

## 4.1 UNDIRECTED GRAPHS

- ▶ introduction
- ▶ graph API
- ▶ depth-first search
- ▶ breadth-first search
- ▶ connected components
- ▶ challenges



## 4.1 UNDIRECTED GRAPHS

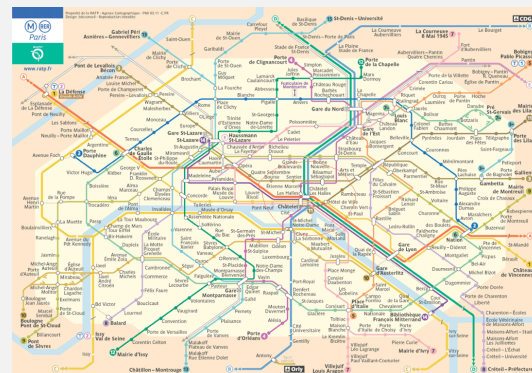
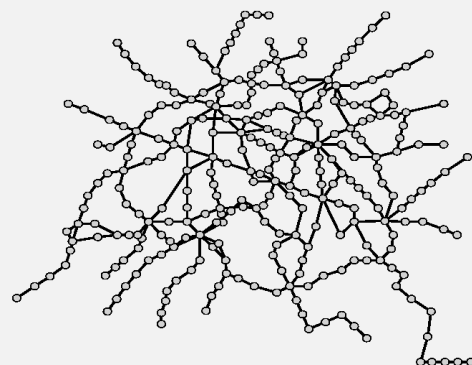
- ▶ introduction
- ▶ graph API
- ▶ depth-first search
- ▶ breadth-first search
- ▶ connected components
- ▶ challenges

## Undirected graphs

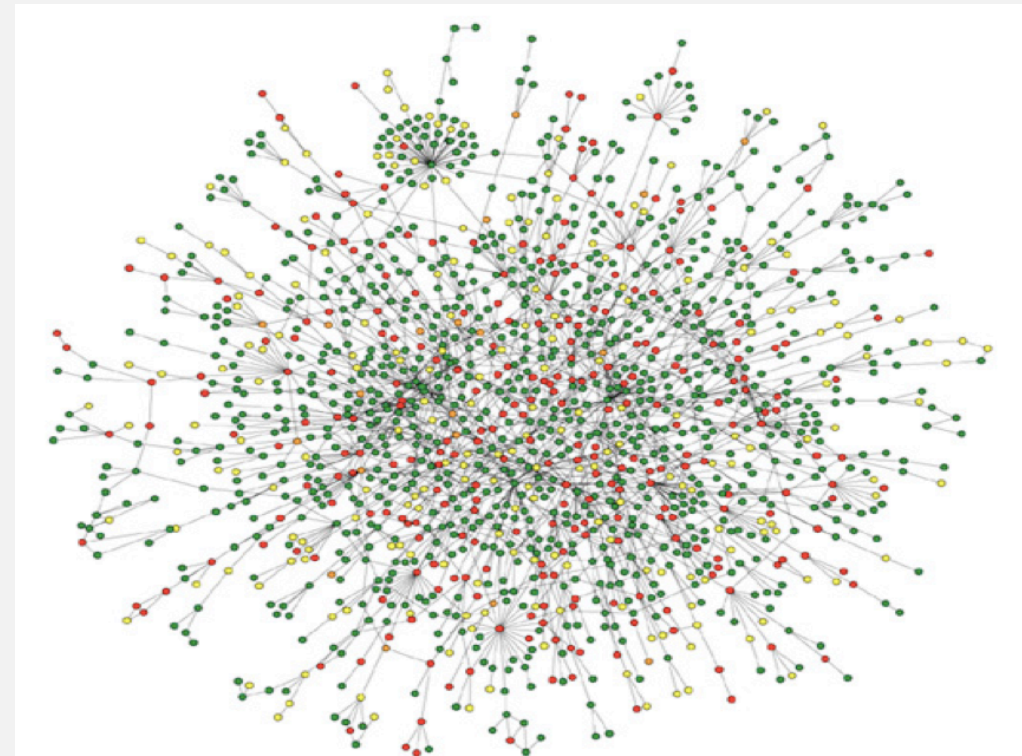
**Graph.** Set of **vertices** connected pairwise by **edges**.

### Why study graph algorithms?

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



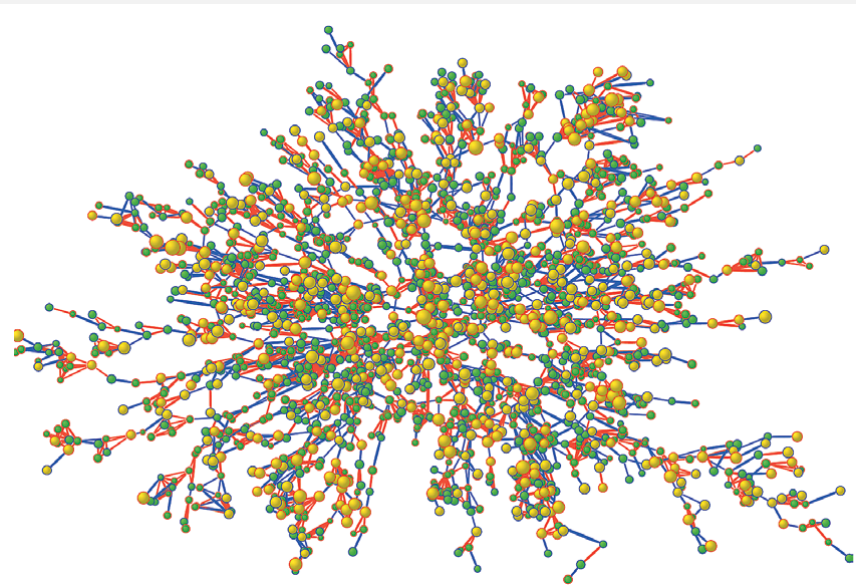
## Protein-protein interaction network



Reference: Jeong et al, Nature Review | Genetics



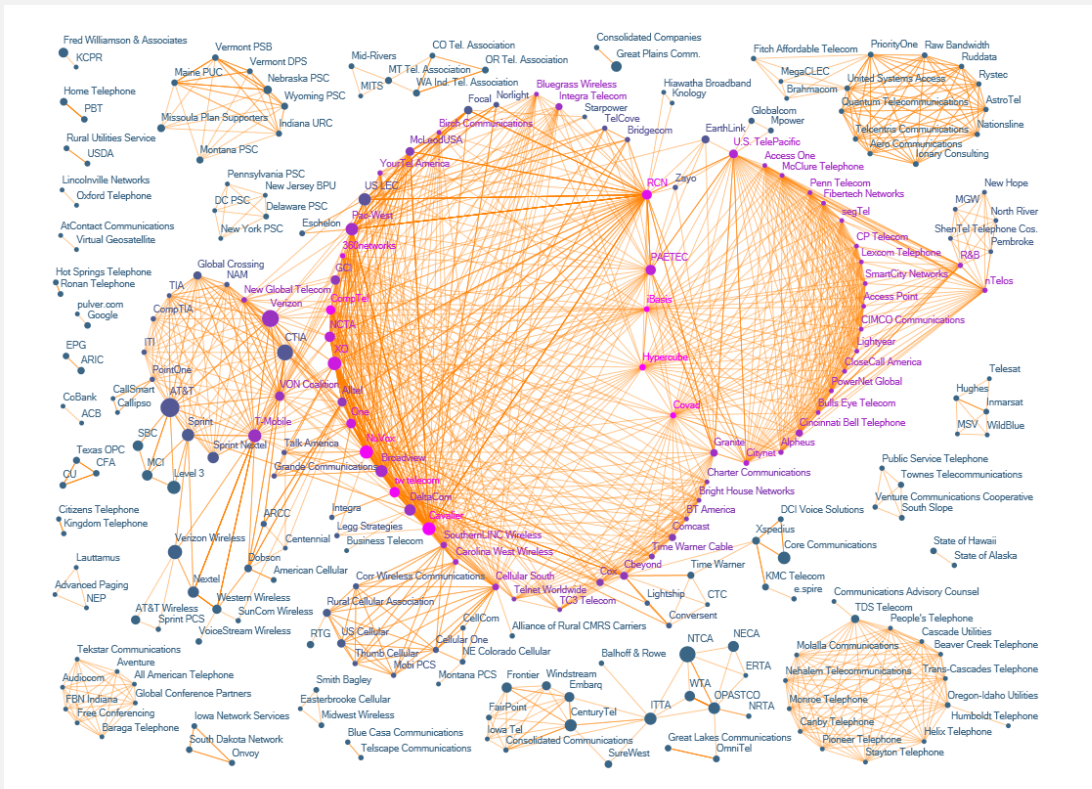
Framingham heart study



**Figure 1.** Largest Connected Subcomponent of the Social Network in the Framingham Heart Study in the Year 2000. Each circle (node) represents one person in the data set. There are 2200 persons in this subcomponent of the social network. Circles with red borders denote women, and circles with blue borders denote men. The size of each circle is proportional to the person's body-mass index. The interior color of the circles indicates the person's obesity status: yellow denotes an obese person (body-mass index,  $\geq 30$ ) and green denotes a nonobese person. The colors of the ties between the nodes indicate the relationship between them: purple denotes a friendship or marital tie and orange denotes a familial tie.

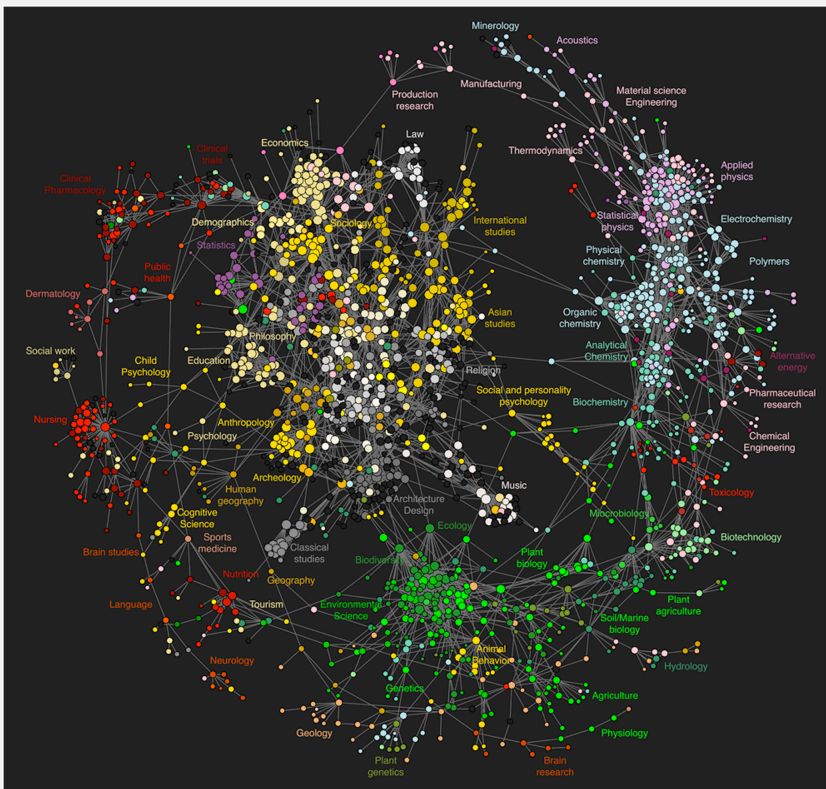
“The Spread of Obesity in a Large Social Network over 32 Years” by Christakis and Fowler in New England Journal of Medicine, 2007

The evolution of FCC lobbying coalitions



“The Evolution of FCC Lobbying Coalitions” by Pierre de Vries in JoSS Visualization Symposium 2010

Map of science clickstreams



<http://www.plosone.org/article/info:doi/10.1371/journal.pone.0004803>

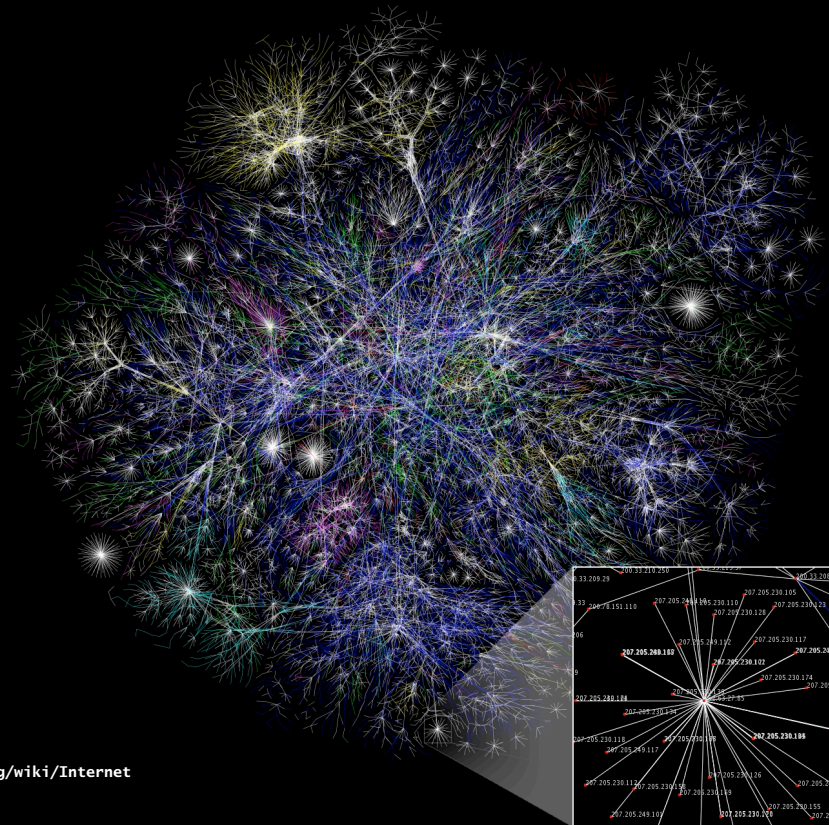
10 million Facebook friends



“Visualizing Friendships” by Paul Butler



## The Internet as mapped by the Opte Project



## 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

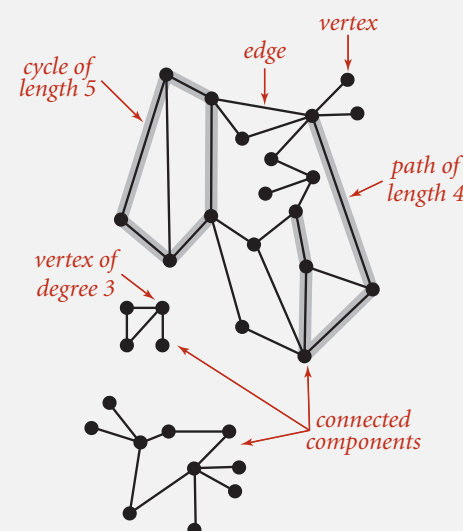
10

## Graph terminology

**Path.** Sequence of vertices connected by edges.

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

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



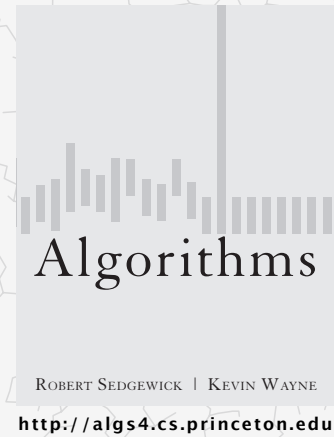
11

## Some graph-processing problems

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

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

12

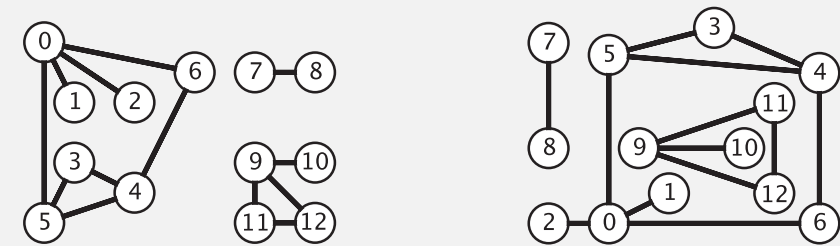


## 4.1 UNDIRECTED GRAPHS

- ▶ introduction
- ▶ graph API
- ▶ depth-first search
- ▶ breadth-first search
- ▶ connected components
- ▶ challenges

## Graph representation

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



two drawings of the same graph

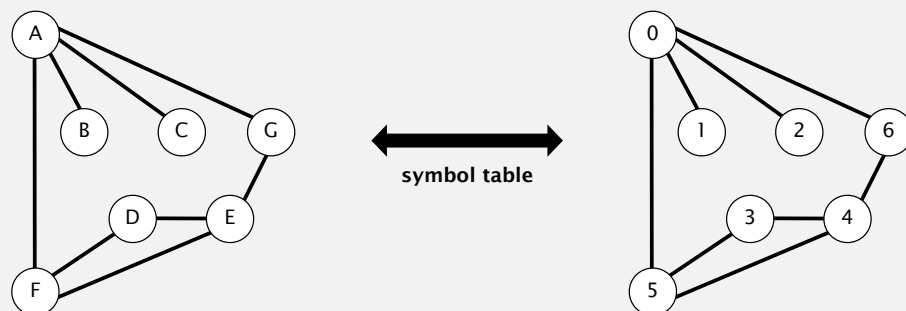
**Caveat.** Intuition can be misleading.

14

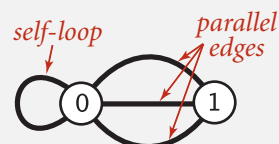
## Graph representation

**Vertex representation.**

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



**Anomalies.**



15

## Graph API

```
public class Graph
{
    Graph(int V)                create an empty graph with V vertices
    Graph(In in)                create a graph from input stream

    void addEdge(int v, int w)  add an edge v-w

    Iterable<Integer> adj(int v) vertices adjacent to v

    int V()                    number of vertices

    int E()                    number of edges
}
```

```
// 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;
}
```

16



## Graph API: sample client

Graph input format.

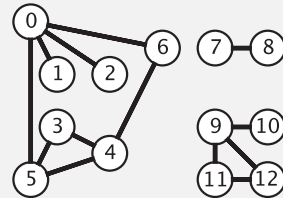
**tinyG.txt**

V → 13  
E → 13

```

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

```



```

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

```

```

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

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

```

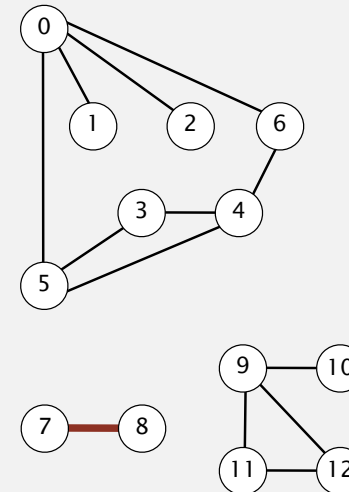
← read graph from  
input stream

← print out each  
edge (twice)

17

## Graph representation: set of edges

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



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

18

## Undirected graphs: quiz 1

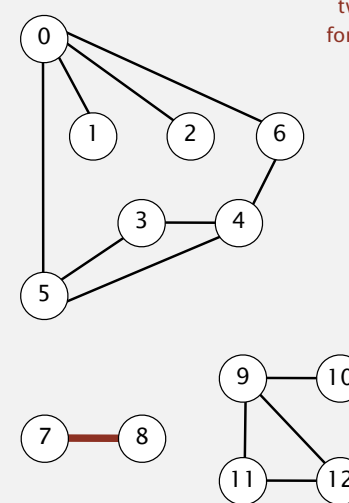
Which is order of growth of running time to iterate over all vertices adjacent to  $v$  using the set-of-edges representation?

- A. 1
- B.  $\text{degree}(v)$
- C.  $V$
- D.  $E$
- E. *I don't know.*

19

## Graph representation: adjacency matrix

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	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

20

## Undirected graphs: quiz 2

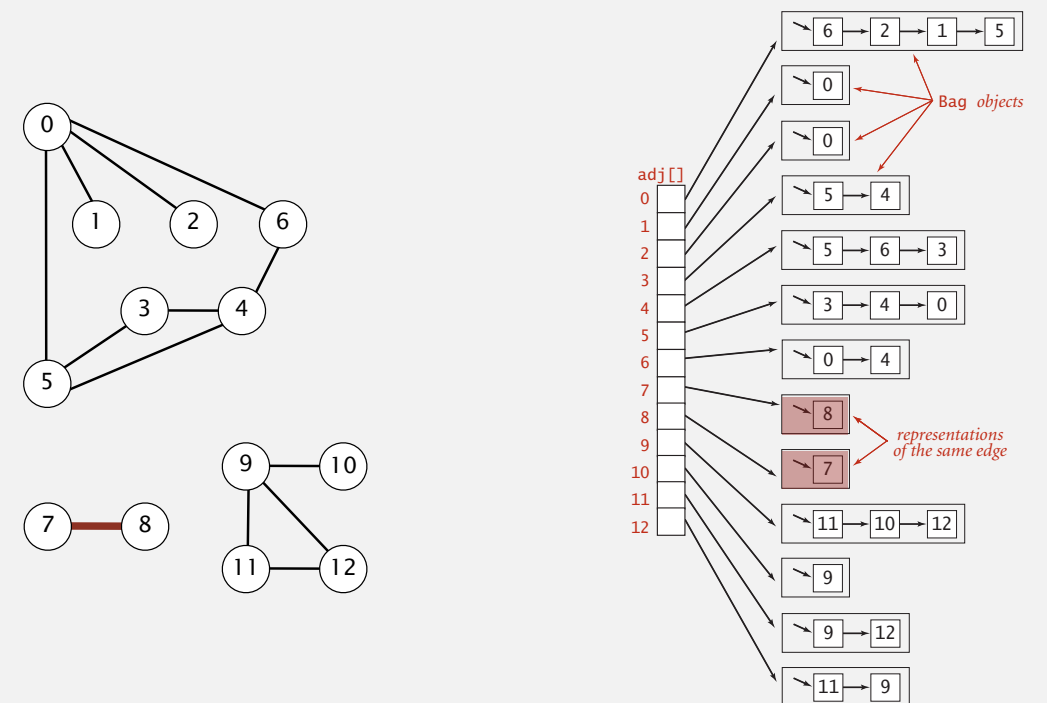
Which is order of growth of running time to iterate over all vertices adjacent to  $v$  using the adjacency-matrix representation?

- A. 1
- B.  $\text{degree}(v)$
- C.  $V$
- D.  $E$
- E. *I don't know.*

21

## Graph representation: adjacency lists

Maintain vertex-indexed array of lists.



22

## Undirected graphs: quiz 3

Which is order of growth of running time to iterate over all vertices adjacent to  $v$  using the adjacency-lists representation?

- A. 1
- B.  $\text{degree}(v)$
- C.  $V$
- D.  $E$
- E. *I don't know.*

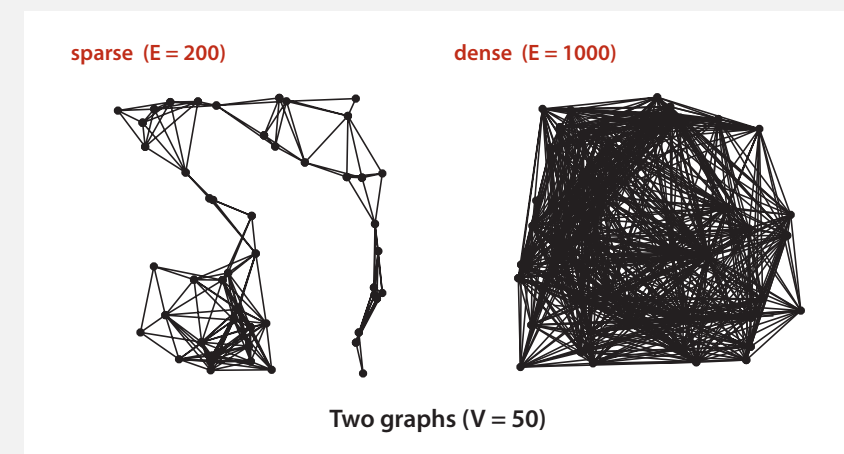
23

## Graph representations

**In practice.** Use adjacency-lists representation.

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

huge number of vertices,  
small average vertex degree



24



## Graph representations

**In practice.** Use adjacency-lists representation.

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

huge number of vertices,  
small average vertex degree

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

\* disallows parallel edges

25

## 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  $v$

26

## 4.1 UNDIRECTED GRAPHS

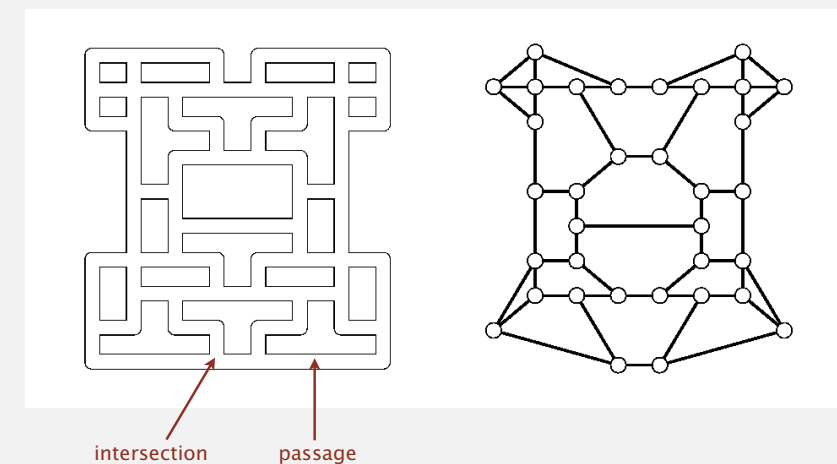
- introduction
- graph API
- depth-first search
- breadth-first search
- connected components
- challenges



## Maze exploration

**Maze graph.**

- Vertex = intersection.
- Edge = passage.



**Goal.** Explore every intersection in the maze.

28

## Maze exploration: National Building Museum



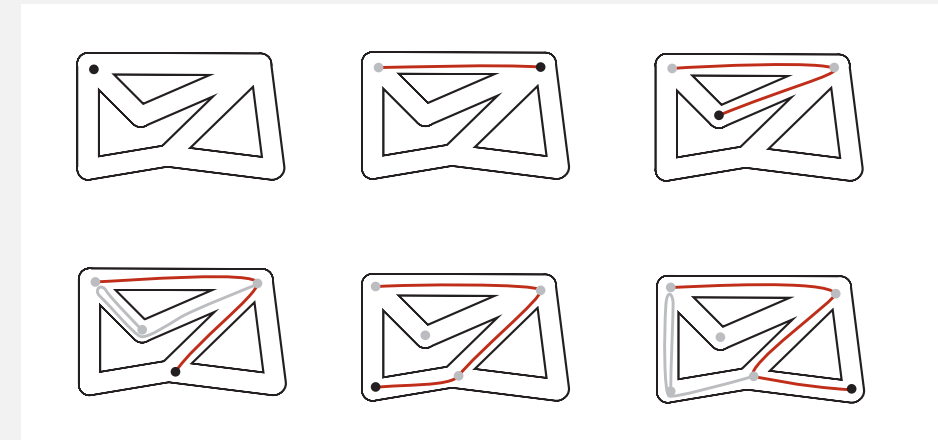
<http://www.smithsonianmag.com/travel/winding-history-maze-180951998/?no-ist>

29

## Trémaux maze exploration

### Algorithm.

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



30

## 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>

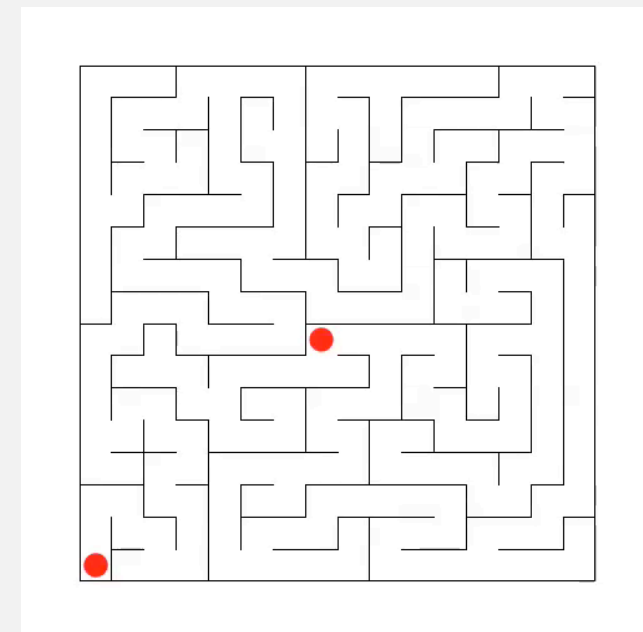


Claude Shannon (with electromechanical mouse)

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

31

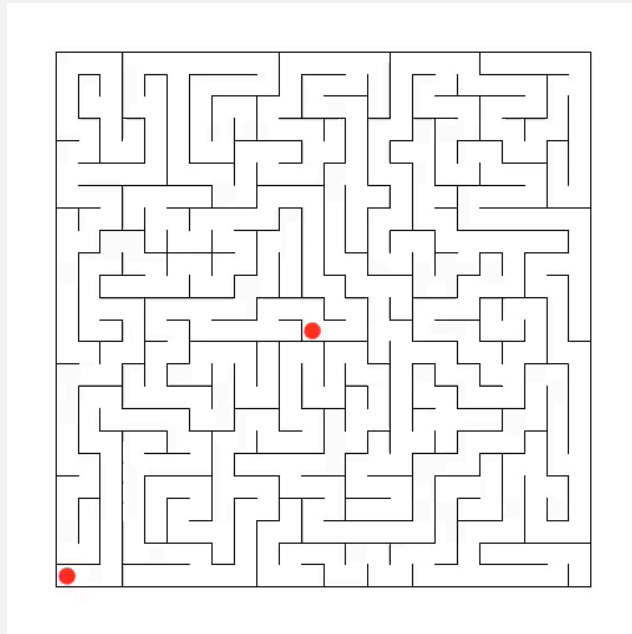
## Maze exploration: easy



32

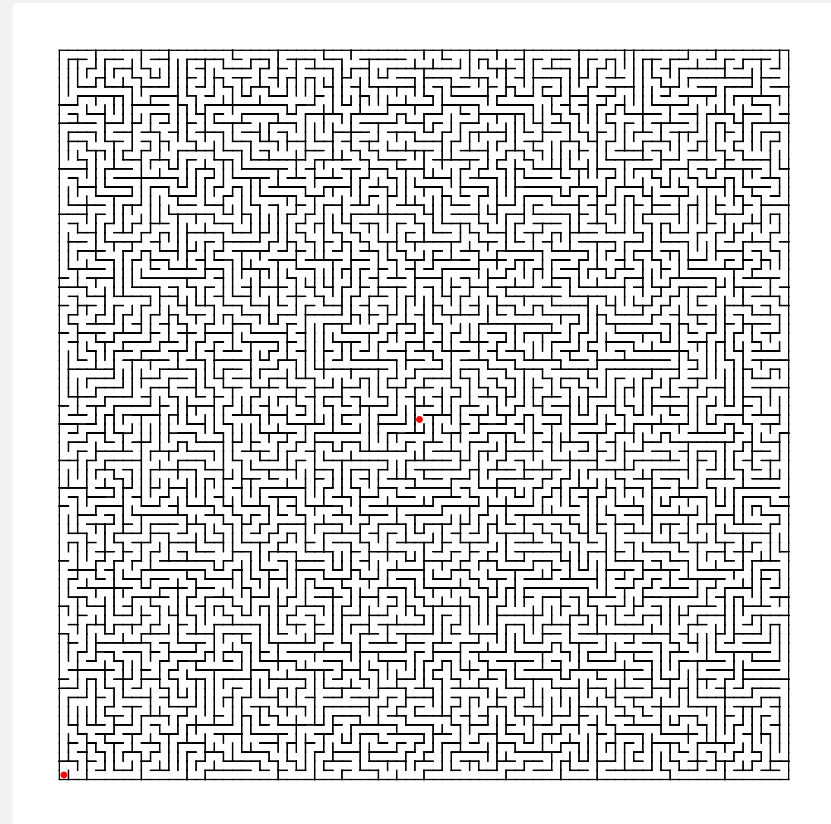


## Maze exploration: medium



33


## Maze exploration: challenge for the bored



34

## Depth-first search

**Goal.** Systematically traverse a graph.

**Idea.** Mimic maze exploration.  function-call stack acts as ball of string

**DFS** (to visit a vertex  $v$ )

Mark vertex  $v$ .

Recursively visit all unmarked  
vertices  $w$  adjacent to  $v$ .

**Typical applications.**

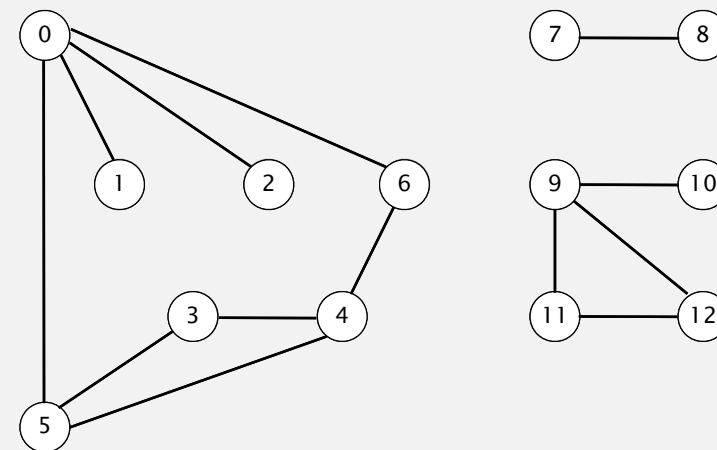
- Find all vertices connected to a given source vertex.
- Find a path between two vertices.

**Design challenge.** How to implement?

## Depth-first search demo

To visit a vertex  $v$ :

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



graph G

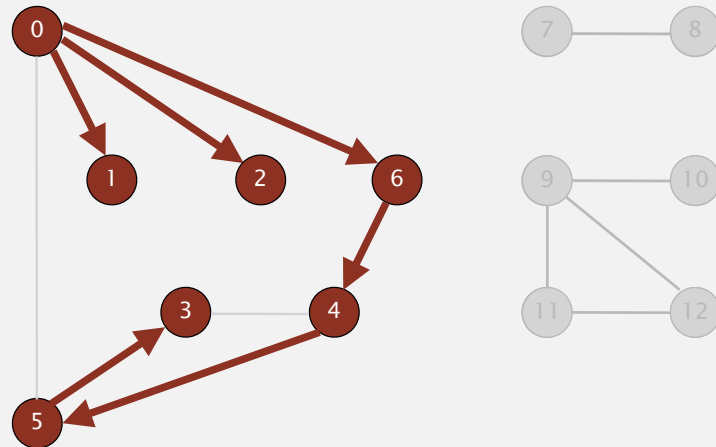
**tinyG.txt**  
 $V \rightarrow$  13  
13  $\leftarrow E$   
0 5  
4 3  
0 1  
9 12  
6 4  
5 4  
0 2  
11 12  
9 10  
0 6  
7 8  
9 11  
5 3

36

## Depth-first search demo

To visit a vertex  $v$ :

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



vertices reachable from 0

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	-

37

## 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)           find paths in G from source s
```

```
    boolean hasPathTo(int v)       is there a path from s to v?
```

```
    Iterable<Integer> pathTo(int v) path from s to v; null if no such path
```

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

← print all vertices  
connected to s

38

## 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  $v$ - $w$  taken to discover vertex  $w$
- Function-call stack for recursion.

## 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);
            }
    }
}
```

← `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

40

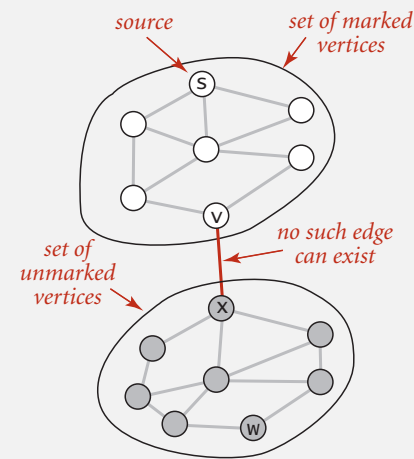


## Depth-first search: properties

**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.

41

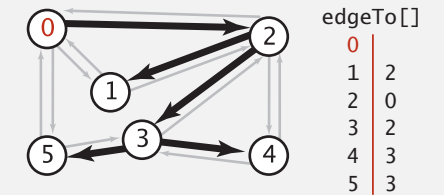
## Depth-first search: properties

**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;
}
```



42

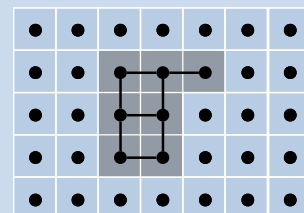
## FLOOD FILL

**Problem.** Implement flood fill (Photoshop magic wand).



**Solution.** Build a **grid graph** (implicitly).

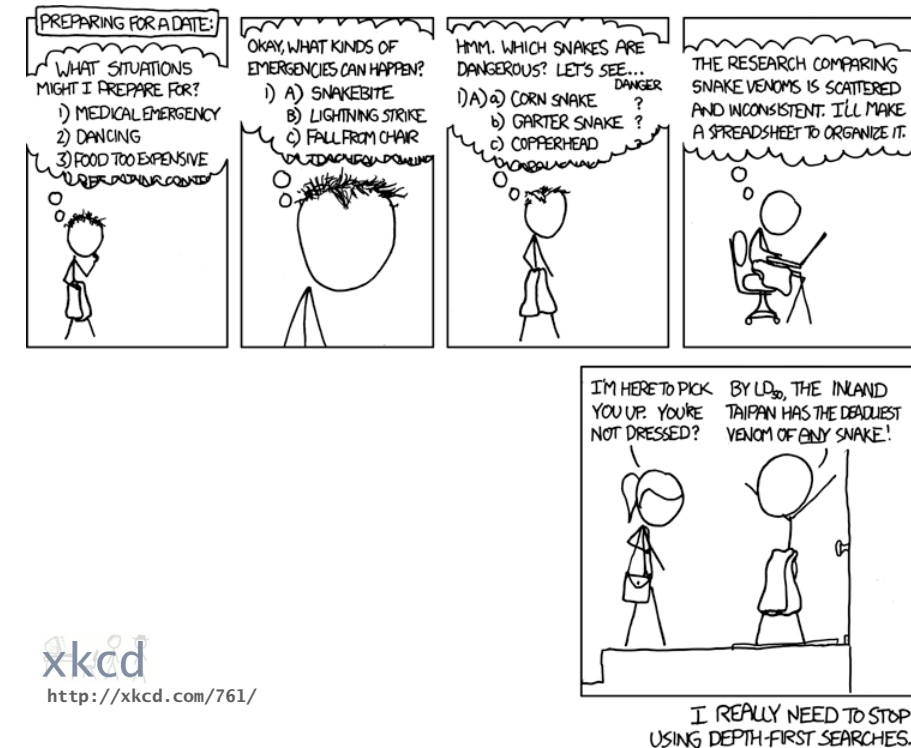
- Vertex: pixel.
- Edge: between two adjacent gray pixels.
- Blob: all pixels connected to given pixel.



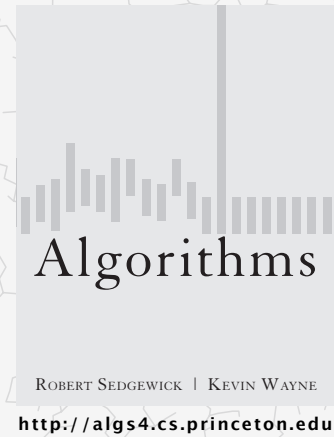
**Extra concern.** Function-call stack depth.

43

## Depth-first search application: preparing for a date



44



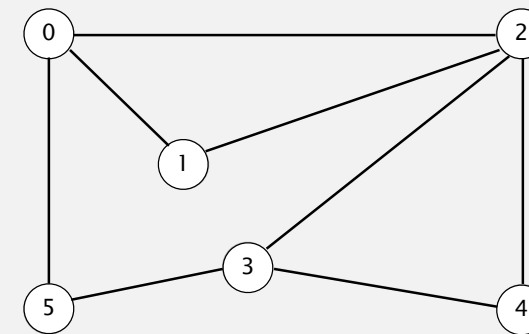
## 4.1 UNDIRECTED GRAPHS

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

## 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.



tinyCG.txt  
V → 6  
8 ← E  
0 5  
2 4  
2 3  
1 2  
0 1  
3 4  
3 5  
0 2

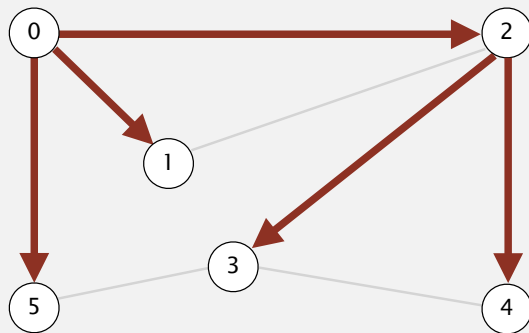
graph G

46

## 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

done

47

## Breadth-first search

Repeat until queue is empty:

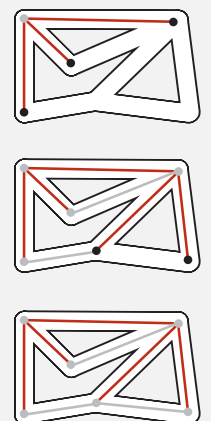
- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent to  $v$  and mark them.

### 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 marked neighbors to the queue, and mark them.



48



## 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;
                }
            }
        }
    }
}
```

initialize FIFO queue of vertices to explore

found new vertex w via edge v-w

49

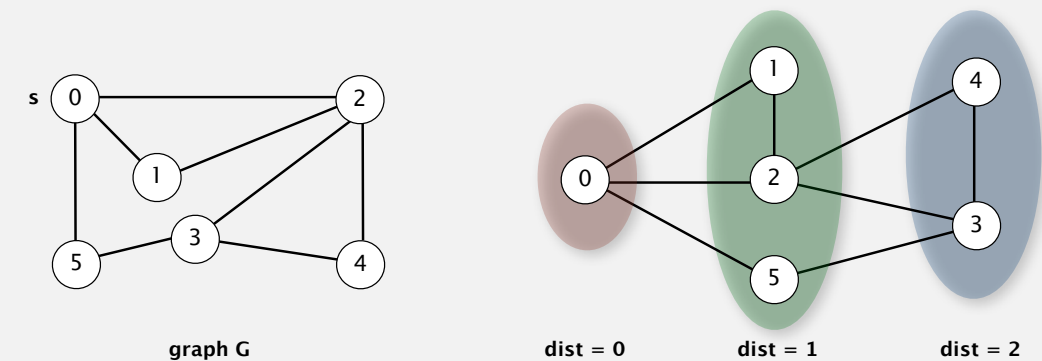
## Breadth-first search properties

Q. In which order does BFS examine vertices?

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

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

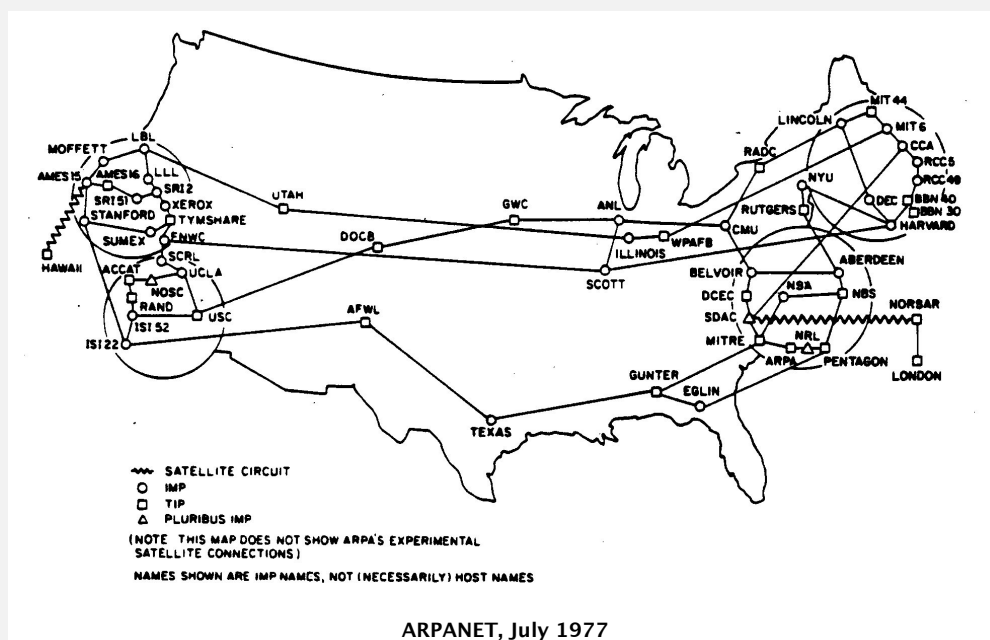
**Proposition.** In any connected graph  $G$ , BFS computes shortest paths from  $s$  to all other vertices in time proportional to  $E + V$ .



50

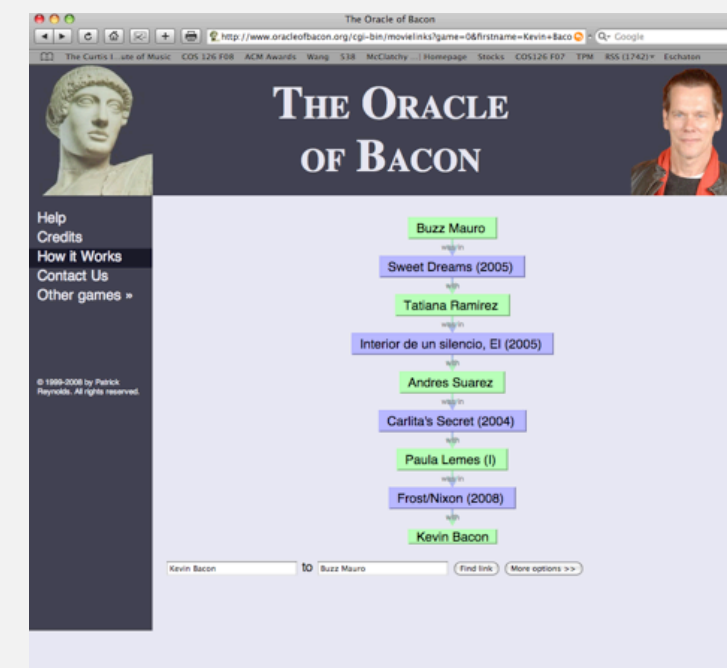
## Breadth-first search application: routing

Fewest number of hops in a communication network.

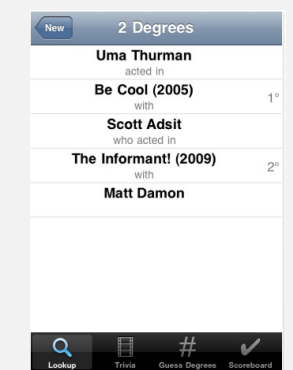


51

## Breadth-first search application: Kevin Bacon numbers



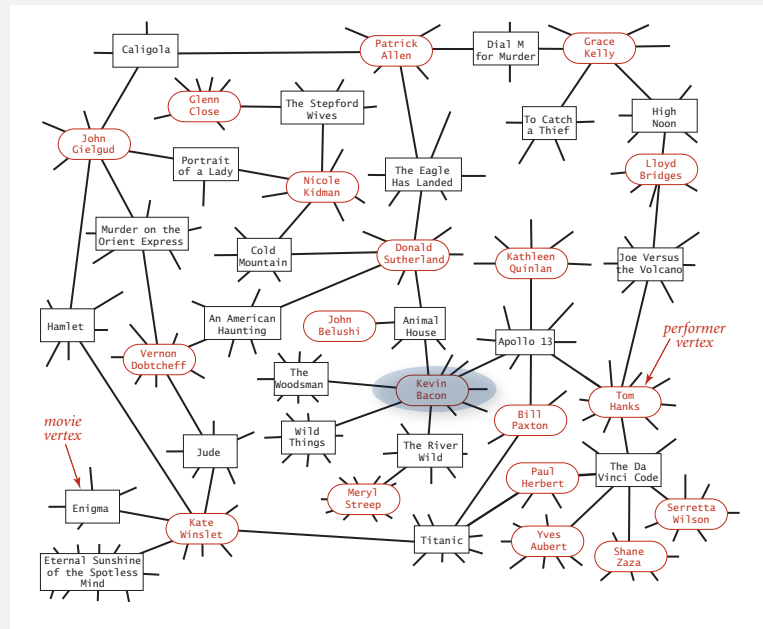
Endless Games board game



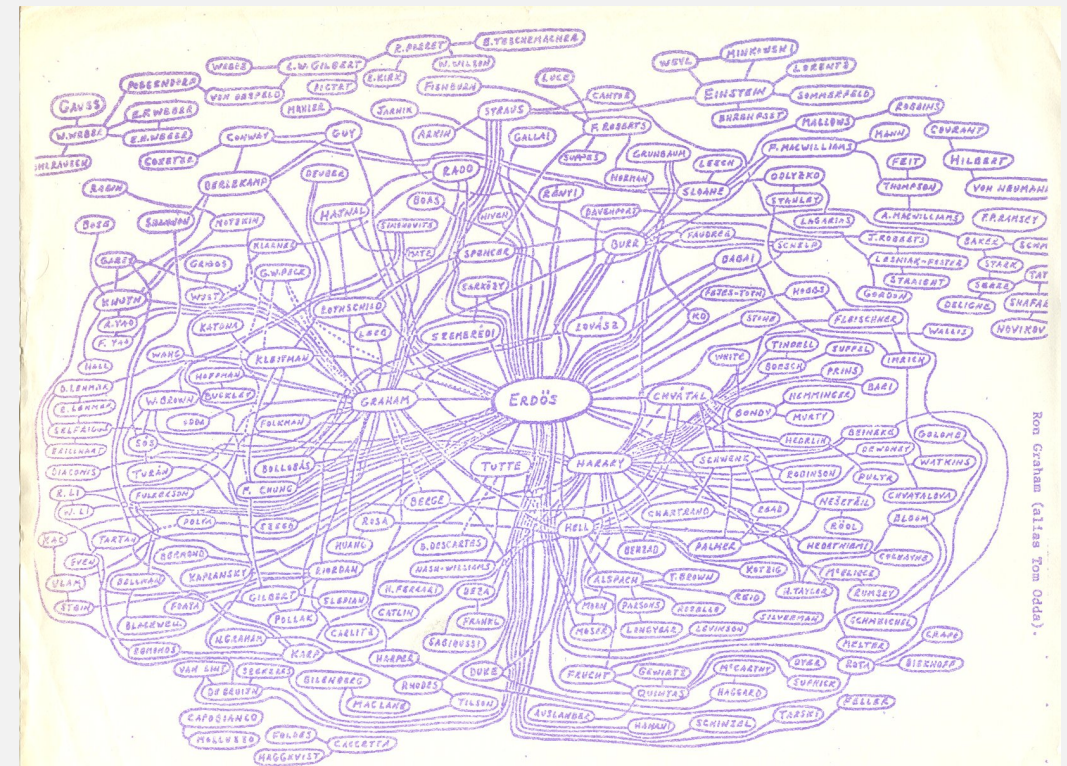
52

## 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



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

## 4.1 UNDIRECTED GRAPHS

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



## Connectivity queries

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

**Goal.** Preprocess graph to answer queries of the form *is  $v$  connected to  $w$ ?* in **constant** time.

```
public class CC
```

$$CC(\text{Graph } G)$$

*find connected components in  $G$*

```
boolean connected(int v, int w)
```

*are  $v$  and  $w$  connected?*

```
int count()
```

*number of connected components*

```
int id(int v)
```

*component identifier for v*  
(between 0 and count() - 1)

Union-Find? Not quite.

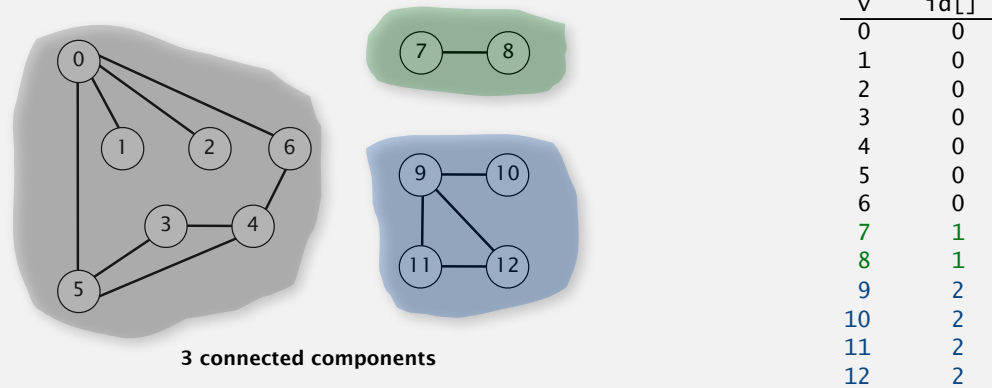
Depth-first search. Yes. [next few slides]

# Connected components

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

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

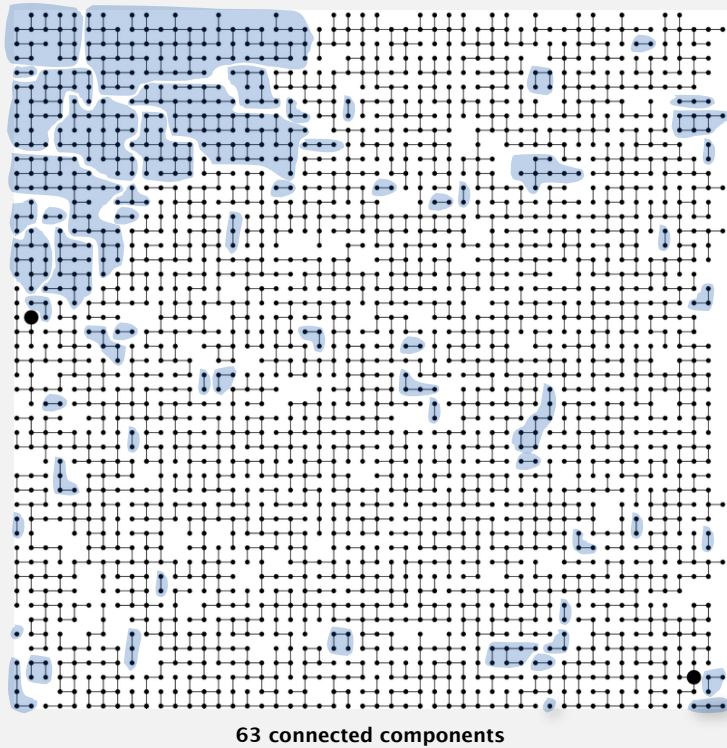
**Def.** A **connected component** is a maximal set of connected vertices.



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

# Connected components

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



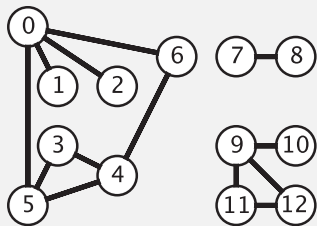
# Connected components

**Goal.** Partition vertices into connected components.

**Connected components**

Initialize all vertices  $v$  as unmarked.

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

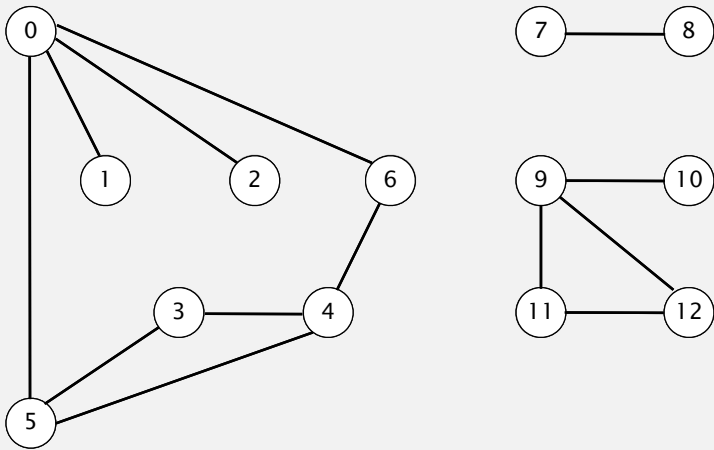


tinyG.txt  
V → 13  
13 ← E  
0 5  
4 3  
0 1  
9 12  
6 4  
5 4  
0 2  
11 12  
9 10  
0 6  
7 8  
9 11  
5 3

# Connected components demo

To visit a vertex  $v$ :

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



graph G

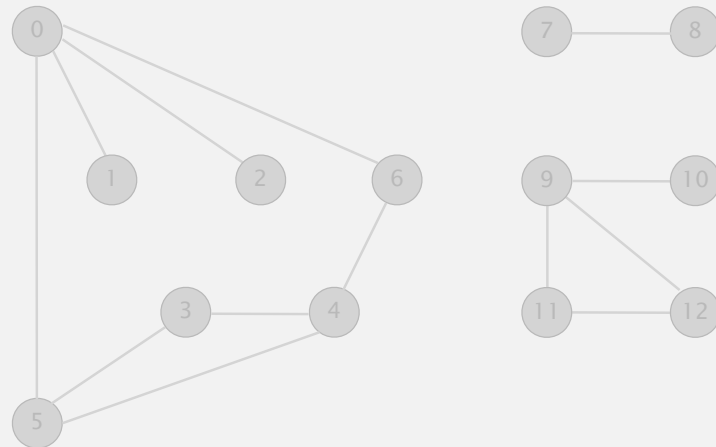
v	marked[]	id[]
0	F	—
1	F	—
2	F	—
3	F	—
4	F	—
5	F	—
6	F	—
7	F	—
8	F	—
9	F	—
10	F	—
11	F	—
12	F	—



## Connected components demo

To visit a vertex  $v$ :

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



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

done

61

## Finding connected components with DFS

```

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

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

    public int count()
    public int id(int v)
    public boolean connected(int v, int w)
    private void dfs(Graph G, int v)
}
    
```

id[v] = id of component containing v  
number of components

run DFS from one vertex in  
each component

see next slide

62

## Finding connected components with DFS (continued)

```

public int count()
{ return count; }

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

public boolean connected(int v, int w)
{ return id[v] == id[w]; }

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

number of components

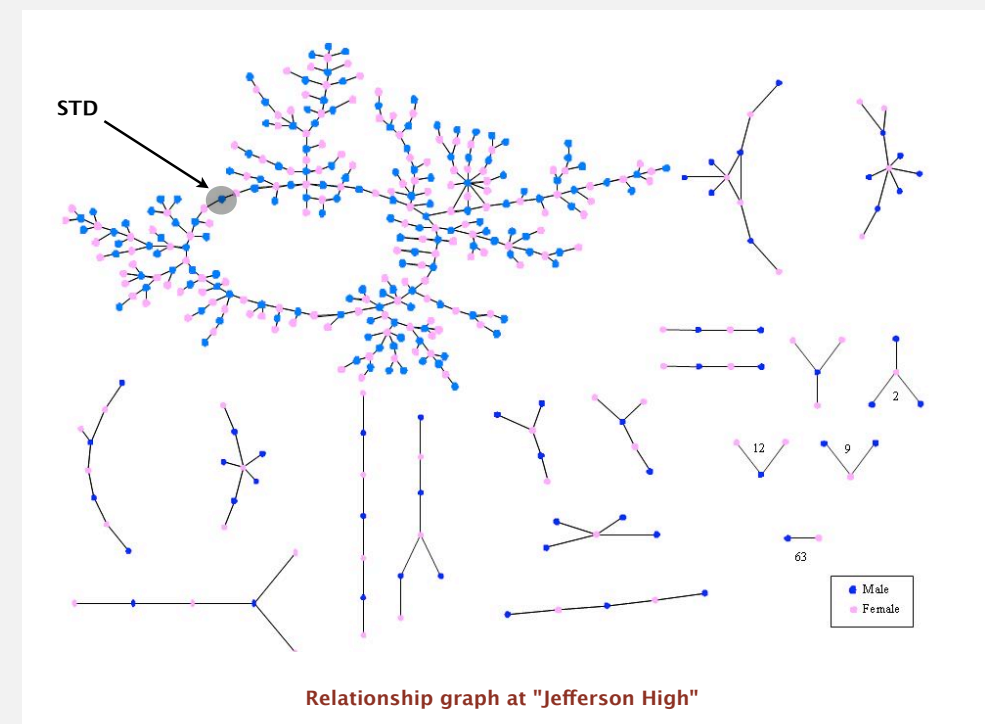
id of component containing v

v and w connected iff same id

all vertices discovered in  
same call of dfs have same id

63

## Connected components application: study spread of STDs



Relationship graph at "Jefferson High"

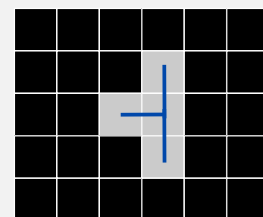
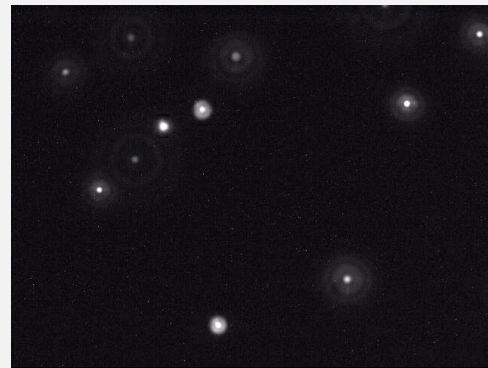
Peter Bearman, James Moody, and Katherine Stovel. Chains of affection: The structure of adolescent romantic and sexual networks. *American Journal of Sociology*, 110(1): 44-99, 2004.

64

## Connected components application: particle detection

**Particle detection.** Given grayscale image of particles, identify "blobs."

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



black = 0  
white = 255

**Particle tracking.** Track moving particles over time.

65

## 4.1 UNDIRECTED GRAPHS

- ▶ introduction
- ▶ graph API
- ▶ depth-first search
- ▶ breadth-first search
- ▶ connected components
- ▶ challenges

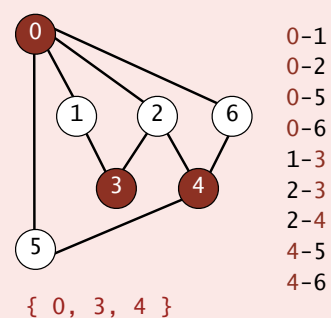
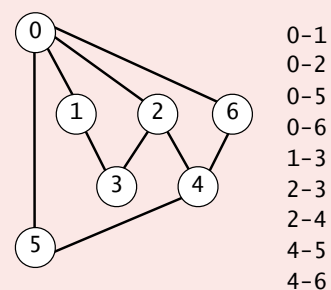
## Graph-processing challenge 1

**Problem.** Is a graph bipartite?

**How difficult?**

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

simple DFS-based solution  
(see textbook)



67

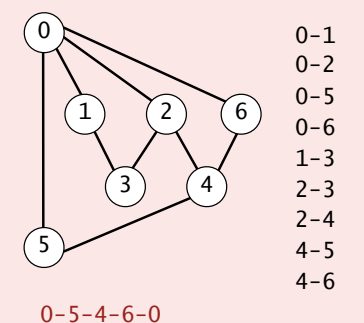
## Graph-processing challenge 2

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

**How difficult?**

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

simple DFS-based solution  
(see textbook)



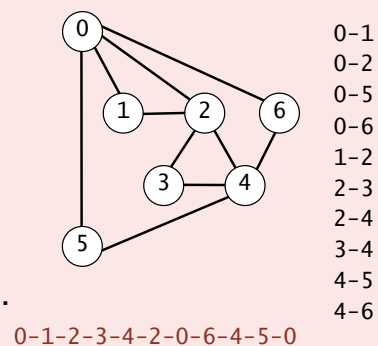
68

### Graph-processing challenge 3

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

**How difficult?**

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



yes if and only if graph is connected  
and every vertex has even degree  
(Leonhard Euler 1786)

moreover, if graph is Eulerian,  
can find a Euler cycle via DFS

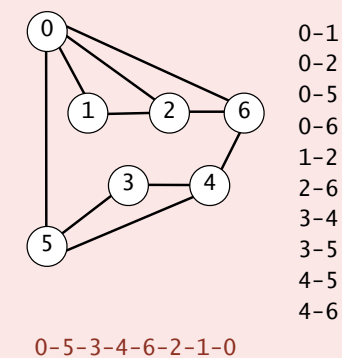
69

### Graph-processing challenge 4

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

**How difficult?**

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



Hamilton cycle  
(classical NP-complete problem)

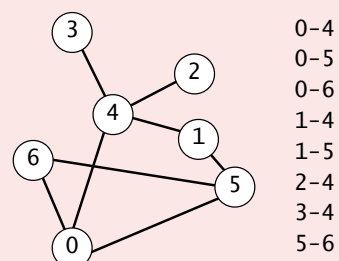
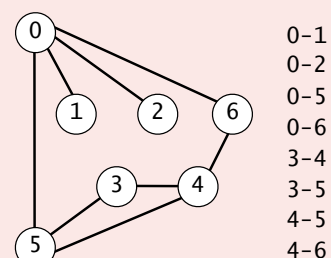
70

### Graph-processing challenge 5

**Problem.** Are two graphs identical except for vertex names?

**How difficult?**

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



graph isomorphism is  
longstanding open problem

0↔4, 1↔3, 2↔2, 3↔6, 4↔5, 5↔0, 6↔1

71

### Graph-processing challenge 6

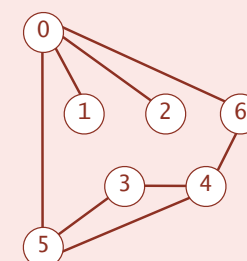
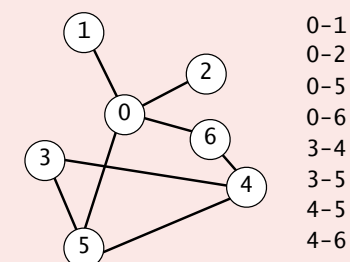
**Problem.** Can you draw a graph in the plane with no crossing edges?

try it yourself at <http://planarity.net>

**How difficult?**

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

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



72



## Graph traversal summary

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

graph problem	BFS	DFS	time
<b>s-t path</b>	✓	✓	$E + V$
<b>shortest s-t path</b>	✓		$E + V$
<b>cycle</b>	✓	✓	$E + V$
<b>Euler cycle</b>		✓	$E + V$
<b>Hamilton cycle</b>			$2^{1.657 V}$
<b>bipartiteness (odd cycle)</b>	✓	✓	$E + V$
<b>connected components</b>	✓	✓	$E + V$
<b>biconnected components</b>		✓	$E + V$
<b>planarity</b>		✓	$E + V$
<b>graph isomorphism</b>			$2^{c\sqrt{V \log V}}$