https://algs4.cs.princeton.edu

4.1 Undirected Graphs

- *introduction*
- *graph API*
- *depth-first search*
- *breadth-first search*
- *challenges*

https://algs4.cs.princeton.edu

4.1 Undirected Graphs

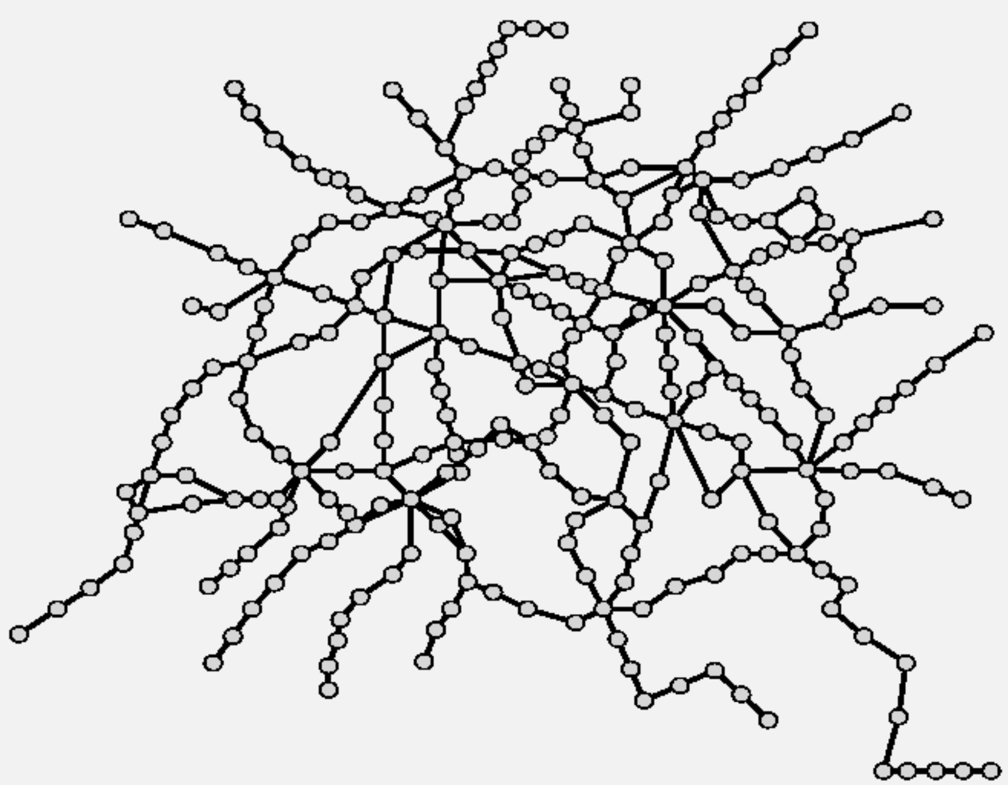
- *introduction*
- *graph API*
- *depth-first search*
- *breadth-first search*
- *challenges*

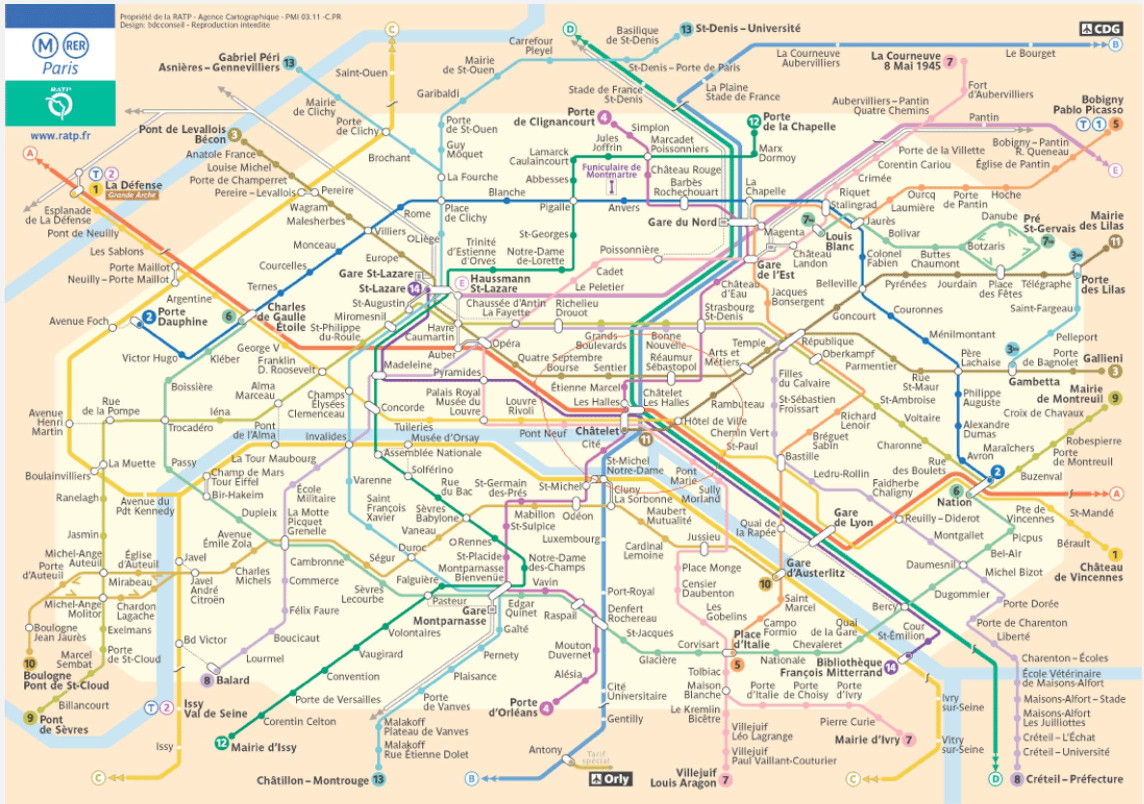
Undirected graphs

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.





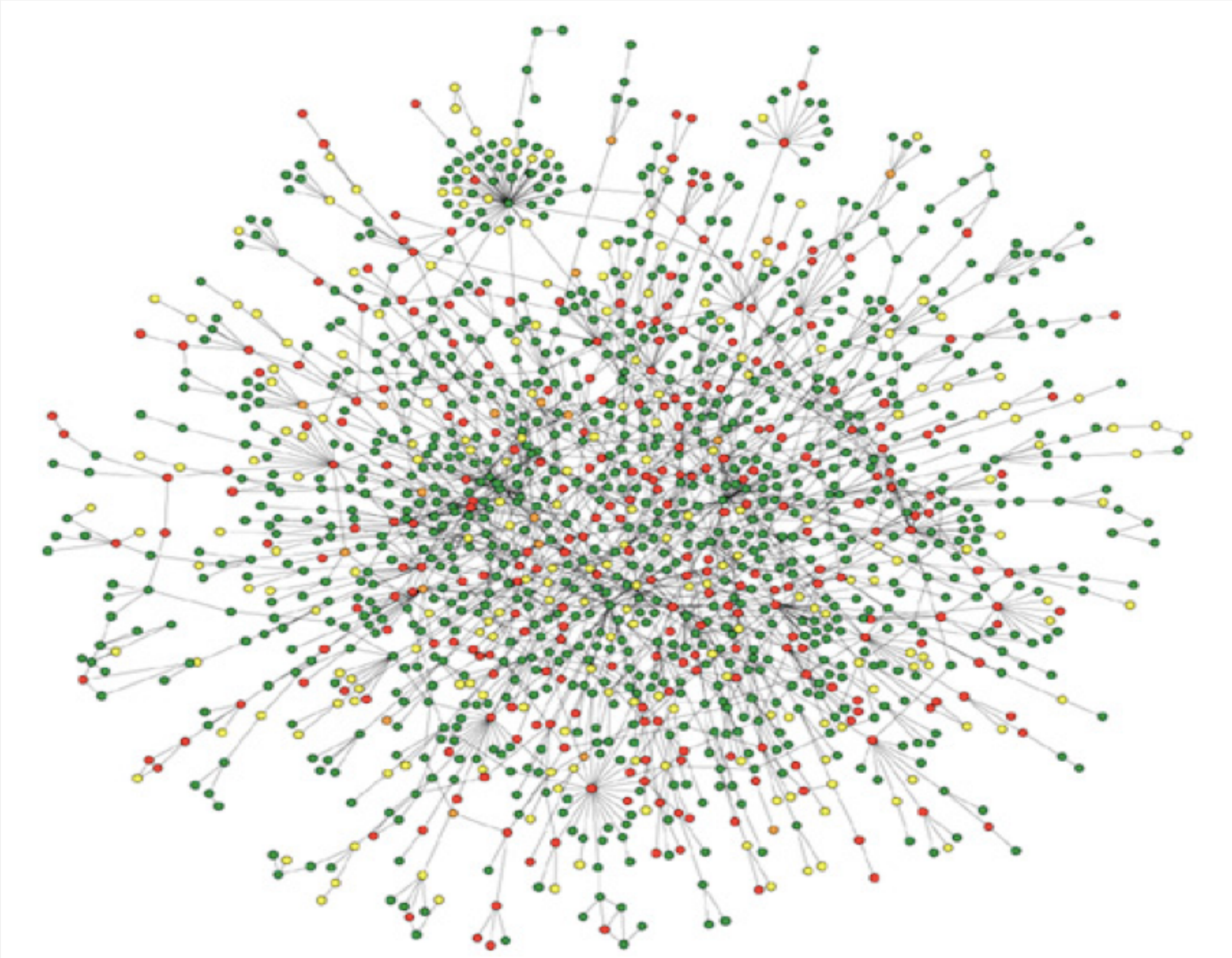
Social networks

Vertex = person; edge = social relationship.

"Visualizing Friendships" by Paul Butler

Protein-protein interaction network

Vertex = protein; edge = interaction.



Reference: Jeong et al, Nature Review | Genetics

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	intersection	street
internet	class C network	connection
game	board position	legal move
social relationship	person	friendship
neural network	neuron	synapse
protein network	protein	protein–protein interaction
molecule	atom	bond

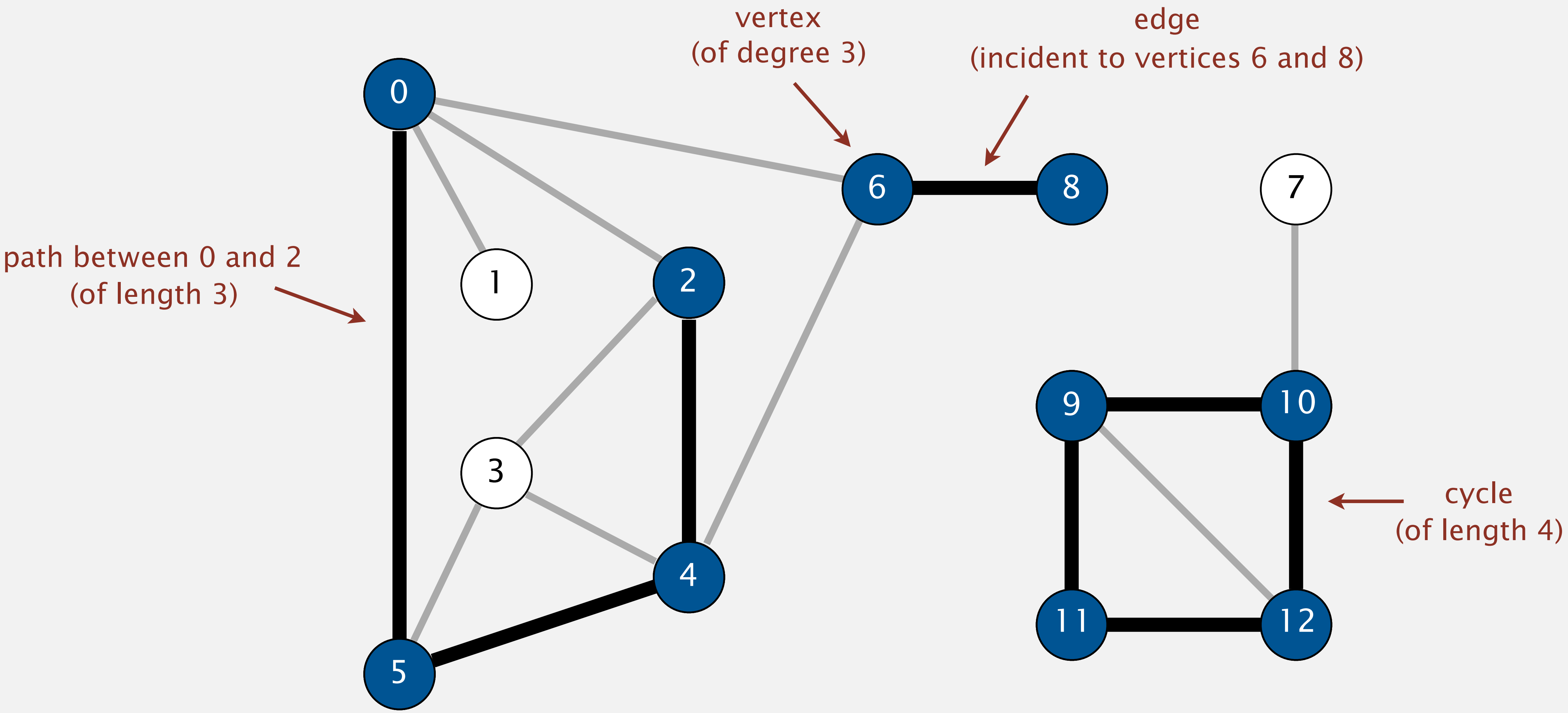
Graph terminology

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.



Some graph-processing problems

problem	description
s-t path	*Is there a path between s and t ?*
shortest s-t path	*What is the shortest path between s and t ?*
cycle	*Is there a cycle in the graph ?*
Euler cycle	*Is there a cycle that uses each edge exactly once ?*
Hamilton cycle	*Is there a cycle that uses each vertex exactly once ?*
connectivity	*Is there a path between every pair of vertices ?*
biconnectivity	*Is there a vertex whose removal disconnects the graph ?*
planarity	*Can the graph be drawn in the plane with no crossing edges ?*
graph isomorphism	*Are two graphs isomorphic?*

Challenge. Which graph problems are easy? Difficult? Intractable?

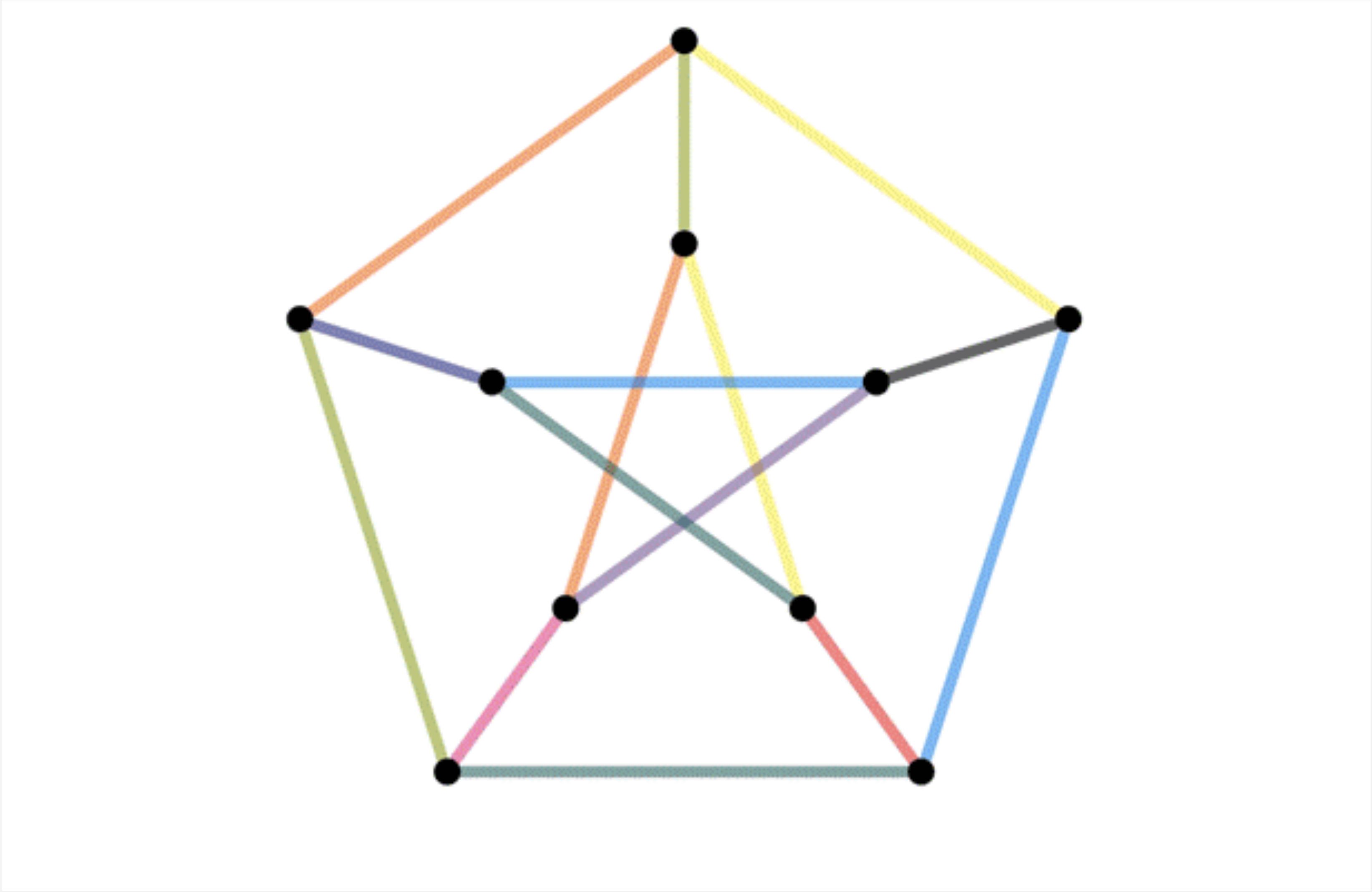
https://algs4.cs.princeton.edu

4.1 Undirected Graphs

- introduction
- graph API
- depth-first search
- breadth-first search
- challenges

Graph representation

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



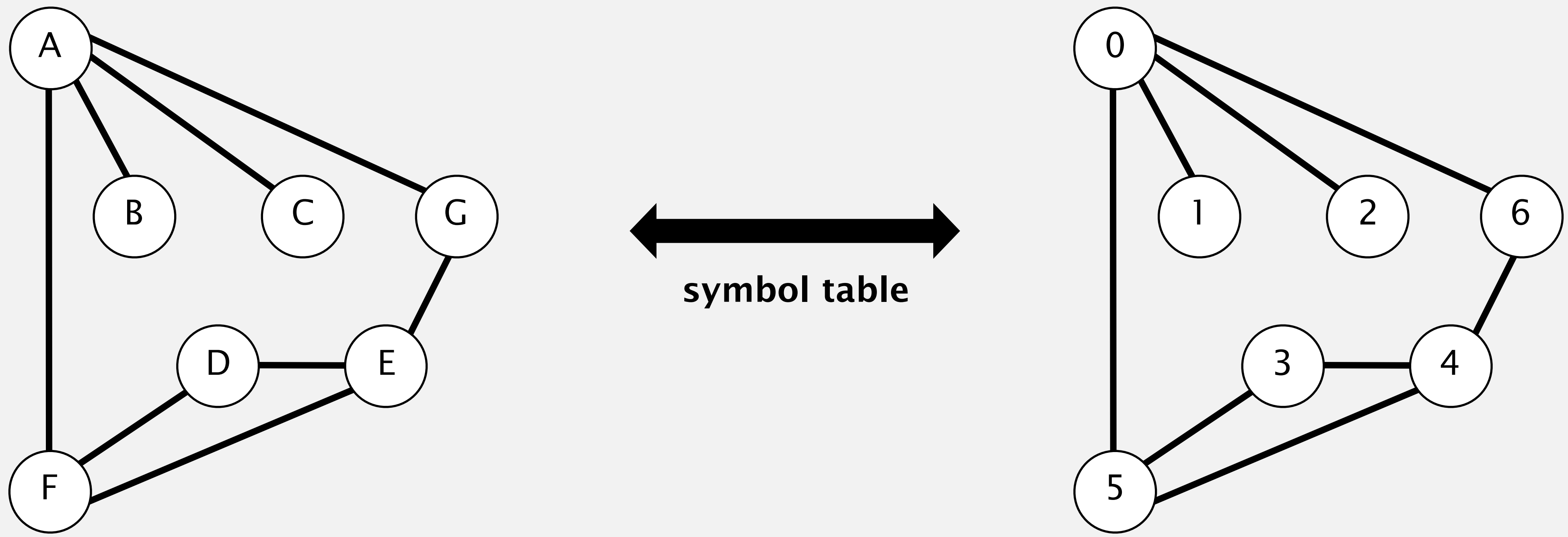
different drawings of the same graph

Caveat. Intuition can be misleading.

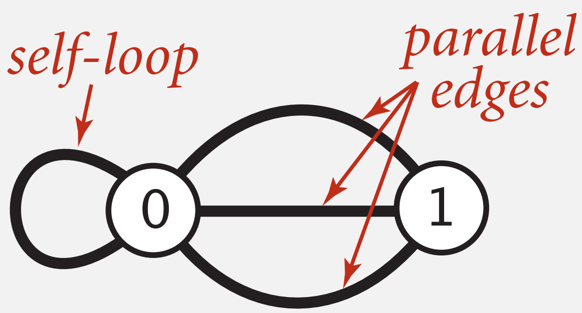
Graph representation

Vertex representation.

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



Anomalies.



Graph API

	`public class Graph`	
	`Graph(int V)`	*create an empty graph with V vertices*
`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*
	⋮	⋮

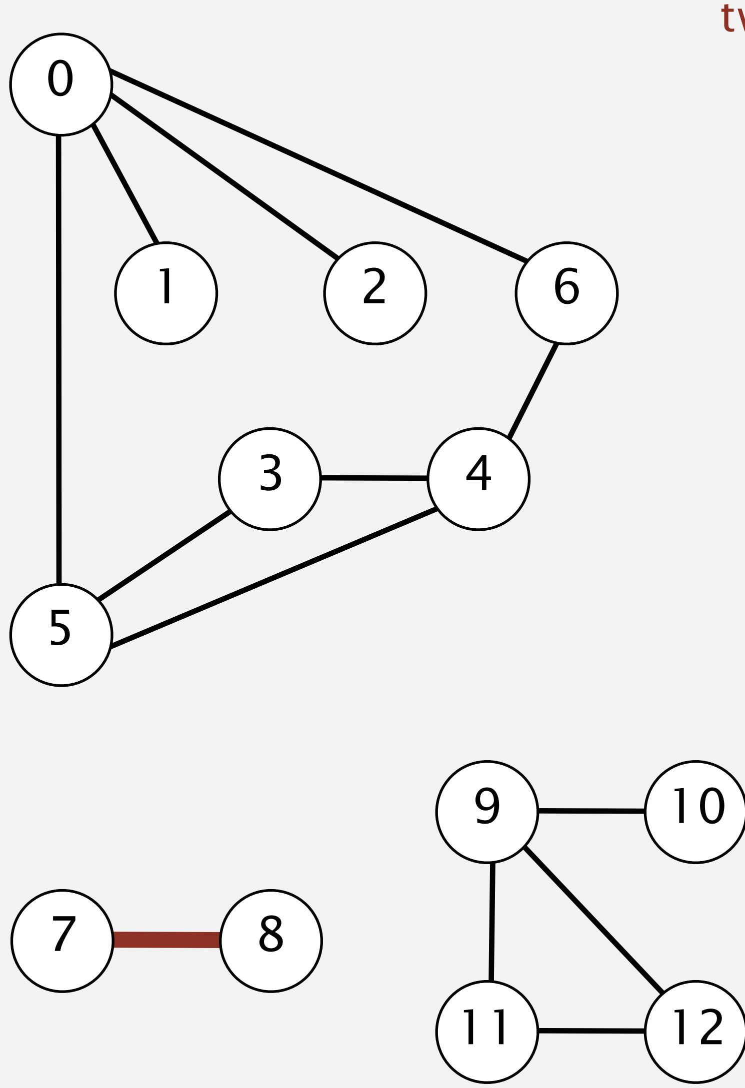
```
// degree of vertex v in graph G
public static int degree(Graph G, int v)
{
    int count = 0;
    for (int w : G.adj(v))
       count++;
    return count;
}
```

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

Graph representation: adjacency matrix

Maintain a V-by-V boolean array; for each edge $v–w$ in graph:

`adj[v][w] = adj[w][v] = true.`



two entries
per 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

Undirected graphs: quiz 1

Which is the order of growth of running time of the following code fragment if the graph uses the adjacency-matrix representation, where V is the number of vertices and E is the number of edges?

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

print each edge twice

A. V

B. $E + V$

C. V^2

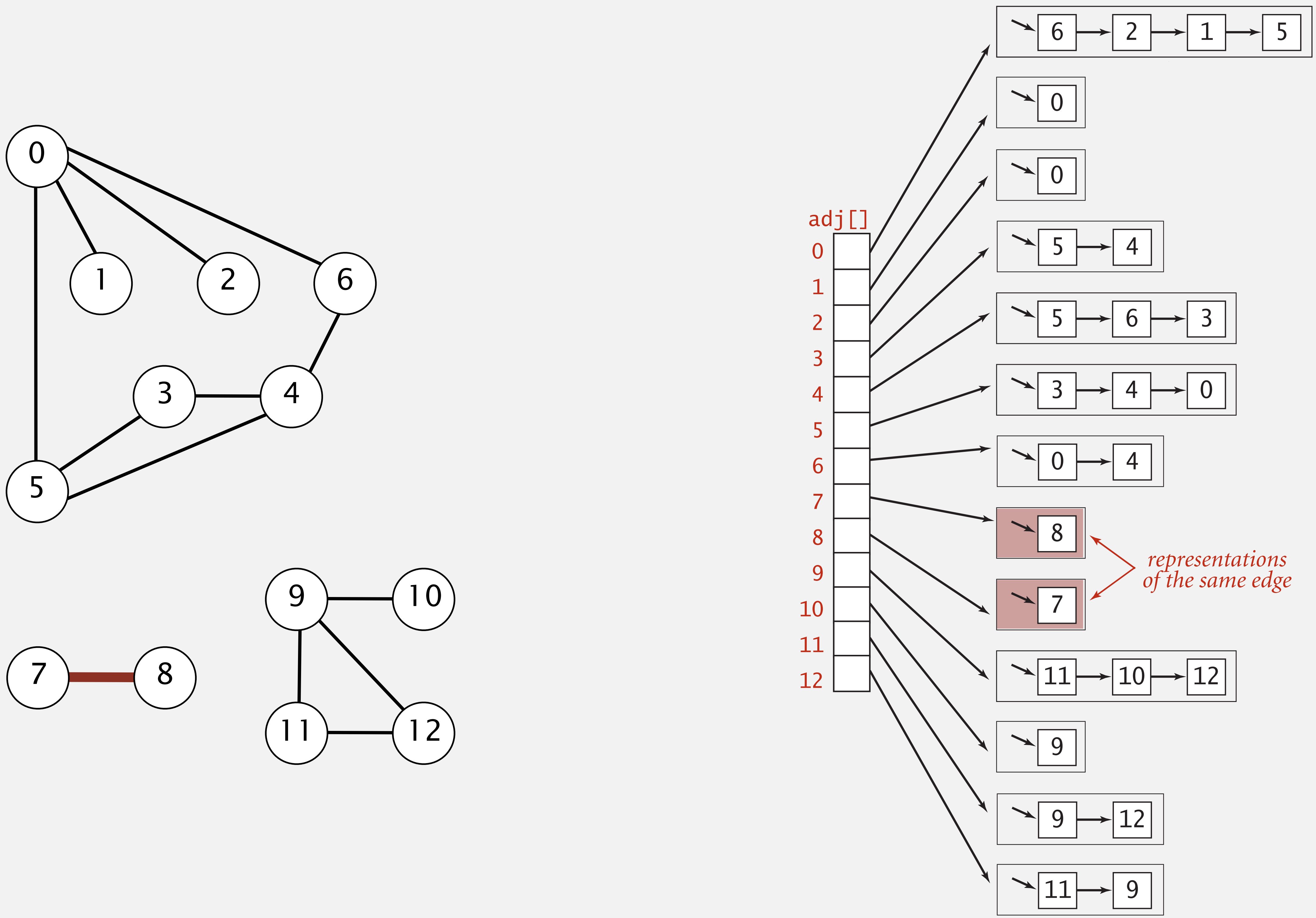
D. $V E$

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

adjacency-matrix representation

Graph representation: adjacency lists

Maintain vertex-indexed array of lists.



Undirected graphs: quiz 2

Which is the order of growth of running time of the following code fragment if the graph uses the adjacency-lists representation, where V is the number of vertices and E is the number of edges?

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

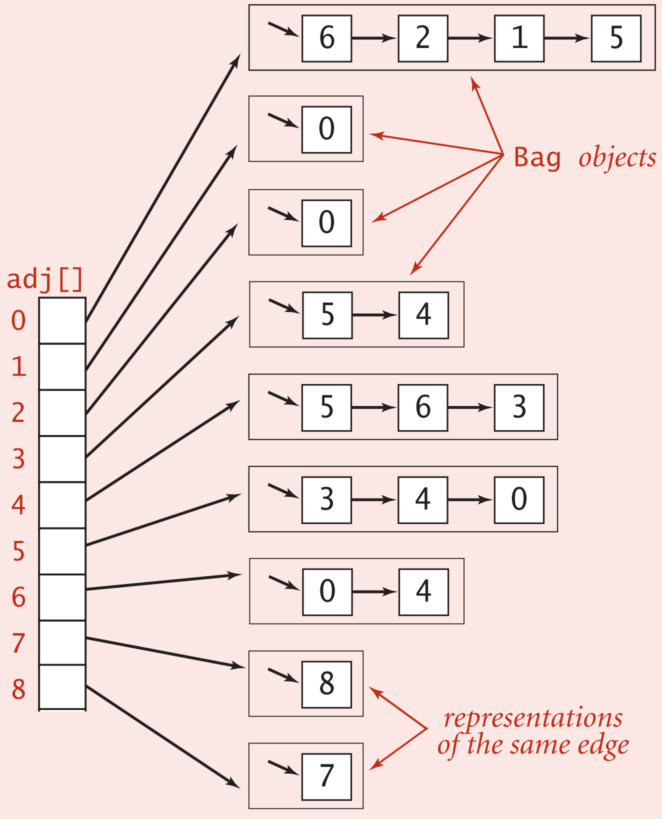
print each edge twice

A. V

B. $E + V$

C. V^2

D. $V E$



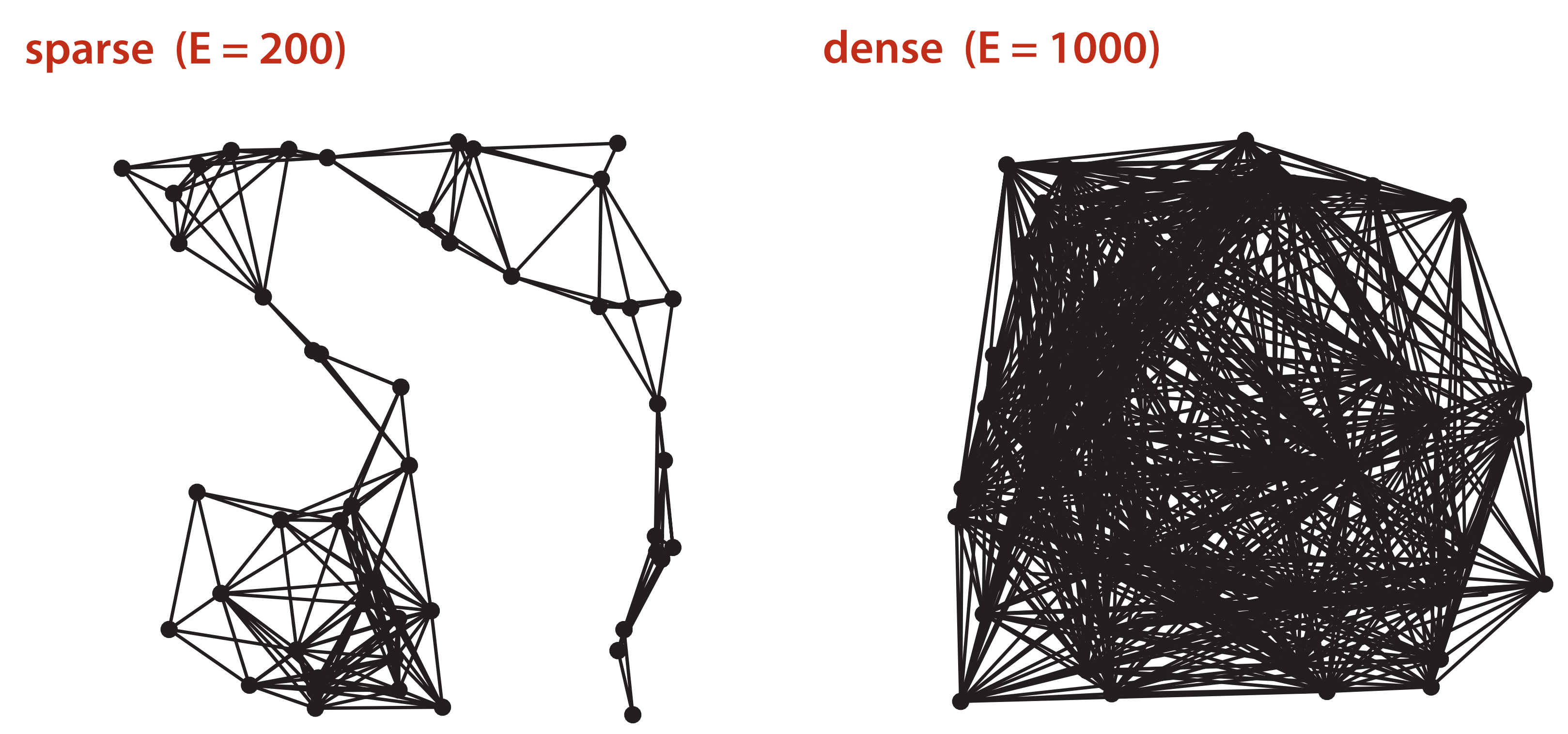
Graph representations

In practice. Use adjacency-lists representation.

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

proportional to V edges

proportional to V^2 edges



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 (not dense).

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

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 and self-loops allowed)

iterator for vertices adjacent to v

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

https://algs4.cs.princeton.edu

4.1 Undirected Graphs

- *introduction*
- *graph API*
- *depth-first search*
- *breadth-first search*
- *challenges*

Depth-first search

Goal. Systematically traverse a graph.

DFS (to visit a vertex v)

Mark vertex v.

Recursively visit all unmarked
vertices w adjacent to v.

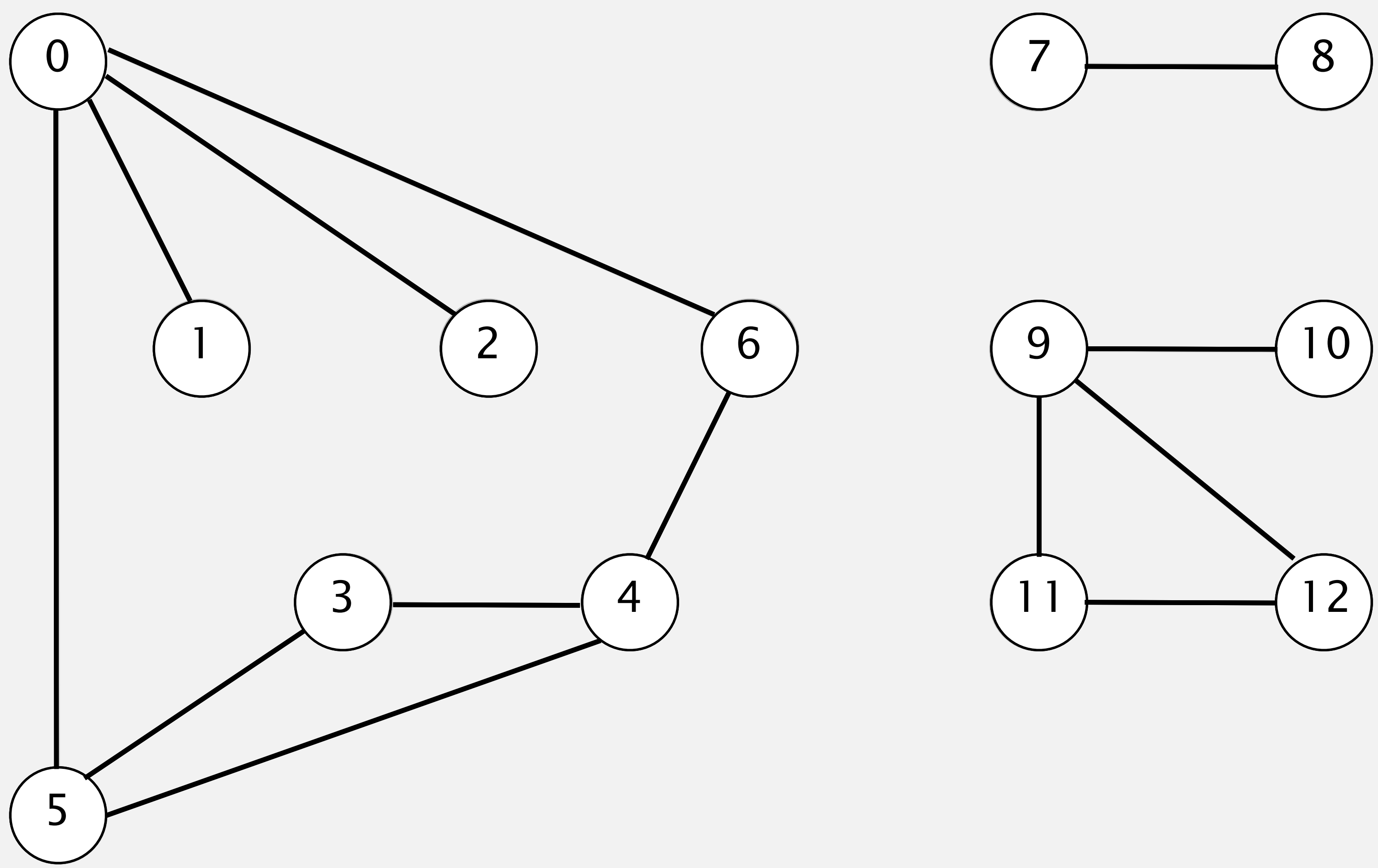
Typical applications.

- 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.



tinyG.txt

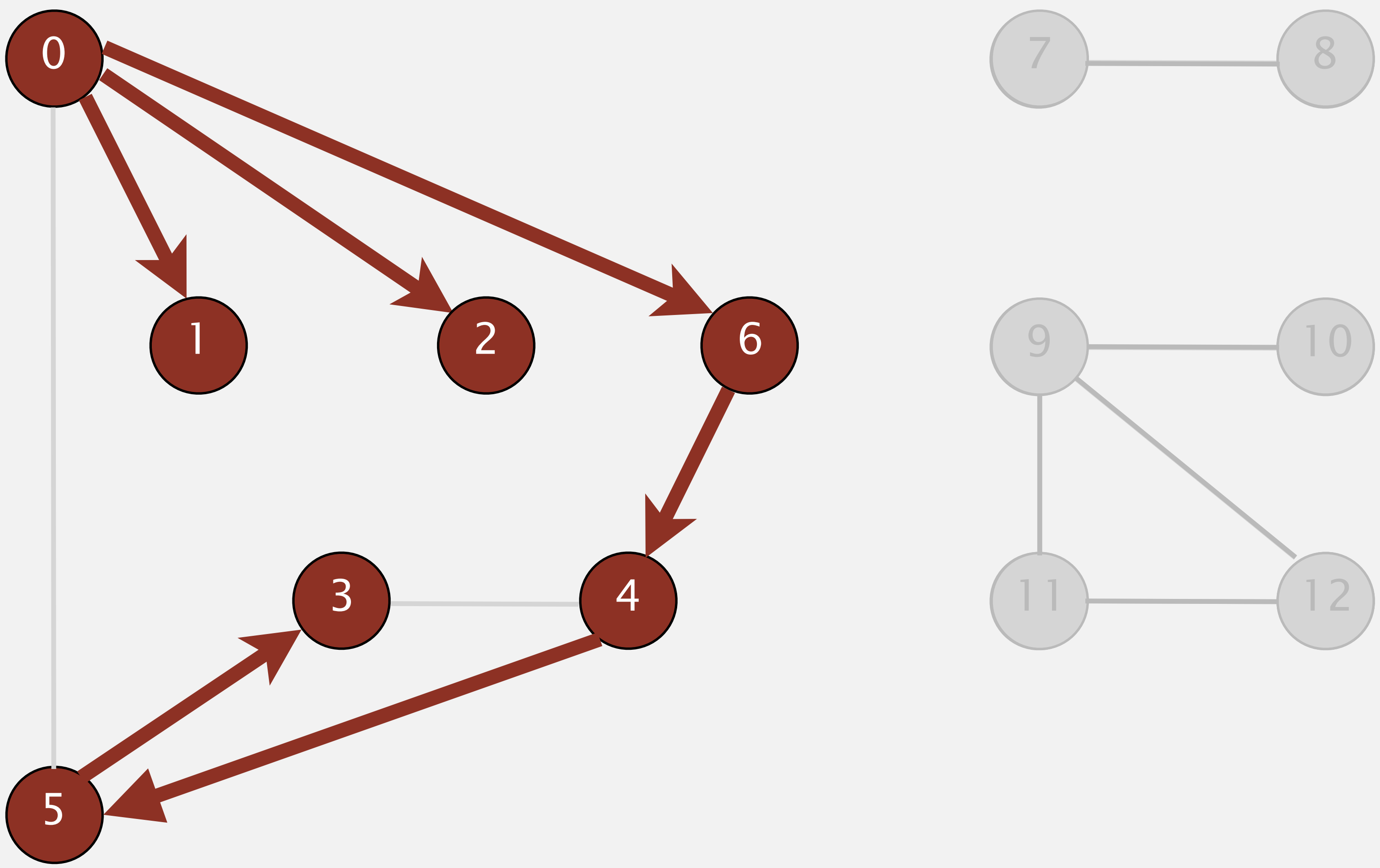
V → 13
13 ← E
0 5
4 3
0 1
9 12
6 4
5 4
0 2
11 12
9 10
0 6
7 8
9 11
5 3

graph G

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)

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

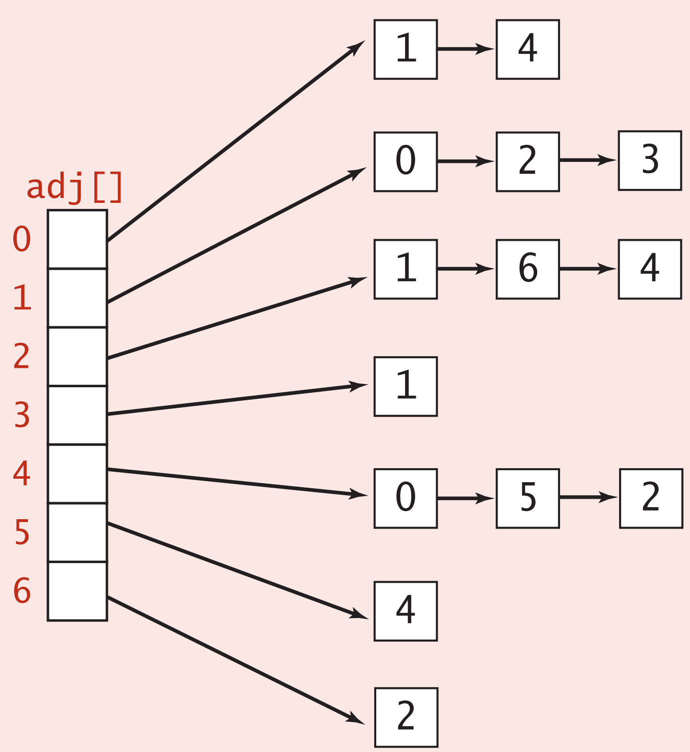
DFS preorder

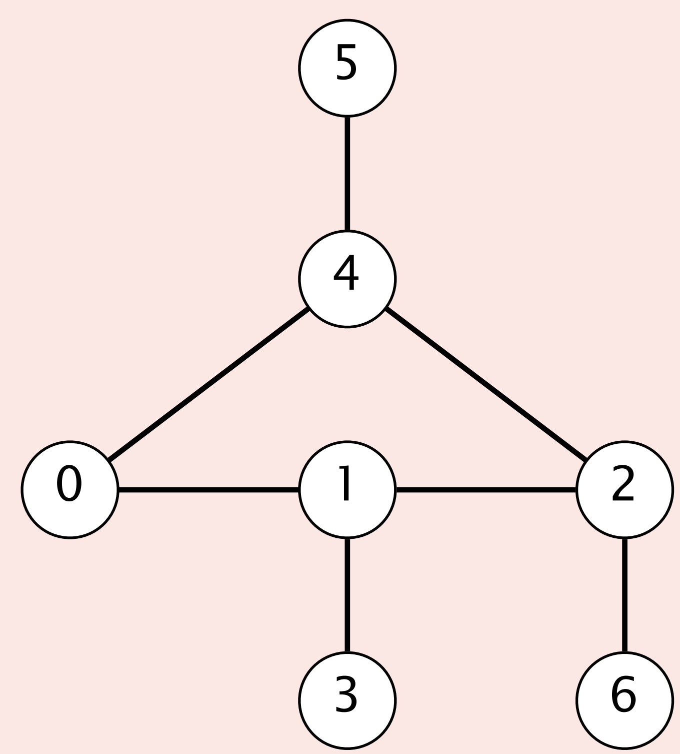
A. 0 1 2 4 5 3 6

B. 0 1 2 4 5 6 3

C. 0 1 4 2 5 3 6

D. 0 1 2 6 4 5 3





Depth-first search: data structures

To visit a vertex v :

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

Data structures.

- Boolean array `marked[]` to mark vertices.
- Integer array `edgeTo[]` to keep track of paths.

  `(edgeTo[w] == v)` means that edge `v-w` used to visit vertex `w`
- Function-call stack for recursion.

Design pattern for graph processing

Goal. Decouple graph data type from graph processing.

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

public class Paths		
	`Paths(Graph G, int s)`	*find paths in G connected to s*
`boolean`	`hasPathTo(int v)`	*is there a path between s and v?*
`Iterable<Integer>`	`pathTo(int v)`	*path between s and 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: Java implementation

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

   public DepthFirstPaths(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])
        {
           edgeTo[w] = v;
           dfs(G, w);
        }
   }
}
```

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

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

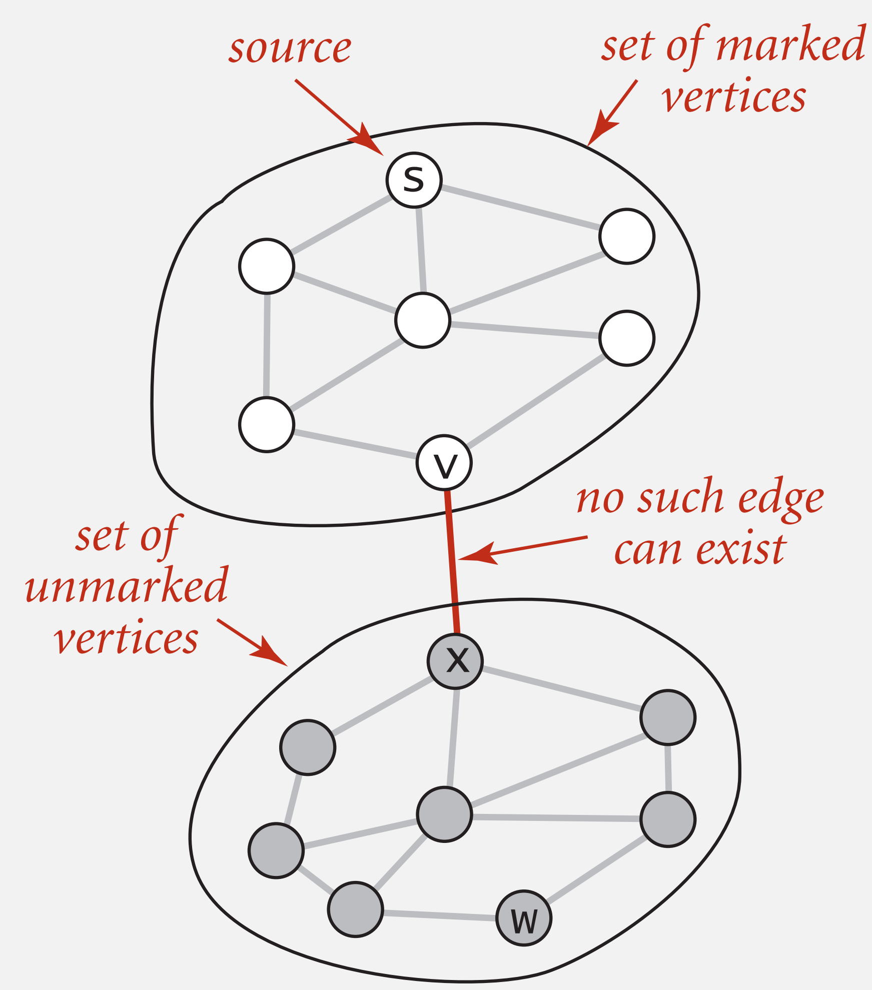
recursive DFS does the work

Depth-first search: properties

Proposition. DFS marks all vertices connected to s.

Pf.

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



skipped in lecture (see videos)

Depth-first search: properties

Proposition. DFS takes time proportional to $V + E$ in the worst case.

Pf.

- Initialize two arrays of length V.
- Each vertex is visited at most once.
  (visiting a vertex takes time proportional to its degree)

$$degree(v_0) + degree(v_1) + degree(v_2) + \ldots = 2E$$

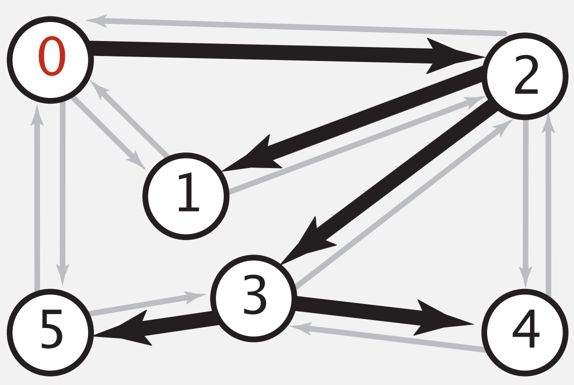
Depth-first search: properties

Proposition. After DFS, can check if vertex *v* is connected to *s* in constant time; can find *v*–*s* path (if one exists) in time proportional to its length.

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

Flood Fill

Problem. Implement flood fill (Photoshop magic wand).

https://algs4.cs.princeton.edu

4.1 Undirected Graphs

- *introduction*
- *graph API*
- *depth-first search*
- ***breadth-first search***
- *challenges*

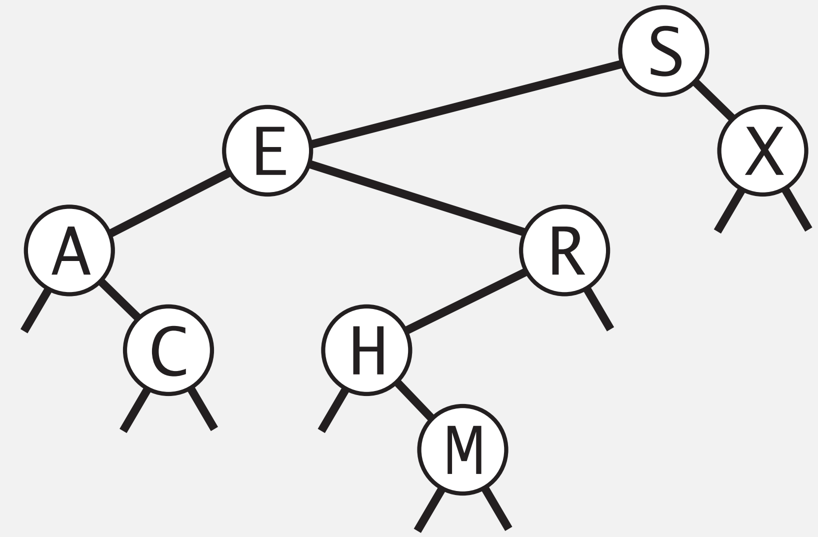
Graph search

Tree traversal. Many ways to explore a binary tree.

- Inorder: `A C E H M R S X`
- Preorder: `S E A C R H M X`
- Postorder: `C A M H R E X S`
- Level-order: `S E X A R C H M`

stack/recursion

queue



Graph search. Many ways to explore a graph.

- DFS preorder: vertices in order of calls to `dfs(G, v)`.
- DFS postorder: vertices in order of returns from `dfs(G, v)`.
- Breadth-first: vertices in increasing order of distance from `s`.

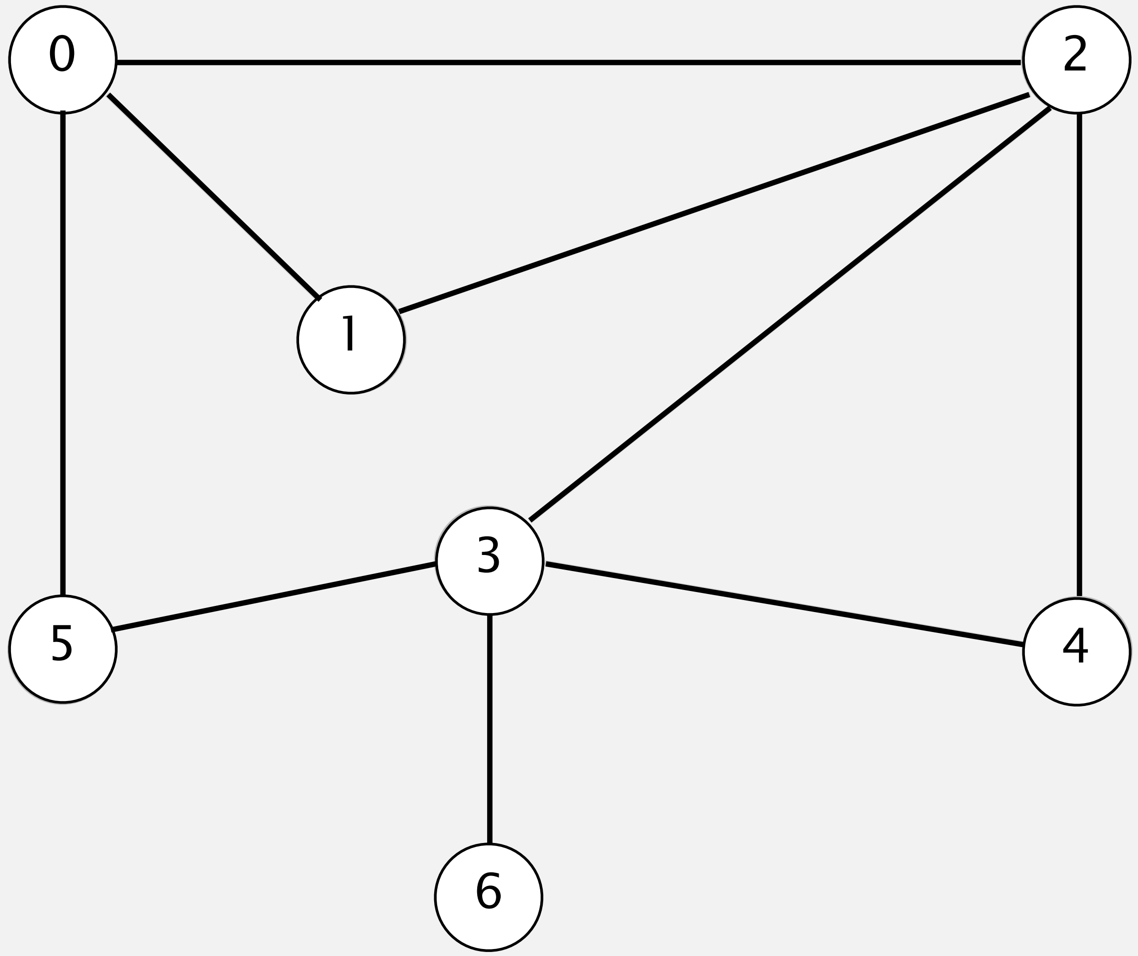
stack/recursion

queue

Breadth-first search demo

Repeat until queue is empty:

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

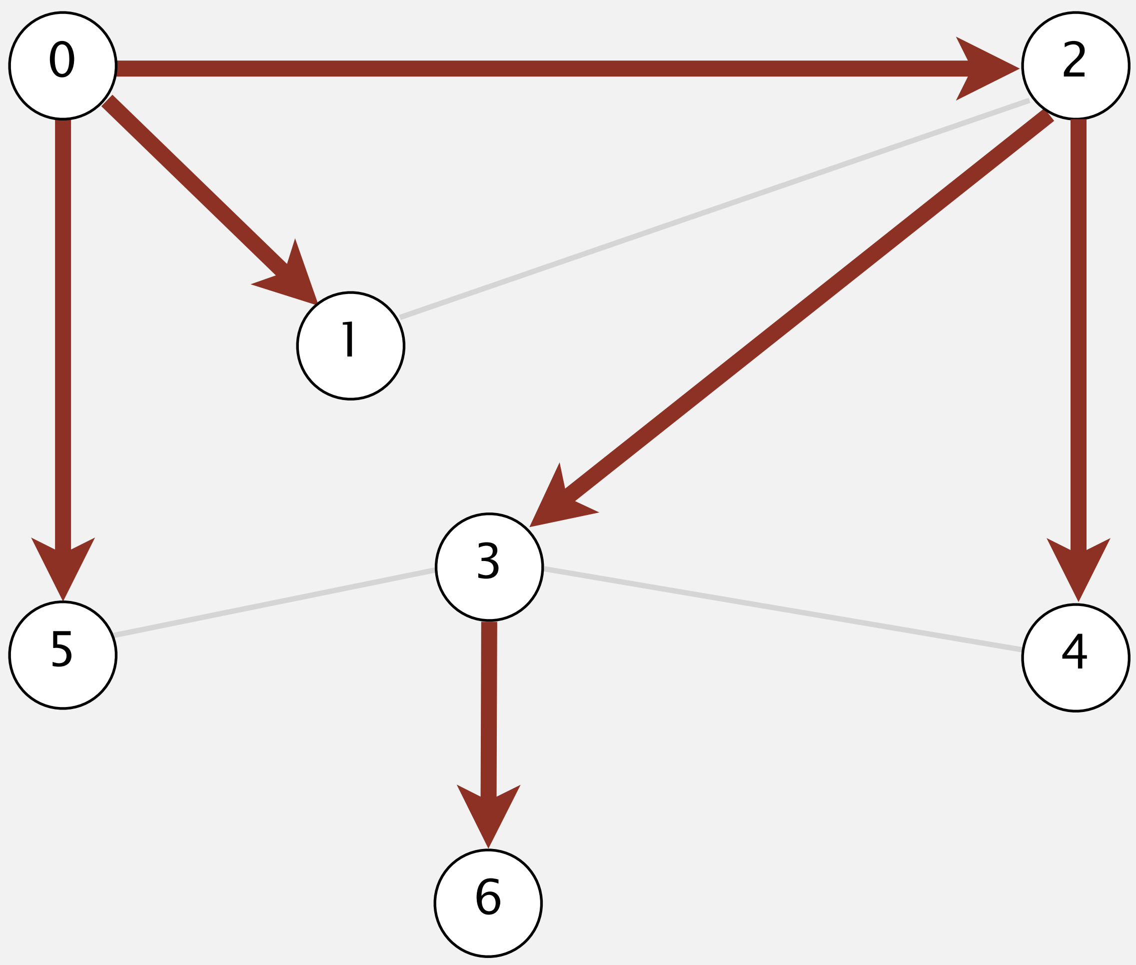


graph G

Breadth-first search demo

Repeat until queue is empty:

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



v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	2	2
4	2	2
5	0	1
6	3	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.

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 unmarked neighbors to the queue,
and mark them.**

Breadth-first search: Java implementation

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

   …

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

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

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

initialize FIFO queue of vertices to explore

found new vertex w via edge v–w

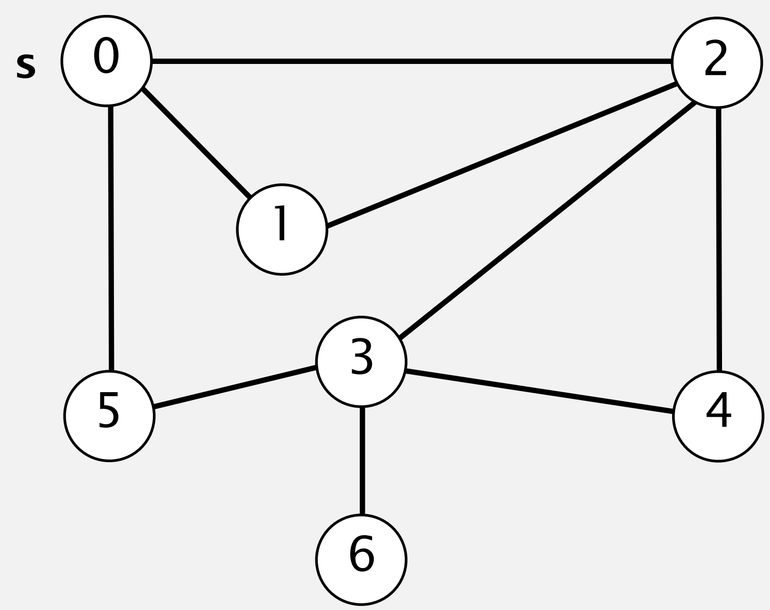
Breadth-first search properties

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

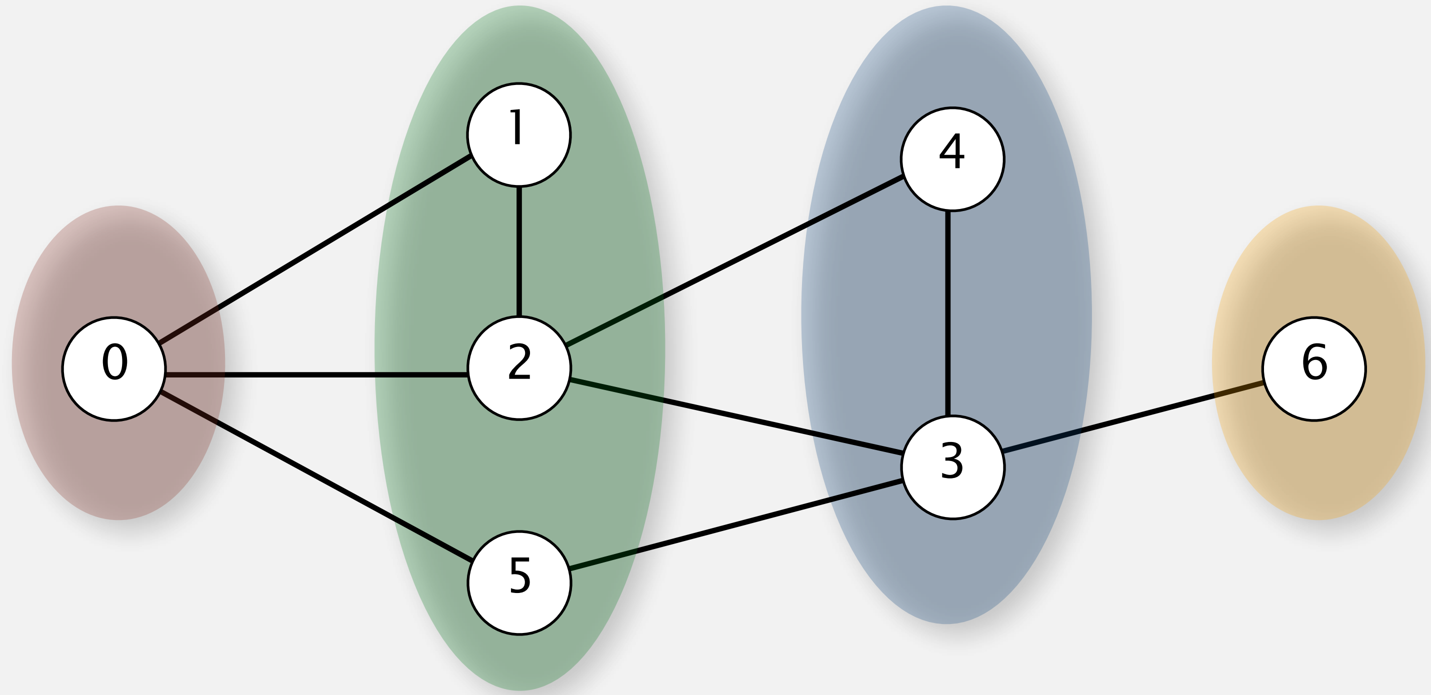
distance = number of edges

Pf idea. BFS examines vertices in increasing distance from s.

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



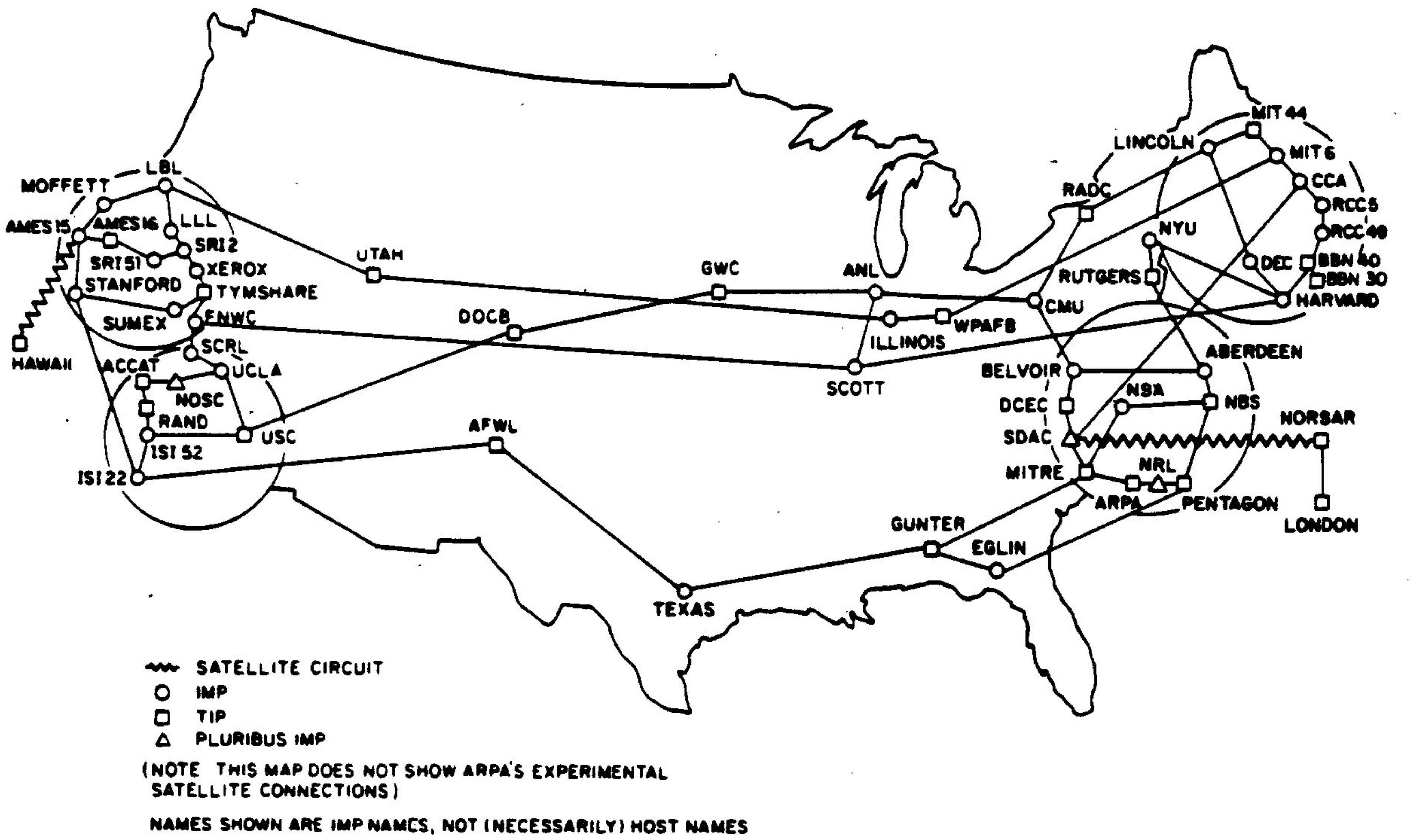
graph G



dist = 0 dist = 1 dist = 2

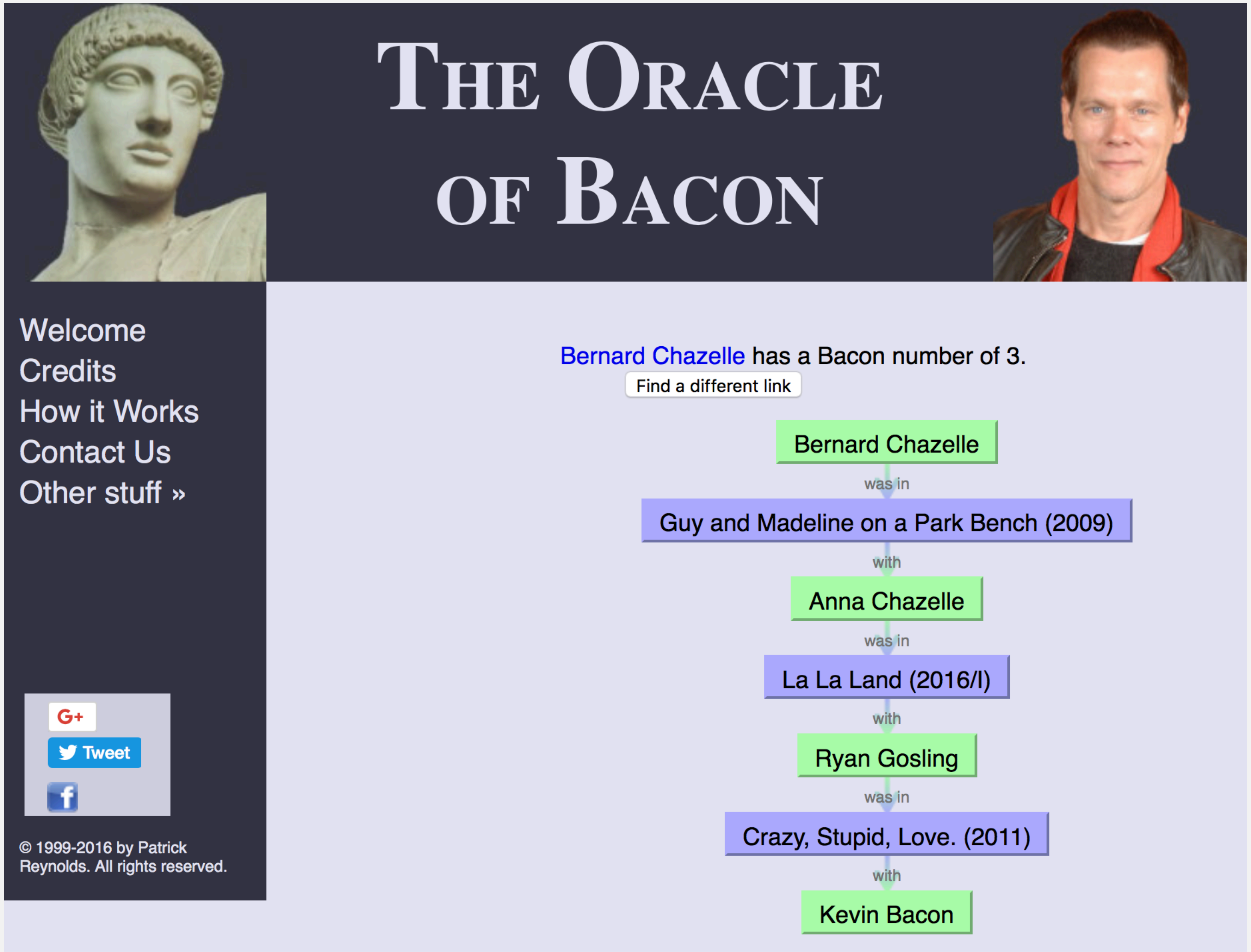
Breadth-first search application: routing

Fewest number of hops in a communication network.



ARPANET, July 1977

Breadth-first search application: Kevin Bacon numbers



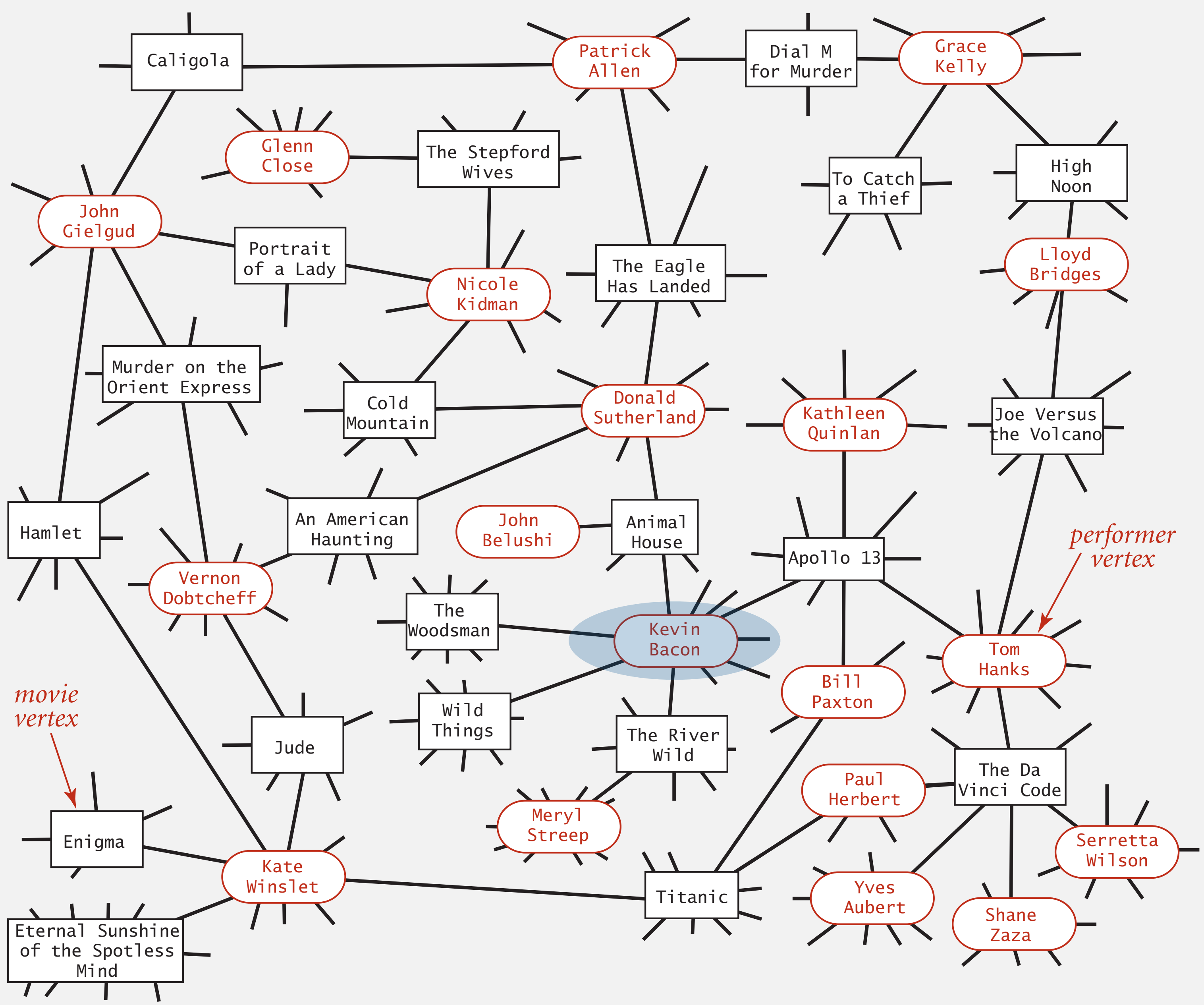
https://oracleofbacon.org

Endless Games board game

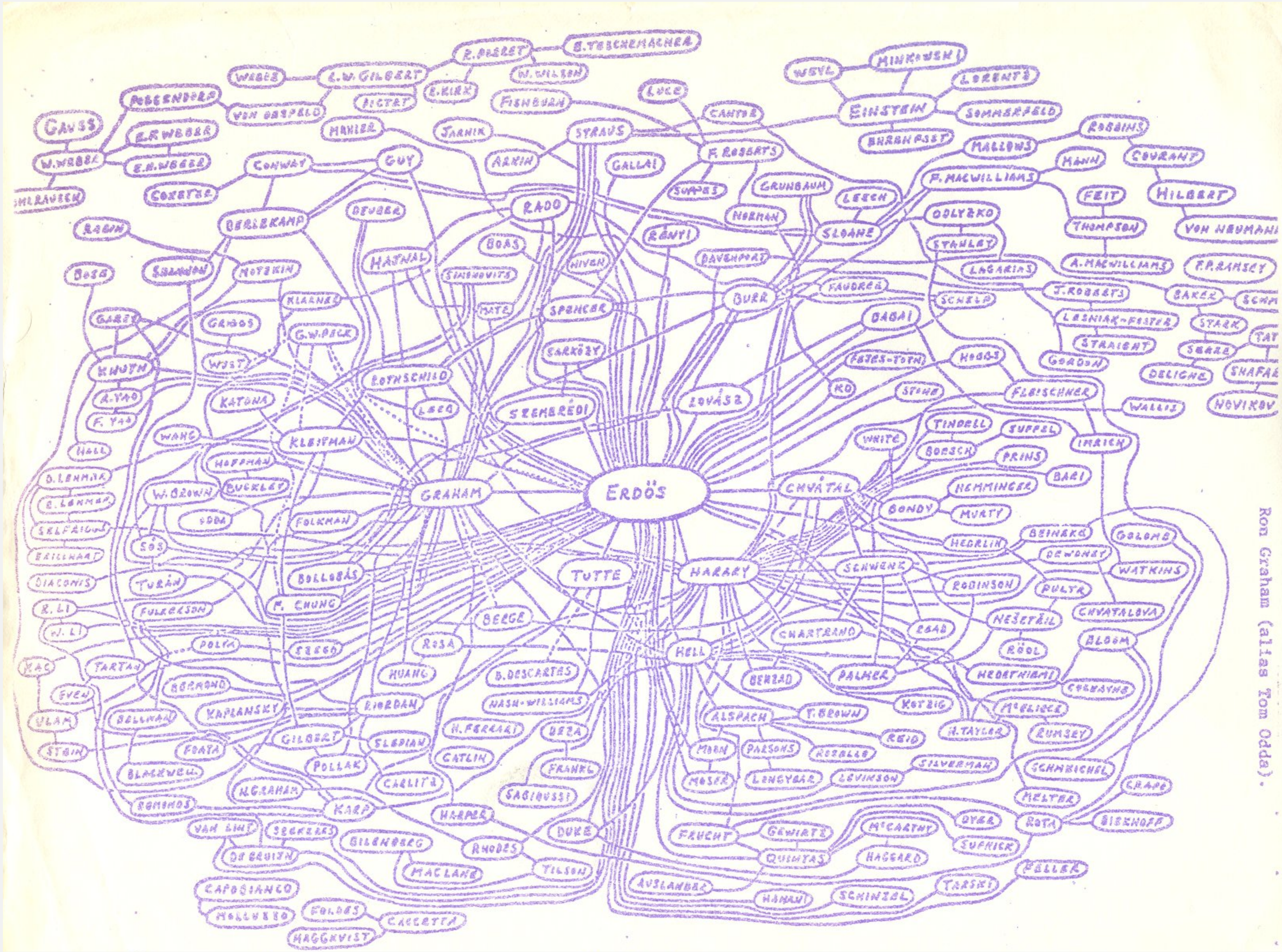
SixDegrees iPhone App

Kevin Bacon graph

- Include one vertex for each performer and one for each movie.
- Connect a movie to all performers that appear in that movie.
- Compute shortest path from s = Kevin Bacon.



Breadth-first search application: Erdös numbers



hand-drawing of part of the Erdös graph by Ron Graham

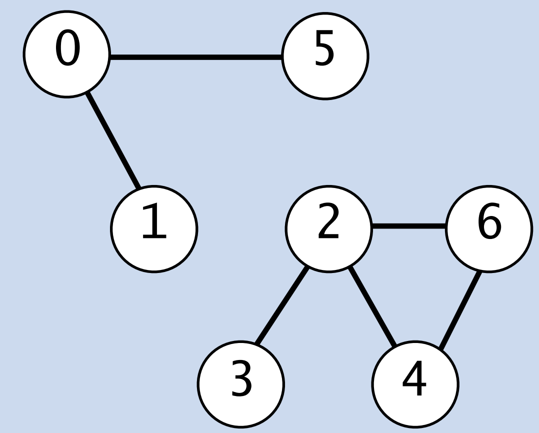
https://algs4.cs.princeton.edu

4.1 Undirected Graphs

- introduction
- graph API
- depth-first search
- breadth-first search
- challenges

Graph-processing challenge 1

Problem. Identify connected components.



0–1
0–5
2–6
2–3
2–4
4–6

How difficult?

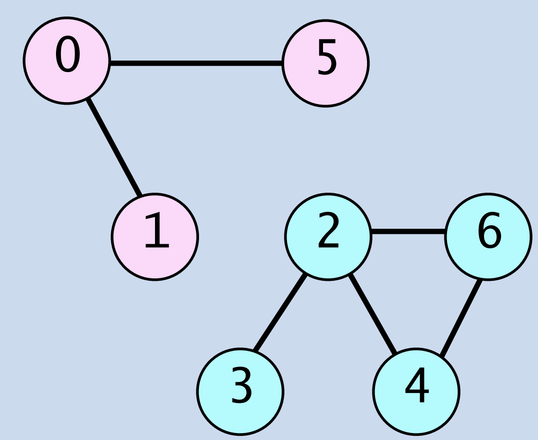
A. Any programmer could do it.

B. Diligent algorithms student could do it.

C. Hire an expert.

D. Intractable.

E. No one knows.



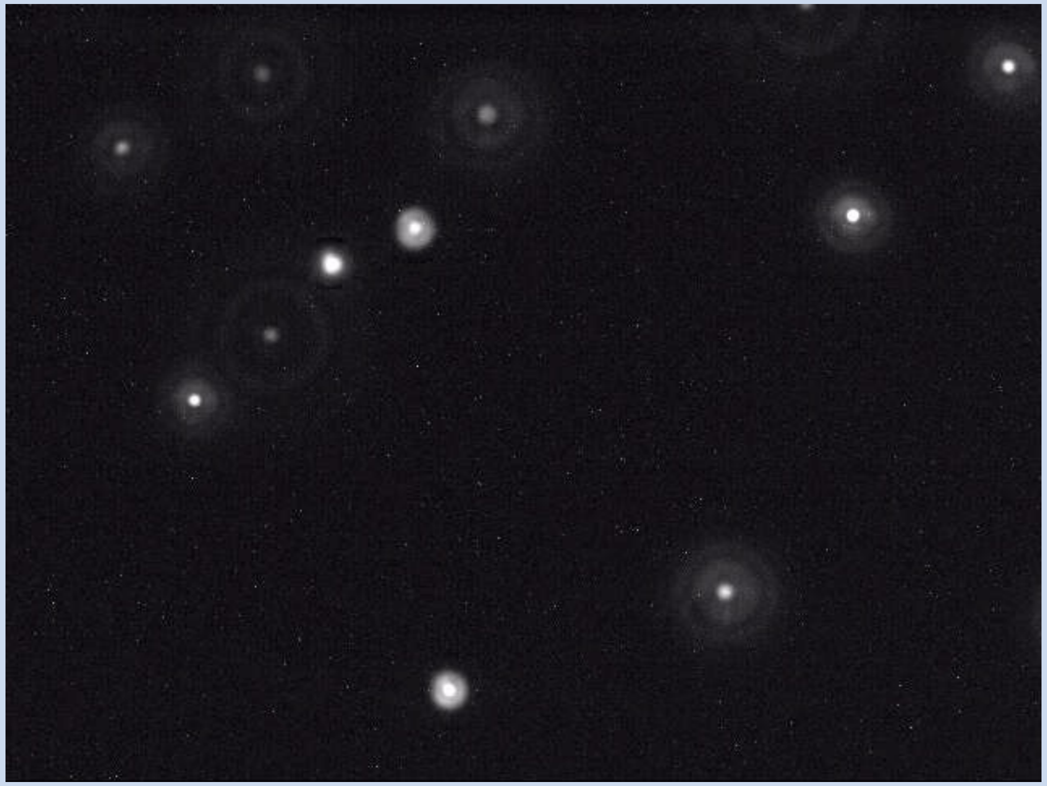
v	id[]
0	0
1	0
2	1
3	1
4	1
5	0
6	1

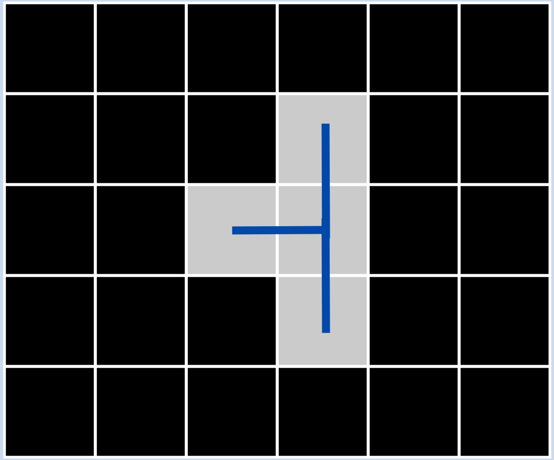
Graph-processing challenge 1

Problem. Identify connected components.

Particle detection. Given grayscale image of particles, identify “blobs.”

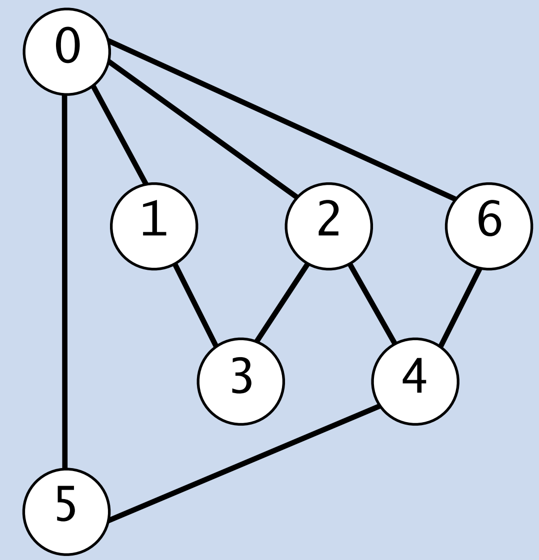
- Vertex: pixel.
- Edge: between two adjacent pixels with grayscale value ≥ 70.
- Blob: connected component of 20–30 pixels.





Graph-processing challenge 2

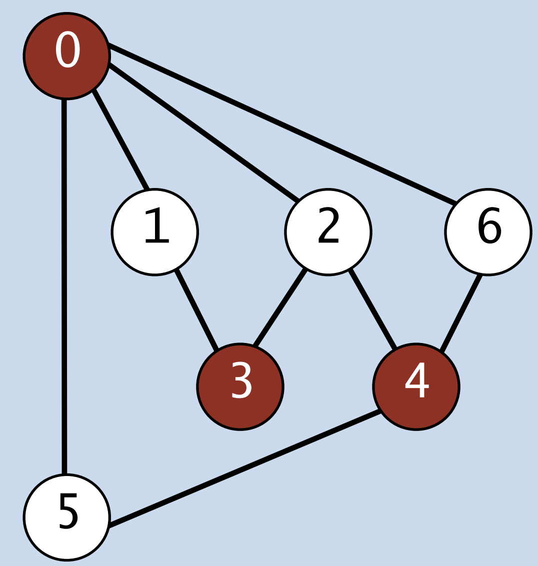
Problem. Is a graph bipartite?



0–1
0–2
0–5
0–6
1–3
2–3
2–4
4–5
4–6

How difficult?

- **A.** Any programmer could do it.
- **B.** Diligent algorithms student could do it.
- **C.** Hire an expert.
- **D.** Intractable.
- **E.** No one knows.

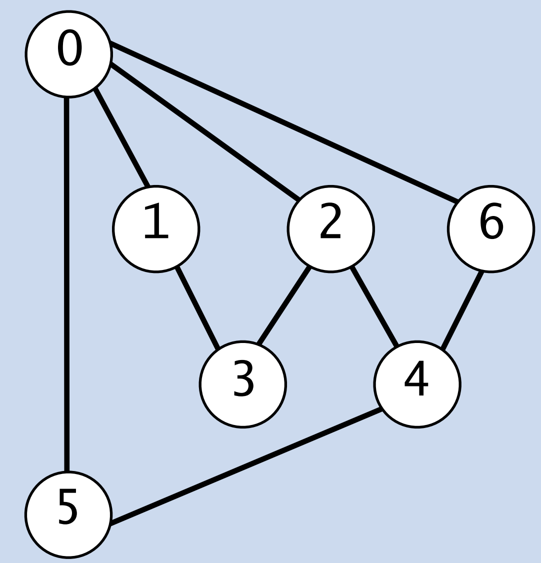


`{ 0, 3, 4 }`

0–1
0–2
0–5
0–6
1–3
2–3
2–4
4–5
4–6

Graph-processing challenge 3

Problem. Find a cycle in a graph (if one exists).



0–1
0–2
0–5
0–6
1–3
2–3
2–4
4–5
4–6

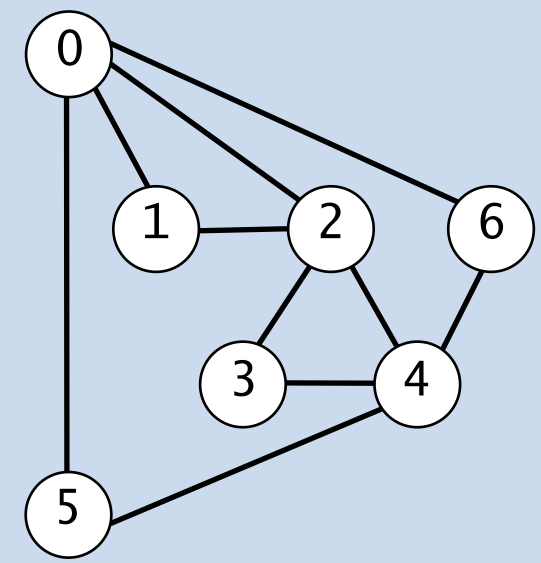
0–5–4–6–0

How difficult?

- **A.** Any programmer could do it.
- **B.** Diligent algorithms student could do it.
- **C.** Hire an expert.
- **D.** Intractable.
- **E.** No one knows.

Graph-processing challenge 4

Problem. Is there a (general) cycle that uses every edge exactly once?



0–1
0–2
0–5
0–6
1–2
2–3
2–4
3–4
4–5
4–6

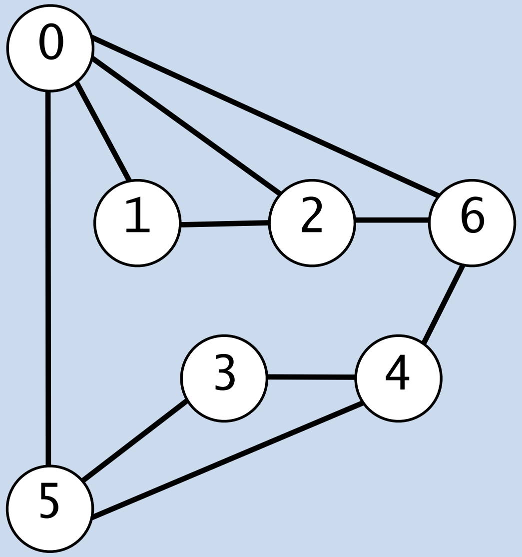
0–1–2–3–4–2–0–6–4–5–0

How difficult?

- **A.** Any programmer could do it.
- **B.** Diligent algorithms student could do it.
- **C.** Hire an expert.
- **D.** Intractable.
- **E.** No one knows.

Graph-processing challenge 5

Problem. Is there a cycle that uses every vertex exactly once?



0–1
0–2
0–5
0–6
1–2
2–6
3–4
3–5
4–5
4–6

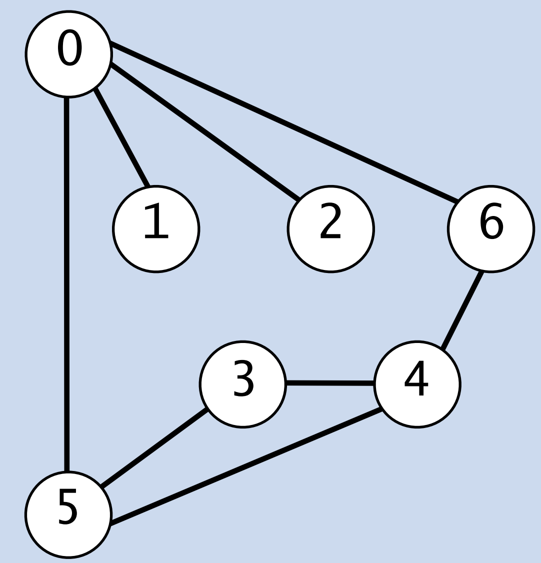
0–5–3–4–6–2–1–0

How difficult?

- **A.** Any programmer could do it.
- **B.** Diligent algorithms student could do it.
- **C.** Hire an expert.
- **D.** Intractable.
- **E.** No one knows.

Graph-processing challenge 6

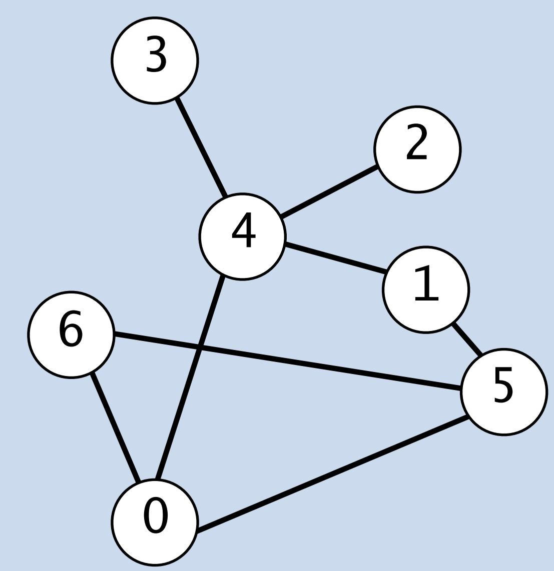
Problem. Are two graphs identical except for vertex names?



0–1
0–2
0–5
0–6
3–4
3–5
4–5
4–6

How difficult?

- **A.** Any programmer could do it.
- **B.** Diligent algorithms student could do it.
- **C.** Hire an expert.
- **D.** Intractable.
- **E.** No one knows.



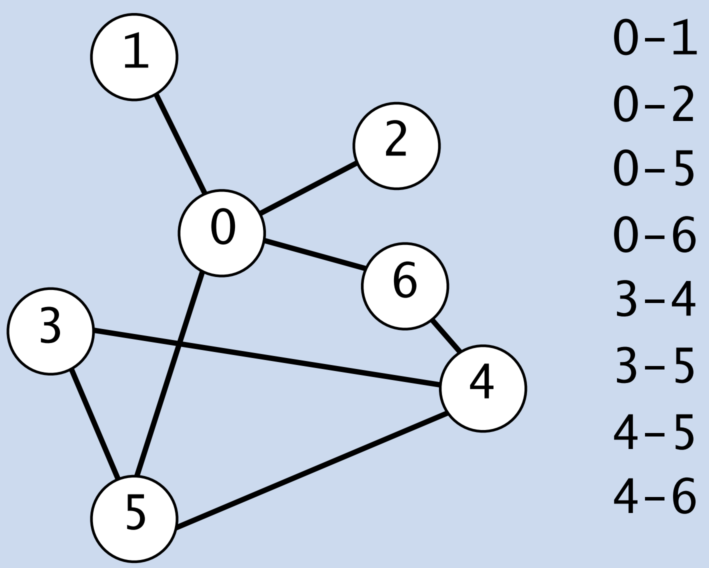
0–4
0–5
0–6
1–4
1–5
2–4
3–4
5–6

0↔4, 1↔3, 2↔2, 3↔6, 4↔5, 5↔0, 6↔1

Graph-processing challenge 7

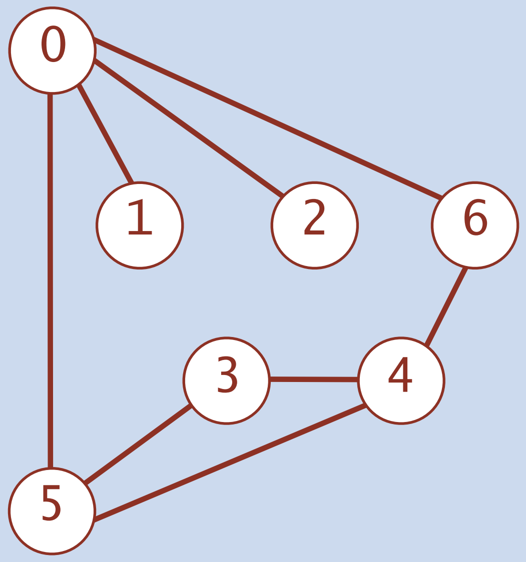
Problem. Can you draw a graph in the plane with no crossing edges?

try it yourself at http://planarity.net



How difficult?

- **A.** Any programmer could do it.
- **B.** Diligent algorithms student could do it.
- **C.** Hire an expert.
- **D.** Intractable.
- **E.** No one knows



Graph traversal summary

BFS and DFS enables efficient solution of many (but not all) graph problems.

graph problem	BFS	DFS	time
s-t path	✔	✔	$E + V$
shortest s-t path	✔		$E + V$
cycle	✔	✔	V
Euler cycle		✔	$E + V$
Hamilton cycle			$2^{1.657\,V}$
bipartiteness (odd cycle)	✔	✔	$E + V$
connected components	✔	✔	$E + V$
biconnected components		✔	$E + V$
planarity		✔	$E + V$
graph isomorphism			$2^{c \ln^3 V}$