BBM 202 - ALGORITHMS

HACETTEPE UNIVERSITY

DEPT. OF COMPUTER ENGINEERING

UNDIRECTED GRAPHS

Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

TODAY

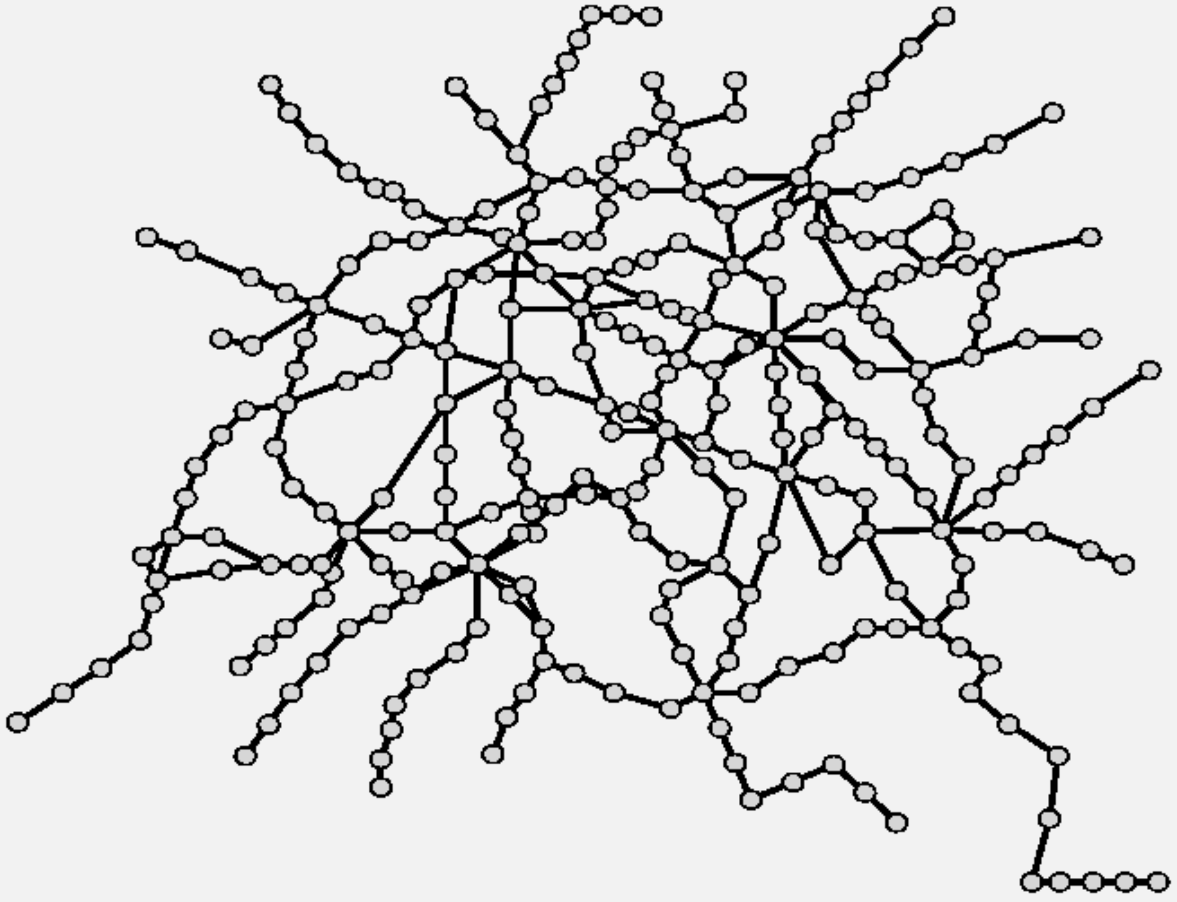
- Undirected Graphs
- Graph API
- Depth-first search
- Breadth-first search
- Connected components
- Challenges

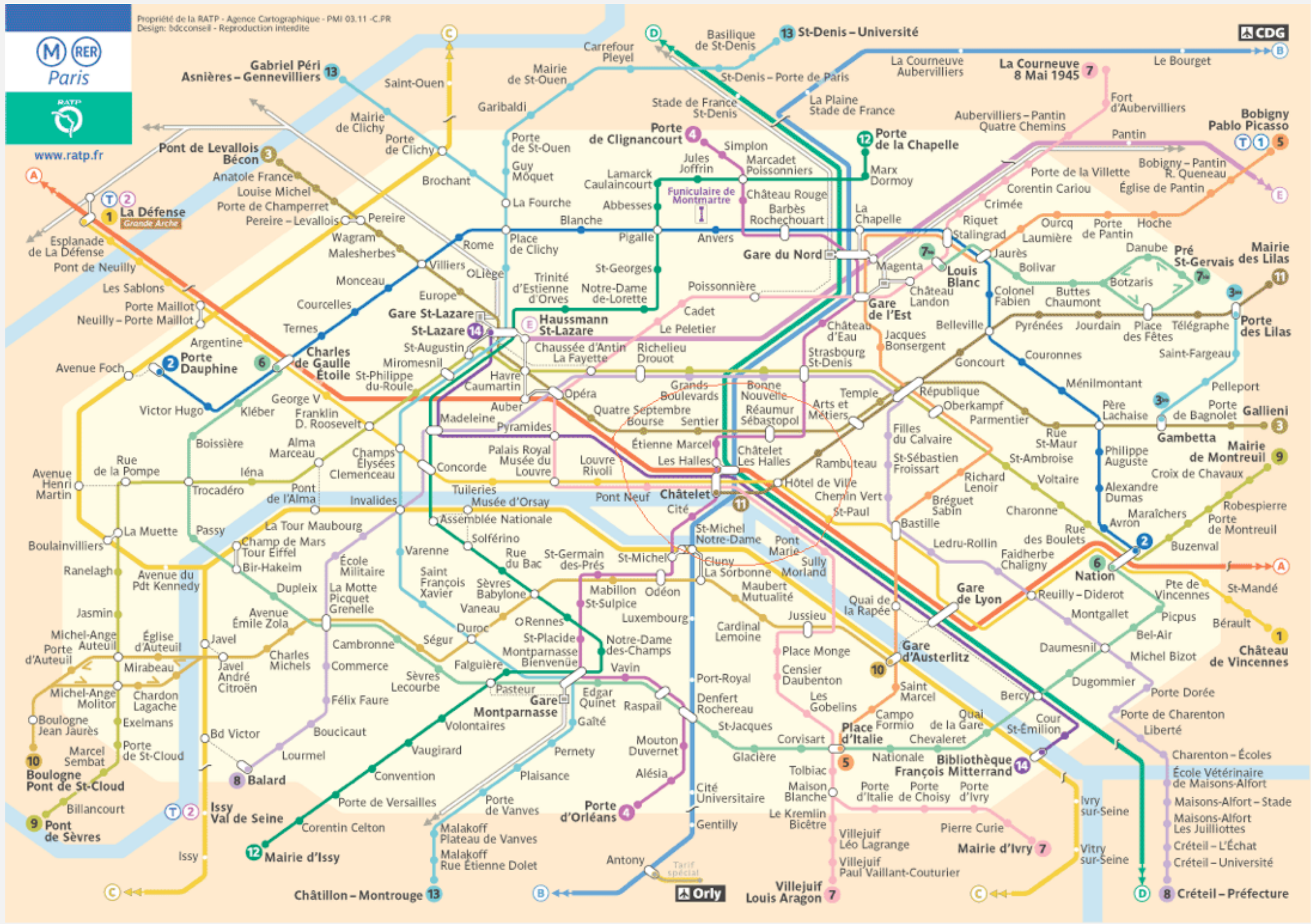
Undirected graphs

Graph. Set of vertices connected pairwise by edges.

Why study graph algorithms?

- Thousands of practical applications.
- Hundreds of graph algorithms known.
- Interesting and broadly useful abstraction.
- Challenging branch of computer science and discrete math.





Graph applications

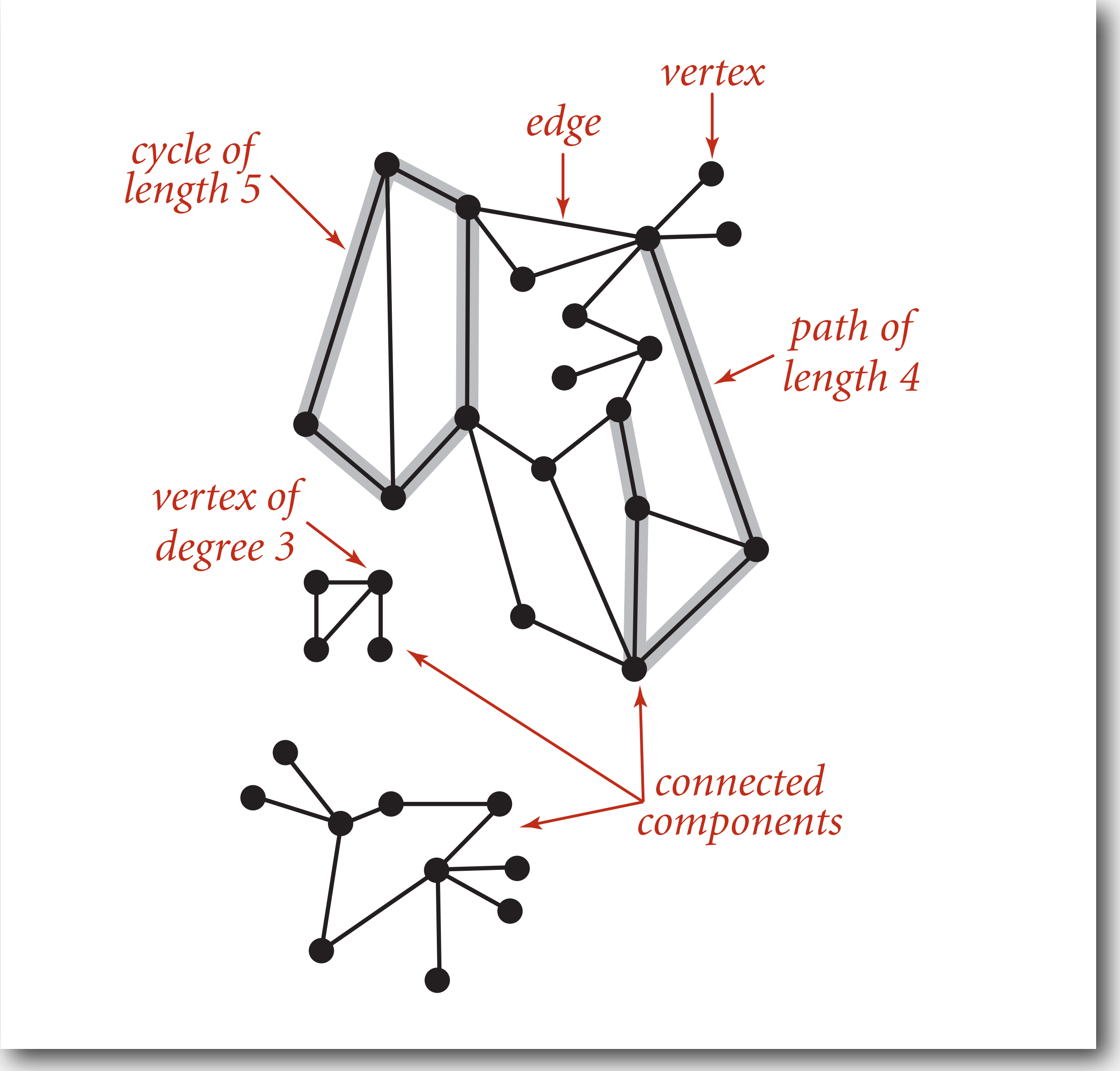
graph	vertex	edge
communication	telephone, computer	fiber optic cable
circuit	gate, register, processor	wire
mechanical	joint	rod, beam, spring
financial	stock, currency	transactions
transportation	street intersection, airport	highway, airway route
internet	class C network	connection
game	board position	legal move
social relationship	person, actor	friendship, movie cast
neural network	neuron	synapse
protein network	protein	protein-protein interaction
chemical compound	molecule	bond

Graph terminology

Path. Sequence of vertices connected by edges.

Cycle. Path whose first and last vertices are the same.

Two vertices are connected if there is a path between them.



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

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

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

MST. 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?

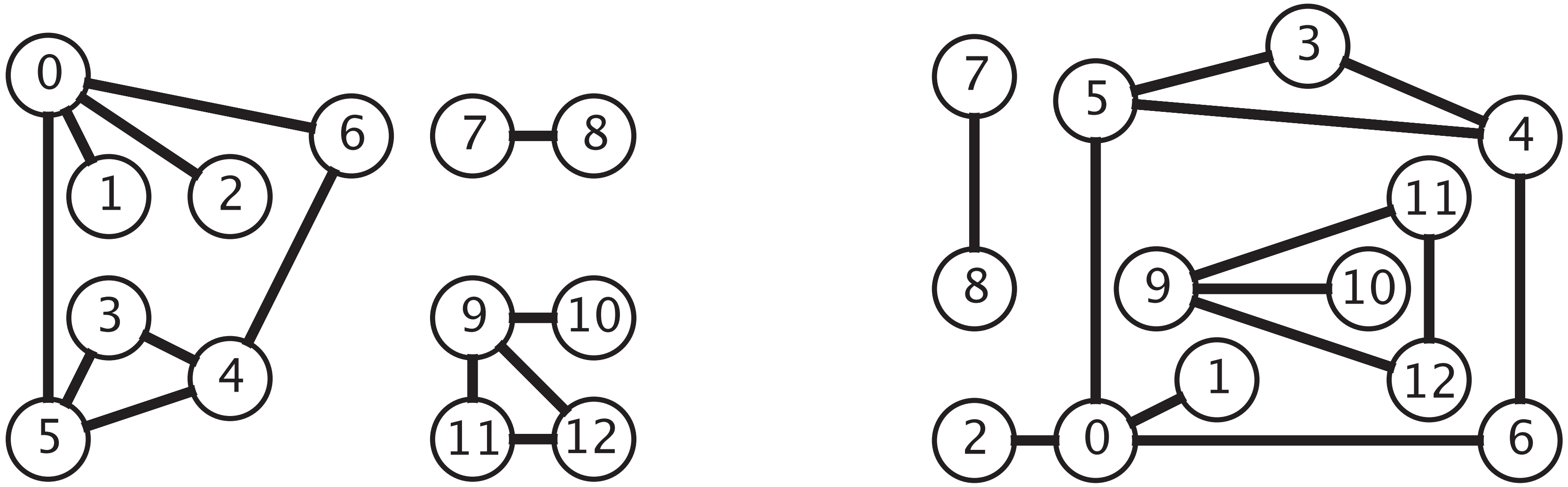
Challenge. Which of these problems are easy? difficult? intractable?

Undirected Graphs

- **Graph API**
- Depth-first search
- Breadth-first search
- Connected components
- Challenges

Graph representation

Graph drawing. Provides intuition about the structure of the graph.



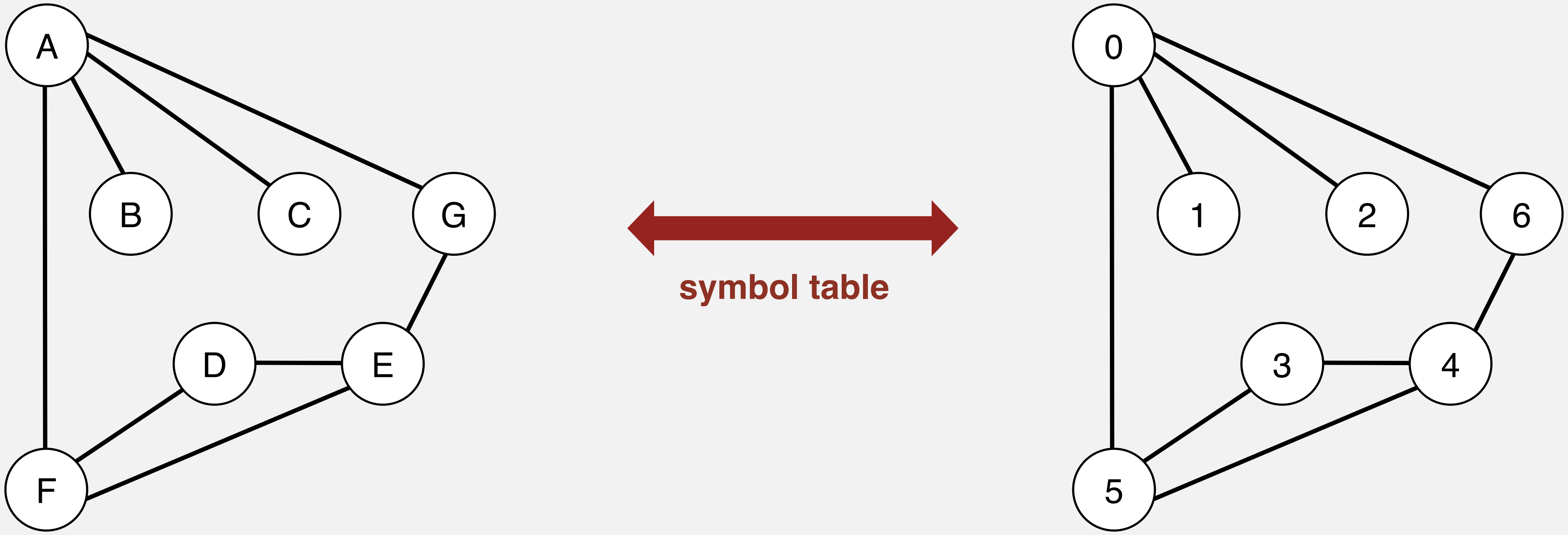
two drawings of the same graph

Caveat. Intuition can be misleading.

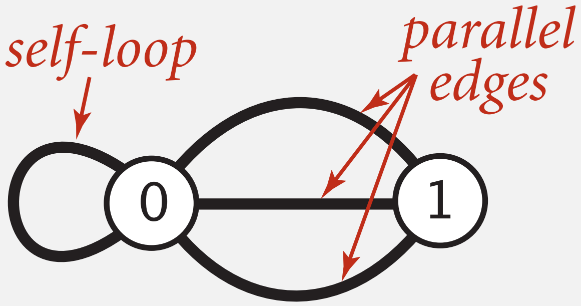
Graph representation

Vertex representation.

- This lecture: use integers between 0 and $V - 1$.
- Applications: convert between names and integers with symbol table.



Anomalies.



Graph API

	public class Graph	
	`Graph(int V)`	*create an empty graph with V vertices*
	`Graph(In in)`	*create a graph from input stream*
`void`	`addEdge(int v, int w)`	*add an edge v-w*
`Iterable<Integer>`	`adj(int v)`	*vertices adjacent to v*
`int`	`V()`	*number of vertices*
`int`	`E()`	*number of edges*
`String`	`toString()`	*string representation*

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

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

read graph from input stream

print out each edge (twice)

Graph API: sample client

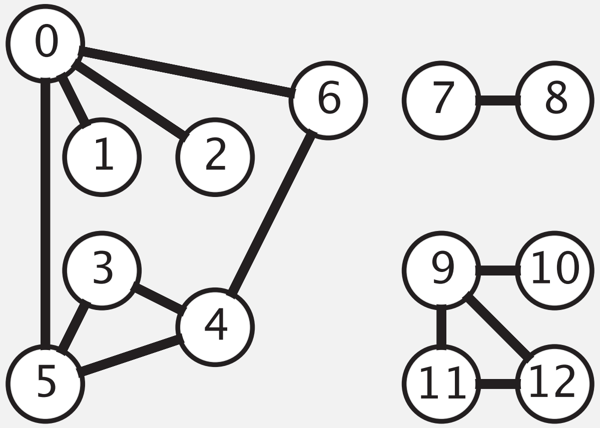
Graph input format.

tinyG.txt

V → 13

E → 13

```
13
13
0  5
4  3
0  1
9  12
6  4
5  4
0  2
11  12
9  10
0  6
7  8
9  11
5  3
```



```
% java Test tinyG.txt
0-6
0-2
0-1
0-5
1-0
2-0
3-5
3-4
…
12-11
12-9
```

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

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

read graph from input stream

print out each edge (twice)

Typical graph-processing code

compute the degree of v

```
public static int degree(Graph G, int v)
{
   int degree = 0;
   for (int w : G.adj(v)) degree++;
   return degree;
}
```

compute maximum degree

```
public static int maxDegree(Graph G)
{
   int max = 0;
   for (int v = 0; v < G.V(); v++)
      if (degree(G, v) > max)
         max = degree(G, v);
   return max;
}
```

compute average degree

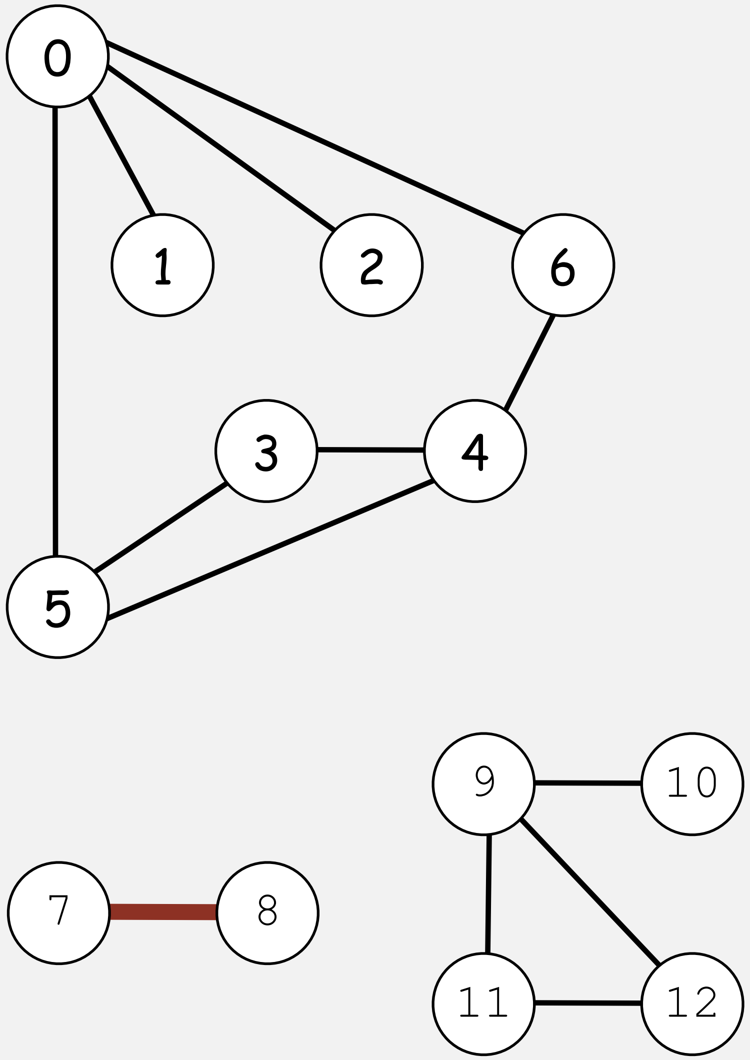
```
public static double averageDegree(Graph G)
{  return 2.0 * G.E() / G.V();  }
```

count self-loops

```
public static int numberOfSelfLoops(Graph G)
{
   int count = 0;
   for (int v = 0; v < G.V(); v++)
      for (int w : G.adj(v))
         if (v == w) count++;
   return count/2;   // each edge counted twice
}
```

Set-of-edges graph representation

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



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

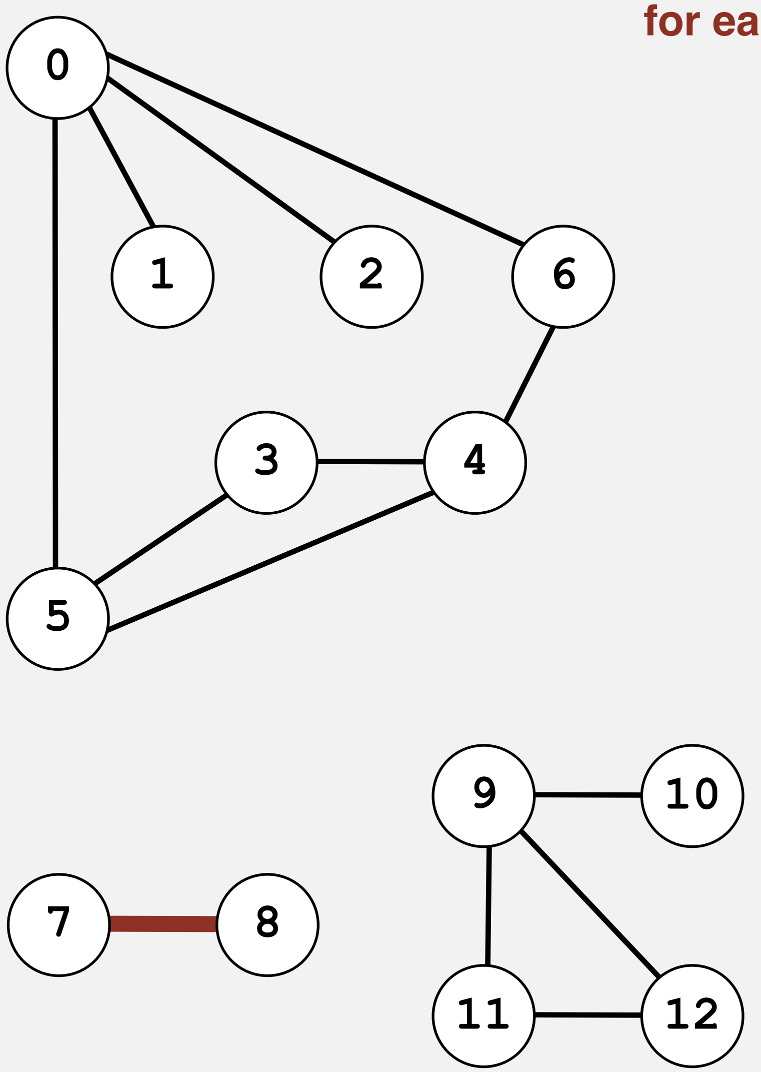
Adjacency-matrix graph representation

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

for each edge v–w in graph: `adj[v][w] = adj[w][v] = true.`

two entries

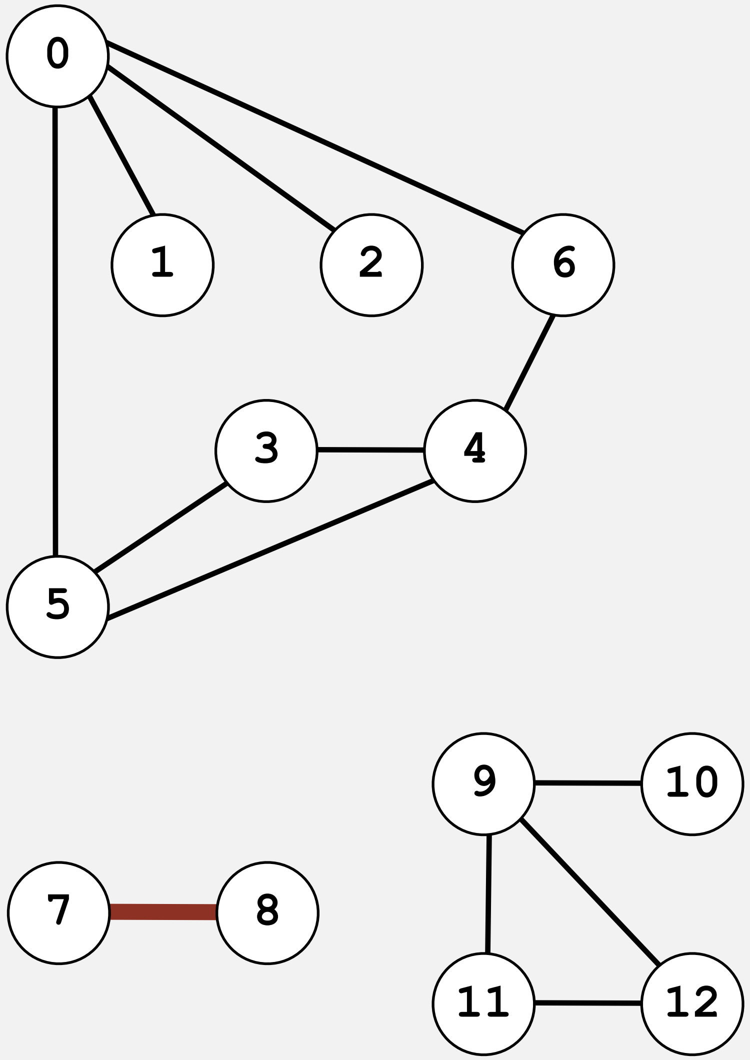
for each edge

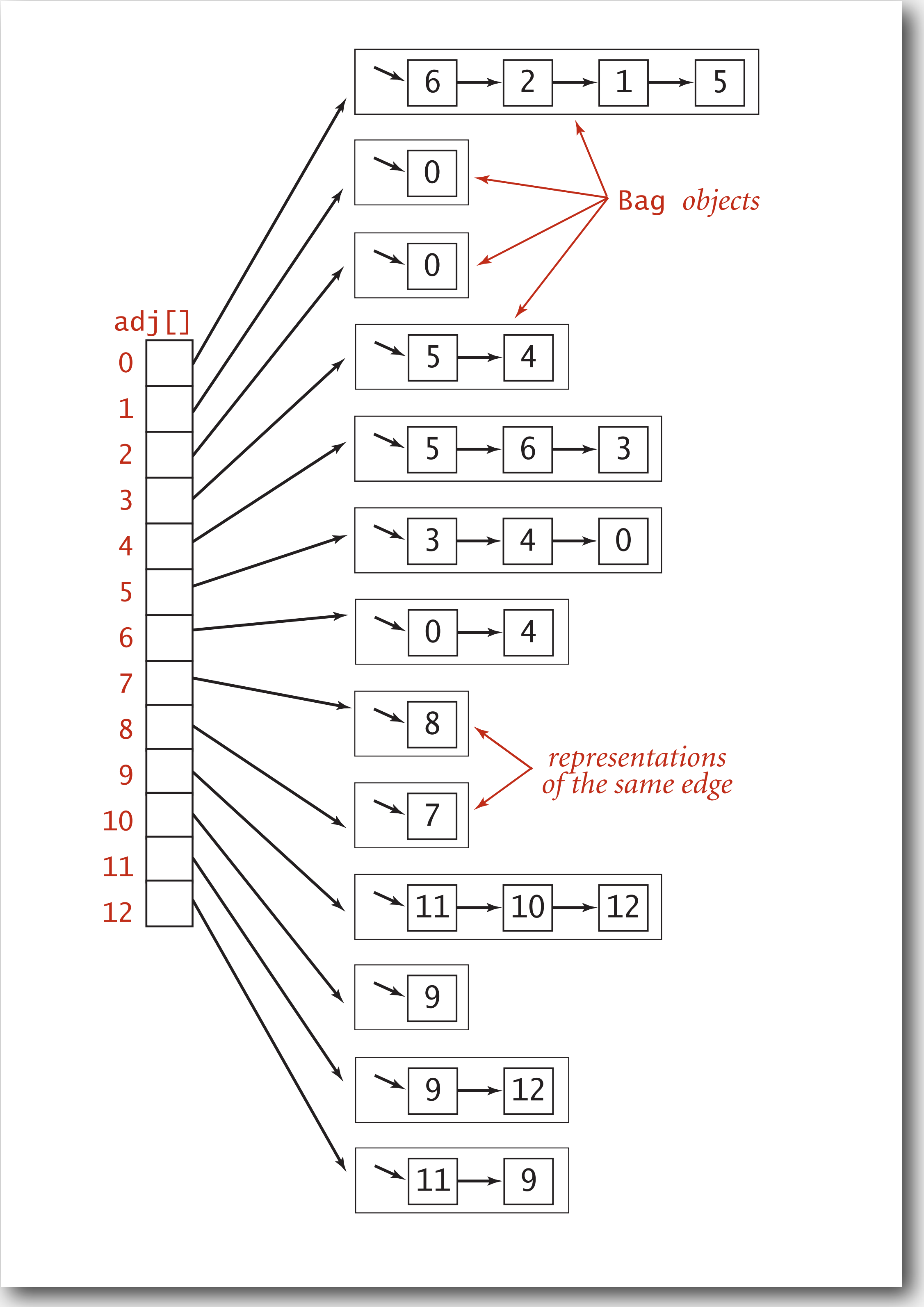


	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0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

Adjacency-list graph representation

Maintain vertex-indexed array of lists.





Adjacency-list graph representation: Java implementation

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

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

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

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

adjacency lists

(using `Bag` data type)

create empty graph

with `V` vertices

add edge `v-w`

(parallel edges allowed)

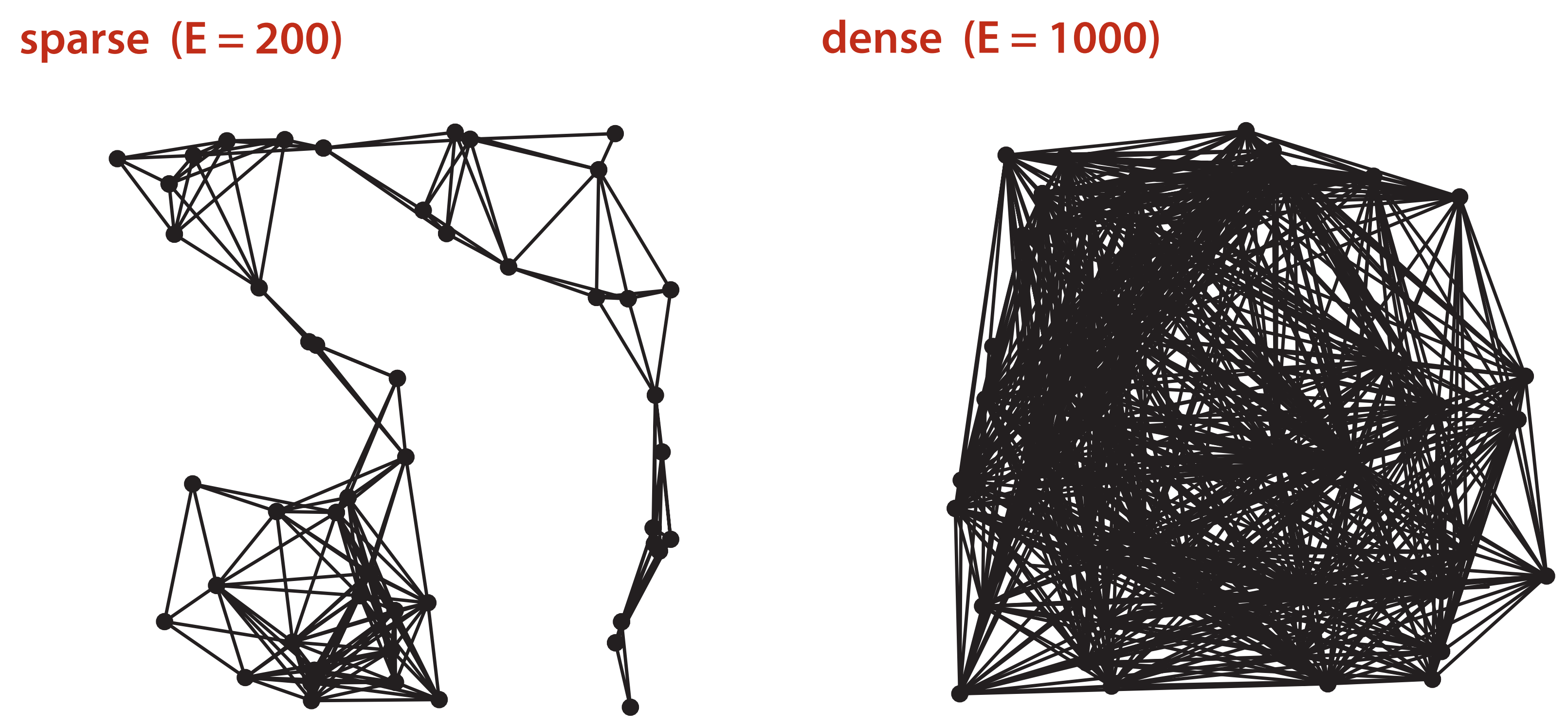
iterator for vertices adjacent to `v`

Graph representations

In practice. Use adjacency-lists representation.

- Algorithms based on iterating over vertices adjacent to *v*.
- Real-world graphs tend to be sparse.

huge number of vertices,
small average vertex degree



Two graphs (V = 50)

Graph representations

In practice. Use adjacency-lists representation.

- Algorithms based on iterating over vertices adjacent to *v*.
- Real-world graphs tend to be sparse.

huge number of vertices,
small average vertex degree

representation	space	add edge	edge between v and w?	iterate over vertices adjacent to v?
list of edges	E	1	E	E
adjacency matrix	V^2	1 *	1	V
adjacency lists	E + V	1	degree(v)	degree(v)

* disallows parallel edges

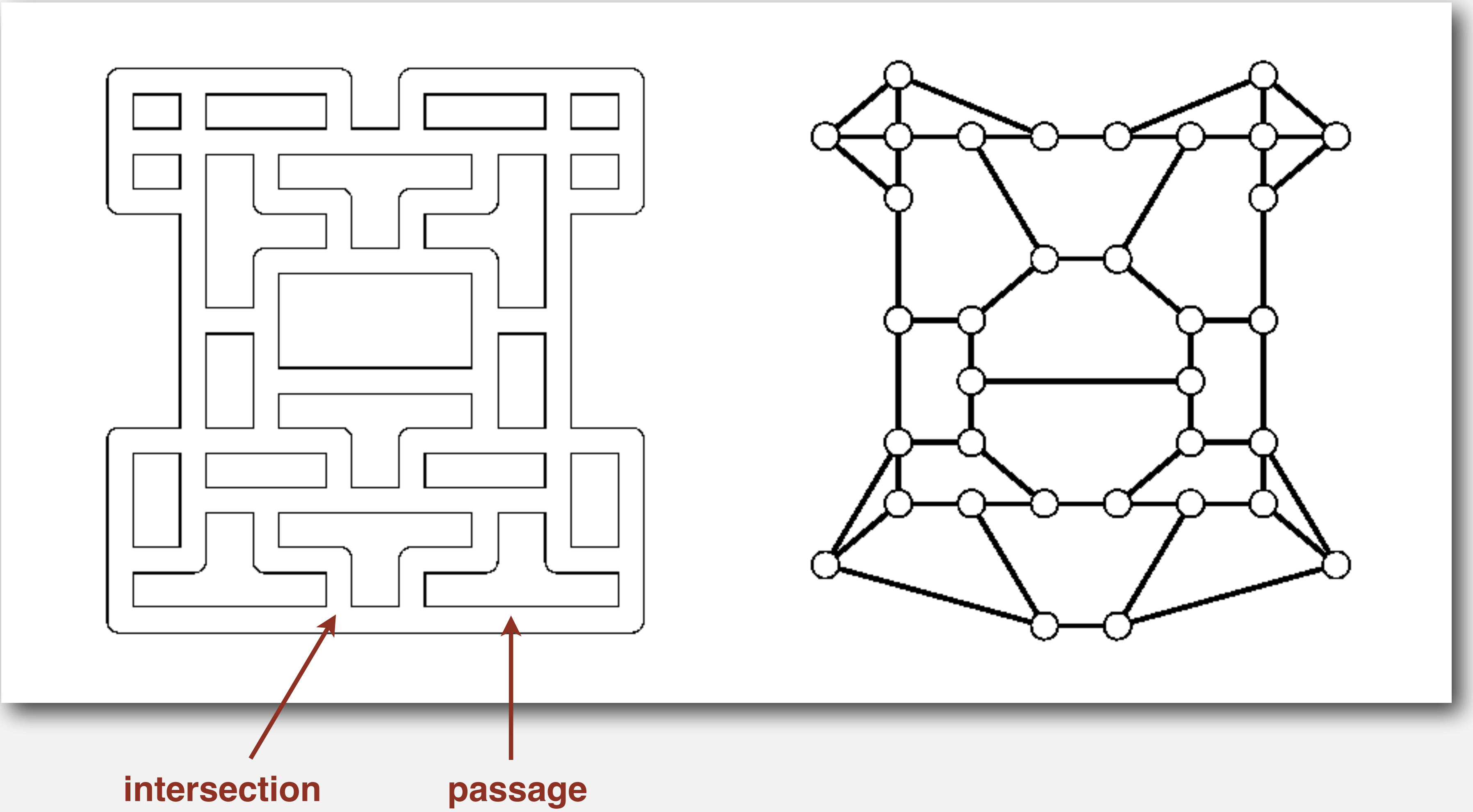
Undirected Graphs

- Graph API
- Depth-first search
- Breadth-first search
- Connected components
- Challenges

Maze exploration

Maze graphs.

- Vertex = intersection.
- Edge = passage.

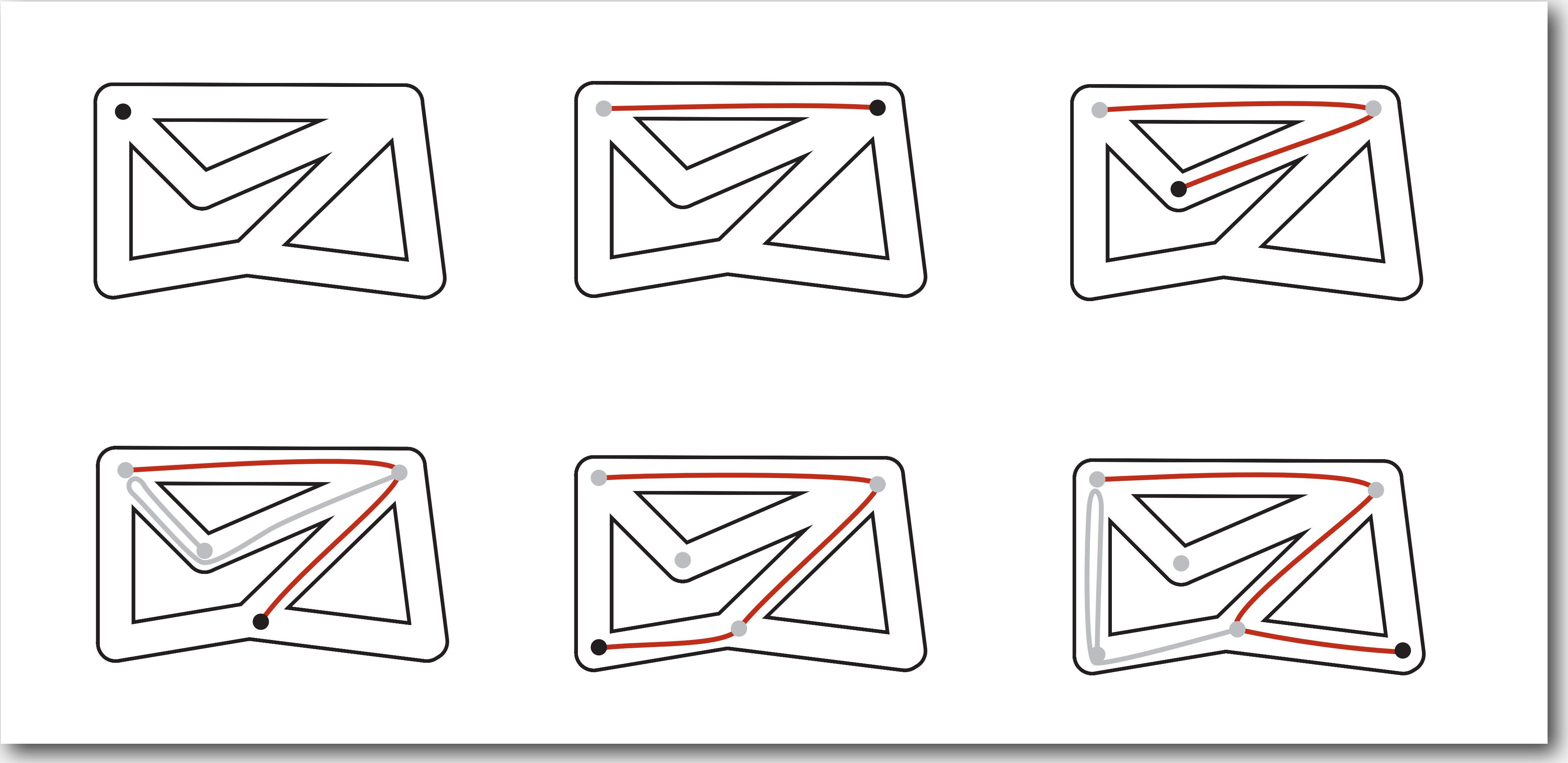


Goal. Explore every intersection in the maze.

Trémaux maze exploration

Algorithm.

- Unroll a ball of string behind you.
- Mark each visited intersection and each visited passage.
- Retrace steps when no unvisited options.



Depth-first search

Goal. Systematically search through a graph.

Idea. Mimic maze exploration.

```
DFS (to visit a vertex v)

Mark v as visited.
Recursively visit all unmarked
        vertices w adjacent to v.
```

Typical applications.

- Find all vertices connected to a given source vertex.
- Find a path between two vertices.

Design challenge. How to implement?

Design pattern for graph processing

Design pattern. Decouple graph data type from graph processing.

- Create a `Graph` object.
- Pass the `Graph` to a graph-processing routine, e.g., `Paths`.
- Query the graph-processing routine for information.

public class Paths		
	`Paths(Graph G, int s)`	*find paths in G from source s*
`boolean`	`hasPathTo(int v)`	*is there a path from s to v?*
`Iterable<Integer>`	`pathTo(int v)`	*path from s to v; null if no such path*

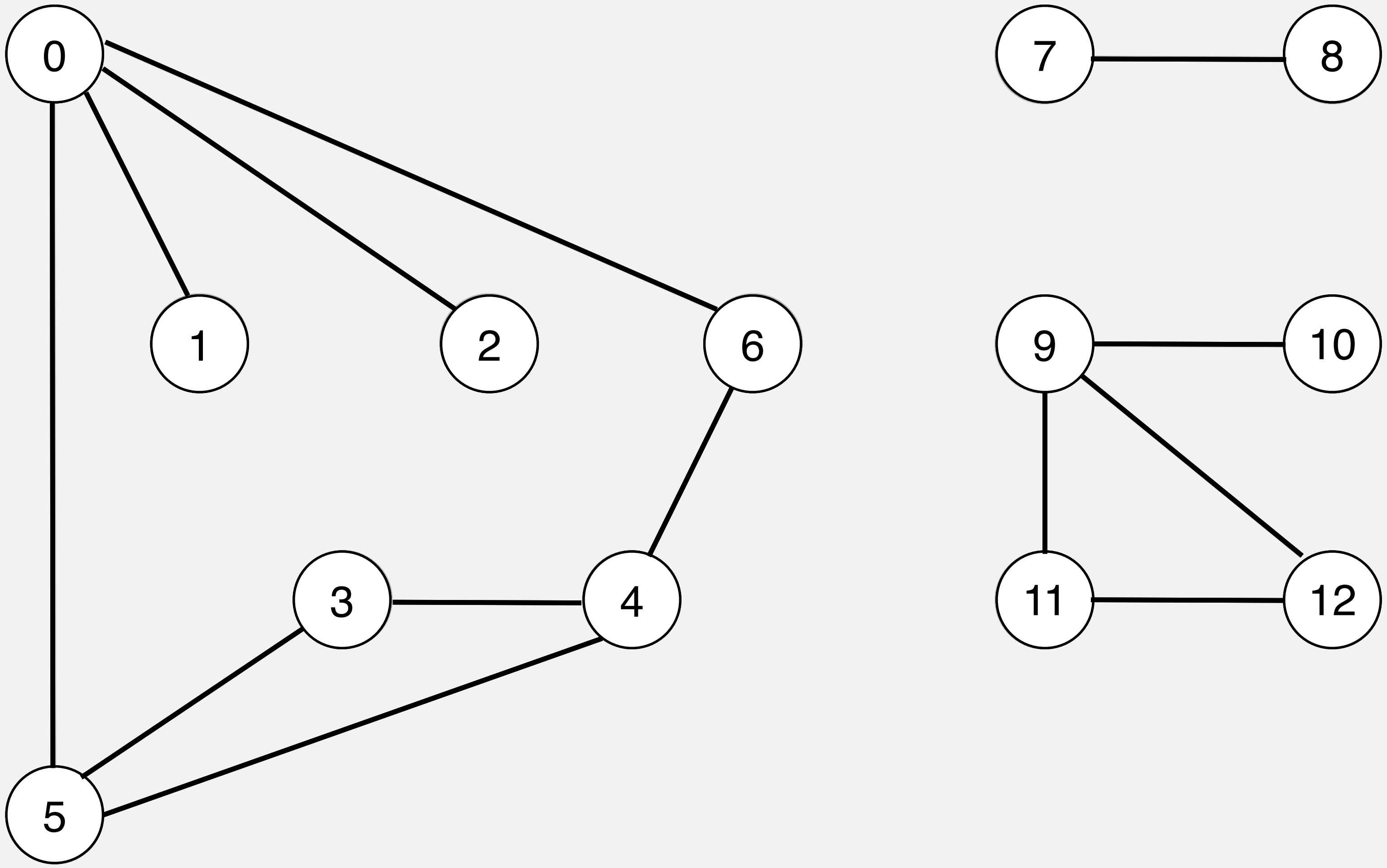
```
Paths paths = new Paths(G, s);
for (int v = 0; v < G.V(); v++)
   if (paths.hasPathTo(v))
      StdOut.println(v);
```

print all vertices connected to s

Depth-first search

To visit a vertex *v* :

- Mark vertex *v* as visited.
- Recursively visit all unmarked vertices adjacent to *v*.



graph G

tinyG.txt

V → 13

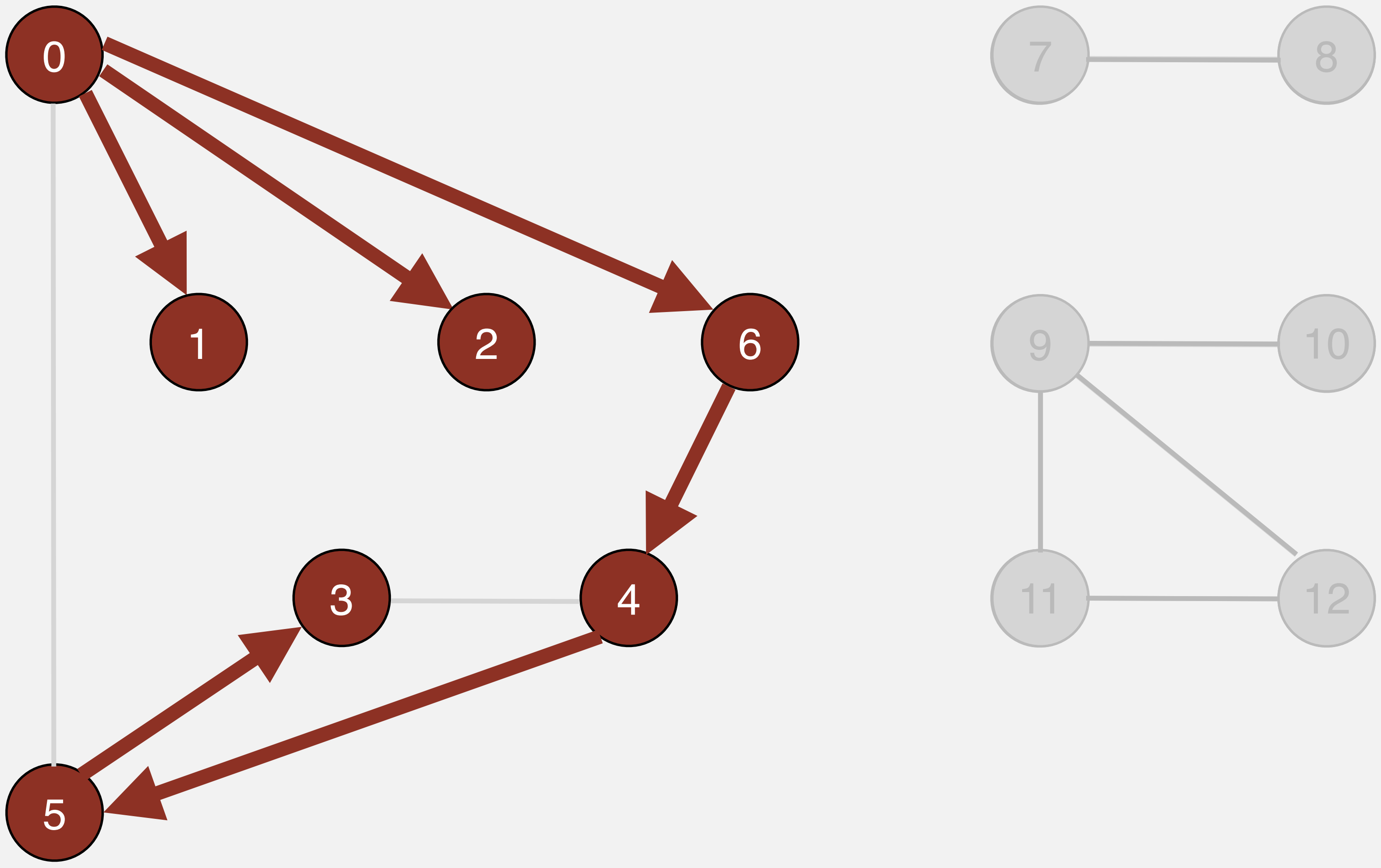
13 ← *E*

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

Depth-first search

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



vertices reachable from 0

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

Depth-first search

Goal. Find all vertices connected to s (and a path).

Idea. Mimic maze exploration.

Algorithm.

- Use recursion (ball of string).
- Mark each visited vertex (and keep track of edge taken to visit it).
- Return (retrace steps) when no unvisited options.

Data structures.

- `boolean[] marked` to mark visited vertices.
- `int[] edgeTo` to keep tree of paths.

  `(edgeTo[w] == v)` means that edge `v-w` taken to visit `w` for first time

Depth-first search

```
public class DepthFirstPaths
{
   private boolean[] marked;
   private int[] edgeTo;
   private int s;

   public DepthFirstSearch(Graph G, int s)
   {
      ...
      dfs(G, s);
   }

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

marked[v] = true
if v connected to s

edgeTo[v] = previous vertex
on path from s to v

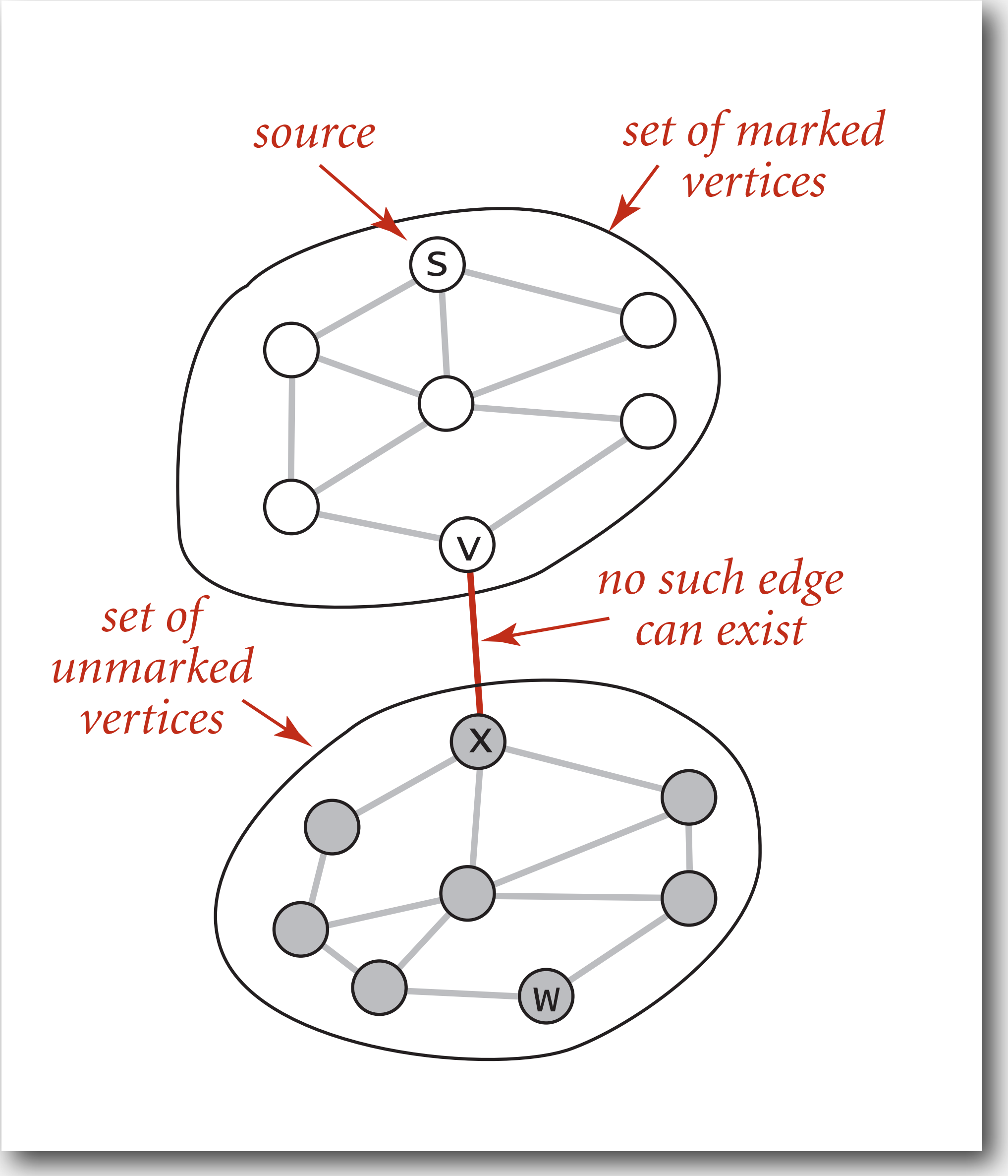
initialize data structures

find vertices connected to s

recursive DFS does the work

Depth-first search properties

Proposition. DFS marks all vertices connected to s in time proportional to the sum of their degrees.



Pf.

- Correctness:
  - if w marked, then w connected to s (why?)
  - if w connected to s, then w marked (if w unmarked, then consider last edge on a path from s to w that goes from a marked vertex to an unmarked one)

- Running time: Each vertex connected to s is visited once.

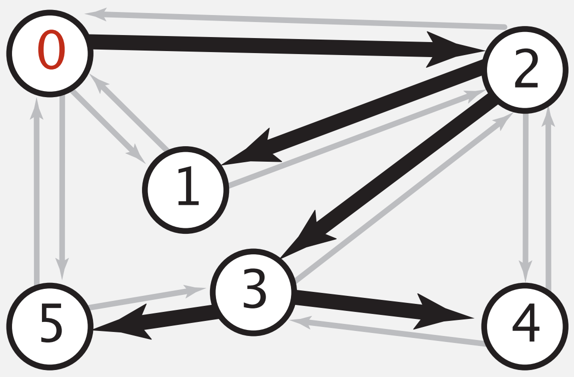
Depth-first search properties

Proposition. After DFS, can find vertices connected to s in constant time and can find a path to s (if one exists) in time proportional to its length.

Pf. `edgeTo[]` is a parent-link representation of a tree rooted at `s`.

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

public Iterable<Integer> pathTo(int v)
{
   if (!hasPathTo(v)) return null;
   Stack<Integer> path = new Stack<Integer>();
   for (int x = v; x != s; x = edgeTo[x])
      path.push(x);
   path.push(s);
   return path;
}
```



edgeTo[]	
0	
1	2
2	0
3	2
4	3
5	3

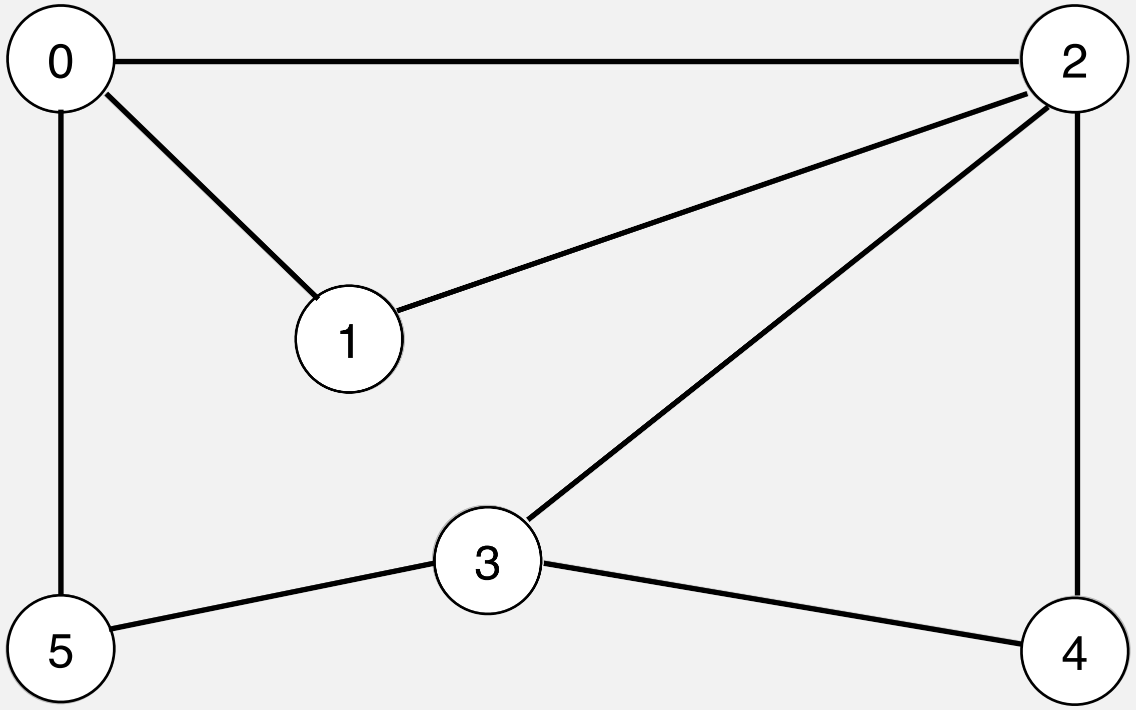
Undirected Graphs

- Graph API
- Depth-first search
- Breadth-first search
- Connected components
- Challenges

Breadth-first search

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

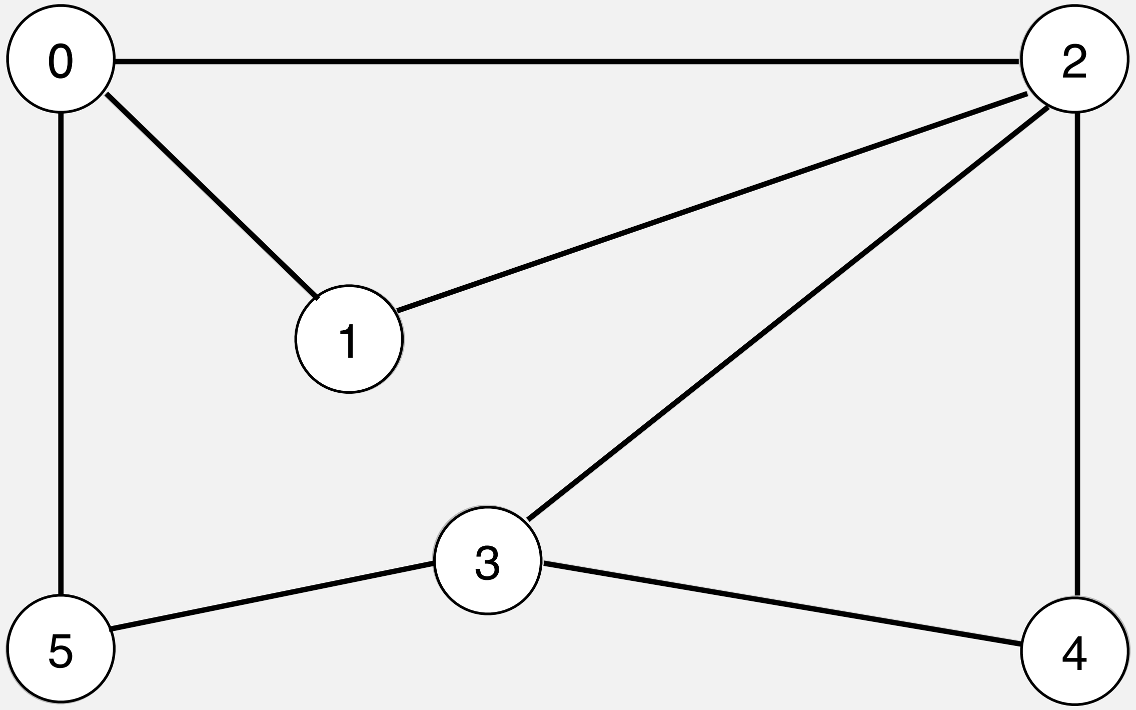
E

```
6
8
0 5
2 4
2 3
1 2
0 1
3 4
3 5
0 2
```

Breadth-first search

Repeat until queue is empty:

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



queue

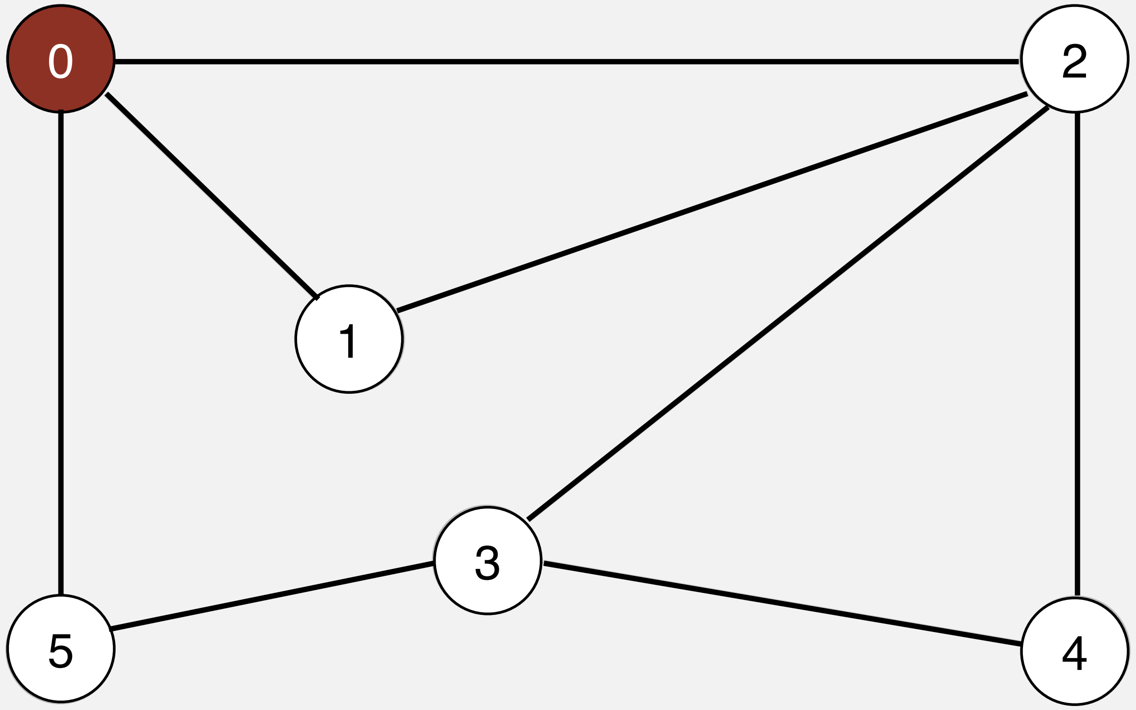
v	edgeTo[v]
0	–
1	–
2	–
3	–
4	–
5	–

add 0 to queue

Breadth-first search

Repeat until queue is empty:

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



queue

0

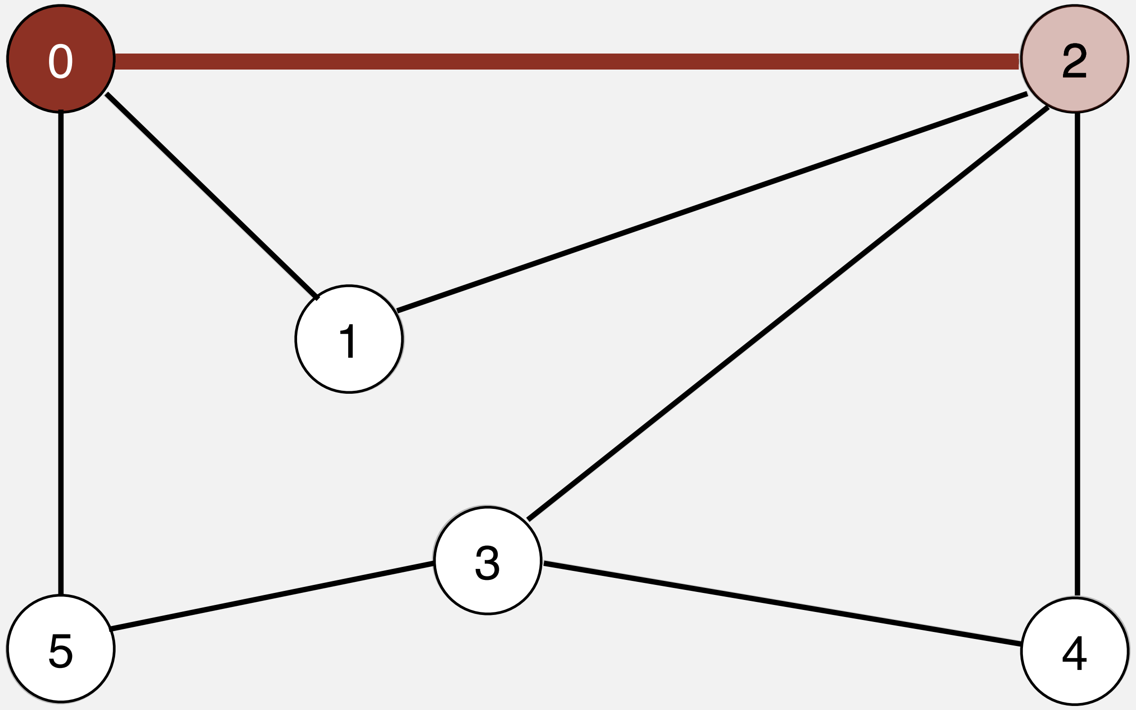
v	edgeTo[v]
0	–
1	–
2	–
3	–
4	–
5	–

dequeue 0

Breadth-first search

Repeat until queue is empty:

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



queue

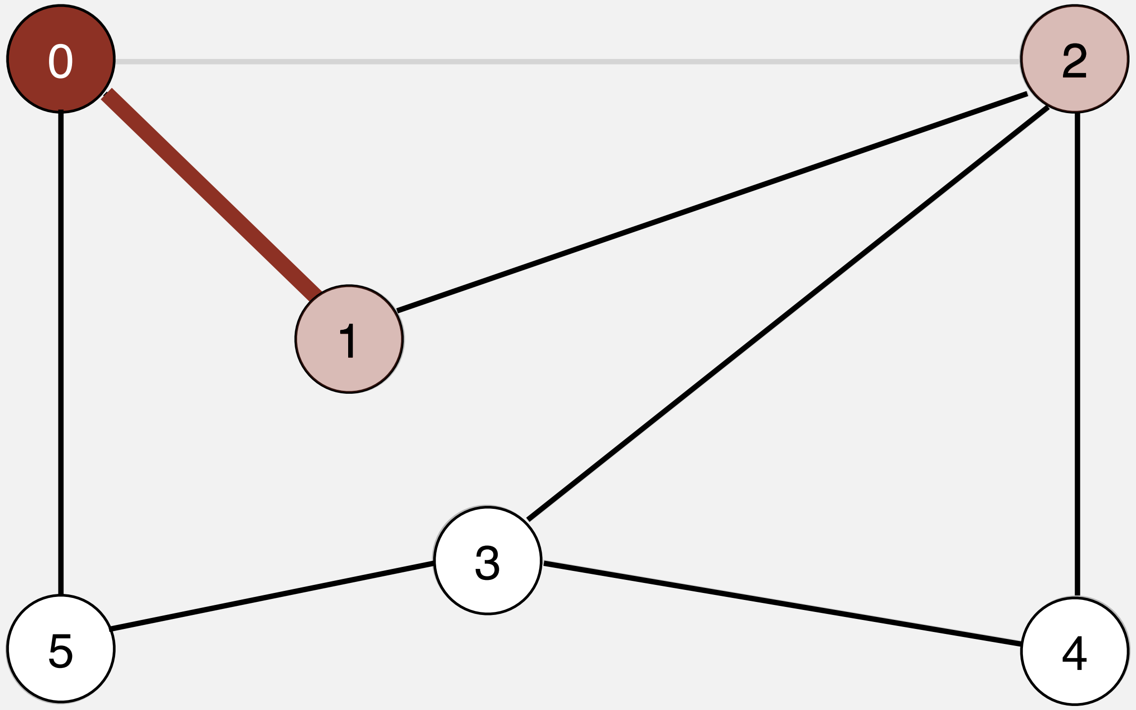
v	edgeTo[v]
0	–
1	–
2	0
3	–
4	–
5	–

dequeue 0

Breadth-first search

Repeat until queue is empty:

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



queue

2

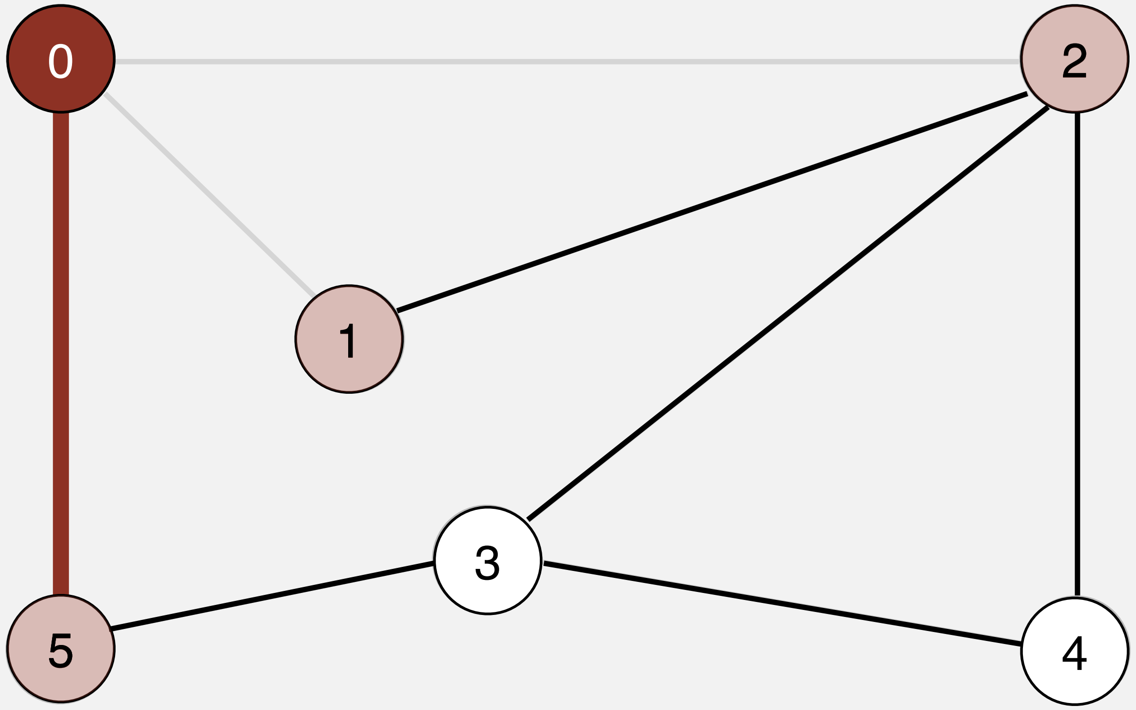
v	edgeTo[v]
0	–
1	0
2	0
3	–
4	–
5	–

dequeue 0

Breadth-first search

Repeat until queue is empty:

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



queue

1

2

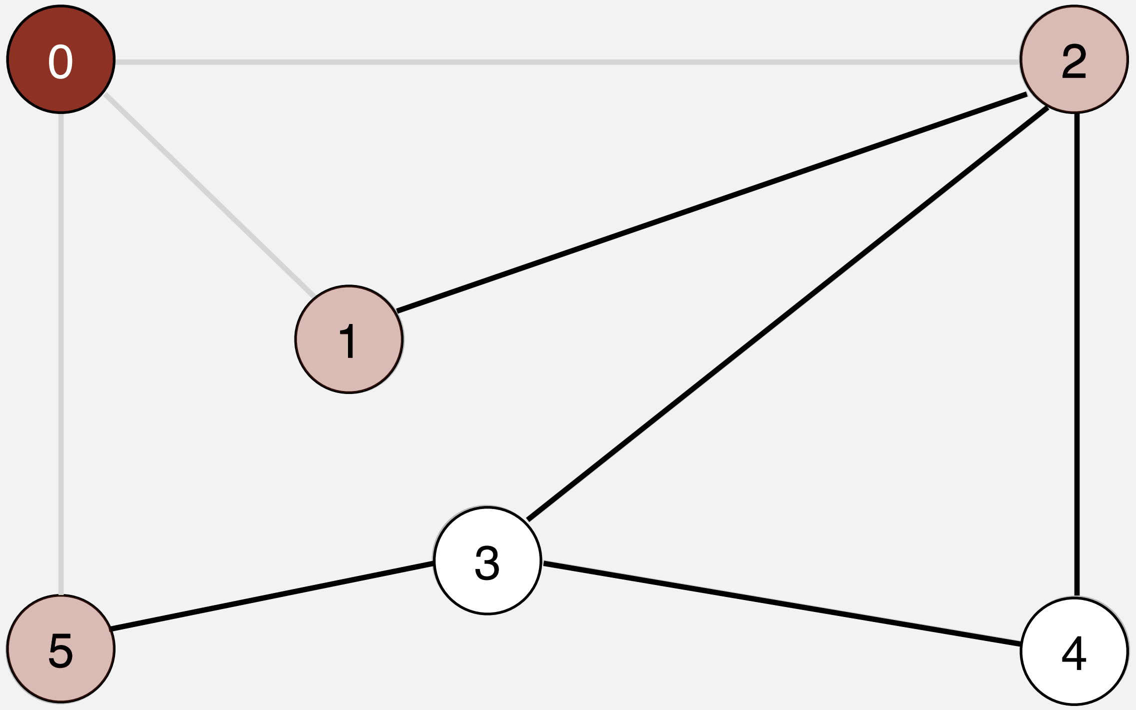
v	edgeTo[v]
0	–
1	0
2	0
3	–
4	–
5	0

dequeue 0

Breadth-first search

Repeat until queue is empty:

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



queue

5

1

2

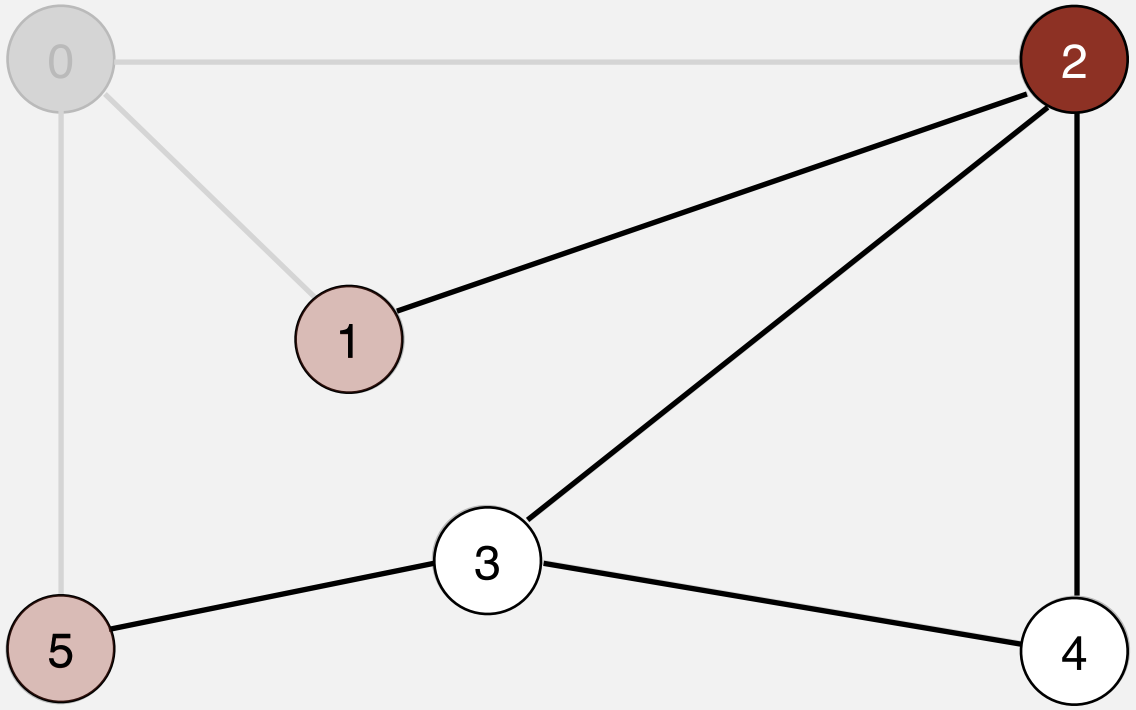
v	edgeTo[v]
0	–
1	0
2	0
3	–
4	–
5	0

0 done

Breadth-first search

Repeat until queue is empty:

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



queue

5

1

2

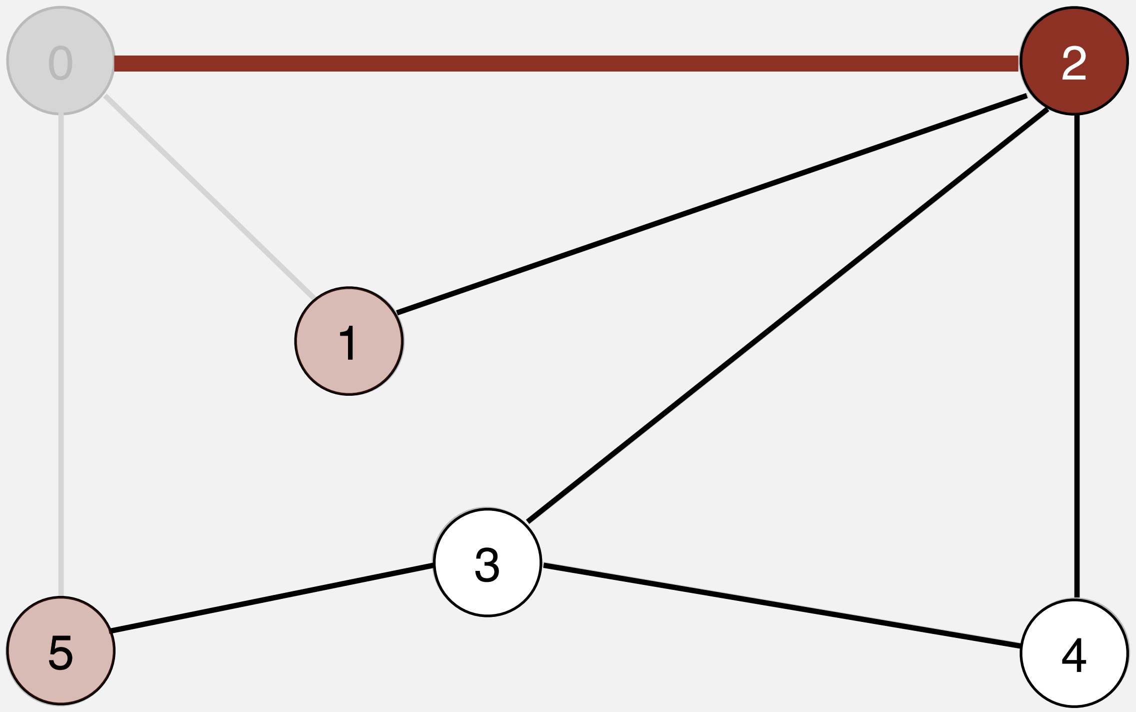
v	edgeTo[v]
0	–
1	0
2	0
3	–
4	–
5	0

dequeue 2

Breadth-first search

Repeat until queue is empty:

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



queue

5

1

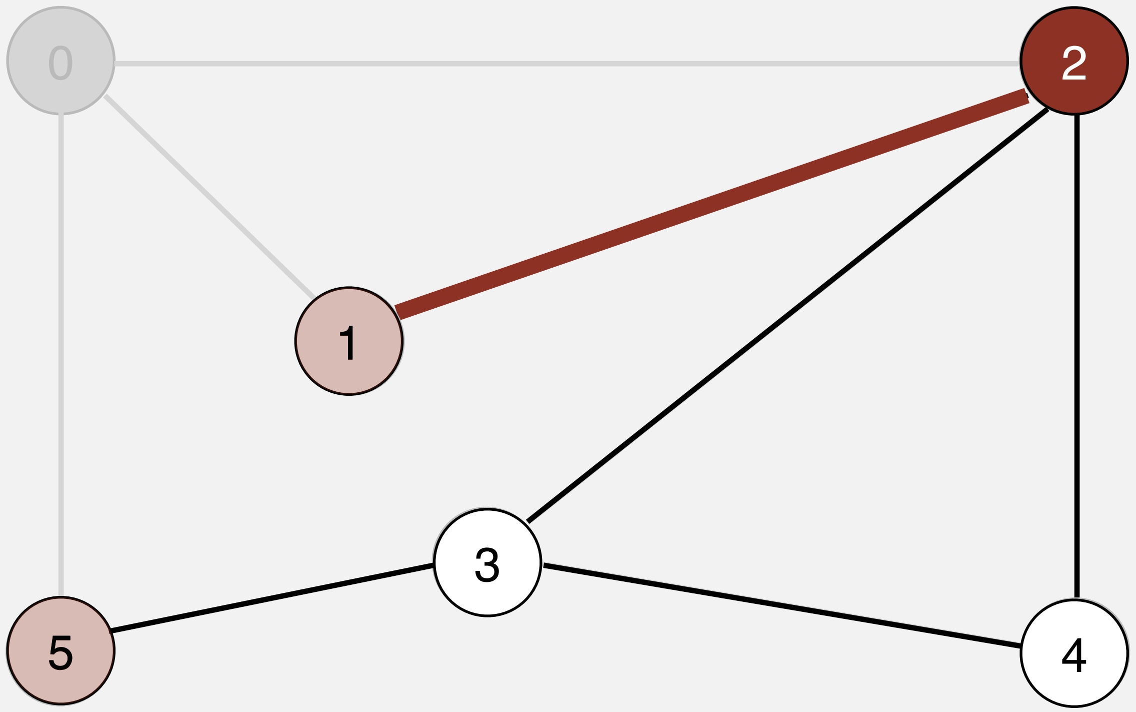
v	edgeTo[v]
0	–
1	0
2	0
3	–
4	–
5	0

dequeue 2

Breadth-first search

Repeat until queue is empty:

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



queue

5

1

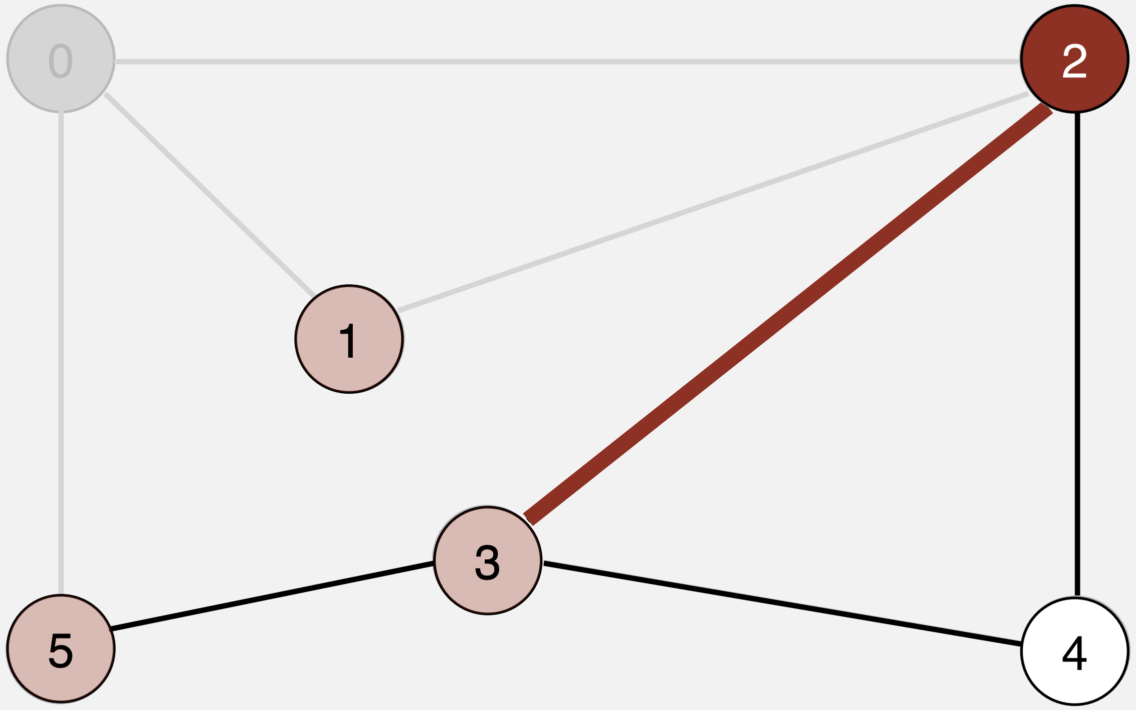
v	edgeTo[v]
0	–
1	0
2	0
3	–
4	–
5	0

dequeue 2

Breadth-first search

Repeat until queue is empty:

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



queue

5

1

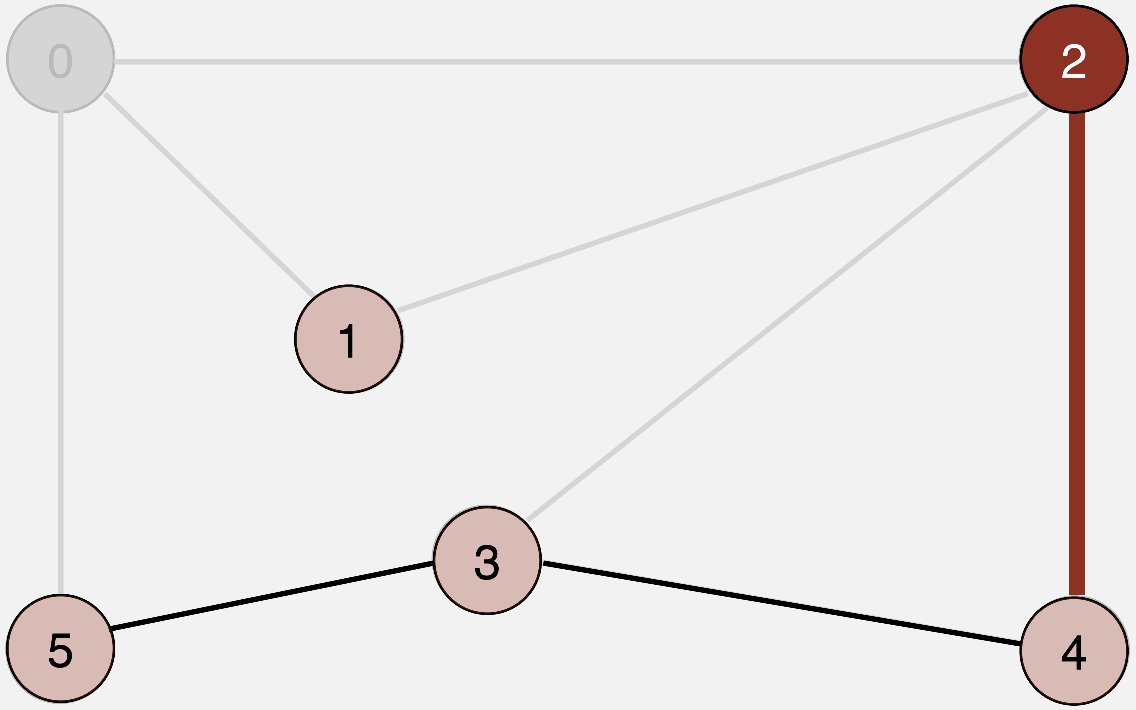
v	edgeTo[v]
0	–
1	0
2	0
3	2
4	–
5	0

dequeue 2

Breadth-first search

Repeat until queue is empty:

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



queue

3

5

1

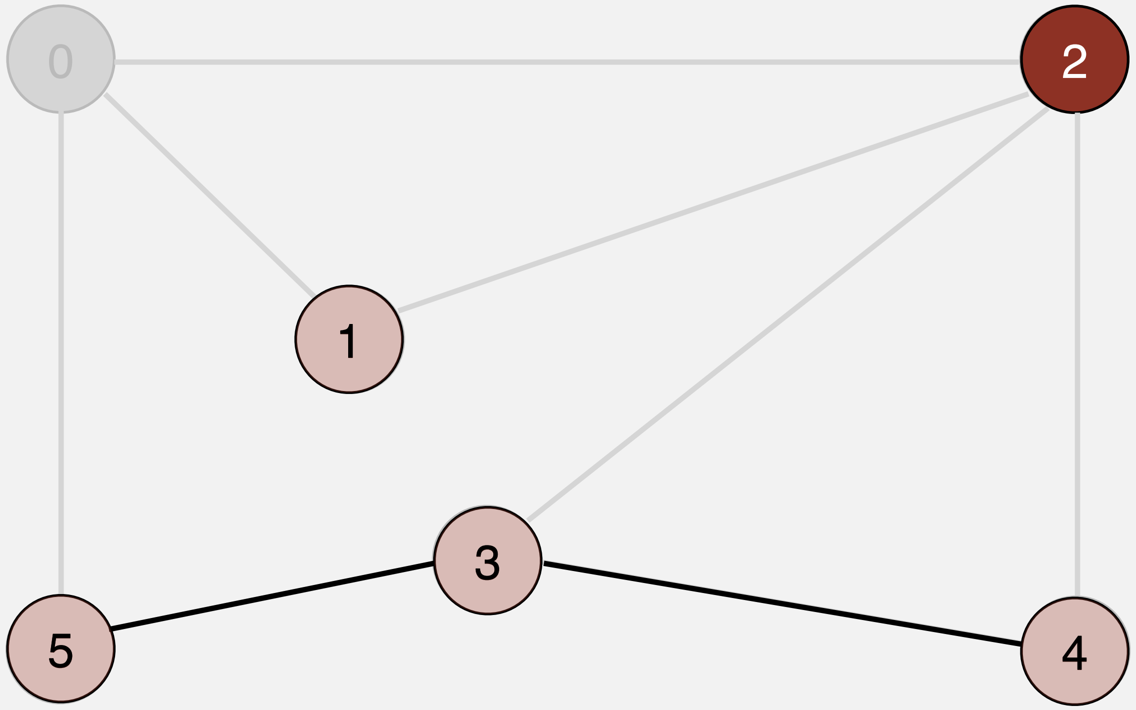
v	edgeTo[v]
0	–
1	0
2	0
3	2
4	2
5	0

dequeue 2

Breadth-first search

Repeat until queue is empty:

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



queue
4
3
5
1

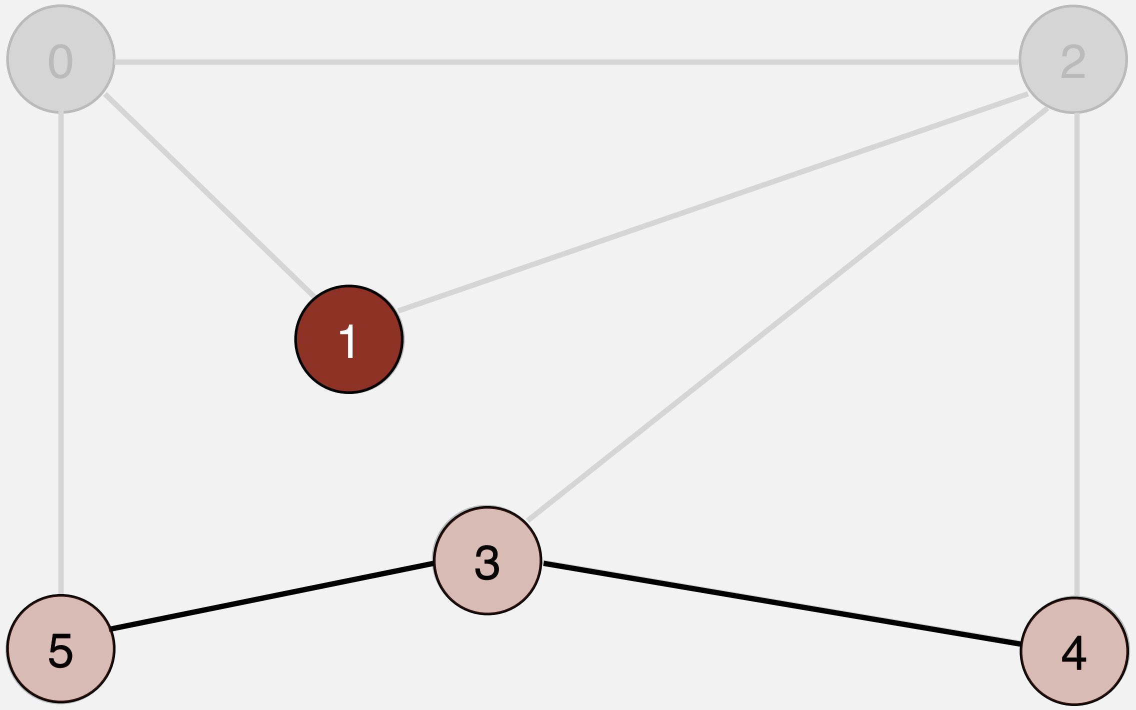
v	edgeTo[v]
0	–
1	0
2	0
3	2
4	2
5	0

2 done

Breadth-first search

Repeat until queue is empty:

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



queue

4
3
5
1

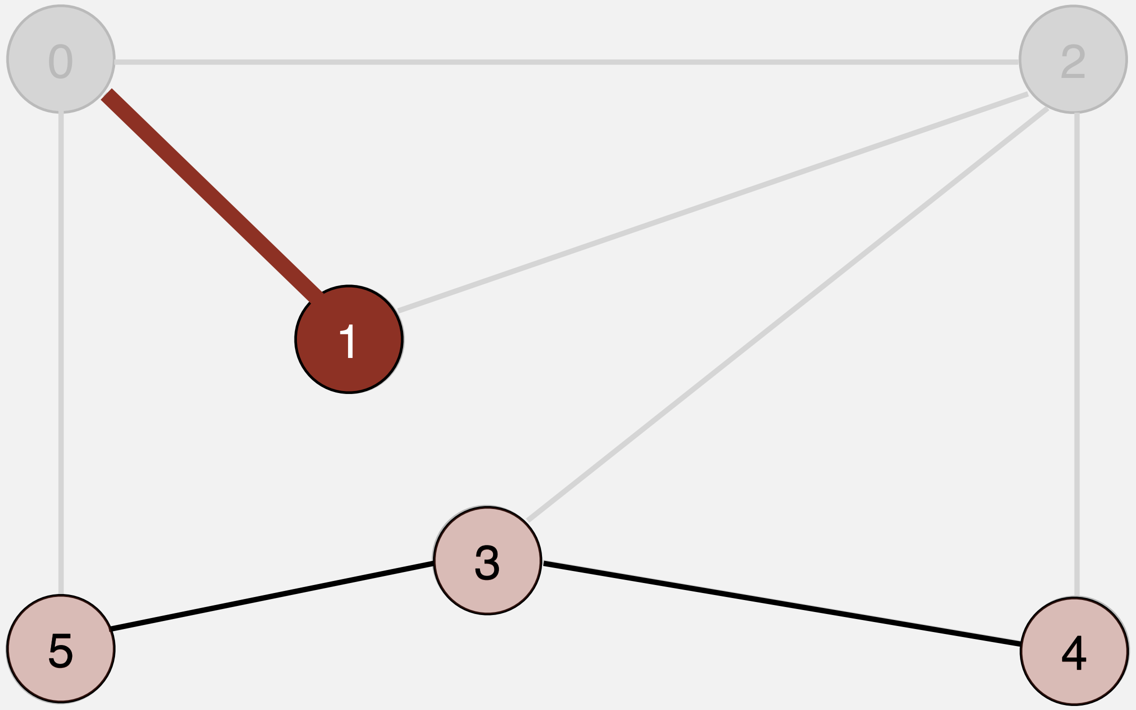
v	edgeTo[v]
0	–
1	0
2	0
3	2
4	2
5	0

dequeue 1

Breadth-first search

Repeat until queue is empty:

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



queue

4

3

5

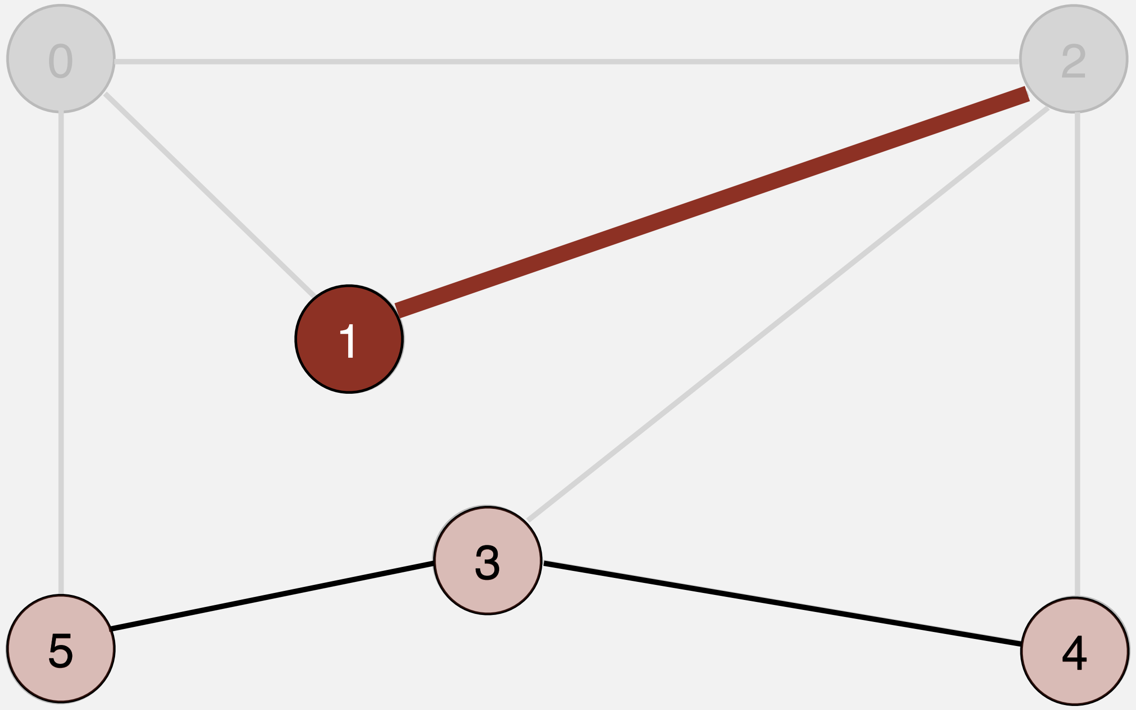
v	edgeTo[v]
0	–
1	0
2	0
3	2
4	2
5	0

dequeue 1

Breadth-first search

Repeat until queue is empty:

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



queue

4

3

5

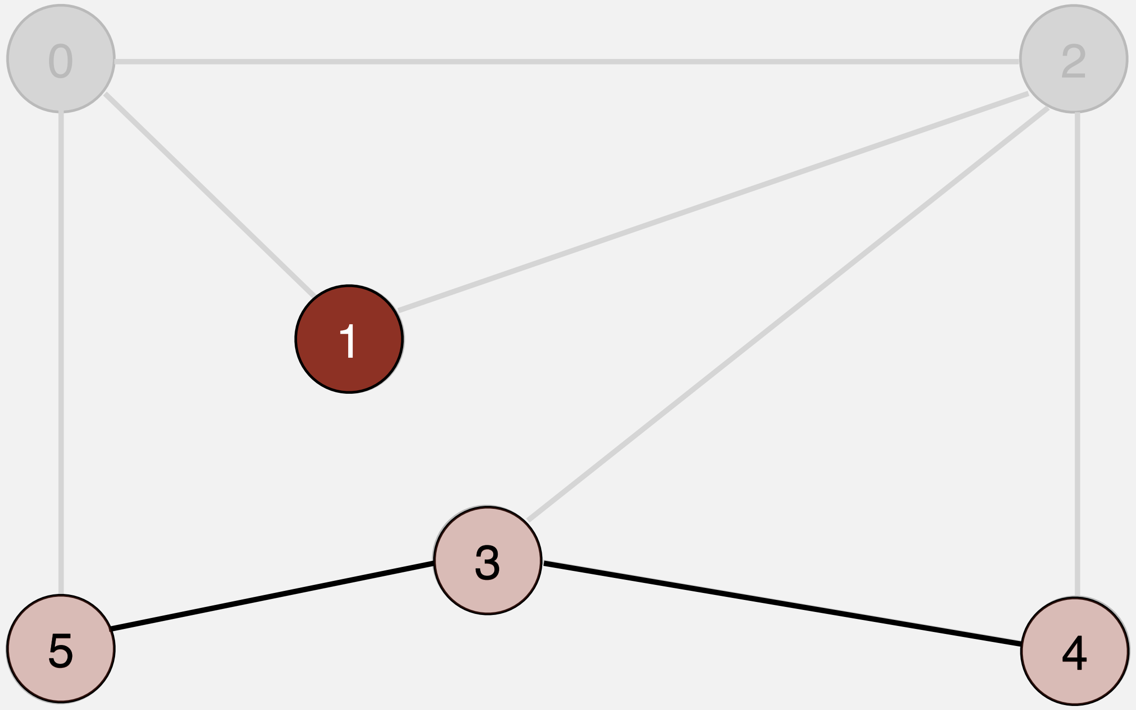
v	edgeTo[v]
0	–
1	0
2	0
3	2
4	2
5	0

dequeue 1

Breadth-first search

Repeat until queue is empty:

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



queue

4

3

5

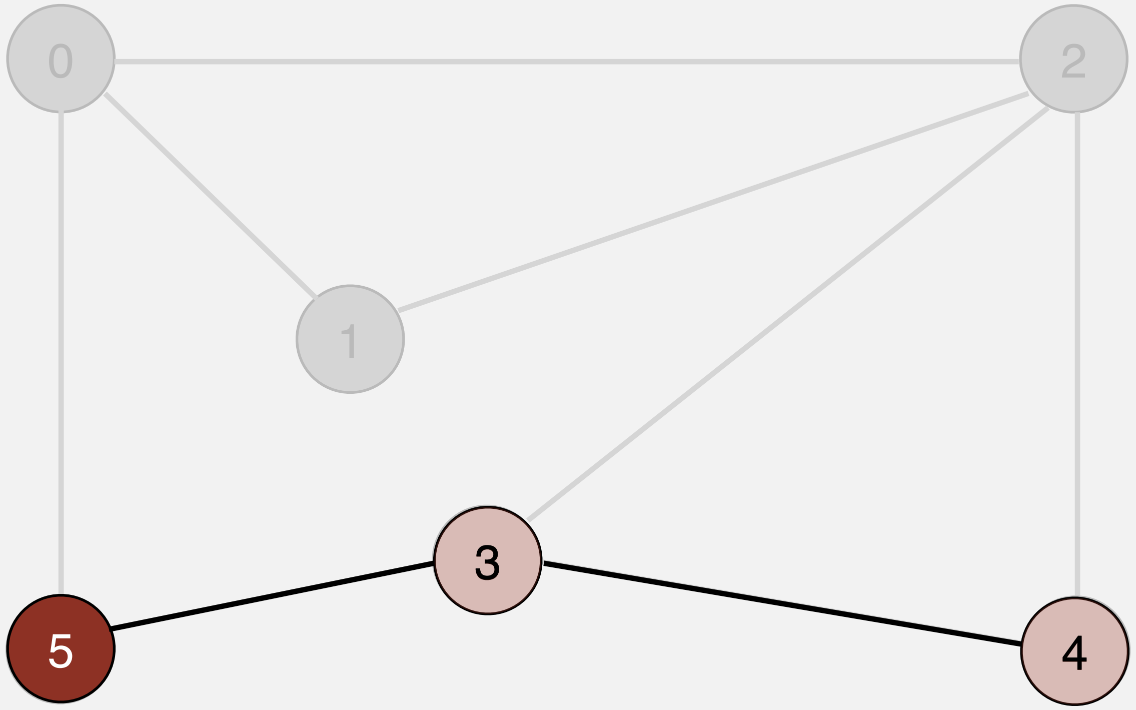
v	edgeTo[v]
0	–
1	0
2	0
3	2
4	2
5	0

1 done

Breadth-first search

Repeat until queue is empty:

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



queue

4

3

5

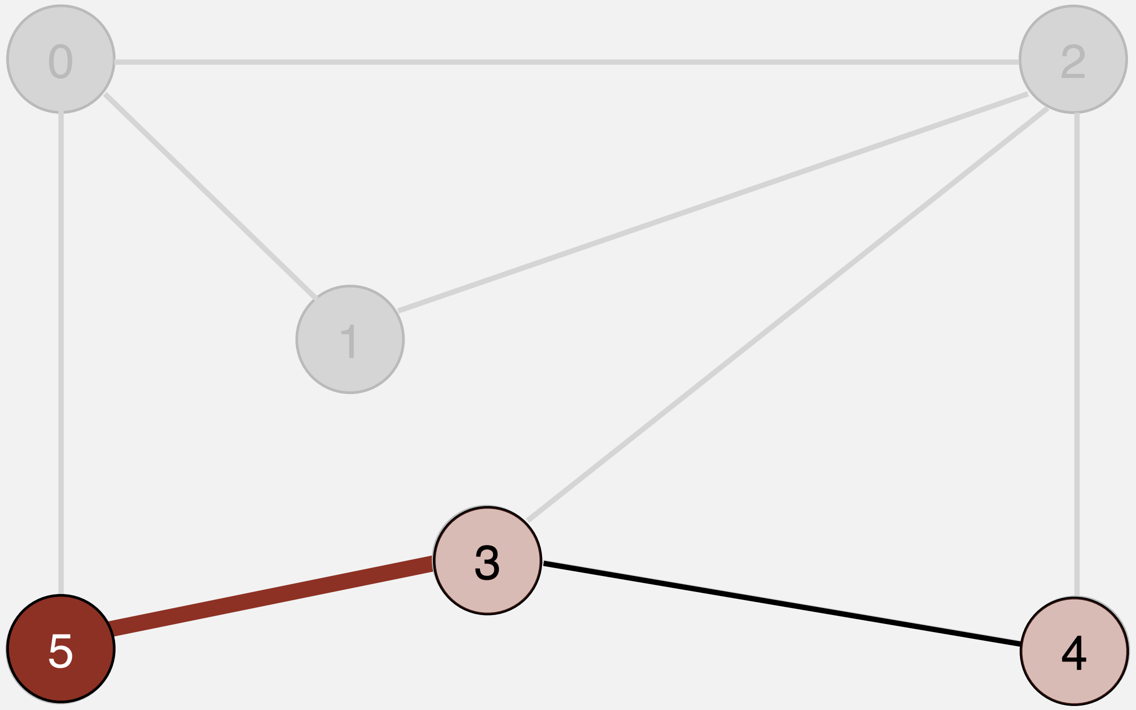
v	edgeTo[v]
0	–
1	0
2	0
3	2
4	2
5	0

dequeue 5

Breadth-first search

Repeat until queue is empty:

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



queue

4

3

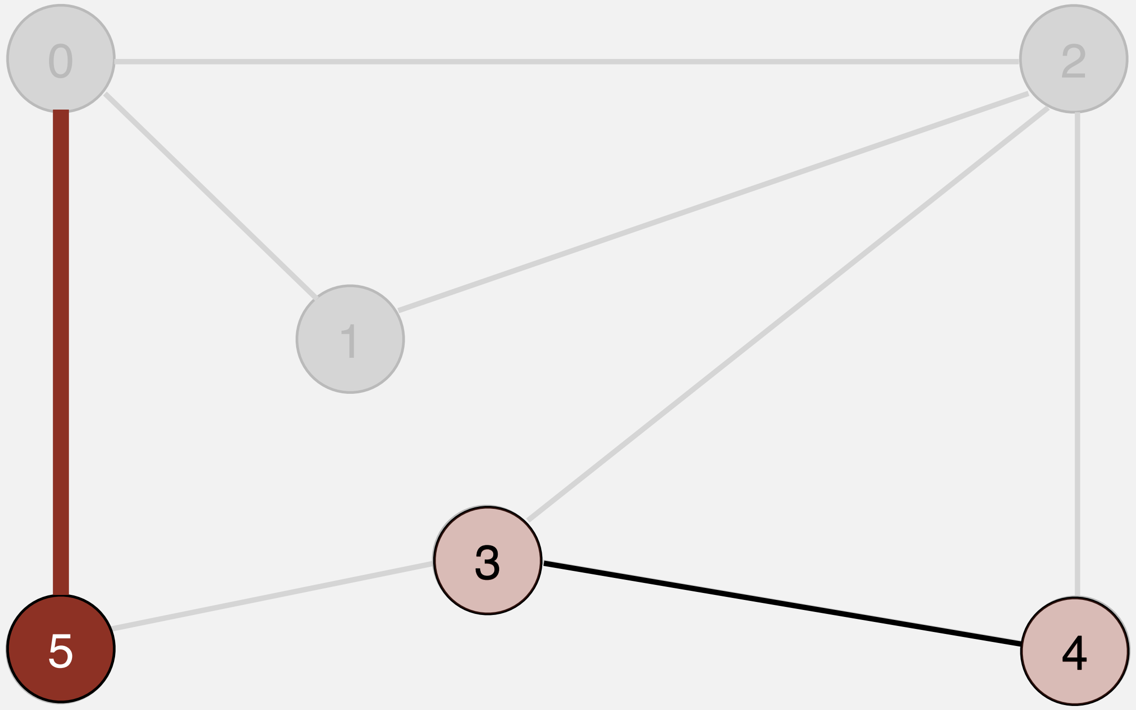
v	edgeTo[v]
0	–
1	0
2	0
3	2
4	2
5	0

dequeue 5

Breadth-first search

Repeat until queue is empty:

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



queue

4

3

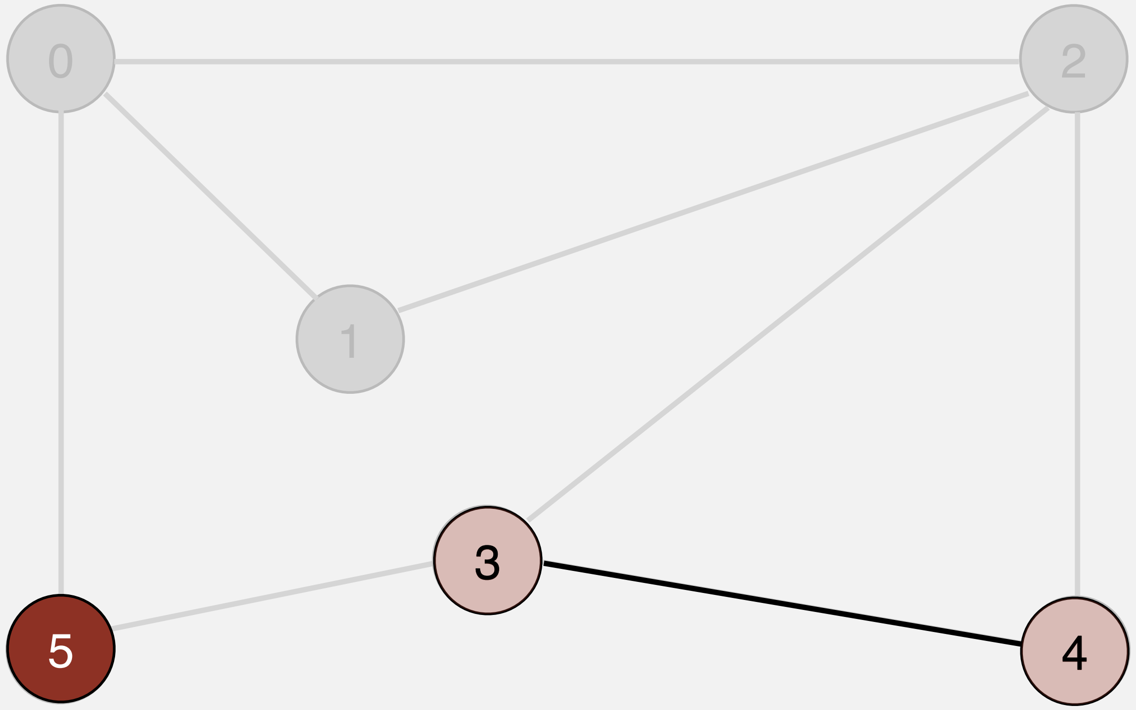
v	edgeTo[v]
0	–
1	0
2	0
3	2
4	2
5	0

dequeue 5

Breadth-first search

Repeat until queue is empty:

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



queue

4

3

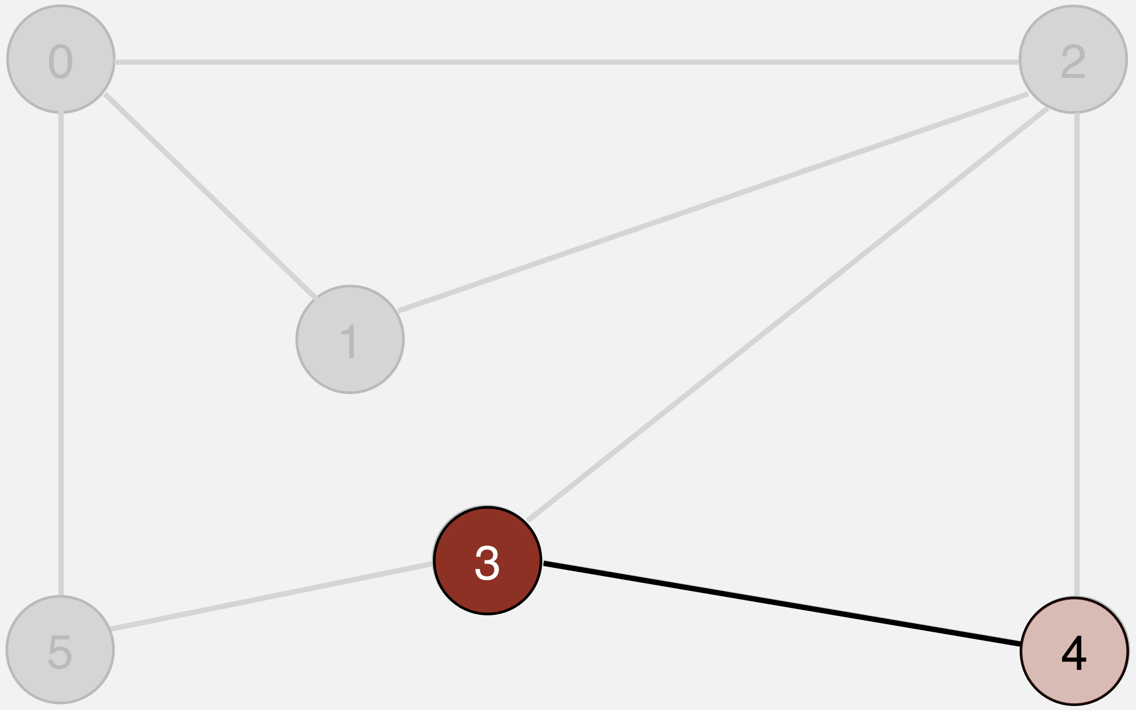
v	edgeTo[v]
0	–
1	0
2	0
3	2
4	2
5	0

5 done

Breadth-first search

Repeat until queue is empty:

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



queue

4

3

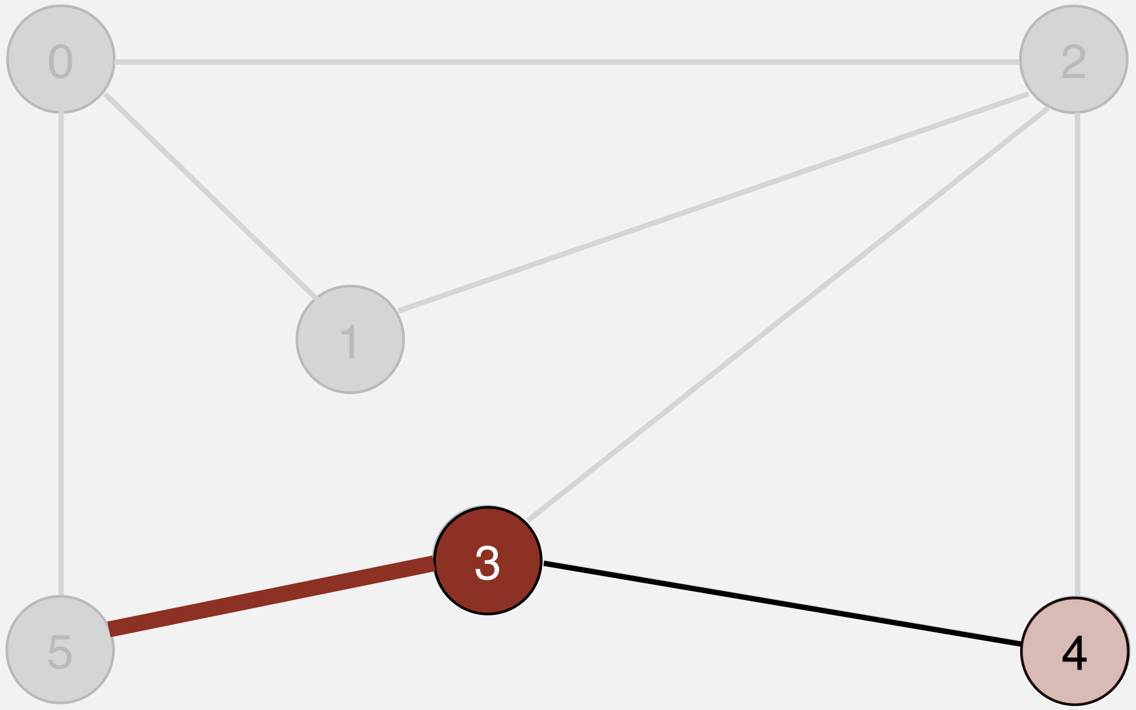
v	edgeTo[v]
0	–
1	0
2	0
3	2
4	2
5	0

dequeue 3

Breadth-first search

Repeat until queue is empty:

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



queue

4

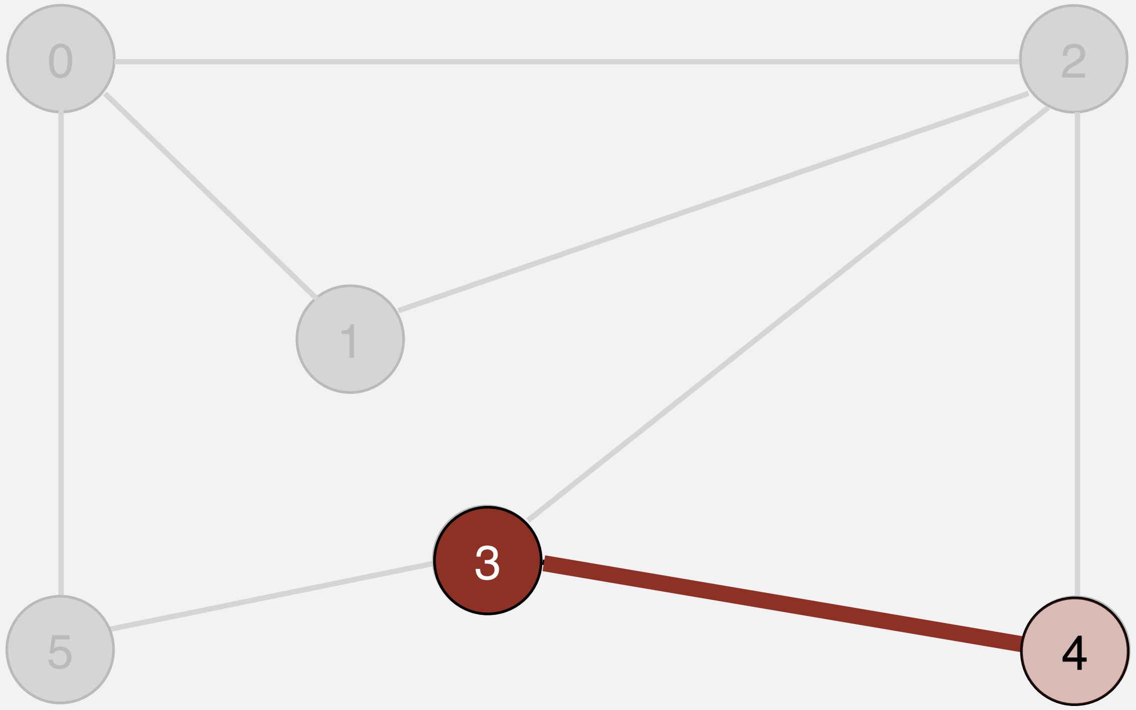
v	edgeTo[v]
0	–
1	0
2	0
3	2
4	2
5	0

dequeue 3

Breadth-first search

Repeat until queue is empty:

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



queue

4

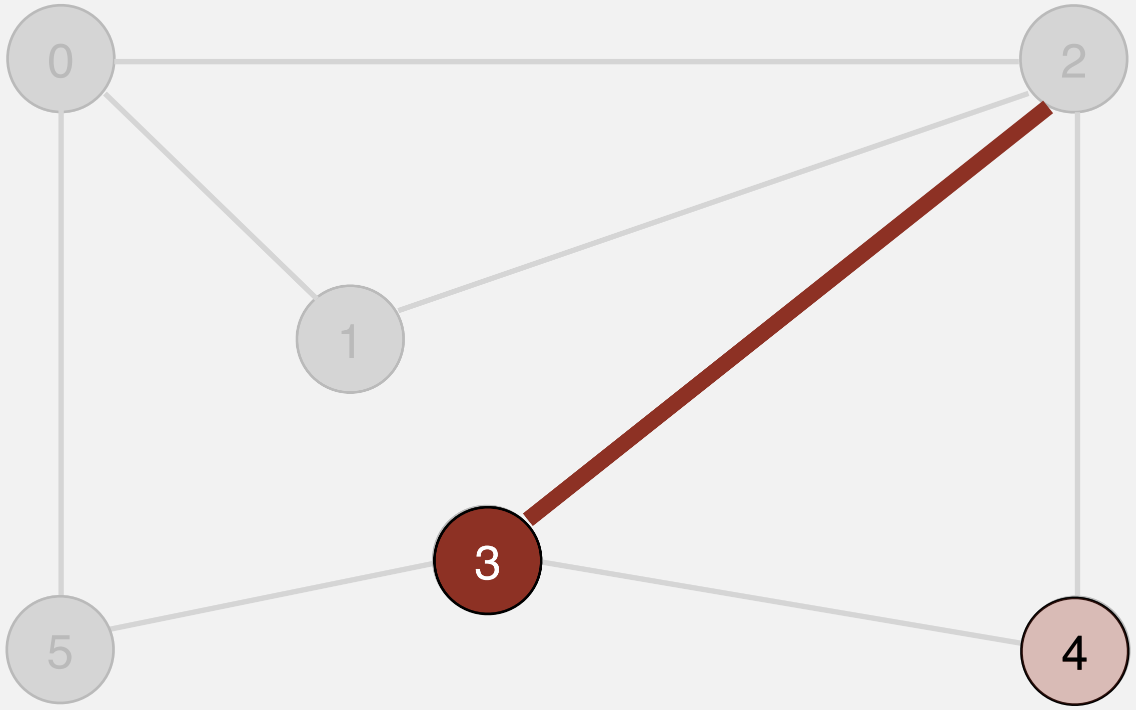
v	edgeTo[v]
0	–
1	0
2	0
3	2
4	2
5	0

dequeue 3

Breadth-first search

Repeat until queue is empty:

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



queue

4

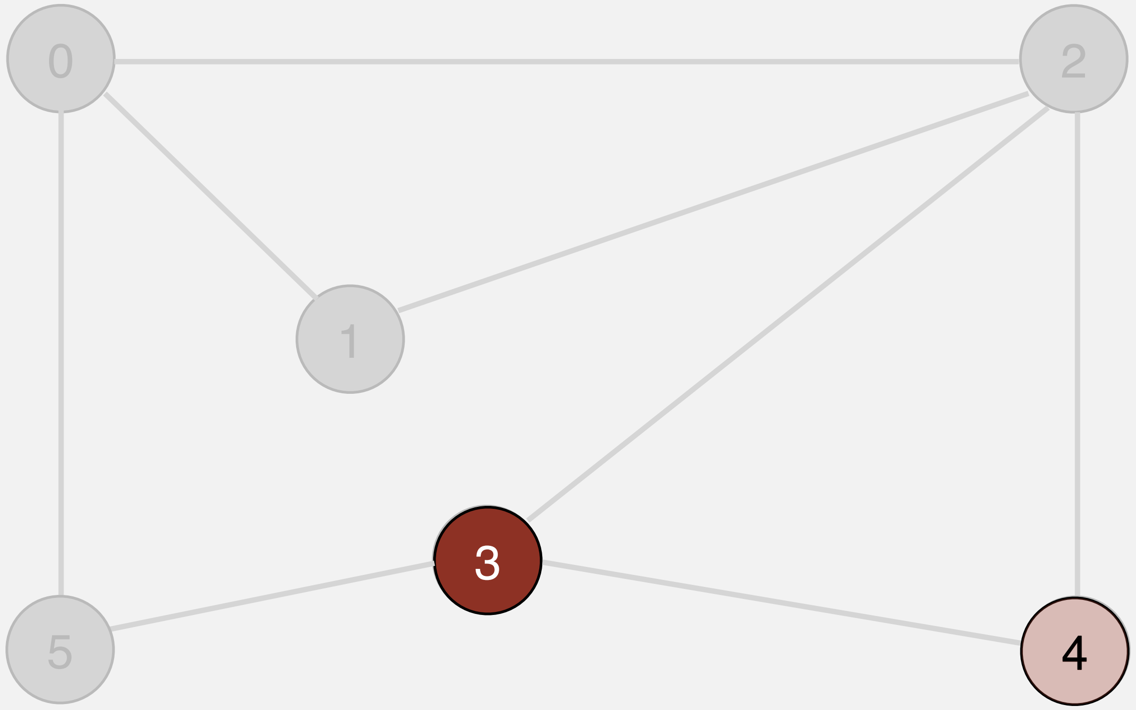
v	edgeTo[v]
0	–
1	0
2	0
3	2
4	2
5	0

dequeue 3

Breadth-first search

Repeat until queue is empty:

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



queue

4

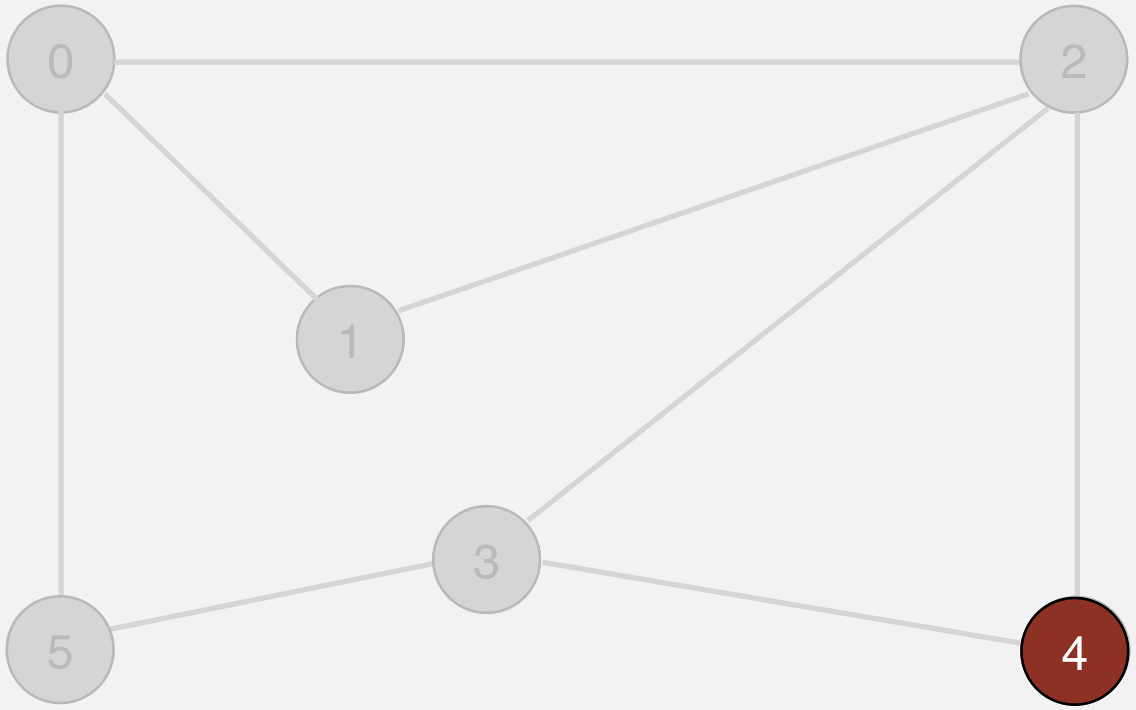
v	edgeTo[v]
0	–
1	0
2	0
3	2
4	2
5	0

3 done

Breadth-first search

Repeat until queue is empty:

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



queue

4

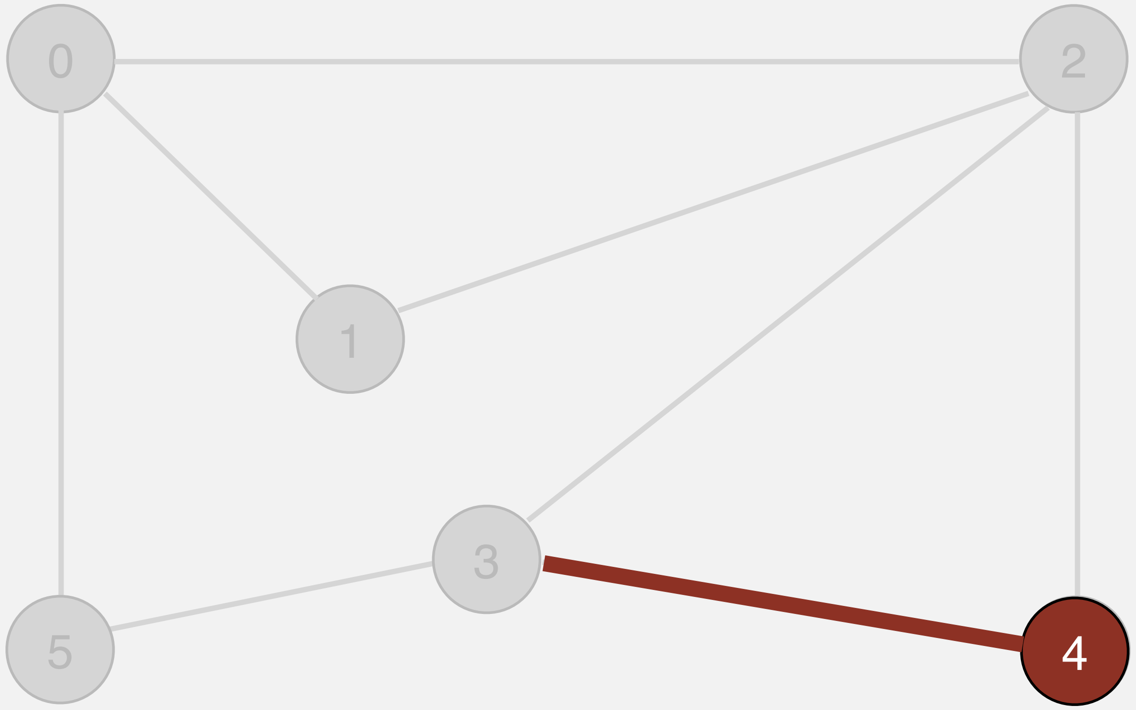
v	edgeTo[v]
0	–
1	0
2	0
3	2
4	2
5	0

dequeue 4

Breadth-first search

Repeat until queue is empty:

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



queue

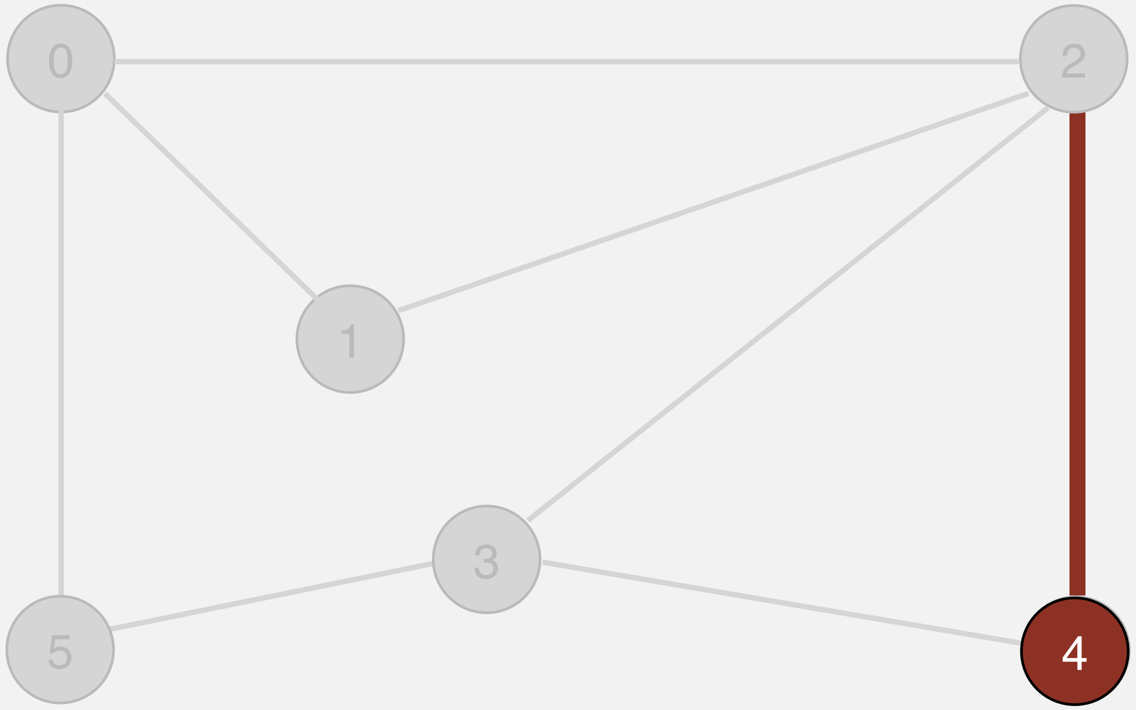
v	edgeTo[v]
0	–
1	0
2	0
3	2
4	2
5	0

dequeue 4

Breadth-first search

Repeat until queue is empty:

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



queue

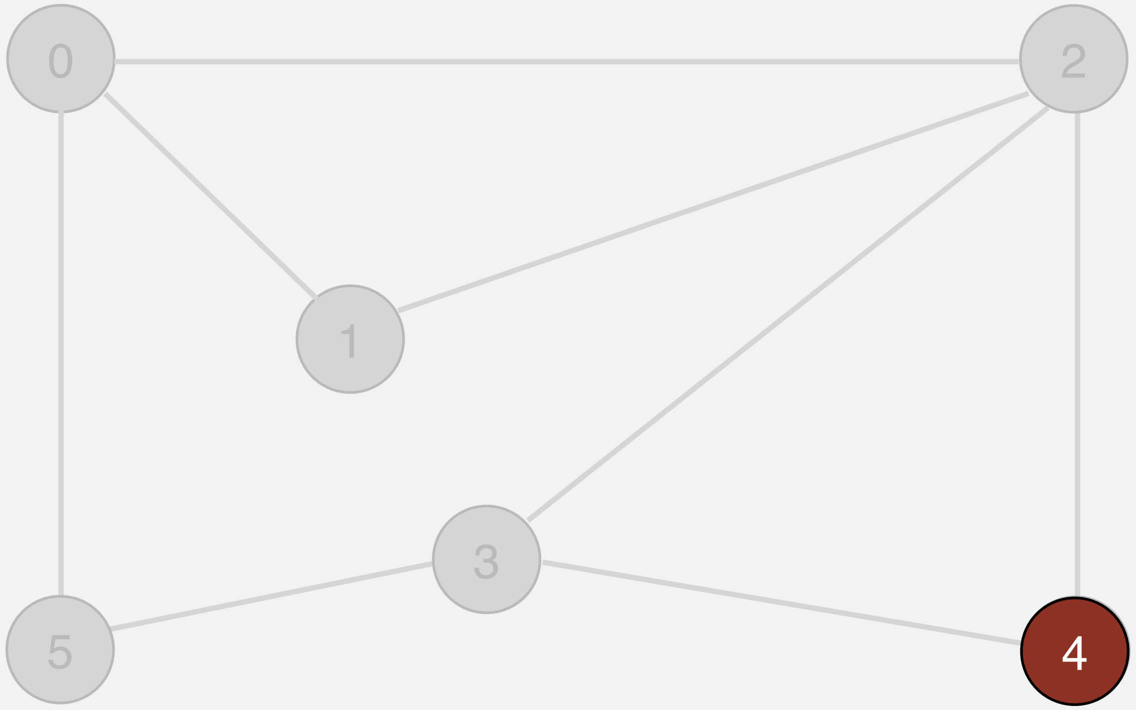
v	edgeTo[v]
0	–
1	0
2	0
3	2
4	2
5	0

dequeue 4

Breadth-first search

Repeat until queue is empty:

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



queue

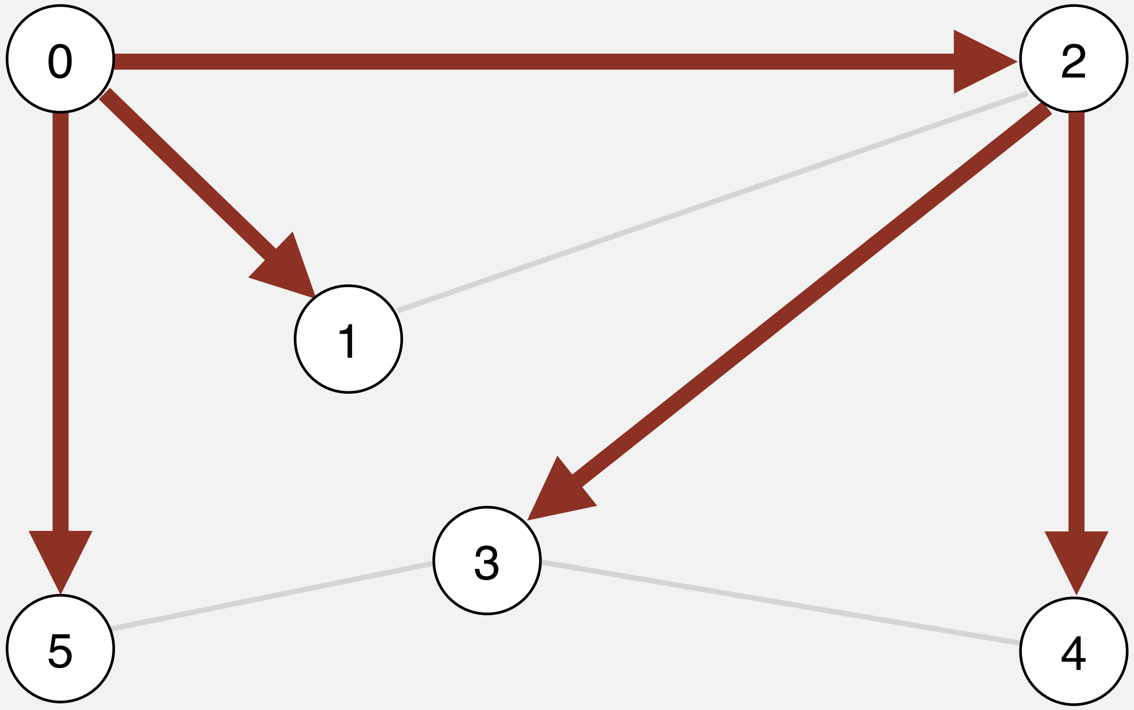
v	edgeTo[v]
0	–
1	0
2	0
3	2
4	2
5	0

4 done

Breadth-first search

Repeat until queue is empty:

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



v	edgeTo[v]
0	–
1	0
2	0
3	2
4	2
5	0

done

Breadth-first search

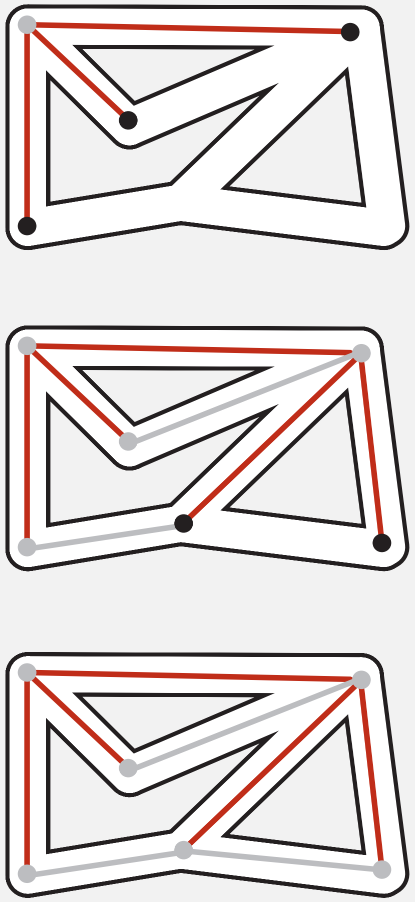
Depth-first search. Put unvisited vertices on a stack.

Breadth-first search. Put unvisited vertices on a queue.

Shortest path. Find path from s to t that uses fewest number of edges.

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
  - add each of v's unvisited neighbors to the queue,
    and mark them as visited.
```



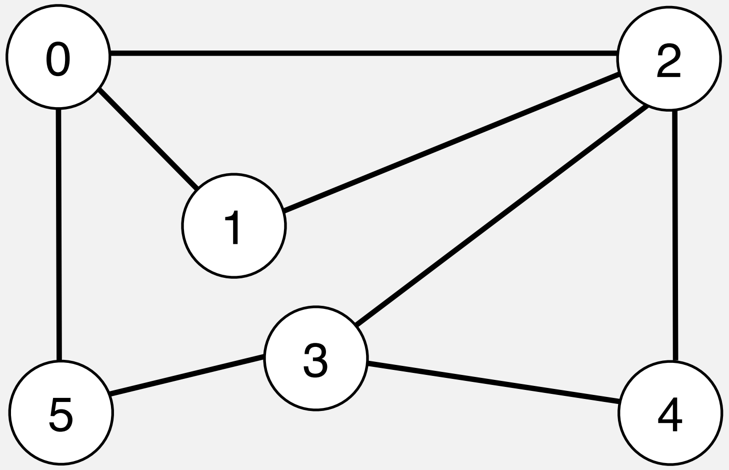
Intuition. BFS examines vertices in increasing distance from s.

Breadth-first search properties

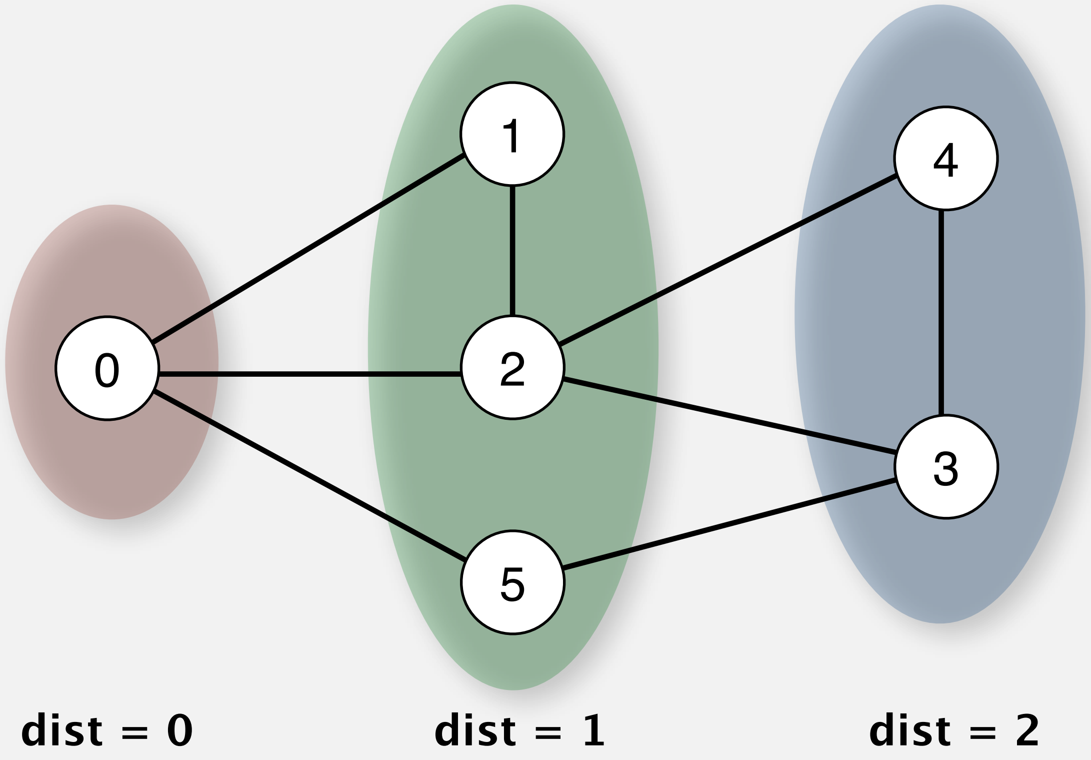
Proposition. BFS computes shortest path (number of edges) from s in a connected graph in time proportional to $E + V$.

Pf. [correctness] Queue always consists of zero or more vertices of distance k from s, followed by zero or more vertices of distance $k + 1$.

Pf. [running time] Each vertex connected to s is visited once.



standard drawing



Breadth-first search

```
public class BreadthFirstPaths
{
   private boolean[] marked;
   private boolean[] edgeTo[];
   private final int s;
   …

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

Undirected Graphs

- Graph API
- Depth-first search
- Breadth-first search
- **Connected components**
- Challenges

Connectivity queries

Def. Vertices v and w are connected if there is a path between them.

Goal. Preprocess graph to answer queries: is v connected to w ? in constant time.

public class CC		
	`CC(Graph G)`	*find connected components in G*
`boolean`	`connected(int v, int w)`	*are v and w connected?*
`int`	`count()`	*number of connected components*
`int`	`id(int v)`	*component identifier for v*

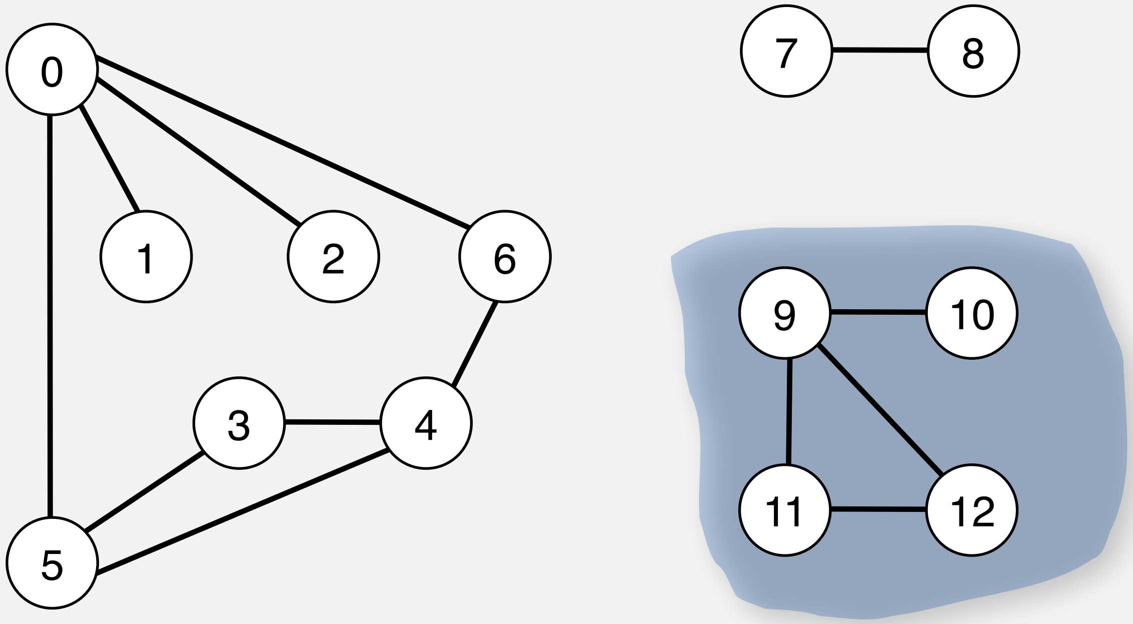
Depth-first search. [next few slides]

Connected components

The relation "is connected to" is an equivalence relation:

- Reflexive: v is connected to v.
- Symmetric: if v is connected to w, then w is connected to v.
- Transitive: if v connected to w and w connected to x, then v connected to x.

Def. A connected component is a maximal set of connected vertices.



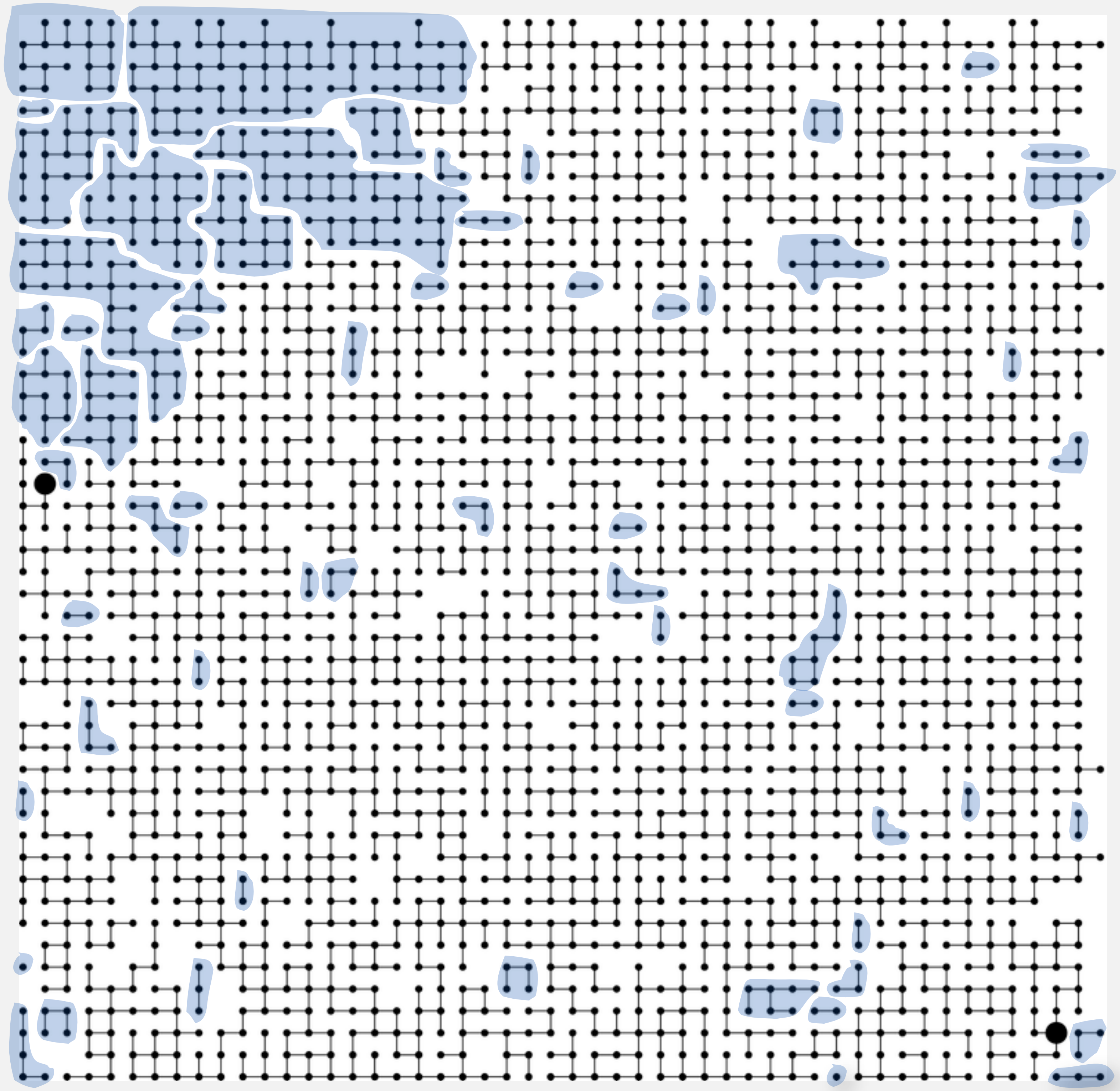
3 connected components

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

Remark. Given connected components, can answer queries in constant time.

Connected components

Def. A connected component is a maximal set of connected vertices.



63 connected components

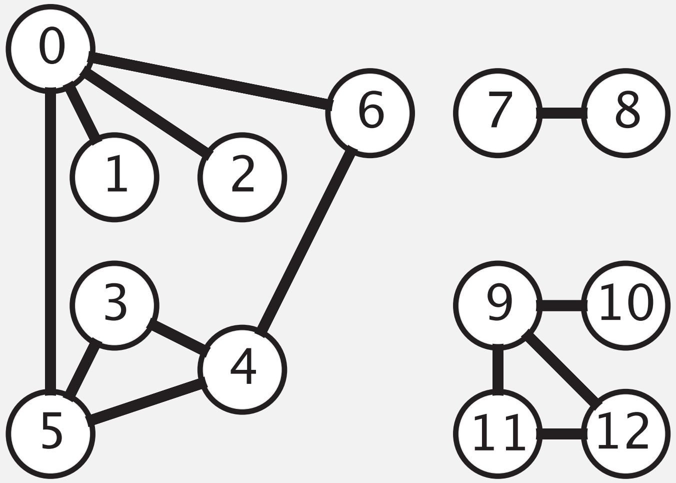
Connected components

Goal. Partition vertices into connected components.

Connected components

Initialize all vertices v as unmarked.

For each unmarked vertex v, run DFS to identify all vertices discovered as part of the same component.



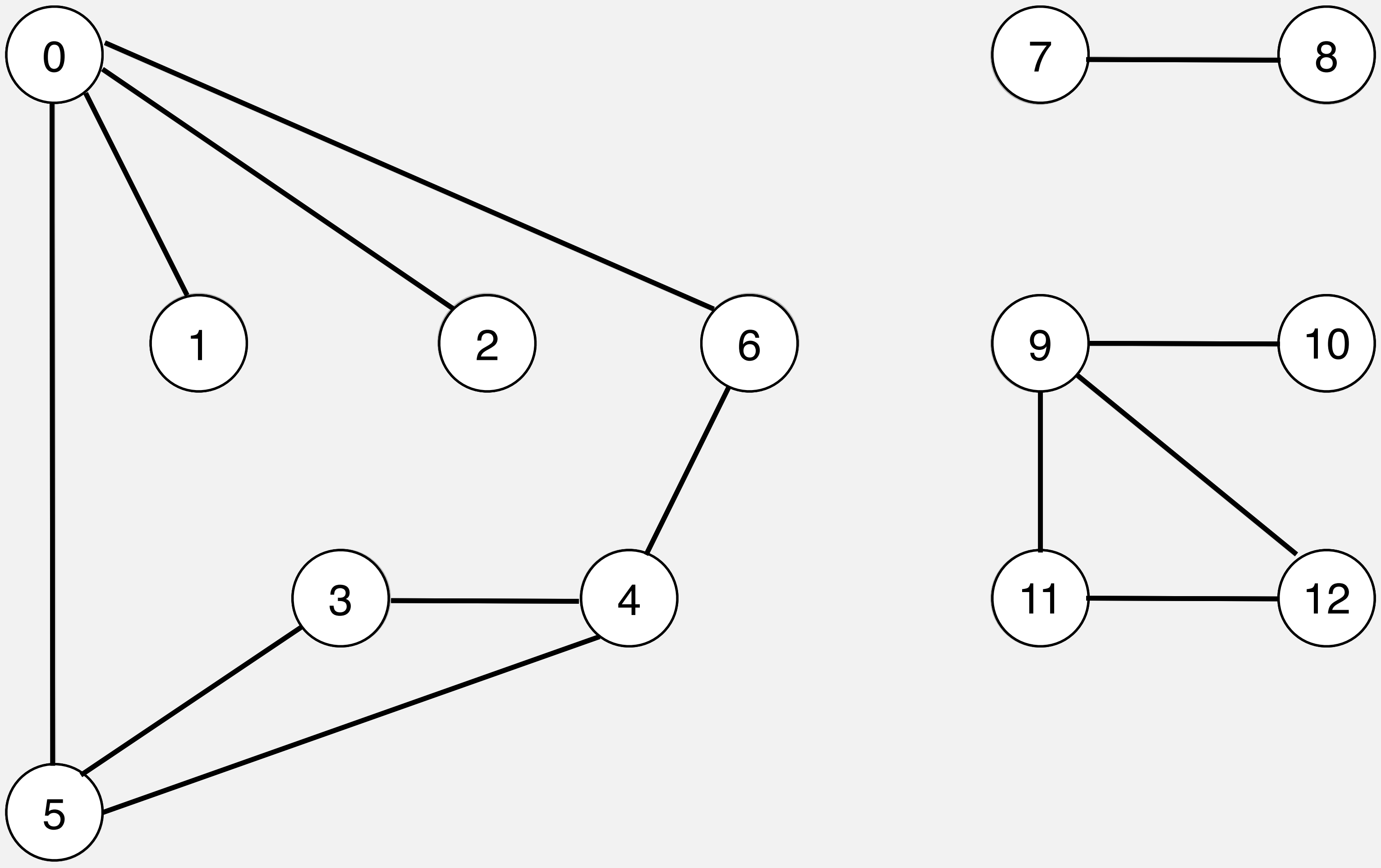
tinyG.txt

```
V → 13
    13 ← E
    0  5
    4  3
    0  1
    9  12
    6  4
    5  4
    0  2
    11  12
    9  10
    0  6
    7  8
    9  11
    5  3
```

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



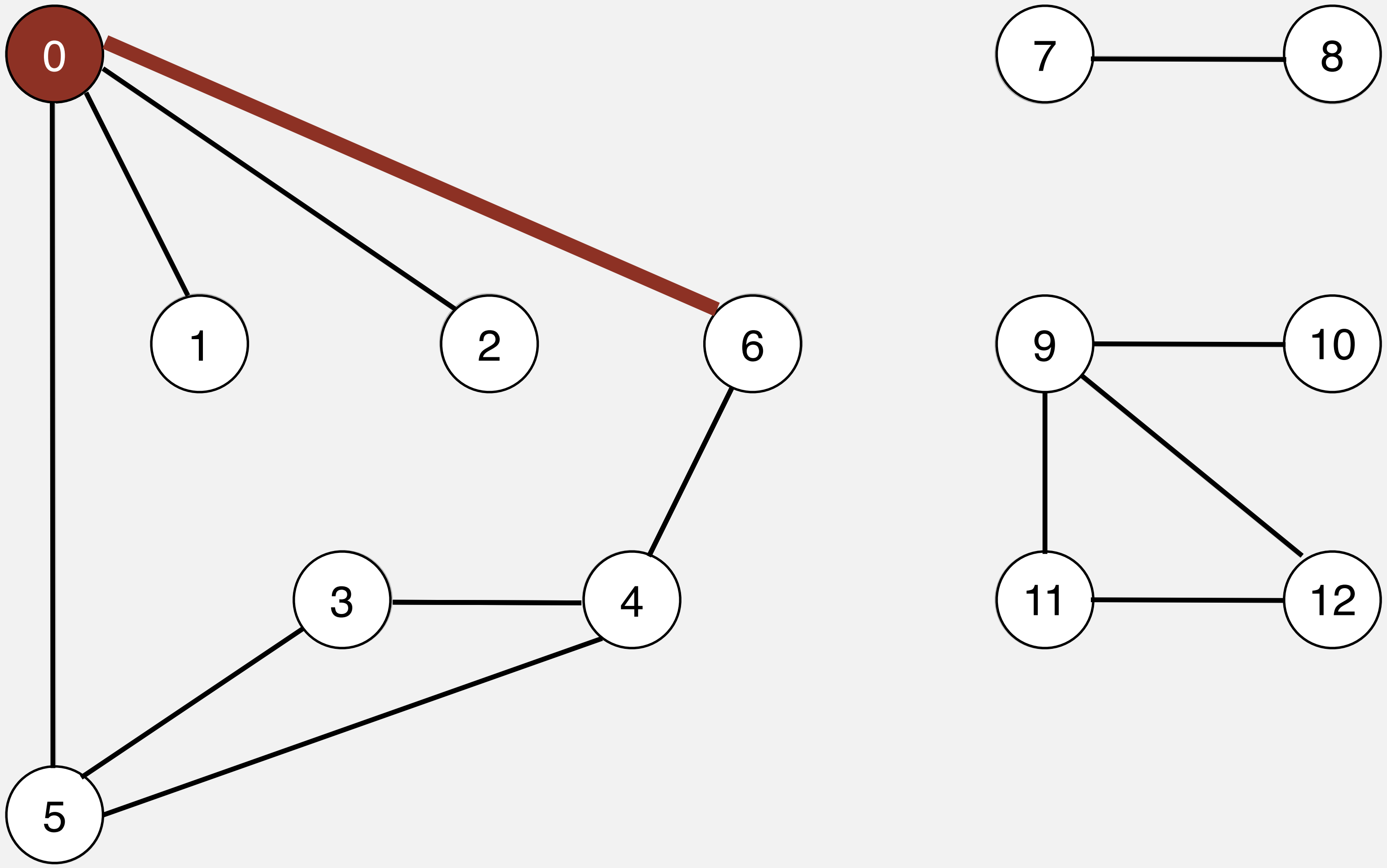
graph G

v	marked[]	cc[]
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	–

Connected components

To visit a vertex *v* :

- Mark vertex *v* as visited.
- Recursively visit all unmarked vertices adjacent to *v*.



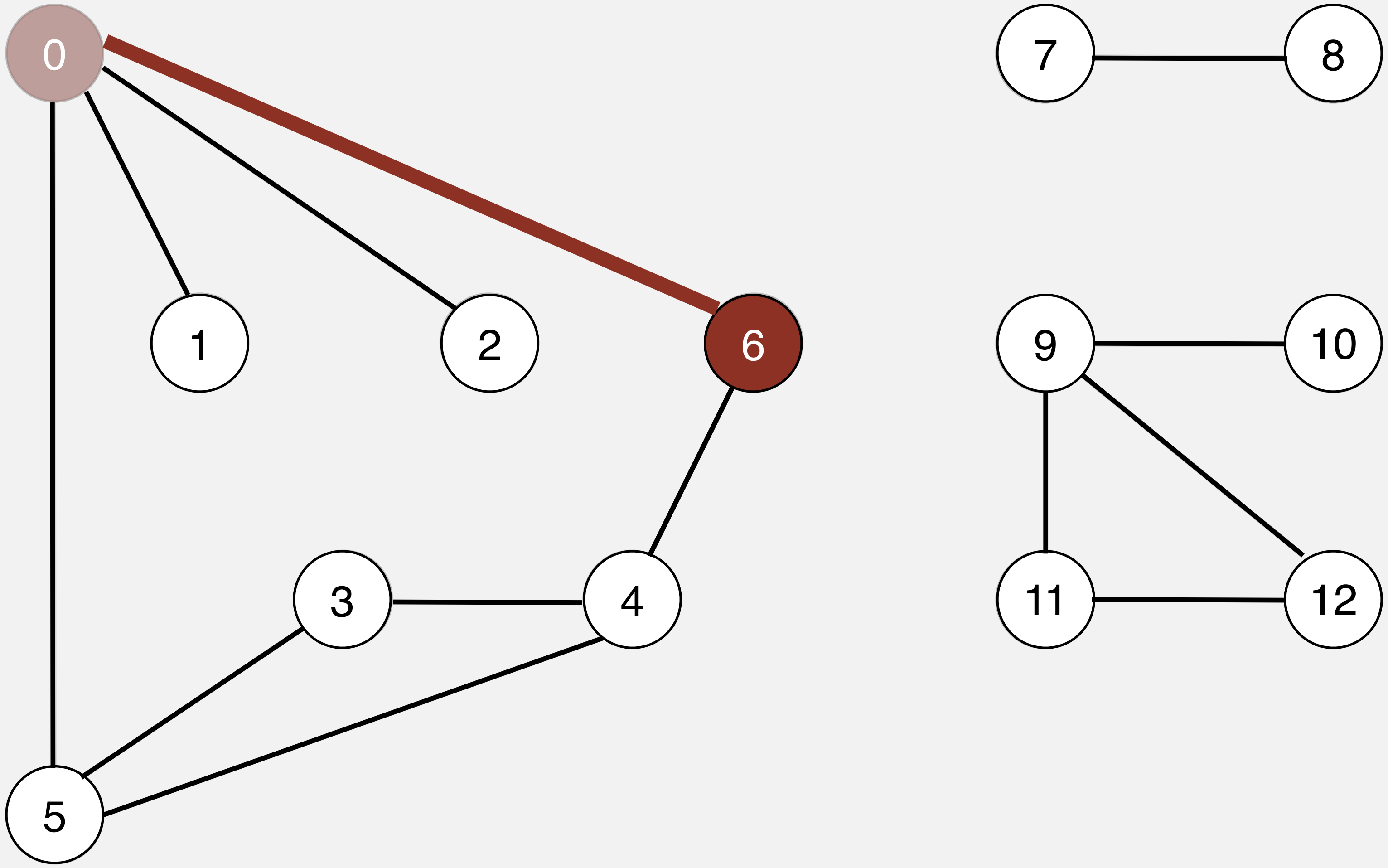
v	marked[]	cc[]
0	T	0
1	F	–
2	F	–
3	F	–
4	F	–
5	F	–
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 0: check 6, check 2, check 1 and check 5

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



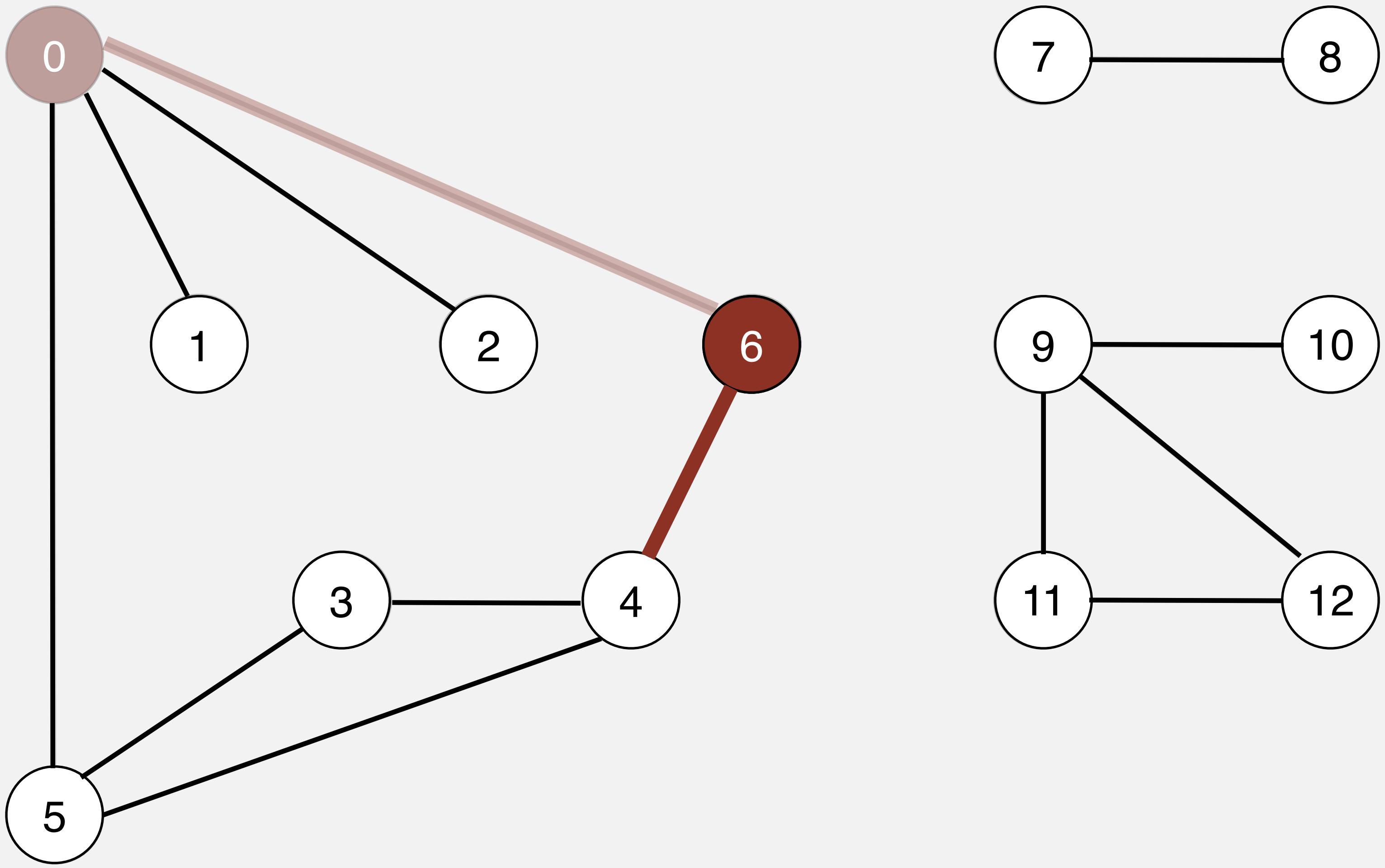
v	marked[]	cc[]
0	T	0
1	F	–
2	F	–
3	F	–
4	F	–
5	F	–
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 6: check 0 and check 4

Connected components

To visit a vertex *v* :

- Mark vertex *v* as visited.
- Recursively visit all unmarked vertices adjacent to *v*.



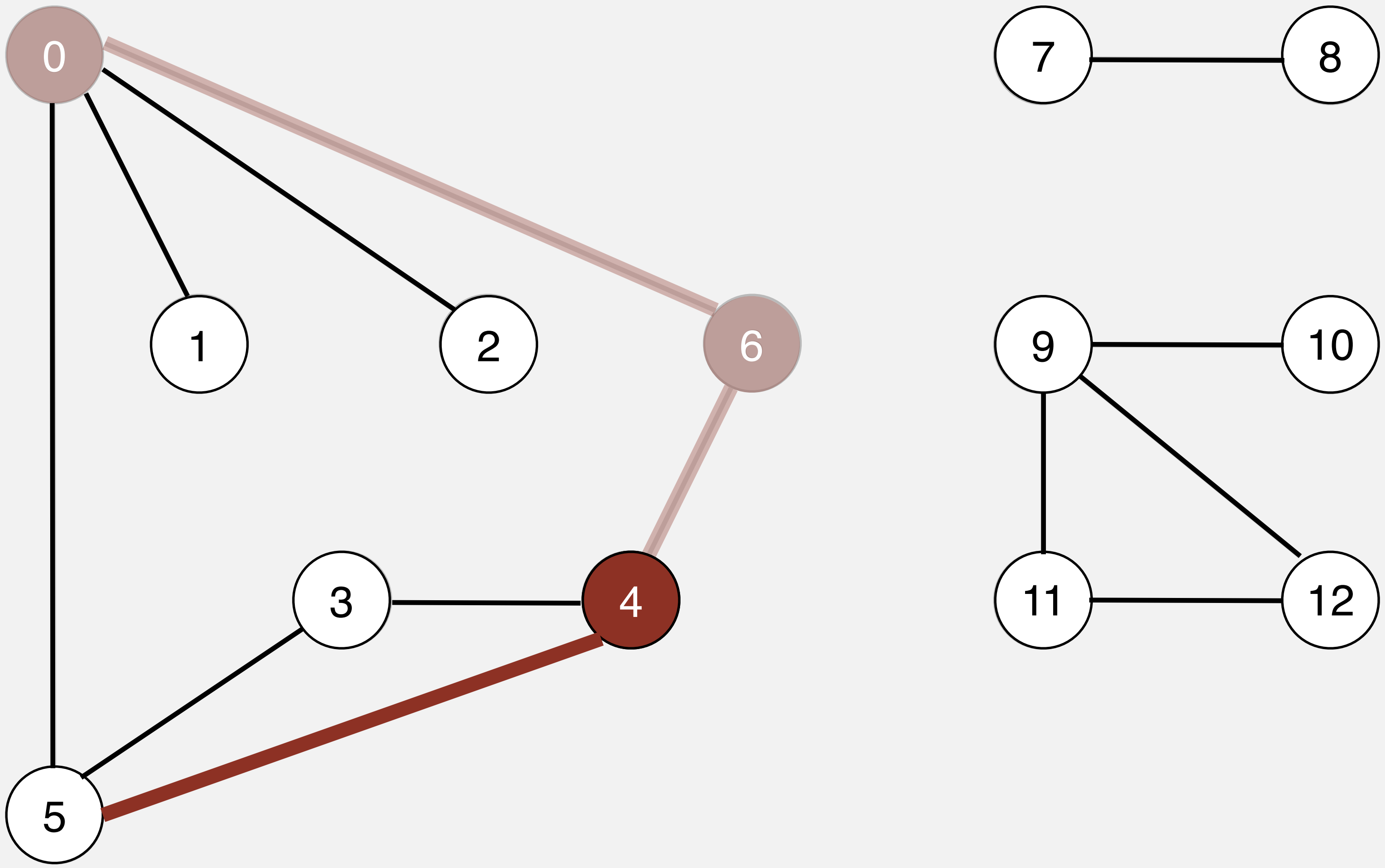
visit 6: check 0 and check 4

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

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



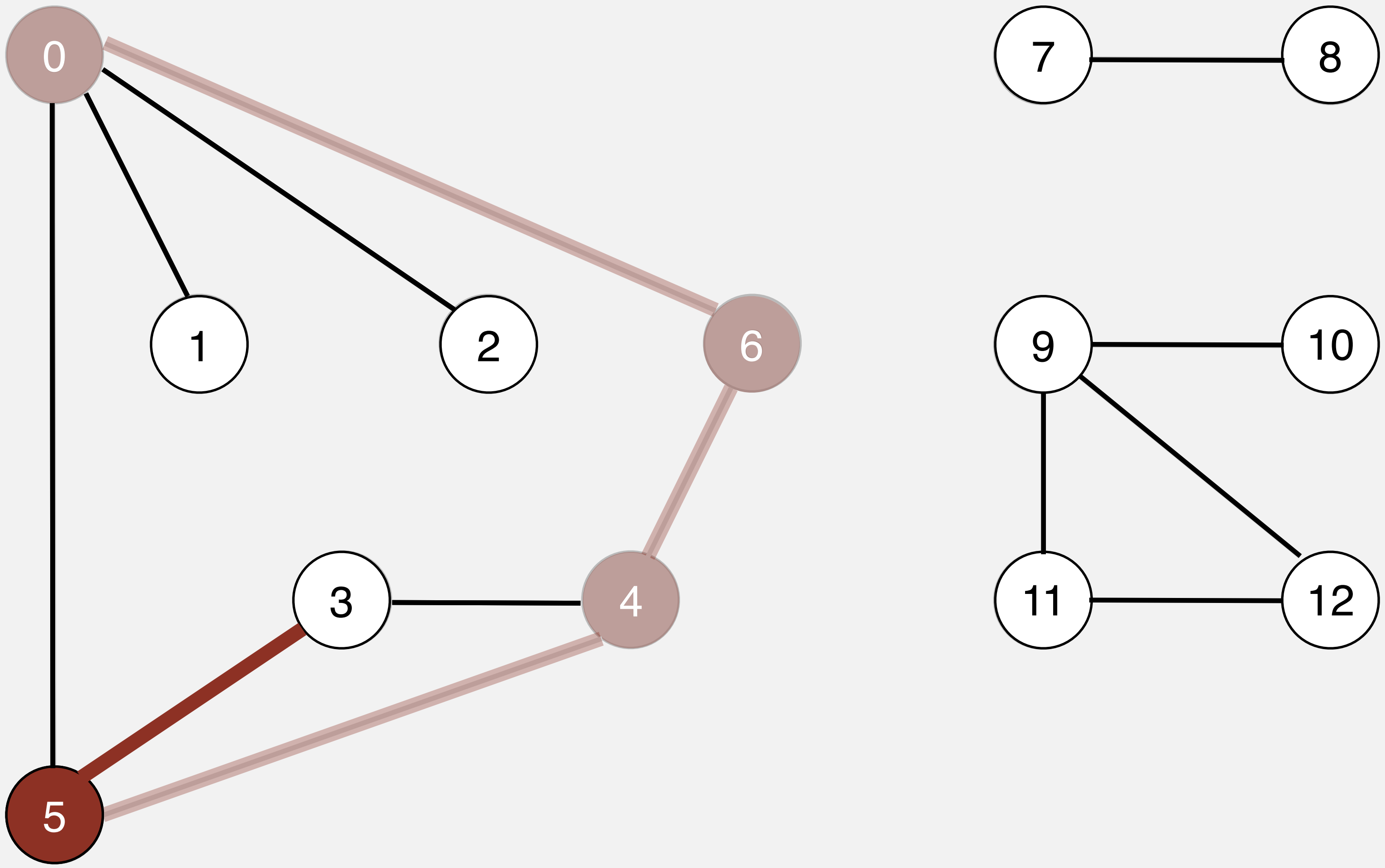
visit 4: check 5, check 6 and check 3

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

Connected components

To visit a vertex *v* :

- Mark vertex *v* as visited.
- Recursively visit all unmarked vertices adjacent to *v*.



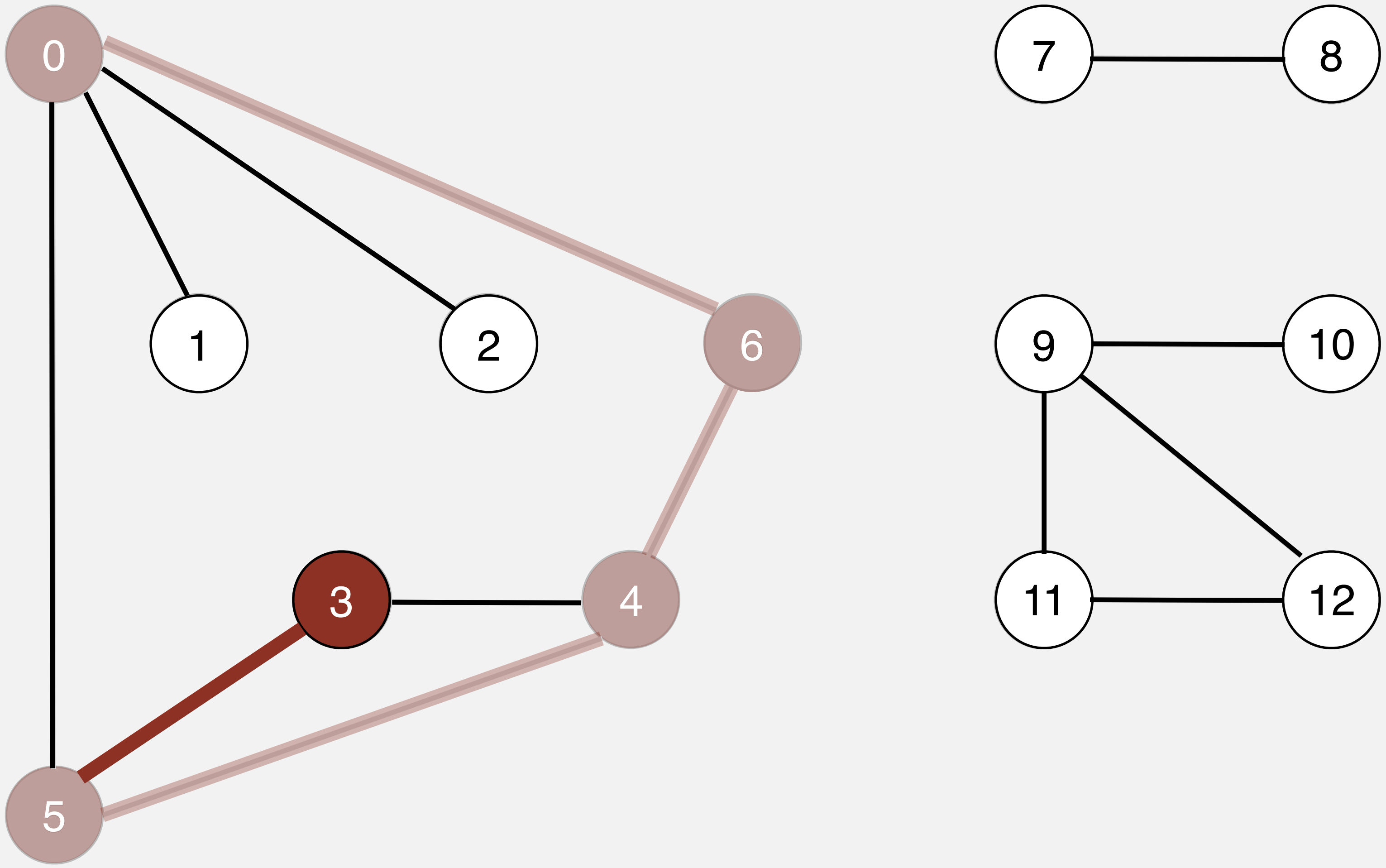
visit 5: check 3, check 4 and check 0

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

Connected components

To visit a vertex *v* :

- Mark vertex *v* as visited.
- Recursively visit all unmarked vertices adjacent to *v*.



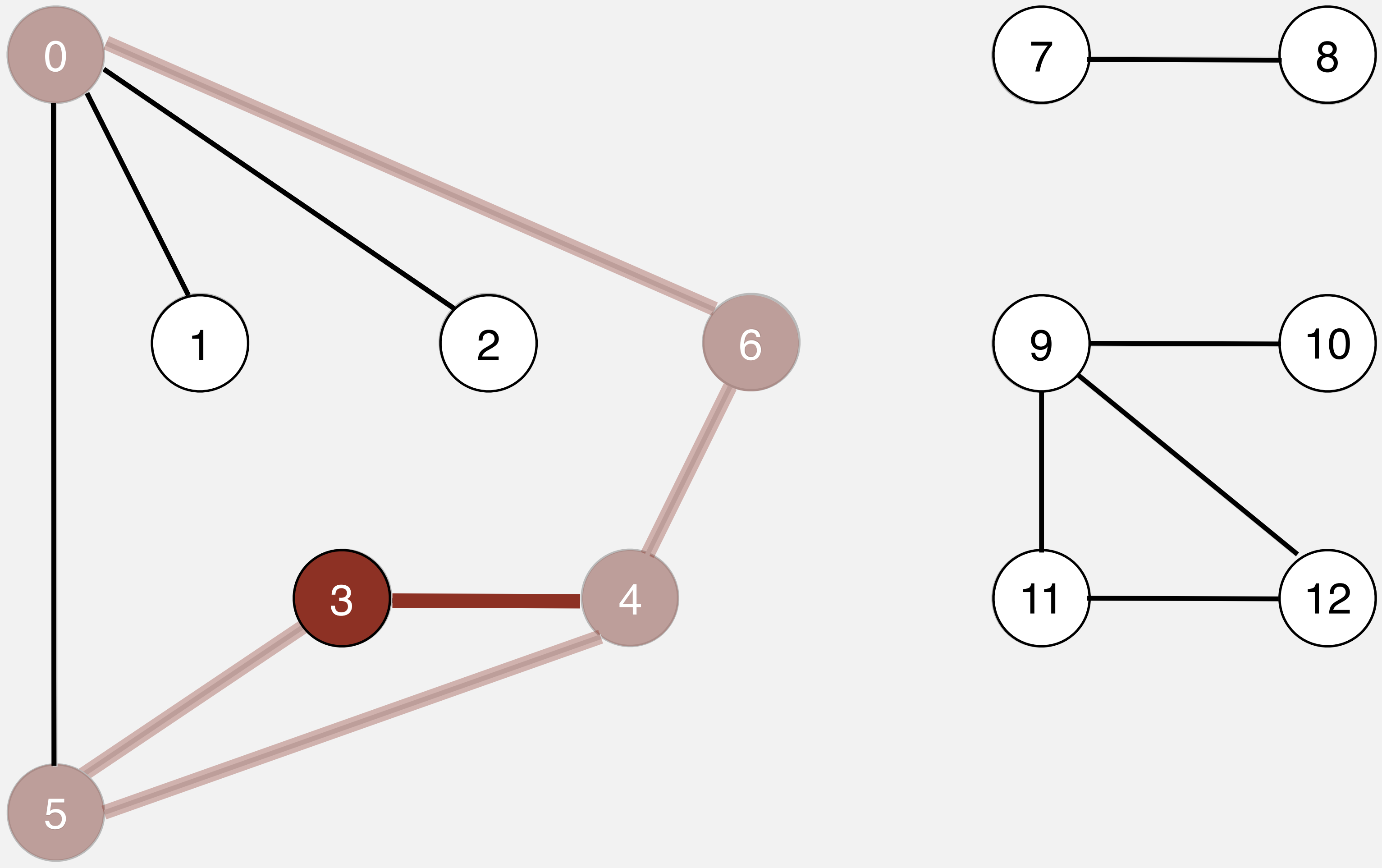
visit 3: check 5 and check 4

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

Connected components

To visit a vertex *v* :

- Mark vertex *v* as visited.
- Recursively visit all unmarked vertices adjacent to *v*.



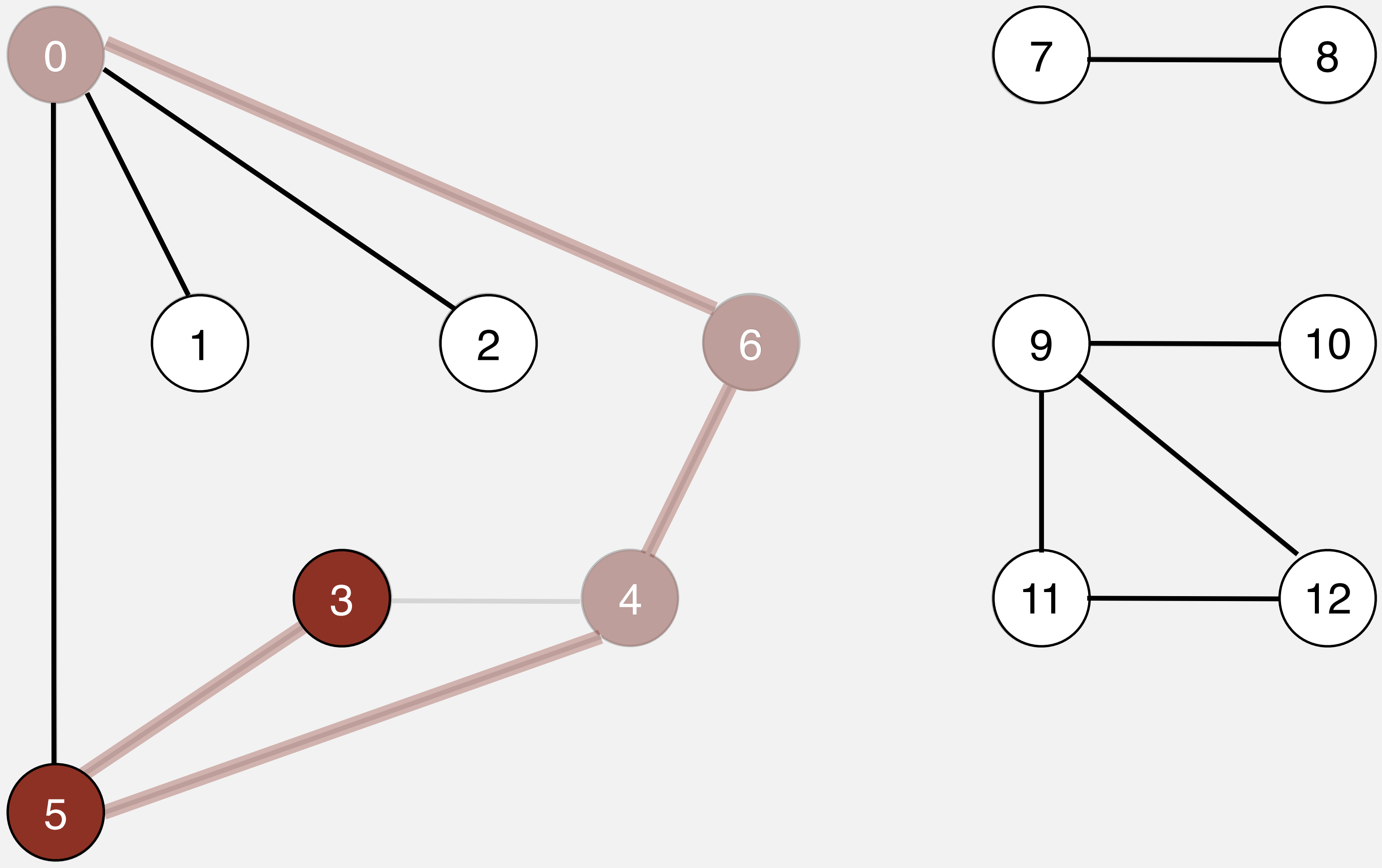
visit 3: check 5 and check 4

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

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



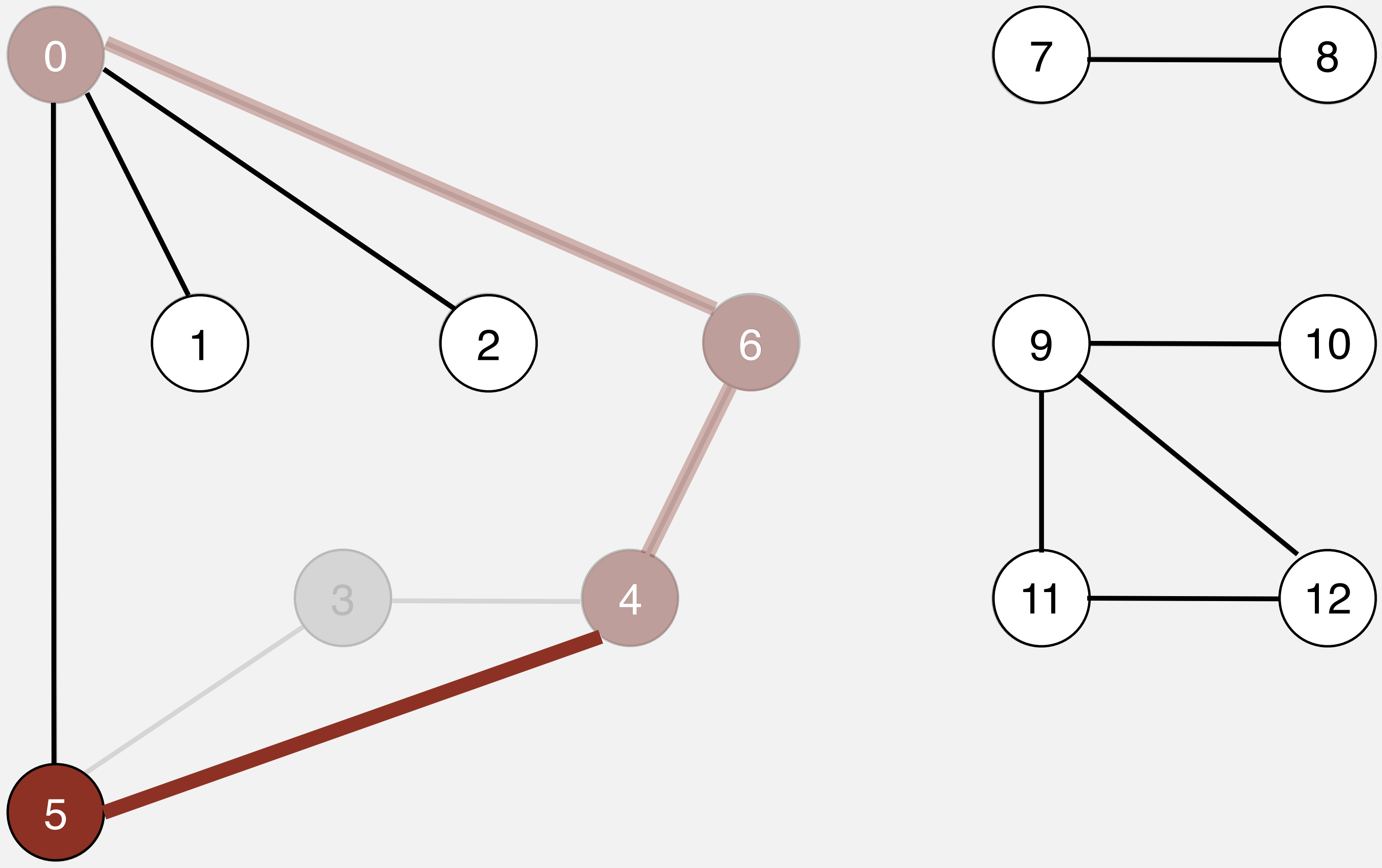
3 done

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

Connected components

To visit a vertex *v* :

- Mark vertex *v* as visited.
- Recursively visit all unmarked vertices adjacent to *v*.



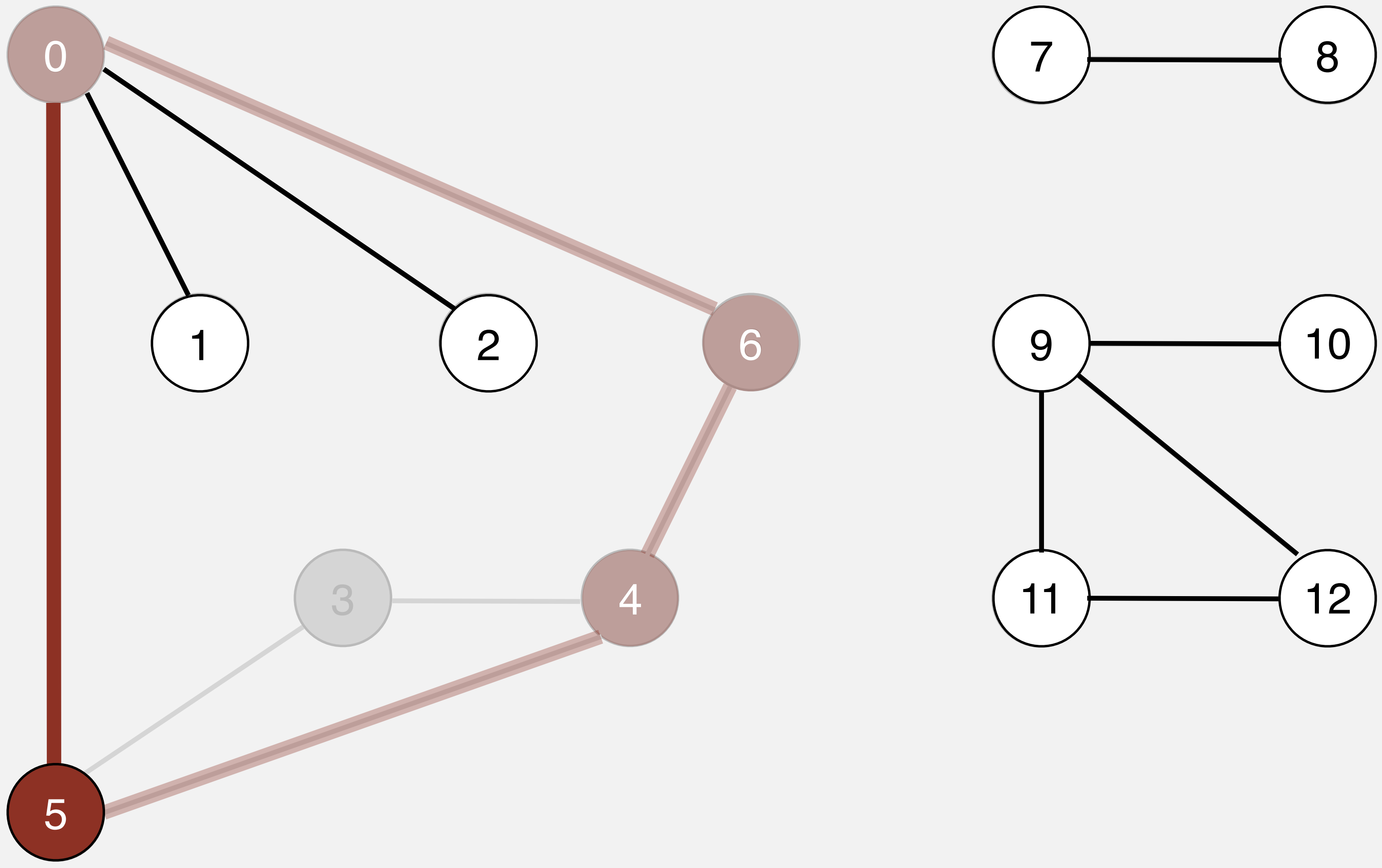
v	marked[]	cc[]
0	T	0
1	F	–
2	F	–
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 5: check 3, check 4 and check 0

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



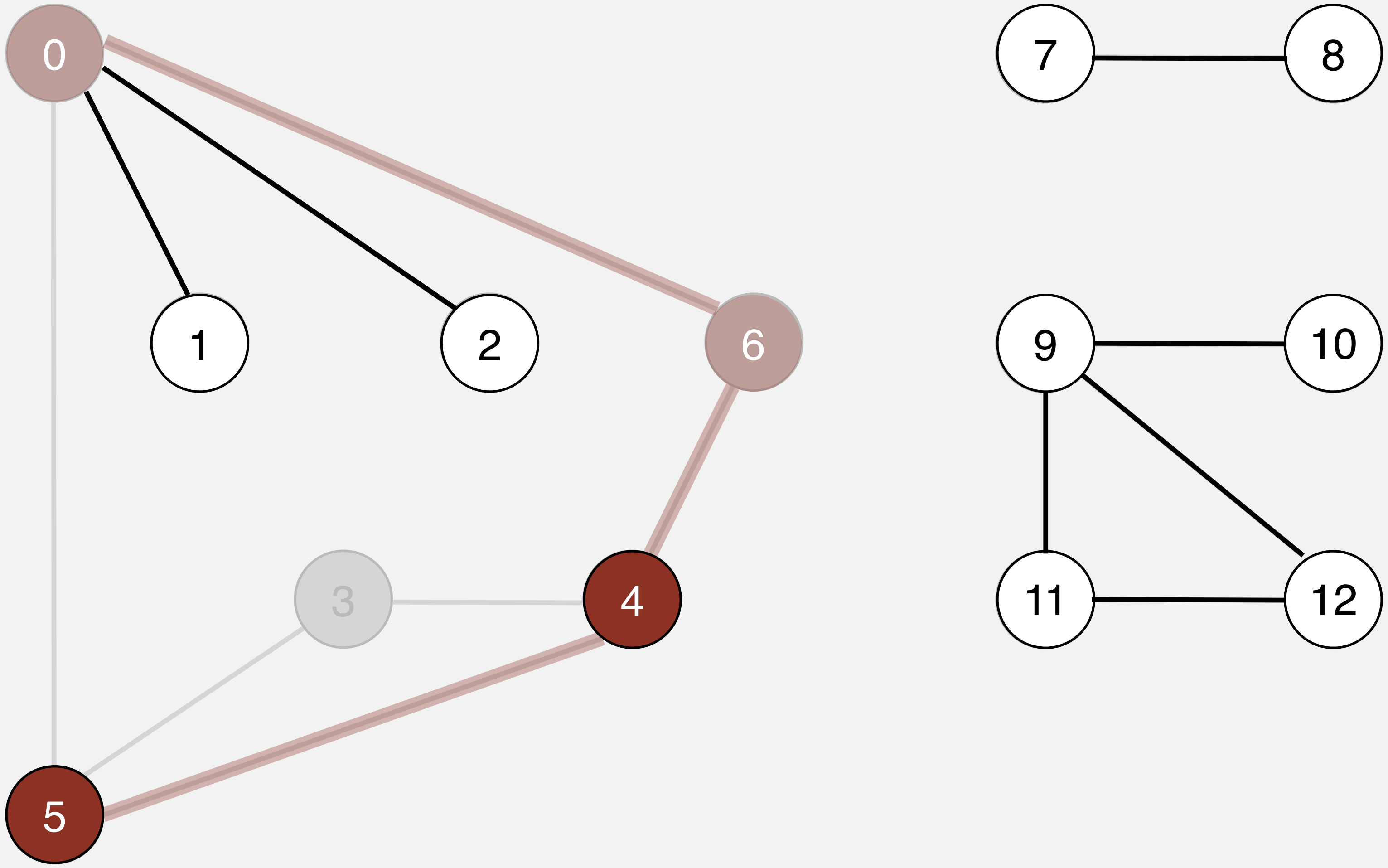
visit 5: check 3, check 4 and check 0

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

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



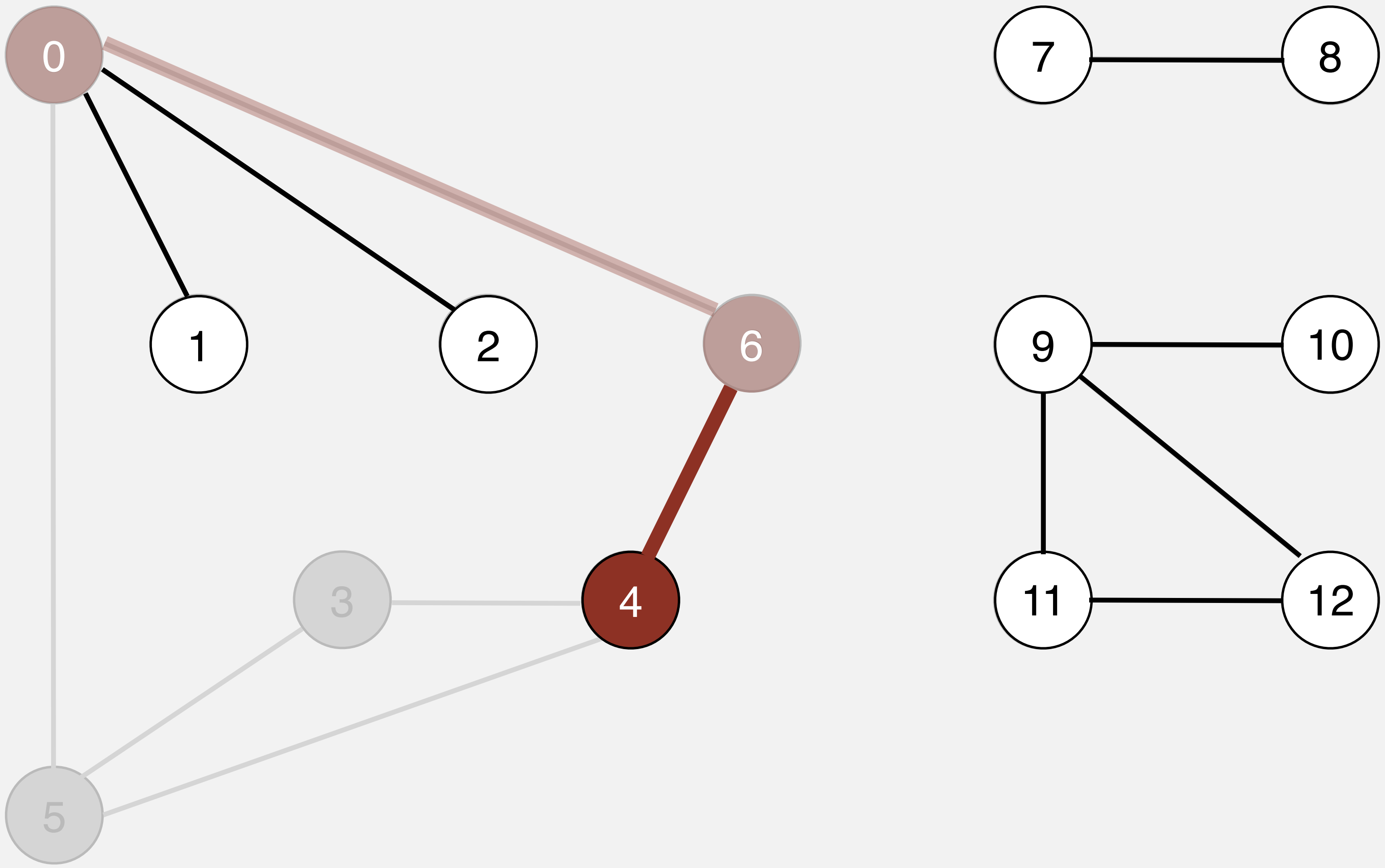
5 done

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

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



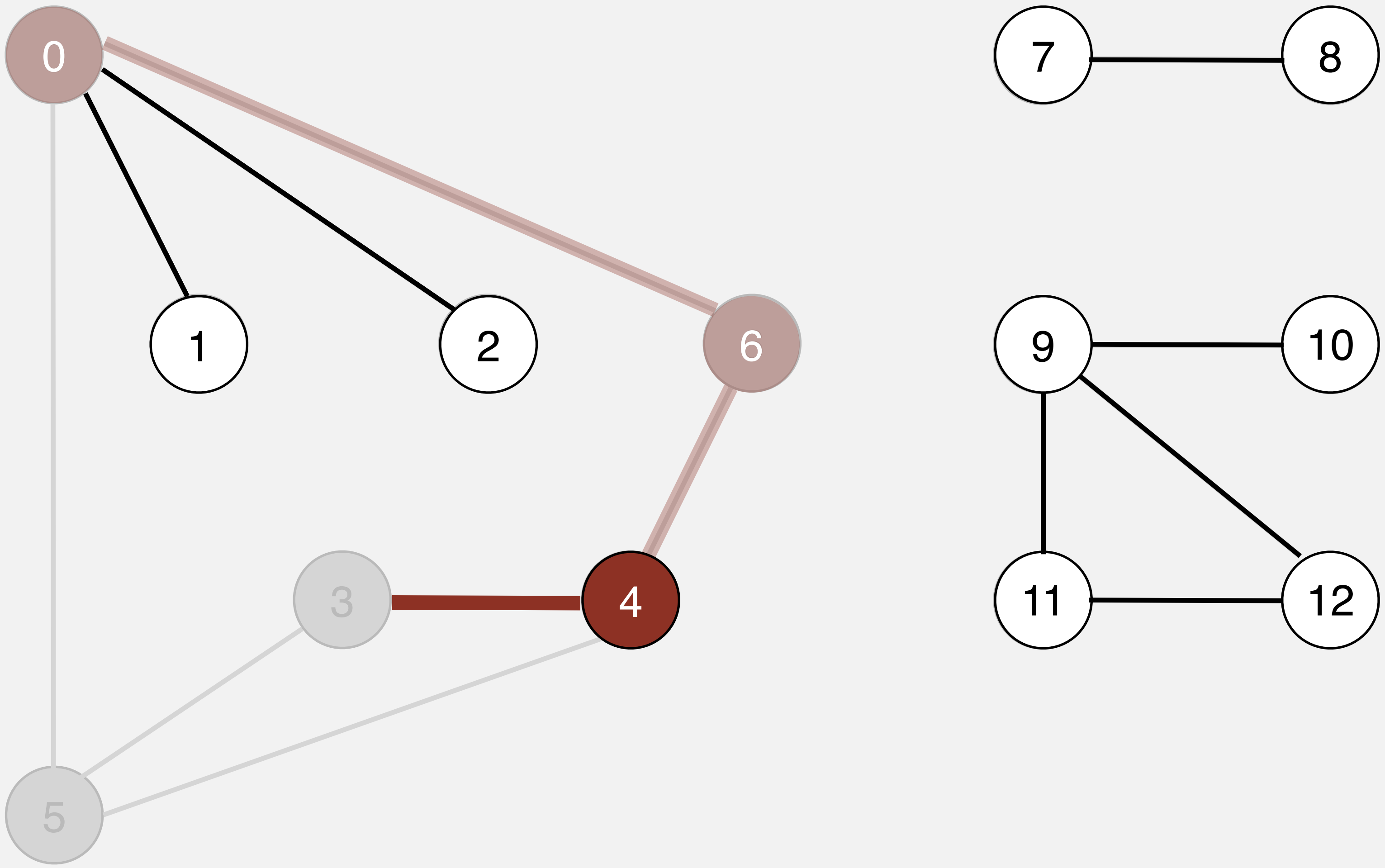
visit 4: check 5, check 6 and check 3

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

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



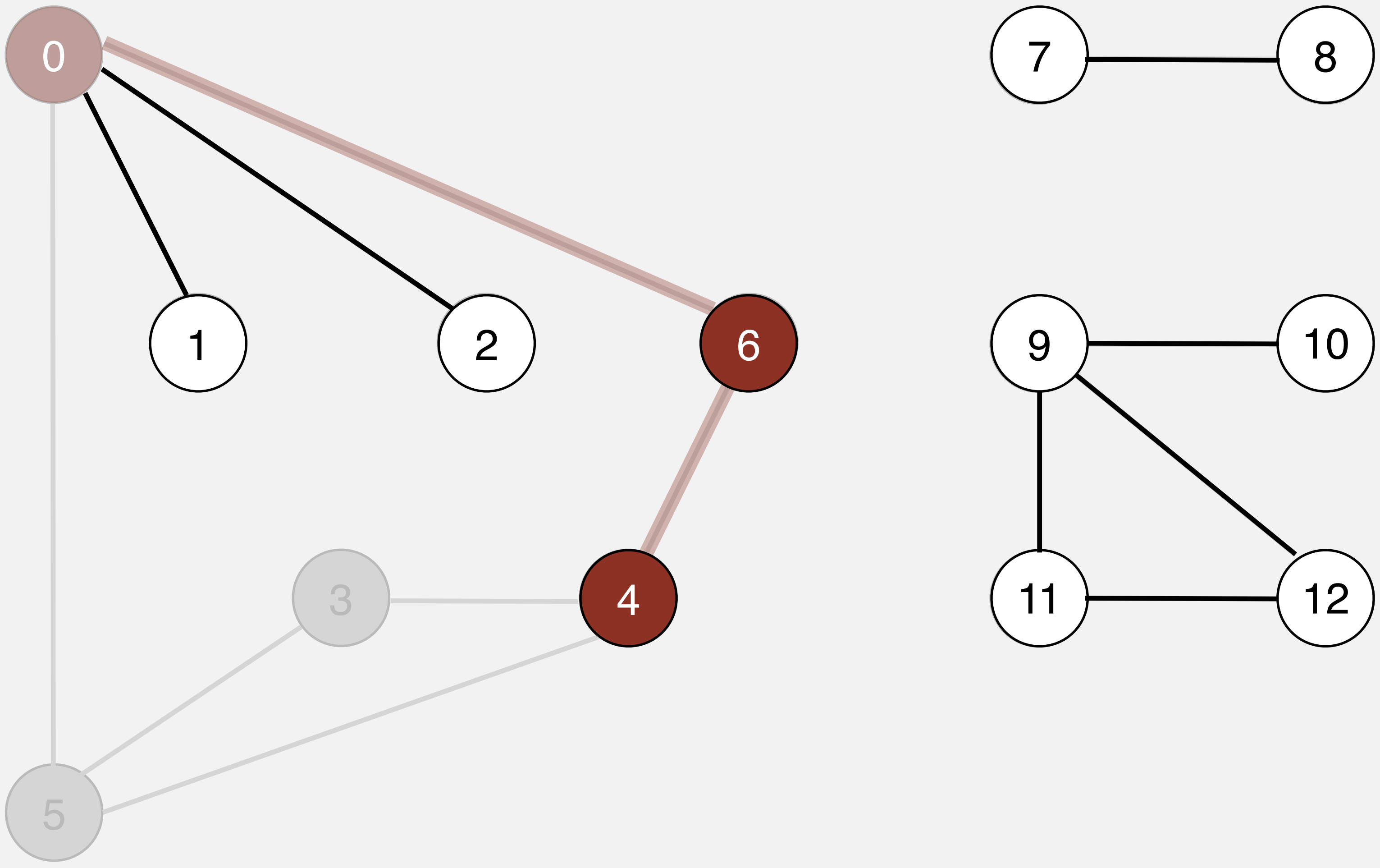
visit 4: check 5, check 6 and check 3

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

Connected components

To visit a vertex *v* :

- Mark vertex *v* as visited.
- Recursively visit all unmarked vertices adjacent to *v*.



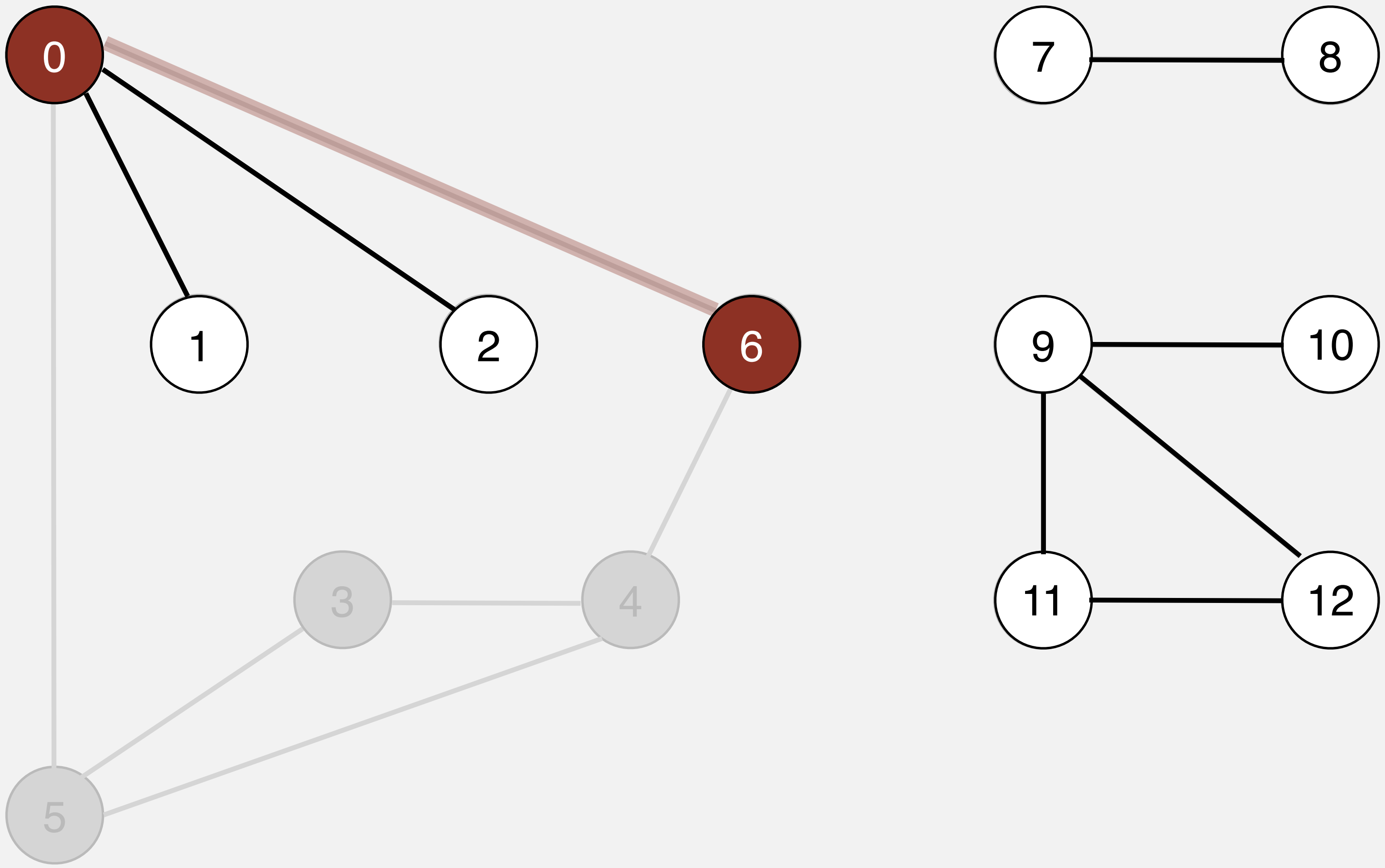
4 done

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

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



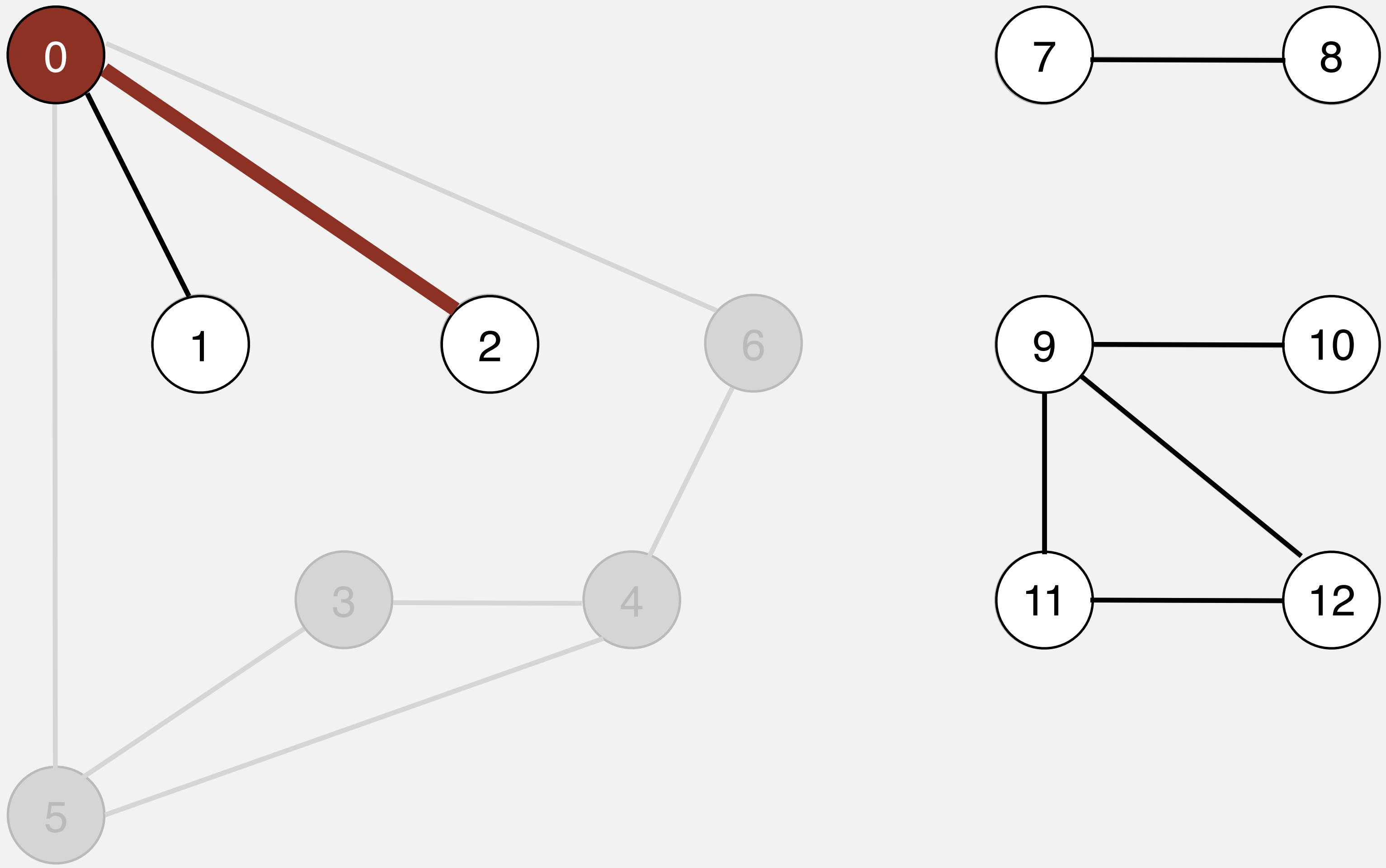
v	marked[]	cc[]
0	T	0
1	F	–
2	F	–
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

6 done

Connected components

To visit a vertex *v* :

- Mark vertex *v* as visited.
- Recursively visit all unmarked vertices adjacent to *v*.



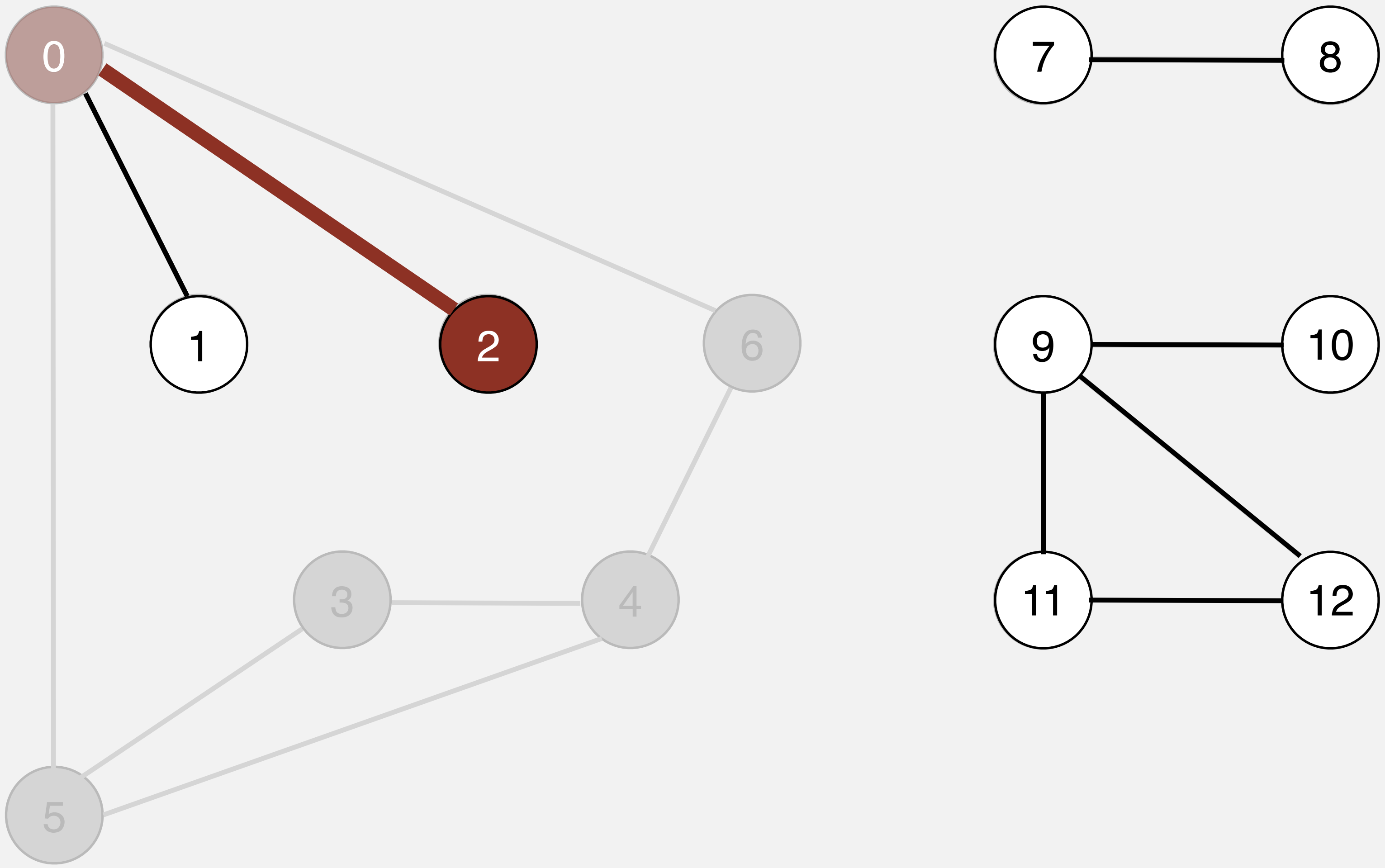
v	marked[]	cc[]
0	T	0
1	F	–
2	F	–
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 0: check 6, **check 2**, check 1 and check 5

Connected components

To visit a vertex *v* :

- Mark vertex *v* as visited.
- Recursively visit all unmarked vertices adjacent to *v*.



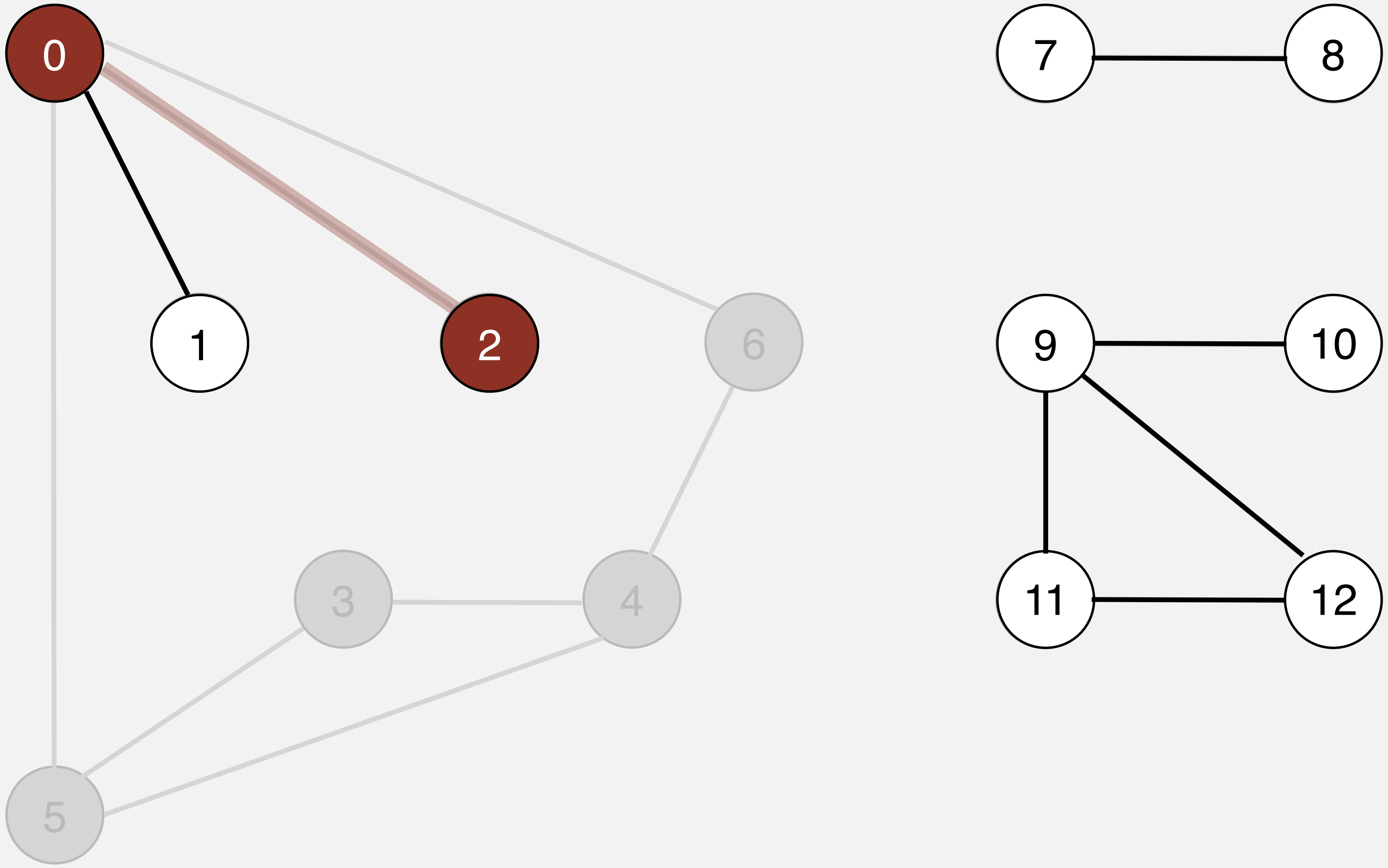
visit 2: check 0

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

Connected components

To visit a vertex *v* :

- Mark vertex *v* as visited.
- Recursively visit all unmarked vertices adjacent to *v*.



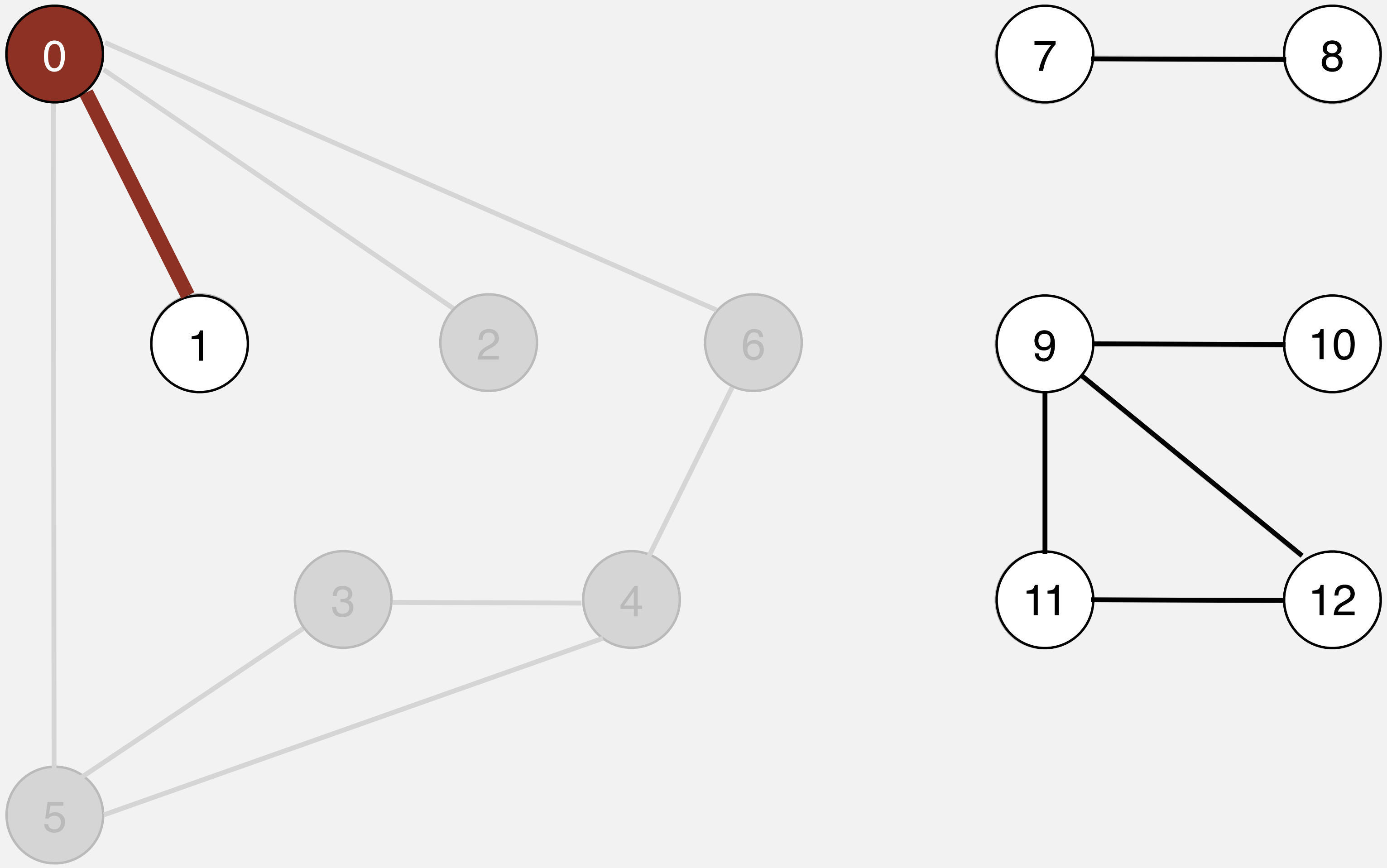
2 done

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

Connected components

To visit a vertex *v* :

- Mark vertex *v* as visited.
- Recursively visit all unmarked vertices adjacent to *v*.



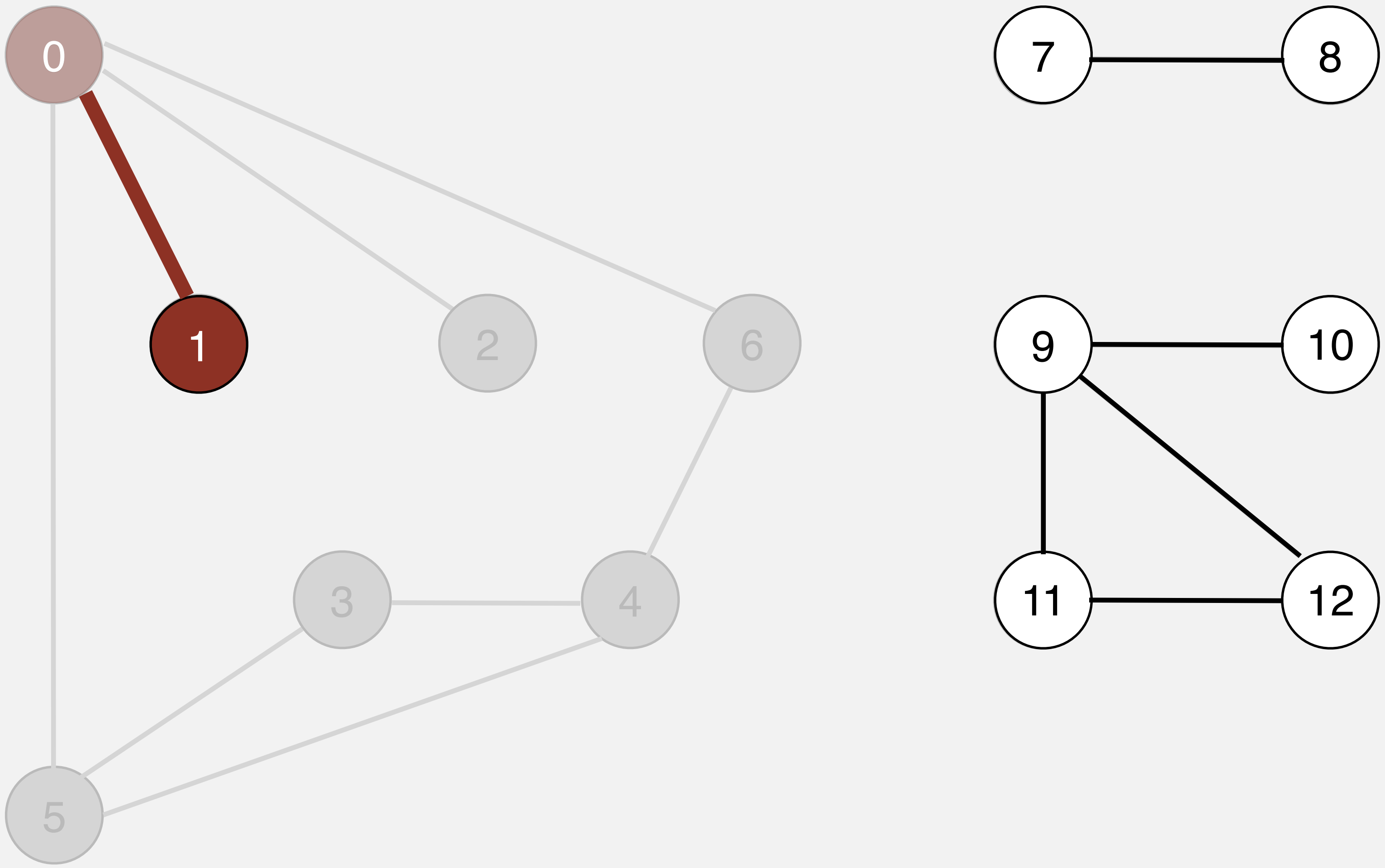
v	marked[]	cc[]
0	T	0
1	F	–
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 0: check 6, check 2, **check 1** and check 5

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



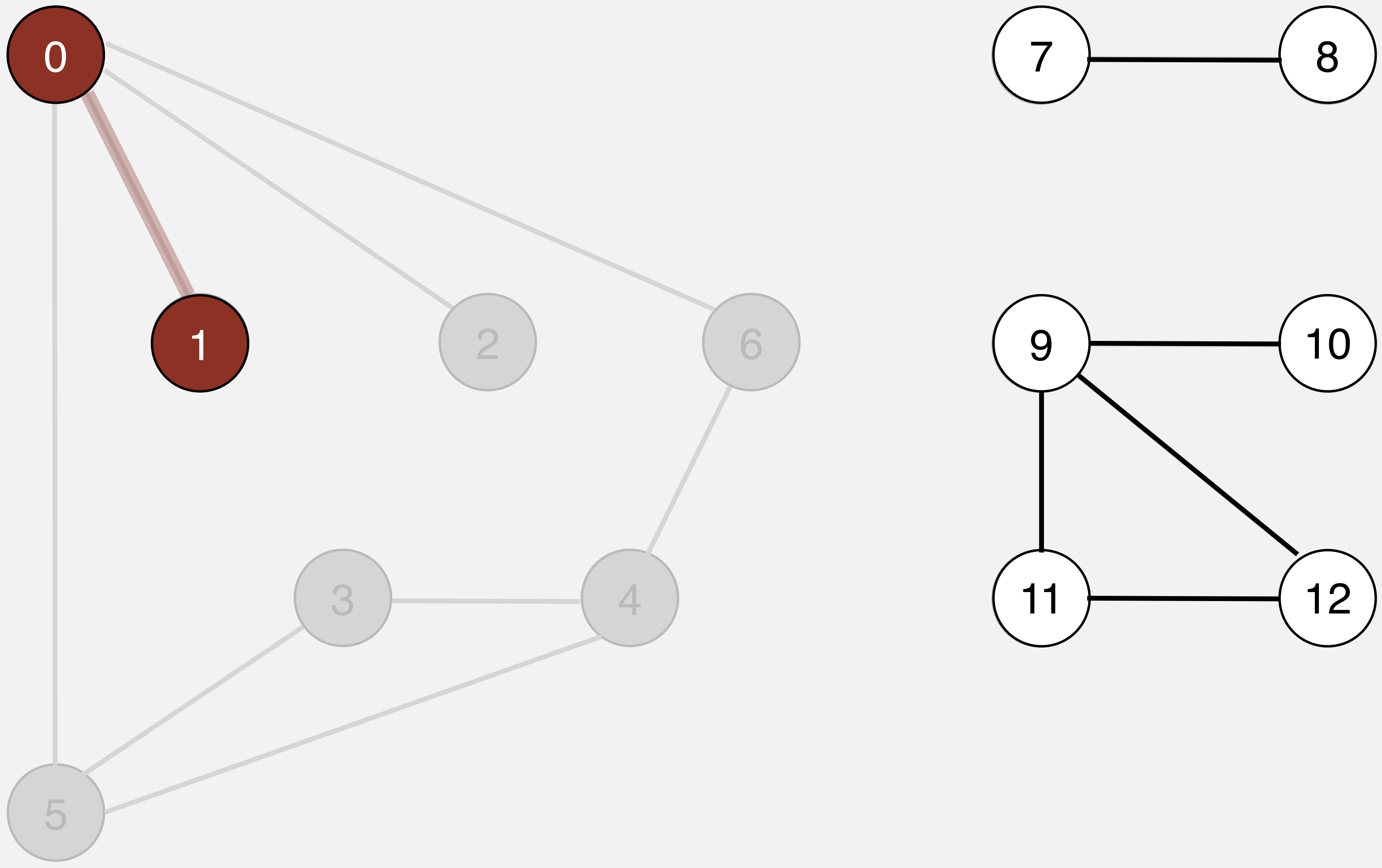
visit 1: check 0

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

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



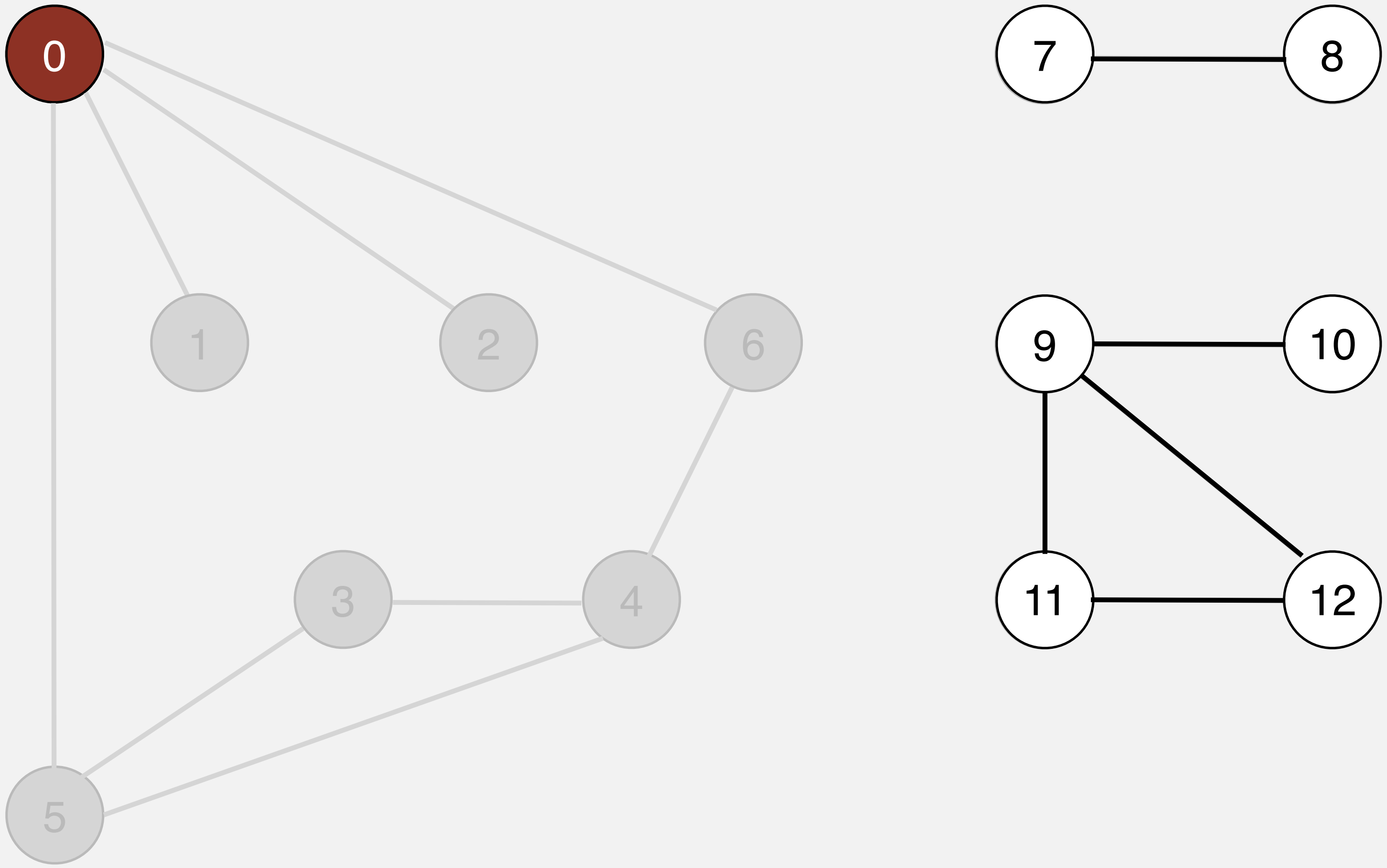
1 done

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

Connected components

To visit a vertex *v* :

- Mark vertex *v* as visited.
- Recursively visit all unmarked vertices adjacent to *v*.



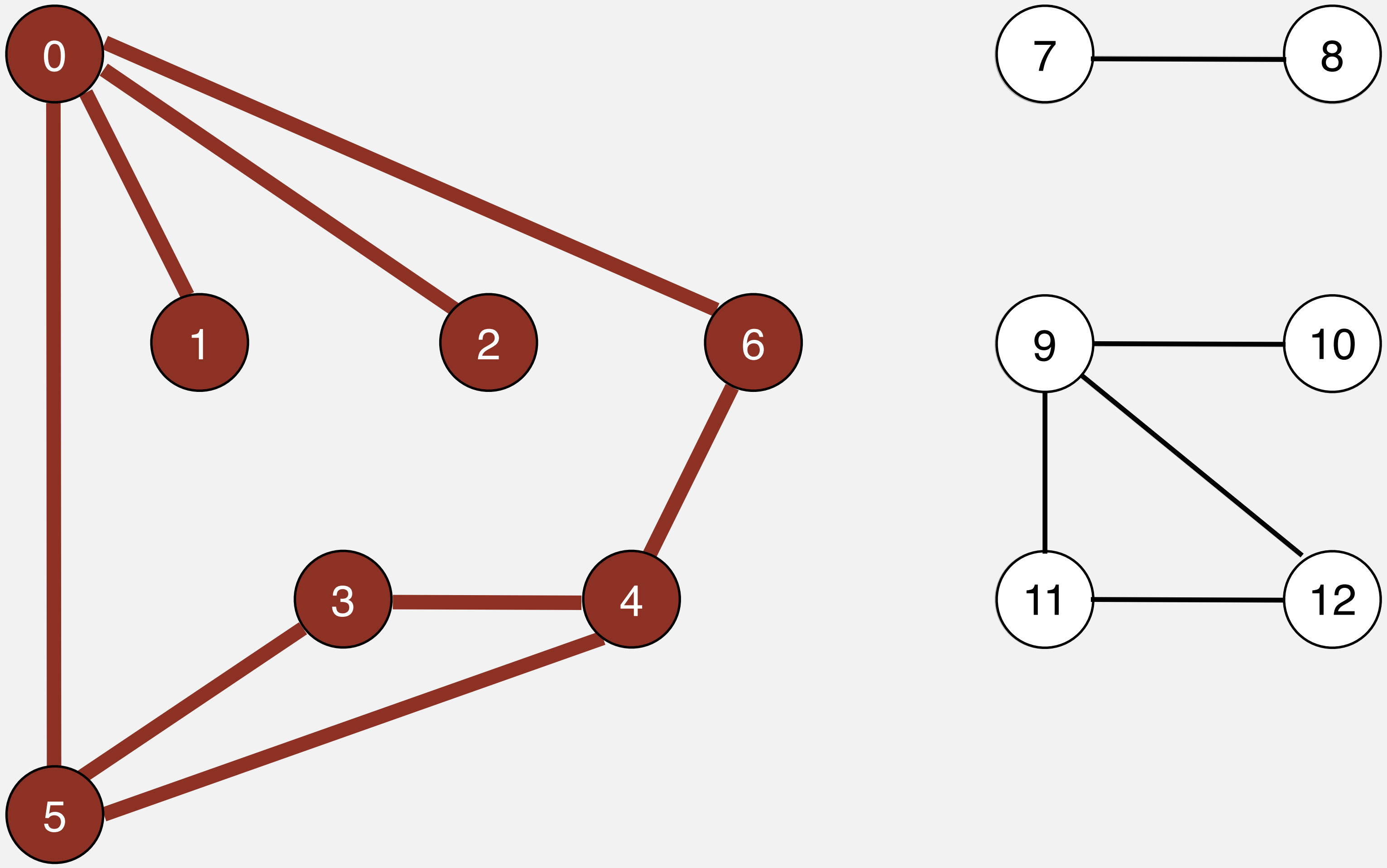
0 done

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

Connected components

To visit a vertex *v* :

- Mark vertex *v* as visited.
- Recursively visit all unmarked vertices adjacent to *v*.



connected component: 0 1 2 3 4 5 6

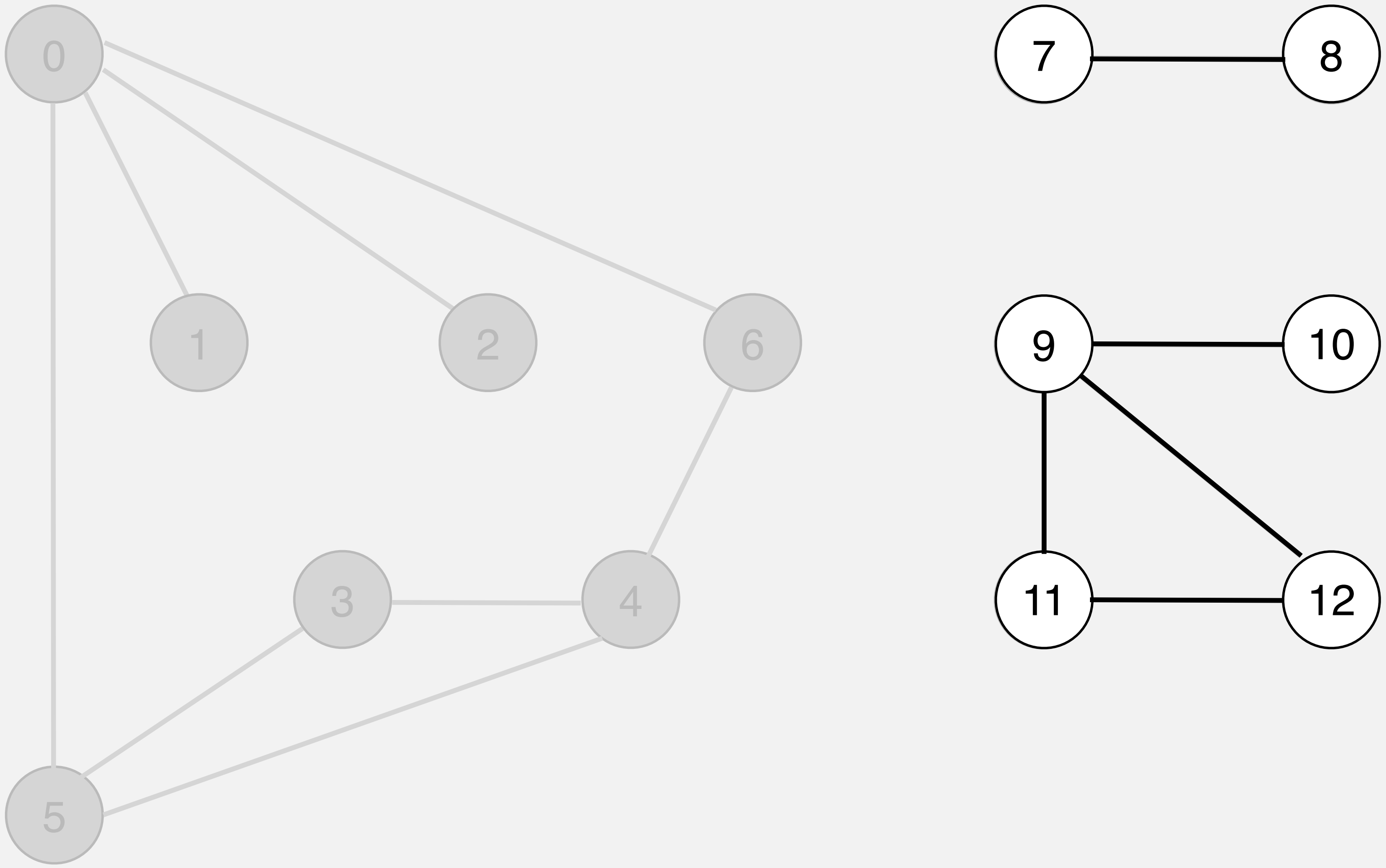
connected component

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

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



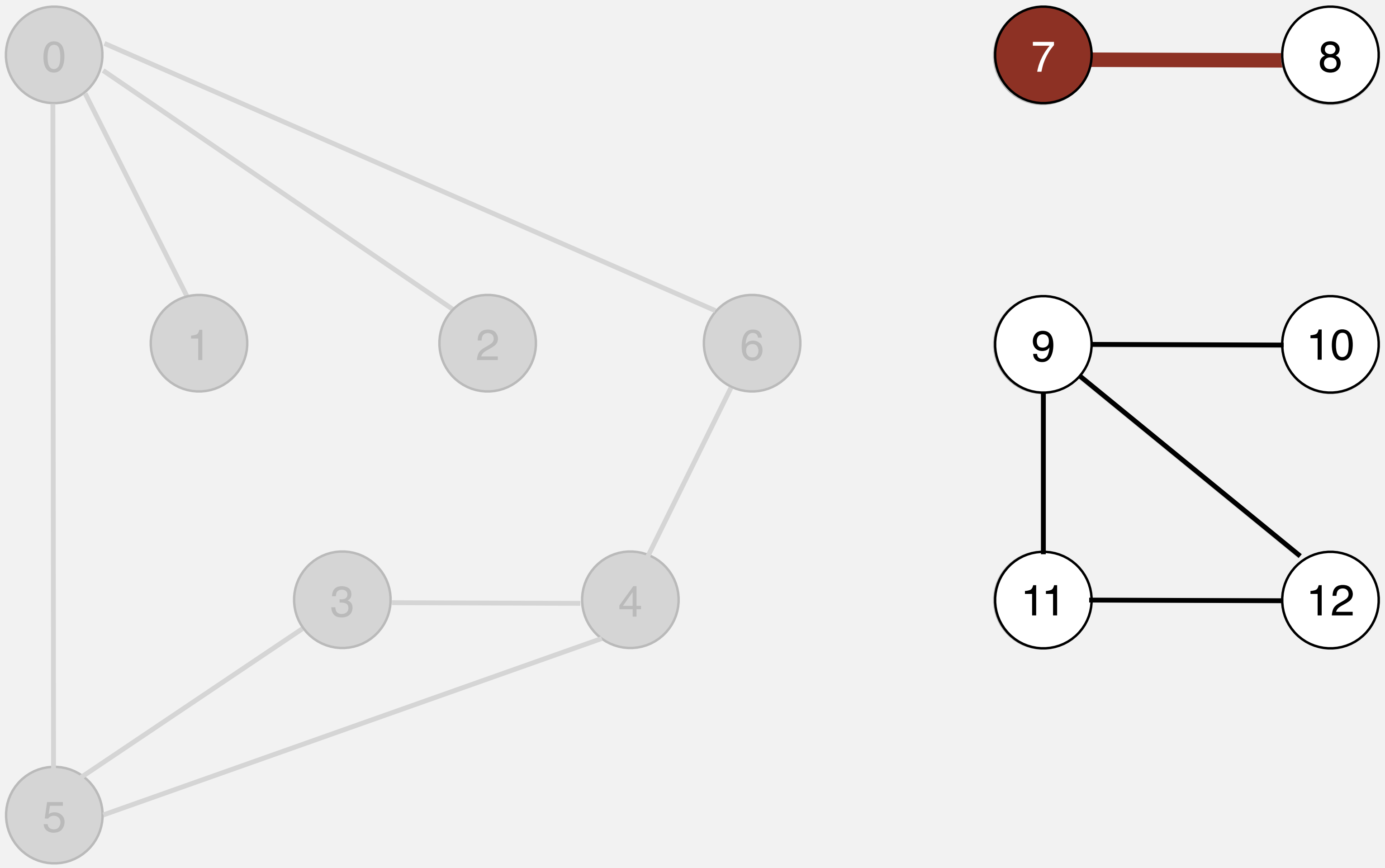
check 1 2 3 4 5 6

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

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



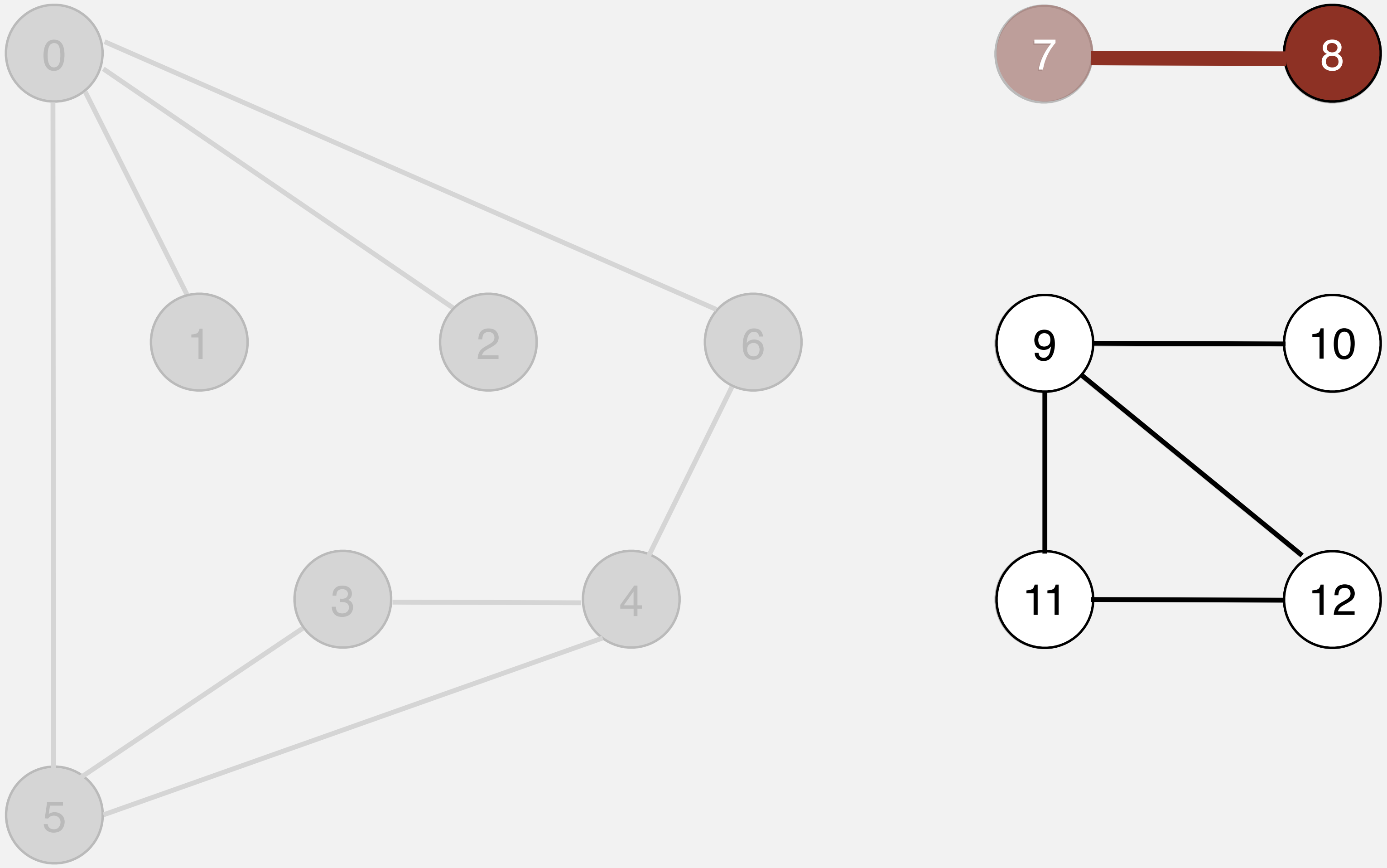
visit 7: check 8

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

Connected components

To visit a vertex *v* :

- Mark vertex *v* as visited.
- Recursively visit all unmarked vertices adjacent to *v*.



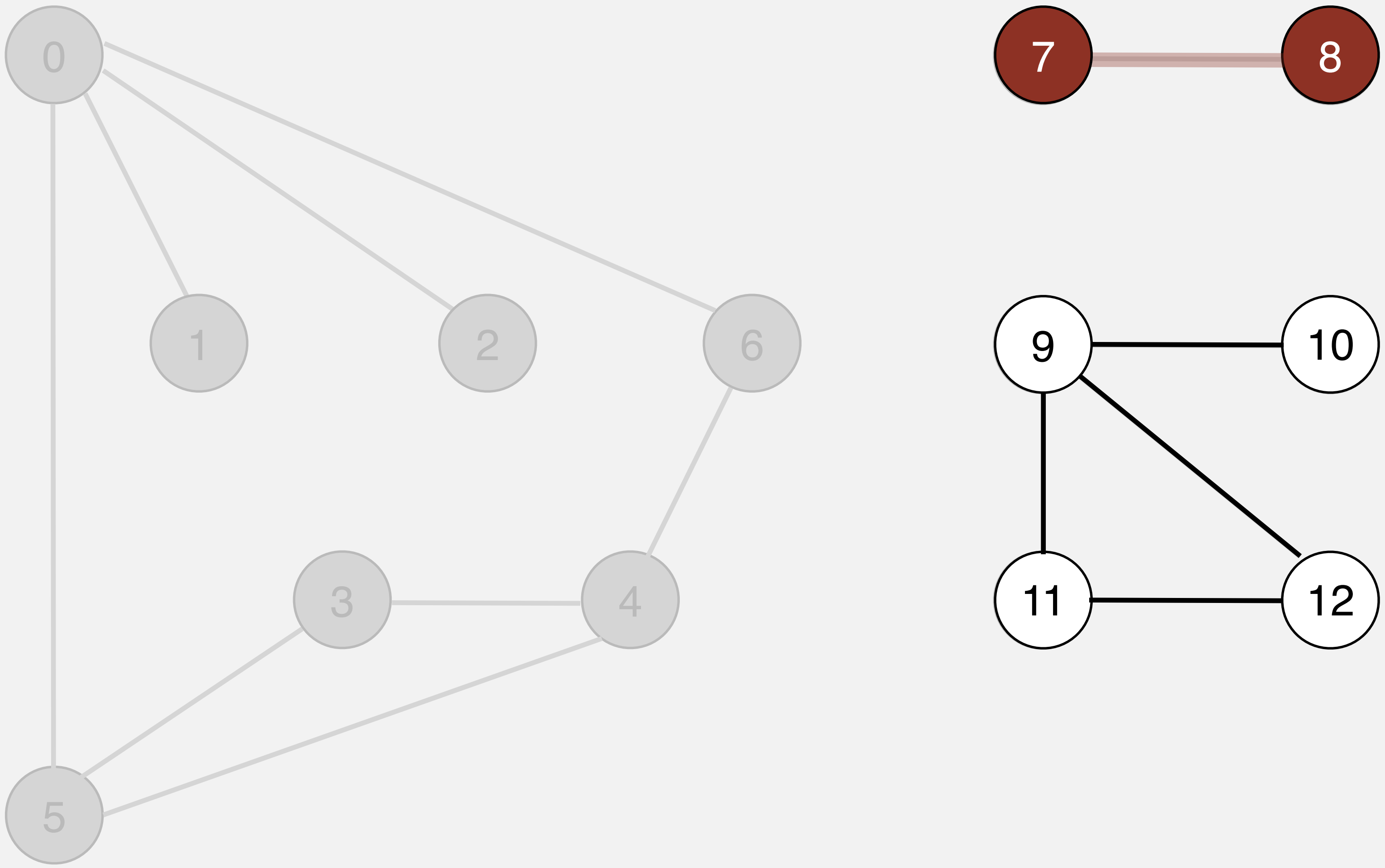
v	marked[]	cc[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	F	–
10	F	–
11	F	–
12	F	–

visit 8: check 7

Connected components

To visit a vertex *v* :

- Mark vertex *v* as visited.
- Recursively visit all unmarked vertices adjacent to *v*.



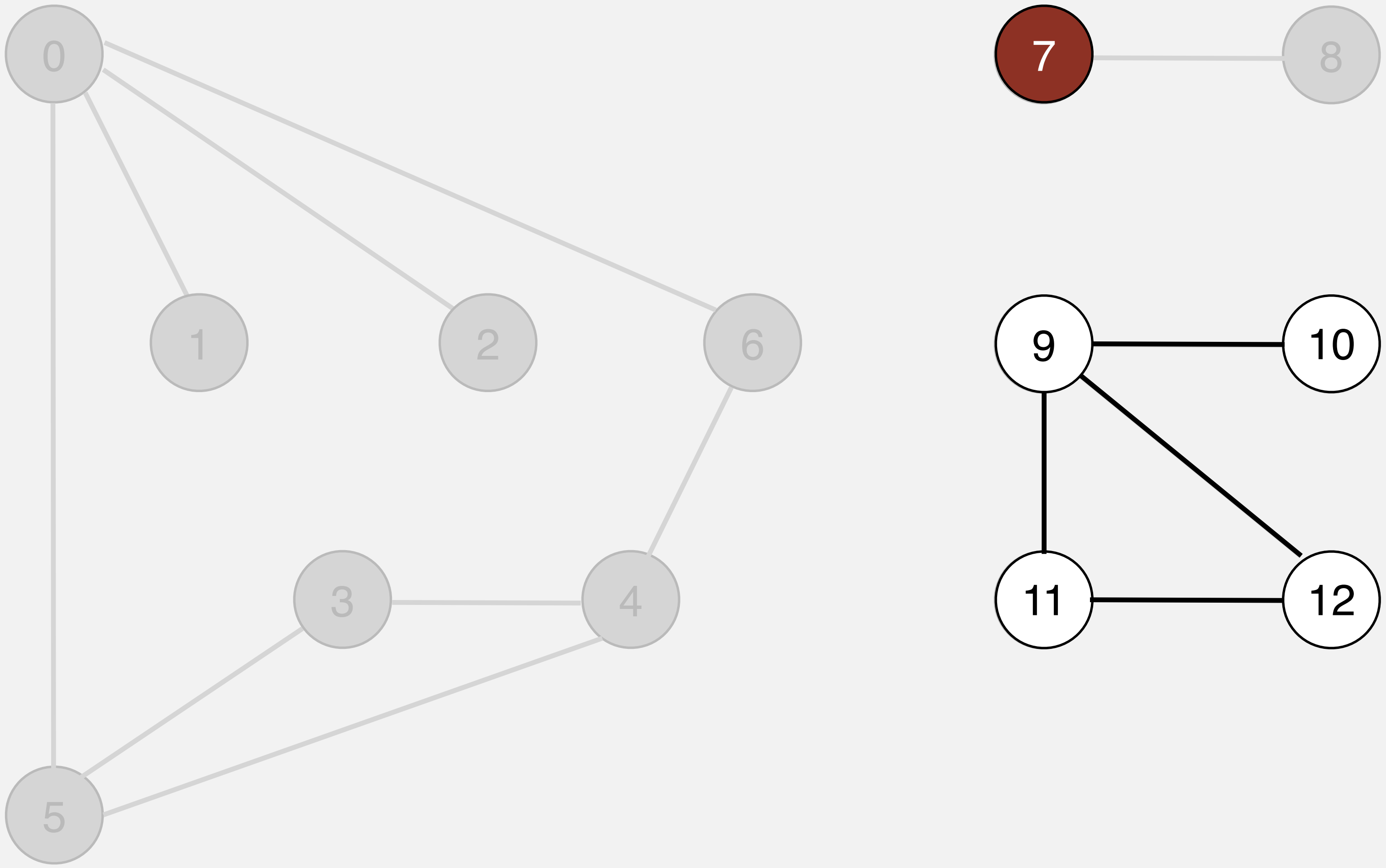
8 done

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

Connected components

To visit a vertex *v* :

- Mark vertex *v* as visited.
- Recursively visit all unmarked vertices adjacent to *v*.



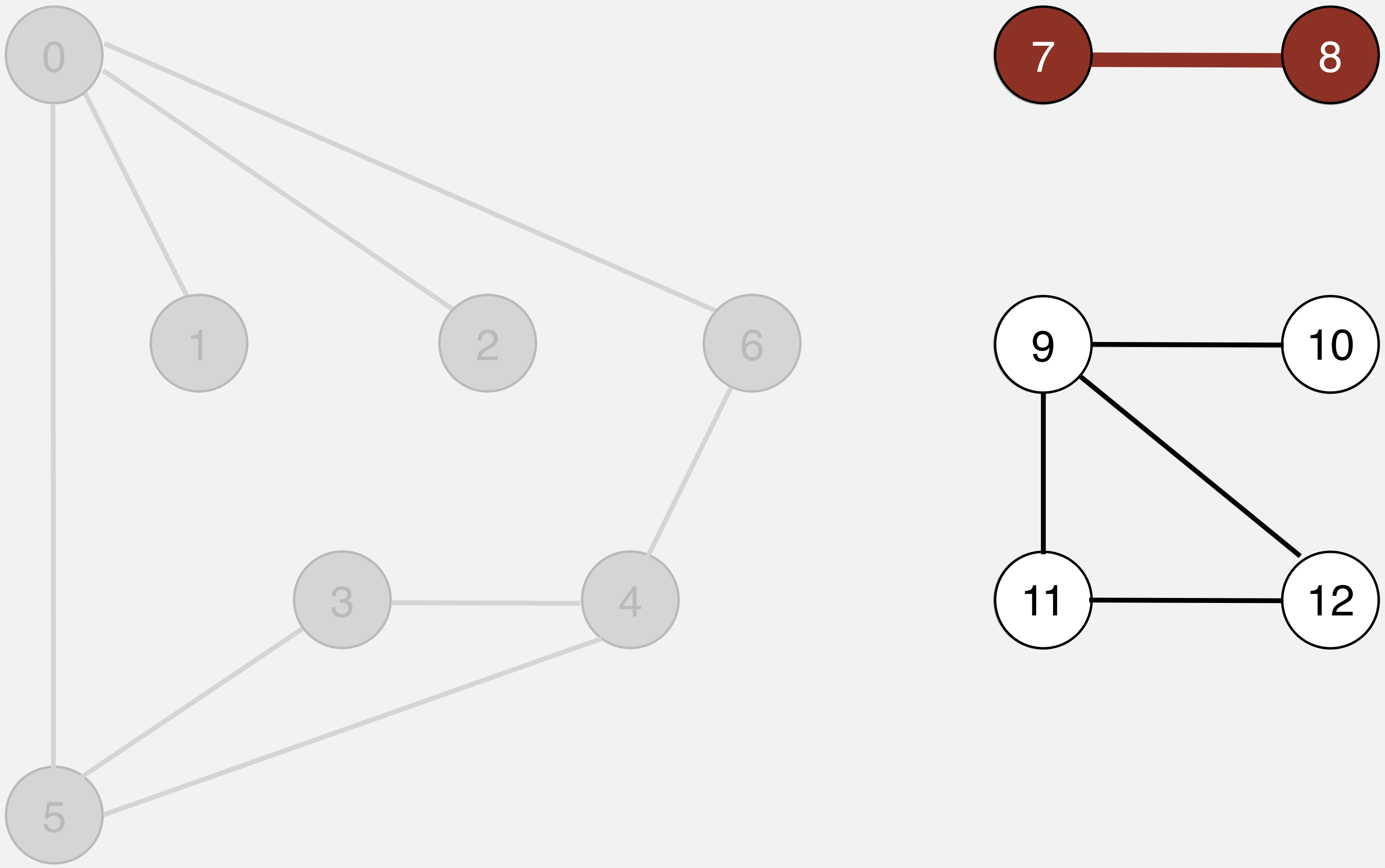
v	marked[]	cc[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	F	–
10	F	–
11	F	–
12	F	–

7 done

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



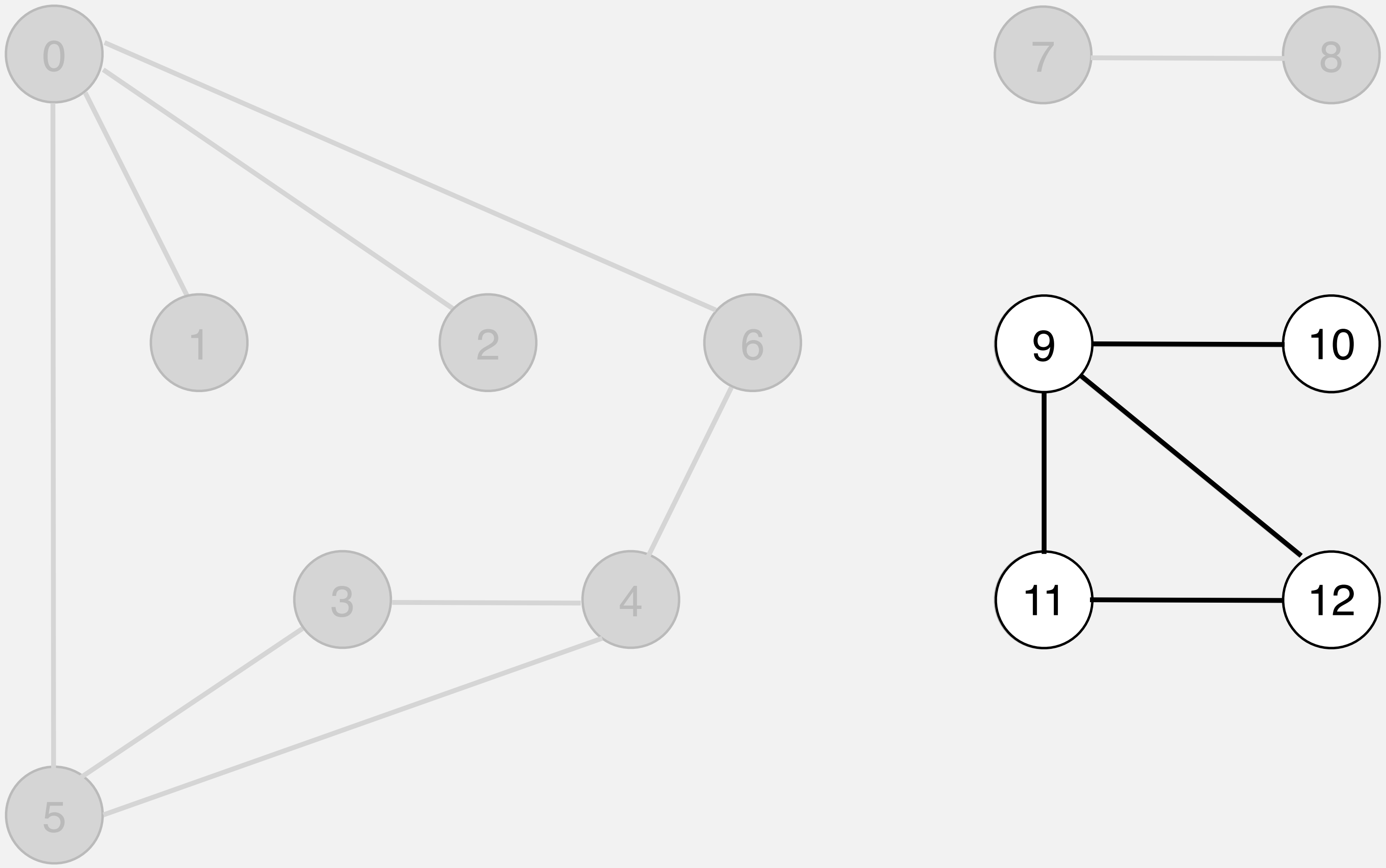
connected component: 7 8

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

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



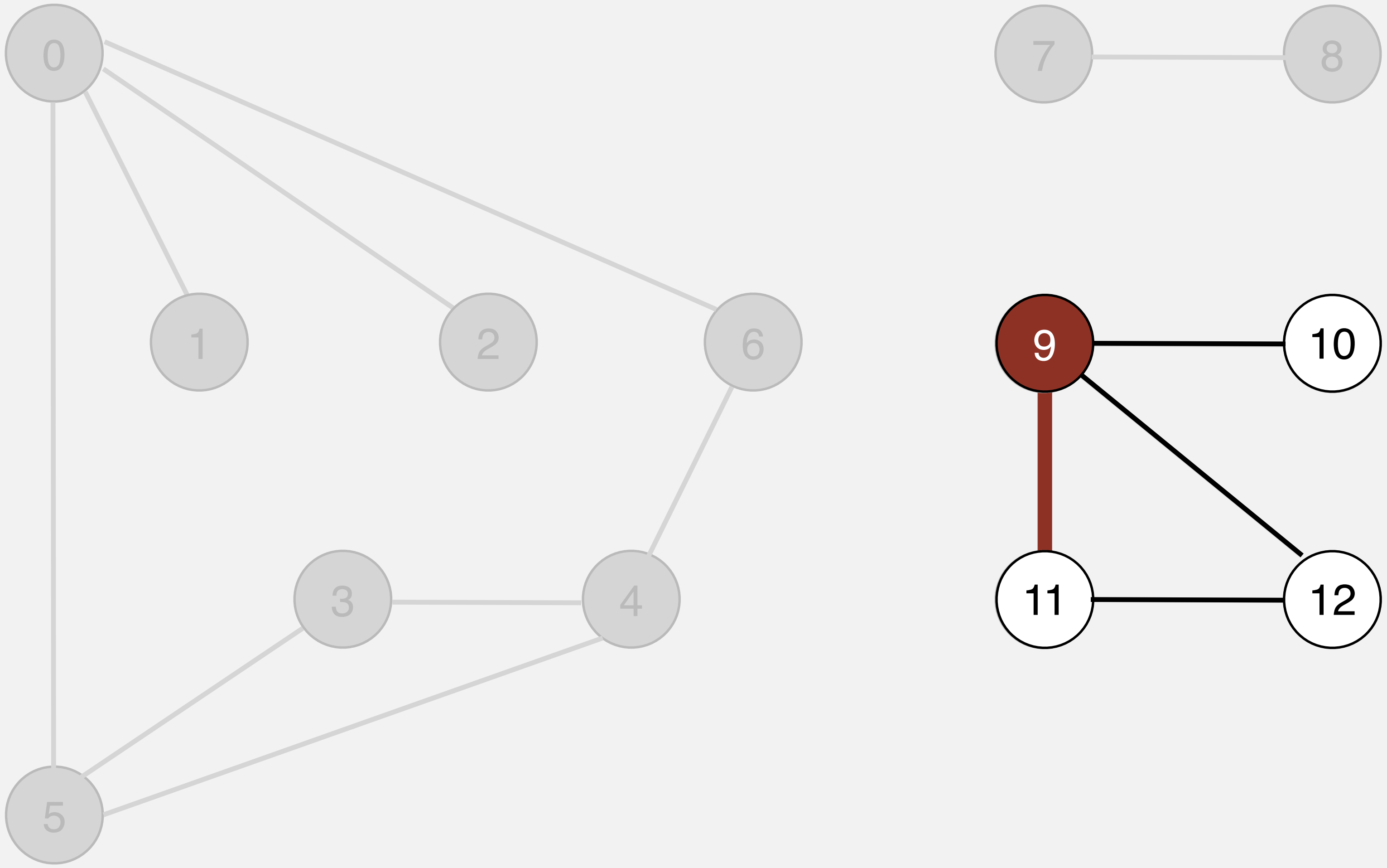
check 8

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

Connected components

To visit a vertex *v* :

- Mark vertex *v* as visited.
- Recursively visit all unmarked vertices adjacent to *v*.



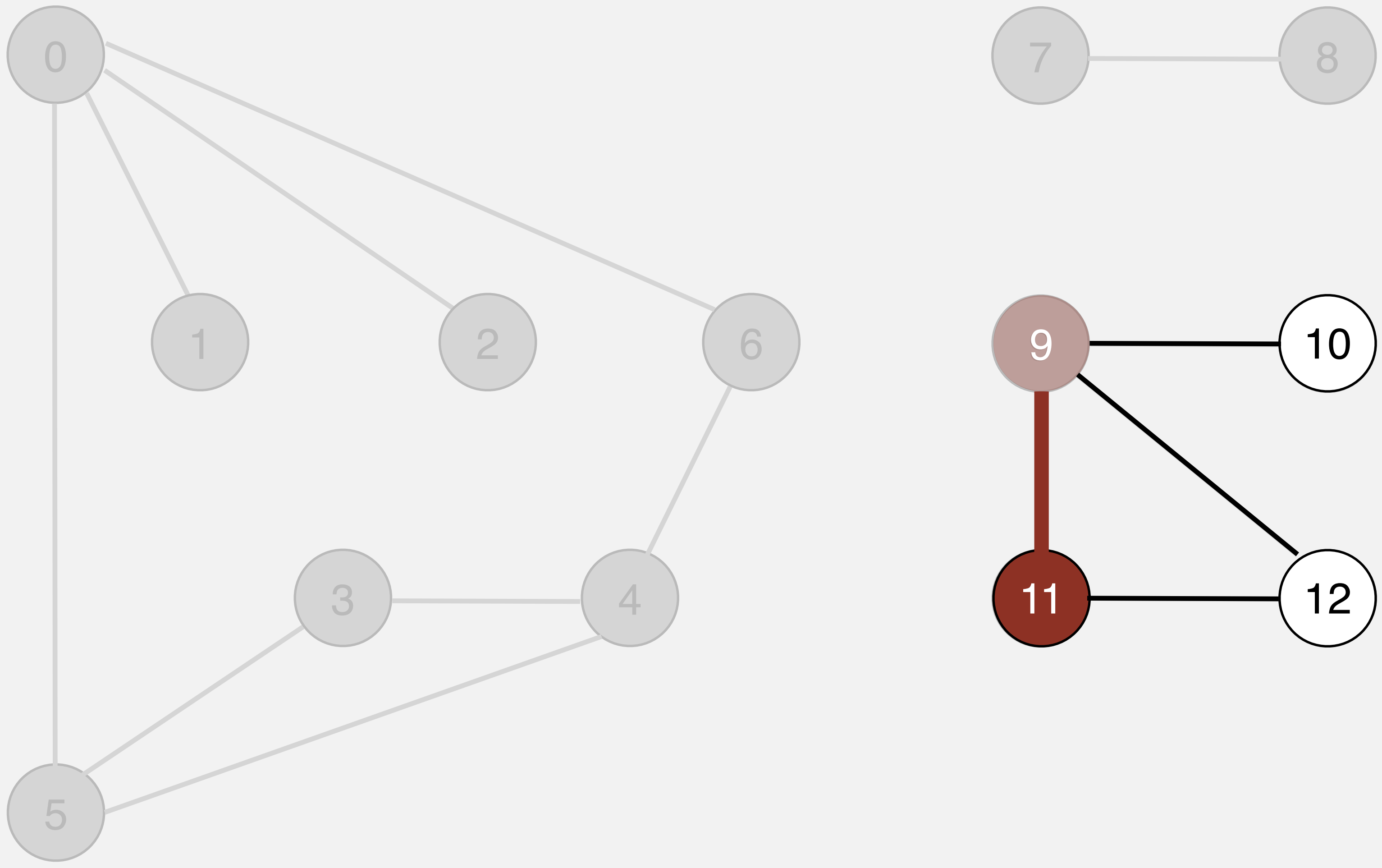
visit 9: check 11, check 10 and check 12

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

Connected components

To visit a vertex *v* :

- Mark vertex *v* as visited.
- Recursively visit all unmarked vertices adjacent to *v*.



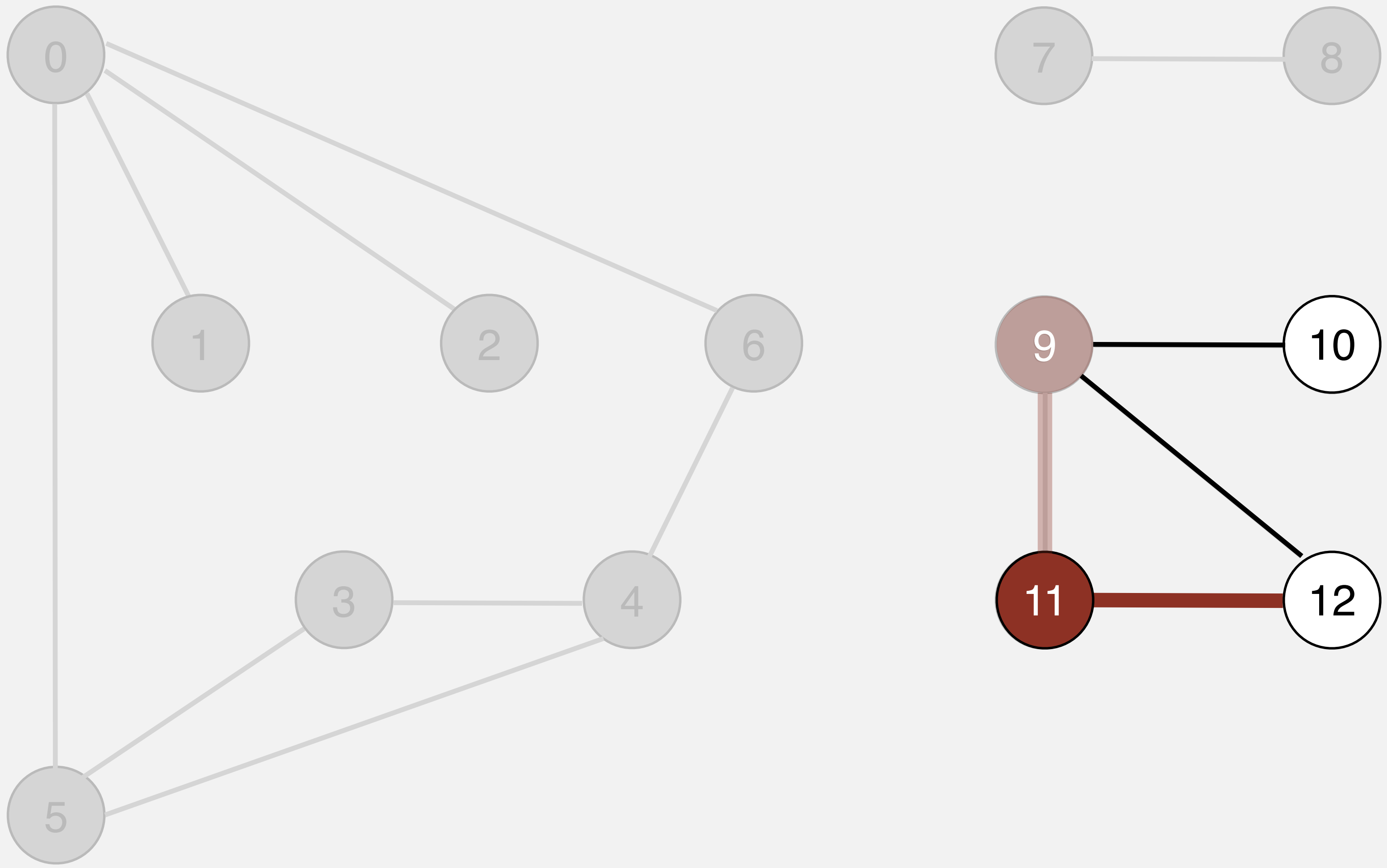
visit 11: check 9 and check 12

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

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



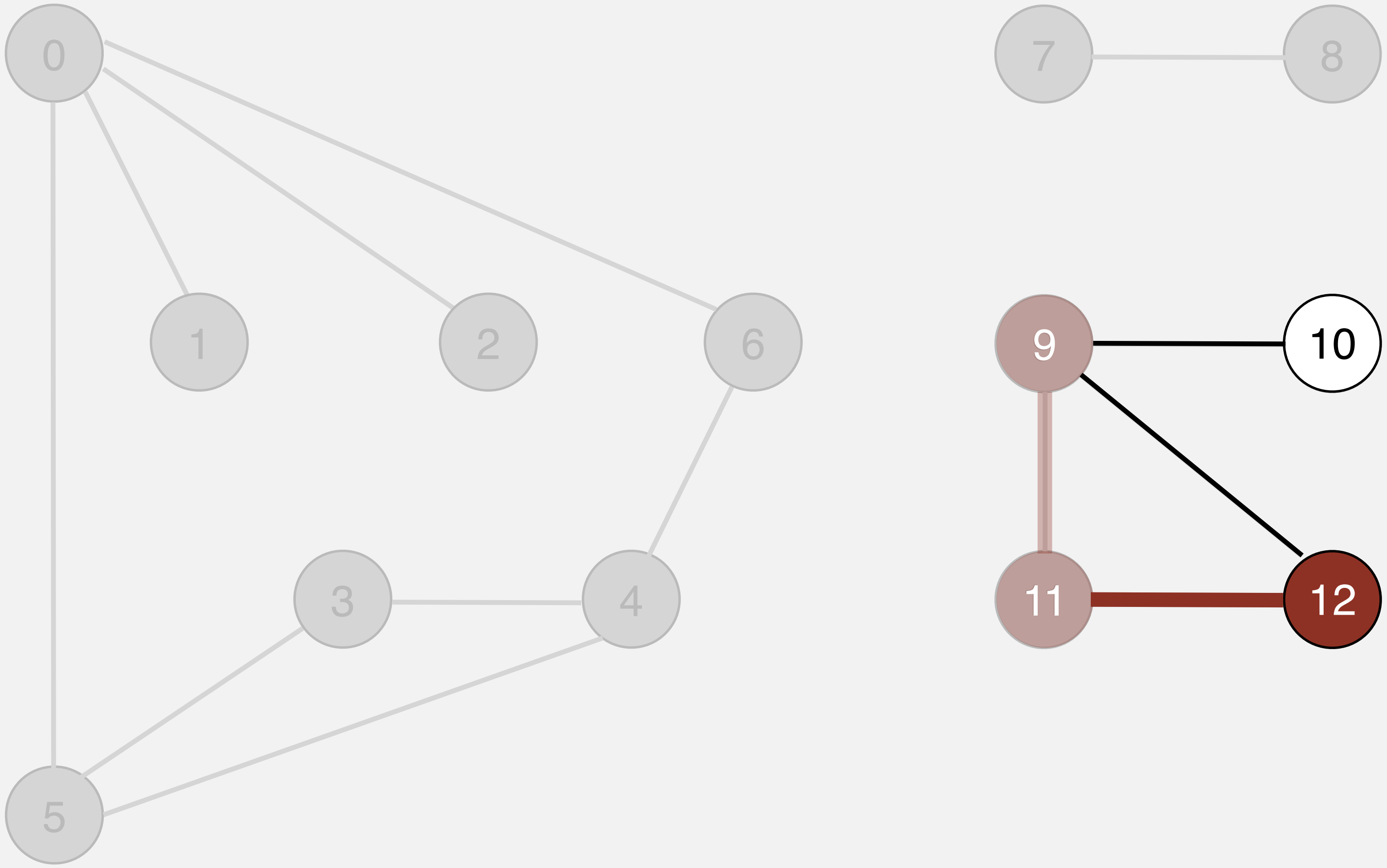
visit 11: check 9 and check 12

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

Connected components

To visit a vertex *v* :

- Mark vertex *v* as visited.
- Recursively visit all unmarked vertices adjacent to *v*.



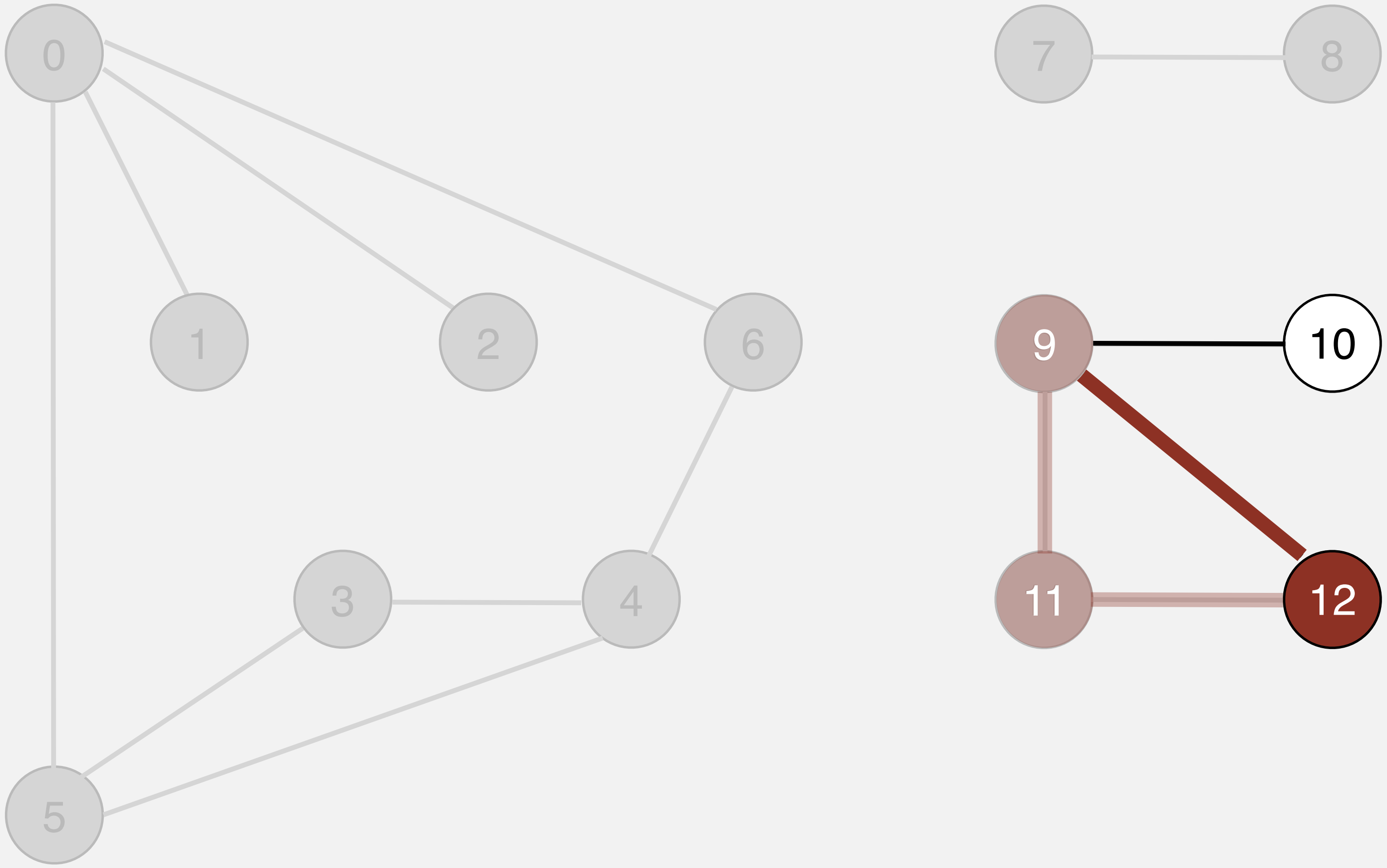
visit 12: check 11 and check 9

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

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



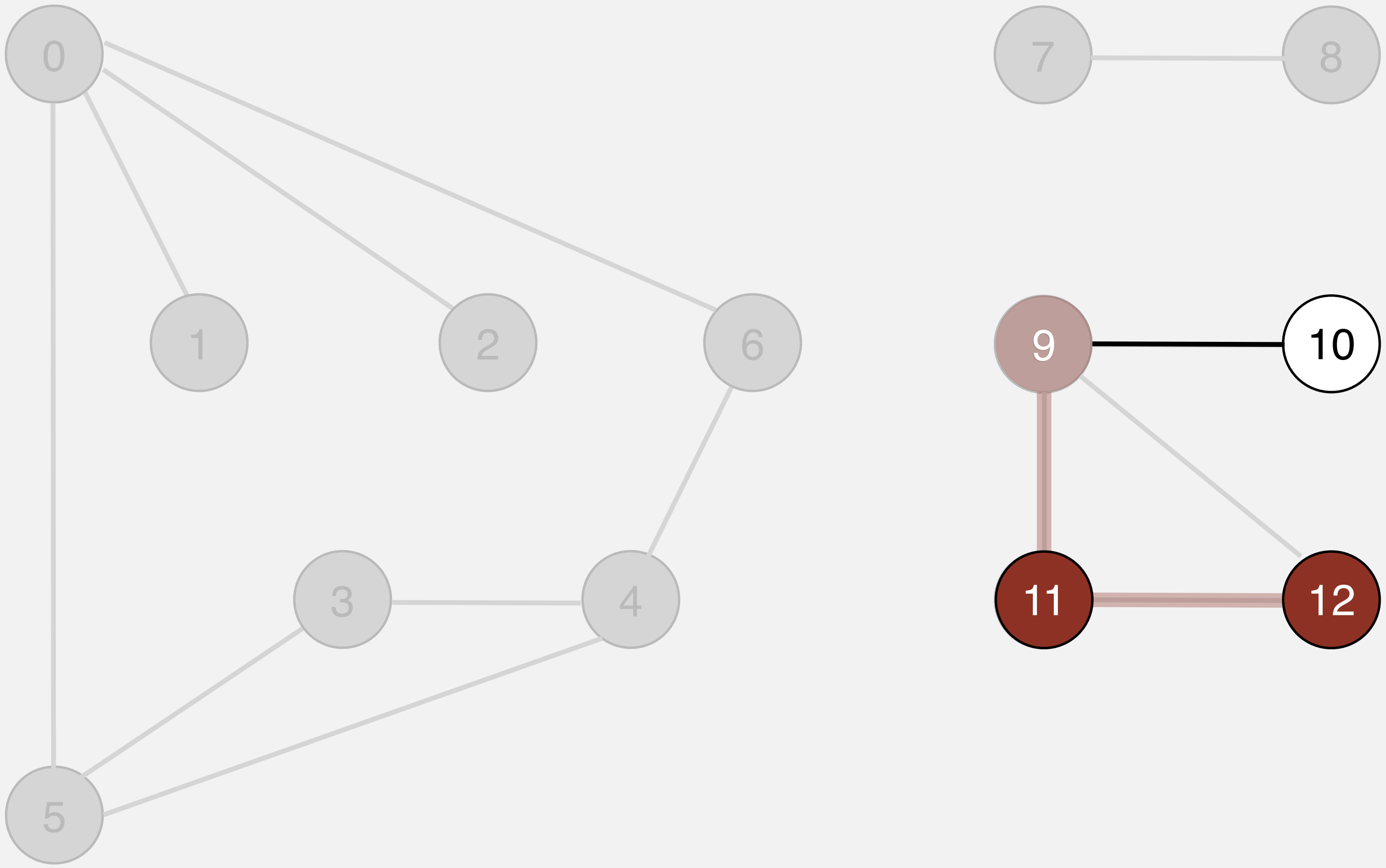
visit 12: check 11 and check 9

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

Connected components

To visit a vertex *v* :

- Mark vertex *v* as visited.
- Recursively visit all unmarked vertices adjacent to *v*.



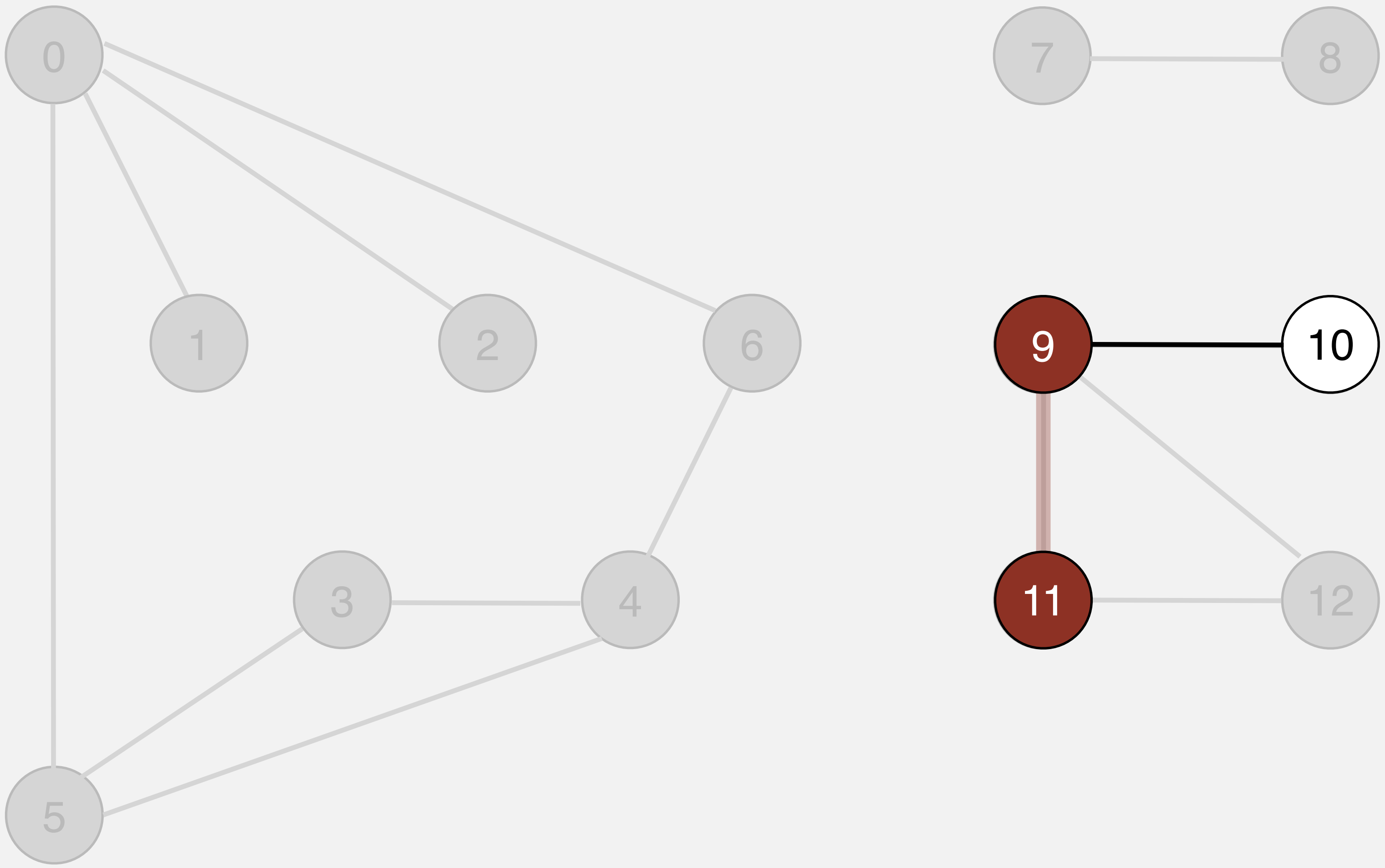
12 done

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

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



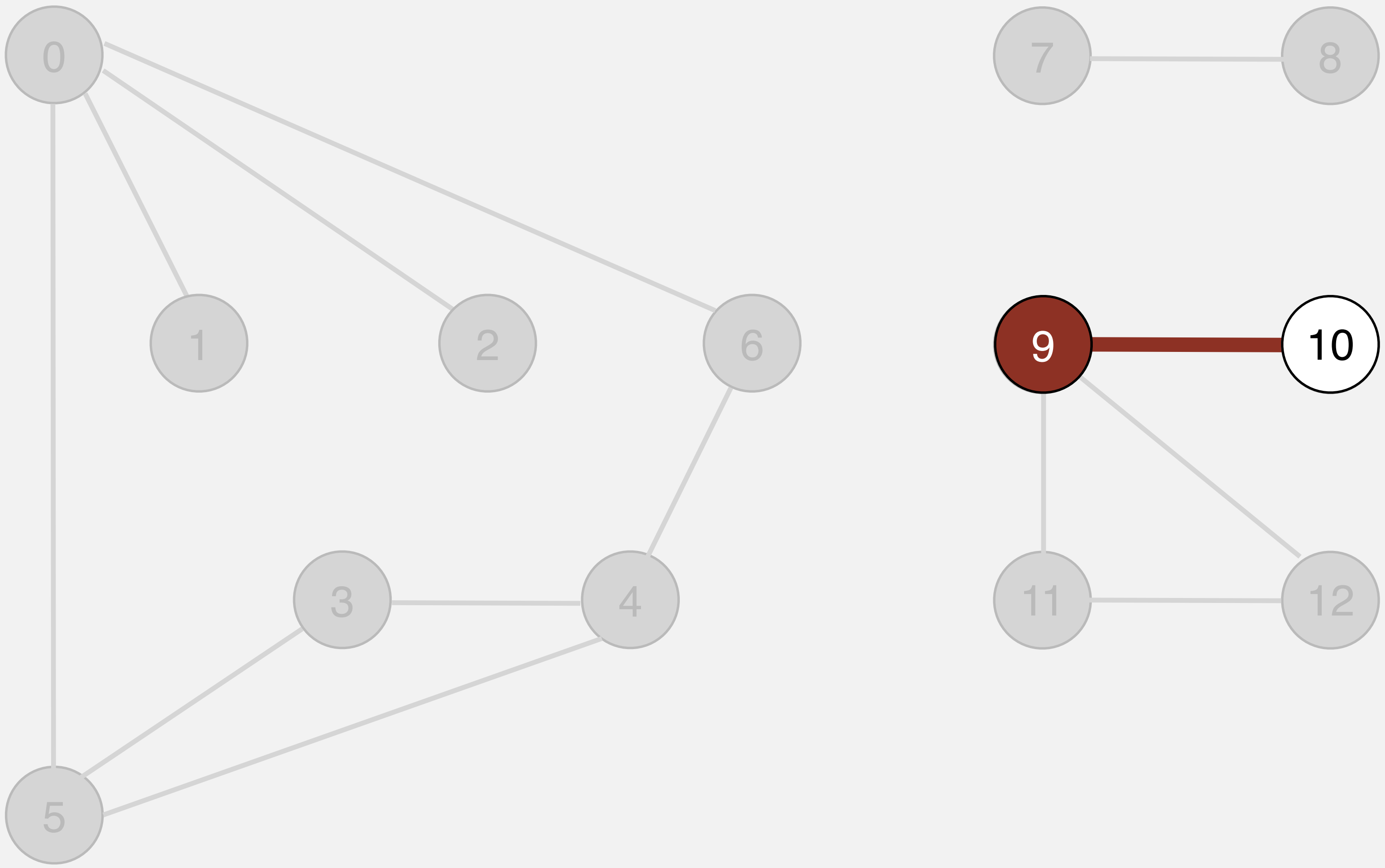
11 done

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

Connected components

To visit a vertex *v* :

- Mark vertex *v* as visited.
- Recursively visit all unmarked vertices adjacent to *v*.



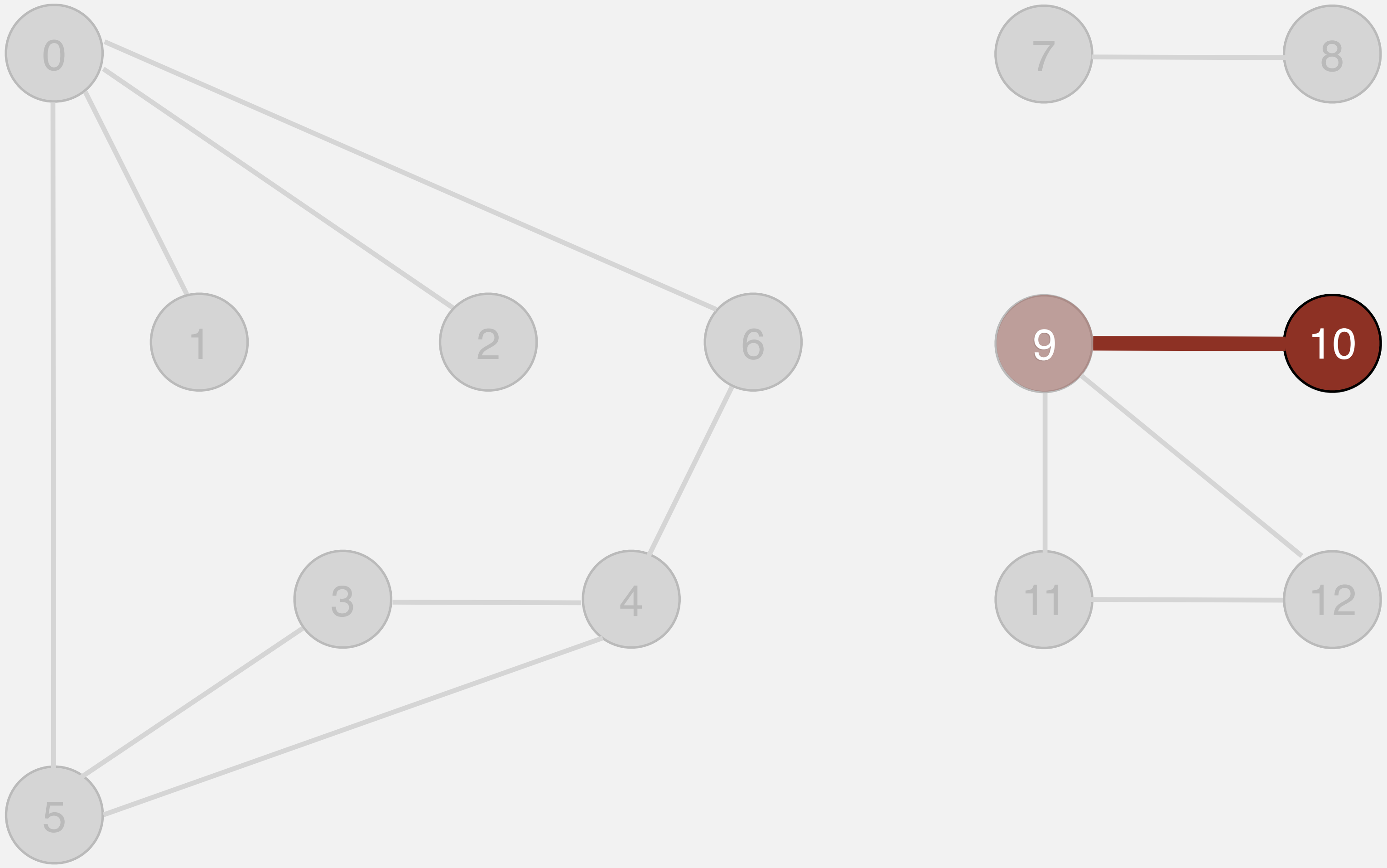
visit 9: check 11, **check 10** and check 12

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

Connected components

To visit a vertex *v* :

- Mark vertex *v* as visited.
- Recursively visit all unmarked vertices adjacent to *v*.



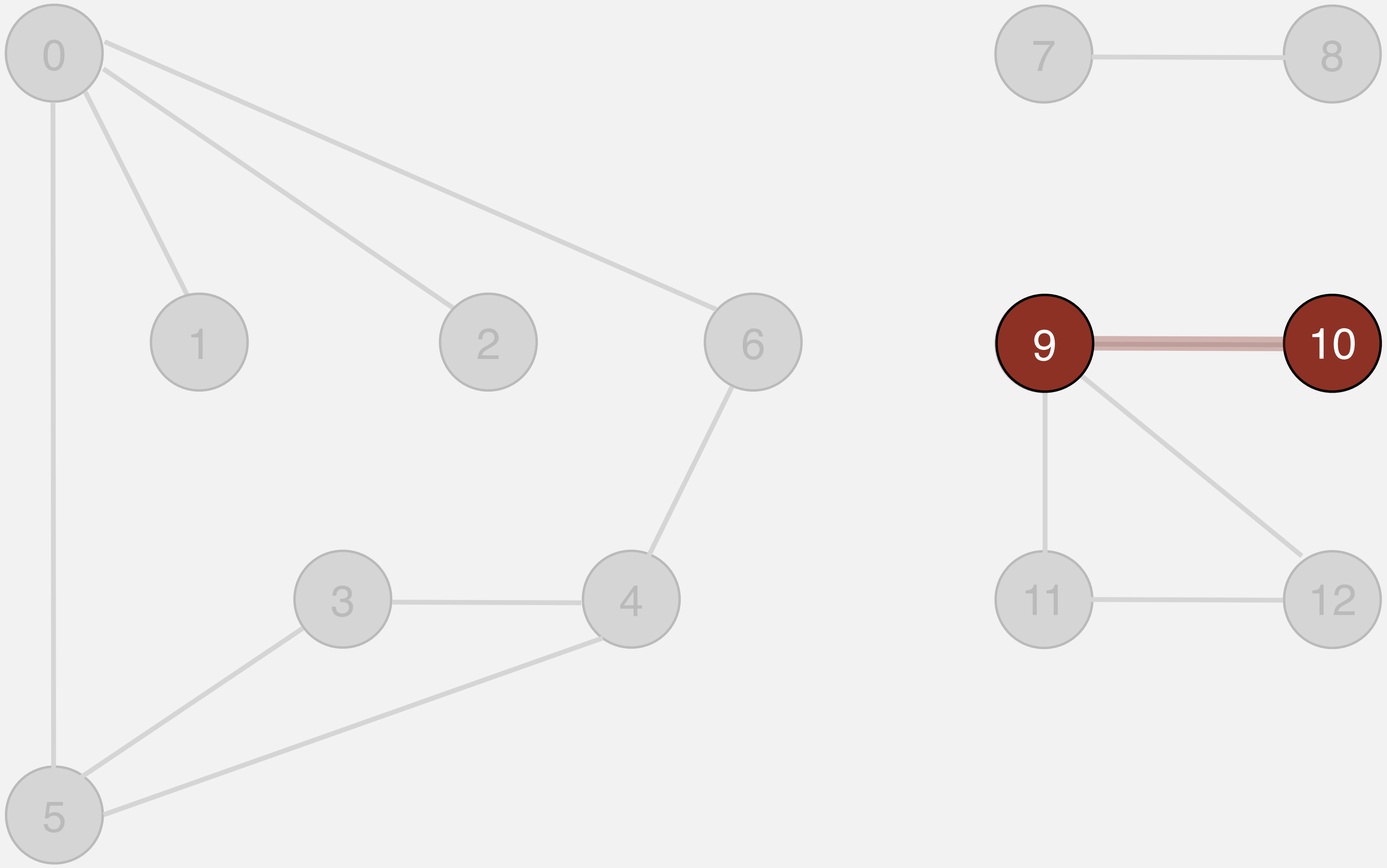
visit 10: check 9

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

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



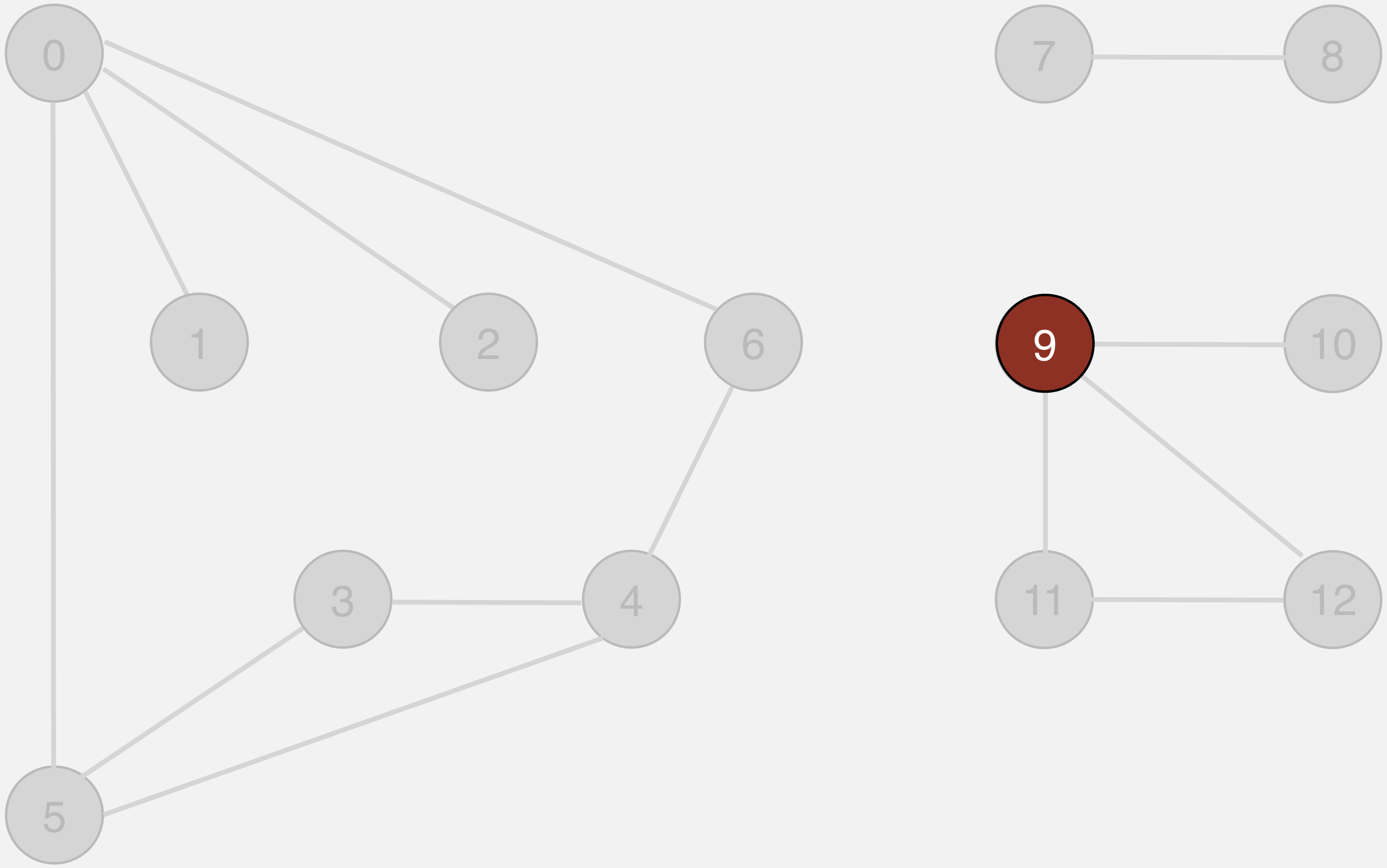
10 done

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

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



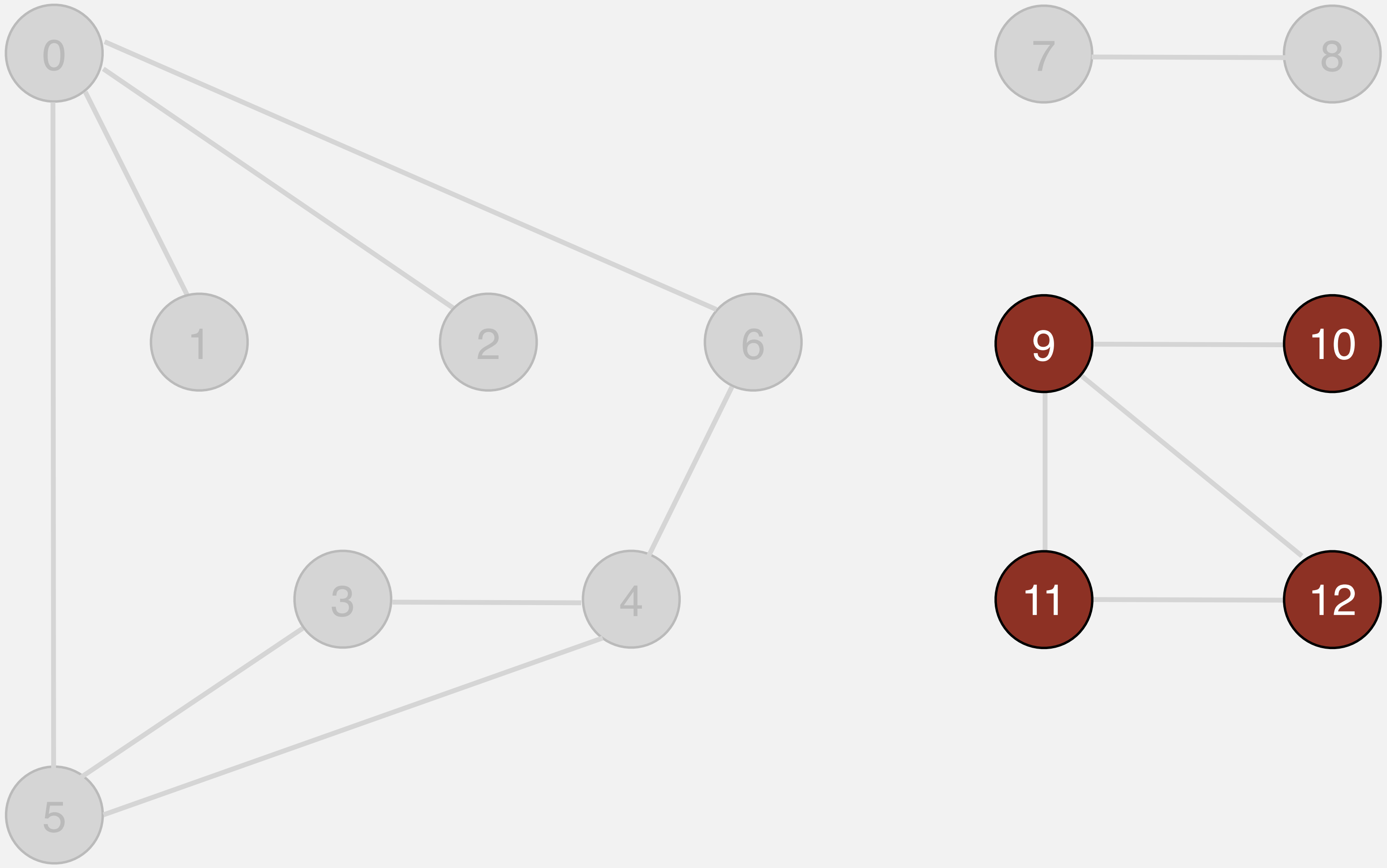
9 done

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

Connected components

To visit a vertex *v* :

- Mark vertex *v* as visited.
- Recursively visit all unmarked vertices adjacent to *v*.



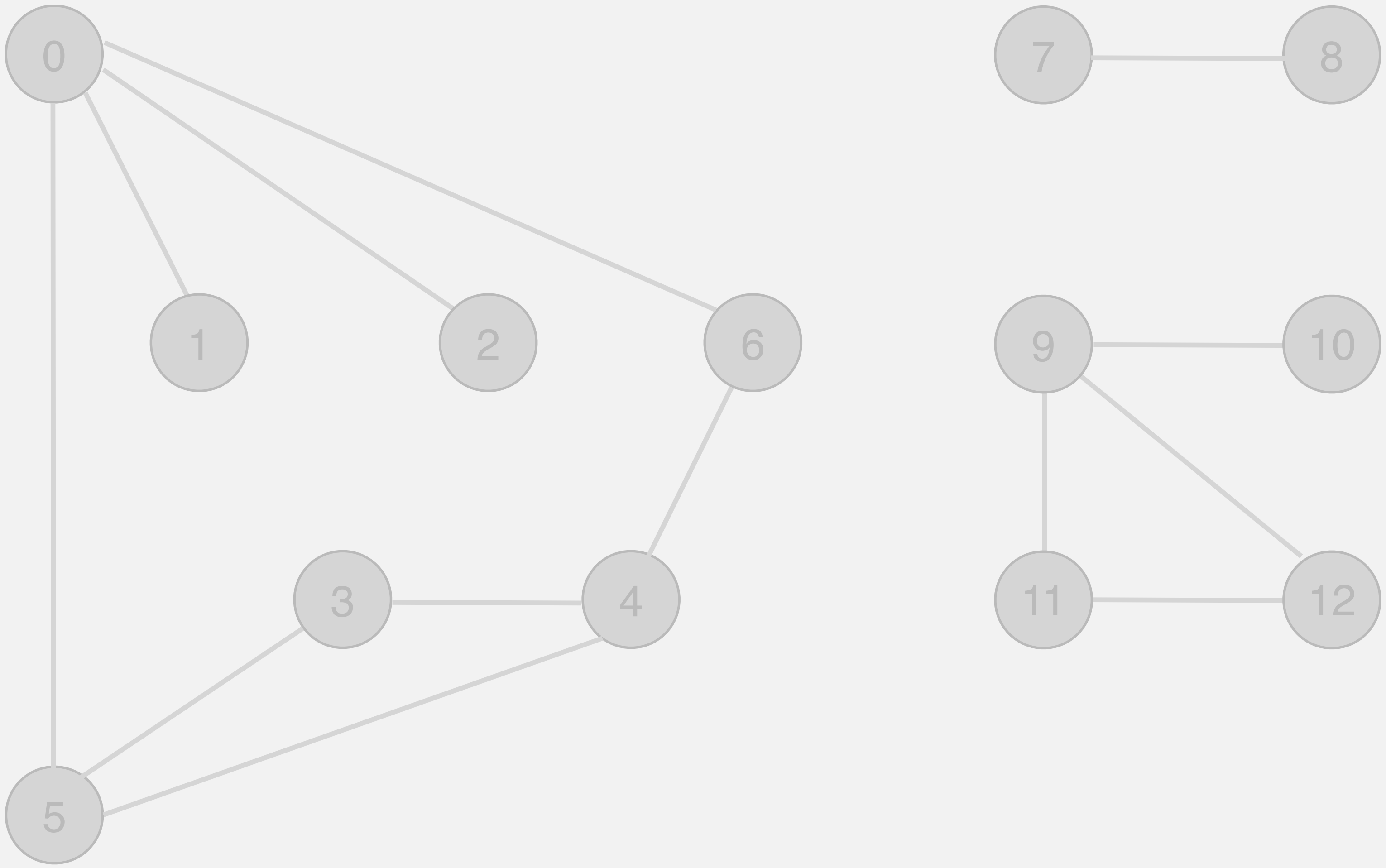
connected component: 9 10 11 12

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

Connected components

To visit a vertex *v* :

- Mark vertex *v* as visited.
- Recursively visit all unmarked vertices adjacent to *v*.



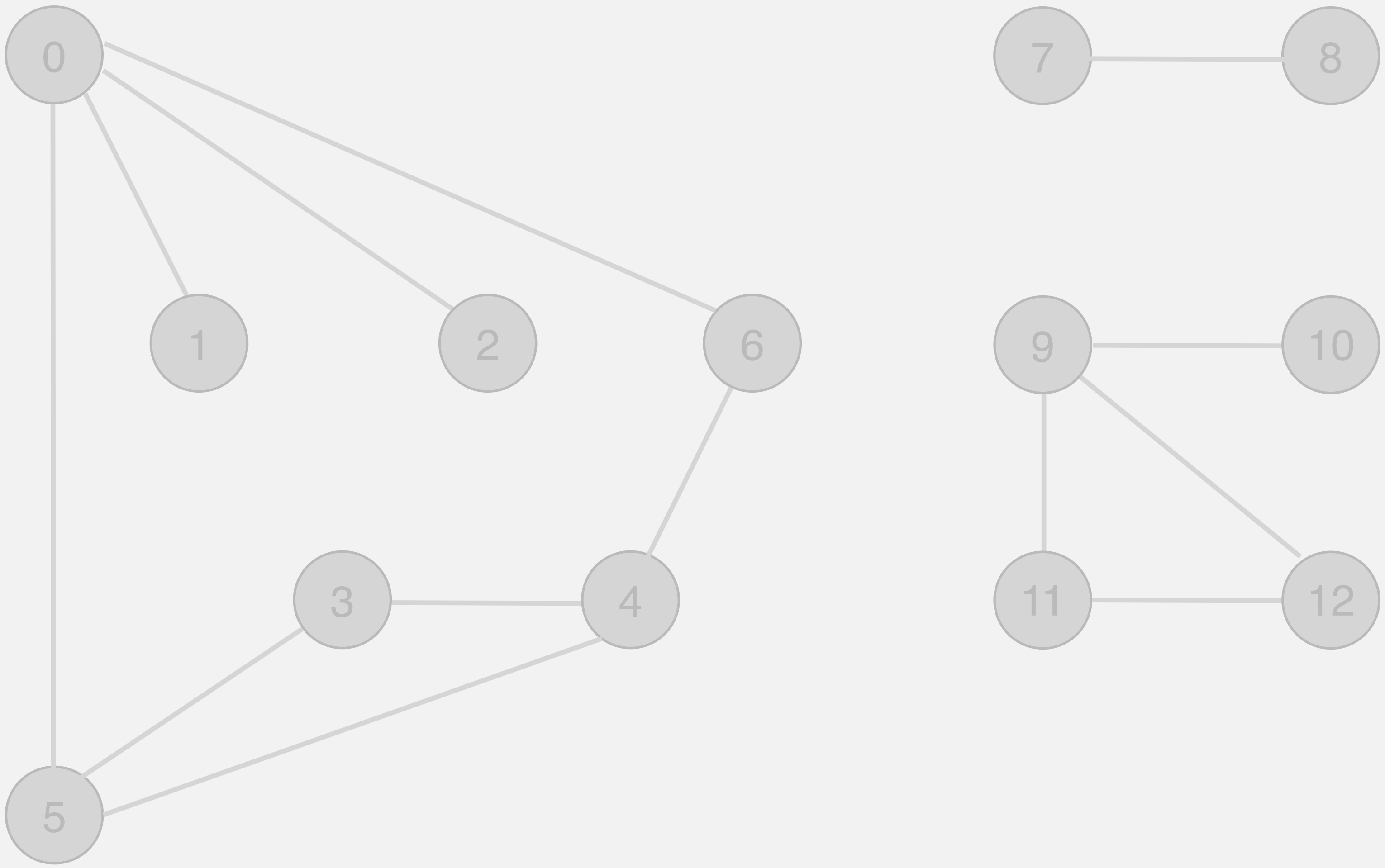
check 10 11 12

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

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



done

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

Finding connected components with DFS

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

   public CC(Graph G)
   {
      marked = new boolean[G.V()];
      id = new int[G.V()];
      for (int v = 0; v < G.V(); v++)
      {
         if (!marked[v])
         {
            dfs(G, v);
            count++;
         }
      }
   }

   public int count()
   public int id(int v)
   private void dfs(Graph G, int v)

}
```

id[v] = id of component containing v

number of components

run DFS from one vertex in each component

see next slide

Finding connected components with DFS (continued)

```
public int count()
{  return count;  }

public int id(int v)
{  return id[v];  }

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

number of components

id of component containing v

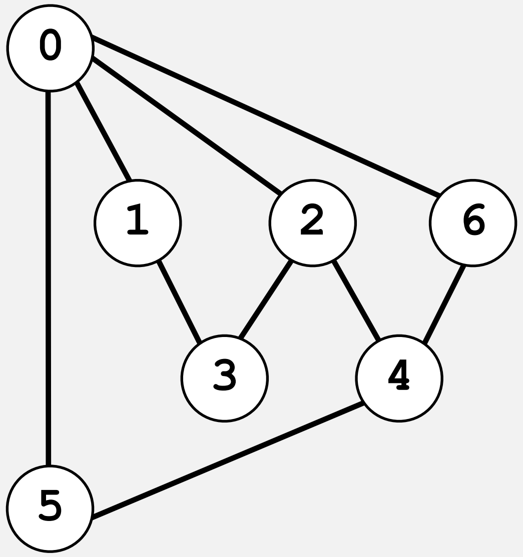
all vertices discovered in same call of dfs have same id

Undirected Graphs

- Graph API
- Depth-first search
- Breadth-first search
- Connected components
- Challenges

Graph-processing challenge 1

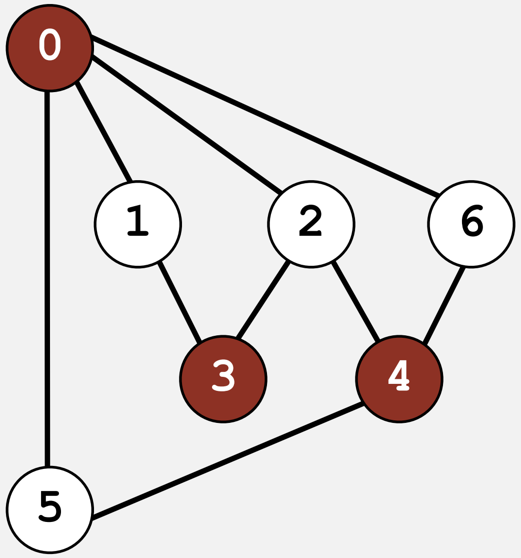
Problem. Is a graph bipartite?



0-1
0-2
0-5
0-6
1-3
2-3
2-4
4-5
4-6

How difficult?

- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

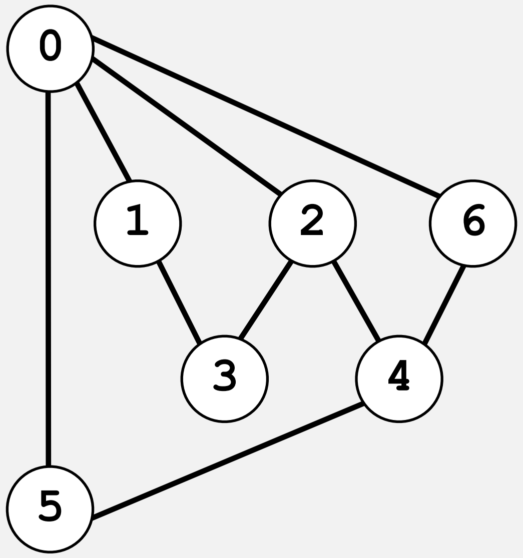


0-1
0-2
0-5
0-6
1-3
2-3
2-4
4-5
4-6

{ 0, 3, 4 }

Graph-processing challenge 1

Problem. Is a graph bipartite?

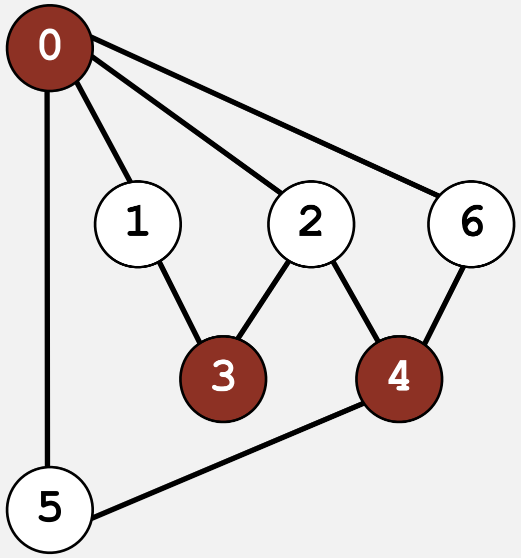


0-1
0-2
0-5
0-6
1-3
2-3
2-4
4-5
4-6

How difficult?

- Any programmer could do it.
- ✓ Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

simple DFS-based solution (see textbook)



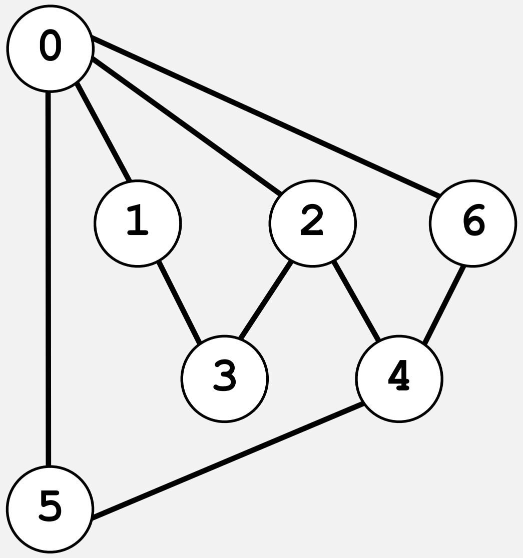
0-1
0-2
0-5
0-6
1-3
2-3
2-4
4-5
4-6

Graph-processing challenge 2

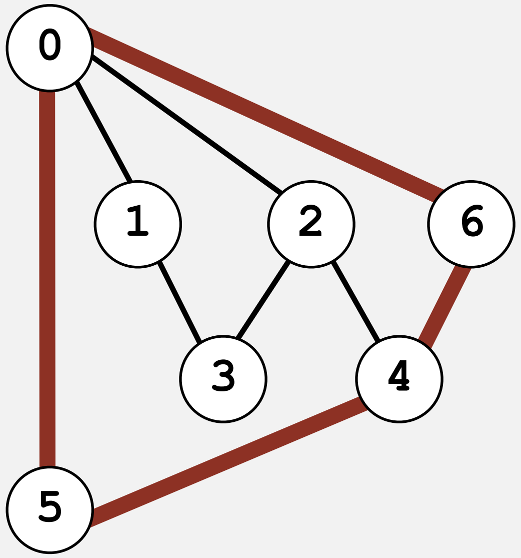
Problem. Find a cycle.

How difficult?

- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.



0-1
0-2
0-5
0-6
1-3
2-3
2-4
4-5
4-6



0-5-4-6-0

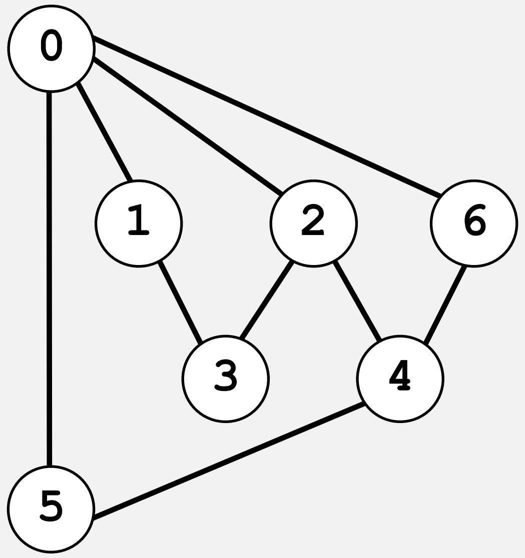
Graph-processing challenge 2

Problem. Find a cycle.

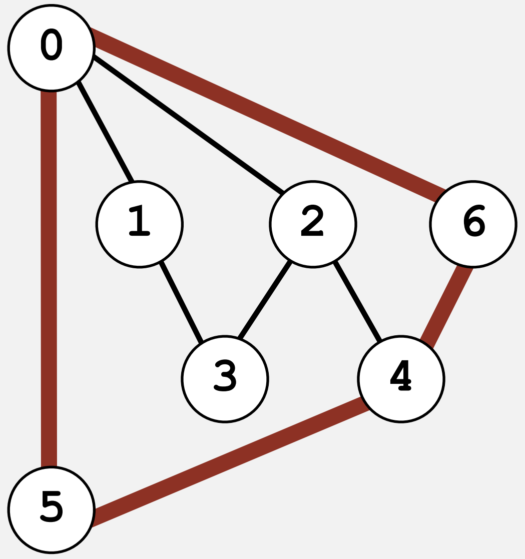
How difficult?

- Any programmer could do it.
- ✓ Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

simple DFS-based solution (see textbook)



```
0-1
0-2
0-5
0-6
1-3
2-3
2-4
4-5
4-6
```

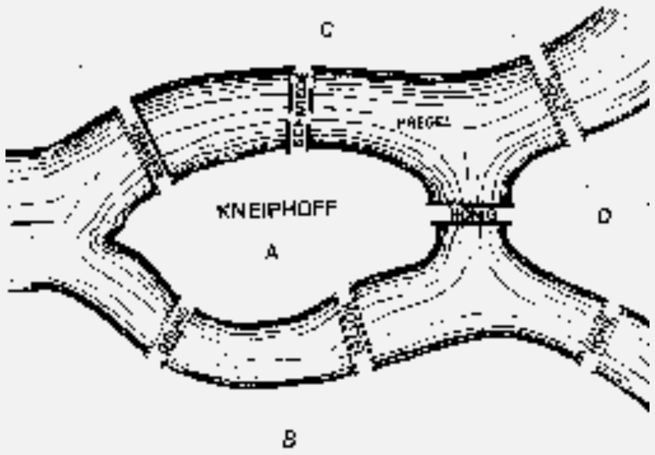


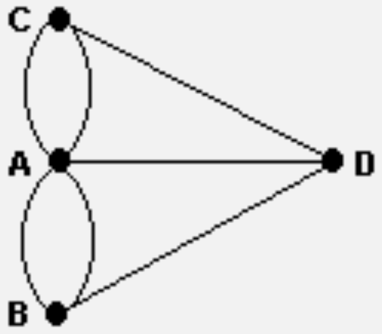
0-5-4-6-0

Bridges of Königsberg

The Seven Bridges of Königsberg. [Leonhard Euler 1736]

> *“ … in Königsberg in Prussia, there is an island A, called the Kneiphof; the river which surrounds it is divided into two branches … and these branches are crossed by seven bridges. Concerning these bridges, it was asked whether anyone could arrange a route in such a way that he could cross each bridge once and only once. ”*





Euler tour. Is there a (general) cycle that uses each edge exactly once?

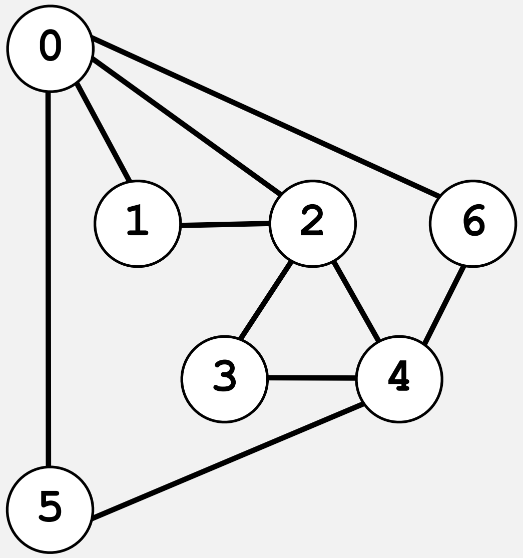
Answer. Yes iff connected and all vertices have even degree.

To find path. DFS-based algorithm (see textbook).

Graph-processing challenge 3

Problem. Find a cycle that uses every edge.

Assumption. Need to use each edge exactly once.



0-1
0-2
0-5
0-6
1-2
2-3
2-4
3-4
4-5
4-6

0-1-2-3-4-2-0-6-4-5-0

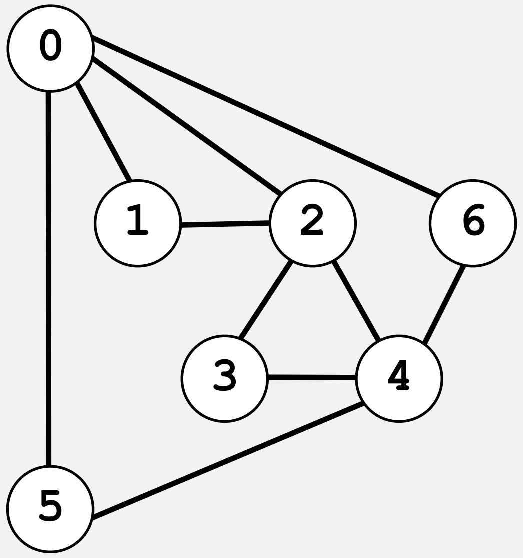
How difficult?

- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

Graph-processing challenge 3

Problem. Find a cycle that uses every edge.

Assumption. Need to use each edge exactly once.



0-1
0-2
0-5
0-6
1-2
2-3
2-4
3-4
4-5
4-6

0-1-2-3-4-2-0-6-4-5-0

How difficult?

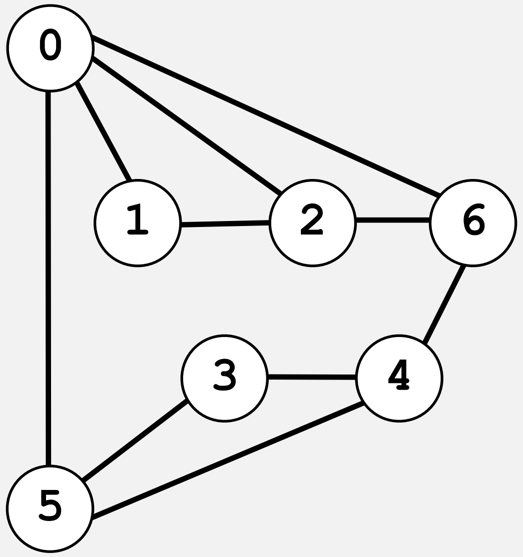
- Any programmer could do it.
- ✓ Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

Eulerian tour

(classic graph-processing problem)

Graph-processing challenge 4

Problem. Find a cycle that visits every vertex exactly once.



0-5-3-4-6-2-1-0

```
0-1
0-2
0-5
0-6
1-2
2-6
3-4
3-5
4-5
4-6
```

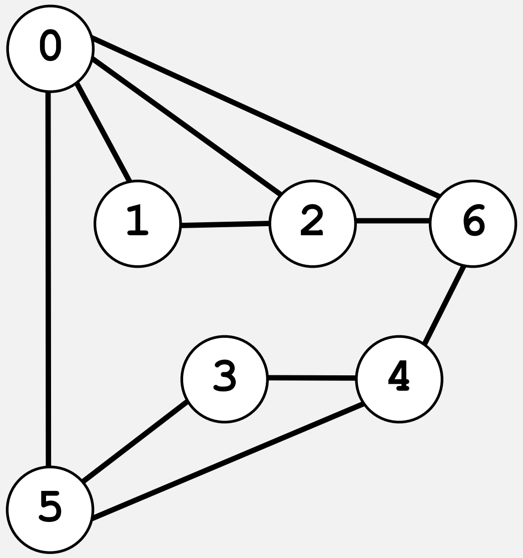
How difficult?

- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

Graph-processing challenge 4

Problem. Find a cycle that visits every vertex.

Assumption. Need to visit each vertex exactly once.



0-5-3-4-6-2-1-0

0-1
0-2
0-5
0-6
1-2
2-6
3-4
3-5
4-5
4-6

How difficult?

- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- ✓ Intractable.
- No one knows.
- Impossible.

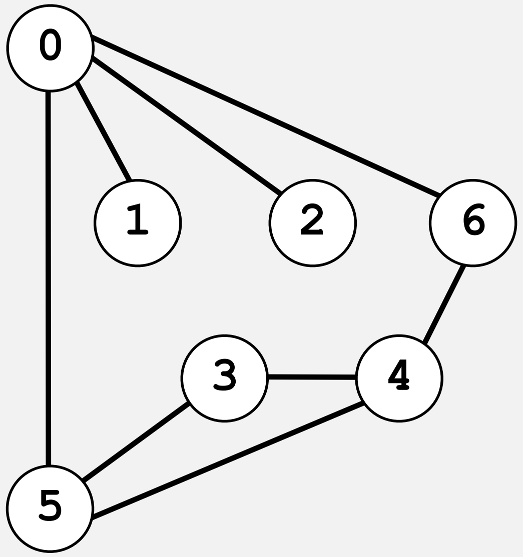
Hamiltonian cycle
(classical NP-complete problem)

Graph-processing challenge 5

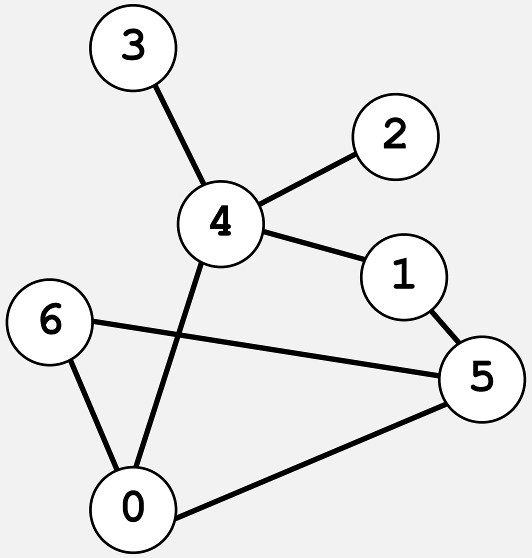
Problem. Are two graphs identical except for vertex names?

How difficult?

- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.



0-1
0-2
0-5
0-6
3-4
3-5
4-5
4-6



0-4
0-5
0-6
1-4
1-5
2-4
3-4
5-6

0↔4, 1↔3, 2↔2, 3↔6, 4↔5, 5↔0, 6↔1

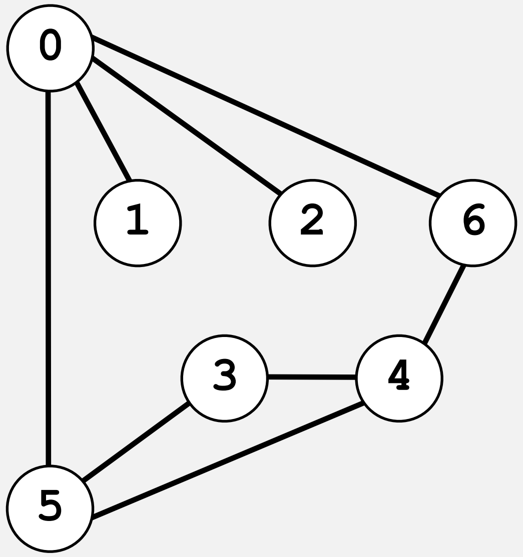
Graph-processing challenge 5

Problem. Are two graphs identical except for vertex names?

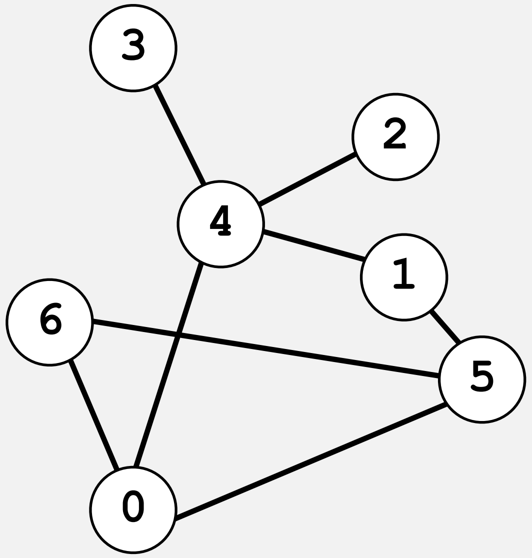
How difficult?

- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- ✓ No one knows.
- Impossible.

graph isomorphism is longstanding open problem



0-1
0-2
0-5
0-6
3-4
3-5
4-5
4-6

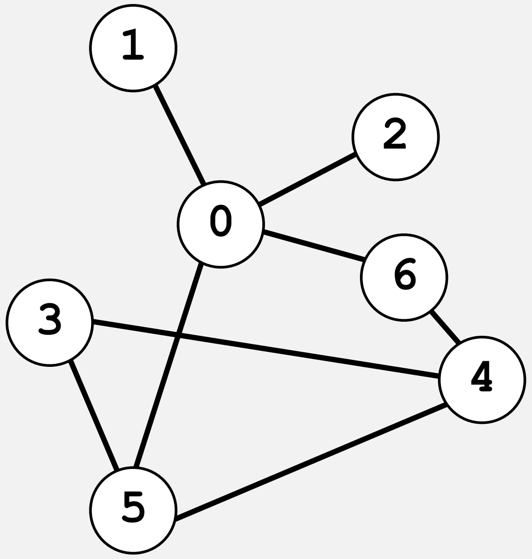


0-4
0-5
0-6
1-4
1-5
2-4
3-4
5-6

0↔4, 1↔3, 2↔2, 3↔6, 4↔5, 5↔0, 6↔1

Graph-processing challenge 6

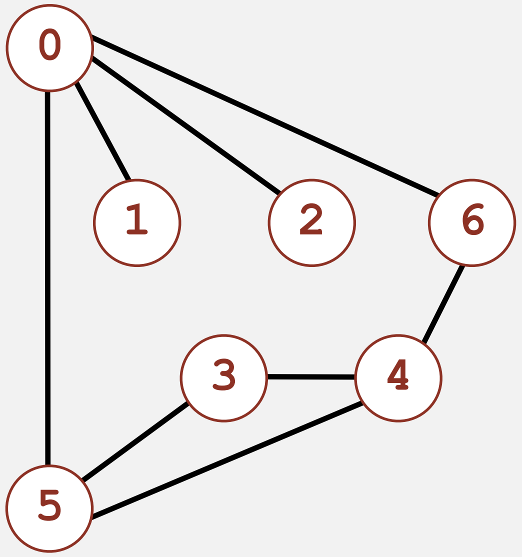
Problem. Lay out a graph in the plane without crossing edges?



```
0-1
0-2
0-5
0-6
3-4
3-5
4-5
4-6
```

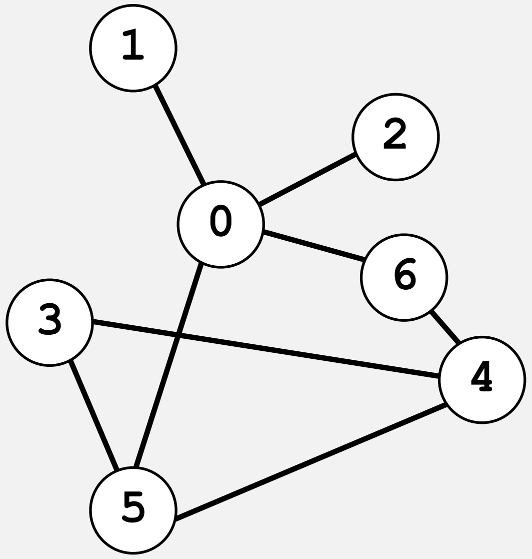
How difficult?

- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.



Graph-processing challenge 6

Problem. Lay out a graph in the plane without crossing edges?



0-1
0-2
0-5
0-6
3-4
3-5
4-5
4-6

How difficult?

- Any programmer could do it.
- Typical diligent algorithms student could do it.
- ✓ Hire an expert.
- Intractable.
- No one knows.
- Impossible.

linear-time DFS-based planarity algorithm
discovered by Tarjan in 1970s
(too complicated for practitioners)