

INTRODUCTION TO DATA STRUCTURES and ALGORITHMS

Rutgers University

UNDIRECTED GRAPHS

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



UNDIRECTED GRAPHS

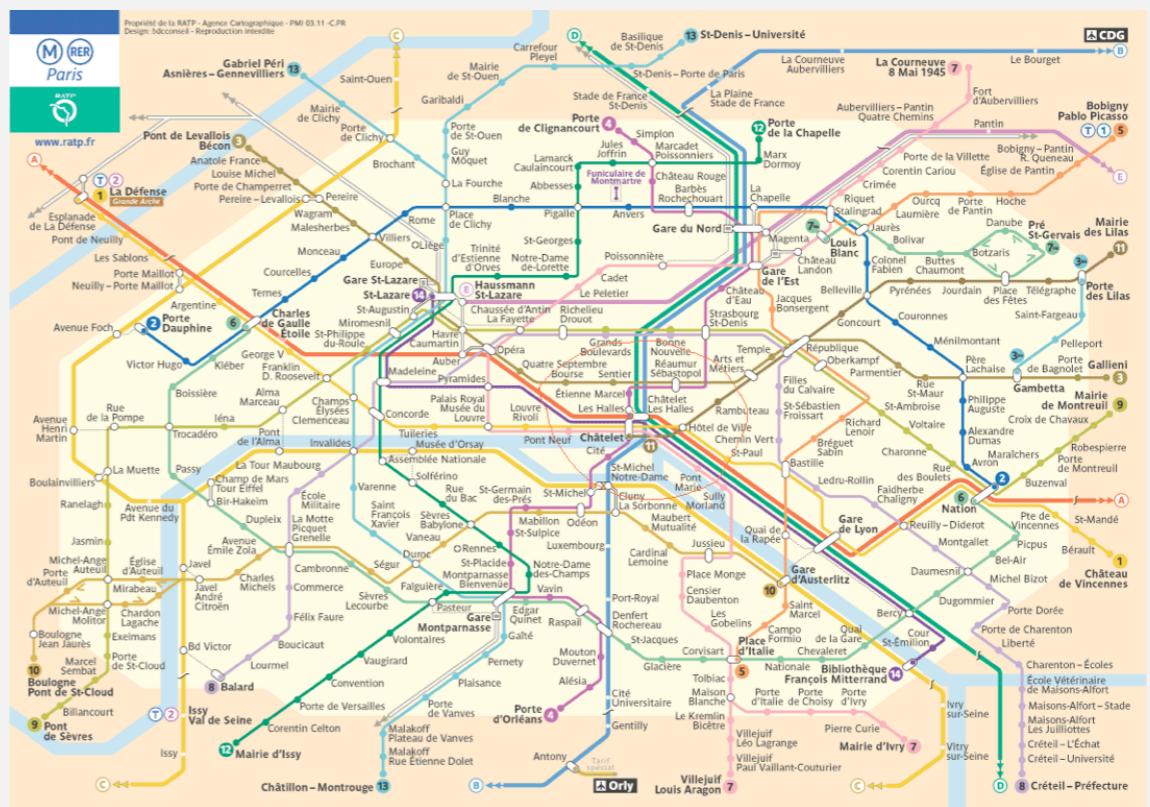
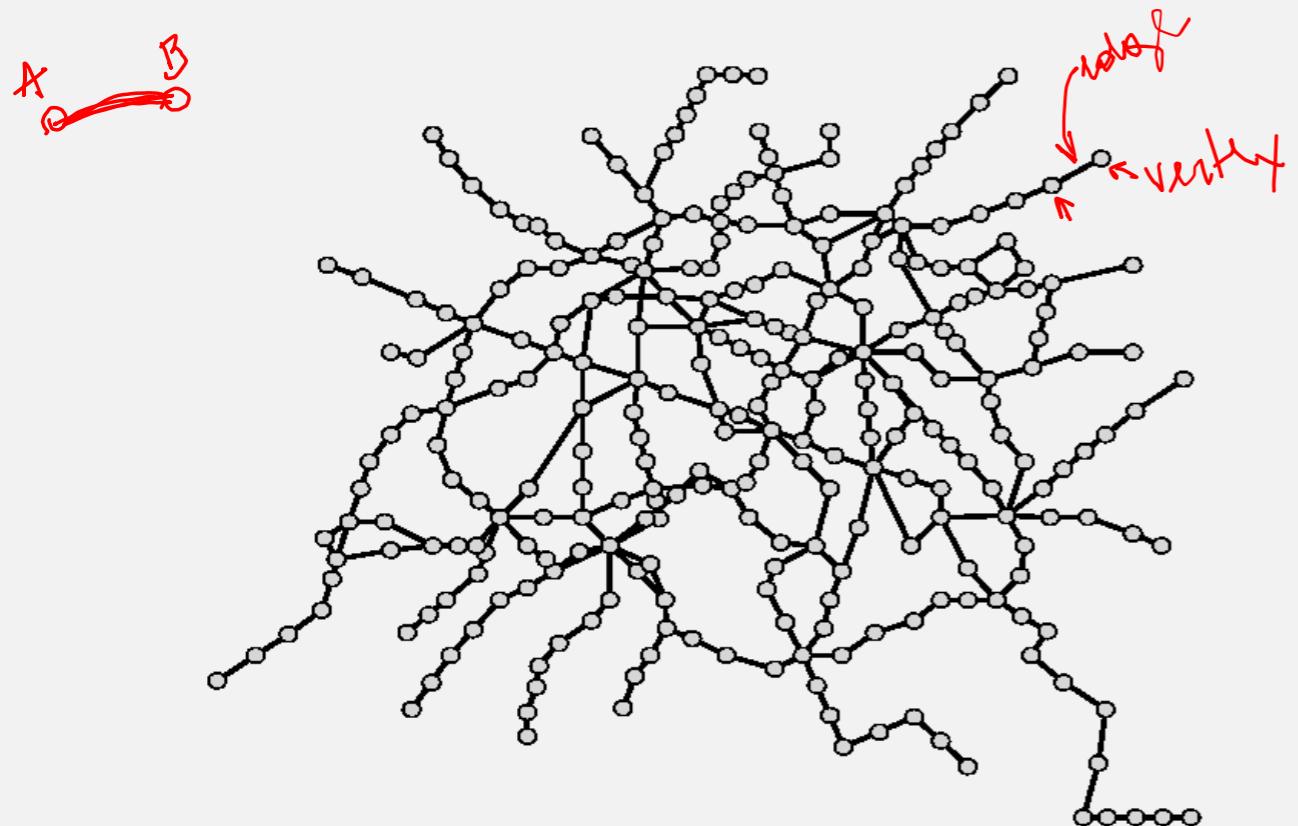
- ▶ *introduction*
- ▶ *graph API*
- ▶ *depth-first search*
- ▶ *breadth-first search*
- ▶ *challenges*

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.



Vertex = person; edge = social relationship.

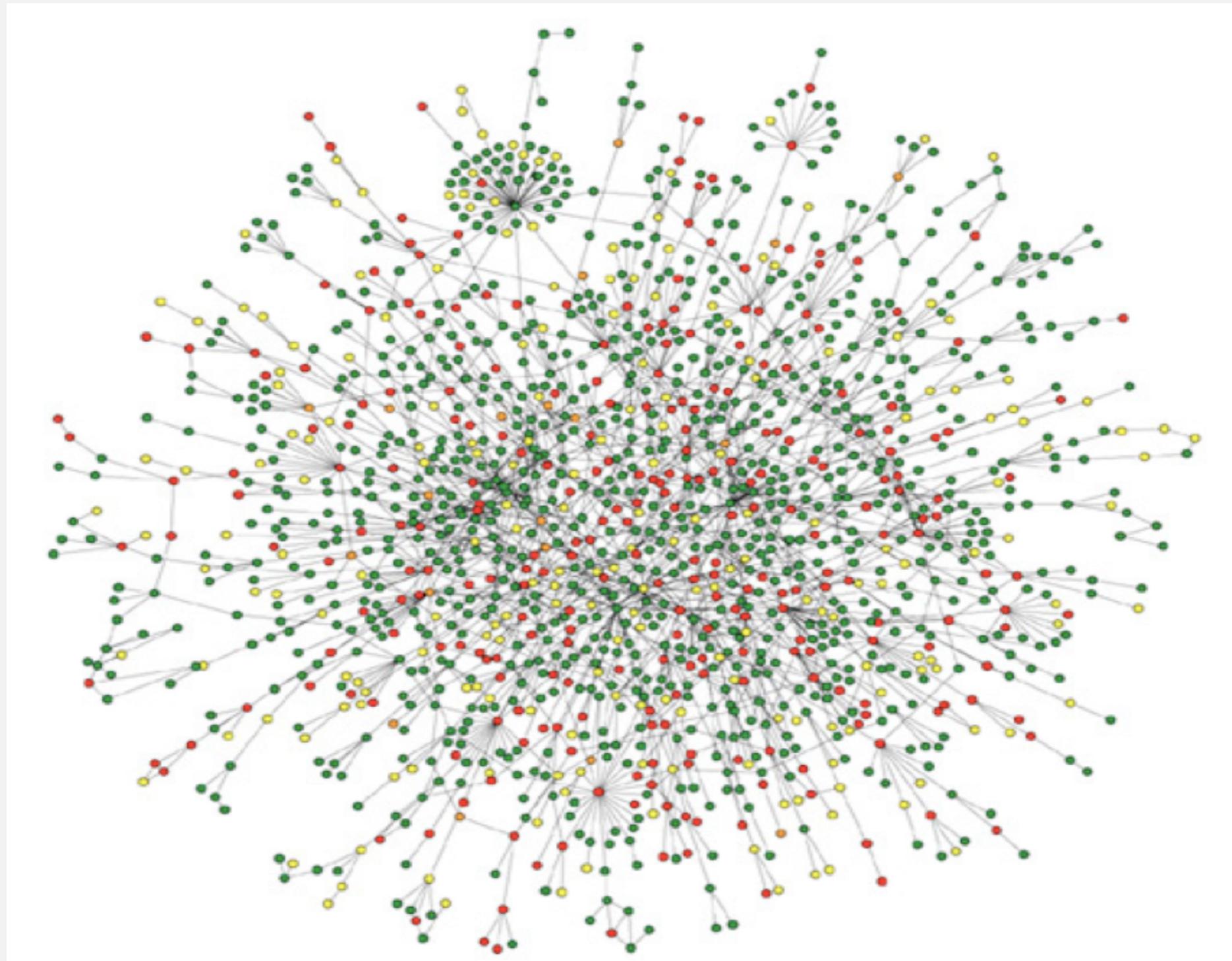


“Visualizing Friendships” by Paul Butler

Protein-protein interaction network

LO 9A.1, 9A.3

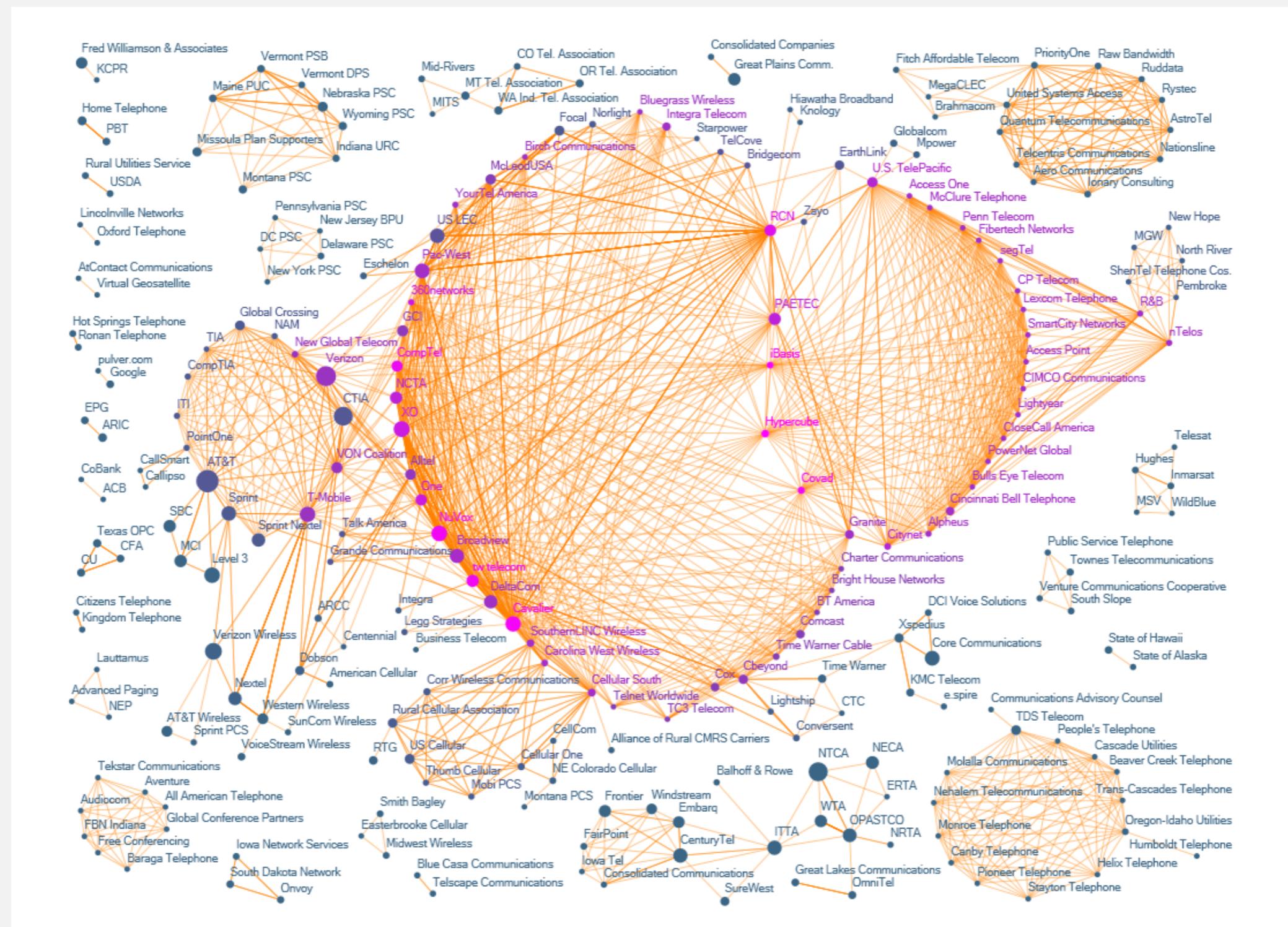
Vertex = protein; edge = interaction.



The evolution of FCC lobbying coalitions

LO 9A.1, 9A.3

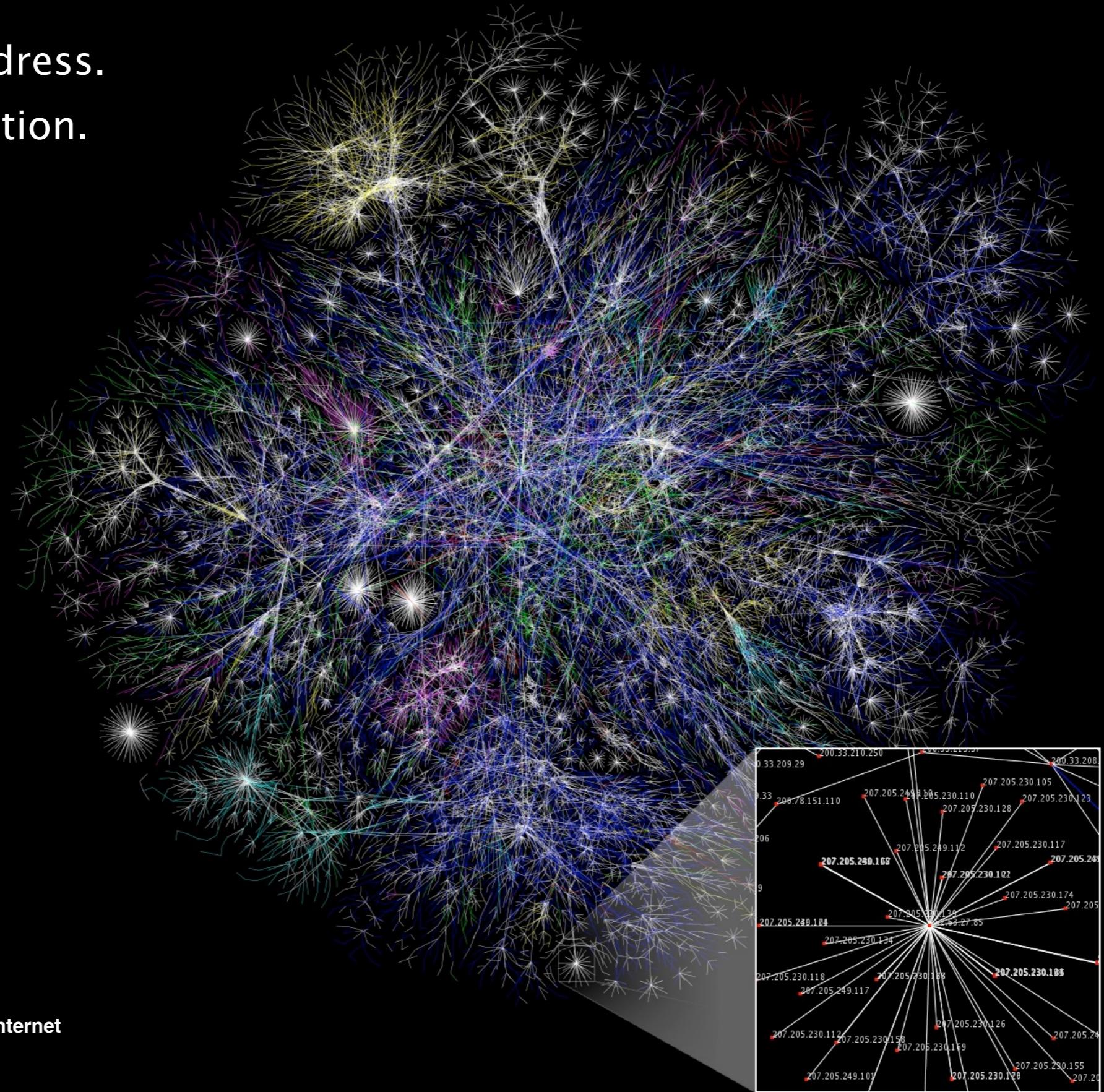
Vertex = company; edge = lobbying partner.



The Internet as mapped by the Opte Project

Vertex = IP address.

Edge = connection.



Graph applications

LO 9A.3

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

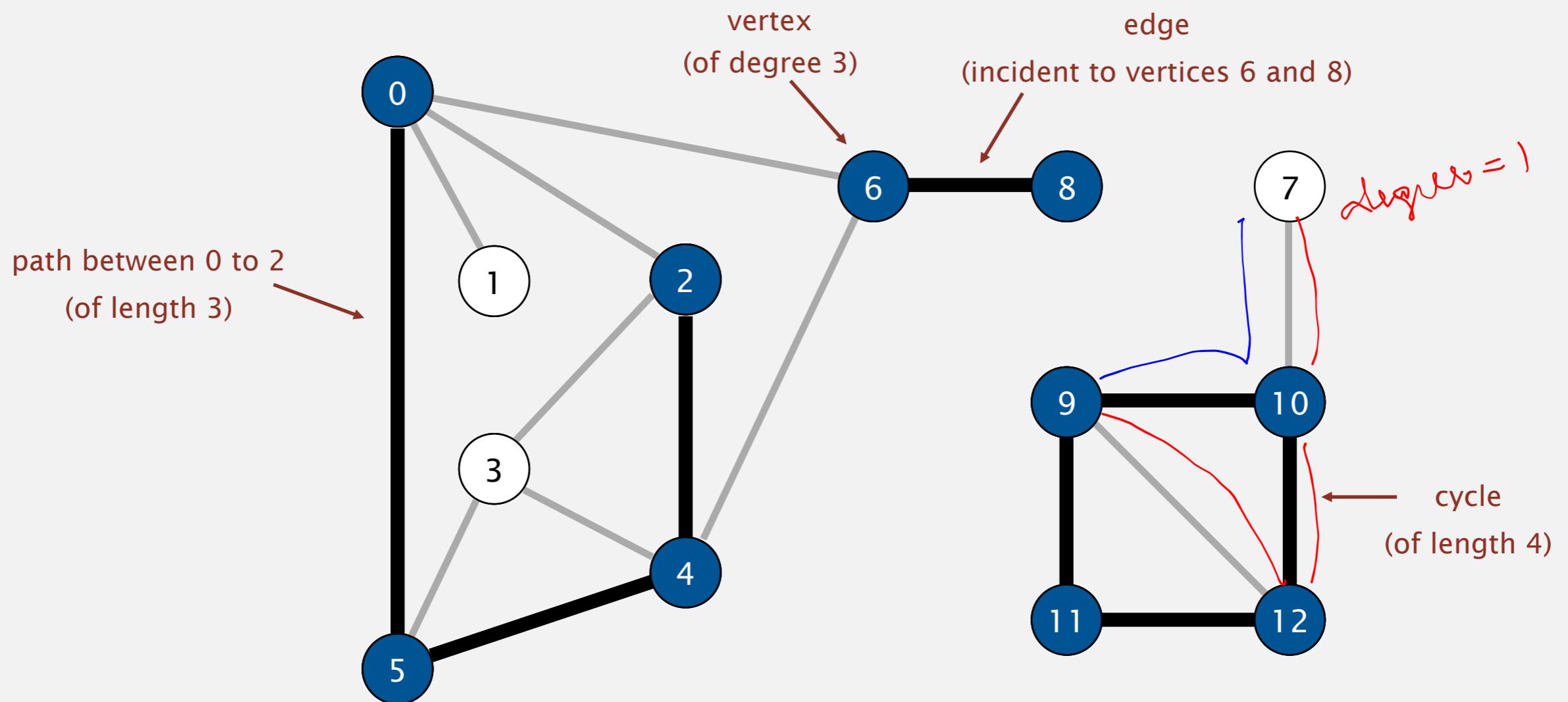
Graph terminology

Graph. Set of **vertices** connected pairwise by **edges**.

Path. Sequence of vertices connected by edges.

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

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



Some graph-processing problems

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

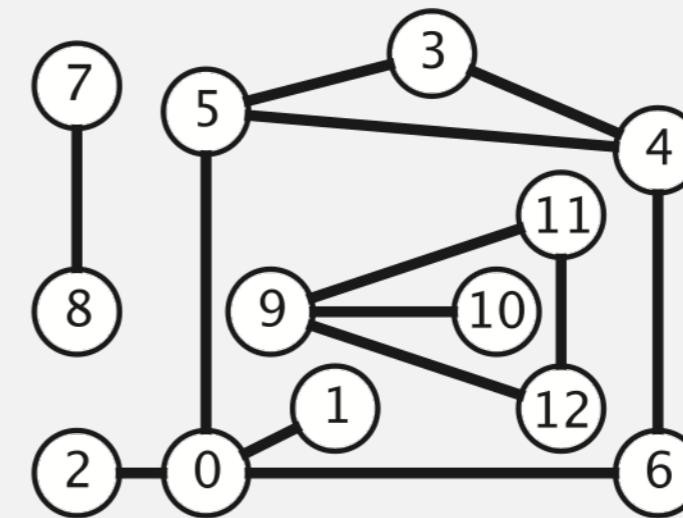
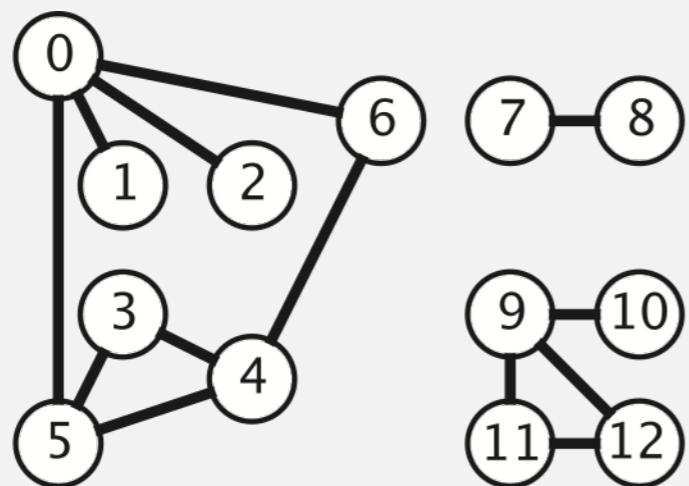
Challenge. Which graph problems are easy? difficult? intractable?

UNDIRECTED GRAPHS

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

Graph representation

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



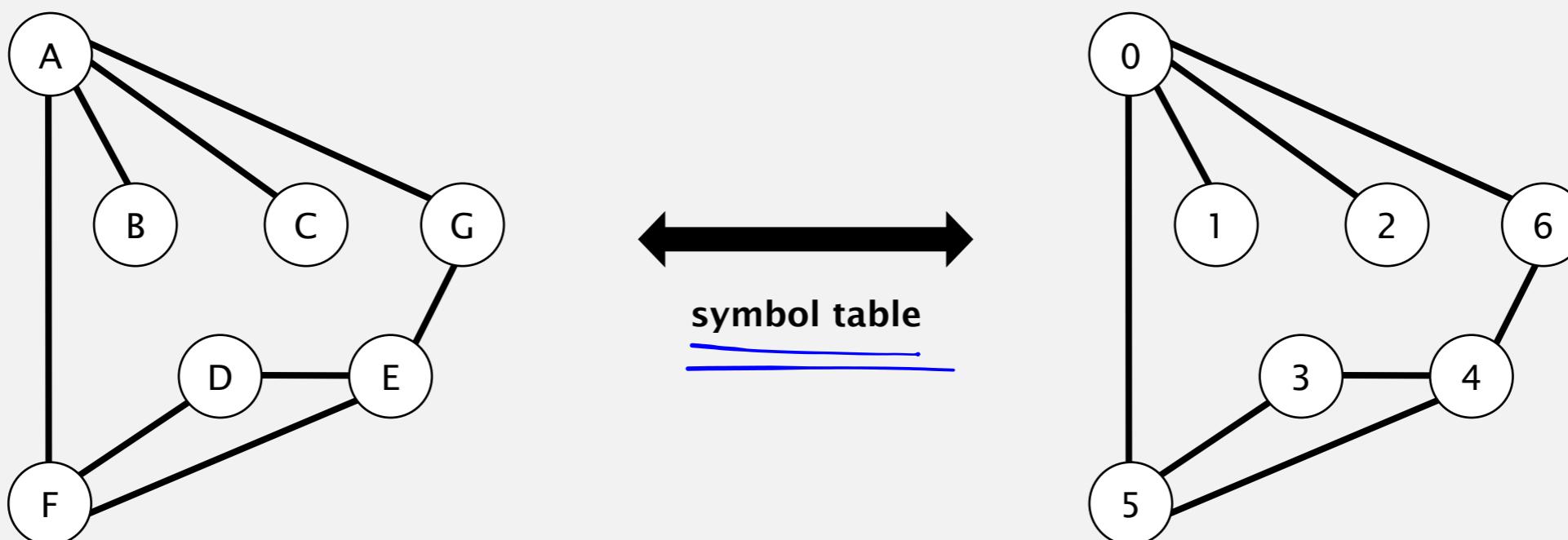
two drawings of the same graph

Caveat. Intuition can be misleading.

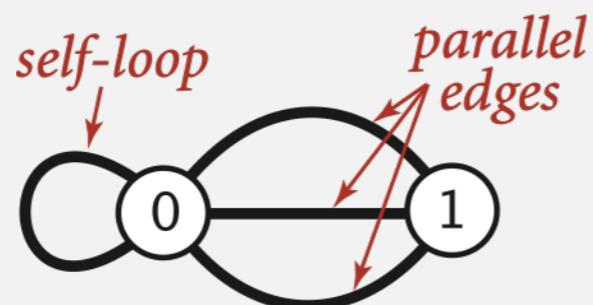
Graph representation

Vertex representation.

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



Anomalies.



```
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) → α, β, γ

vertices adjacent to v

int V()

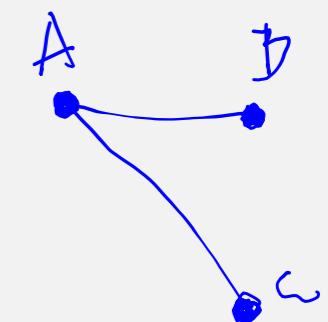
number of vertices

int E()

number of edges

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

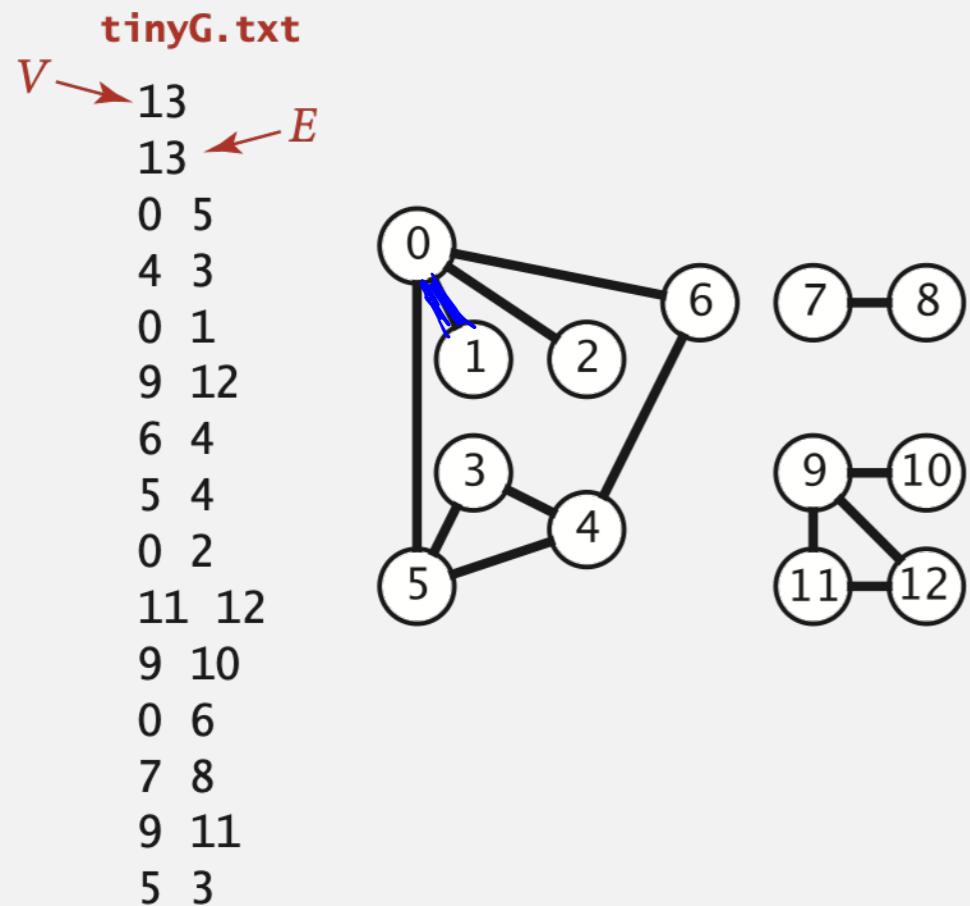
vertices adjacent to v



$$\begin{aligned} \text{degree}(A) &= 2 \\ \text{degree}(B) &= 1 \\ \text{degree}(C) &= 1 \end{aligned}$$

Graph API: sample client

Graph input format.



% java Test tinyG.txt

0-6

0-2

0-1

0-5

1-0

2-0

3-5

3-4

:

12-11

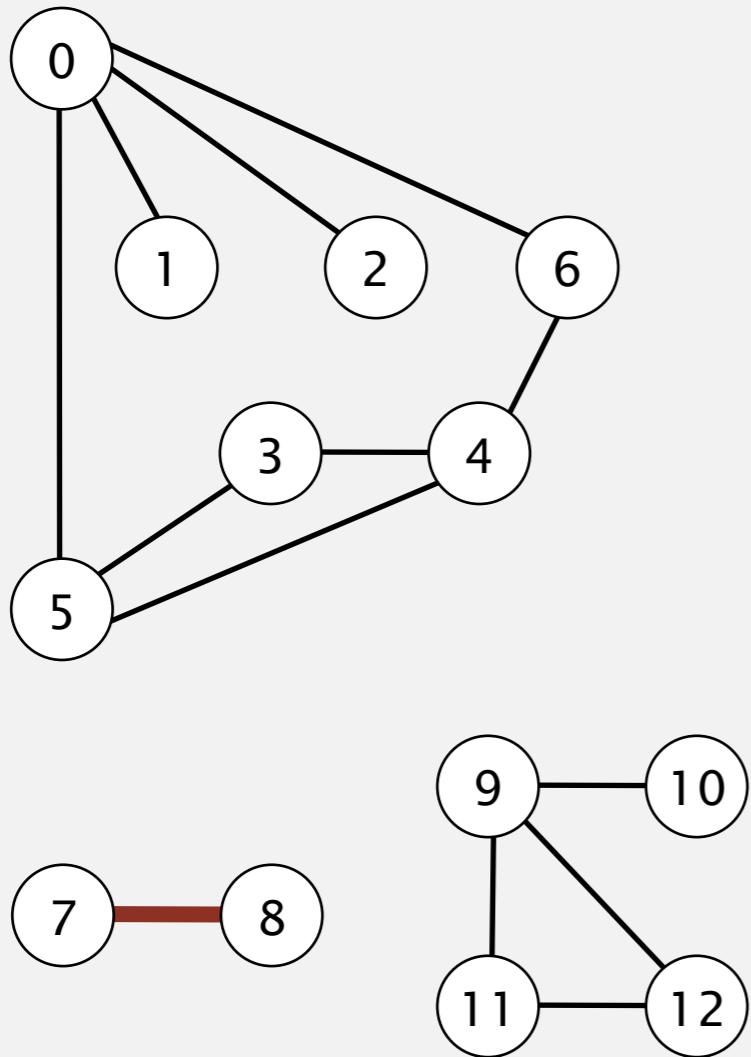
12-9

```
In in = new In(args[0]);
Graph G = new Graph(in);           ← read graph from
                                         input stream

for (int v = 0; v < G.V(); v++)
    for (int w : G.adj(v))        1,2,5,6
        StdOut.println(v + "-" + w); ← print out each
                                         edge (twice)
```

Graph representation: set of edges

Maintain a list of the edges (linked list or array). (2)
(1)

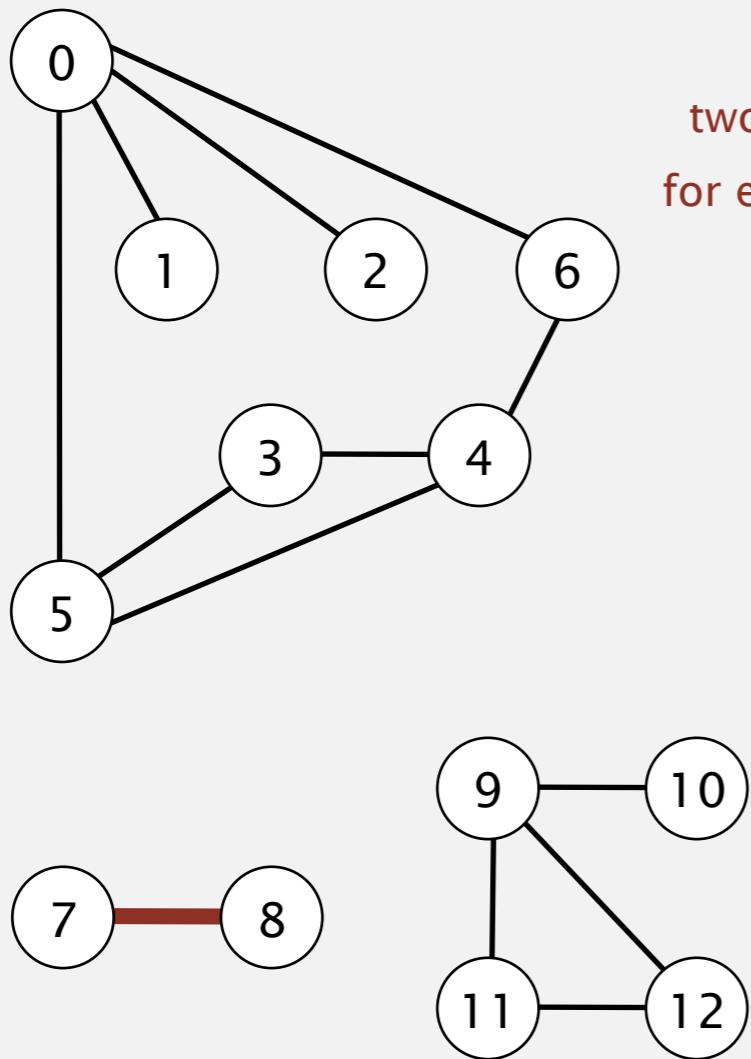


0 1
0 2
0 5
0 6
3 4
3 5
4 5
4 6
7 8
9 10
9 11
9 12
11 12

Graph representation: adjacency matrix

LO 9A.5

Maintain a two-dimensional V -by- V boolean array;
for each edge $v-w$ in graph: $\text{adj}[v][w] = \text{adj}[w][v] = \text{true}$.



two entries
for each edge

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	0	0	0	0	0	0	0	0	0
7	1	0	0	0	1	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	1	0	0	0	0
9	0	0	0	0	0	0	0	0	1	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	1	1	1
12	0	0	0	0	0	0	0	0	0	0	1	0	0
13	0	0	0	0	0	0	0	0	0	0	1	0	1

Undirected graphs: quiz 2

Which is 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

	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

B. $E + V$

C. V^2

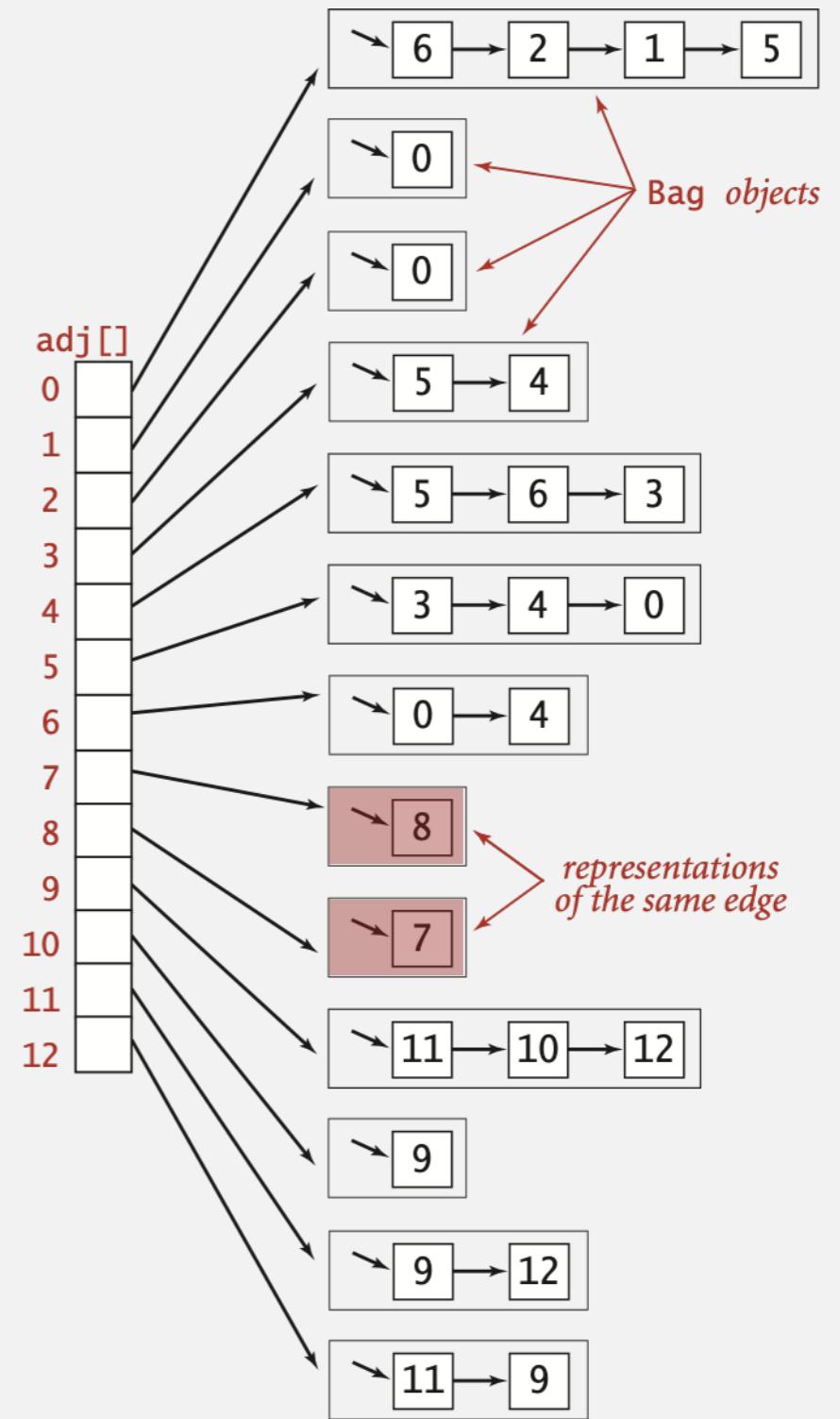
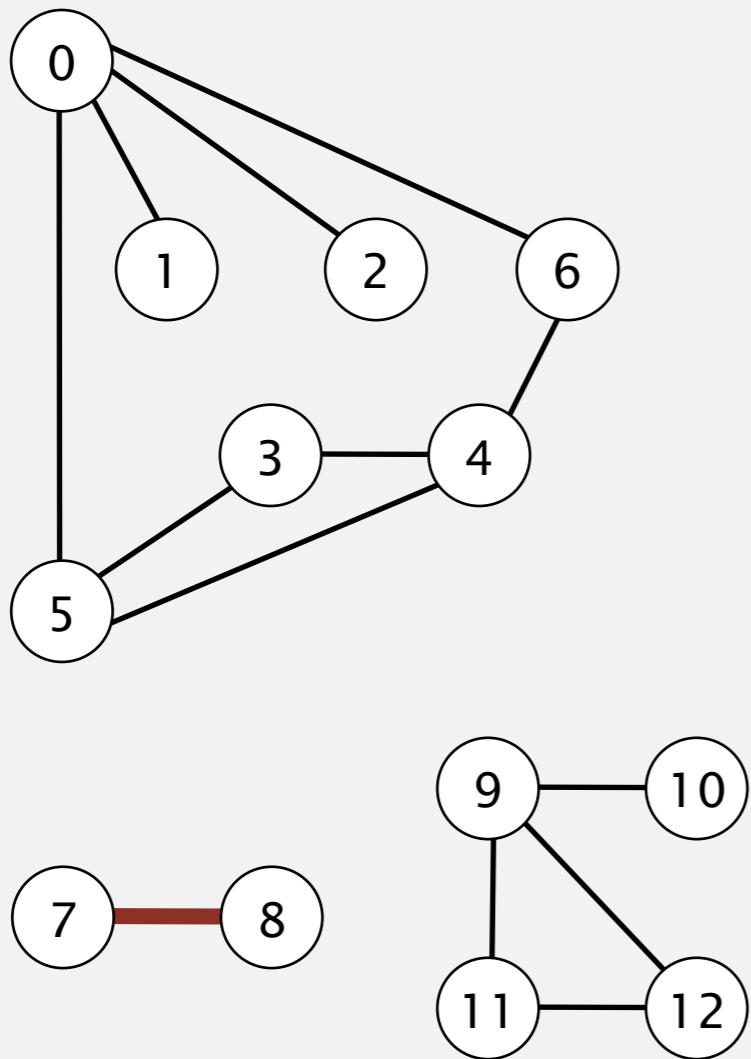
D. VE

adjacency-matrix representation

Graph representation: adjacency lists

LO 9A.6

Maintain vertex-indexed array of lists.



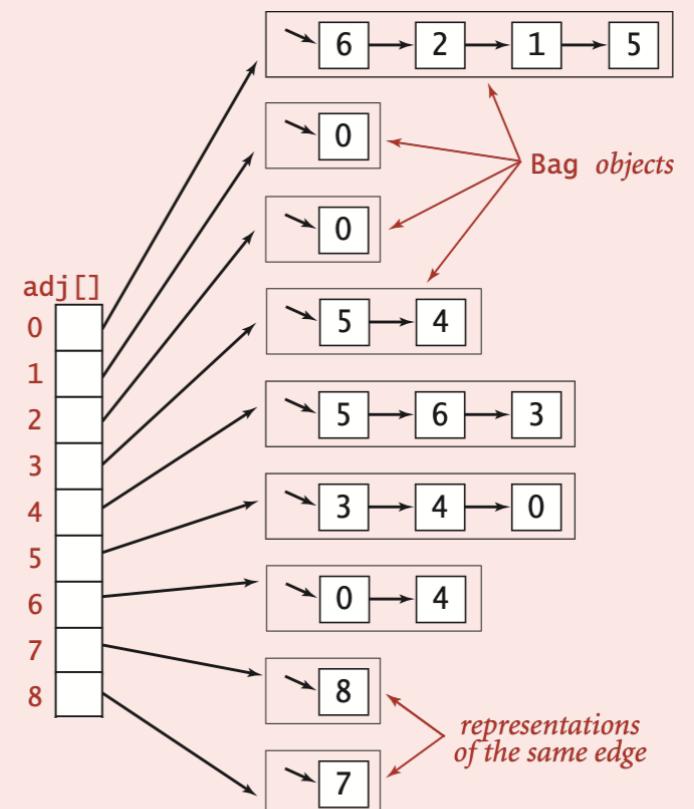
Undirected graphs: quiz 3

Which is 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. VE



Graph representations

In practice. Use adjacency-lists representation.

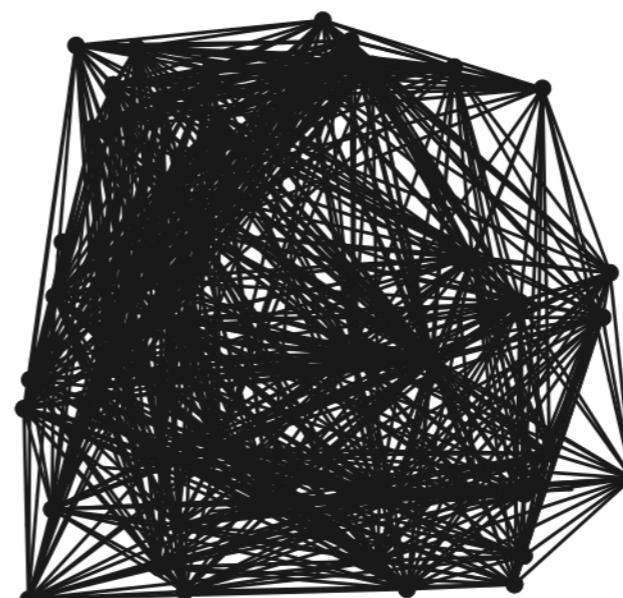
- Algorithms based on iterating over vertices adjacent to v .
- Real-world graphs tend to be **sparse** (not **dense**).

↑
proportional
to V edges ↑
proportional
to V^2 edges

sparse ($E = 200$)



dense ($E = 1000$)



Two graphs ($V = 50$)

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^\dagger	1	V
adjacency lists	$E + V$	1	$degree(v)$	$degree(v)$

\dagger disallows parallel edges

Adjacency-list graph representation: Java implementation

LO 9A.7

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

adjacency lists
(using Bag data type)

```
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>();
}
```

create empty graph
with V vertices

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

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

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

iterator for vertices adjacent to v

```
}
```

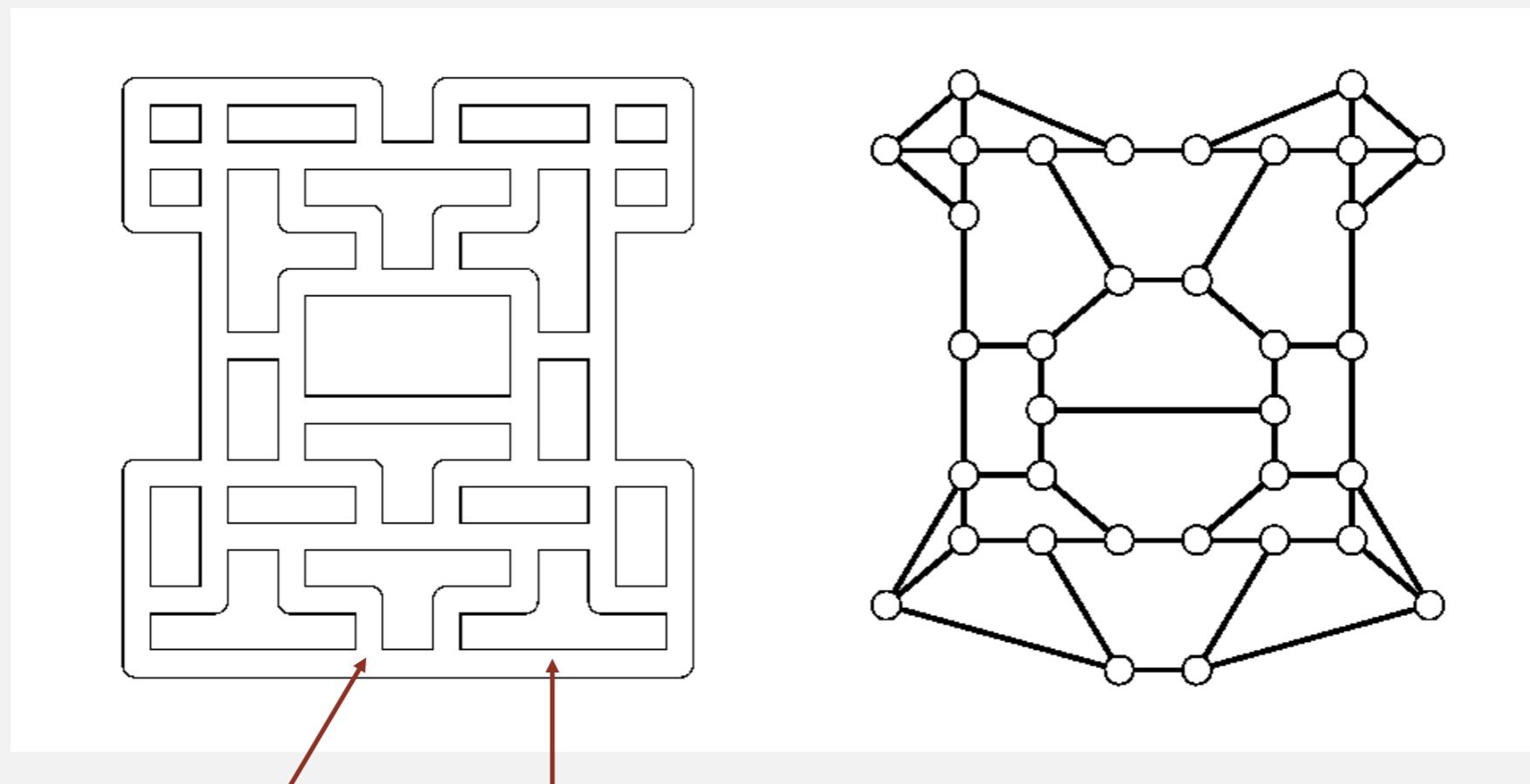
UNDIRECTED GRAPHS

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

Maze exploration

Maze graph.

- Vertex = intersection.
- Edge = passage.



Goal. Explore every intersection in the maze.

Maze exploration: National Building Museum

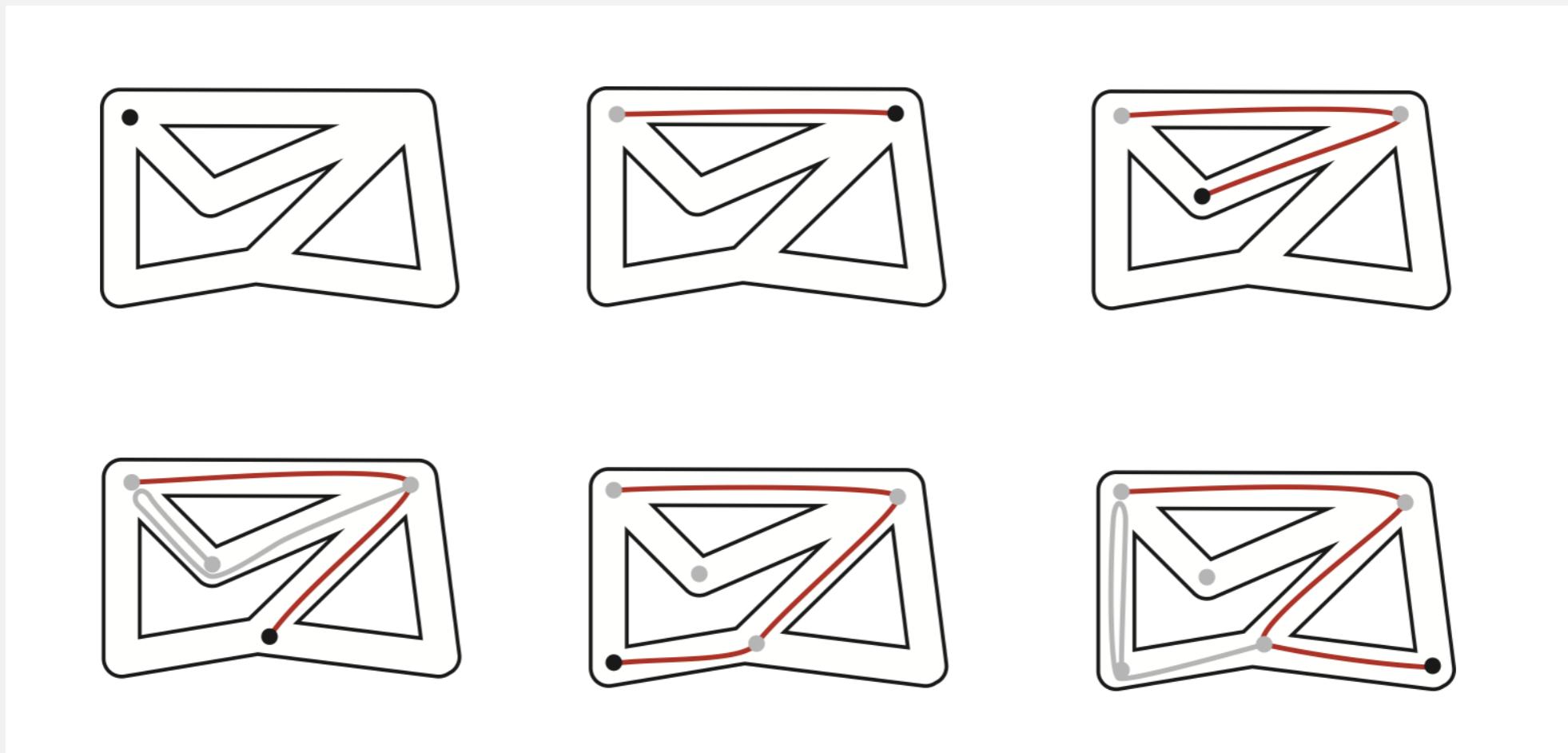


<http://www.smithsonianmag.com/travel/winding-history-maze-180951998/?no-ist>

Trémaux maze exploration

Algorithm.

- Unroll a ball of string behind you.
- Mark each newly discovered intersection and passage.
- Retrace steps when no unmarked options.



Trémaux maze exploration

Algorithm.

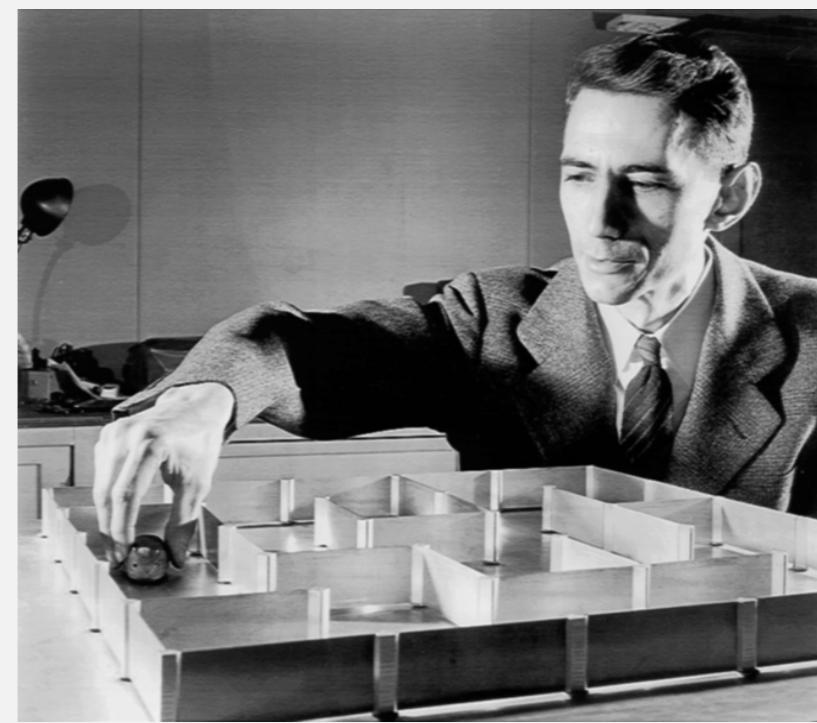
- Unroll a ball of string behind you.
- Mark each newly discovered intersection and passage.
- Retrace steps when no unmarked options.

First use? Theseus entered Labyrinth to kill the monstrous Minotaur; Ariadne instructed Theseus to use a ball of string to find his way back out.



The Cretan Labyrinth (with Minotaur)

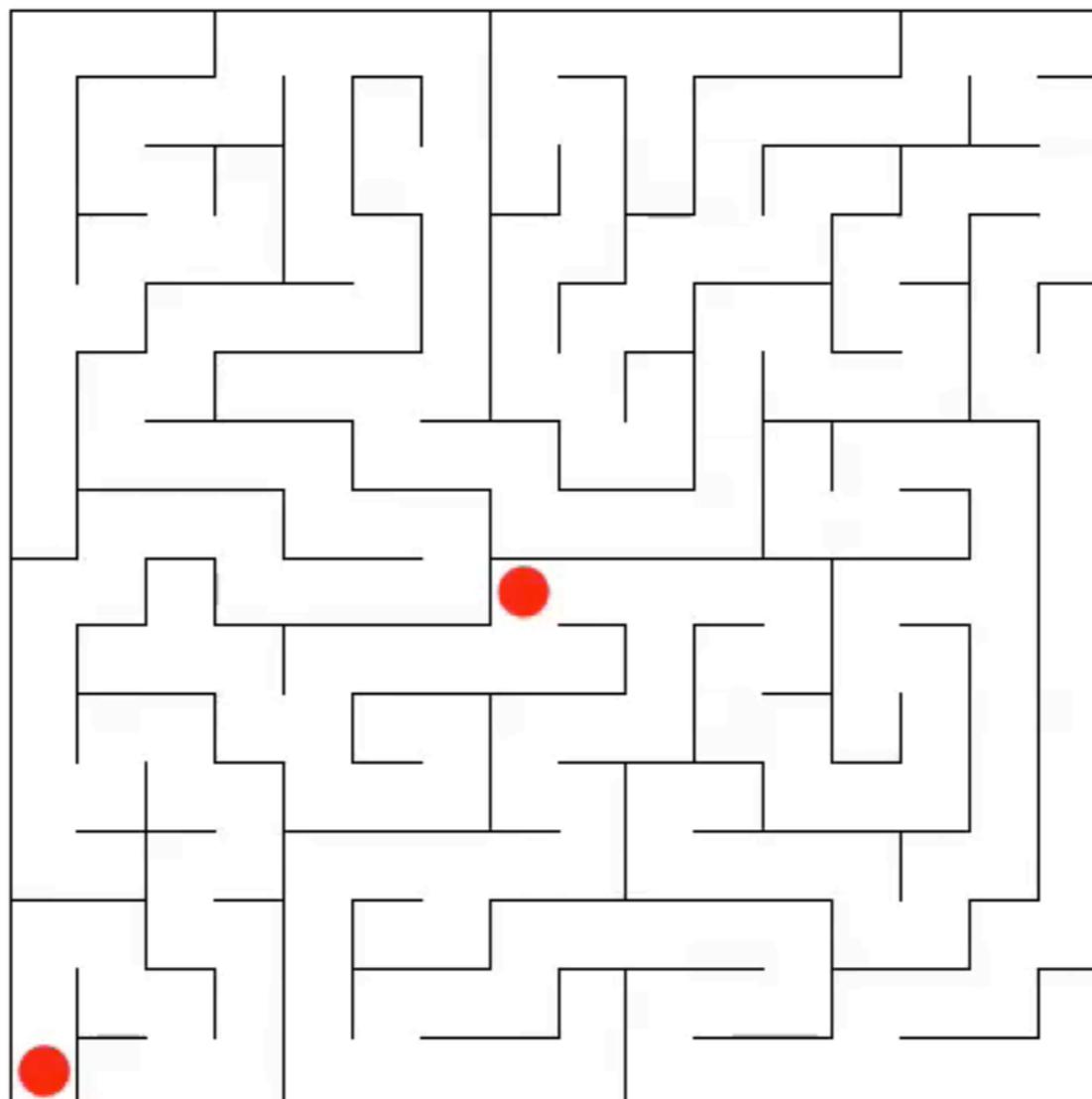
<http://commons.wikimedia.org/wiki/File:Minotaurus.gif>



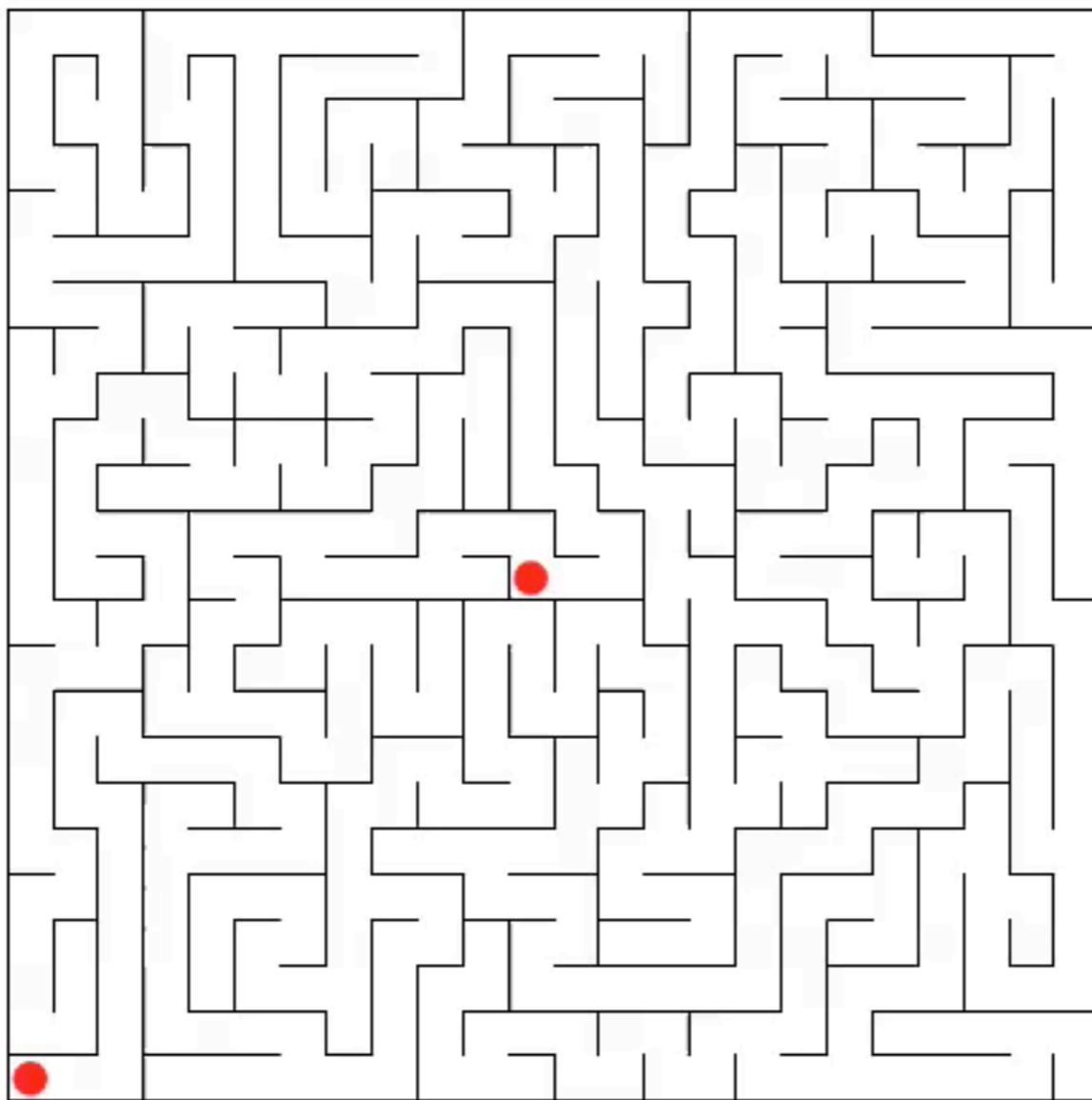
Claude Shannon (with electromechanical mouse)

<http://www.corp.att.com/attlabs/reputation/timeline/16shannon.html>

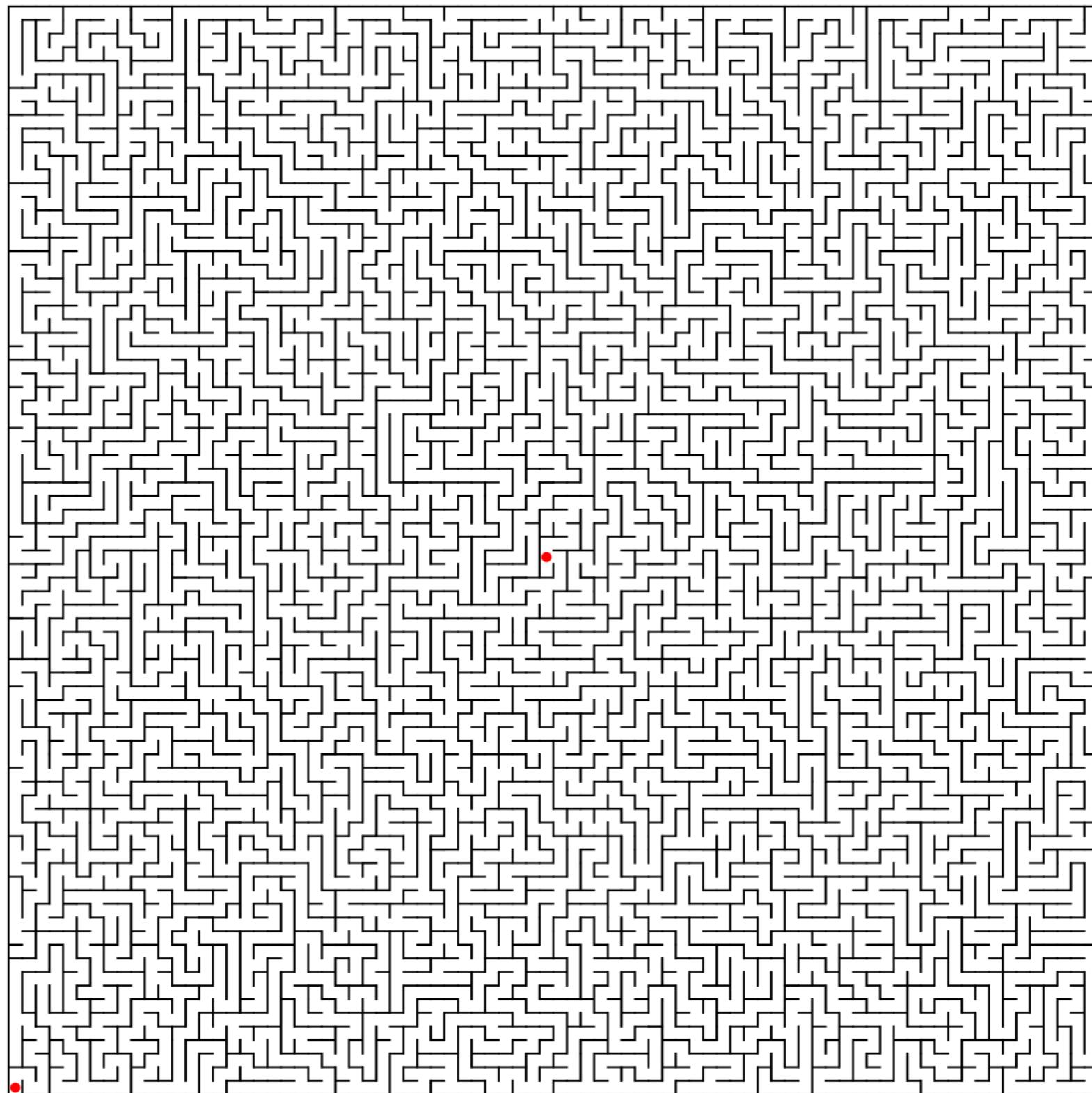
Maze exploration: easy



Maze exploration: medium



Maze exploration: challenge for the bored



Depth-first search

Goal. Systematically traverse a graph.

Idea. Mimic maze exploration. ← function-call stack
plays role of ball of string

DFS (to visit a vertex v)

Mark vertex v

**Recursively visit all unmarked
vertices w adjacent to v**

Typical applications.

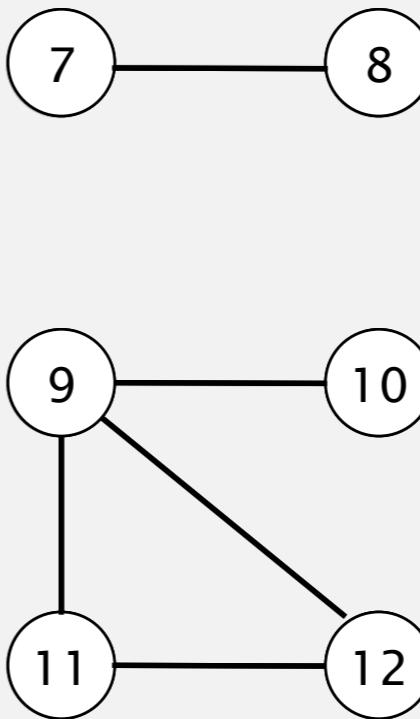
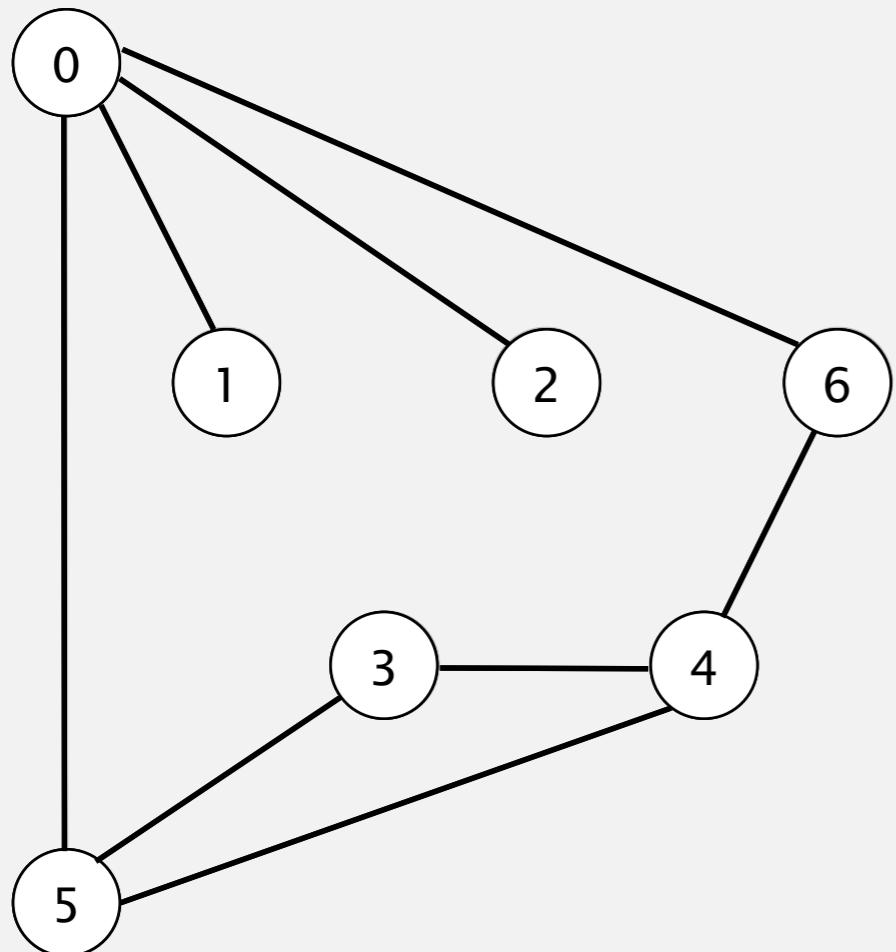
- Find all vertices connected to a given source vertex.
- Find a path between two vertices.

Design challenge. How to implement?

Depth-first search demo

To visit a vertex v :

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



tinyG.txt

$V \rightarrow$ 13
 $E \leftarrow$ 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

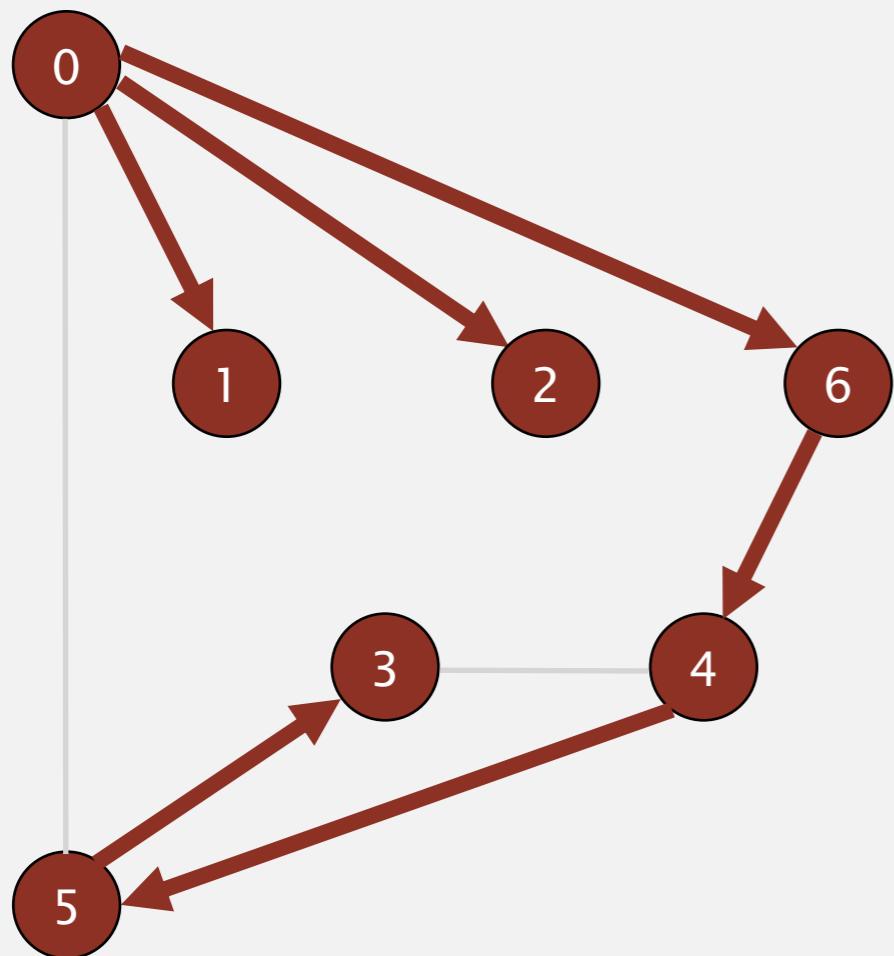
graph G

Depth-first search demo

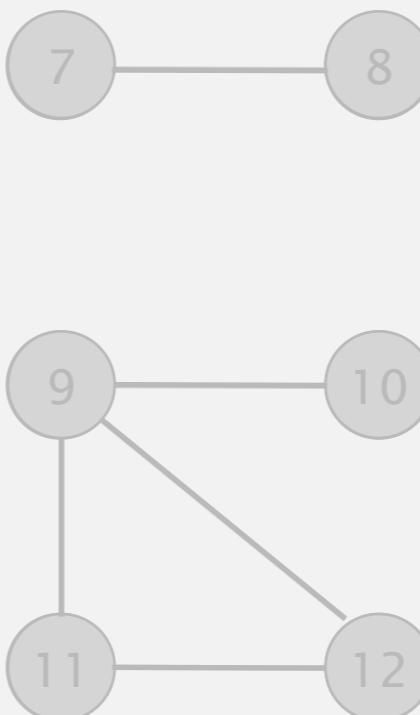
LO 9A.10

To visit a vertex v :

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



vertices reachable from 0



<code>v</code>	<code>marked[]</code>	<code>edgeTo[]</code>
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	-

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.
- Query the graph-processing routine for information.

```
public class Paths
```

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

```
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: 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.
 $(\text{edgeTo}[w] == v)$ means that edge $v-w$ taken to discover vertex w
- Function-call stack for recursion.

Depth-first search: Java implementation

DFS, BFS

LO 9A.11

```
public class DepthFirstPaths
```

```
{
```

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

marked[v] = true

```
    private int[] edgeTo; ←
```

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

```
    private int s;
```

```
    public DepthFirstPaths(Graph G, int s)
```

```
{
```

```
    ...
```

```
    dfs(G, s);
```

```
}
```

```
    private void dfs(Graph G, int v)
```

```
{
```

✓

```
    marked[v] = true; ←
```

 initialize data structures

2 E

```
    for (int w : G.adj(v)) ←
```

 find vertices connected to s

2 E

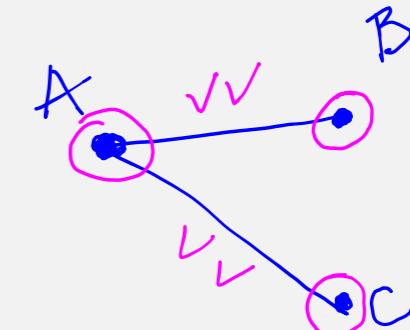
```
    if (!marked[w])
```

```
    {  
        edgeTo[w] = v;  
        dfs(G, w); ←
```

recursive DFS does the work

```
}
```

```
}
```



✓ vertices

E edges

✓ + 2E

O (V + E)

DFS

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

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

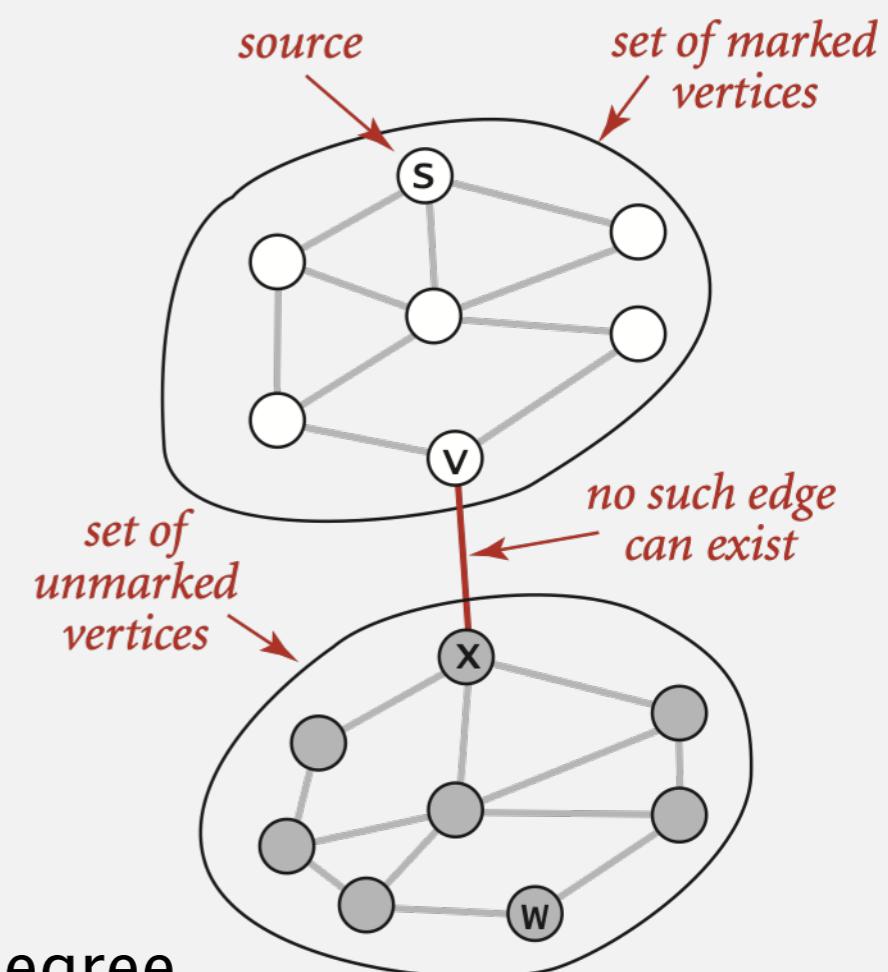
    private void dfs(Graph G, int v)
    {
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w])
            {
                edgeTo[w] = v;
                dfs(G, w);
            }
    }
}
```

Depth-first search: properties

Proposition. DFS marks all vertices connected to s in time proportional to the sum of their degrees (plus time to initialize the marked[] array).

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).



Pf. [running time]

- Each vertex connected to s is visited once.
- Visiting a vertex takes time proportional to its degree.

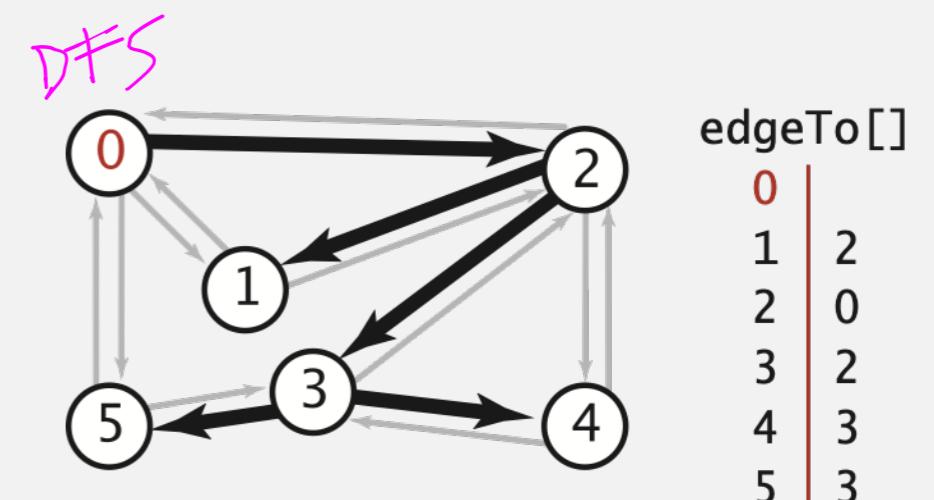
Depth-first search: properties

Proposition. After DFS, can check if vertex v is connected to s in constant time and 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;
}
```



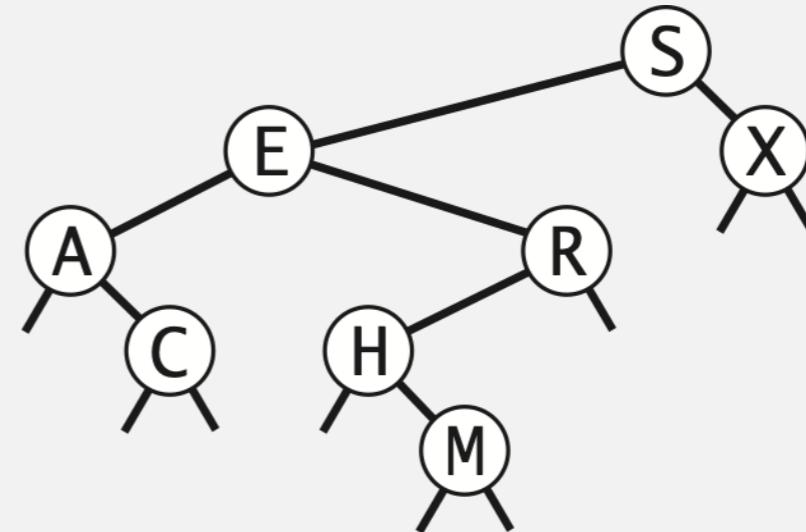
UNDIRECTED GRAPHS

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

Graph search

Tree traversal. Many ways to explore every vertex in 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

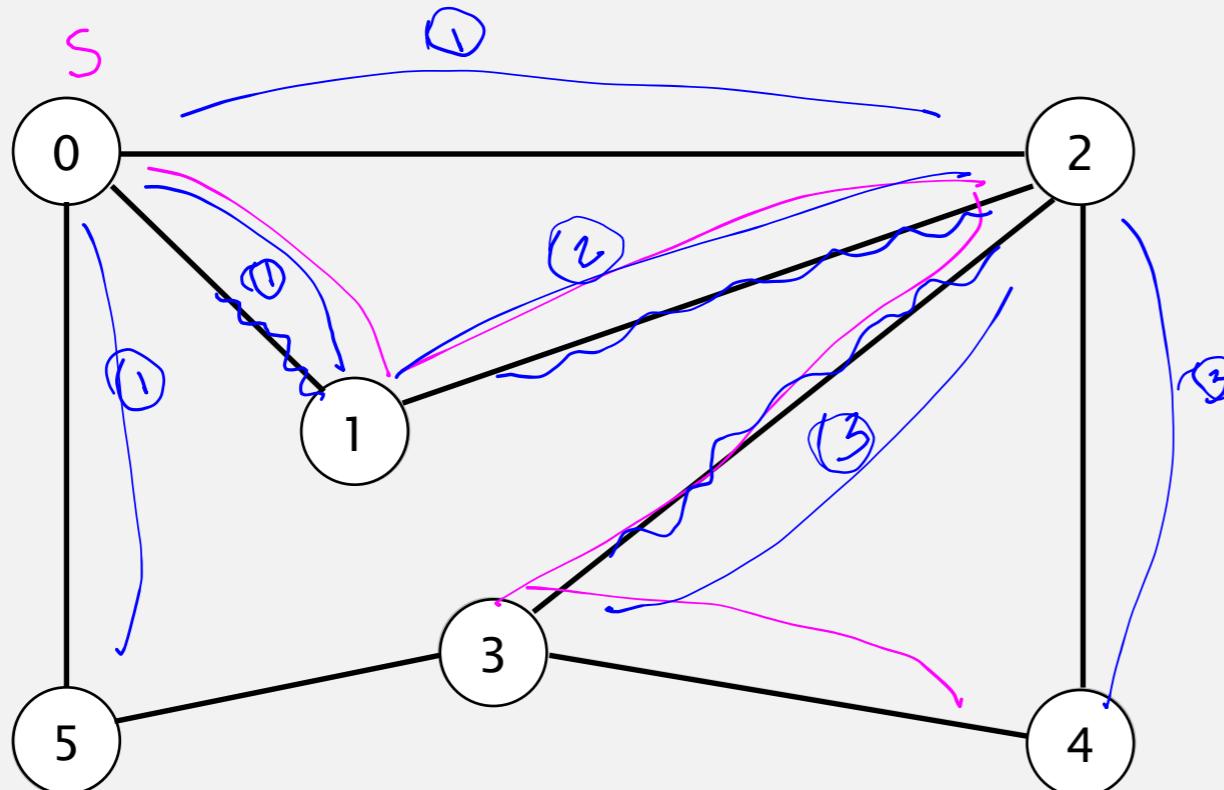


Graph search. Many ways to explore every vertex in a graph.

- **Preorder:** vertices in order of calls to $\text{dfs}(G, v)$.
- **Postorder:** vertices in order of returns from $\text{dfs}(G, v)$.
- **Level-order:** vertices in increasing order of distance from s .

Repeat until queue is empty:

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



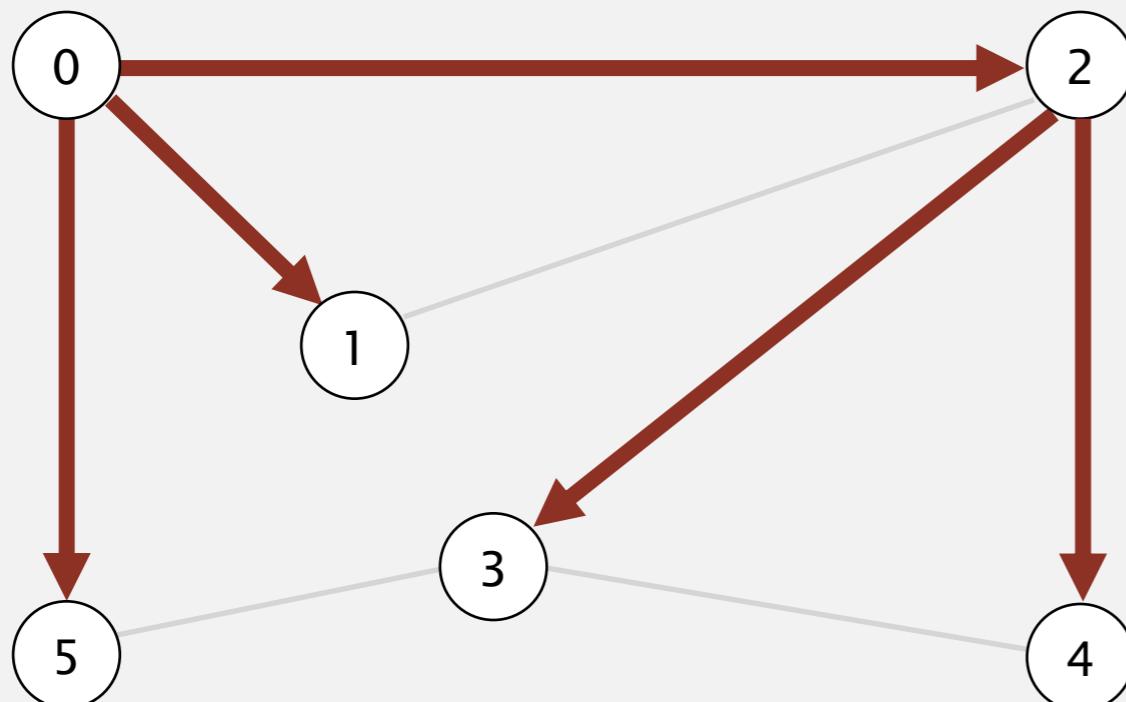
tinyCG.txt

<i>V</i> →	6	<i>E</i> →	8
	0 5		2 4
	2 3		2 3
	1 2		1 2
	0 1		0 1
	3 4		3 4
	3 5		3 5
	0 2		0 2

graph G

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

done

Repeat until queue is empty:

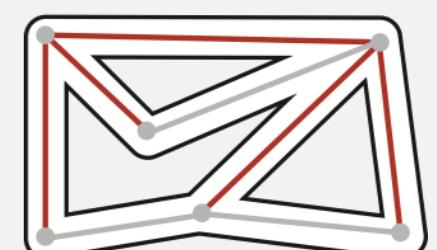
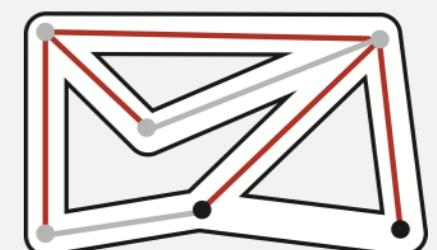
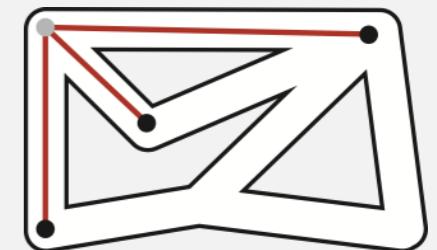
- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.

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 \Rightarrow dequeue (\downarrow)
- add each of v 's unmarked neighbors to the queue,
and mark them.



Breadth-first search: Java implementation

LO 9A.13

```
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; visiting
        distTo[s] = 0;

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

initialize FIFO queue of vertices to explore

found new vertex w via edge v-w

Breadth-first search: Java implementation

LO 9A.13

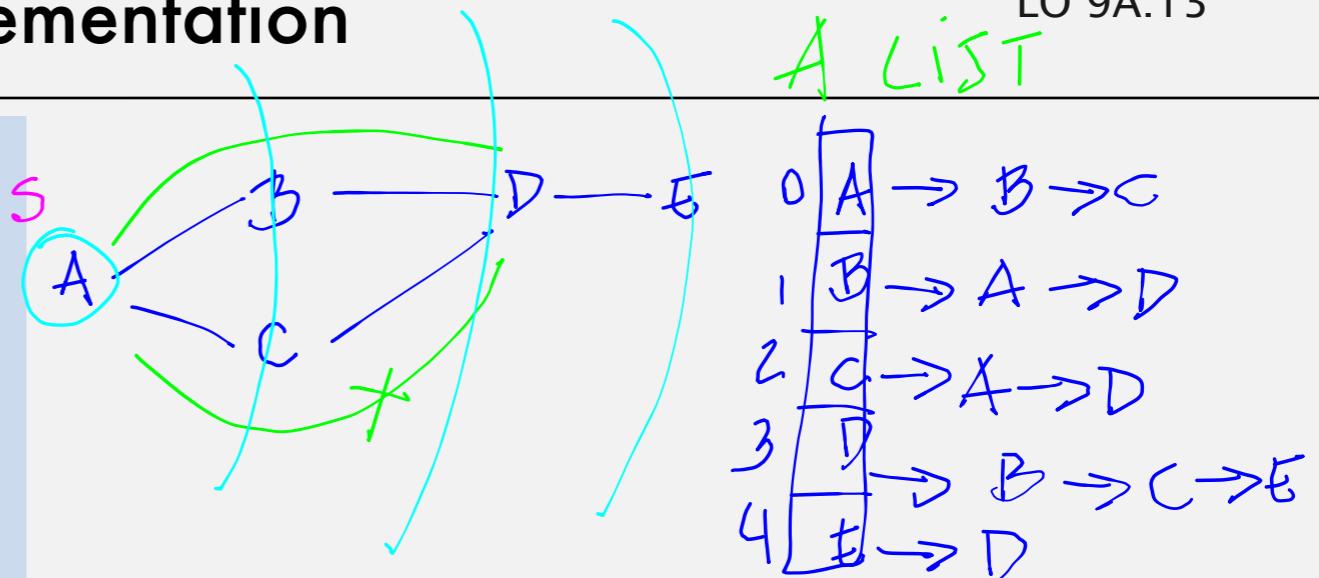
```

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

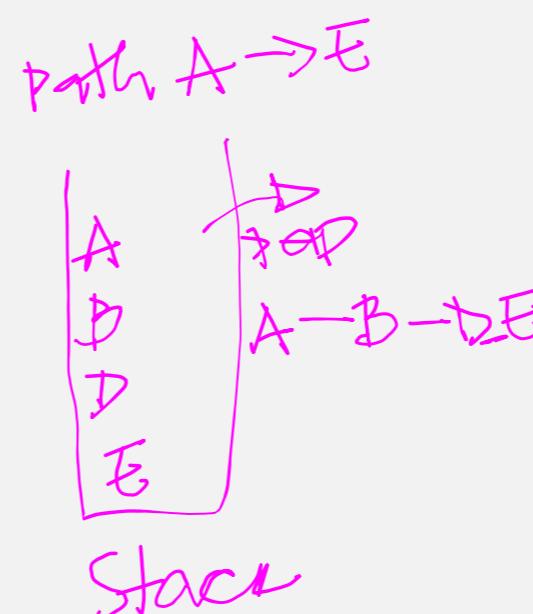
    private void bfs(Graph G, int s) {
        Queue<Integer> q = new Queue<Integer>();
        q.enqueue(s);
        marked[s] = true;
        distTo[s] = 0;

        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;
                }
            }
        }
    }
}

```



Queue
A B C D E



	marked	edgeTo	distTo
A	✓		0
B	✓	A	1
C	✓	A	1
D	✓	B	2
E	✓	D	3

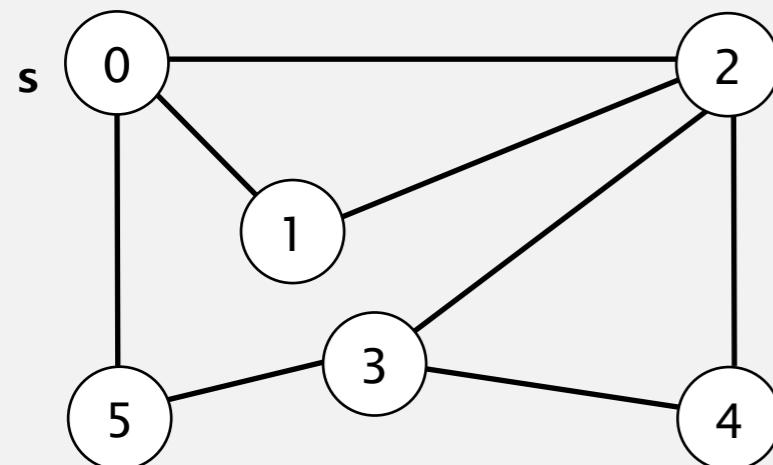
Breadth-first search properties

Q. In which order does BFS examine vertices?

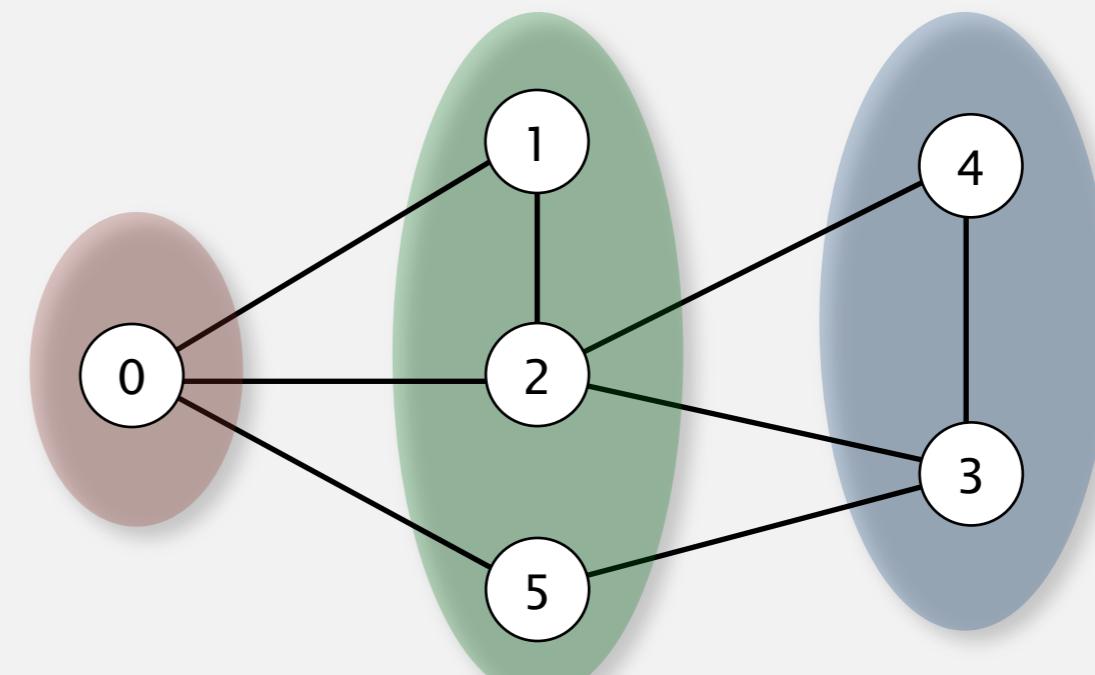
A. Increasing distance (number of edges) from s .

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

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



graph G



dist = 0

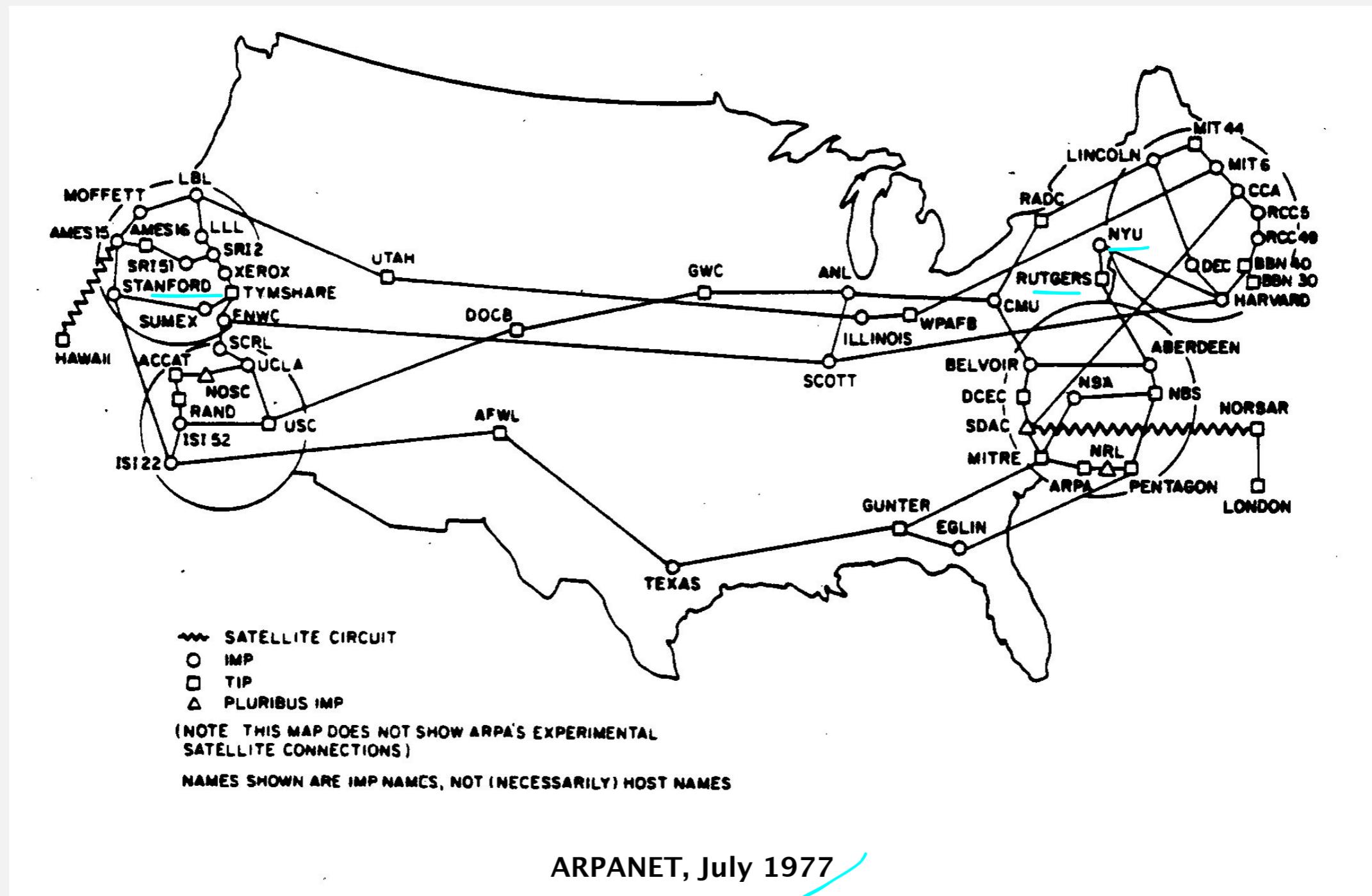
dist = 1

dist = 2

Breadth-first search application: routing

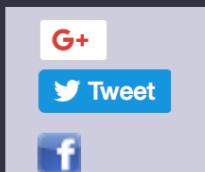
LO 9A.14

Fewest number of hops in a communication network.





Welcome
Credits
How it Works
Contact Us
Other stuff »



© 1999-2016 by Patrick Reynolds. All rights reserved.

[Bernard Chazelle](#) has a Bacon number of 3.

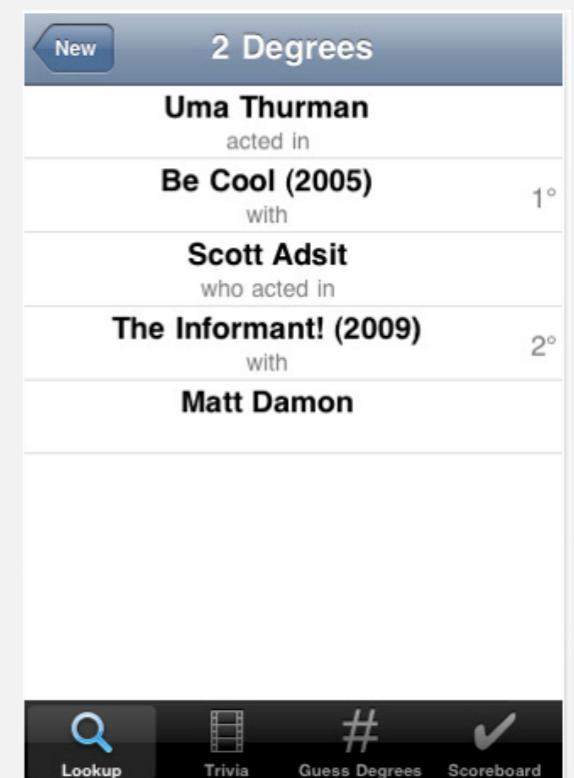
[Find a different link](#)



<http://oracleofbacon.org>



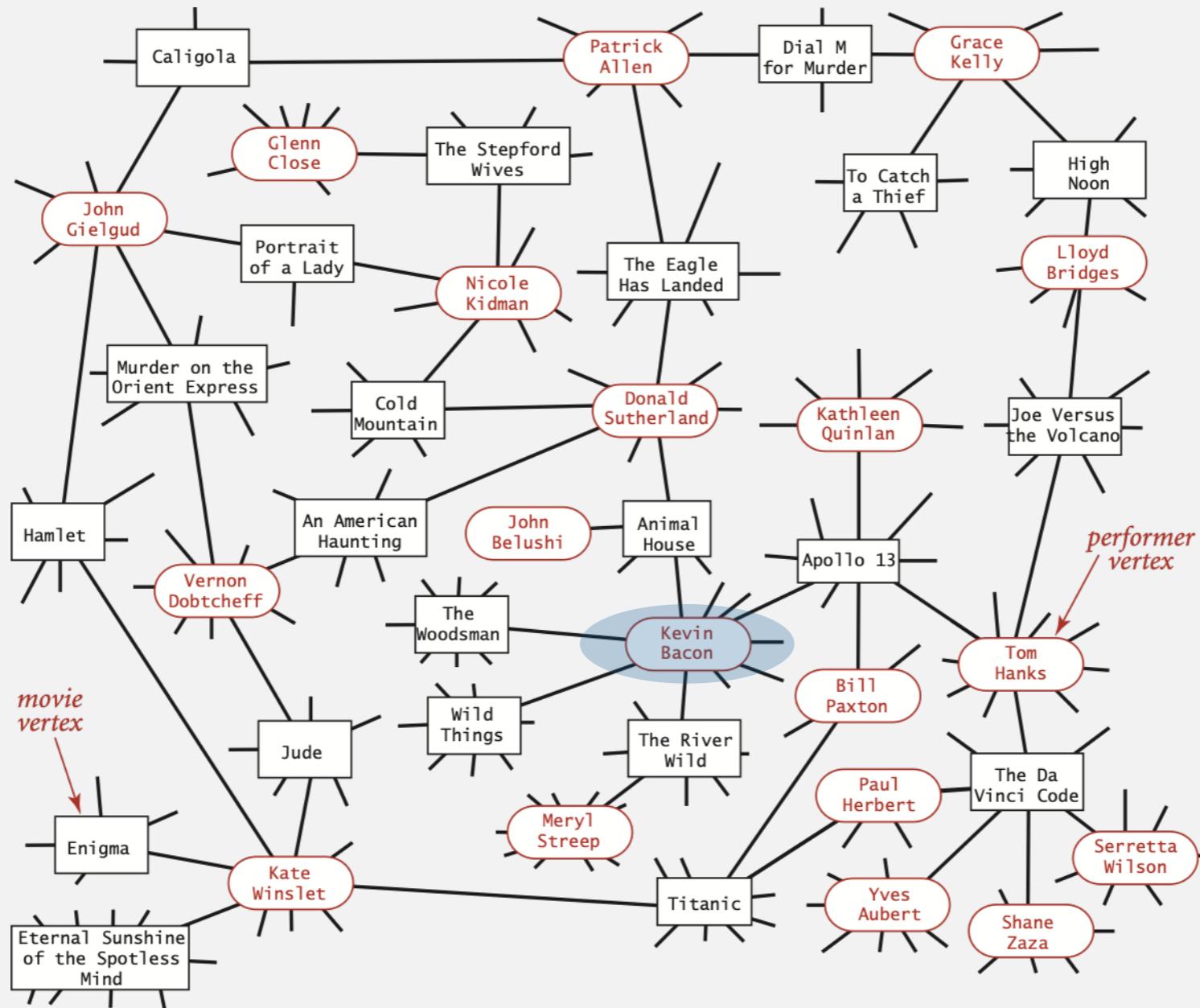
Endless Games board game



SixDegrees iPhone App

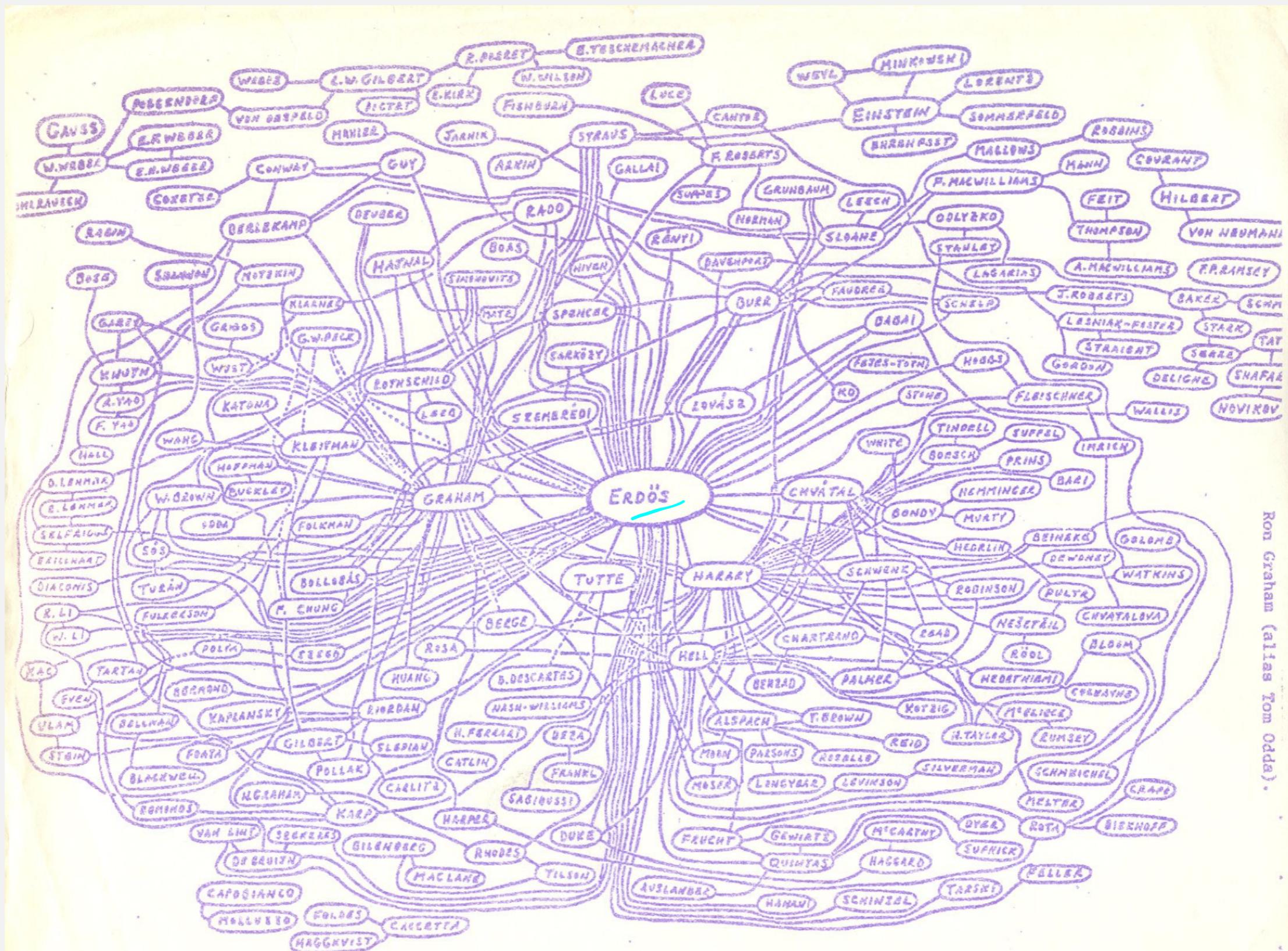
Kevin Bacon graph

- 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 $s = \text{Kevin Bacon}$.



Breadth-first search application: Erdős numbers

LO 9A.14



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

UNDIRECTED GRAPHS

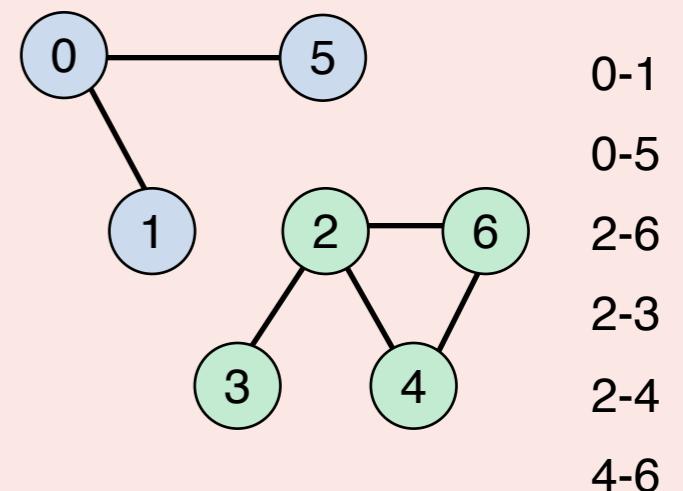
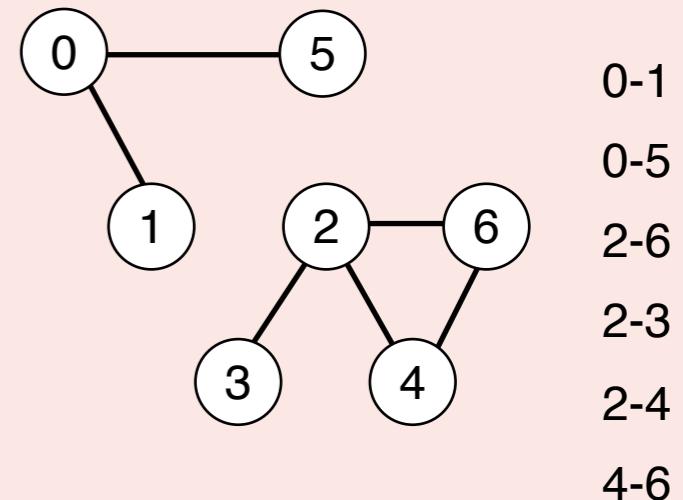
- ▶ *introduction*
- ▶ *graph API*
- ▶ *depth-first search*
- ▶ *breadth-first search*
- ▶ ***challenges***

Graph-processing challenge 1

Problem. Identify connected components.

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.

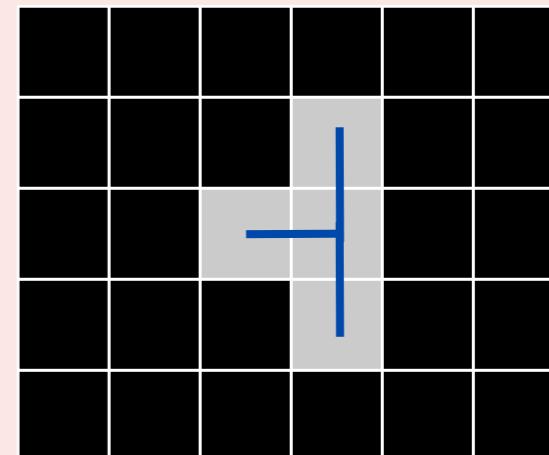
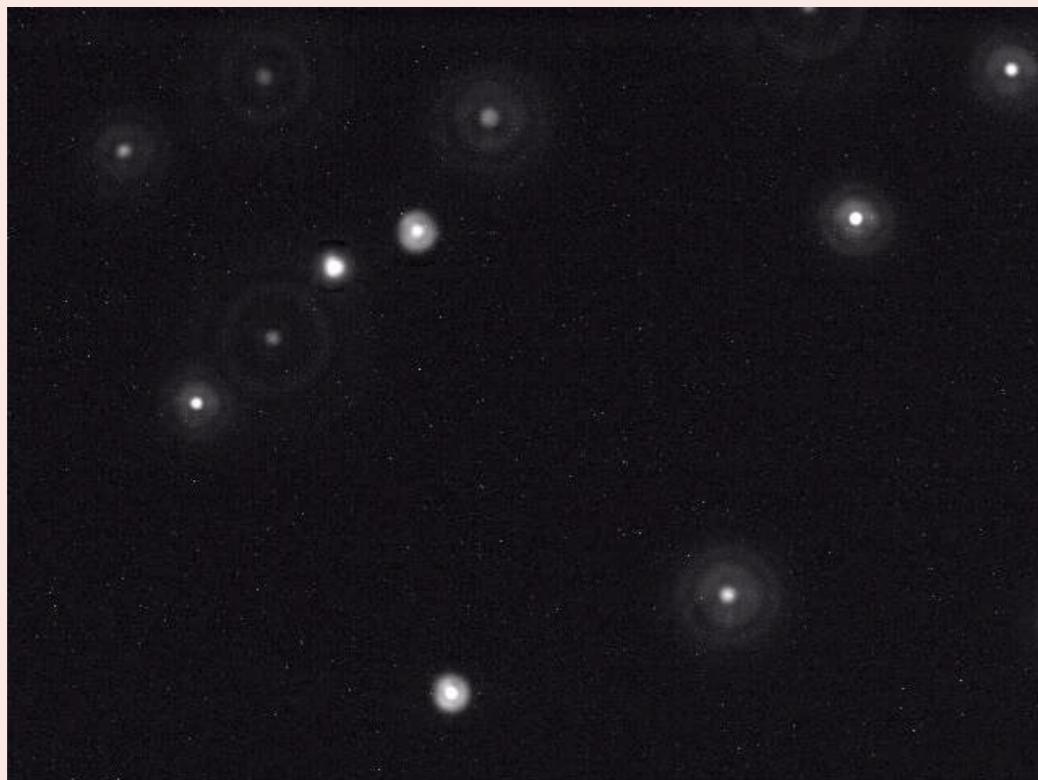


Graph-processing challenge 1

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.

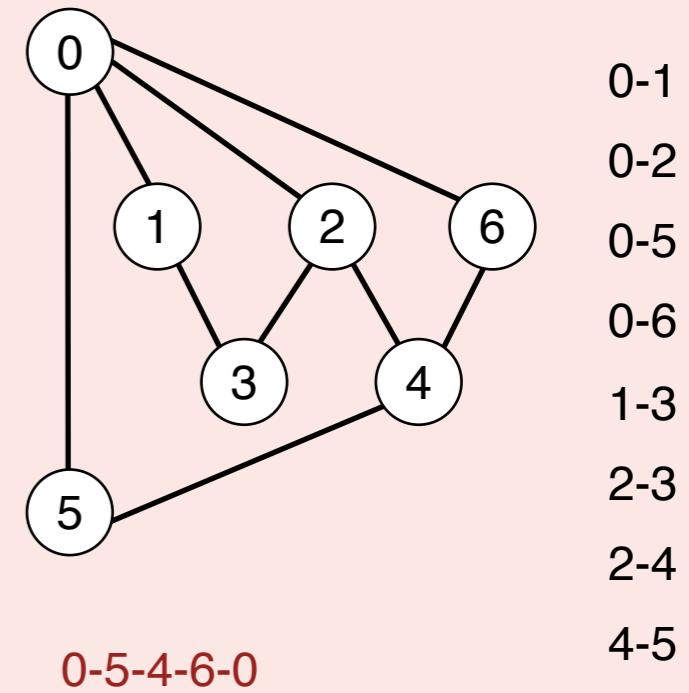


Graph-processing challenge 3

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

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.



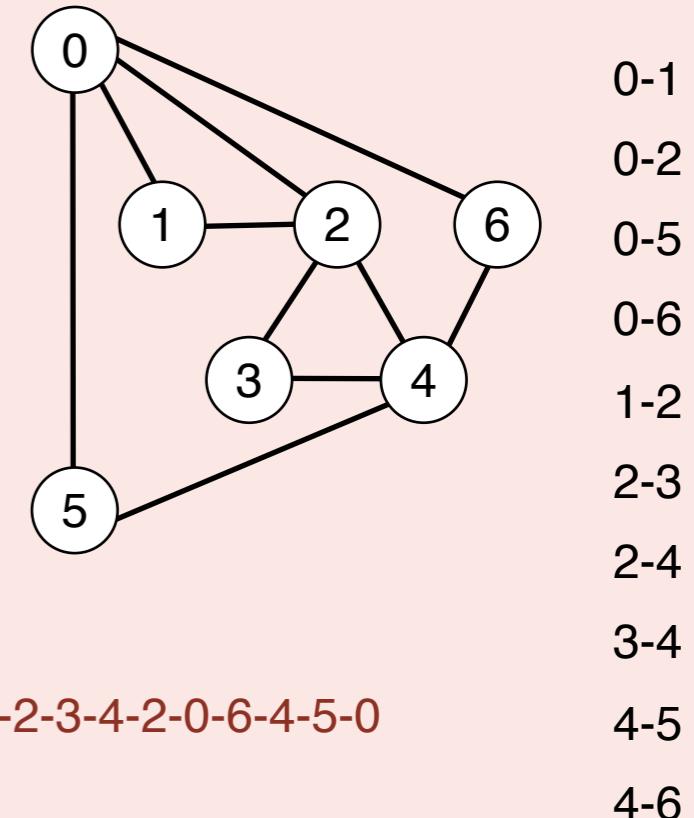
Graph-processing challenge 4

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

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.

(Leonhard Euler 1786)

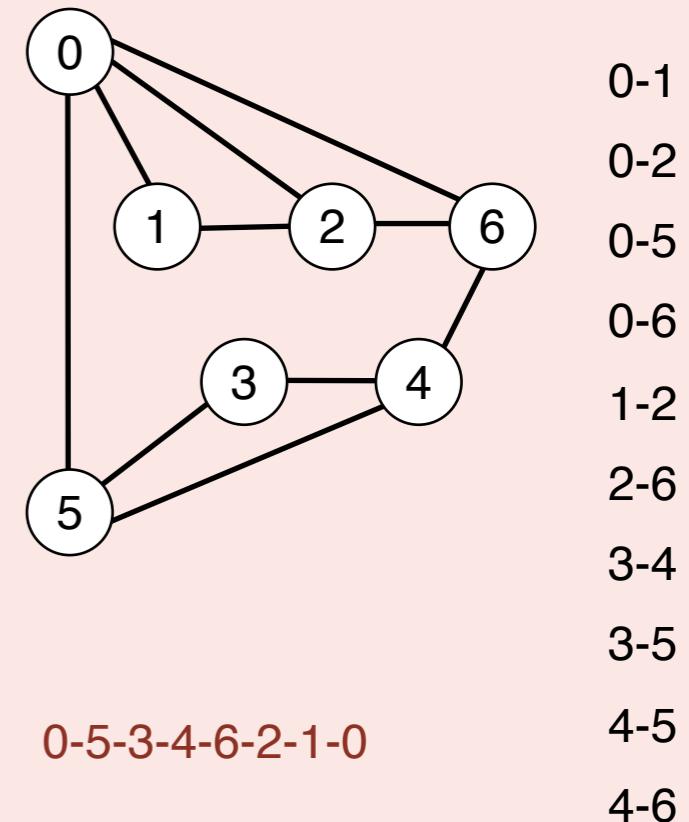


Graph-processing challenge 5

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

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.

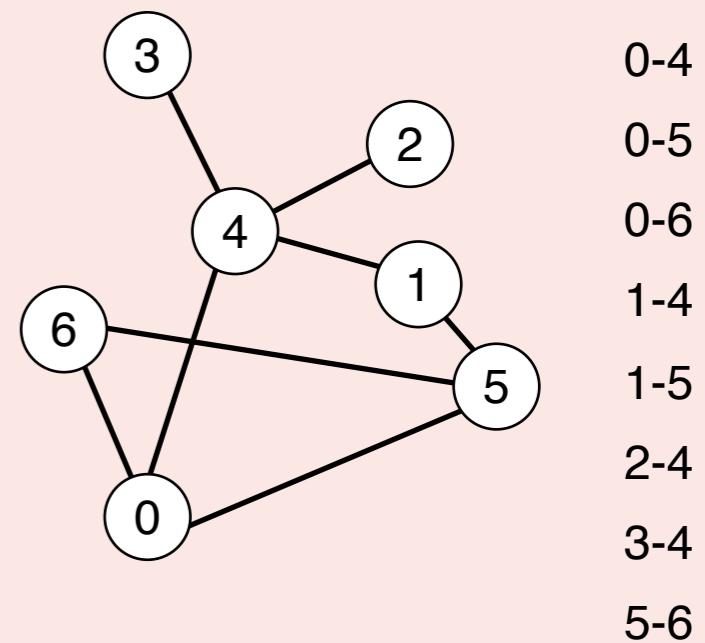
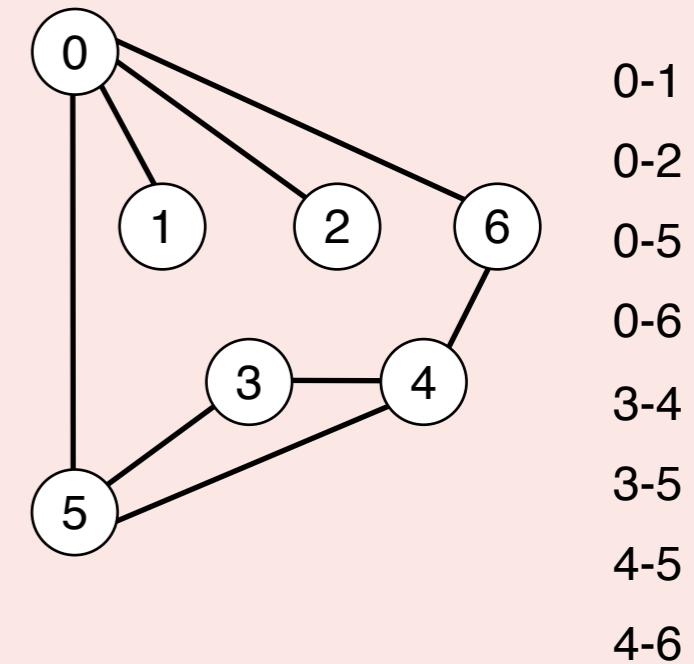


Graph-processing challenge 6

Problem. Are two graphs identical except for vertex names?

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.



$0 \leftrightarrow 4, 1 \leftrightarrow 3, 2 \leftrightarrow 2, 3 \leftrightarrow 6, 4 \leftrightarrow 5, 5 \leftrightarrow 0, 6 \leftrightarrow 1$

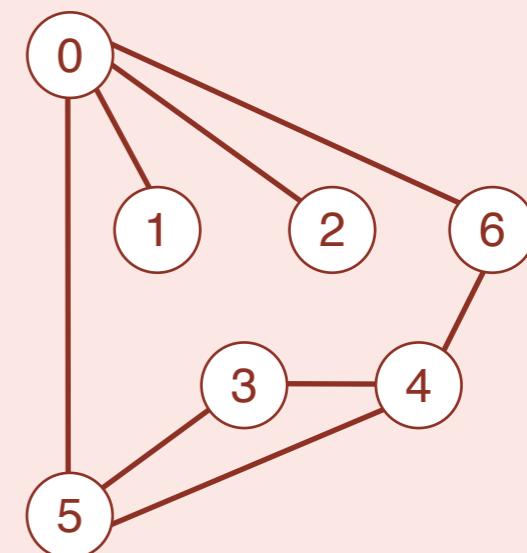
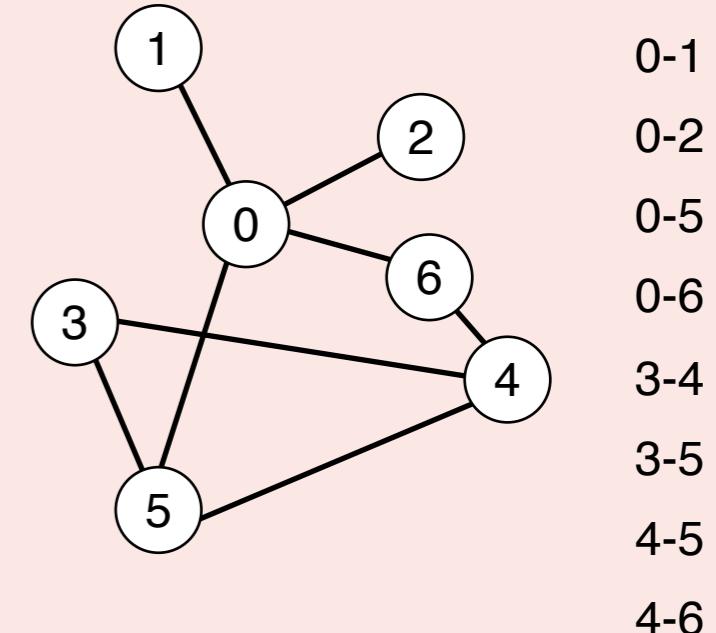
Graph-processing challenge 7

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	✓	✓	$E + V$
Euler cycle		✓	$E + V$
Hamilton cycle			$2^{1.657V}$
bipartiteness (odd cycle)	✓	✓	$E + V$
connected components	✓	✓	$E + V$
biconnected components		✓	$E + V$
planarity		✓	$E + V$
graph isomorphism			$2^{c \ln^3 V}$

INTRODUCTION TO DATA STRUCTURES and ALGORITHMS

Rutgers University
UNDIRECTED GRAPHS

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

