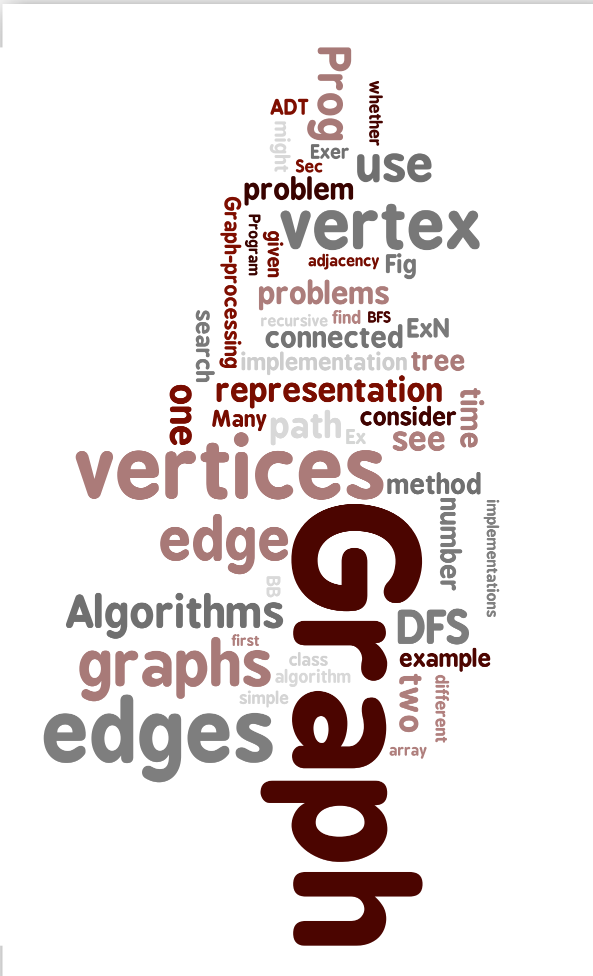
4.1 Undirected Graphs

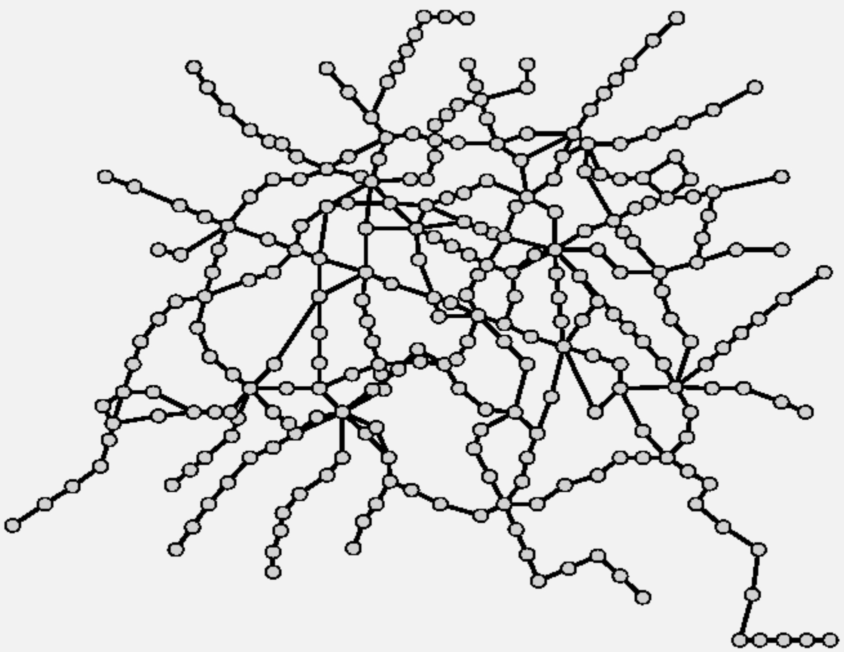


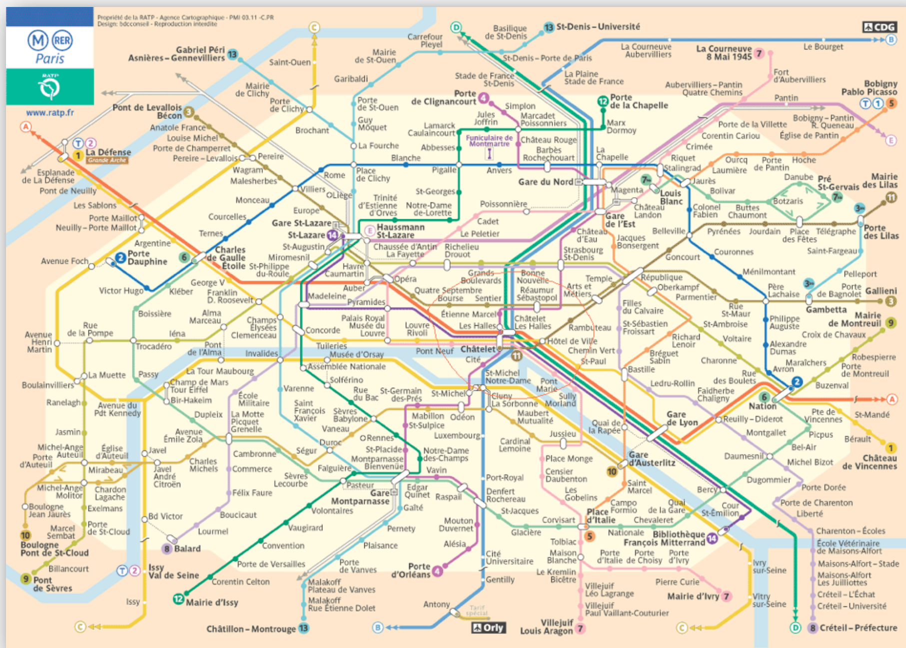
- graph API
- depth-first search
- breadth-first search
- connected components
- challenges

Graph. Set of vertices connected pairwise by edges.

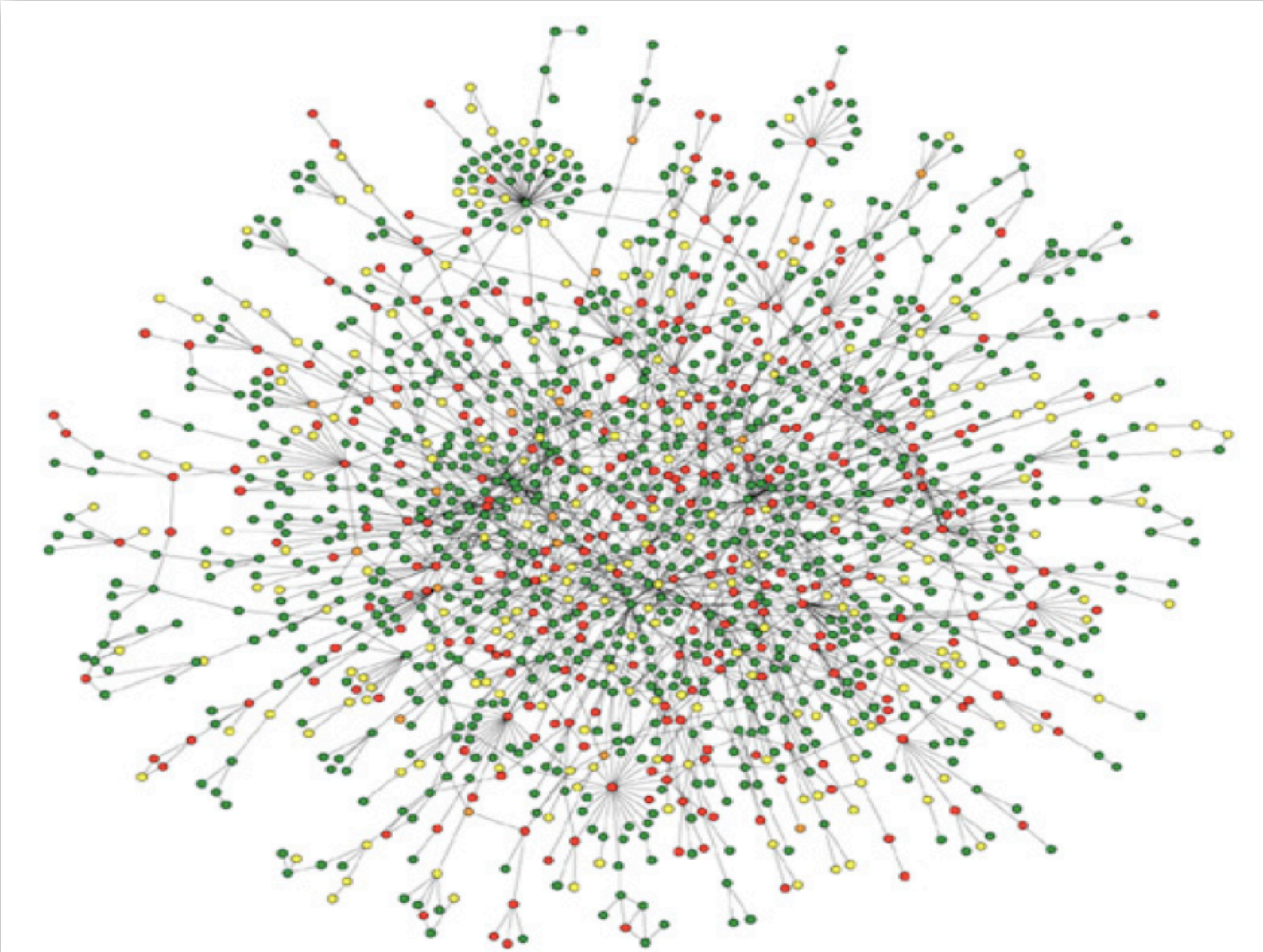
Why study graph algorithms?

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



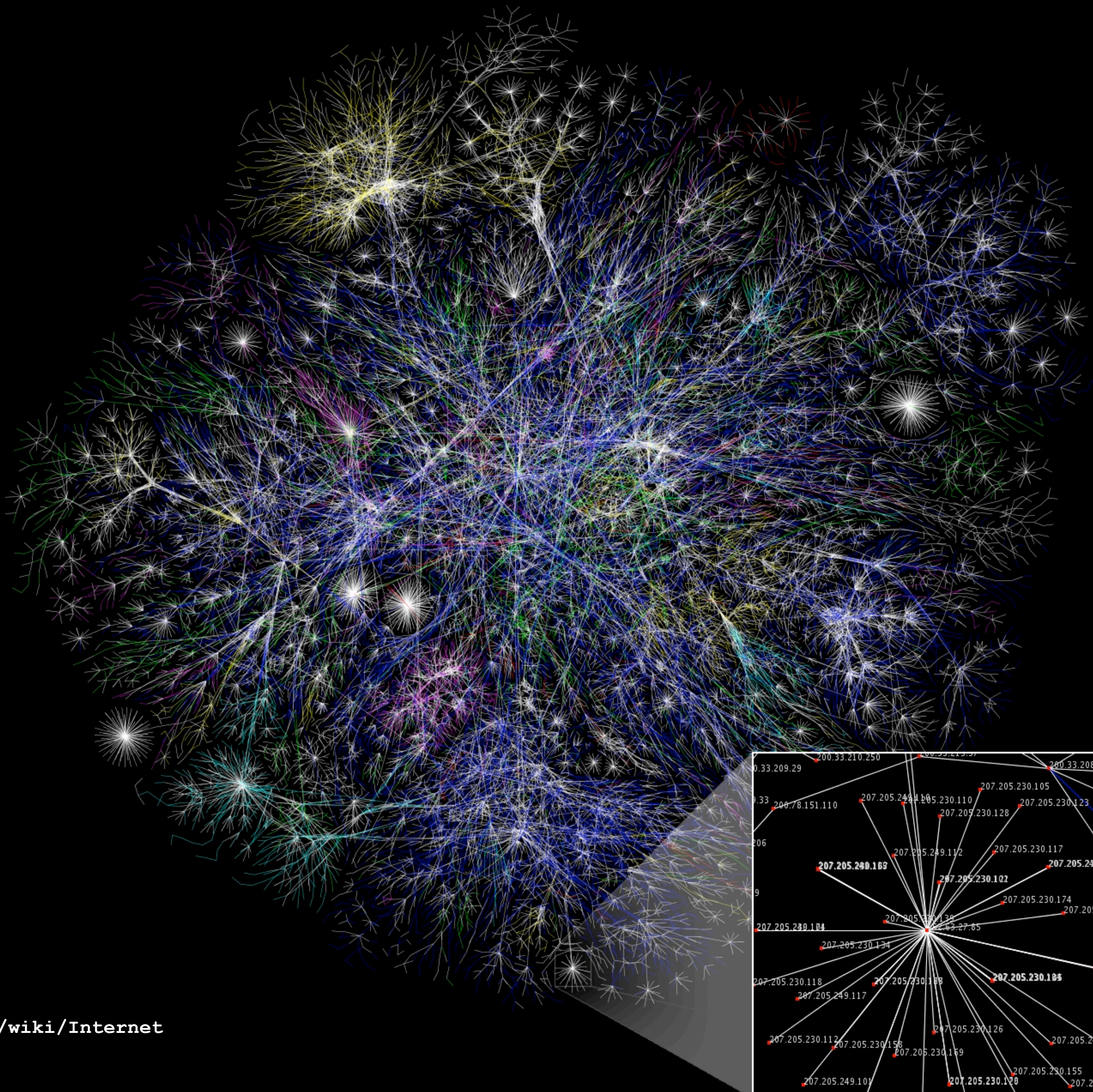


Protein-protein interaction network



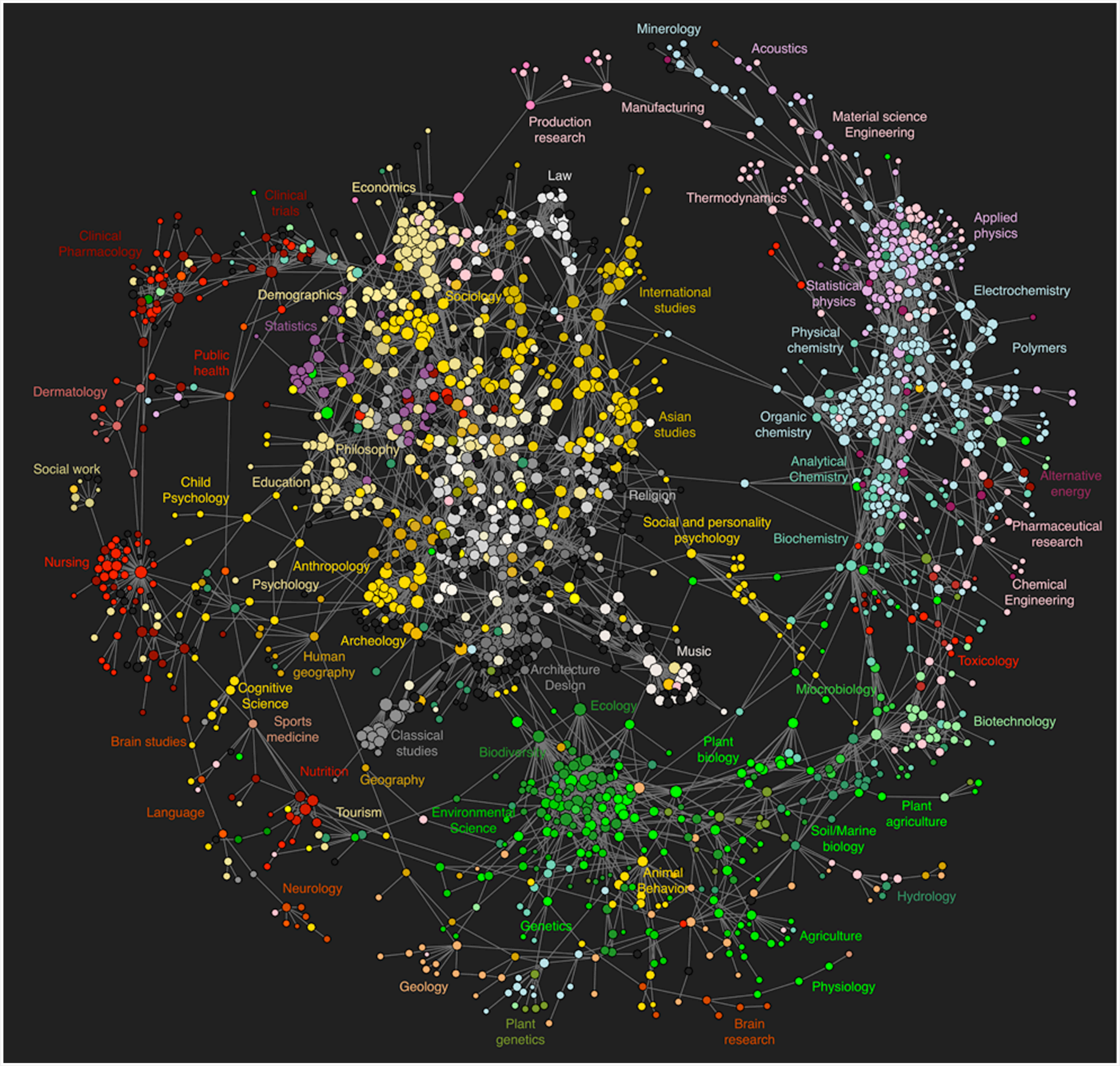
Reference: Jeong et al, Nature Review | Genetics

The Internet as mapped by the Opte Project



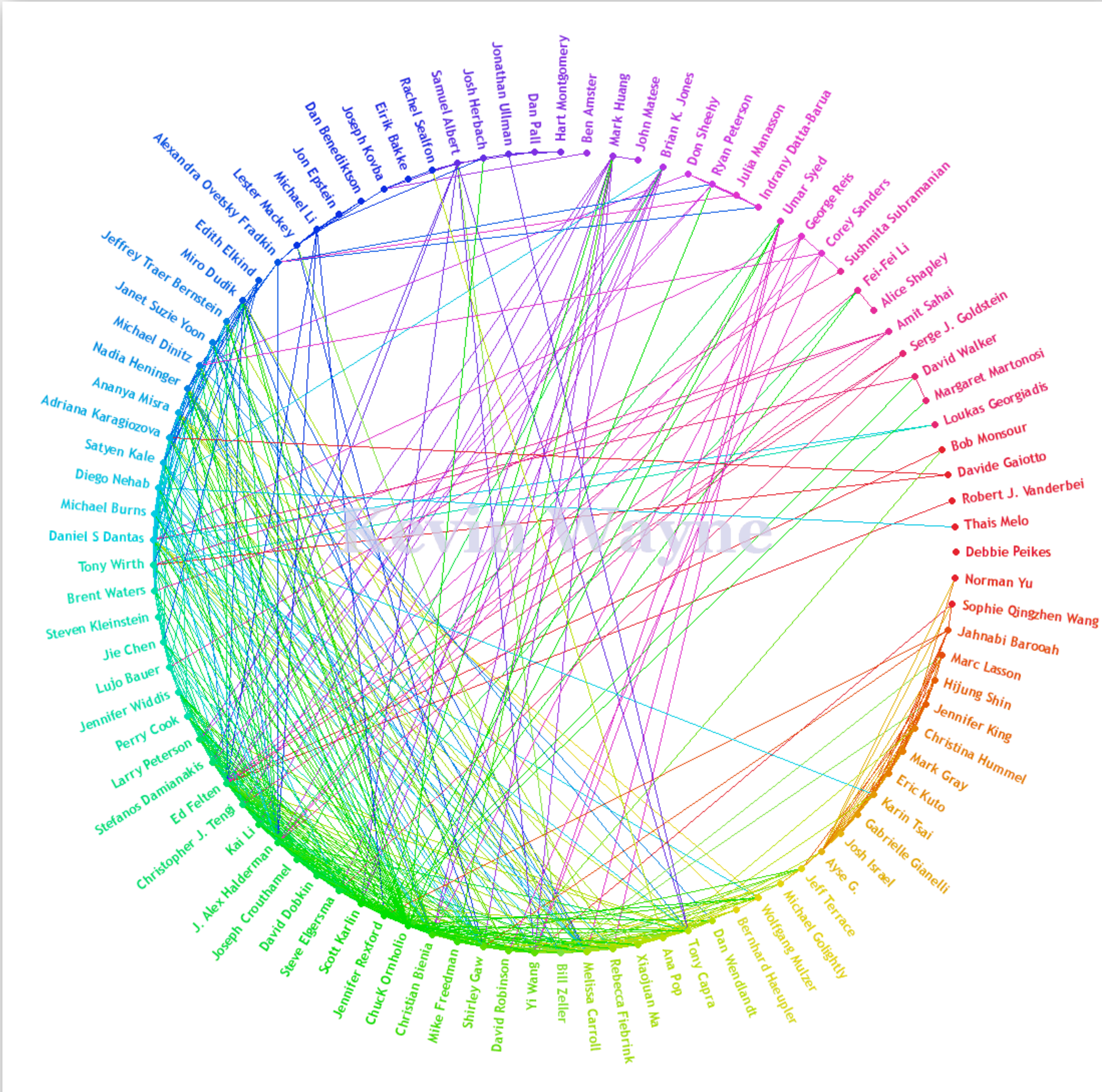
http://en.wikipedia.org/wiki/Internet

Map of science clickstreams

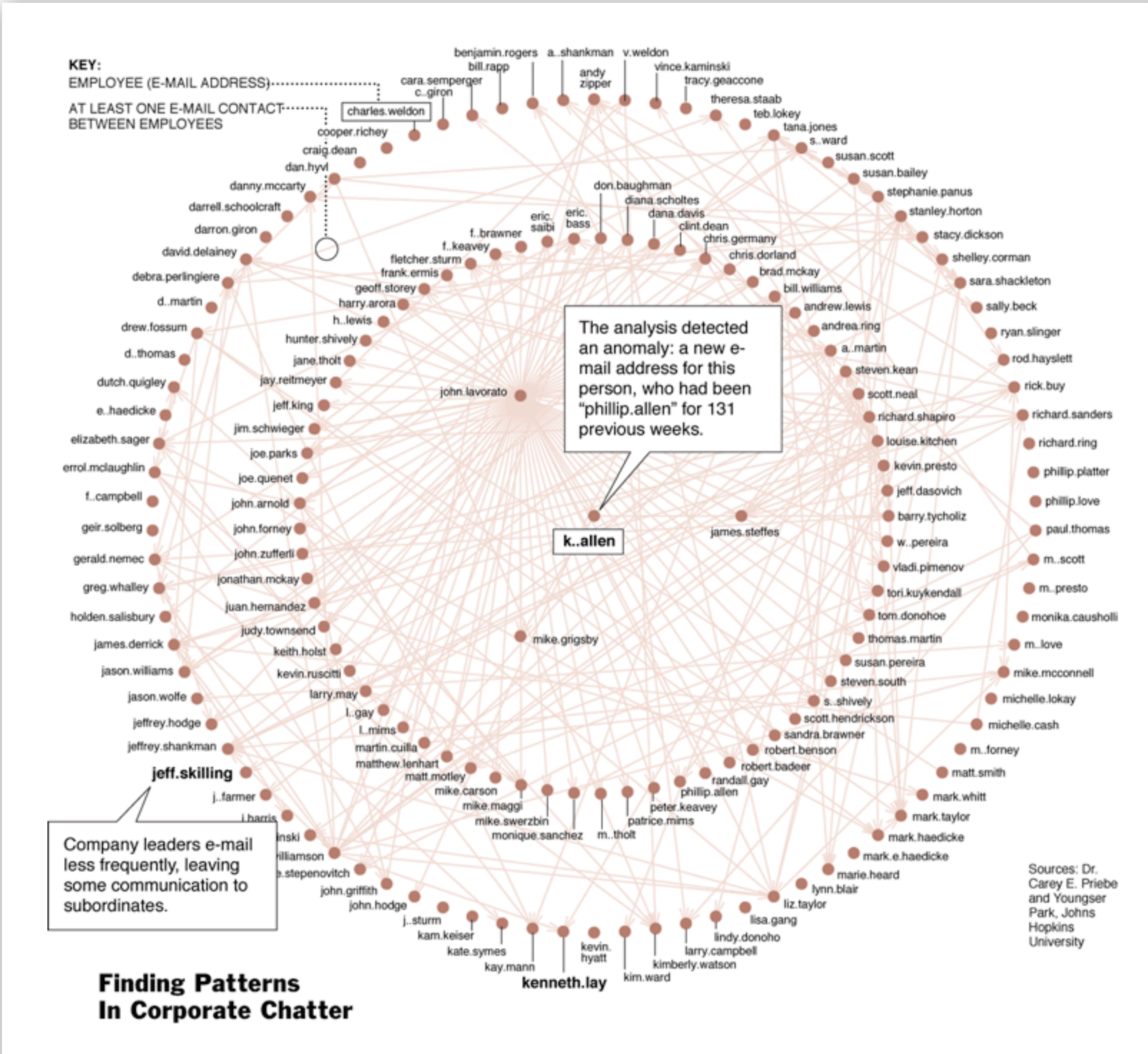


http://www.plosone.org/article/info:doi/10.1371/journal.pone.0004803

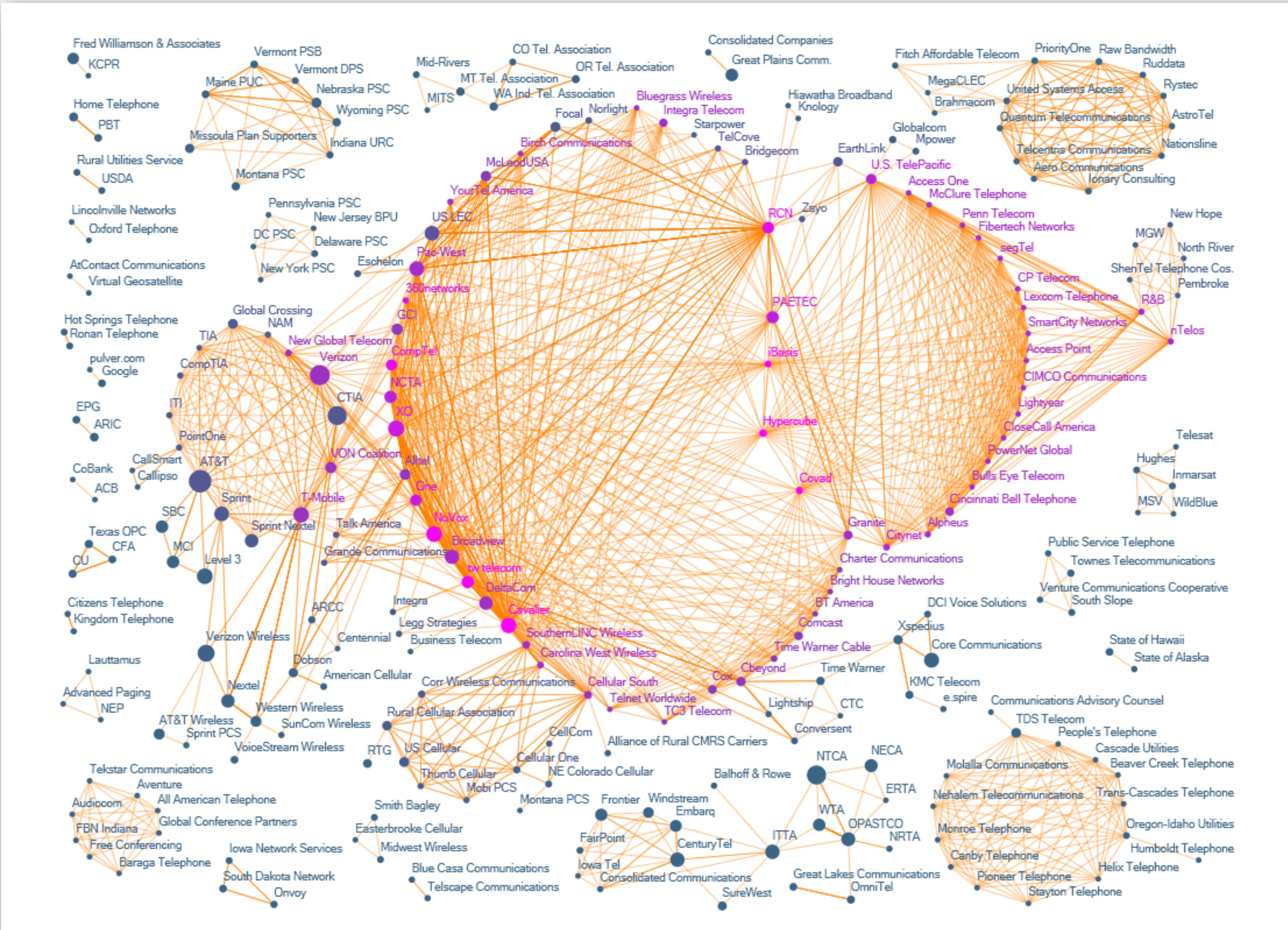
Kevin's facebook friends (Princeton network)



One week of Enron emails



The evolution of FCC lobbying coalitions



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

Graph applications

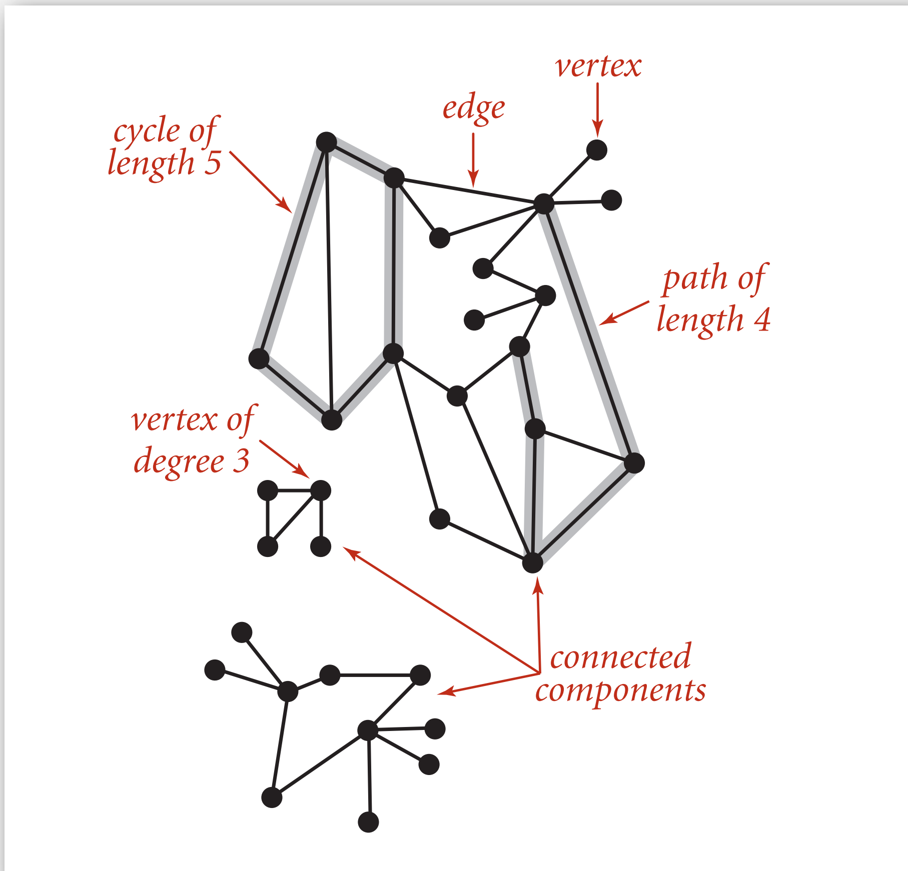
graph	vertex	edge
communication	telephone, computer	fiber optic cable
circuit	gate, register, processor	wire
mechanical	joint	rod, beam, spring
financial	stock, currency	transactions
transportation	street intersection, airport	highway, airway route
internet	class C network	connection
game	board position	legal move
social relationship	person, actor	friendship, movie cast
neural network	neuron	synapse
protein network	protein	protein-protein interaction
chemical compound	molecule	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

Path. Is there a path between s and t?

Shortest path. What is the shortest path between s and t?

Cycle. Is there a cycle in the graph?

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

Hamilton tour. Is there a cycle that uses each vertex exactly once?

Connectivity. Is there a way to connect all of the vertices?

MST. What is the best way to connect all of the vertices?

Biconnectivity. Is there a vertex whose removal disconnects the graph?

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

Graph isomorphism. Do two adjacency lists represent the same graph?

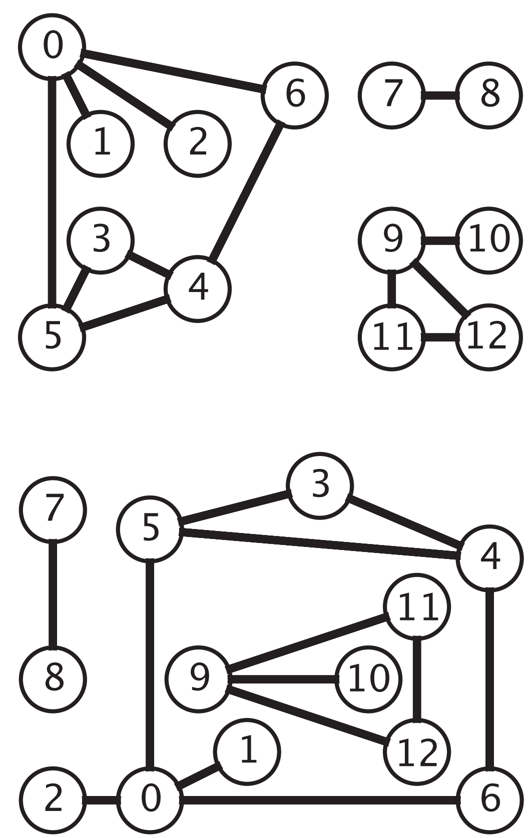
Challenge. Which of these problems are easy? difficult? intractable?

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

Graph representation

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

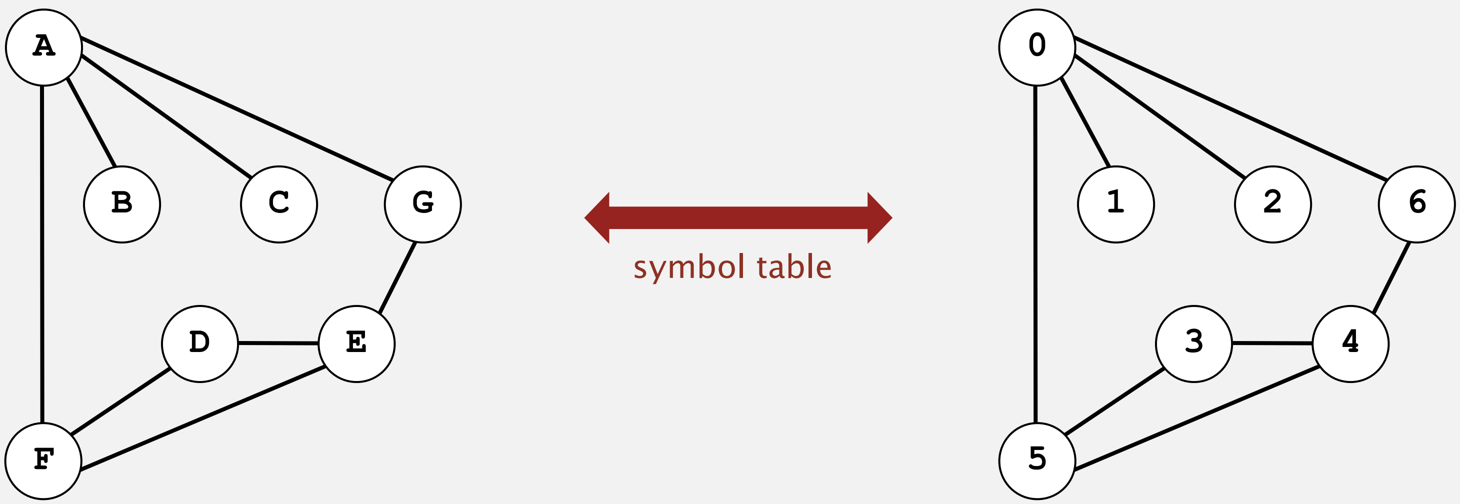
Caveat. Intuition can be misleading.



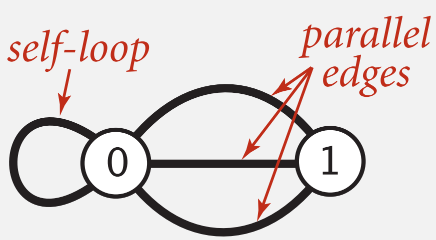
Two drawings of the same graph

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*
`String`	`toString()`	*string representation*

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

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

← read graph from input stream

← print out each edge (twice)

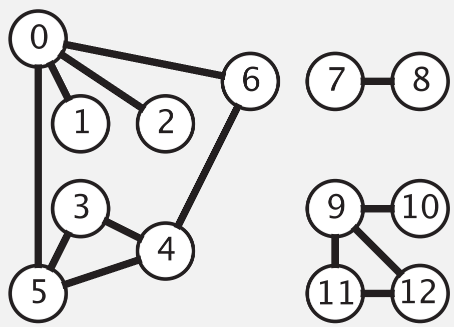
Graph API: sample client

Graph input format.

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
```




```
% 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(w))
      StdOut.println(v + "-" + w);
```

read graph from input stream

print out each edge (twice)

Typical graph-processing code

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

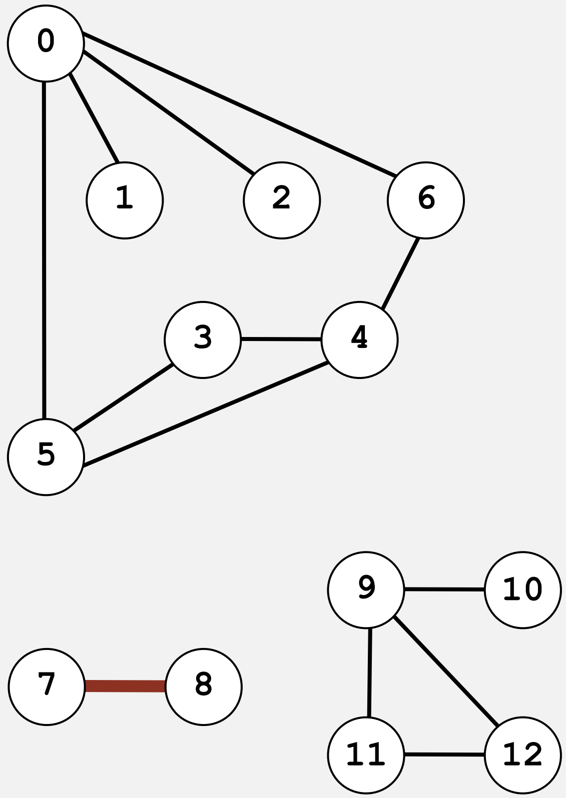
compute maximum degree	`public static int maxDegree(Graph G)`
	`{`
	`   int max = 0;`
	`   for (int v = 0; v < G.V(); v++)`
	`      if (degree(G, v) > max)`
	`         max = degree(G, v);`
	`   return max;`
	`}`

compute average degree	`public static int avgDegree(Graph G)`
	`{`
	`   return 2 * G.E() / G.V();`
	`}`

count self-loops	`public static int numberOfSelfLoops(Graph G)`
	`{`
	`   int count = 0;`
	`   for (int v = 0; v < G.V(); v++)`
	`      for (int w : G.adj(v))`
	`         if (v == w) count++;`
	`   return count/2;`
	`}`

Set-of-edges graph representation

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

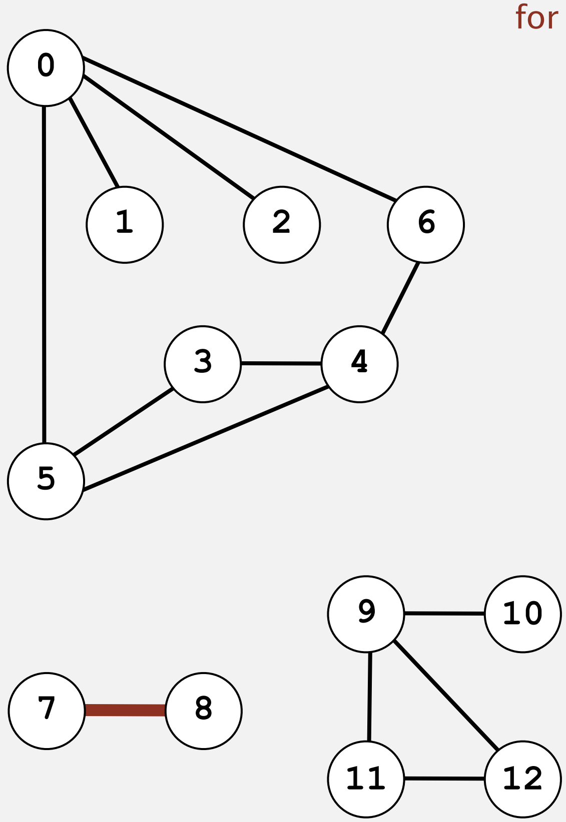


0	1
0	2
0	5
0	6
3	4
3	5
4	5
4	6
7	8
9	10
9	11
9	12
11	12

Adjacency-matrix graph representation

Maintain a two-dimensional V-by-V boolean array;

for each edge v-w in graph: `adj[v][w] = adj[w][v] = true`.



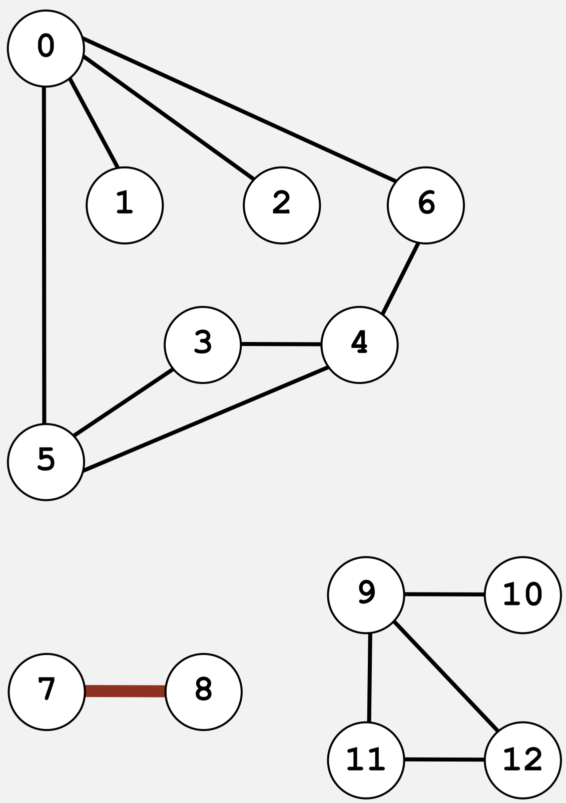
two entries for each edge

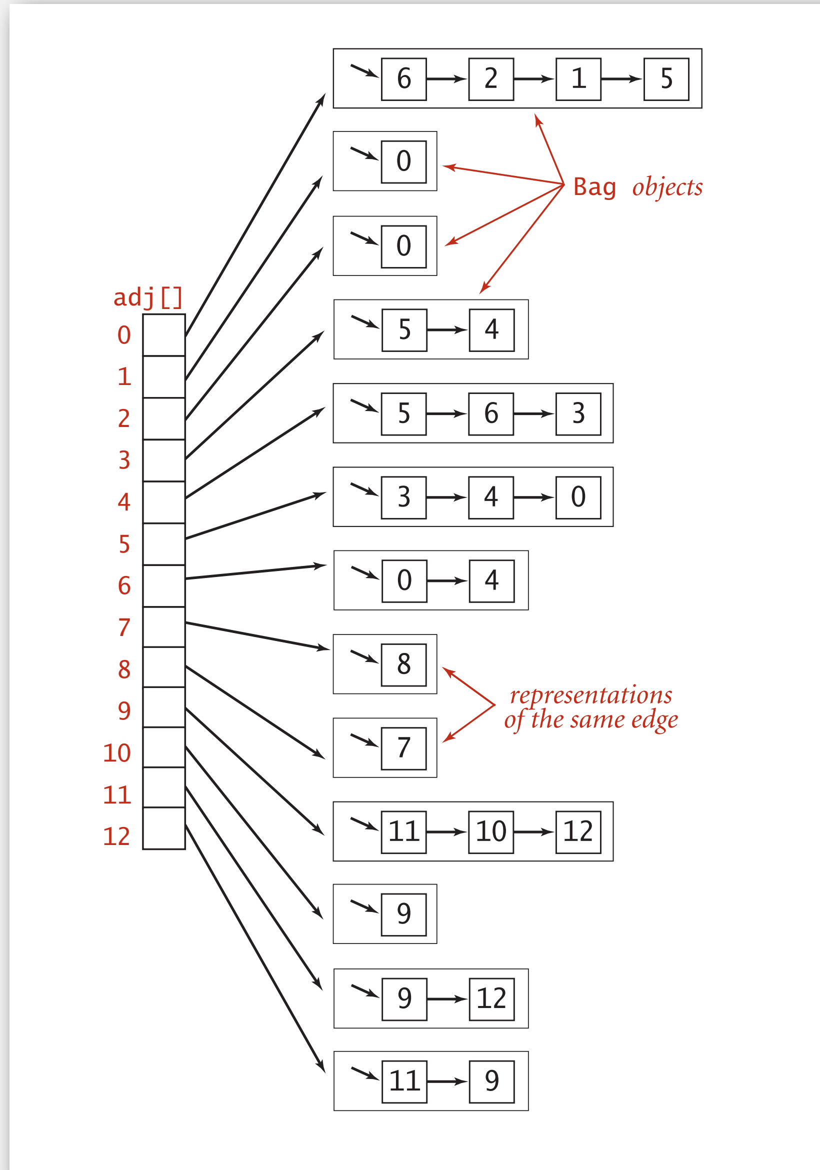
	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0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

Adjacency-list graph representation

Maintain vertex-indexed array of lists.

(use `Bag` abstraction)





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
(use **Bag** data type)

create empty graph
with **V** vertices

add edge **v-w**
(parallel edges allowed)

iterator for vertices adjacent to **v**

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

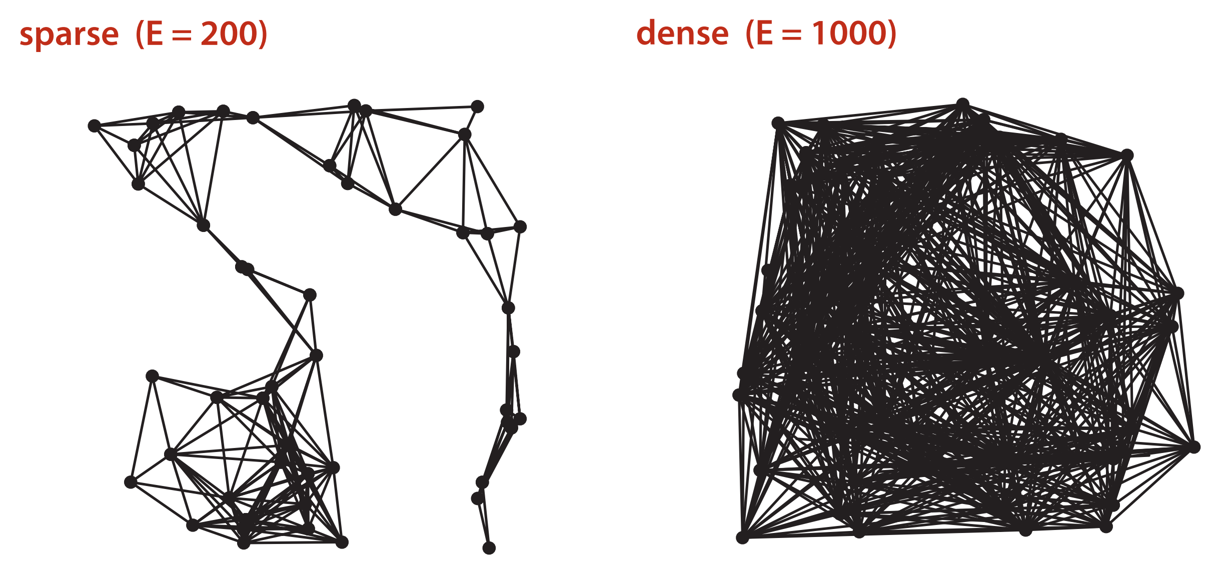
representation	space	add edge	edge between v and w?	iterate over vertices adjacent to v?
list of edges	E	1	E	E
adjacency matrix	V^2	1 *	1	V
adjacency lists	E + V	1	degree(v)	degree(v)

* disallows parallel edges

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



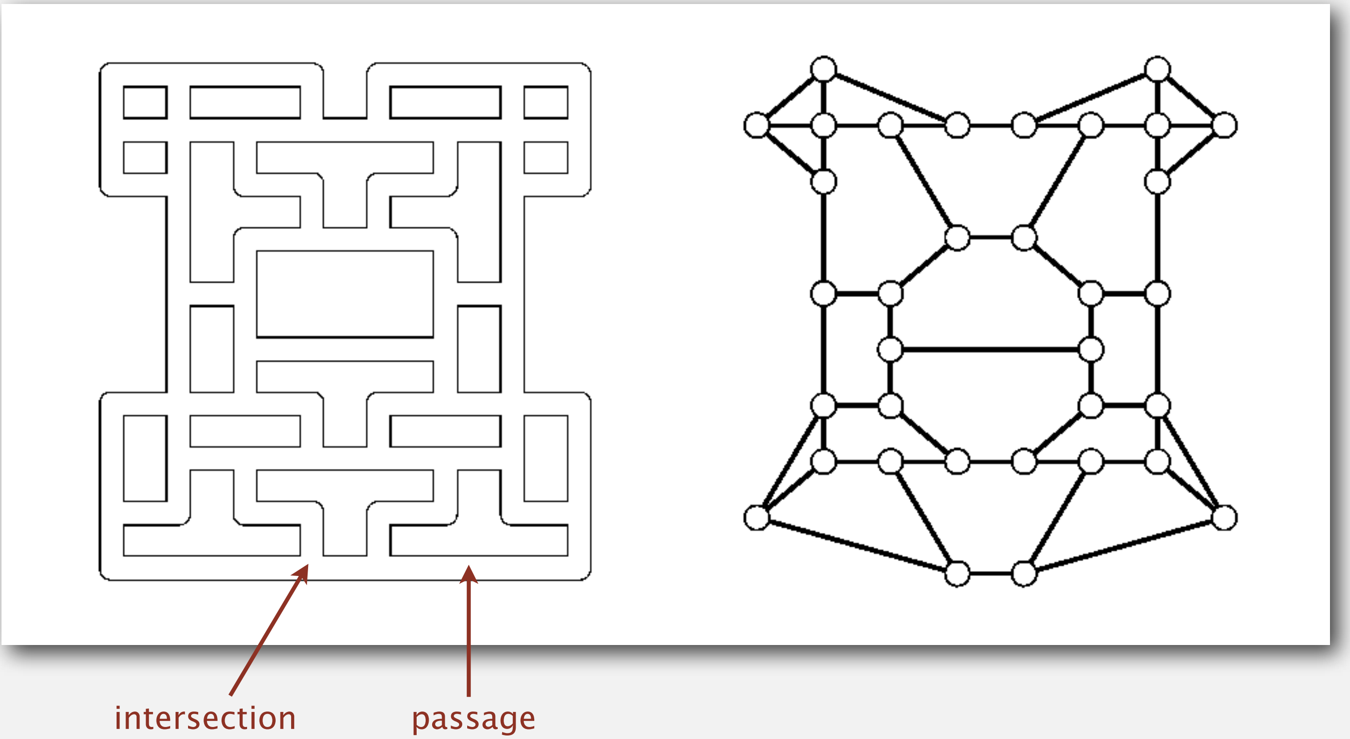
Two graphs (V = 50)

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

Maze exploration

Maze graphs.

- Vertex = intersection.
- Edge = passage.

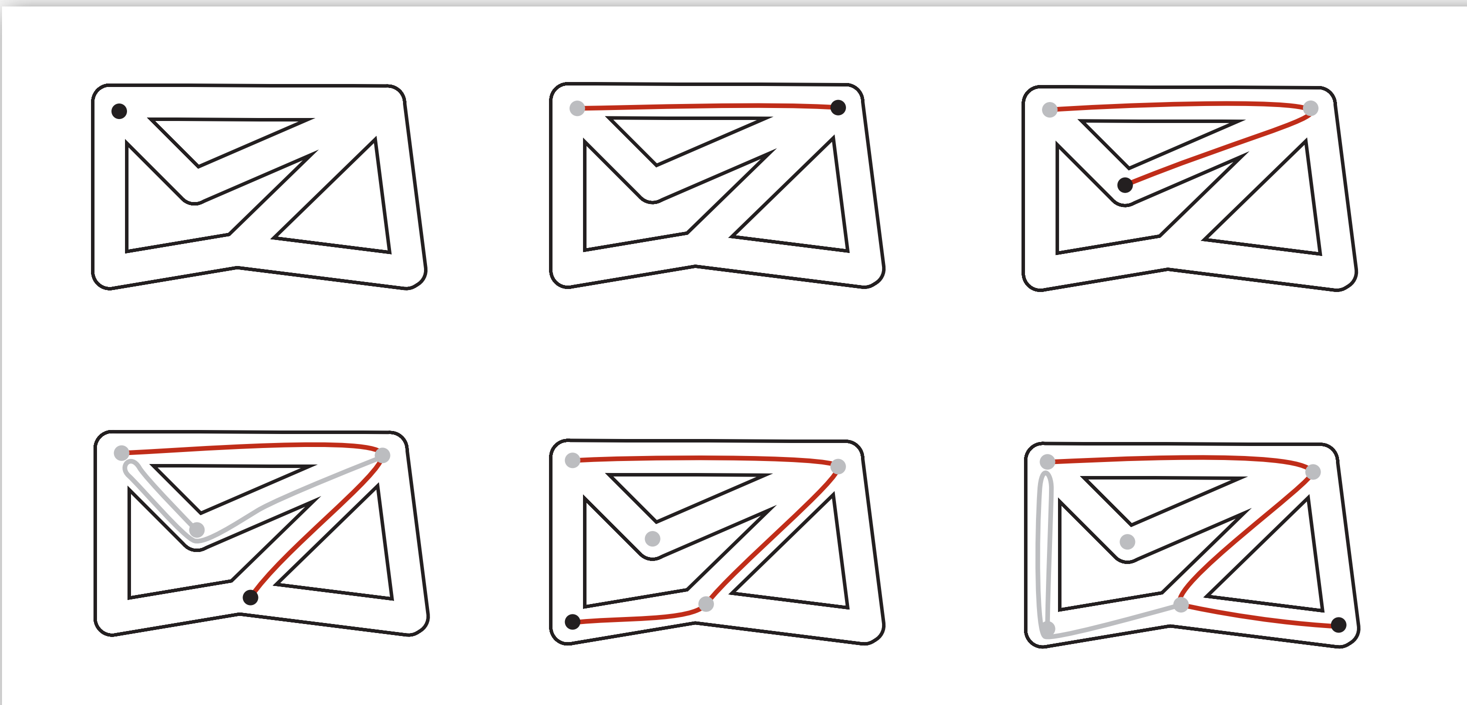


Goal. Explore every intersection in the maze.

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.

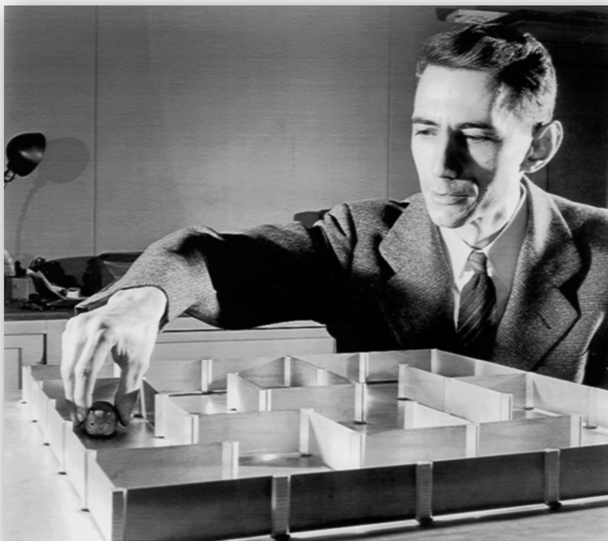


Trémaux maze exploration

Algorithm.

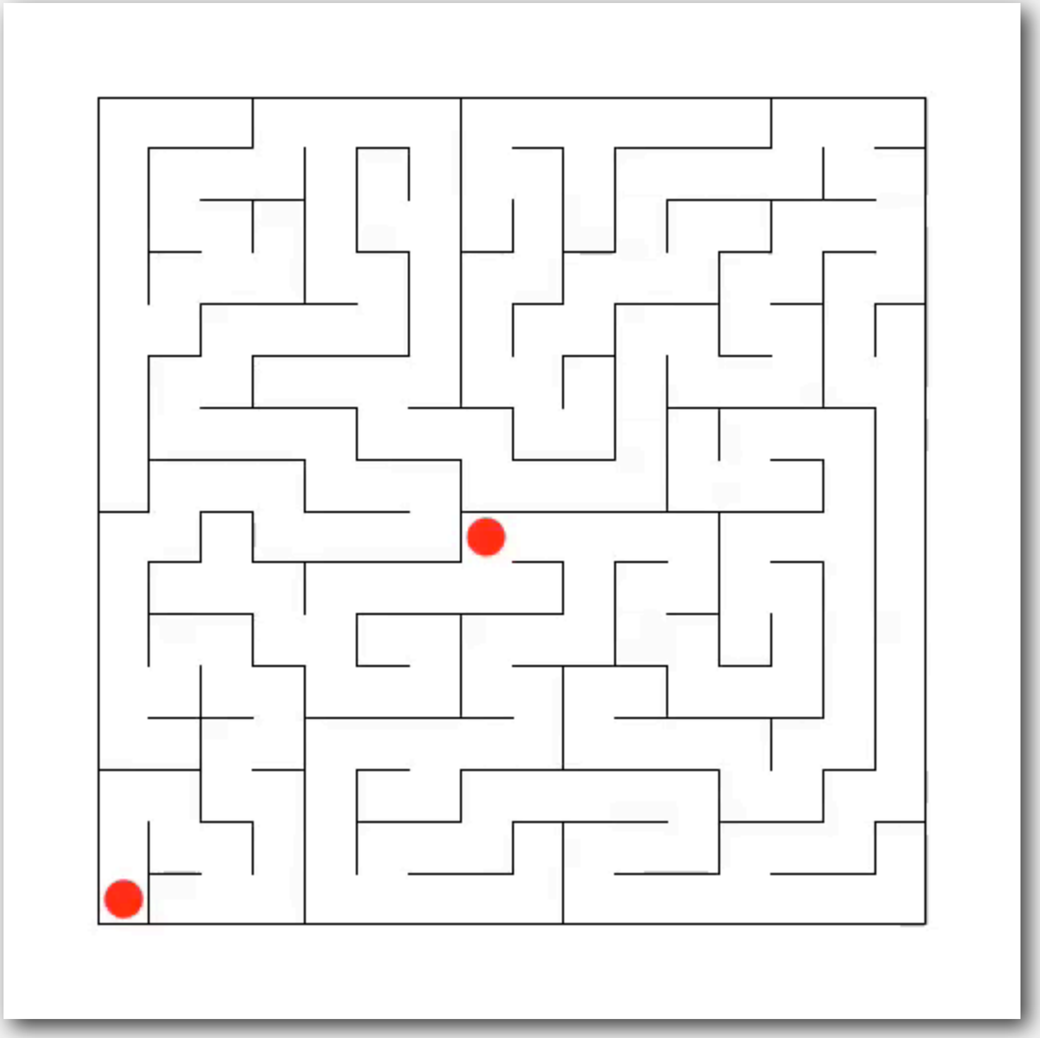
- Unroll a ball of string behind you.
- Mark each visited intersection and each visited passage.
- Retrace steps when no unvisited options.

First use? Theseus entered labyrinth to kill the monstrous Minotaur; Ariadne held ball of string.

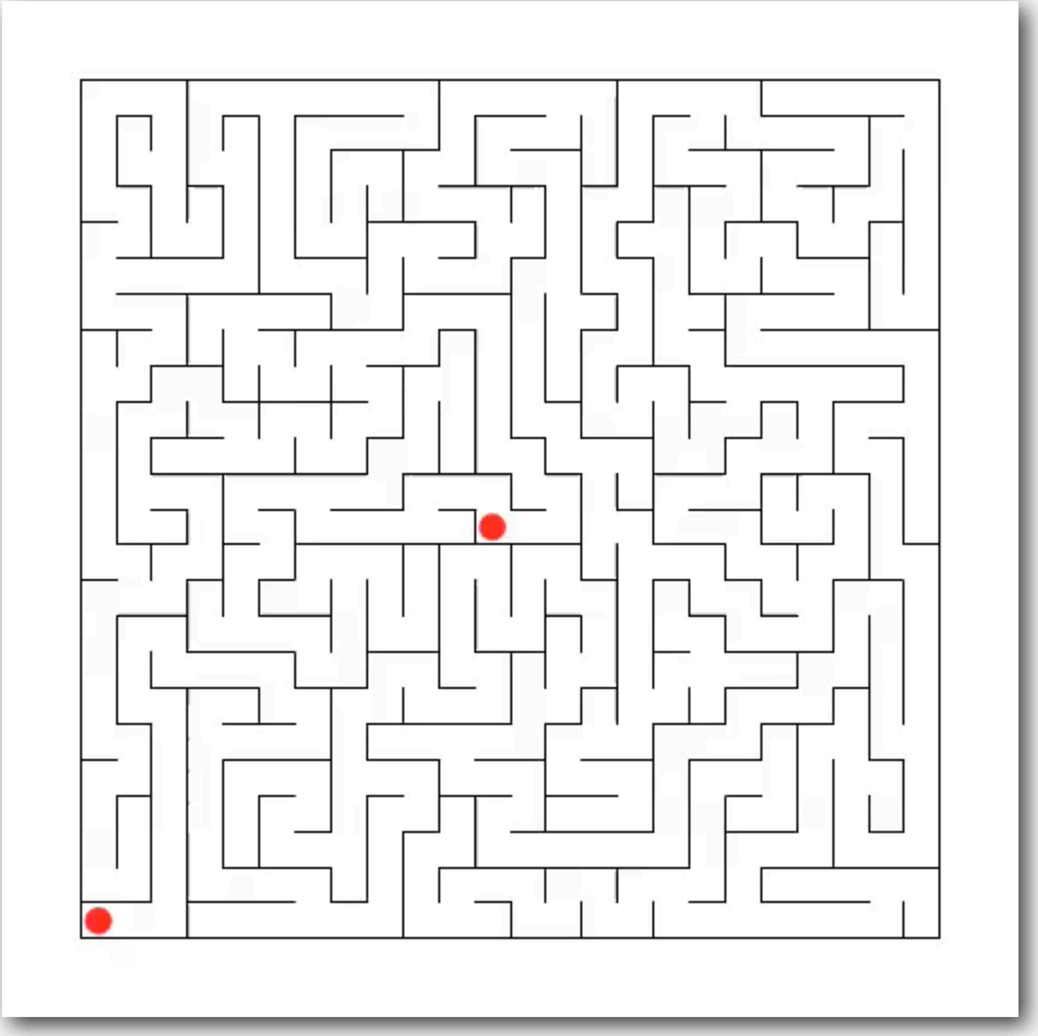


Claude Shannon (with Theseus mouse)

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. [ahead]

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

Design pattern for graph processing

Design pattern. Decouple graph data type from graph processing.

public class Search		
	`Search(Graph G, int s)`	*find vertices connected to s*
`boolean`	`marked(int v)`	*is vertex v connected to s?*
`int`	`count()`	*how many vertices connected to s?*

Typical client program.

- Create a `Graph`.
- Pass the `Graph` to a graph-processing routine, e.g., `Search`.
- Query the graph-processing routine for information.

```
Search search = new Search(G, s);
for (int v = 0; v < G.V(); v++)
   if (search.marked(v))
      StdOut.println(v);
```

print all vertices connected to s

Depth-first search (warmup)

Goal. Find all vertices connected to s.

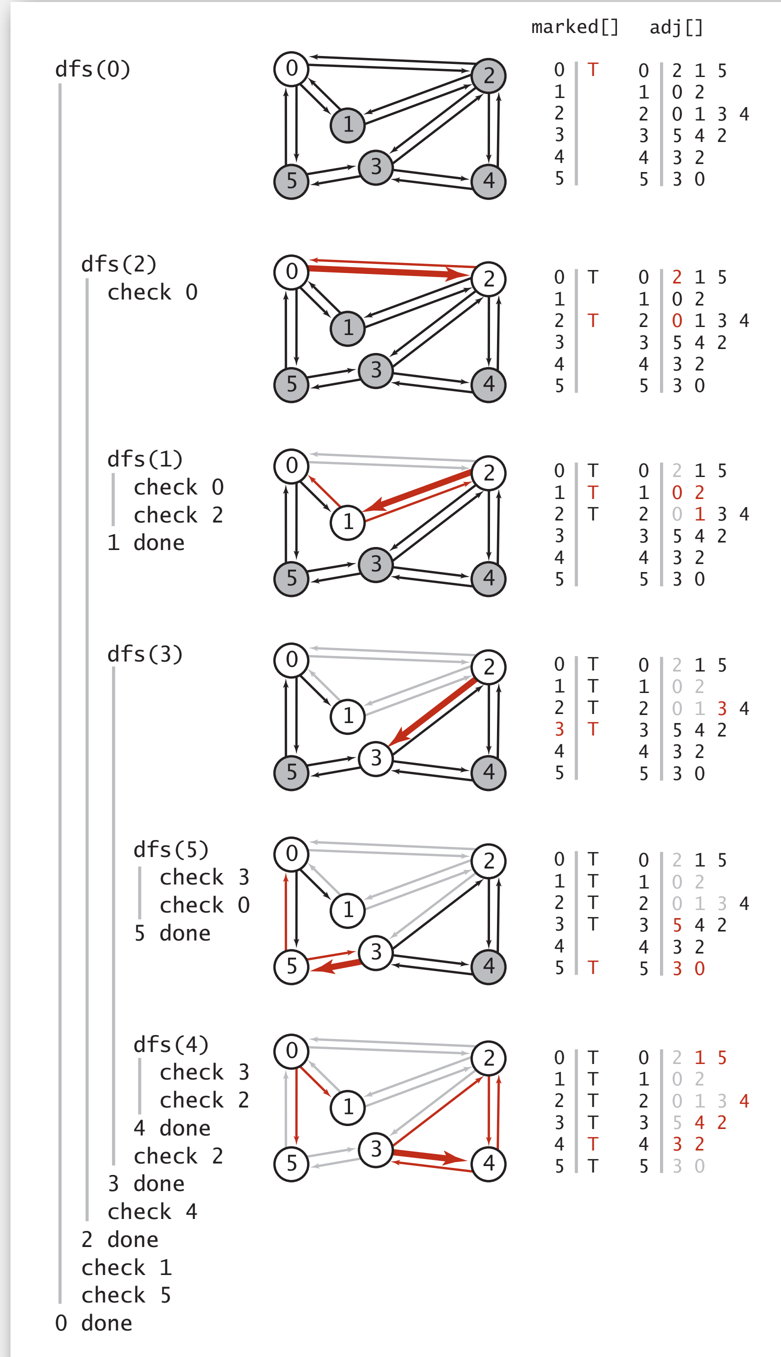
Idea. Mimic maze exploration.

Algorithm.

- Use recursion (ball of string).
- Mark each visited vertex.
- Return (retrace steps) when no unvisited options.

Data structure.

- `boolean[] marked` to mark visited vertices.



Depth-first search (warmup)

```
public class DepthFirstSearch
{
   private boolean[] marked;

   public DepthFirstSearch(Graph G, int s)
   {
      marked = new boolean[G.V()];
      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);
   }

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

true if connected to s

constructor marks vertices connected to s

recursive DFS does the work

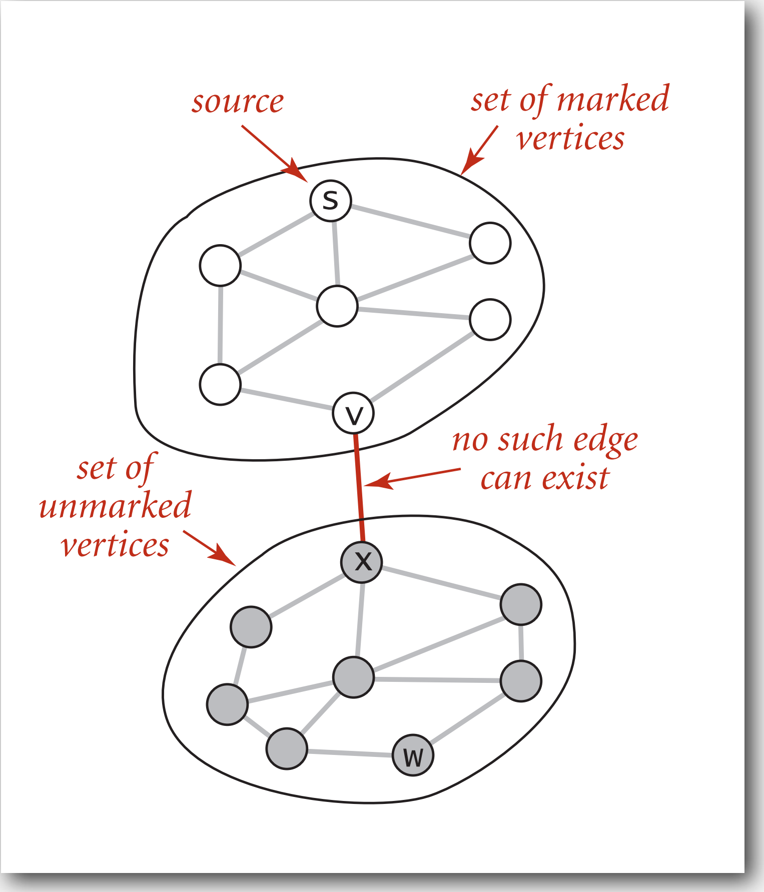
client can ask whether vertex v is connected to s

Depth-first search properties

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

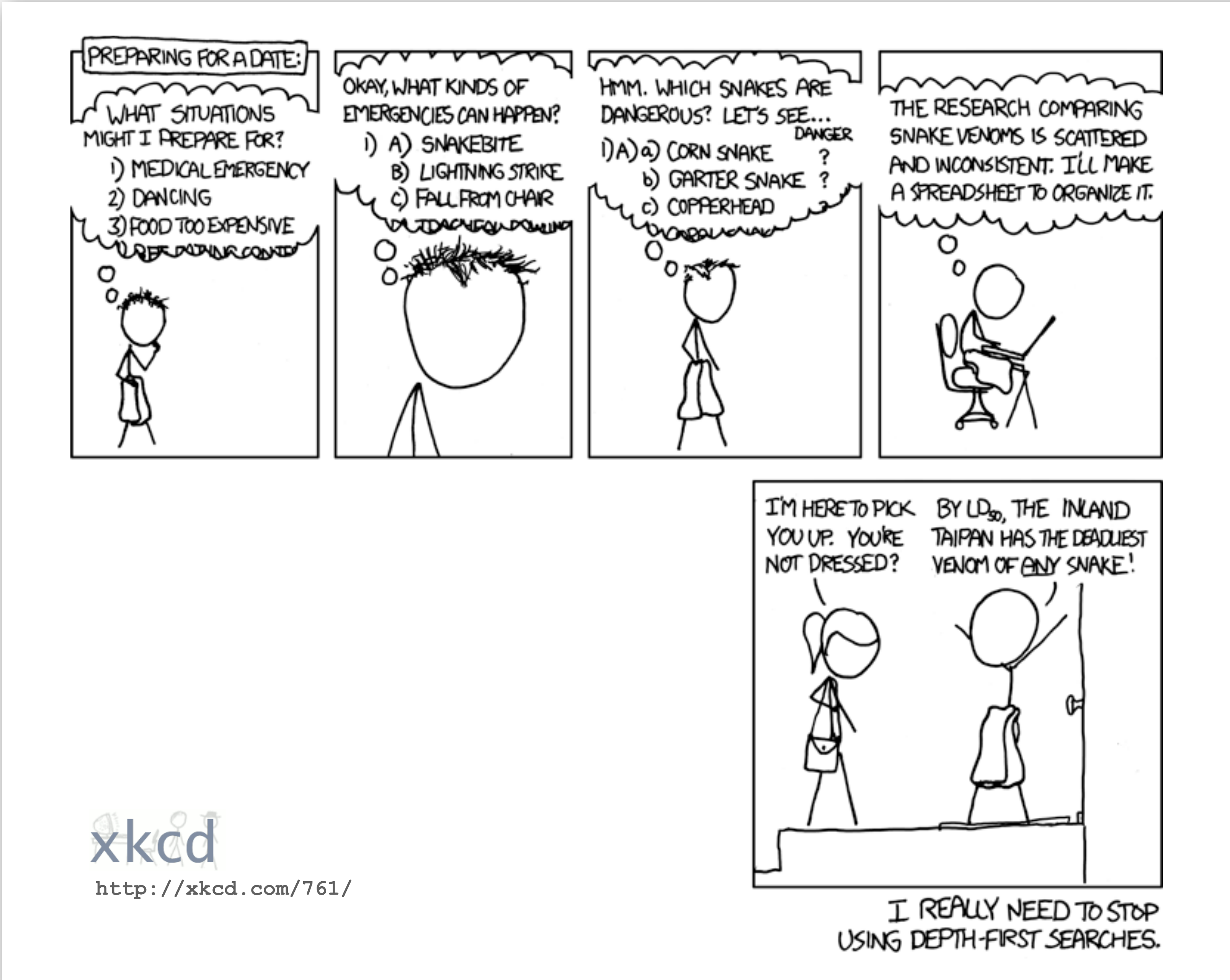
Pf.

- 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)



- Running time: each vertex connected to s is visited once.

Depth-first search application: preparing for a date



Depth-first search application: flood fill

Challenge. Flood fill (Photoshop magic wand).

Assumptions. Picture has millions to billions of pixels.

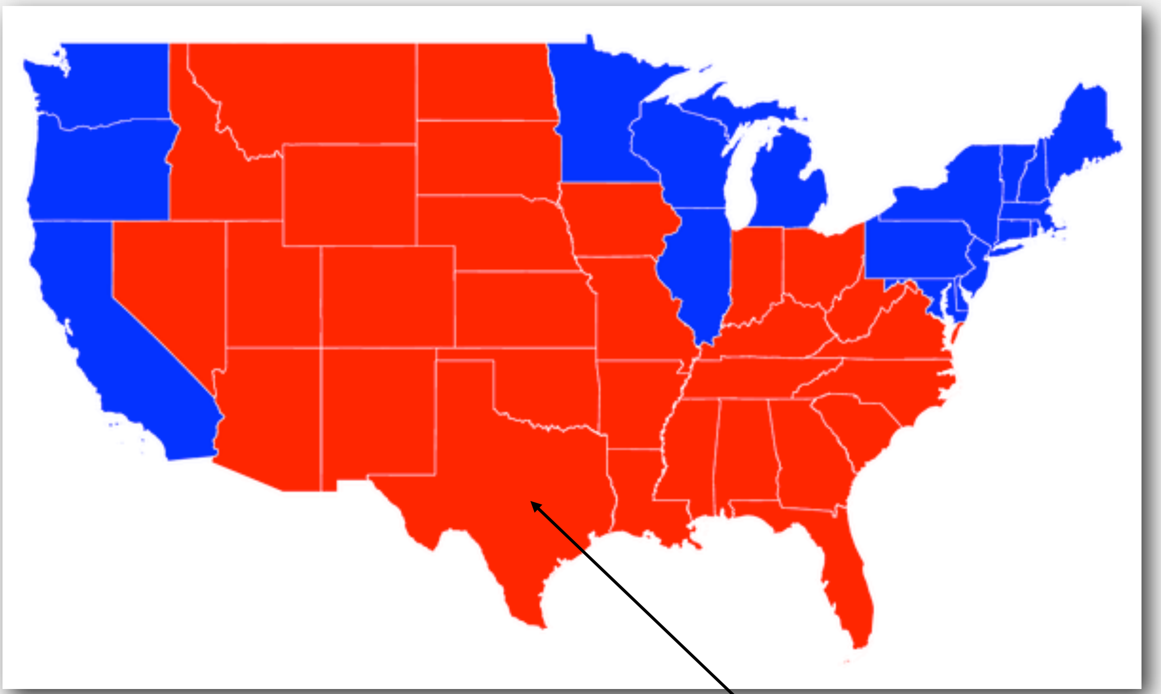
Q. How difficult?

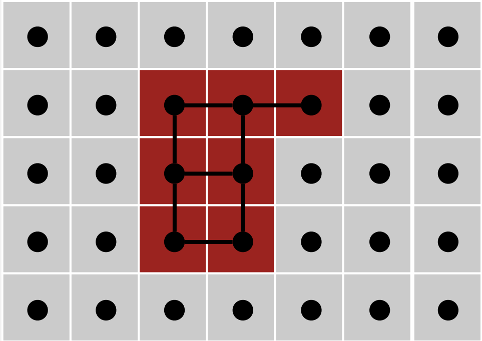
Depth-first search application: flood fill

Change color of entire blob of neighboring red pixels to blue.

Build a grid graph.

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



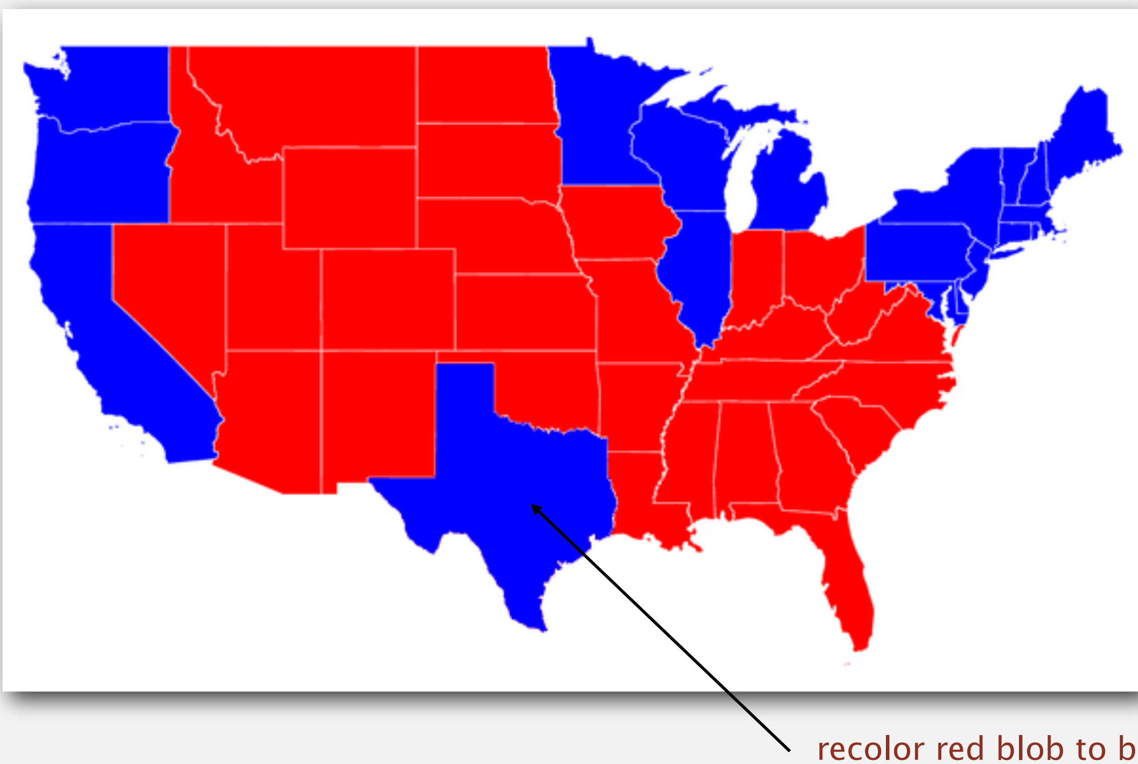


Depth-first search application: flood fill

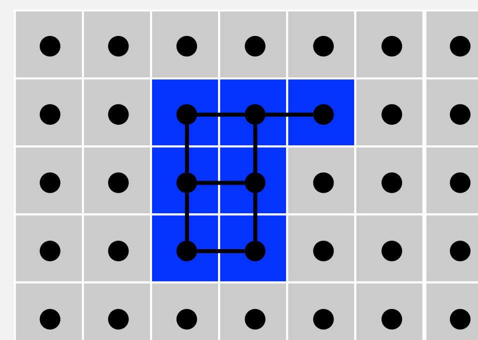
Change color of entire blob of neighboring red pixels to blue.

Build a grid graph.

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

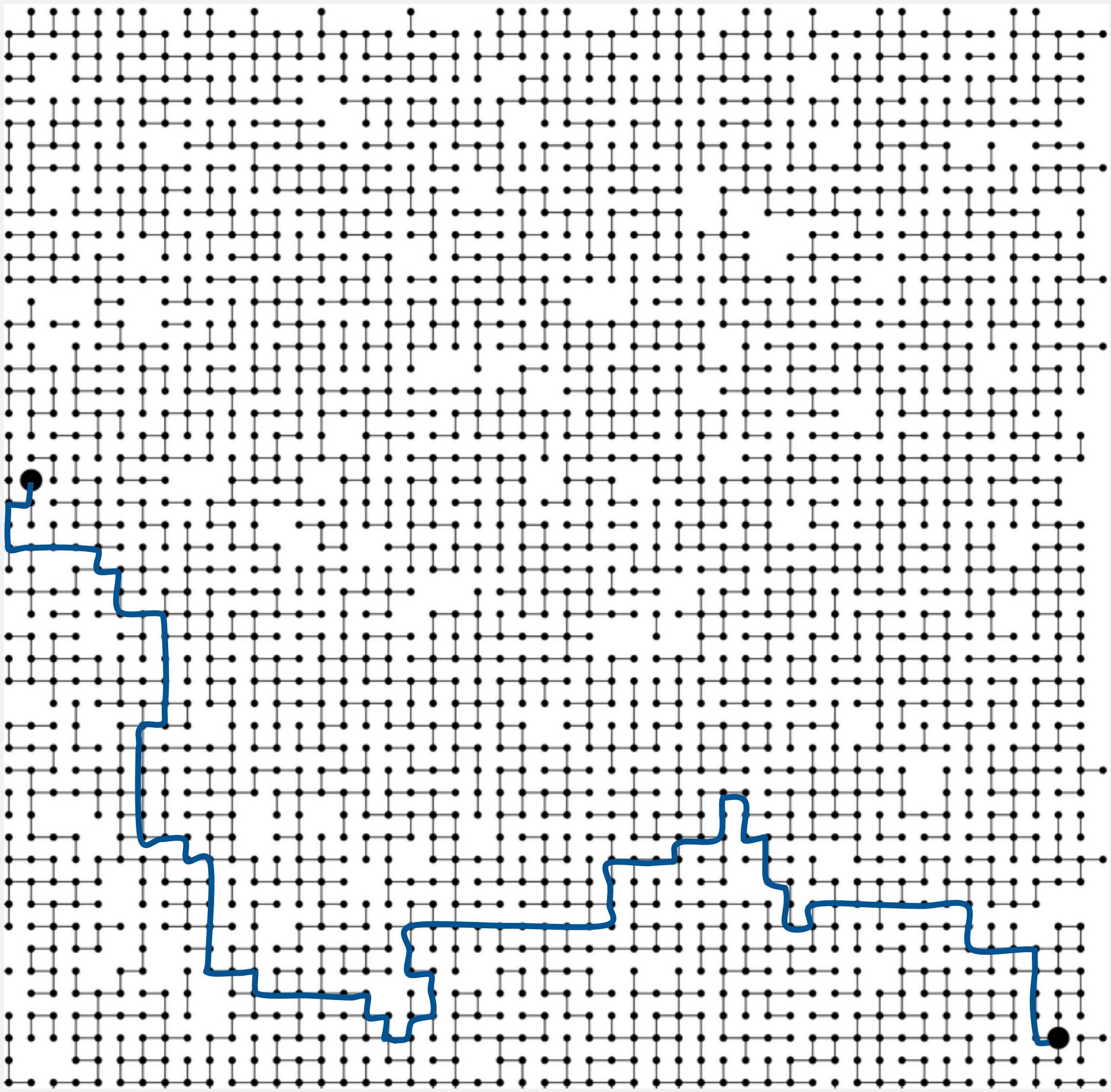


recolor red blob to blue



Paths in graphs

Goal. Does there exist a path from s to t ?



Paths in graphs: union-find vs. DFS

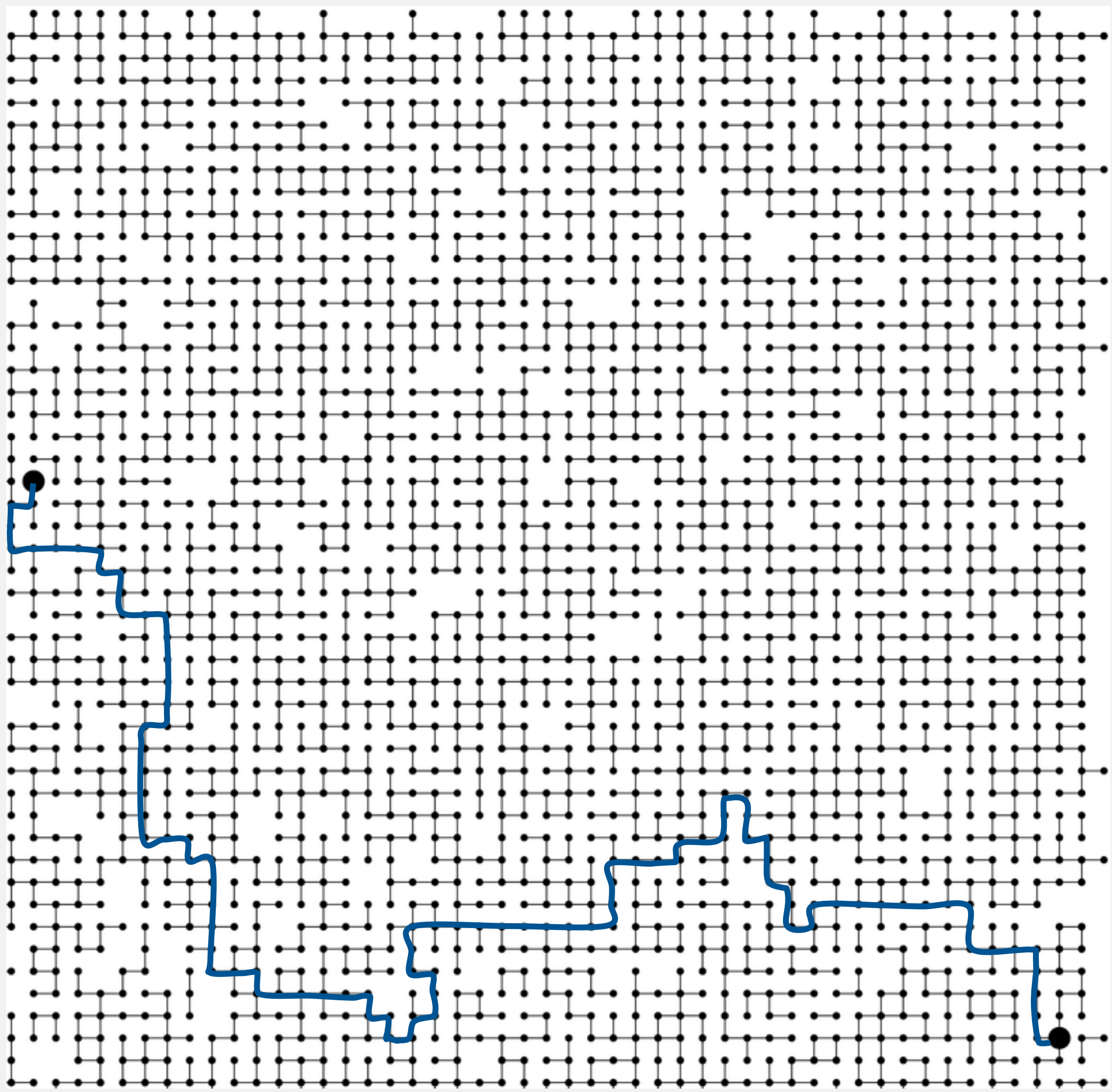
Goal. Does there exist a path from s to t ?

method	preprocessing time	query time	space
union-find	V + E log* V	log* V †	V
DFS	E + V	1	E + V

Union-find. Can intermix queries and edge insertions.

Depth-first search. Constant time per query.

Goal. Does there exist a path from s to t? If yes, find any such path.



Goal. Does there exist a path from s to t? If yes, find any such path.

```
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*

Union-find. Not much help.

Depth-first search. After linear-time preprocessing, can recover path itself in time proportional to its length.

easy modification (stay tuned)

Depth-first search (pathfinding)

Goal. Find paths to all vertices connected to a given source s.

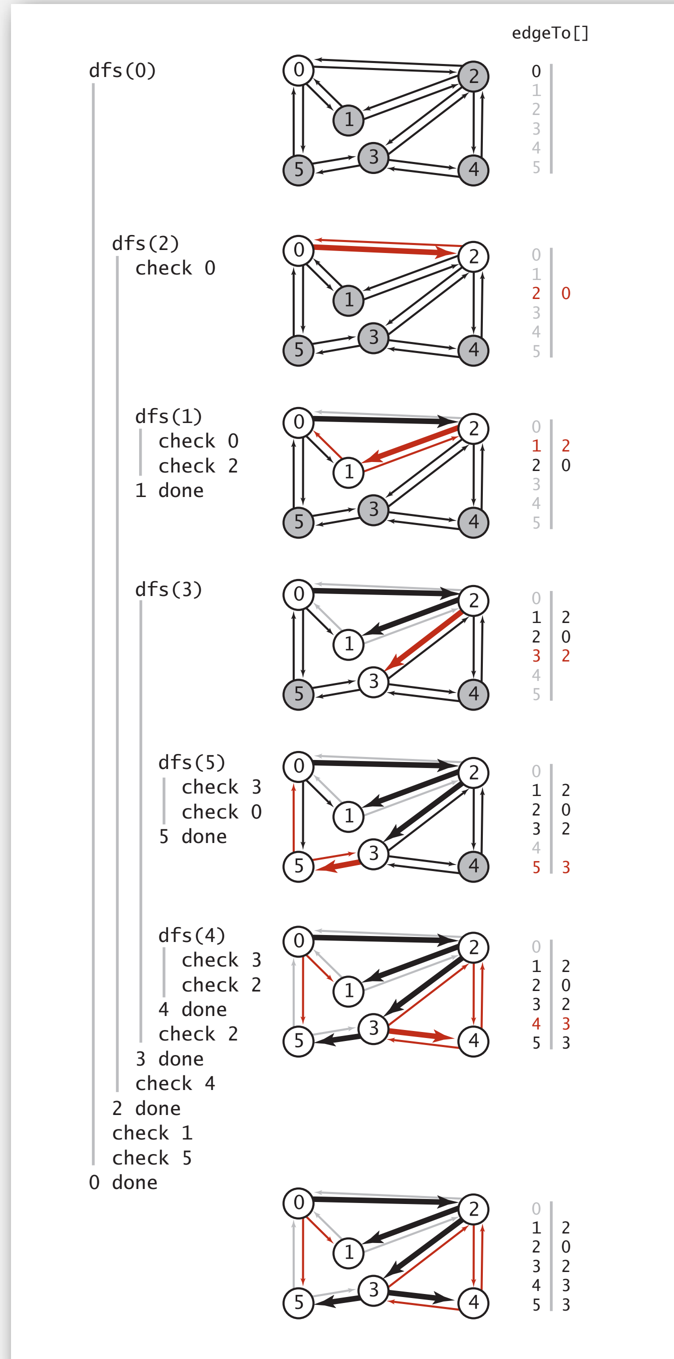
Idea. Mimic maze exploration.

Algorithm.

- Use recursion (ball of string).
- Mark each visited vertex by keeping
- 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` was taken to visit `w` the first time



Depth-first search (pathfinding)

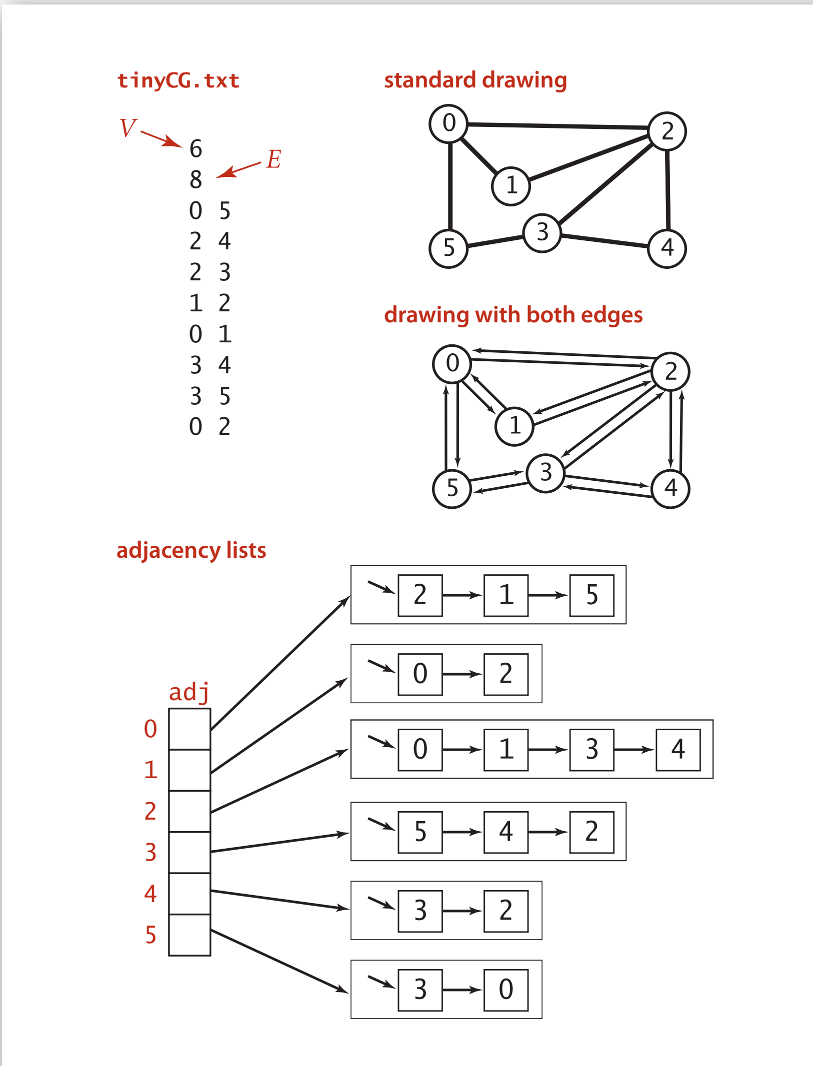
```
public class DepthFirstPaths
{
   private boolean[] marked;
   private int[] edgeTo;
   private final int s;

   public DepthFirstPaths(Graph G, int s)
   {
      marked = new boolean[G.V()];
      edgeTo = new int[G.V()];
      this.s = 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);
         }
   }

   public boolean hasPathTo(int v)
   public Iterable<Integer> pathTo(int v)
}
```

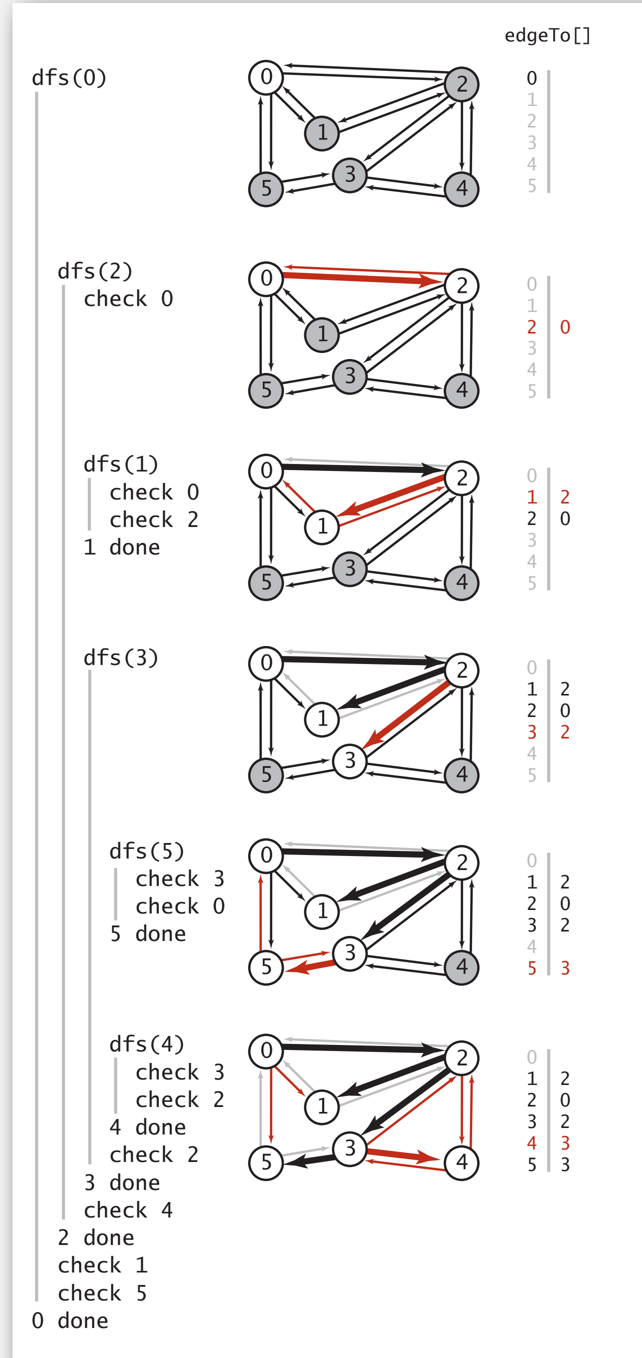
parent-link representation of DFS tree

set parent link

ahead

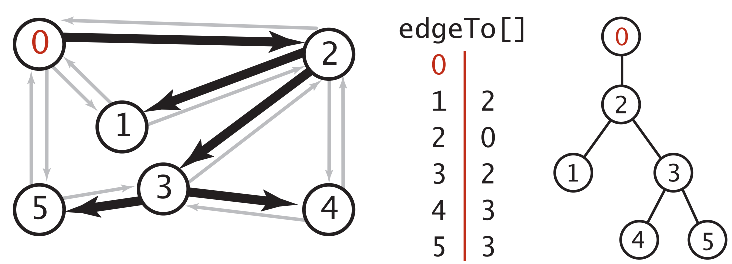
Depth-first search (pathfinding trace)





Depth-first search (pathfinding iterator)

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



x	path
5	5
3	3 5
2	2 3 5
0	0 2 3 5

```
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;
}
```

Depth-first search summary

Enables direct solution of simple graph problems.

- ✓ Does there exists a path between s and t?
- ✓ Find path between s and t.
- Connected components (stay tuned).
- Euler tour (see book).
- Cycle detection (see book).
- Bipartiteness checking (see book).

Basis for solving more difficult graph problems.

- Biconnected components (beyond scope).
- Planarity testing (beyond scope).

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.

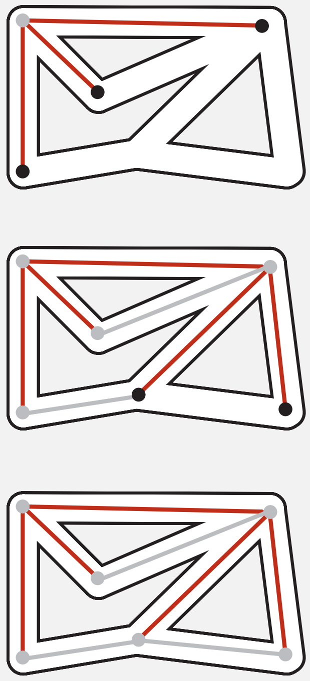
BFS (from source vertex s)

Put s onto a FIFO queue, and mark s as visited.

Repeat until the queue is empty:

- remove the least recently added vertex v

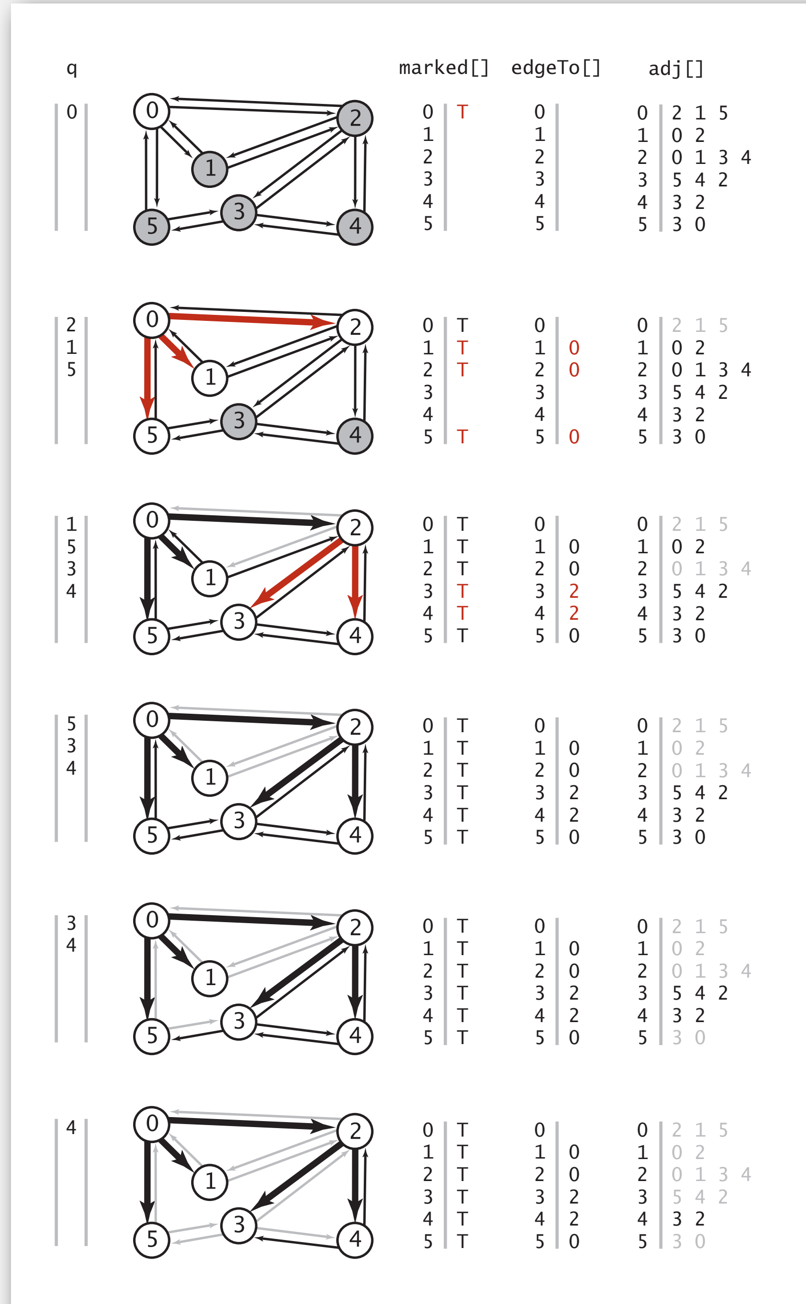
- add each of v's unvisited neighbors to the queue, and mark them as visited.



Intuition. BFS examines vertices in increasing distance from s.

Breadth-first search (pathfinding)

```
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;
         }
   }
}
```



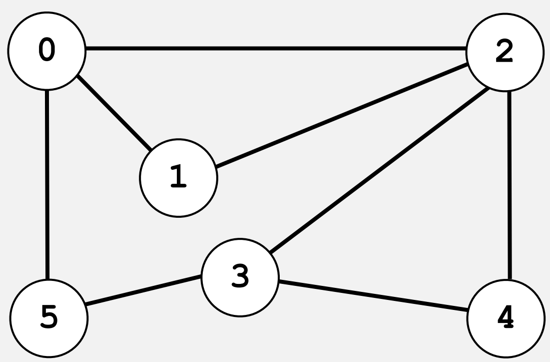
Breadth-first search properties

Proposition. BFS computes shortest path (number of edges) from s in a connected graph in time proportional to $E + V$.

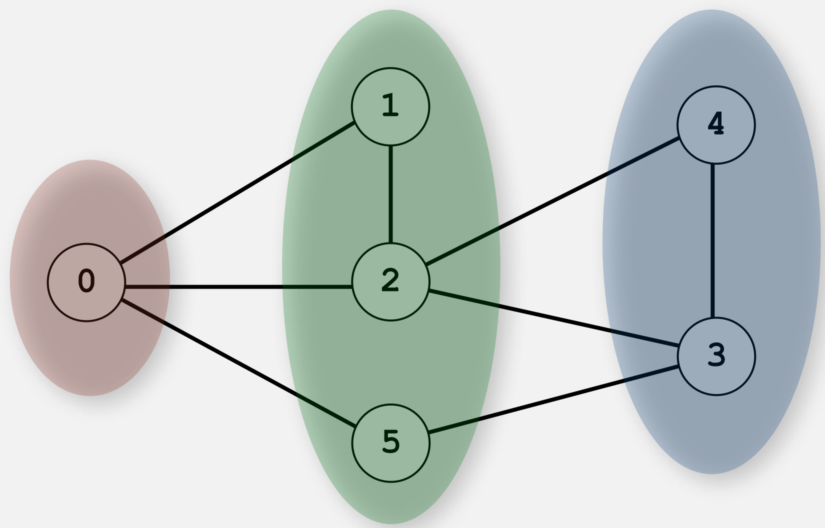
Pf.

- Correctness: queue always consists of zero or more vertices of distance k from s, followed by zero or more vertices of distance $k + 1$.

- Running time: each vertex connected to s is visited once.



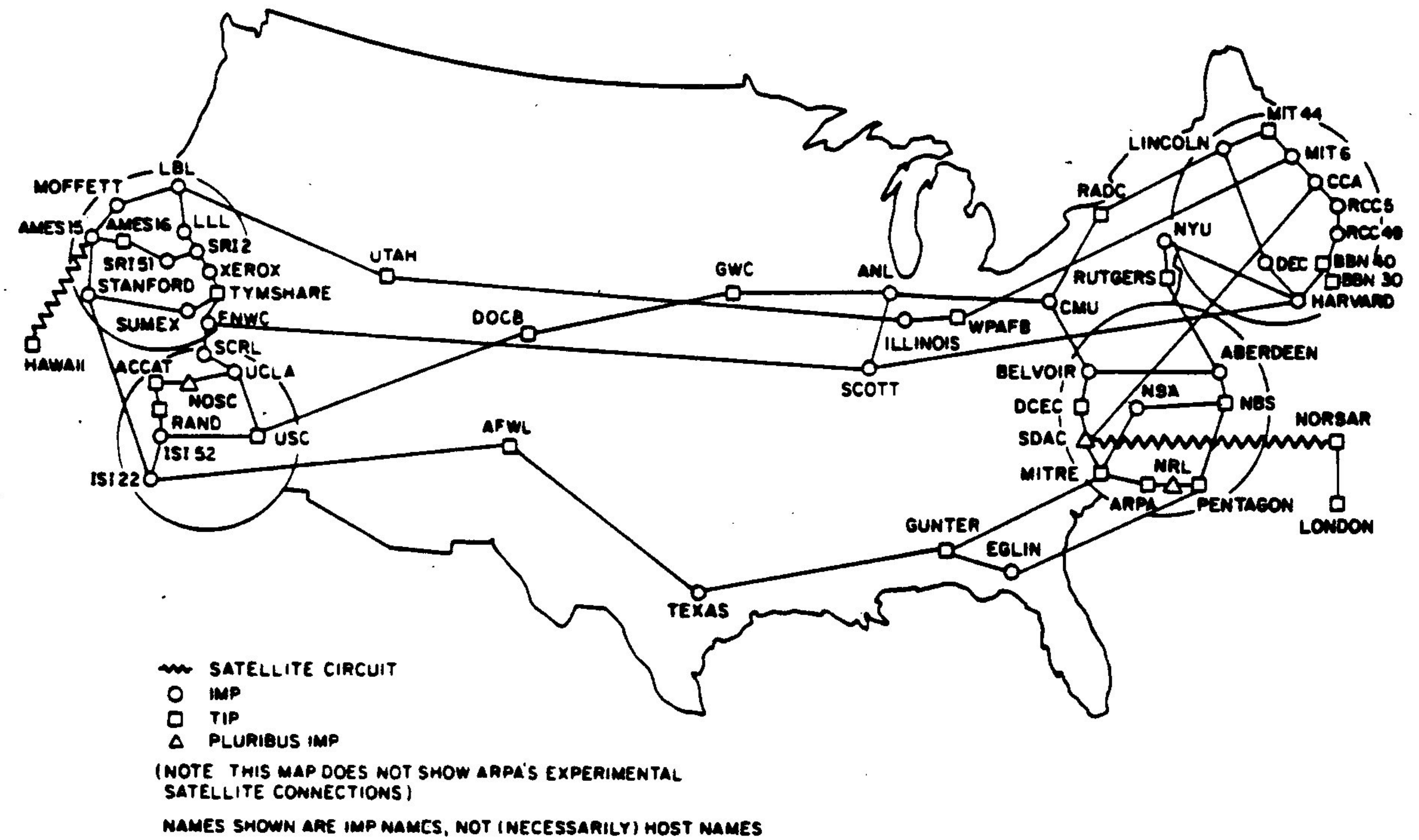
standard drawing



dist = 0 dist = 1 dist = 2

Breadth-first search application: routing

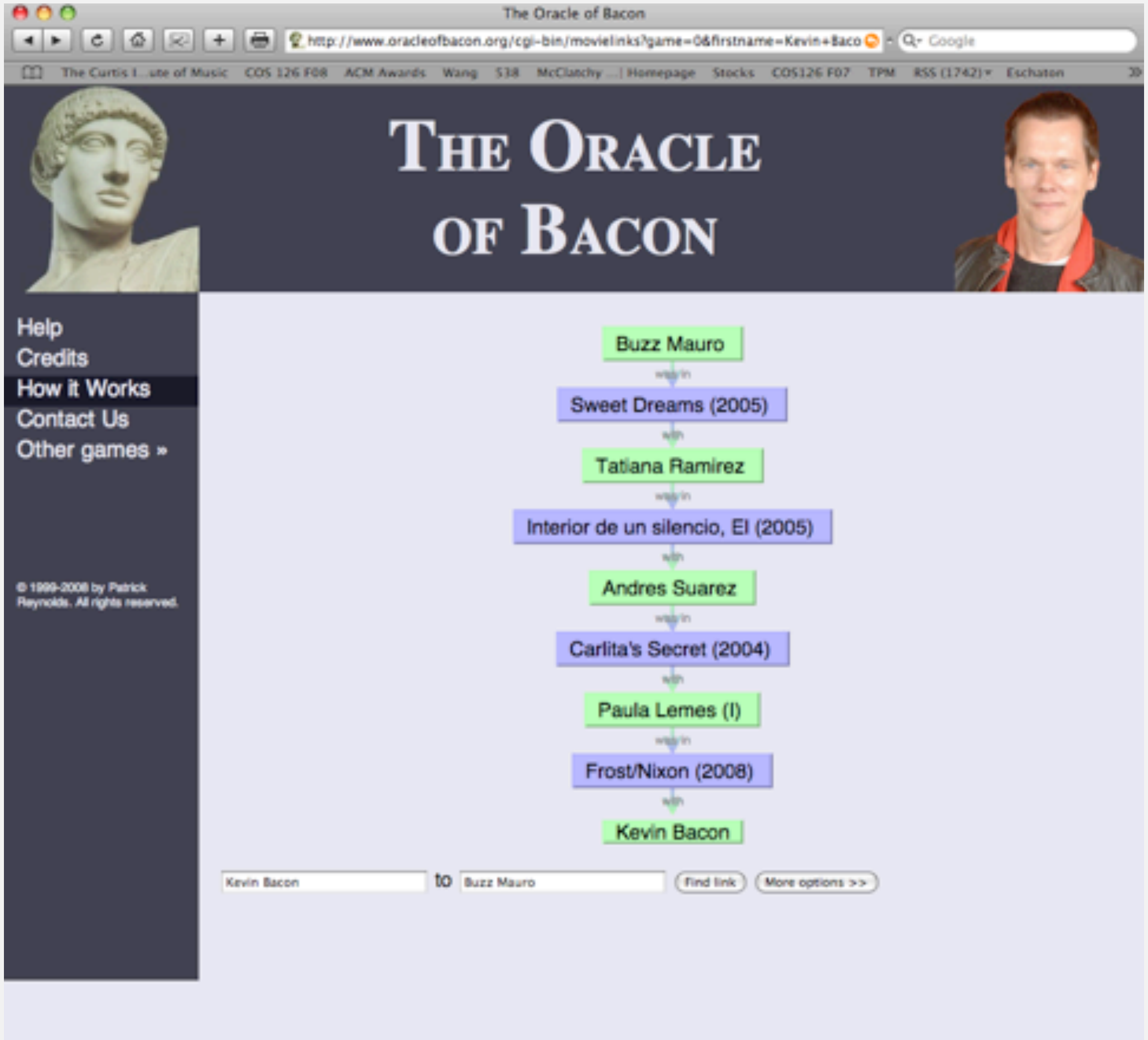
Fewest number of hops in a communication network.



ARPANET, July 1977

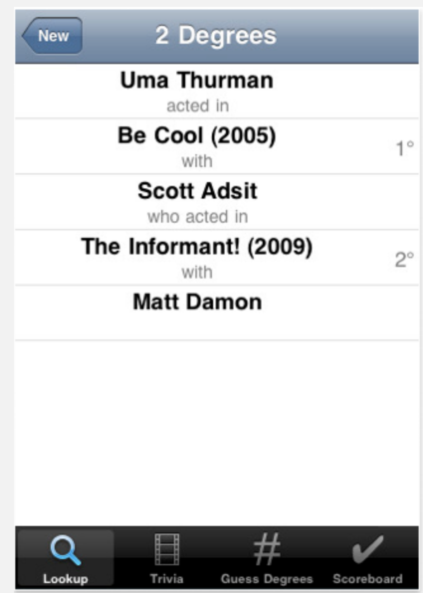
Breadth-first search application: Kevin Bacon numbers

Kevin Bacon numbers.



http://oracleofbacon.org

Endless Games board game



SixDegrees iPhone App

Kevin Bacon graph

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