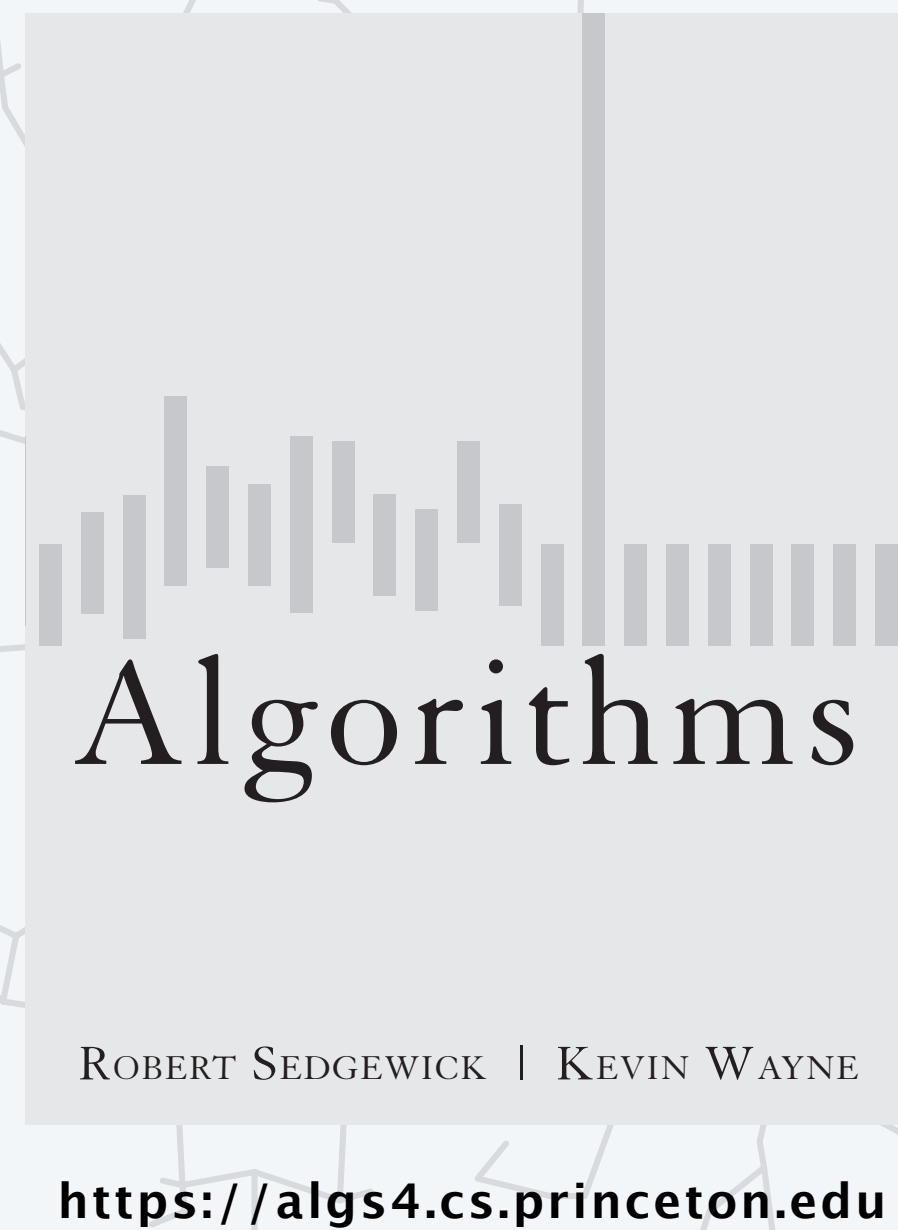




<https://algs4.cs.princeton.edu>

4. GRAPHS AND DIGRAPHS I

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



4. GRAPHS AND DIGRAPHS I

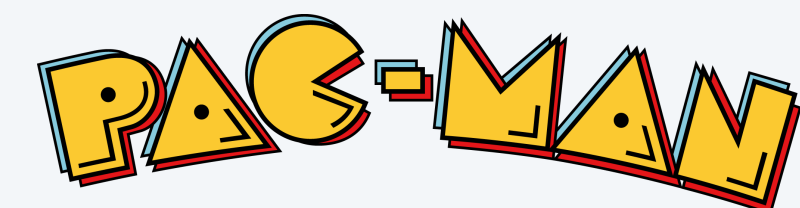
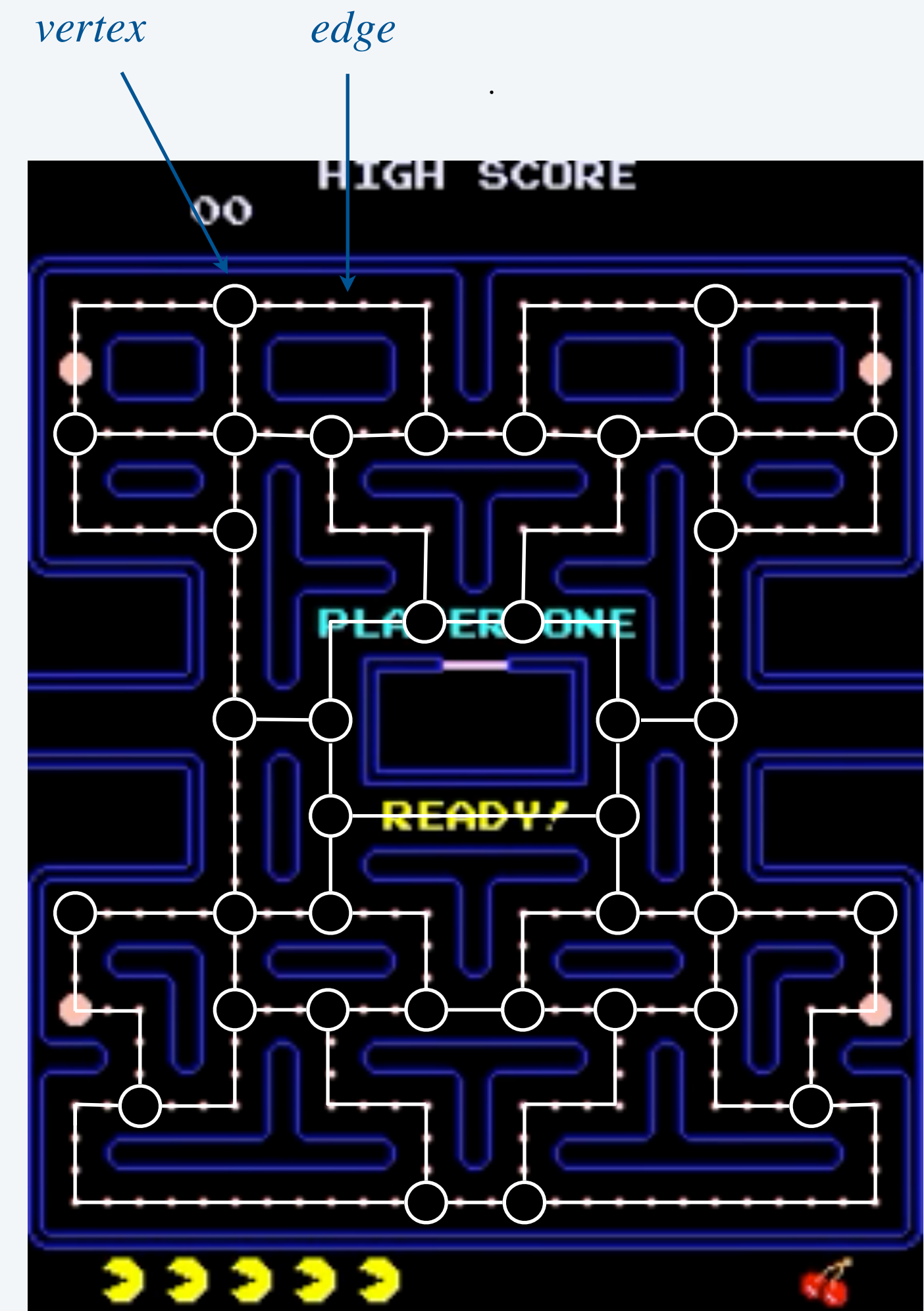
- *introduction*
- *graph representation*
- *depth-first search*
- *path finding*
- *undirected graphs*

Graphs

Def. A **graph** is a set of **vertices** connected pairwise by **edges**.

Why study graphs and graph algorithms?

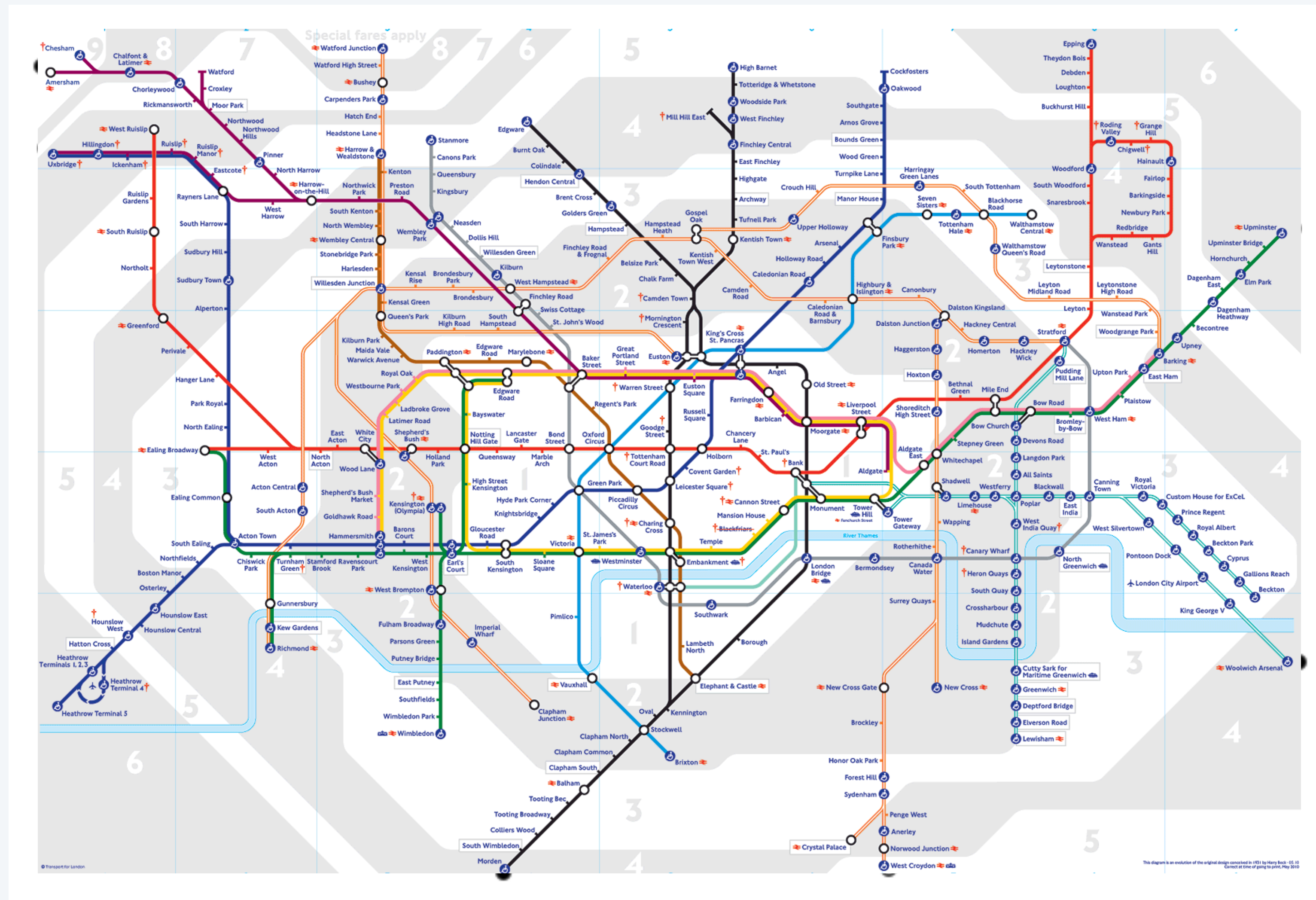
- Hundreds of graph algorithms.
- Thousands of real-world applications.
- Fascinating branch of computer science and discrete math.



Transportation networks

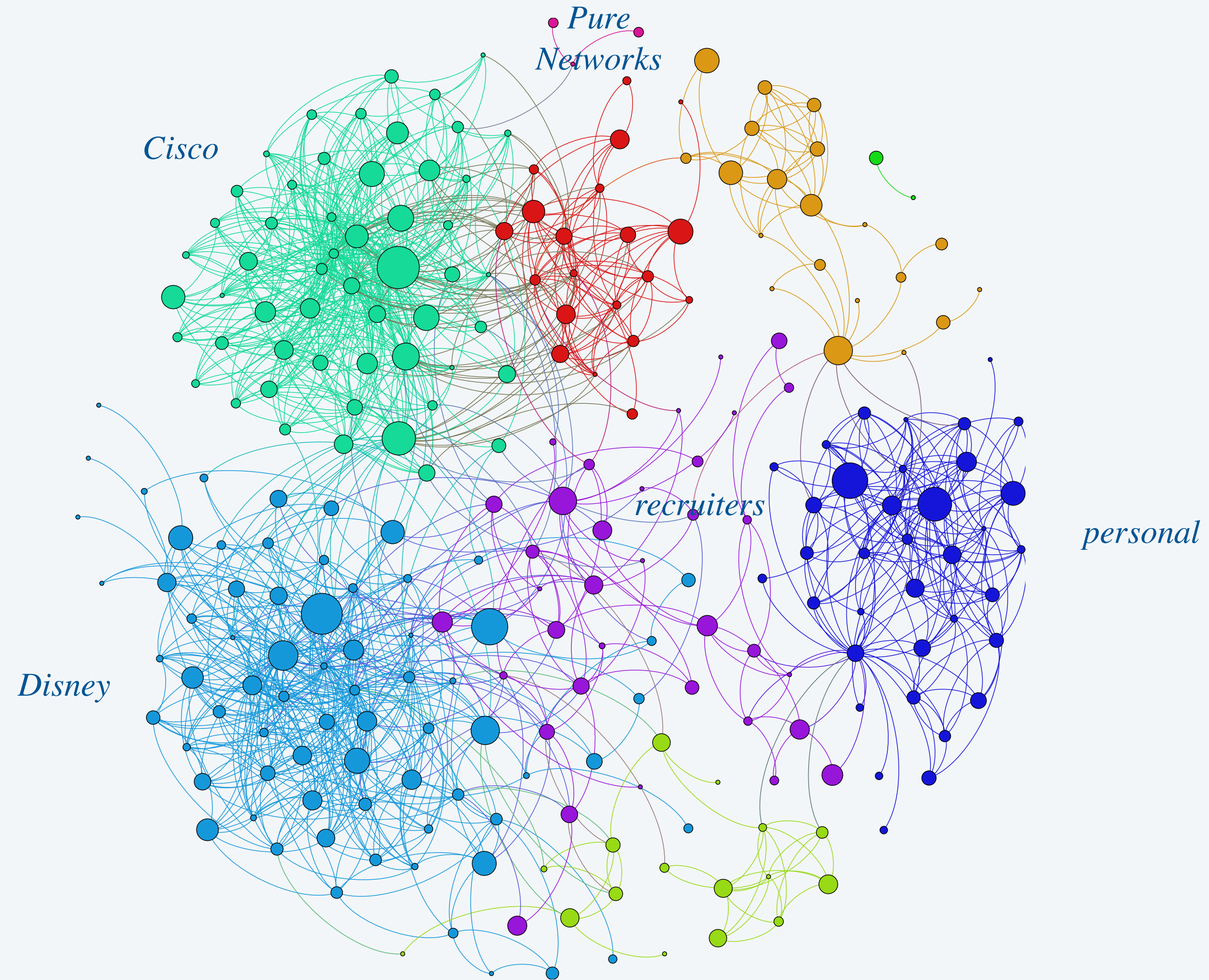


Vertex = subway stop; edge = direct connection.



London Underground (Tube) Map

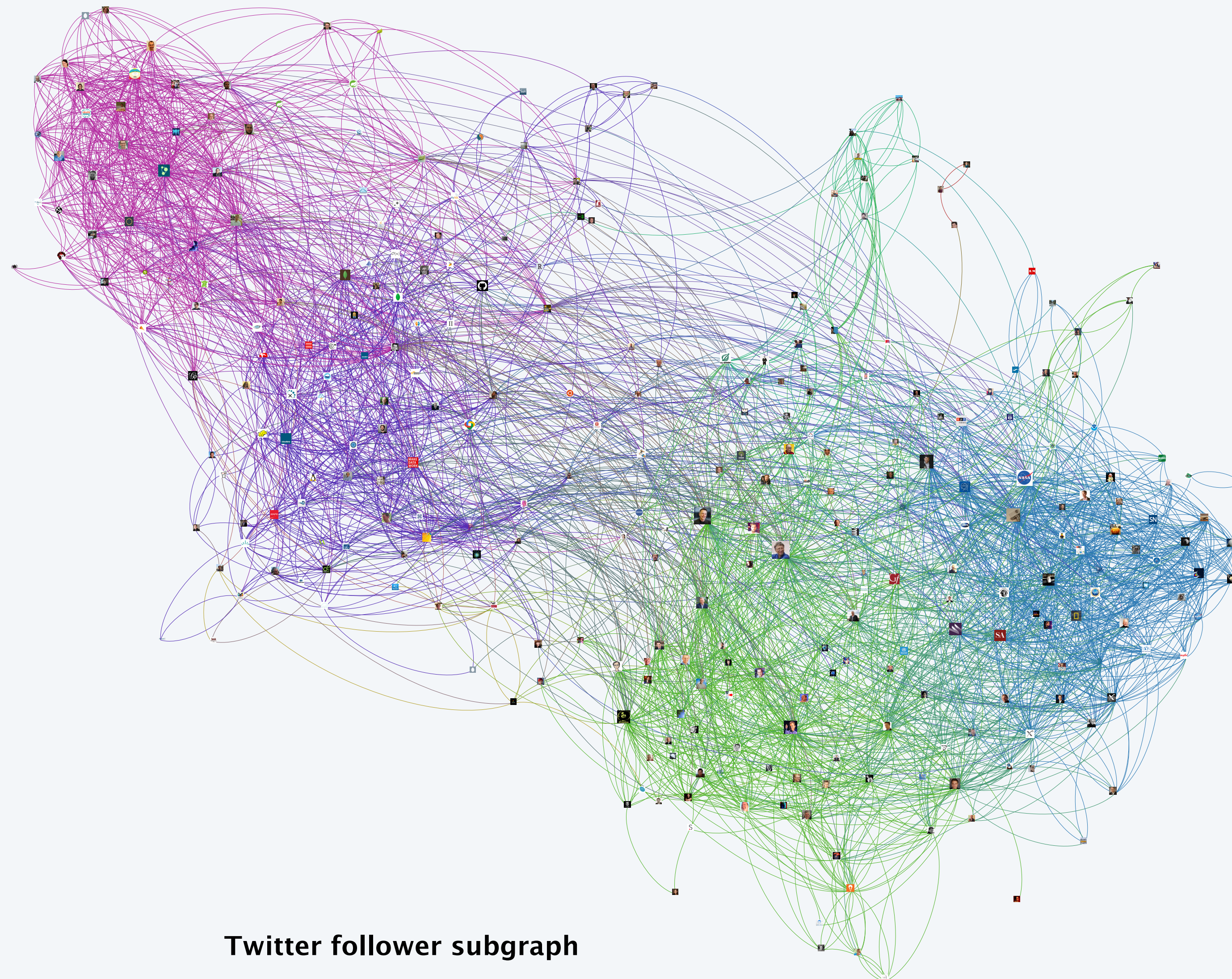
Vertex = person; edge = LinkedIn connection.



LinkedIn social network

Twitter followers

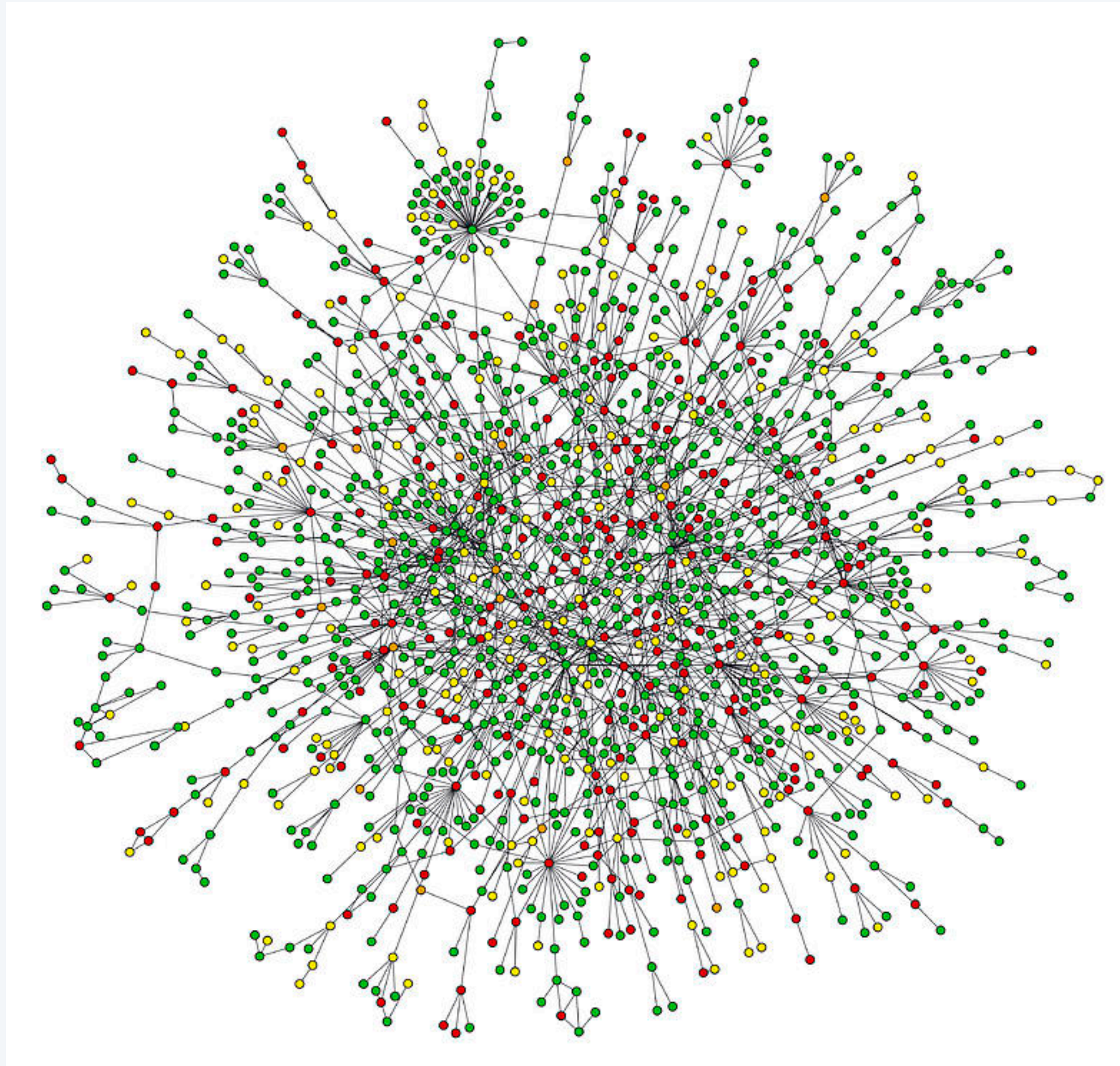
Vertex = Twitter account; edge = Twitter follower.



Twitter follower subgraph

Protein-protein interaction network

Vertex = protein; edge = interaction.



yeast protein interaction map

Graph applications

graph	vertex	edge
<i>cell phone</i>	phone	placed call
<i>infectious disease</i>	person	infection
<i>financial</i>	stock, currency	transactions
<i>game</i>	board position	legal move
<i>transportation</i>	intersection	street
<i>internet</i>	router	fiber optic cable
<i>web</i>	web page	URL link
<i>social relationship</i>	person	friendship
<i>object</i>	object	pointer / reference
<i>protein network</i>	protein	protein–protein interaction
<i>circuit</i>	logic gate	wire
<i>neural network</i>	neuron	synapse

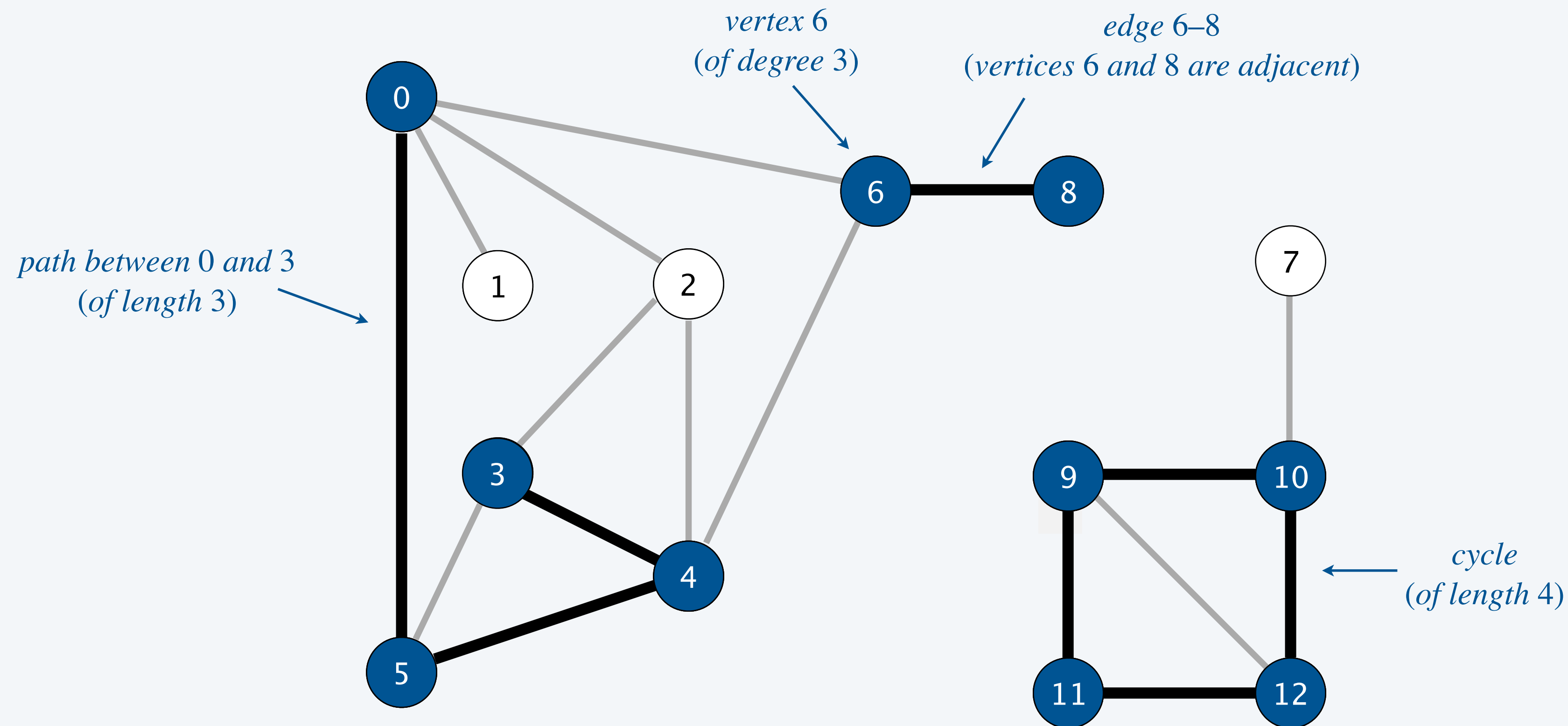
Undirected graph terminology

Def. A **graph** is a set of **vertices** connected pairwise by **edges**.

Def. A **path** is a sequence of vertices connected by edges (with no repeated edges).

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

Def. A **cycle** is a path (with ≥ 1 edge) whose first and last vertices are the same.



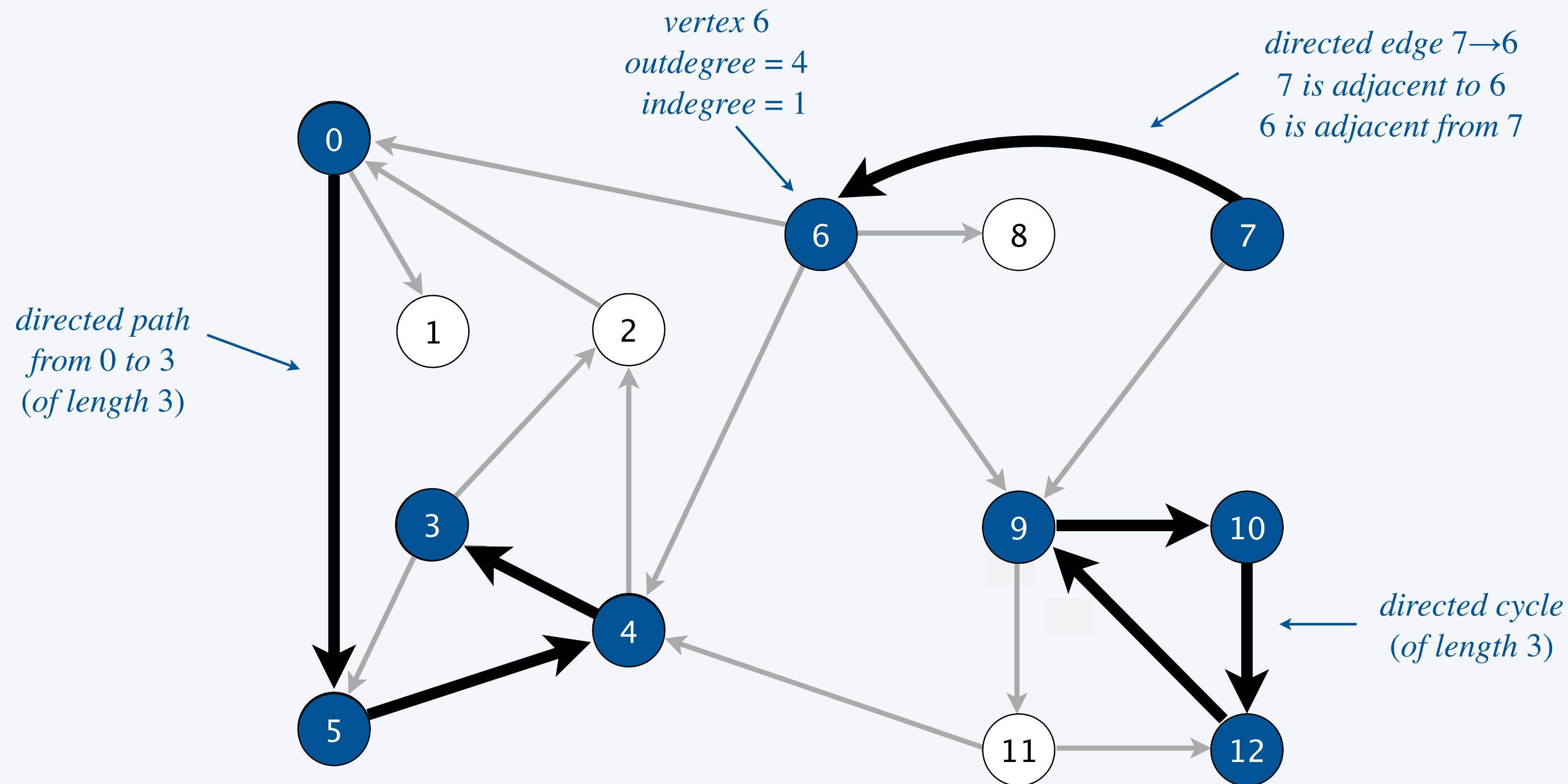
Directed graph terminology

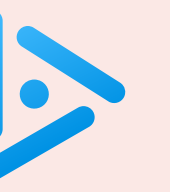
Def. A **digraph** is a set of **vertices** connected pairwise by **directed edges**.

Def. A **directed path** is a 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 .

Def. A **directed cycle** is a directed path (with ≥ 1 edge) whose first and last vertices are the same.





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

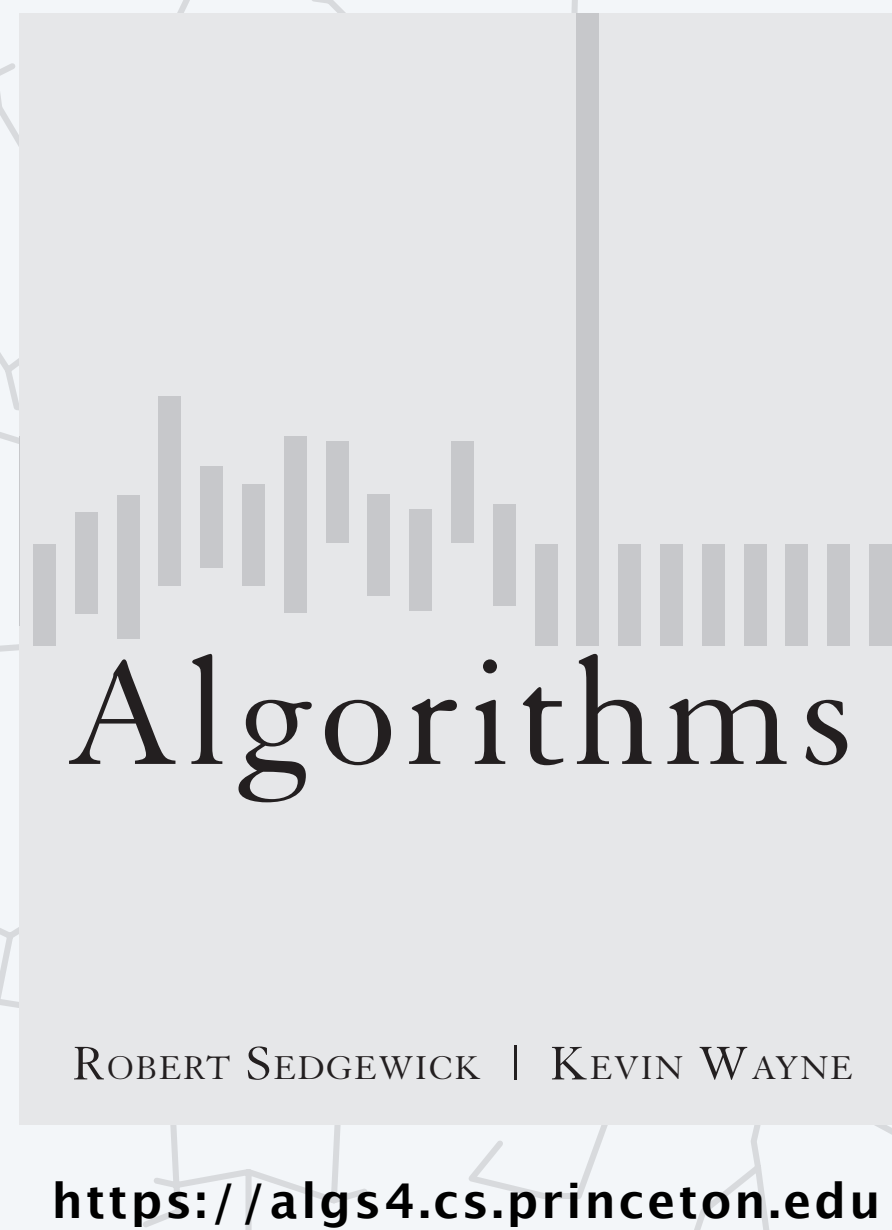
- A. Facebook: vertex = person; edge = friendship.
- B. Web: vertex = webpage; edge = URL link.
- C. Internet: vertex = router; edge = fiber optic cable.
- D. Molecule: vertex = atom; edge = chemical bond.

Some graph-processing problems

	graph problem	description
😊	s–t path	<i>Find a path between s and t.</i>
😊	shortest s–t path	<i>Find a path with the fewest edges between s to t.</i>
😊	cycle	<i>Find a cycle.</i>
😊	Euler cycle	<i>Find a cycle that uses each edge exactly once.</i>
😈	Hamilton cycle	<i>Find a cycle that uses each vertex exactly once.</i>
😊	connected components	<i>Find connected components.</i>
🙋	graph isomorphism	<i>Find an isomorphism between two graphs.</i>
😊	planarity	<i>Draw graph in the plane with no crossing edges.</i>

← also digraph versions

Challenge. Which problems are easy? Difficult? Intractable?



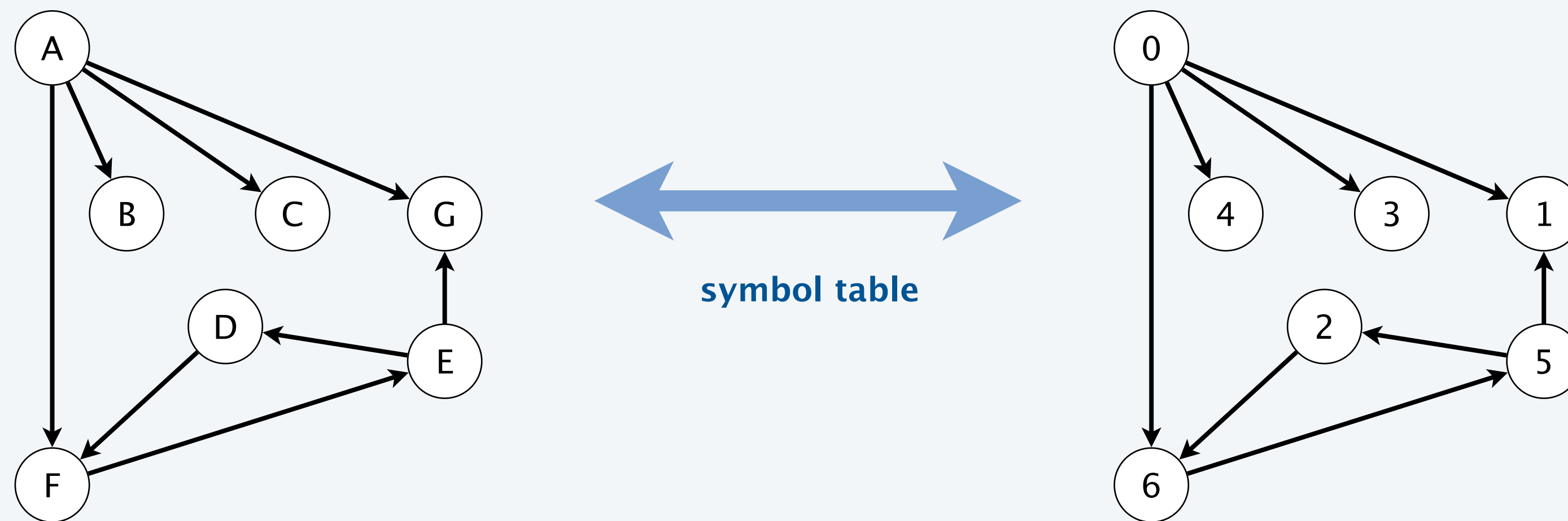
4. GRAPHS AND DIGRAPHS I

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

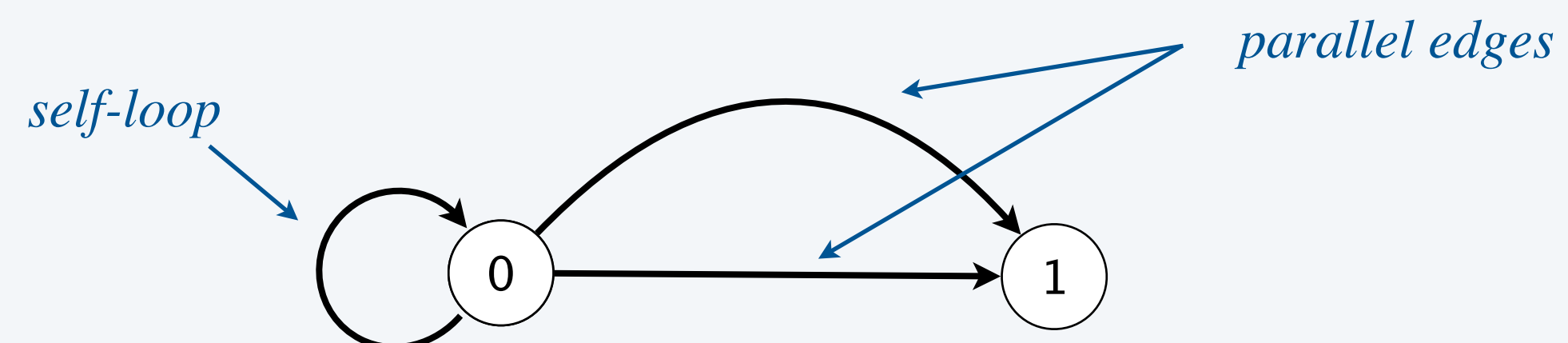
Digraph representation

Vertex representation.

- This lecture: integers between 0 and $V - 1$.
- Real-world 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 $v \rightarrow w$*  *our API allows self-loops and parallel edges*

`Iterable<Integer> adj(int v)` *vertices adjacent from v*


`int V()` *number of vertices*

`Digraph reverse()` *reverse digraph*

`:` `:`

```
// outdegree of vertex v in digraph G
```

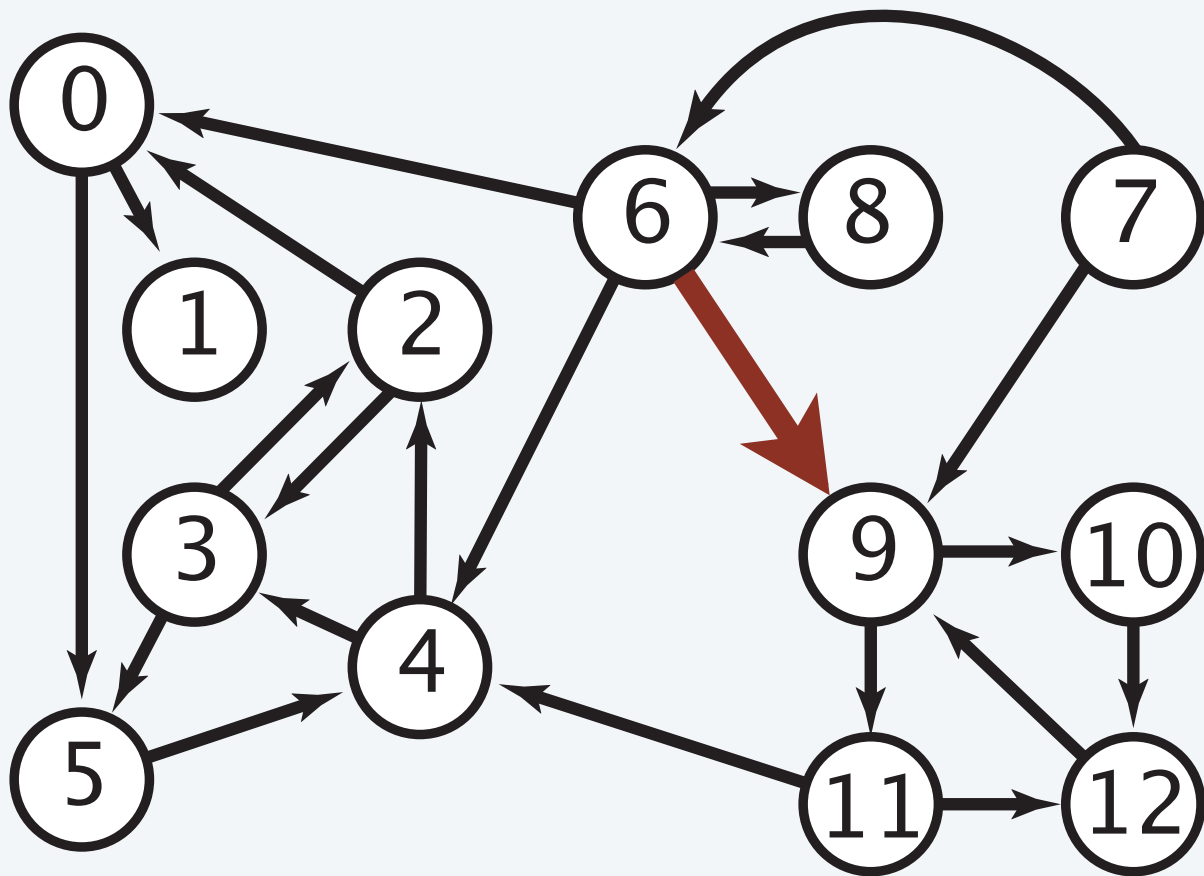
```
public static int outdegree(Digraph digraph, int v) {  
    int count = 0;  
    for (int w : digraph.adj(v))  
        count++;  
    return count;  
}
```

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

Digraph representation: adjacency matrix

Maintain a V -by- V boolean array `adj[][]` with `adj[v][w]` true if and only if $v \rightarrow w$ is an edge.

Memory. Uses $\Theta(V^2)$ space.



`adj[][]`

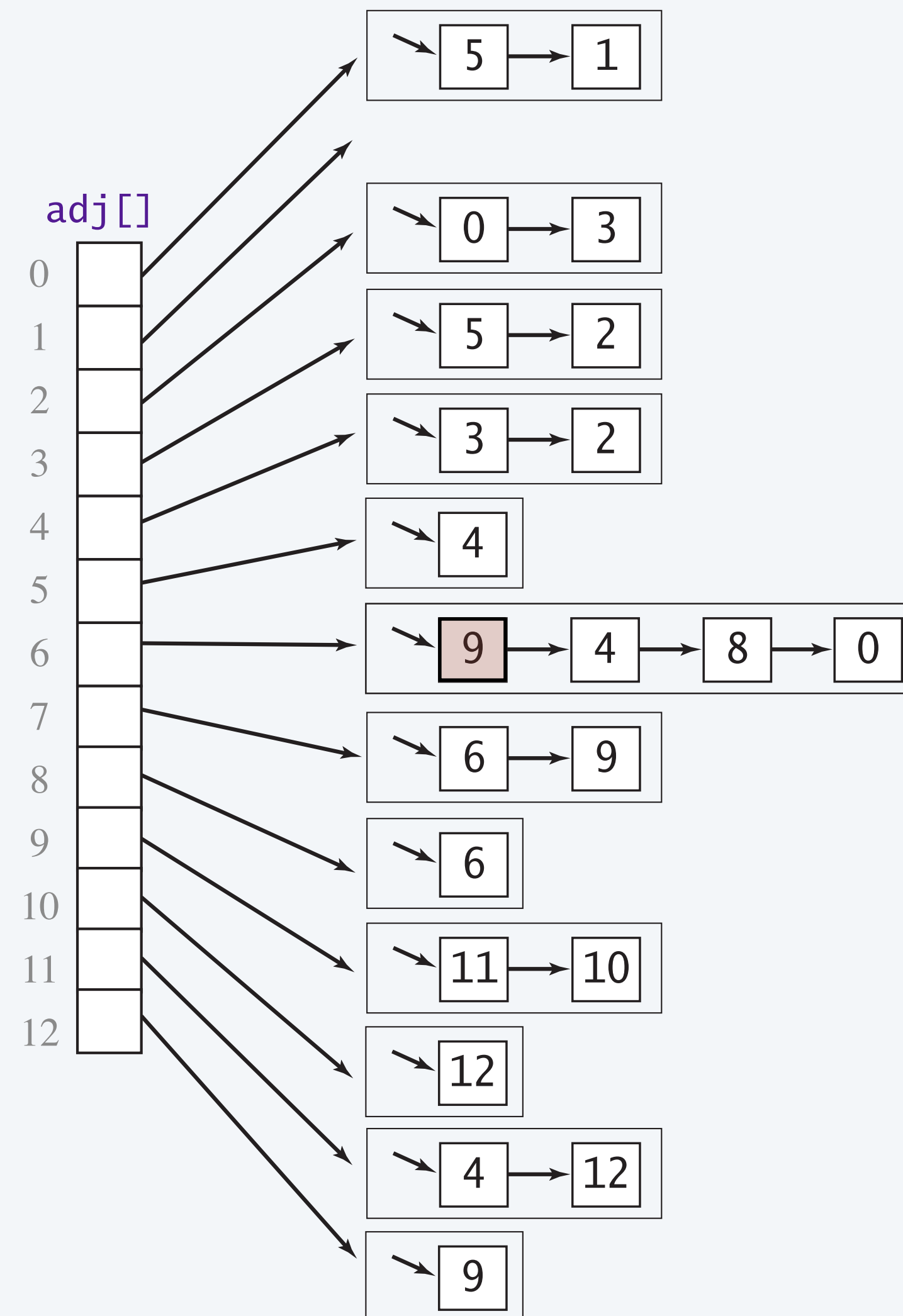
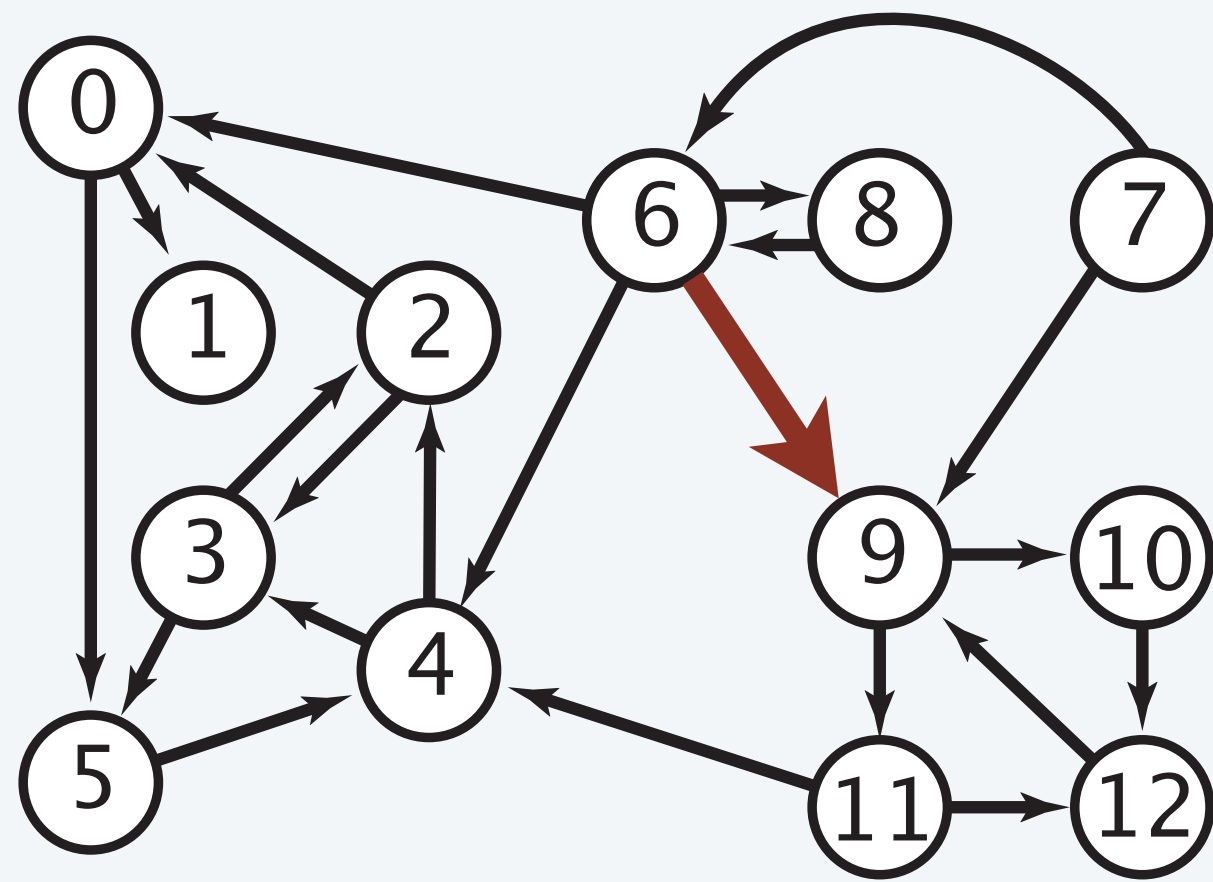
	to												
from	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	0	0	0	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	1	0	0	0	0	0	0	0	0	0
3	0	0	1	0	0	1	0	0	0	0	0	0	0
4	0	0	1	1	0	0	0	0	0	0	0	0	0
5	0	0	0	0	1	0	0	0	0	0	0	0	0
6	0	0	0	0	1	0	0	0	1	1	0	0	0
7	0	0	0	0	0	0	1	0	0	1	0	0	0
8	0	0	0	0	0	0	1	0	0	0	0	0	0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

Digraph representation: adjacency lists

Maintain vertex-indexed array of lists: `adj[v]` contains vertices adjacent from vertex v .

Memory. Uses $\Theta(E + V)$ space.





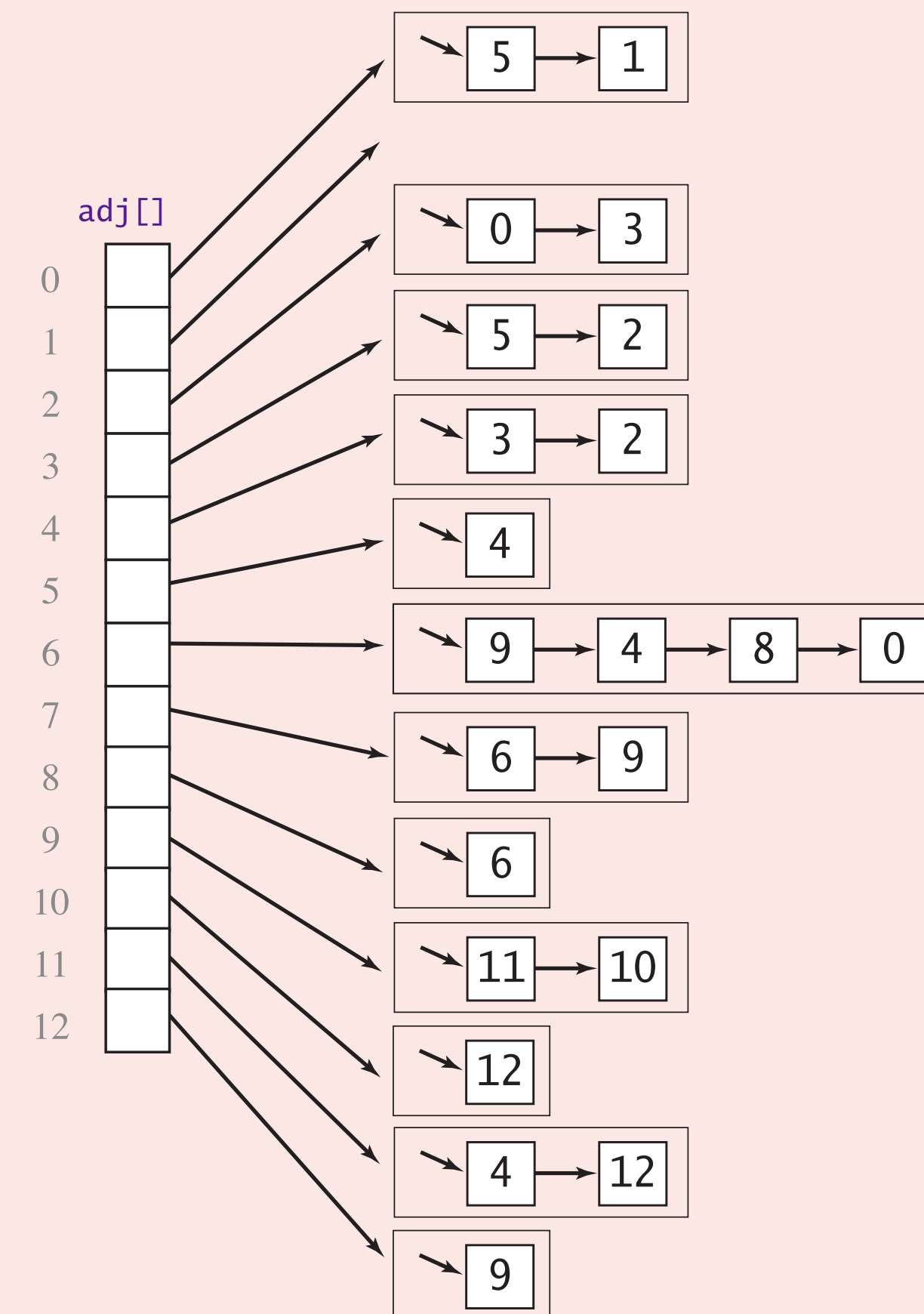
What is the running time of the following code fragment in the worst case?

Assume **adjacency-lists** representation, with $V = \#$ vertices and $E = \#$ edges.

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

print each directed edge once

- A. $\Theta(V)$
- B. $\Theta(E + V)$
- C. $\Theta(V^2)$
- D. $\Theta(EV)$



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**).

↑

$\Theta(V)$ edges

↑

$\Theta(V^2)$ 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 Queue<Integer>[] adj;
```

← adjacency lists
(could also use a stack)

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

← create empty digraph with V vertices

← can't create an array of a parameterized type

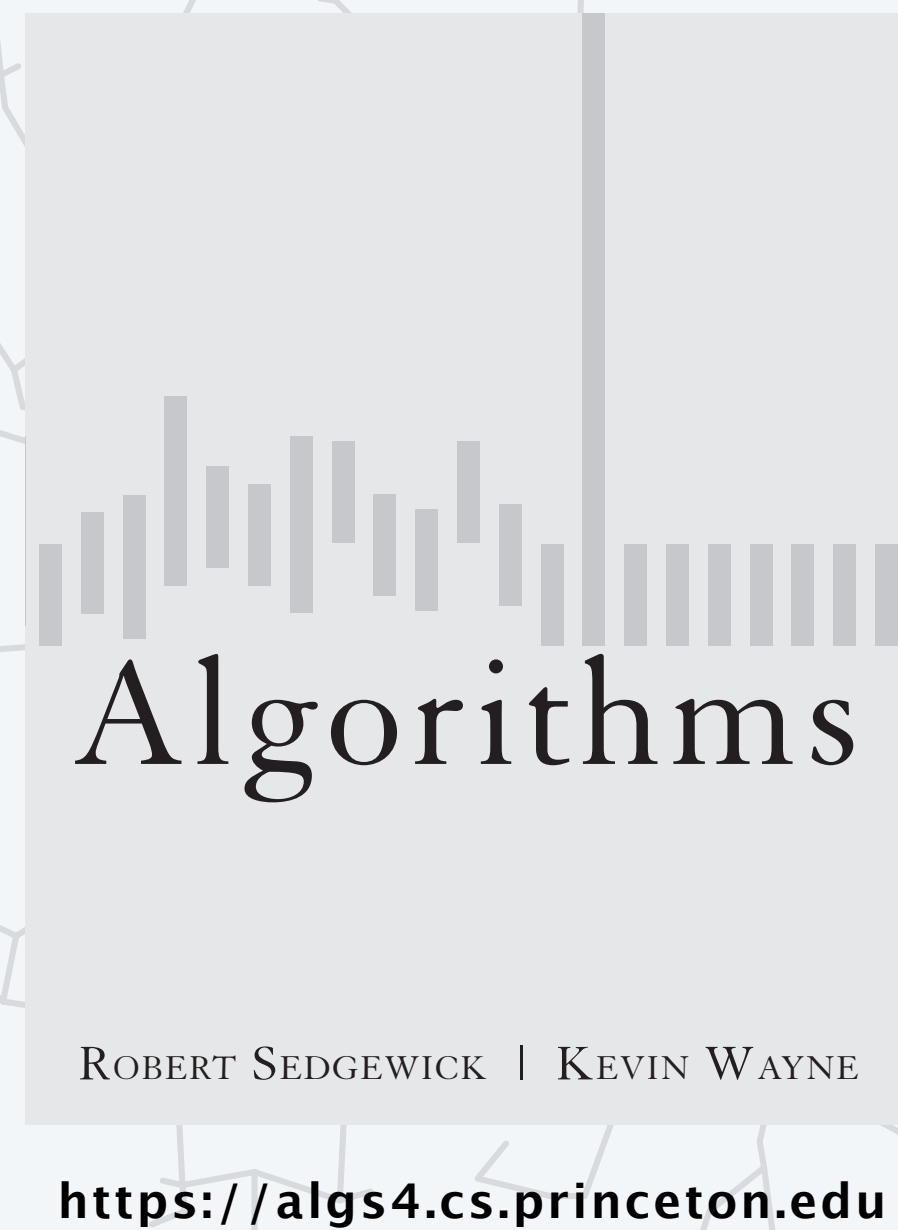
```
    public void addEdge(int v, int w) {  
        adj[v].enqueue(w);  
    }
```

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

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

← iterate over vertices adjacent from v

```
}
```

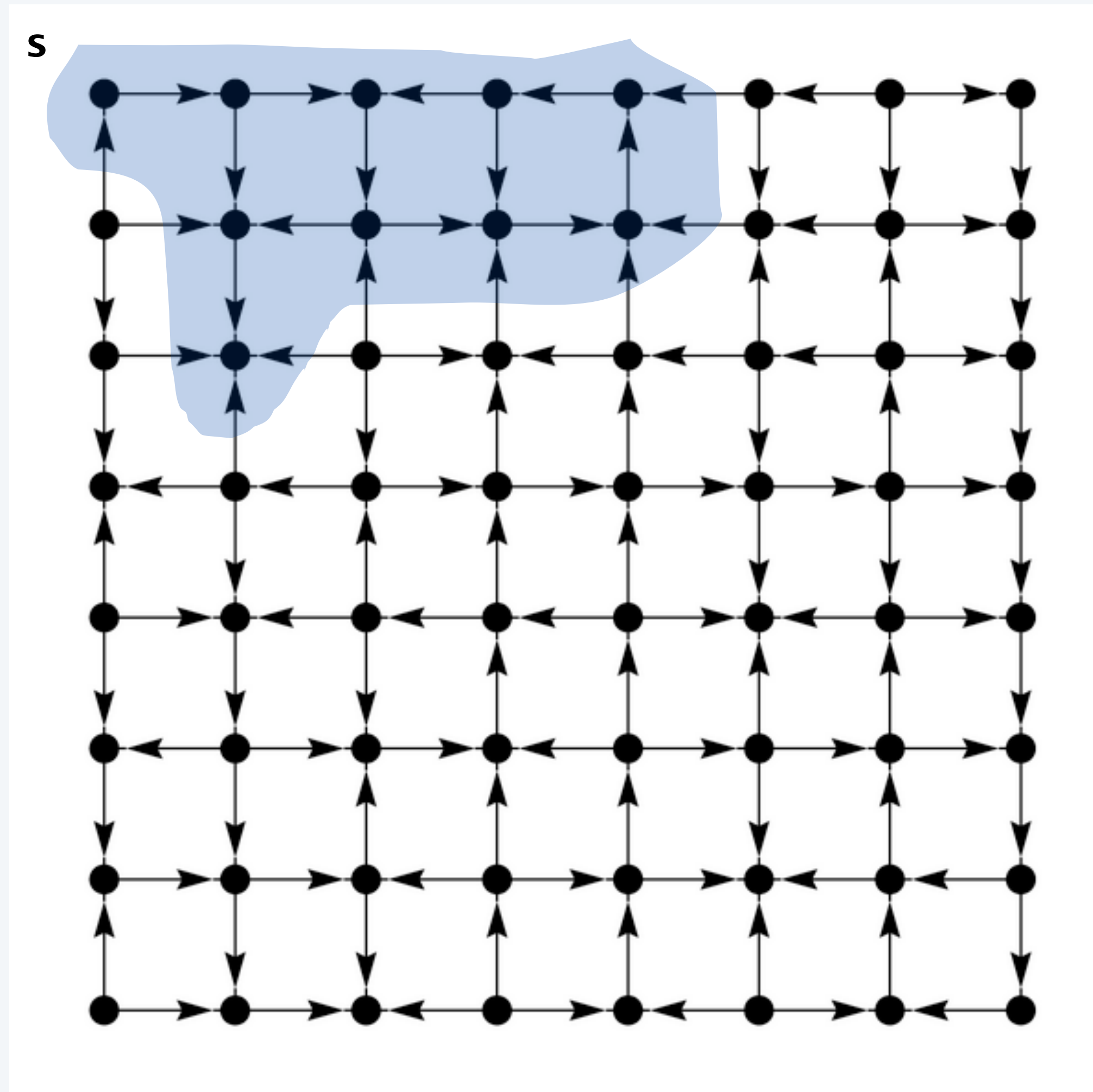


4. GRAPHS AND DIGRAPHS I

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

Reachability problem in a digraph

Reachability problem. Given a digraph G and vertex s , find all vertices **reachable** from s .



Reachability problem in a digraph

Reachability problem. Given a digraph G and vertex s , find all vertices **reachable** from s .

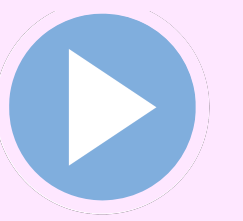
Depth-first search. A systematic method to explore all vertices reachable from s .

DFS (to visit a vertex v)

Mark vertex v .

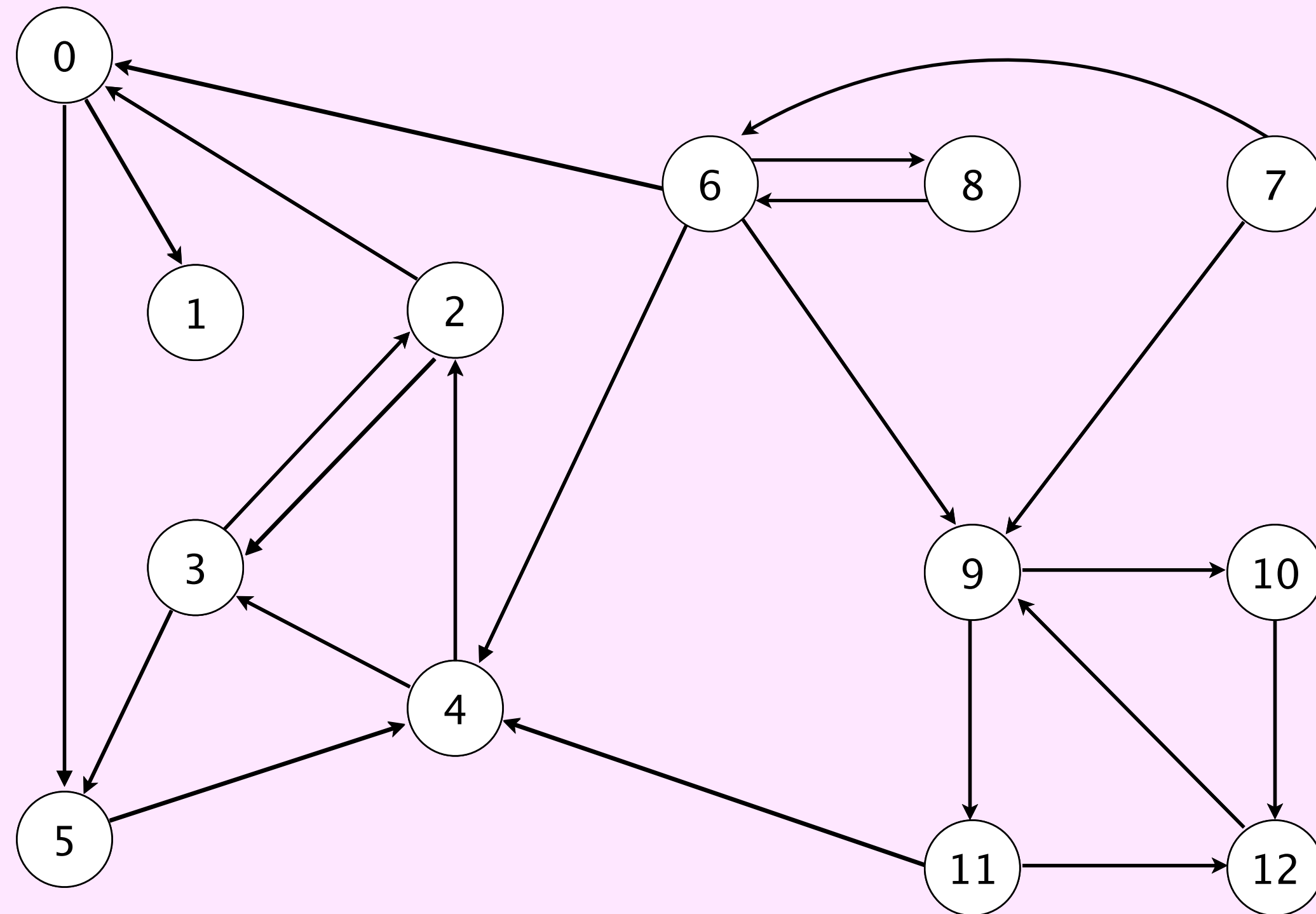
Recursively visit all unmarked
vertices w adjacent from v .

Depth-first search (in a digraph) demo



To visit a vertex v :

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



a directed graph

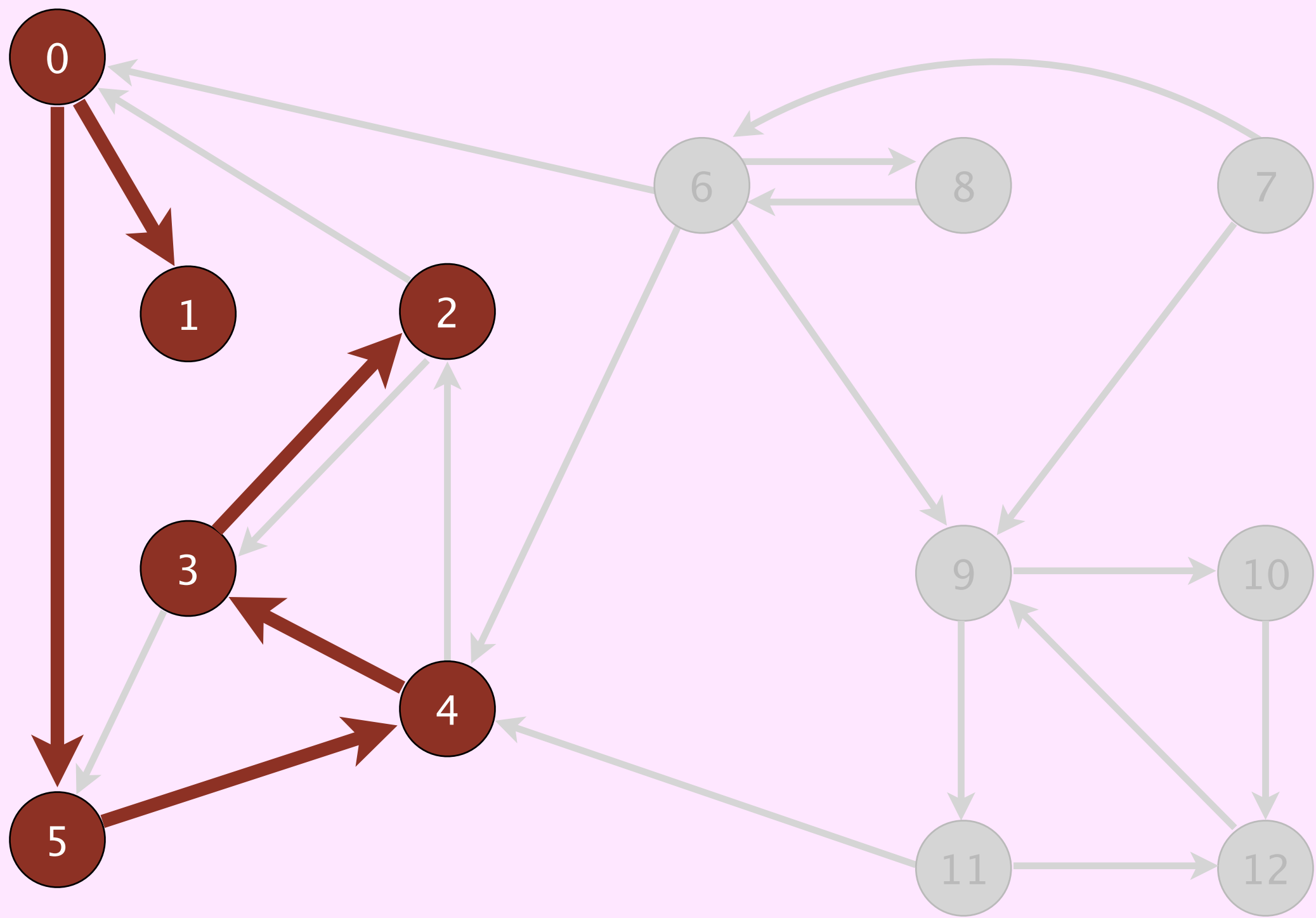
4→2
2→3
3→2
6→0
0→1
2→0
11→12
12→9
9→10
9→11
8→9
10→12
11→4
4→3
3→5
6→8
8→6
5→4
0→5
6→4
6→9
7→6

Depth-first search (in a digraph) 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*

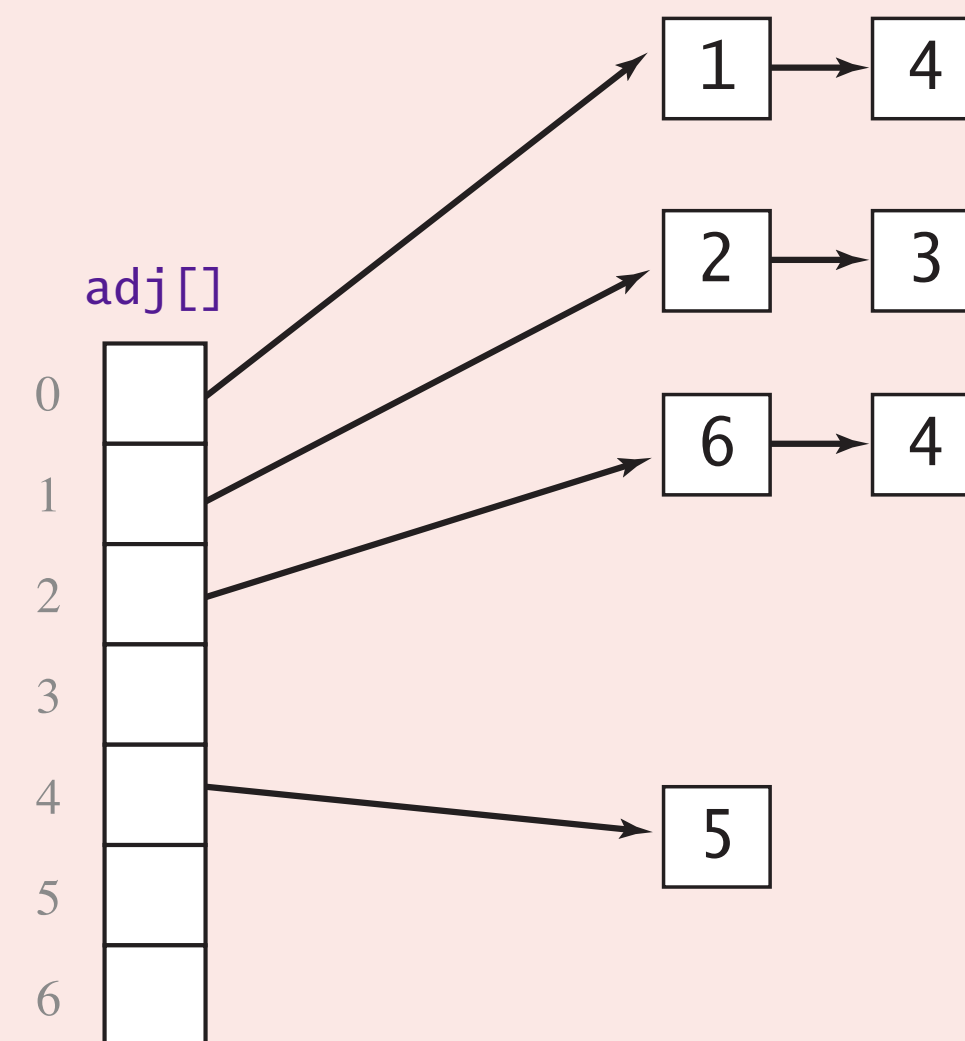
reachable from 0



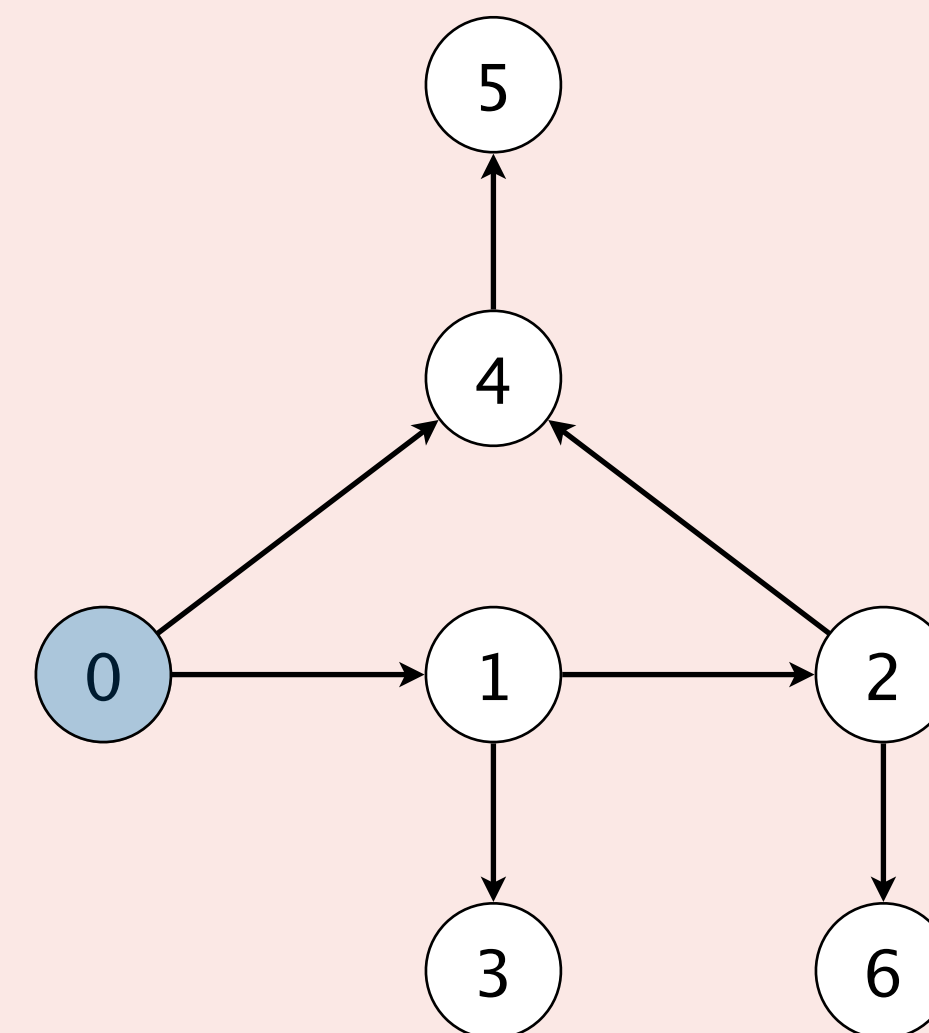
Run DFS using the given adjacency-lists representation of digraph G , starting at vertex 0. In which order is `dfs(digraph, v)` called?

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;
```

← *marked[v] = true if v is reachable from s*

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

← *constructor marks vertices reachable from s*

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

← *recursive DFS does the work*

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


← *is v reachable from s ?*

```
}
```

Depth-first search: analysis

Proposition. DFS uses $\Theta(V)$ extra space (not including the digraph itself).

Pf.

- The `marked[]` array uses $\Theta(V)$ space.
- The function-call stack uses $O(V)$ space.  *dfs() called at most once per vertex*

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

Pf.

- Initializing the `marked[]` array takes $\Theta(V)$ time.
- Each vertex is visited at most once.
- Visiting a vertex takes time proportional to its outdegree:

$$\text{outdegree}(v_0) + \text{outdegree}(v_1) + \text{outdegree}(v_2) + \dots = E$$

 *in the worst case,
all vertices are reachable from s*

Note. If all vertices are reachable from s , then $E \geq V - 1$ and running time simplifies to $\Theta(E)$.



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 digraph, int v) {  
    marked[v] = true;  
    for (int w : digraph.adj(v))  
        if (!marked[w])  
            dfs(digraph, w);  
    marked[v] = true;  
}
```


Reachability application: program control-flow analysis

Every program is a digraph.

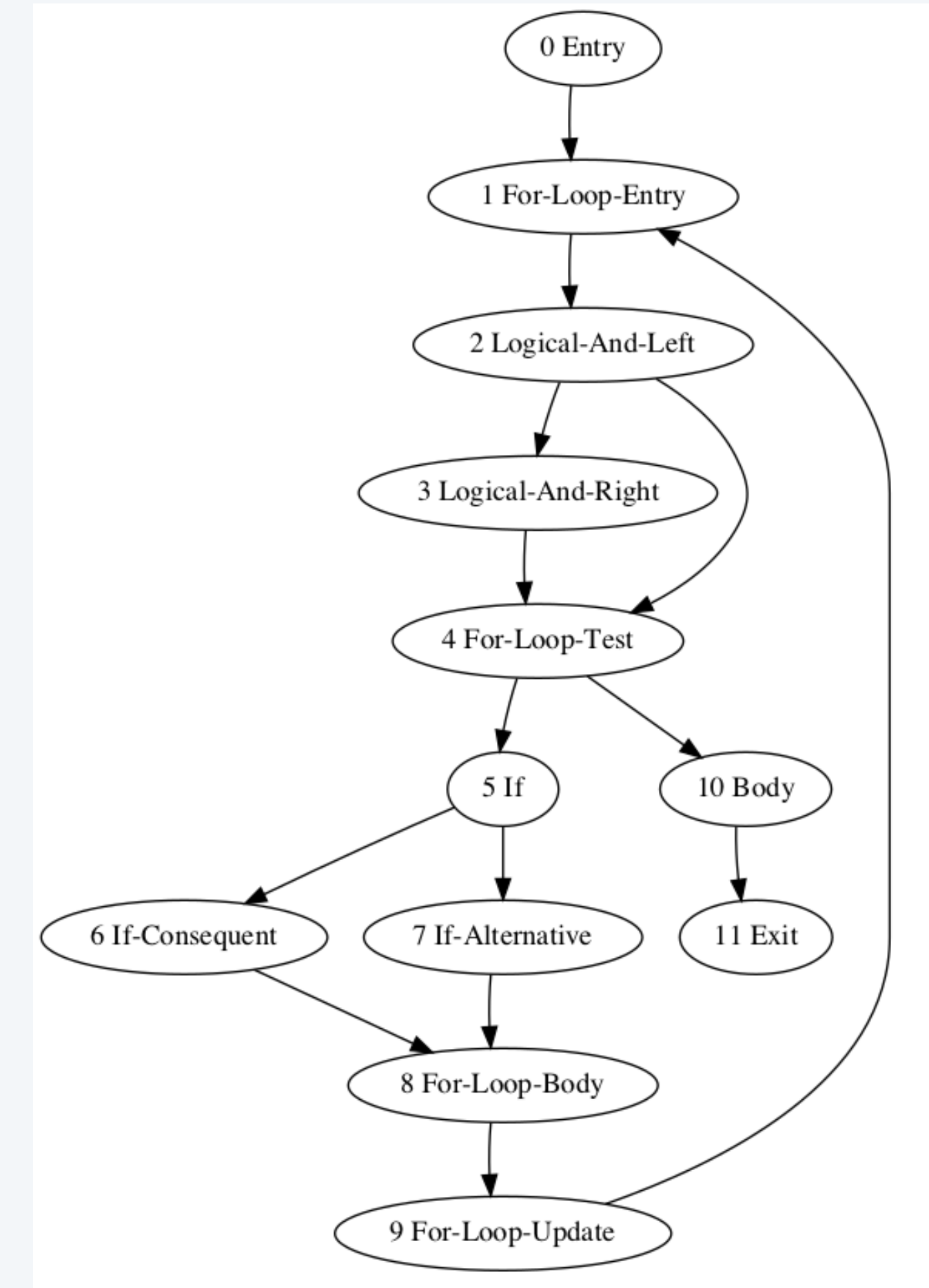
- Vertex = basic block of instructions (straight-line program).
- Edge = jump.

Dead-code elimination.

Find (and remove) unreachable code.

Infinite-loop detection.

Determine whether exit is unreachable.



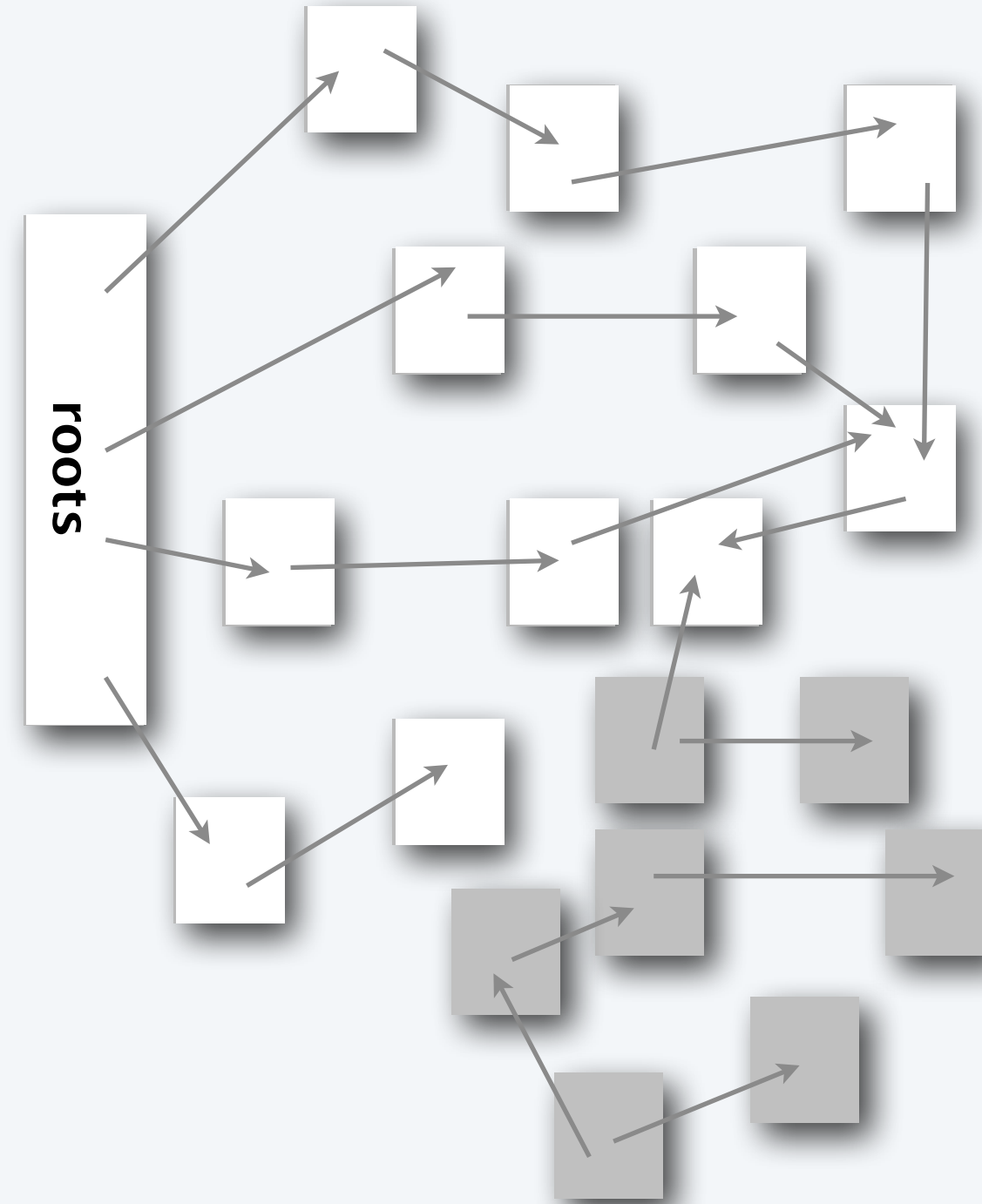
Reachability application: mark-sweep garbage collector

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).

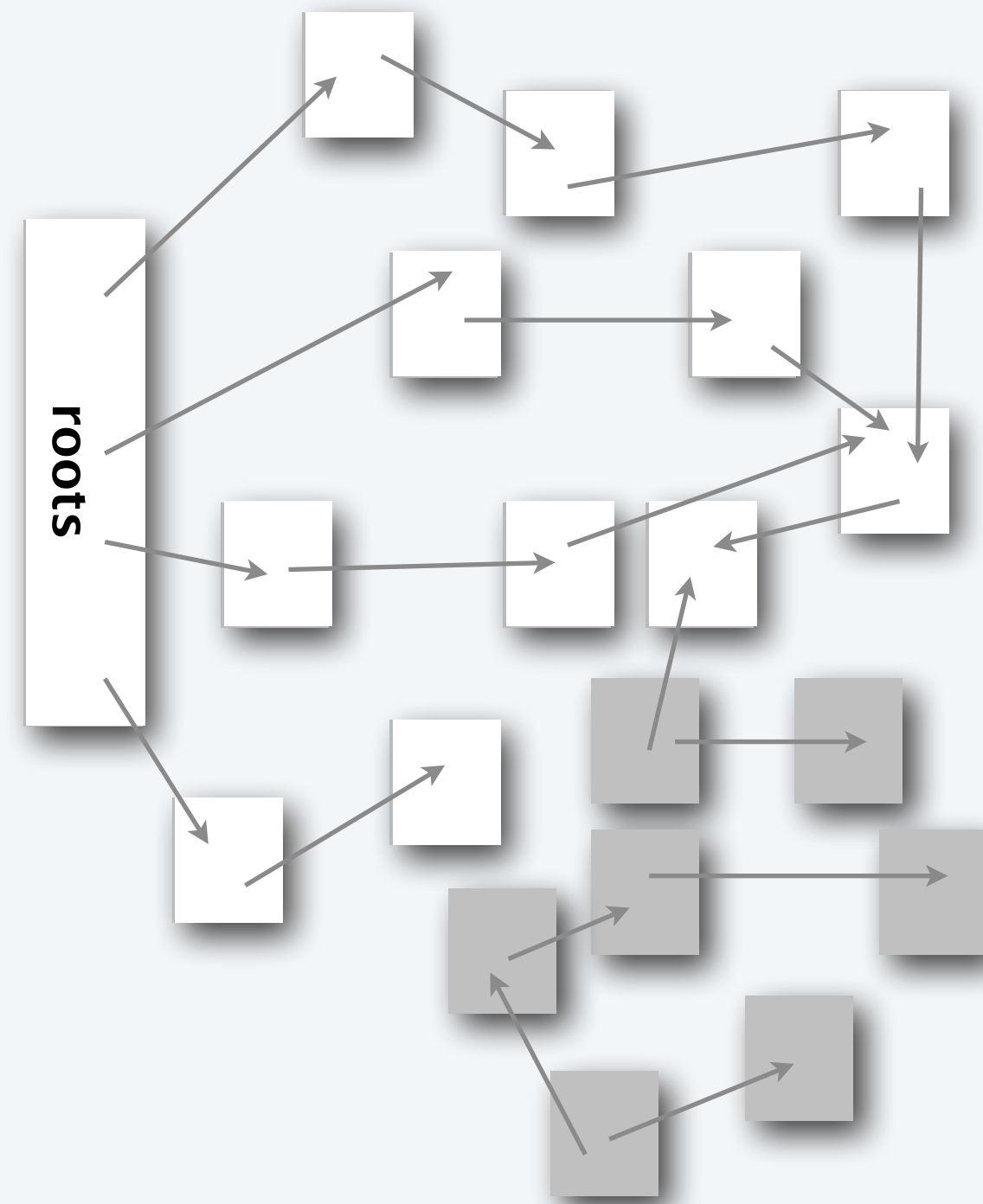


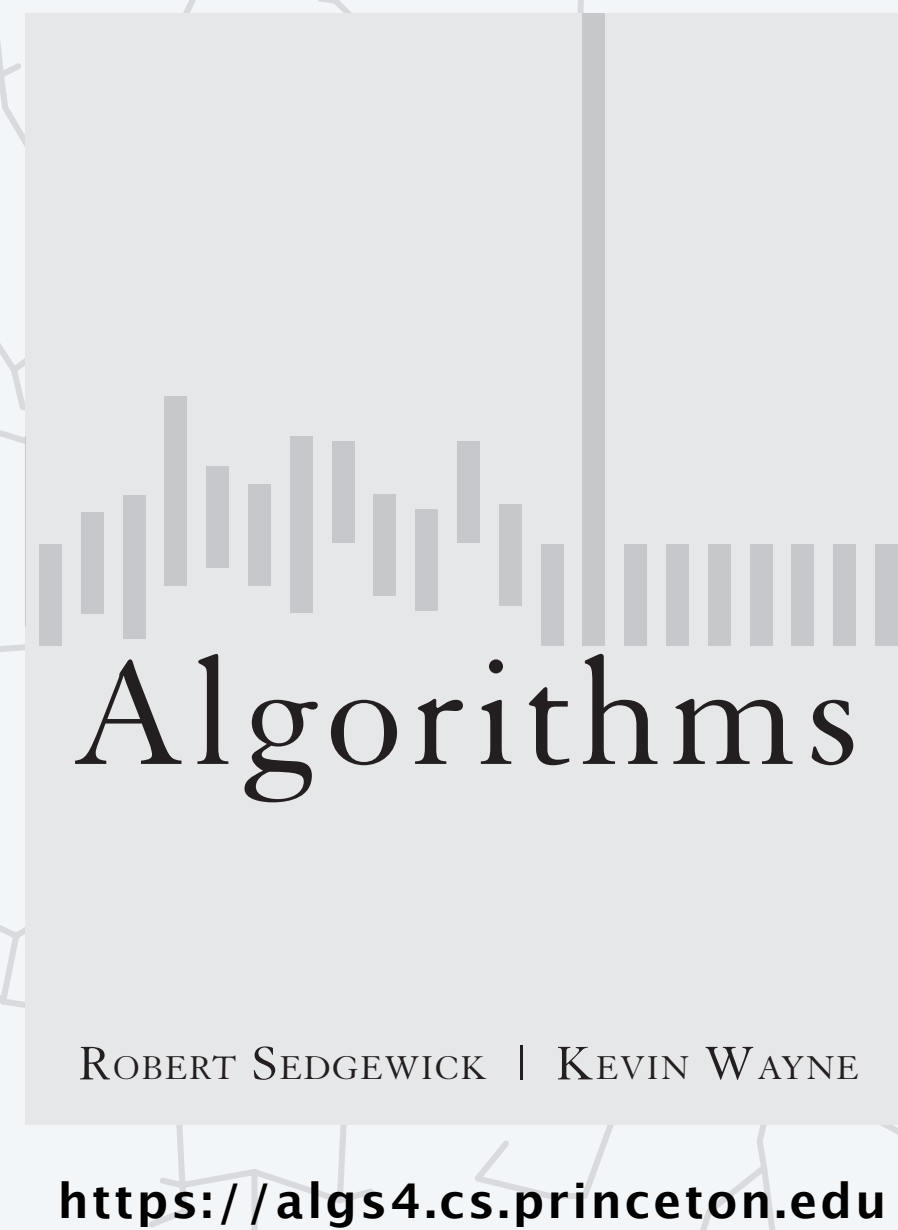
Reachability application: mark-sweep garbage collector

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 one extra mark bit per object (plus DFS function-call stack).





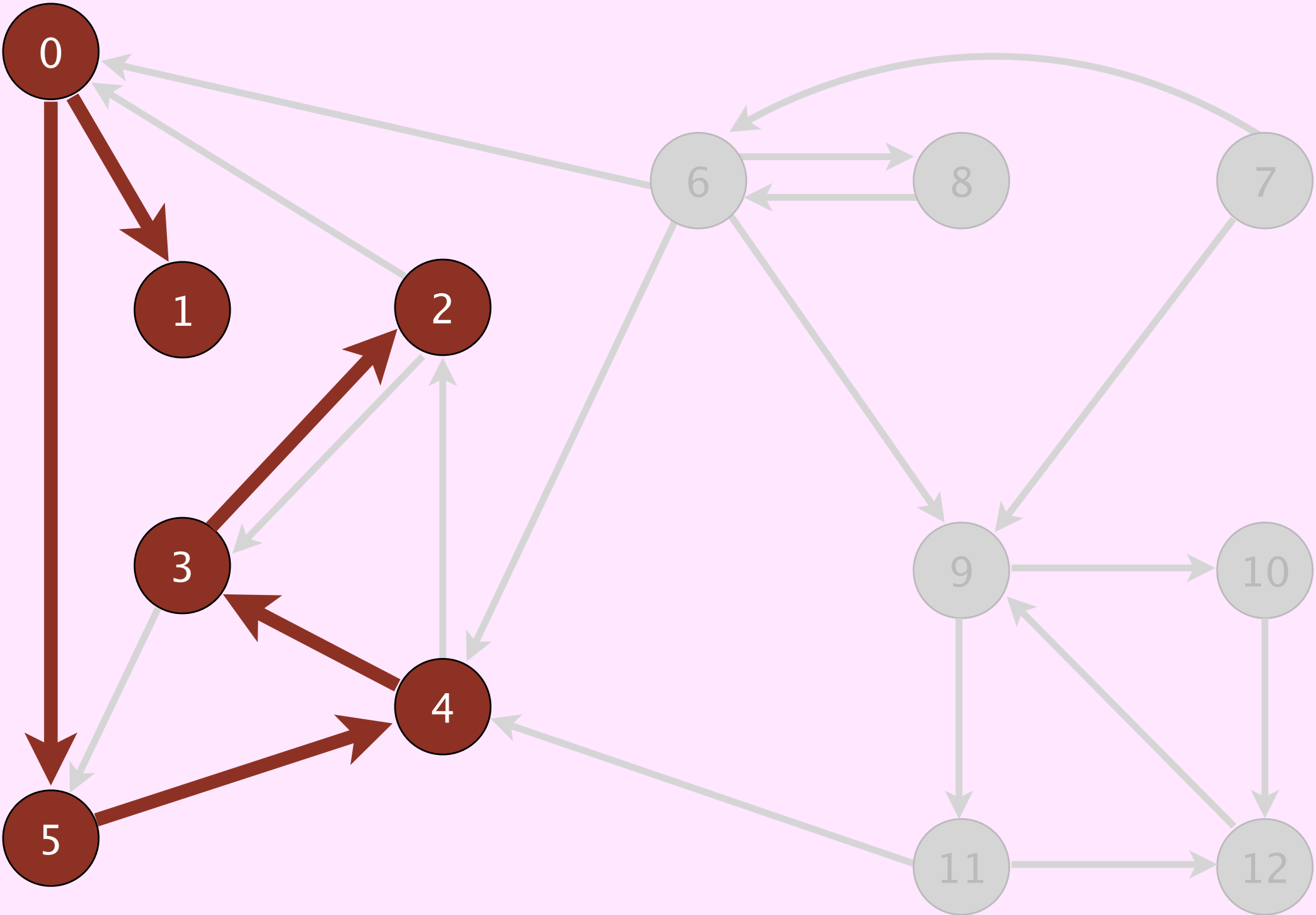
4. GRAPHS AND DIGRAPHS I

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



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

Solution. Use **parent-link representation**.



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	-

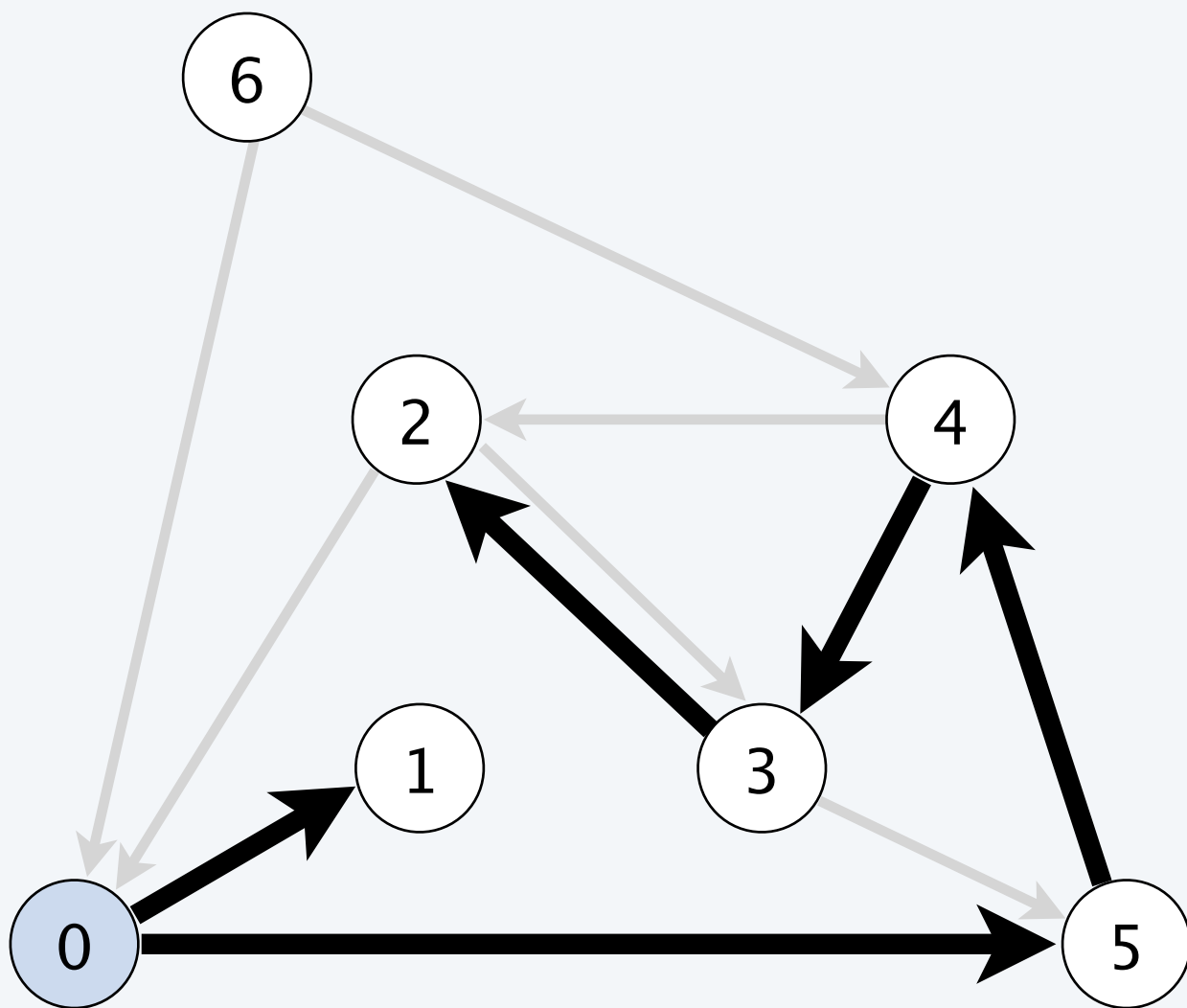
← *parent-link representation
of paths from vertex 0*

vertices reachable from 0
(and directed paths)

Depth-first search: path finding

Parent-link representation of paths from vertex s .

- Maintain an integer array `edgeTo[]`.
- Interpretation: `edgeTo[v]` is the next-to-last vertex on a directed 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;
```

← *edgeTo[v] = previous vertex
on a directed path from s to v*

```
    public DepthFirstDirectedPaths(Digraph digraph, int s) {
```

```
        ...
```

```
        dfs(digraph, s);
```

```
    }
```

```
    private void dfs(Digraph digraph, int v) {
```

```
        marked[v] = true;
```

```
        for (int w : digraph.adj(v)) {
```

```
            if (!marked[w]) {
```

```
                dfs(digraph, w);
```

```
                edgeTo[w] = v;
```

```
            }
```

```
        }
```

```
    }
```

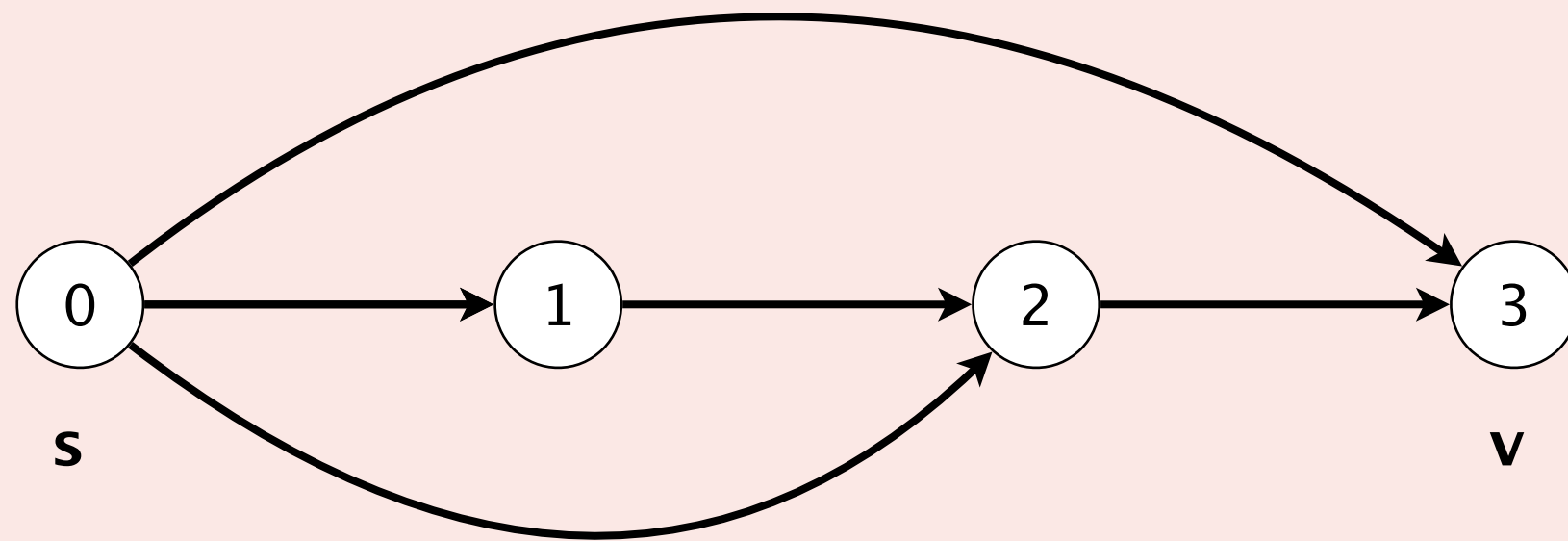
← *v → w is edge that led
to the discovery of w*

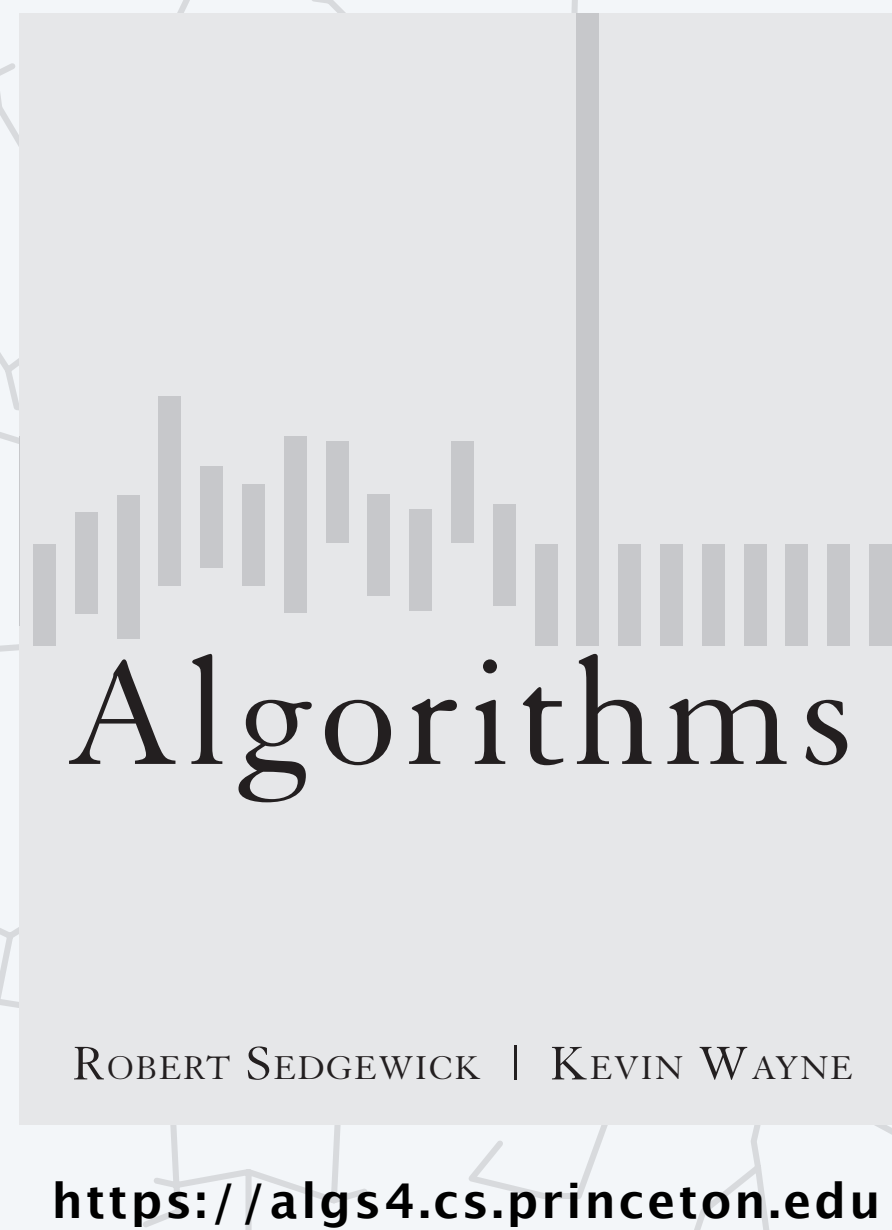
```
}
```



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.





4. GRAPHS AND DIGRAPHS I


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

Problem. Implement flood fill (Photoshop magic wand).



Depth-first search in undirected graphs

Connectivity problem. Given an undirected graph G and vertex s , find all vertices **connected to** s .

Solution. Use DFS.  *but now, for each undirected edge $v-w$:
 v is adjacent with w (and w is adjacent with v)*

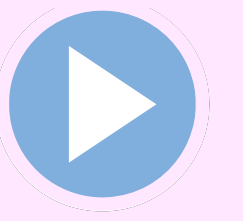
DFS (to visit a vertex v)

Mark vertex v .

Recursively visit all unmarked
vertices w **adjacent with** v .

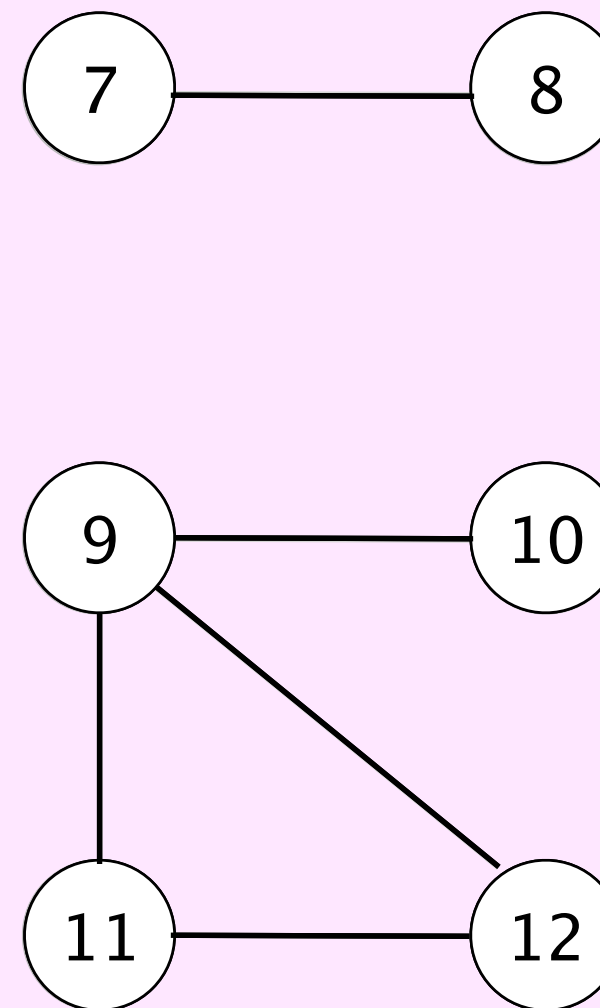
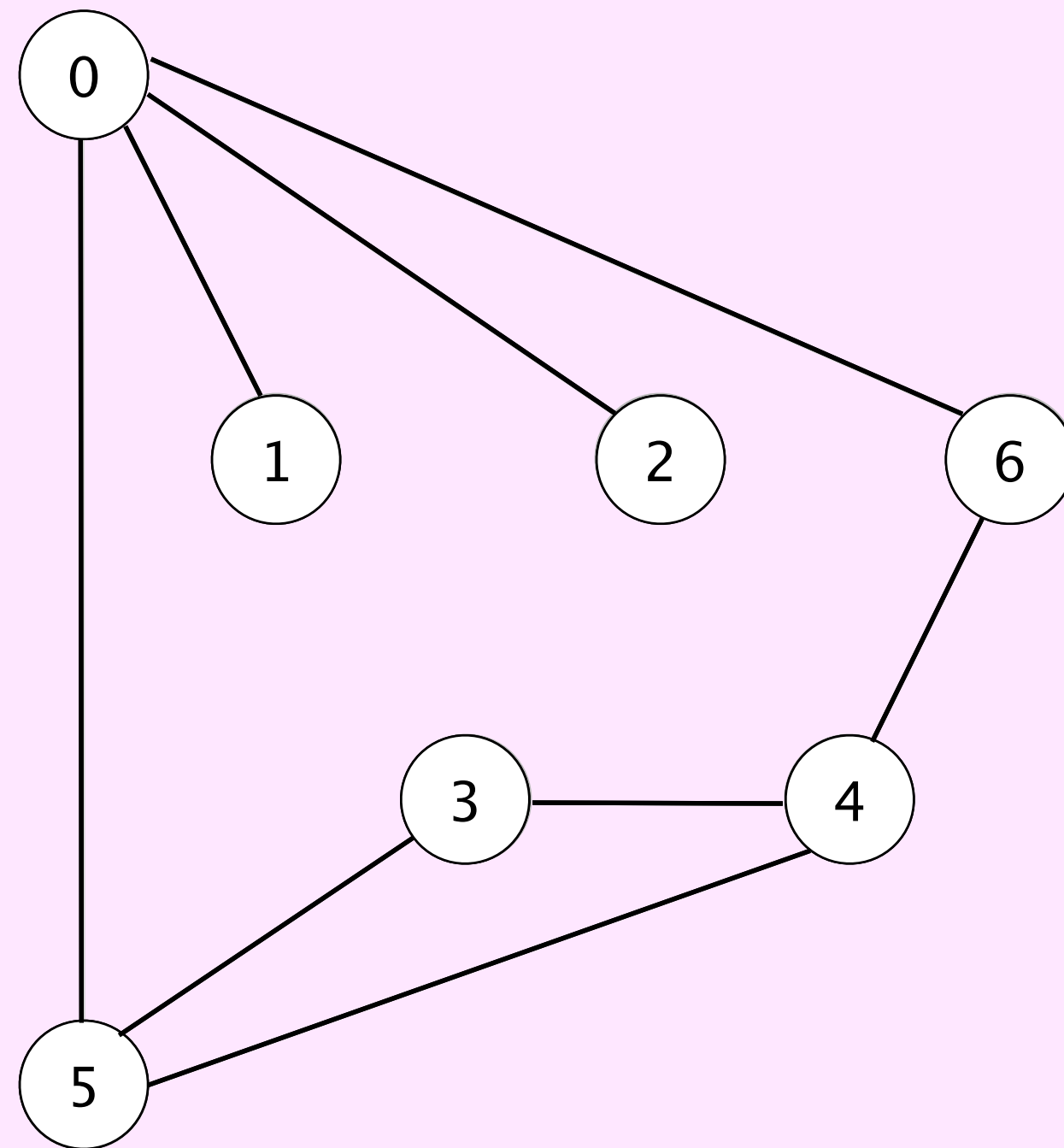
Proposition. DFS marks all vertices connected to s in $\Theta(E + V)$ time in the worst case.

Depth-first search (in an undirected graph) demo



To visit a vertex v :

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



tinyG.txt

$V \rightarrow$ 13
13 $\leftarrow E$
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

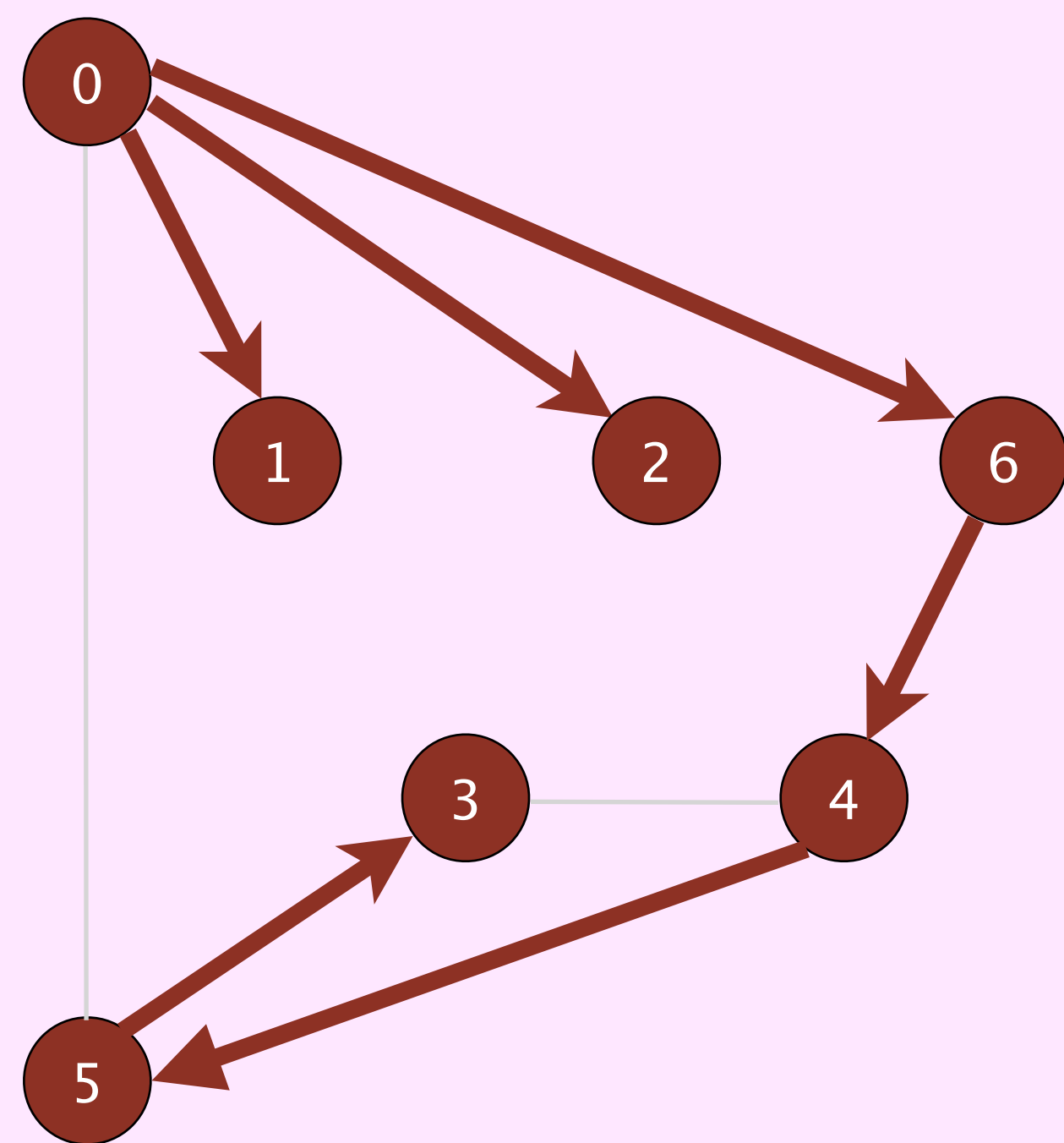
graph G

Depth-first search (in an undirected graph) demo

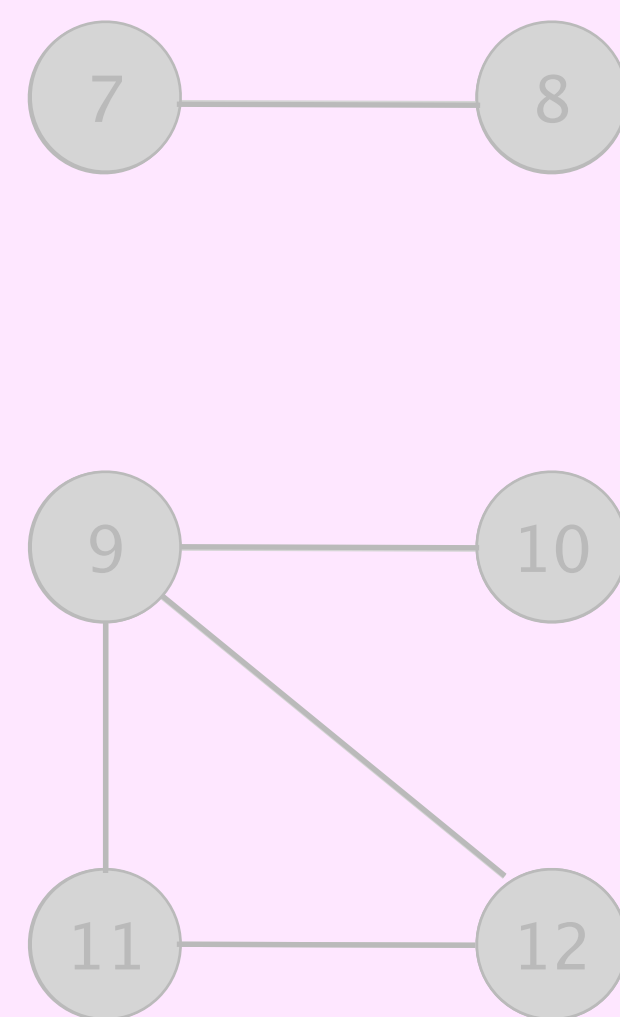


To visit a vertex v :

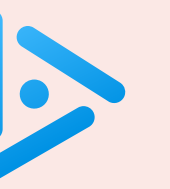
- Mark vertex v .
- Recursively visit all unmarked vertices adjacent with v .



vertices connected to 0
(and associated paths)

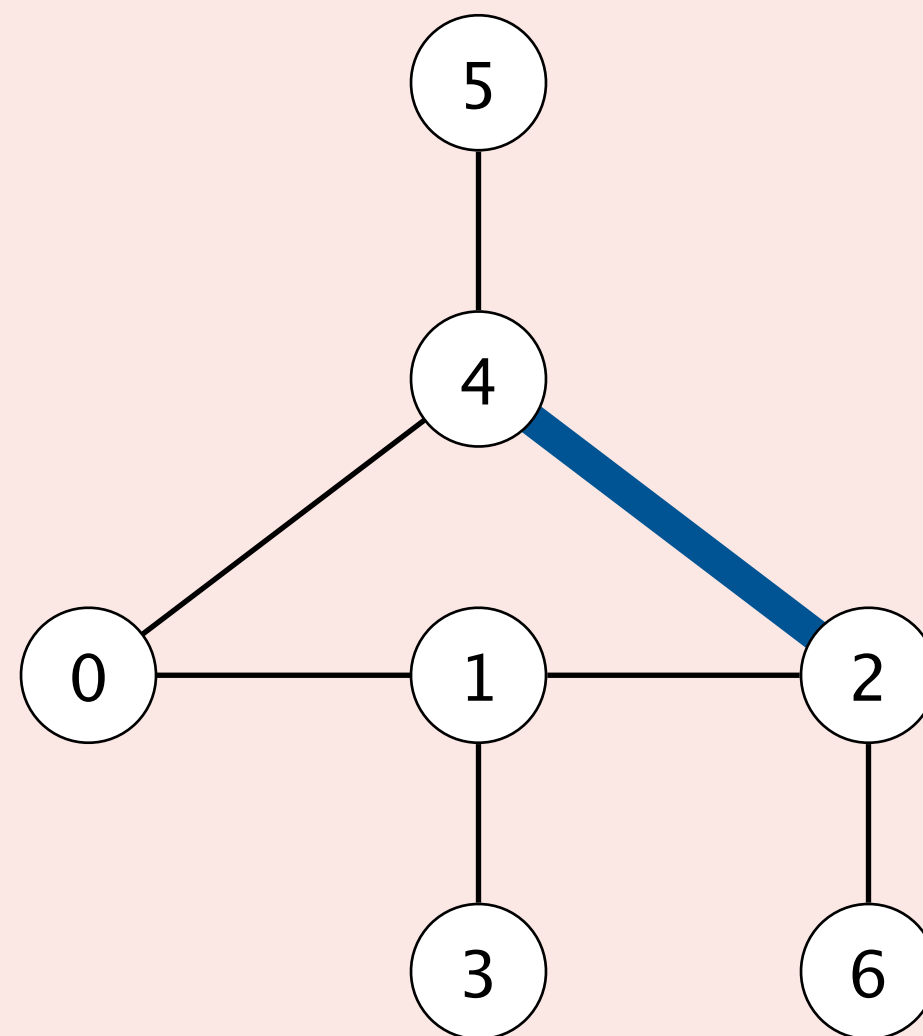


v	marked[]	edgeTo[]
0	T	-
1	T	0
2	T	0
3	T	5
4	T	6
5	T	4
6	T	0
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-



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.



Directed graph representation (review)

```
public class Digraph {
```

```
    private final int V;  
    private Queue<Integer>[] adj;
```

← *adjacency lists*

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

← *create empty digraph with V vertices*

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

← *add edge $v \rightarrow w$*

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

← *iterate over vertices adjacent from v*

```
}
```

Undirected graph representation

```
public class Graph {
```

```
    private final int V;  
    private Queue<Integer>[] adj;
```

← *adjacency lists*

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

← *create empty graph with V vertices*

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

← *add edge v-w*

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

← *iterate over vertices adjacent with v*

```
}
```


Depth-first search (in directed graphs)

```
public class DirectedDFS {
```

```
    private boolean[] marked;
```

← *marked[v] = true if v is reachable from s*

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

← *constructor marks vertices reachable from s*

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

← *recursive DFS does the work*

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

← *is v reachable from s ?*

```
}
```

Depth-first search (in undirected graphs)

```
public class DepthFirstSearch {
```

```
    private boolean[] marked;
```

← *marked[v] = true if vertices s and v are connected*

```
    public DirectedDFS(Graph graph, int s) {  
        marked = new boolean[graph.V()];  
        dfs(graph, s);  
    }
```

← *constructor marks vertices connected with s*

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

← *recursive DFS does the work*

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

← *are vertices s and v connected?*

```
}
```

Depth-first search summary

DFS enables direct solution of several elementary 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 is also core of solution to more advanced problems.

- Euler cycle.
- Biconnectivity.
- 2-satisfiability.
- Planarity testing.
- Strong components.
- Nonbipartite matching.
- ...

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

DEPTH-FIRST SEARCH AND LINEAR GRAPH ALGORITHMS*

ROBERT TARJAN†

Abstract. The value of depth-first search or “backtracking” as a technique for solving problems is illustrated by two examples. An improved version of an algorithm for finding the strongly connected components of a directed graph and an algorithm for finding the biconnected components of an undirected graph are presented. The space and time requirements of both algorithms are bounded by $k_1V + k_2E + k_3$ 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.

Credits

media	source	license
<i>Function Graph</i>	<u>Adobe Stock</u>	<u>Education License</u>
<i>Pac-Man Graph</i>	<u>Oatzy</u>	
<i>Pac-Man Game</i>	<u>Old Classic Retro Gaming</u>	
<i>London Tube Map</i>	<u>Transport for London</u>	
<i>London Tube Graph</i>	<u>visualize.org</u>	
<i>LinkedIn Social Network</i>	<u>Caleb Jones</u>	
<i>Twitter Graph</i>	<u>Caleb Jones</u>	
<i>Protein Interaction Graph</i>	<u>Hawing Jeong / KAIST</u>	
<i>PageRank</i>	<u>Wikipedia</u>	<u>public domain</u>
<i>Control Flow Graph</i>	<u>Stack Exchange</u>	
<i>DFS Graph Visualization</i>	<u>Gerry Jenkins</u>	

DFS visualization (by Gerry Jenkins)

