https://algs4.cs.princeton.edu

4. GRAPHS AND DIGRAPHS I

- *introduction*
- *graph representation*
- *depth-first search*
- *path finding*
- *undirected graphs*

https://algs4.cs.princeton.edu

4. Graphs and Digraphs I

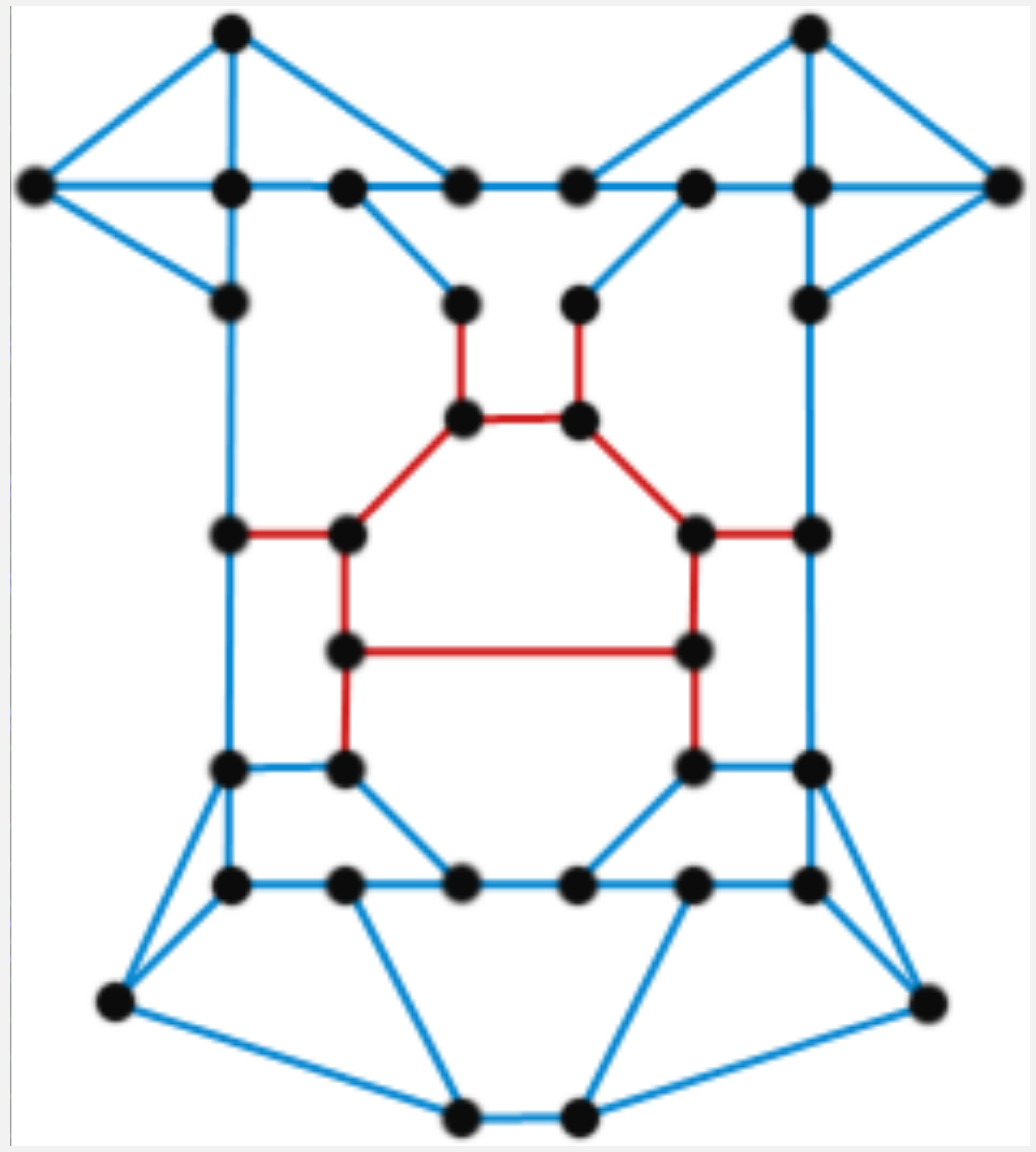
- *introduction*
- *graph representation*
- *depth-first search*
- *path finding*
- *undirected graphs*

Graphs

Graph. Set of vertices connected pairwise by edges.

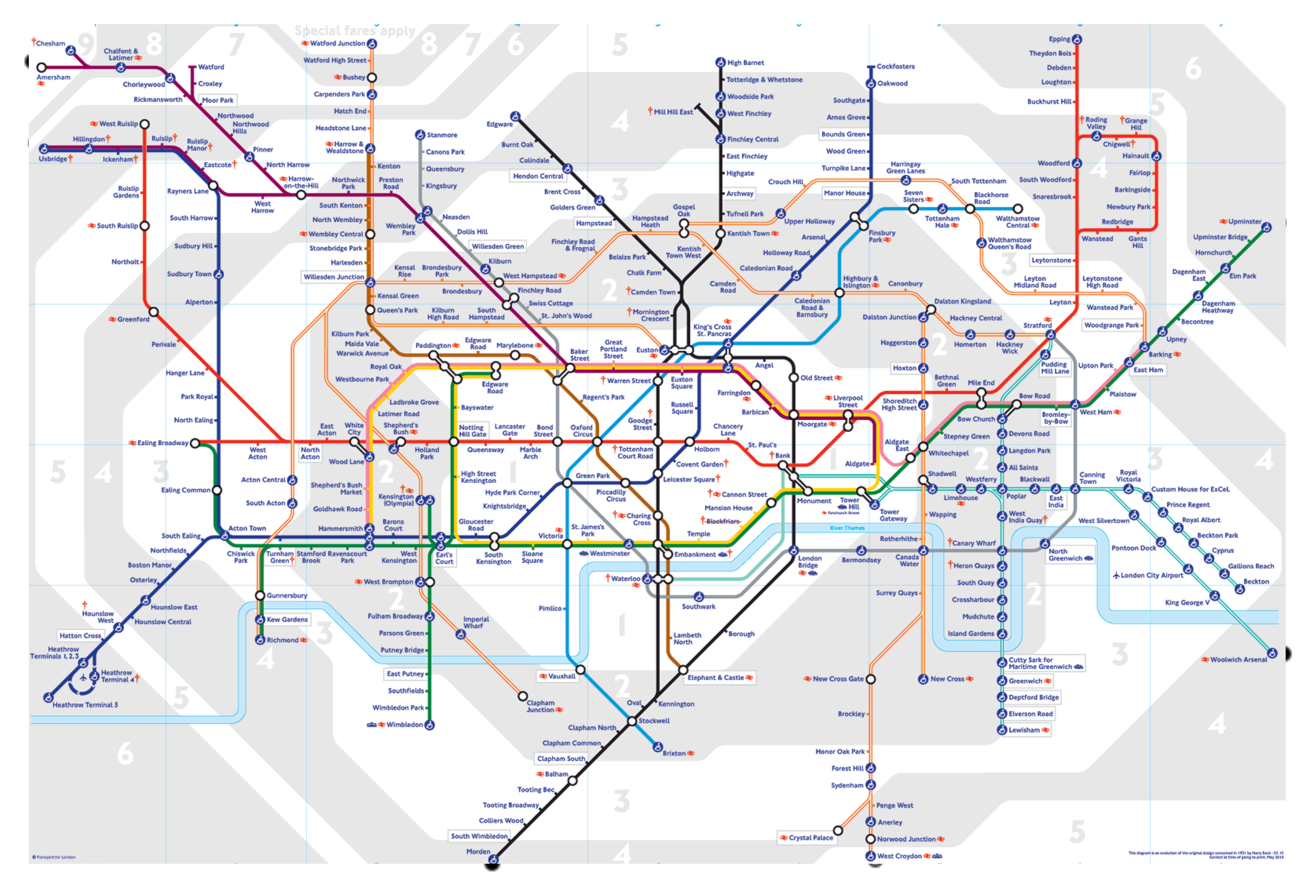
Why study graphs and graph algorithms?

- Broadly useful abstraction.
- Hundreds of graph algorithms.
- Thousands of real-world applications.
- Fascinating branch of computer science and discrete math.



Transportation networks

Vertex = subway stop; edge = direct route.



London Underground (Tube) Map

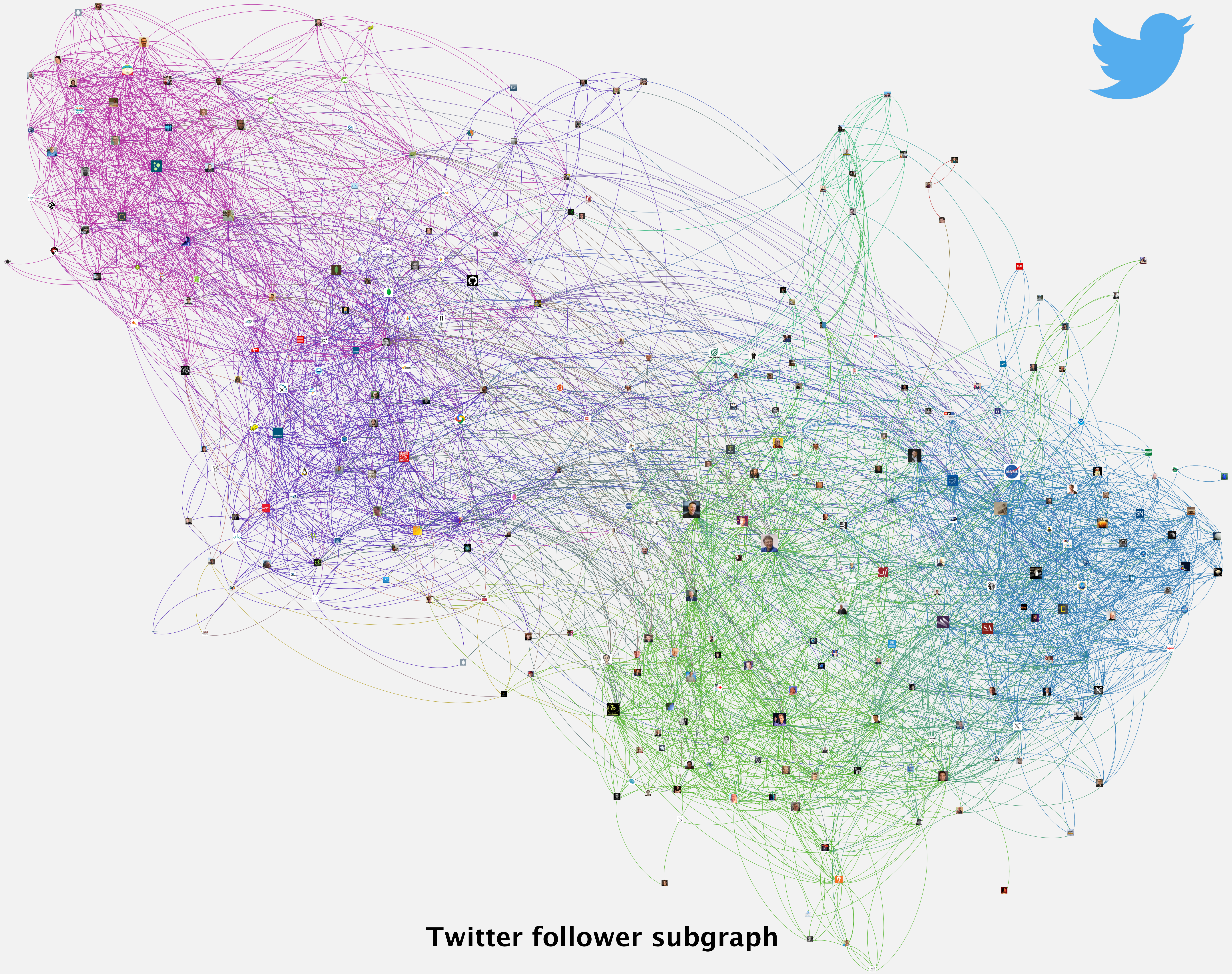
Social networks

Vertex = person; edge = social relationship.

"Visualizing Friendships" by Paul Butler

Twitter followers

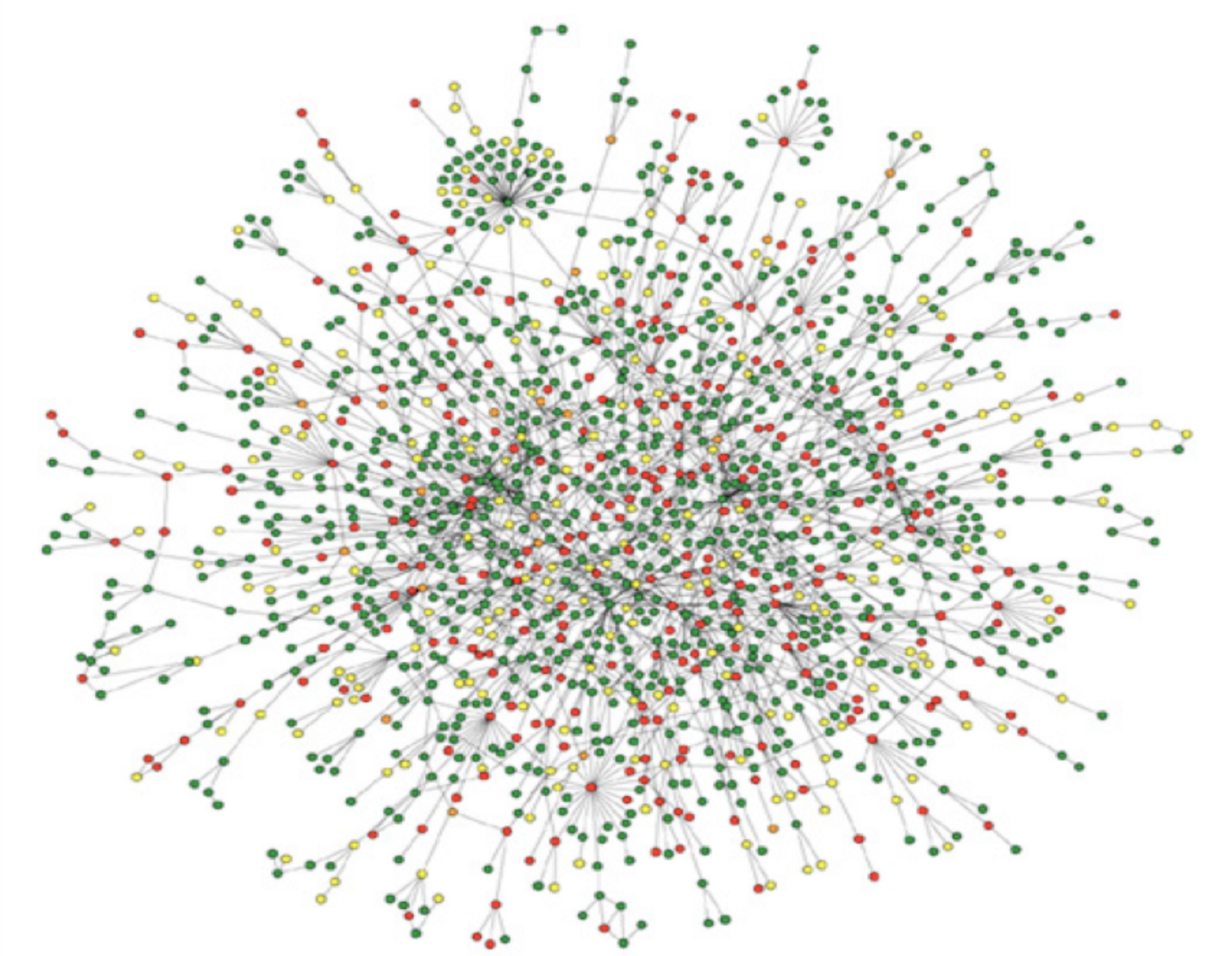
Vertex = Twitter account; edge = Twitter follower.



Twitter follower subgraph

Protein-protein interaction network

Vertex = protein; edge = interaction.



Reference: Jeong et al, Nature Review | Genetics

Graph applications

graph	vertex	edge
cell phone	phone	placed call
infectious disease	person	infection
financial	stock, currency	transactions
transportation	intersection	street
internet	router	fiber cable
web	web page	URL link
social relationship	person	friendship
object graph	object	pointer
protein network	protein	protein–protein interaction
circuit	gate, register, processor	wire
neural network	neuron	synapse

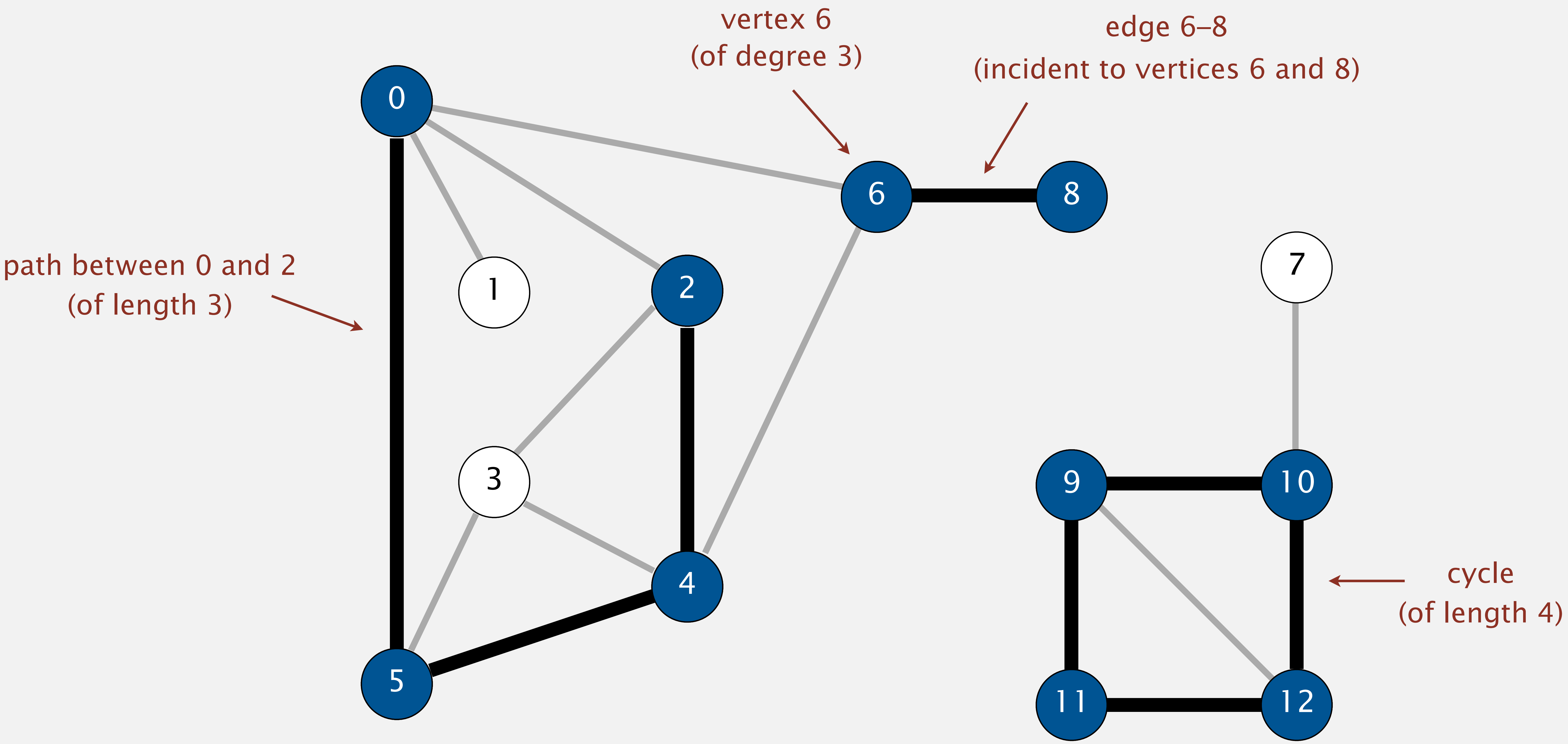
Undirected graph terminology

Graph. Set of vertices connected pairwise by edges.

Path. Sequence of vertices connected by edges, with no repeated edges.

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

Cycle. Path (with ≥ 1 edge) whose first and last vertices are the same.



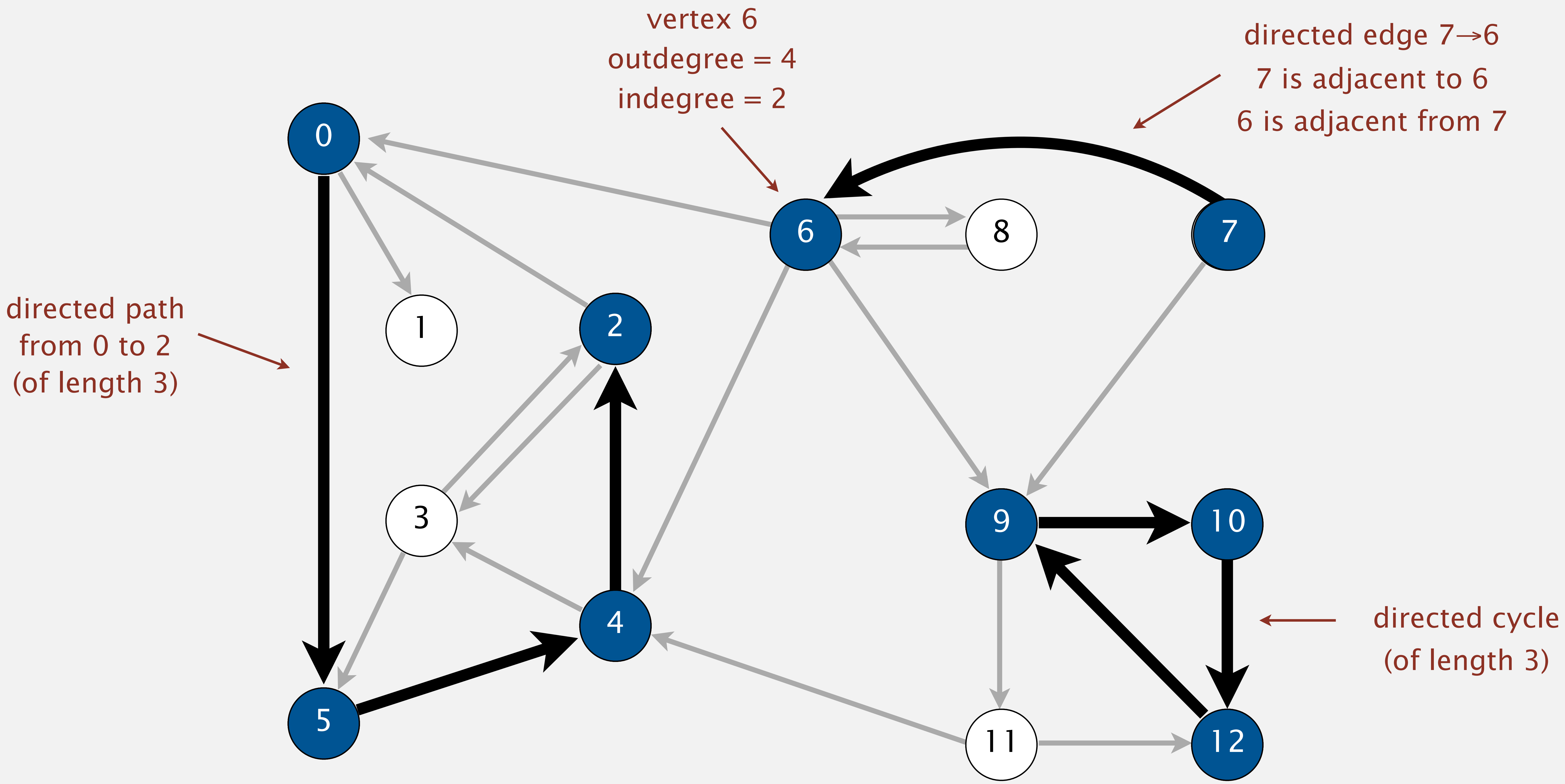
Directed graph terminology

Digraph. Set of vertices connected pairwise by directed edges.

Directed path. Sequence of vertices connected by directed edges, with no repeated edges.

Def. Vertex w is reachable from vertex v if there is a directed path from v to w.

Directed cycle. Directed path (with ≥ 1 edge) whose first and last vertices are the same.



Which of these graphs is best modeled as a directed graph?

A. Facebook: vertex = person; edge = friendship.

B. Web: vertex = webpage; edge = URL link.

C. Internet: vertex = router; edge = fiber optic cable.

D. Molecule: vertex = atom; edge = chemical bond.

Some graph-processing problems

graph problem	description
s-t path	*Find a path between s and t.*
shortest s-t path	*Find a path with the fewest edges between s to t.*
cycle	*Find a cycle.*
Euler cycle	*Find a cycle that uses each edge exactly once.*
Hamilton cycle	*Find a cycle that uses each vertex exactly once.*
connectivity	*Is there a path between every pair of vertices ?*
graph isomorphism	*Are two graphs isomorphic?*
planarity	*Draw the graph in the plane with no crossing edges.*

← digraph versions

Challenge. Which problems are easy? Difficult? Intractable?

https://algs4.cs.princeton.edu

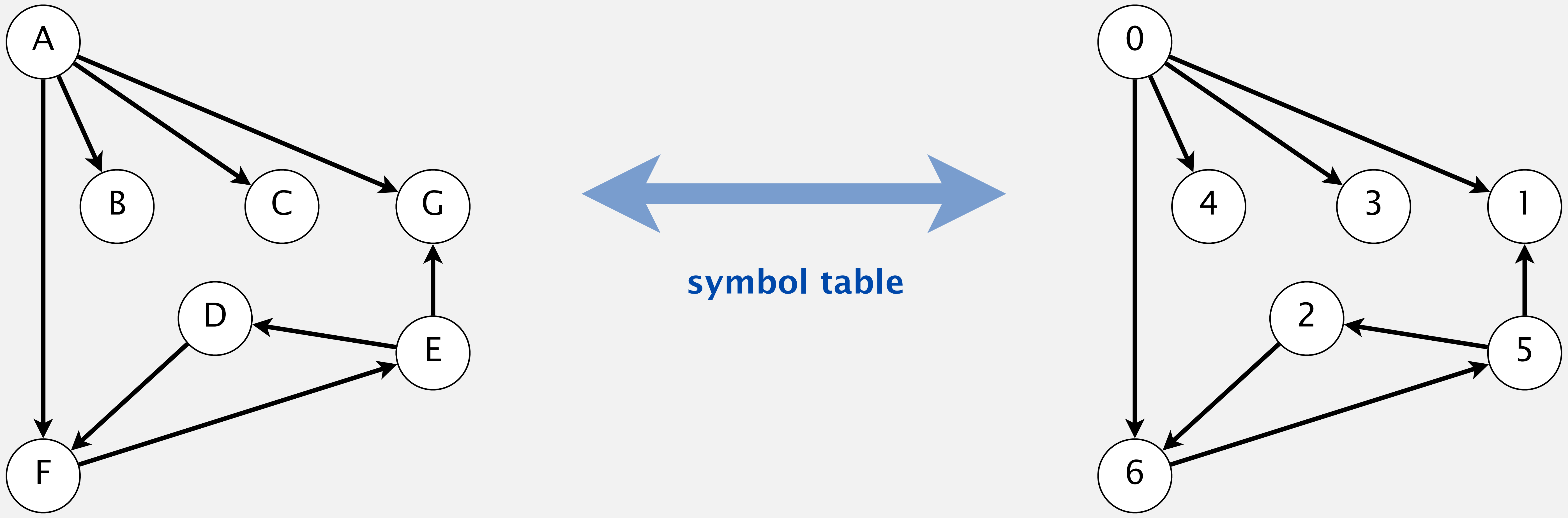
4. Graphs and Digraphs I

- introduction
- **graph representation**
- depth-first search
- path finding
- undirected graphs

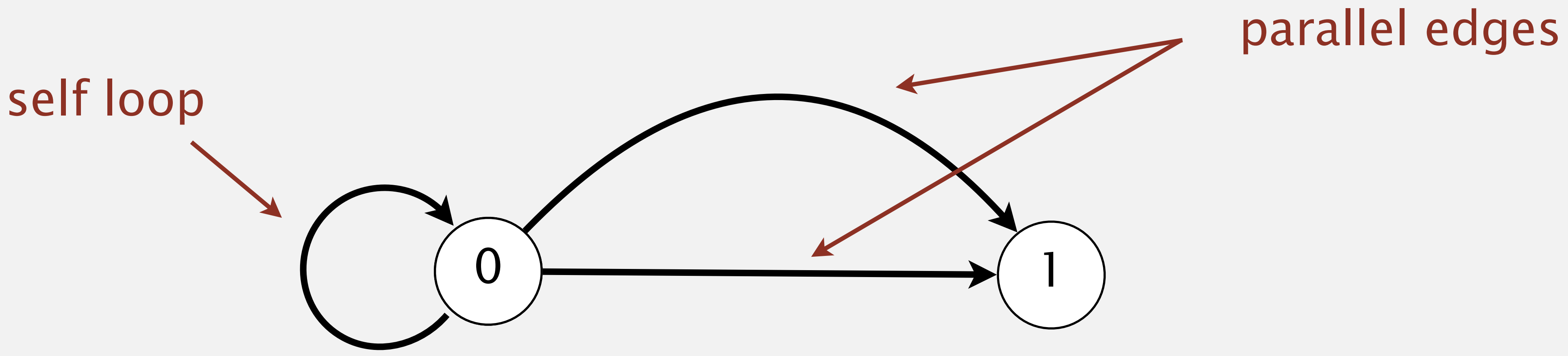
Digraph representation

Vertex representation.

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



Def. A digraph is simple if it has no self-loops or parallel edges.



Digraph API

	`public class Digraph`	
	`Digraph(int V)`	*create an empty digraph with V vertices*
`void`	`addEdge(int v, int w)`	*add a directed edge v→w*
`Iterable<Integer>`	`adj(int v)`	*vertices adjacent from v*
`int`	`V()`	*number of vertices*
	⋮	⋮

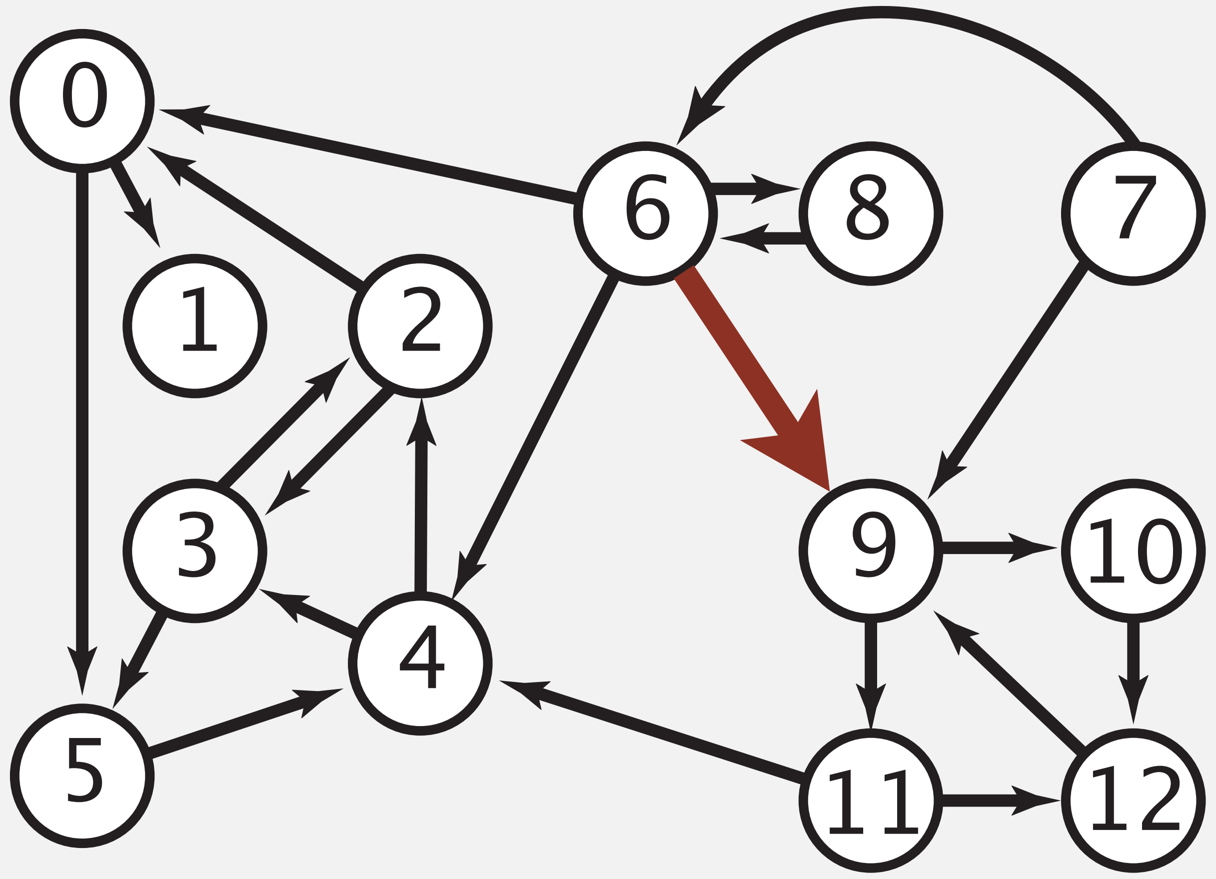
← this API allows self loops and parallel edges

```
// outdegree of vertex v in digraph G
public static int outdegree(Digraph G, int v)
{
    int count = 0;
    for (int w : G.adj(v))
       count++;
    return count;
}
```

← Note: this method is in full Digraph API, so no need to re-implement

Adjacency-matrix representation

Maintain a V-by-V boolean array; for each edge $v \to w$ 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

Note: parallel edges disallowed

Graphs and digraphs: quiz 2

What is the running time of the following code fragment?

Assume adjacency-matrix representation, V = # vertices, E = # edges.

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

print each edge once

A. $\Theta(V)$

B. $\Theta(E + V)$

C. $\Theta(V^2)$

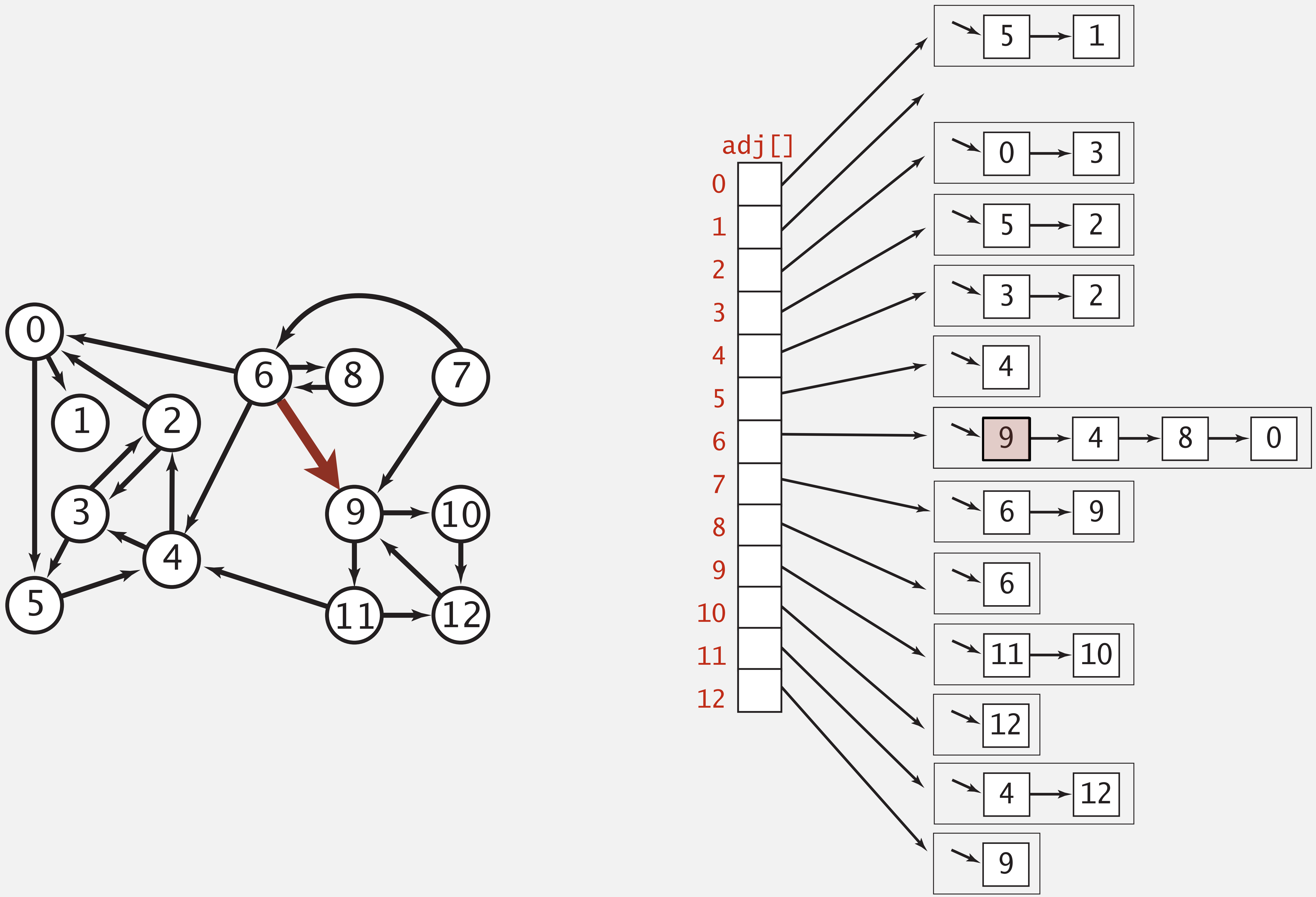
D. $\Theta(E\ V)$

	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

adjacency-matrix representation

Adjacency-lists representation

Maintain vertex-indexed array of lists.

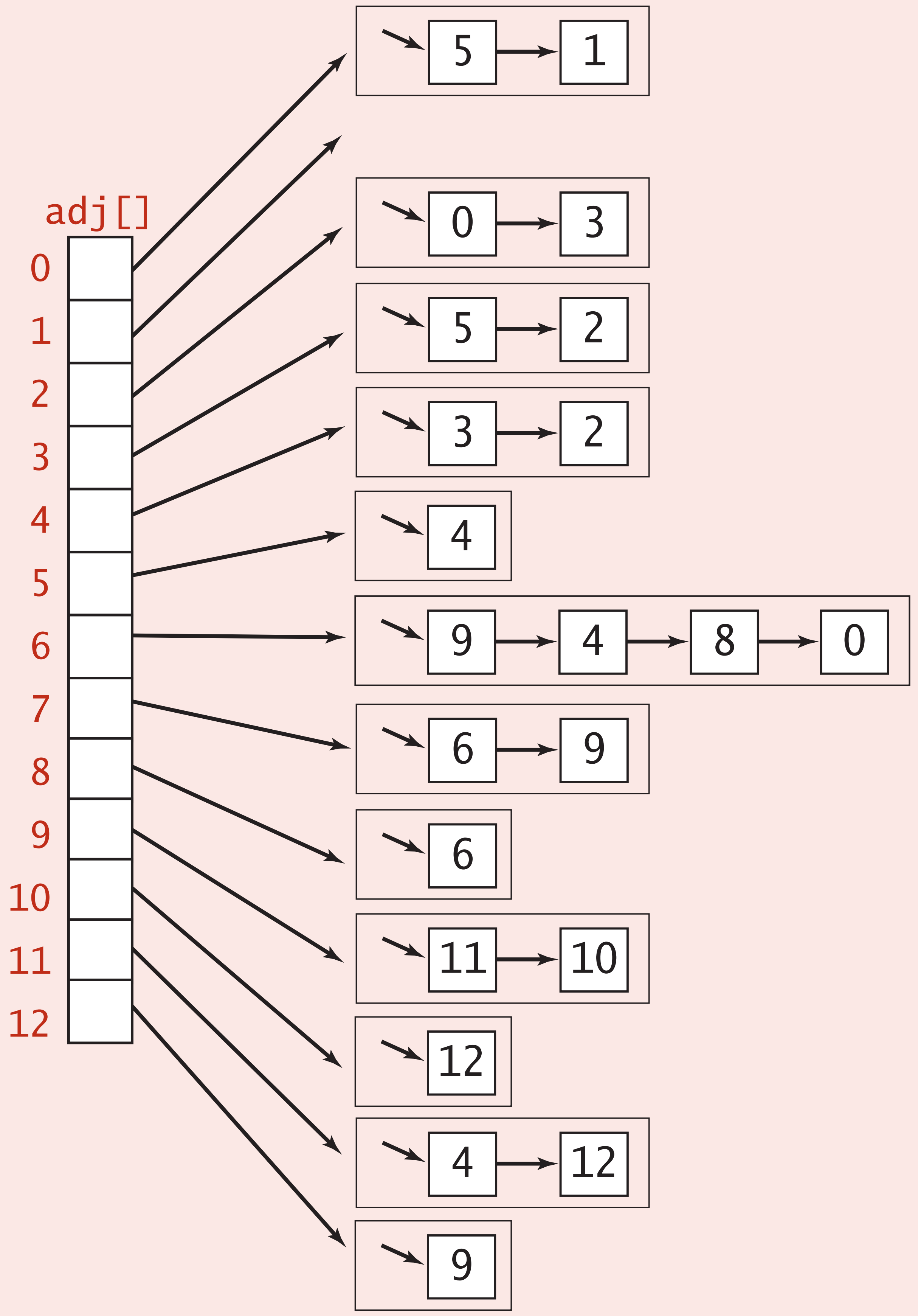


What is the running time of the following code fragment?

Assume adjacency-lists representation, V = # vertices, E = # edges.

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

print each edge once



A. $\Theta(V)$

B. $\Theta(E + V)$

C. $\Theta(V^2)$

D. $\Theta(E\ V)$

Digraph representations

In practice. Use adjacency-lists representation.

- Algorithms based on iterating over vertices adjacent from v.
- Real-world graphs tend to be sparse (not dense).

sparse: $\Theta(V)$ edges   dense: $\Theta(V^2)$ edges

representation	space	add edge from v to w	has edge from v to w?	iterate over vertices adjacent from v?
adjacency matrix	V^2	1 †	1	V
adjacency lists	$E + V$	1	$outdegree(v)$	$outdegree(v)$

† disallows parallel edges

Digraph representation (adjacency lists): Java implementation

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

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

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

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

}
```

adjacency lists

create empty digraph with V vertices

add edge $v \to w$
(parallel edges and self-loops allowed)

iterator for vertices adjacent from v

https://algs4.cs.princeton.edu/42digraph/Digraph.java.html

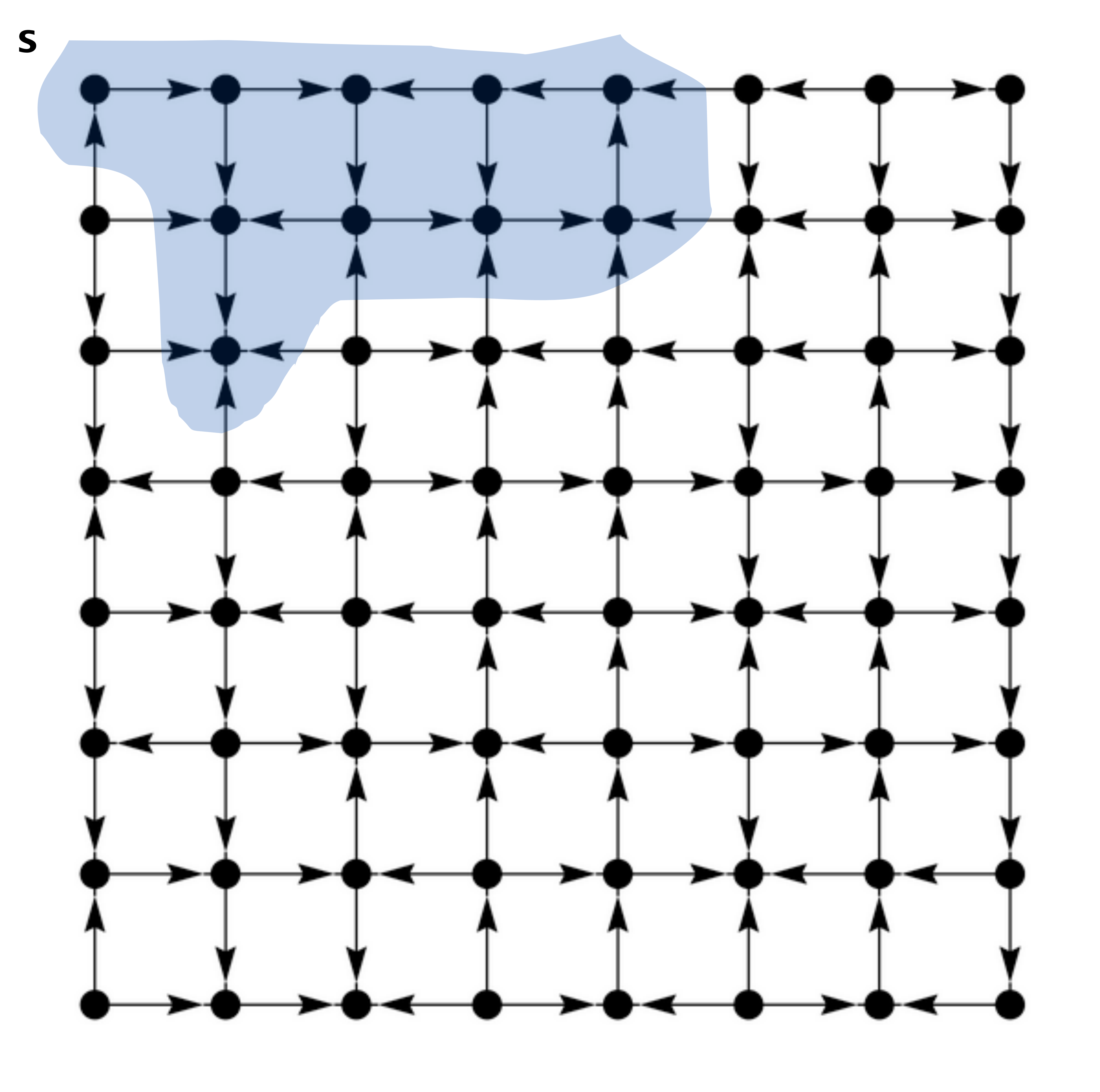
https://algs4.cs.princeton.edu

4. Graphs and Digraphs I

- *introduction*
- *graph representation*
- *depth-first search*
- *path finding*
- *undirected graphs*

Digraph reachability

Problem. Given a digraph G and vertex s, find all vertices reachable from s.



Depth-first search

Goal. Systematically traverse a digraph.

DFS (to visit a vertex v)

Mark vertex v.

Recursively visit all unmarked
vertices w adjacent from v.

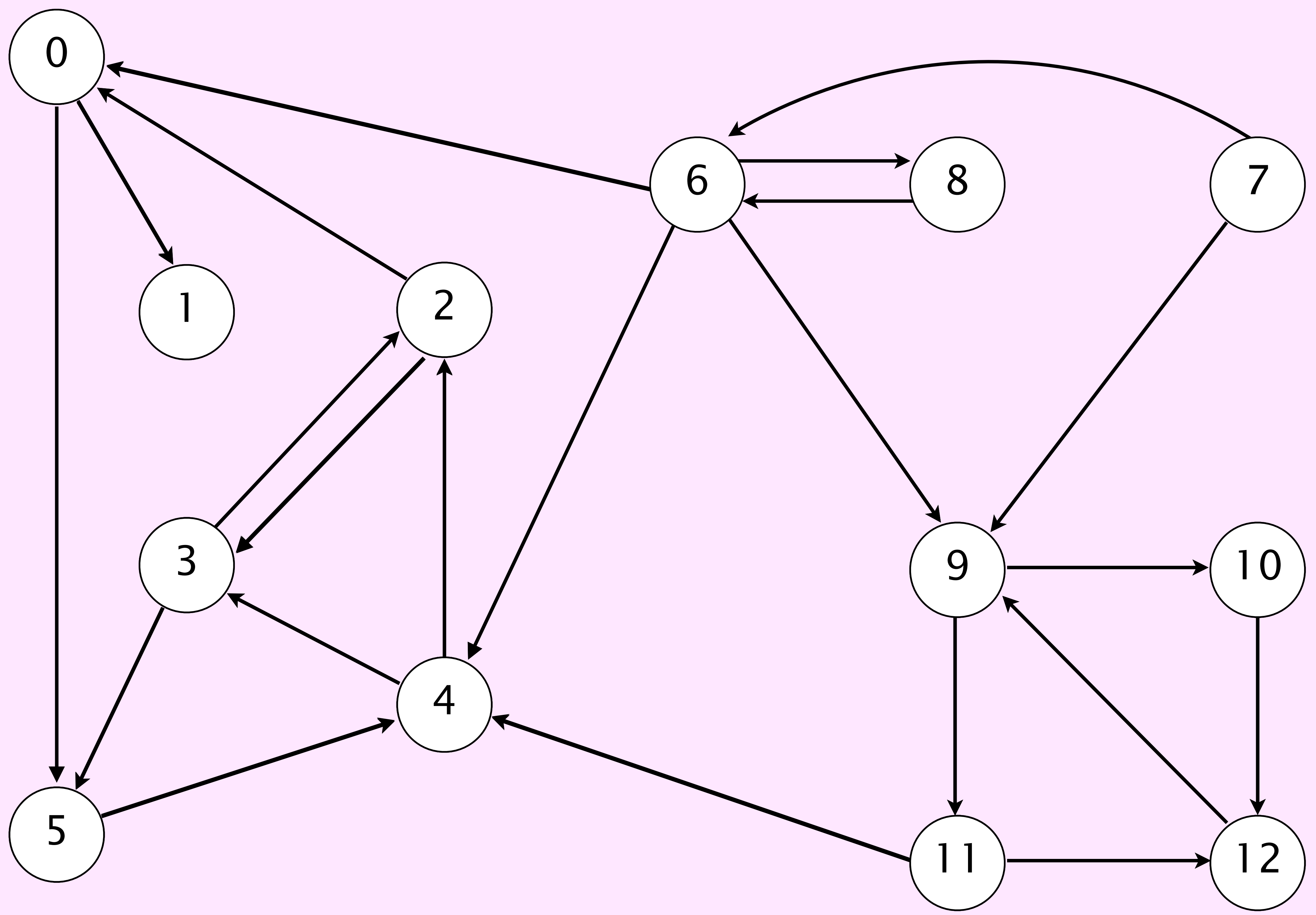
Typical applications.

- Reachability: find all vertices reachable from a given vertex.
- Path finding: find a directed path from one vertex to another vertex.

Directed depth-first search demo

To visit a vertex v :

- Mark vertex v.
- Recursively visit all unmarked vertices adjacent from v.



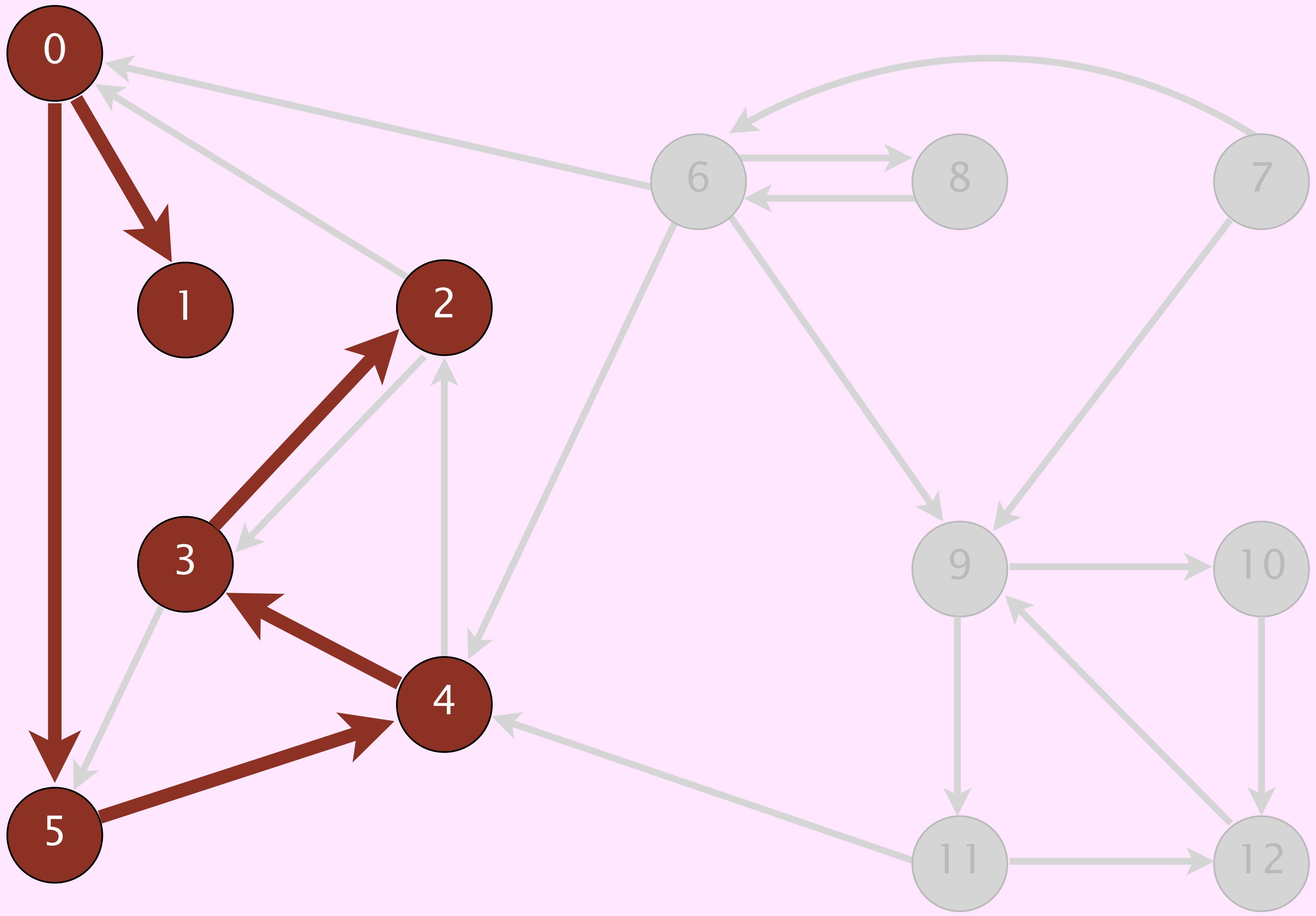
a directed graph

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.
- Recursively visit all unmarked vertices adjacent from v.



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

reachable from vertex 0

reachable from 0

Run DFS using the following adjacency-lists representation of digraph G, starting at vertex 0. In which order is dfs(G, v) called?

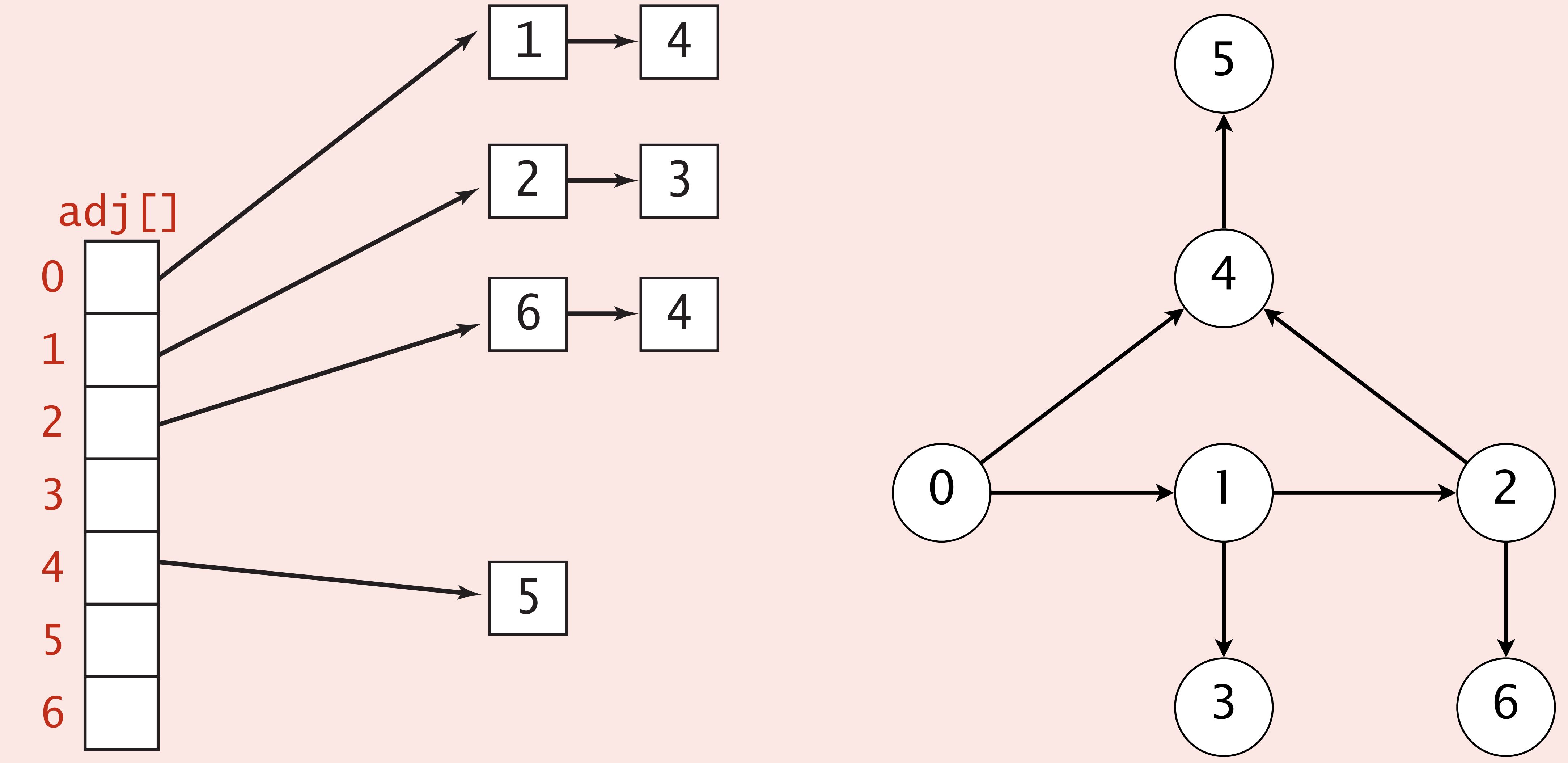
DFS preorder

A. 0 1 2 4 5 3 6

B. 0 1 2 4 5 6 3

C. 0 1 3 2 6 4 5

D. 0 1 2 6 4 5 3



adjacency-lists representation **digraph G**

Depth-first search: Java implementation

```
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 isReachable(int v)
   {  return marked[v];  }
}
```

marked[v] = true if v reachable from s

constructor marks vertices reachable from s

recursive DFS does the work

is v reachable from s ?

Depth-first search: properties

Proposition. DFS marks all vertices reachable from s in $\Theta(E + V)$ time in the worst case.

Pf.

- Initializing an array of length V takes $\Theta(V)$ time.
- Each vertex is visited at most once.
- Visiting a vertex takes time proportional to its outdegree:

$$outdegree(v_0) \ + \ outdegree(v_1) \ + \ outdegree(v_2) \ + \ldots \ = \ E$$

↑ in worst case, all V vertices reachable from s

Note. If all vertices are reachable from s, then $E \ge V - 1$, so V is a lower-order term.

Graphs and digraphs: quiz 5

What could happen if we marked a vertex at the end of the DFS call (instead of beginning)?

- **A.** Marks a vertex not reachable from s.
- **B.** Compile-time error.
- **C.** Infinite loop / stack overflow.
- **D.** None of the above.

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

}
```

Reachability application: program control-flow analysis

Every program is a digraph.

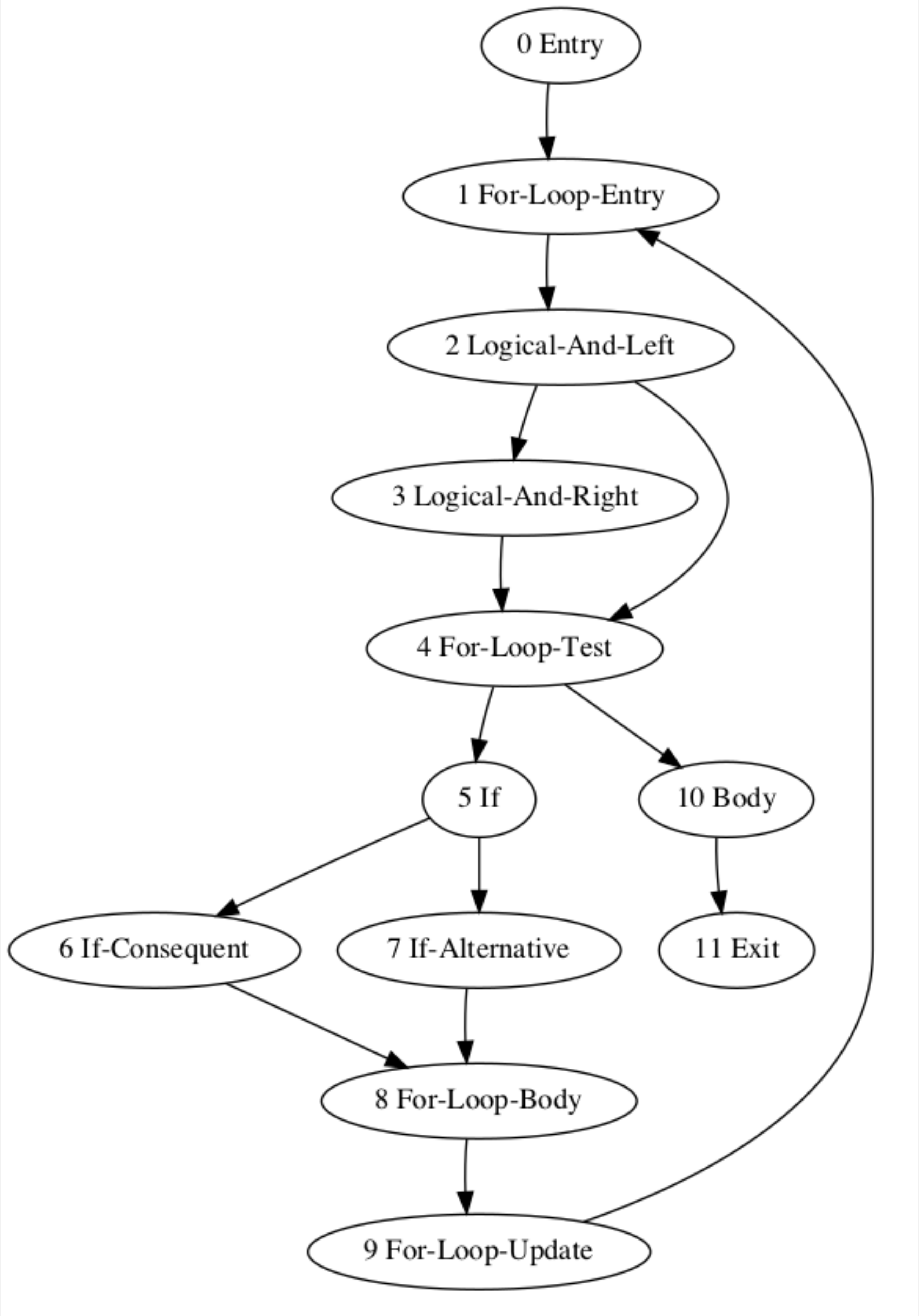
- Vertex = basic block of instructions (straight-line program).
- Edge = jump.

Dead-code elimination.

Find (and remove) unreachable code.

Infinite-loop detection.

Determine whether exit is unreachable.



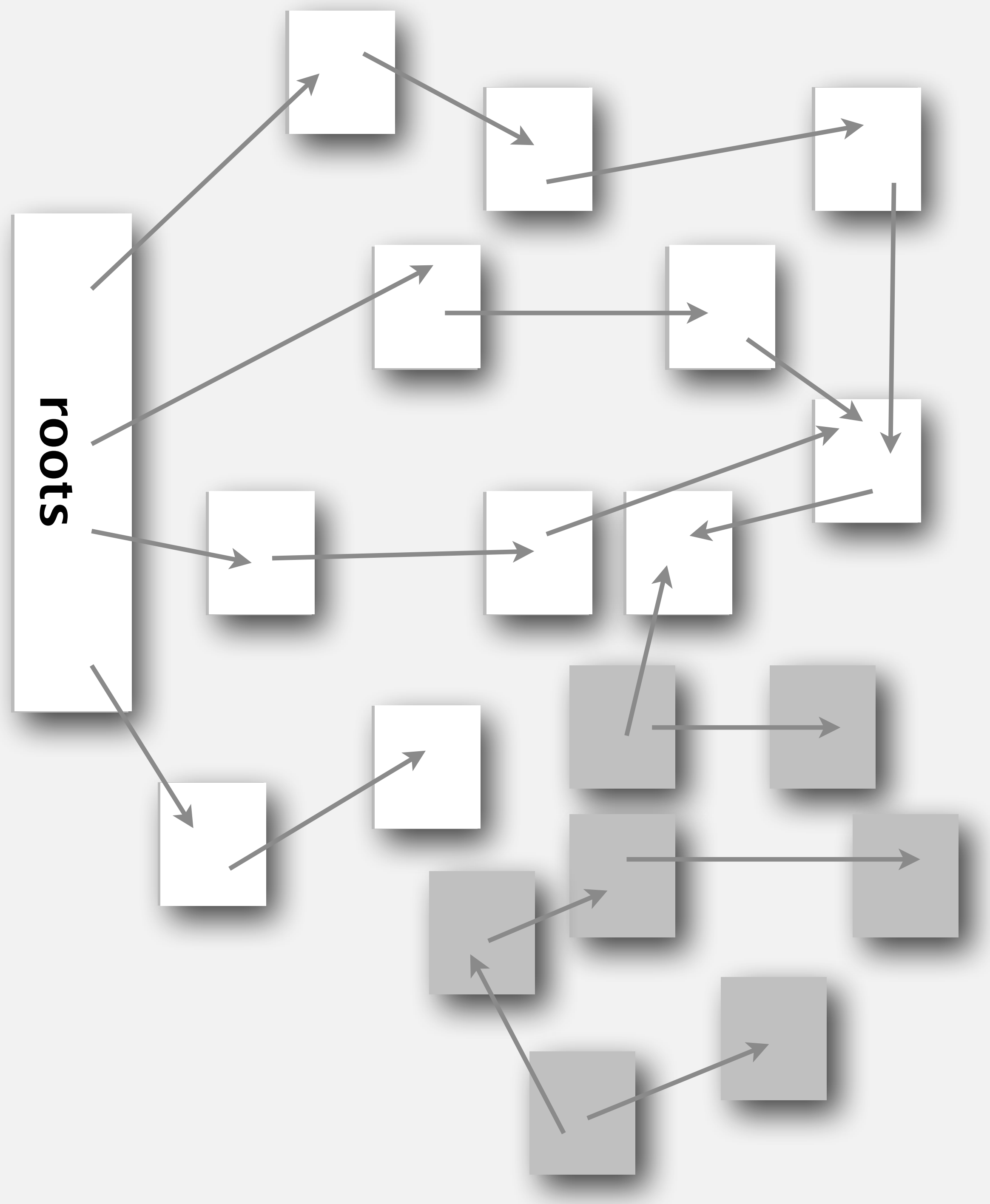
Reachability application: mark-sweep garbage collector

Every data structure is a digraph.

- Vertex = object.
- Edge = reference/pointer.

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

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

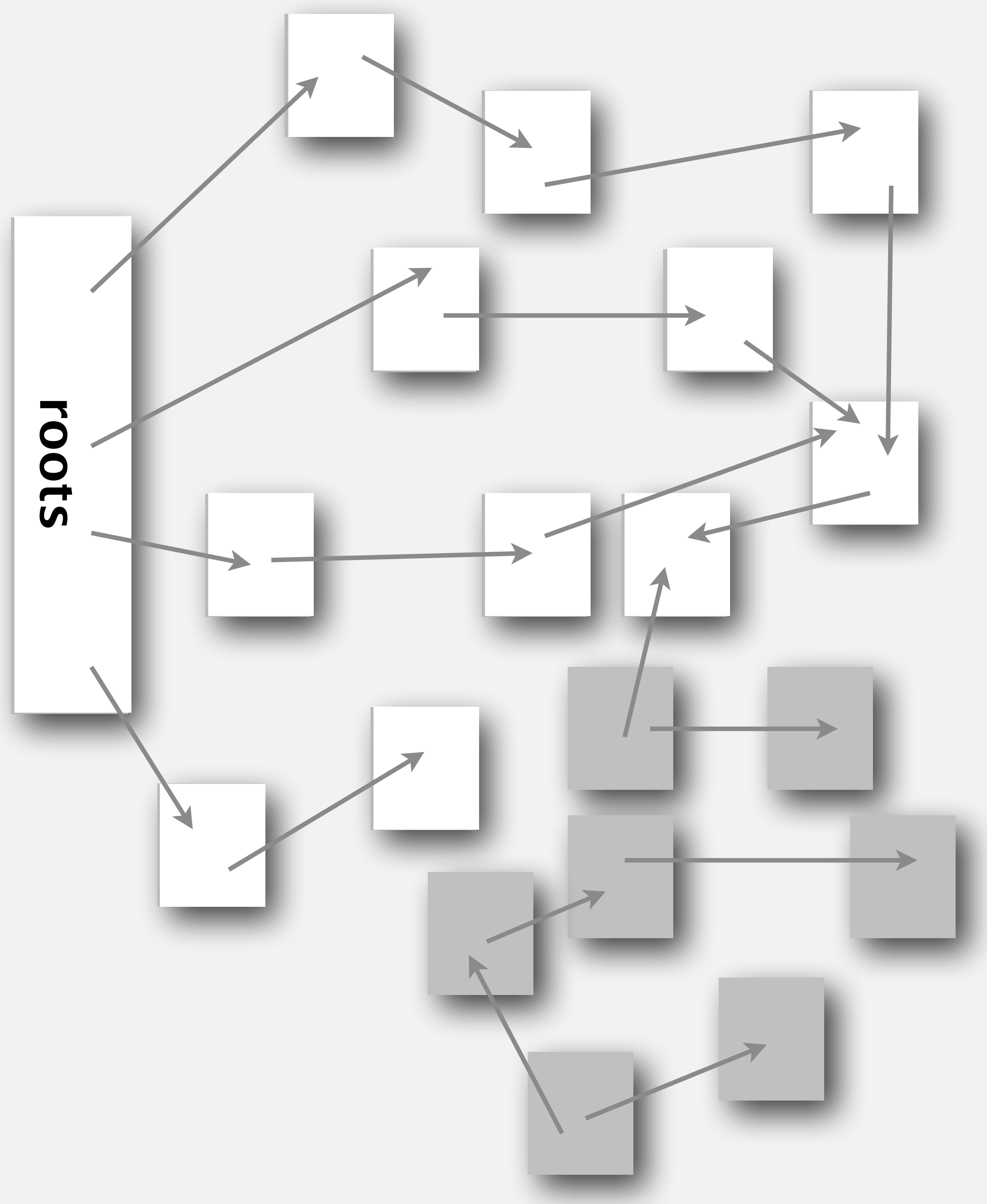


Reachability application: mark–sweep garbage collector

Mark–sweep algorithm. [McCarthy, 1960]

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

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



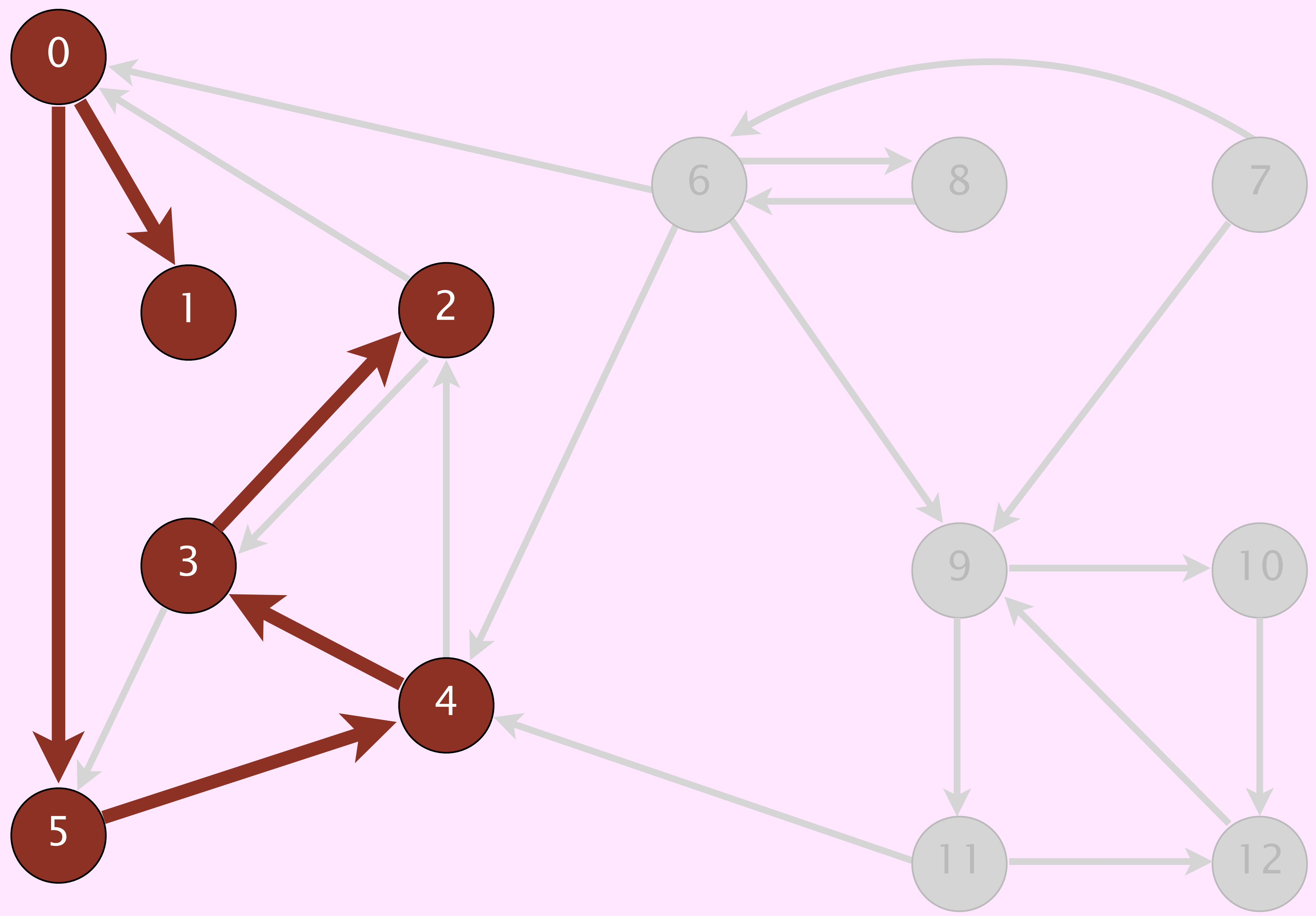
4. Graphs and Digraphs I

- *introduction*
- *graph representation*
- *depth-first search*
- ***path finding***
- *undirected graphs*

Directed paths DFS demo

Goal. DFS determines which vertices are reachable from s. How to reconstruct paths?

Solution. Use parent-link representation.



reachable from 0

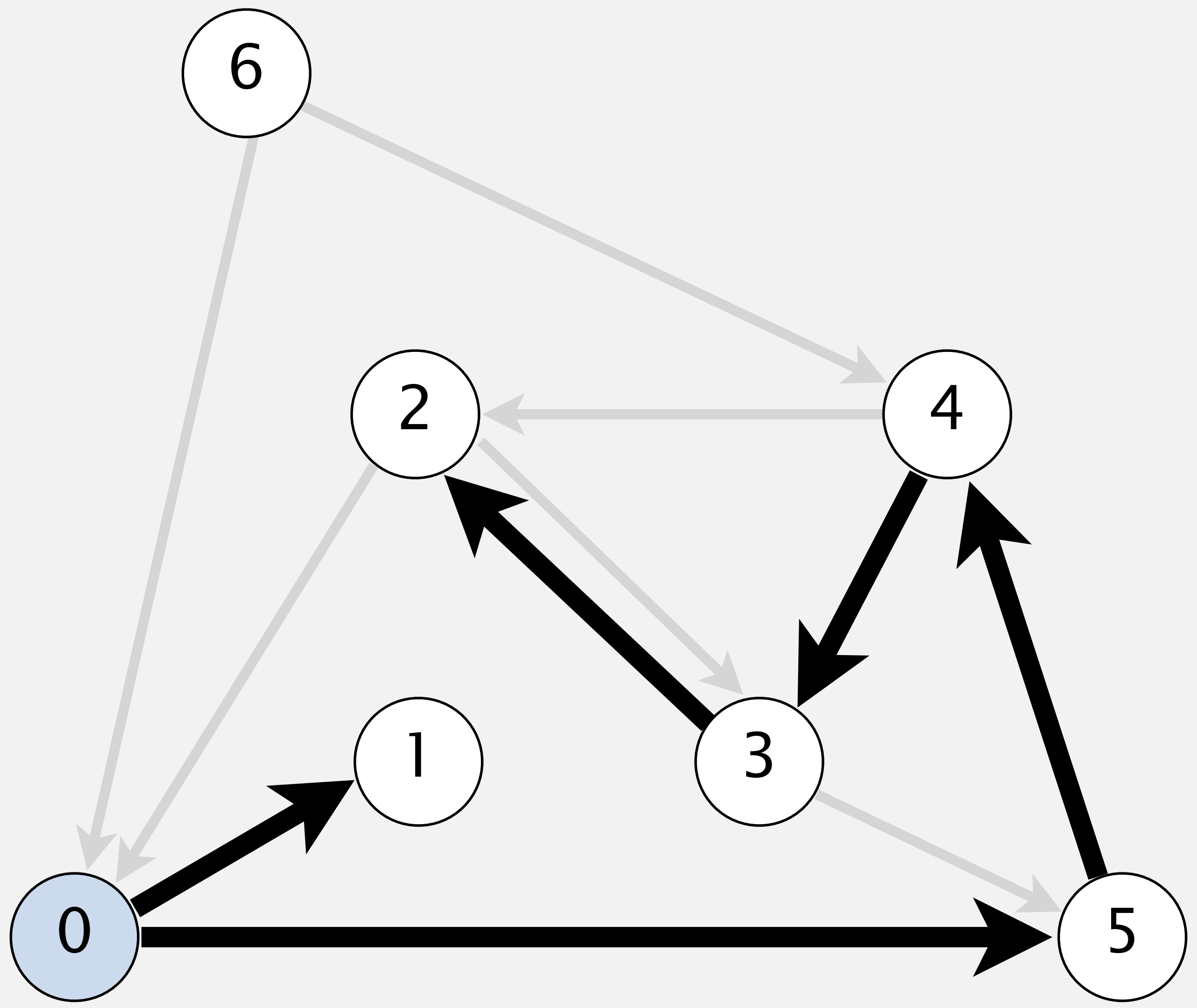
v	marked[]	edgeTo[]
0	T	–
1	T	0
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

parent-link representation of paths from vertex 0

Depth-first search: path finding

Parent-link representation of paths from *s*.

- Maintain an integer array `edgeTo[]`.
- Interpretation: `edgeTo[v]` is the next-to-last vertex on a path from *s* to *v*.
- To reconstruct path from *s* to *v*, trace `edgeTo[]` backward from *v* to *s* (and reverse).



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

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

Depth-first search (with path finding): Java implementation

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

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

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

https://algs4.cs.princeton.edu/42digraph/DepthFirstDirectedPaths.java.html

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

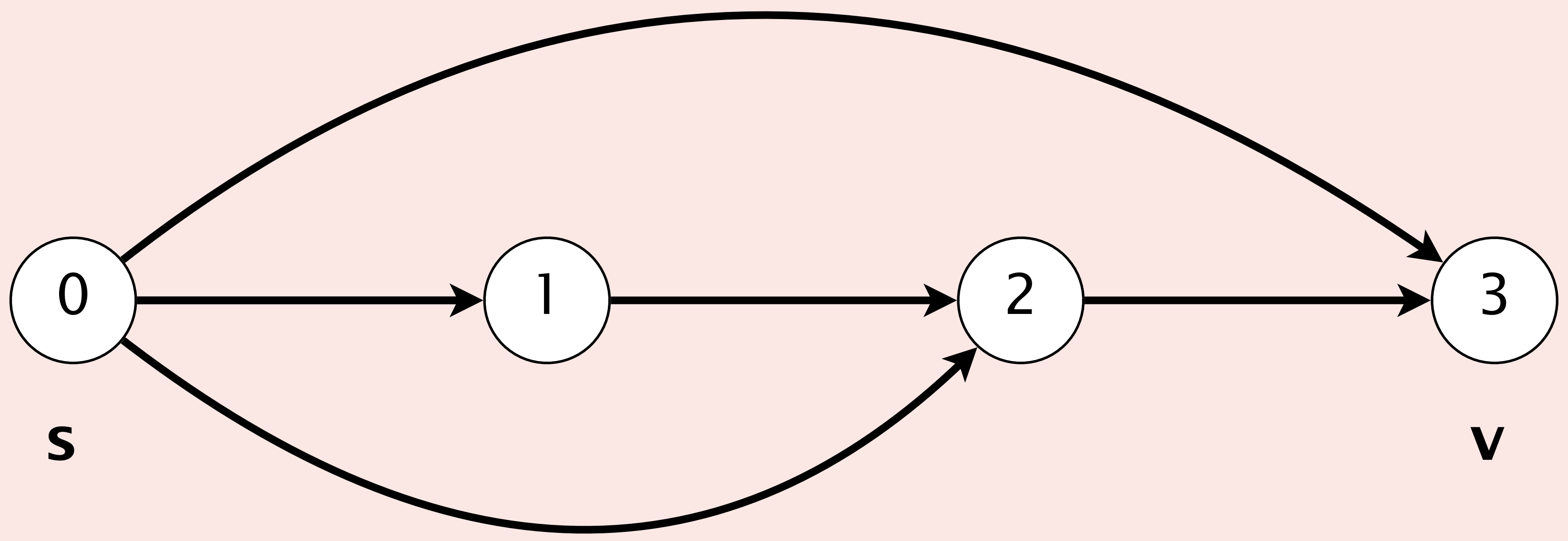
$v \rightarrow w$ is edge that led to w

Suppose there are many paths from s to v. Which one does DepthFirstDirectedPaths find?

A. A shortest path (fewest edges).

B. A longest path (most edges).

C. Depends on digraph representation.



Algorithms

Robert Sedgewick | Kevin Wayne

https://algs4.cs.princeton.edu

4. Graphs and Digraphs I

- *introduction*
- *graph representation*
- *depth-first search*
- *path finding*
- ***undirected graphs***

Flood Fill

Problem. Implement flood fill (Photoshop magic wand).

Depth-first search in undirected graphs

Problem. Given an undirected graph G and vertex s, find all vertices connected to s.

Solution. Treat undirected graph as a digraph, replacing each edge with two antiparallel edges.

DFS (to visit a vertex v)

Mark vertex v.

Recursively visit all unmarked vertices w adjacent to v.

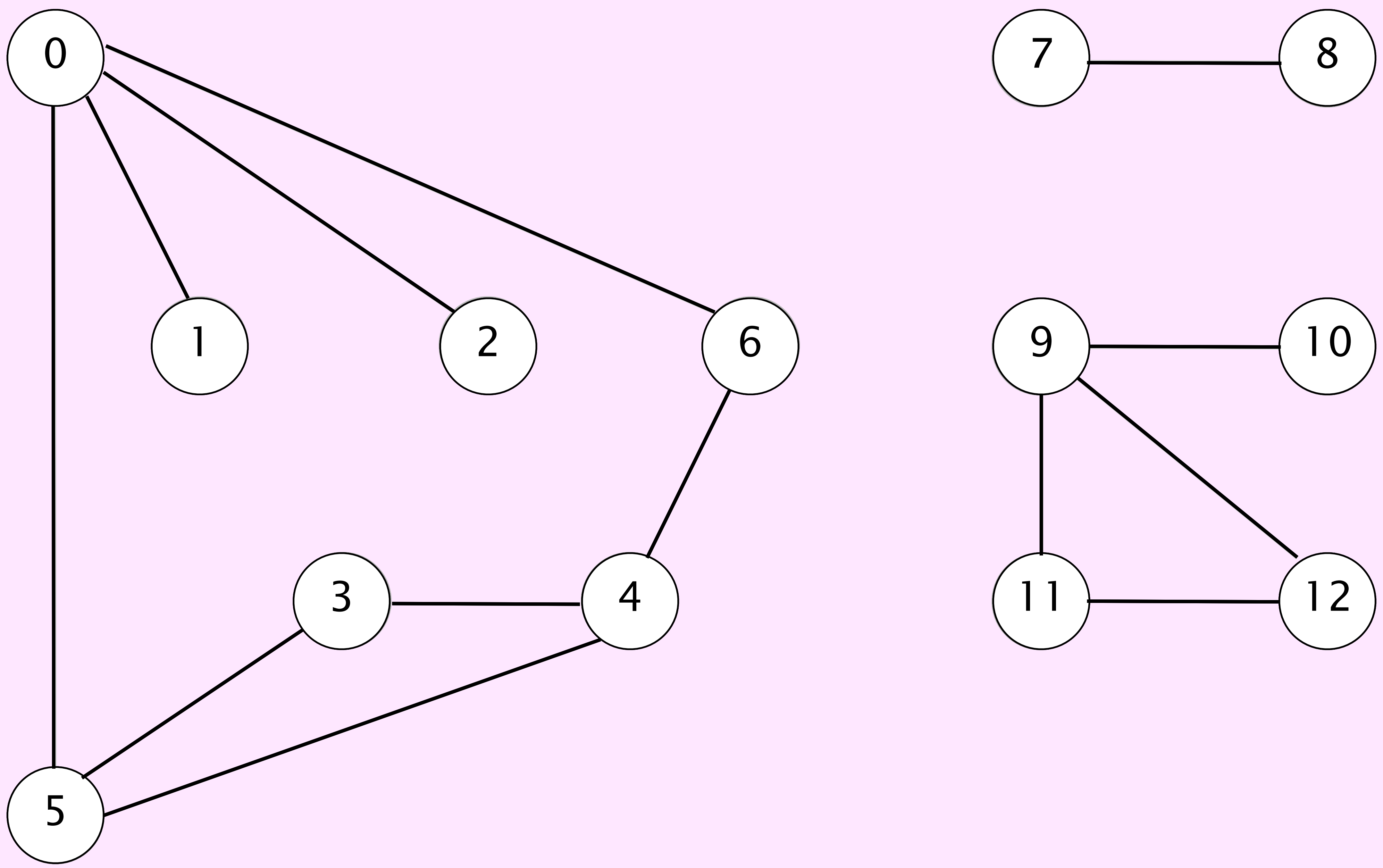
Typical applications.

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

Depth-first search demo

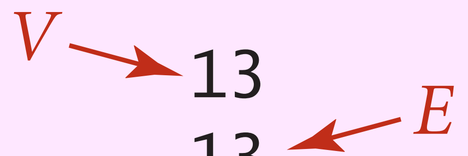
To visit a vertex v :

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



graph G

tinyG.txt

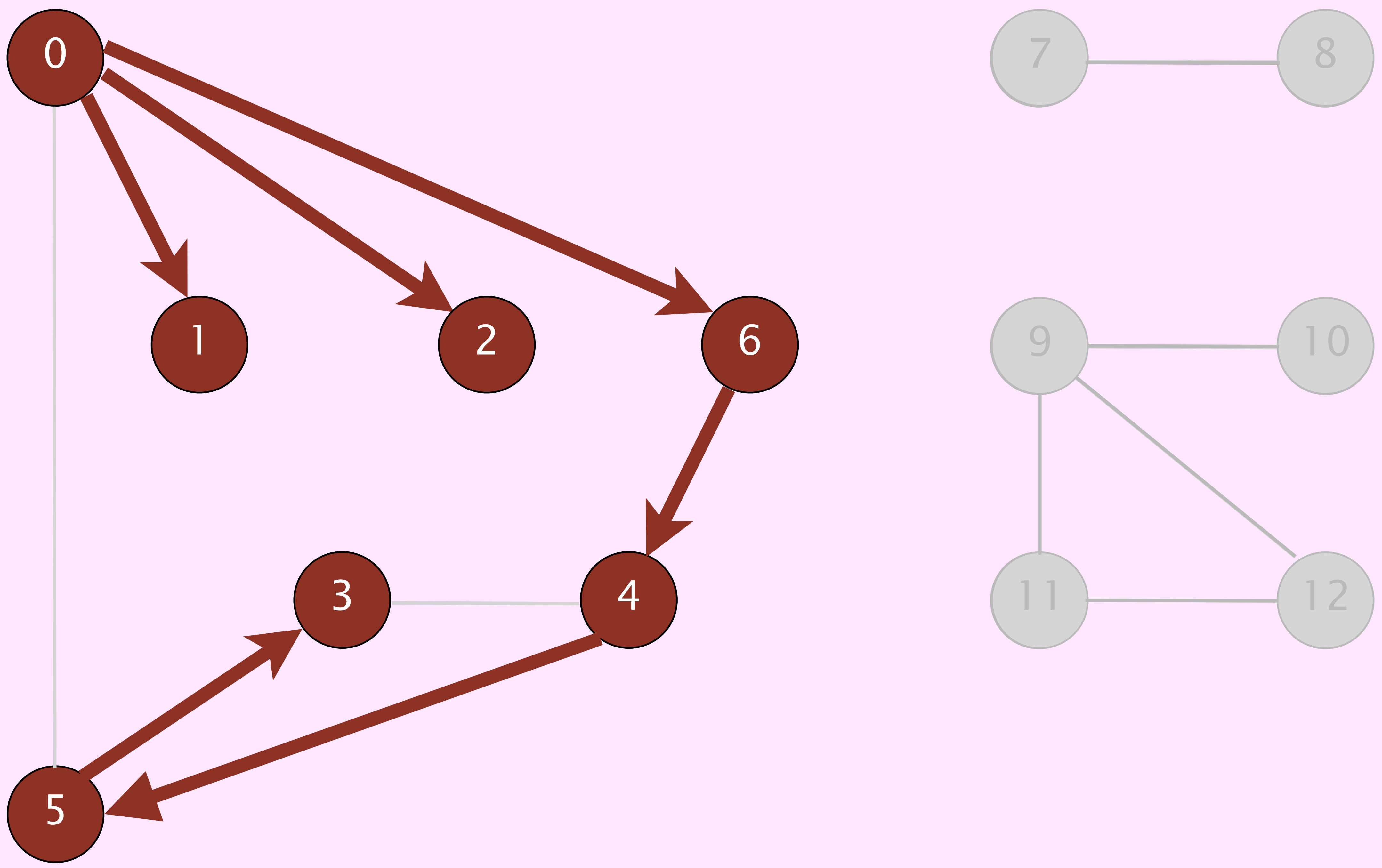



```
13
13
0 5
4 3
0 1
9 12
6 4
5 4
0 2
11 12
9 10
0 6
7 8
9 11
5 3
```

Depth-first search demo

To visit a vertex v :

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



v	marked[]	edgeTo[]
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	–

vertices connected to 0
(and associated paths)

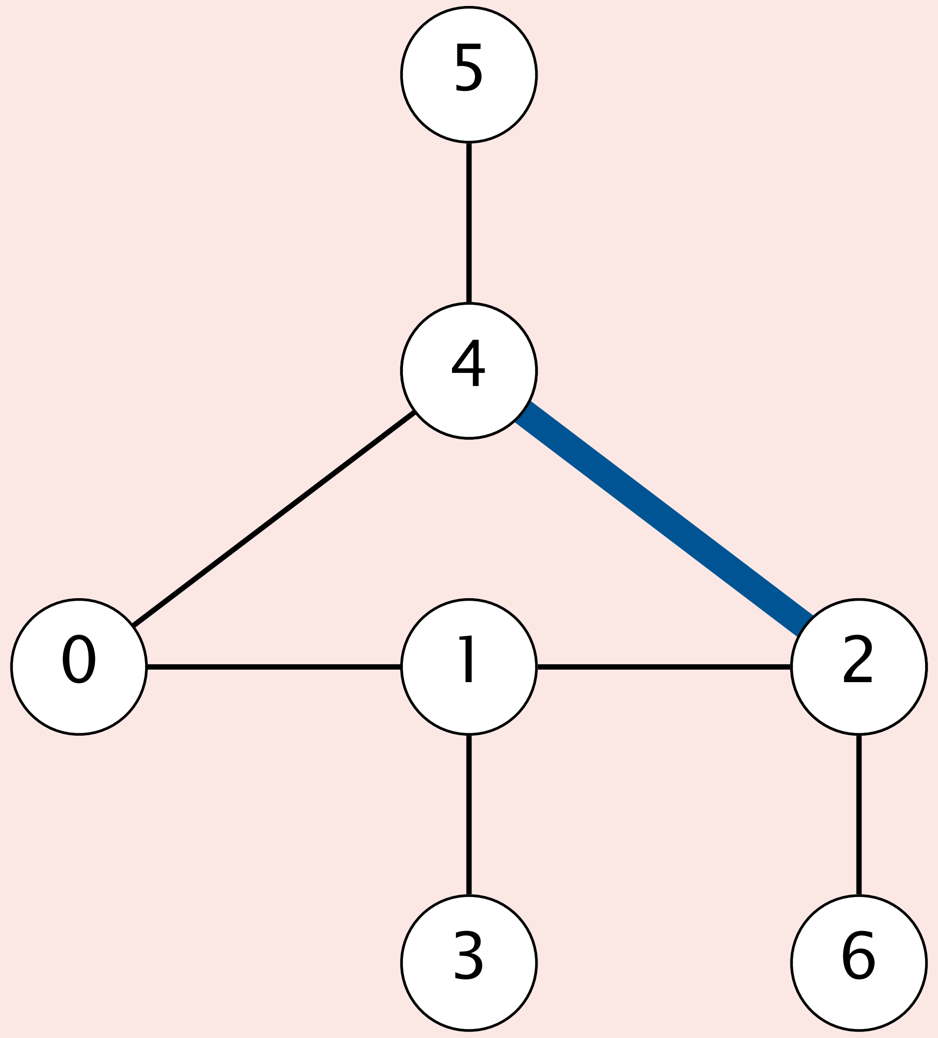
How to represent an undirected edge v–w using adjacency lists?

A. Add w to adjacency list for v.

B. Add v to adjacency list for w.

C. Both A and B.

D. None of the above.



Digraph representation (review)

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

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

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

   }

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

}
```

adjacency lists

create empty digraph with V vertices

add edge $v \to w$
(parallel edges and self-loops allowed)

iterator for vertices adjacent from v

https://algs4.cs.princeton.edu/42digraph/Digraph.java.html

Graph representation

```
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

create empty graph with V vertices

add edge v–w
(parallel edges and self-loops allowed)

iterator for vertices adjacent to v

https://algs4.cs.princeton.edu/41undirected/Graph.java.html

Depth-first search (in digraphs)

Recall code for digraphs.

```
public class DirectedFS
{
   private boolean[] marked;                          // marked[v] = true if v reachable from s

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

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

   public boolean visited(int v)                      // is vertex v is reachable from s ?
   {  return marked[v];  }
}
```

Depth-first search (in undirected graphs)

Code for undirected graphs is essentially identical to code for digraphs.

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

   public DepthFirstSearch(Graph G, int s)
   {
     marked = new boolean[G.V()];
     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);
   }

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

marked[v] = true if v connected to s

constructor marks vertices connected to s

recursive DFS does the work

is vertex v is connected to s ?

Depth-first search summary

DFS enables direct solution of simple graph and digraph problems.

- Reachability (in a digraph). ✓
- Connectivity (in a graph). ✓
- Path finding (in a graph or digraph). ✓
- Topological sort. ← next lecture
- Directed cycle detection. ← precept

DFS provides basis for solving difficult graph problems.

- Euler cycle.
- 2-satisfiability.
- Planarity testing.
- Strong components.

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

DEPTH-FIRST SEARCH AND LINEAR GRAPH ALGORITHMS*

ROBERT TARJAN†

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