http://algs4.cs.princeton.edu

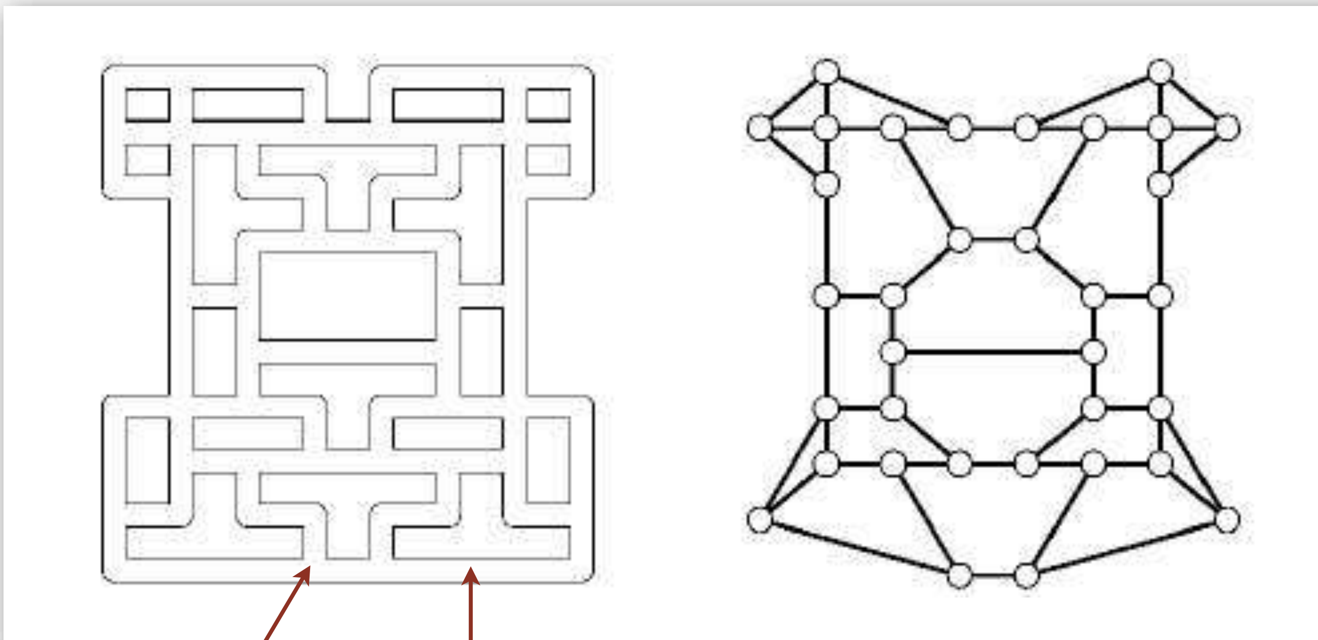
4.1 Undirected Graphs

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

Maze exploration

Maze graph.

- Vertex = intersection.
- Edge = passage.



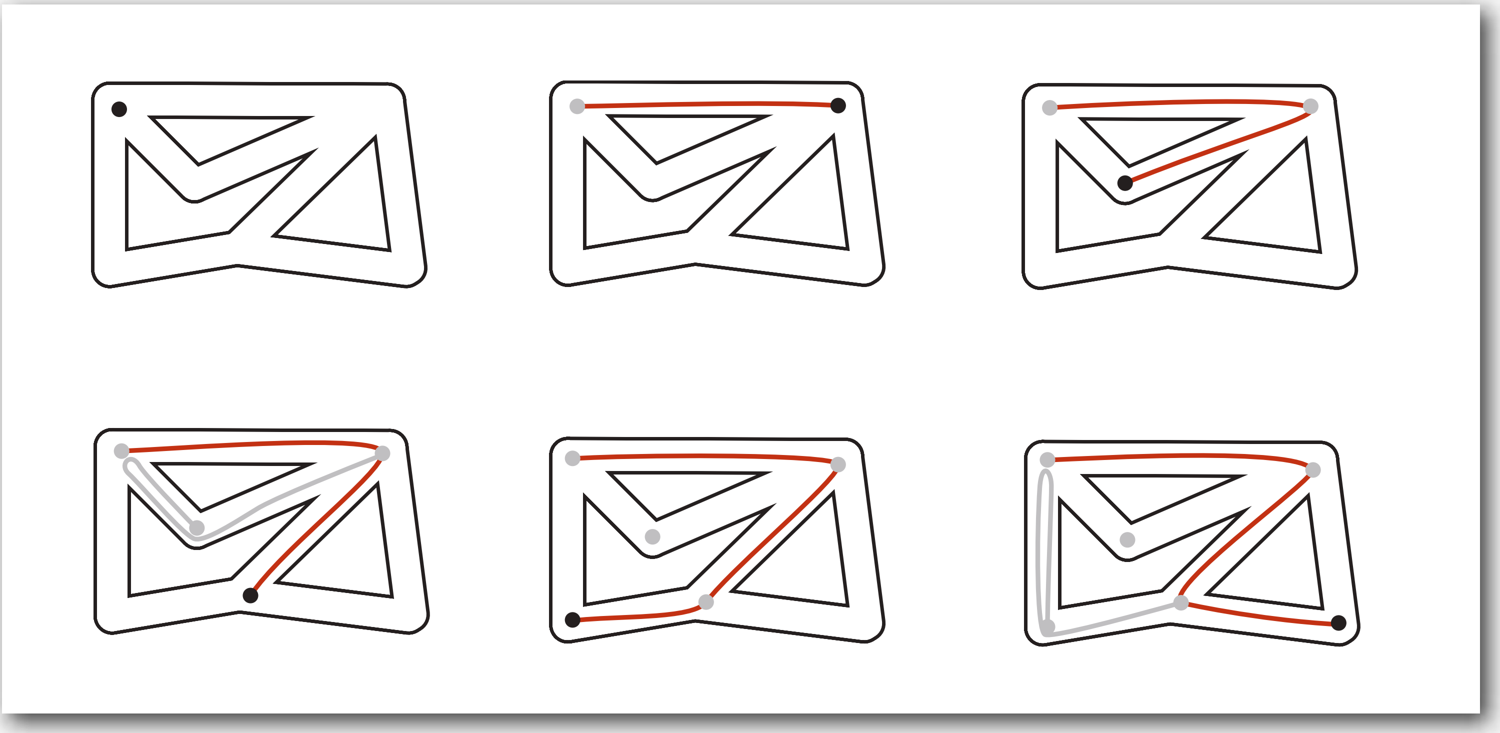
intersection passage

Goal. Explore every intersection in the maze.

Trémaux maze exploration

Algorithm.

- Unroll a ball of string behind you.
- Mark each visited intersection and each visited passage.
- Retrace steps when no unvisited options.



Trémaux maze exploration

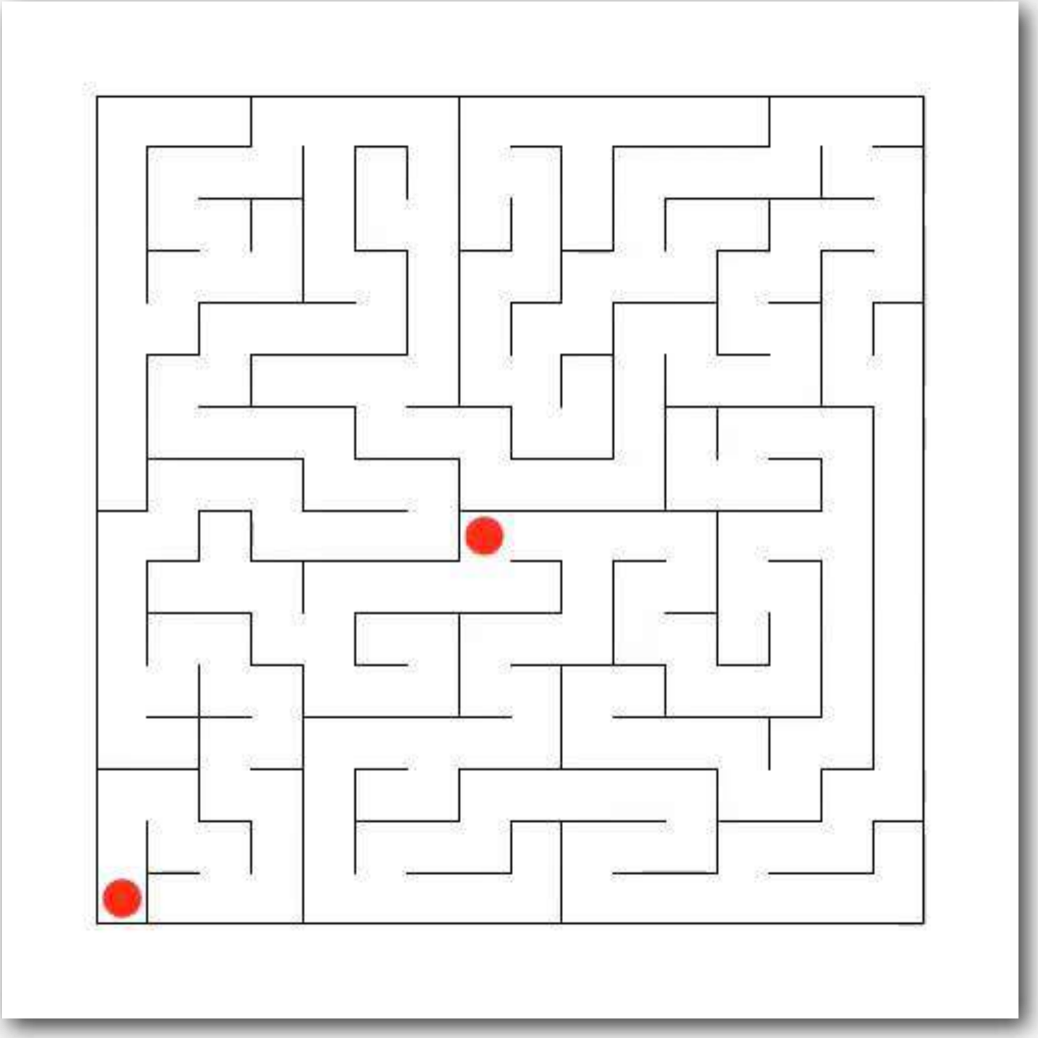
Algorithm.

- Unroll a ball of string behind you.
- Mark each visited intersection and each visited passage.
- Retrace steps when no unvisited options.

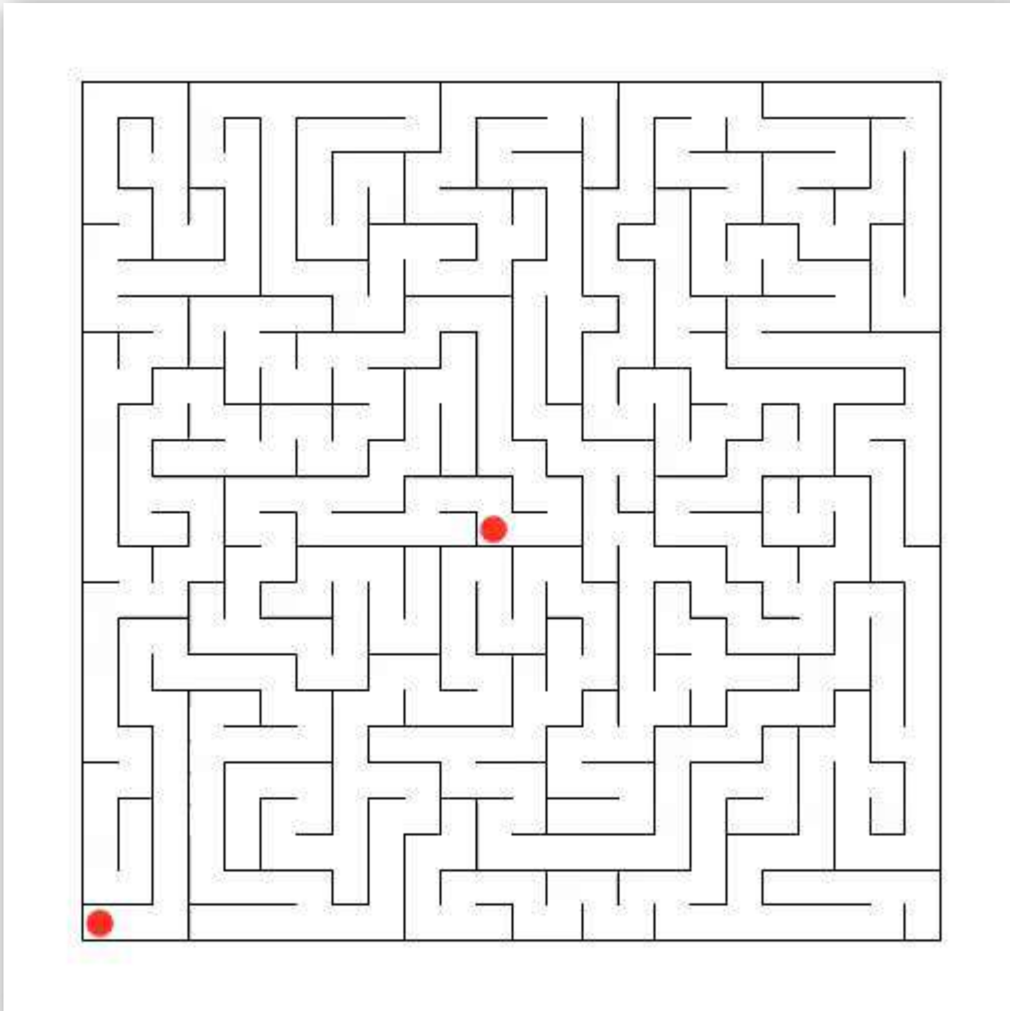
First use? Theseus entered Labyrinth to kill the monstrous Minotaur; Ariadne instructed Theseus to use a ball of string to find his way back out.

Claude Shannon (with Theseus mouse)

Maze exploration



Maze exploration



Depth-first search

Goal. Systematically search through a graph.

Idea. Mimic maze exploration.

DFS (to visit a vertex v)

Mark v as visited.

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?

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*

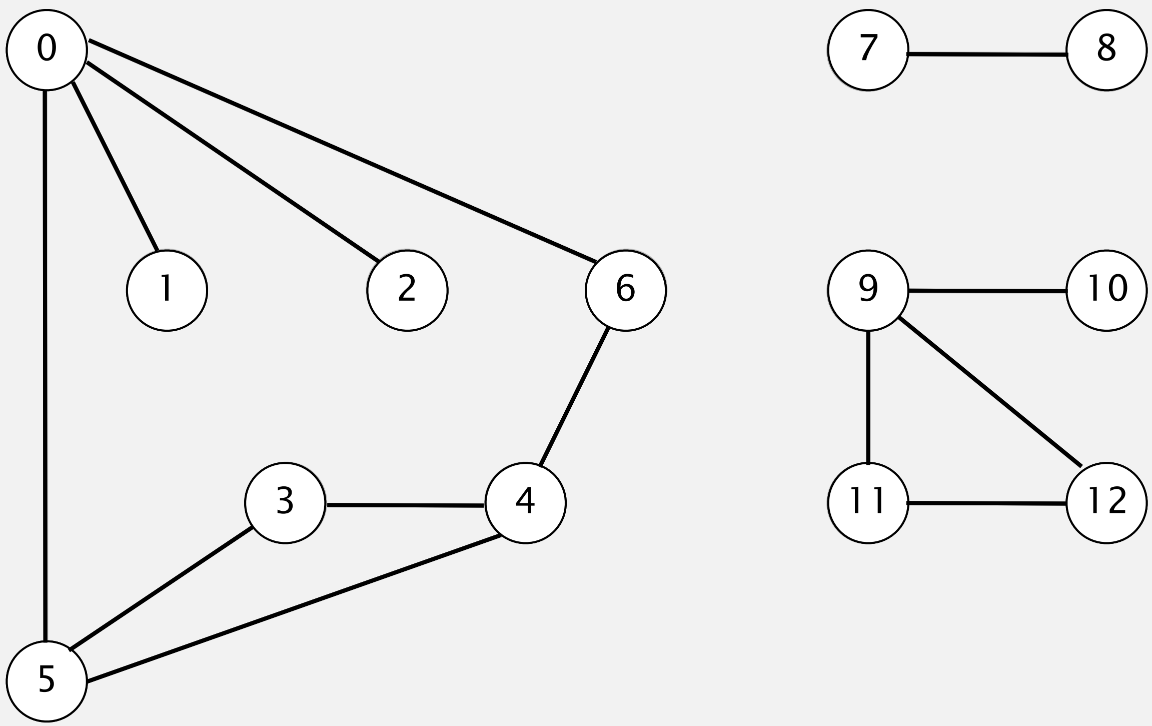
```
Paths paths = new Paths(G, s);
for (int v = 0; v < G.V(); v++)
   if (paths.hasPathTo(v))
      StdOut.println(v);
```

print all vertices connected to s

Depth-first search demo

To visit a vertex v :

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



tinyG.txt

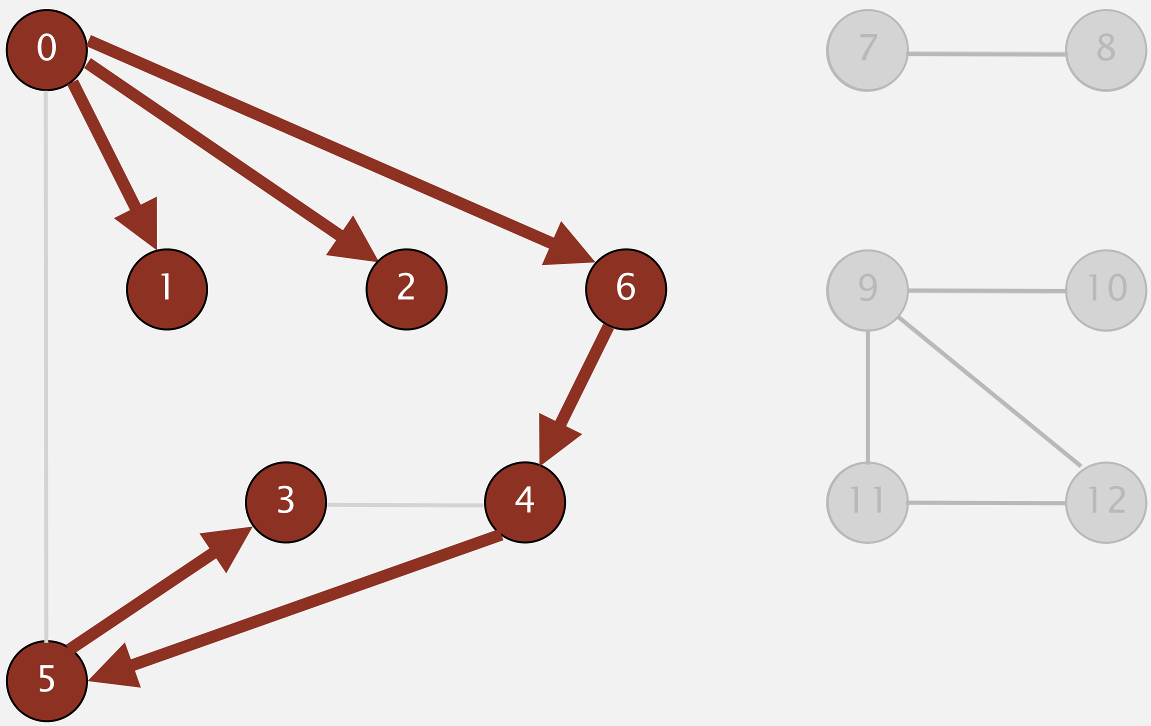
V → 13
13 ← E
0 5
4 3
0 1
9 12
6 4
5 4
0 2
11 12
9 10
0 6
7 8
9 11
5 3

graph G

Depth-first search demo

To visit a vertex v :

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



v	marked[]	edgeTo[v]
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

Depth-first search

Goal. Find all vertices connected to s (and a corresponding path).

Idea. Mimic maze exploration.

Algorithm.

- Use recursion (ball of string).
- Mark each visited vertex (and keep track of edge taken to visit it).
- Return (retrace steps) when no unvisited options.

Data structures.

- `boolean[] marked` to mark visited vertices.
- `int[] edgeTo` to keep tree of paths.

  `(edgeTo[w] == v)` means that edge `v-w` taken to visit `w` for first time

Depth-first search

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

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

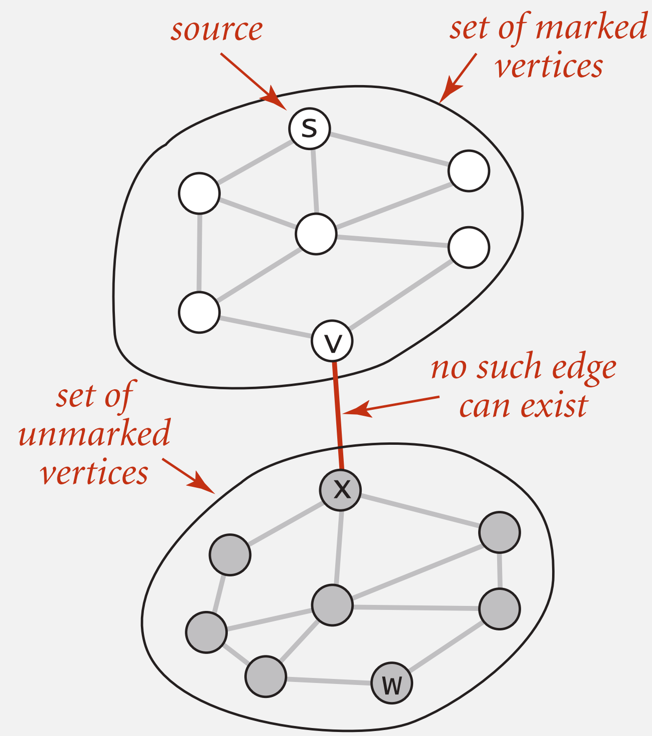
recursive DFS does the work

Depth-first search properties

Proposition. DFS marks all vertices connected to s in time proportional to the sum of their degrees.

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]

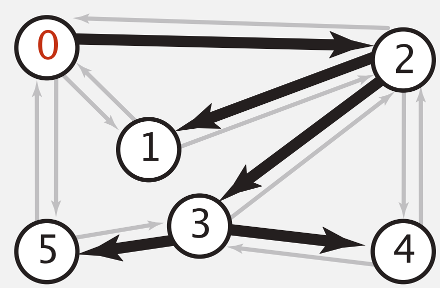
Each vertex connected to s is visited once.

Depth-first search properties

Proposition. After DFS, can find vertices connected to s in constant time and can find a path to s (if one exists) in time proportional to its length.

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

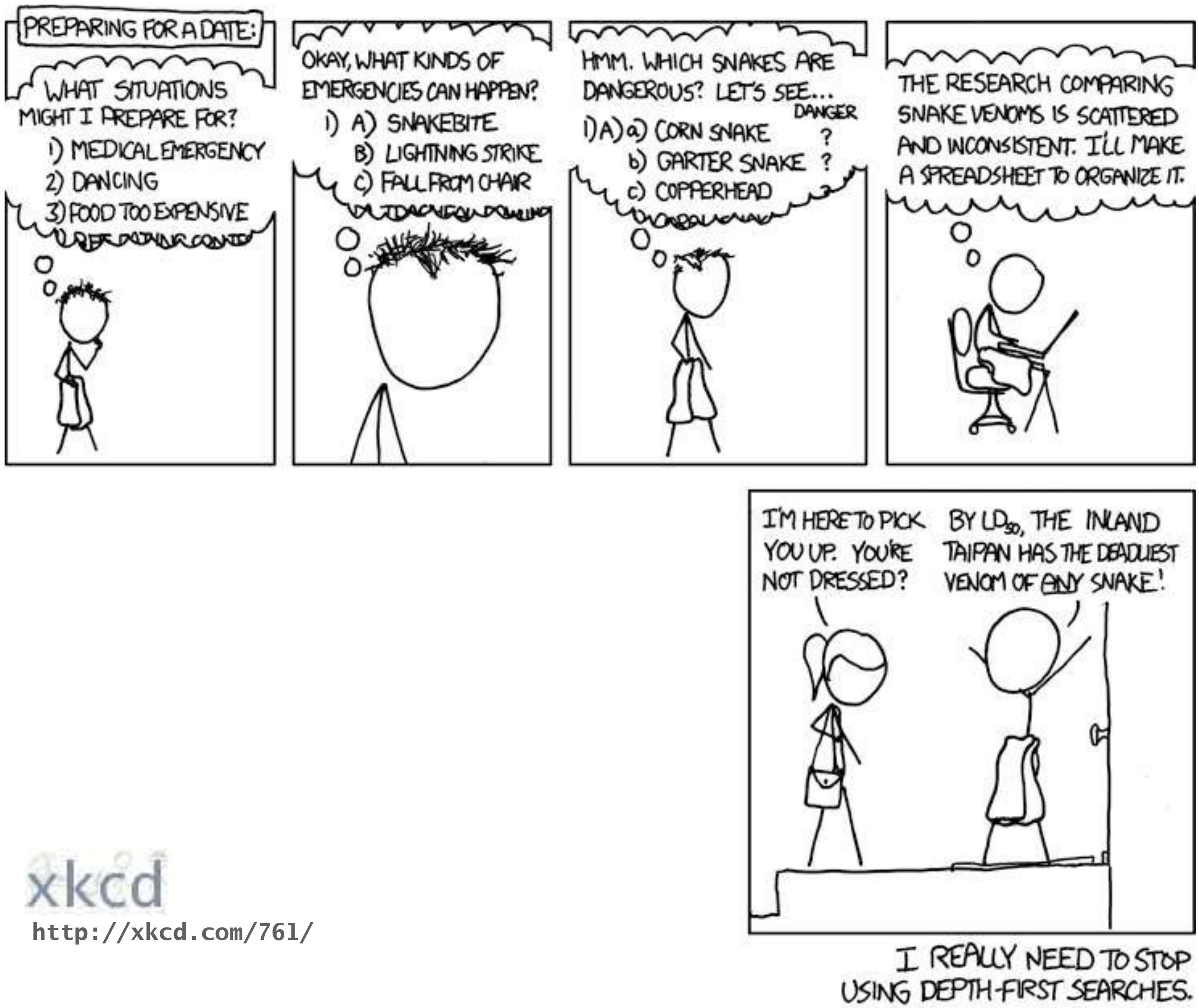
```
public boolean hasPathTo(int v)
{  return marked[v];  }

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

Depth-first search application: preparing for a date



http://xkcd.com/761/

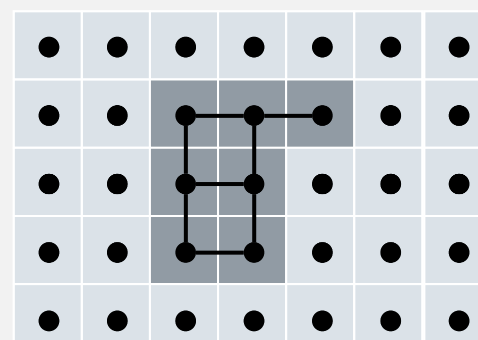
Depth-first search application: flood fill

Challenge. Flood fill (Photoshop magic wand).

Assumptions. Picture has millions to billions of pixels.

Solution. Build a grid graph.

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



http://algs4.cs.princeton.edu

4.1 Undirected Graphs

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

http://algs4.cs.princeton.edu

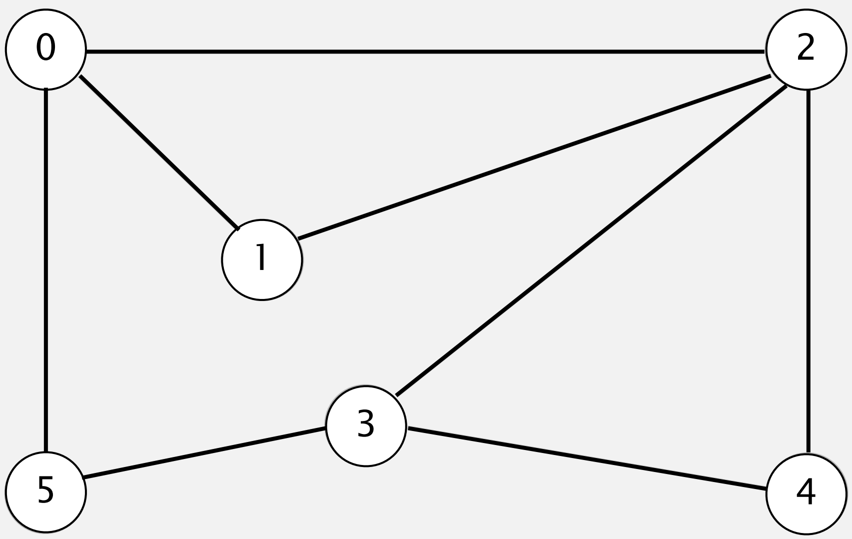
4.1 UNDIRECTED GRAPHS

- *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

E → 8

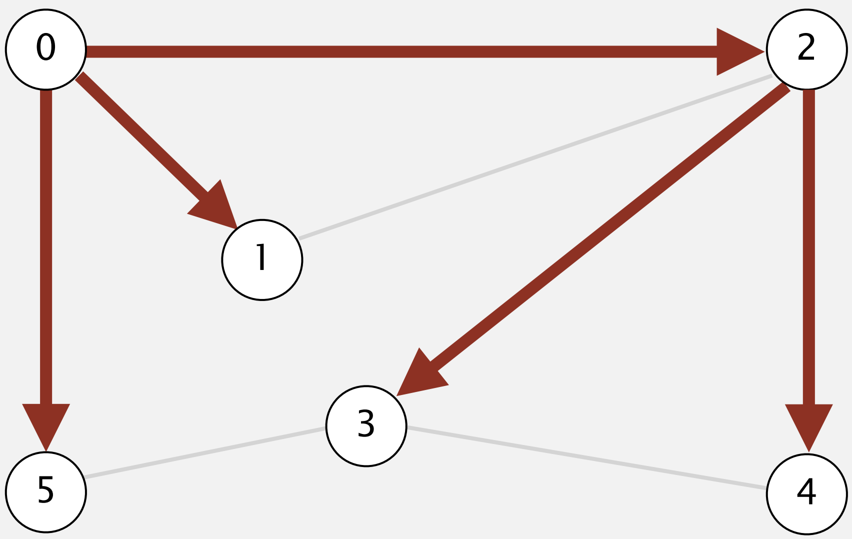
```
0 5
2 4
2 3
1 2
0 1
3 4
3 5
0 2
```

graph G

Breadth-first search demo

Repeat until queue is empty:

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



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

done

Breadth-first search

Depth-first search. Put unvisited vertices on a stack.

Breadth-first search. Put unvisited vertices on a queue.

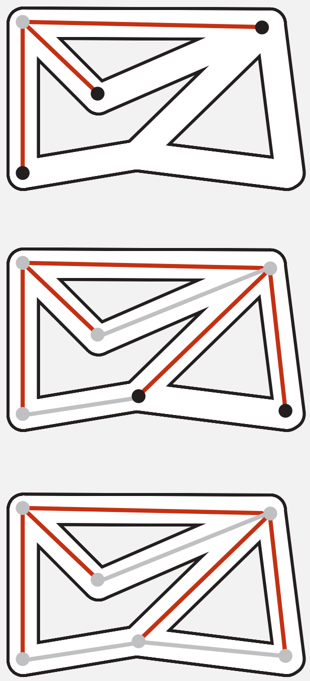
Shortest path. Find path from s to t that uses fewest number of edges.

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 unvisited neighbors to the queue, and mark them as visited.**



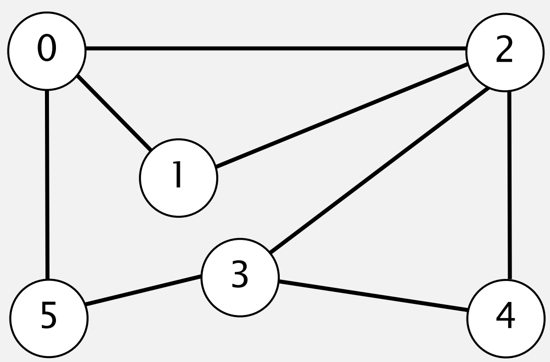
Intuition. BFS examines vertices in increasing distance from s.

Breadth-first search properties

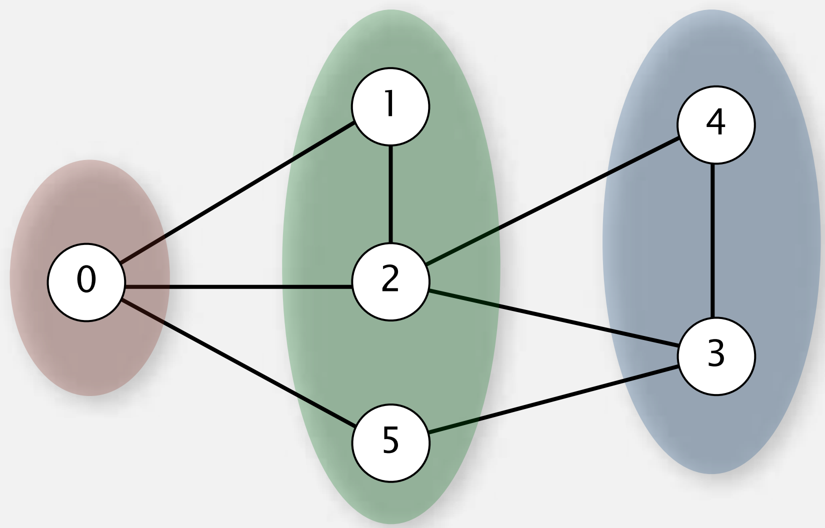
Proposition. BFS computes shortest paths (fewest number of edges) from s to all other vertices in a graph in time proportional to $E + V$.

Pf. [correctness] Queue always consists of zero or more vertices of distance k from s, followed by zero or more vertices of distance $k + 1$.

Pf. [running time] Each vertex connected to s is visited once.



graph



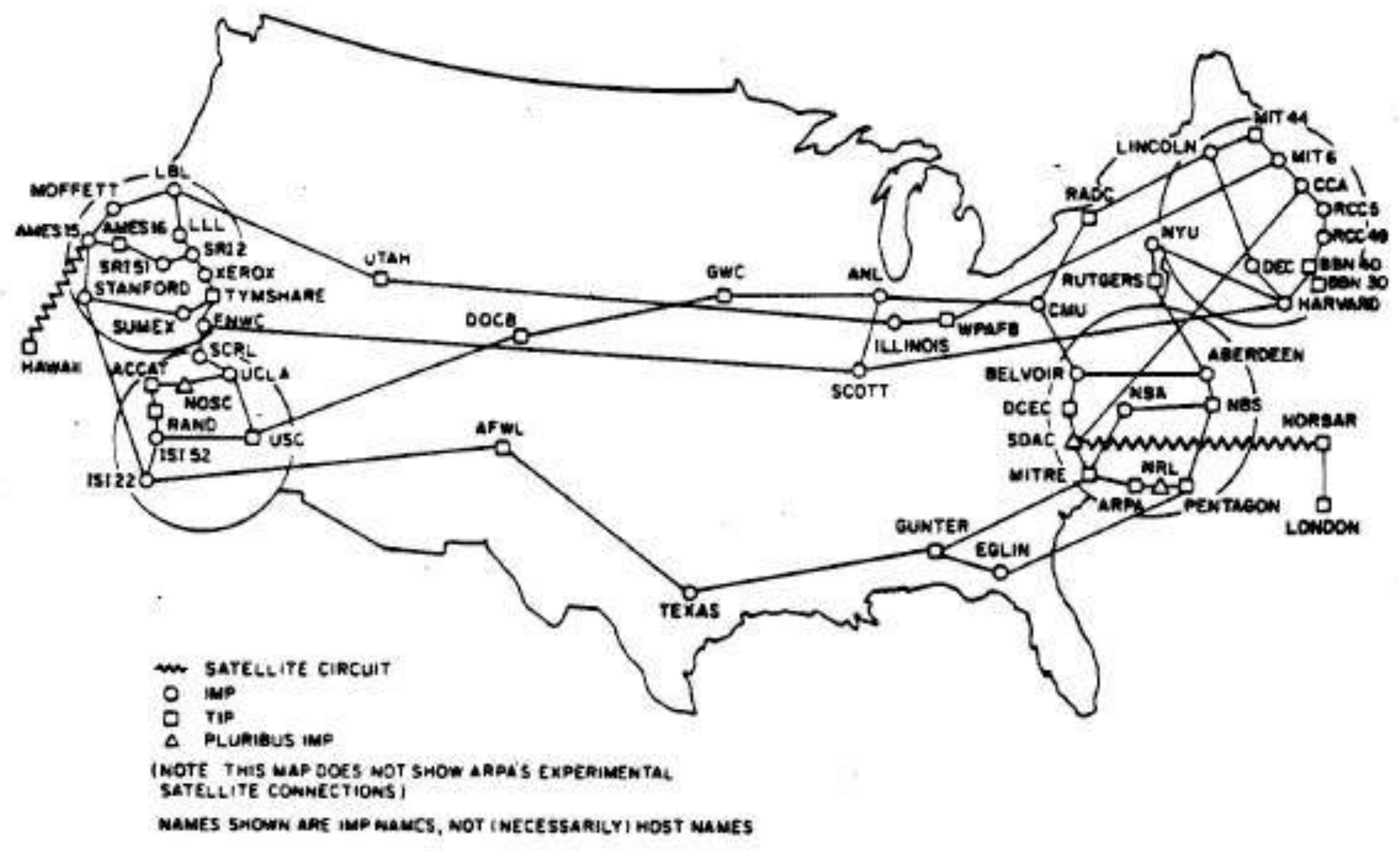
dist = 0 dist = 1 dist = 2

Breadth-first search

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

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

Breadth-first search application: routing

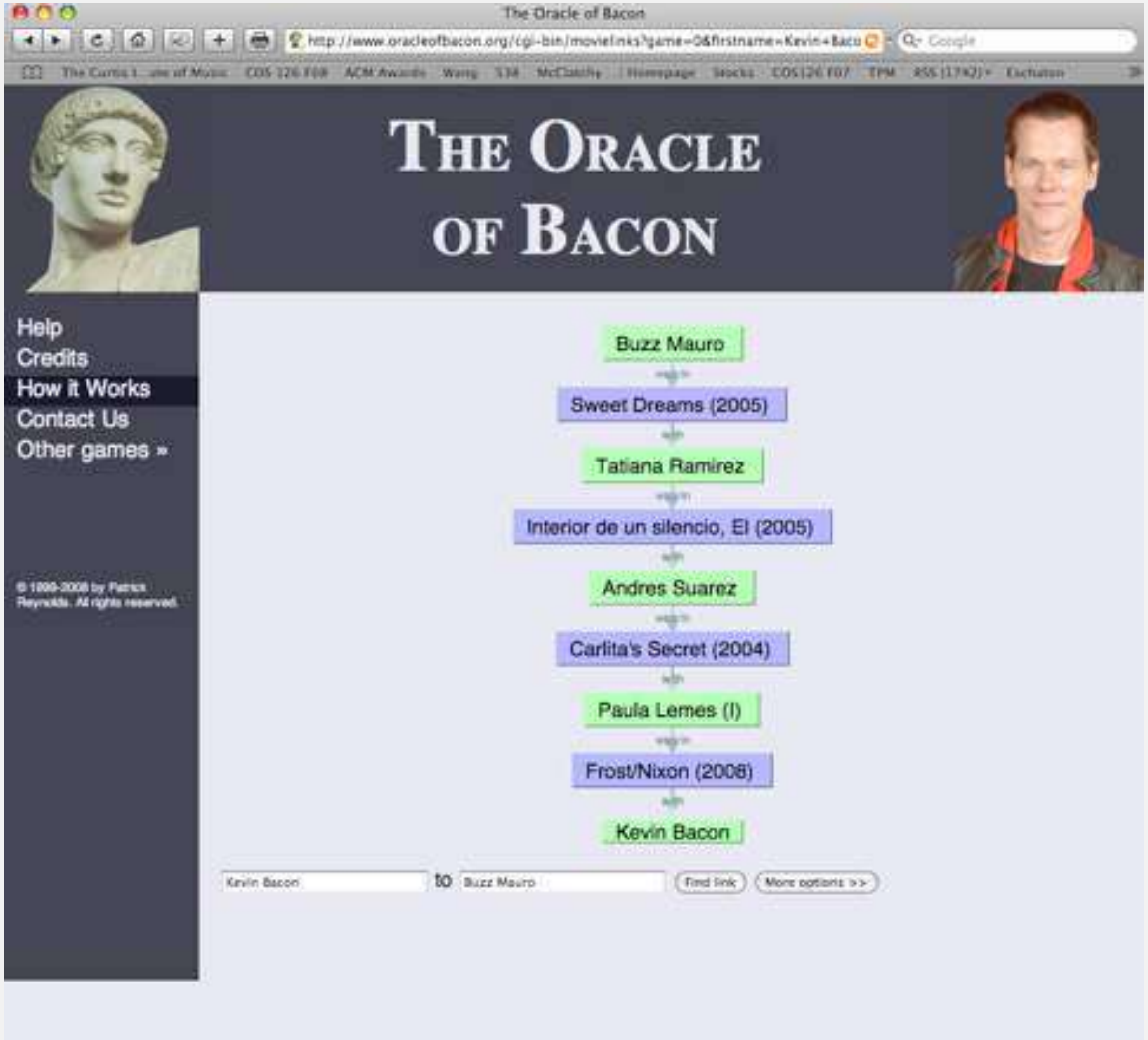
Fewest number of hops in a communication network.



ARPANET, July 1977

Breadth-first search application: Kevin Bacon numbers

Kevin Bacon numbers.



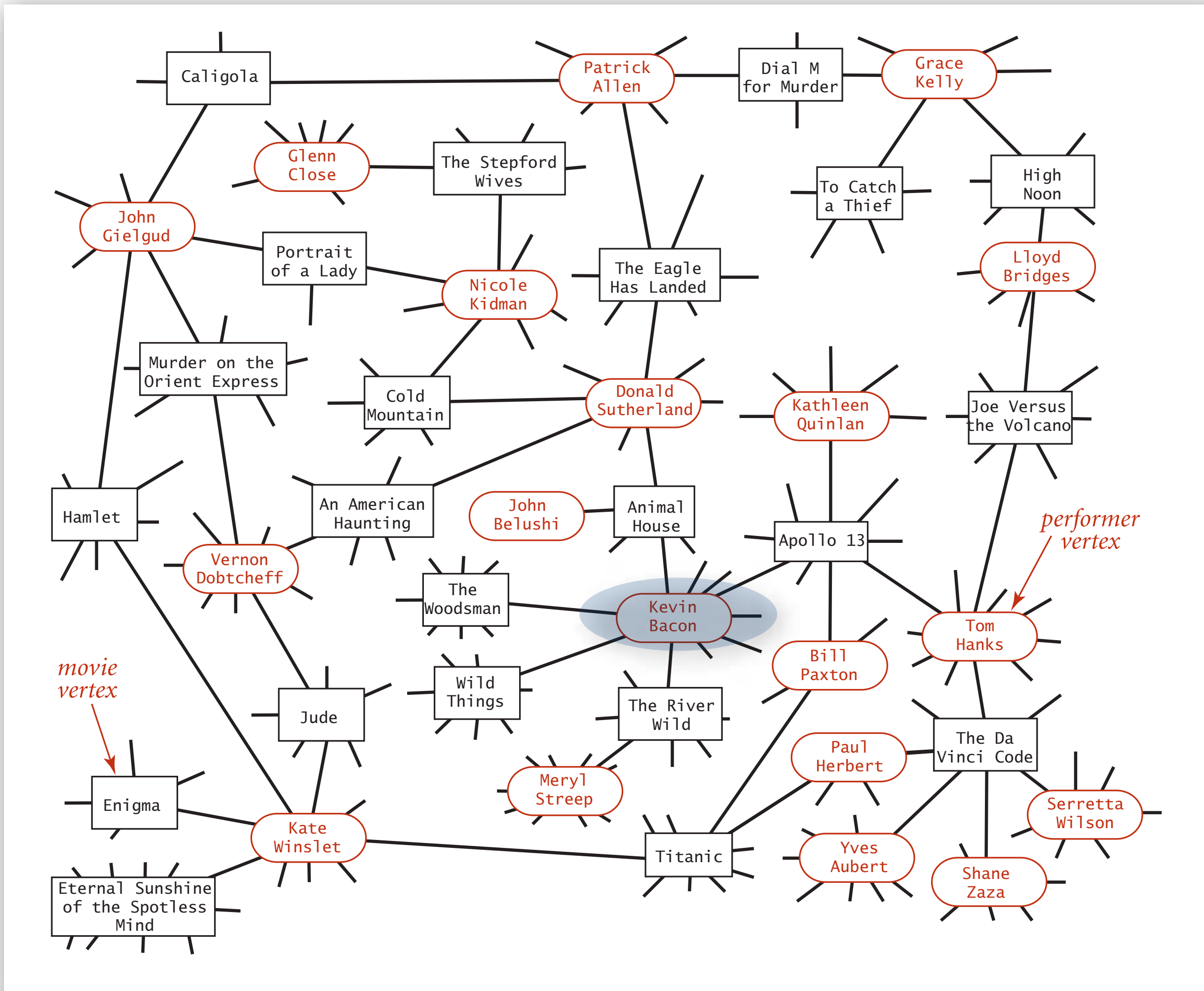
http://oracleofbacon.org

Endless Games board game

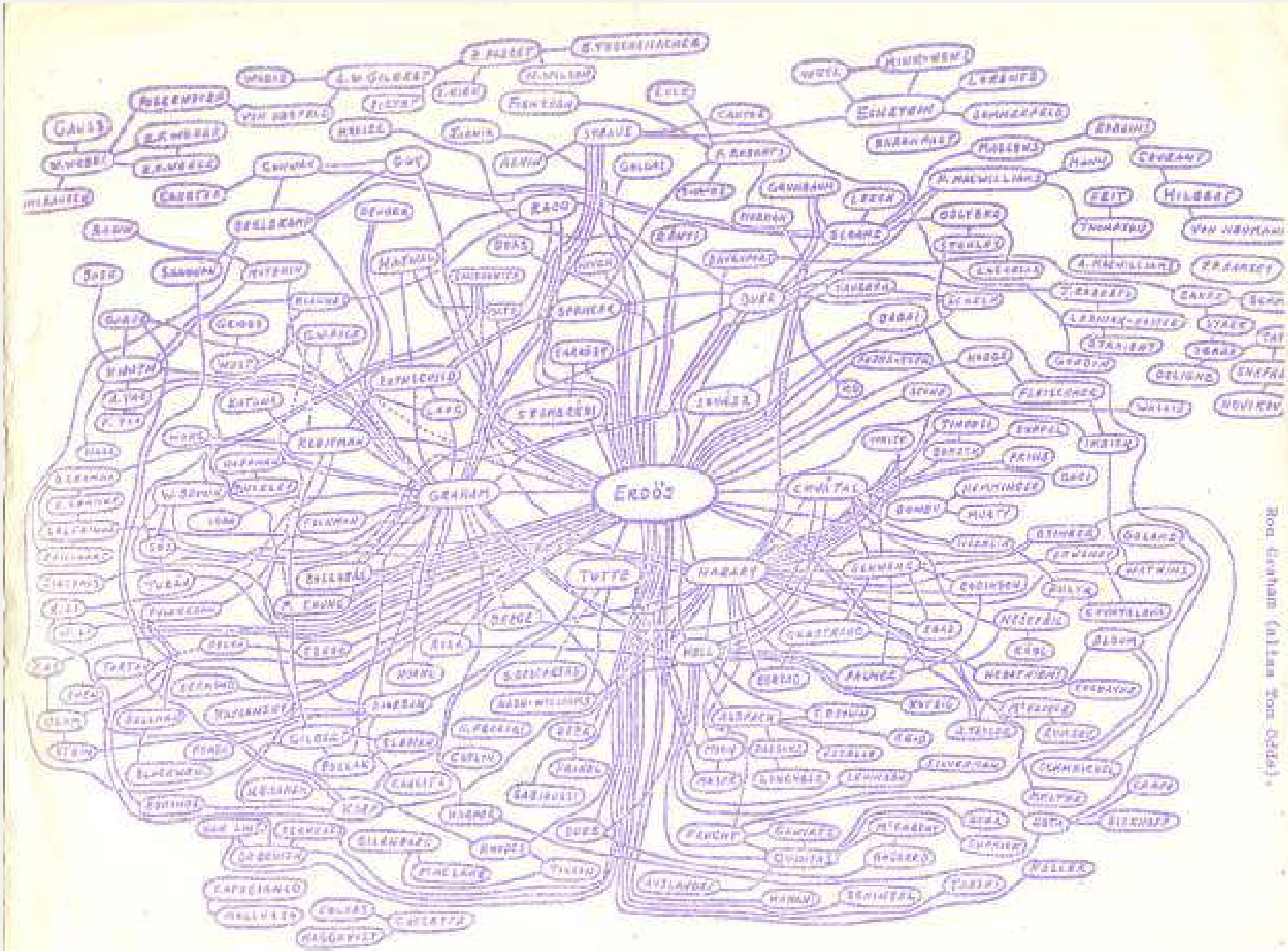
SixDegrees iPhone App

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

Algorithms

Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

4.1 Undirected Graphs

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

http://algs4.cs.princeton.edu

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(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*

Union-Find? Not quite.

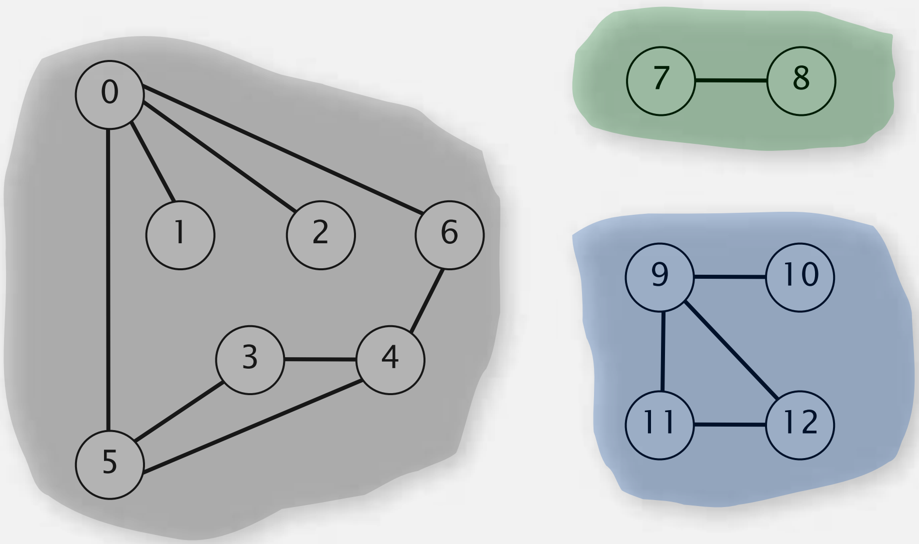
Depth-first search. Yes. [next few slides]

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.



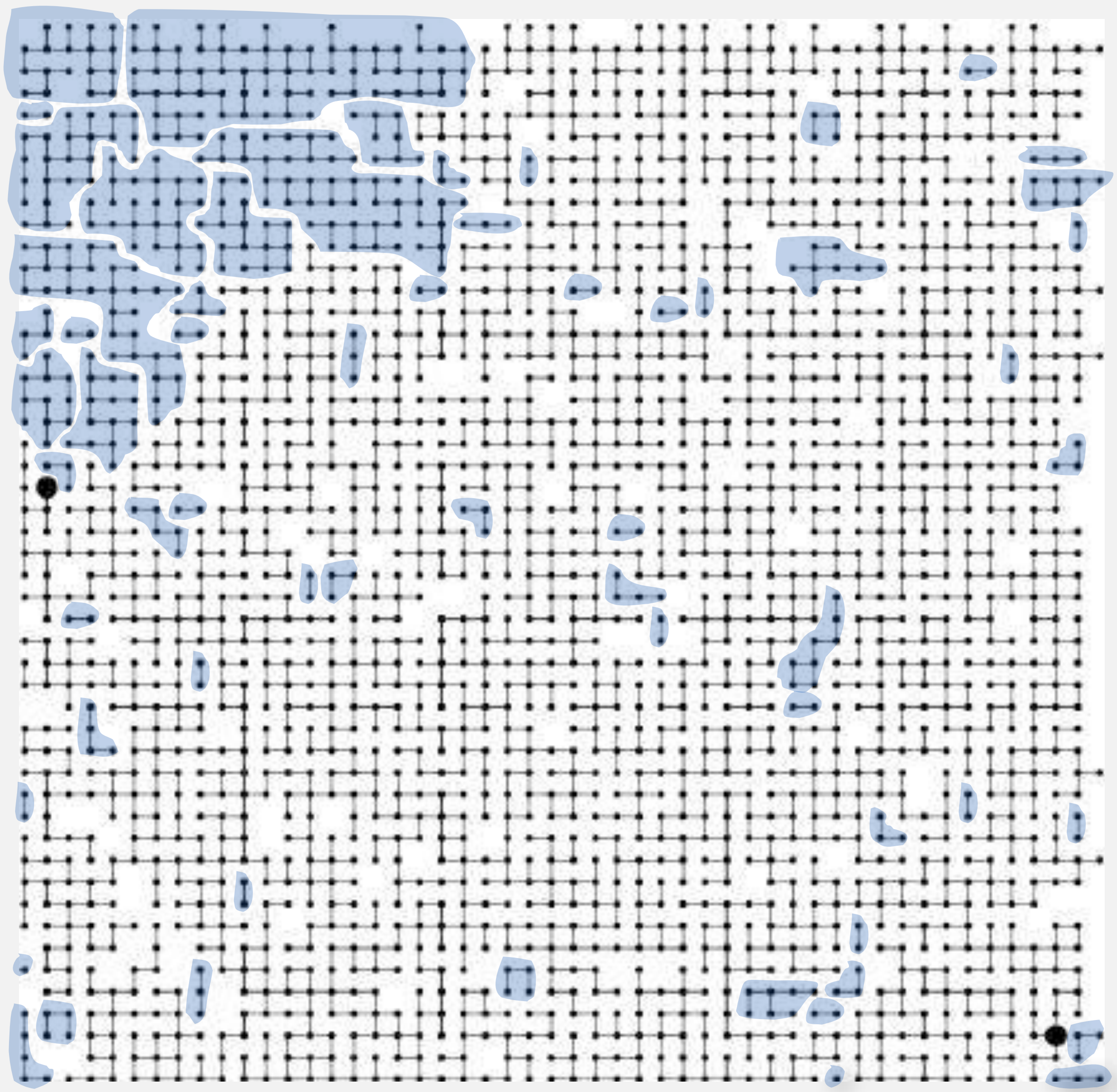
3 connected components

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

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

Connected components

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

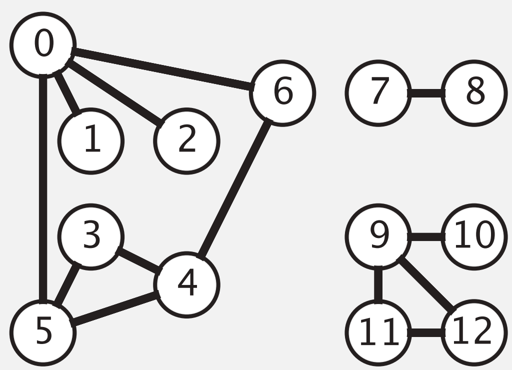


63 connected components

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.**



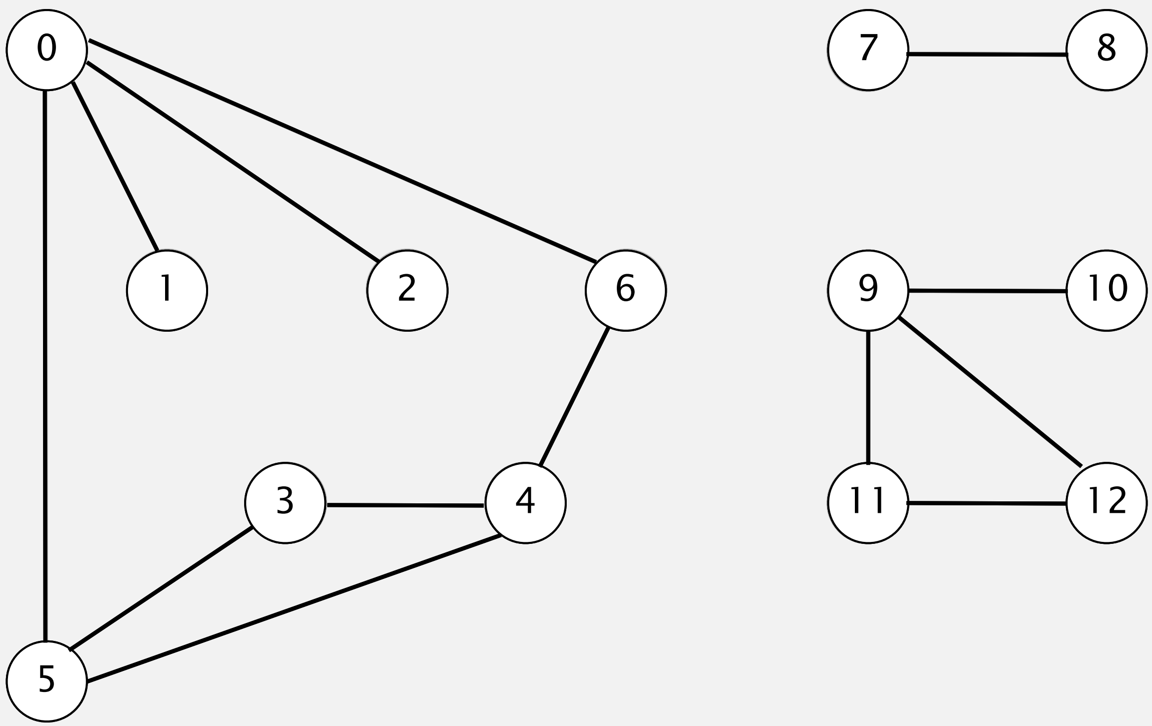
tinyG.txt

```
13   ← V
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 as visited.
- Recursively visit all unmarked vertices adjacent to v.



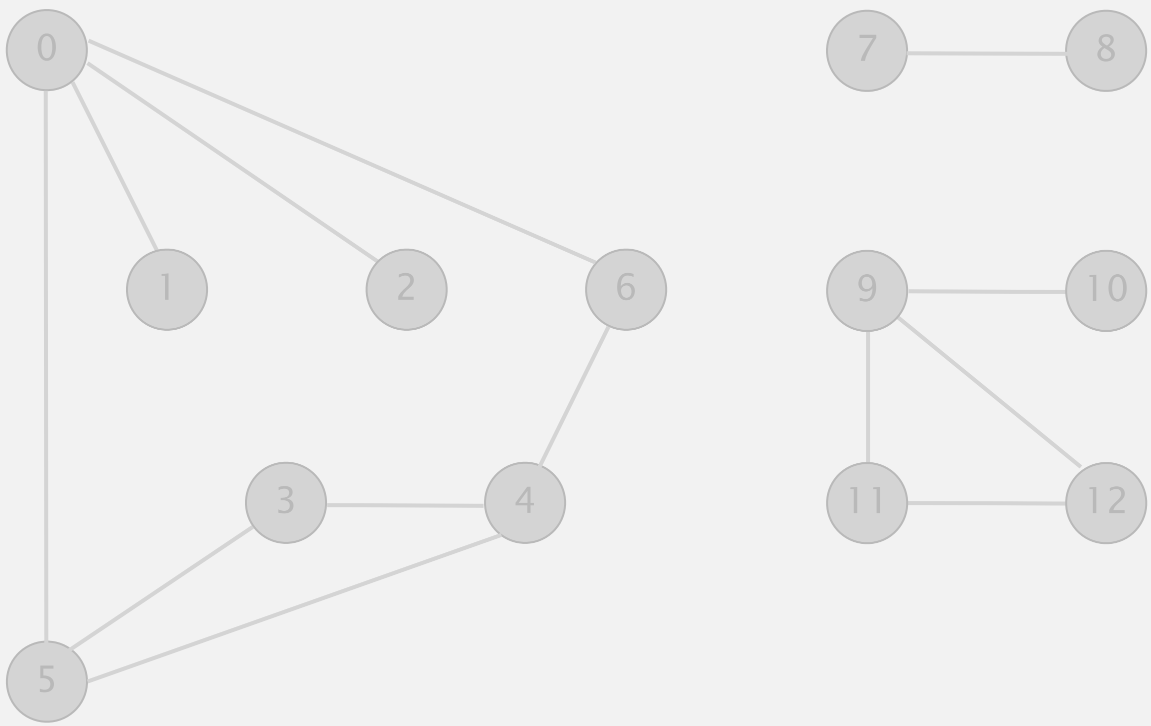
v	marked[]	id[]
0	F	–
1	F	–
2	F	–
3	F	–
4	F	–
5	F	–
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

graph G

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- 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

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)
   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

Finding connected components with DFS (continued)

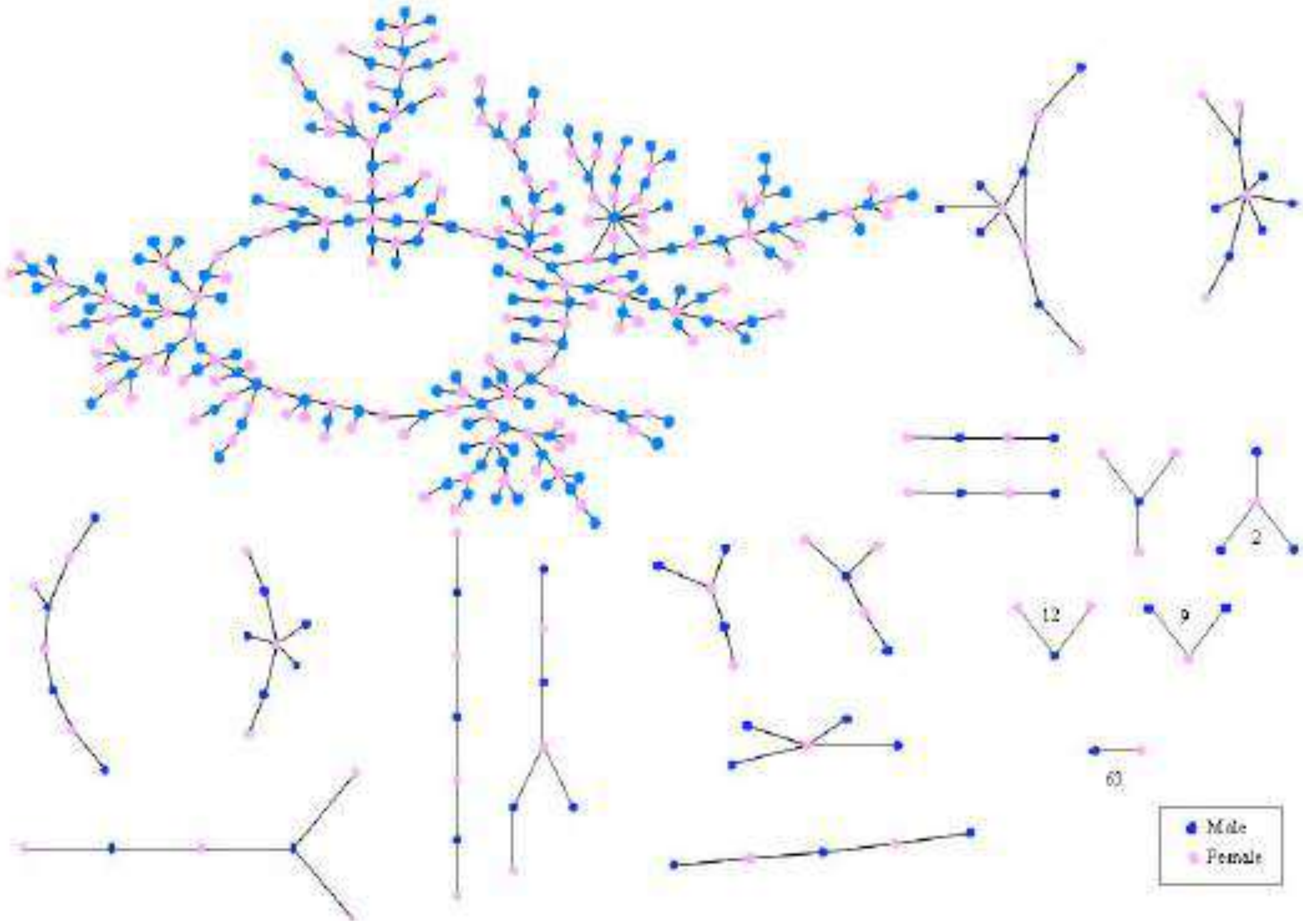
```
public int count()
{  return count;  }

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

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

all vertices discovered in same call of dfs have same id

Connected components application: study spread of STDs



Relationship graph at "Jefferson High"

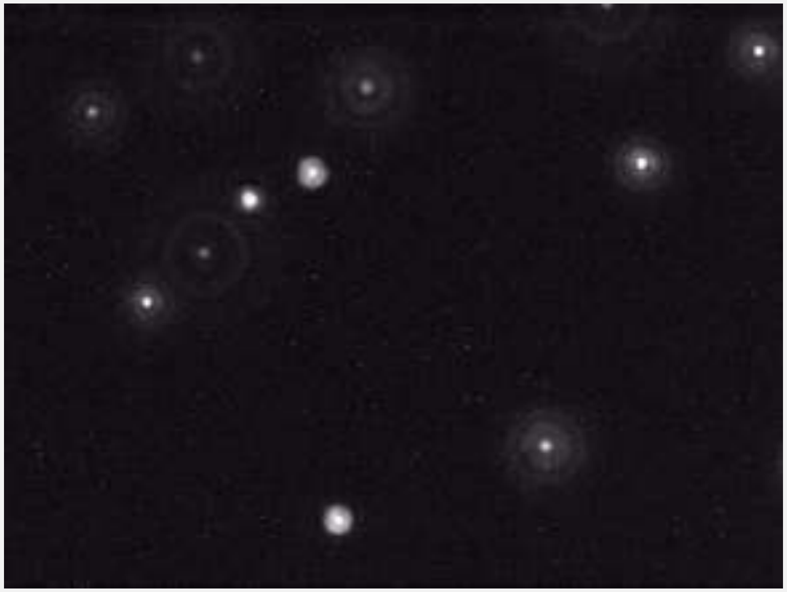
Peter Bearman, James Moody, and Katherine Stovel. Chains of affection: The structure of adolescent romantic and sexual networks. American Journal of Sociology, 110(1): 44–99, 2004.

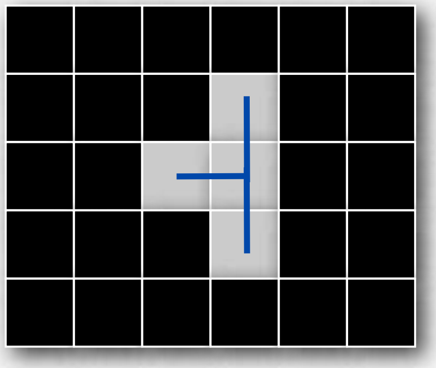
Connected components application: particle detection

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.

black = 0
white = 255





Particle tracking. Track moving particles over time.

http://algs4.cs.princeton.edu

4.1 Undirected Graphs

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

http://algs4.cs.princeton.edu

4.1 Undirected Graphs

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

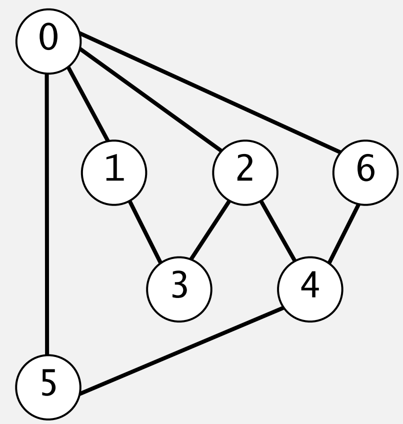
Graph-processing challenge 1

Problem. Is a graph bipartite?

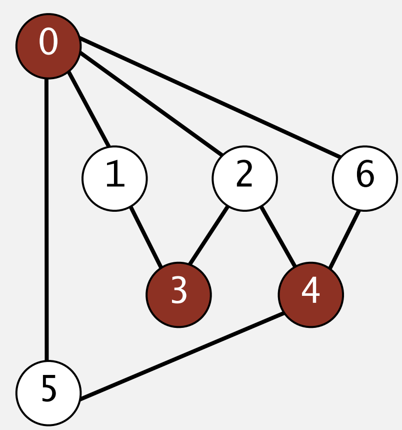
How difficult?

- Any programmer could do it.
- ✓ Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

simple DFS-based solution
(see textbook)



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



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

{ 0, 3, 4 }

Bipartiteness application: is dating graph bipartite?

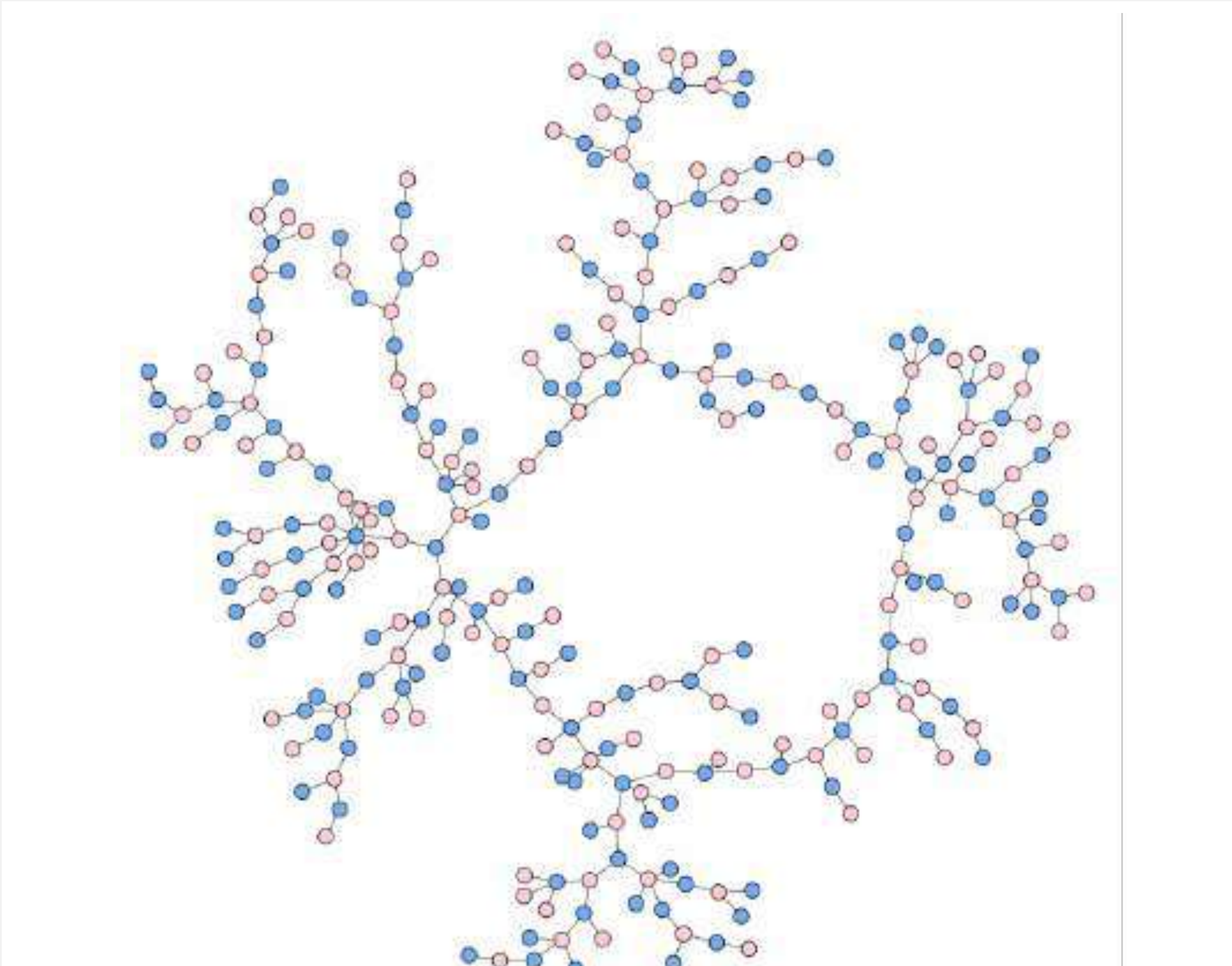
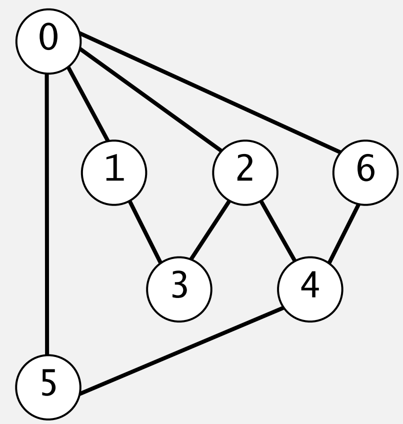


Image created by Mark Newman.

Graph-processing challenge 2

Problem. Find a cycle.



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?

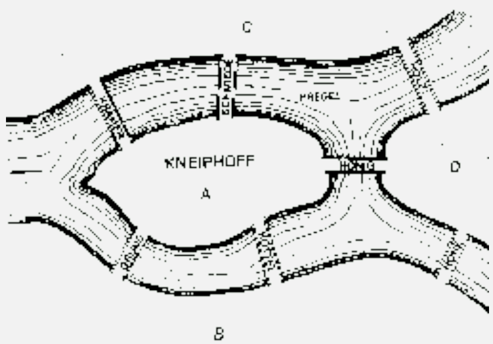
- Any programmer could do it.
- ✓ Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

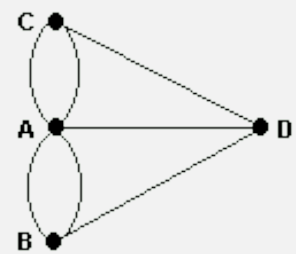
simple DFS-based solution
(see textbook)

Bridges of Königsberg

The Seven Bridges of Königsberg. [Leonhard Euler 1736]

> *" … in Königsberg in Prussia, there is an island A, called the Kneiphof; the river which surrounds it is divided into two branches ... and these branches are crossed by seven bridges. Concerning these bridges, it was asked whether anyone could arrange a route in such a way that he could cross each bridge once and only once. "*





Euler tour. Is there a (general) cycle that uses each edge exactly once?

Answer. A connected graph is Eulerian iff all vertices have even degree.

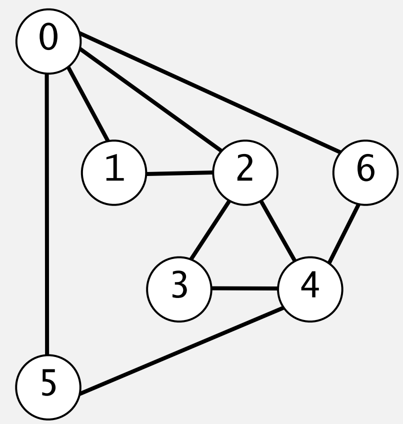
Graph-processing challenge 3

Problem. Find a (general) cycle that uses every edge exactly once.

How difficult?

- Any programmer could do it.
- ✓ Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

Eulerian tour
(classic graph-processing problem)

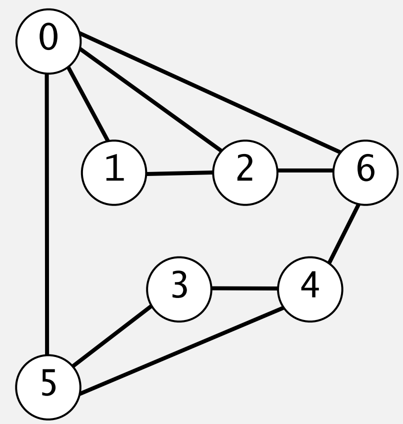


0-1
0-2
0-5
0-6
1-2
2-3
2-4
3-4
4-5
4-6

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

Graph-processing challenge 4

Problem. Find a cycle that visits 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

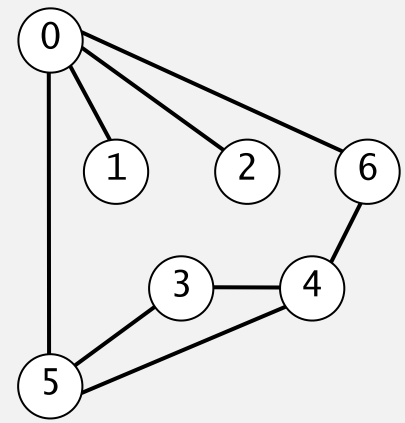
How difficult?

- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- ✓ Intractable.
- No one knows.
- Impossible.

Hamiltonian cycle
(classical NP-complete problem)

Graph-processing challenge 5

Problem. Are two graphs identical except for vertex names?

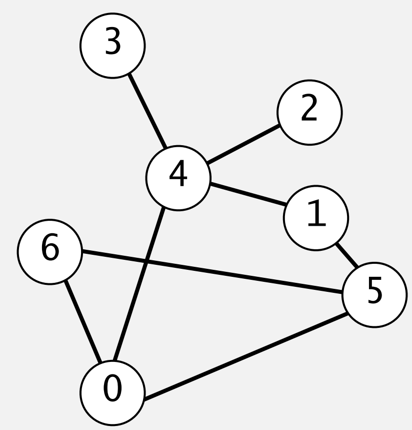


0-1
0-2
0-5
0-6
3-4
3-5
4-5
4-6

How difficult?

- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- ✓ No one knows.
- Impossible.

graph isomorphism is
longstanding open problem

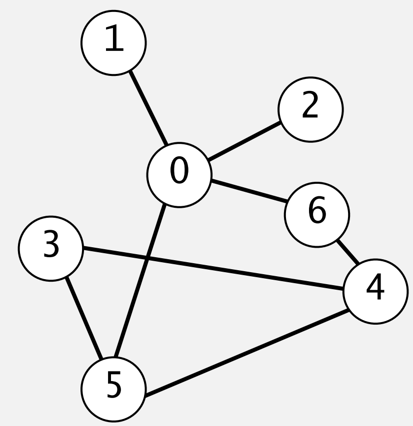


0-4
0-5
0-6
1-4
1-5
2-4
3-4
5-6

0↔4, 1↔3, 2↔2, 3↔6, 4↔5, 5↔0, 6↔1

Graph-processing challenge 6

Problem. Lay out a graph in the plane without crossing edges?

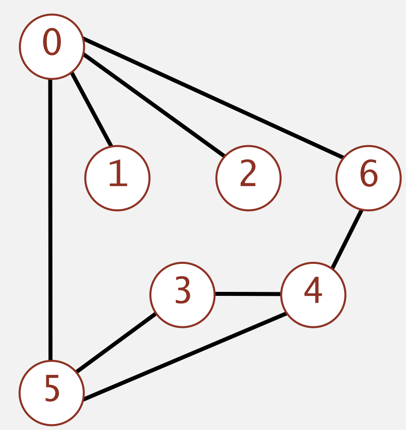


0-1
0-2
0-5
0-6
3-4
3-5
4-5
4-6

How difficult?

- Any programmer could do it.
- Typical diligent algorithms student could do it.
- ✓ Hire an expert.
- Intractable.
- No one knows.
- Impossible.

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



http://algs4.cs.princeton.edu

4.1 Undirected Graphs

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

http://algs4.cs.princeton.edu

4.1 UNDIRECTED GRAPHS

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