
http://algs4.cs.princeton.edu

## 4.1 UNDIRECTED GRAPHS

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


http://algs4.cs.princeton.edu

## 4.1 UNDIRECTED GRAPHS

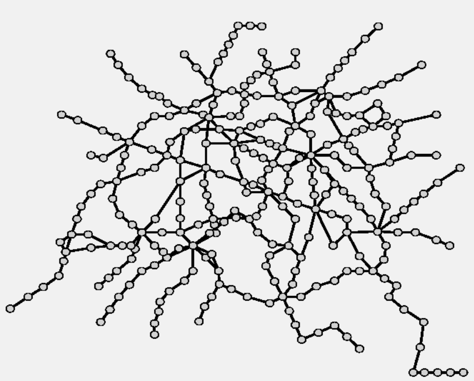
- *introduction*
- *graph API*
- *depth-first search*
- *breadth-first search*
- *challenges*

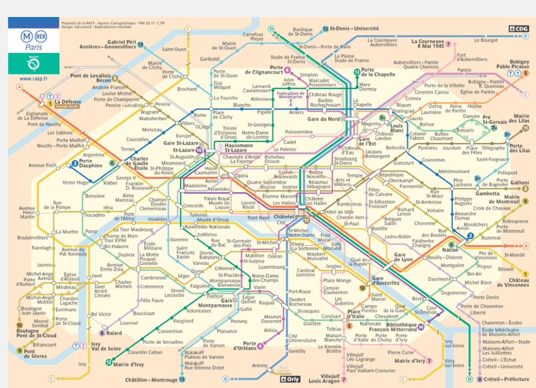
## Undirected graphs

Graph. Set of vertices connected pairwise by edges.

Why study graph algorithms?

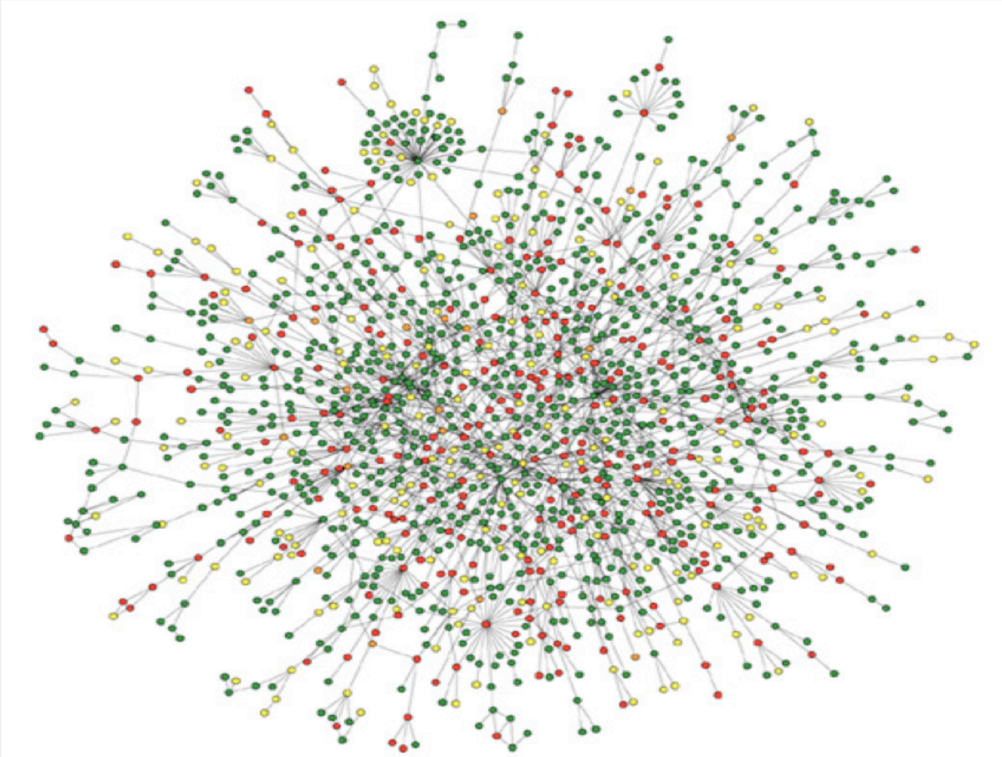
- Thousands of practical applications.
- Hundreds of graph algorithms known.
- Interesting and broadly useful abstraction.
- Challenging branch of computer science and discrete math.







## Protein-protein interaction network



Reference: Jeong et al, Nature Review | Genetics

## Framingham heart study

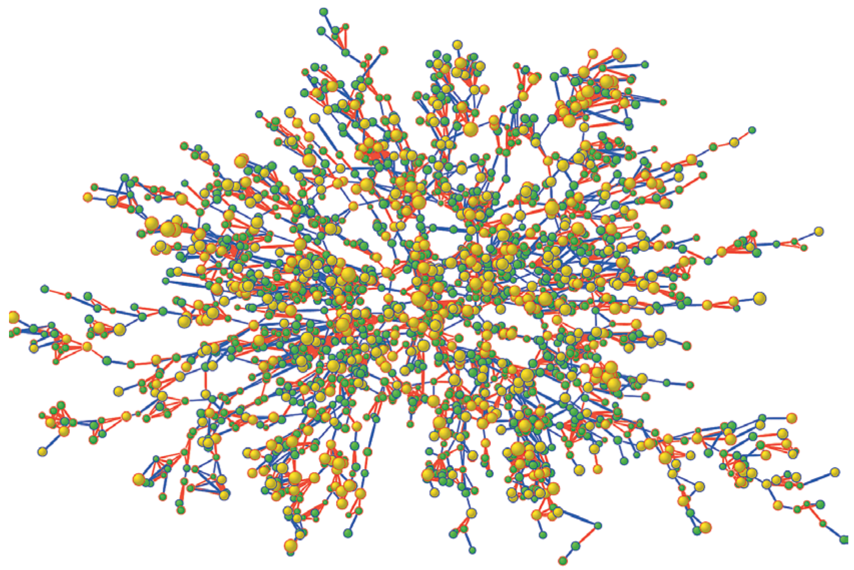
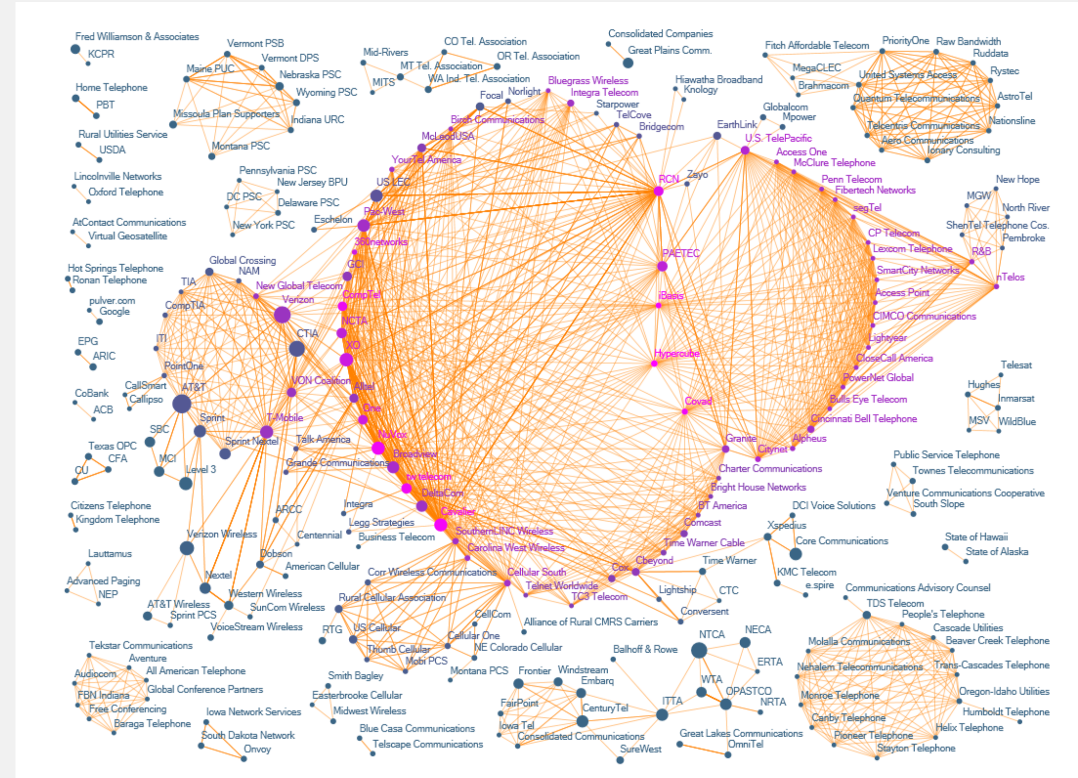


Figure 1. **Largest Connected Subcomponent of the Social Network in the Framingham Heart Study in the Year 2000.**
Each circle (node) represents one person in the data set. There are 2200 persons in this subcomponent of the social network. Circles with red borders denote women, and circles with blue 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, ≥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**



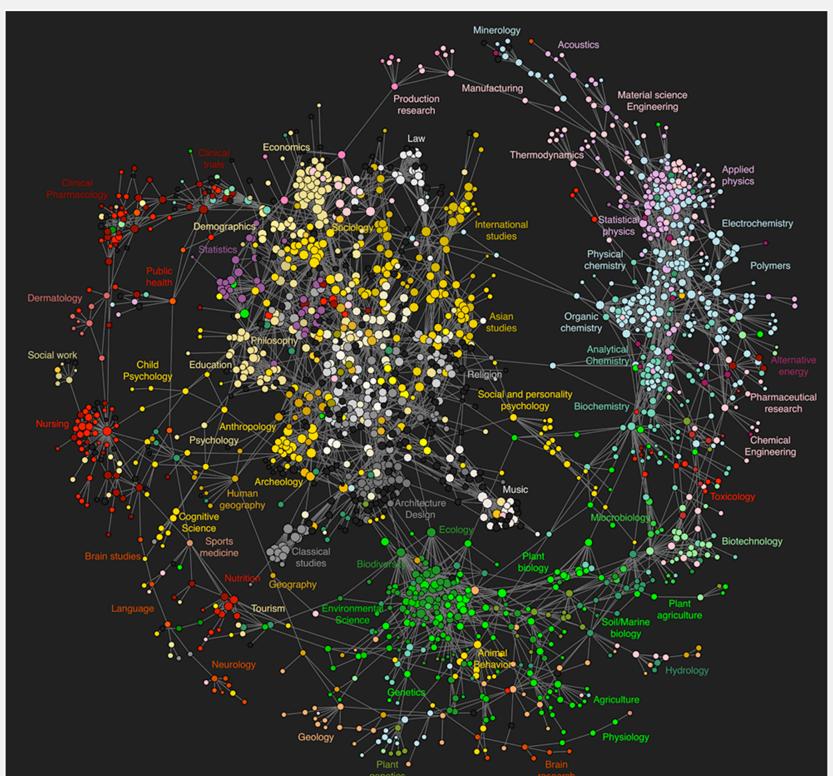
## The evolution of FCC lobbying coalitions



**"The Evolution of FCC Lobbying Coalitions" by Pierre de Vries in JoSS Visualization Symposium 2010**



## Map of science clickstreams



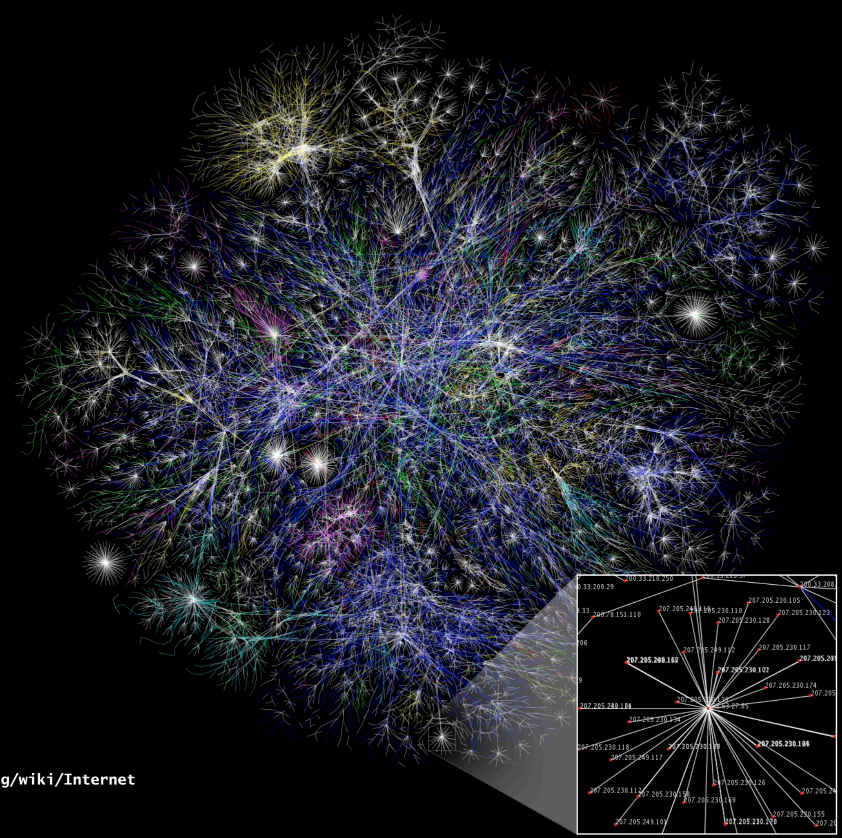
**http://www.plosone.org/article/info:doi/10.1371/journal.pone.0004803**



## 10 million Facebook friends


**"Visualizing Friendships" by Paul Butler**

## The Internet as mapped by the Opte Project



http://en.wikipedia.org/wiki/Internet

## Graph applications

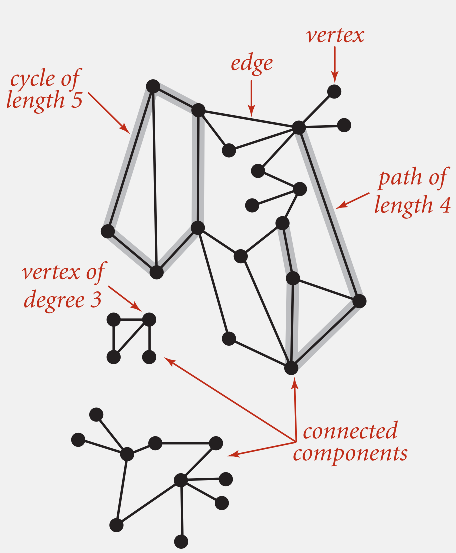
| graph | vertex | edge |
|---|---|---|
| **communication** | telephone, computer | fiber optic cable |
| **circuit** | gate, register, processor | wire |
| **mechanical** | joint | rod, beam, spring |
| **financial** | stock, currency | transactions |
| **transportation** | intersection | street |
| **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

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.






## 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?

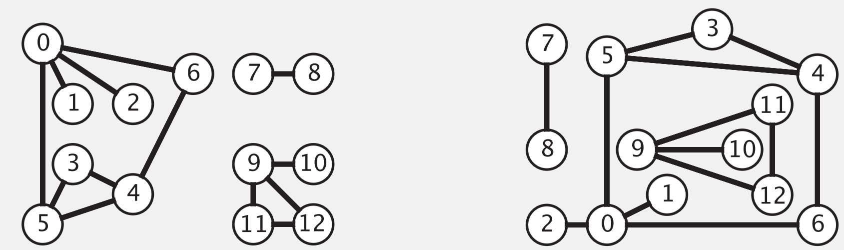

http://algs4.cs.princeton.edu

# 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.




two drawings of the same graph

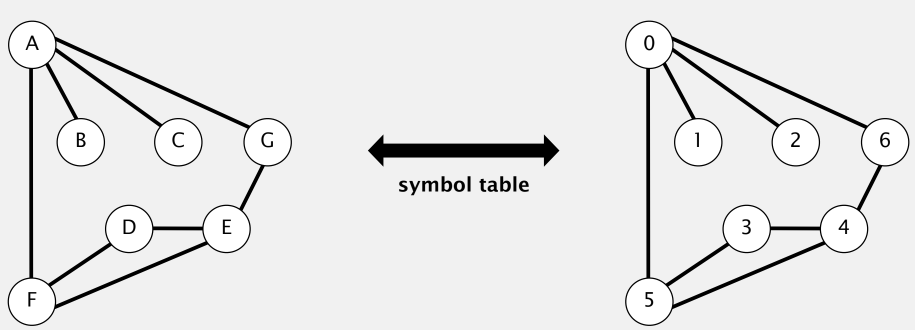
Caveat. Intuition can be misleading.



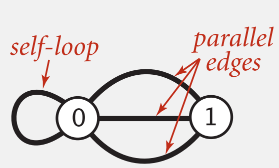
## Graph representation

Vertex representation.

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




Anomalies.





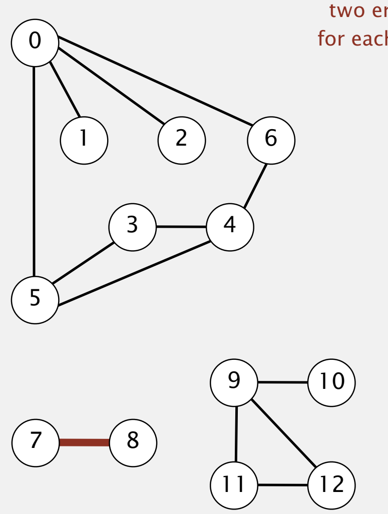

## Graph API

| | | |
|---|---|---|
| | 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;
}
```

## Graph representation: adjacency matrix

Maintain a two-dimensional $V$-by-$V$ boolean array;
for each edge $v–w$ in graph: `adj[v][w] = adj[w][v] = 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 |



## 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?

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

**A.** $V$

**B.** $E + V$

**C.** $V^2$

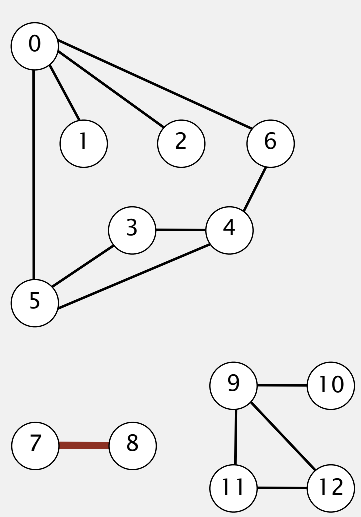
**D.** $V E$

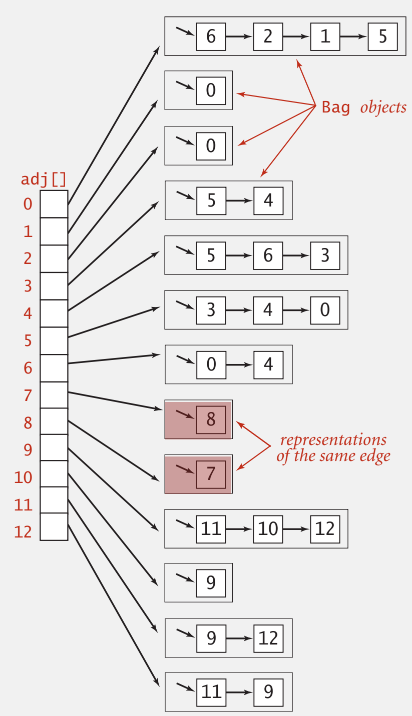
**E.** *I don't know.*



## Graph representation: adjacency lists

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?

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

**A.** $V$

**B.** $E + V$

**C.** $V^2$

**D.** $V E$

**E.** *I don't know.*

## Graph representations

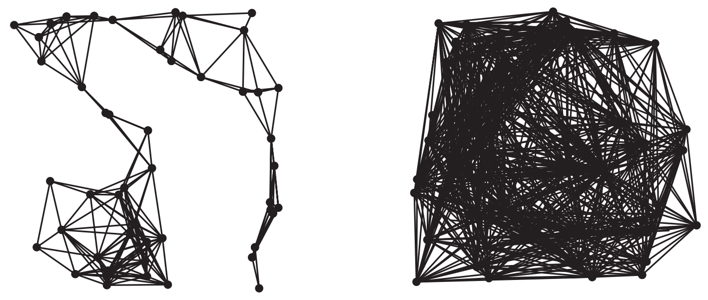
In practice. Use adjacency-lists representation.

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

huge number of vertices,
small average vertex degree

sparse (E = 200)

dense (E = 1000)



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.

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** | $E + V$ | $1$ | $degree(v)$ | $degree(v)$ |

† 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




http://algs4.cs.princeton.edu

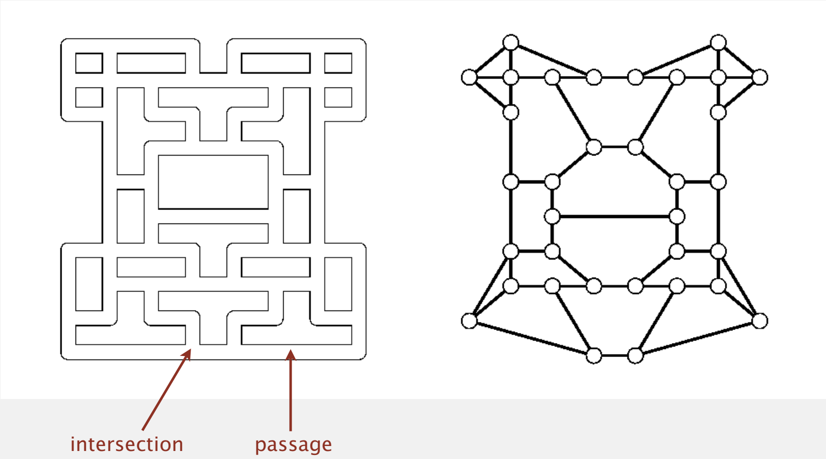
## 4.1 Undirected Graphs

- 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

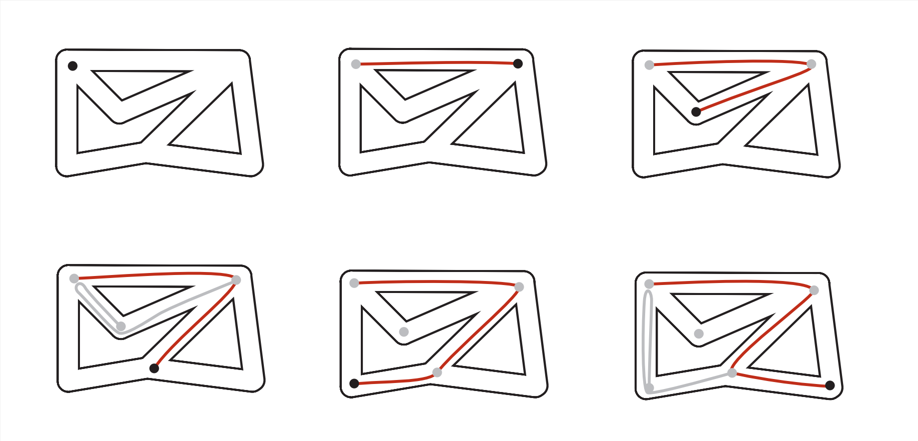
http://www.smithsonianmag.com/travel/winding-history-maze-180951998/?no-ist



## 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

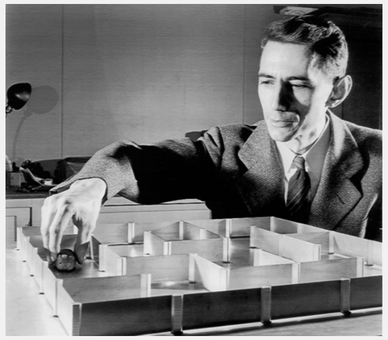
Algorithm.

- 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)

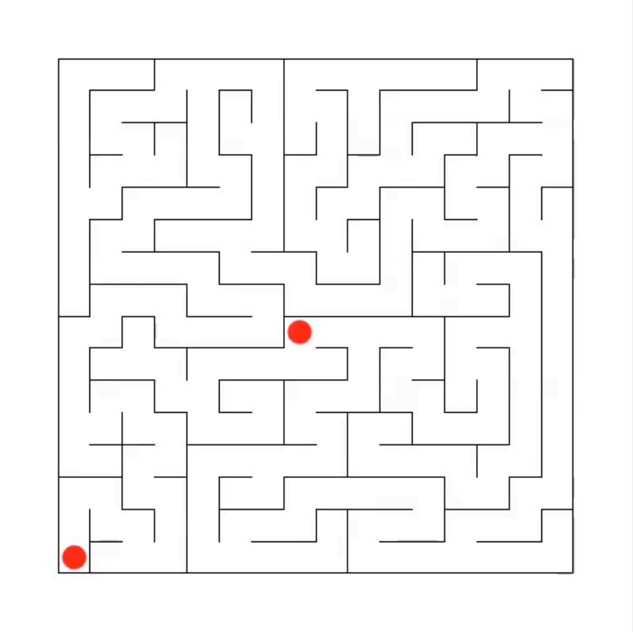
http://commons.wikimedia.org/wiki/File:Minotaurus.gif



Claude Shannon (with electromechanical mouse)

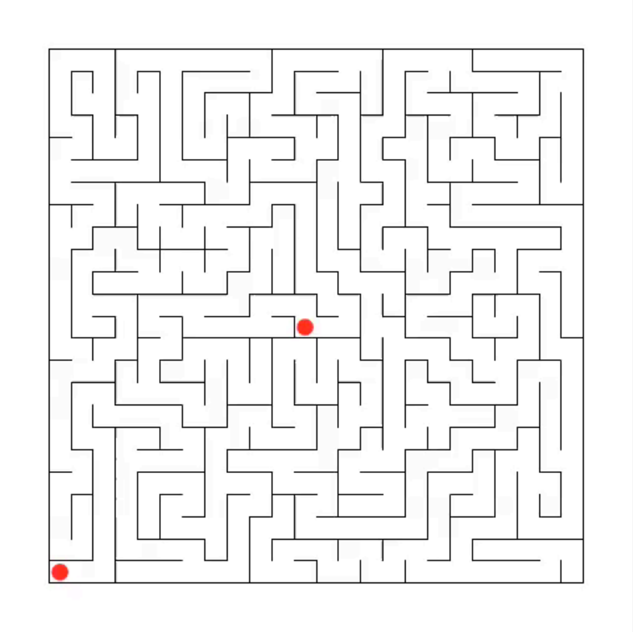
http://www.corp.att.com/attlabs/reputation/timeline/16shannon.html

## Maze exploration: easy



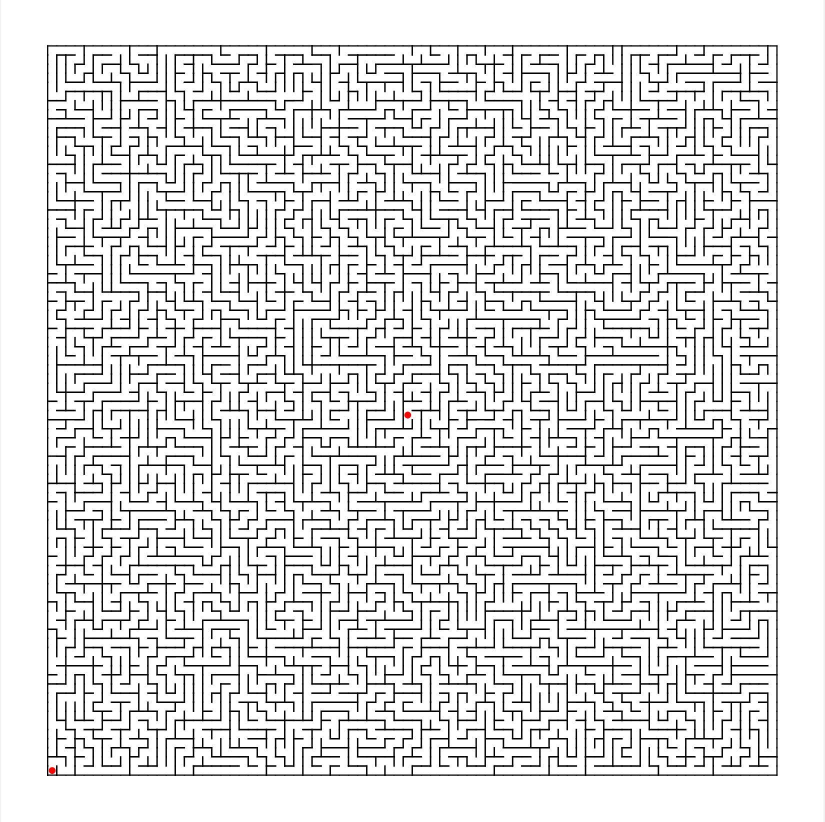


## Maze exploration: medium





## Maze exploration: challenge for the bored





## Depth-first search

Goal. Systematically traverse a graph.

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

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

**Mark vertex v.**
**Recursively visit all unmarked vertices w adjacent to v.**

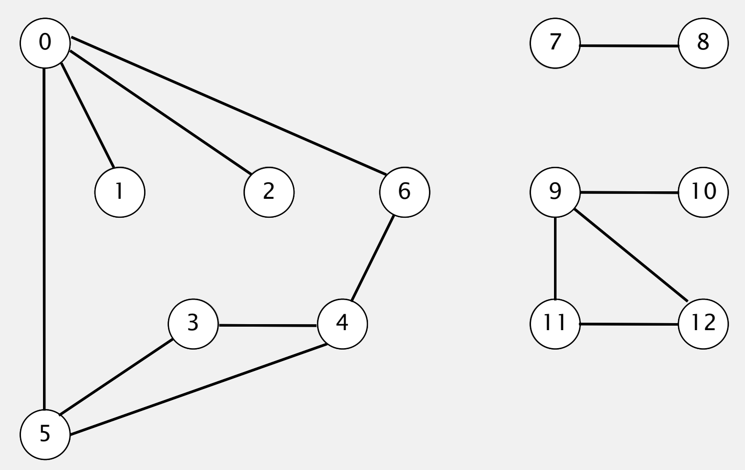
Typical applications.

- 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 $v$.




tinyG.txt

$V$ → 13
13 ← $E$

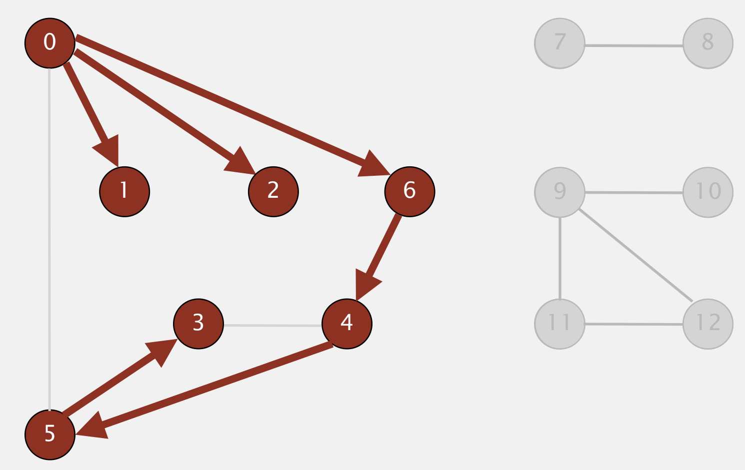
```
0 5
4 3
0 1
9 12
6 4
5 4
0 2
11 12
9 10
0 6
7 8
9 11
5 3
```

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 reachable from 0



## 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



## 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

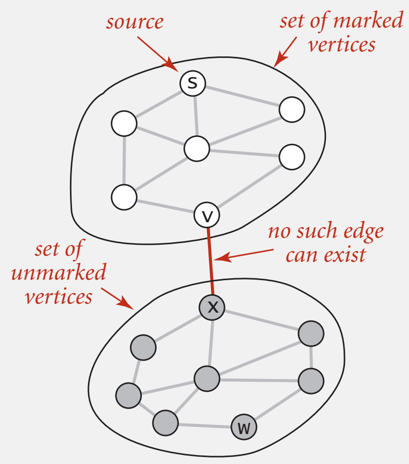


## Depth-first search: properties

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.



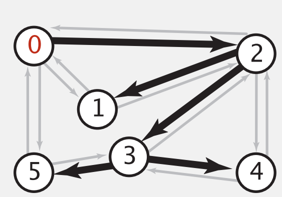
## Depth-first search: properties

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;
}
```




edgeTo[]

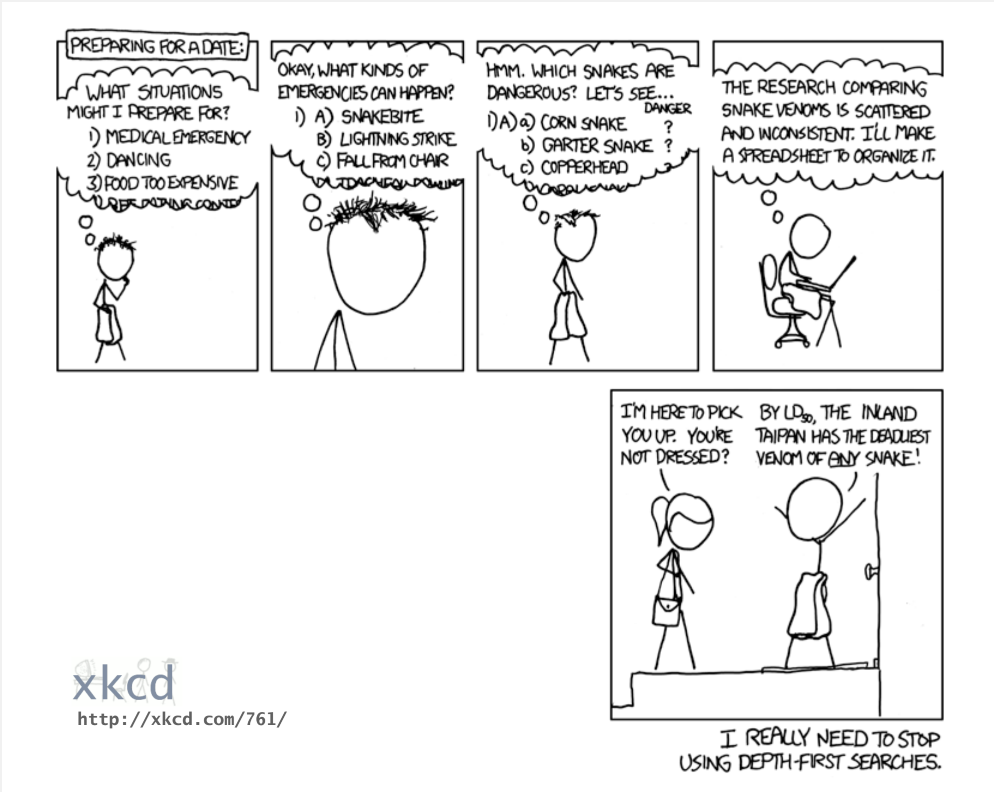
| | |
|---|---|
| 0 | |
| 1 | 2 |
| 2 | 0 |
| 3 | 2 |
| 4 | 3 |
| 5 | 3 |



# Flood Fill

Problem. Implement flood fill (Photoshop magic wand).

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







http://algs4.cs.princeton.edu

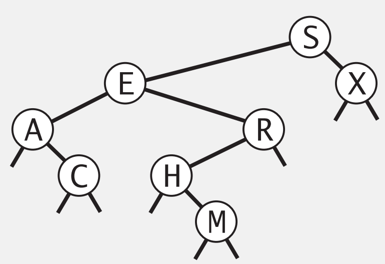
## 4.1 Undirected Graphs

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

## Graph search

Tree traversal. Many ways to explore 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.

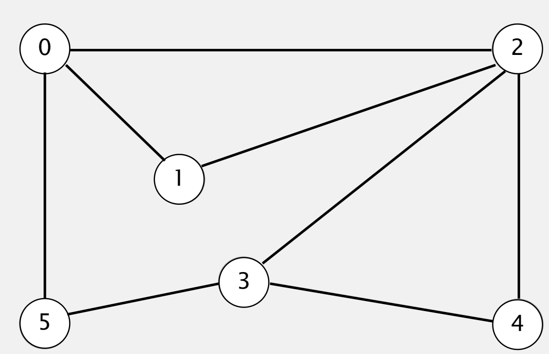
- Preorder: vertices in order DFS calls `dfs(G, v)`.
- Postorder: vertices in order DFS returns from `dfs(G, v)`.
- Level-order: vertices in increasing order of distance from `s`.



## 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

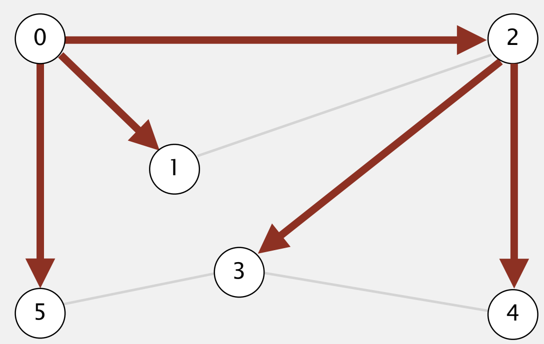
tinyCG.txt

$V$ → 6
8 ← $E$
0 5
2 4
2 3
1 2
0 1
3 4
3 5
0 2

## 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**



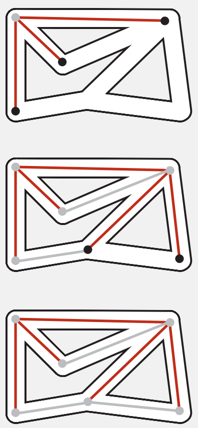
## Breadth-first search

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.**





## 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



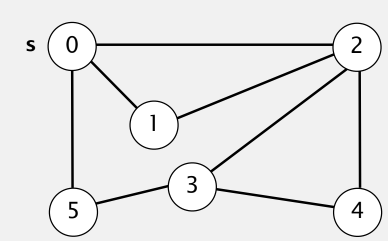
## Breadth-first search properties

Q. In which order does BFS examine vertices?

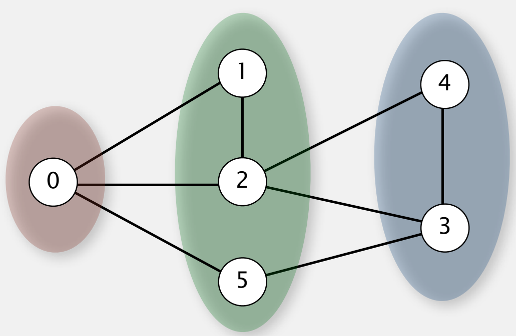
A. Increasing distance (number of edges) from $s$.

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 $E + V$.



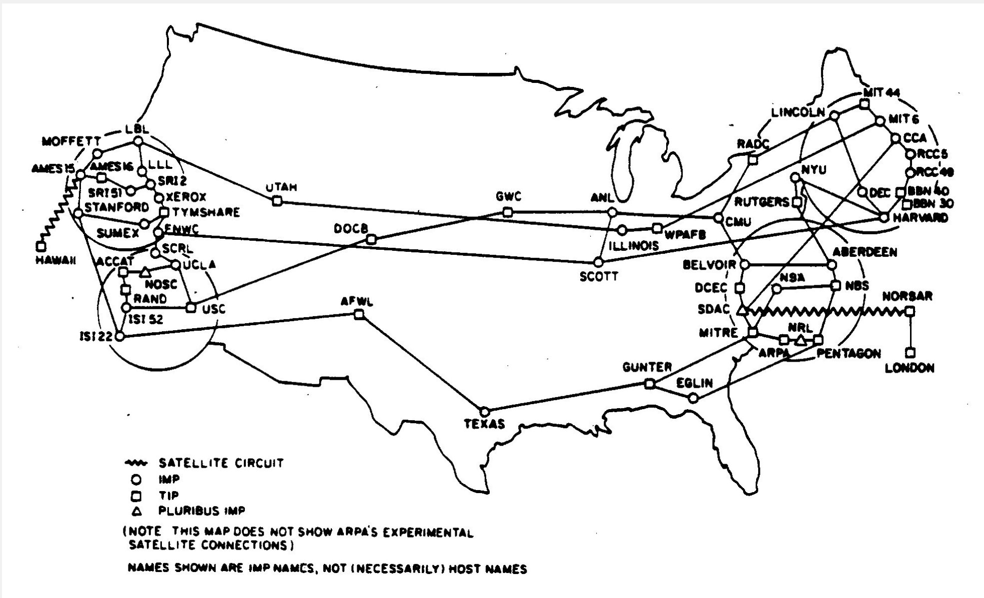

graph G




dist = 0 dist = 1 dist = 2

## Breadth-first search application: routing

Fewest number of hops in a communication network.



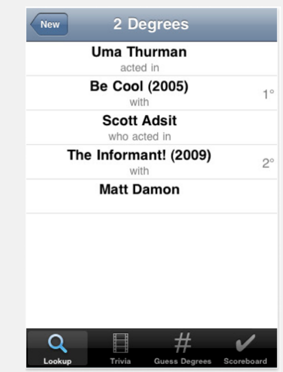

ARPANET, July 1977



## Breadth-first search application: Kevin Bacon numbers


http://oracleofbacon.org


Endless Games board game

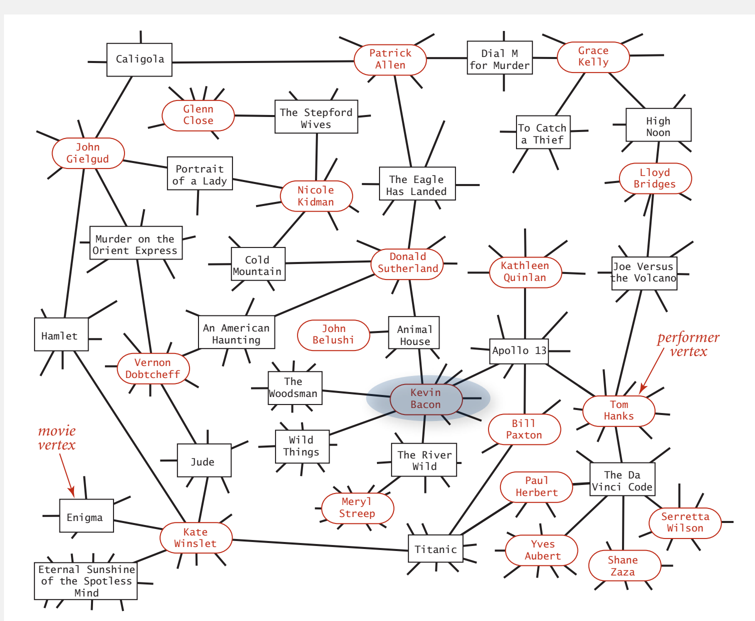



SixDegrees iPhone App



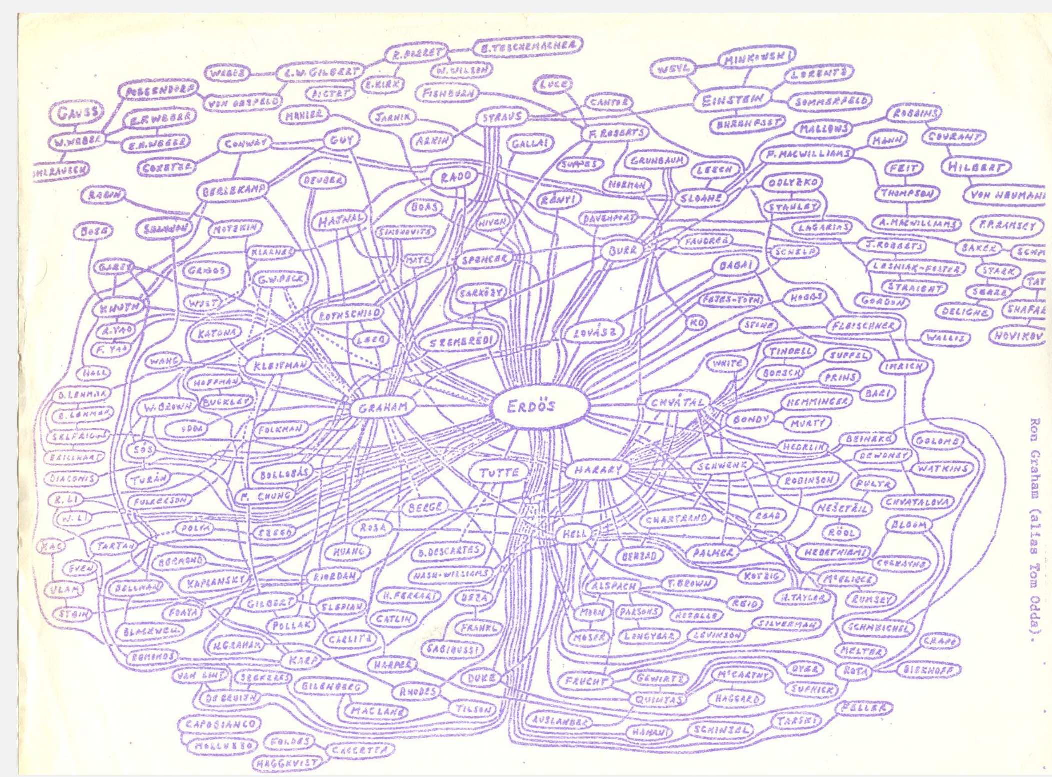
## Kevin Bacon graph

- 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 $s$ = Kevin Bacon.






## Breadth-first search application: Erdös numbers




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

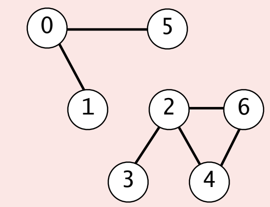

http://algs4.cs.princeton.edu

# 4.1 Undirected Graphs

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

## Graph-processing challenge 1

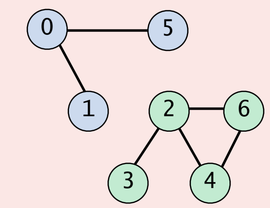
Problem. Identify connected components.




0-1
0-5
2-6
2-3
2-4
4-6

**How difficult?**

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

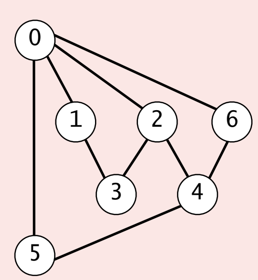



0-1
0-5
2-6
2-3
2-4
4-6



## Graph-processing challenge 2

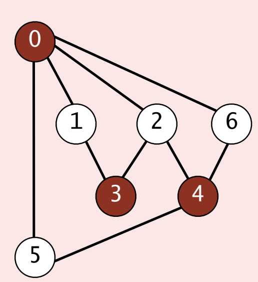
Problem. Is a graph bipartite?




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.** Typical diligent algorithms student could do it.
- **C.** Hire an expert.
- **D.** Intractable.
- **E.** No one knows.



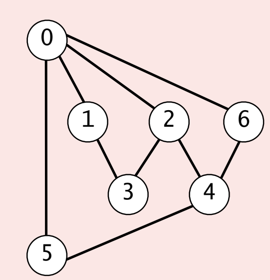

0-1
0-2
0-5
0-6
1-3
2-3
2-4
4-5
4-6

{ 0, 3, 4 }



## 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

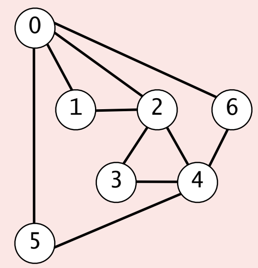
0-5-4-6-0

**How difficult?**

- **A.** Any programmer could do it.
- **B.** Typical 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

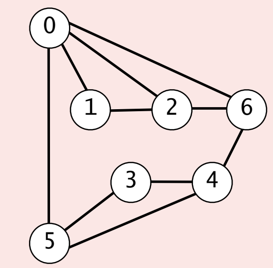
**How difficult?**

- **A.** Any programmer could do it.
- **B.** Typical 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 contains 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

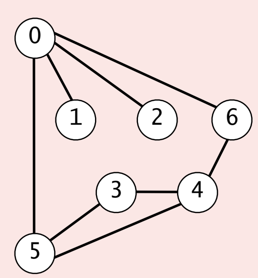
**How difficult?**

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



## Graph-processing challenge 6

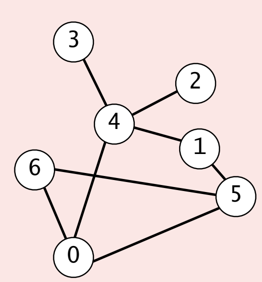
Problem. Are two graphs identical except for vertex names?




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.** Typical 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

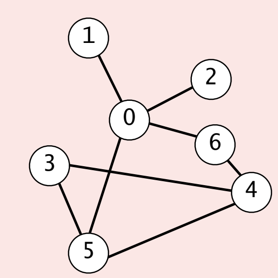
0↔4, 1↔3, 2↔2, 3↔6, 4↔5, 5↔0, 6↔1



## Graph-processing challenge 7

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

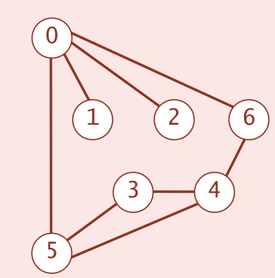
try it yourself at http://planarity.net




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.** Typical 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** | | | $2^{1.657\,V}$ |
| **bipartiteness (odd cycle)** | ✔ | ✔ | $E + V$ |
| **connected components** | ✔ | ✔ | $E + V$ |
| **biconnected components** | | ✔ | $E + V$ |
| **planarity** | | ✔ | $E + V$ |
| **graph isomorphism** | | | $2^{c\sqrt{V \log V}}$ |