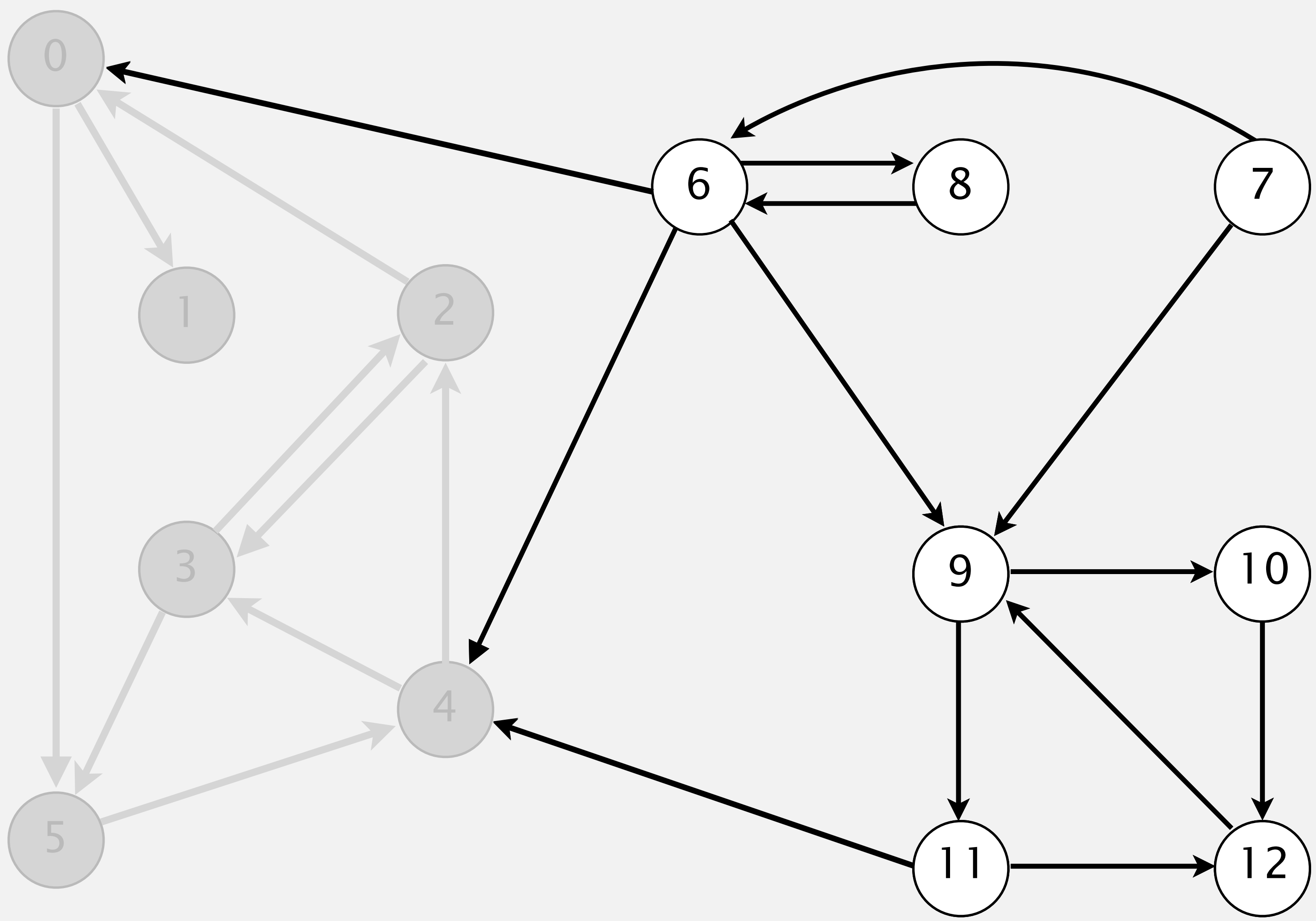
v	marked[]	edgeTo[]
0	T	–
1	T	0
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

done 0

Directed depth-first search demo

To visit a vertex v :

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



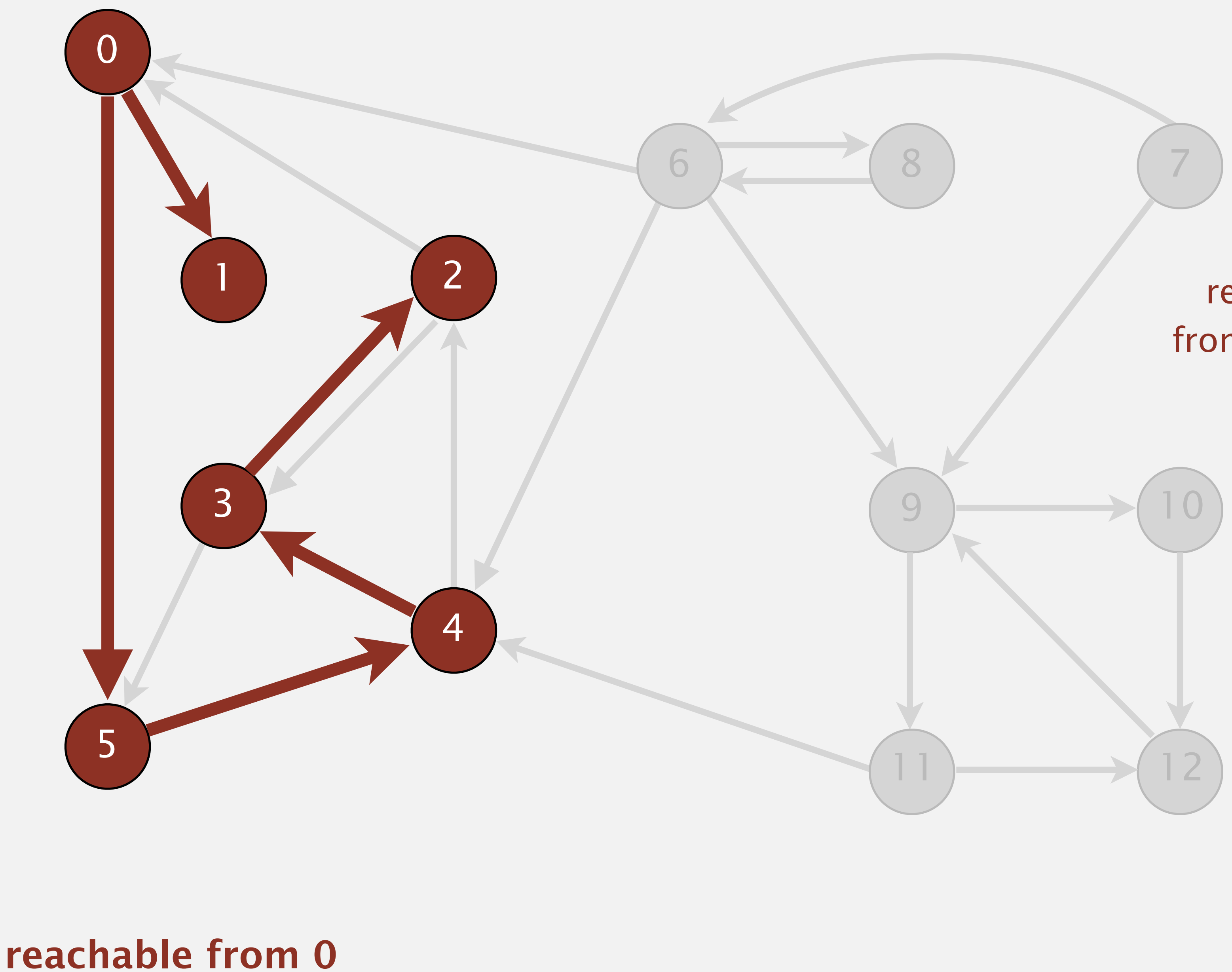
v	marked[]	edgeTo[]
0	T	–
1	T	0
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

done

Directed depth-first search demo

To visit a vertex v :

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



reachable from vertex 0

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 directed graphs)

Code for directed graphs identical to undirected one.

Good practice to separate your graph processing from graph

```
public class DirectedDFS
{
   private boolean[] marked;

   public DirectedDFS(Digraph G, int s)
   {
      marked = new boolean[G.V()];
      dfs(G, s);
   }

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

   public boolean visited(int v)
   {  return marked[v];   }
}
```

true if path from s

constructor marks vertices reachable from s

recursive DFS does the work

client can ask whether any vertex is reachable from s

BREATH FIRST SEARCH

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:

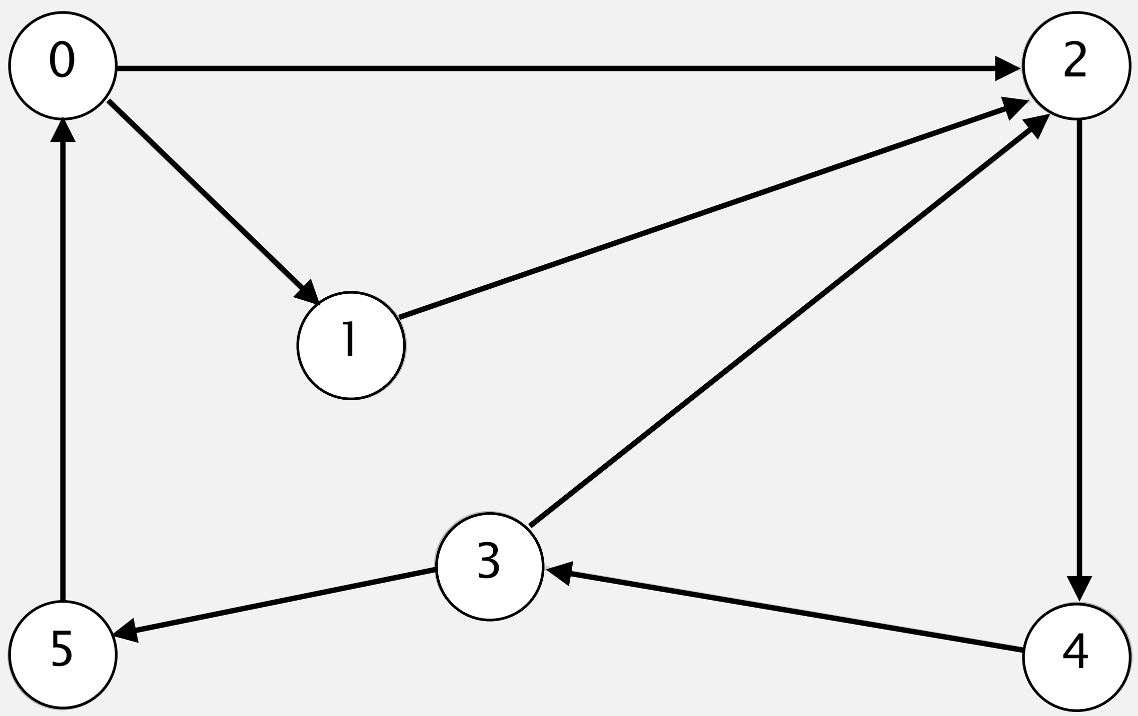
- **remove the least recently added vertex v**
- **for each unmarked vertex pointing from v:
  add to queue and mark as visited.**

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

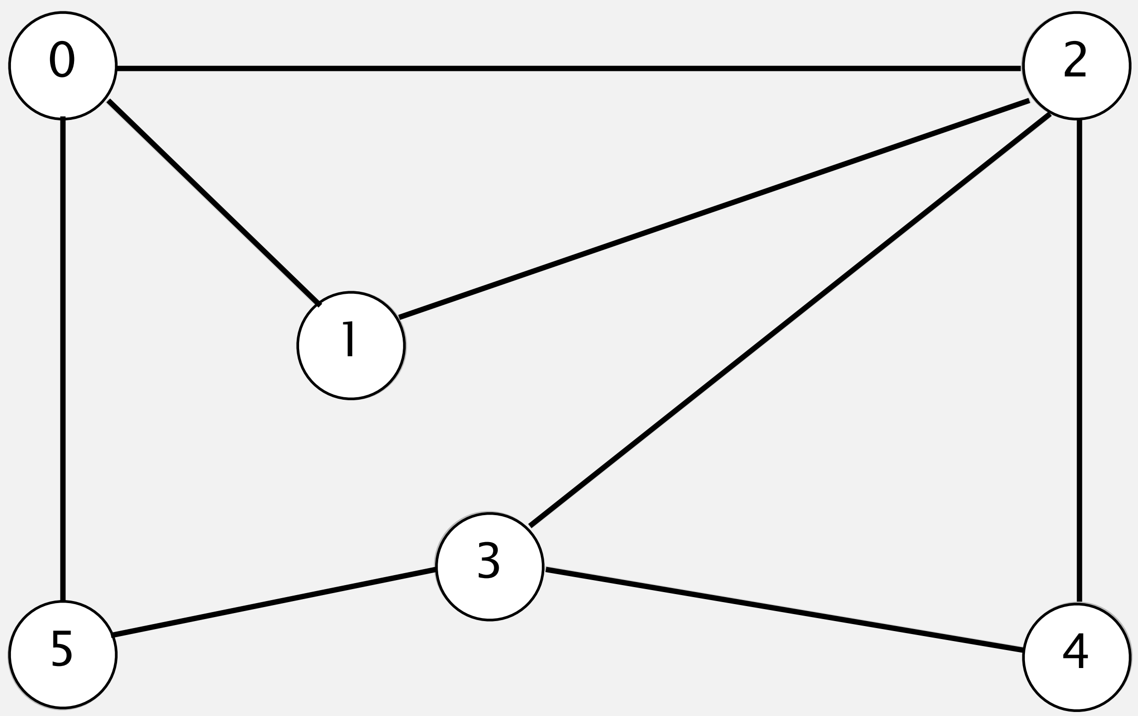
5	0
2	4
3	2
1	2
0	1
4	3
3	5
0	2

graph G

Breadth-first search demo

Repeat until queue is empty:

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



graph G

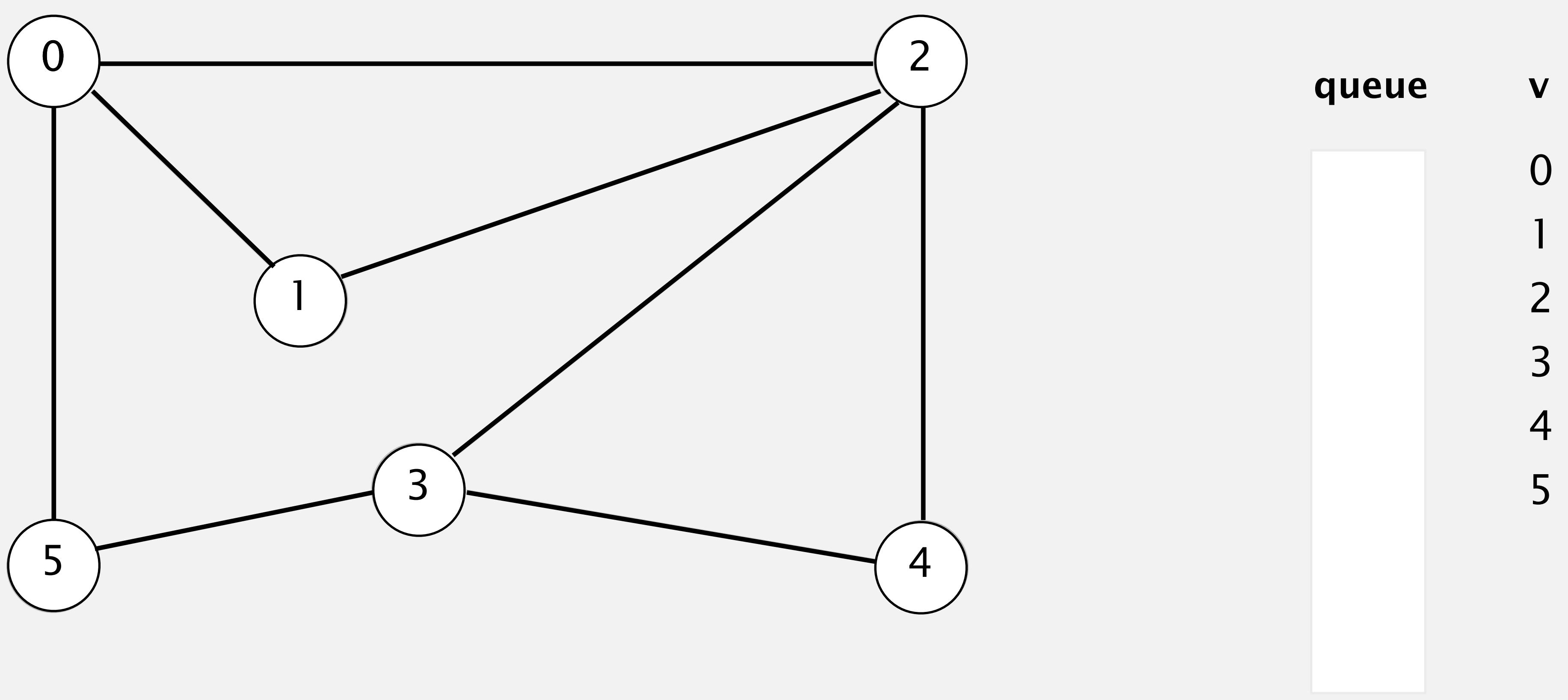
tinyCG.txt

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

Breadth-first search demo

Repeat until queue is empty:

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

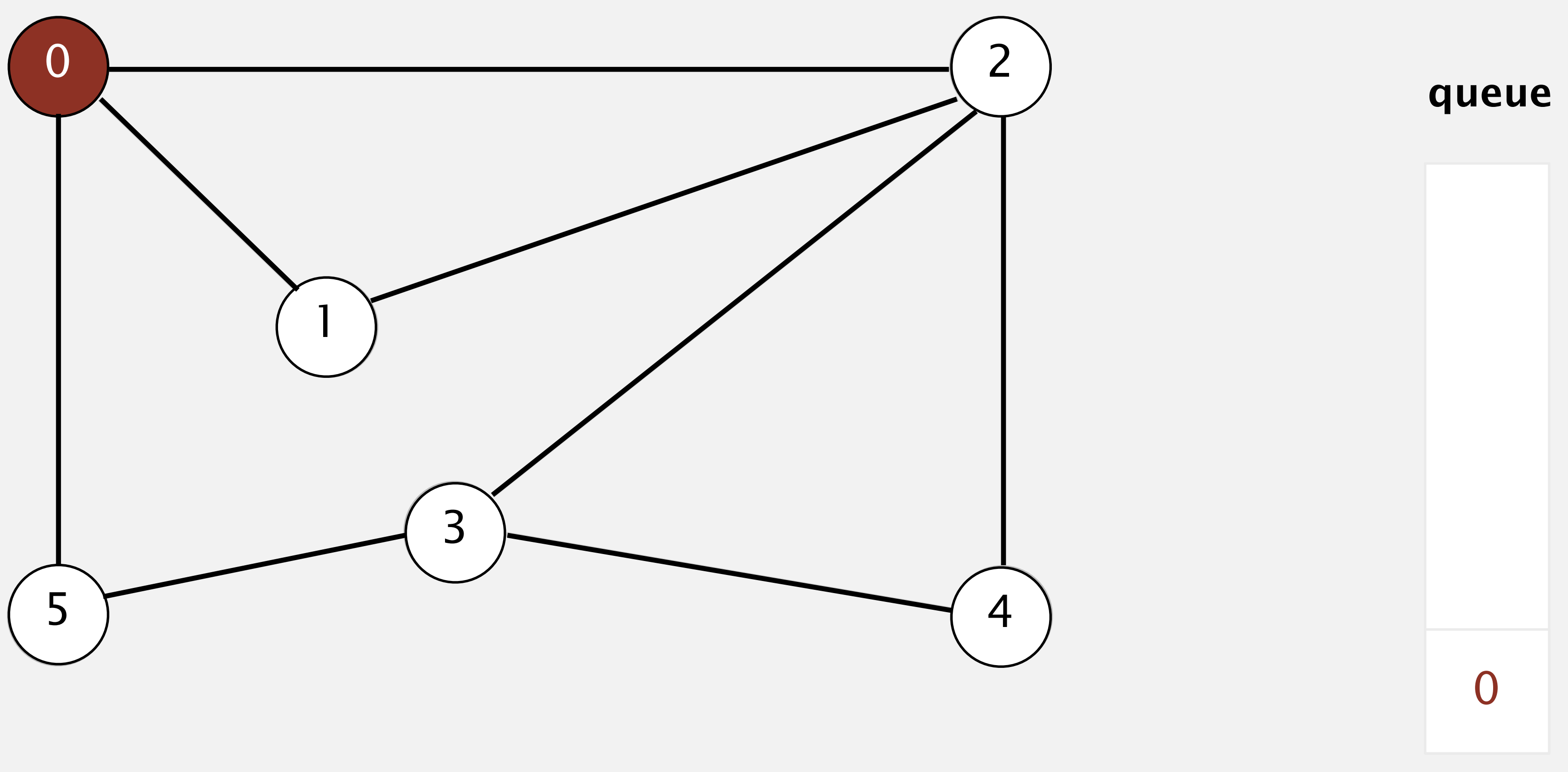


add 0 to queue

Breadth-first search demo

Repeat until queue is empty:

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

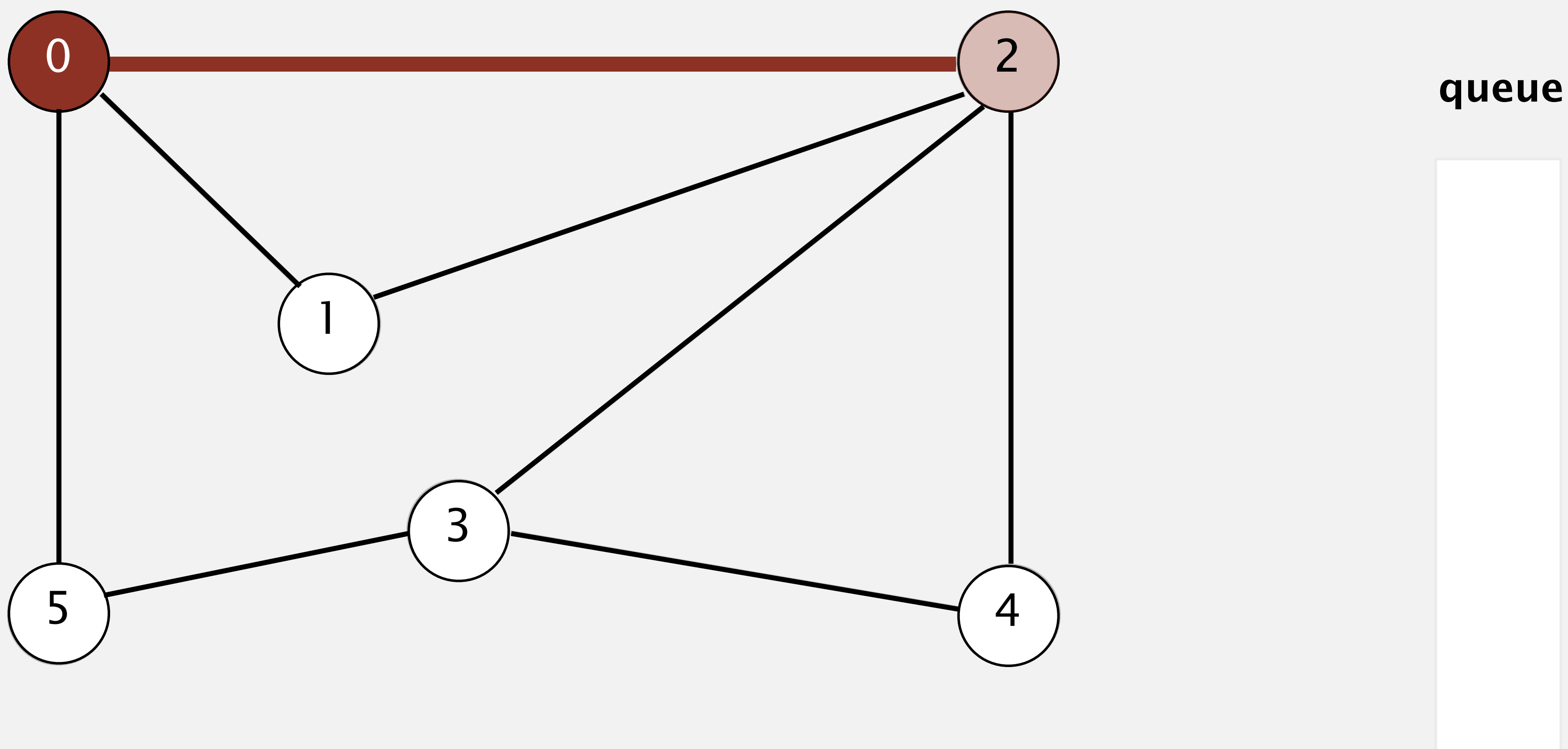


dequeue 0

Breadth-first search demo

Repeat until queue is empty:

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

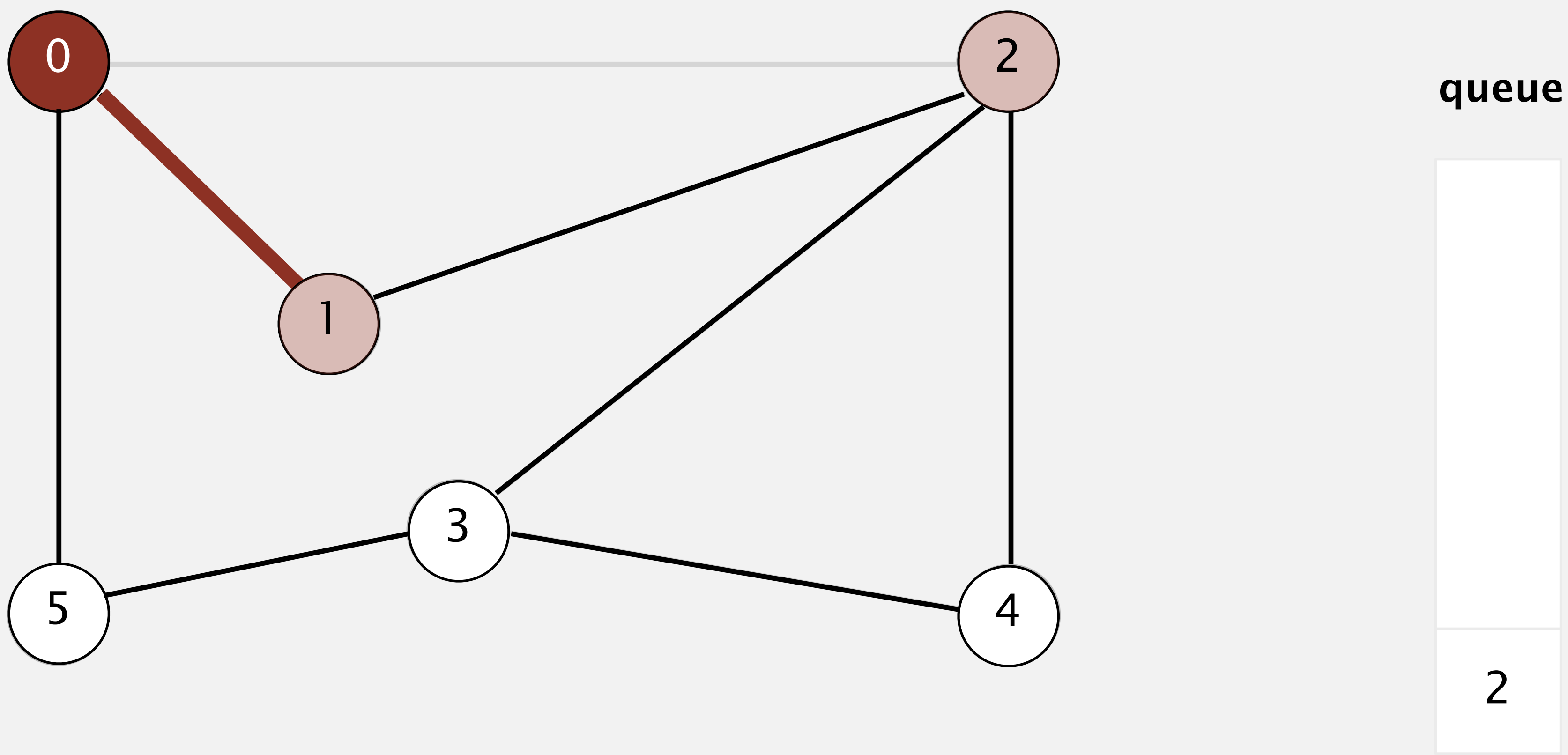


dequeue 0

Breadth-first search demo

Repeat until queue is empty:

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

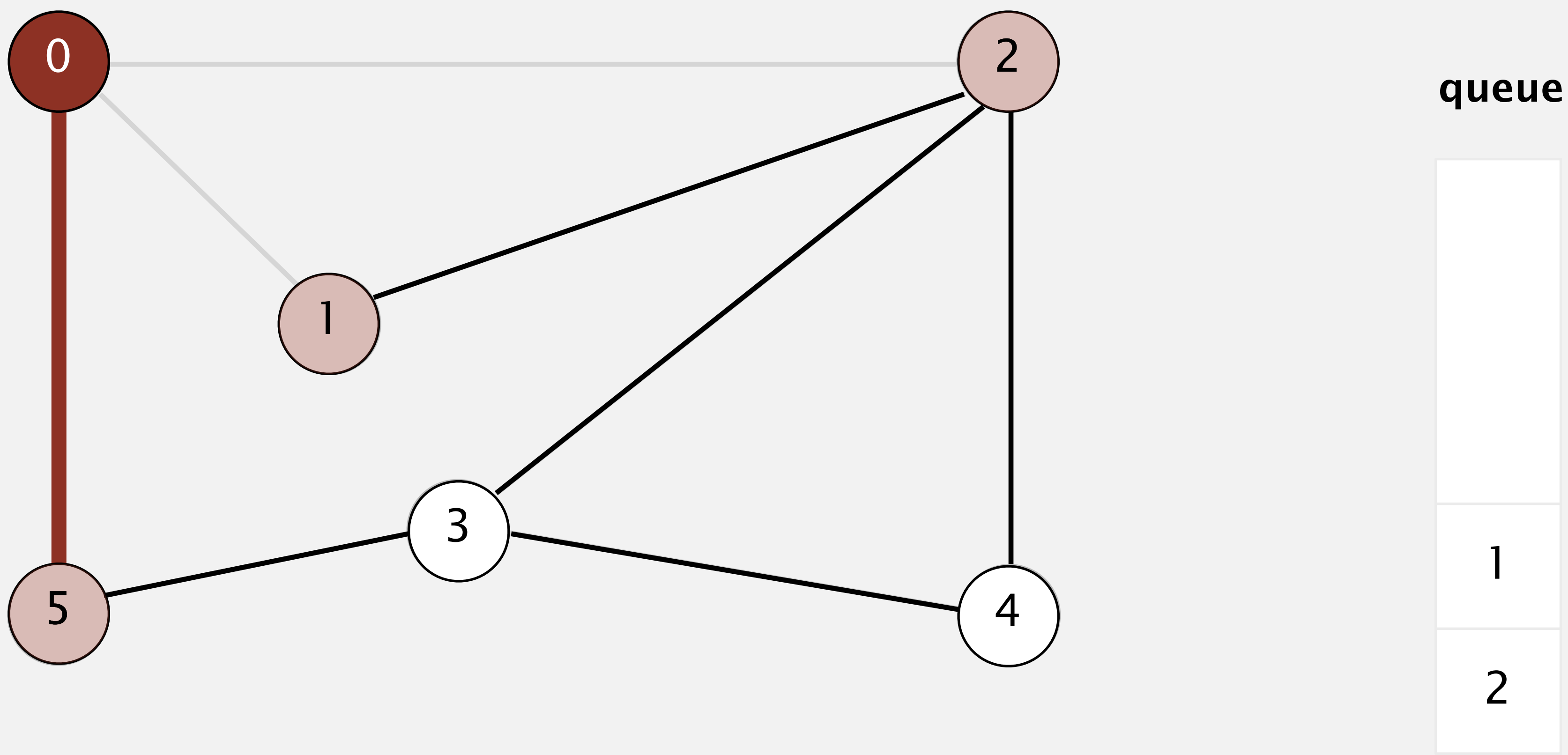


dequeue 0

Breadth-first search demo

Repeat until queue is empty:

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

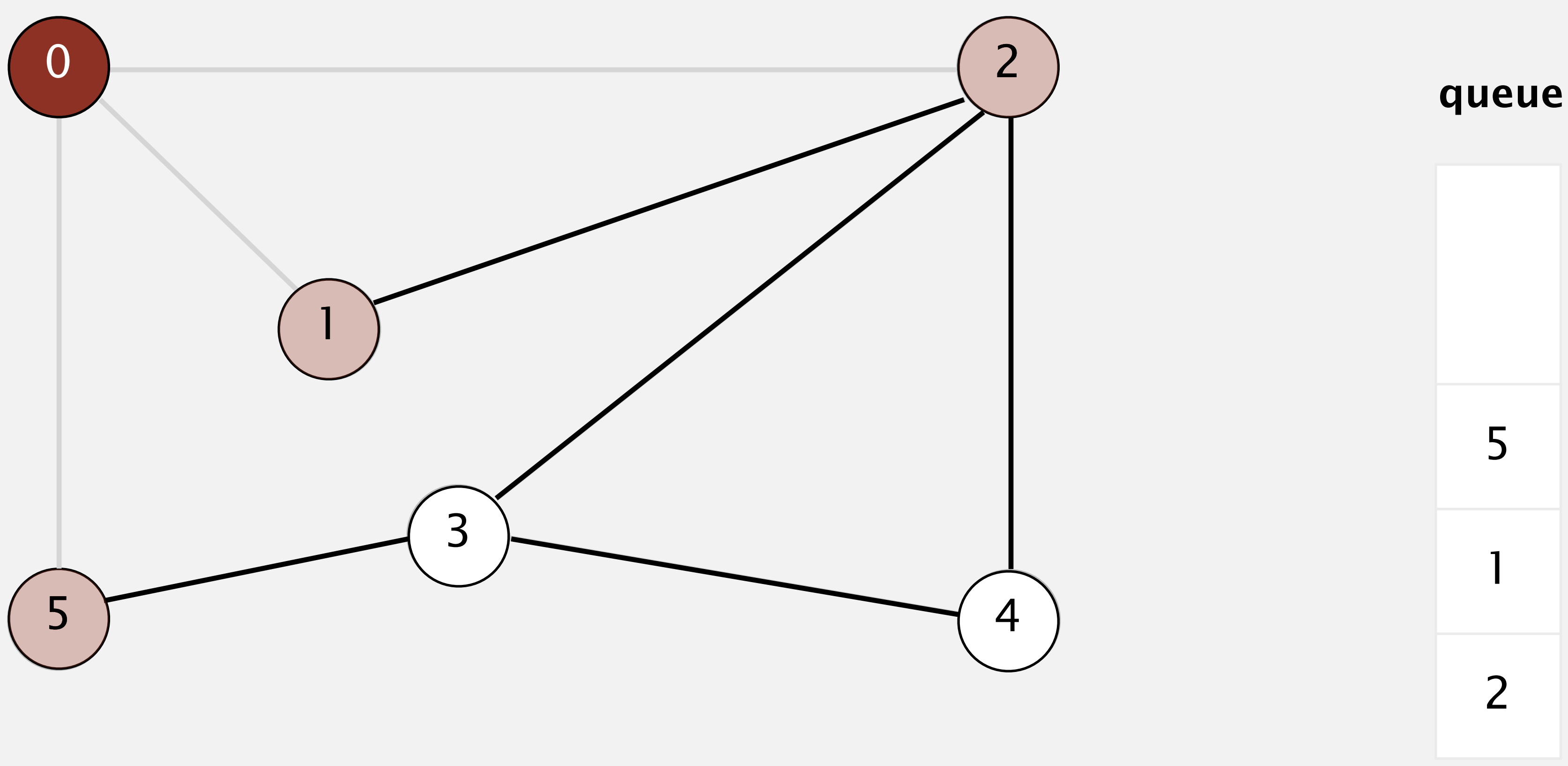


dequeue 0

Breadth-first search demo

Repeat until queue is empty:

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

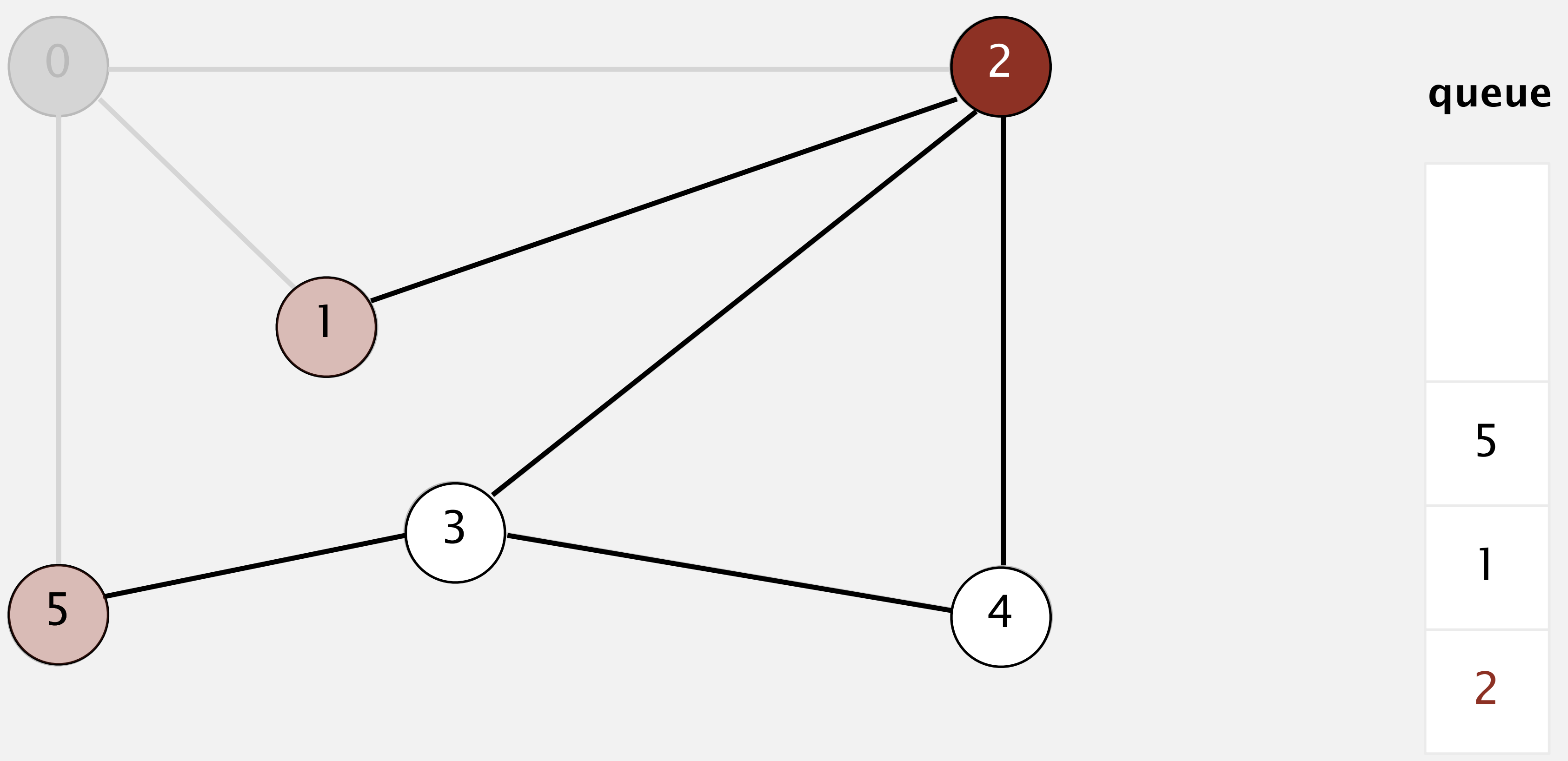


0 done

Breadth-first search demo

Repeat until queue is empty:

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

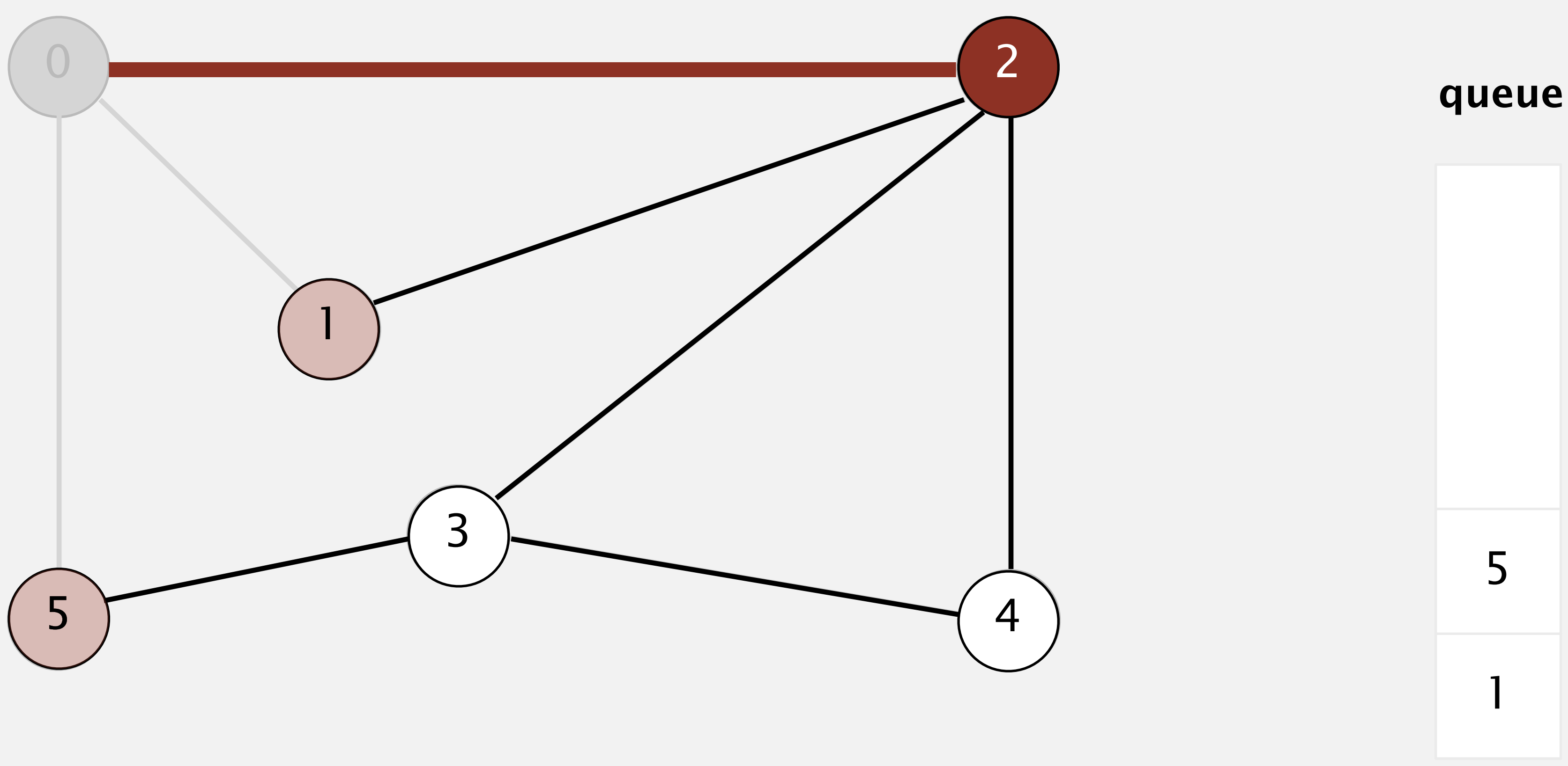


dequeue 2

Breadth-first search demo

Repeat until queue is empty:

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

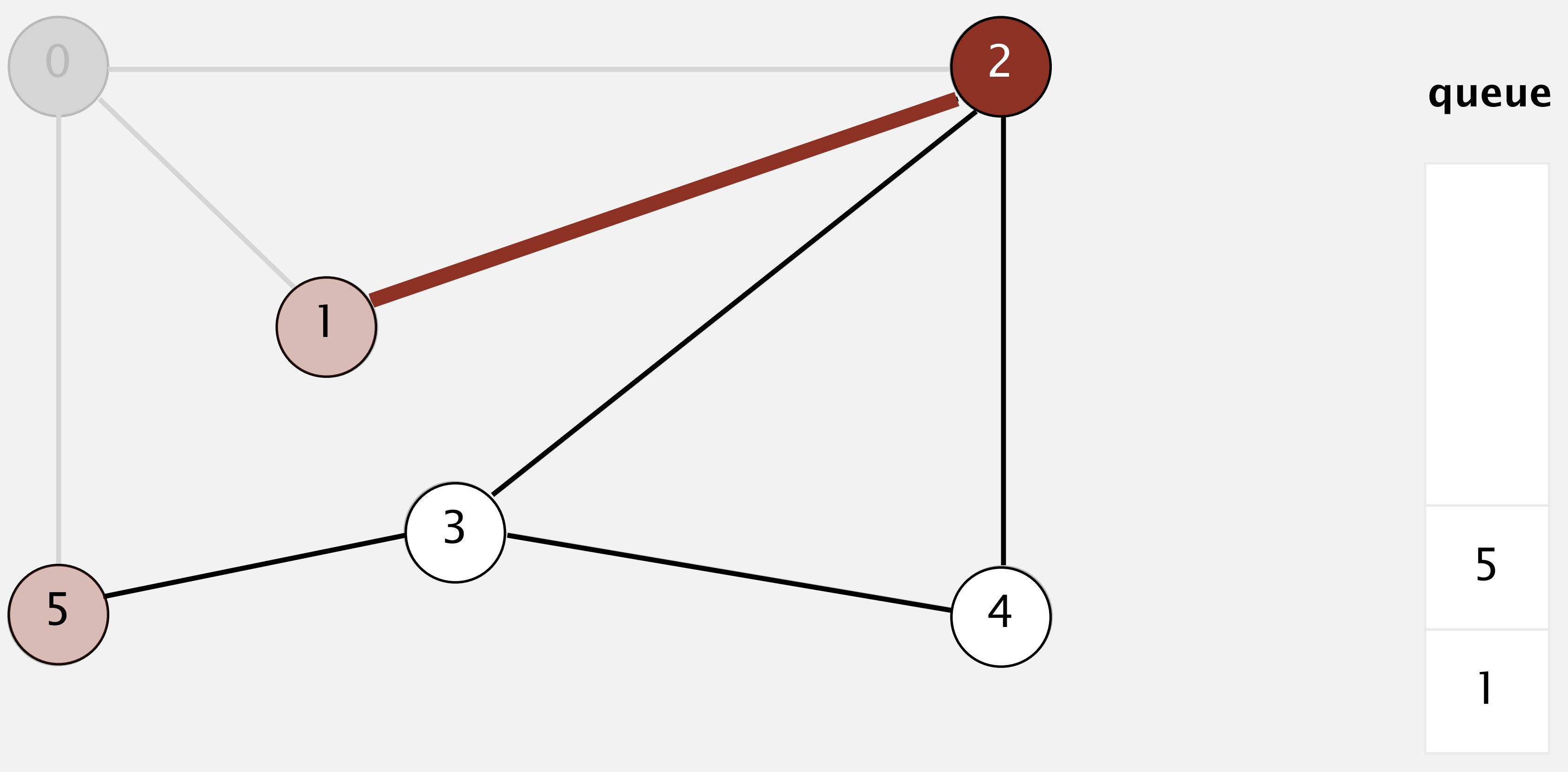


dequeue 2

Breadth-first search demo

Repeat until queue is empty:

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

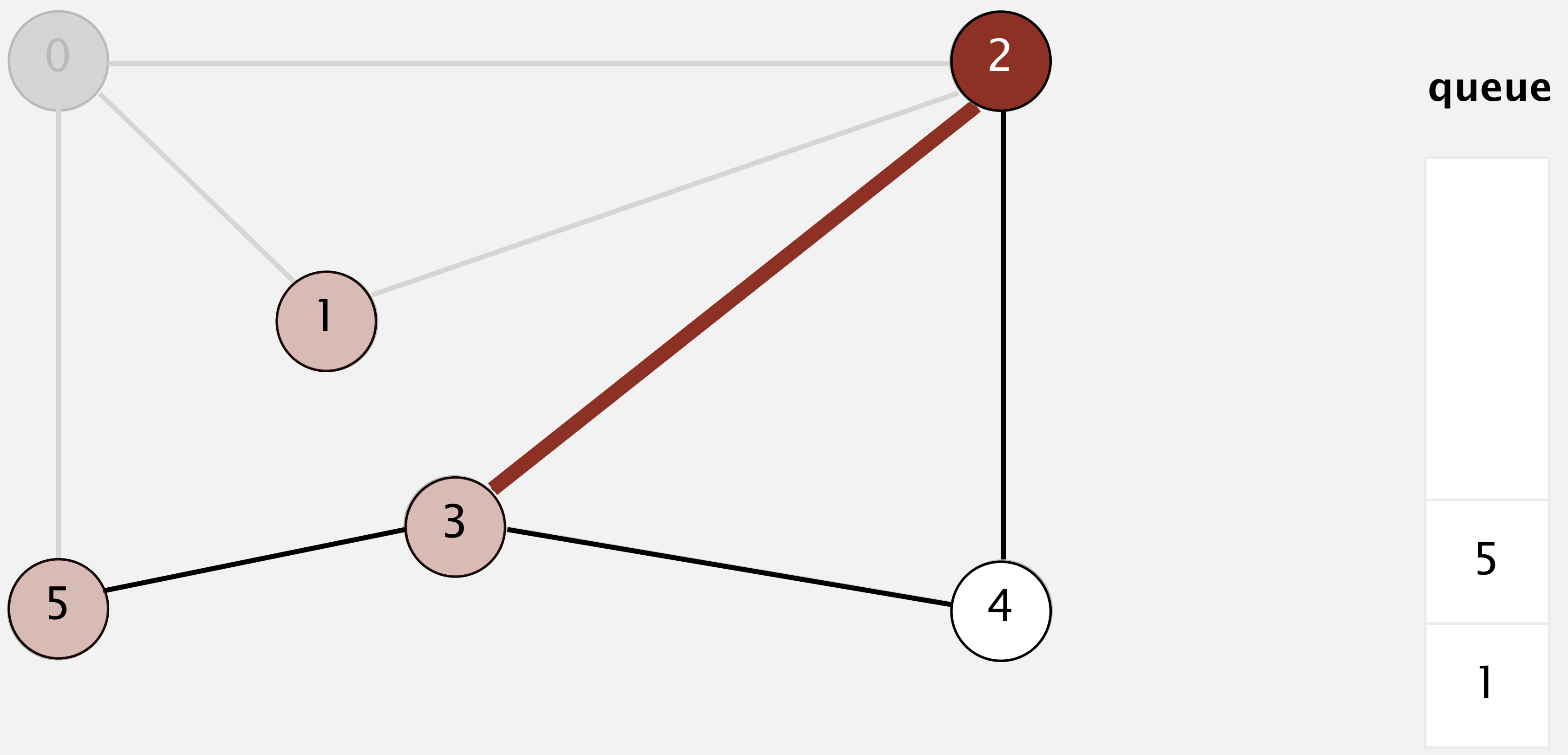


dequeue 2

Breadth-first search demo

Repeat until queue is empty:

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

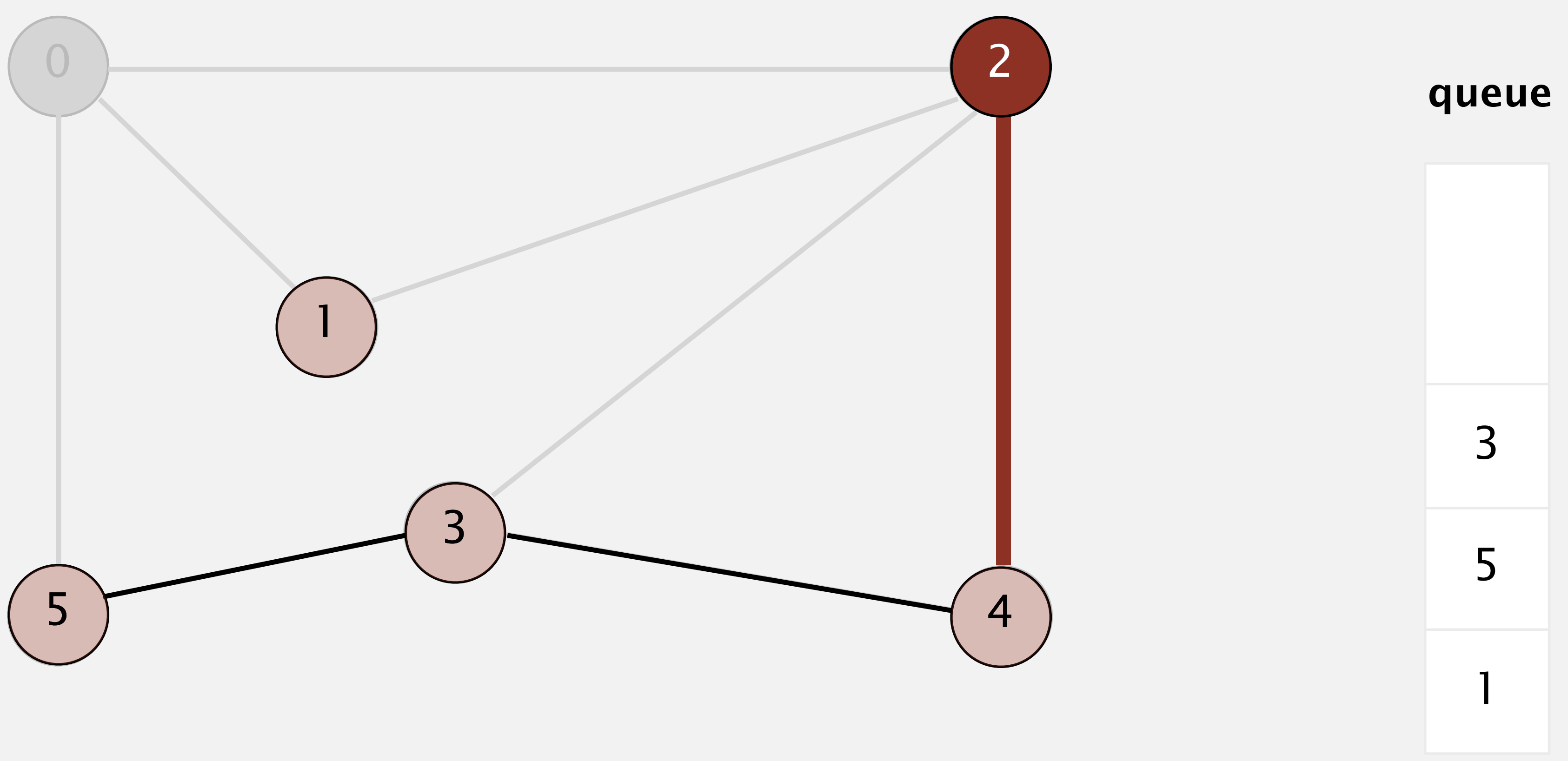


dequeue 2

Breadth-first search demo

Repeat until queue is empty:

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

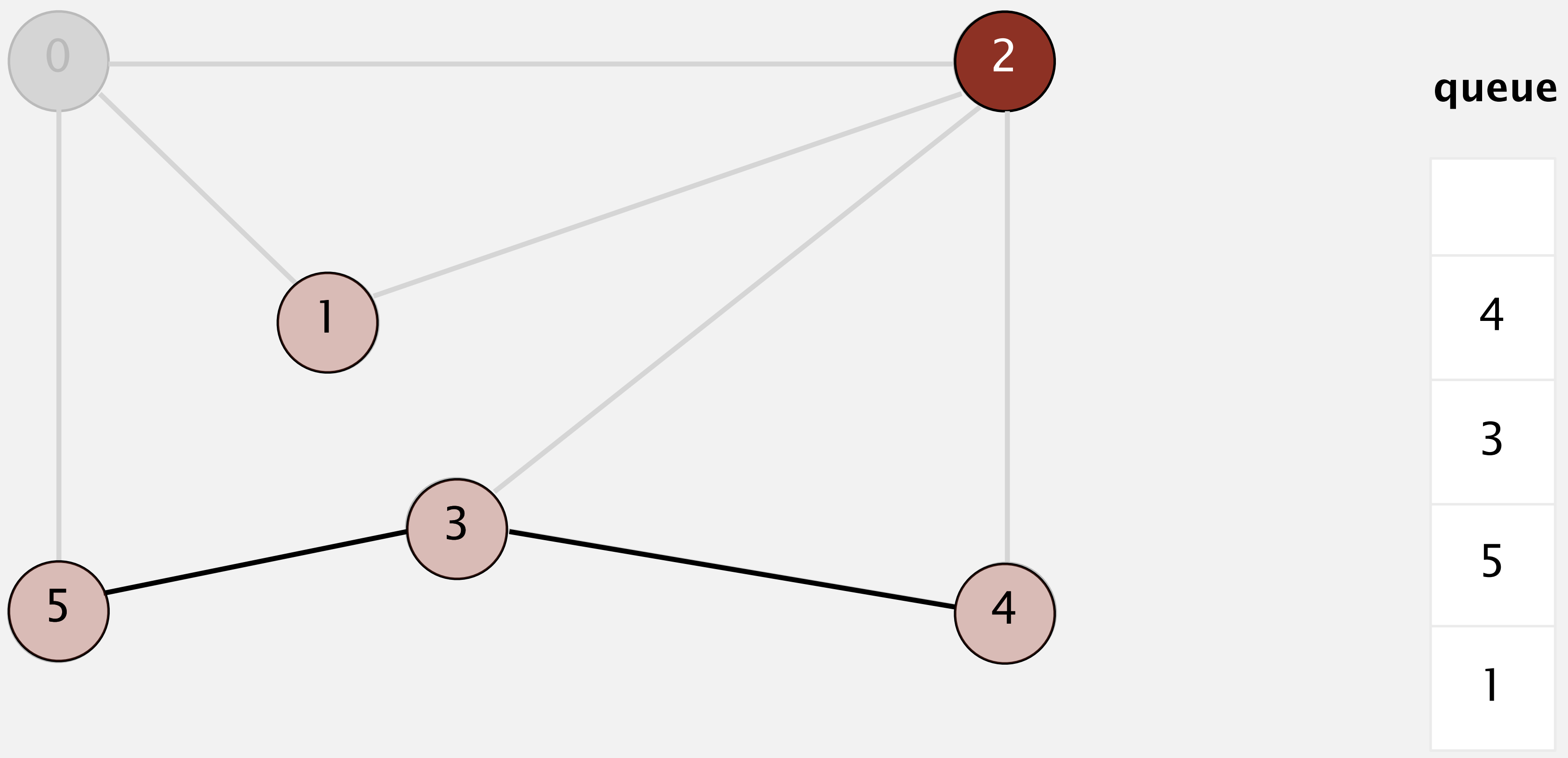


dequeue 2

Breadth-first search demo

Repeat until queue is empty:

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

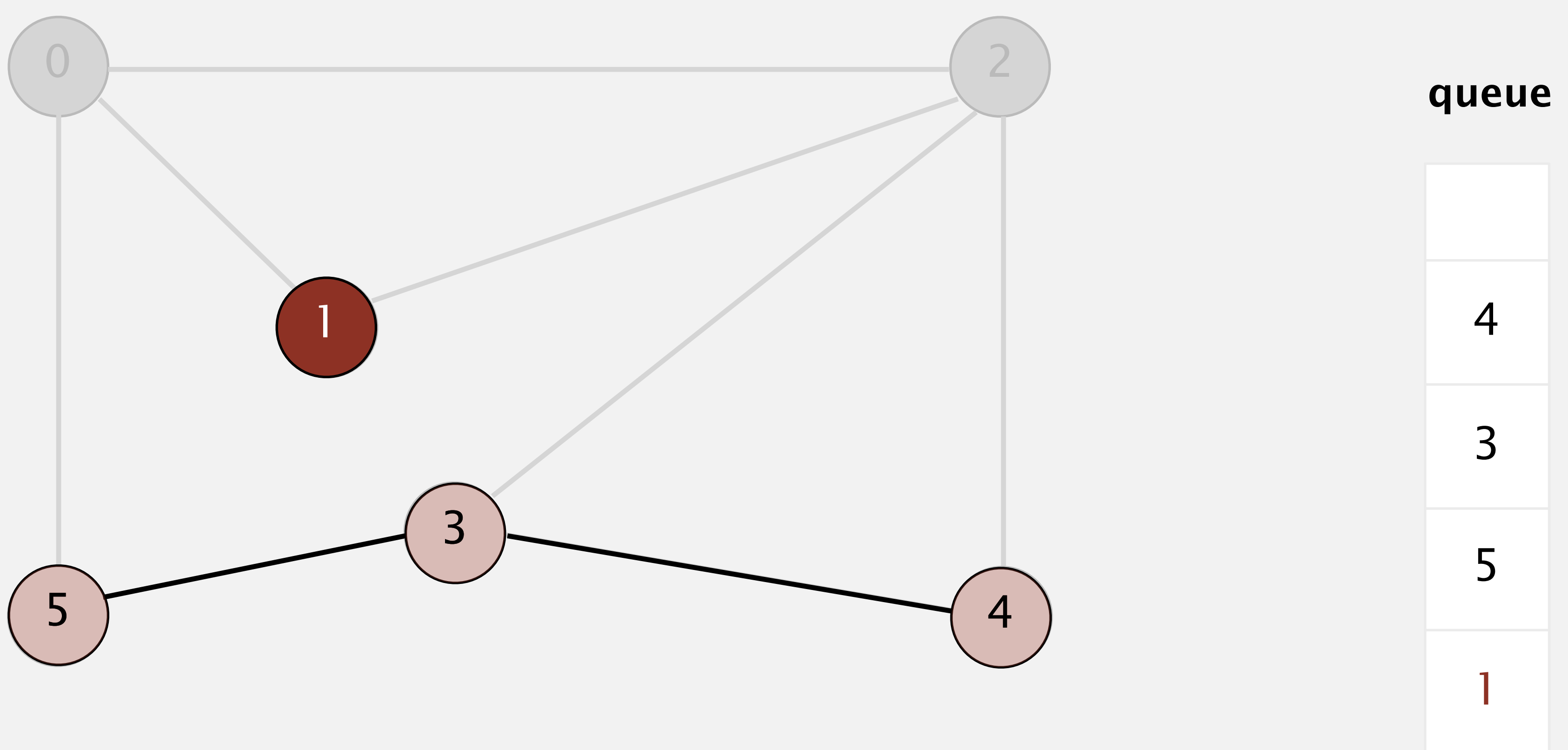


2 done

Breadth-first search demo

Repeat until queue is empty:

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

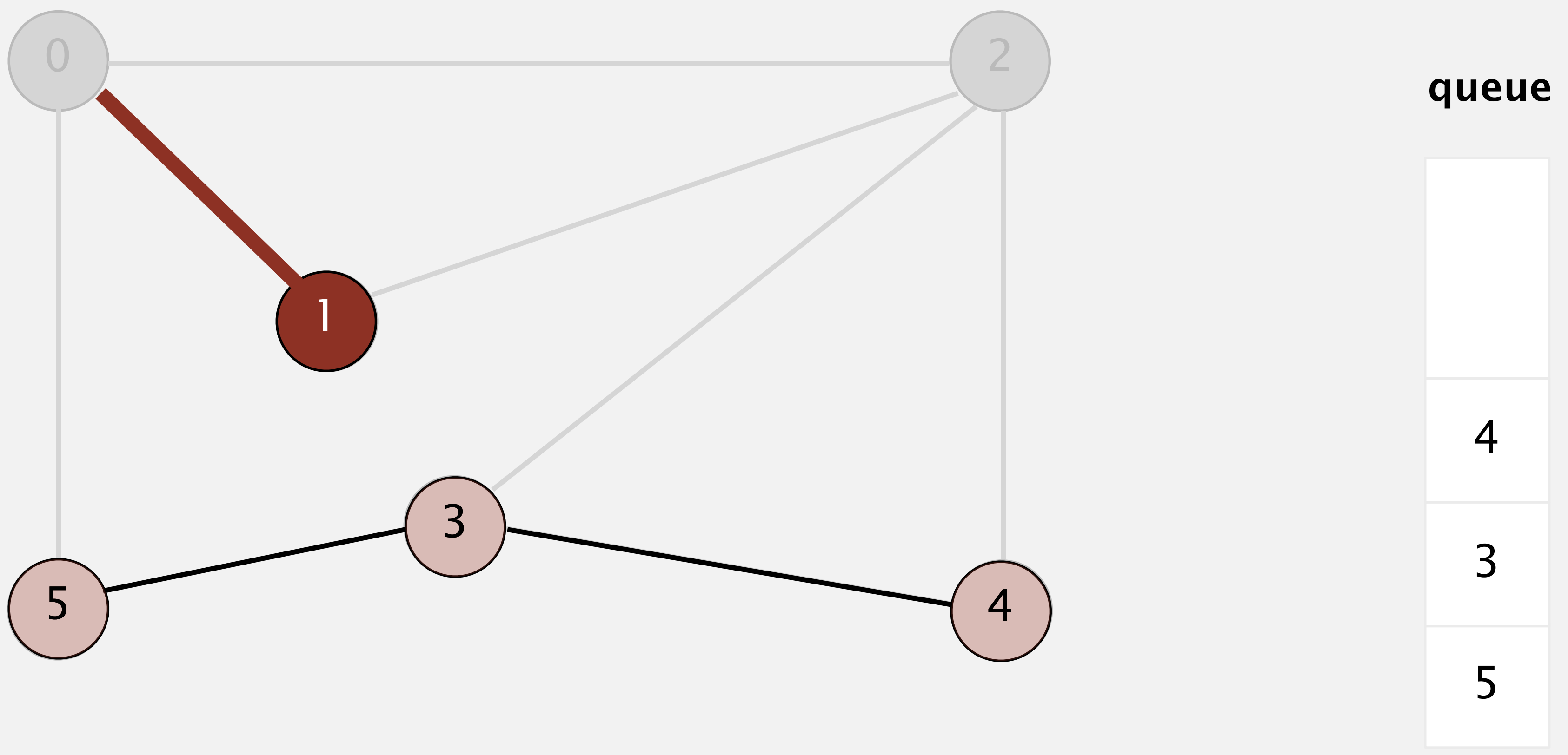


dequeue 1

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex *v* from queue.
- Add to queue all unmarked vertices adjacent to *v* and mark them.

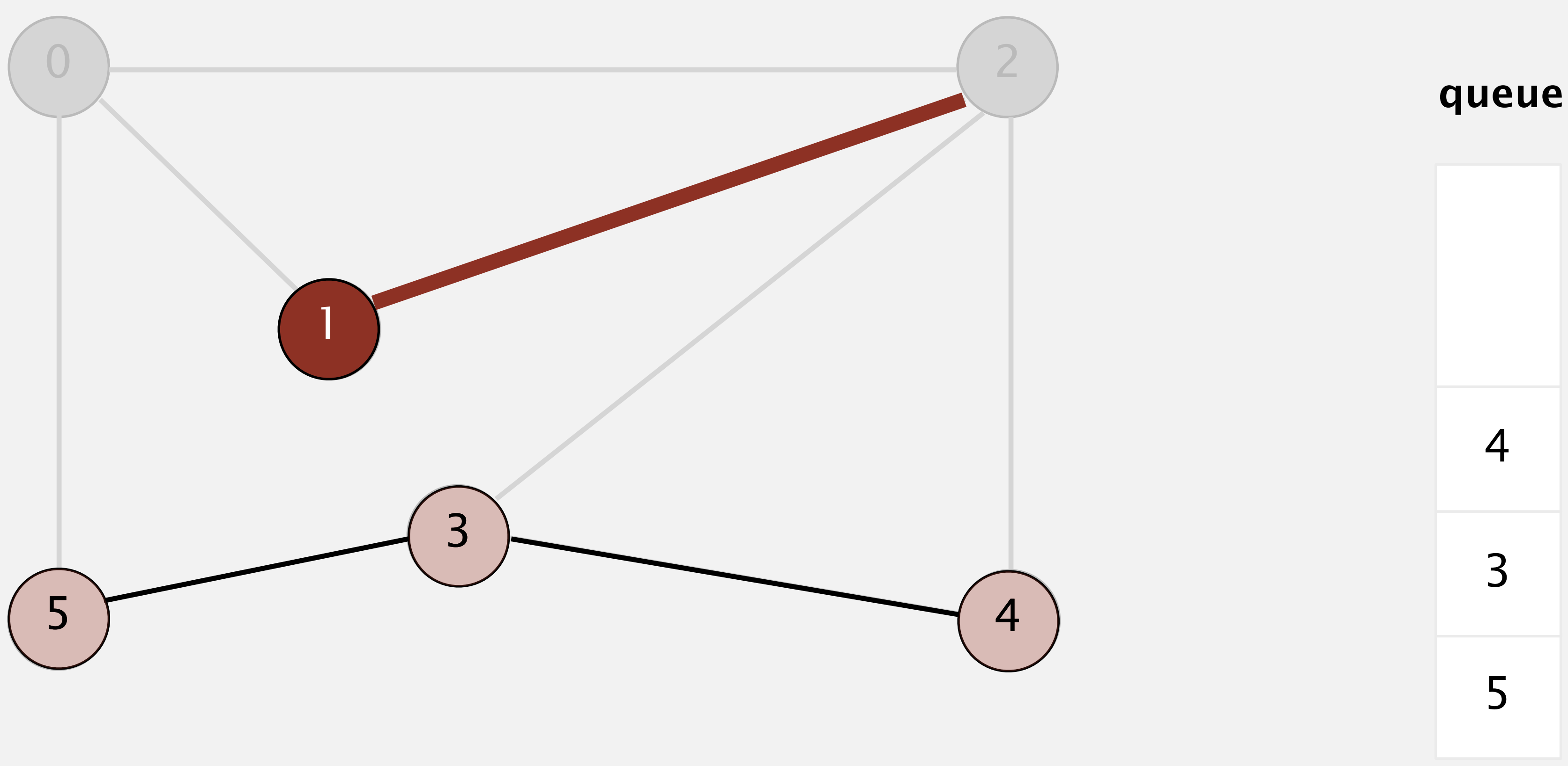


dequeue 1

Breadth-first search demo

Repeat until queue is empty:

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

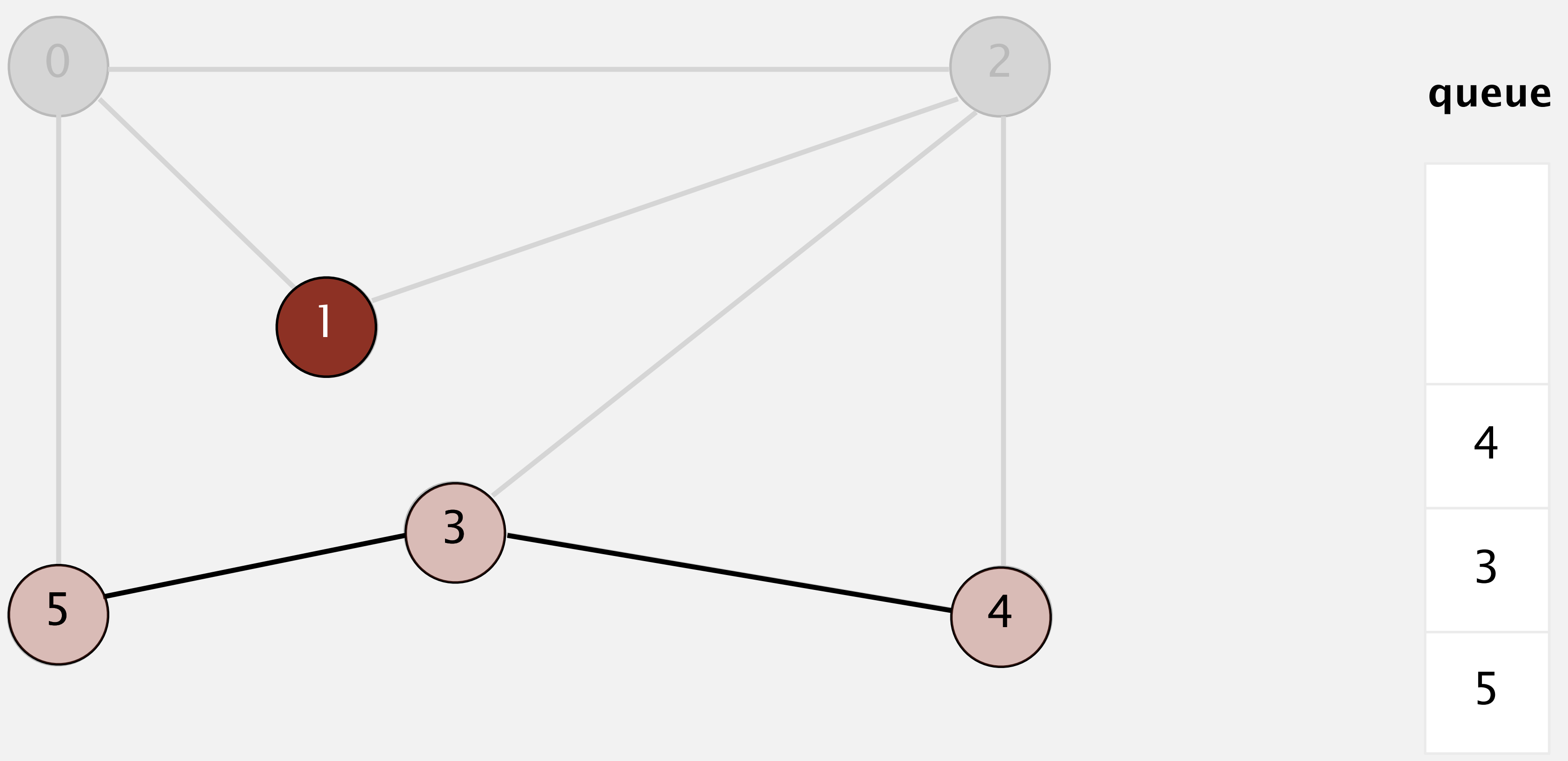


dequeue 1

Breadth-first search demo

Repeat until queue is empty:

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

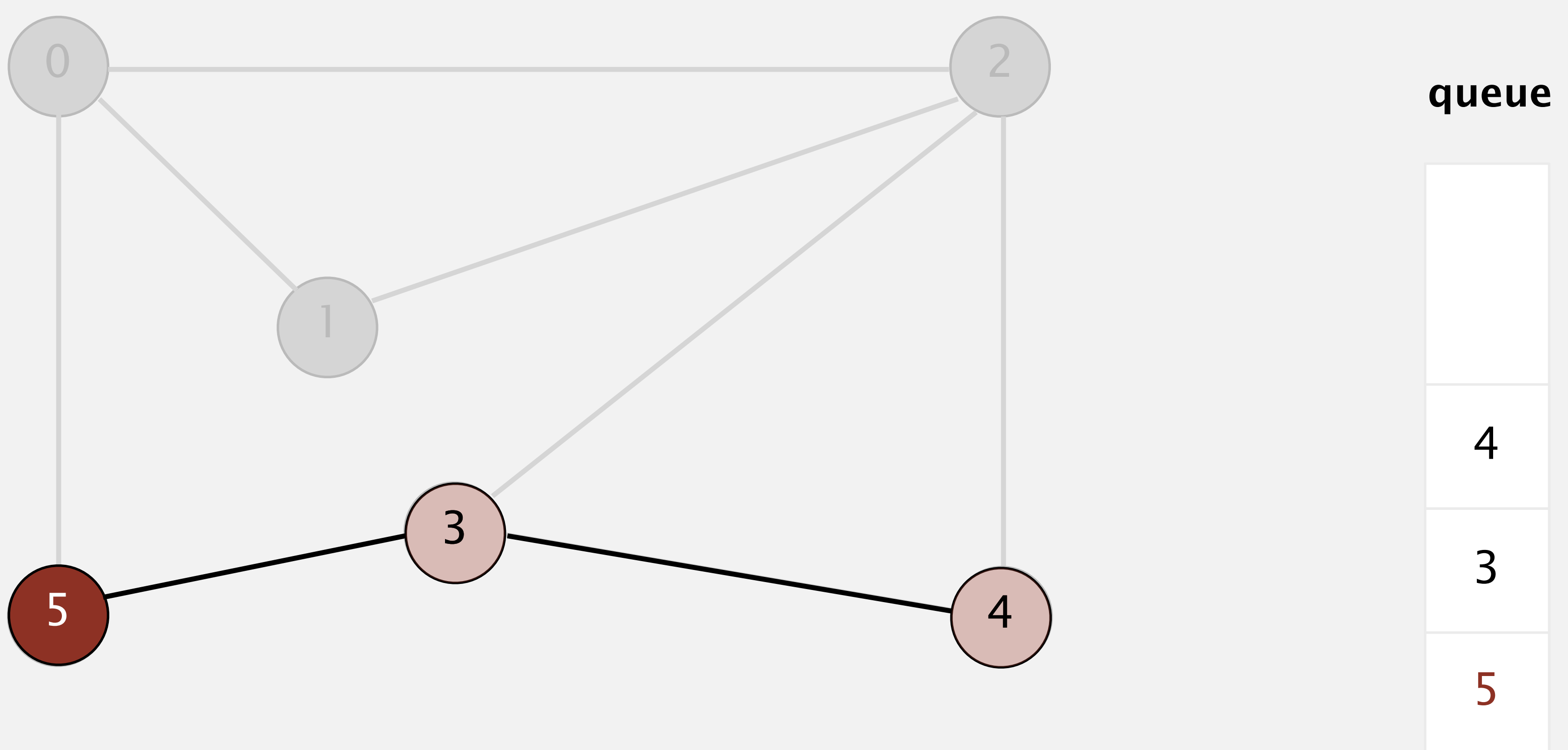


1 done

Breadth-first search demo

Repeat until queue is empty:

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

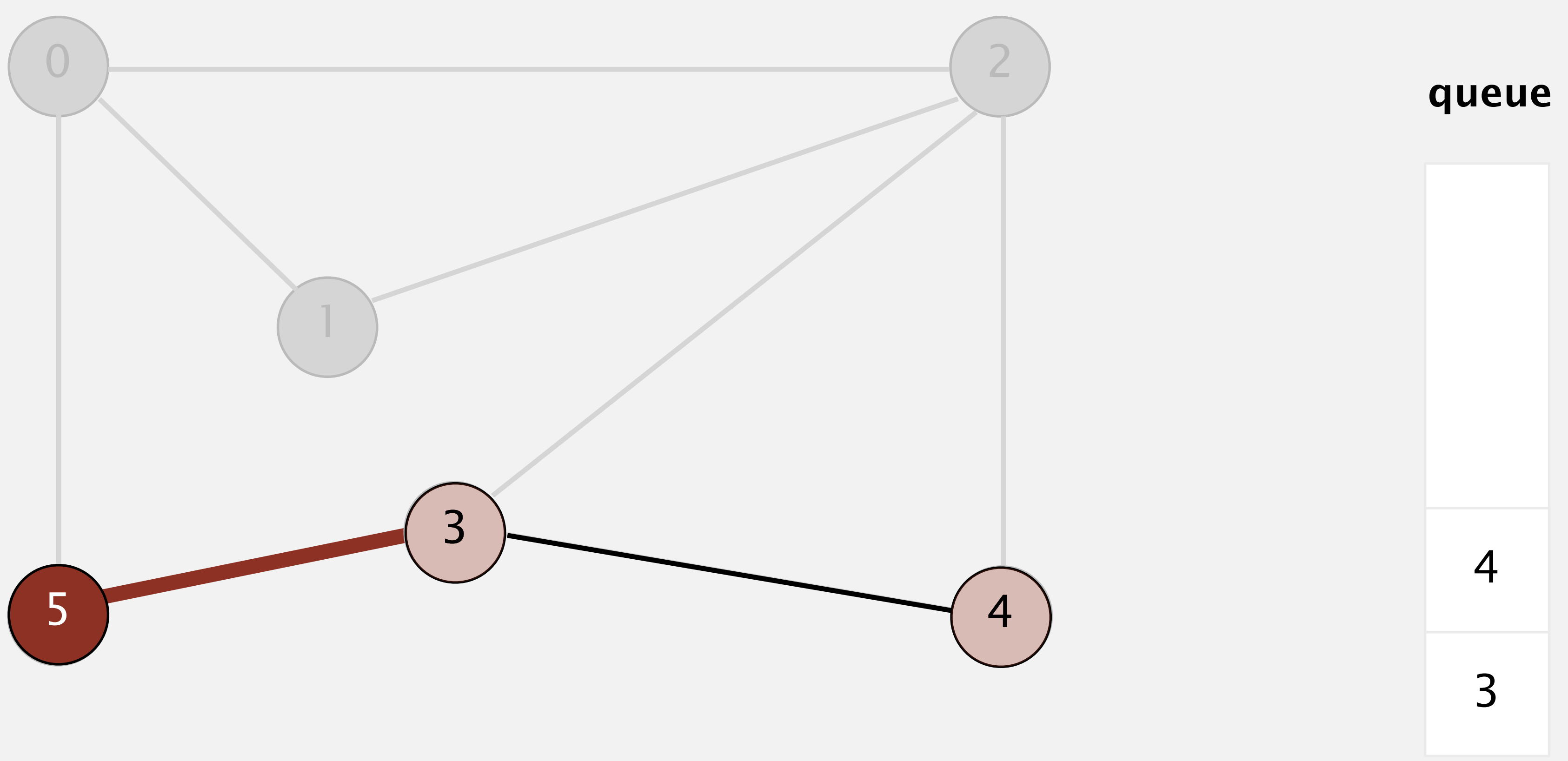


dequeue 5

Breadth-first search demo

Repeat until queue is empty:

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

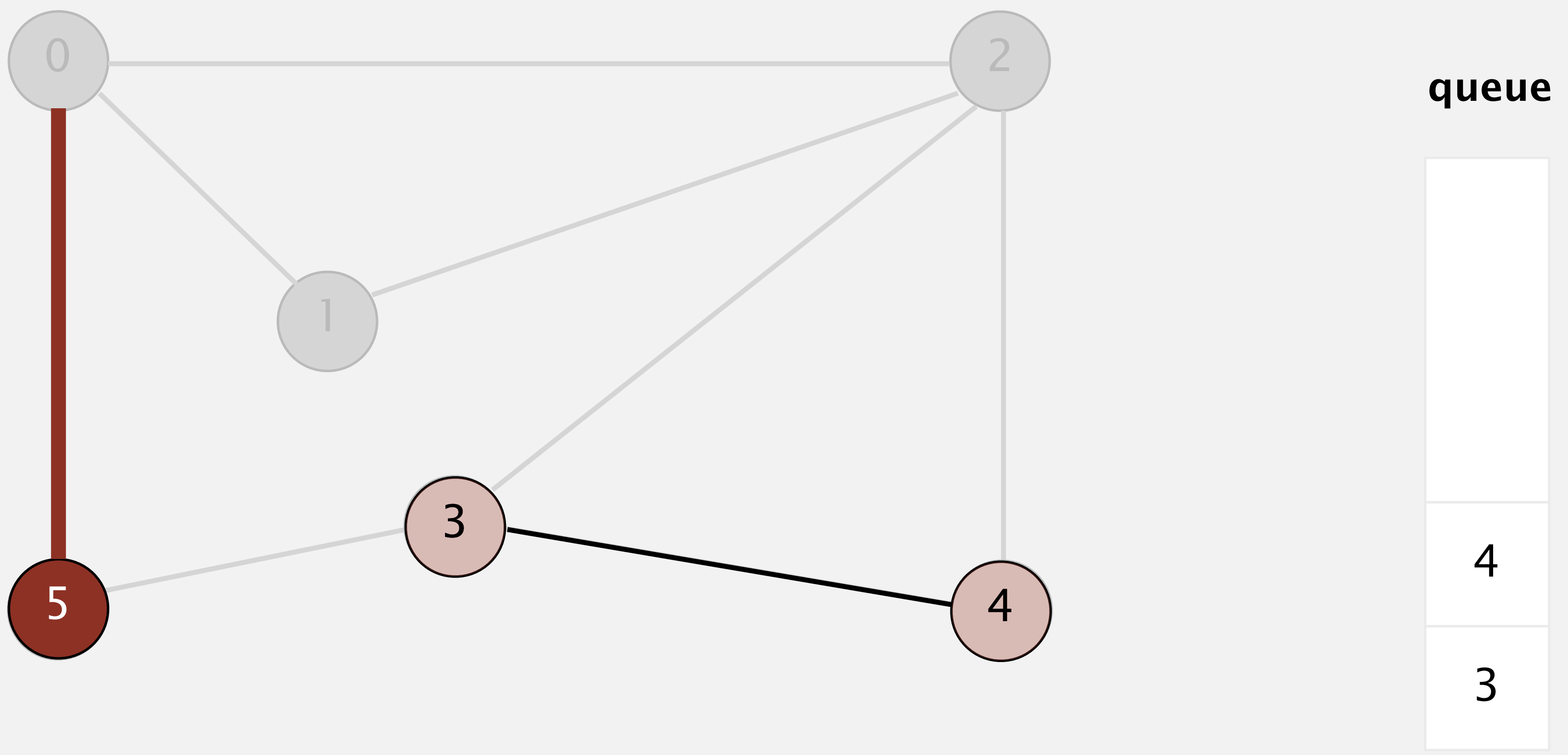


dequeue 5

Breadth-first search demo

Repeat until queue is empty:

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

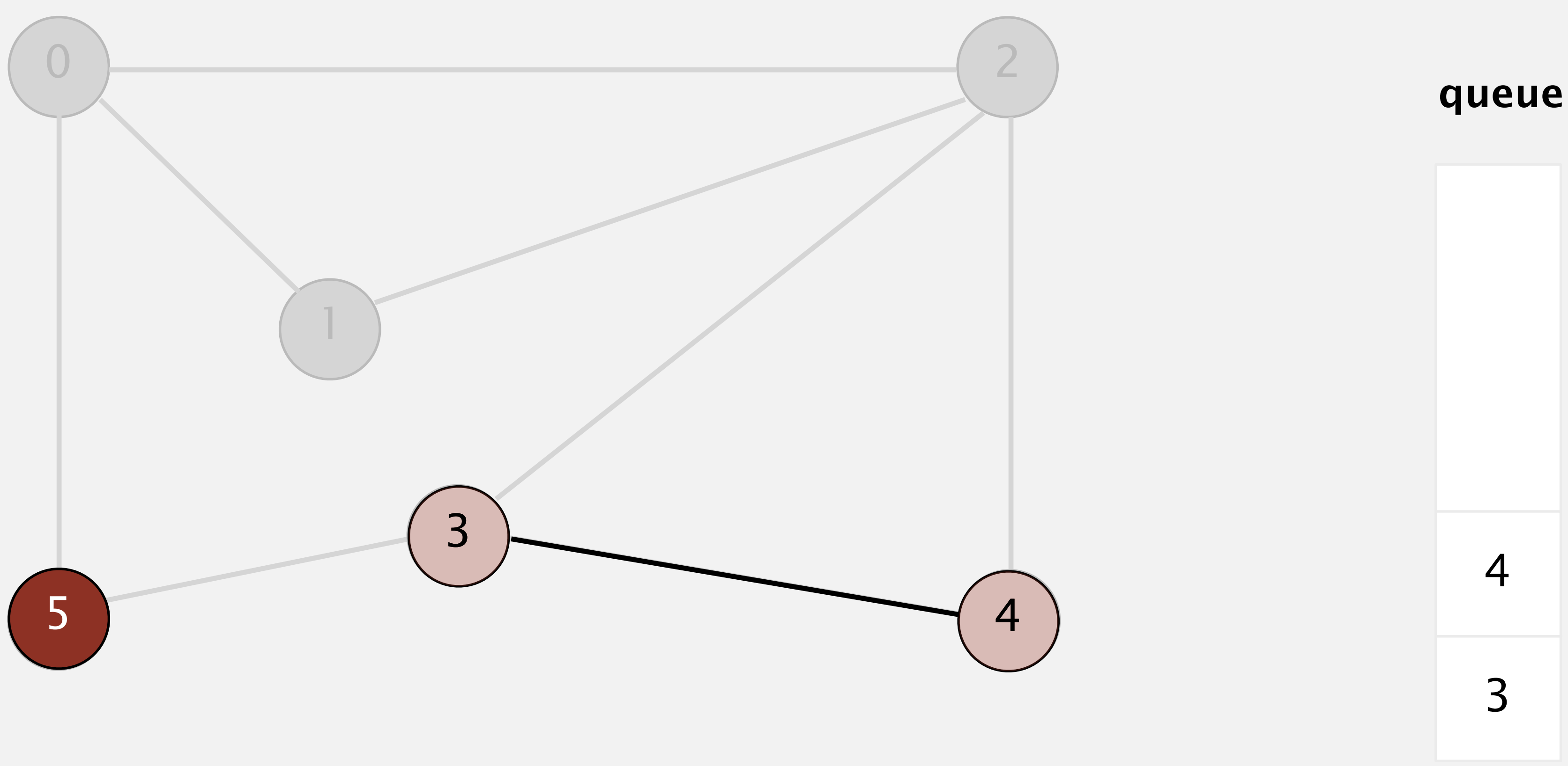


dequeue 5

Breadth-first search demo

Repeat until queue is empty:

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

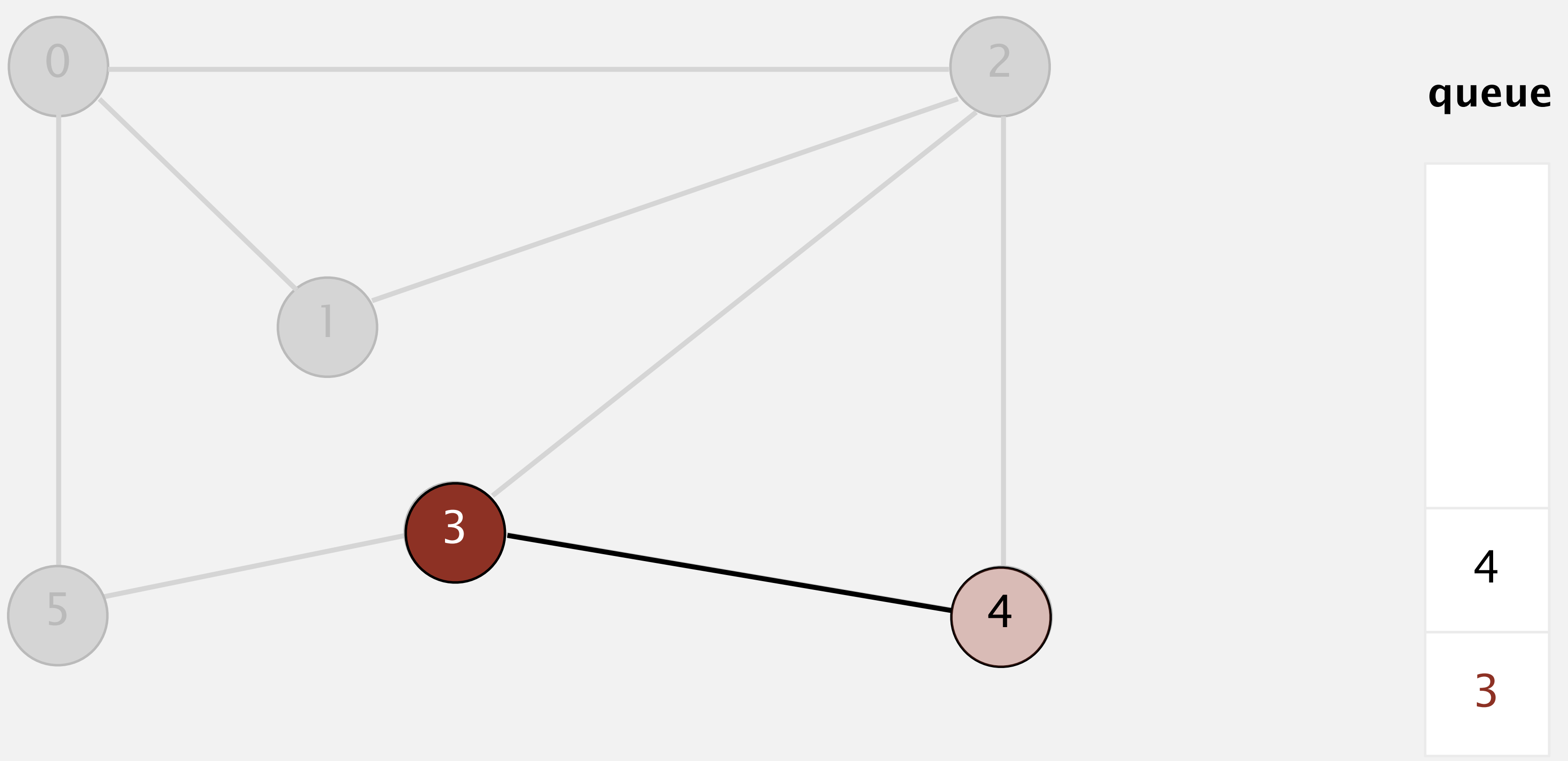


5 done

Breadth-first search demo

Repeat until queue is empty:

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

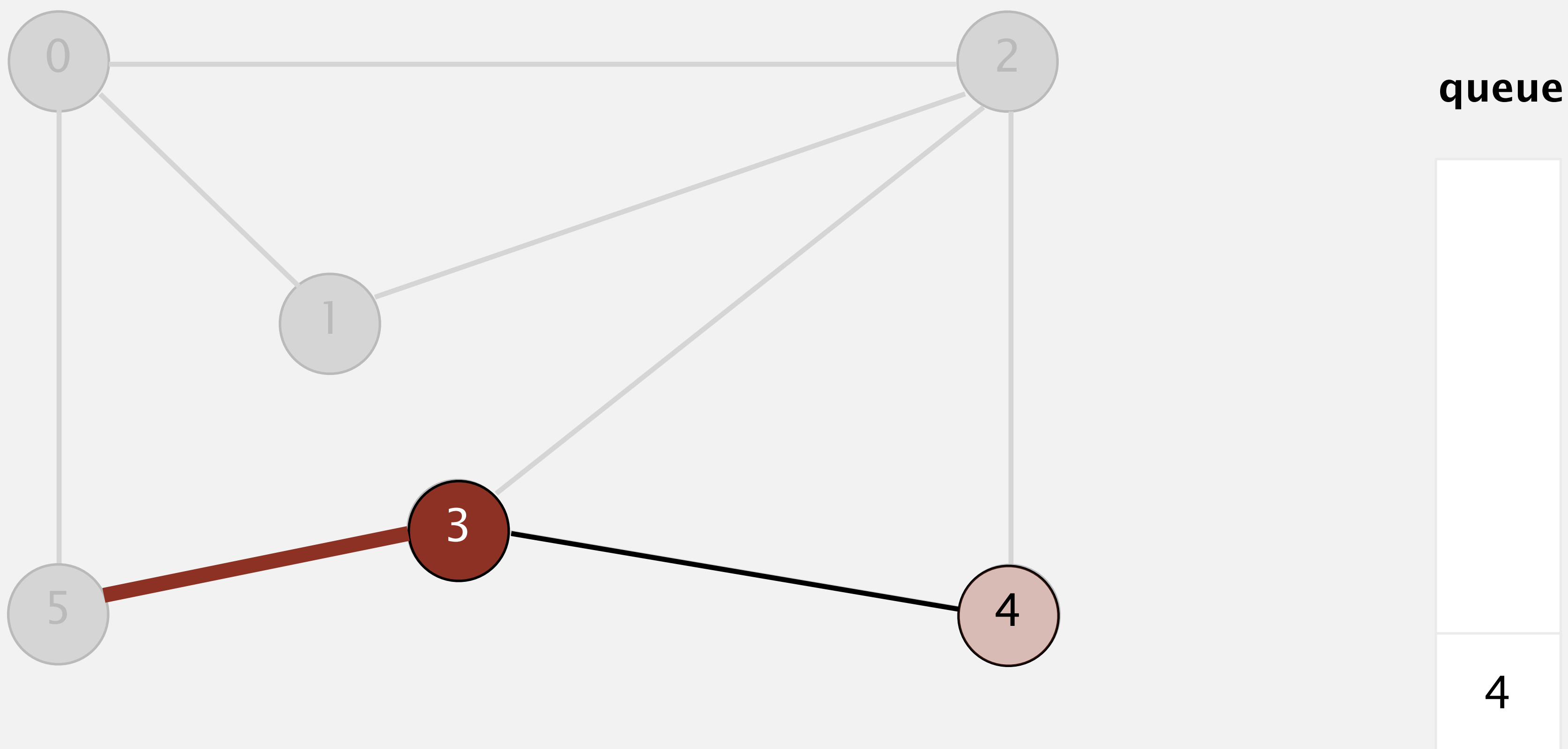


dequeue 3

Breadth-first search demo

Repeat until queue is empty:

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

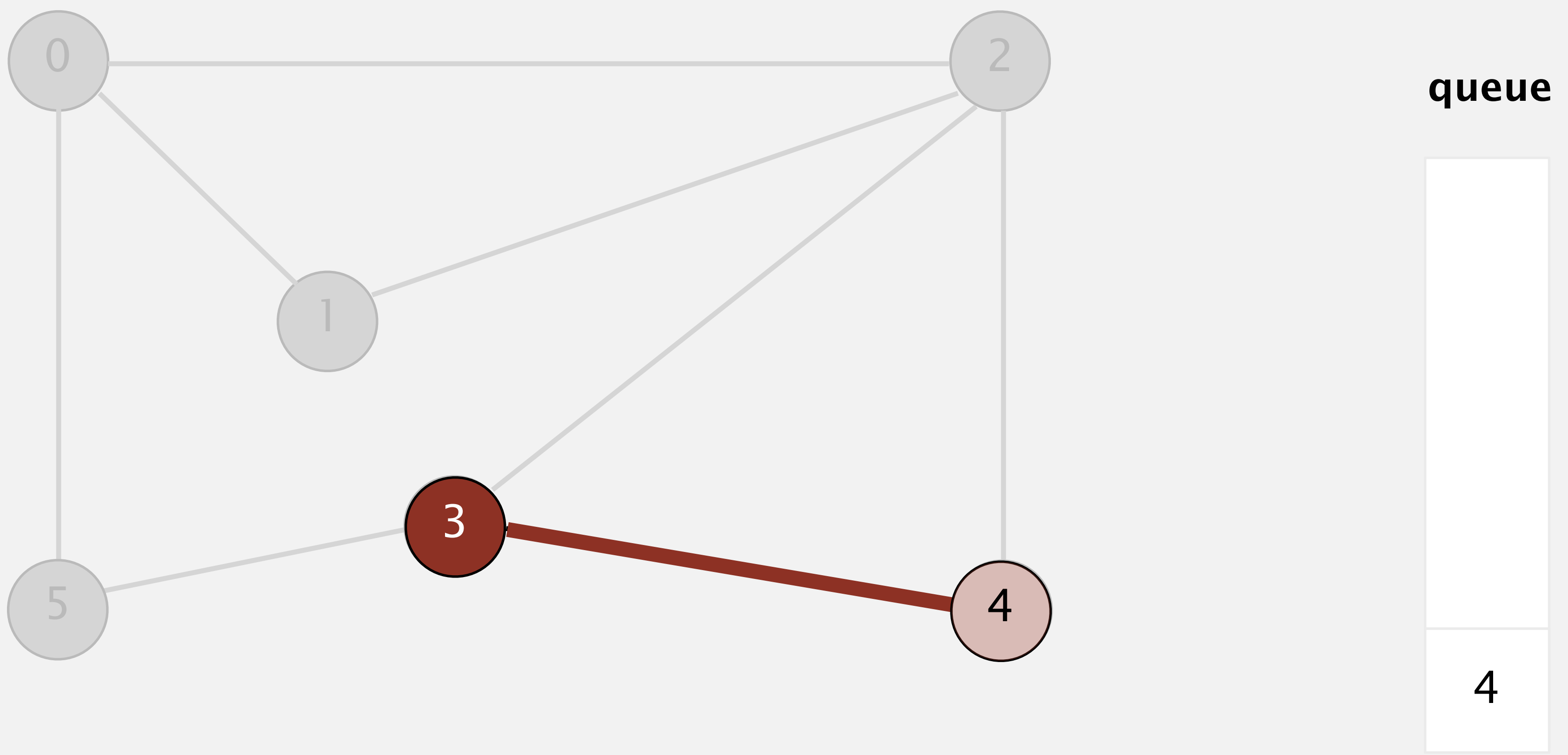


dequeue 3

Breadth-first search demo

Repeat until queue is empty:

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

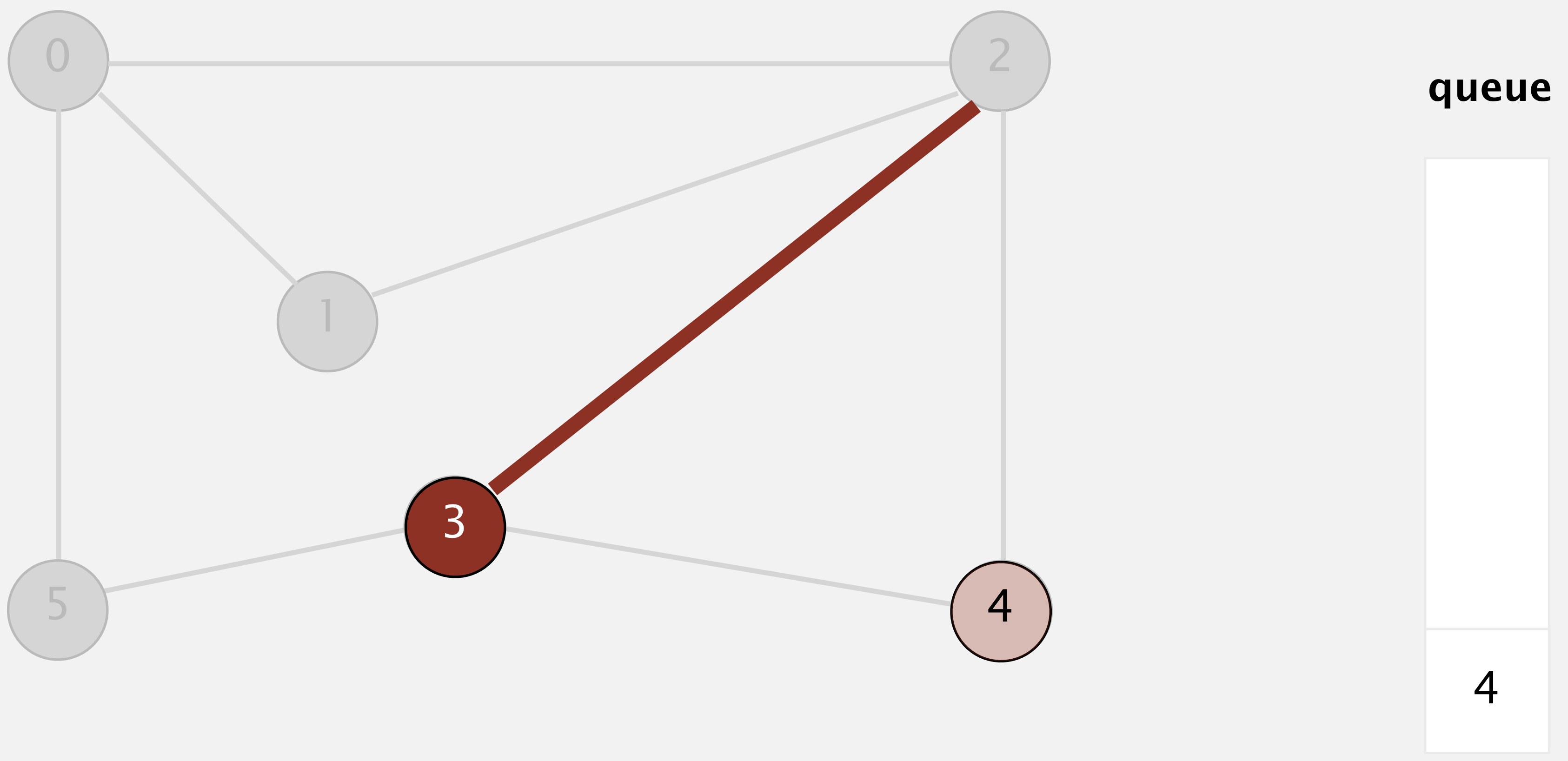


dequeue 3

Breadth-first search demo

Repeat until queue is empty:

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

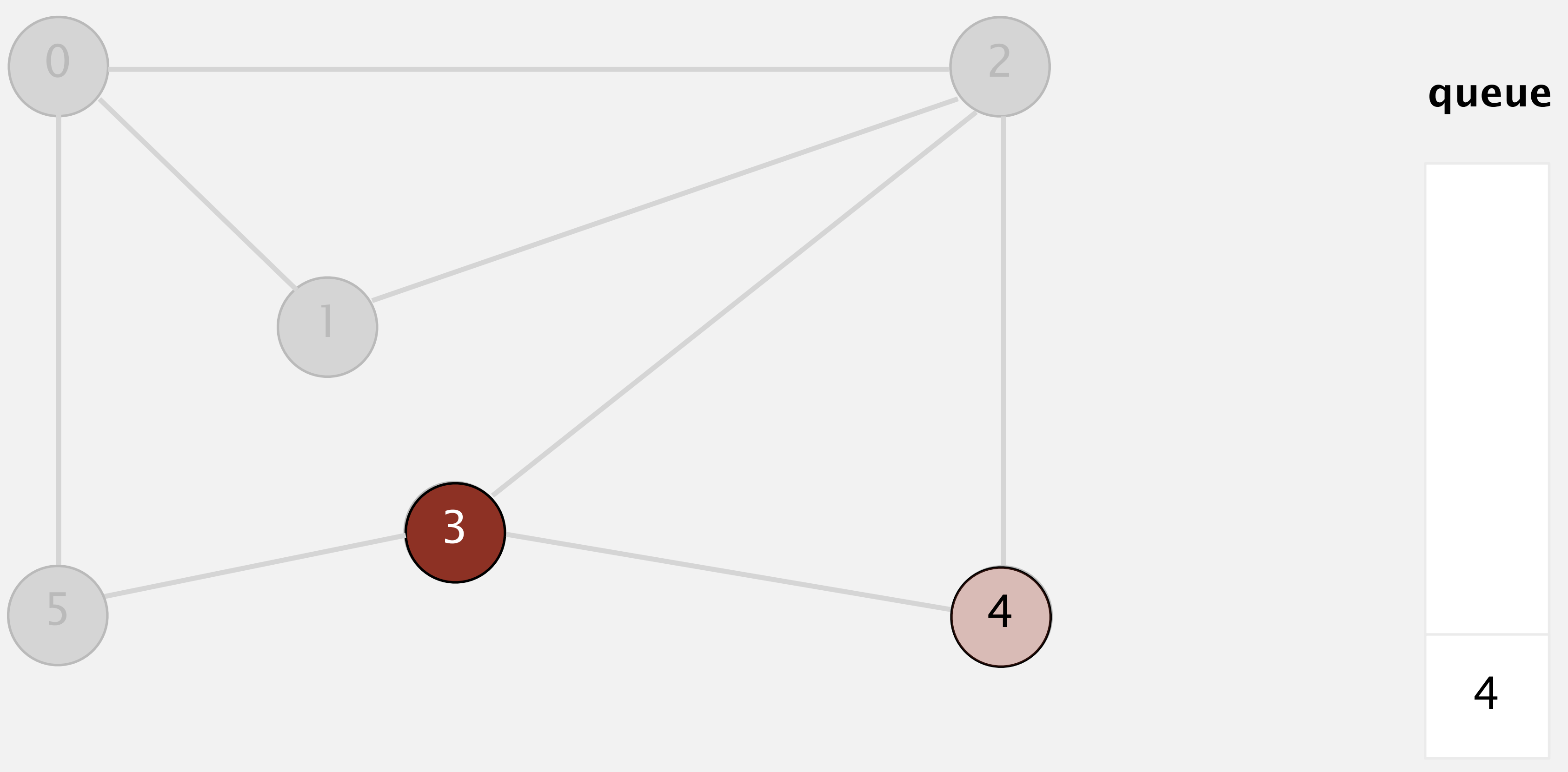


dequeue 3

Breadth-first search demo

Repeat until queue is empty:

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

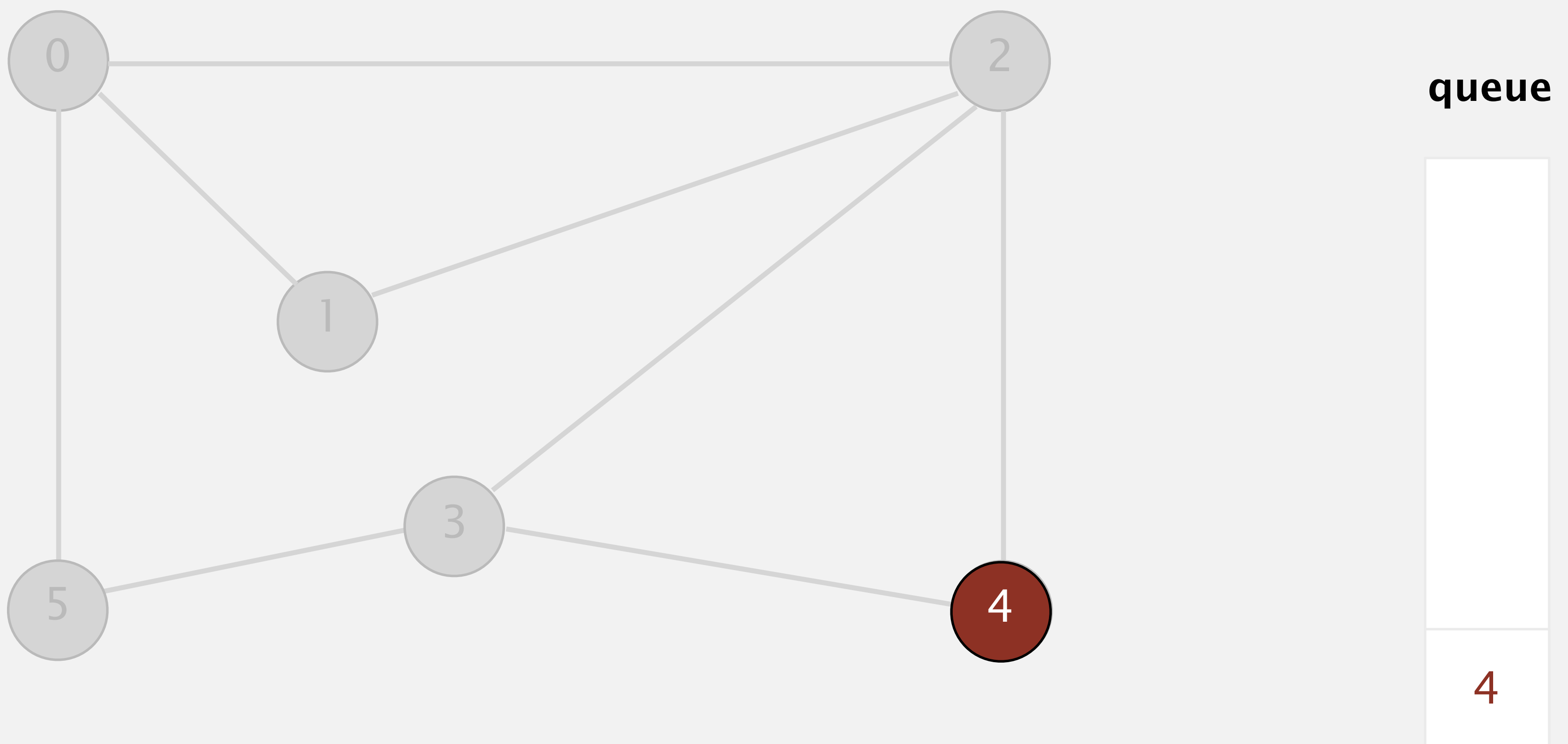


3 done

Breadth-first search demo

Repeat until queue is empty:

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

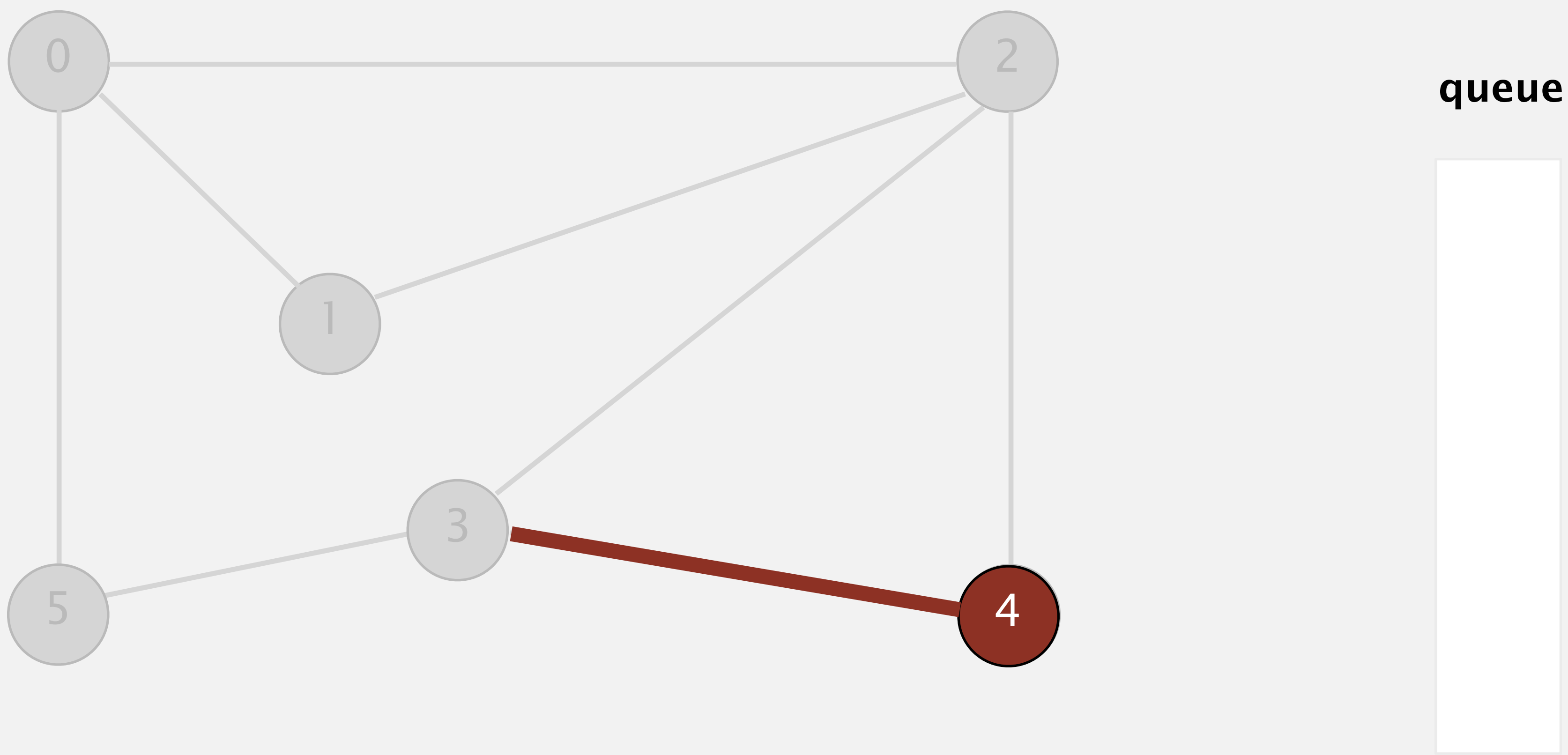


dequeue 4

Breadth-first search demo

Repeat until queue is empty:

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

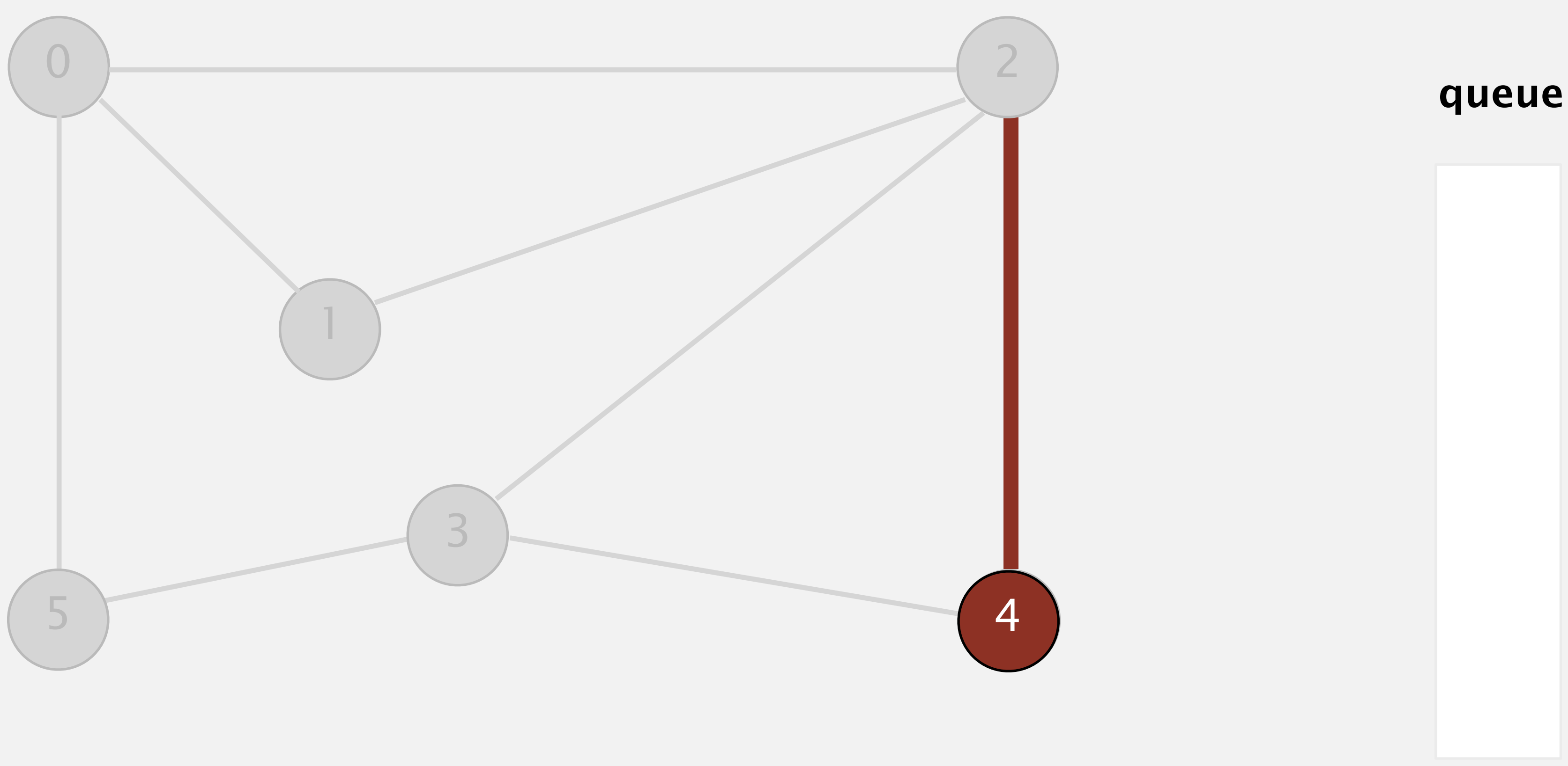


dequeue 4

Breadth-first search demo

Repeat until queue is empty:

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

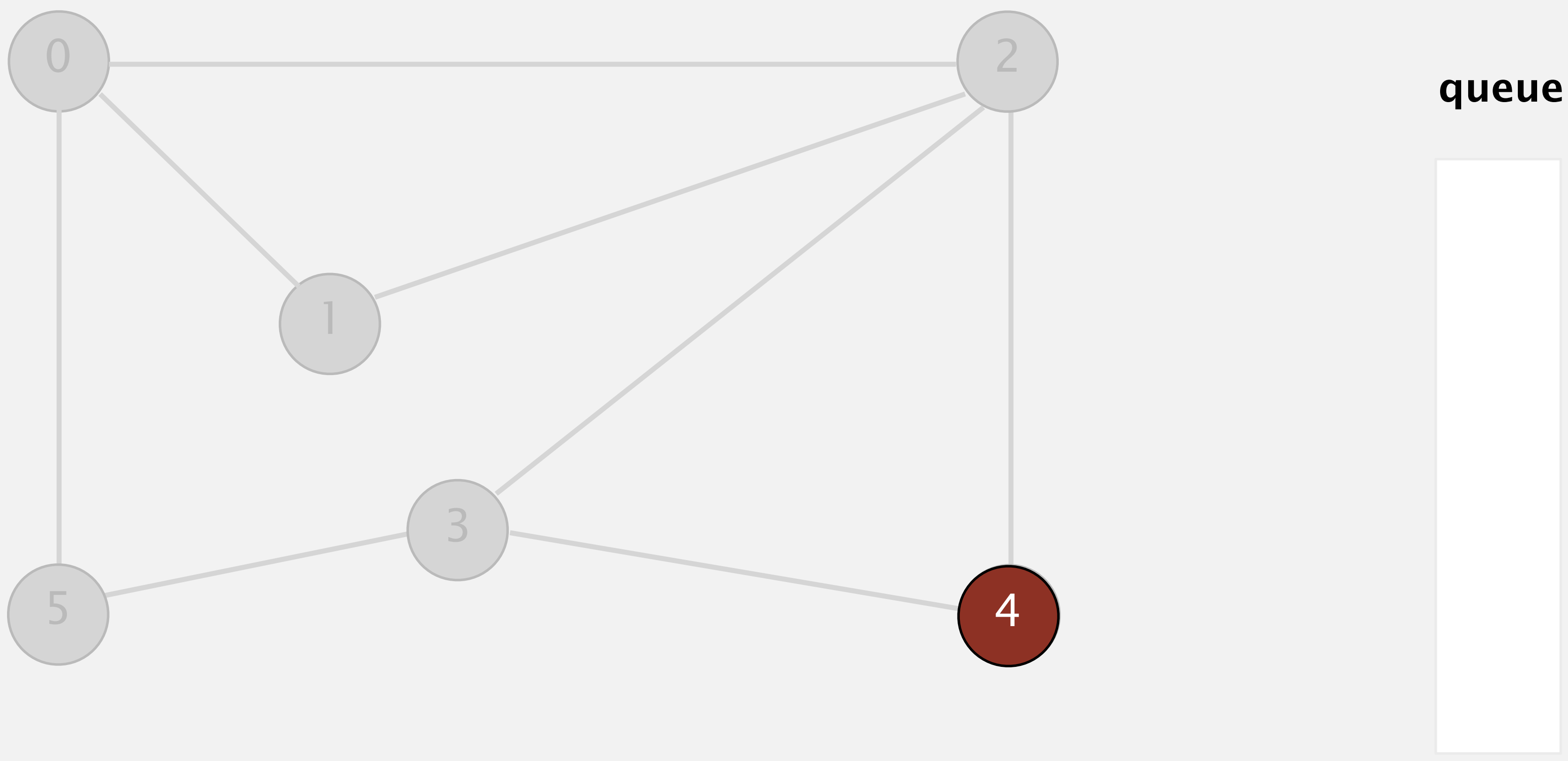


dequeue 4

Breadth-first search demo

Repeat until queue is empty:

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

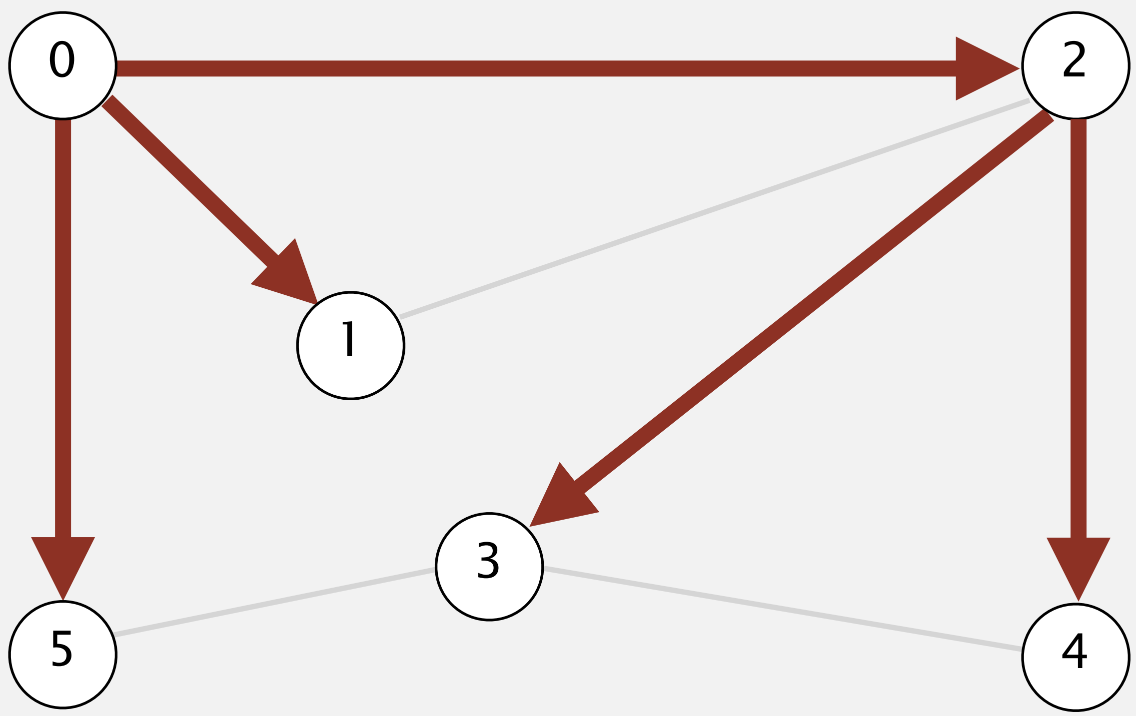


4 done

Breadth-first search demo

Repeat until queue is empty:

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



done

Breadth-first search: Java implementation

```
public class BreadthFirstPaths
{
   private boolean[] marked;
   private int[] edgeTo;
   private int[] distTo;

   …

   private void bfs(Graph G, int s) {
      Queue<Integer> q = new Queue<Integer>();
      q.enqueue(s);
      marked[s] = true;
      distTo[s] = 0;

      while (!q.isEmpty()) {
         int v = q.dequeue();
         for (int w : G.adj(v)) {
            if (!marked[w]) {
               q.enqueue(w);
               marked[w] = true;
               edgeTo[w] = v;
               distTo[w] = distTo[v] + 1;
            }
         }
      }
   }
}
```

initialize FIFO queue of vertices to explore

found new vertex w via edge v-w

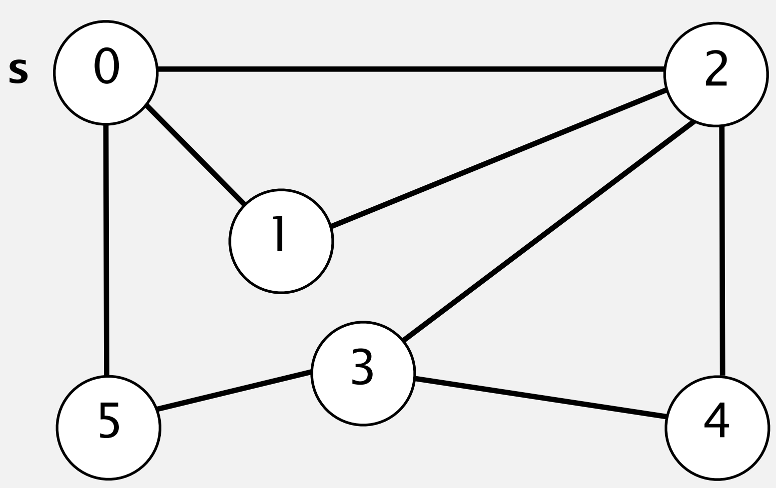
Breadth-first search properties

Q. In which order does BFS examine vertices?

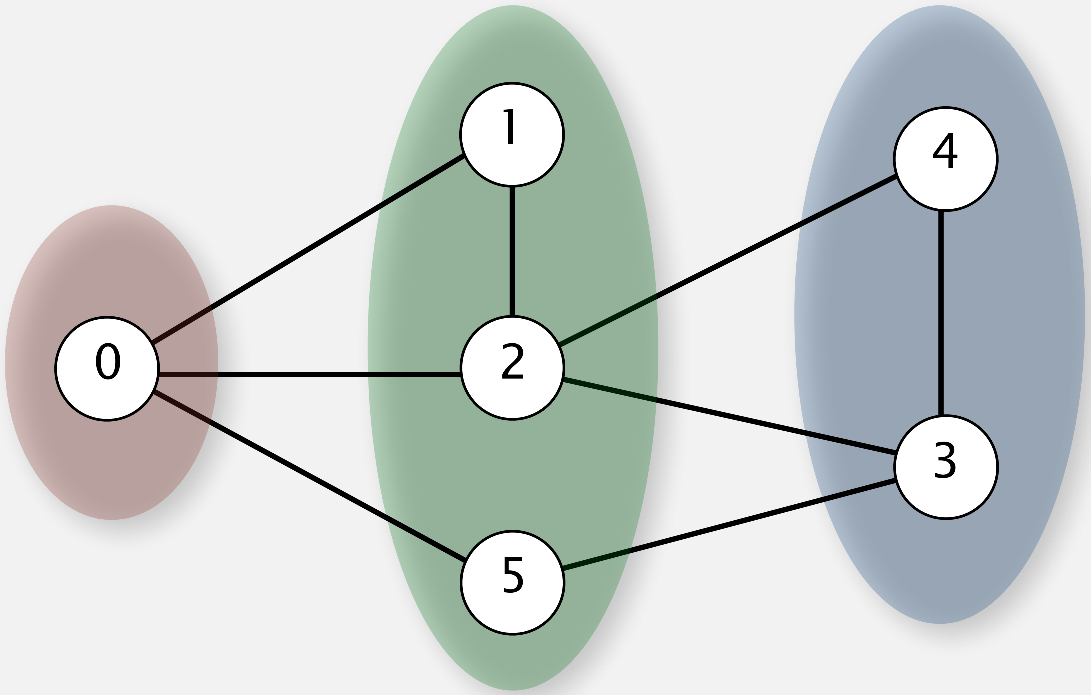
A. Increasing distance (number of edges) from s.

queue always consists of ≥ 0 vertices of distance k from s,
followed by ≥ 0 vertices of distance $k+1$

Proposition. In any connected graph G, BFS computes shortest paths from s to all other vertices in time proportional to $E + V$.



graph G



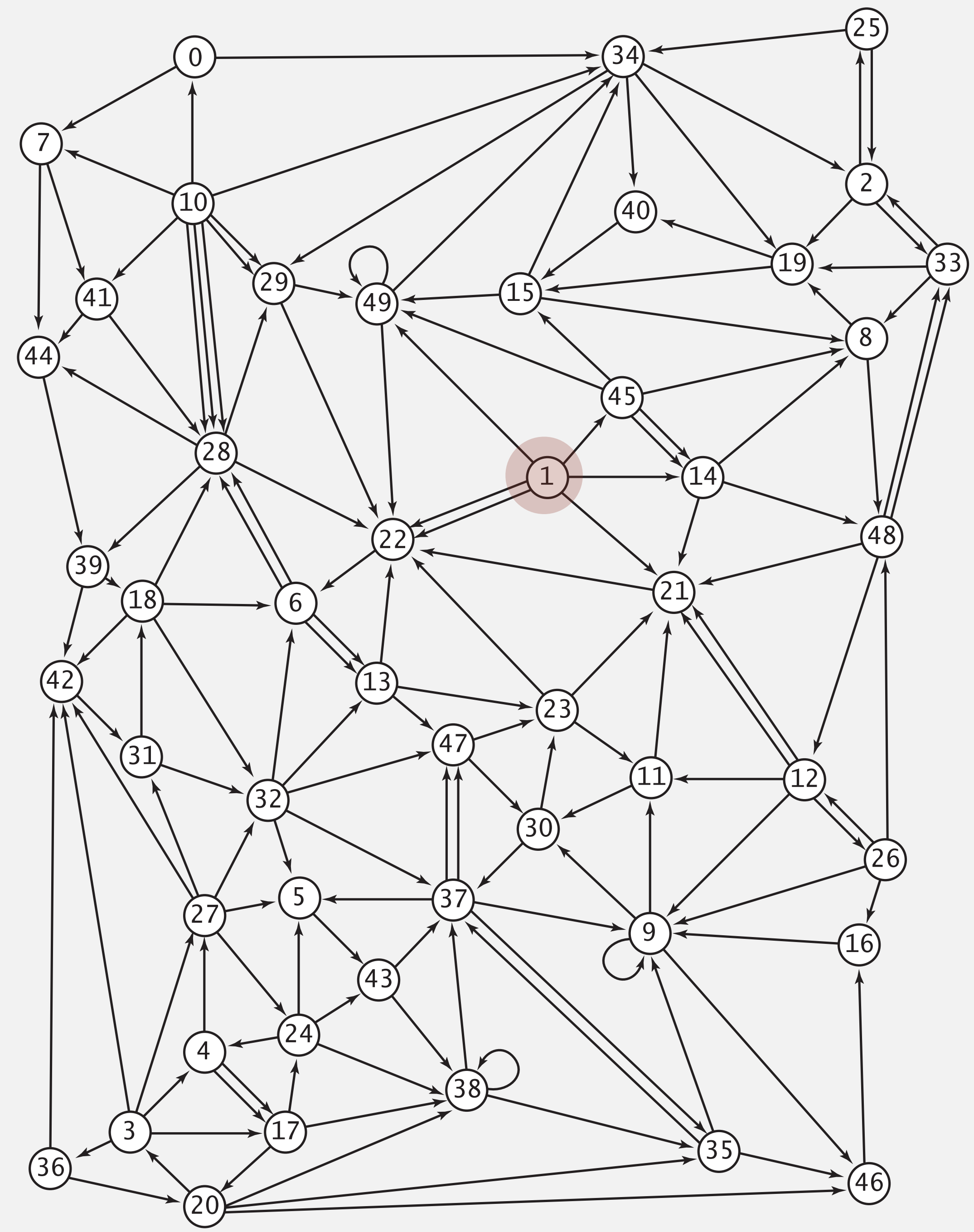
dist = 0 dist = 1 dist = 2

Breadth-first search in digraphs application: web crawler

Goal. Crawl web, starting from some root web page, say `www.viginia.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>();
SET<String> marked = new SET<String>();

String root = "http://www.virginia.edu";
queue.enqueue(root);
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+)";
   Pattern pattern = Pattern.compile(regexp);
   Matcher matcher = pattern.matcher(input);
   while (matcher.find())
   {
      String w = matcher.group();
      if (!marked.contains(w))
      {
         marked.add(w);
         queue.enqueue(w);
      }
   }
}
```

queue of websites to crawl

set of marked websites

start crawling from root website

read in raw html from next website in queue

use regular expression to find all URLs in website of form `http://xxx.yyy.zzz` [crude pattern misses relative URLs]

if unmarked, mark it and put on the queue

TOPOLOGICAL SORT

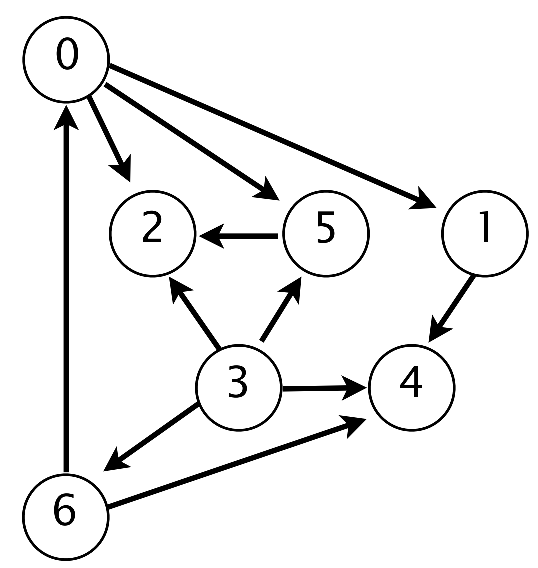
Precedence scheduling

Goal. Given a set of tasks to be completed with precedence constraints, in which order should we schedule the tasks?

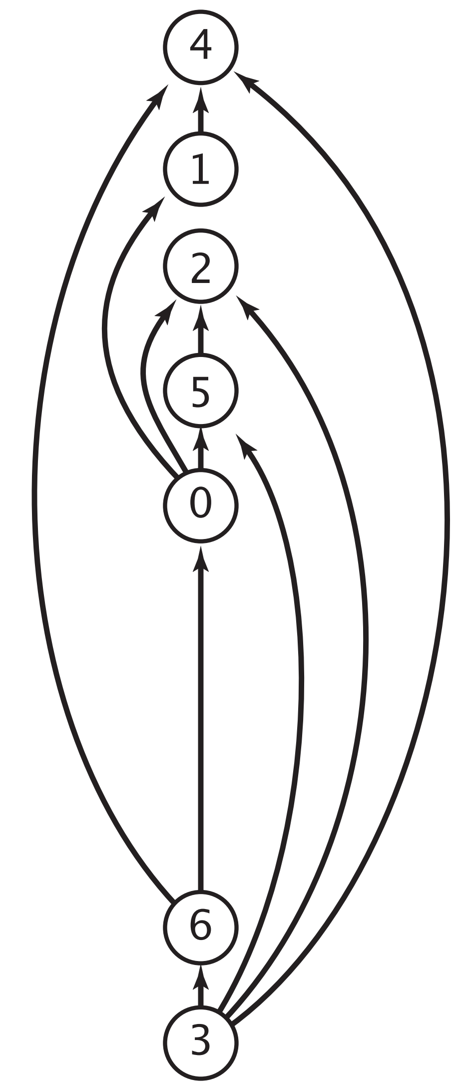
Digraph model. vertex = task; edge = precedence constraint.

0. Algorithms
1. Complexity Theory
2. Artificial Intelligence
3. Intro to CS
4. Cryptography
5. Scientific Computing
6. Advanced Programming

tasks



precedence constraint graph



feasible schedule

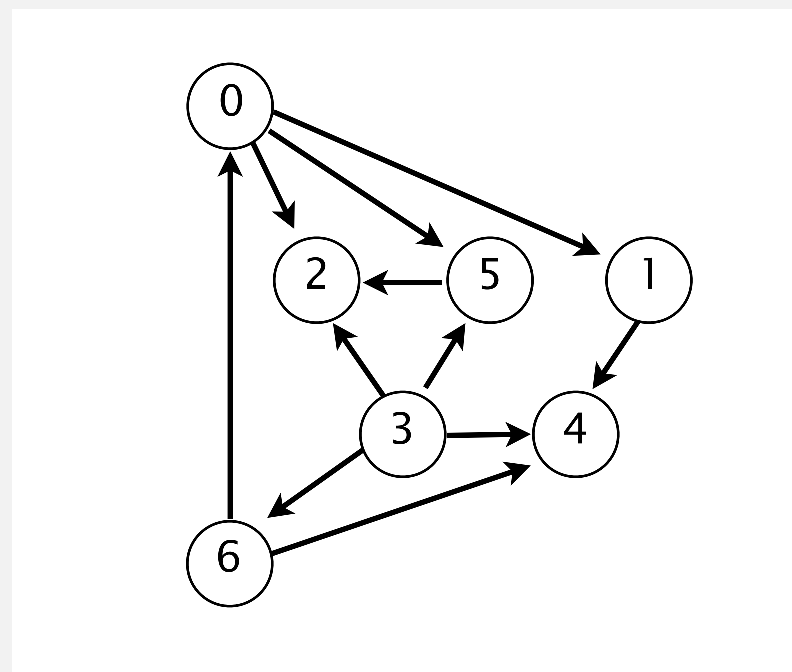
Topological sort

DAG. Directed acyclic graph.

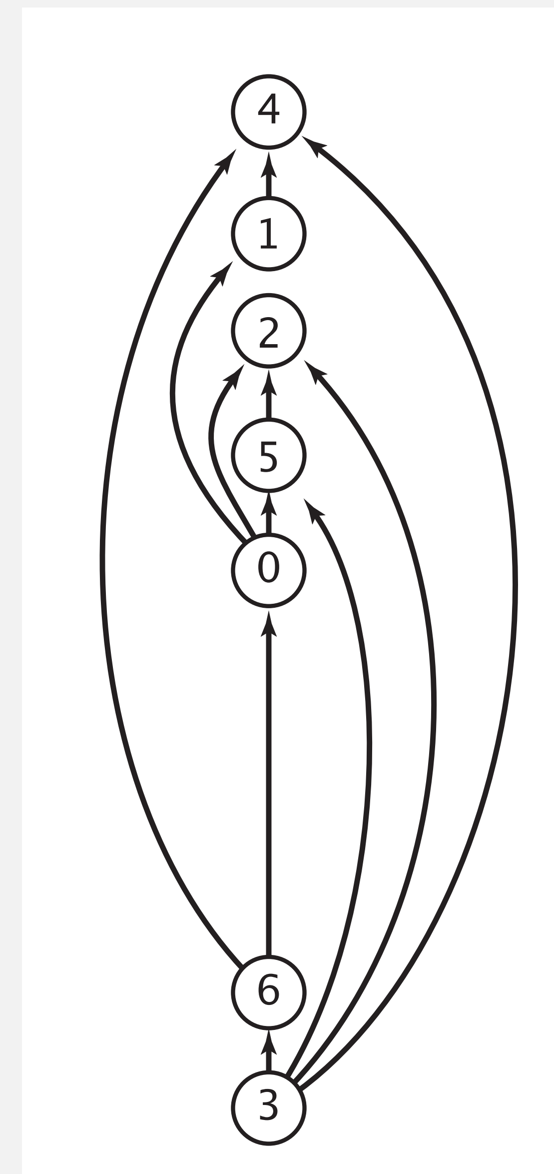
Topological sort. Redraw DAG so all edges point upwards.

0→5	0→2
0→1	3→6
3→5	3→4
5→2	6→4
6→0	3→2
1→4	

directed edges



DAG

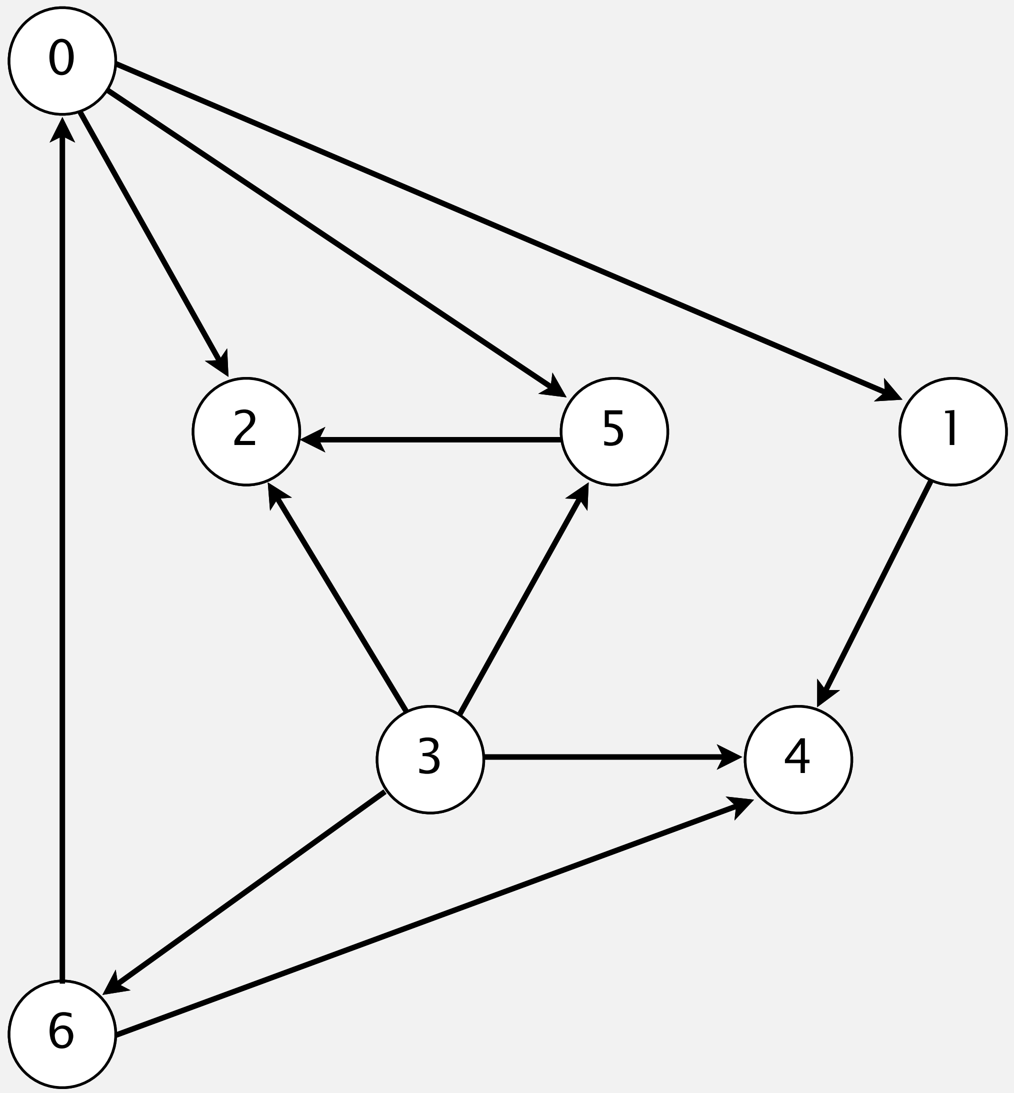


topological order

Solution. DFS. What else?

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



a directed acyclic graph

tinyDAG7.txt

7	
11	
0	5
0	2
0	1
3	6
3	5
3	4
5	2
6	4
6	0
3	2

Depth-first search orders

Observation. DFS visits each vertex exactly once. The order in which it does so can be important.

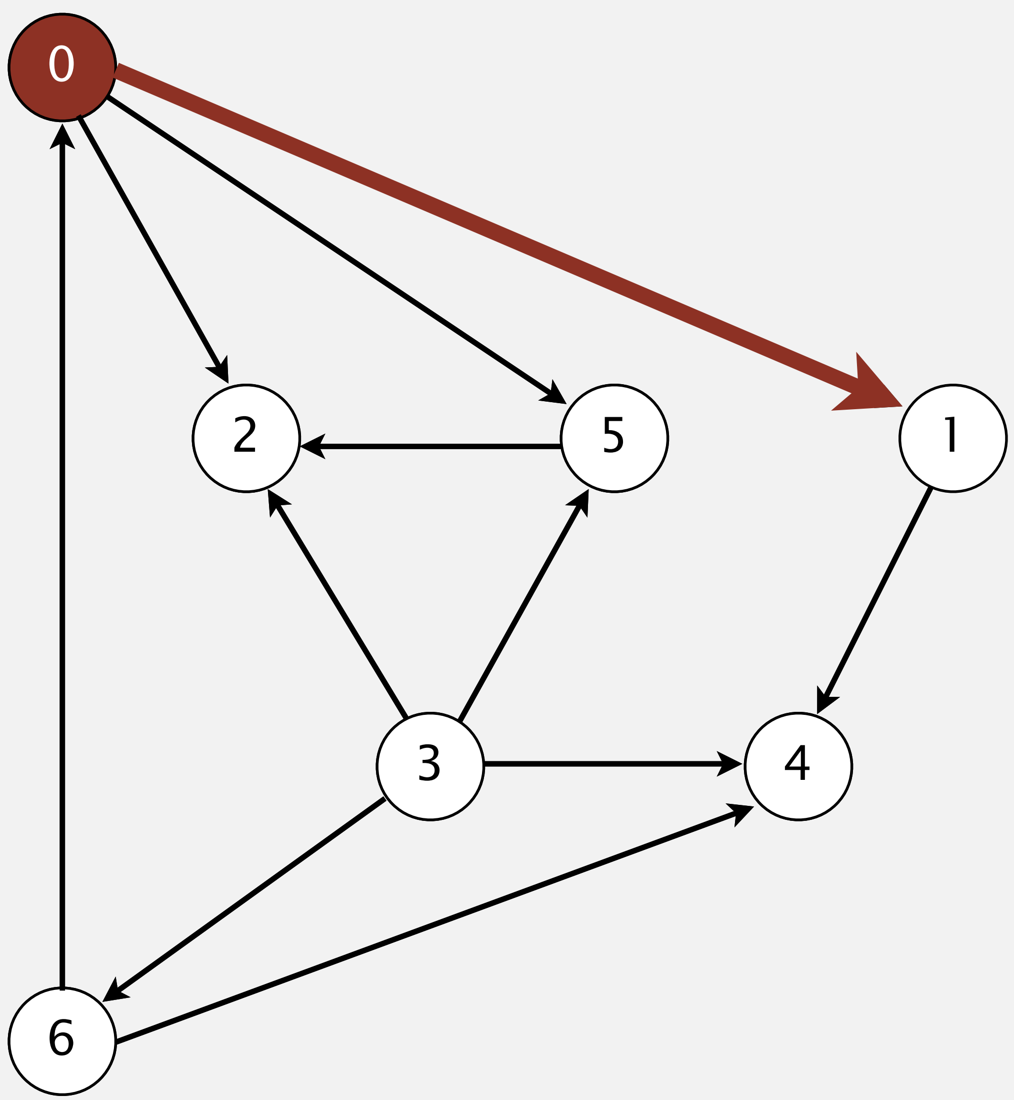
Orderings.

- Preorder: order in which `dfs()` is called.
- Postorder: order in which `dfs()` returns.
- Reverse postorder: reverse order in which `dfs()` returns.

```
private void dfs(Graph G, int v)
{
   marked[v] = true;
   preorder.enqueue(v);
   for (int w : G.adj(v))
      if (!marked[w]) dfs(G, w);
   postorder.enqueue(v);
   reversePostorder.push(v);
}
```

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



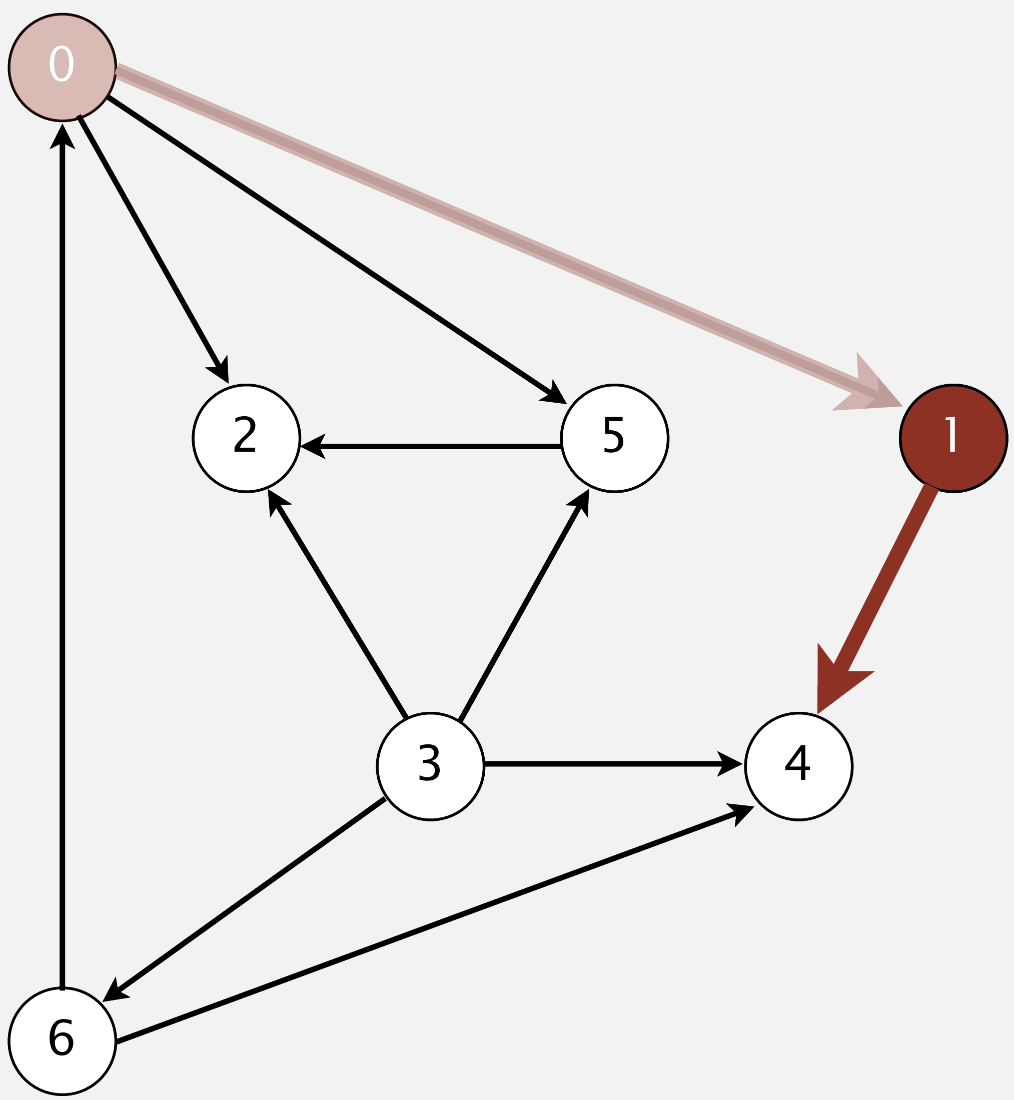
postorder

v	marked[]
0	T
1	F
2	F
3	F
4	F
5	F
6	F

visit 0: check 1, check 2, and check 5

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



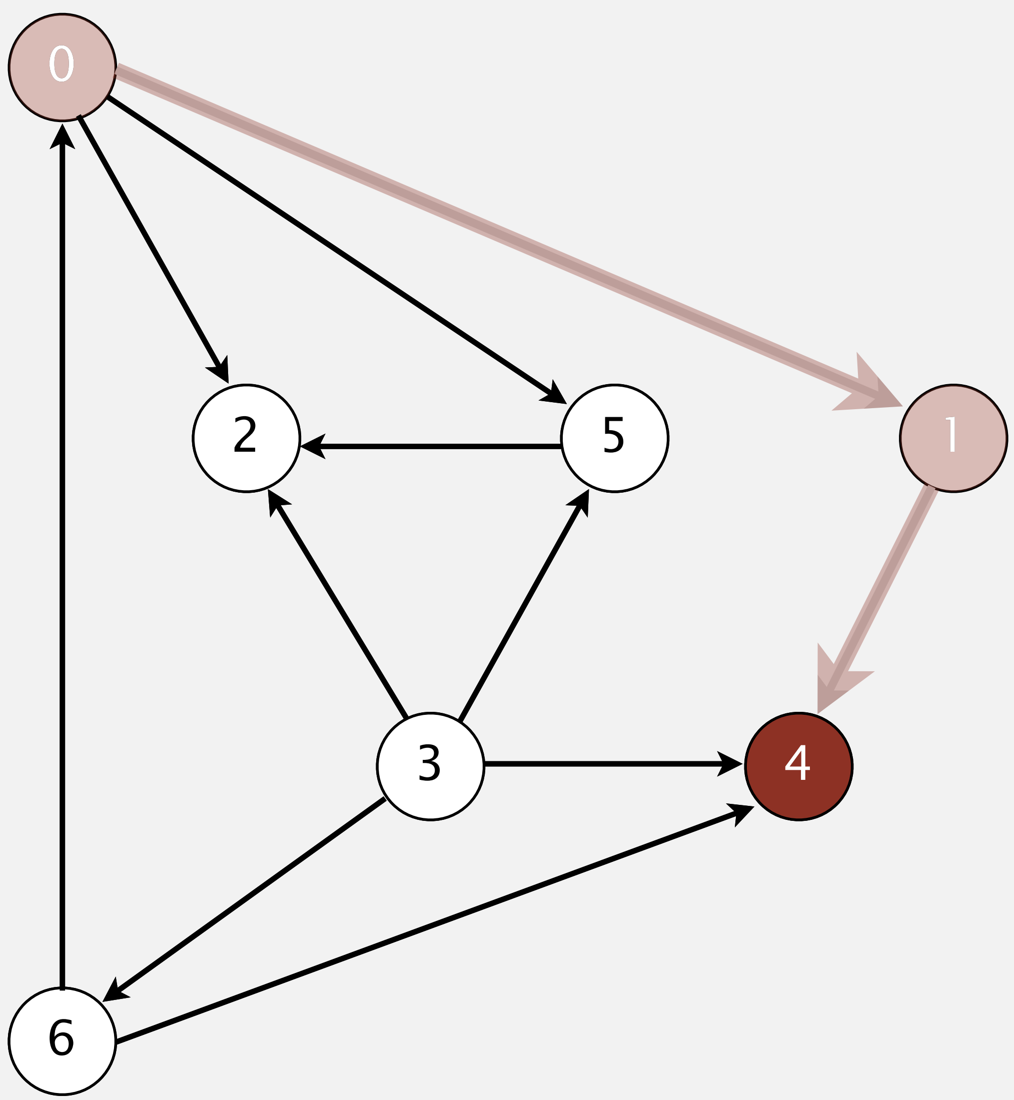
postorder

v	marked[]
0	T
1	T
2	F
3	F
4	F
5	F
6	F

visit 1: check 4

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



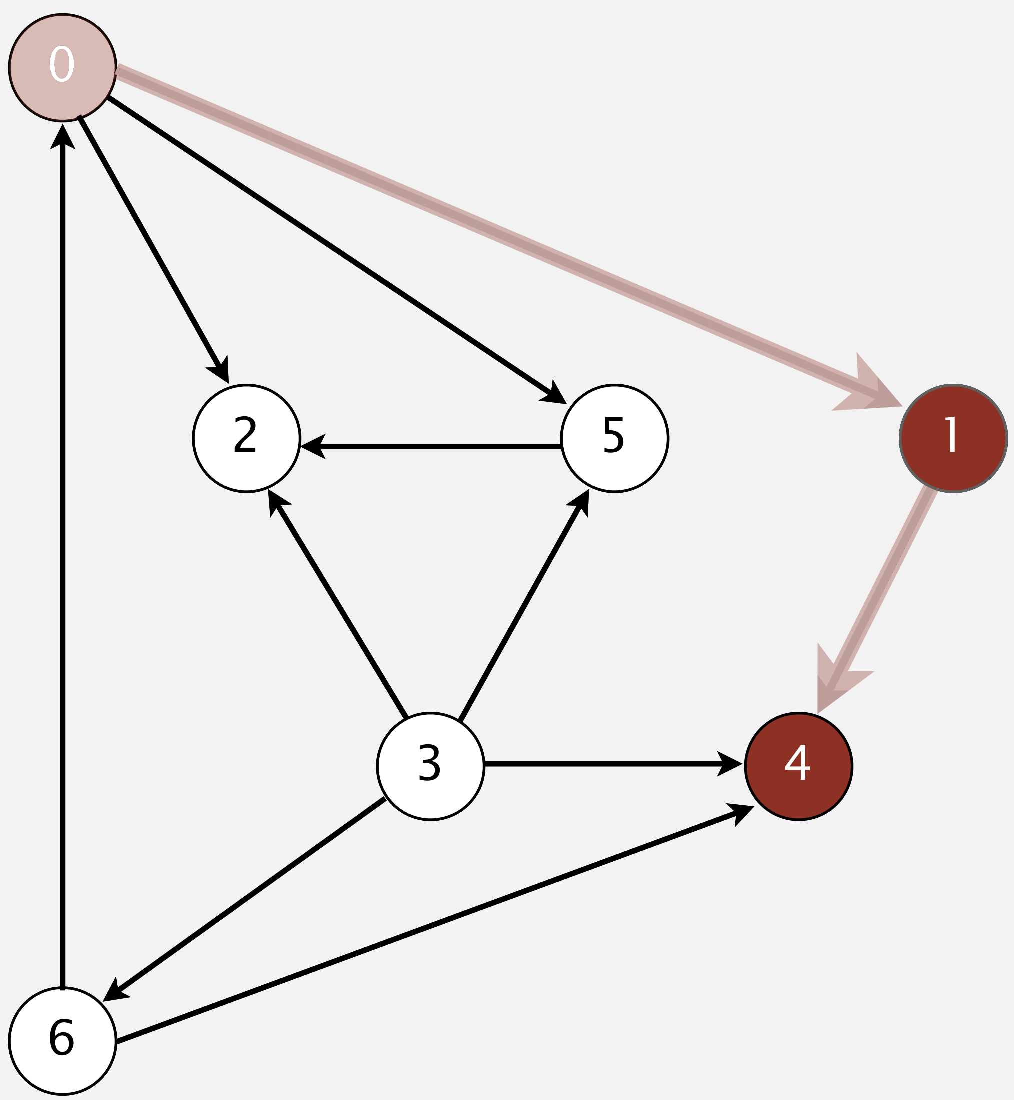
postorder

v	marked[]
0	T
1	T
2	F
3	F
4	T
5	F
6	F

visit 4

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



postorder

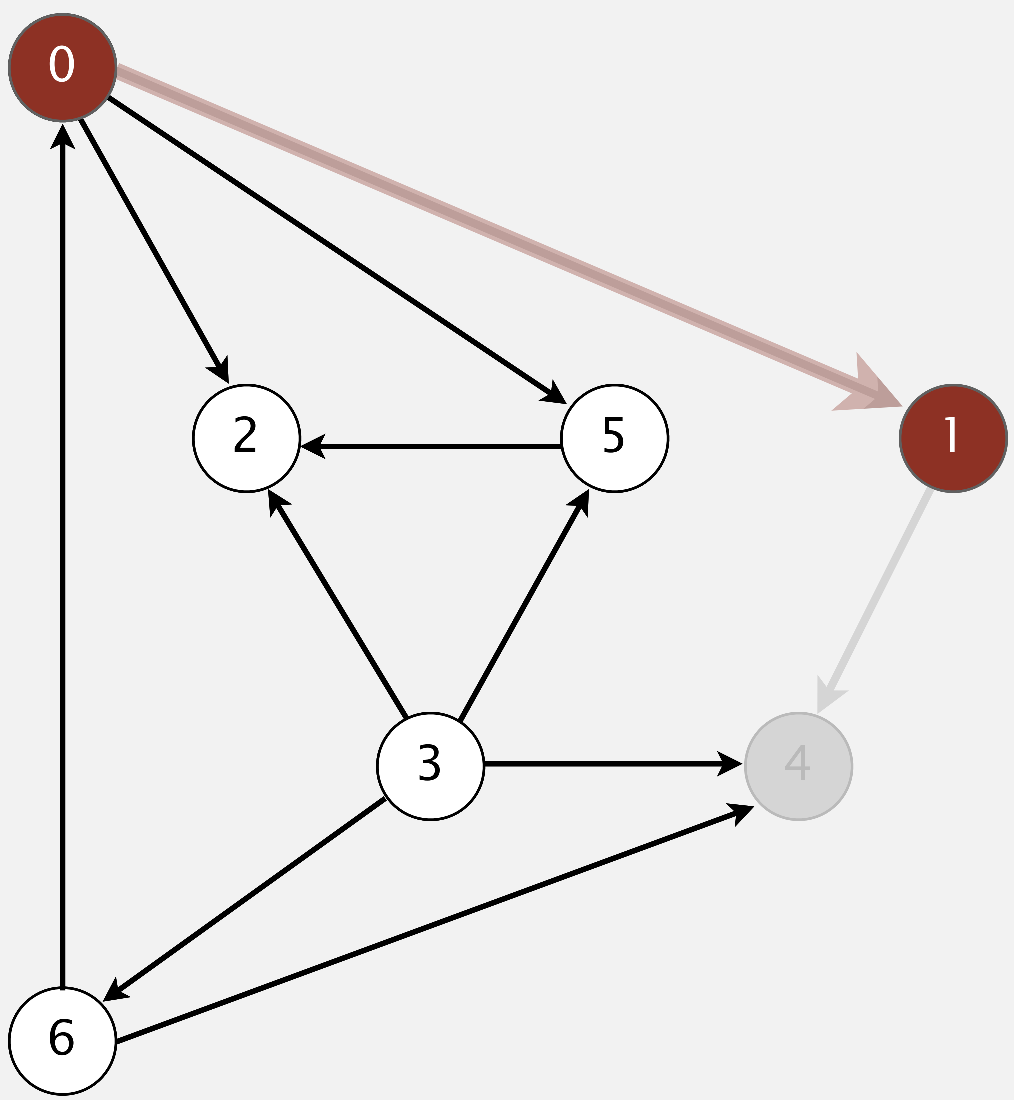
4

v	marked[]
0	T
1	T
2	F
3	F
4	T
5	F
6	F

4 done

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



postorder

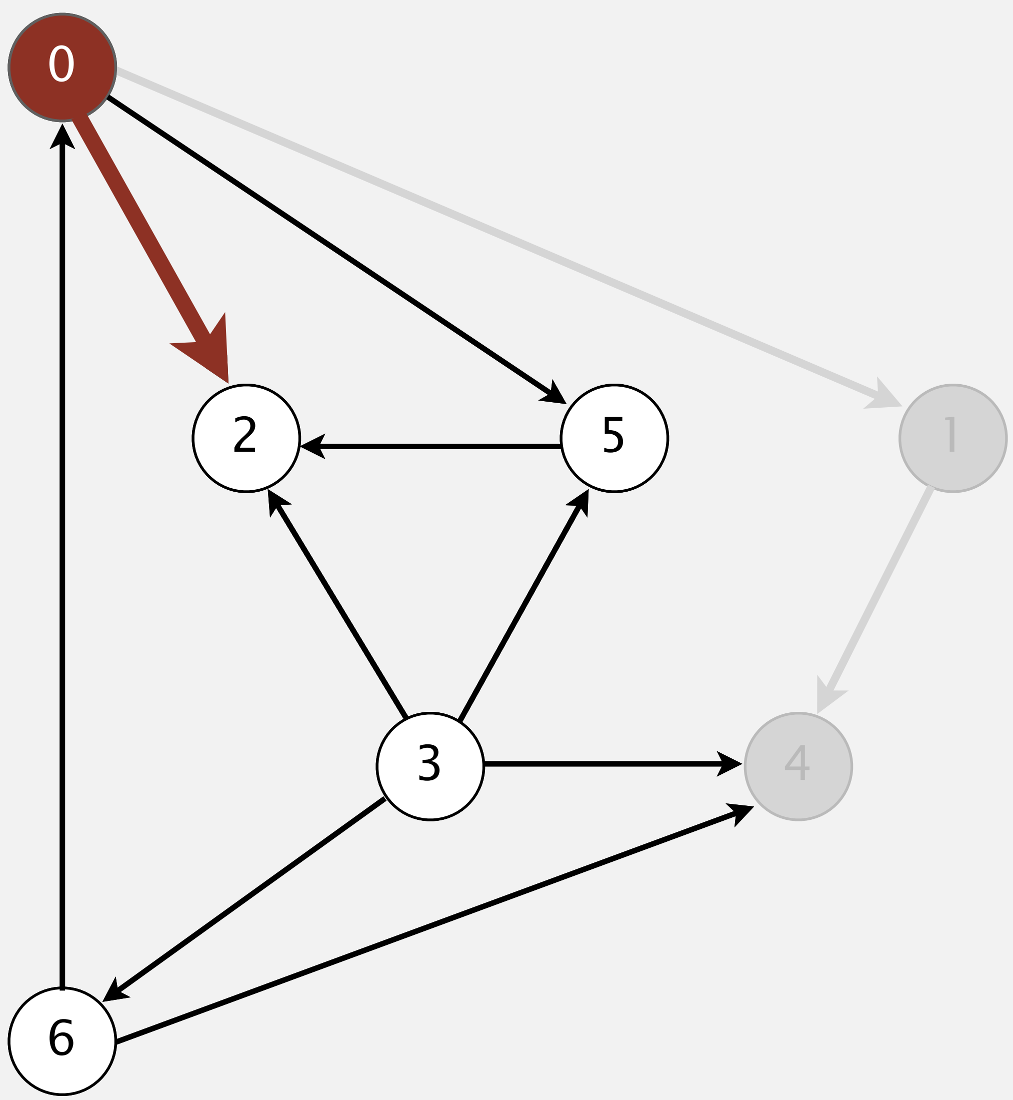
4 1

v	marked[]
0	T
1	T
2	F
3	F
4	T
5	F
6	F

1 done

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



postorder

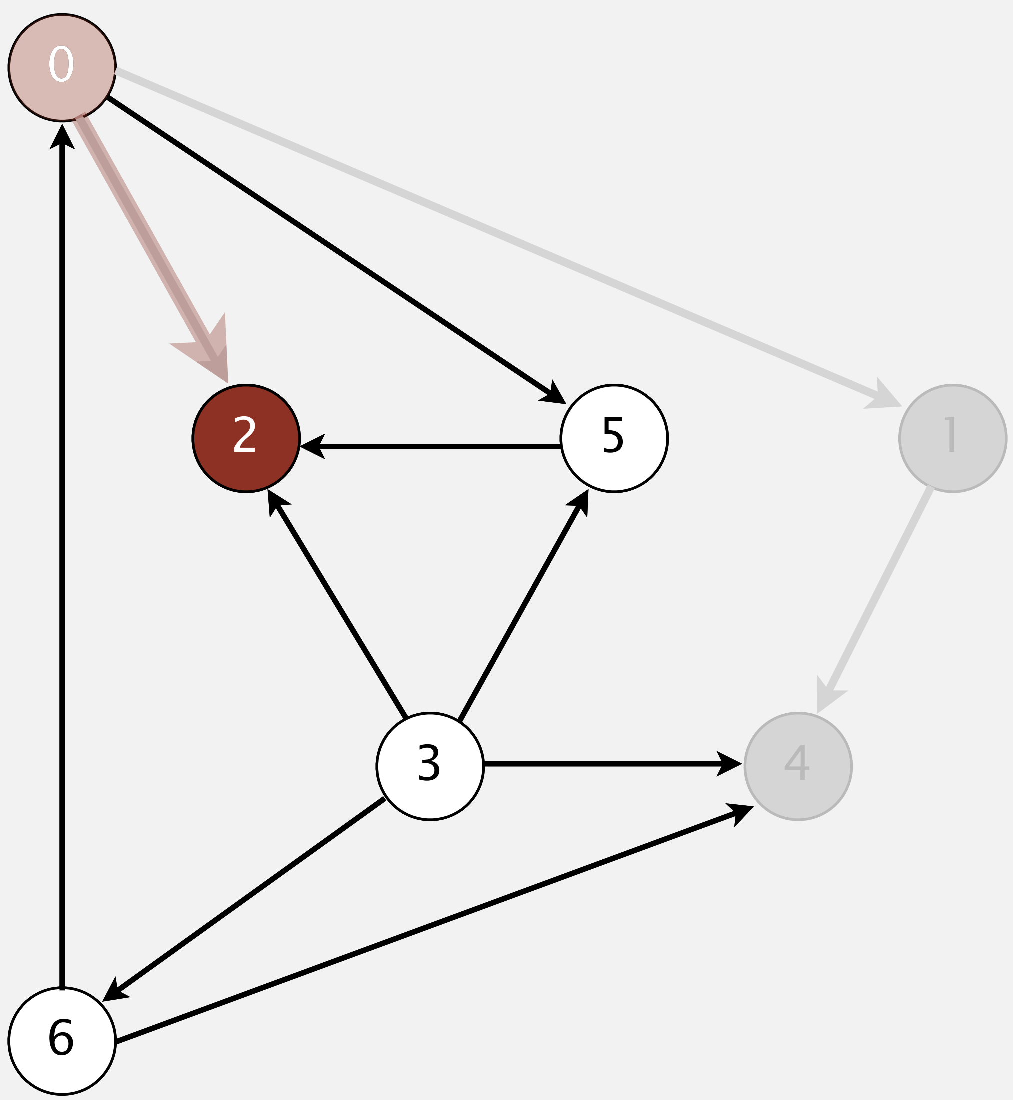
4 1

v	marked[]
0	T
1	T
2	F
3	F
4	T
5	F
6	F

visit 0: check 1, **check 2**, and check 5

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



postorder

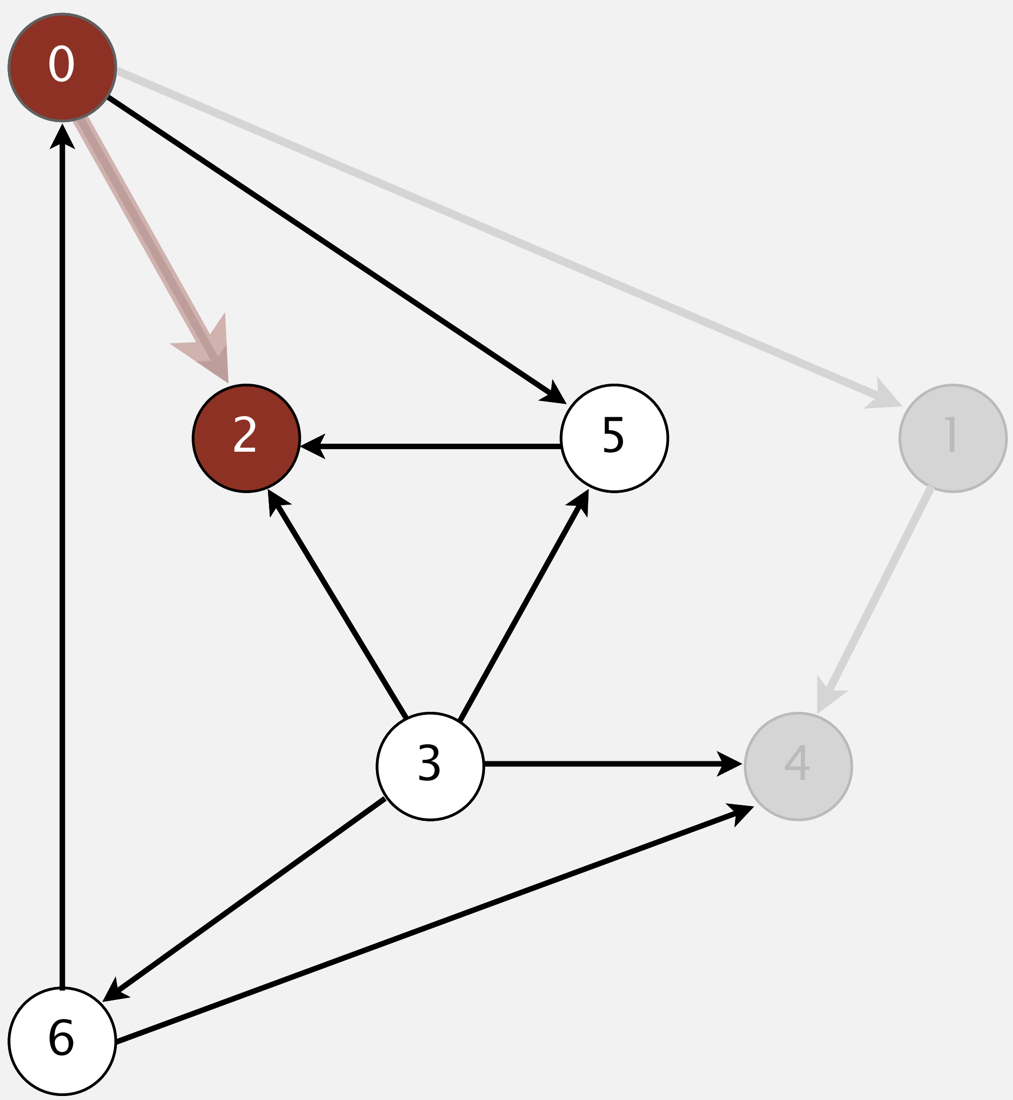
4 1

v	marked[]
0	T
1	T
2	T
3	F
4	T
5	F
6	F

visit 2

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



postorder

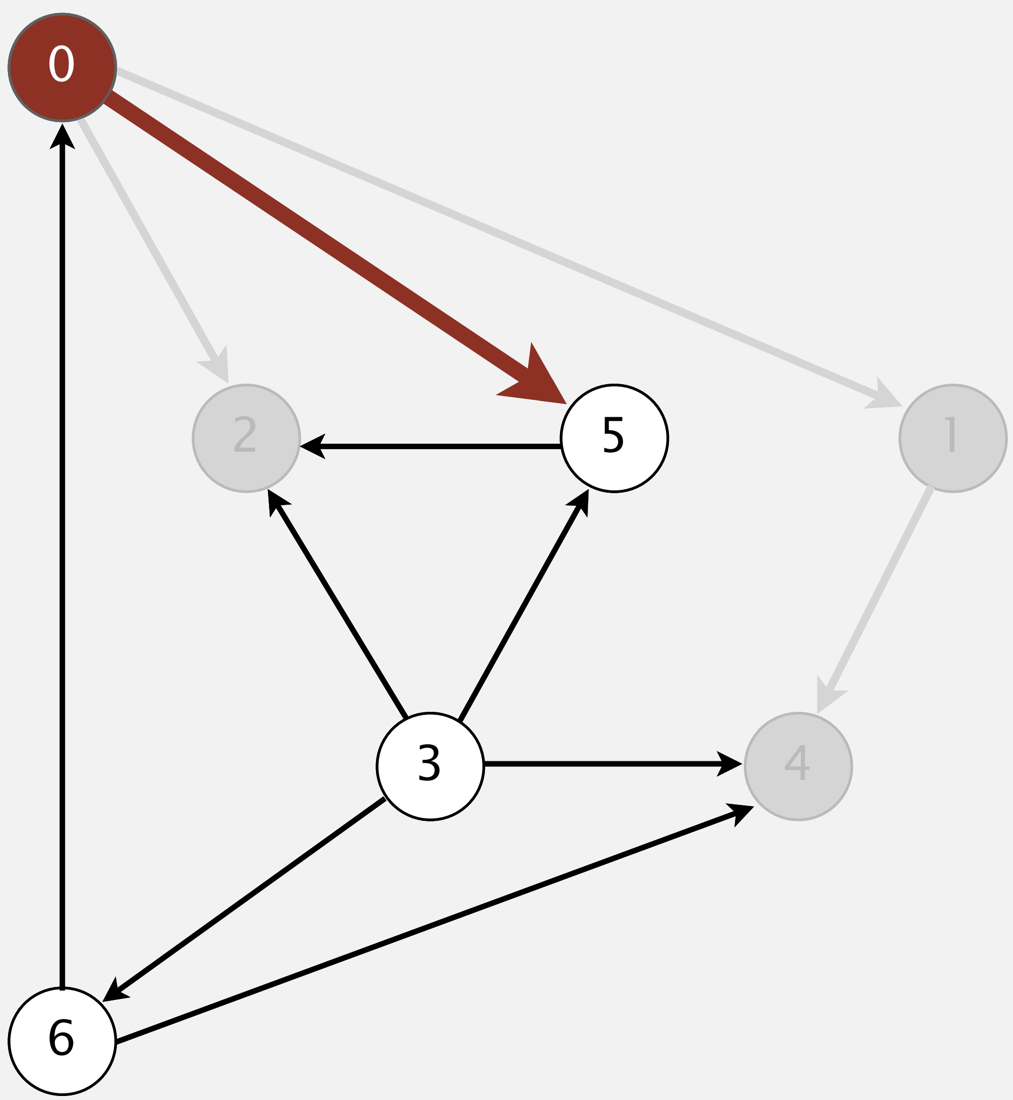
4 1 2

v	marked[]
0	T
1	T
2	T
3	F
4	T
5	F
6	F

2 done

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



postorder

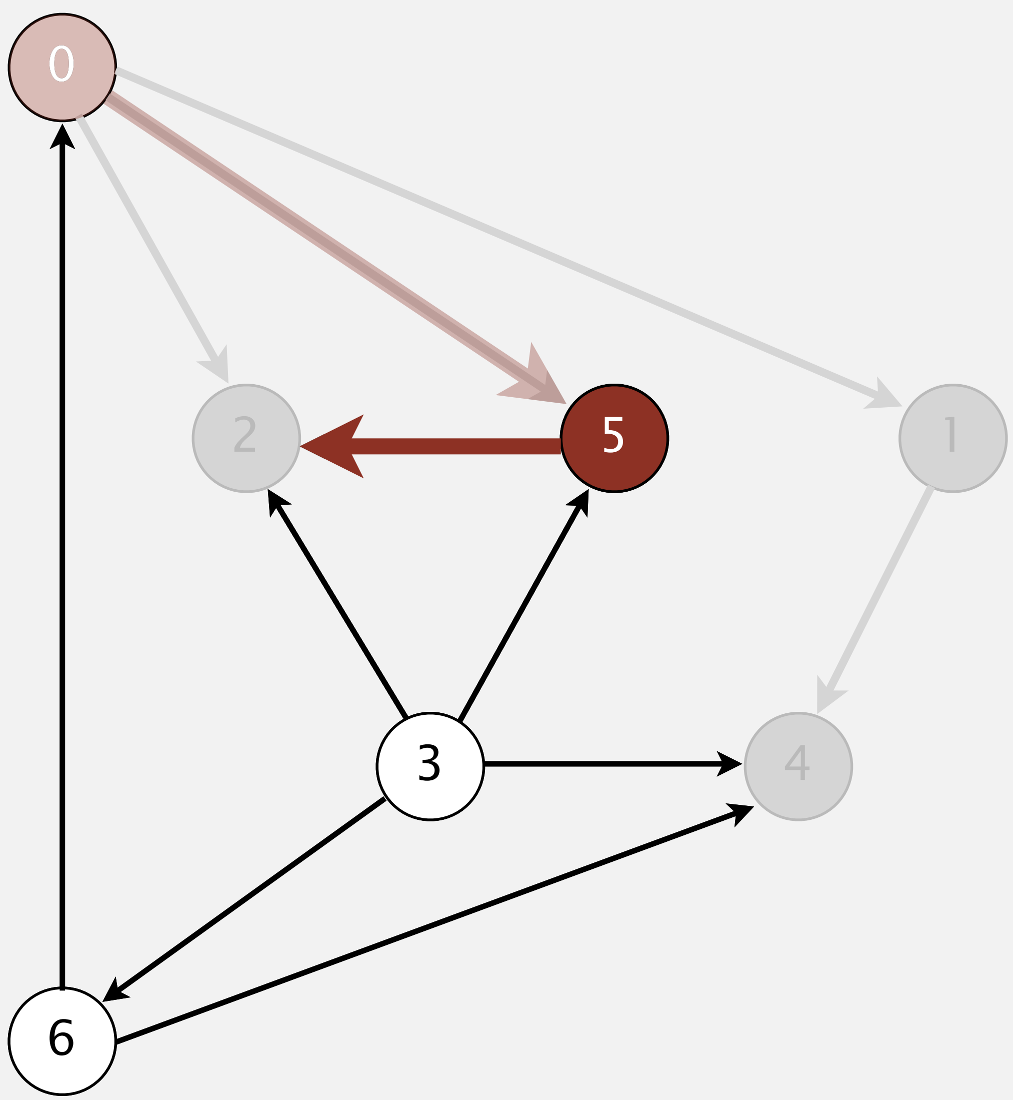
4 1 2

v	marked[]
0	T
1	T
2	T
3	F
4	T
5	F
6	F

visit 0: check 1, check 2, and **check 5**

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



postorder

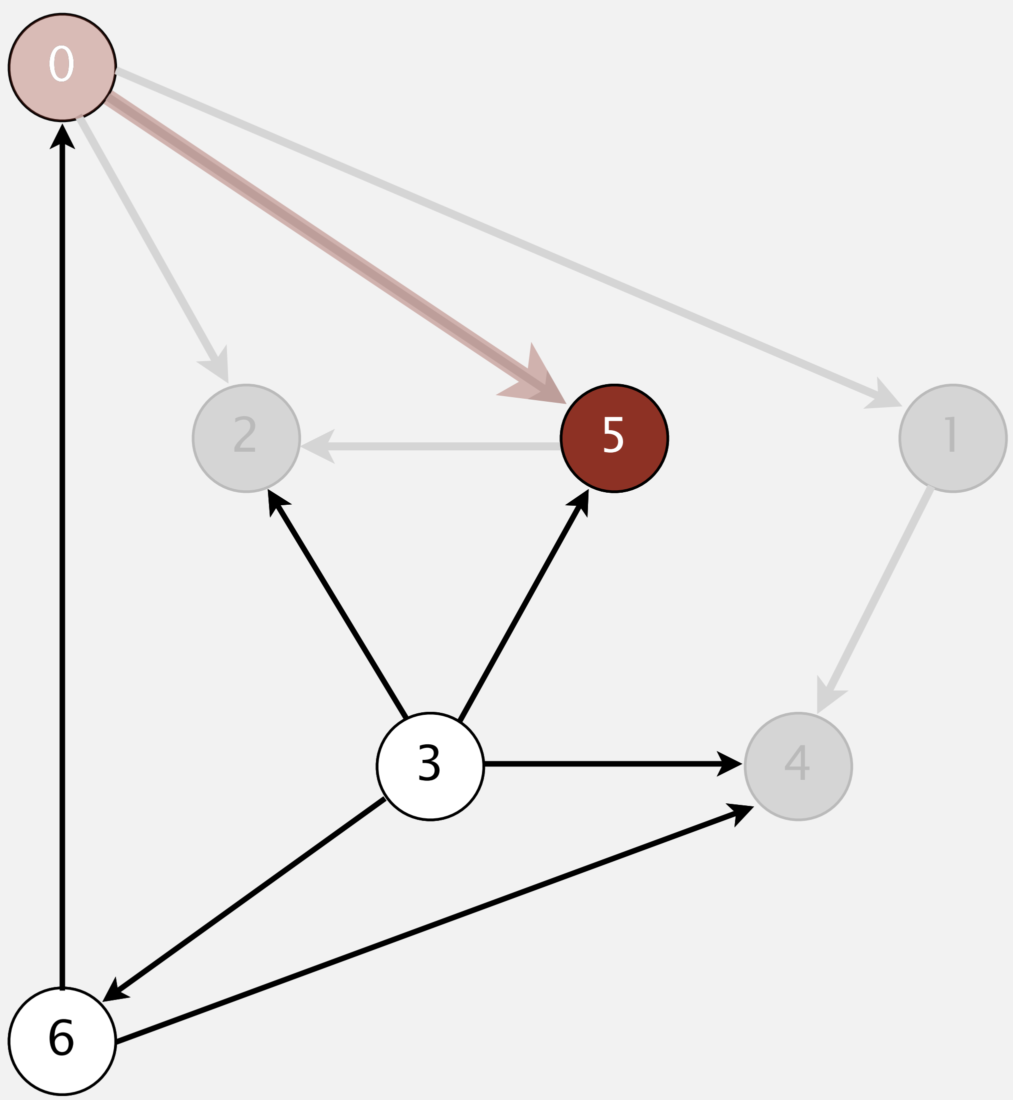
4 1 2

v	marked[]
0	T
1	T
2	T
3	F
4	T
5	T
6	F

visit 5: check 2

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



postorder

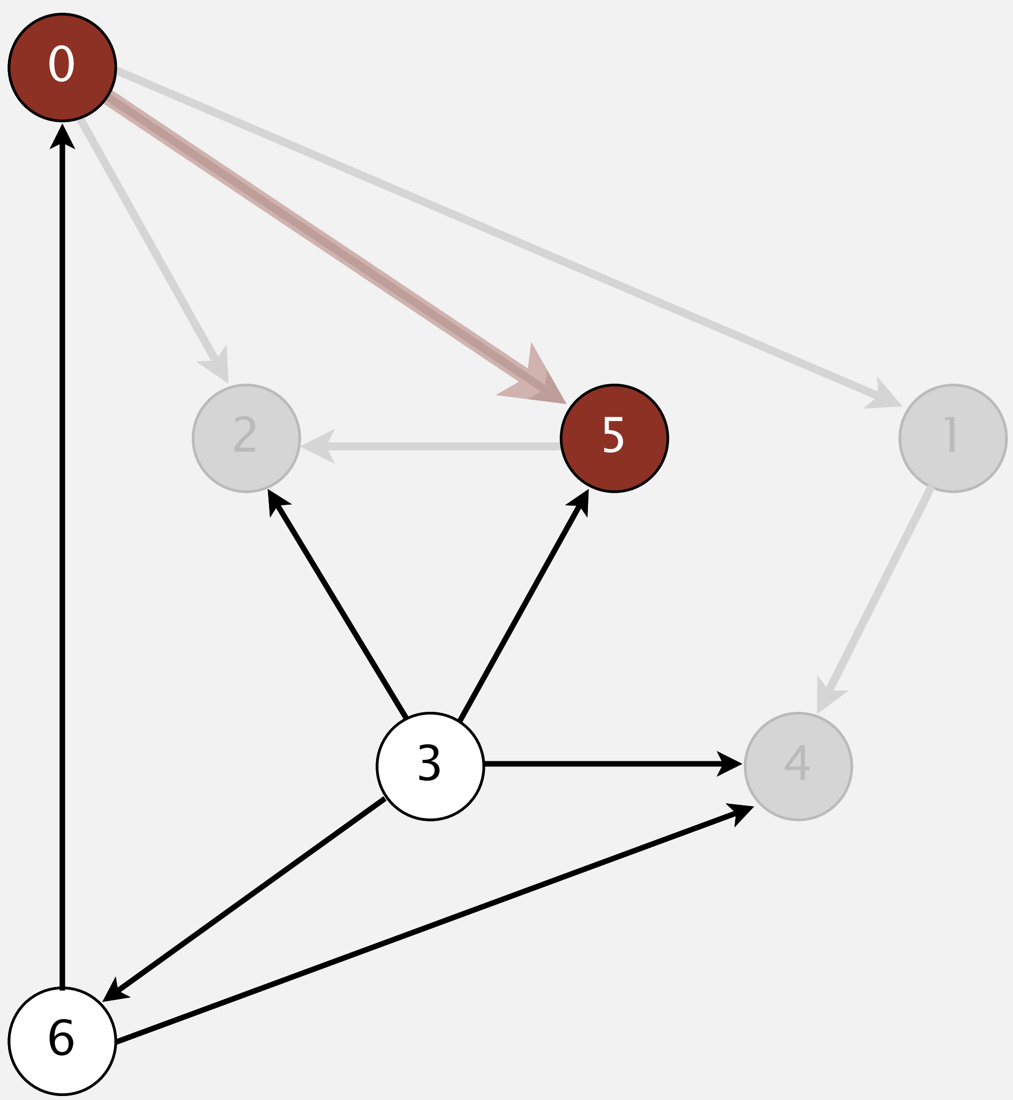
4 1 2

v	marked[]
0	T
1	T
2	T
3	F
4	T
5	T
6	F

visit 5

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



postorder

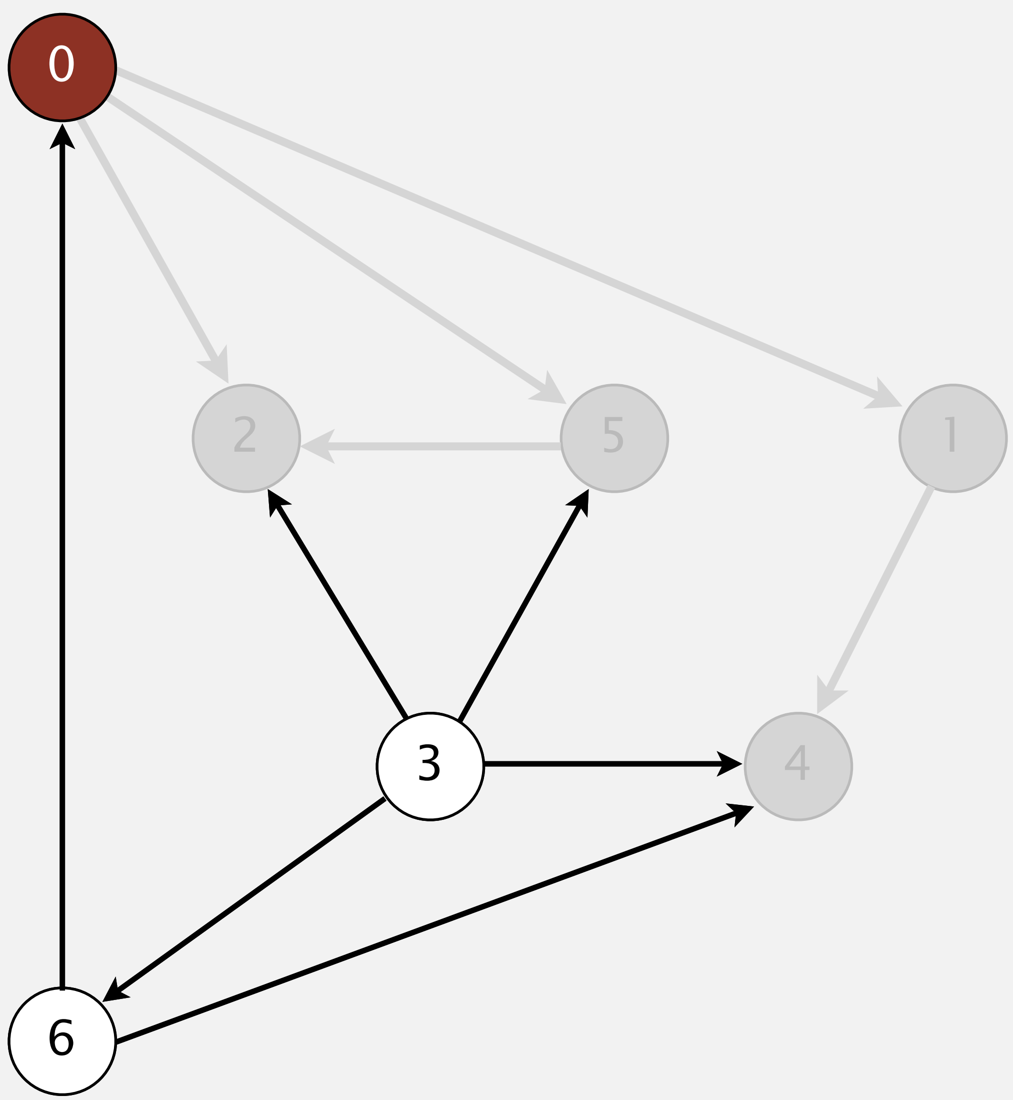
4 1 2 5

v	marked[]
0	T
1	T
2	T
3	F
4	T
5	T
6	F

5 done

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



postorder

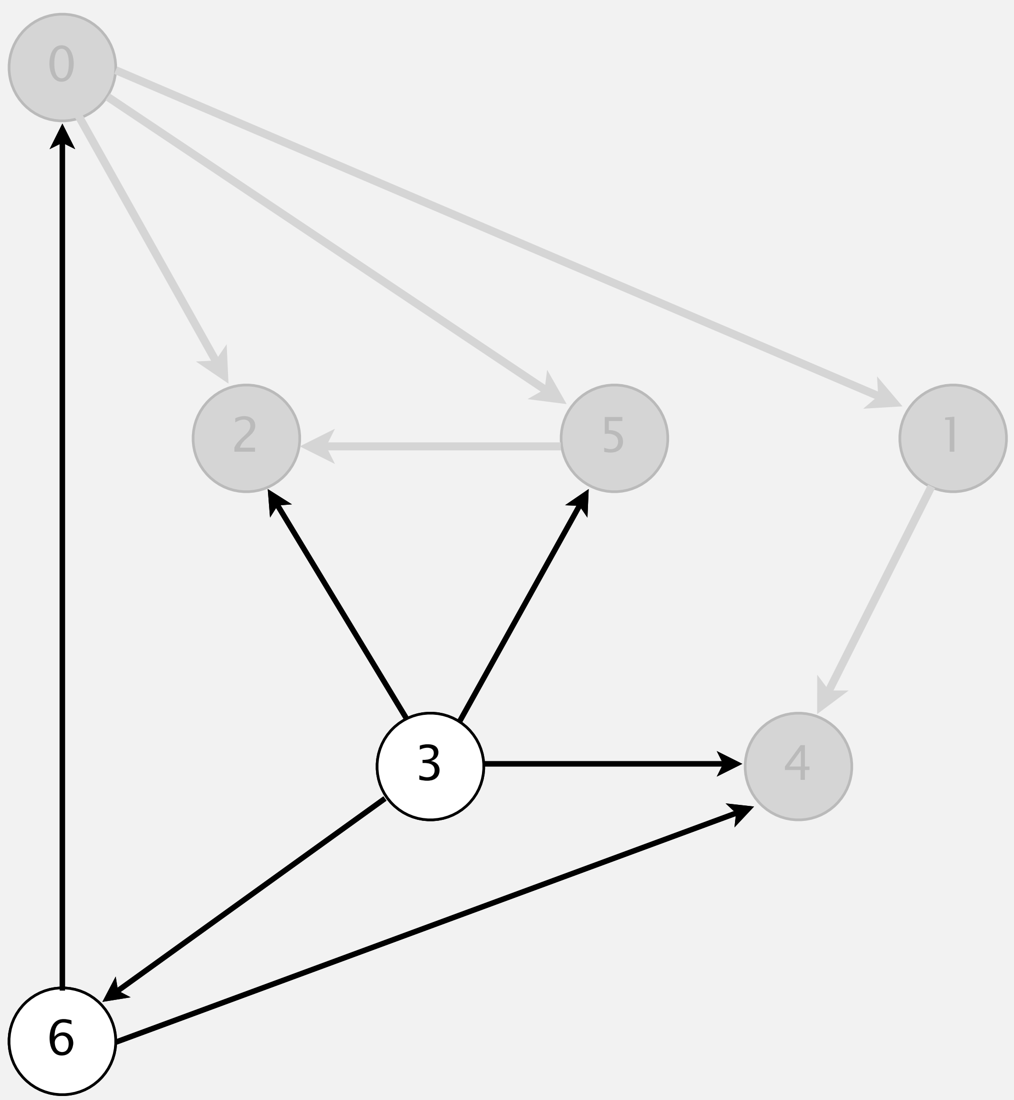
4 1 2 5 0

v	marked[]
0	T
1	T
2	T
3	F
4	T
5	T
6	F

0 done

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



postorder

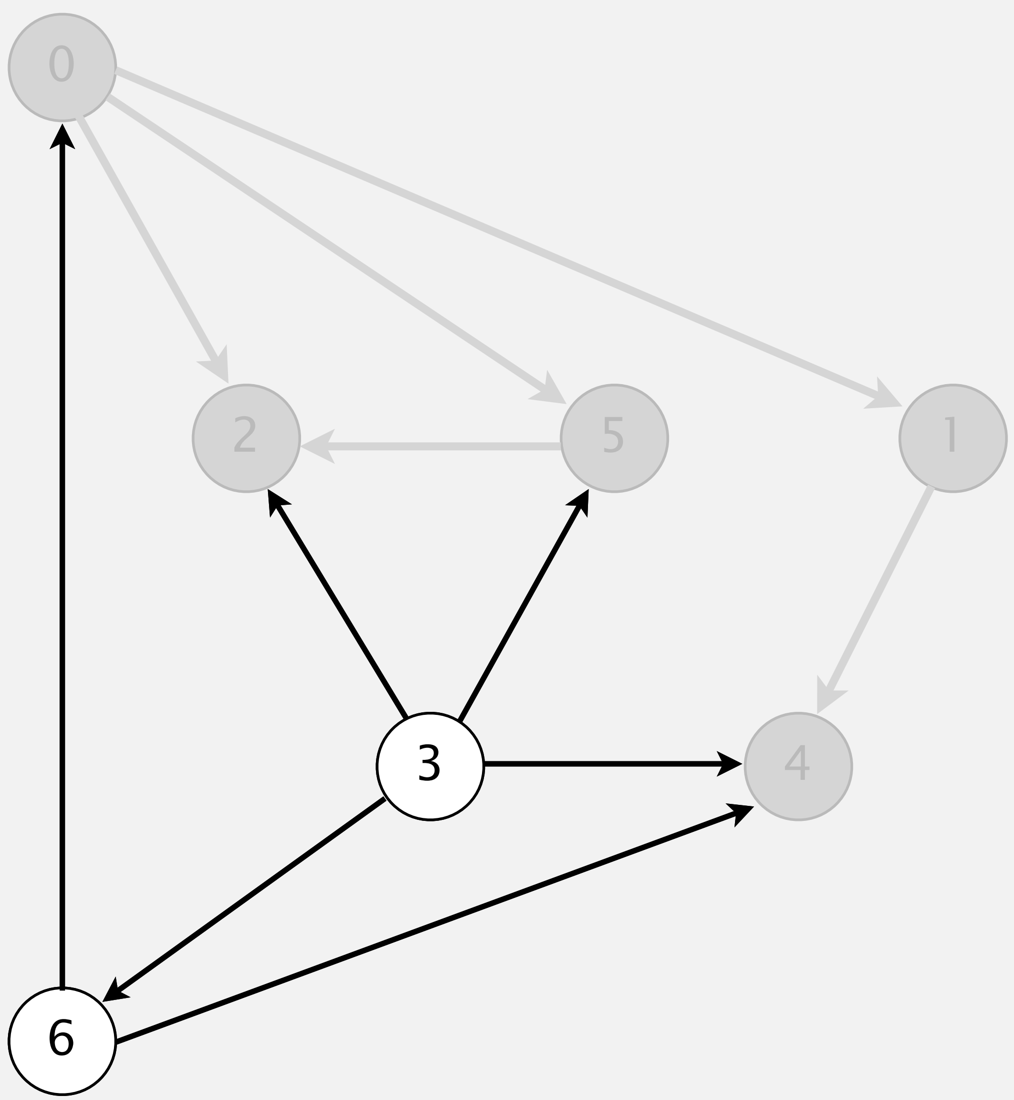
4 1 2 5 0

v	marked[]
0	T
1	T
2	T
3	F
4	T
5	T
6	F

check 1

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



postorder

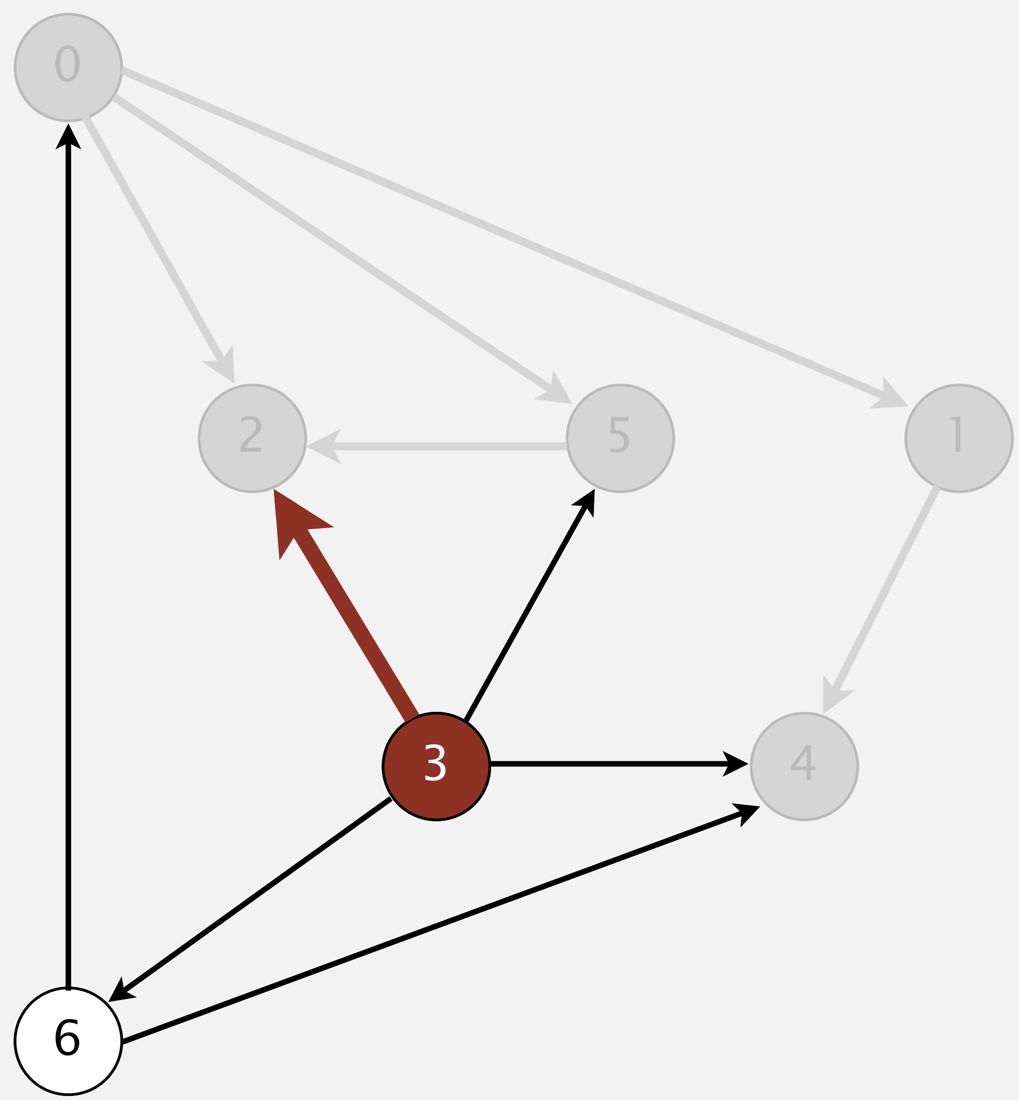
4 1 2 5 0

v	marked[]
0	T
1	T
2	T
3	F
4	T
5	T
6	F

check 2

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



postorder

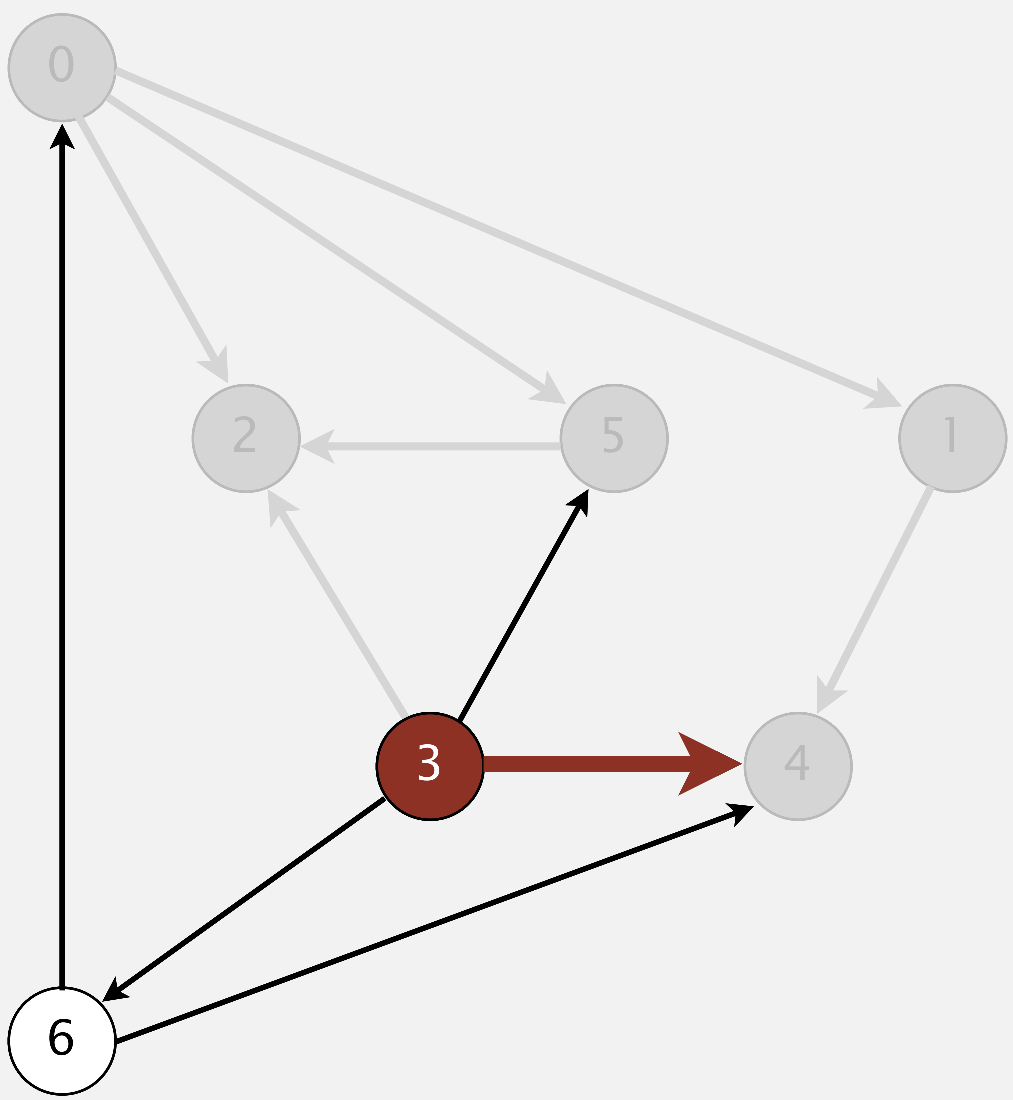
4 1 2 5 0

v	marked[]
0	T
1	T
2	T
3	T
4	T
5	T
6	F

visit 3: check 2, check 4, check 5, and check 6

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



postorder

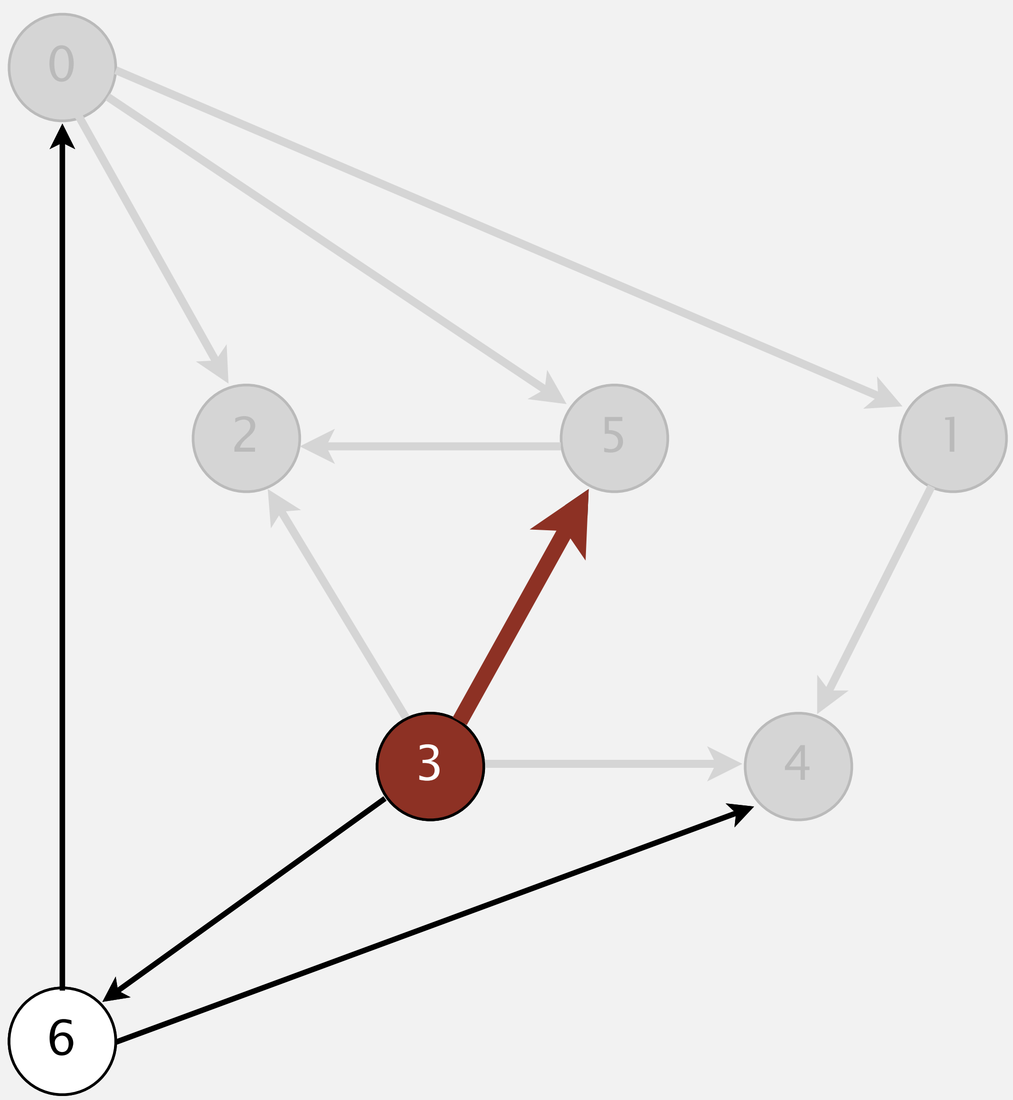
4 1 2 5 0

v	marked[]
0	T
1	T
2	T
3	T
4	T
5	T
6	F

visit 3: check 2, **check 4**, check 5, and check 6

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



postorder

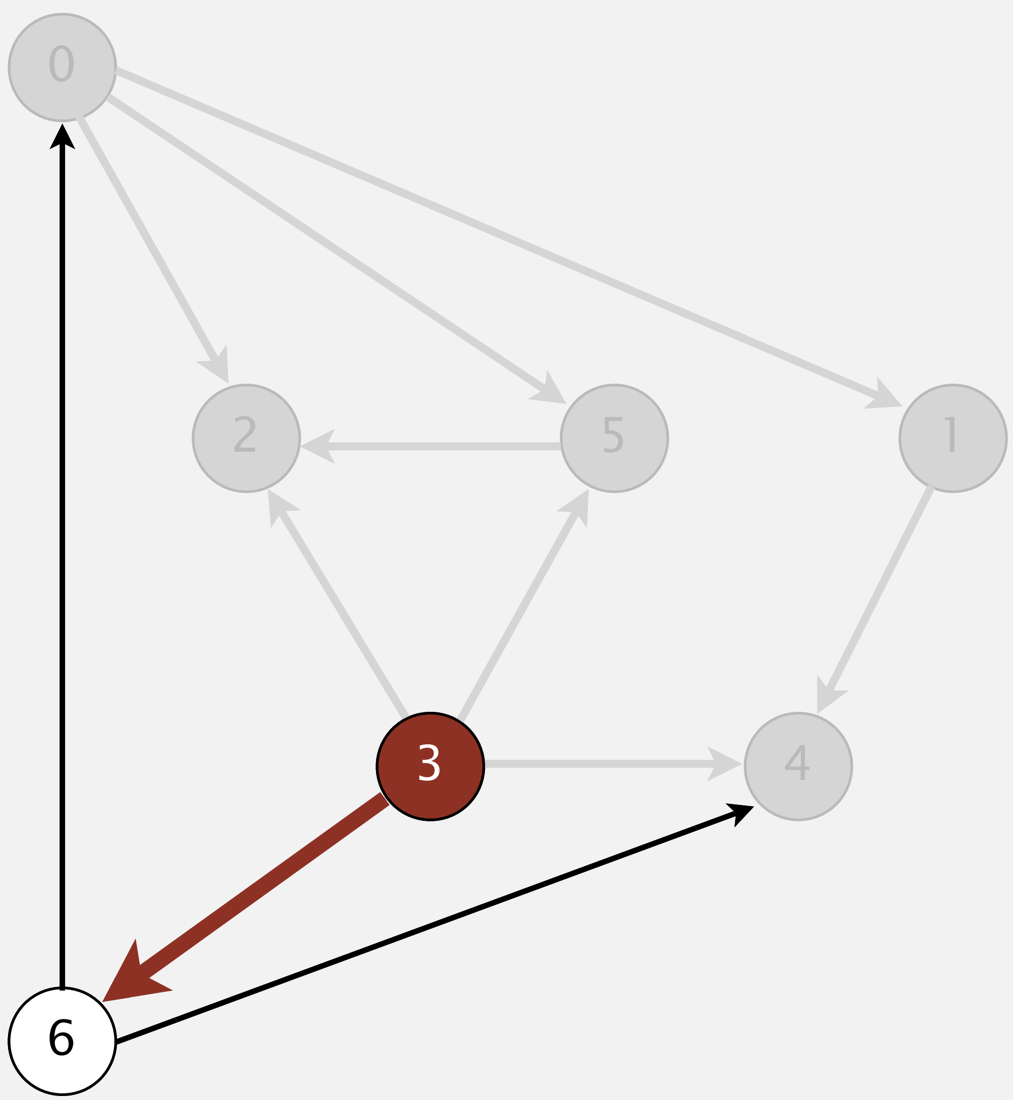
4 1 2 5 0

v	marked[]
0	T
1	T
2	T
3	T
4	T
5	T
6	F

visit 3: check 2, check 4, **check 5**, and check 6

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



postorder

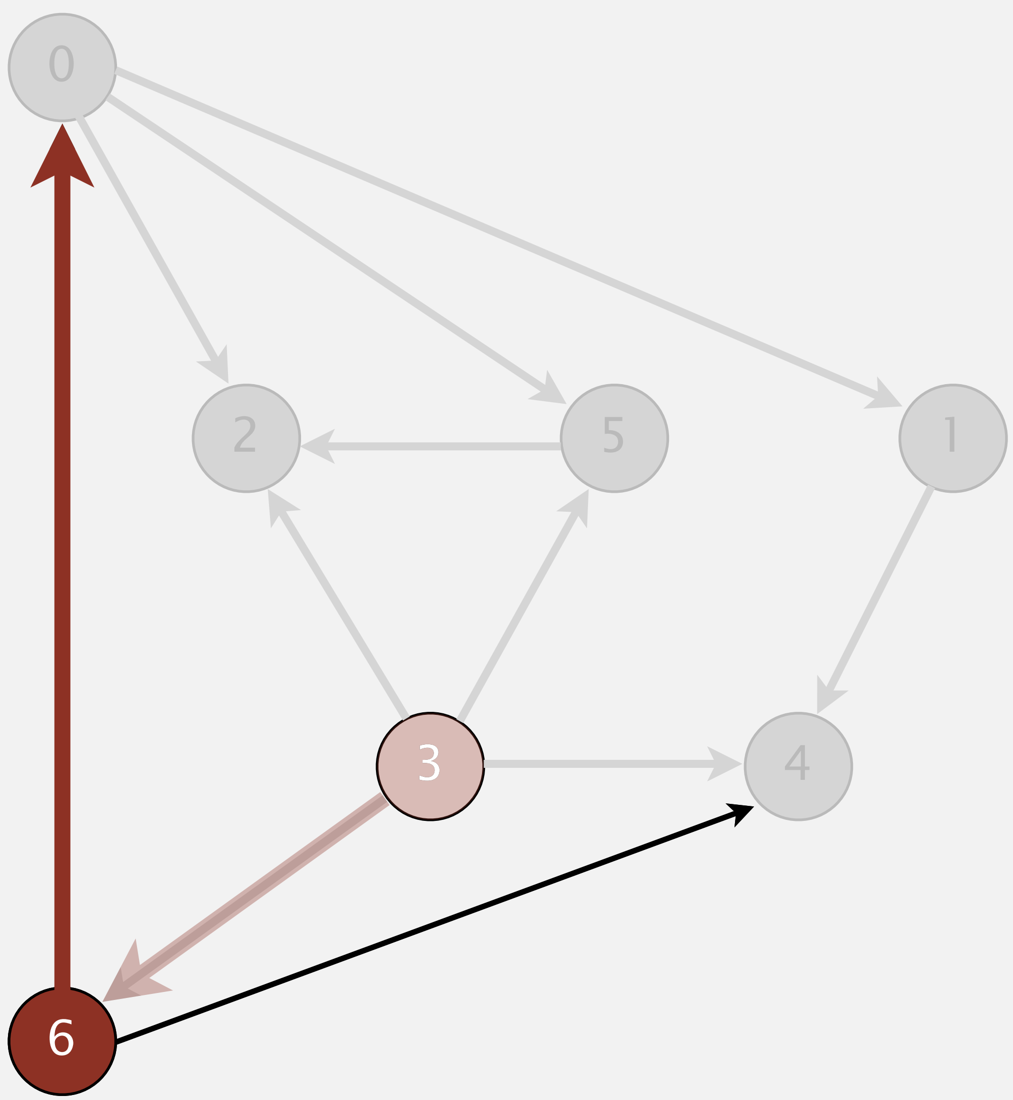
4 1 2 5 0

v	marked[]
0	T
1	T
2	T
3	T
4	T
5	T
6	F

visit 3: check 2, check 4, check 5, and **check 6**

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



postorder

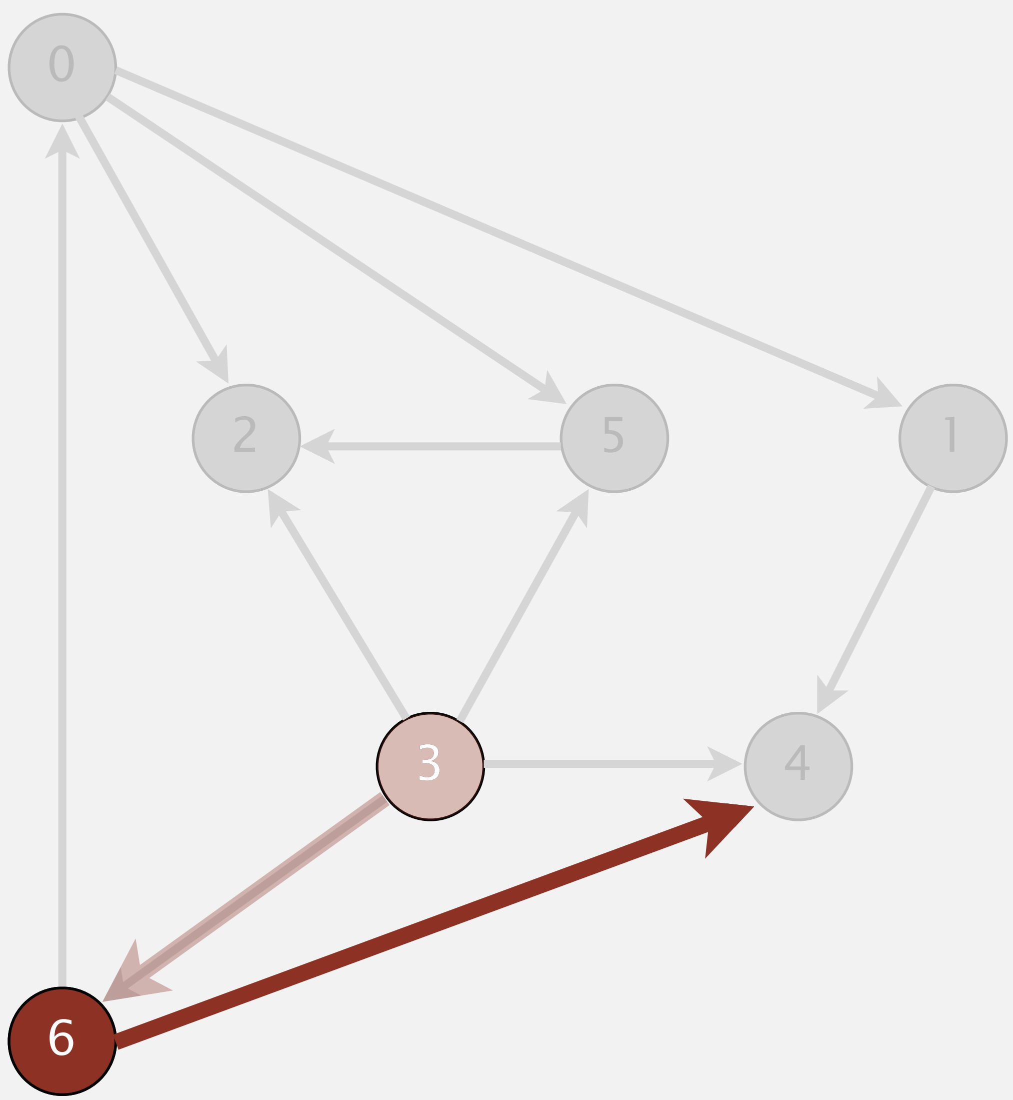
4 1 2 5 0

v	marked[]
0	T
1	T
2	T
3	T
4	T
5	T
6	T

visit 6: check 0 and check 4

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



postorder

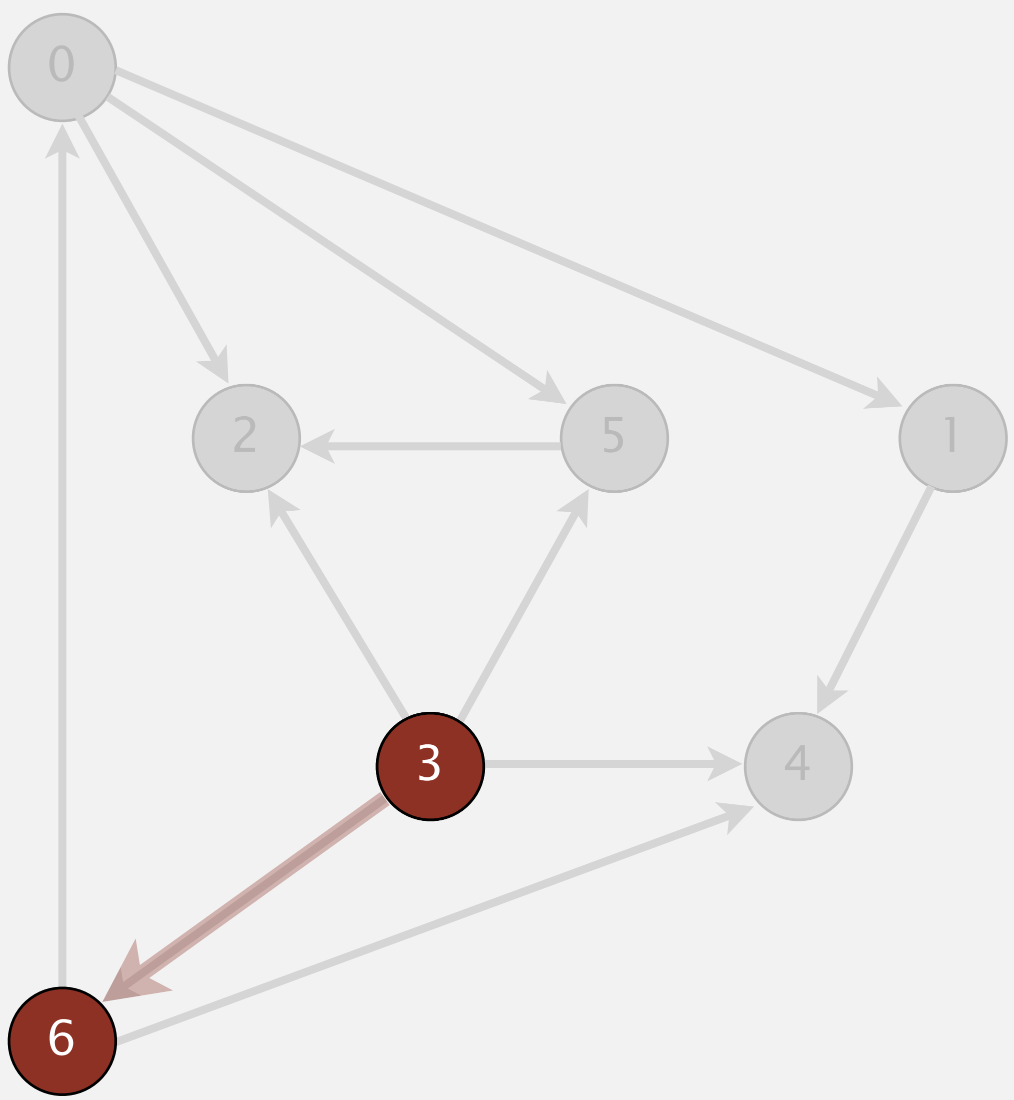
4 1 2 5 0

v	marked[]
0	T
1	T
2	T
3	T
4	T
5	T
6	T

visit 6: check 0 and **check 4**

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



6 done

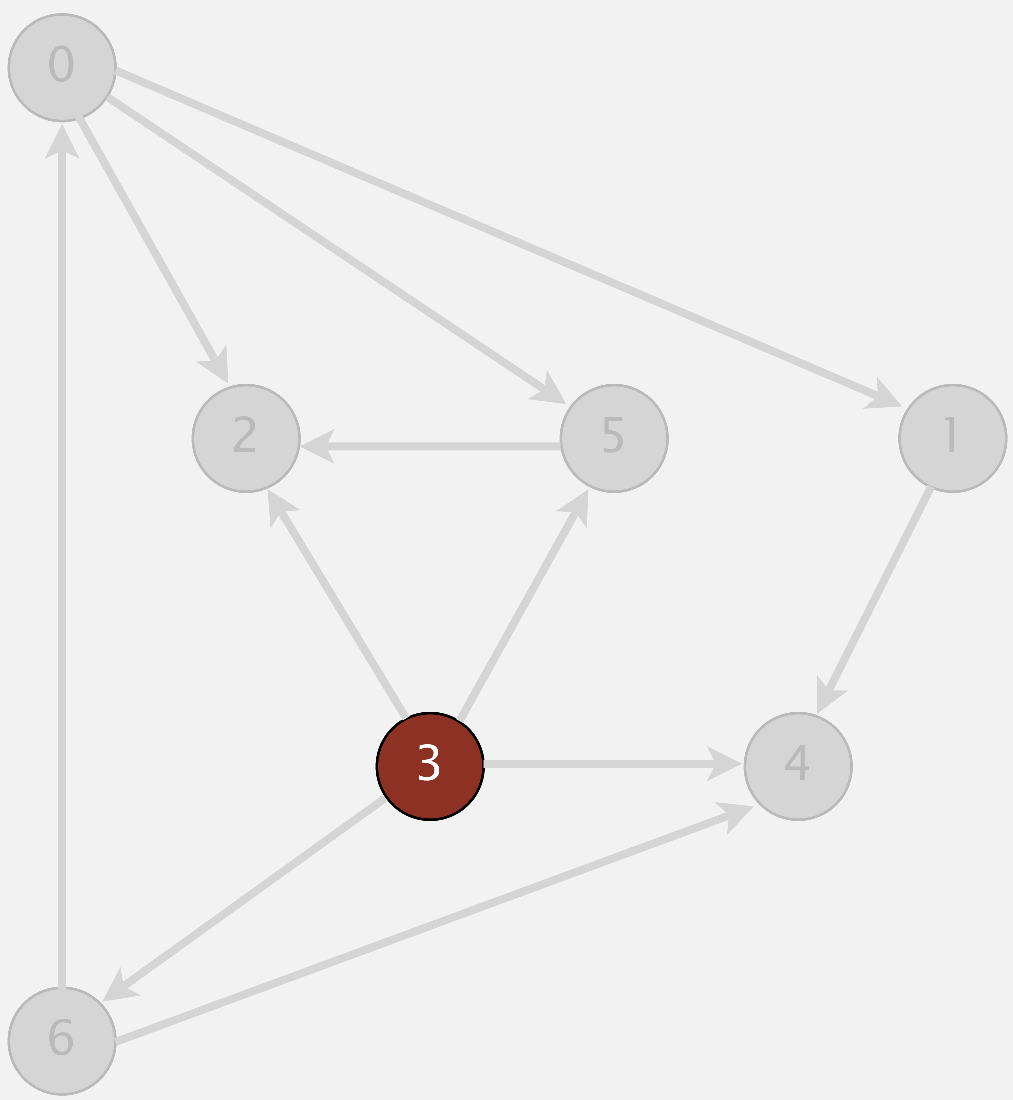
postorder

4 1 2 5 0 6

v	marked[]
0	T
1	T
2	T
3	T
4	T
5	T
6	T

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



postorder

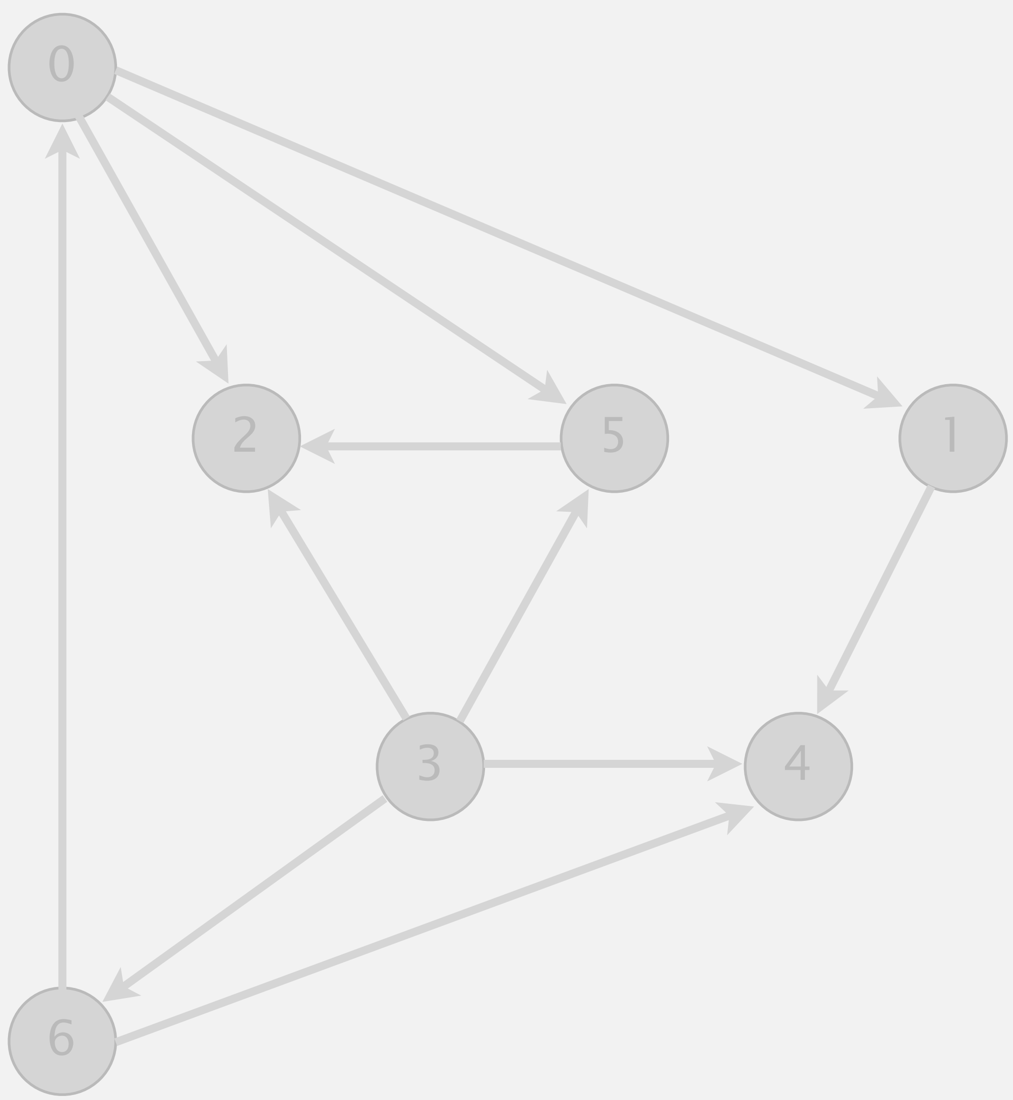
4 1 2 5 0 6 3

v	marked[]
0	T
1	T
2	T
3	T
4	T
5	T
6	T

3 done

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



postorder

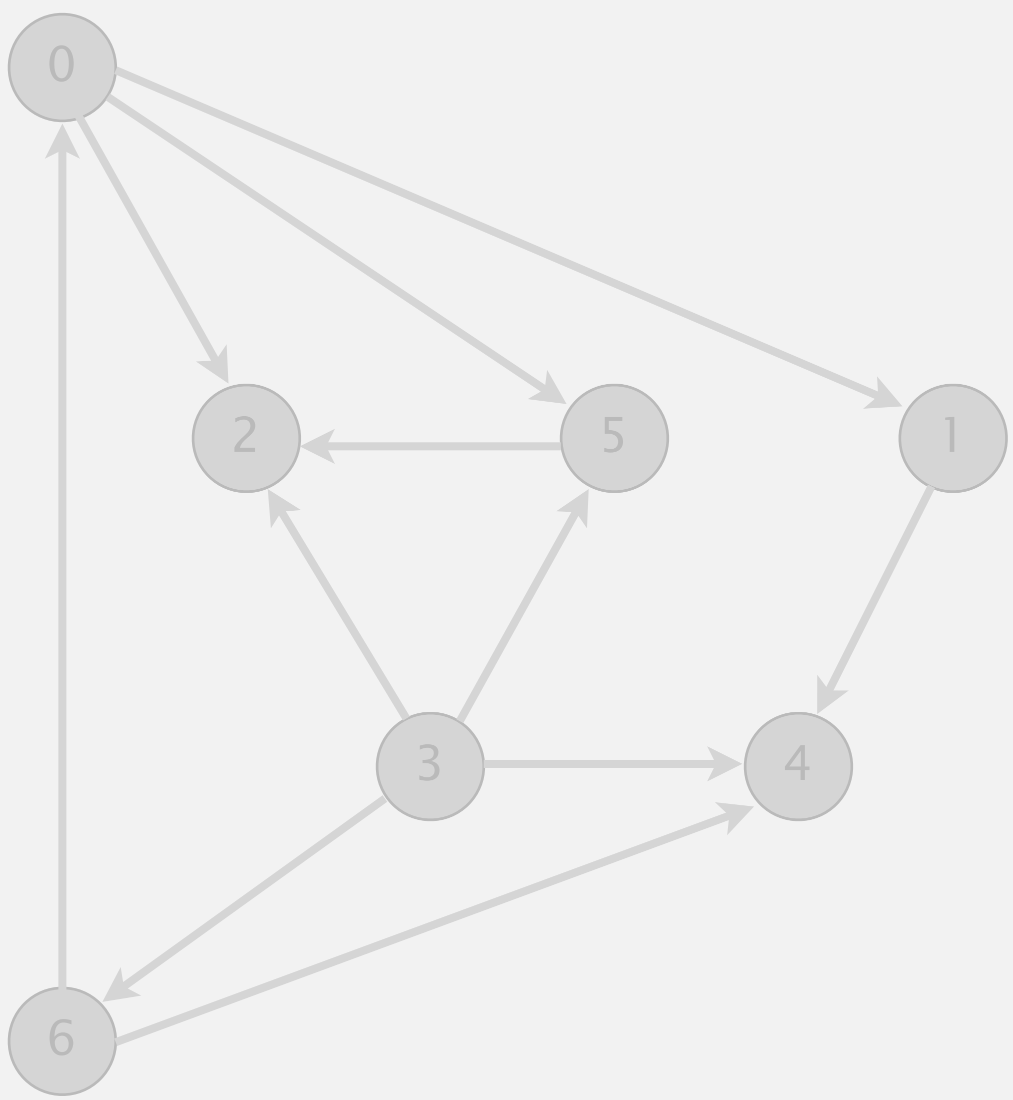
4 1 2 5 0 6 3

v	marked[]
0	T
1	T
2	T
3	T
4	T
5	T
6	T

check 4

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



postorder

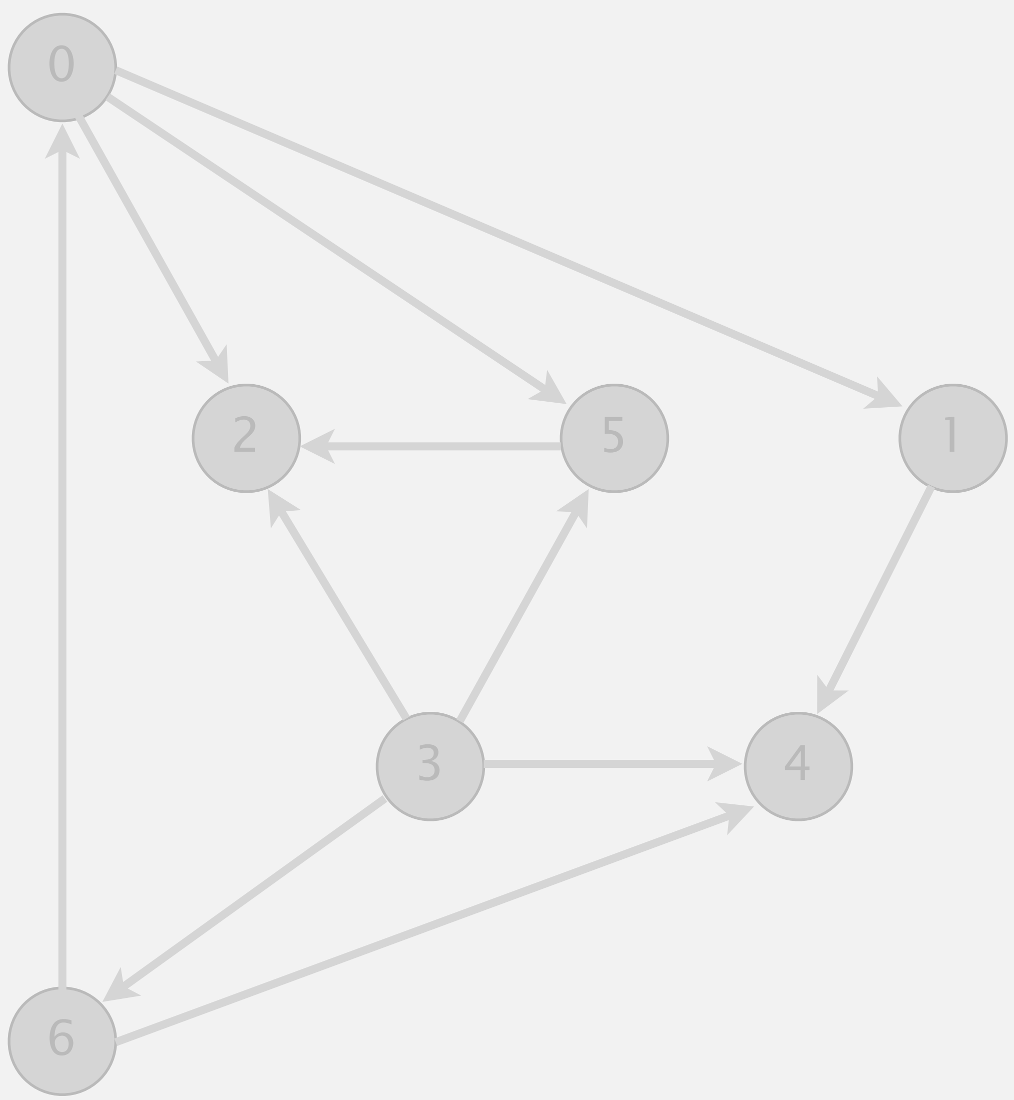
4 1 2 5 0 6 3

v	marked[]
0	T
1	T
2	T
3	T
4	T
5	T
6	T

check 5

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



postorder

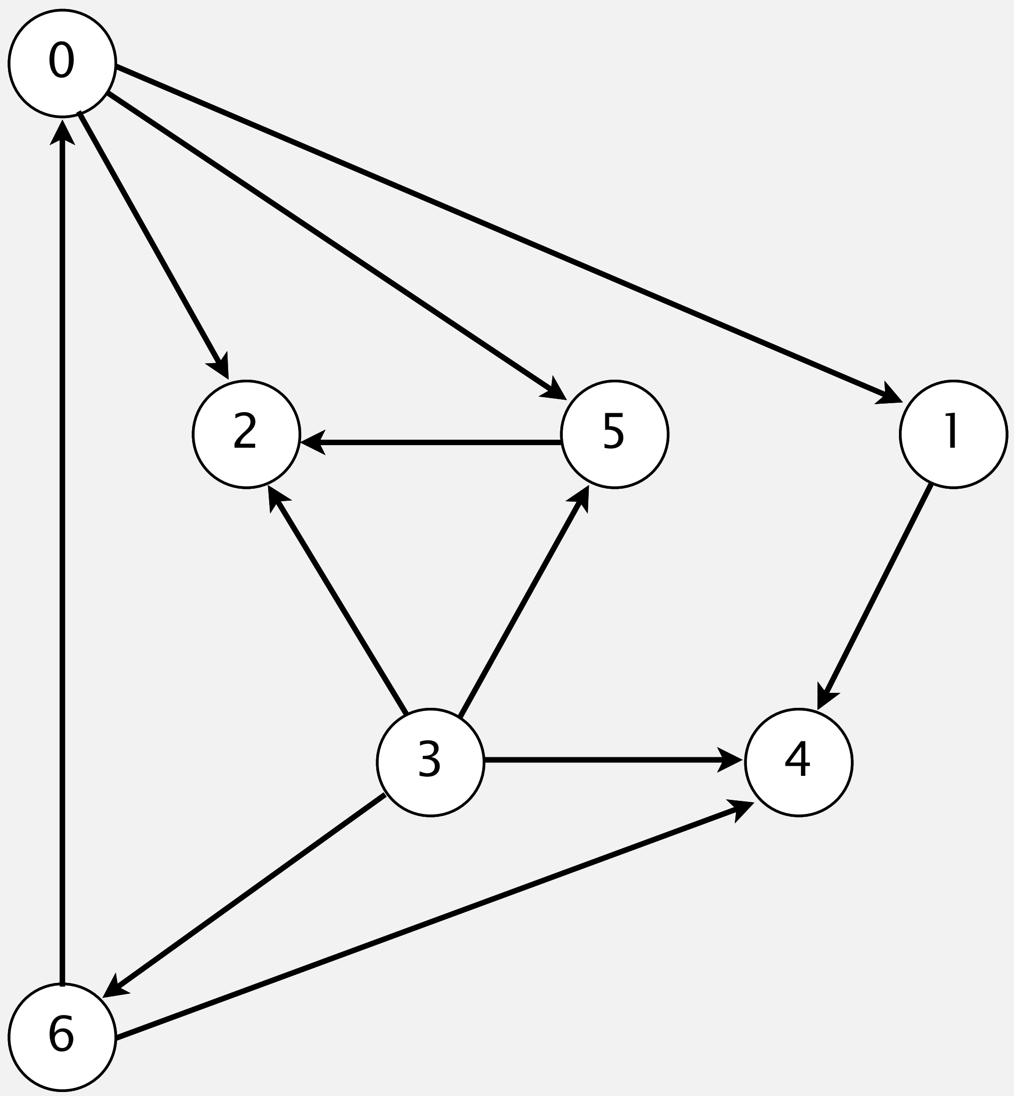
4 1 2 5 0 6 3

v	marked[]
0	T
1	T
2	T
3	T
4	T
5	T
6	T

check 6

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



postorder

4 1 2 5 0 6 3

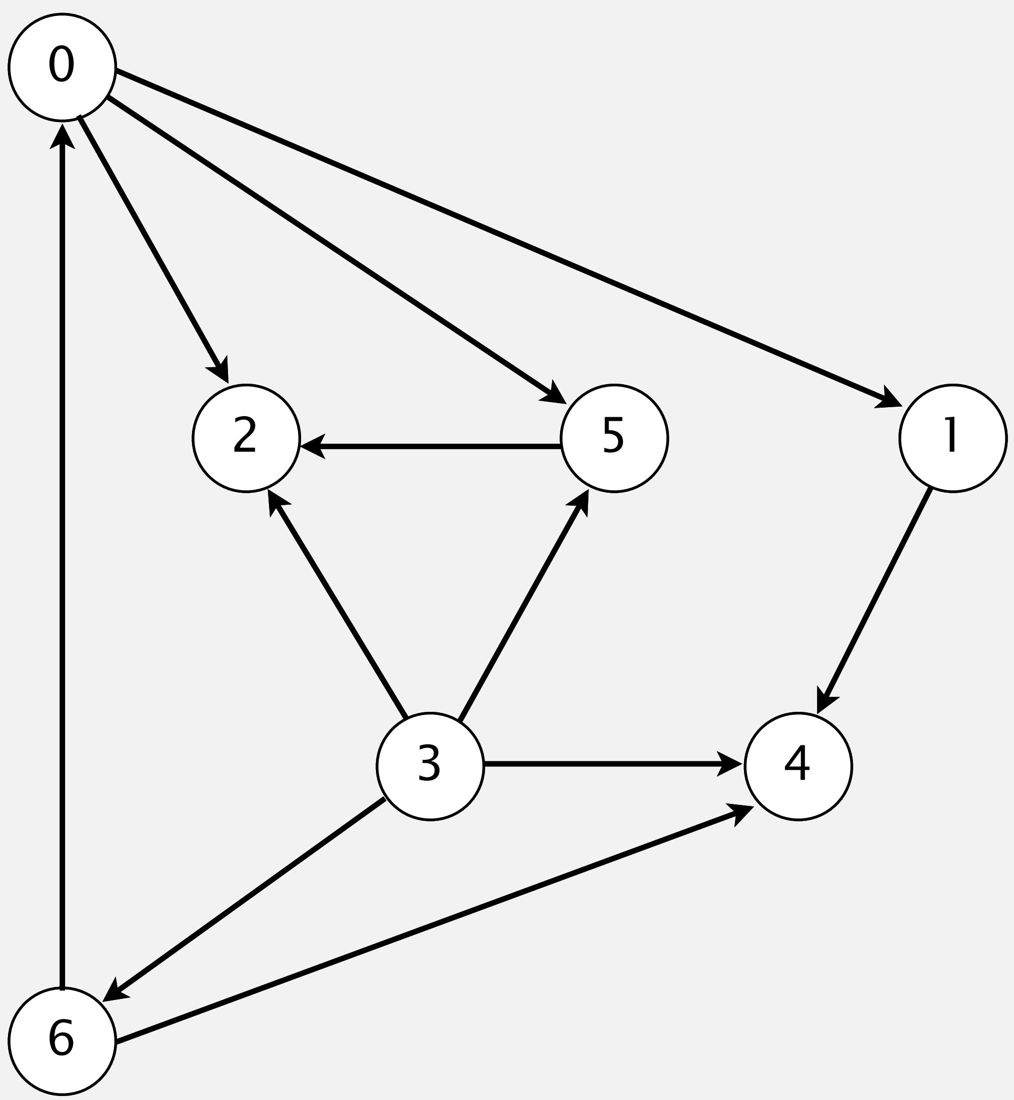
topological order

3 6 0 5 2 1 4

done

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



postorder

4 1 2 5 0 6 3

topological order

3 6 0 5 2 1 4

done

Depth-first search order

```
public class DepthFirstOrder
{
   private boolean[] marked;
   private Stack<Integer> reversePostorder;

   public DepthFirstOrder(Digraph G)
   {
      reversePostorder = new Stack<Integer>();
      marked = new boolean[G.V()];
      for (int v = 0; v < G.V(); v++)
         if (!marked[v]) dfs(G, v);
   }

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

   public Iterable<Integer> reversePostorder()
   {  return reversePostorder;  }
}
```

returns all vertices in "reverse DFS postorder"

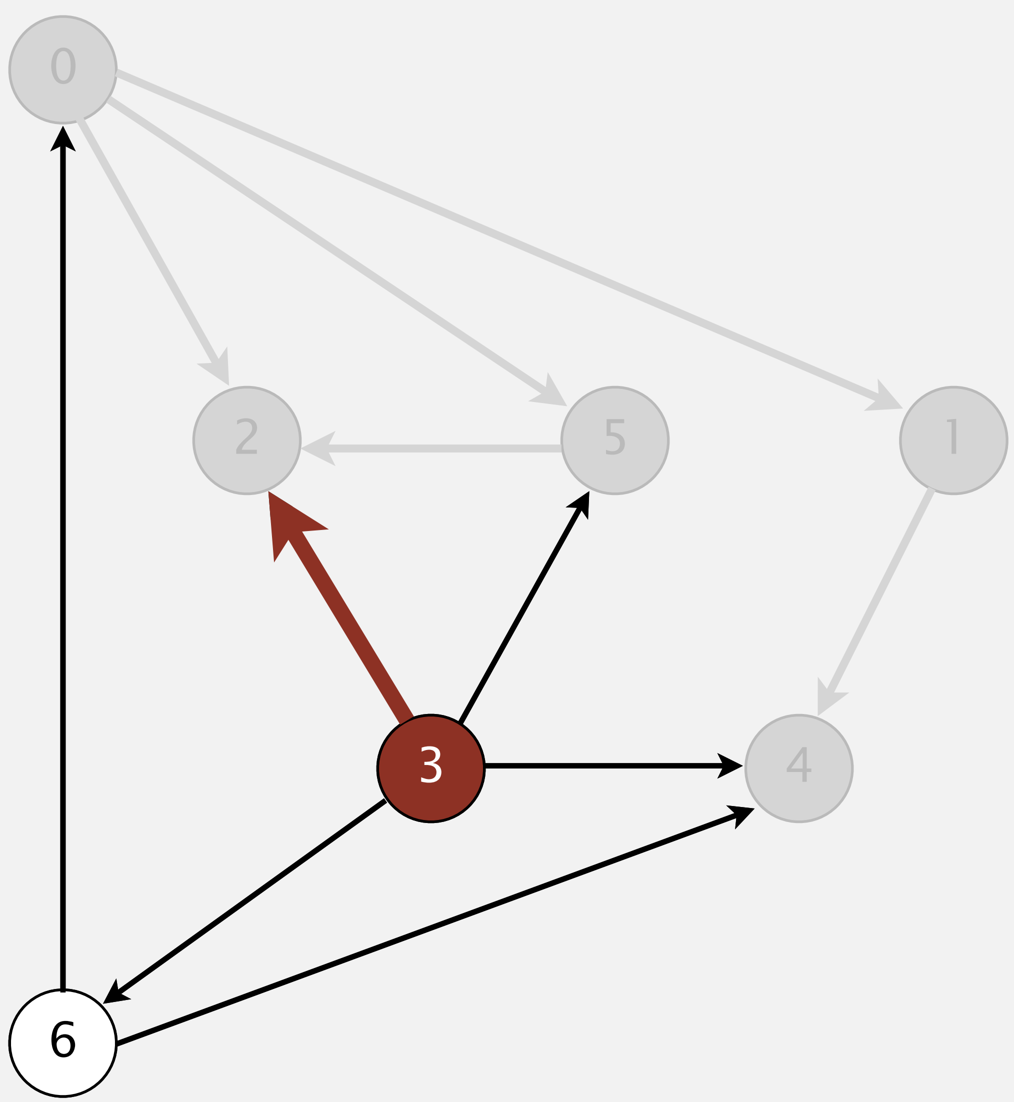
Topological sort in a DAG: correctness proof

Proposition. Reverse DFS postorder of a DAG is a topological order.

Pf. Consider any edge `v→w`. When `dfs(v)` is called:

Need to show the W gets returned before v

- Case 1: `dfs(w)` has already been called and returned.
  Thus, `w` was done before `v`.



v = 3

case 1

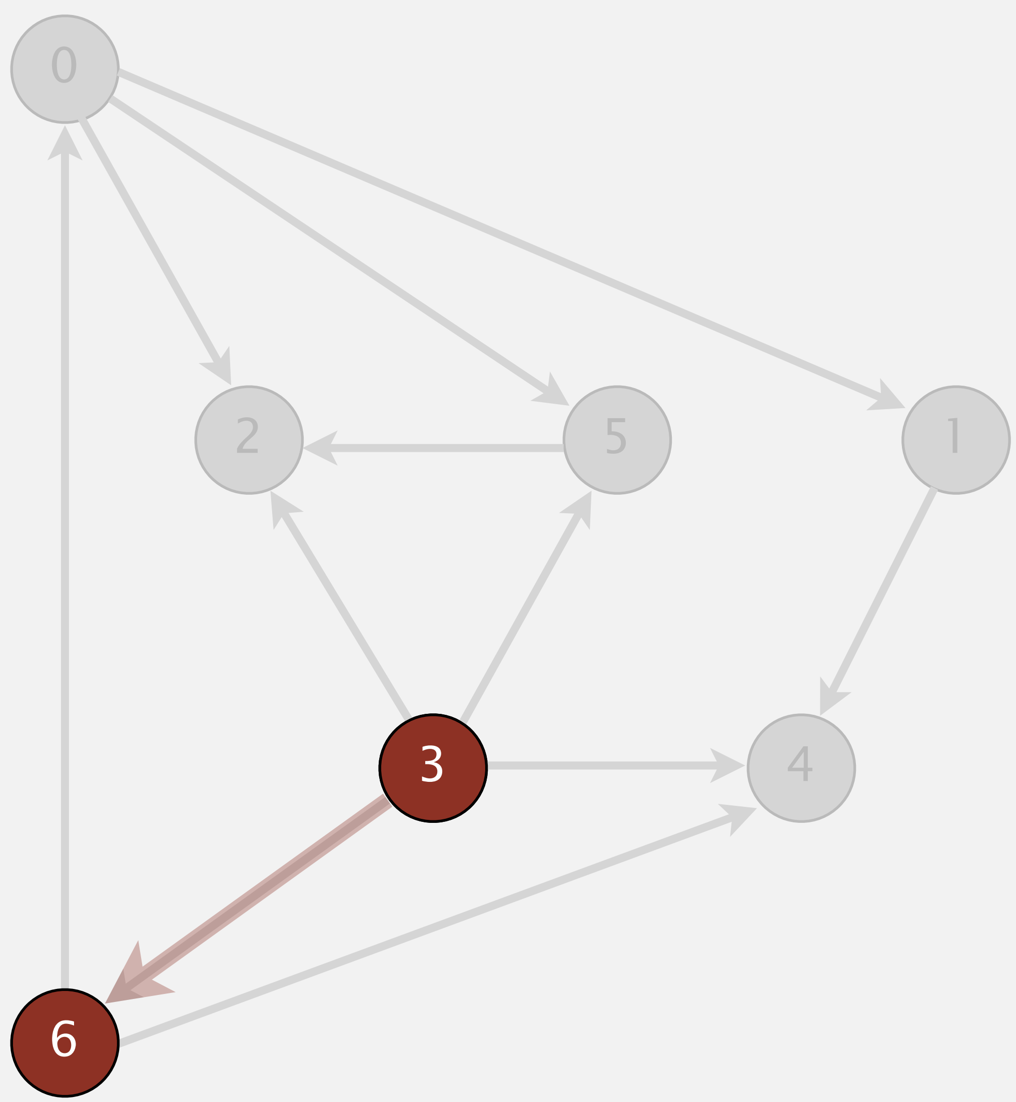
```
dfs(0)
  dfs(1)
    dfs(4)
    4 done
  1 done
  dfs(2)
  2 done
  dfs(5)
    check 2
  5 done
0 done
check 1
check 2
dfs(3)
  check 2
  check 4
  check 5
  dfs(6)
    check 0
    check 4
  6 done
3 done
check 4
check 5
check 6
done
```

Topological sort in a DAG: correctness proof

Proposition. Reverse DFS postorder of a DAG is a topological order.

Pf. Consider any edge `v→w`. When `dfs(v)` is called:

- Case 2: `dfs(w)` has not yet been called. `dfs(w)` will get called directly or indirectly by `dfs(v)` and will finish before `dfs(v)`. Thus, `w` will be done before `v`.



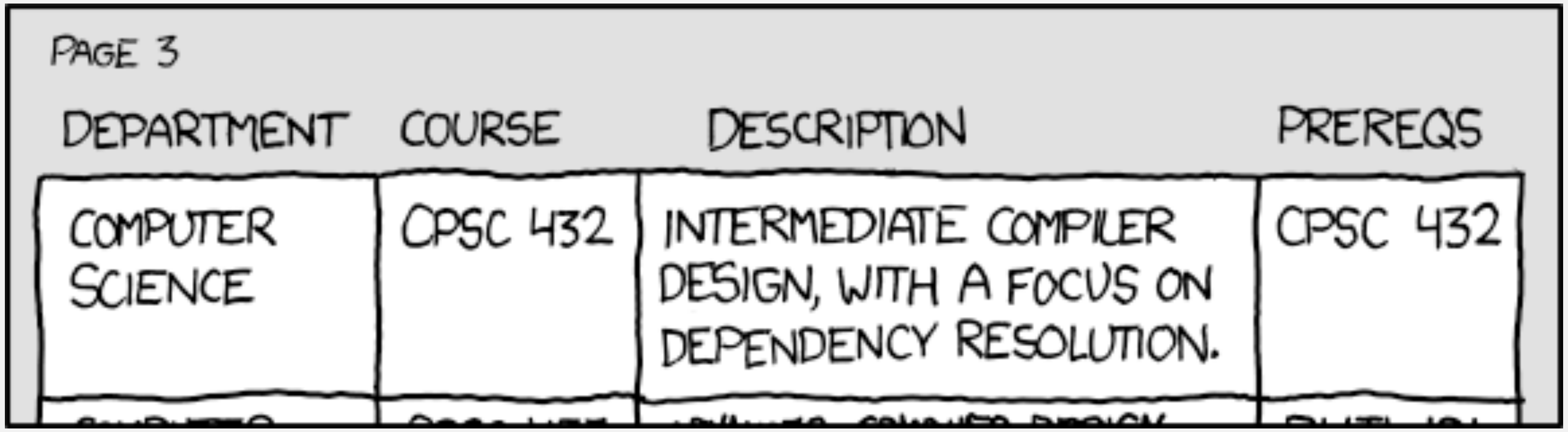
W= 6

case 2

```
dfs(0)
  dfs(1)
    dfs(4)
    4 done
  1 done
  dfs(2)
  2 done
  dfs(5)
    check 2
  5 done
0 done
check 1
check 2
dfs(3)
  check 2
  check 4
  check 5
  dfs(6)
    check 0
    check 4
  6 done
3 done
check 4
check 5
check 6
done
```

Directed cycle detection application: precedence scheduling

Scheduling. Given a set of tasks to be completed with precedence constraints, in what order should we schedule the tasks?



http://xkcd.com/754

Remark. A directed cycle implies scheduling problem is infeasible.

STRONGLY CONNECTED COMPONENTS

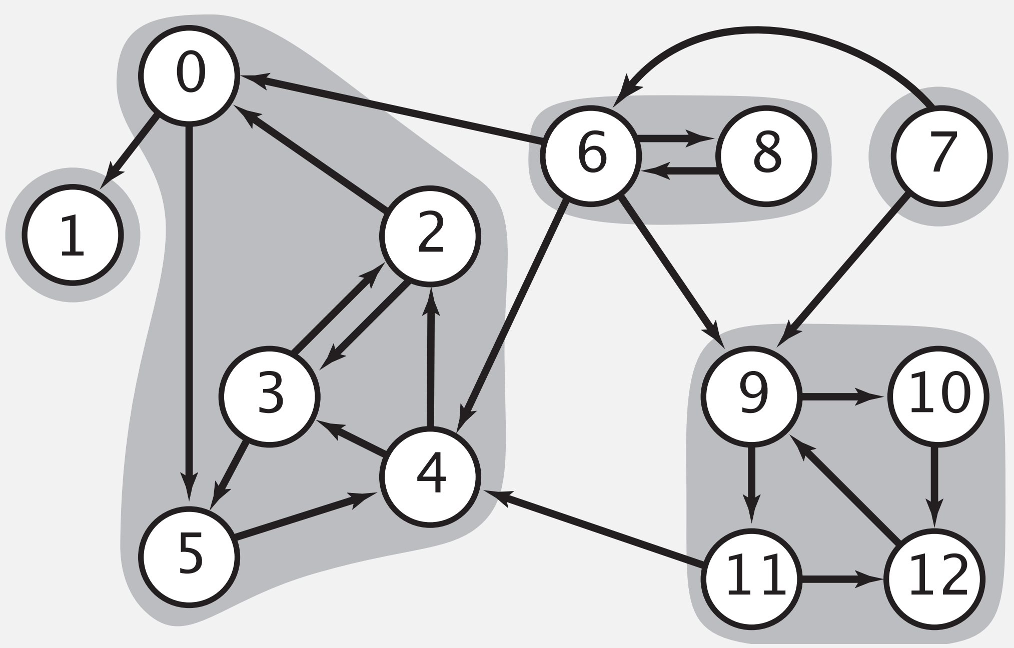
Strongly-connected components

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.

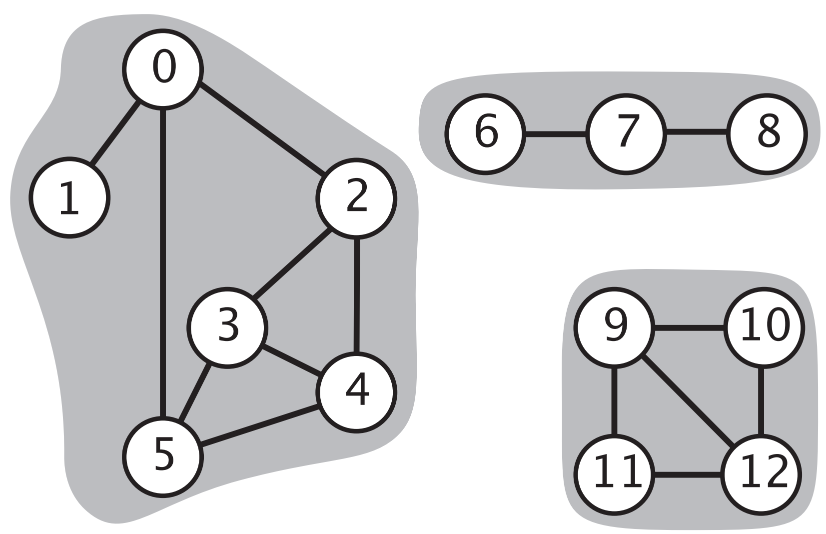
Def. A strong component is a maximal subset of strongly-connected vertices.



5 strongly-connected components

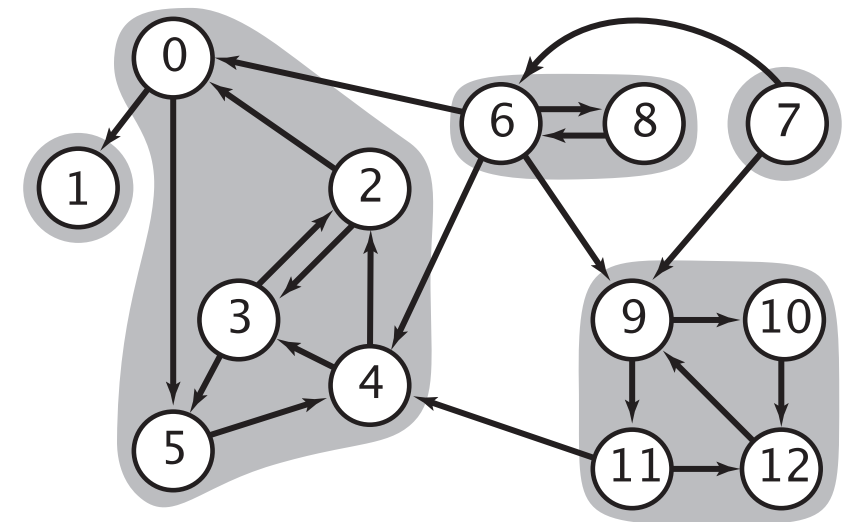
Connected components vs. strongly-connected components

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



3 connected components

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



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
id[]	0	0	0	0	0	0	1	1	1	2	2	2	2

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

constant-time client connectivity query

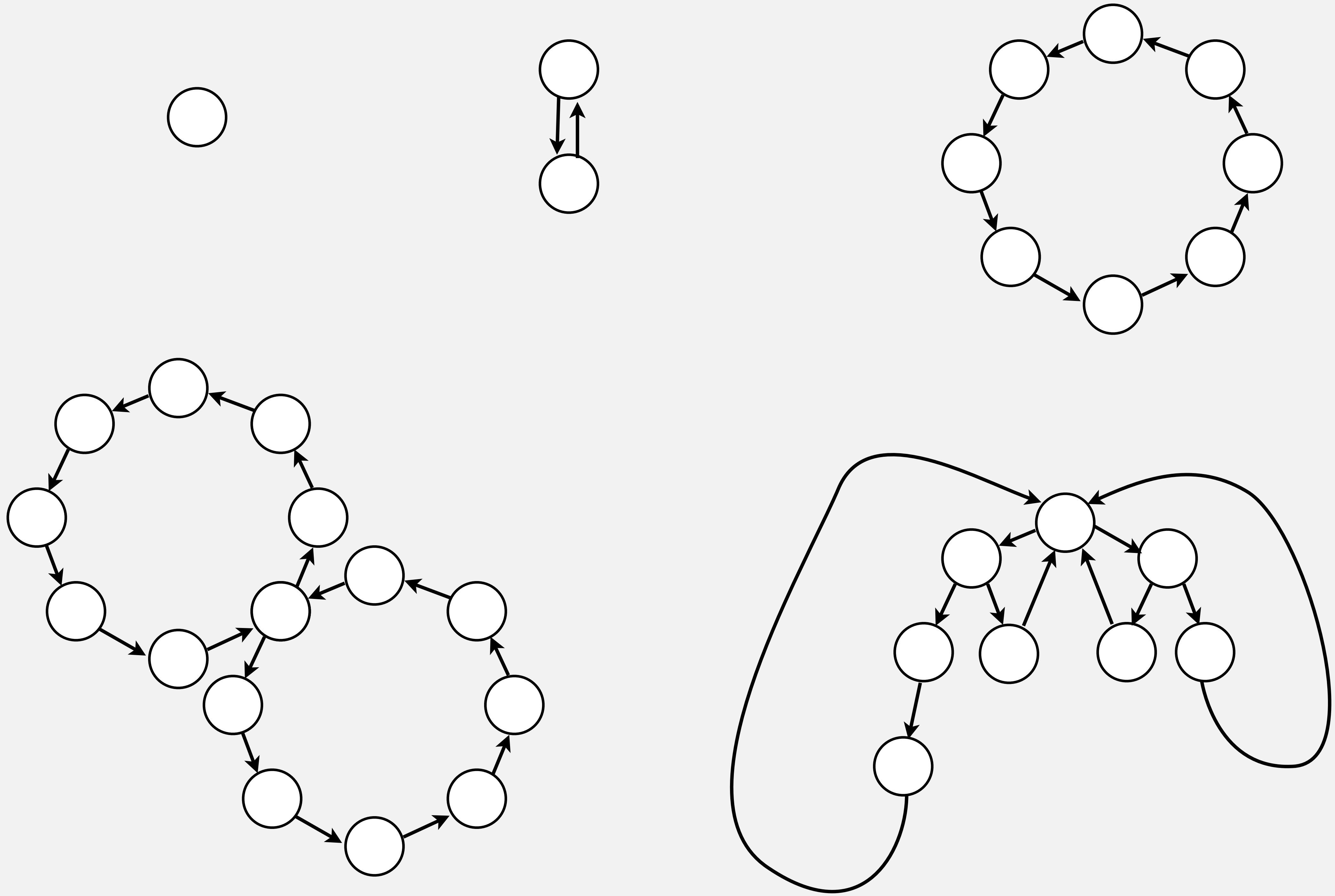
strongly-connected component id (how to compute?)

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

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

constant-time client strong-connectivity query

Examples of strongly-connected digraphs



Strong components algorithms: brief history

1972: linear-time DFS algorithm (Tarjan).

- Classic algorithm.
- 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).

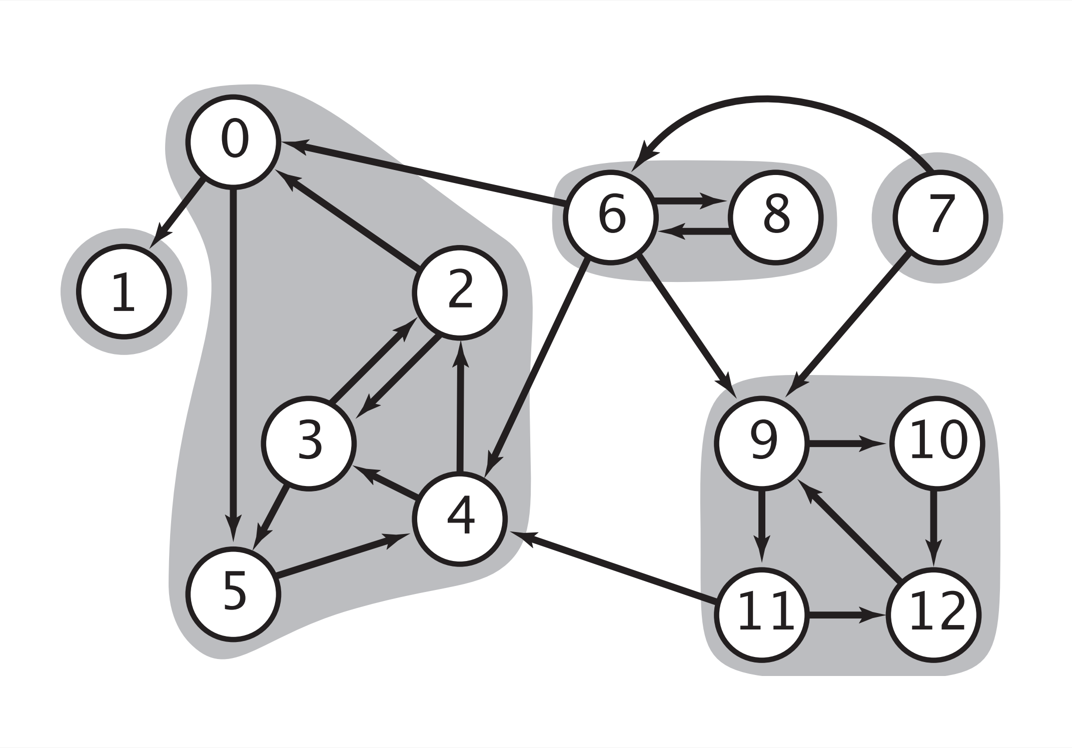
Kosaraju-Sharir algorithm: intuition

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

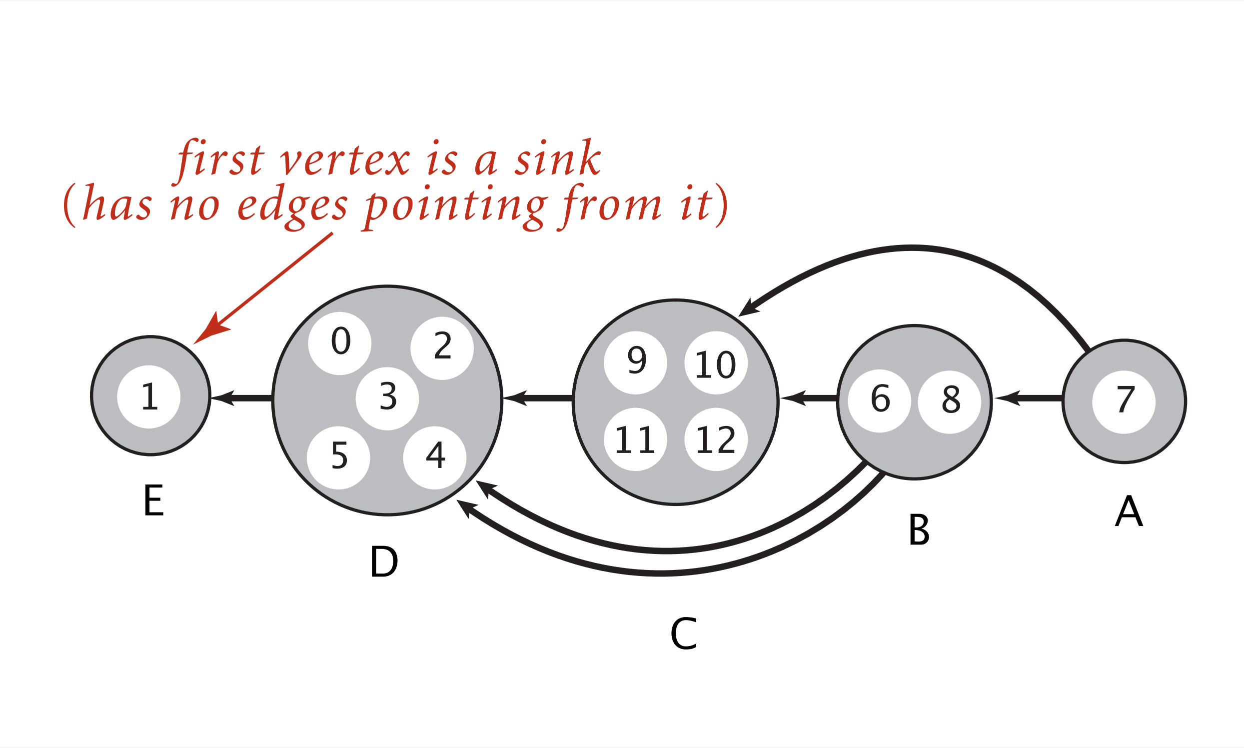
Kernel DAG. Contract each strong component into a single vertex.

Idea.

- Compute topological order (reverse postorder) in kernel DAG.
- Run DFS, considering vertices in reverse topological order.
- All Vertex that we in the DFS will be in same strong component



digraph G and its strong components

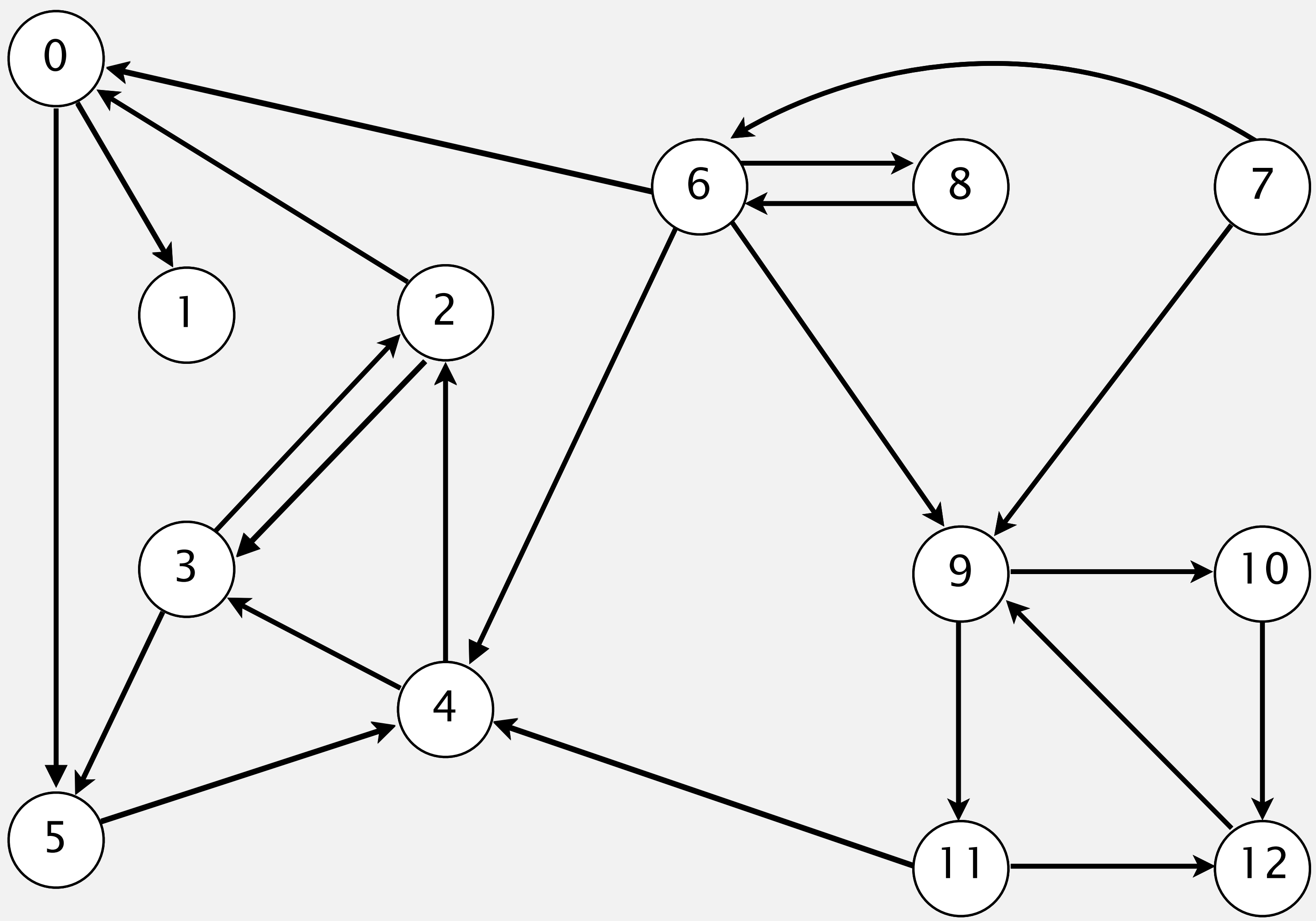


kernel DAG of G (topological order: A B C D E)

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.



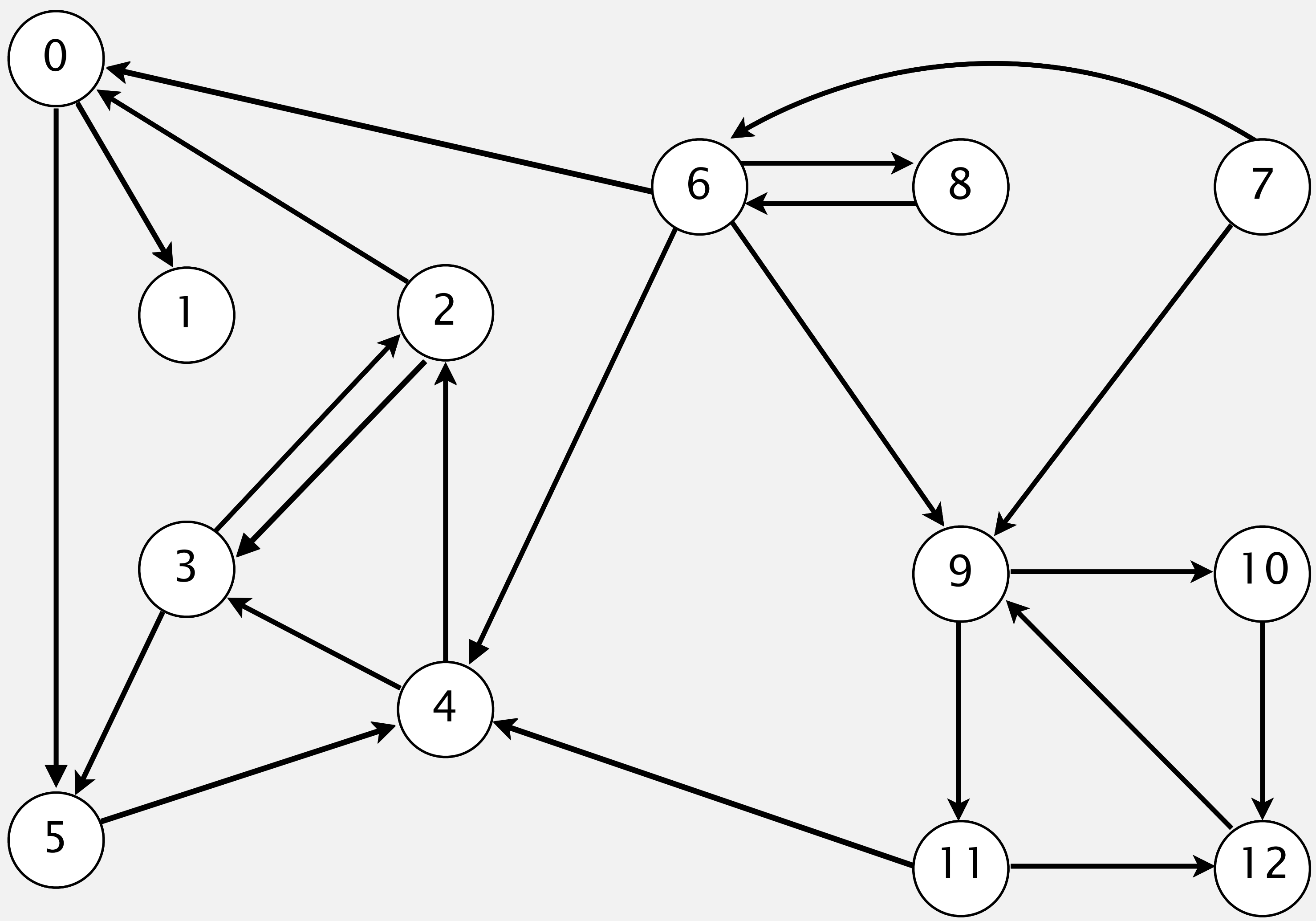
digraph G

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
6→4
6→9
7→6

DFS IN THE REVERSE GRAPH

Kosaraju-Sharir algorithm demo

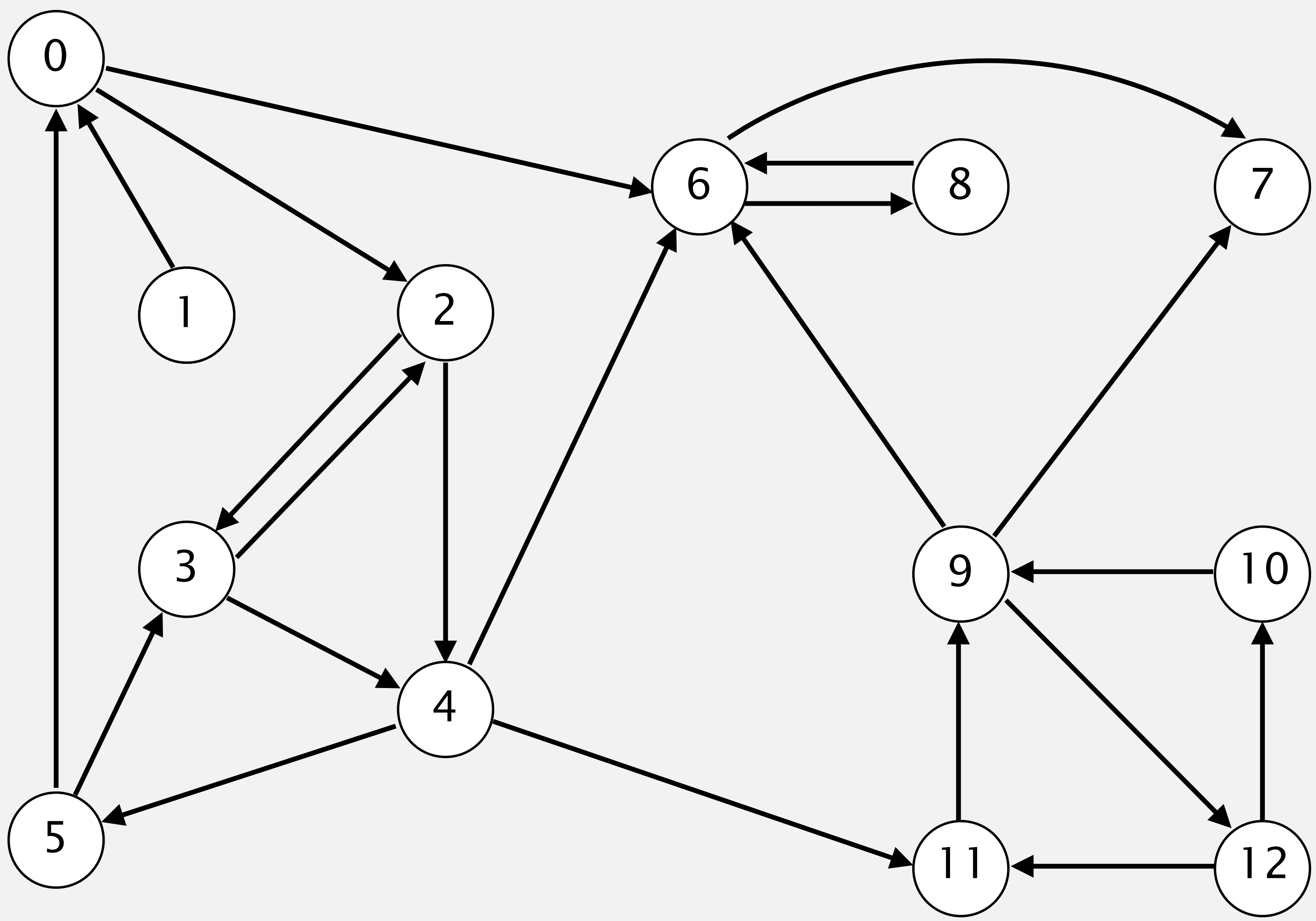
Phase 1. Compute reverse postorder in G^R.



digraph G

Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in G^R.

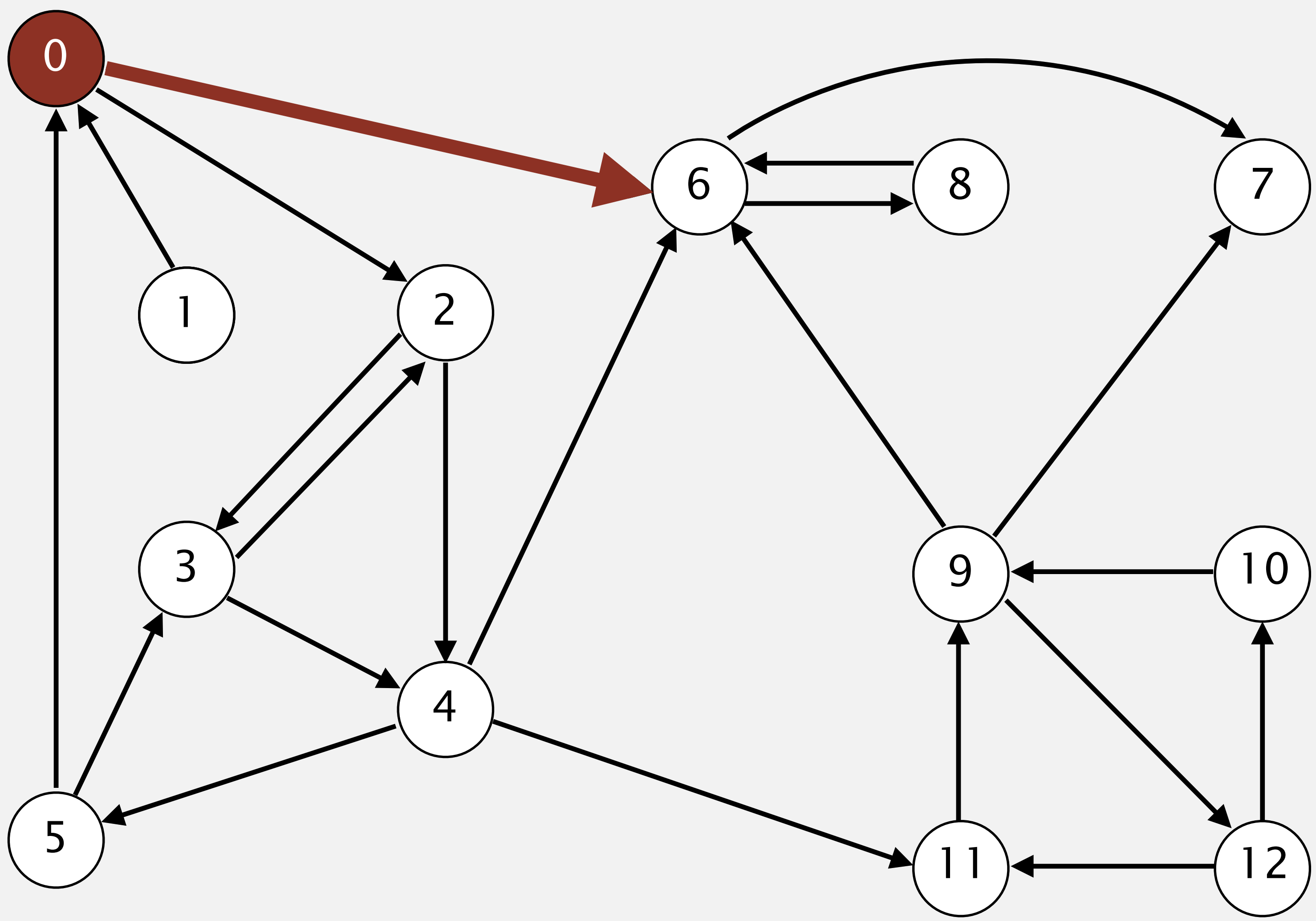


reverse digraph G^R

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.

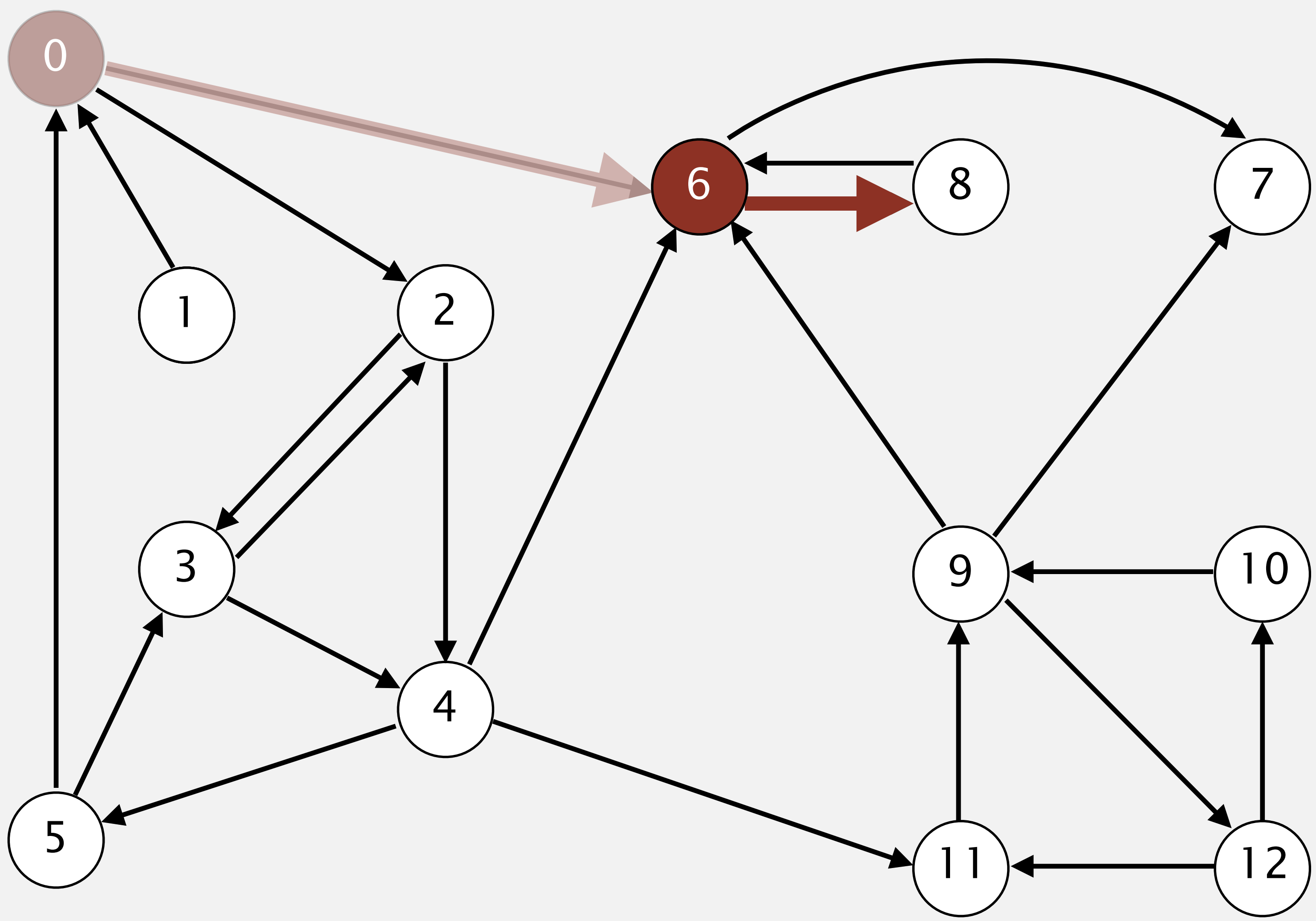


v	marked[]
0	T
1	F
2	F
3	F
4	F
5	F
6	F
7	F
8	F
9	F
10	F
11	F
12	F

visit 0: check 6 and check 2

Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in G^R.

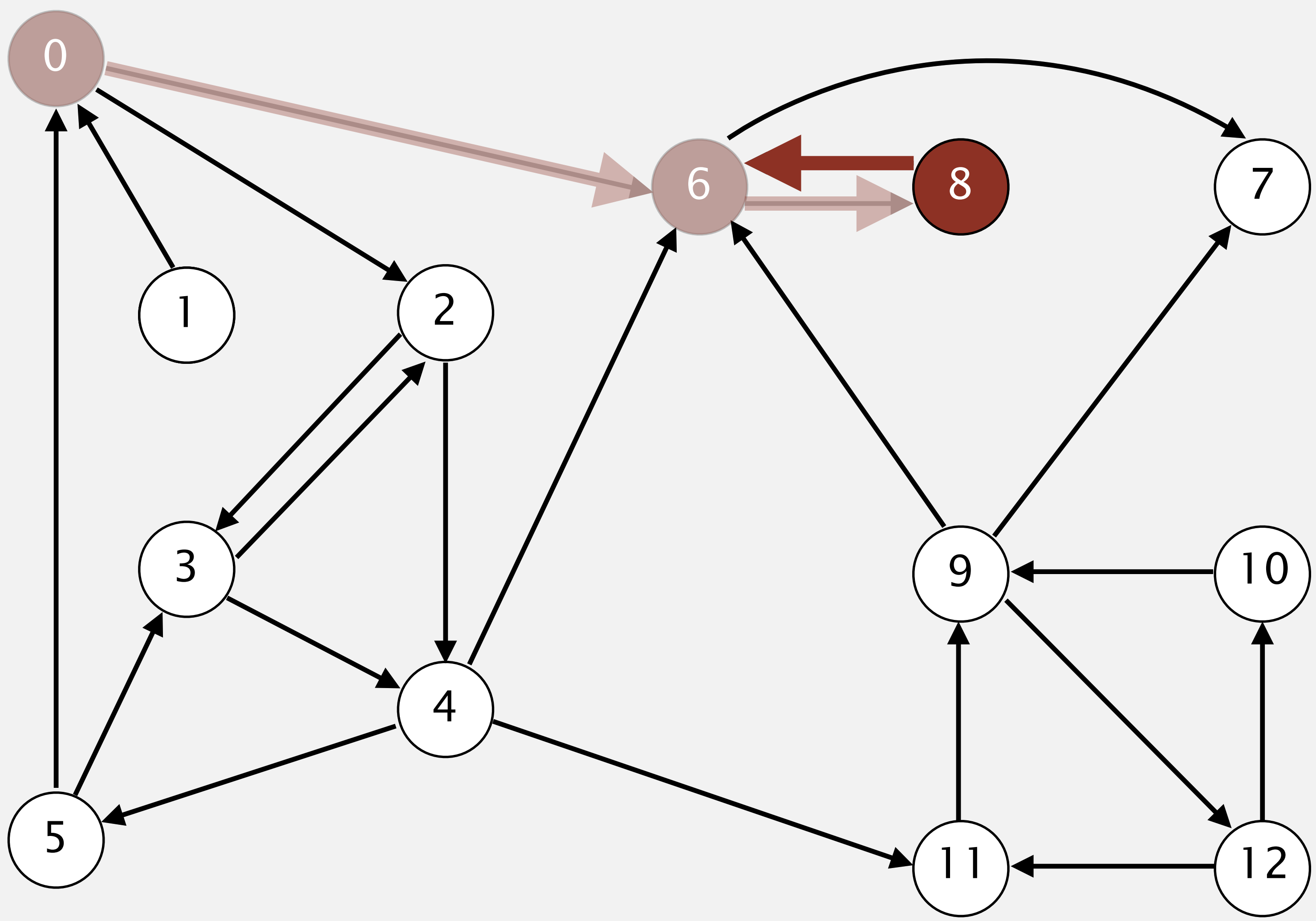


v	marked[]
0	T
1	F
2	F
3	F
4	F
5	F
6	T
7	F
8	F
9	F
10	F
11	F
12	F

visit 6: check 8 and check 7

Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in G^R.



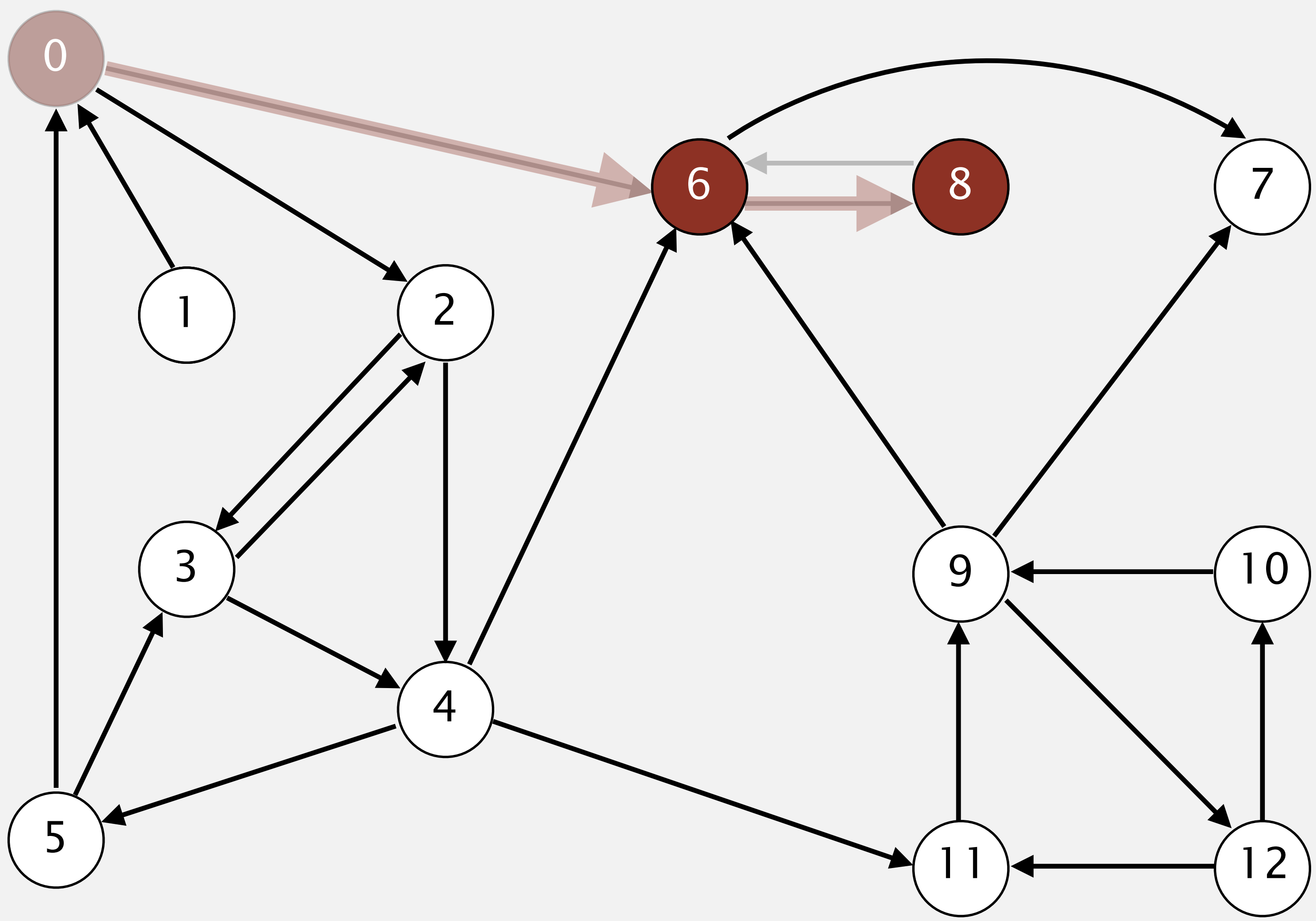
v	marked[]
0	T
1	F
2	F
3	F
4	F
5	F
6	T
7	F
8	T
9	F
10	F
11	F
12	F

visit 8: check 6

Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in G^R.

8



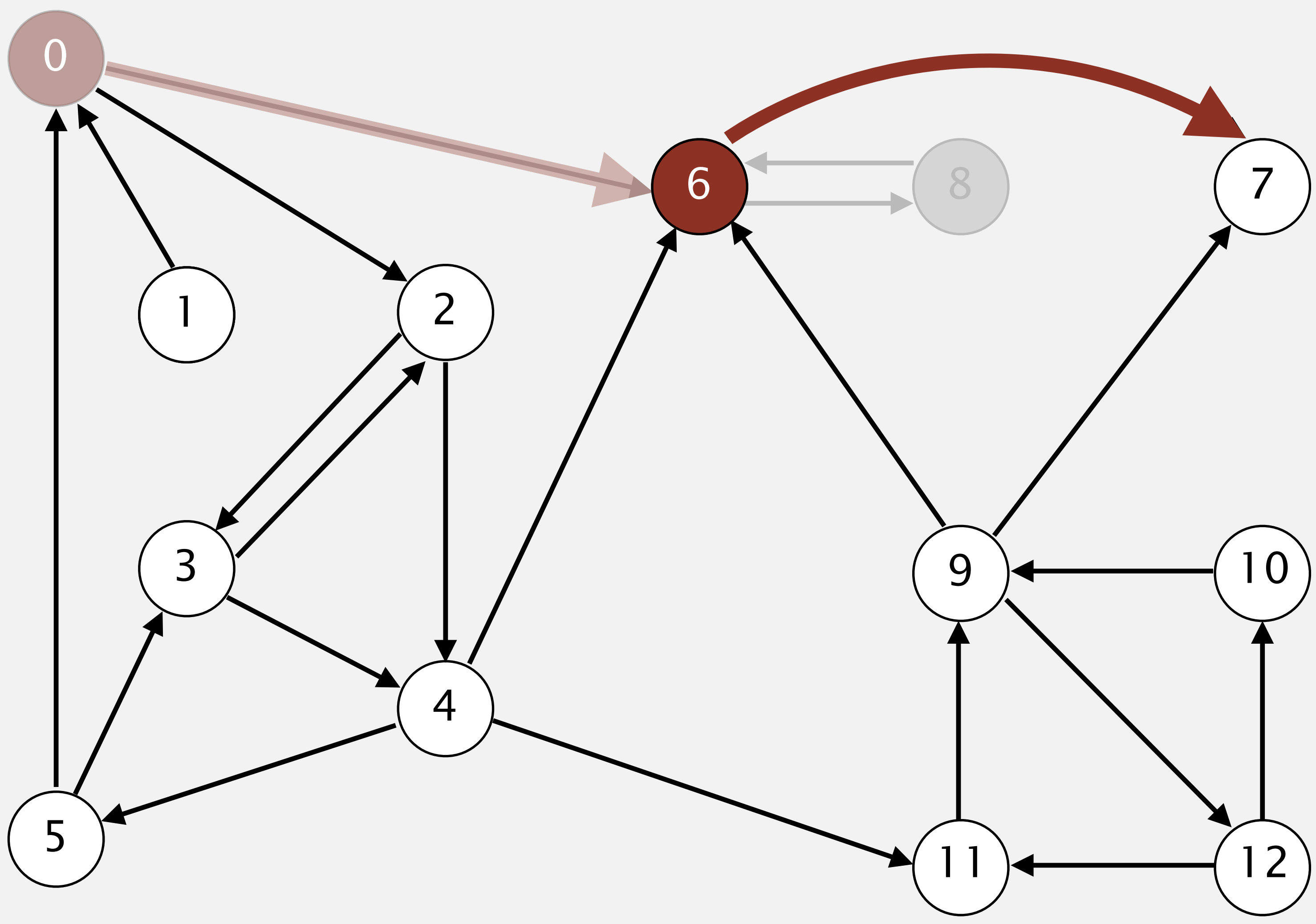
v	marked[]
0	T
1	F
2	F
3	F
4	F
5	F
6	T
7	F
8	T
9	F
10	F
11	F
12	F

8 done

Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in G^R.

8



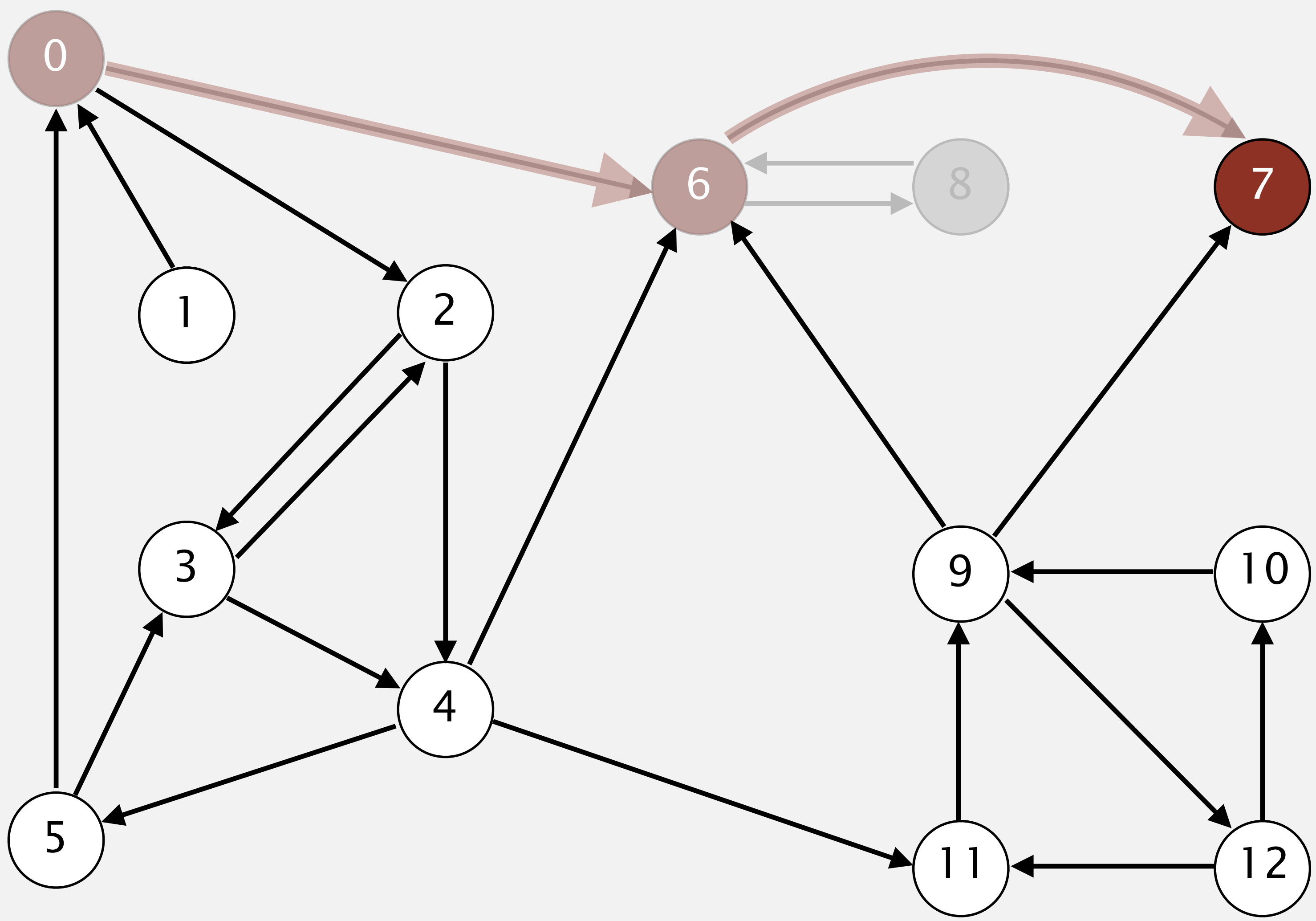
v	marked[]
0	T
1	F
2	F
3	F
4	F
5	F
6	T
7	F
8	T
9	F
10	F
11	F
12	F

visit 6: check 8 and check 7

Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in G^R.

8



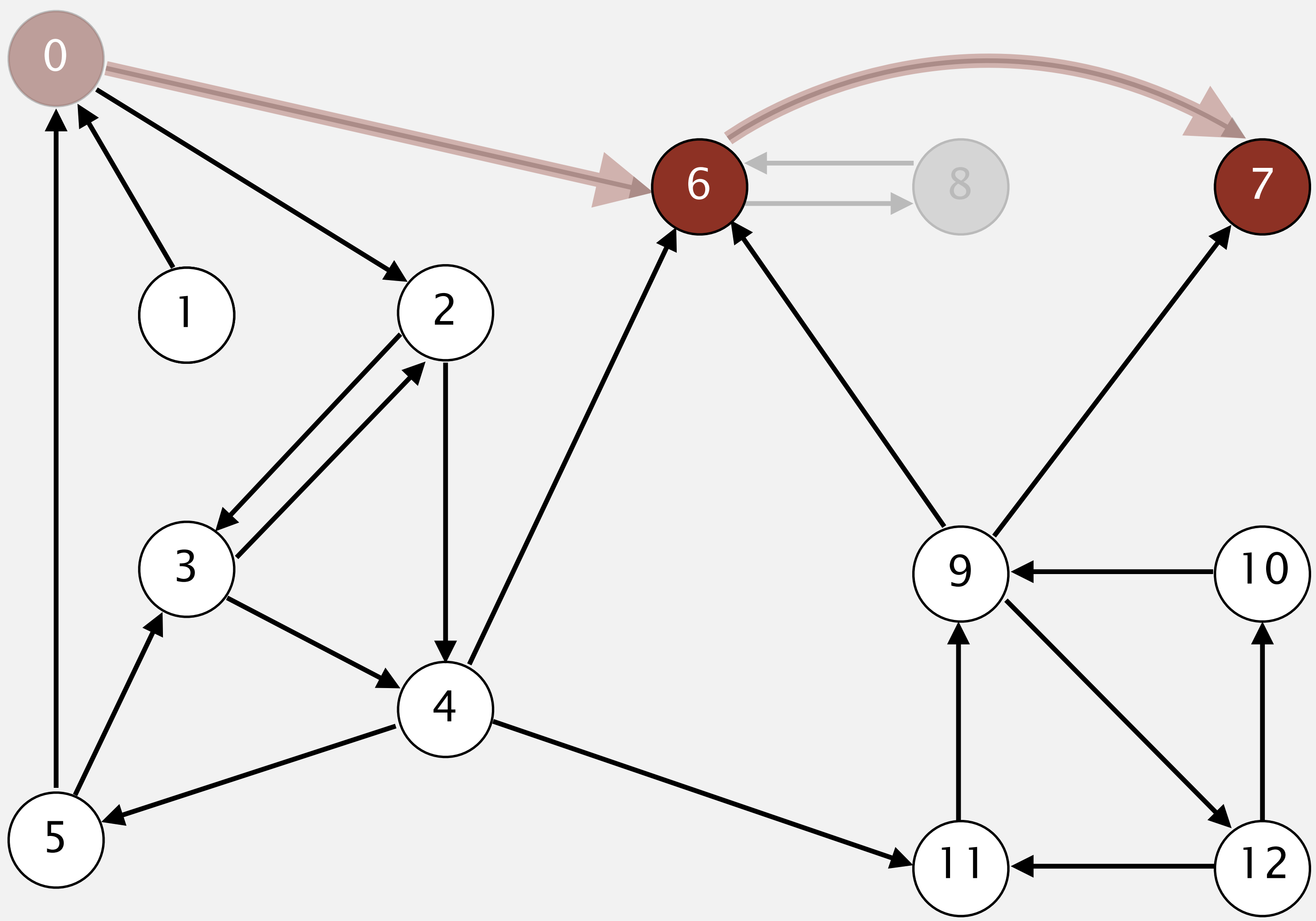
v	marked[]
0	T
1	F
2	F
3	F
4	F
5	F
6	T
7	T
8	T
9	F
10	F
11	F
12	F

visit 7

Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in G^R.

7 8



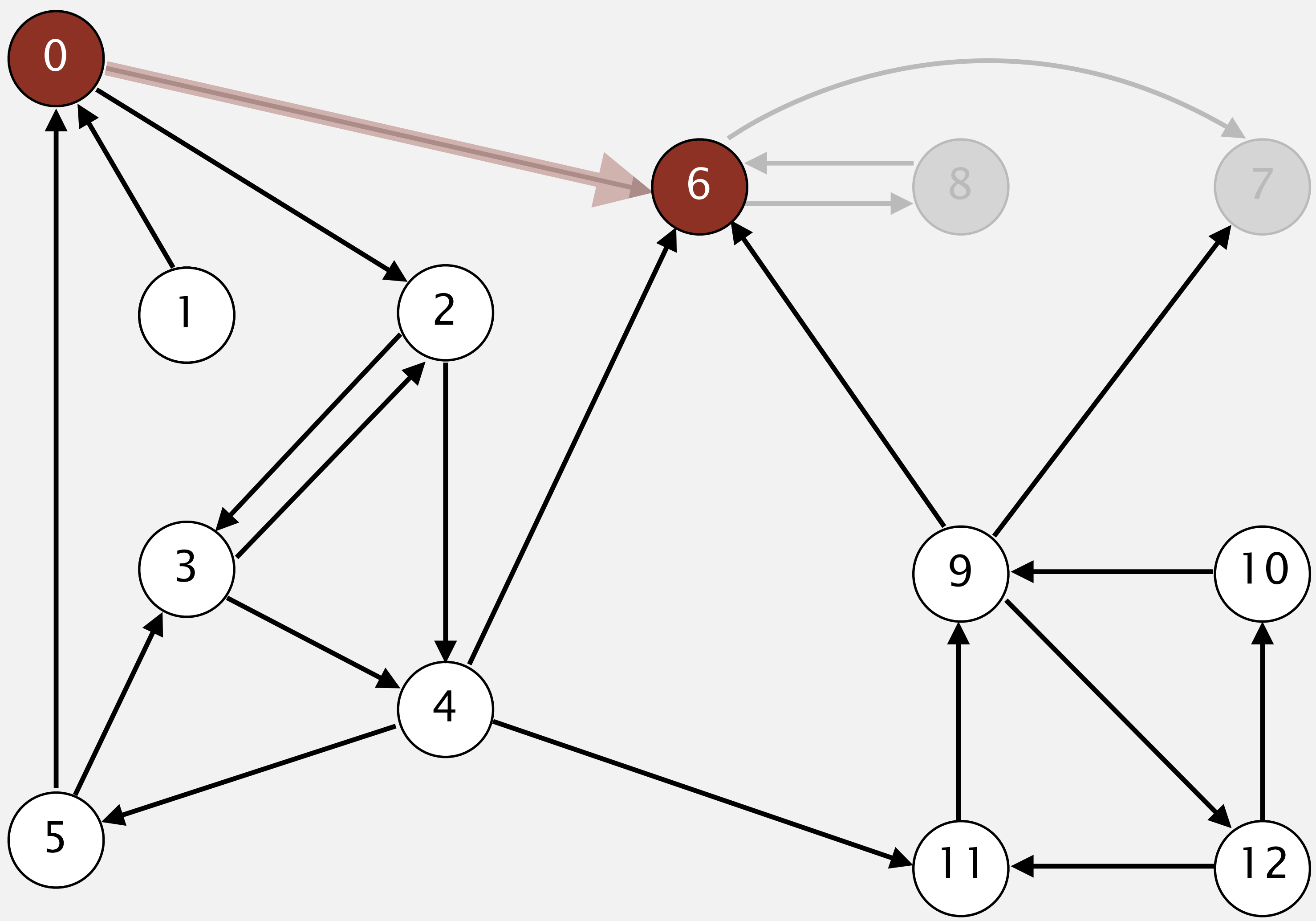
v	marked[]
0	T
1	F
2	F
3	F
4	F
5	F
6	T
7	T
8	T
9	F
10	F
11	F
12	F

7 done

Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in G^R.

6 7 8



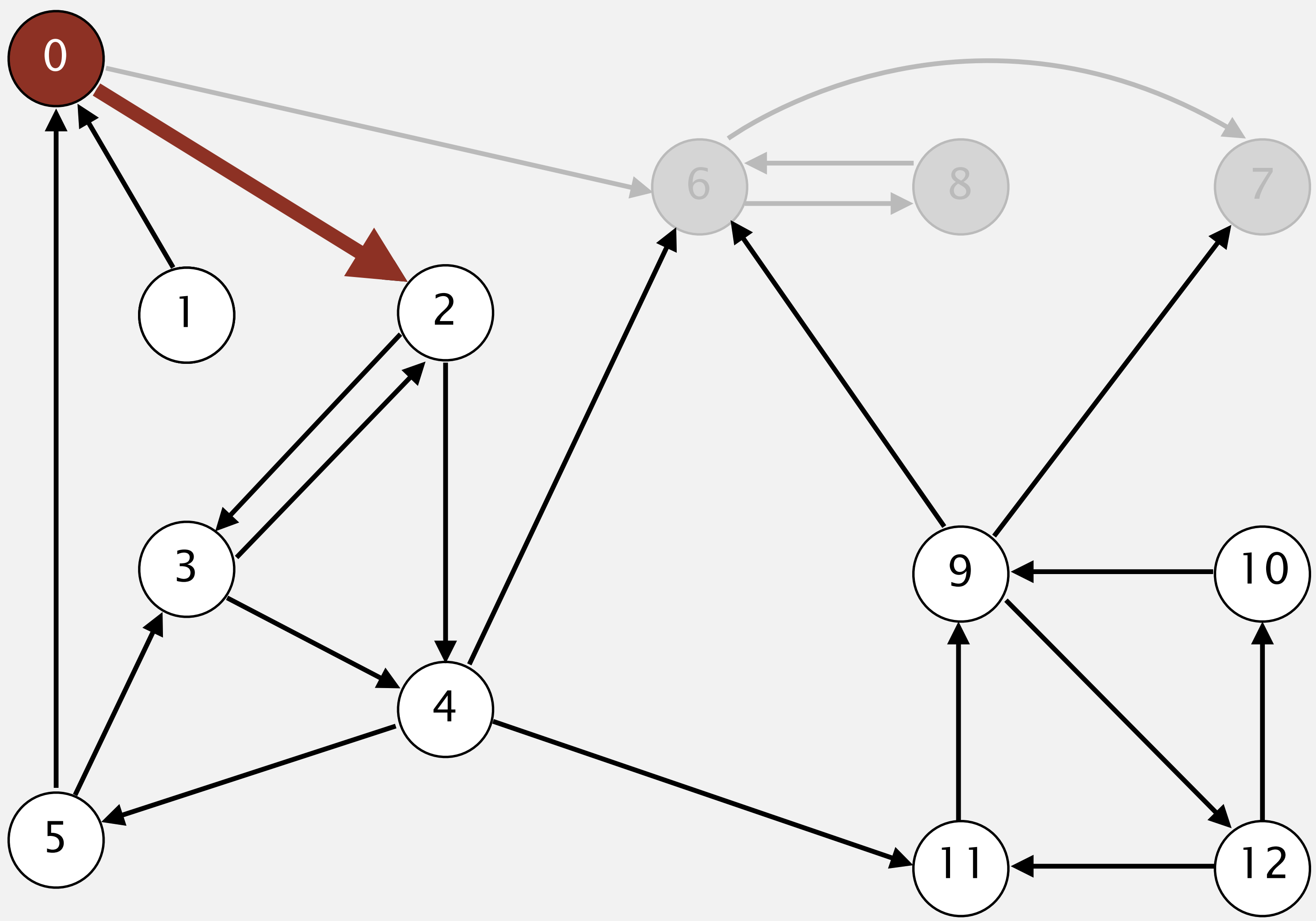
v	marked[]
0	T
1	F
2	F
3	F
4	F
5	F
6	T
7	T
8	T
9	F
10	F
11	F
12	F

6 done

Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in G^R.

6 7 8



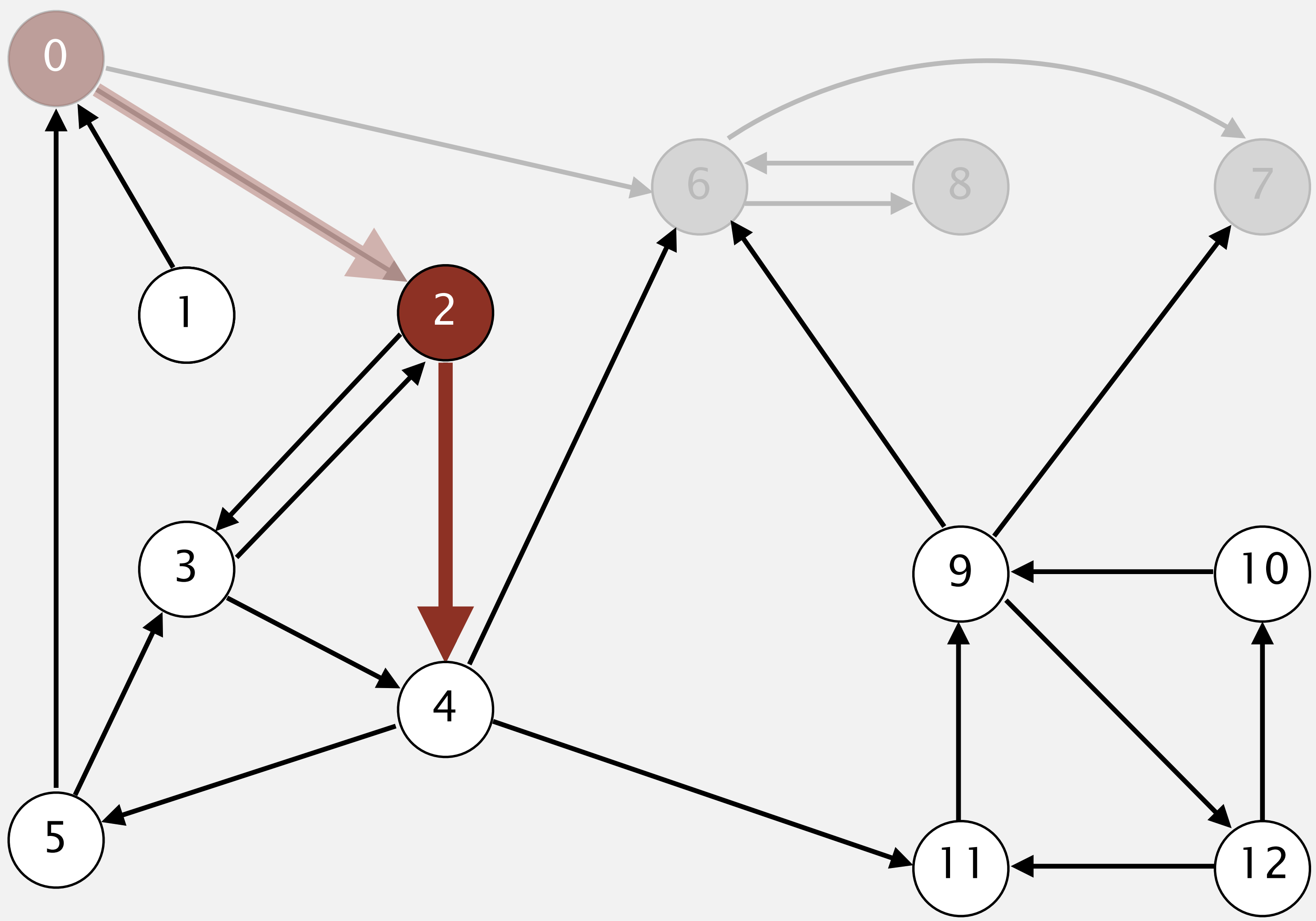
v	marked[]
0	T
1	F
2	F
3	F
4	F
5	F
6	T
7	T
8	T
9	F
10	F
11	F
12	F

visit 0: check 6 and check 2

Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in G^R.

6 7 8



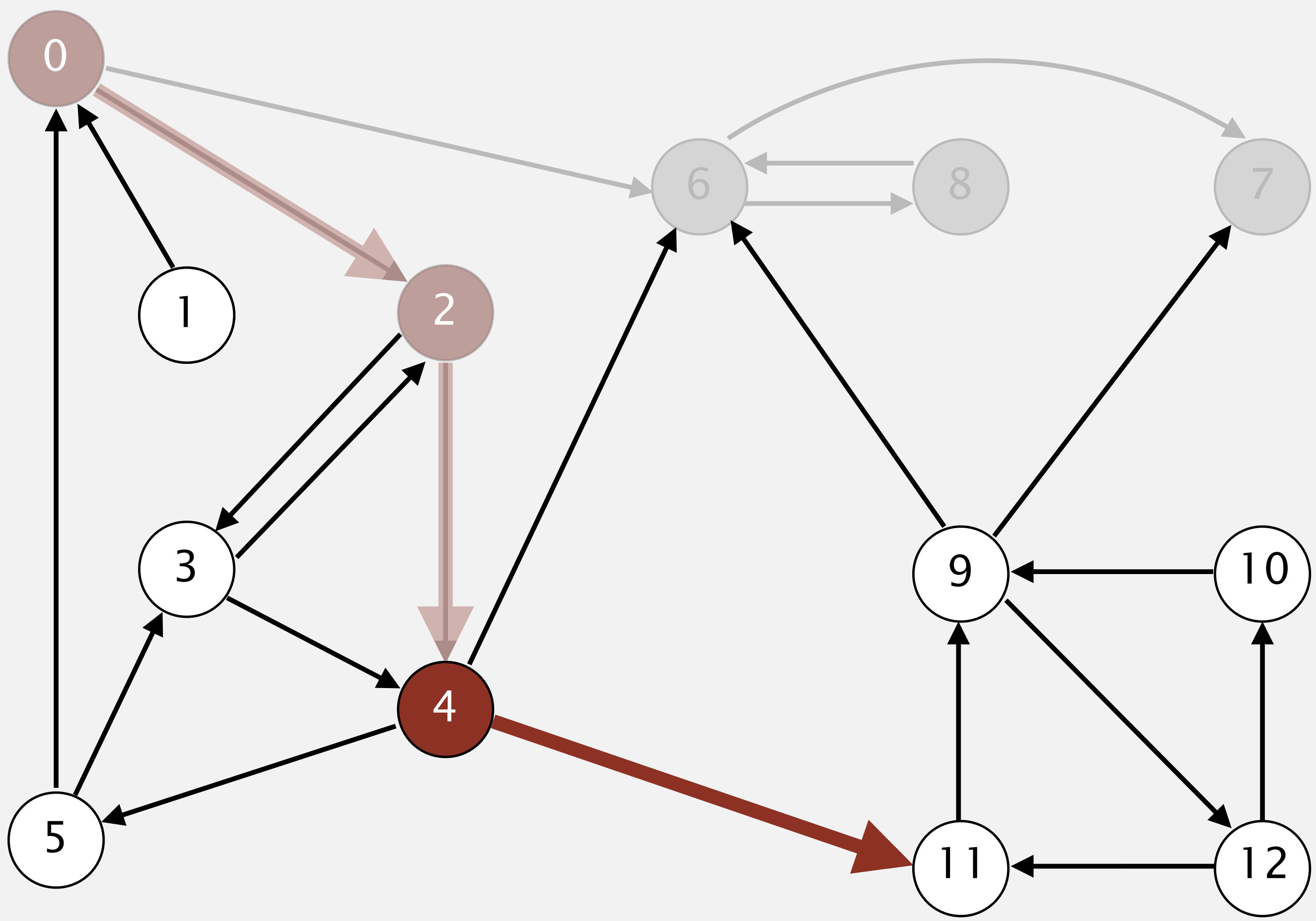
v	marked[]
0	T
1	F
2	T
3	F
4	F
5	F
6	T
7	T
8	T
9	F
10	F
11	F
12	F

visit 2: check 4 and check 3

Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in G^R.

6 7 8



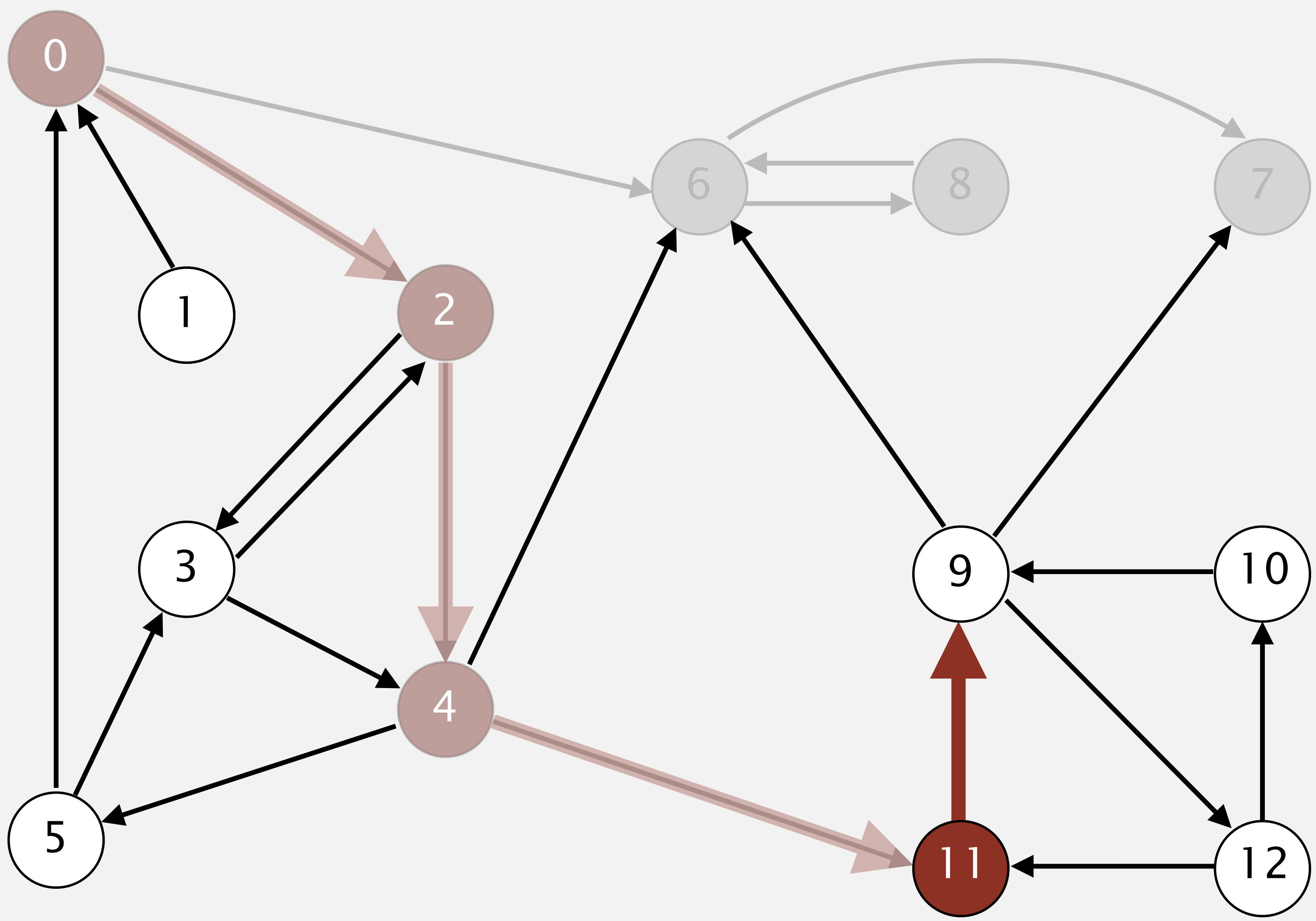
v	marked[]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	F
10	F
11	F
12	F

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

Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in G^R.

6 7 8



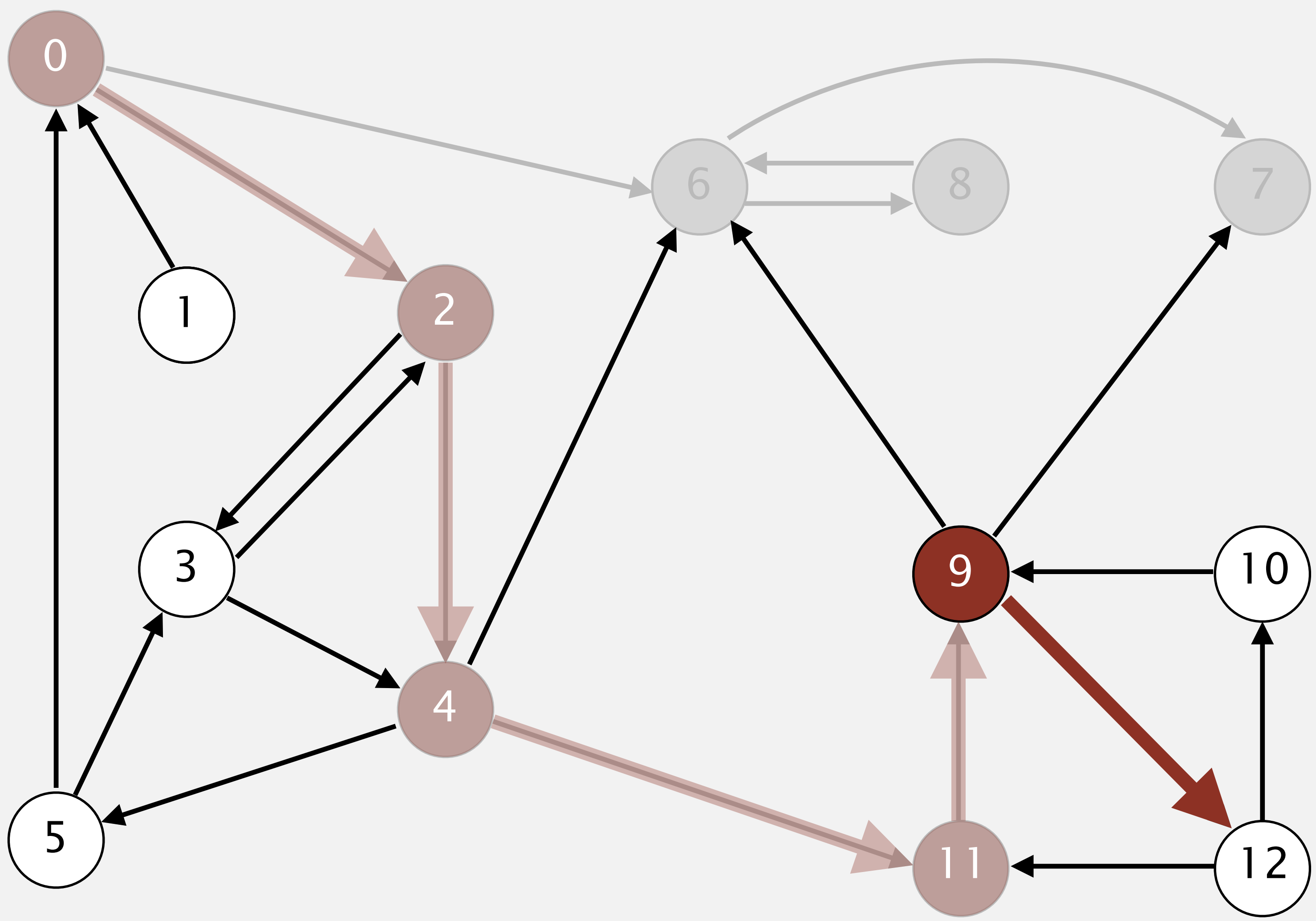
v	marked[]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	F
10	F
11	T
12	F

visit 11: check 9

Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in G^R.

6 7 8



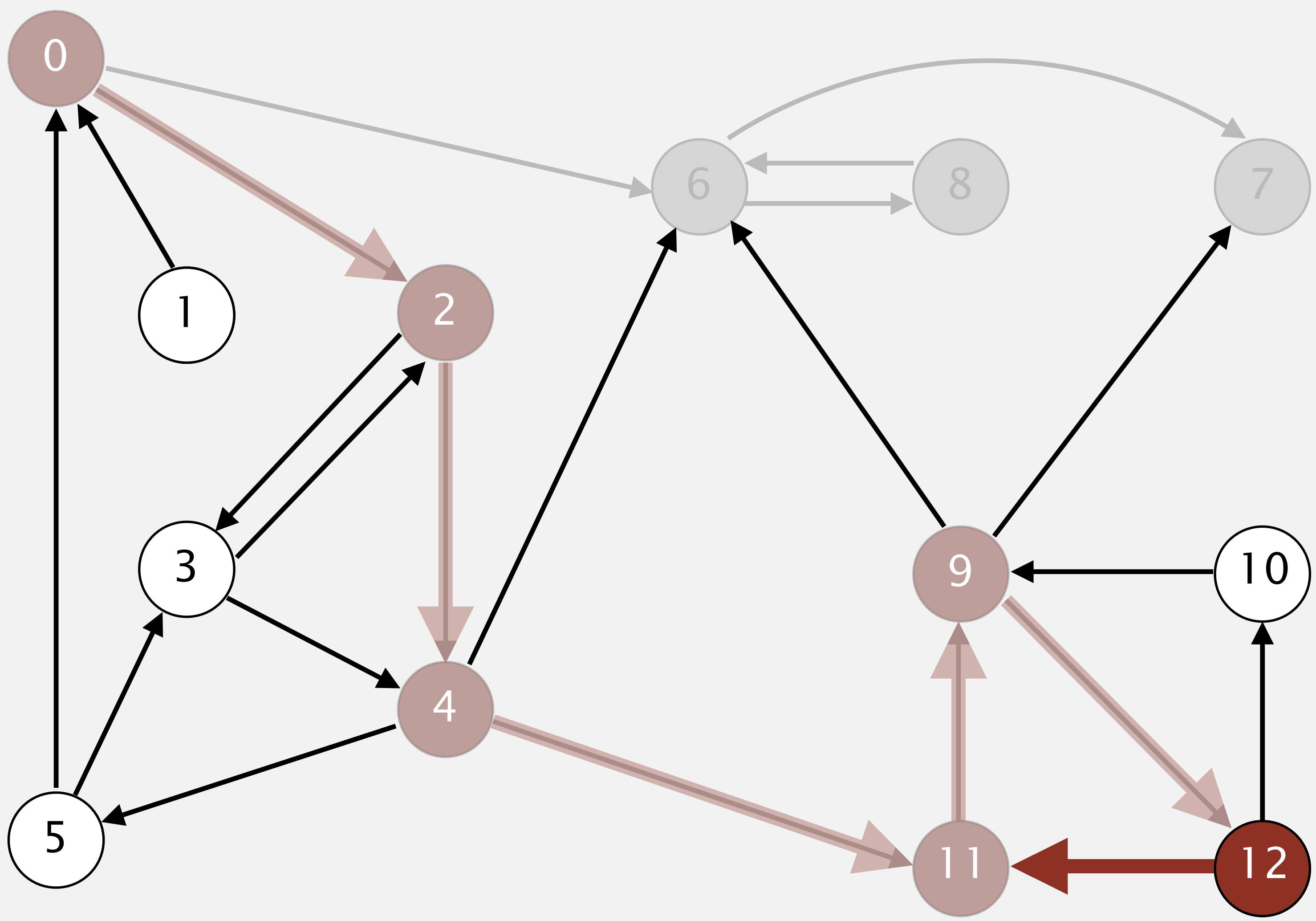
v	marked[]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	T
10	F
11	T
12	F

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

Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in G^R.

6 7 8



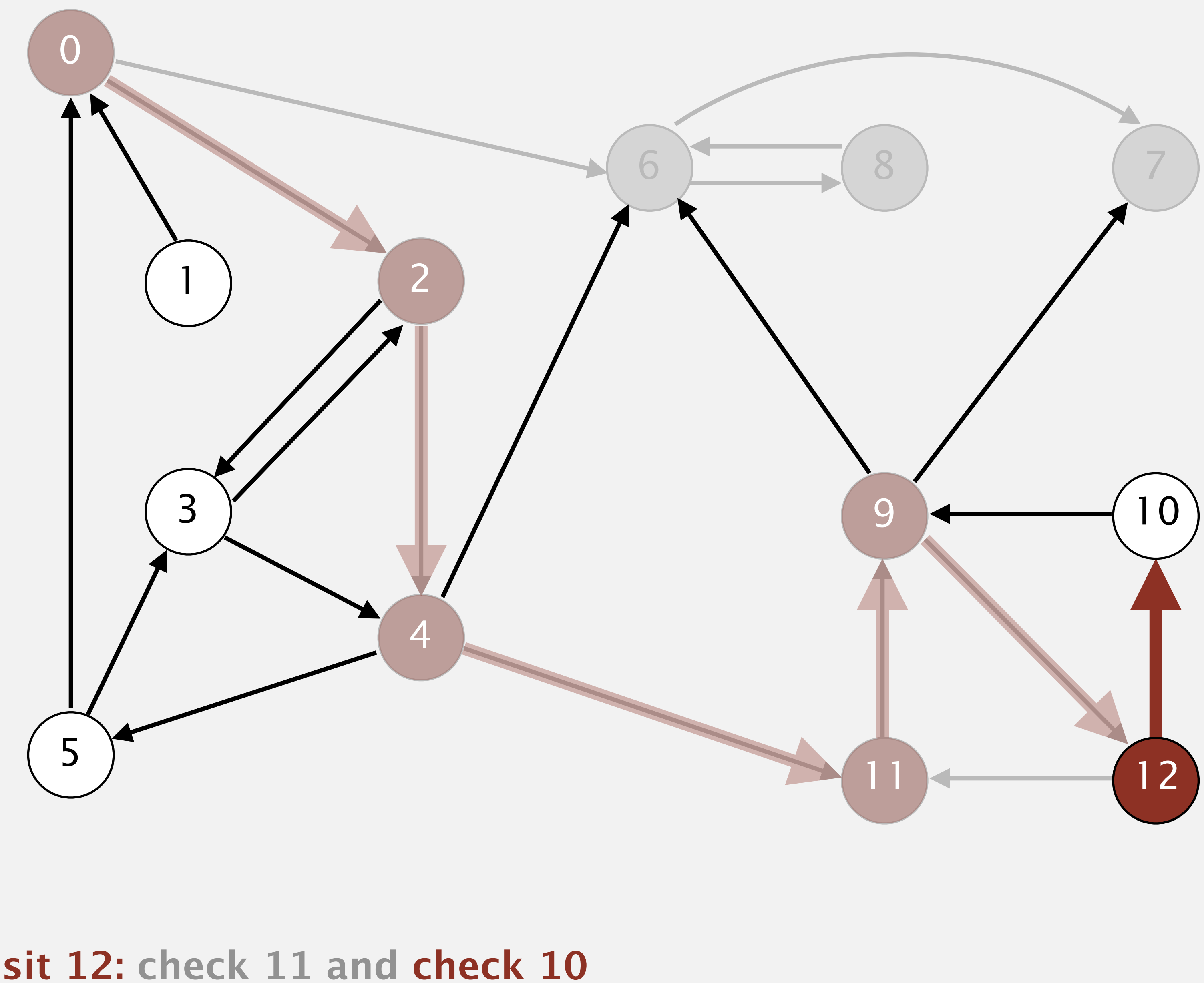
v	marked[]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	T
10	F
11	T
12	T

visit 12: check 11 and check 10

Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in G^R.

6 7 8

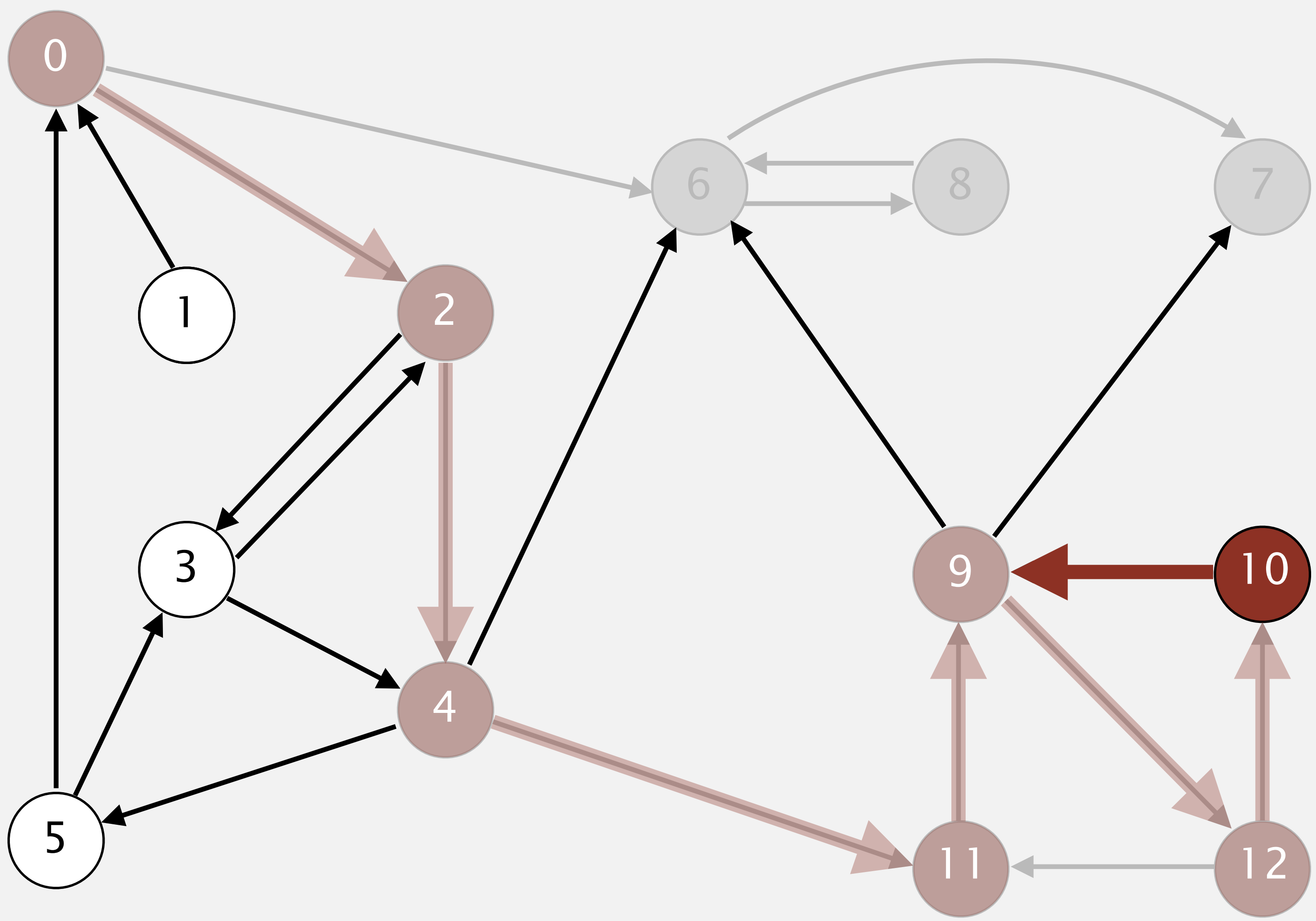


v	marked[]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	T
10	F
11	T
12	T

Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in G^R.

6 7 8



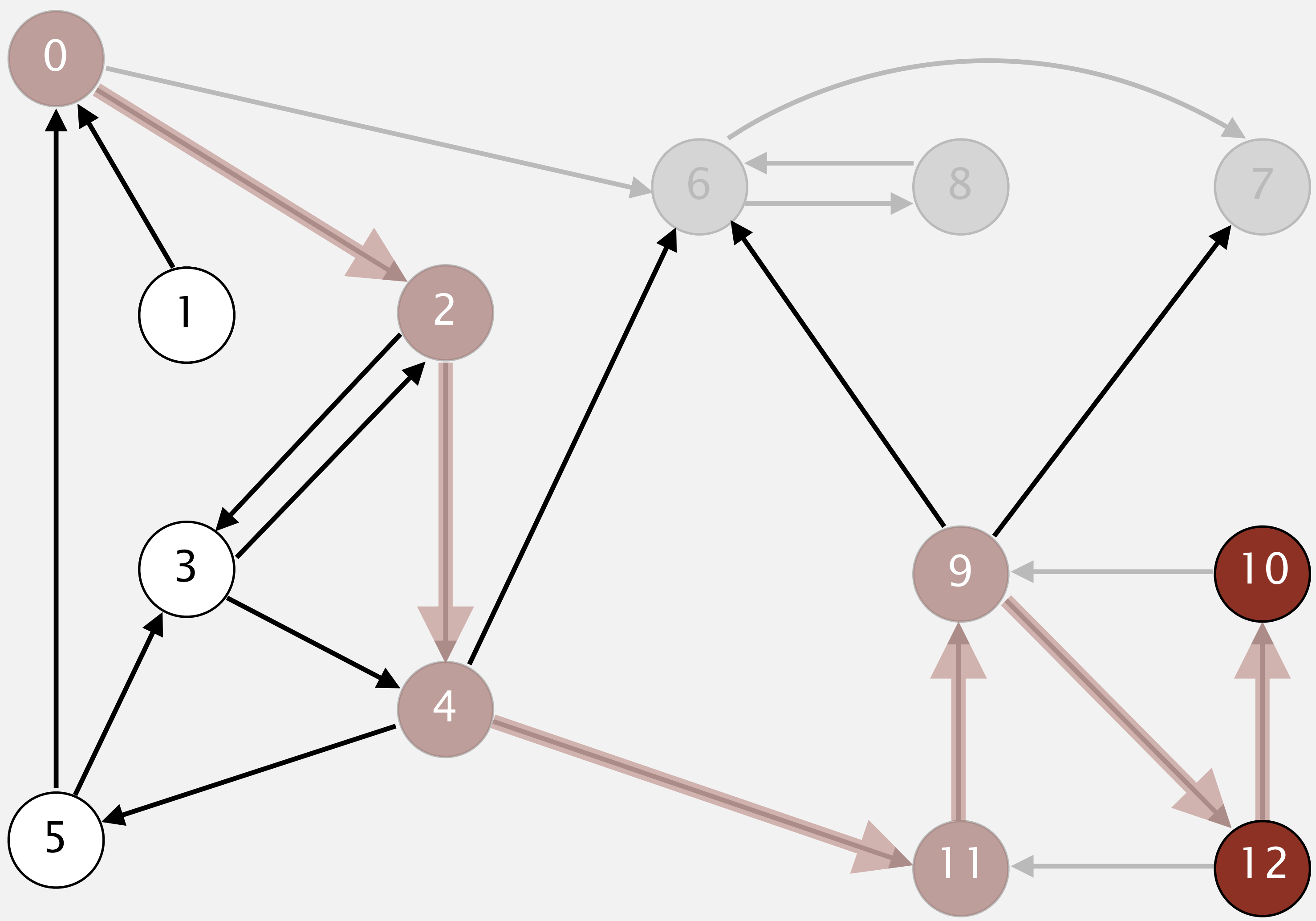
v	marked[]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	T
10	T
11	T
12	T

visit 10: check 9

Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in G^R.

10 6 7 8



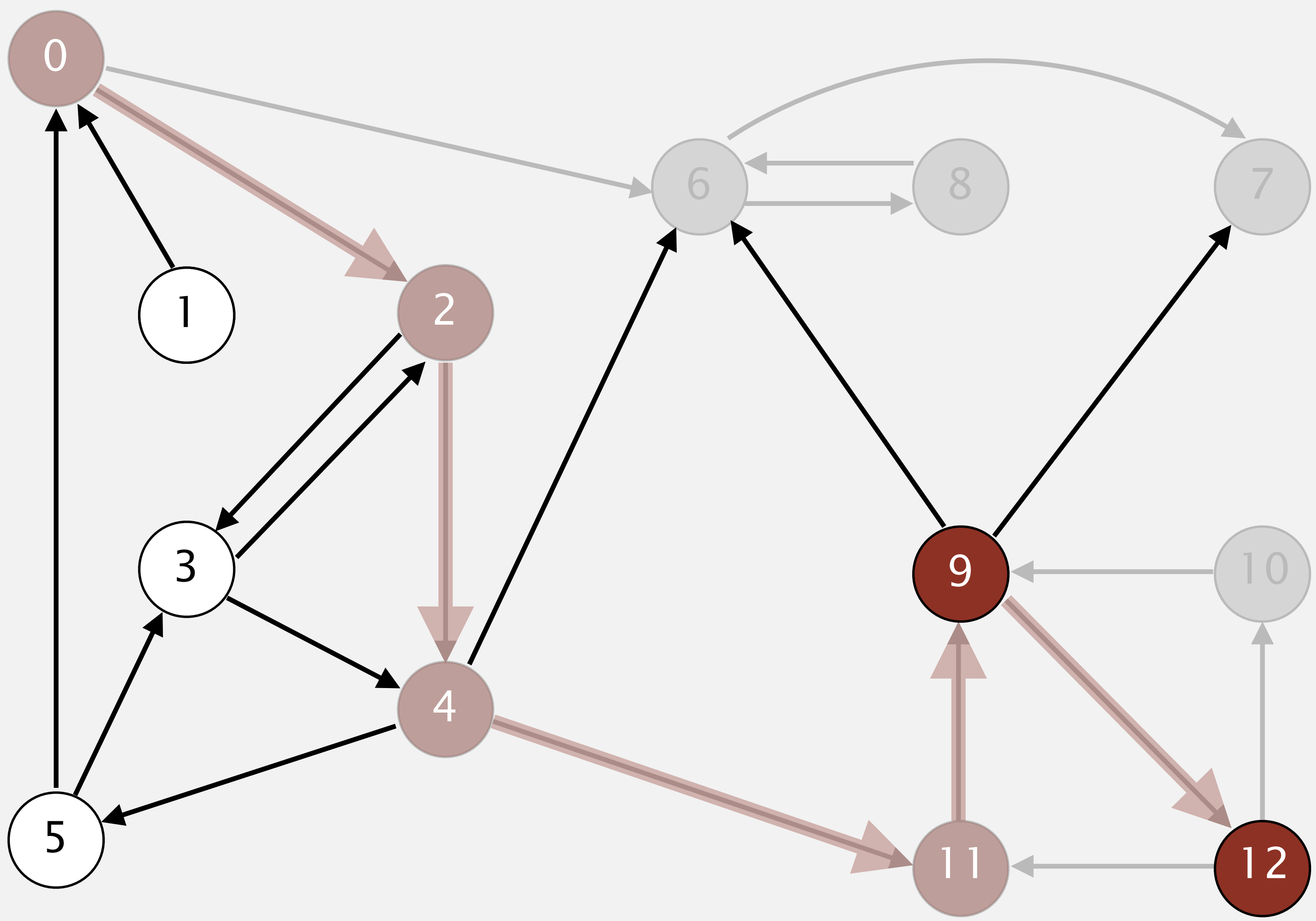
v	marked[]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	T
10	T
11	T
12	T

10 done

Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in G^R.

12 10 6 7 8



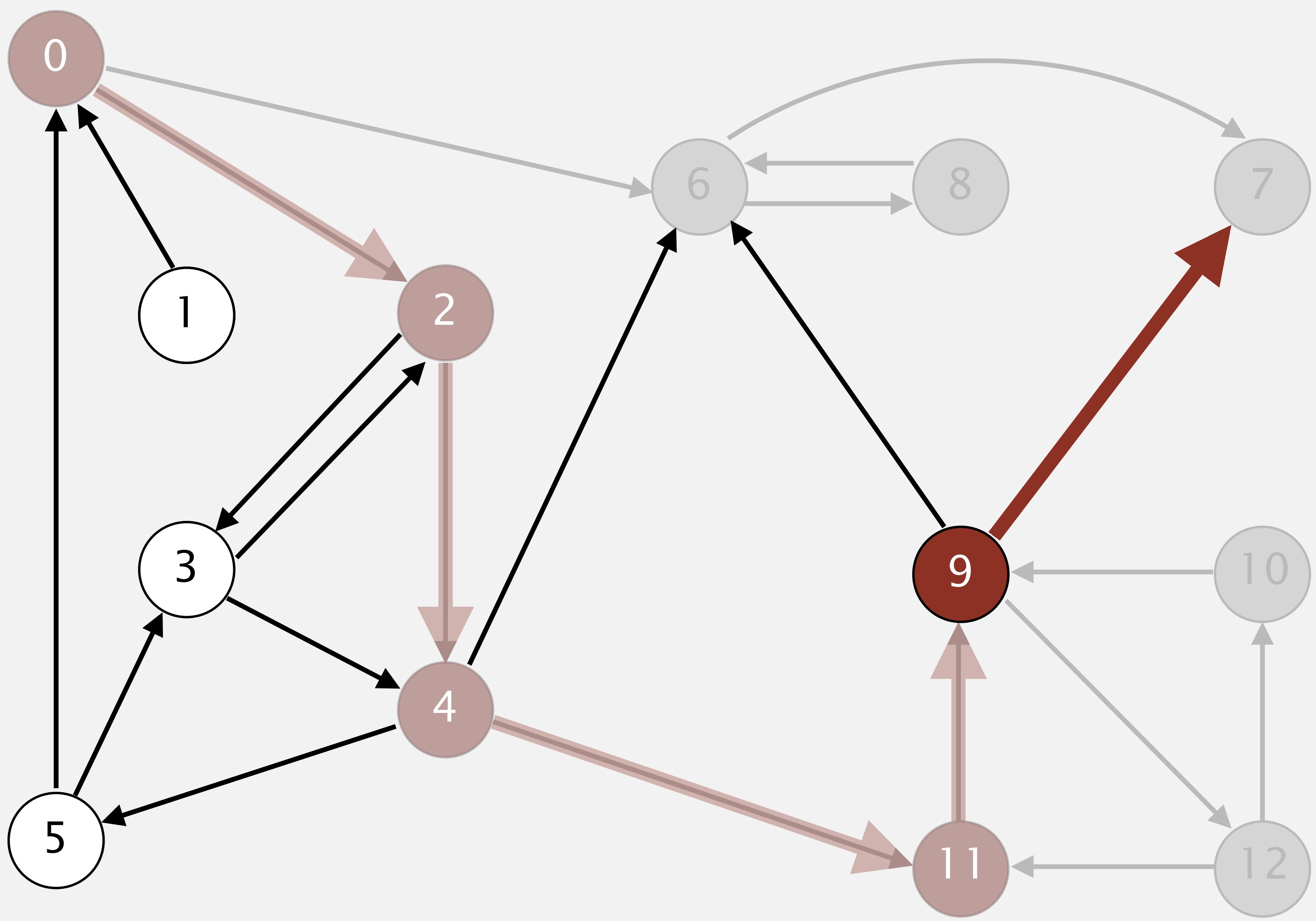
v	marked[]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	T
10	T
11	T
12	T

12 done

Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in G^R.

12 10 6 7 8



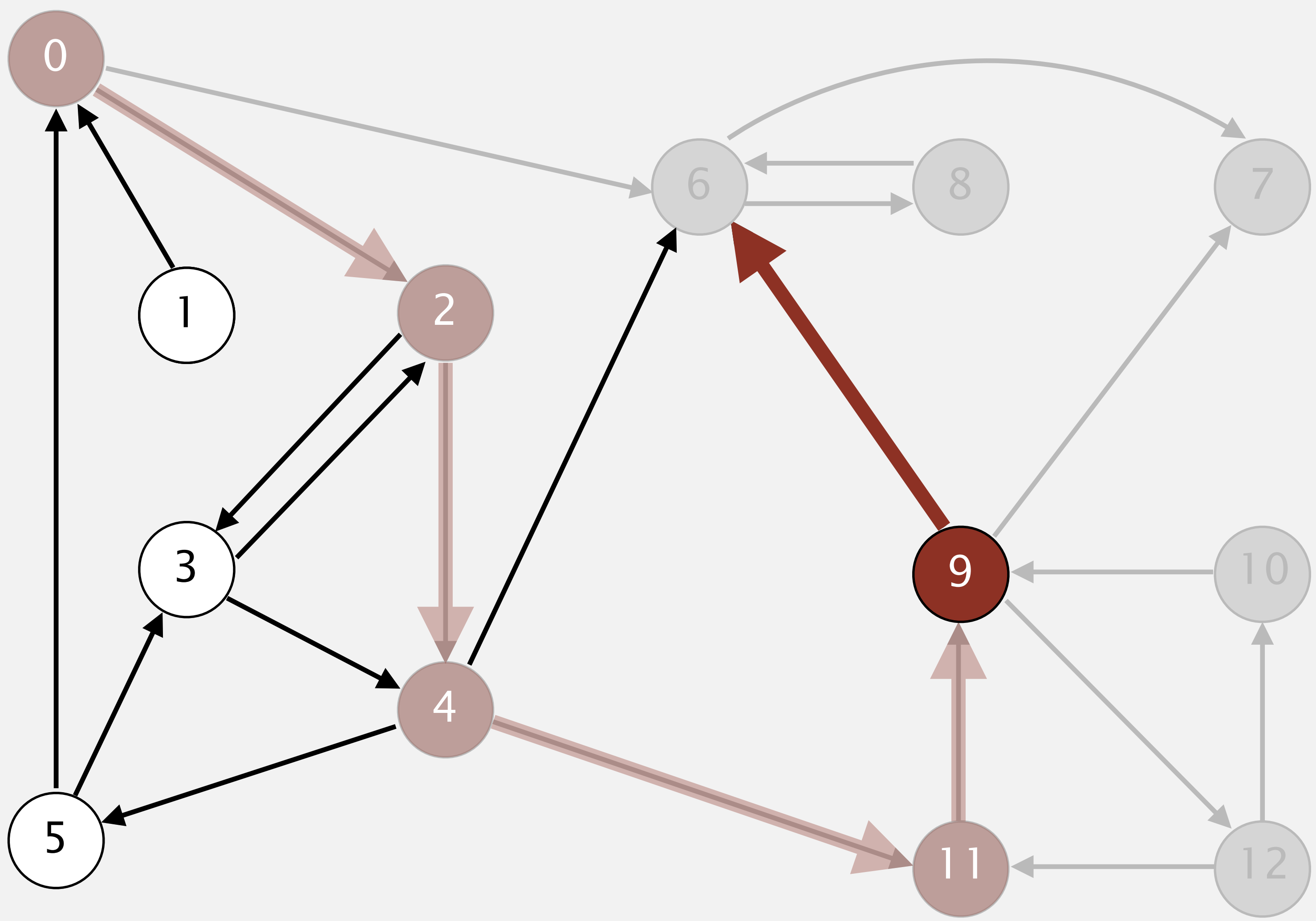
v	marked[]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	T
10	T
11	T
12	T

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



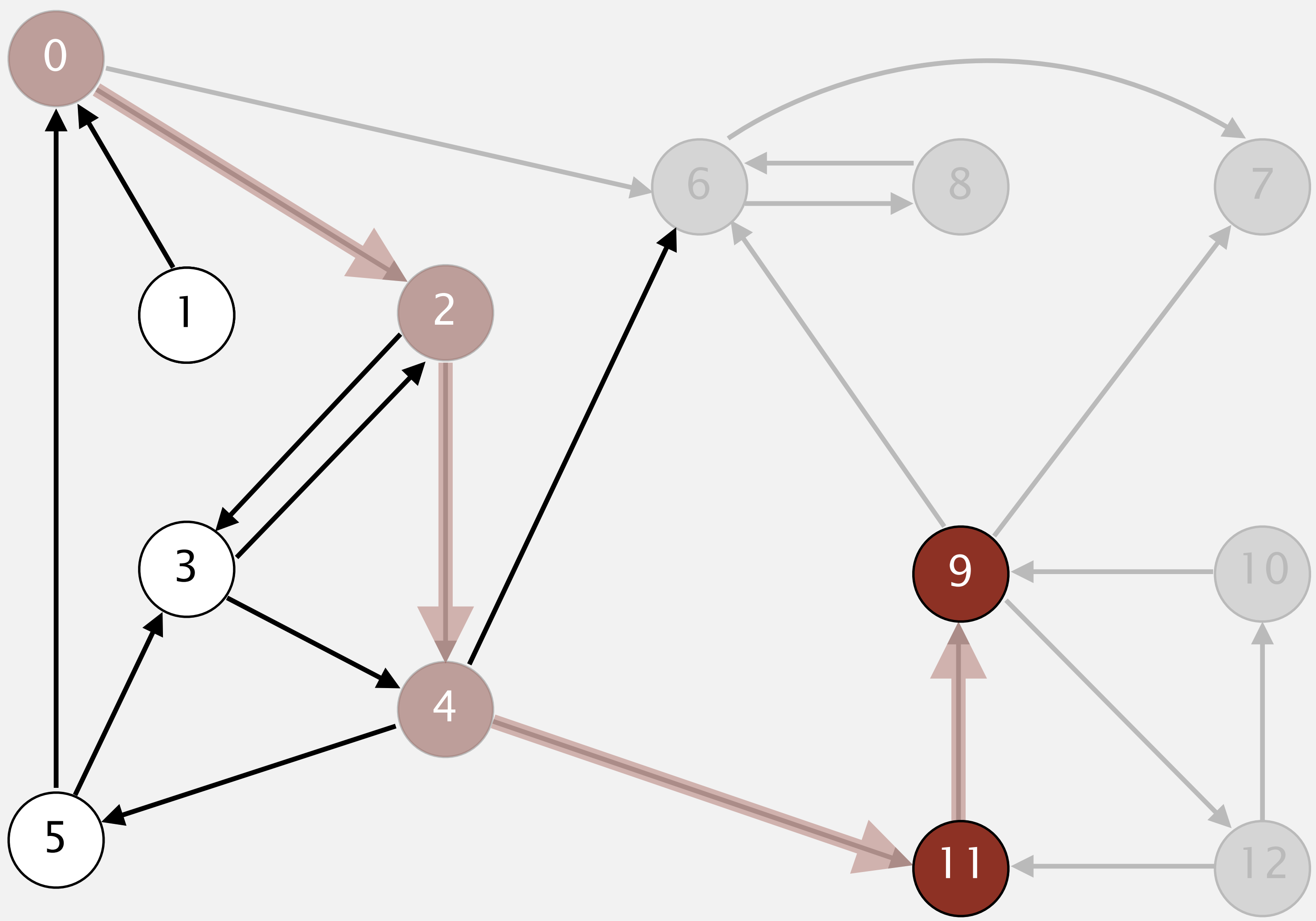
v	marked[]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	T
10	T
11	T
12	T

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



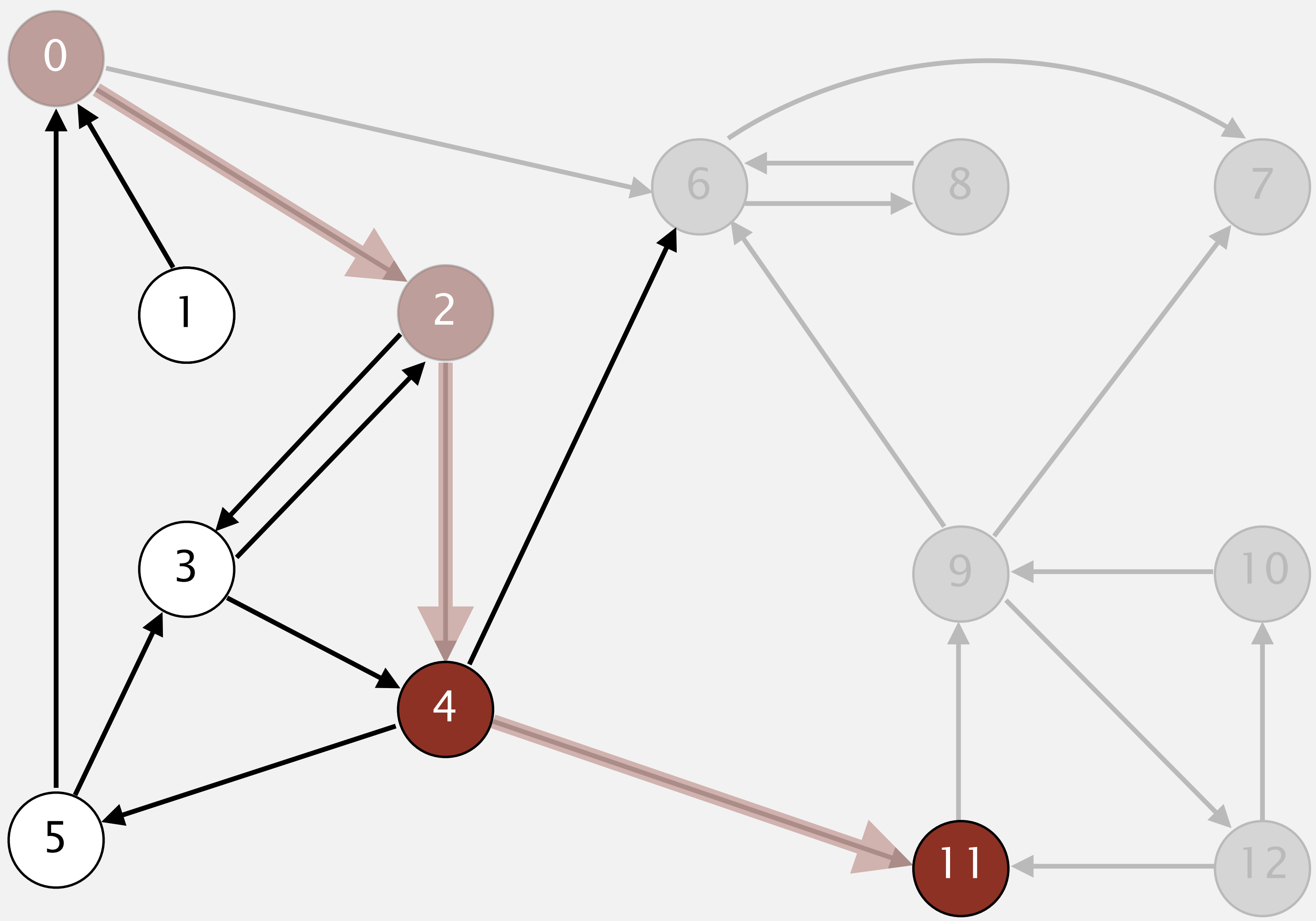
v	marked[]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	T
10	T
11	T
12	T

9 done

Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in G^R.

11 9 12 10 6 7 8



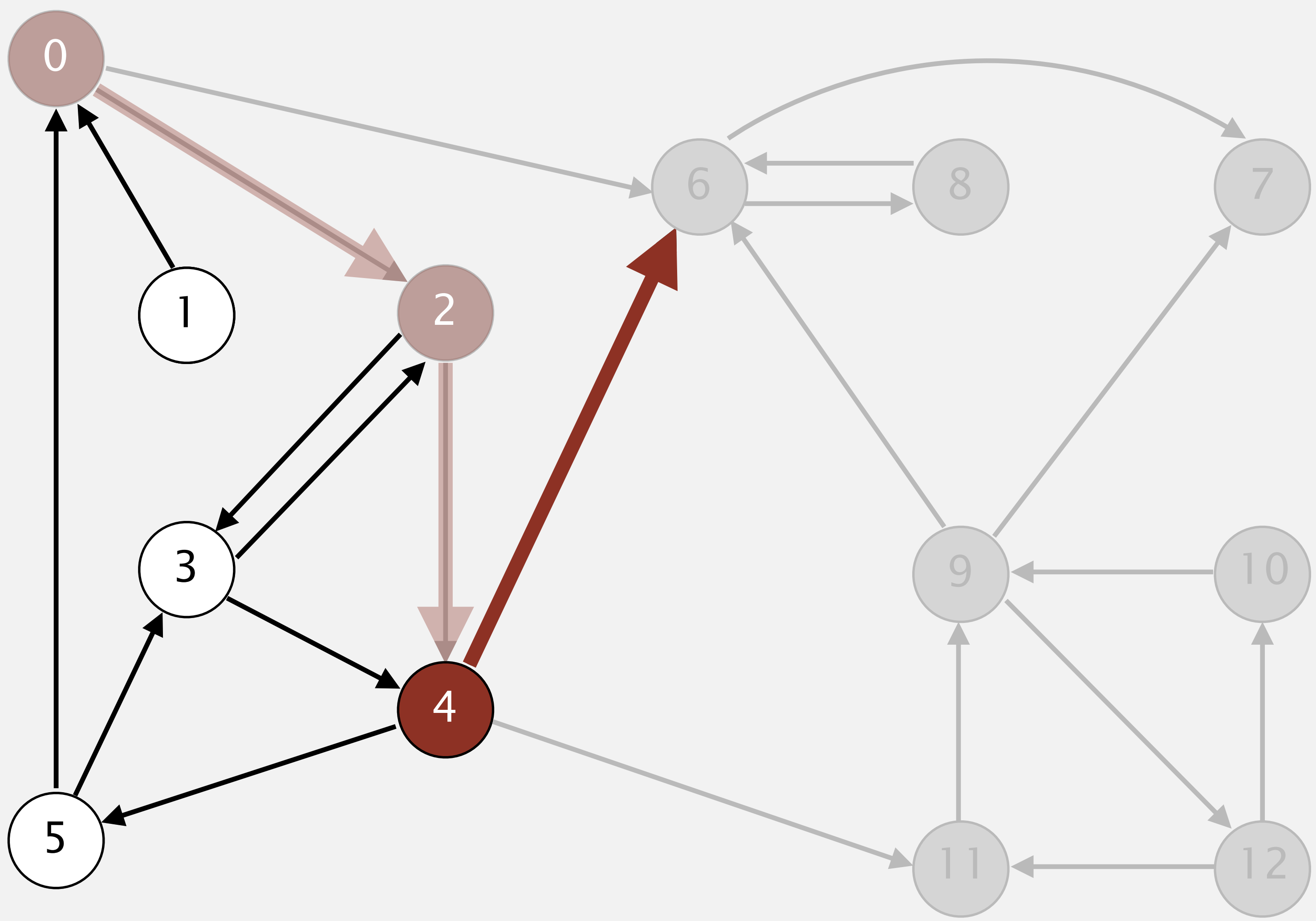
v	marked[]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	T
10	T
11	T
12	T

11 done

Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in G^R.

11 9 12 10 6 7 8



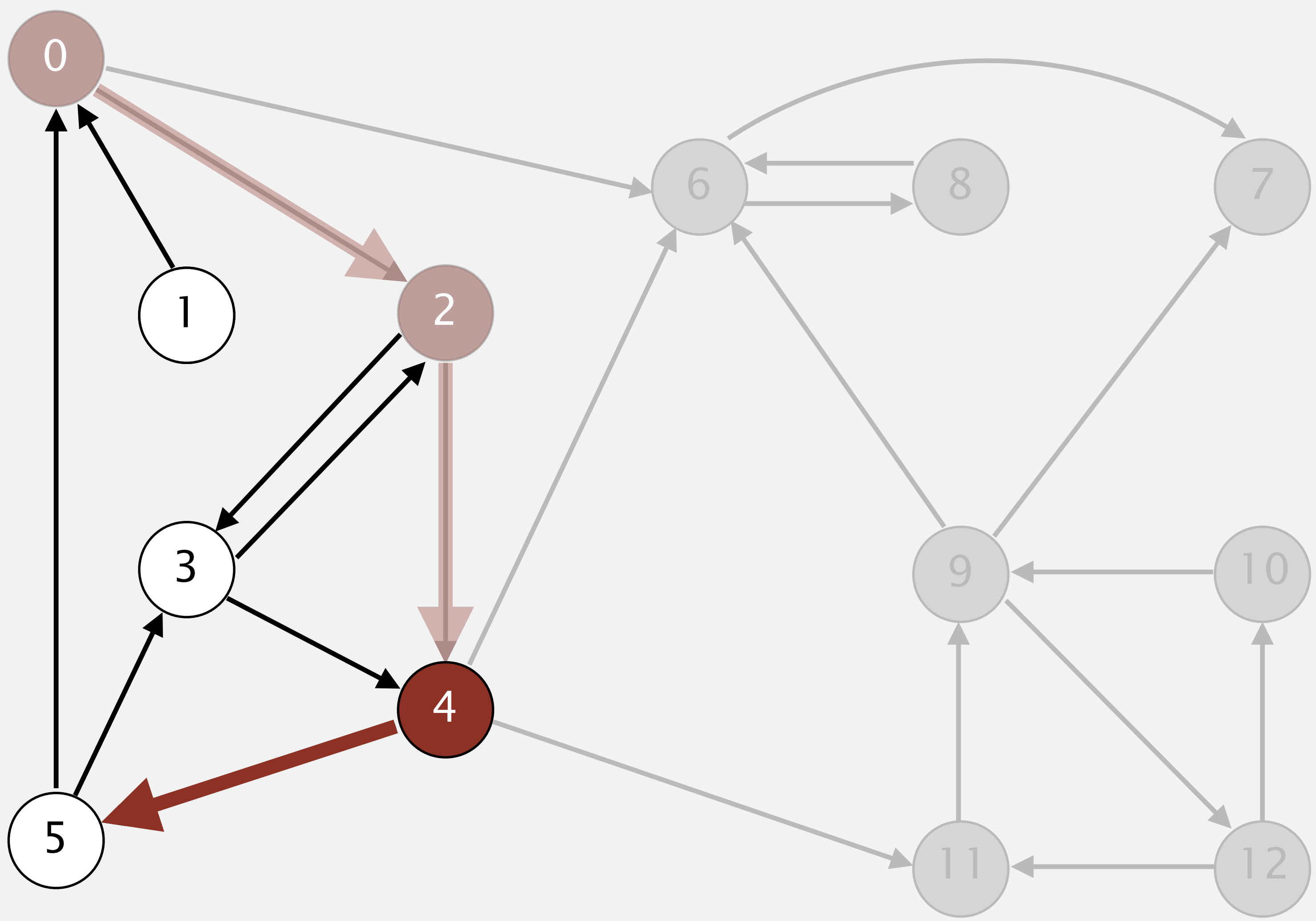
v	marked[]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	T
10	T
11	T
12	T

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



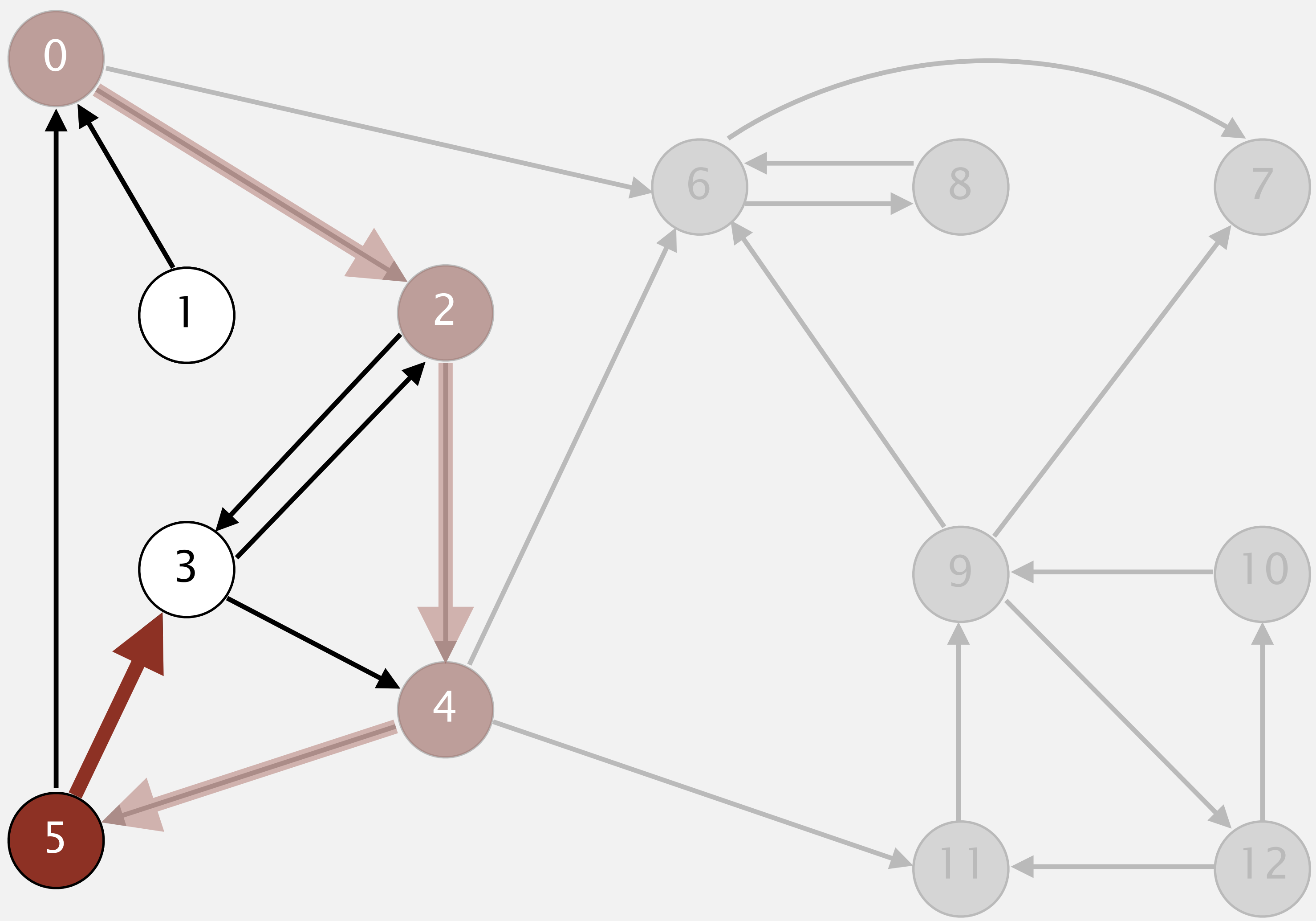
v	marked[]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	T
10	T
11	T
12	T

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



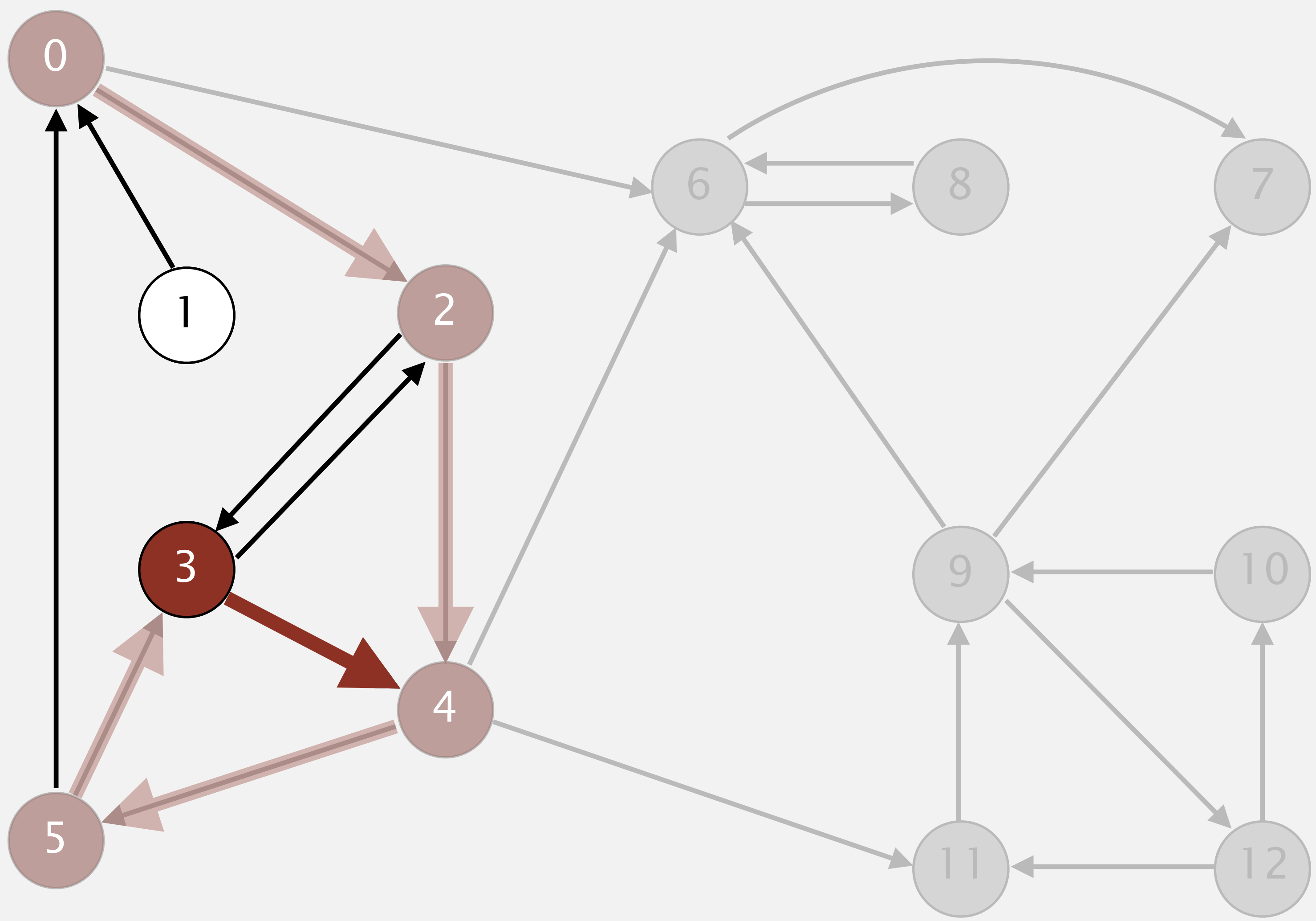
v	marked[]
0	T
1	F
2	T
3	F
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

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



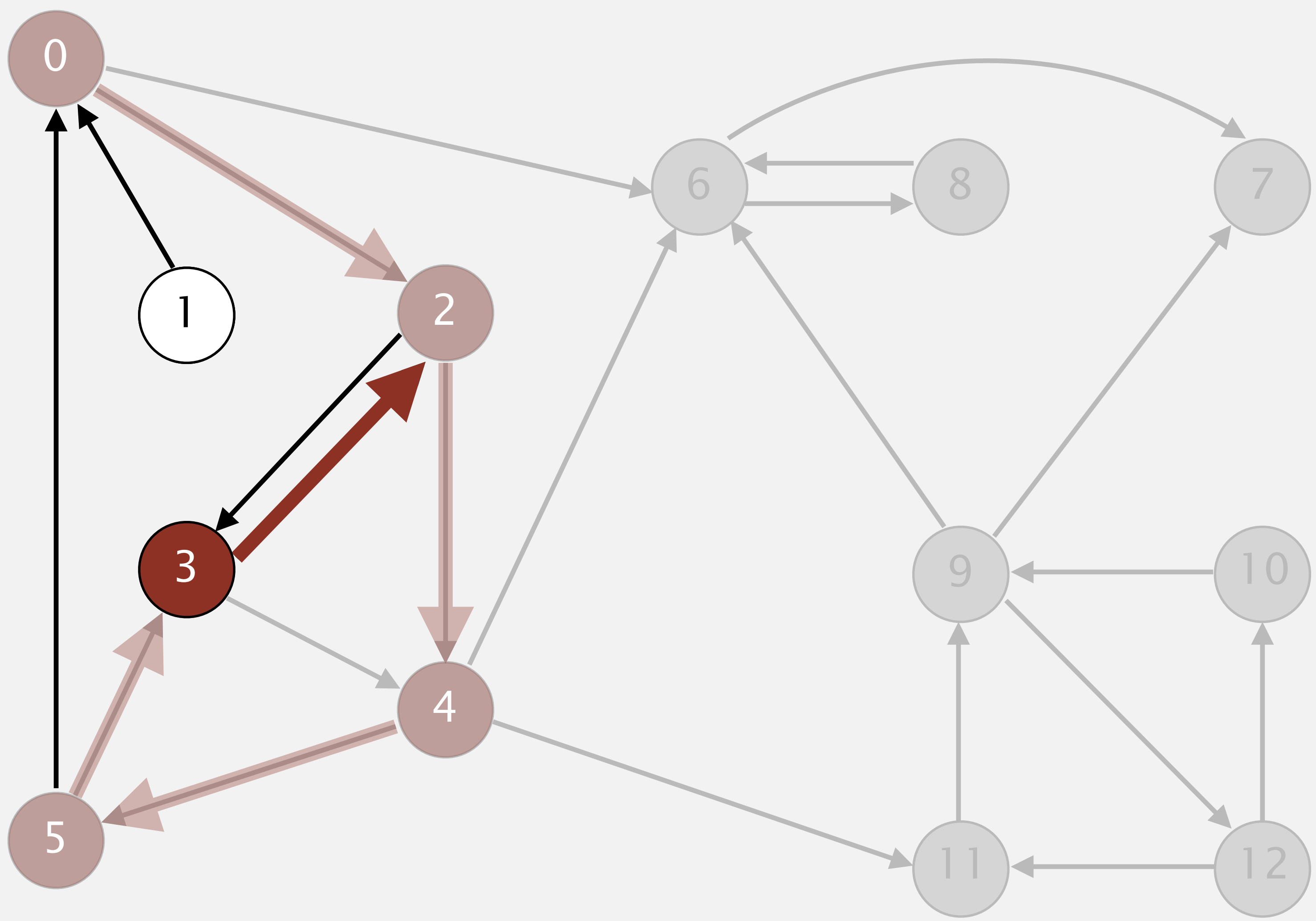
v	marked[]
0	T
1	F
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

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



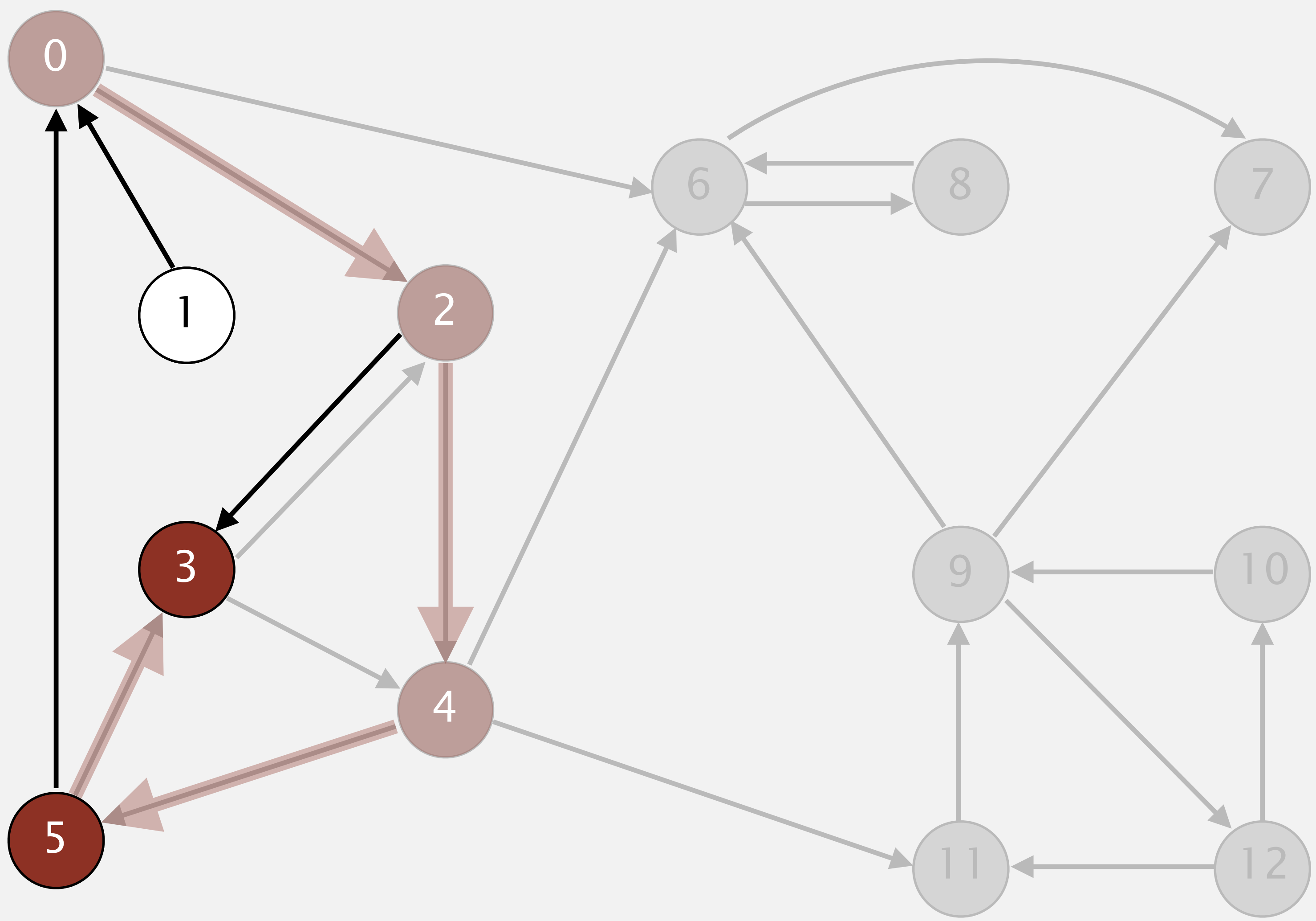
v	marked[]
0	T
1	F
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

visit 3: check 4 and **check 2**

Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in G^R.

3 11 9 12 10 6 7 8



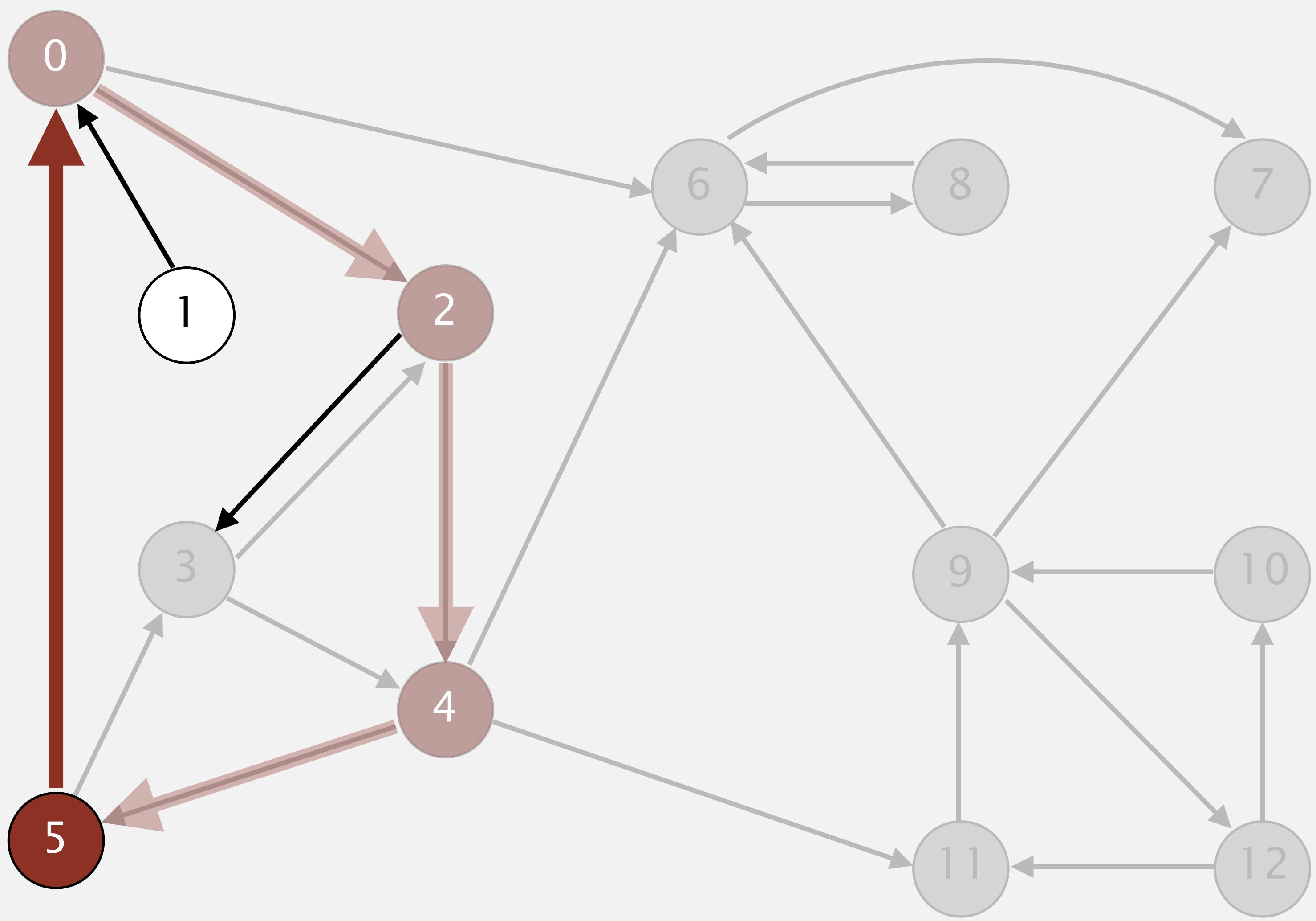
v	marked[]
0	T
1	F
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

3 done

Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in G^R.

3 11 9 12 10 6 7 8



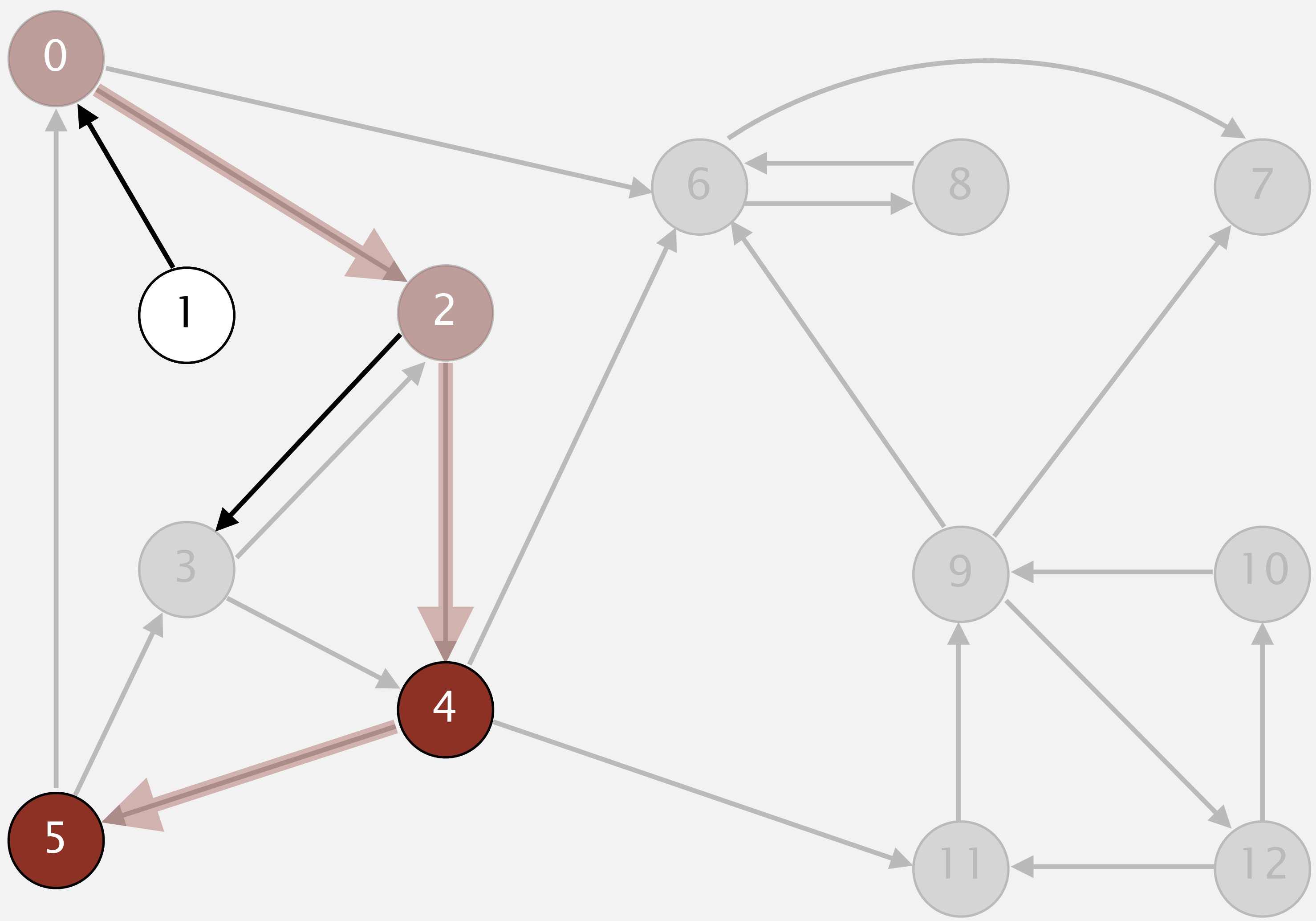
v	marked[]
0	T
1	F
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

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



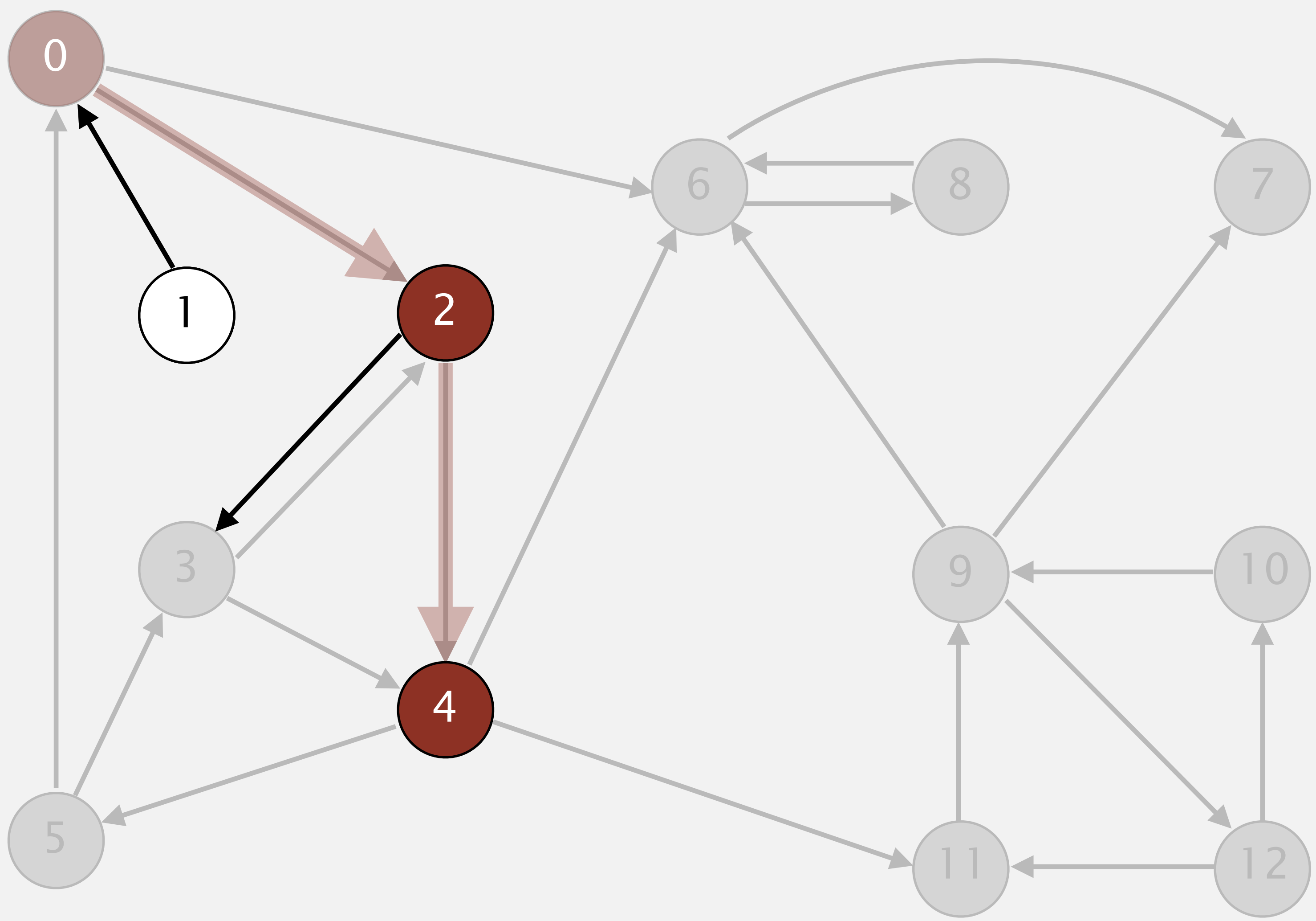
v	marked[]
0	T
1	F
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

5 done

Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in G^R.

4 5 3 11 9 12 10 6 7 8



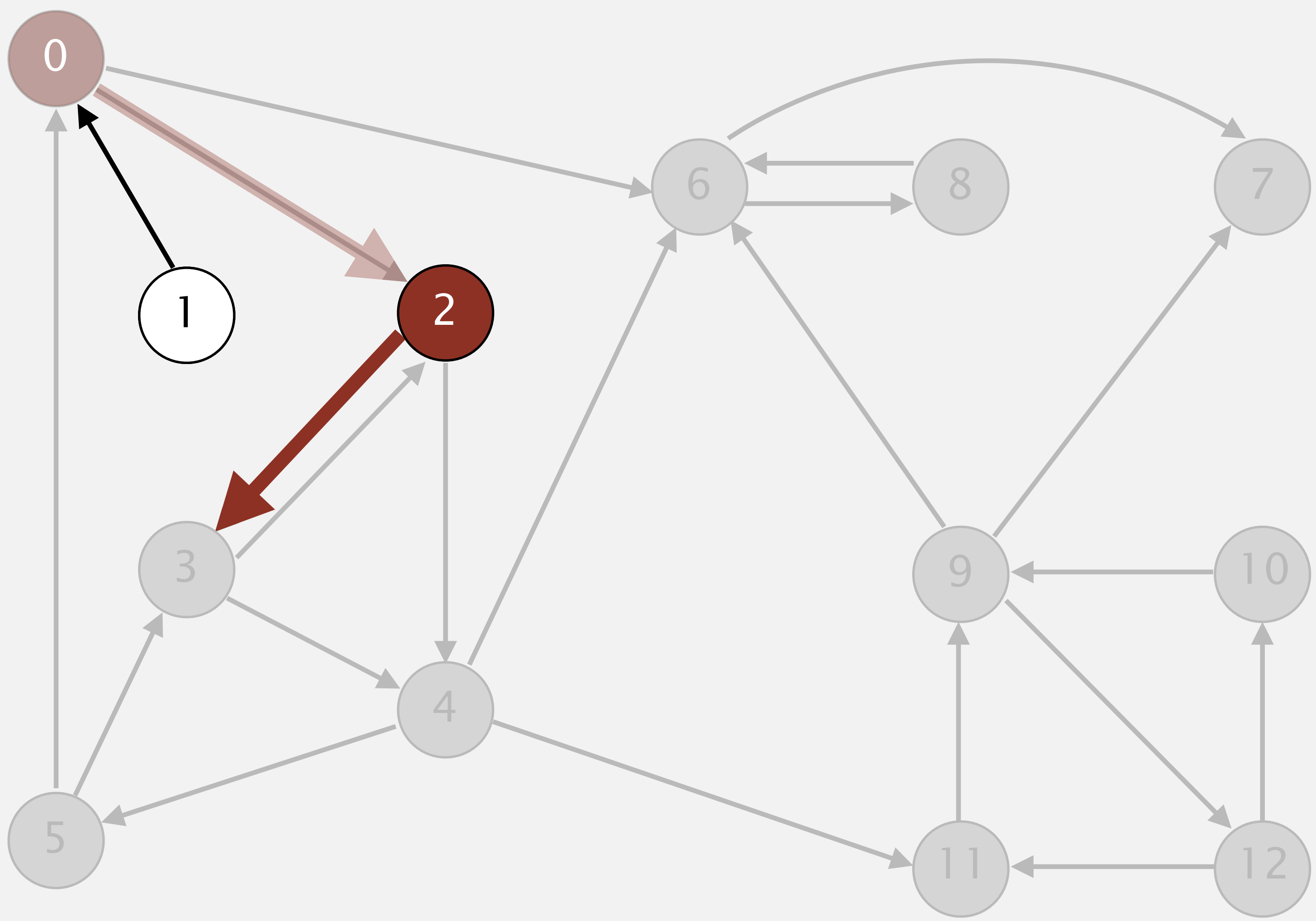
v	marked[]
0	T
1	F
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

4 done

Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in G^R.

4 5 3 11 9 12 10 6 7 8



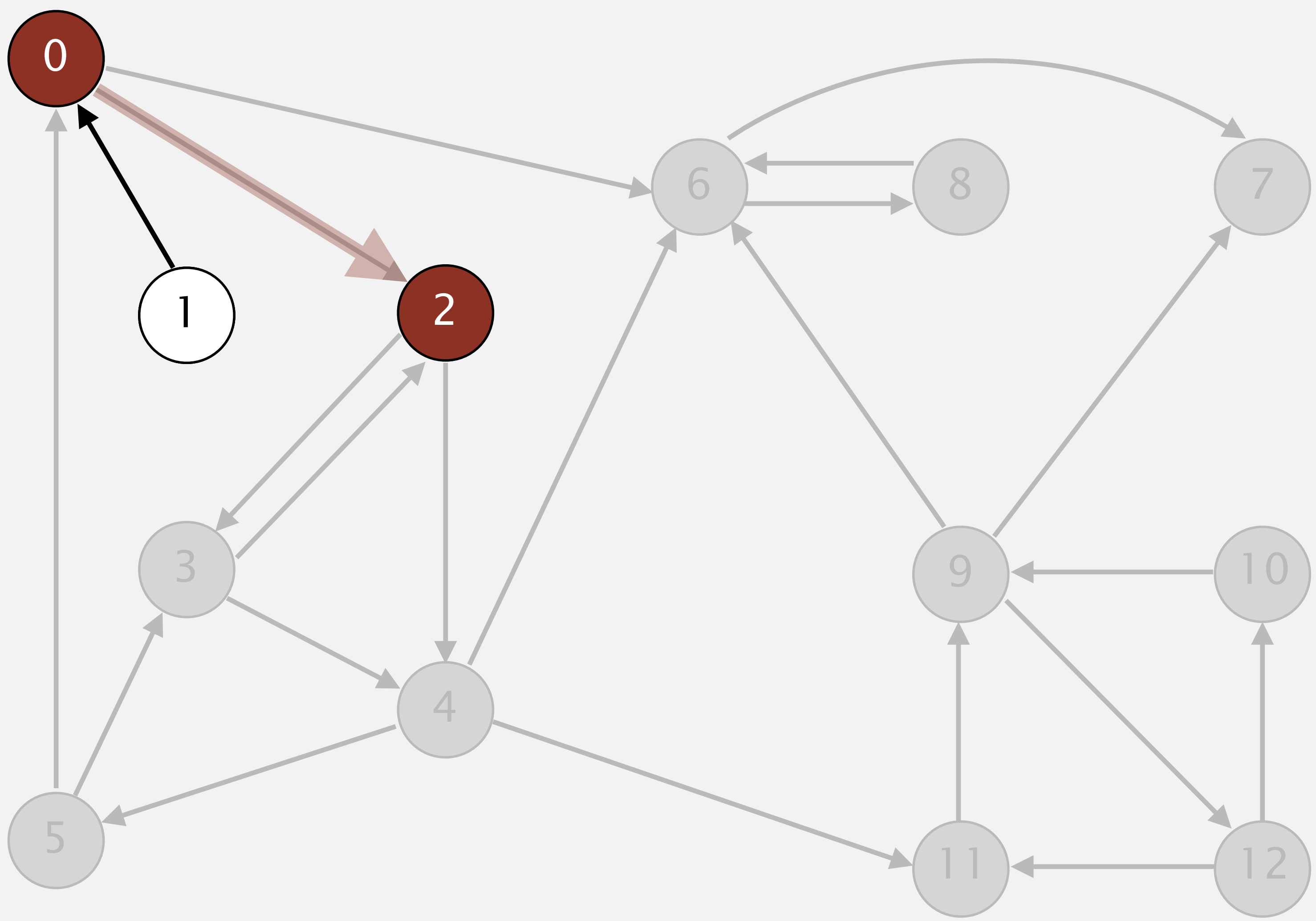
v	marked[]
0	T
1	F
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

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



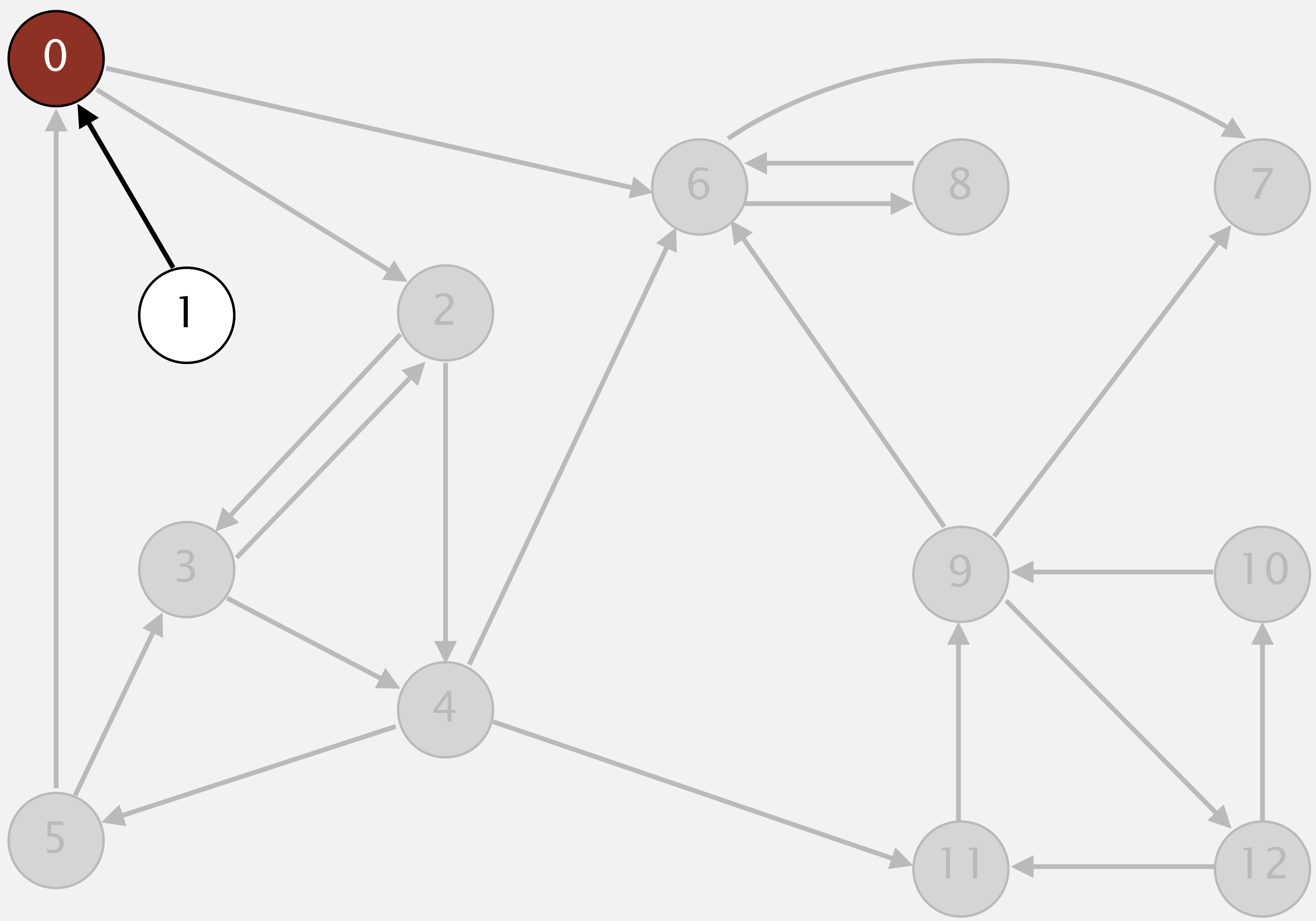
v	marked[]
0	T
1	F
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

2 done

Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in G^R.

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



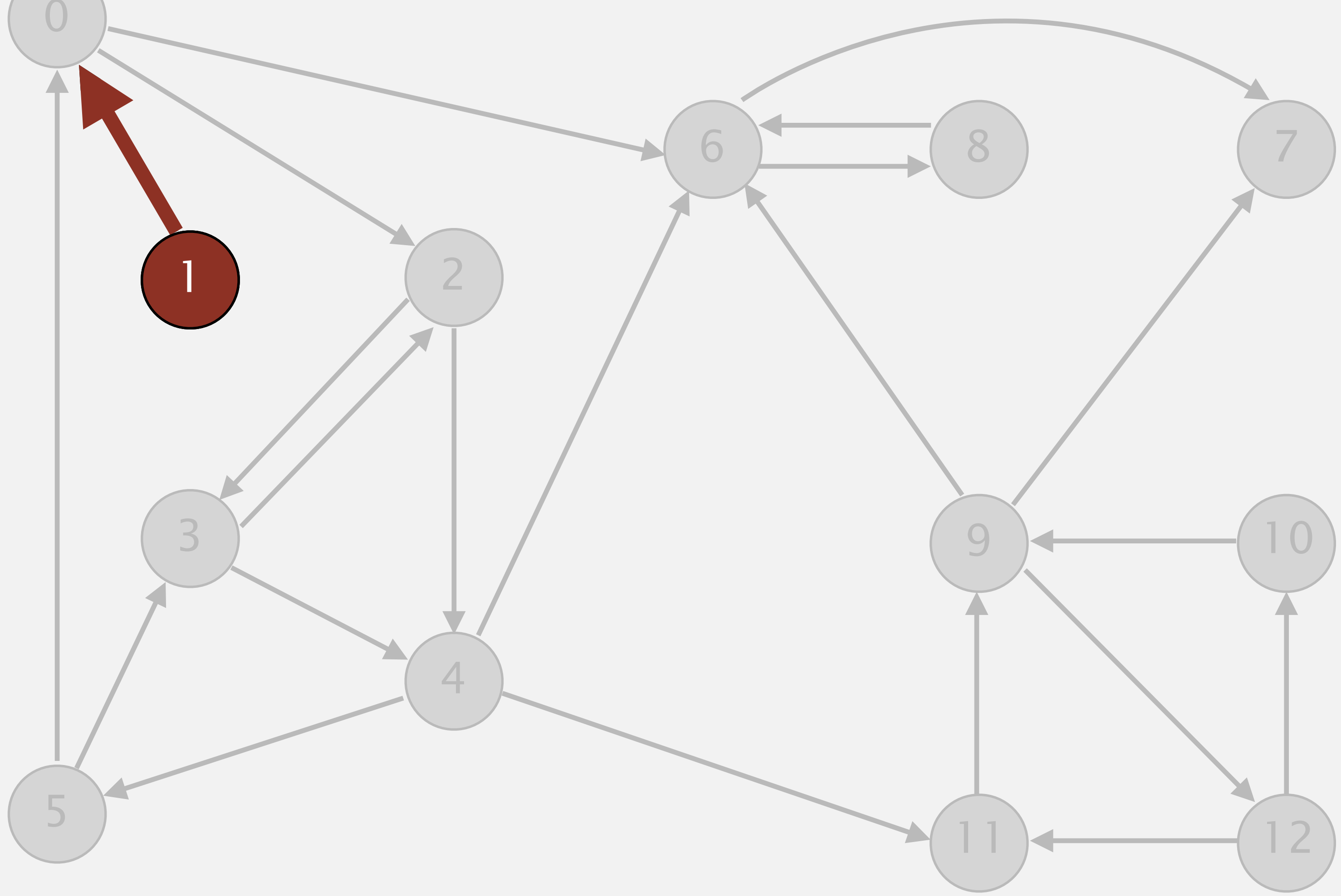
v	marked[]
0	T
1	F
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

0 done

Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in G^R.

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



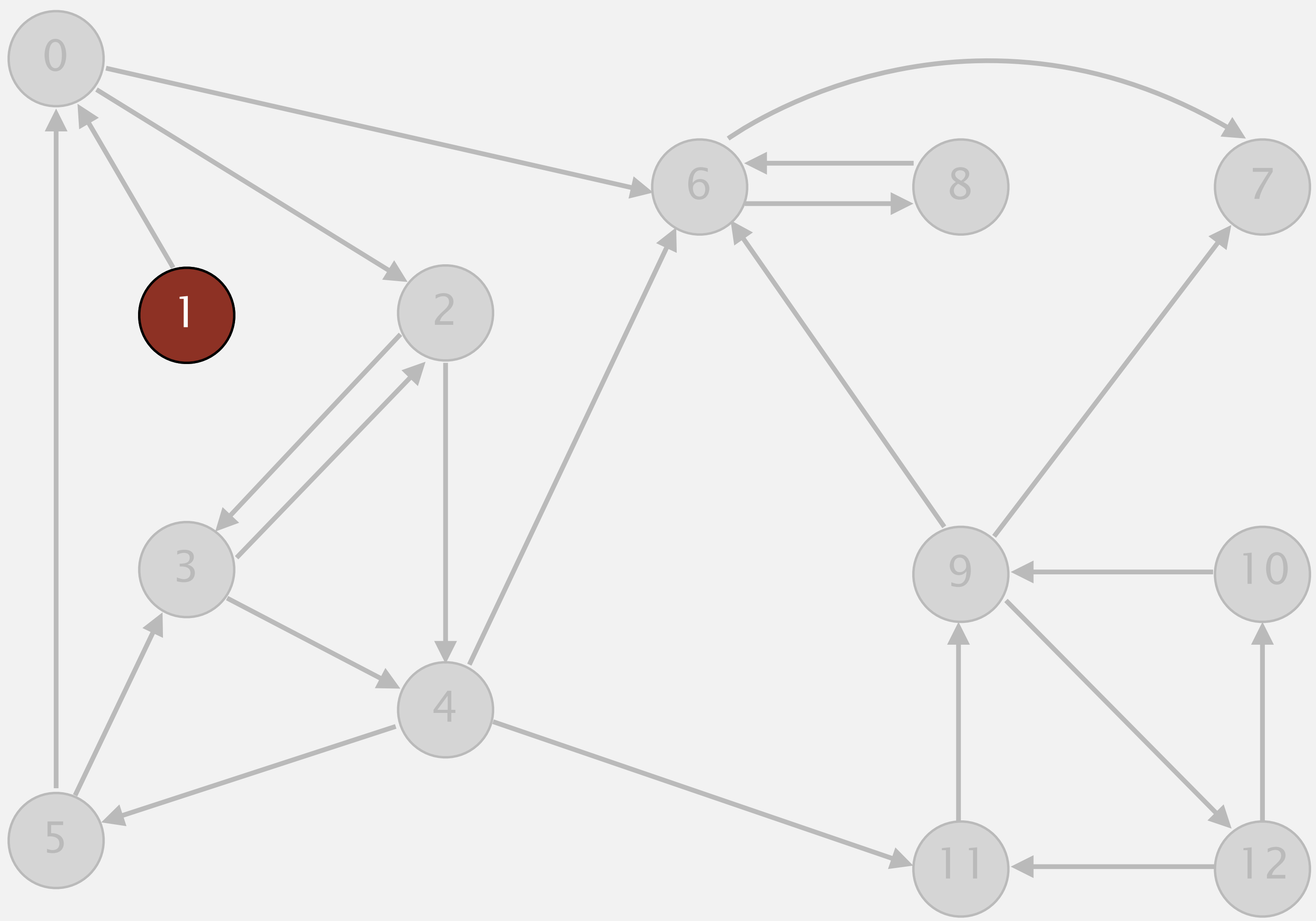
v	marked[]
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

visit 1: check 0

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



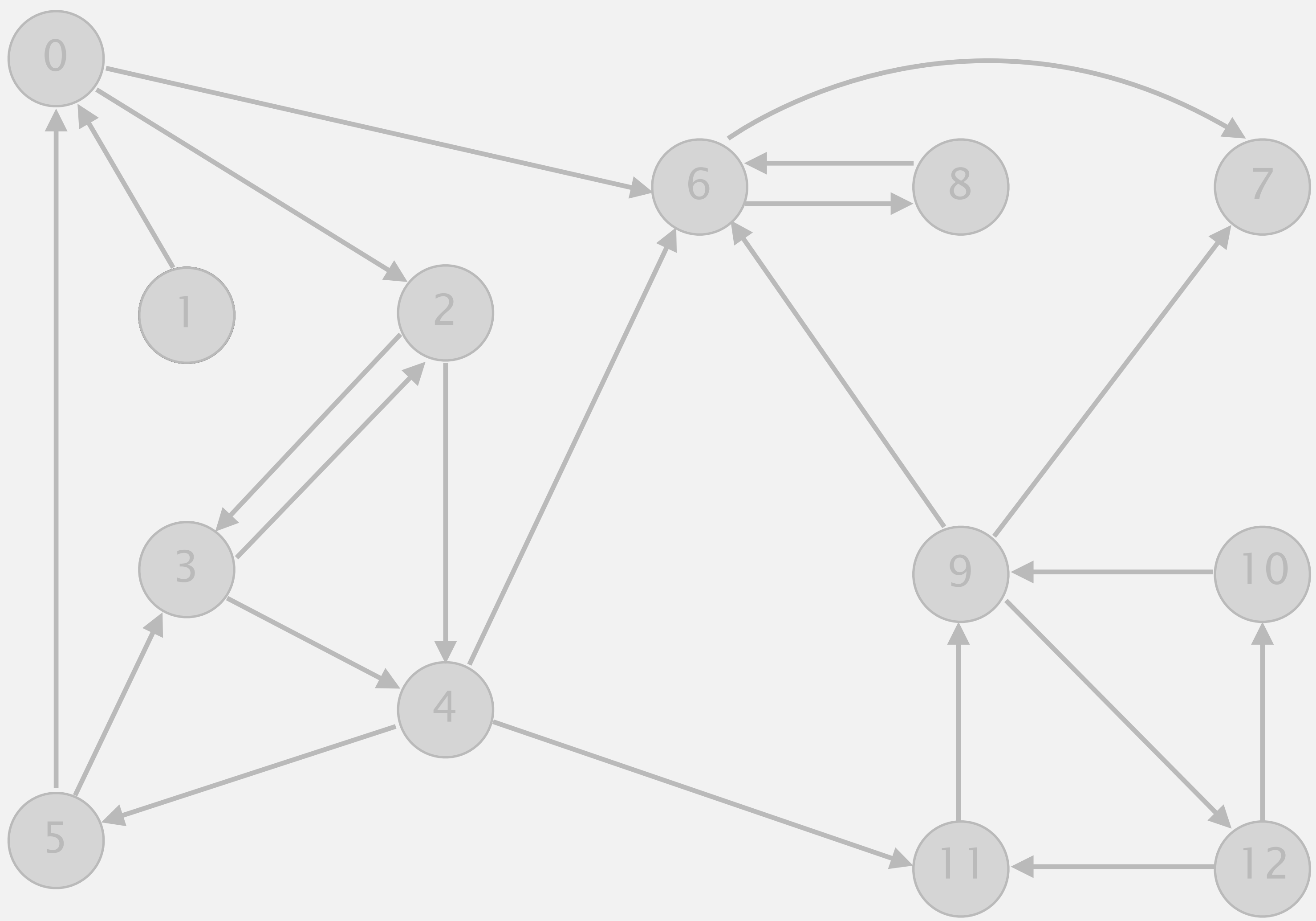
v	marked[]
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

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



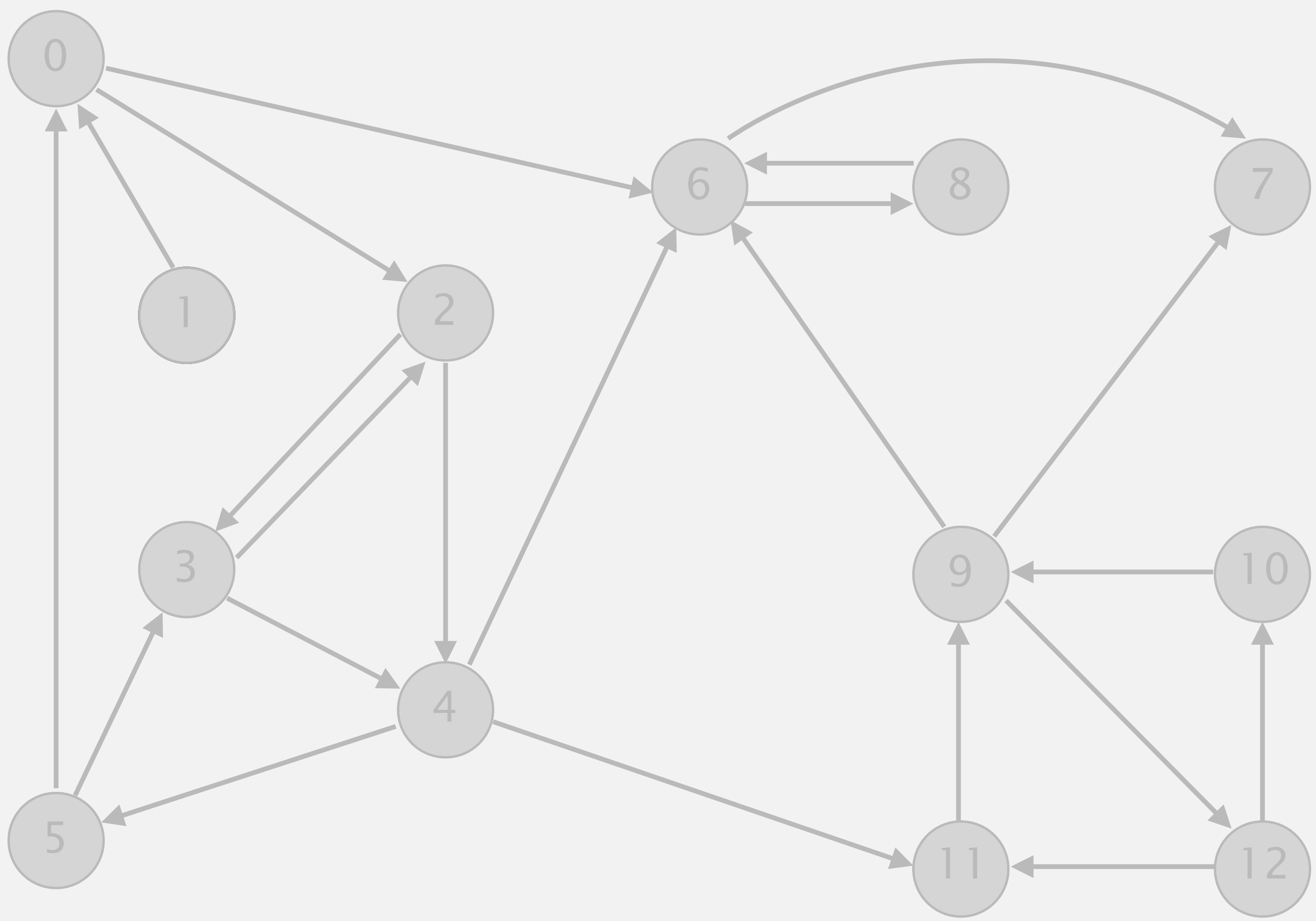
v	marked[]
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

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



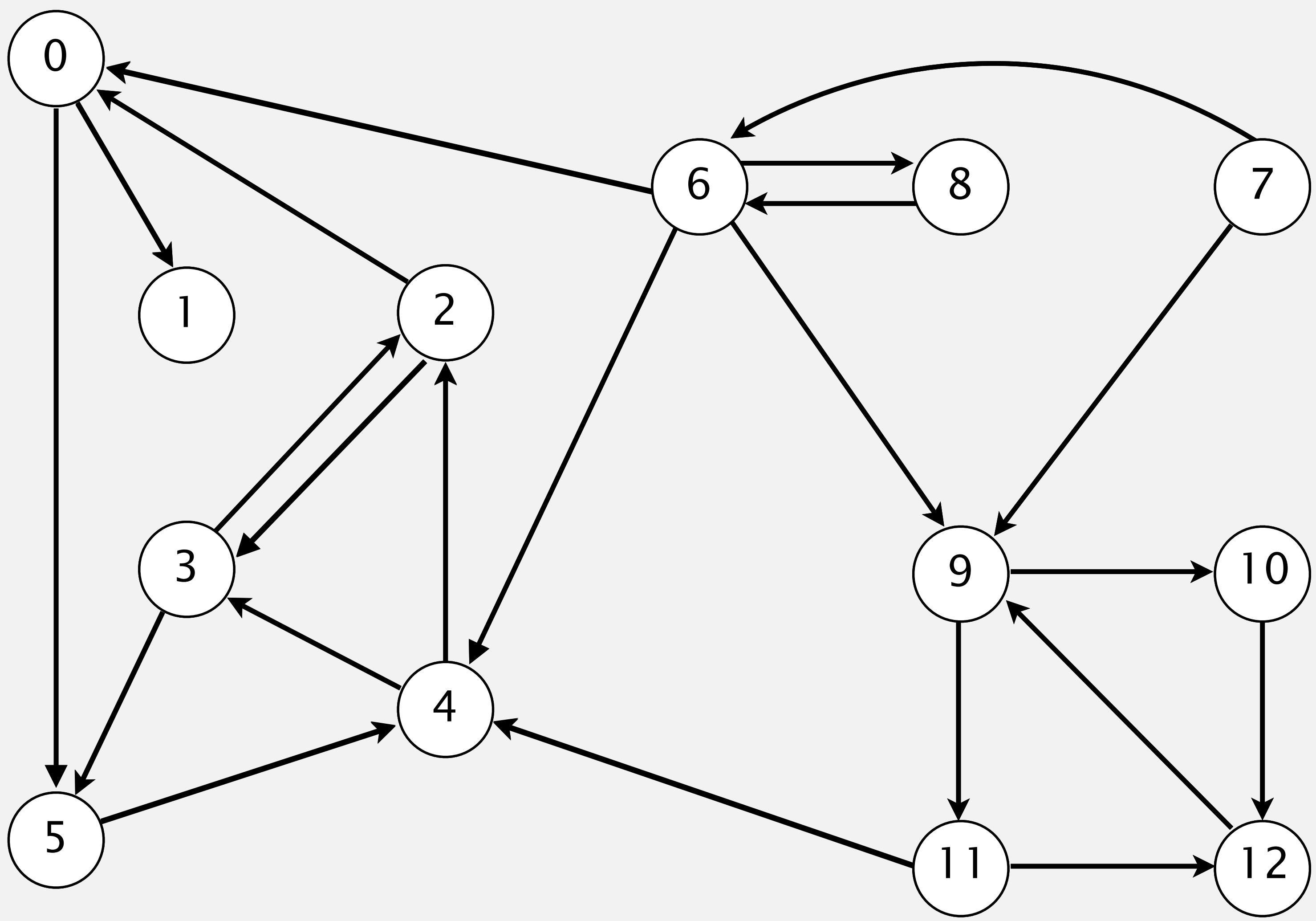
reverse digraph G^R

DFS IN THE ORGINAL GRAPH

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



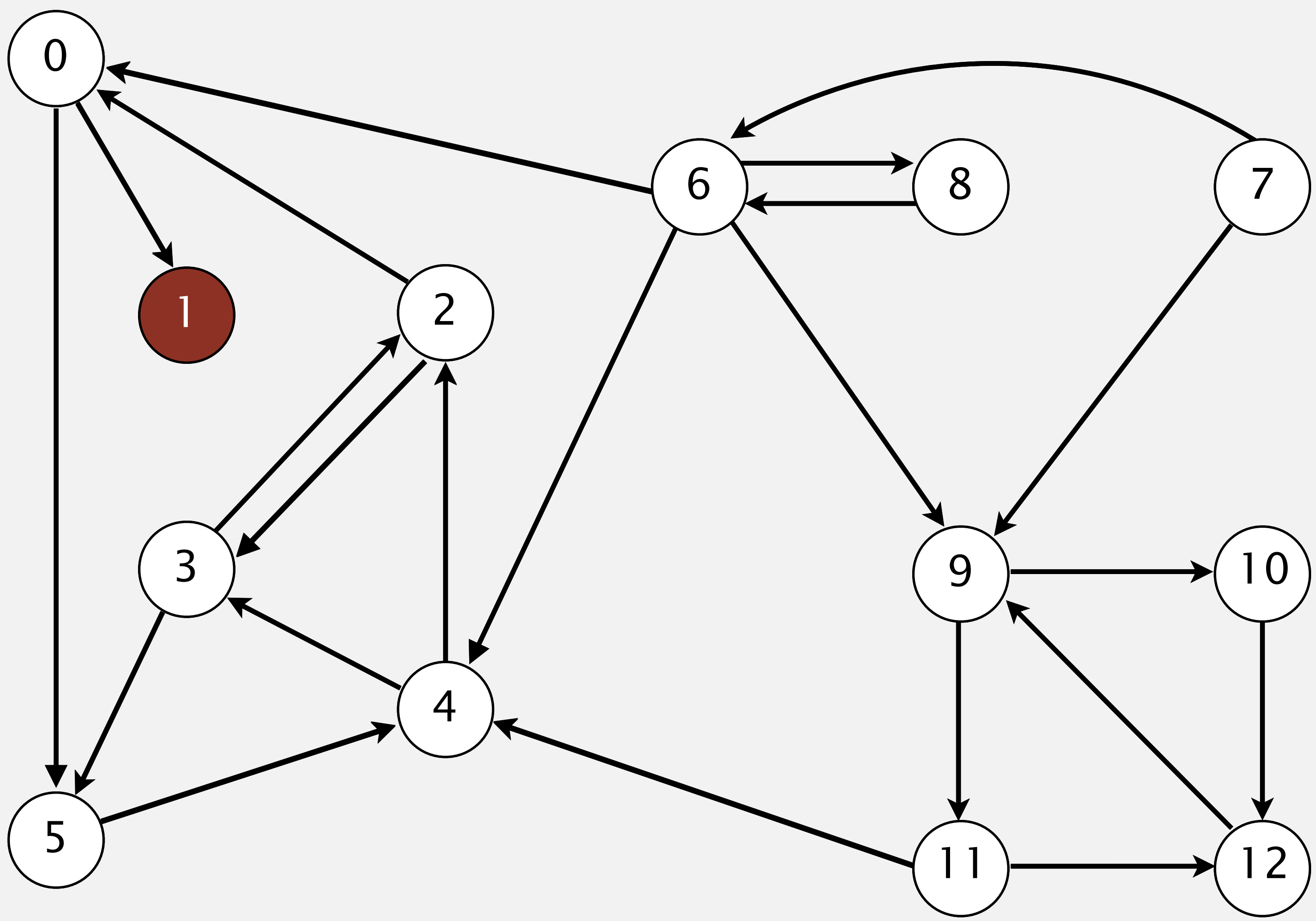
v	id[]
0	–
1	–
2	–
3	–
4	–
5	–
6	–
7	–
8	–
9	–
10	–
11	–
12	–

original digraph G

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



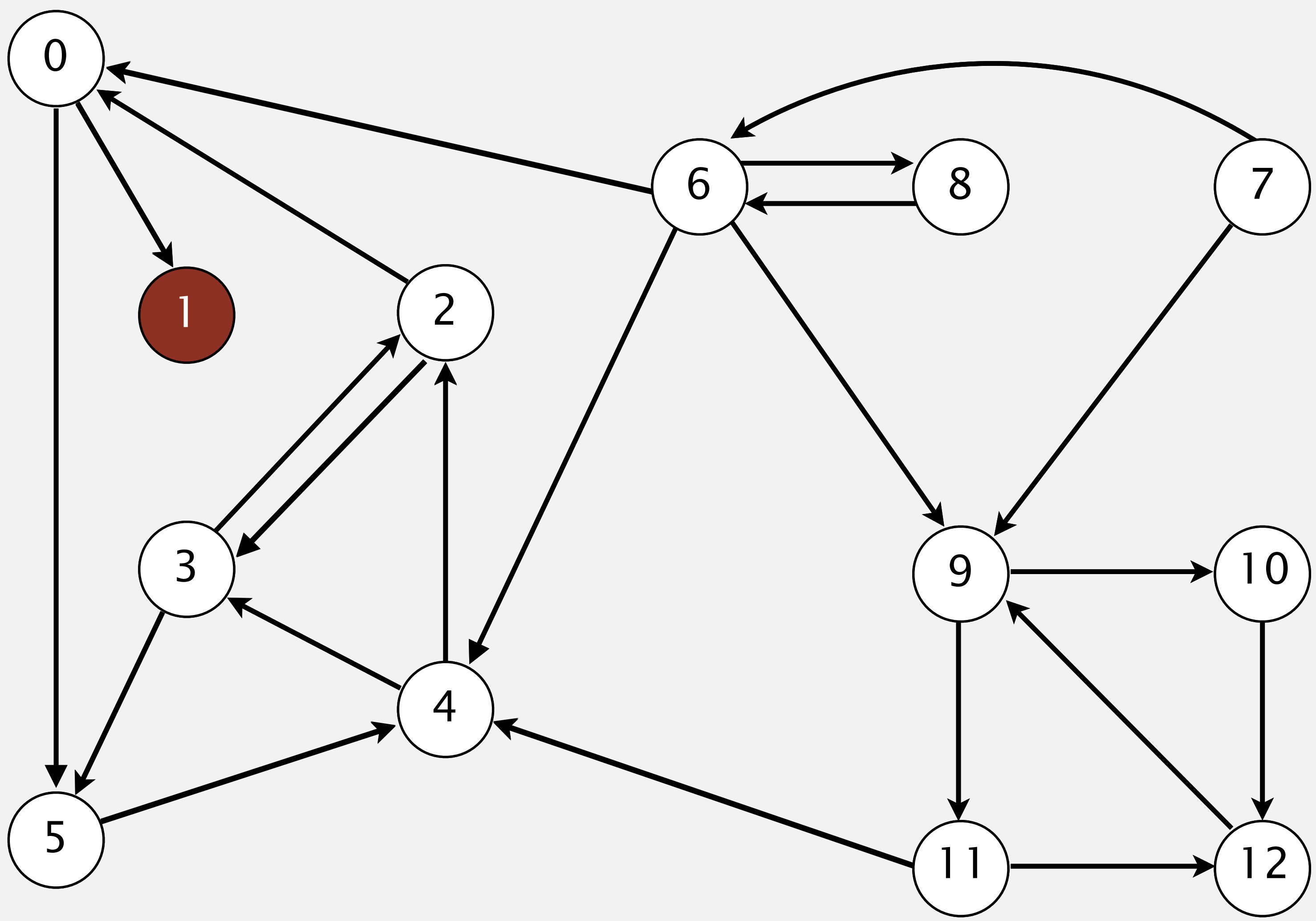
v	id[]
0	–
1	0
2	–
3	–
4	–
5	–
6	–
7	–
8	–
9	–
10	–
11	–
12	–

visit 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



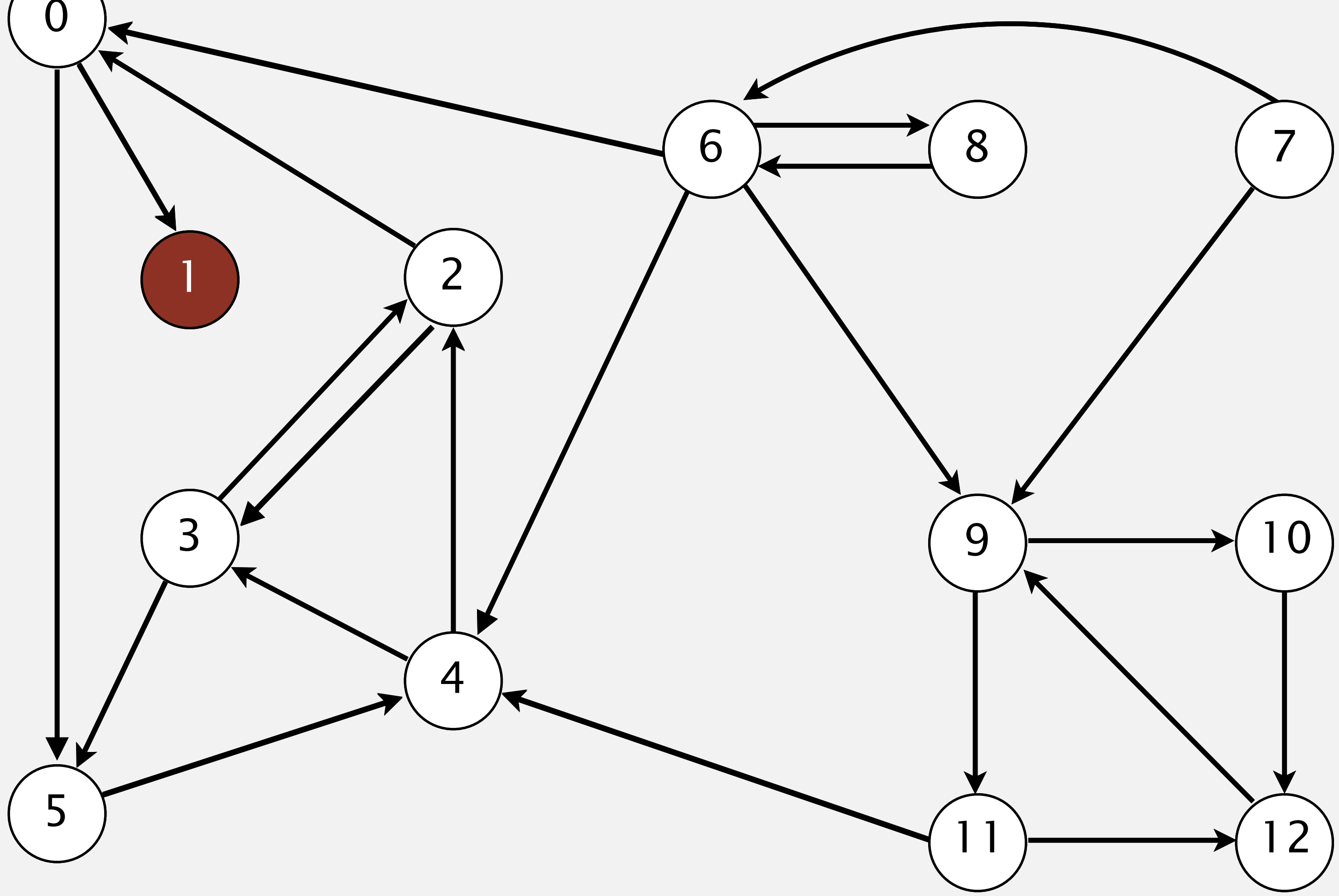
v	id[]
0	–
1	0
2	–
3	–
4	–
5	–
6	–
7	–
8	–
9	–
10	–
11	–
12	–

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



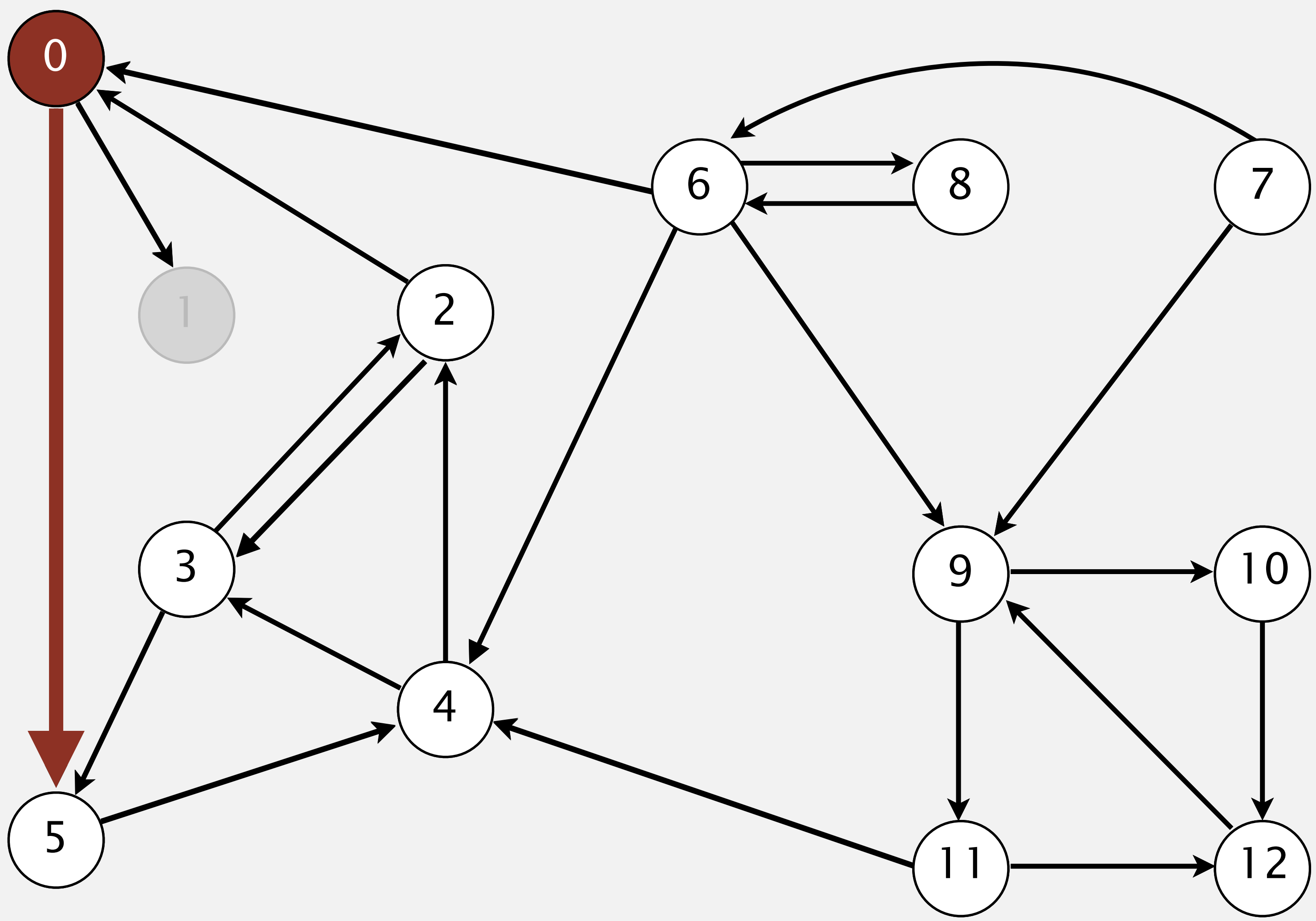
v	id[]
0	–
1	0
2	–
3	–
4	–
5	–
6	–
7	–
8	–
9	–
10	–
11	–
12	–

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



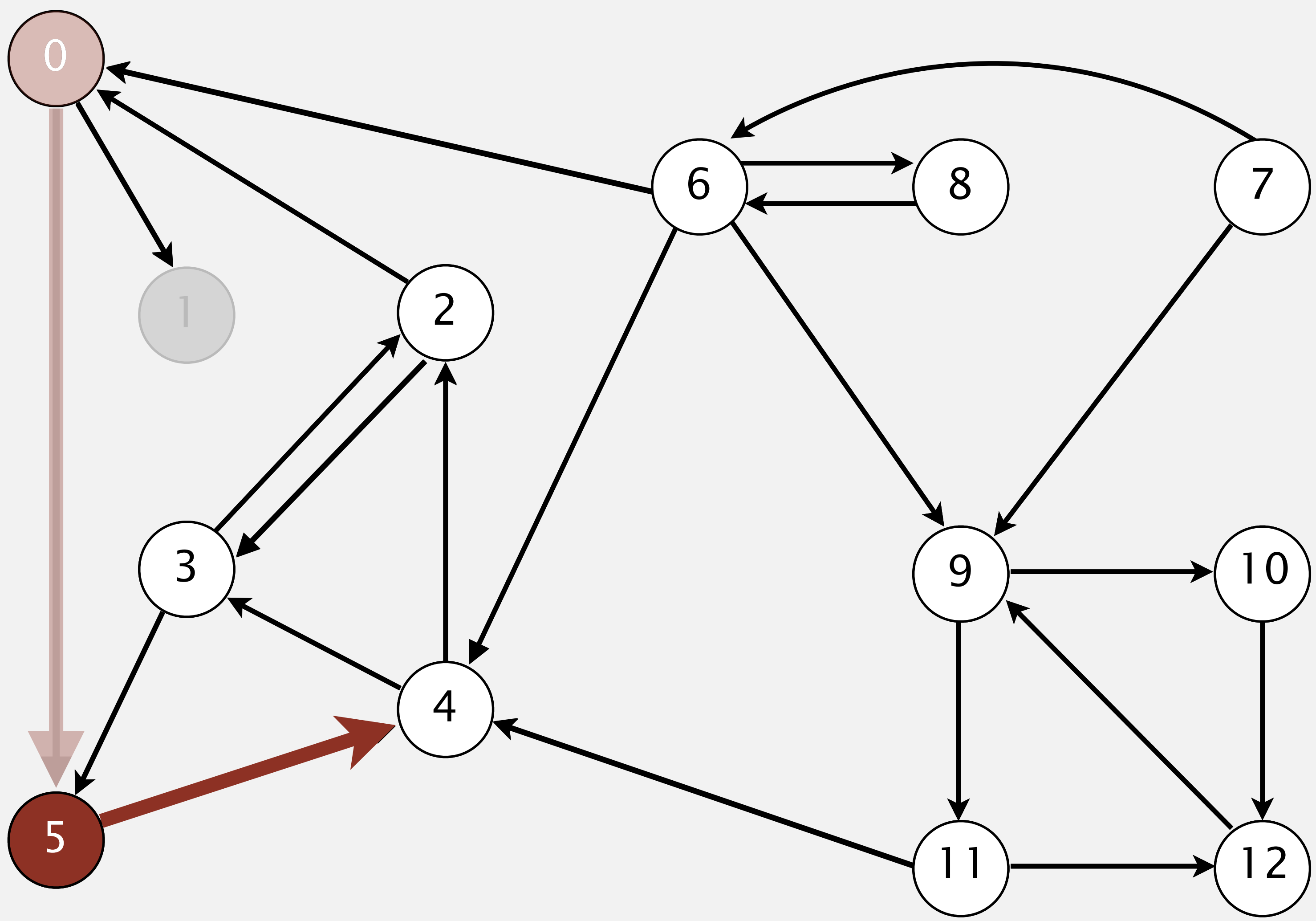
v	id[]
0	1
1	0
2	–
3	–
4	–
5	–
6	–
7	–
8	–
9	–
10	–
11	–
12	–

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



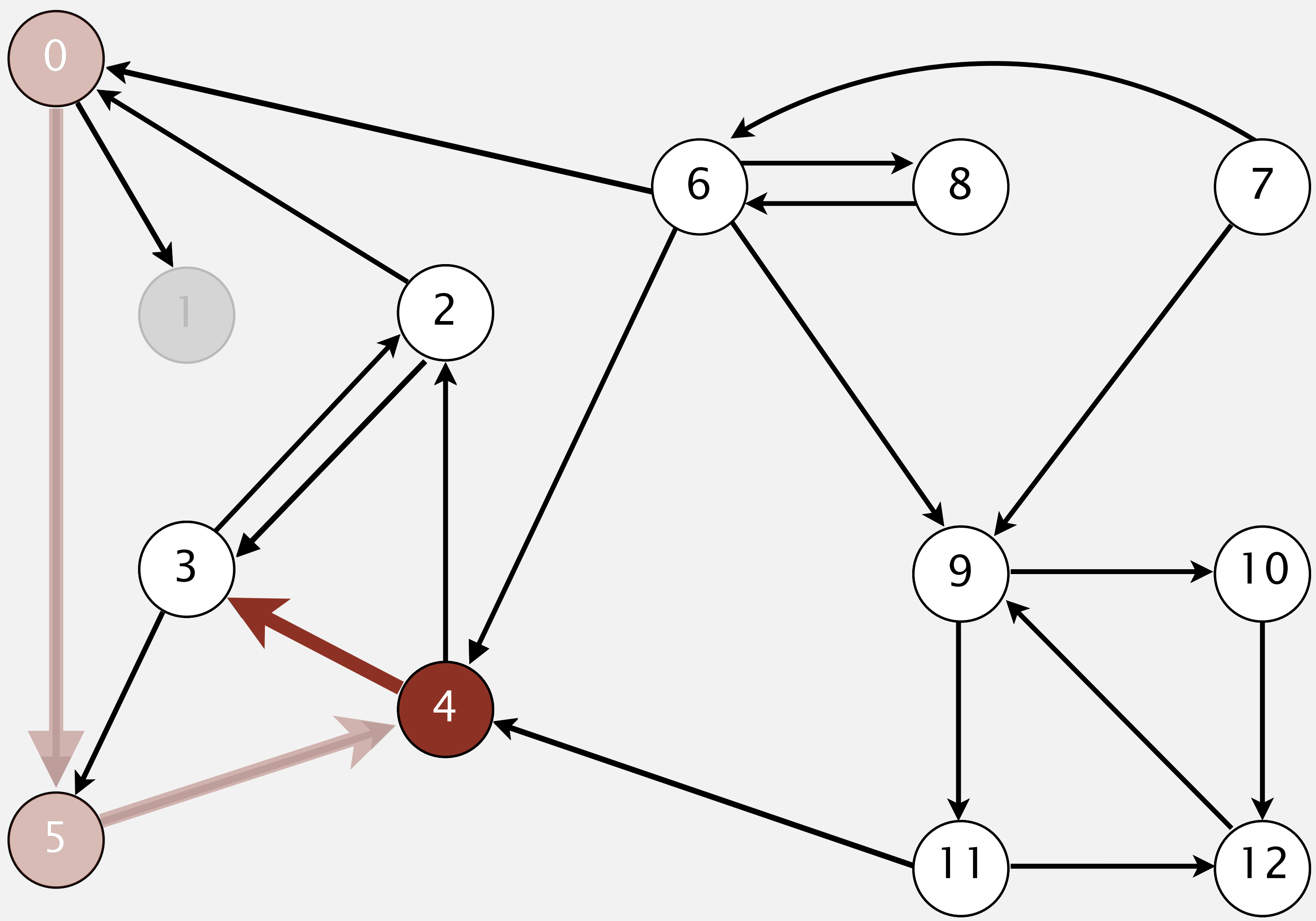
v	id[]
0	1
1	0
2	–
3	–
4	–
5	1
6	–
7	–
8	–
9	–
10	–
11	–
12	–

visit 5: 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



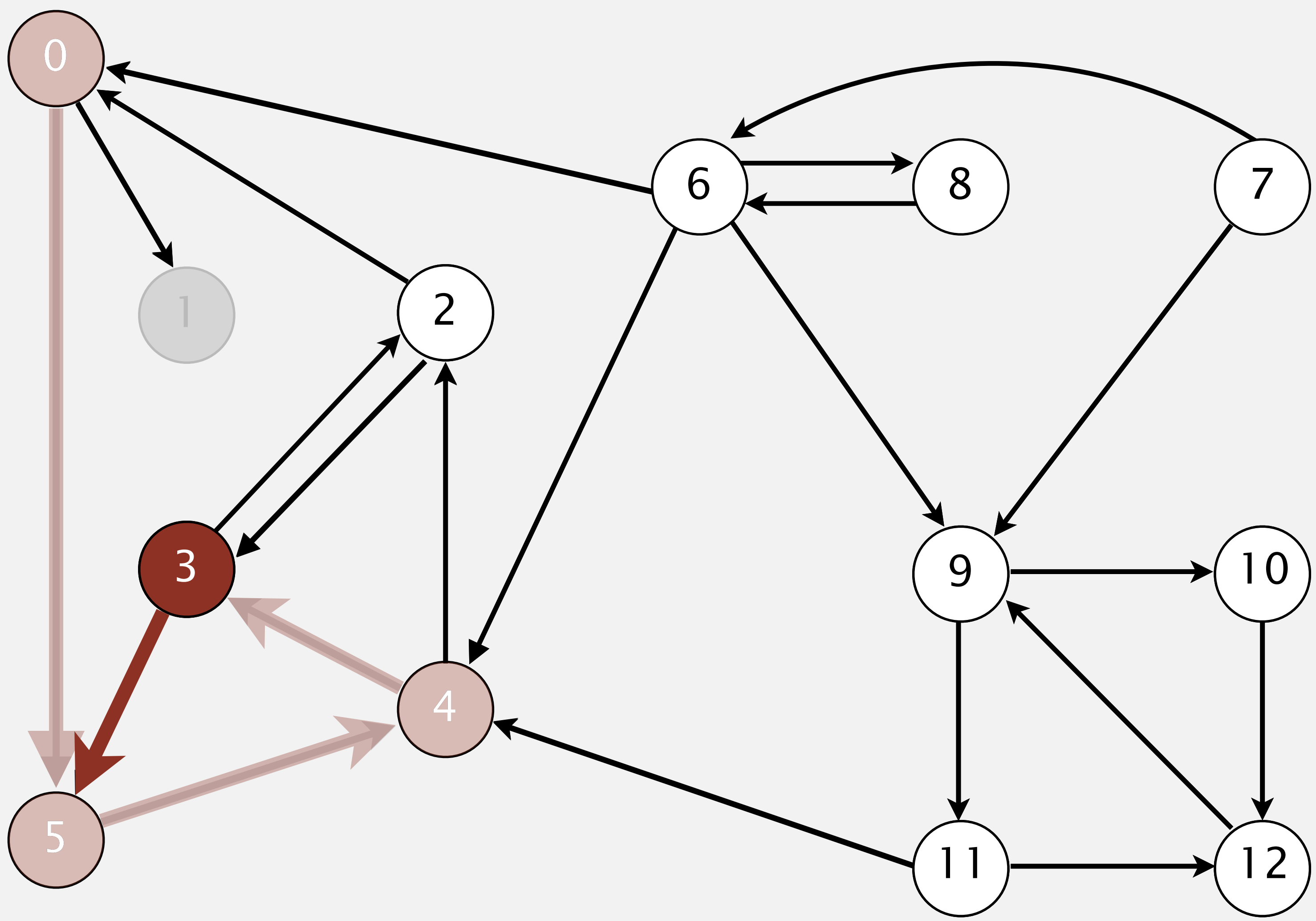
v	id[]
0	1
1	0
2	–
3	–
4	1
5	1
6	–
7	–
8	–
9	–
10	–
11	–
12	–

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



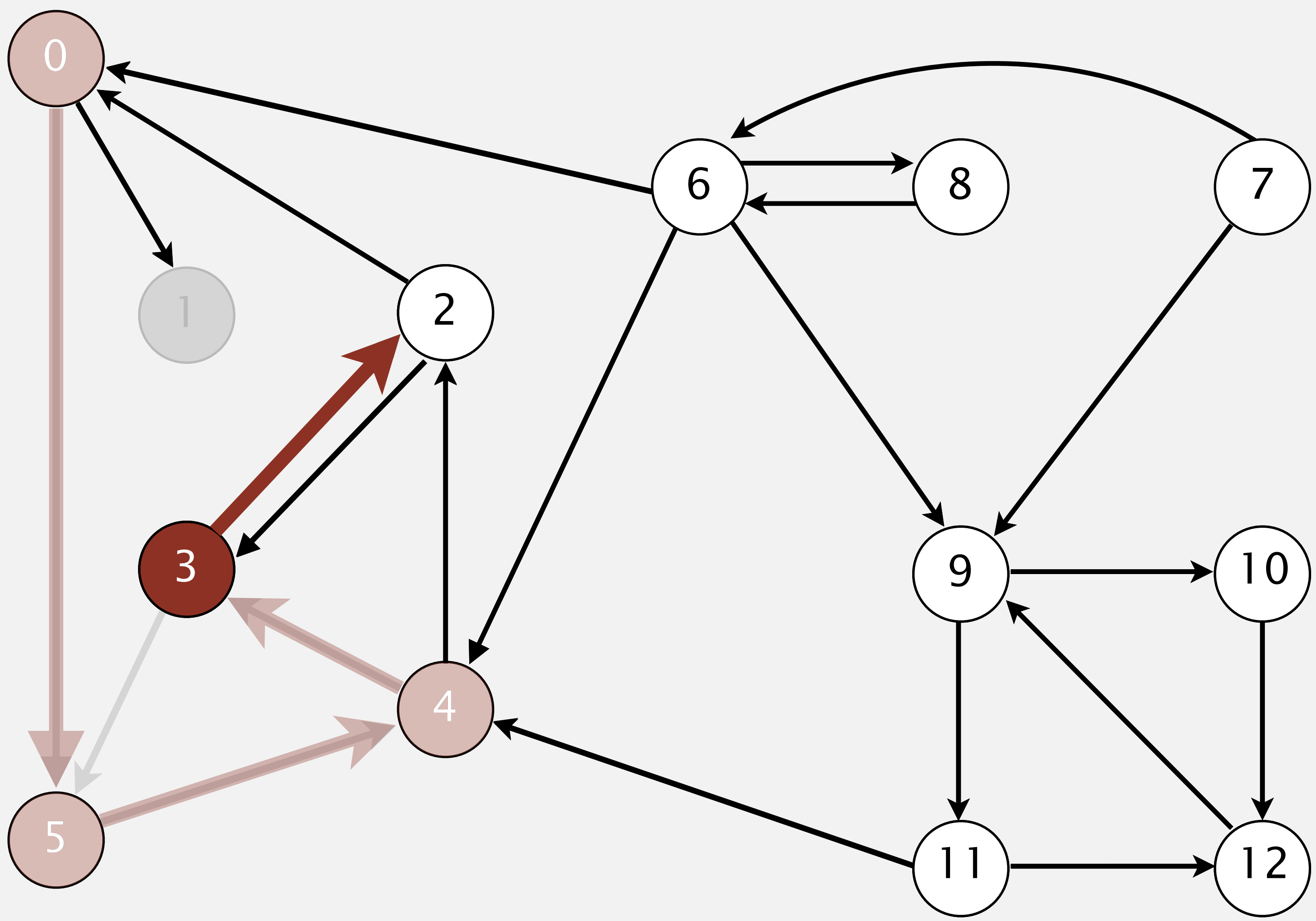
v	id[]
0	1
1	0
2	–
3	1
4	1
5	1
6	–
7	–
8	–
9	–
10	–
11	–
12	–

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



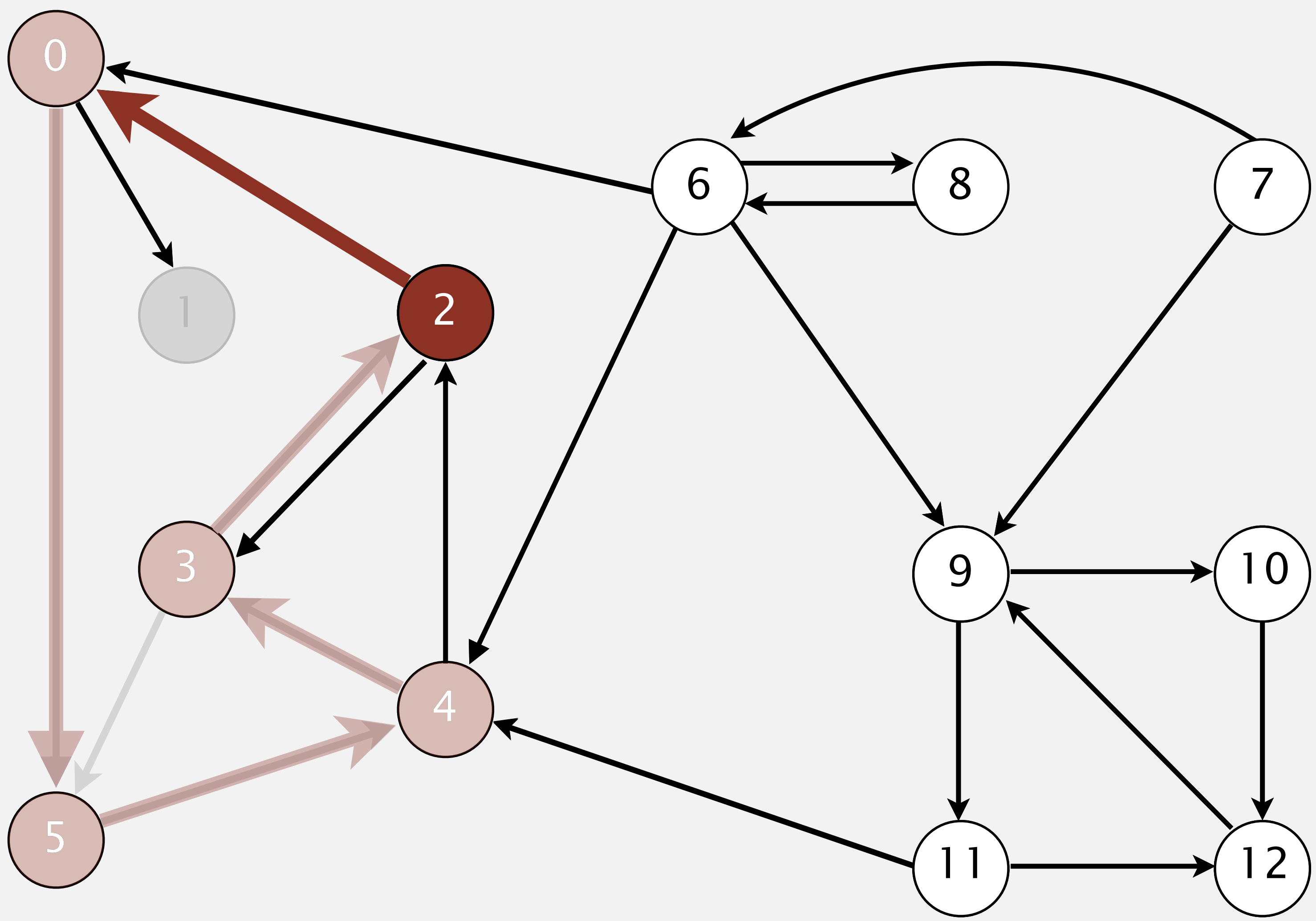
v	id[]
0	1
1	0
2	–
3	1
4	1
5	1
6	–
7	–
8	–
9	–
10	–
11	–
12	–

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



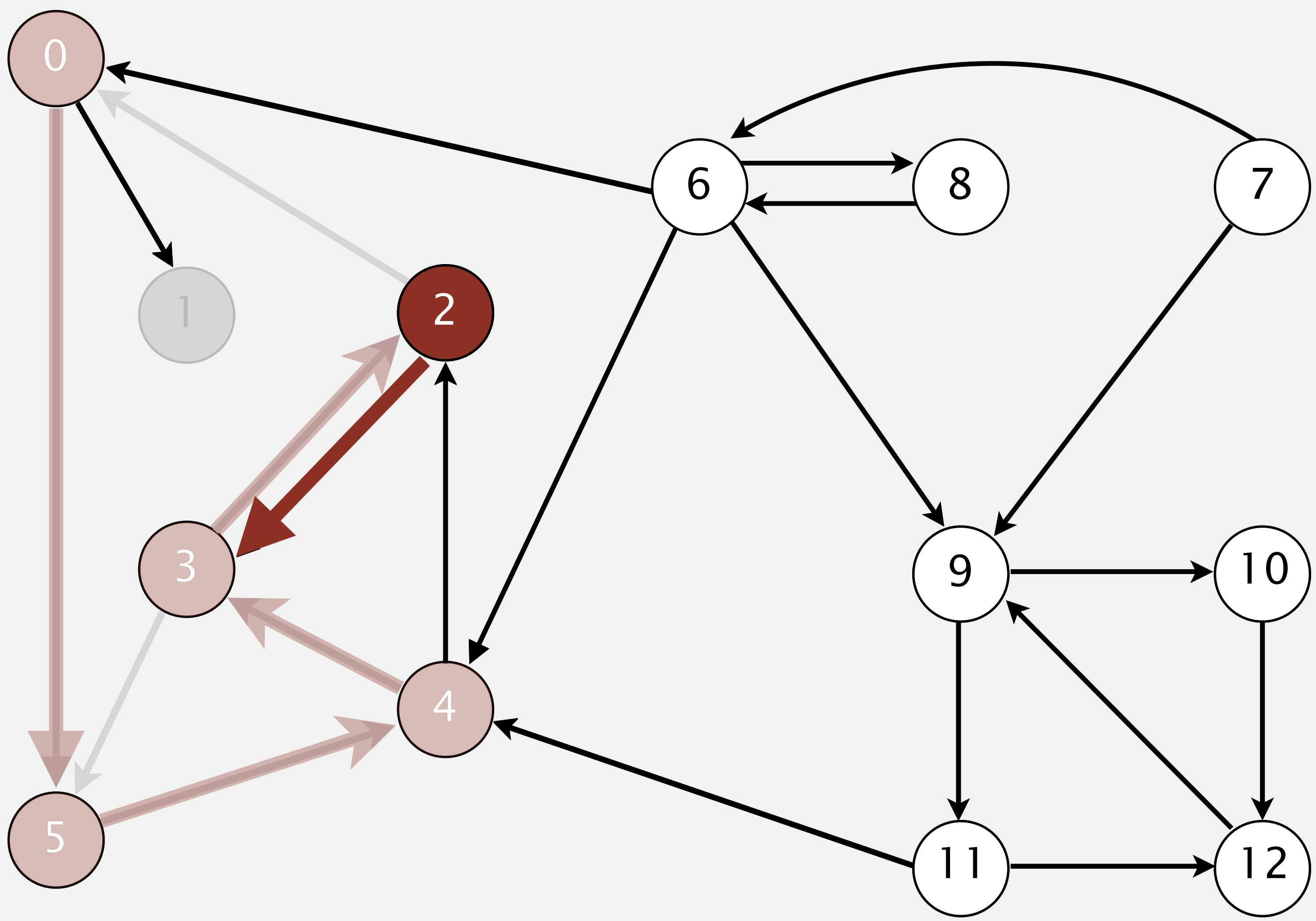
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	–
7	–
8	–
9	–
10	–
11	–
12	–

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



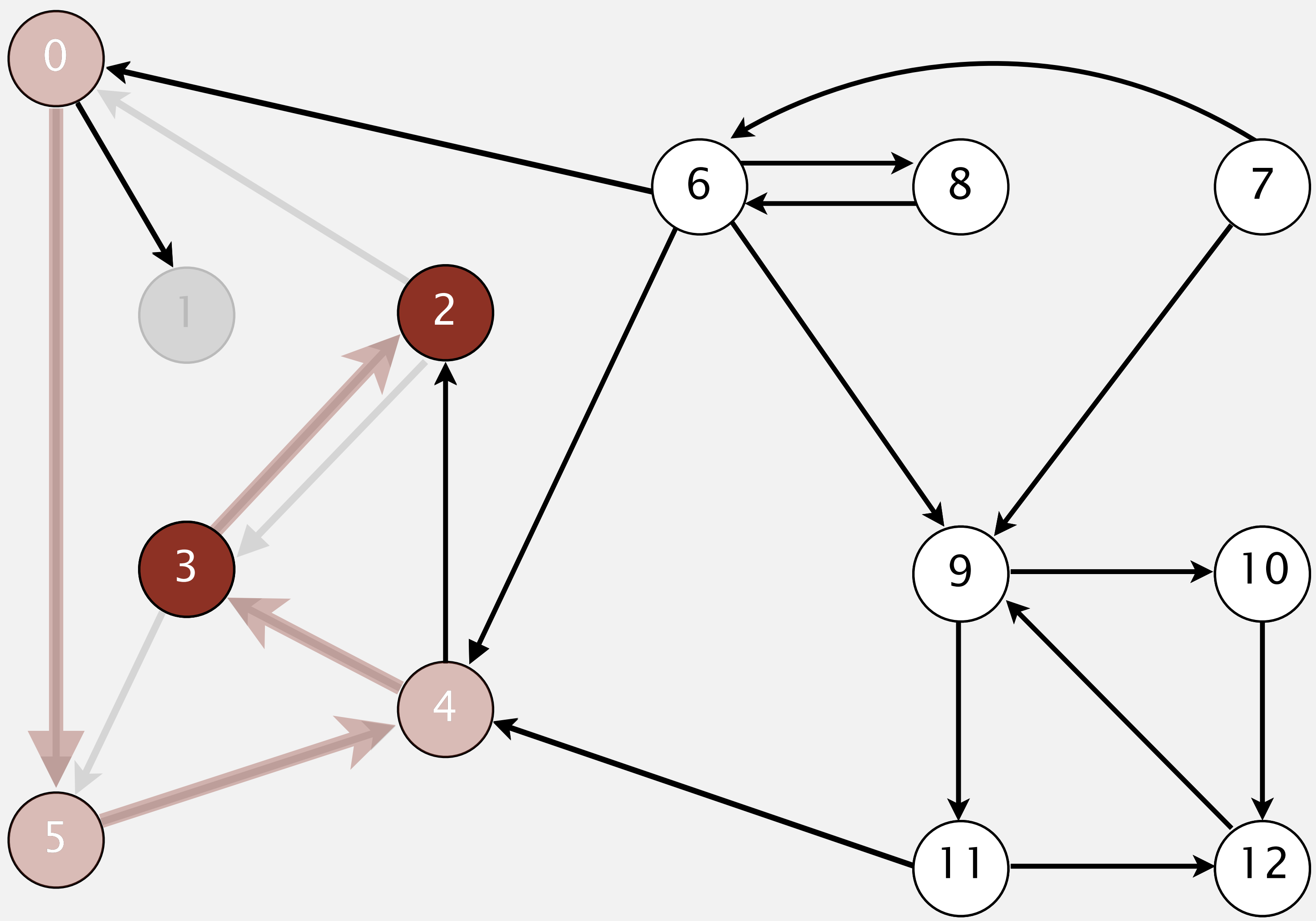
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	–
7	–
8	–
9	–
10	–
11	–
12	–

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



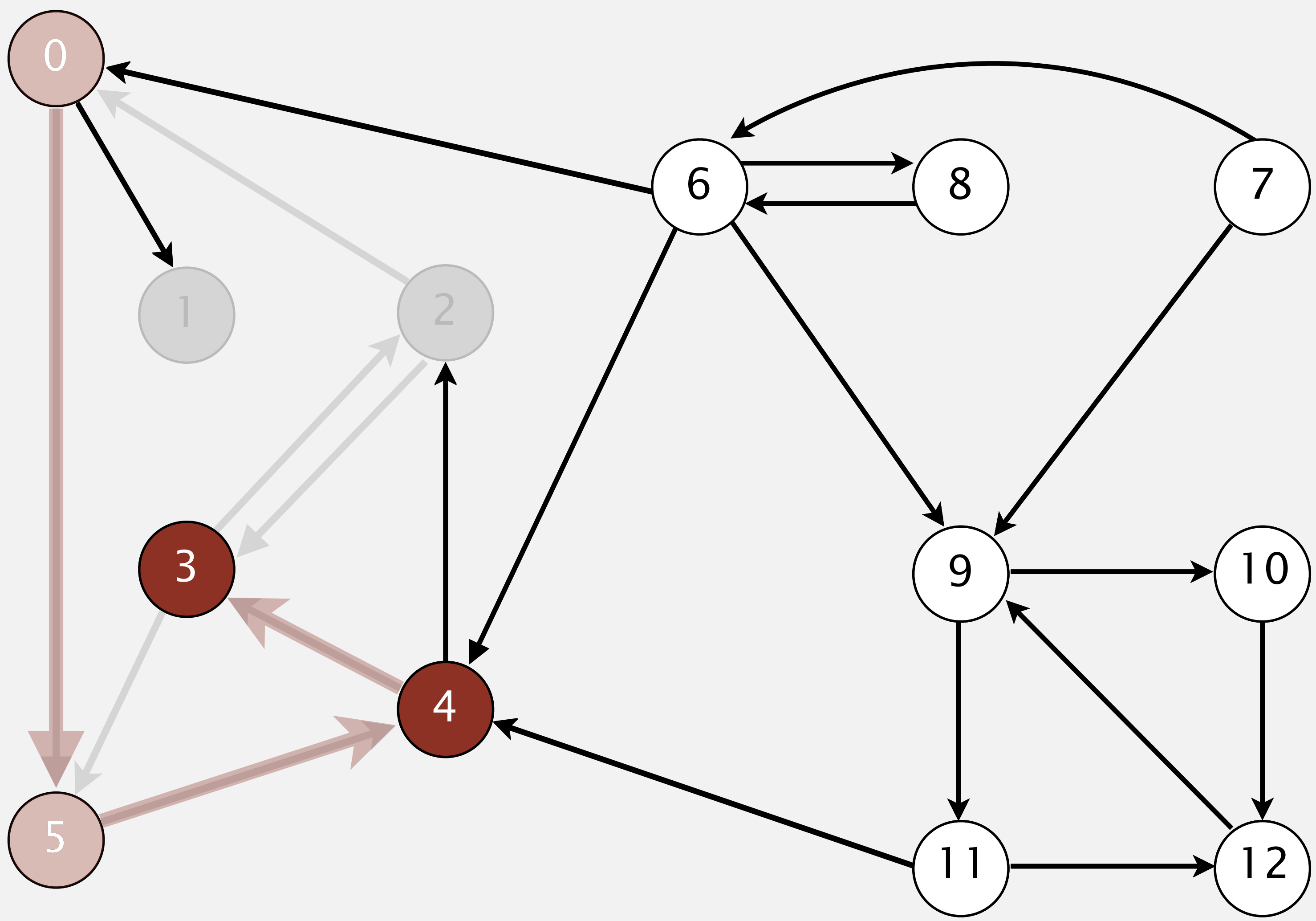
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	–
7	–
8	–
9	–
10	–
11	–
12	–

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



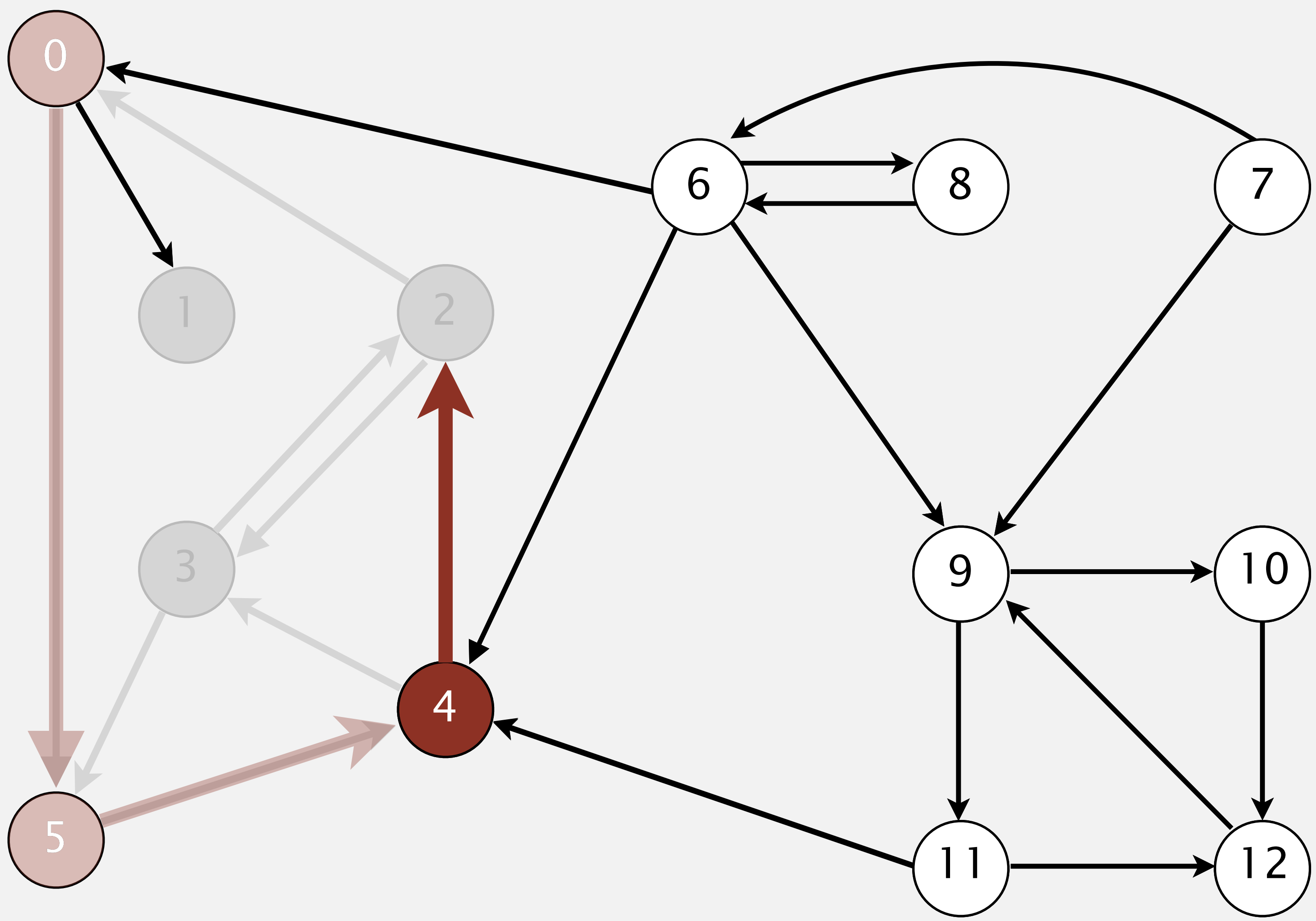
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	–
7	–
8	–
9	–
10	–
11	–
12	–

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



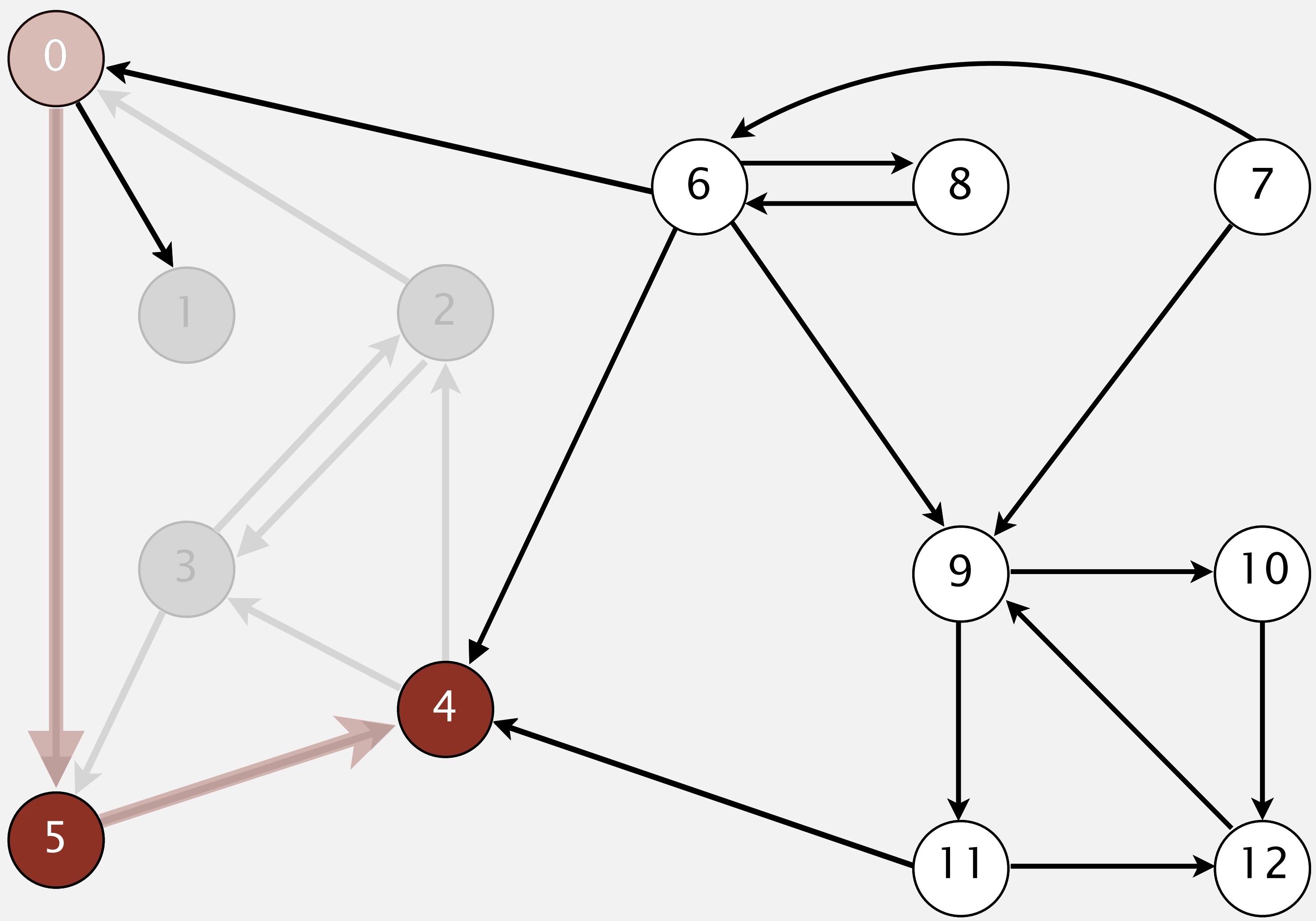
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	–
7	–
8	–
9	–
10	–
11	–
12	–

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



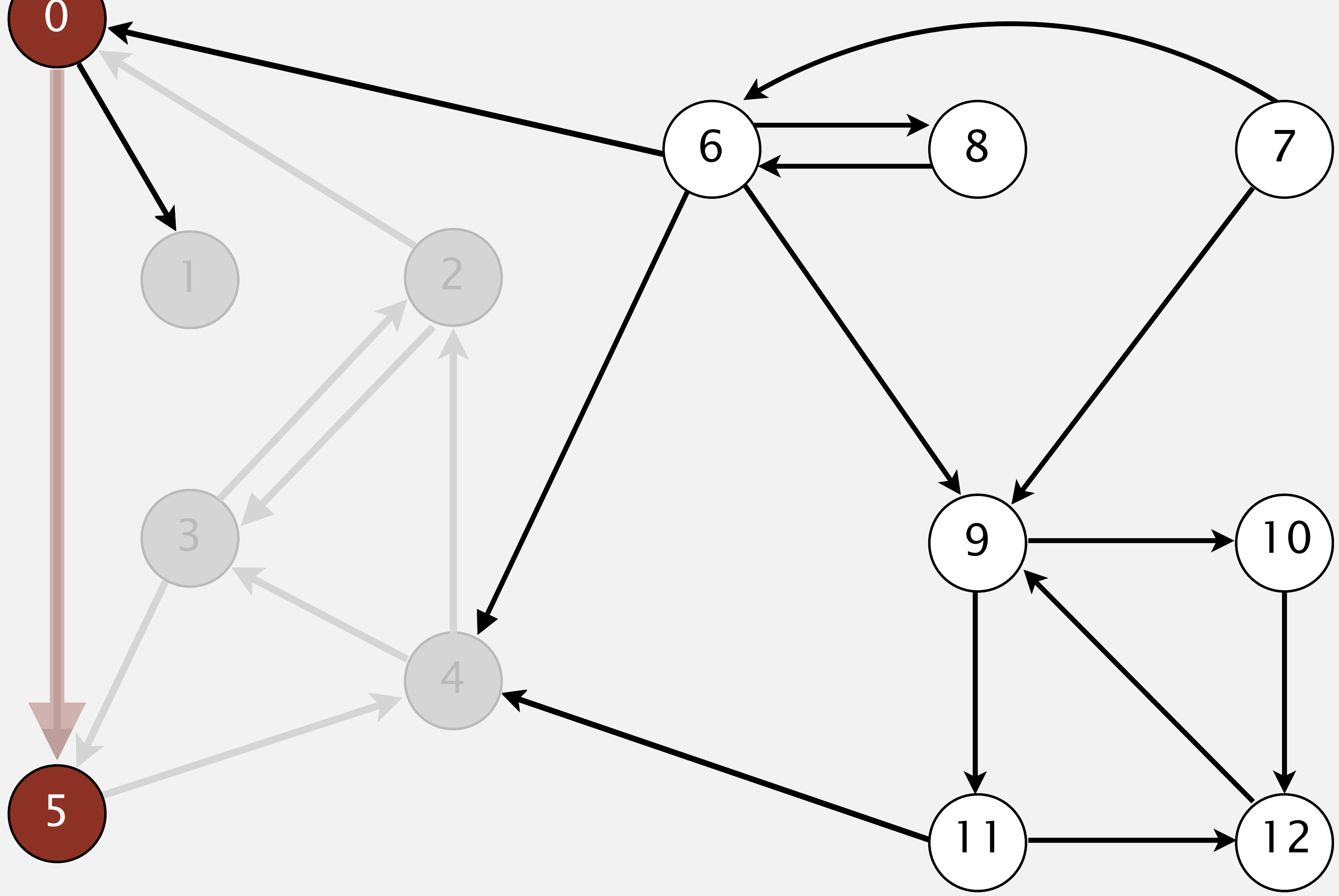
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	–
7	–
8	–
9	–
10	–
11	–
12	–

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



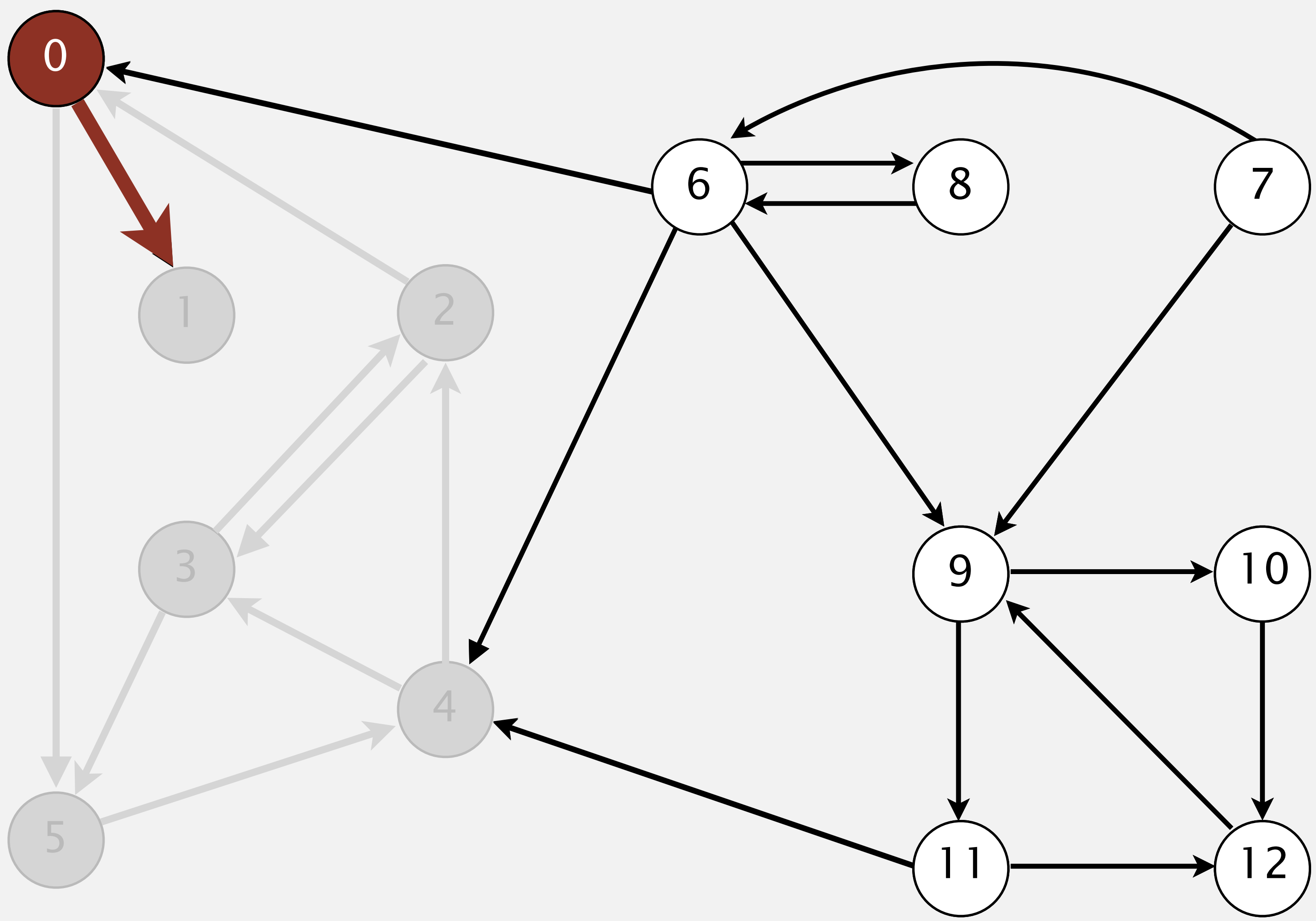
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	–
7	–
8	–
9	–
10	–
11	–
12	–

5 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



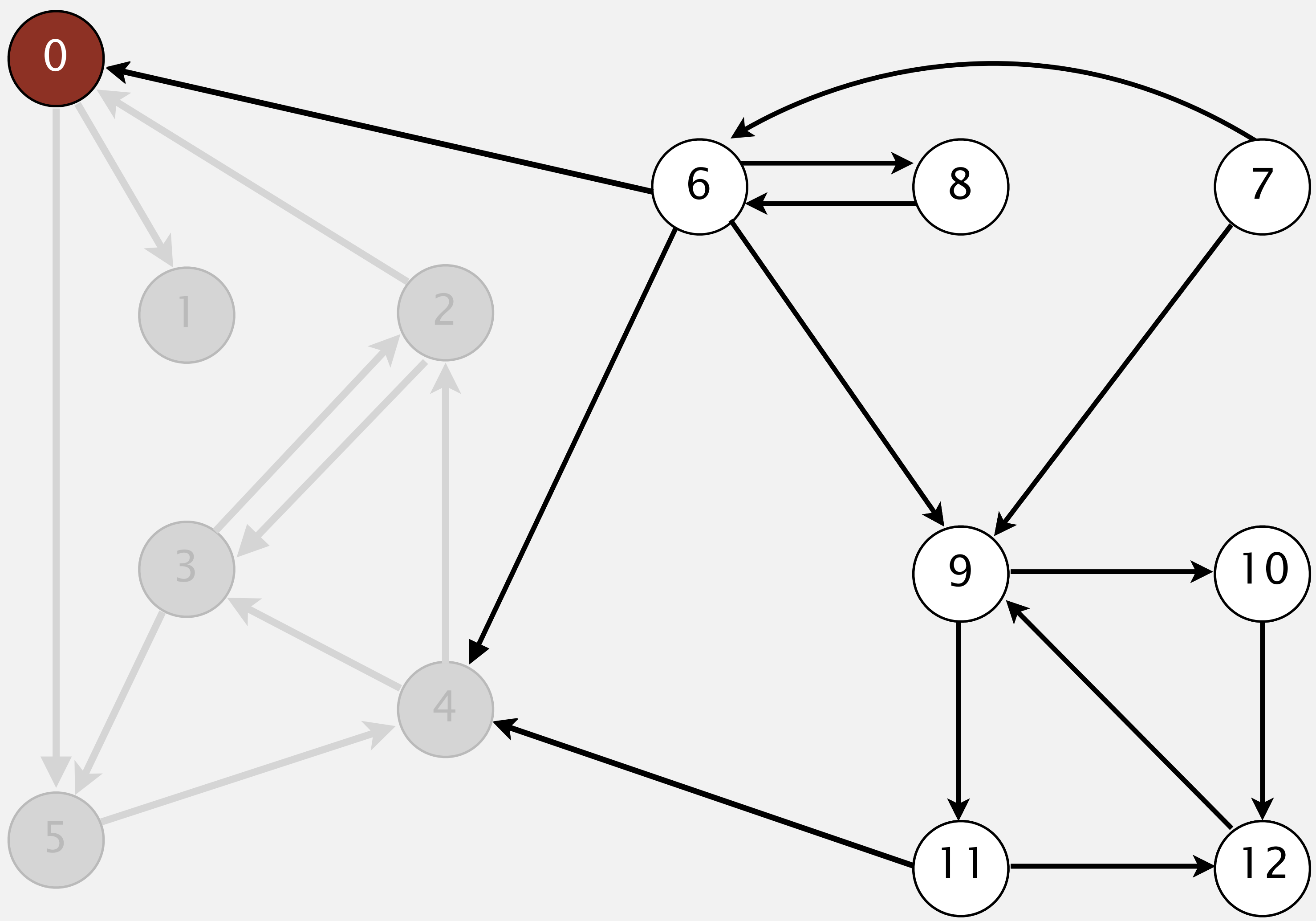
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	–
7	–
8	–
9	–
10	–
11	–
12	–

visit 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



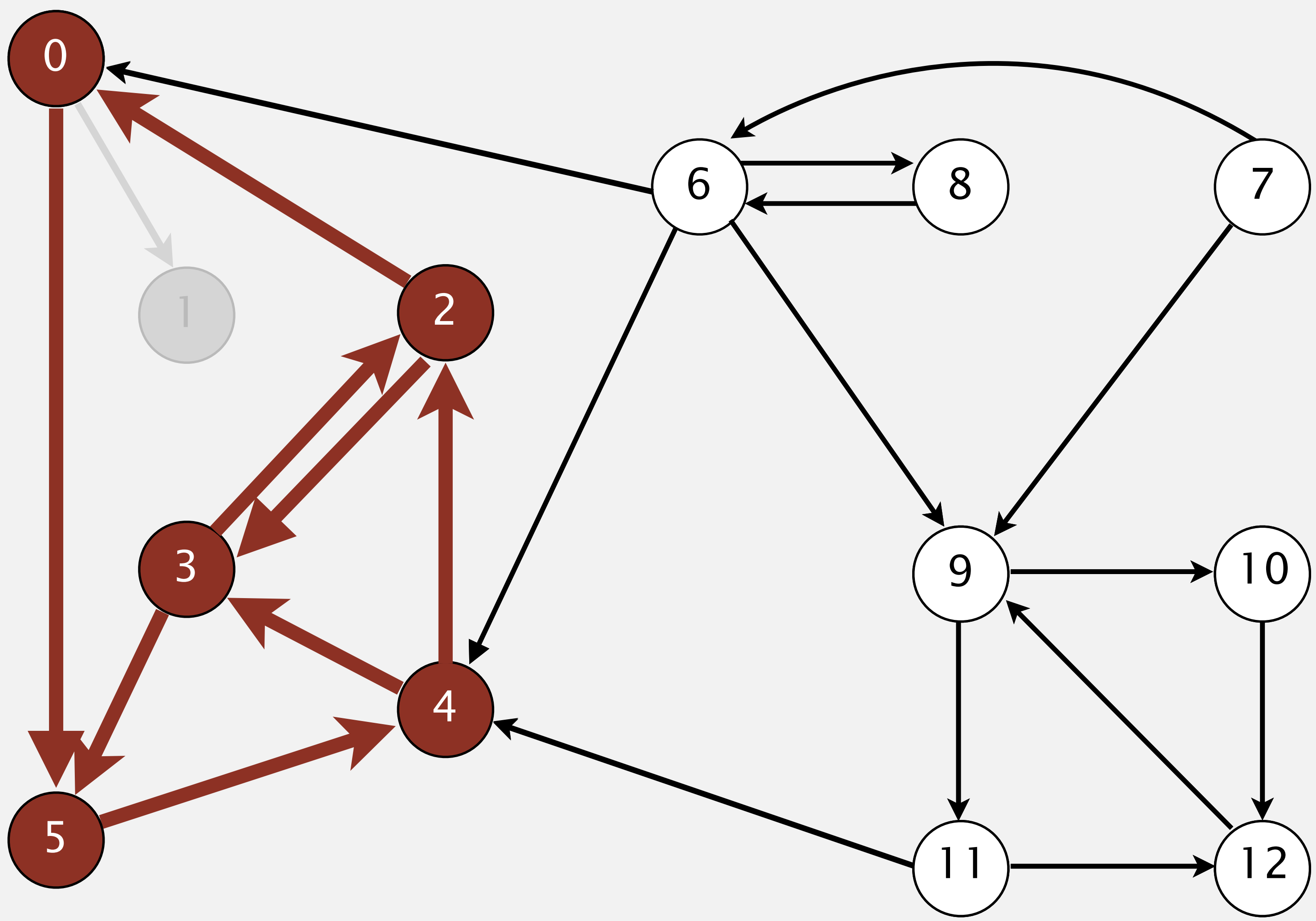
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	–
7	–
8	–
9	–
10	–
11	–
12	–

0 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



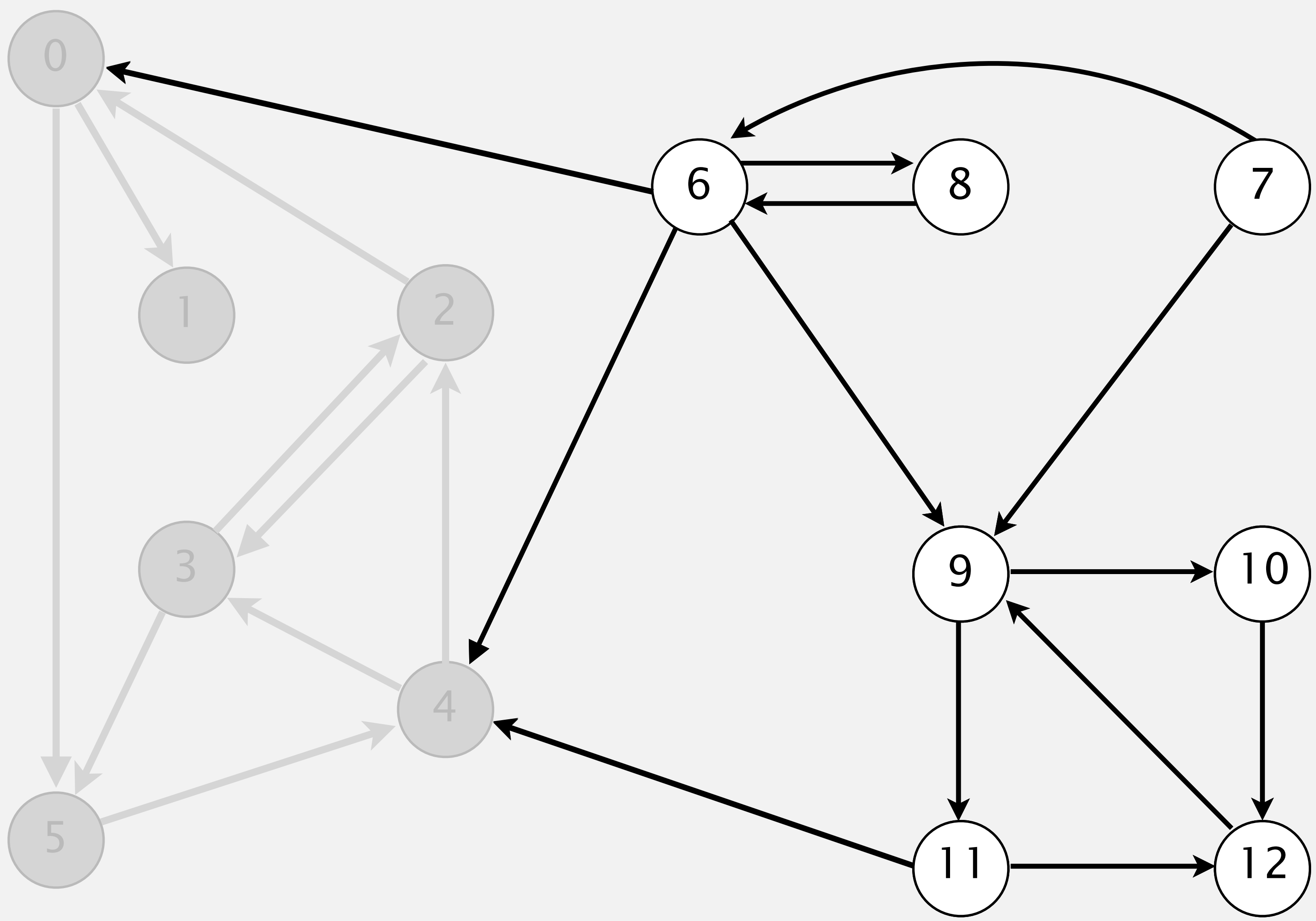
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	–
7	–
8	–
9	–
10	–
11	–
12	–

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



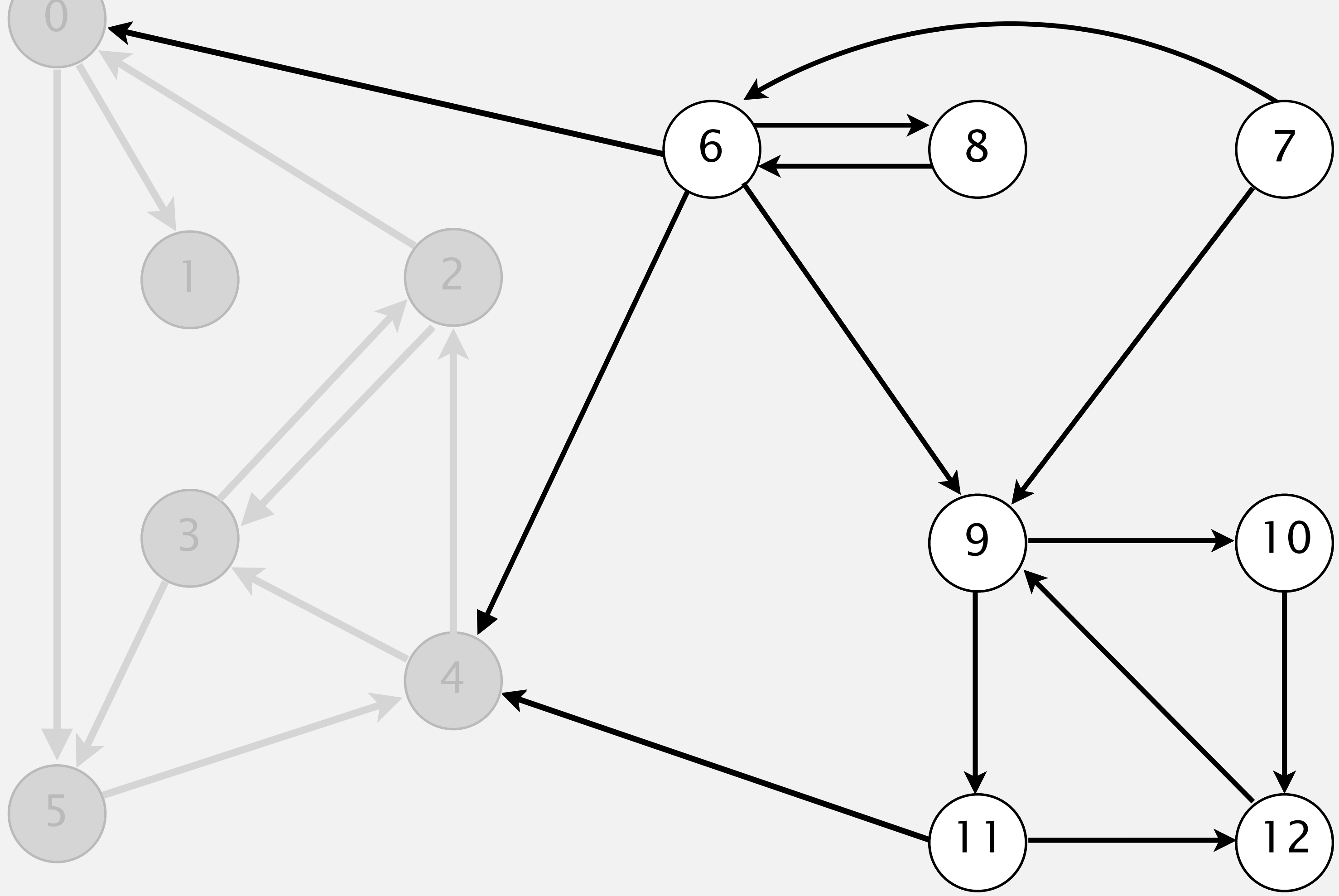
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	–
7	–
8	–
9	–
10	–
11	–
12	–

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



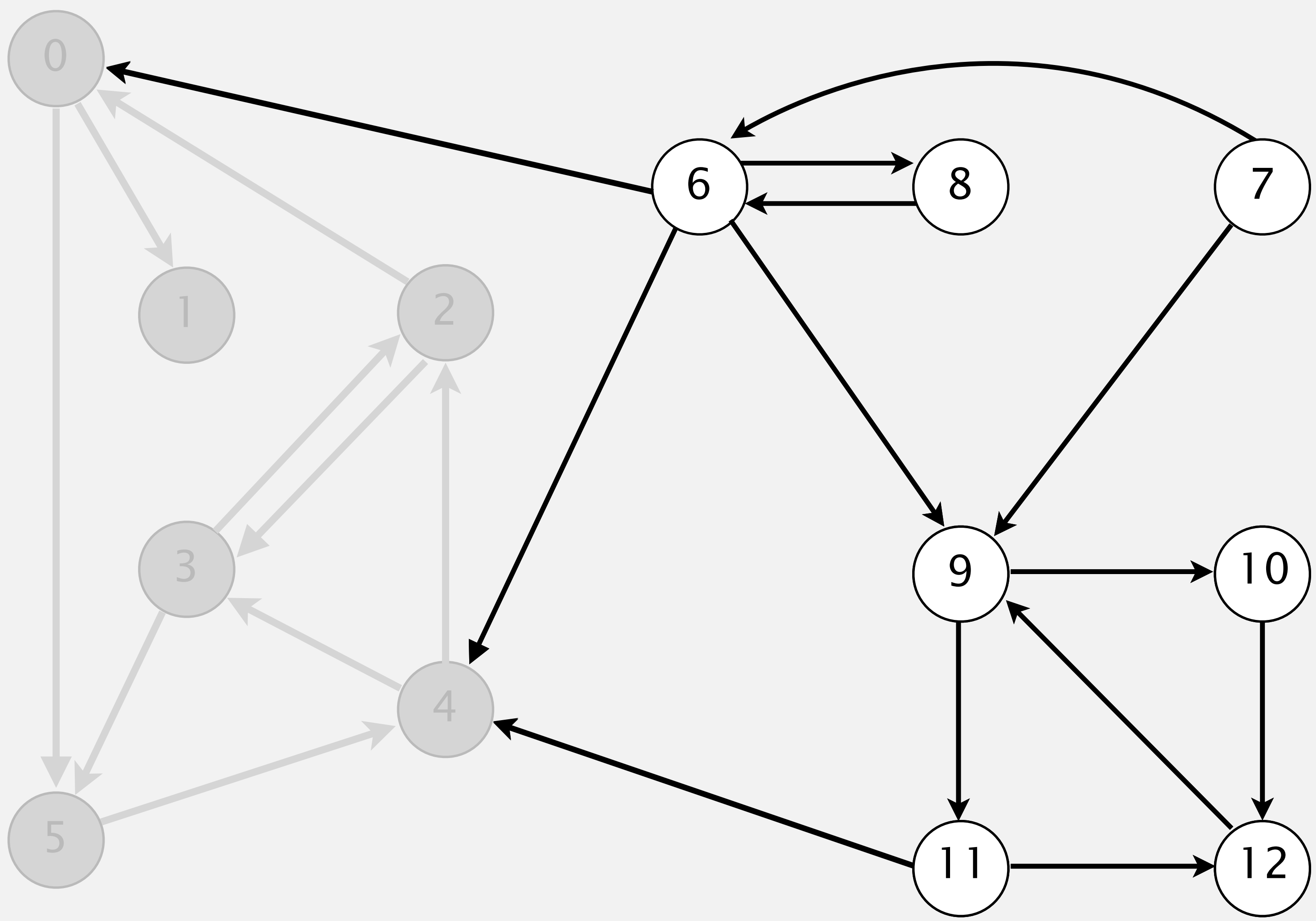
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	–
7	–
8	–
9	–
10	–
11	–
12	–

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



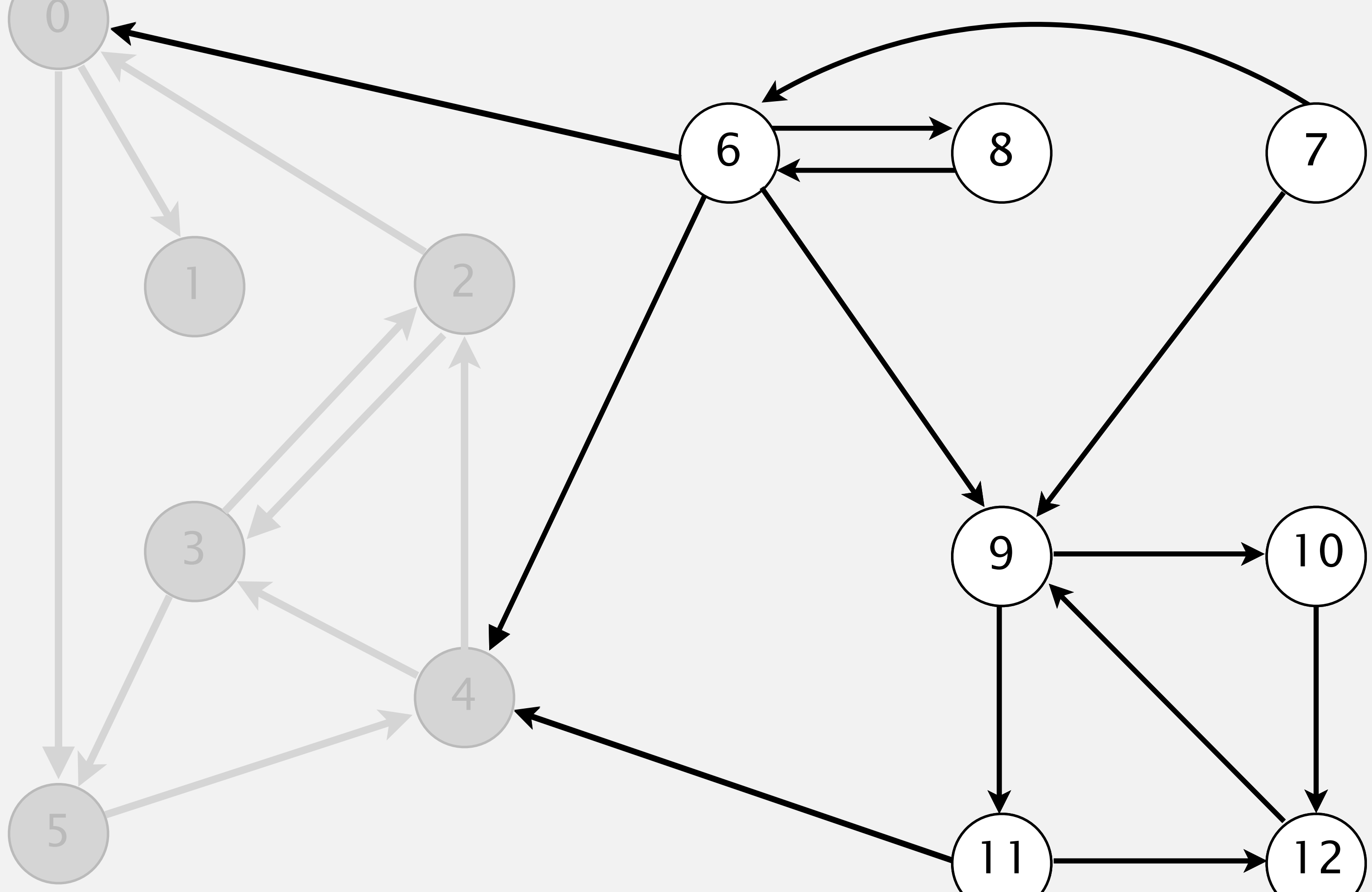
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	–
7	–
8	–
9	–
10	–
11	–
12	–

check 5

Kosaraju-Sharir algorithm demo

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

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



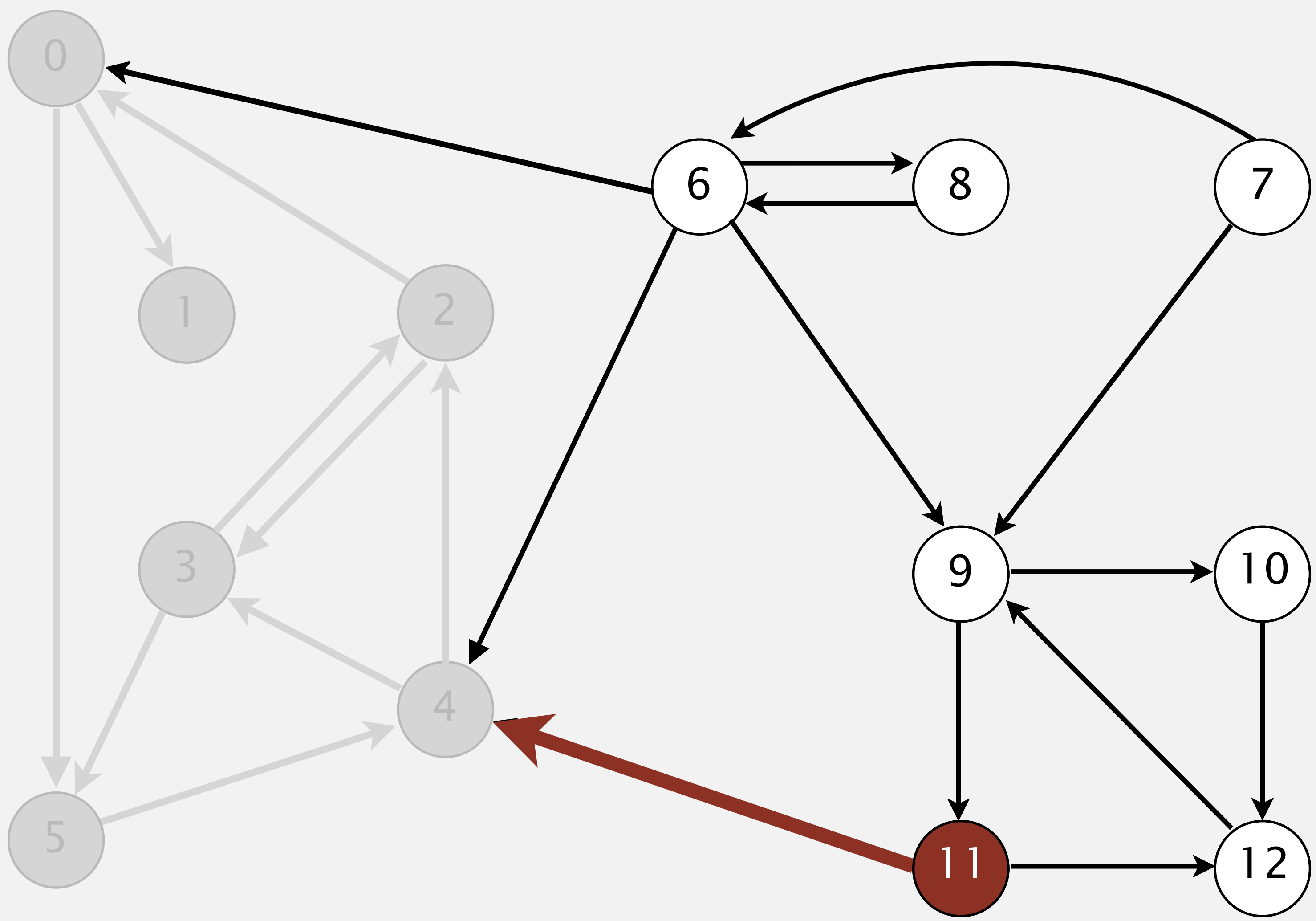
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	–
7	–
8	–
9	–
10	–
11	–
12	–

check 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



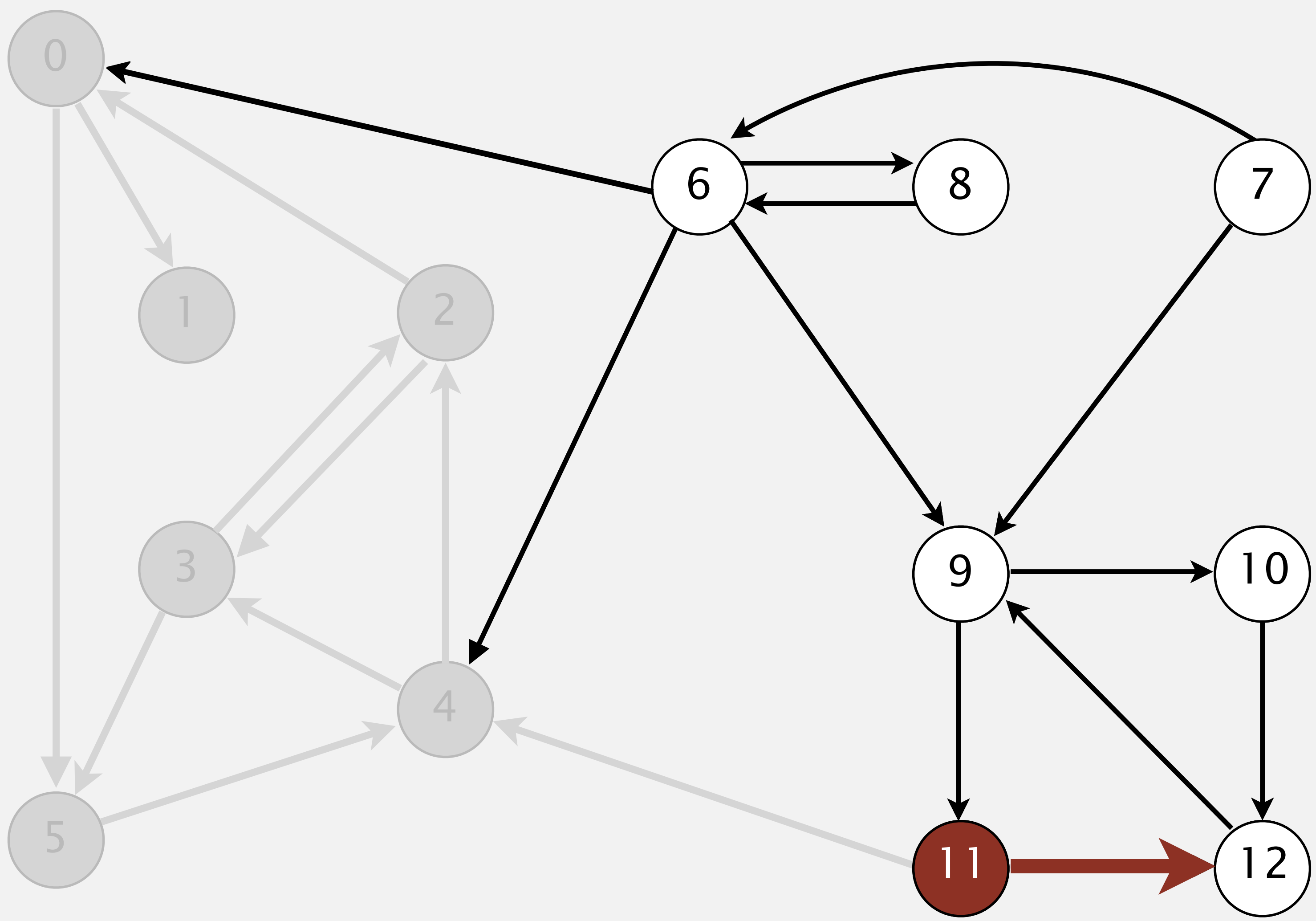
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	–
7	–
8	–
9	–
10	–
11	2
12	–

visit 11: check 4 and check 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



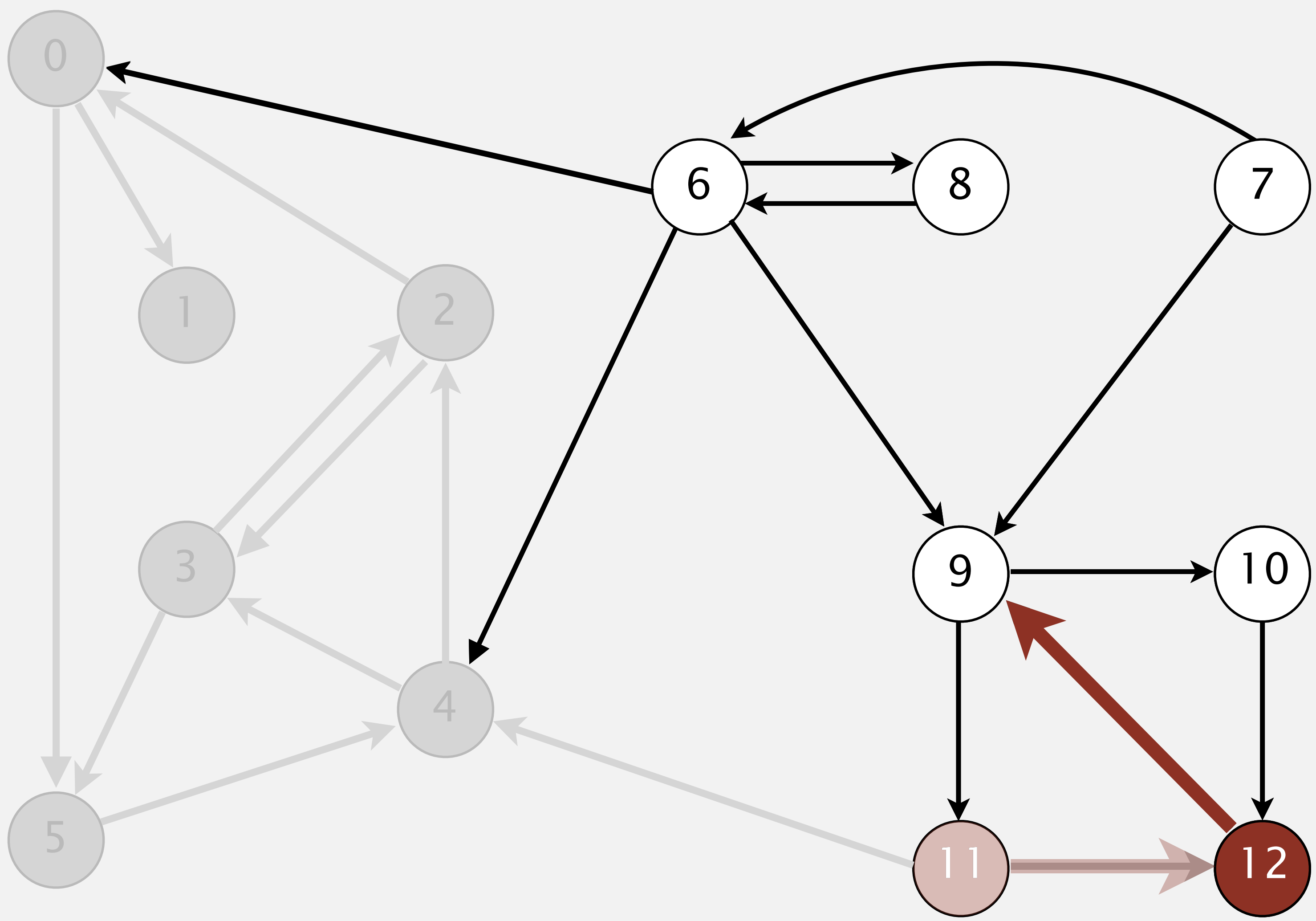
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	–
7	–
8	–
9	–
10	–
11	2
12	–

visit 11: check 4 and check 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



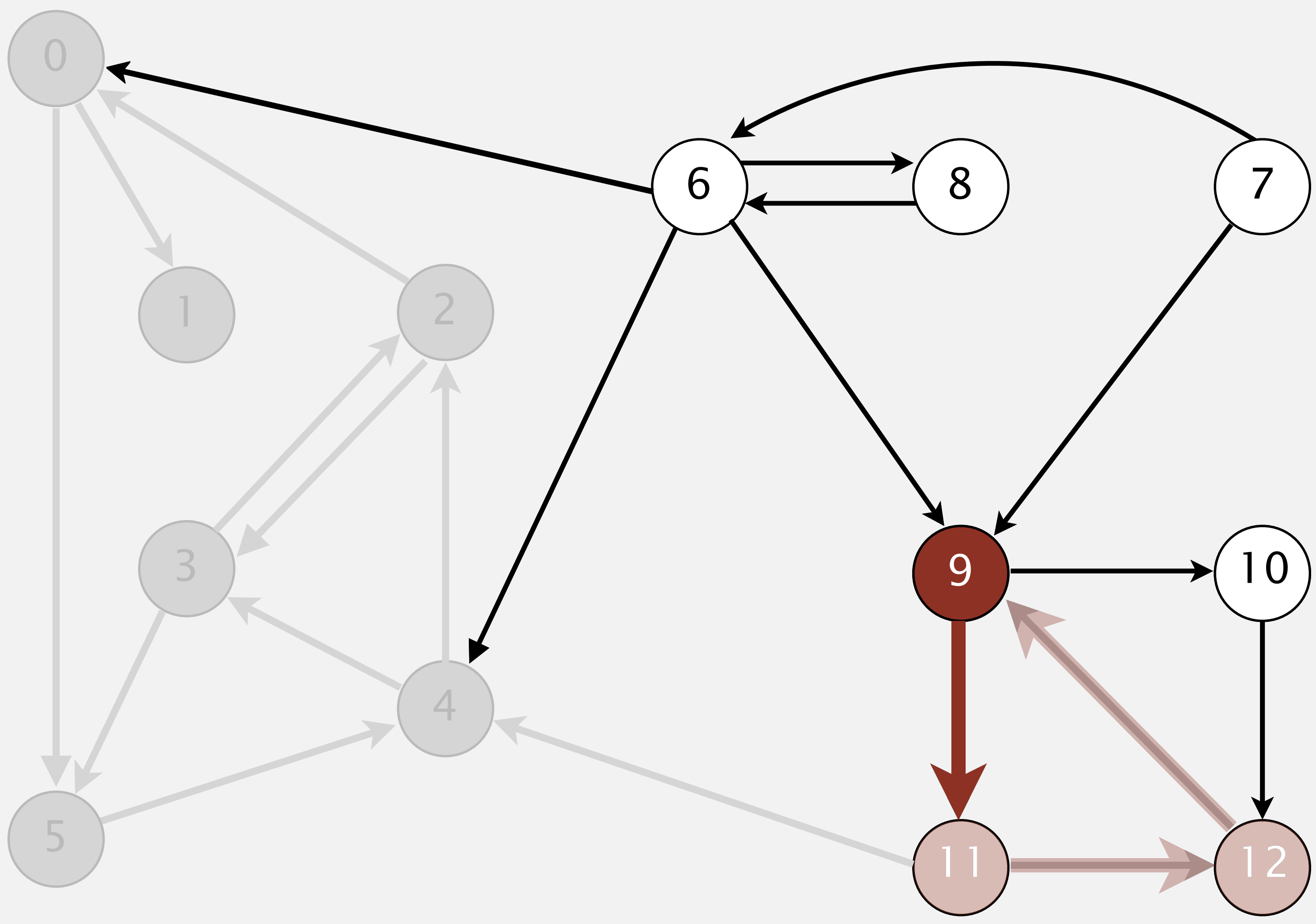
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	–
7	–
8	–
9	–
10	–
11	2
12	2

visit 12: check 9

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**



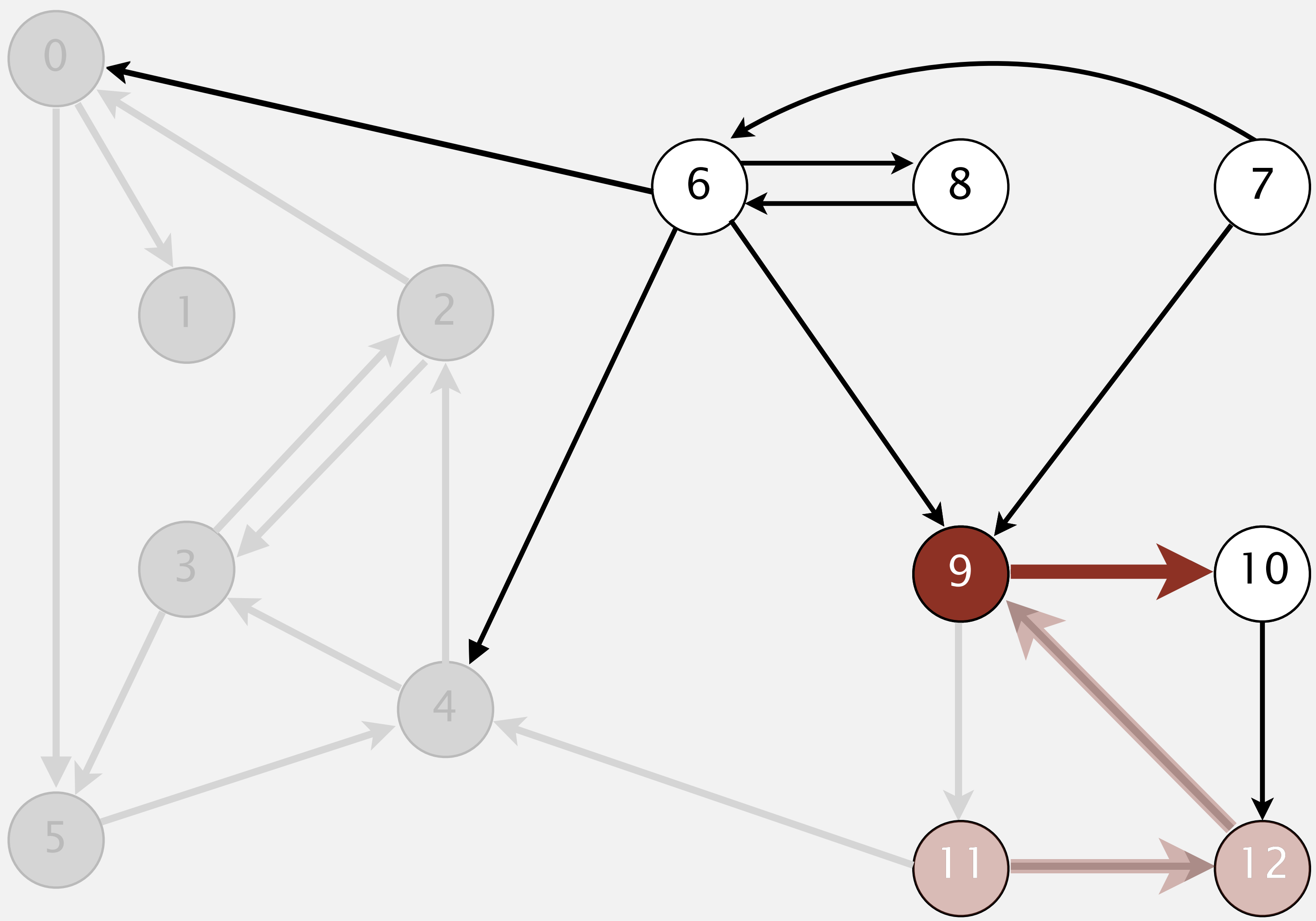
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	–
7	–
8	–
9	2
10	–
11	2
12	2

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.

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



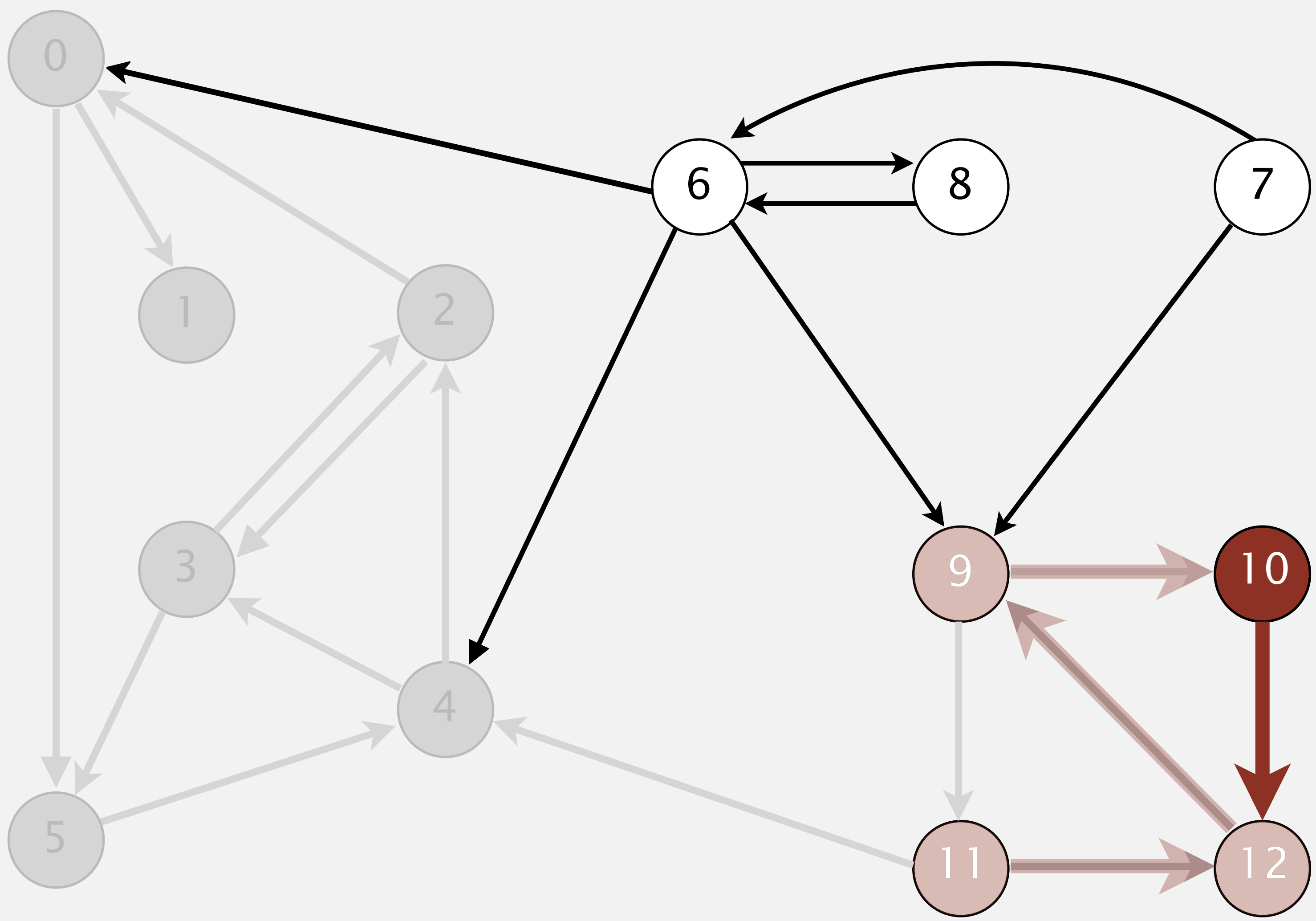
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	–
7	–
8	–
9	2
10	–
11	2
12	2

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.

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



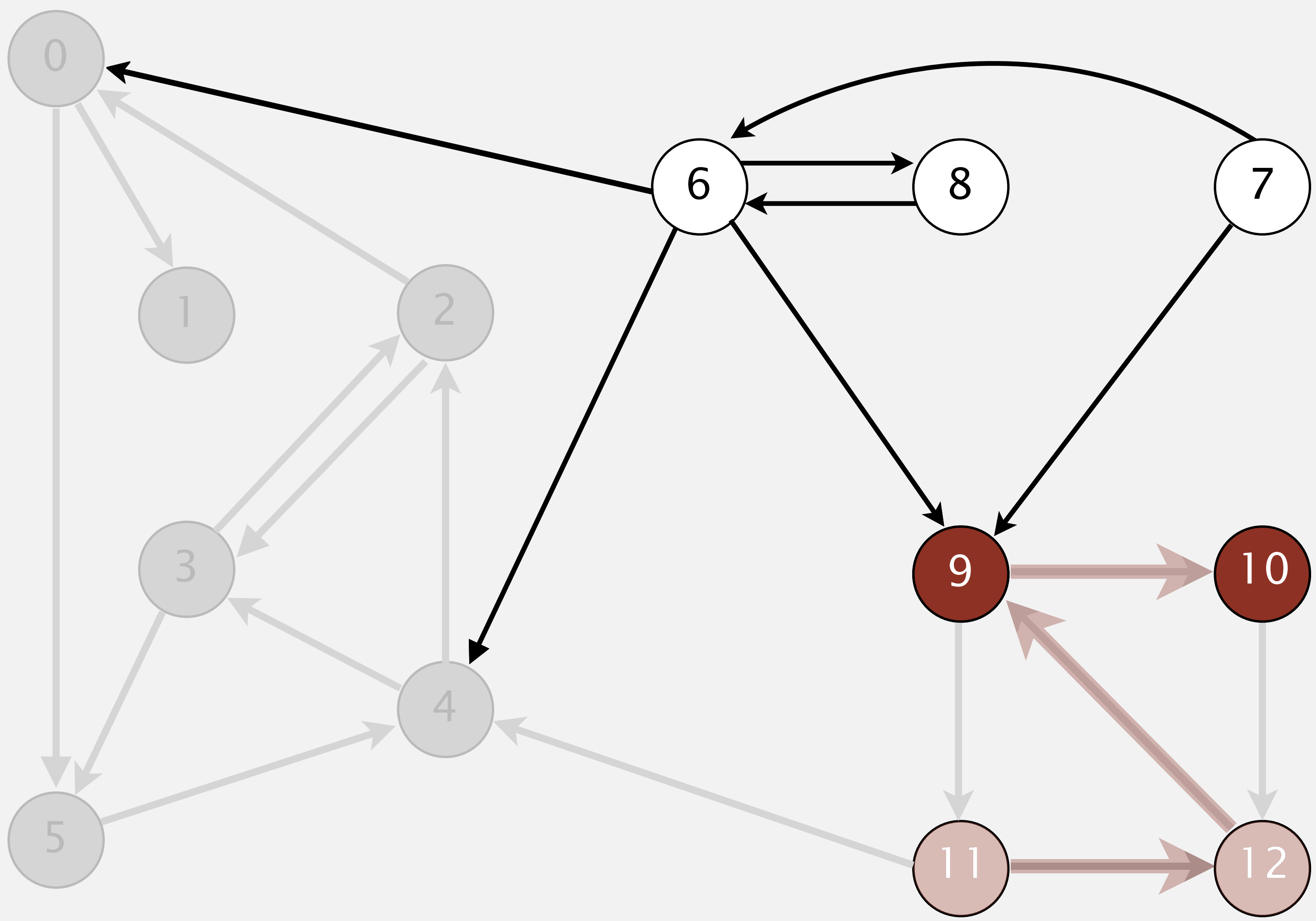
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	–
7	–
8	–
9	2
10	2
11	2
12	2

visit 10: check 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



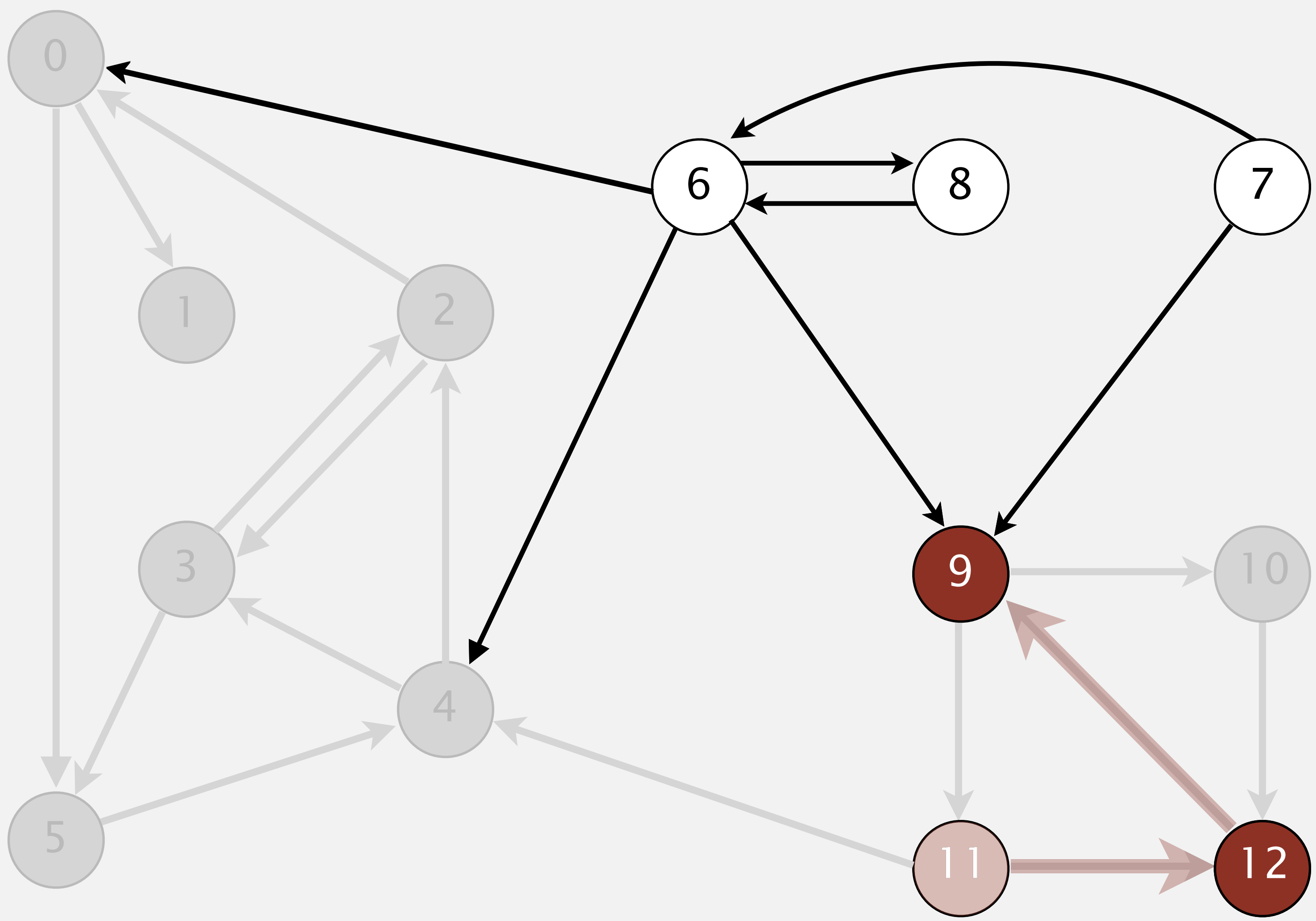
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	–
7	–
8	–
9	2
10	2
11	2
12	2

10 done

Kosaraju-Sharir algorithm demo

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

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



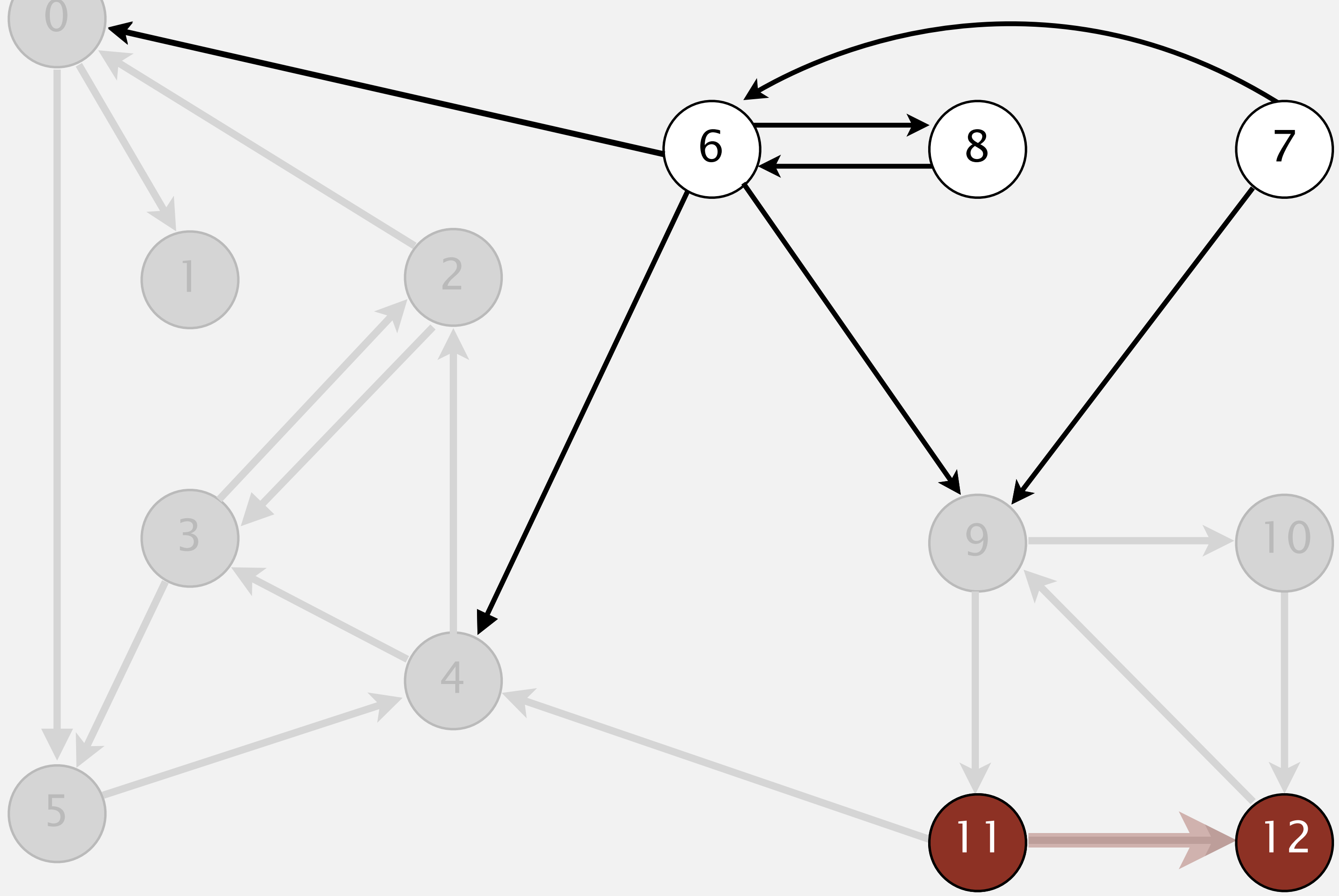
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	–
7	–
8	–
9	2
10	2
11	2
12	2

9 done

Kosaraju-Sharir algorithm demo

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

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



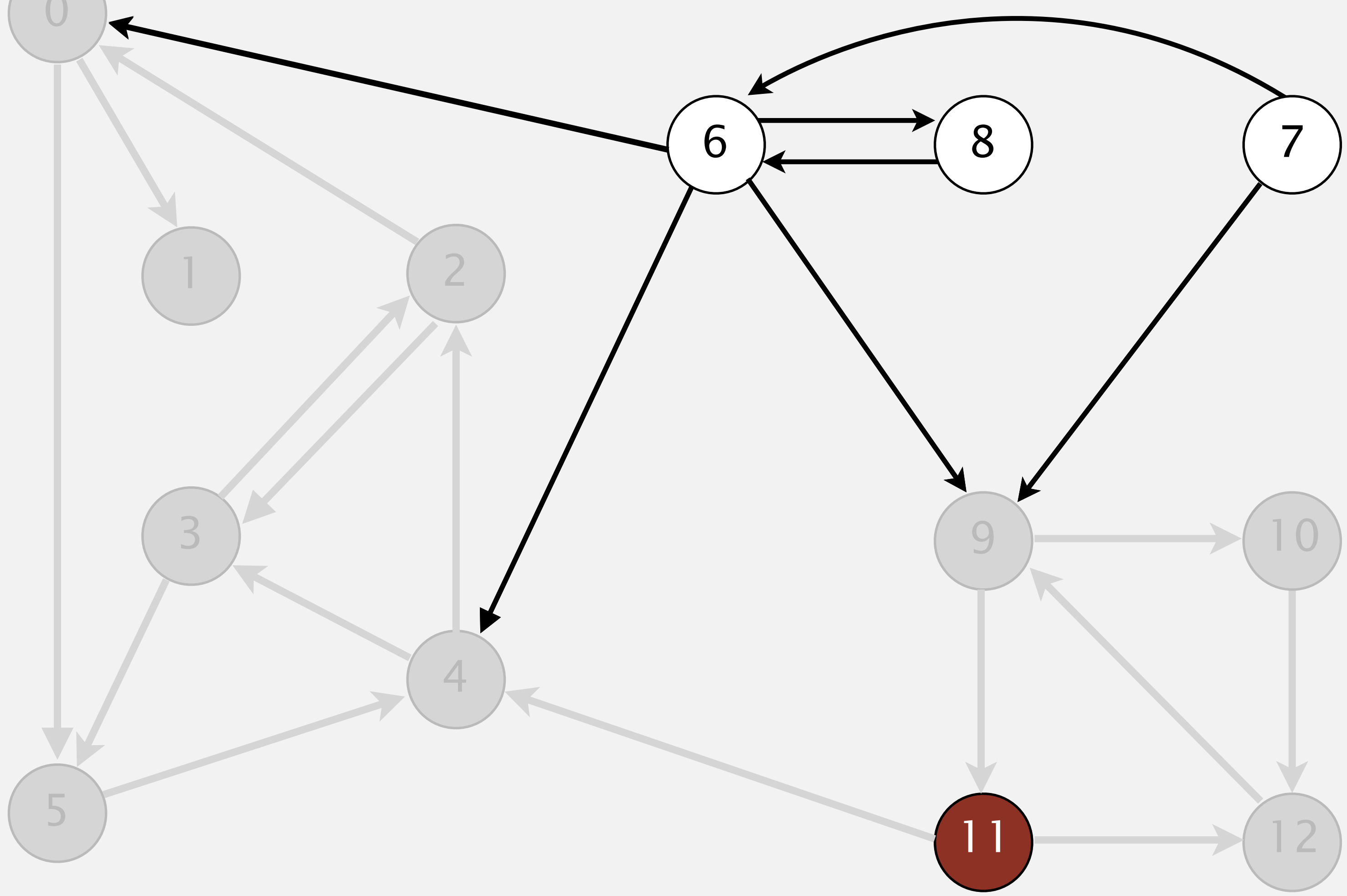
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	–
7	–
8	–
9	2
10	2
11	2
12	2

12 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**



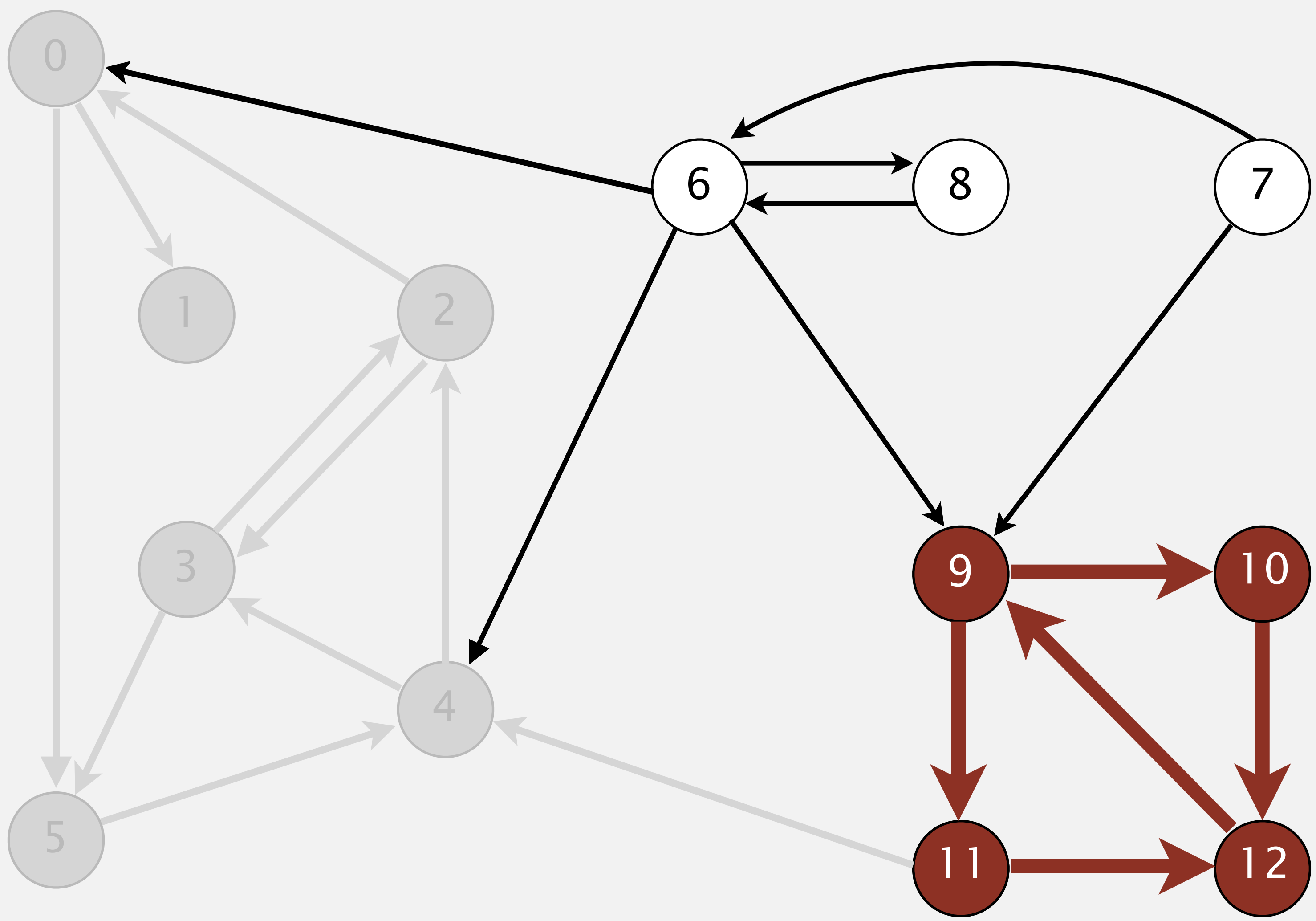
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	–
7	–
8	–
9	2
10	2
11	2
12	2

11 done

Kosaraju-Sharir algorithm demo

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

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



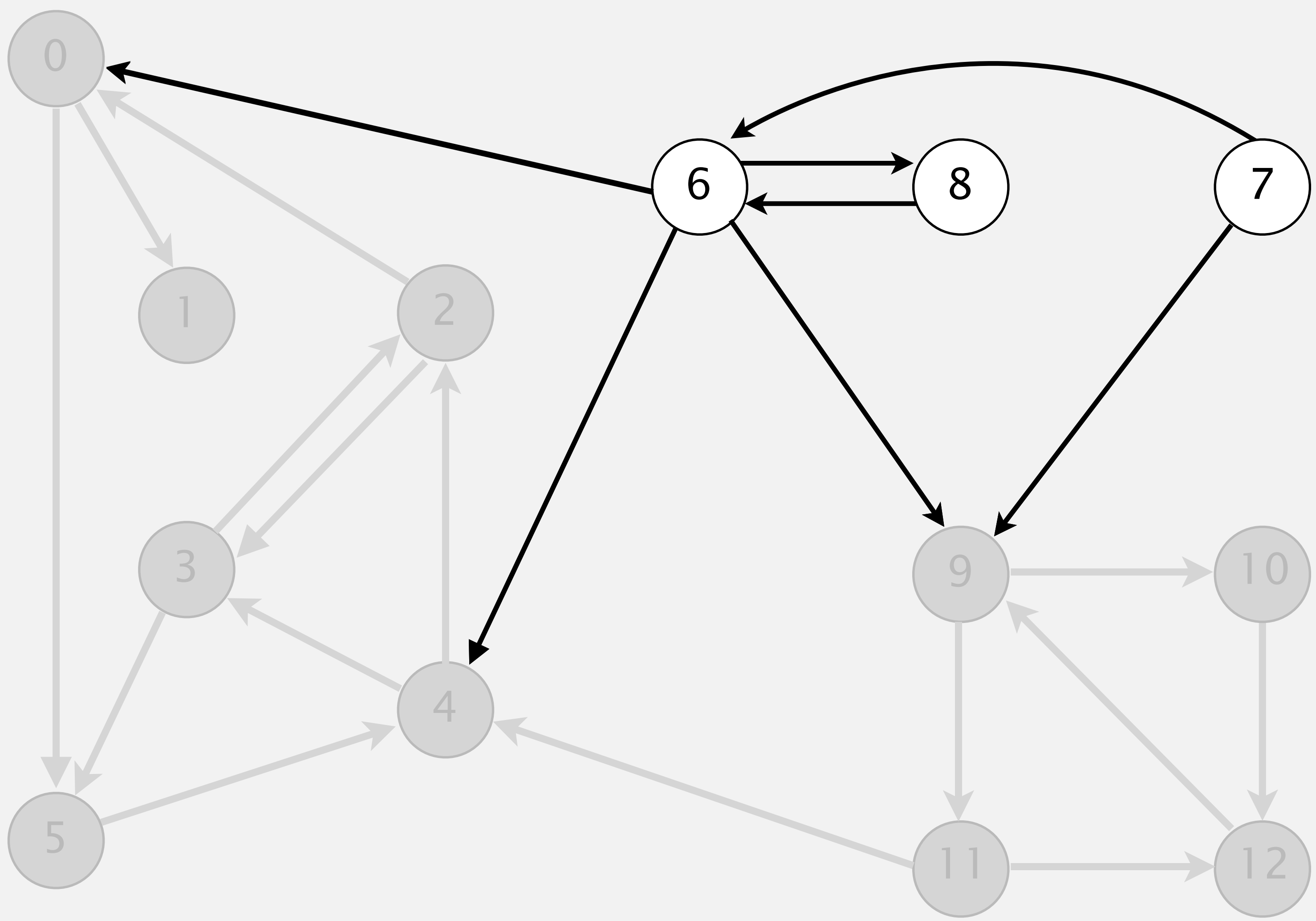
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	–
7	–
8	–
9	2
10	2
11	2
12	2

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.

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



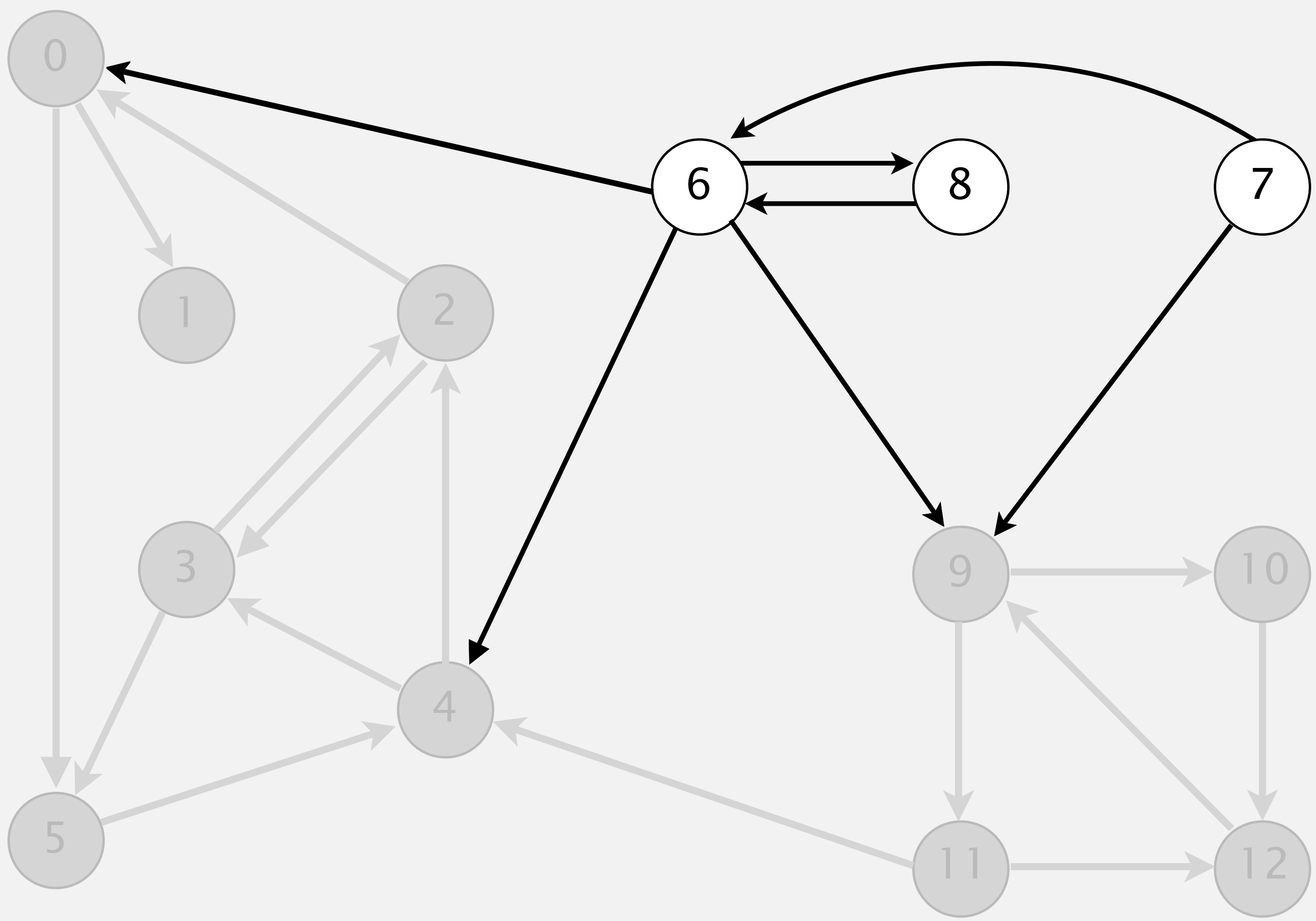
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	–
7	–
8	–
9	2
10	2
11	2
12	2

check 9

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**



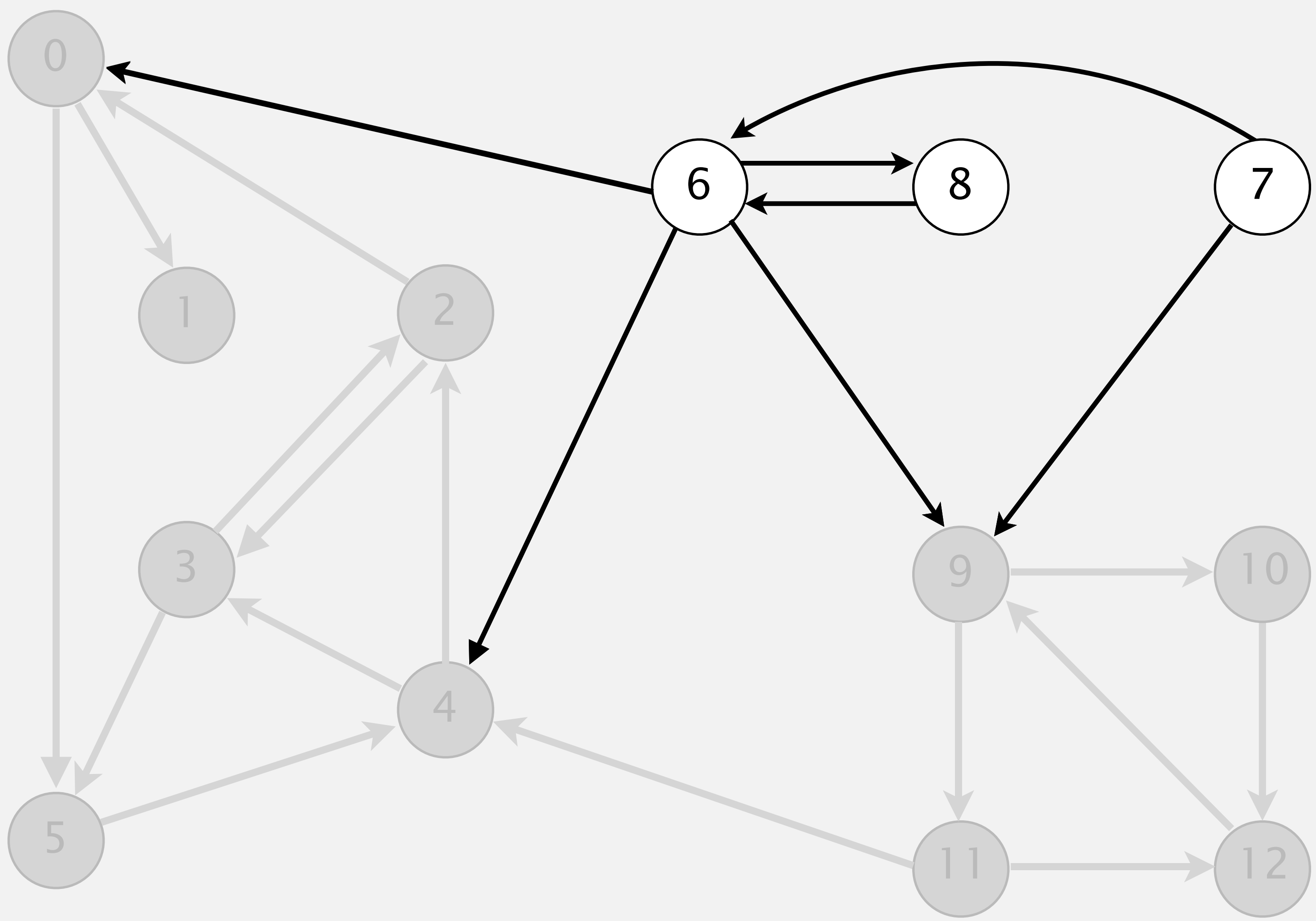
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	–
7	–
8	–
9	2
10	2
11	2
12	2

check 12

Kosaraju-Sharir algorithm demo

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

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



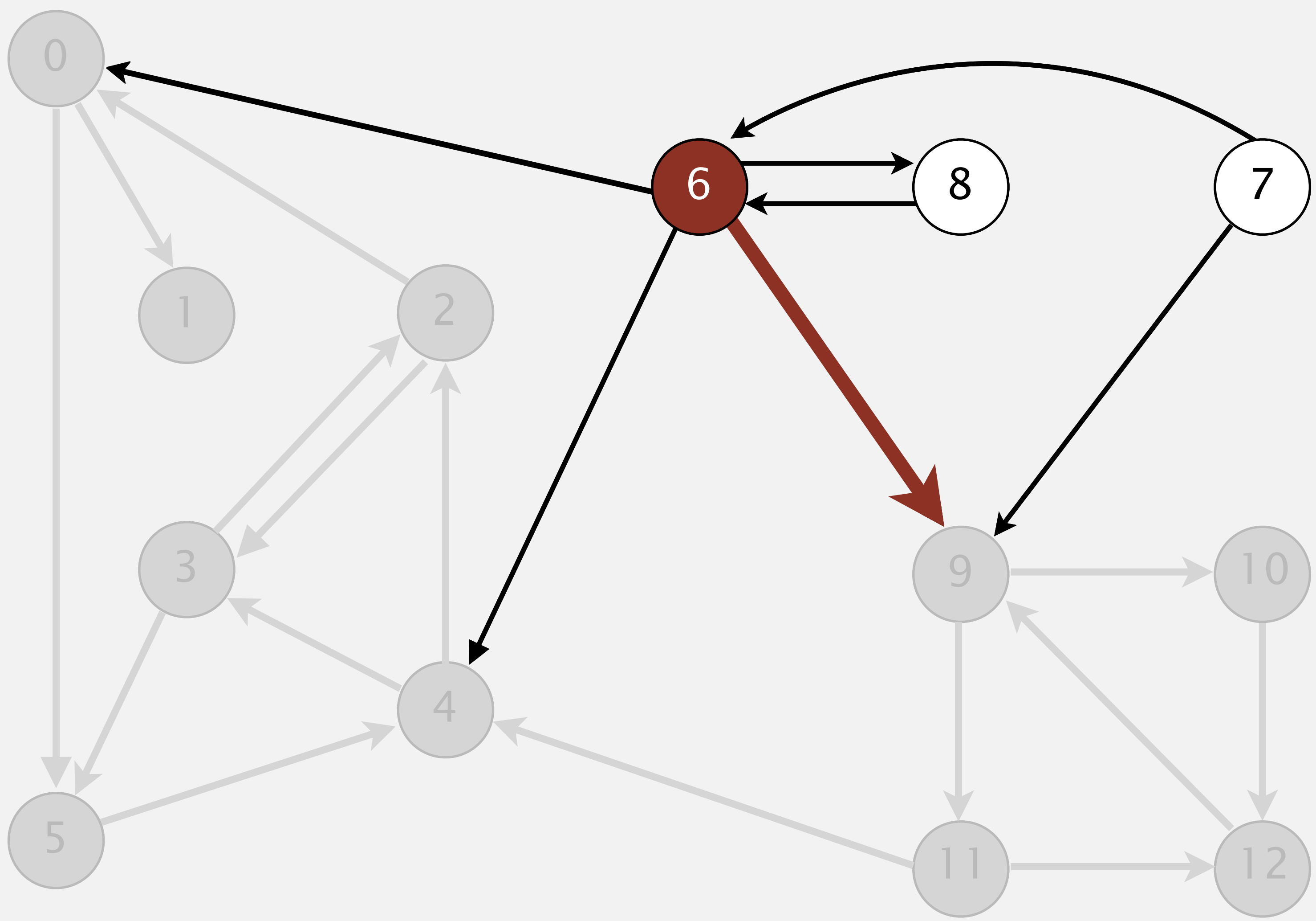
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	–
7	–
8	–
9	2
10	2
11	2
12	2

check 10

Kosaraju-Sharir algorithm demo

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

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



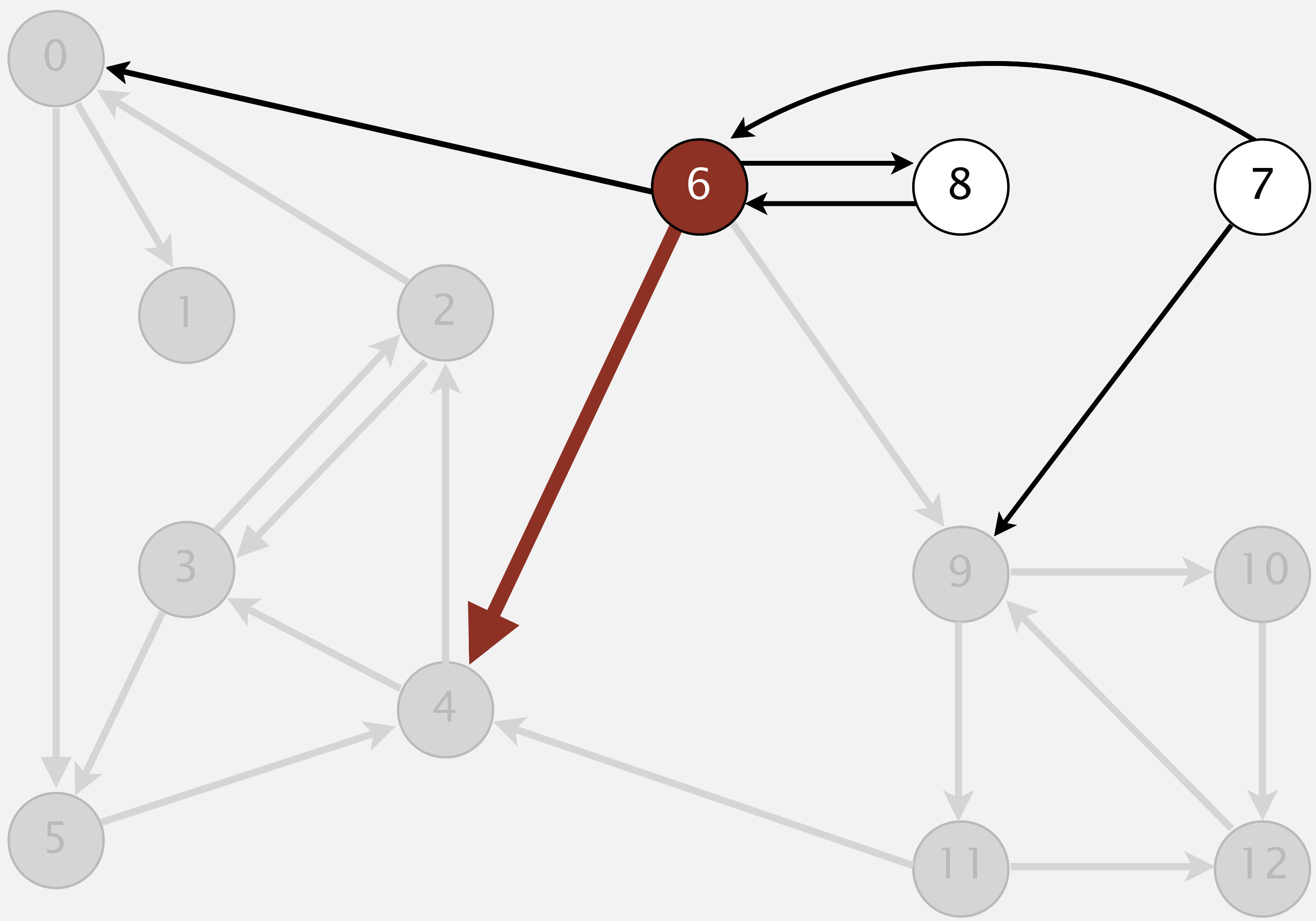
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	–
8	–
9	2
10	2
11	2
12	2

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.

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



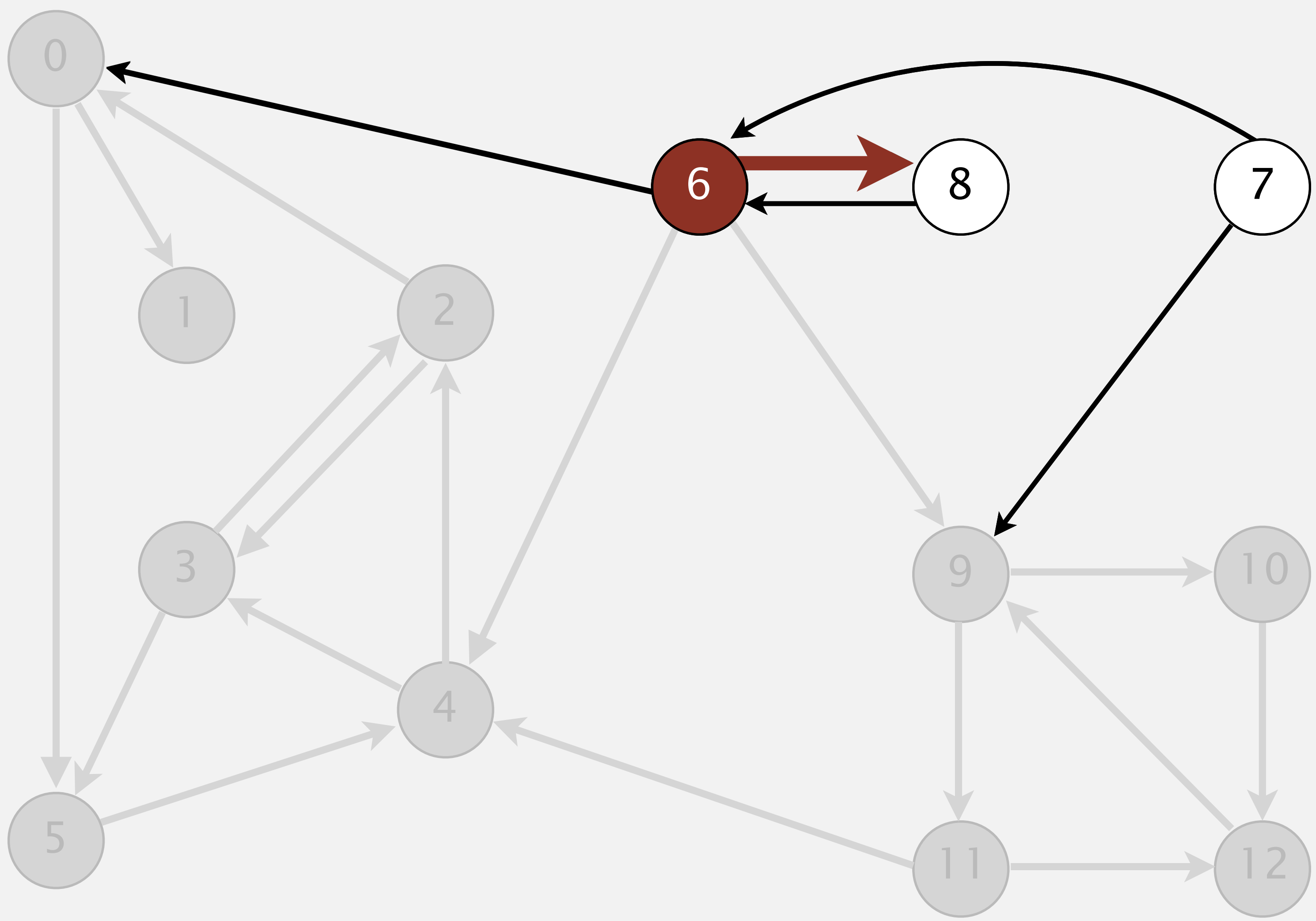
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	–
8	–
9	2
10	2
11	2
12	2

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.

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



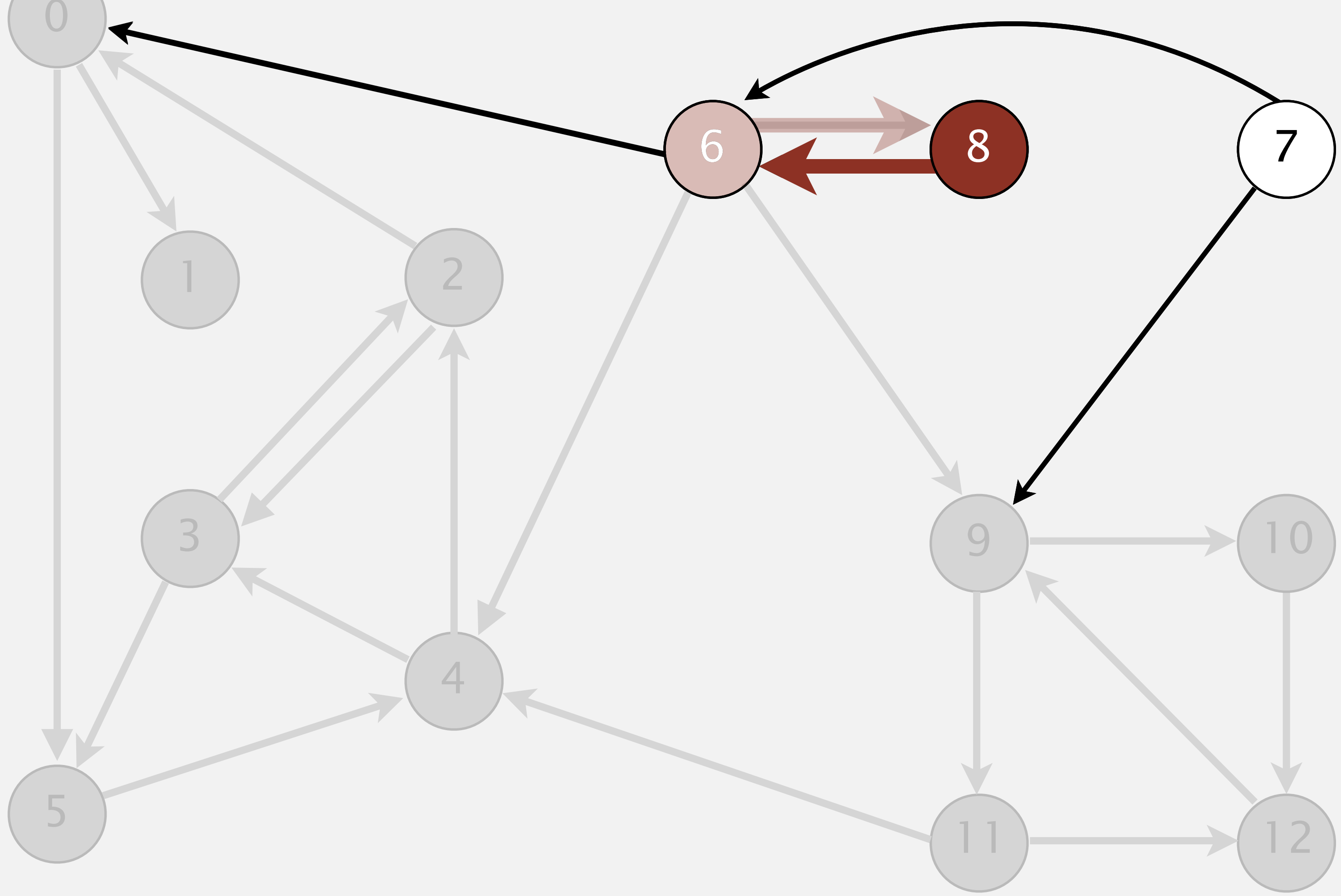
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	–
8	–
9	2
10	2
11	2
12	2

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.

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



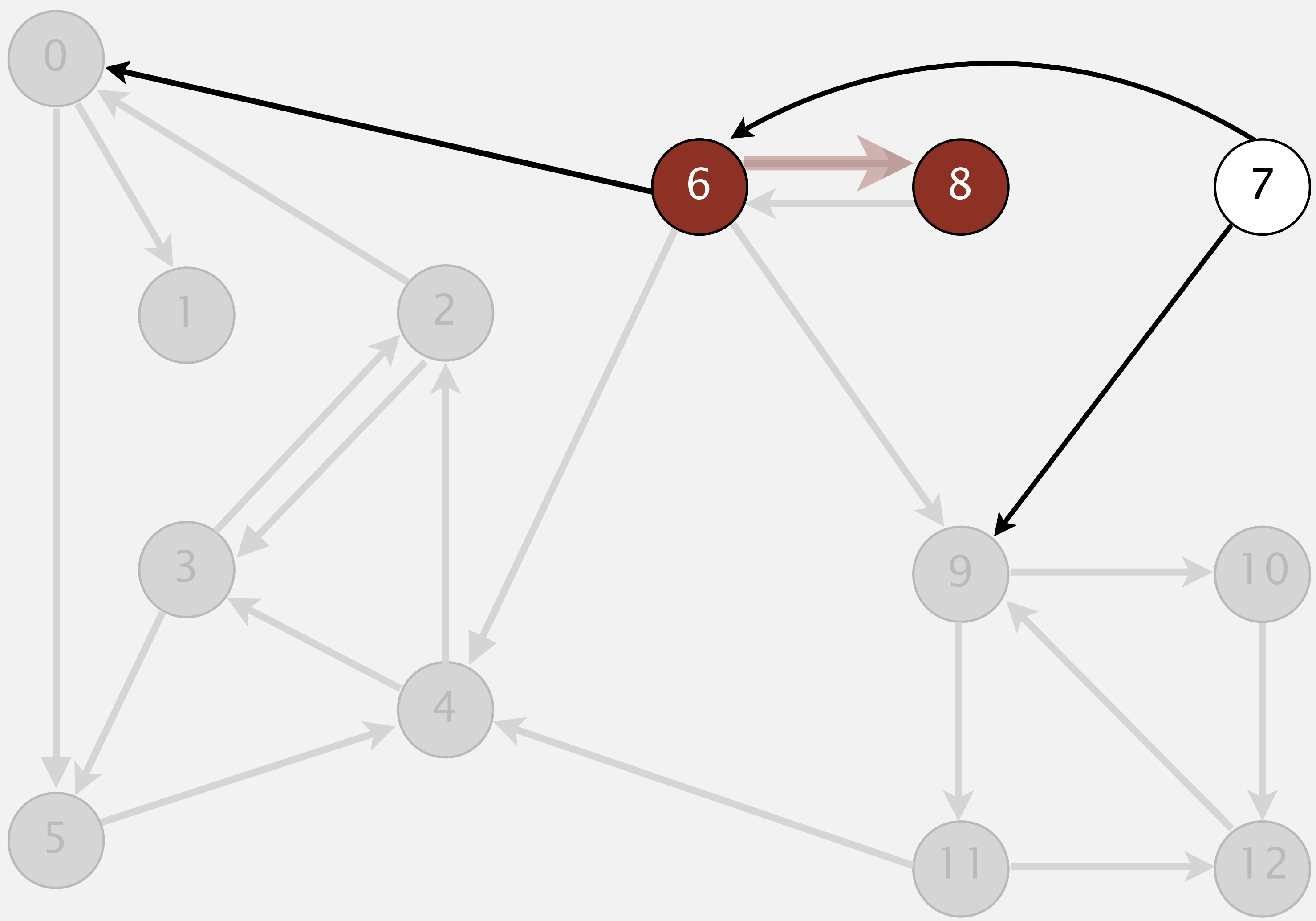
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	–
8	3
9	2
10	2
11	2
12	2

visit 8: check 6

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



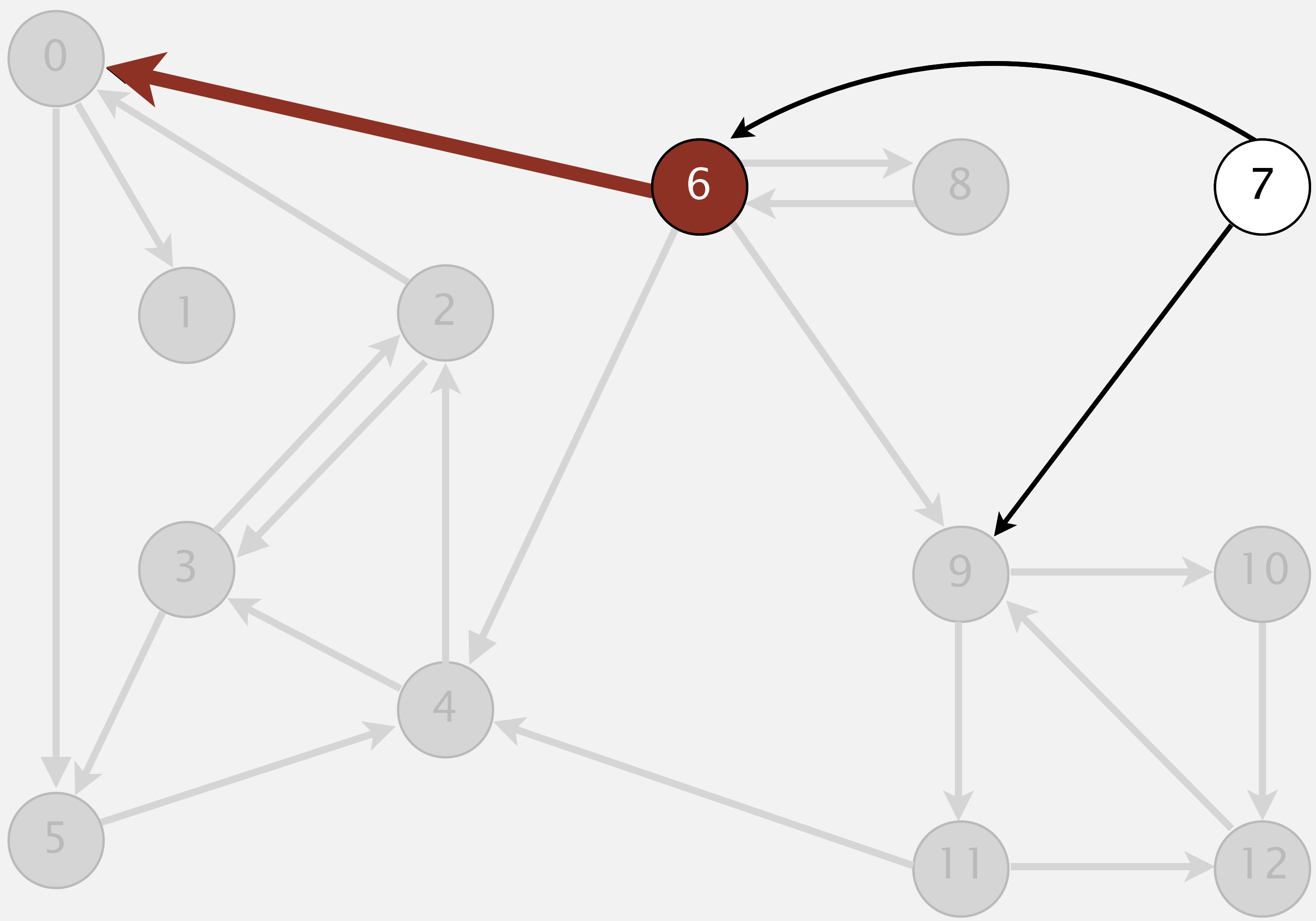
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	–
8	3
9	2
10	2
11	2
12	2

8 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**



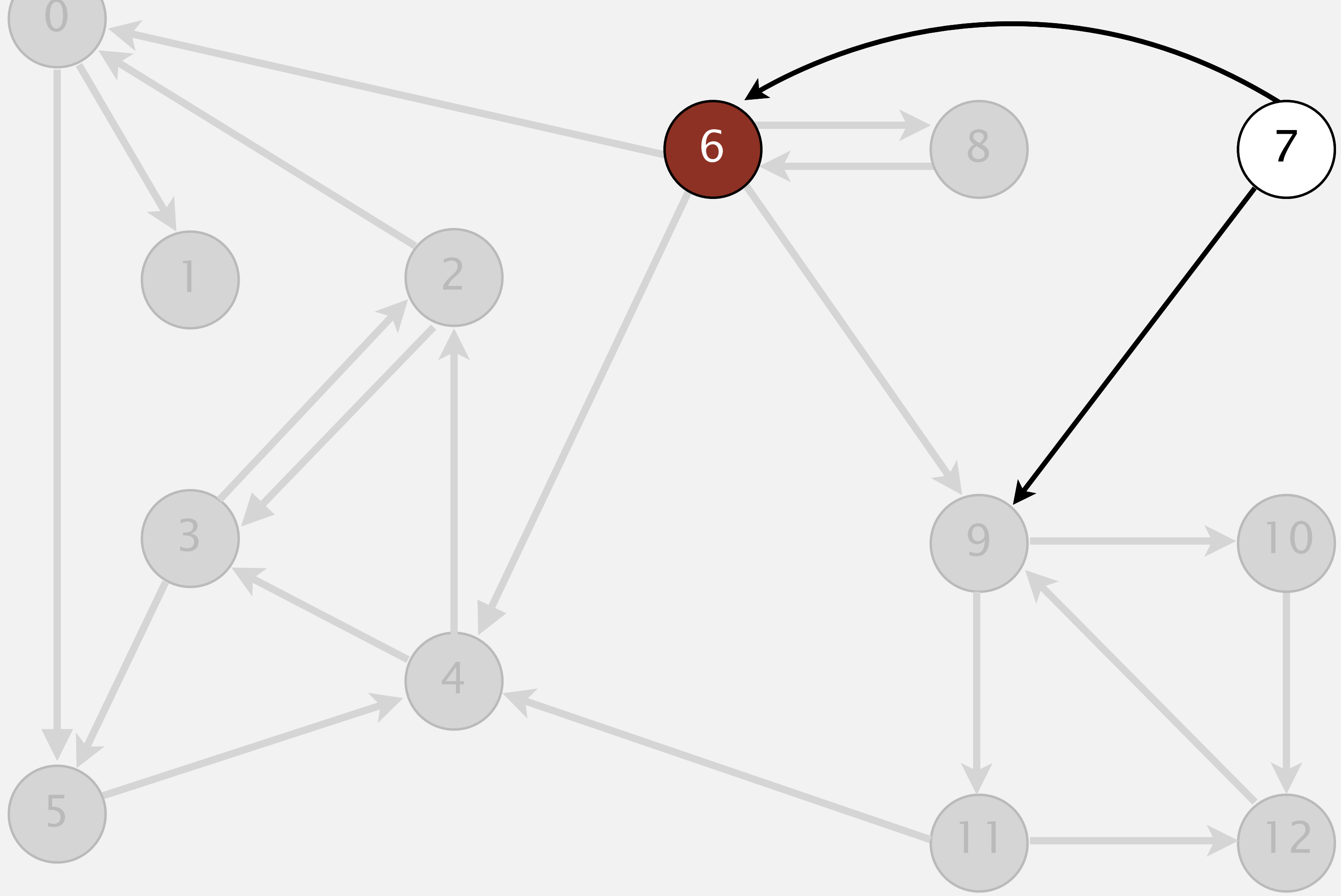
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	–
8	3
9	2
10	2
11	2
12	2

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.

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



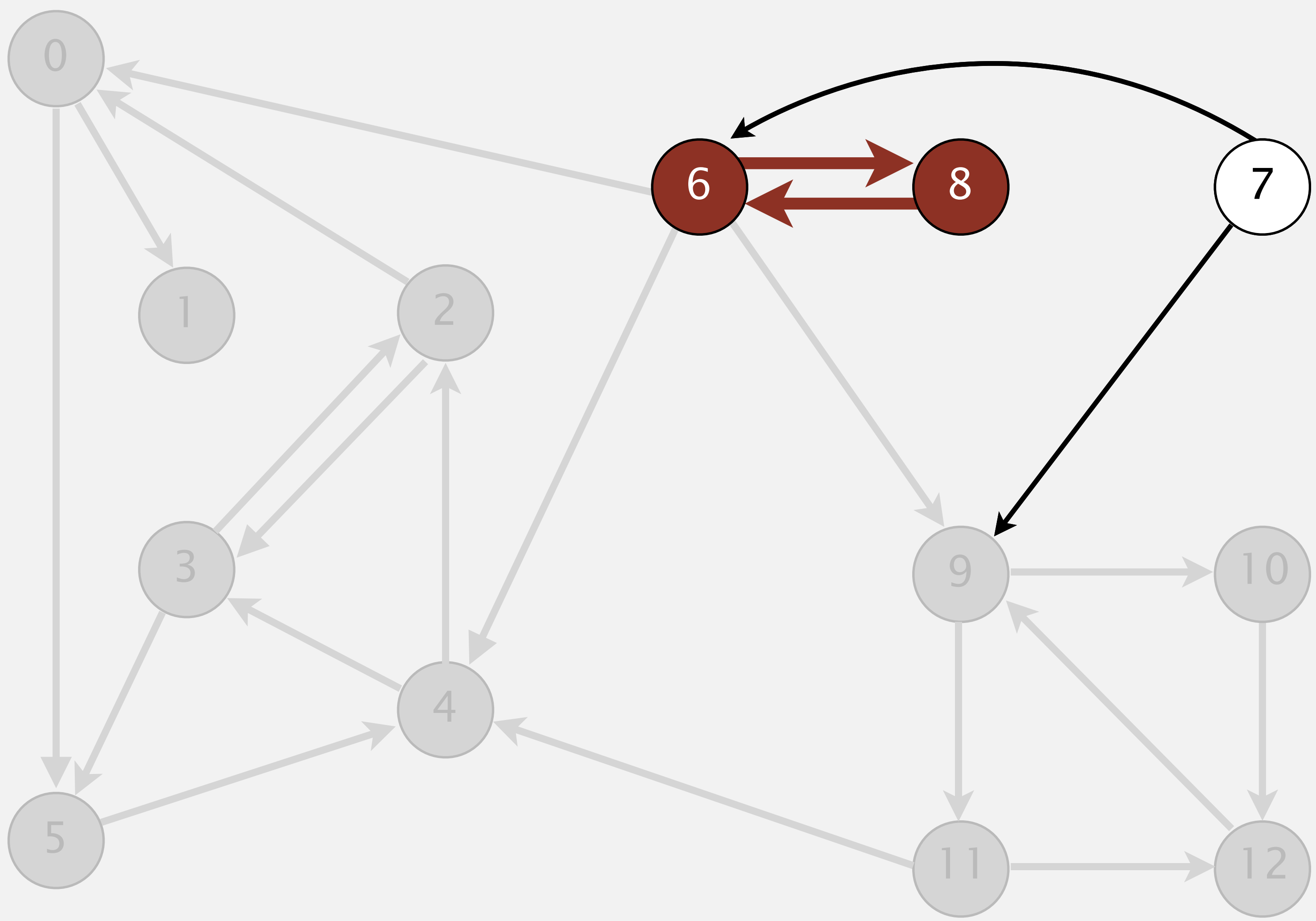
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	–
8	3
9	2
10	2
11	2
12	2

6 done

Kosaraju-Sharir algorithm demo

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

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



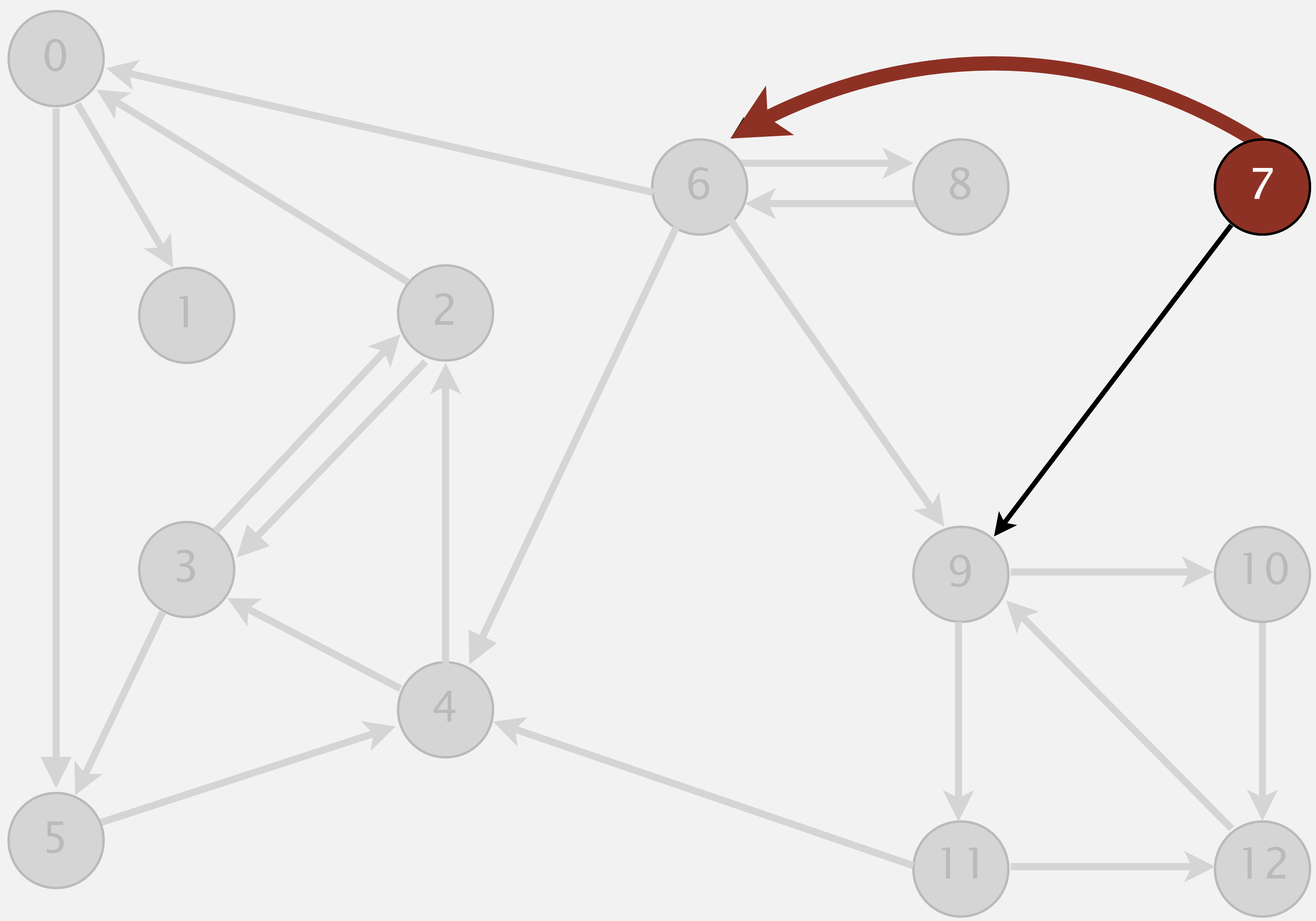
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	–
8	3
9	2
10	2
11	2
12	2

strong component: 6 8

Kosaraju-Sharir algorithm demo

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

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



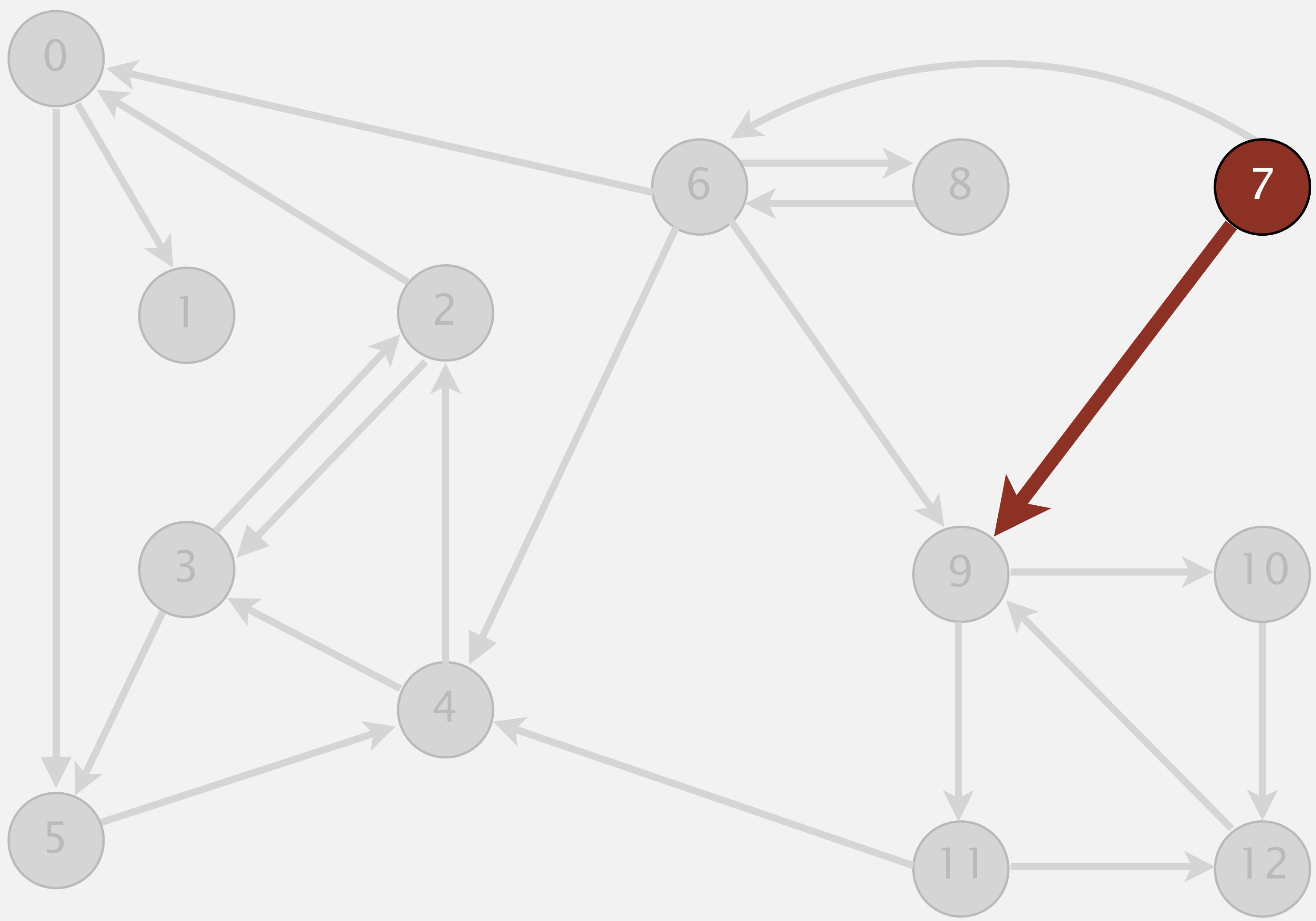
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	4
8	3
9	2
10	2
11	2
12	2

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.

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



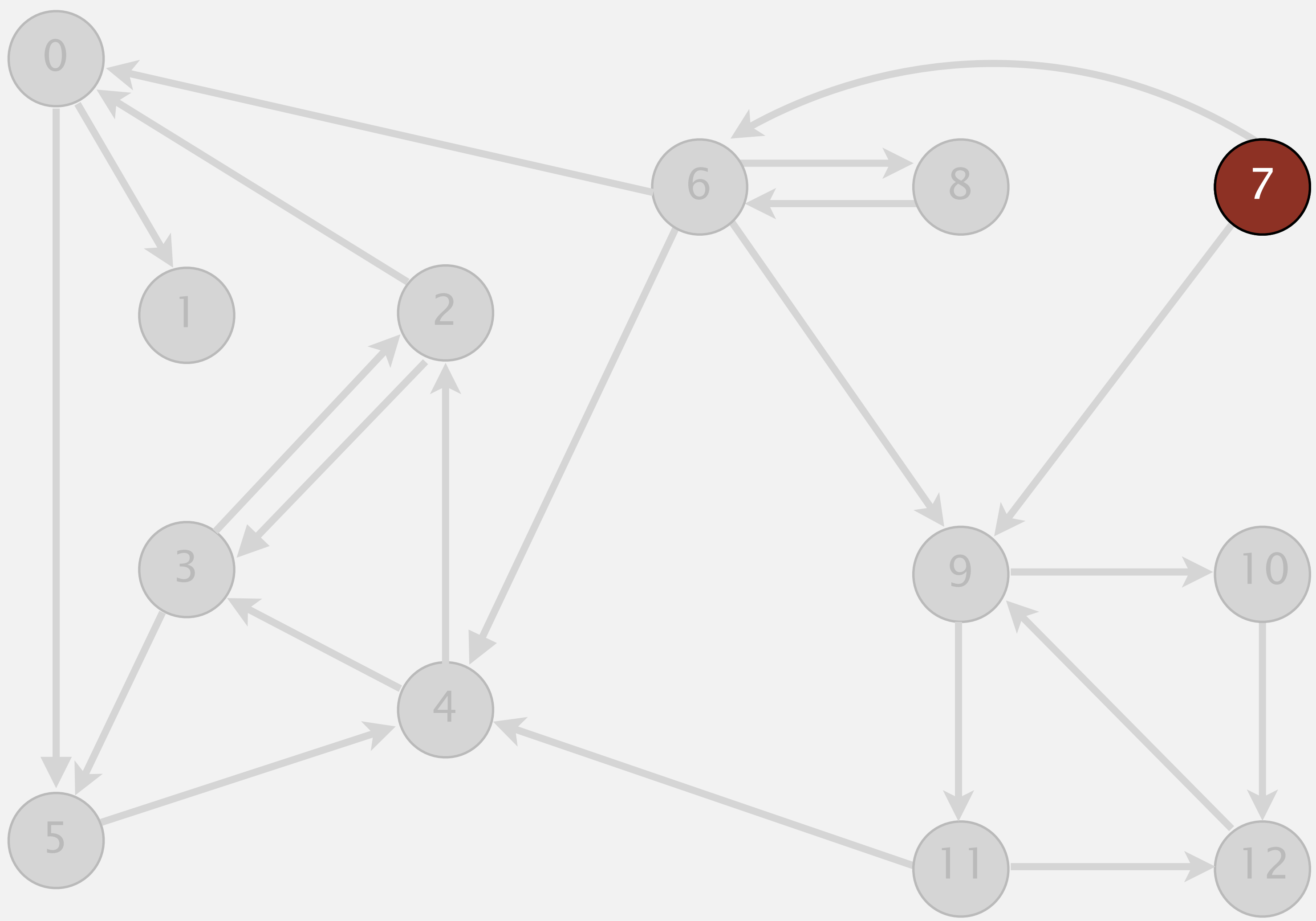
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	4
8	3
9	2
10	2
11	2
12	2

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.

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



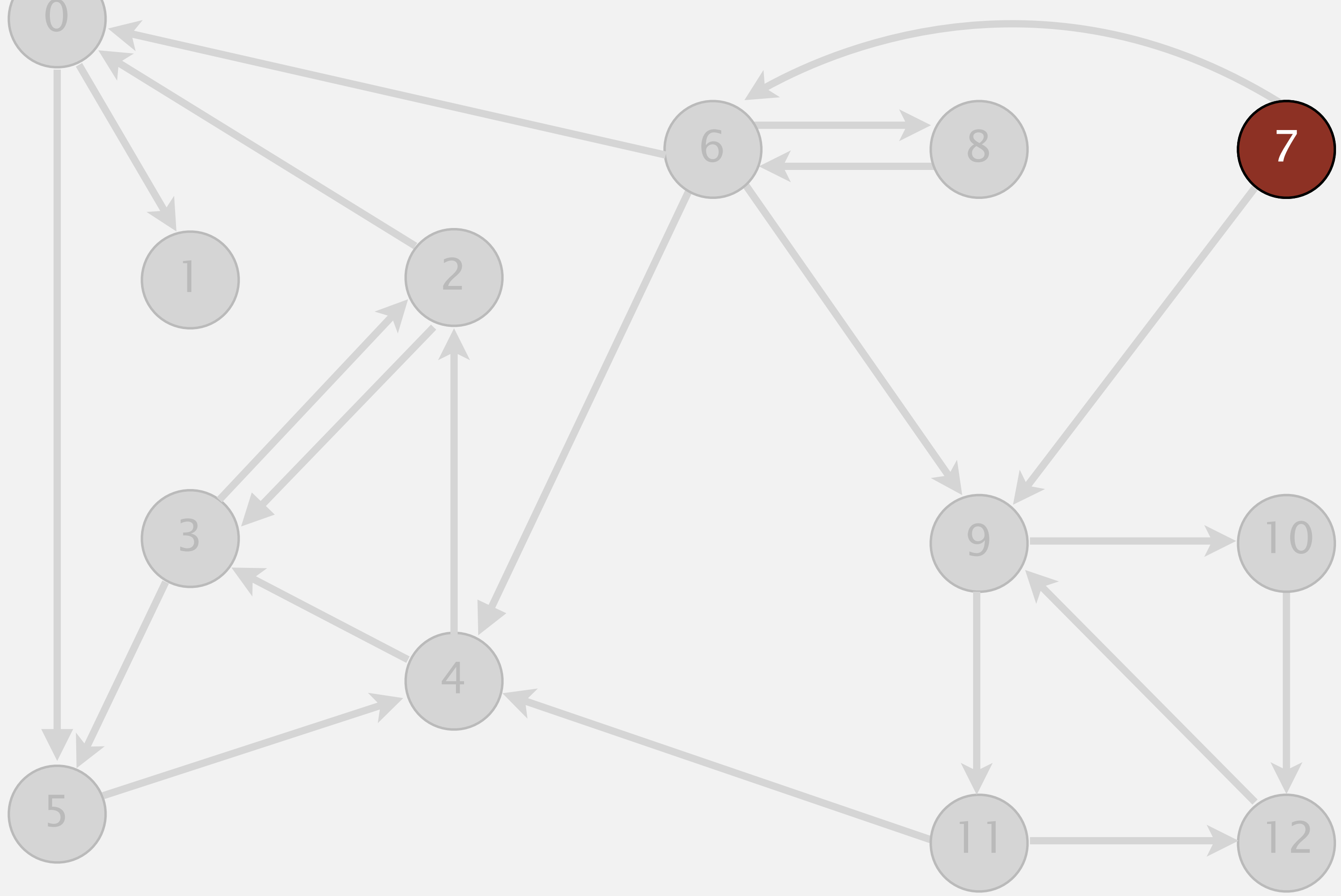
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	4
8	3
9	2
10	2
11	2
12	2

7 done

Kosaraju-Sharir algorithm demo

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

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



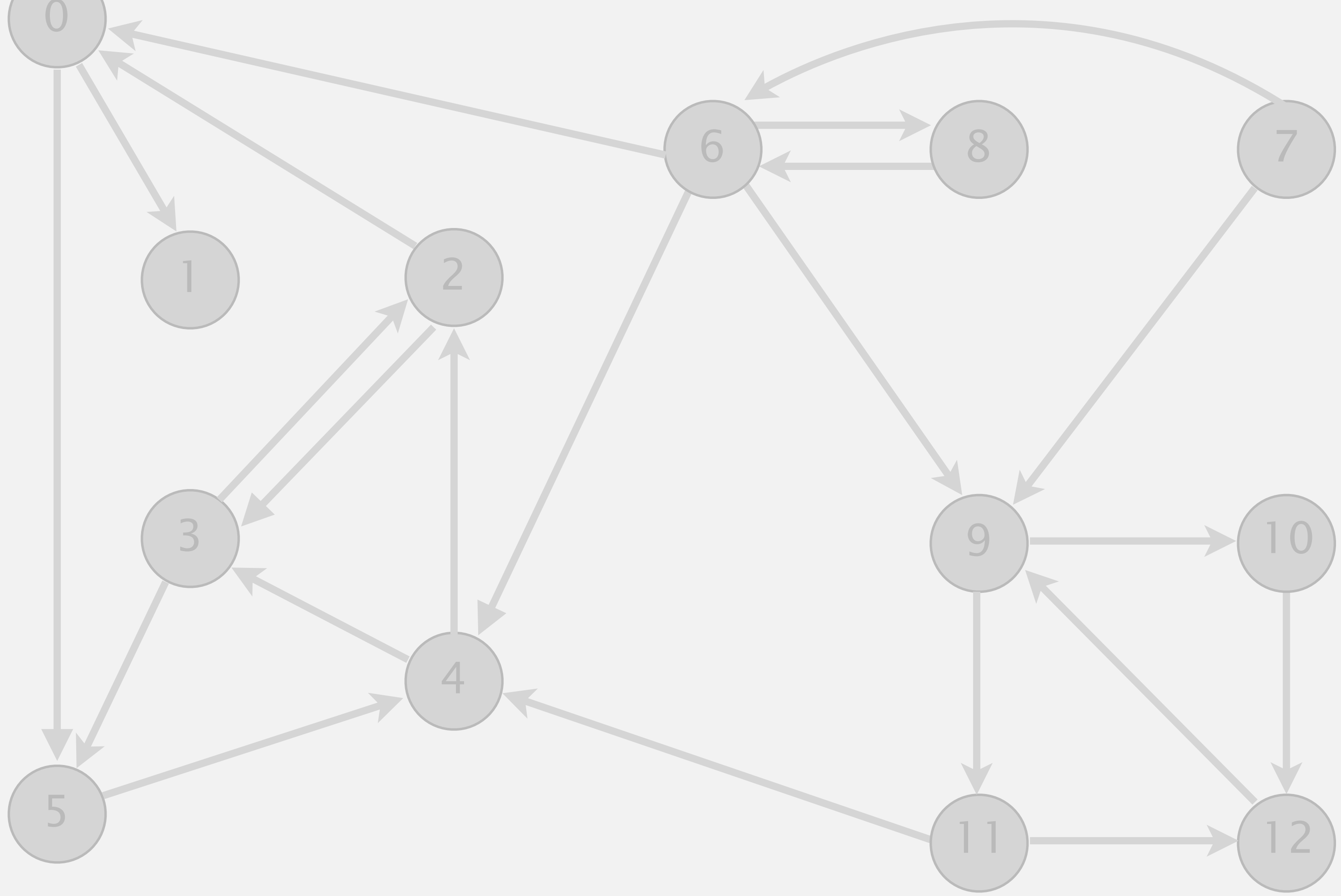
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	4
8	3
9	2
10	2
11	2
12	2

strong component: 7

Kosaraju-Sharir algorithm demo

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

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



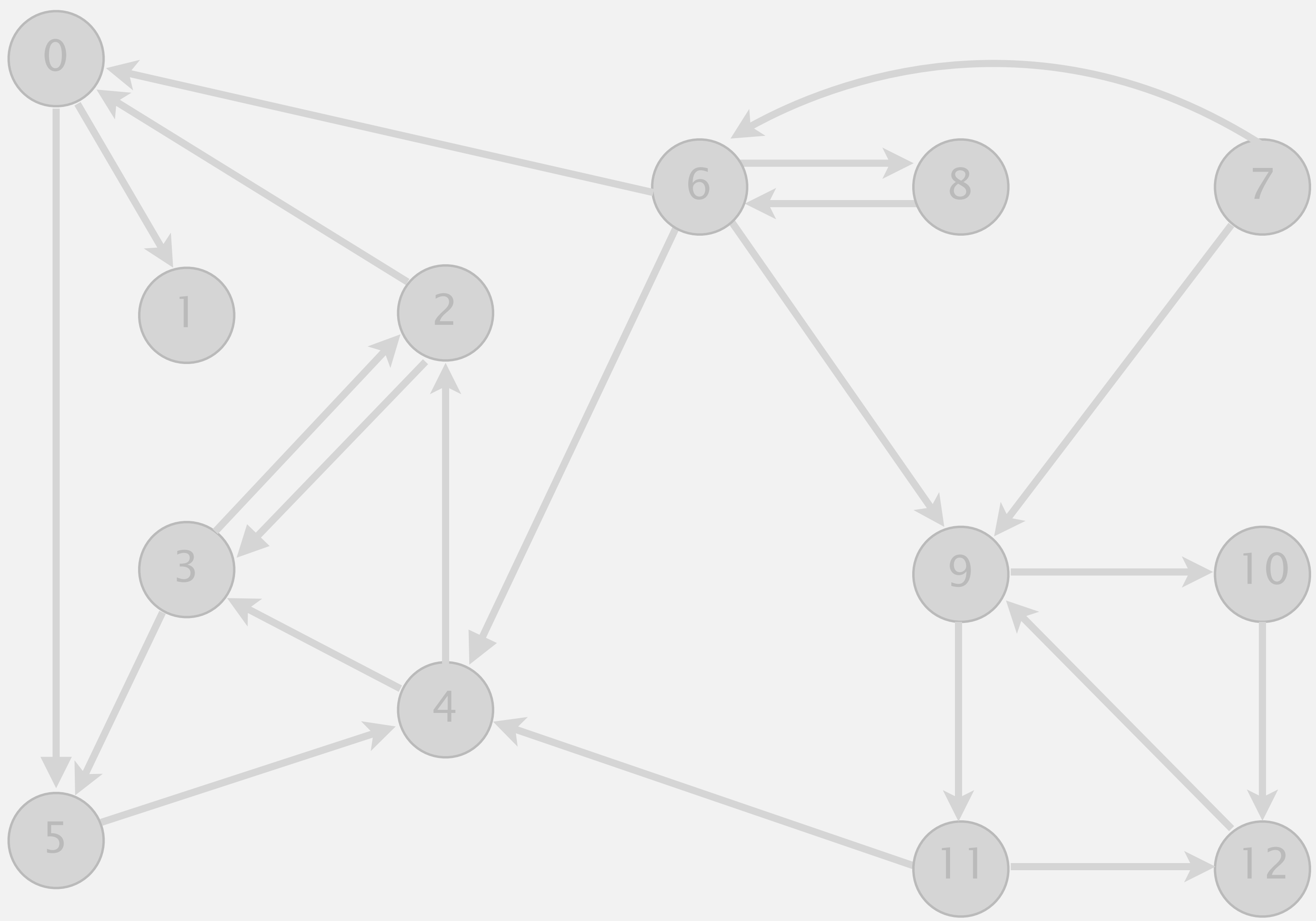
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	4
8	3
9	2
10	2
11	2
12	2

check 8

Kosaraju-Sharir algorithm demo

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

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



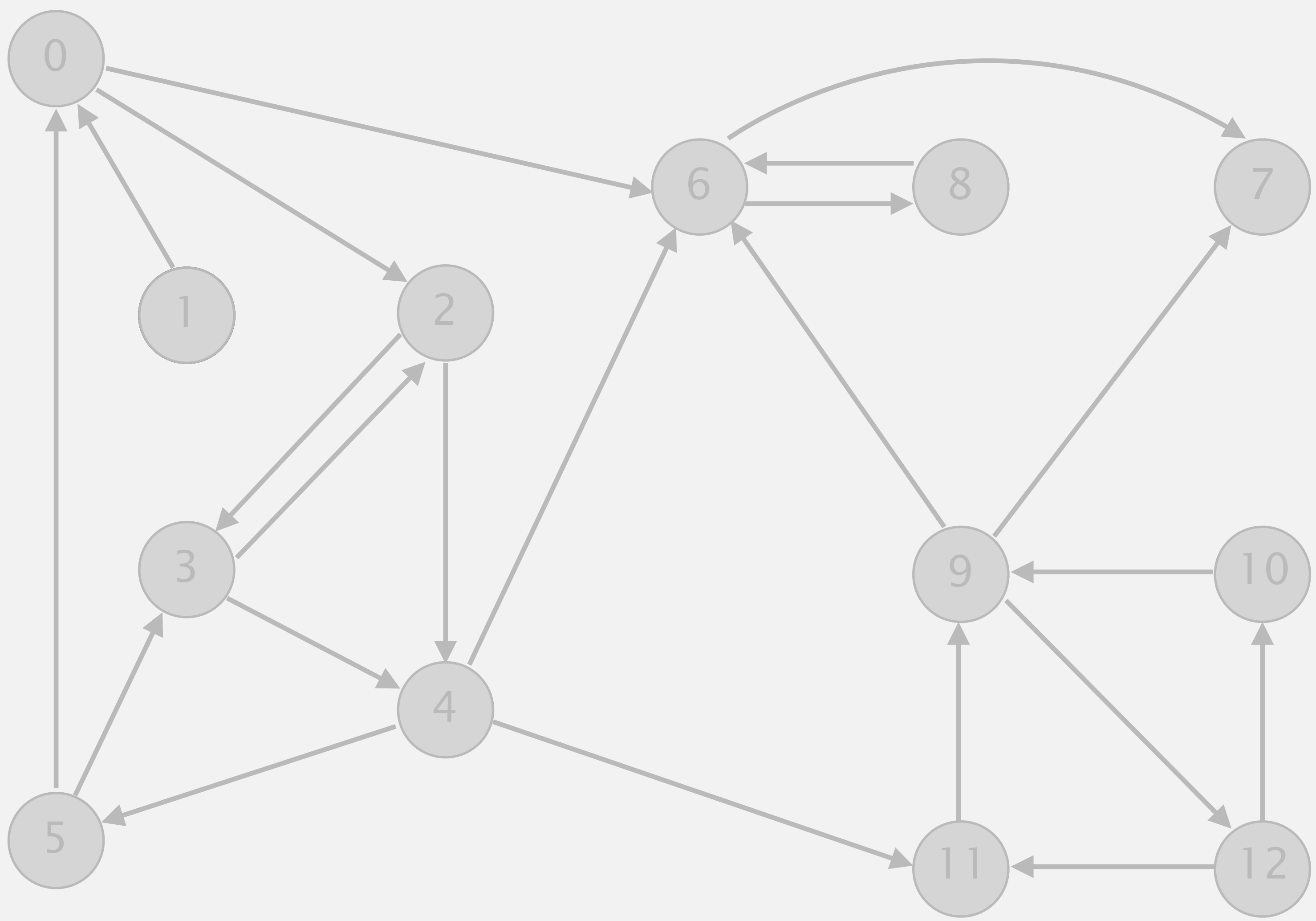
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	4
8	3
9	2
10	2
11	2
12	2

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

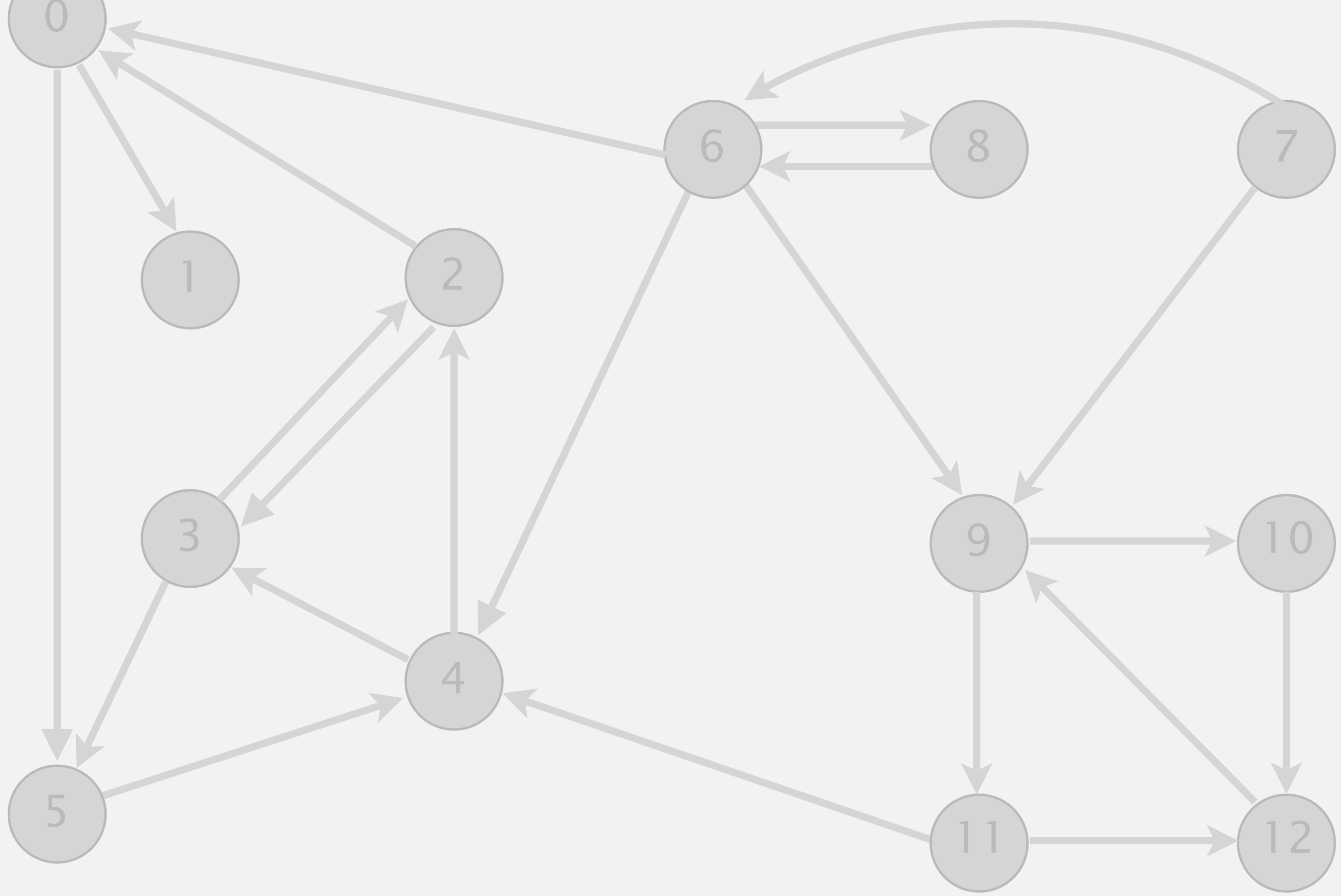


reverse digraph G^R

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	3
7	4
8	3
9	2
10	2
11	2
12	2

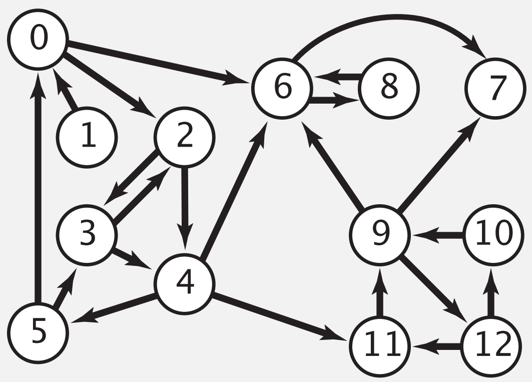
done

Kosaraju-Sharir algorithm

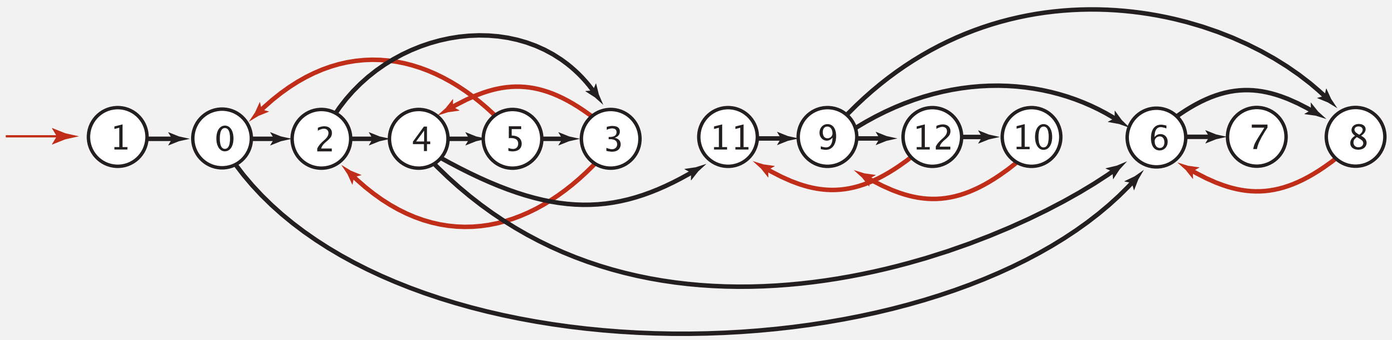
Simple (but mysterious) algorithm for computing strong components.

- 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 reverse digraph G^R



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



reverse postorder for use in second `dfs()`
1 0 2 4 5 3 11 9 12 10 6 7 8

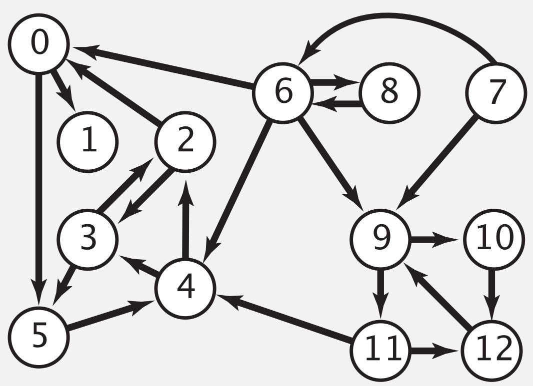
```
dfs(0)
| dfs(6)
| | dfs(8)
| | | check 6
| | 8 done
| | dfs(7)
| | 7 done
| 6 done
| dfs(2)
| | dfs(4)
| | | dfs(11)
| | | | dfs(9)
| | | | | dfs(12)
| | | | | | check 11
| | | | | | dfs(10)
| | | | | | | check 9
| | | | | | 10 done
| | | | | 12 done
| | | | | check 7
| | | | | check 6
...
```

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.

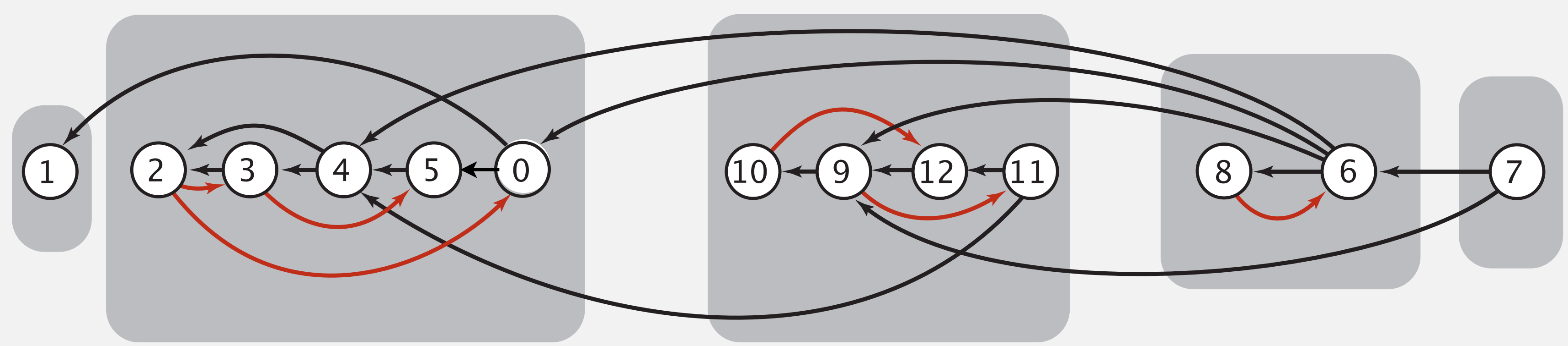
DFS in original digraph G



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 (2nd printing).
- Implementation: easy!

http://algs4.cs.princeton.edu

4.2 DIRECTED GRAPHS

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