Undirected Graphs

- Graph API
- maze exploration
- depth-first search
- breadth-first search
- connected components
- challenges

References:

Algorithms in Java, Chapters 17 and 18

Intro to Programming in Java, Section 4.5

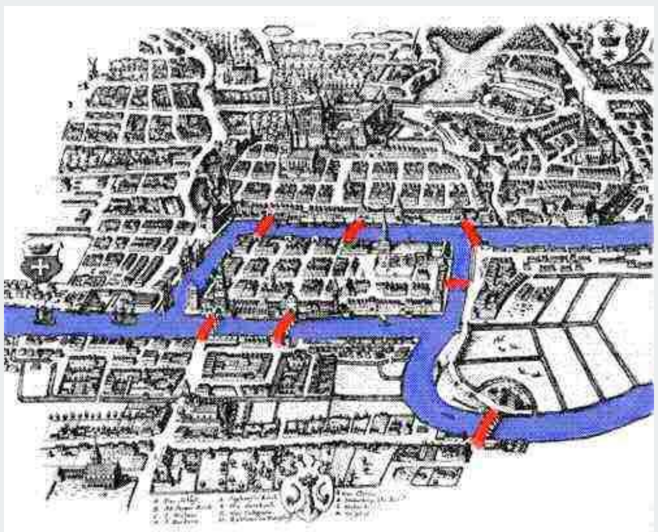
http://www.cs.princeton.edu/introalgsds/51undirected

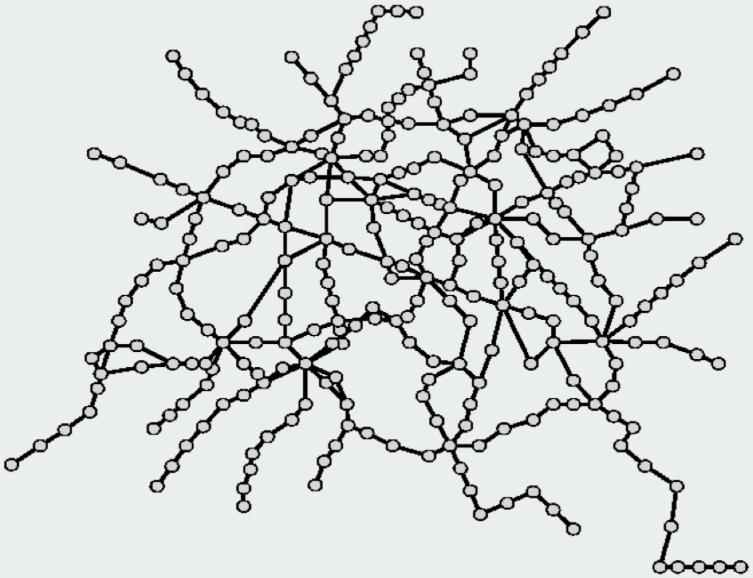
Undirected graphs

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.



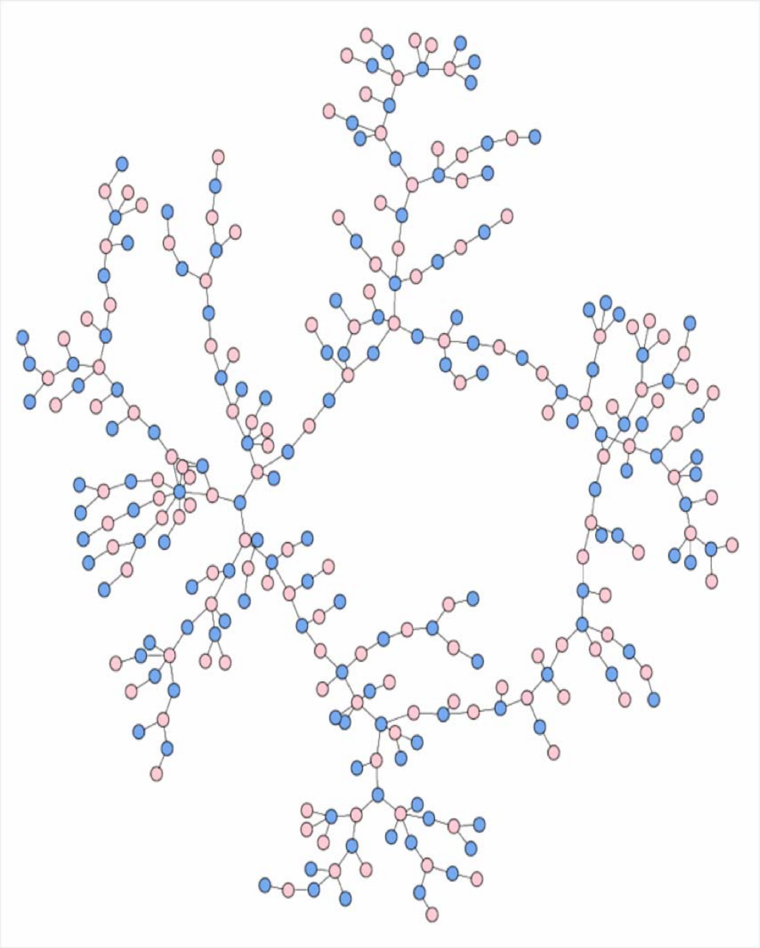


Graph applications

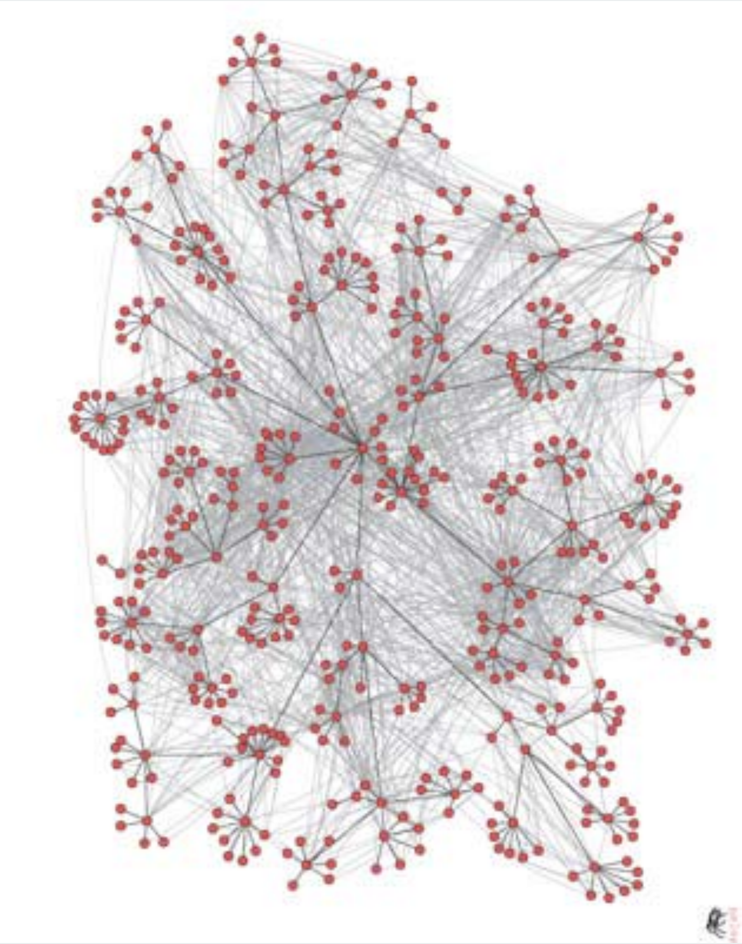
graph	vertices	edges
communication	telephones, computers	fiber optic cables
circuits	gates, registers, processors	wires
mechanical	joints	rods, beams, springs
hydraulic	reservoirs, pumping stations	pipelines
financial	stocks, currency	transactions
transportation	street intersections, airports	highways, airway routes
scheduling	tasks	precedence constraints
software systems	functions	function calls
internet	web pages	hyperlinks
games	board positions	legal moves
social relationship	people, actors	friendships, movie casts
neural networks	neurons	synapses
protein networks	proteins	protein-protein interactions
chemical compounds	molecules	bonds

Social networks

high school dating



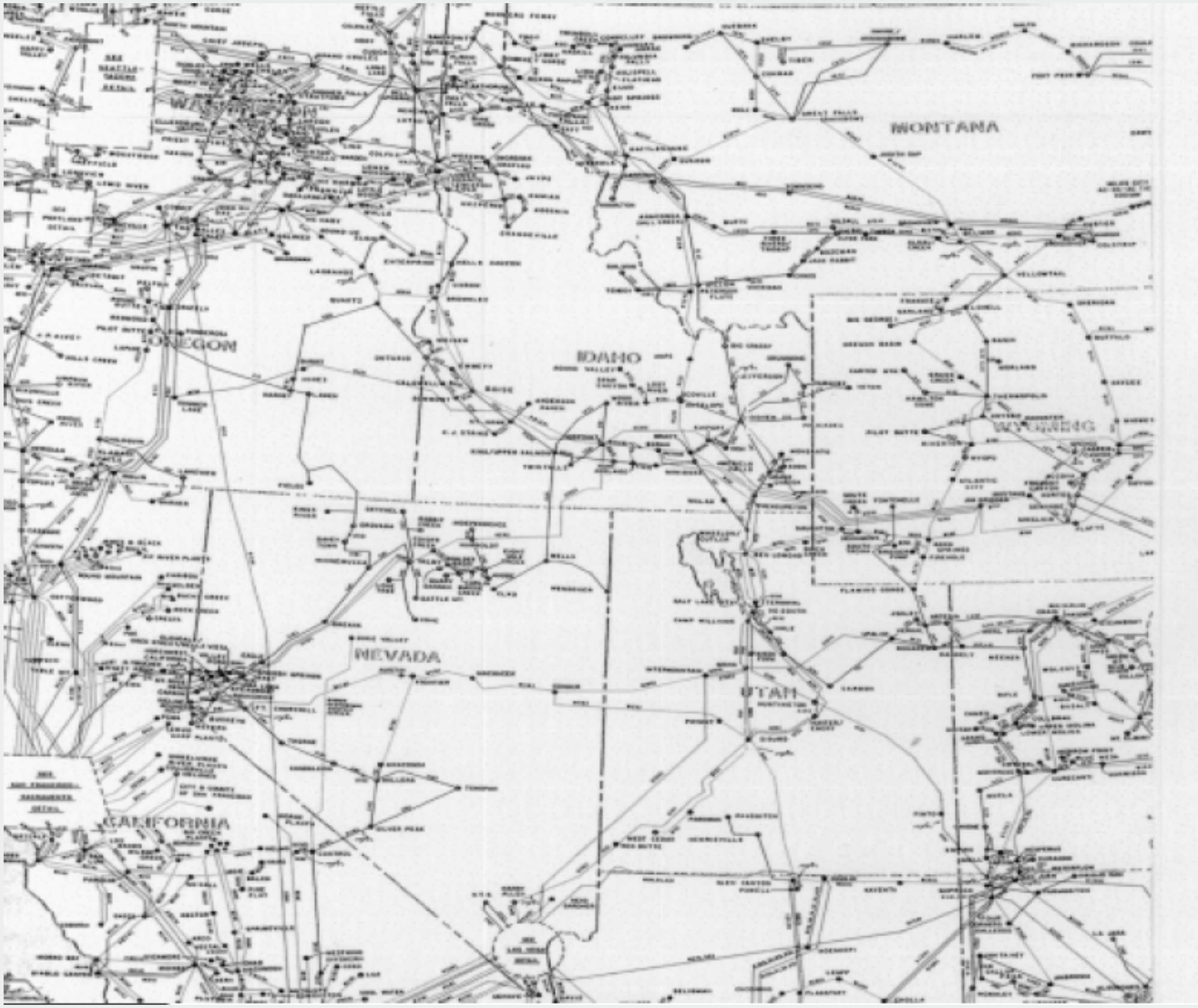
corporate e-mail



Reference: Bearman, Moody and Stovel, 2004
image by Mark Newman

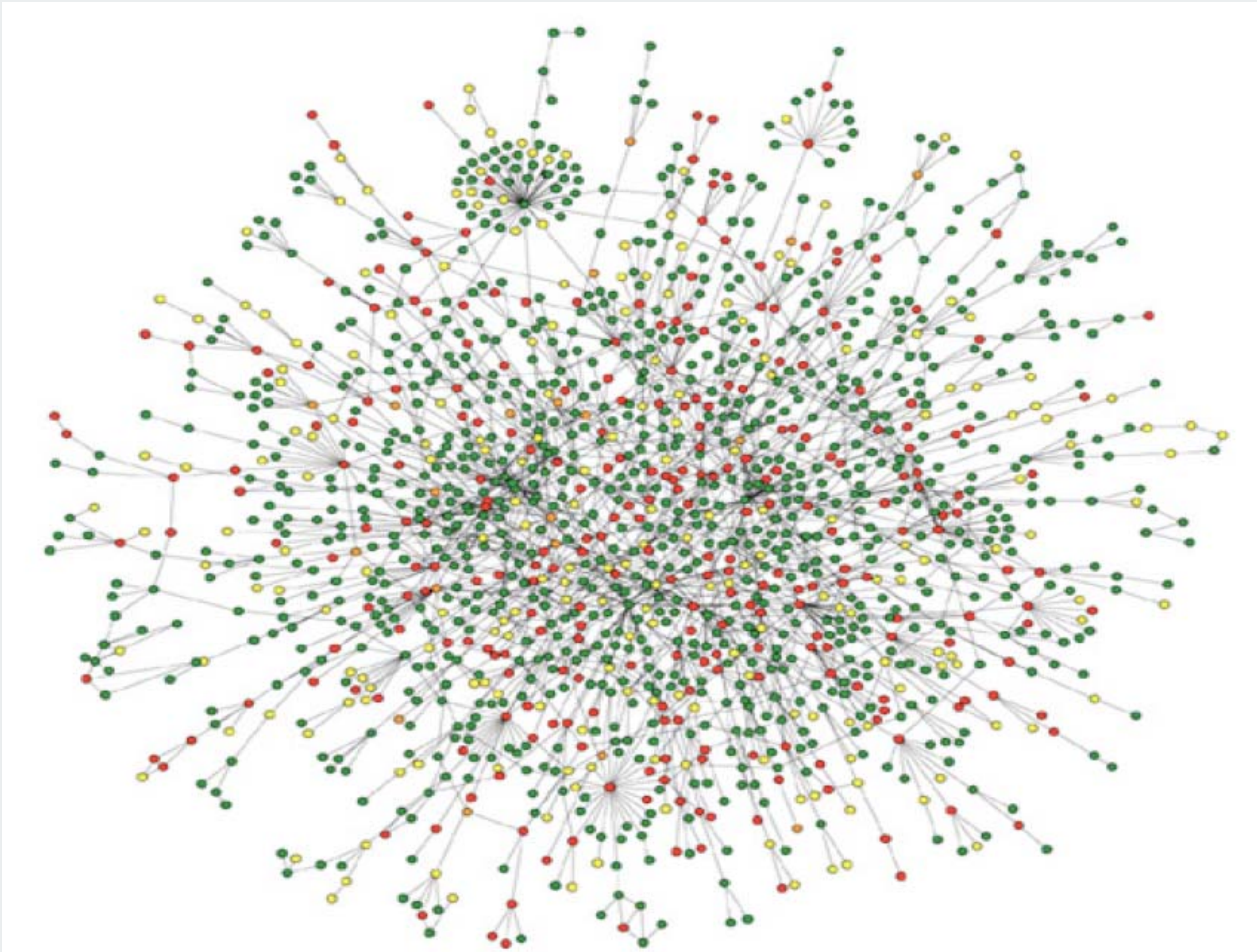
Reference: Adamic and Adar, 2004

Power transmission grid of Western US



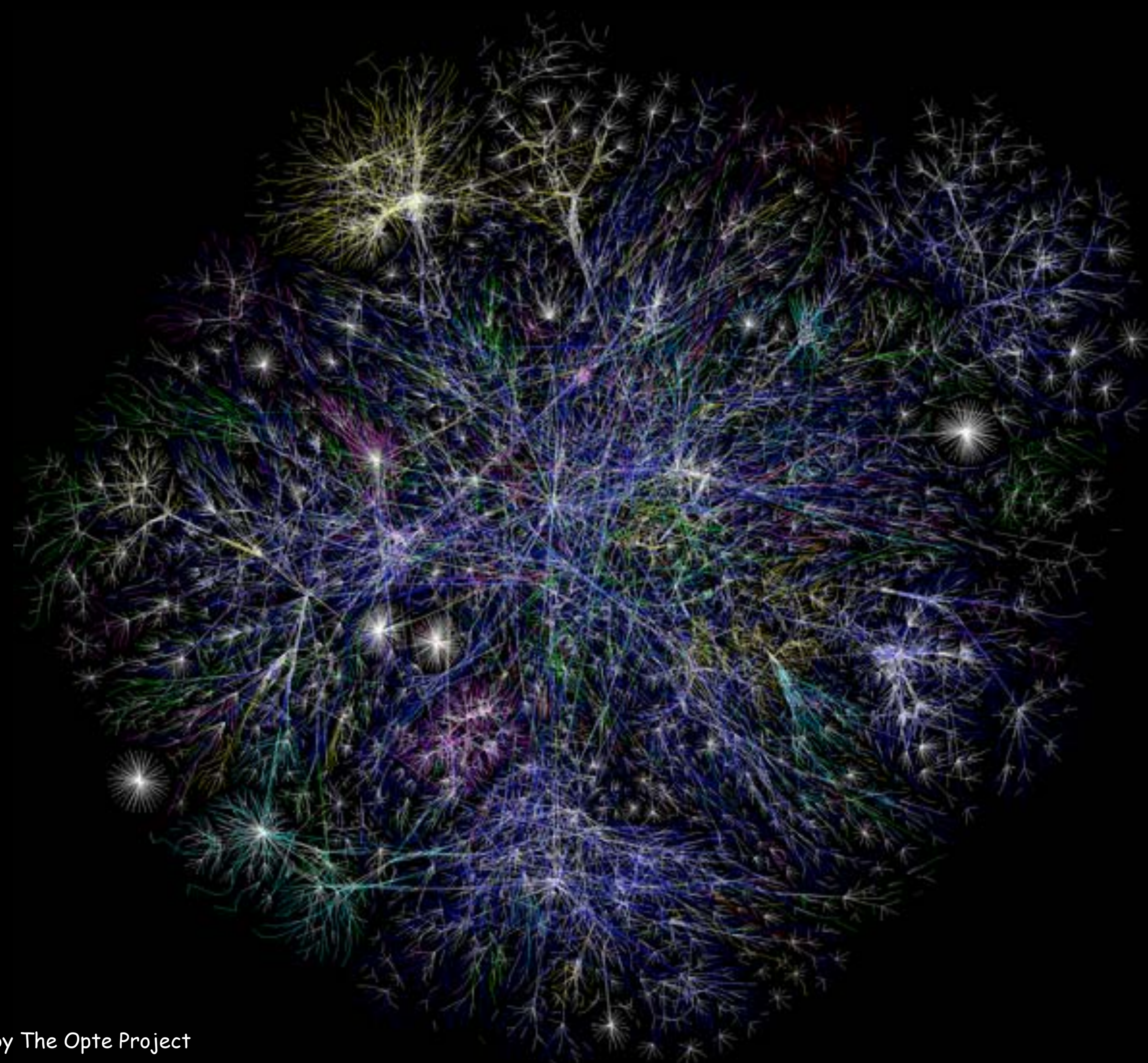
Reference: Duncan Watts

Protein interaction network



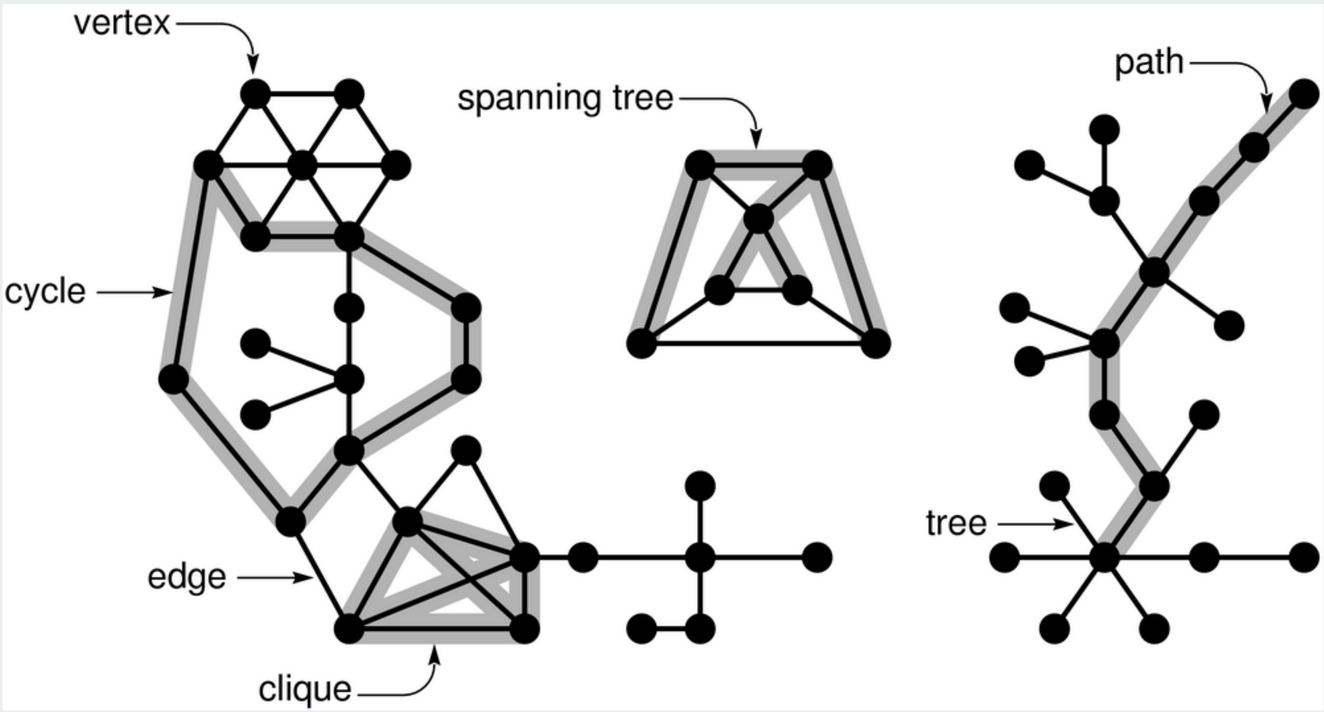
Reference: Jeong et al, Nature Review | Genetics

The Internet



The Internet as mapped by The Opte Project
http://www.opte.org

Graph terminology



Some graph-processing problems

Path. Is there a path between s to t?
Shortest path. What is the shortest path between s and t?
Longest path. What is the longest simple 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?

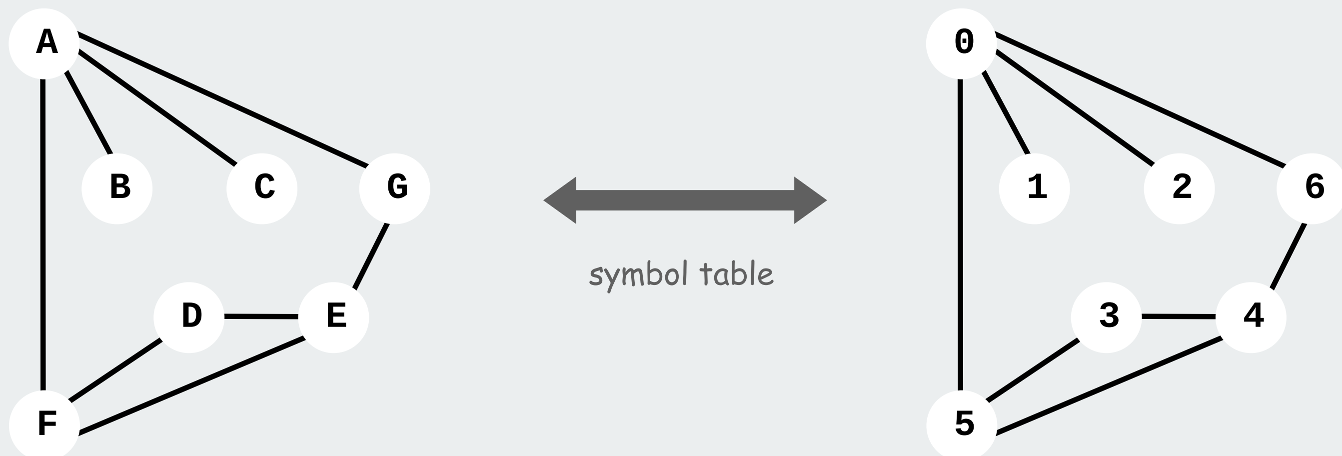
First challenge: Which of these problems is easy? difficult? intractable?

- **Graph API**
- maze exploration
- depth-first search
- breadth-first search
- connected components
- challenges

Graph representation

Vertex representation.

- This lecture: use integers between `0` and `V-1`.
- Real world: convert between names and integers with symbol table.



Other issues. Parallel edges, self-loops.

Graph API

public class Graph	*(graph data type)*	
	`Graph(int V)`	*create an empty graph with V vertices*
	`Graph(int V, int E)`	*create a random graph with V vertices, E edges*
`void`	`addEdge(int v, int w)`	*add an edge v-w*
`Iterable<Integer>`	`adj(int v)`	*return an iterator over the neighbors of v*
`int`	`V()`	*return number of vertices*
`String`	`toString()`	*return a string representation*

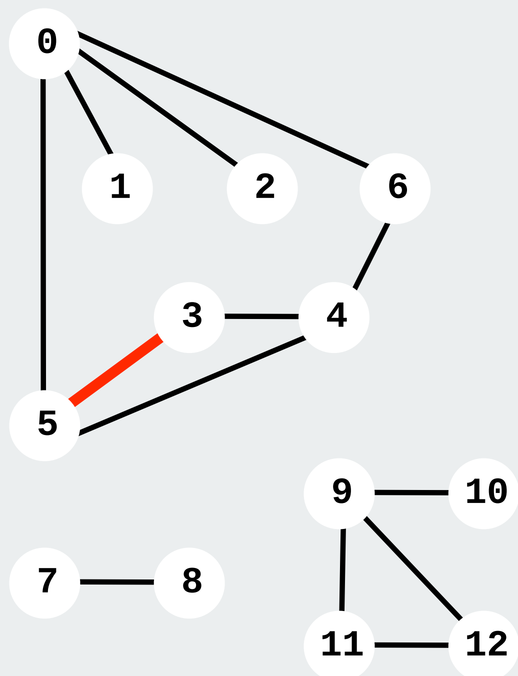
Client that iterates through all edges

```
Graph G = new Graph(V, E);
StdOut.println(G);
for (int v = 0; v < G.V(); v++)
   for (int w : G.adj(v))
     // process edge v-w
```

processes BOTH
v-w and w-v

Set of edges representation

Store a list of the edges (linked list or array)

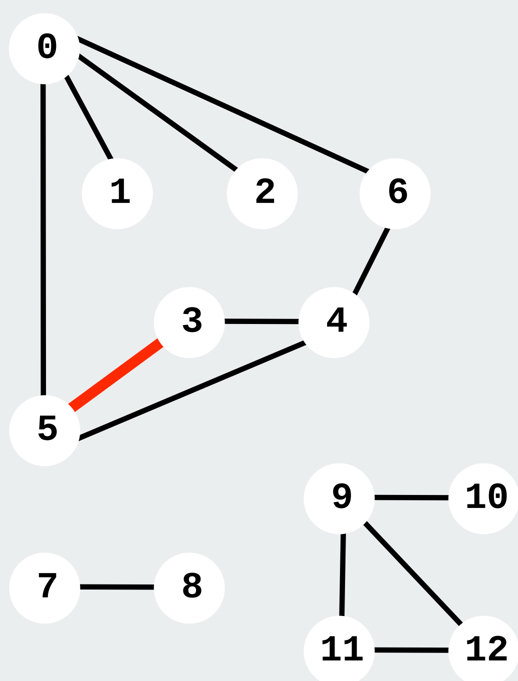


0-1
0-6
0-2
11-12
9-12
9-11
9-10
4-3
5-3
7-8
5-4
0-5
6-4

Adjacency matrix representation

Maintain a two-dimensional $V \times V$ 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-matrix graph representation: Java implementation

```
public class Graph
{
   private int V;
   private boolean[][] adj;

   public Graph(int V)
   {
      this.V = V;
      adj = new boolean[V][V];
   }

   public void addEdge(int v, int w)
   {
      adj[v][w] = true;
      adj[w][v] = true;
   }

   public Iterable<Integer> adj(int v)
   {
      return new AdjIterator(v);
   }
}
```

adjacency matrix

create empty V-vertex graph

add edge v-w (no parallel edges)

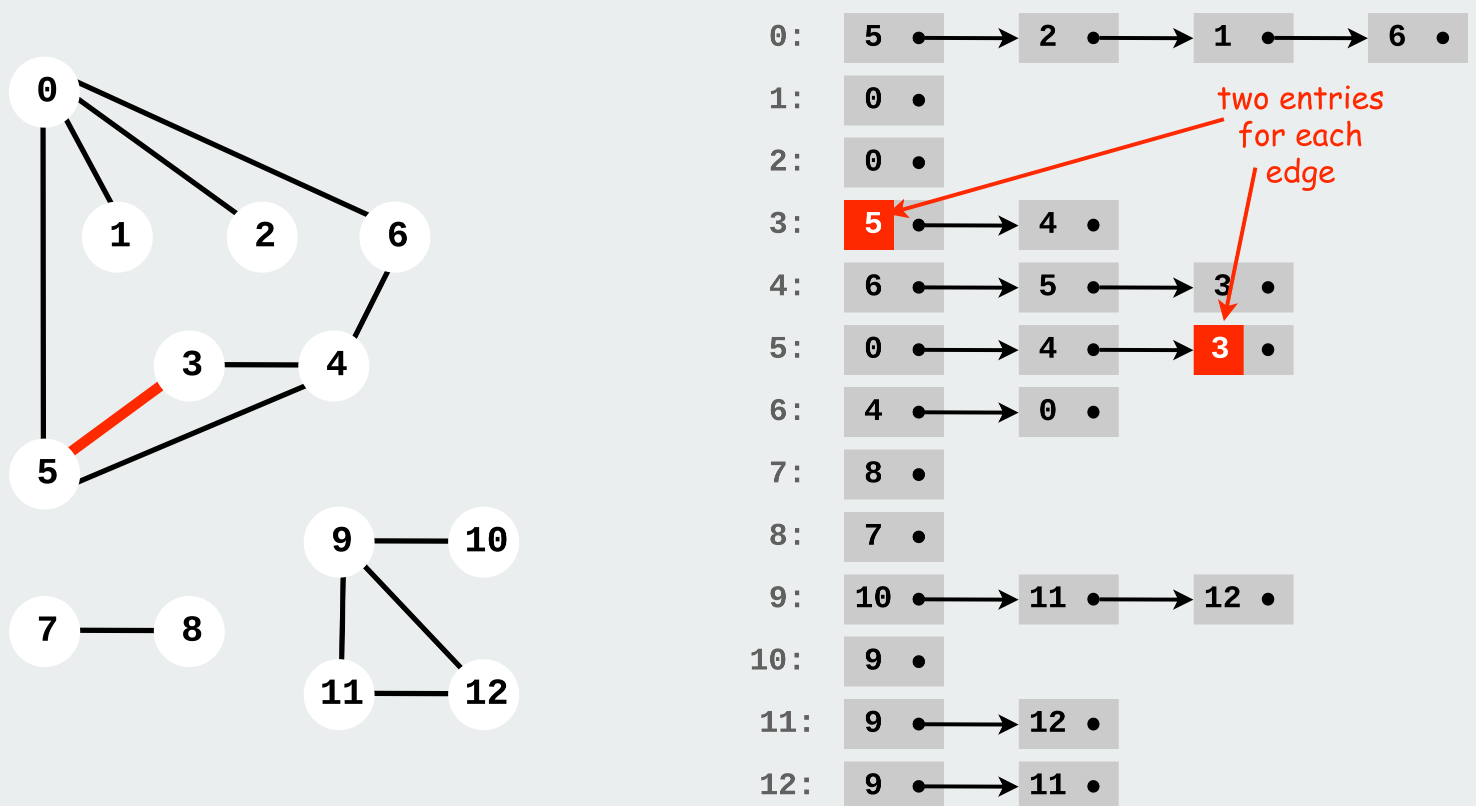
iterator for v's neighbors

Adjacency matrix: iterator for vertex neighbors

```
private class AdjIterator implements Iterator<Integer>,
                                     Iterable<Integer>
{
   int v, w = 0;
   AdjIterator(int v)
   {  this.v = v;  }

   public boolean hasNext()
   {
      while (w < V)
      {  if (adj[v][w]) return true; w++ }
      return false;
   }

   public int next()
   {
      if (hasNext()) return w++ ;
      else throw new NoSuchElementException();
   }

   public Iterator<Integer> iterator()
   { return this; }

}
```

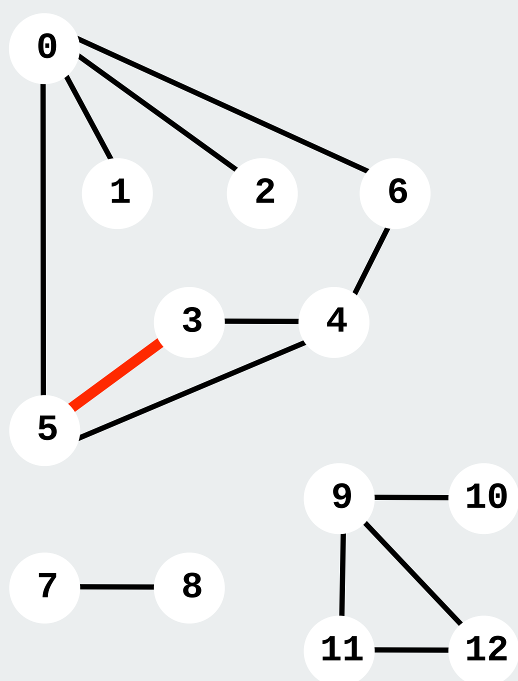
Adjacency-list graph representation

Maintain vertex-indexed array of lists (implementation omitted)



Adjacency-SET graph representation

Maintain vertex-indexed array of SETs
(take advantage of balanced-tree or hashing implementations)




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

two entries for each edge

Adjacency-SET graph representation: Java implementation

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

   public Graph(int V)
   {
      this.V = V;
      adj = (SET<Integer>[]) new SET[V];
      for (int v = 0; v < V; v++)
         adj[v] = new SET<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 sets

create empty V-vertex graph

add edge v-w (no parallel edges)

iterable SET for v's neighbors

Graph representations

Graphs are abstract mathematical objects, BUT

- ADT implementation requires specific representation.
- Efficiency depends on matching algorithms to representations.

representation	space	edge between v and w?	iterate over edges incident to v?
list of edges	E	E	E
adjacency matrix	V^2	1	V
adjacency list	E + V	degree(v)	degree(v)
adjacency SET	E + V	log (degree(v))	degree(v)*

* easy to also support ordered iteration and randomized iteration

In practice: Use adjacency SET representation

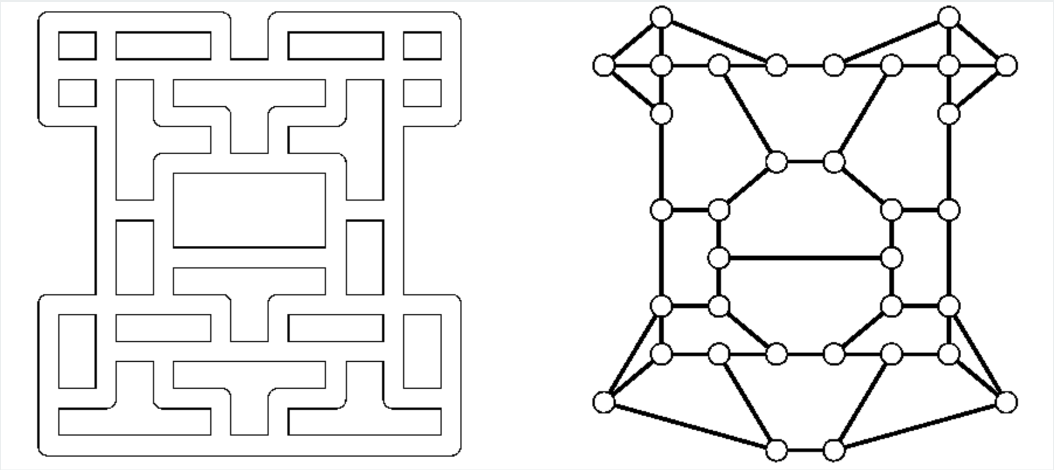
- Take advantage of proven technology
- Real-world graphs tend to be "sparse"
  [huge number of vertices, small average vertex degree]
- Algs all based on iterating over edges incident to v.

- Graph API
- **maze exploration**
- depth-first search
- breadth-first search
- connected components
- challenges

Maze exploration

Maze graphs.

- Vertex = intersections.
- Edge = passage.



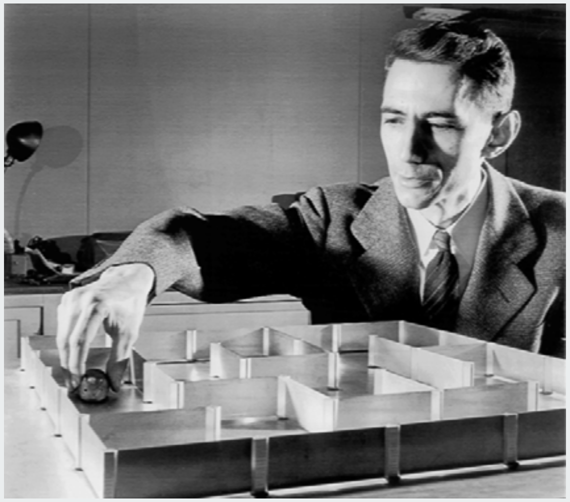
Goal. Explore every passage in the maze.

Trémaux Maze Exploration

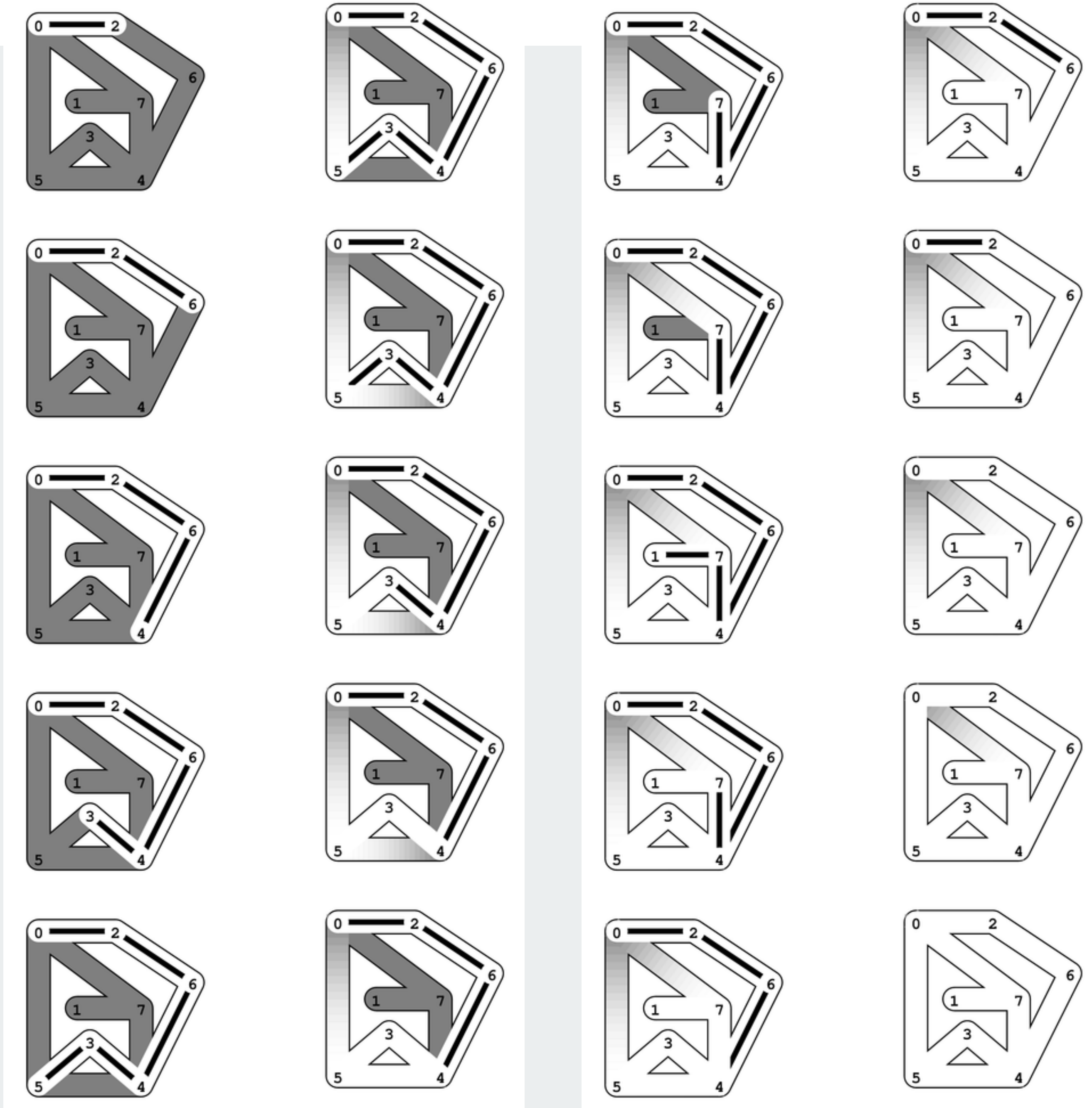
Trémaux maze exploration.

- Unroll a ball of string behind you.
- Mark each visited intersection by turning on a light.
- Mark each visited passage by opening a door.

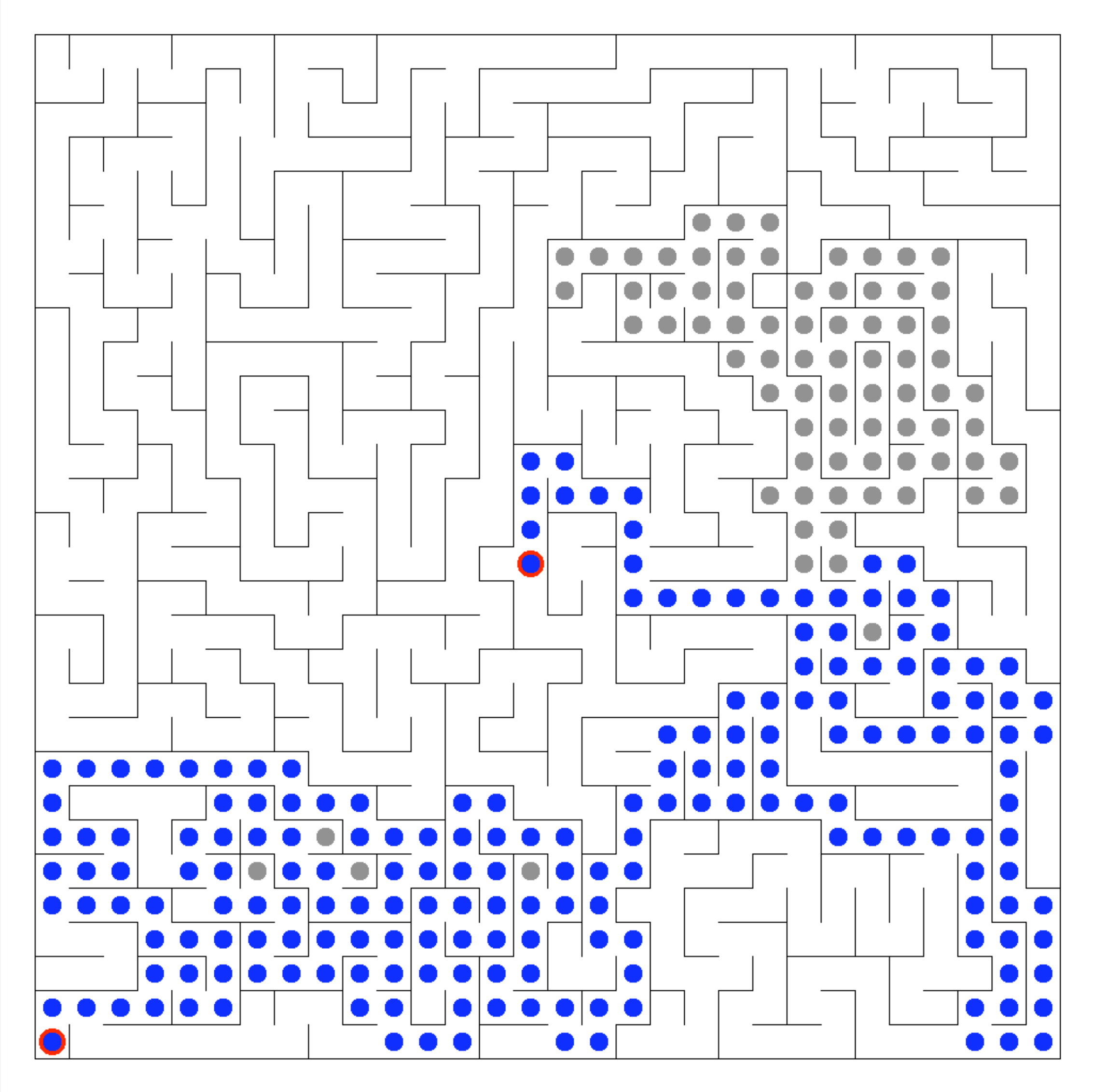
First use? Theseus entered labyrinth to kill the monstrous Minotaur; Ariadne held ball of string.



Claude Shannon (with Theseus mouse)



Maze Exploration



- Graph API
- maze exploration
- **depth-first search**
- breadth-first search
- connected components
- challenges

Flood fill

Photoshop "magic wand"

Graph-processing challenge 1:

Problem: Flood fill

Assumptions: picture has millions to billions of pixels

How difficult?

1) any COS126 student could do it
2) need to be a typical diligent COS226 student
3) hire an expert
4) intractable
5) no one knows

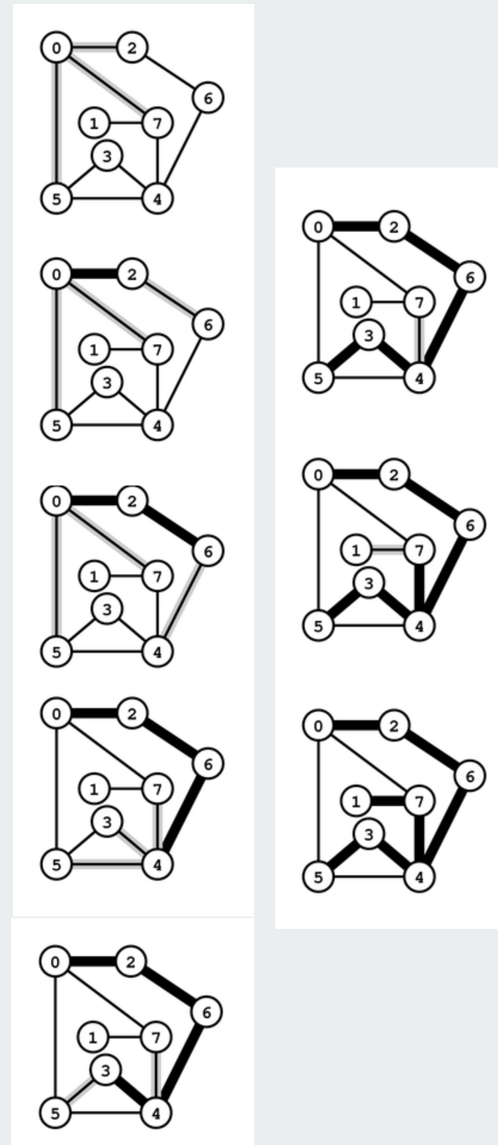
Depth-first search

Goal. Systematically search through a graph.

Idea. Mimic maze exploration.

Typical applications.

- find all vertices connected to a given s
- find a path from s to t



DFS (to visit a vertex s)
Mark s as visited.
Visit all unmarked vertices v adjacent to s.

recursive

Running time.

- O(E) since each edge examined at most twice
- usually less than V to find paths in real graphs

Design pattern for graph processing

Typical client program.

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

Client that prints all vertices connected to (reachable from) s

```
public static void main(String[] args)
{
   In in = new In(args[0]);
   Graph G = new Graph(in);
   int s = 0;
   DFSearcher dfs = new DFSearcher(G, s);
   for (int v = 0; v < G.V(); v++)
      if (dfs.isConnected(v))
         System.out.println(v);
}
```

Decouple graph from graph processing.

Depth-first search (connectivity)

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

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

true if connected to s

constructor marks vertices connected to s

recursive DFS does the work

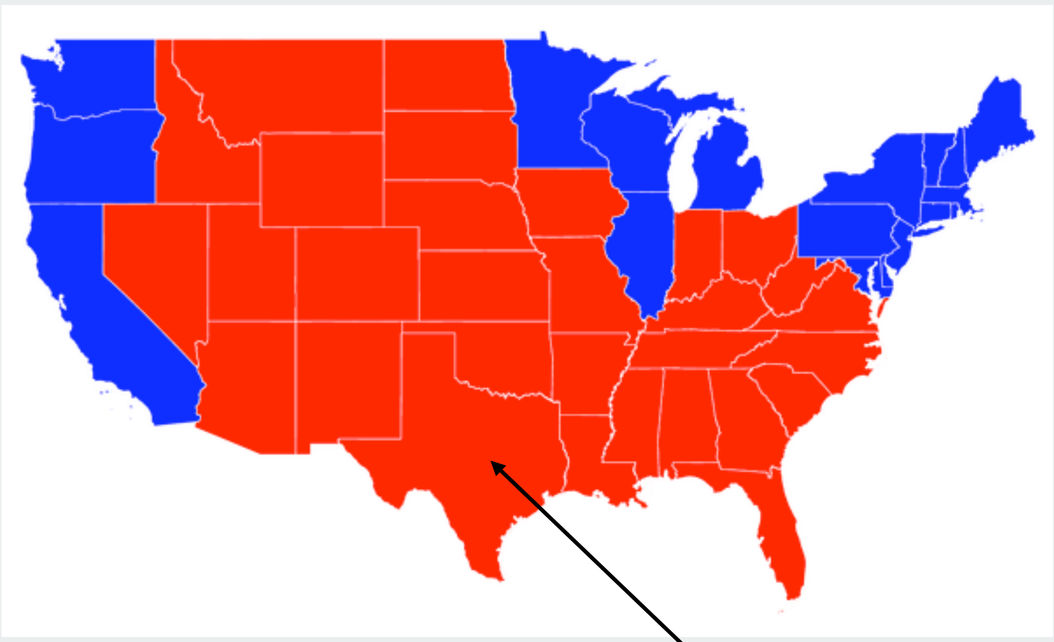
client can ask whether any vertex is connected to s

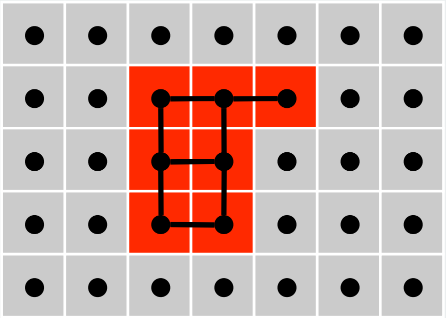
Connectivity application: Flood fill

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

Build a grid graph

- vertex: pixel.
- edge: between two adjacent lime pixels.
- blob: all pixels connected to given pixel.





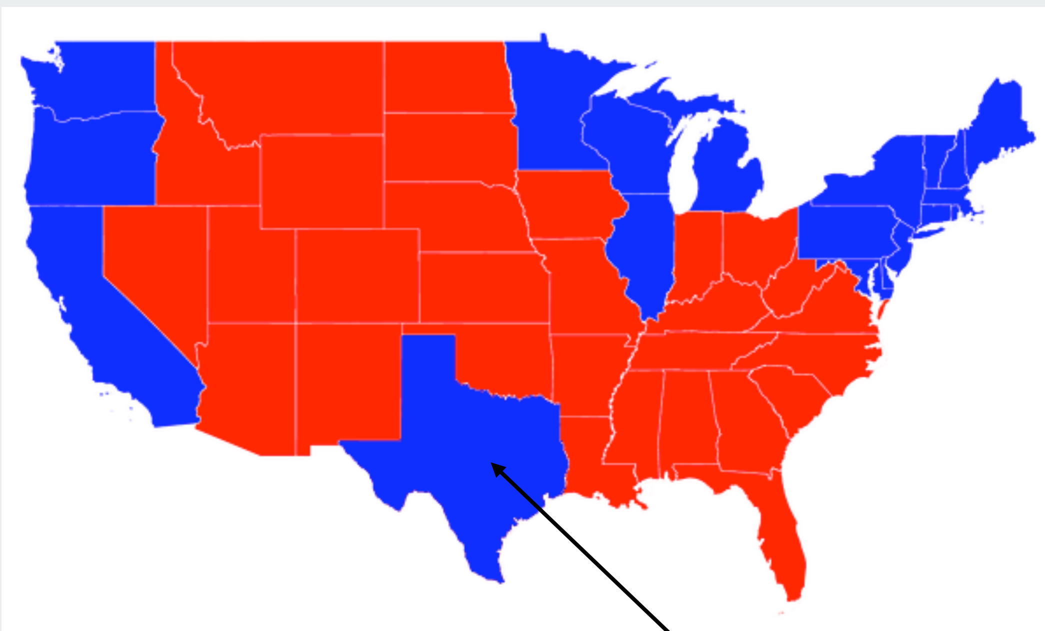
recolor red blob to blue

Connectivity Application: Flood Fill

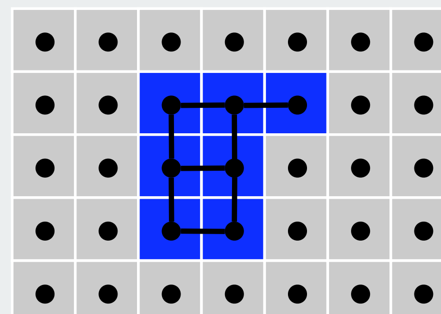
Change color of entire blob of neighboring red pixels to blue.

Build a grid graph

- vertex: pixel.
- edge: between two adjacent red pixels.
- blob: all pixels connected to given pixel.

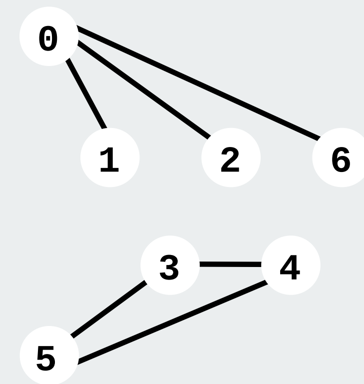


recolor red blob to blue



Graph-processing challenge 2:

Problem: Is there a path from s to t ?



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

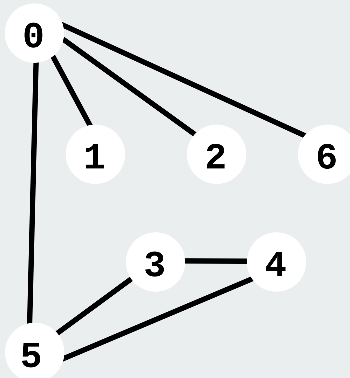
How difficult?

1) any CS126 student could do it
2) need to be a typical diligent CS226 student
3) hire an expert
4) intractable
5) no one knows

Graph-processing challenge 3:

Problem: Find a path from s to t.

Assumptions: any path will do



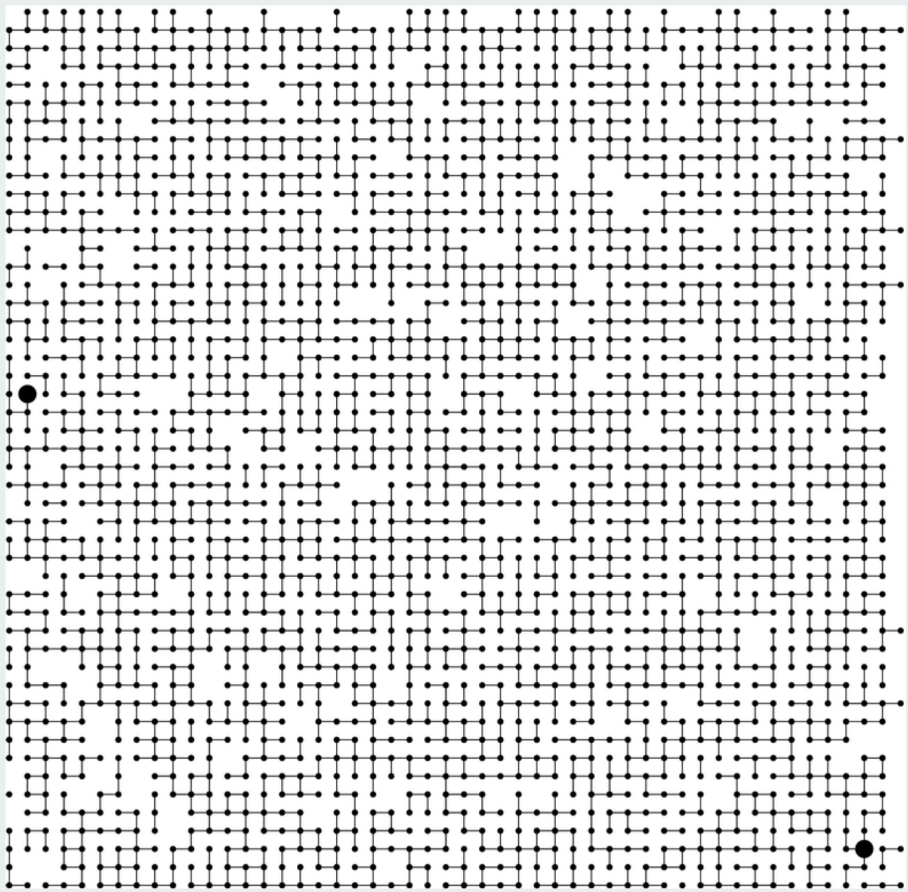
0-1
0-6
0-2
4-3
5-3
5-4
0-5

How difficult?

1) any CS126 student could do it
2) need to be a typical diligent CS226 student
3) hire an expert
4) intractable
5) no one knows

Paths in graphs

Is there a path from s to t? If so, find one.



Paths in graphs

Is there a path from s to t?

method	preprocess time	query time	space
Union Find	V + E log* V	log* V †	V
DFS	E + V	1	E + V

† amortized

If so, find one.

- Union-Find: no help (use DFS on connected subgraph)
- DFS: easy (stay tuned)

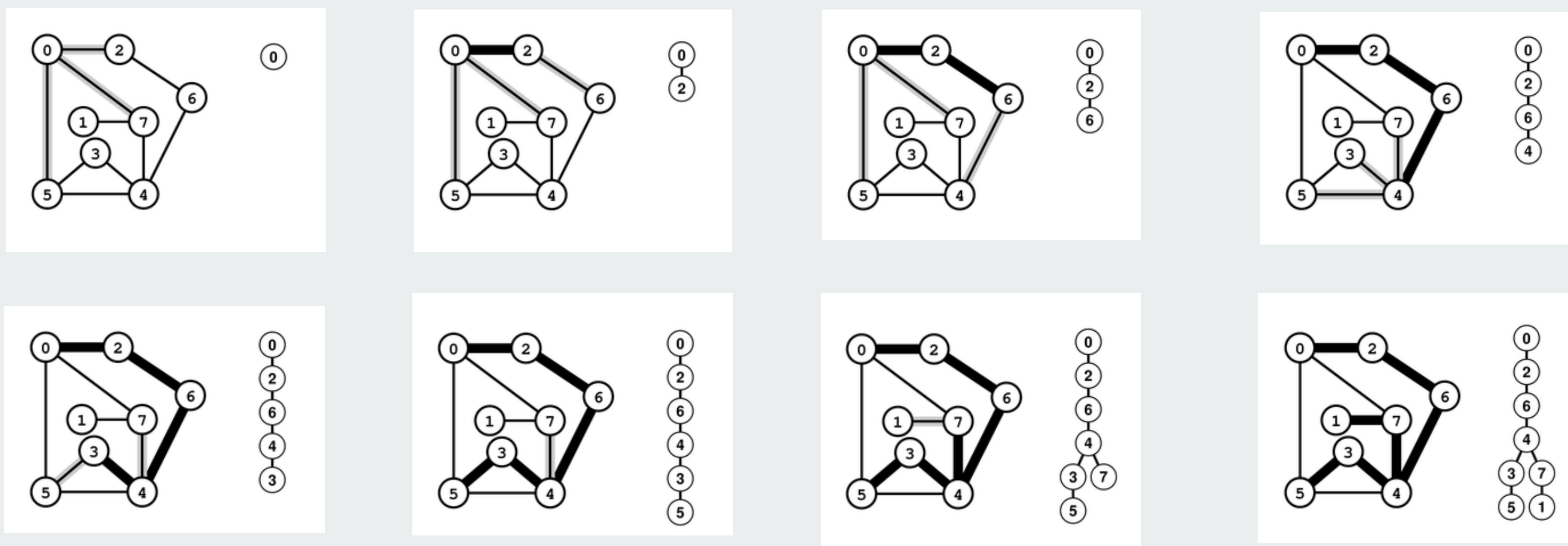
UF advantage. Can intermix queries and edge insertions.

DFS advantage. Can recover path itself in time proportional to its length.

Keeping track of paths with DFS

DFS tree. Upon visiting a vertex v for the first time, remember that you came from **pred[v]** (parent-link representation).

Retrace path. To find path between s and v, follow **pred** back from v.



Depth-first-search (pathfinding)

```
public class DFSearcher
{
   ...
   private int[] pred;
   public DFSearcher(Graph G, int s)
   {
      ...
      pred = new int[G.V()];
      for (int v = 0; v < G.V(); v++)
         pred[v] = -1;
      ...
   }
   private void dfs(Graph G, int v)
   {
      marked[v] = true;
      for (int w : G.adj(v))
         if (!marked[w])
         {
            pred[w] = v;
            dfs(G, w);
         }
   }

   public Iterable<Integer> path(int v)
   {  // next slide  }
}
```

add instance variable for parent-link representation of DFS tree

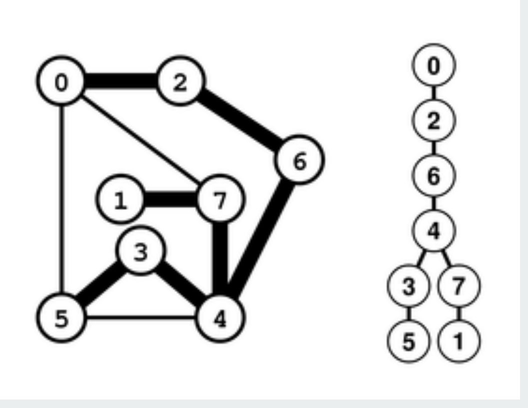
initialize it in the constructor

set parent link

add method for client to iterate through path

Depth-first-search (pathfinding iterator)

```
    public Iterable<Integer> path(int v)
    {
       Stack<Integer> path = new Stack<Integer>();
       while (v != -1 && marked[v])
       {
          list.push(v);
          v = pred[v];
       }
       return path;
    }
}
```



DFS summary

Enables direct solution of simple graph problems.

- Find path from s to t. ✓
- Connected components (stay tuned).
- Euler tour (see book).
- Cycle detection (simple exercise).
- Bipartiteness checking (see book).

Basis for solving more difficult graph problems.

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

- Graph API
- maze exploration
- depth-first search
- **breadth-first search**
- connected components
- challenges

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.

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.

Property. BFS examines vertices in increasing distance from s.

Breadth-first search scaffolding

```
public class BFSearcher
{
   private int[] dist;

   public BFSearcher(Graph G, int s)
   {
      dist = new int[G.V()];
      for (int v = 0; v < G.V(); v++)
         dist[v] = G.V() + 1;
      dist[s] = 0;

      bfs(G, s);
   }

   public int distance(int v)
   {  return dist[v];  }

   private void bfs(Graph G, int s)
   {  // See next slide.   }

}
```

distances from s

initialize distances

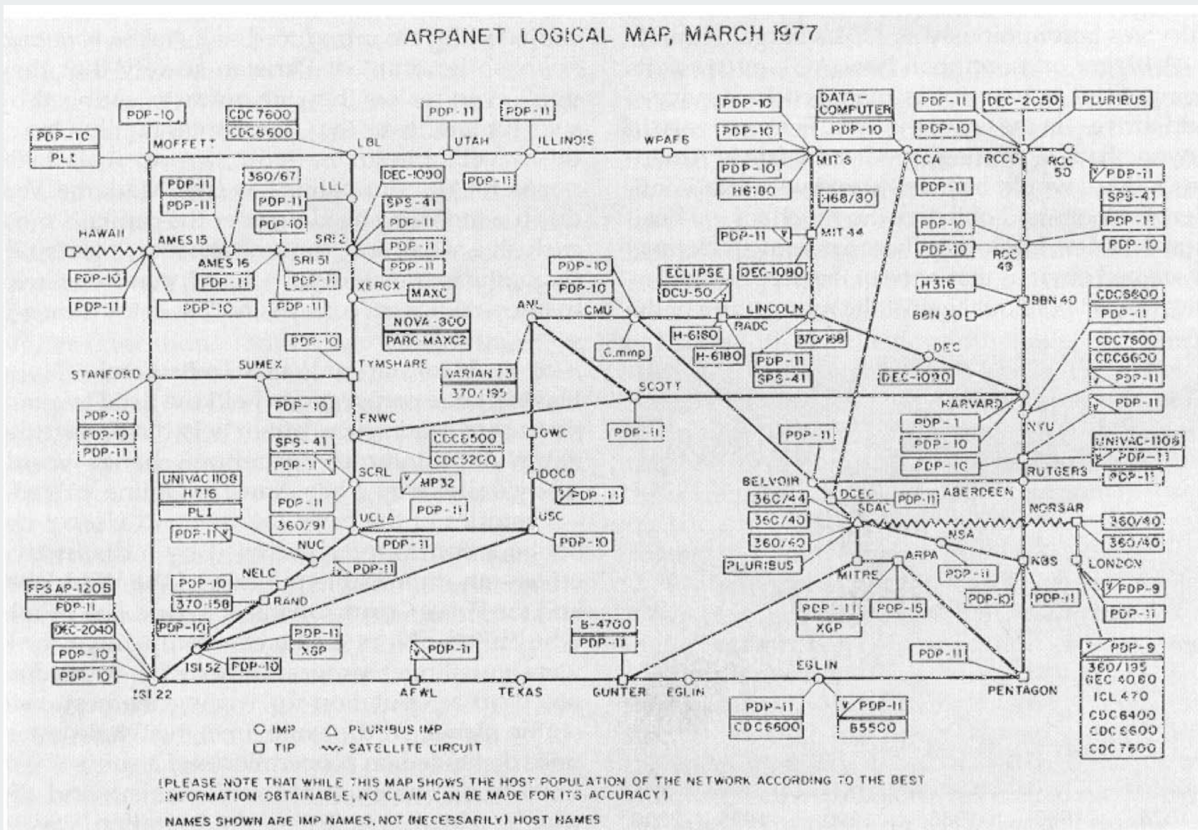
compute distances

answer client query

Breadth-first search (compute shortest-path distances)

```
private void bfs(Graph G, int s)
{
   Queue<Integer> q = new Queue<Integer>();
   q.enqueue(s);
   while (!q.isEmpty())
   {
      int v = q.dequeue();
      for (int w : G.adj(v))
      {
         if (dist[w] > G.V())
         {
            q.enqueue(w);
            dist[w] = dist[v] + 1;
         }
      }
   }
}
```

- Kevin Bacon numbers.
- Facebook.
- Fewest number of hops in a communication network.



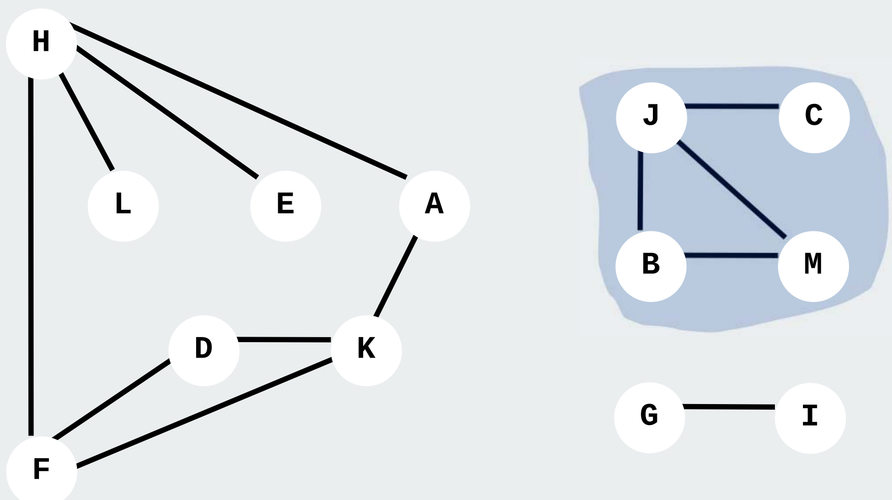
ARPANET

- Graph API
- maze exploration
- 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.
Def. A connected component is a maximal set of connected vertices.

Goal. Preprocess graph to answer queries: is v connected to w?
in constant time



Vertex	Component
A	0
B	1
C	1
D	0
E	0
F	0
G	2
H	0
I	2
J	1
K	0
L	0
M	1

Union-Find? not quite

Goal. Partition vertices into connected components.

Connected components

Initialize all vertices **v** as unmarked.

For each unmarked vertex **v**, run DFS and identify all vertices discovered as part of the same connected component.

---

preprocess Time	query Time	extra Space
E + V	1	V

Depth-first search for connected components

```
public class CCFinder
{
   private final static int UNMARKED = -1;
   private int components;
   private int[] cc;
   public CCFinder(Graph G)
   {
      cc = new int[G.V()];
      for (int v = 0; v < G.V(); v++)
         cc[v] = UNMARKED;
      for (int v = 0; v < G.V(); v++)
         if (cc[v] == UNMARKED)
            { dfs(G, v); components++; }
   }
   private void dfs(Graph G, int v)
   {
      cc[v] = components;
      for (int w : G.adj(v))
         if (cc[w] == UNMARKED) dfs(G, w);
   }

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

}
```

component labels

DFS for each component

standard DFS

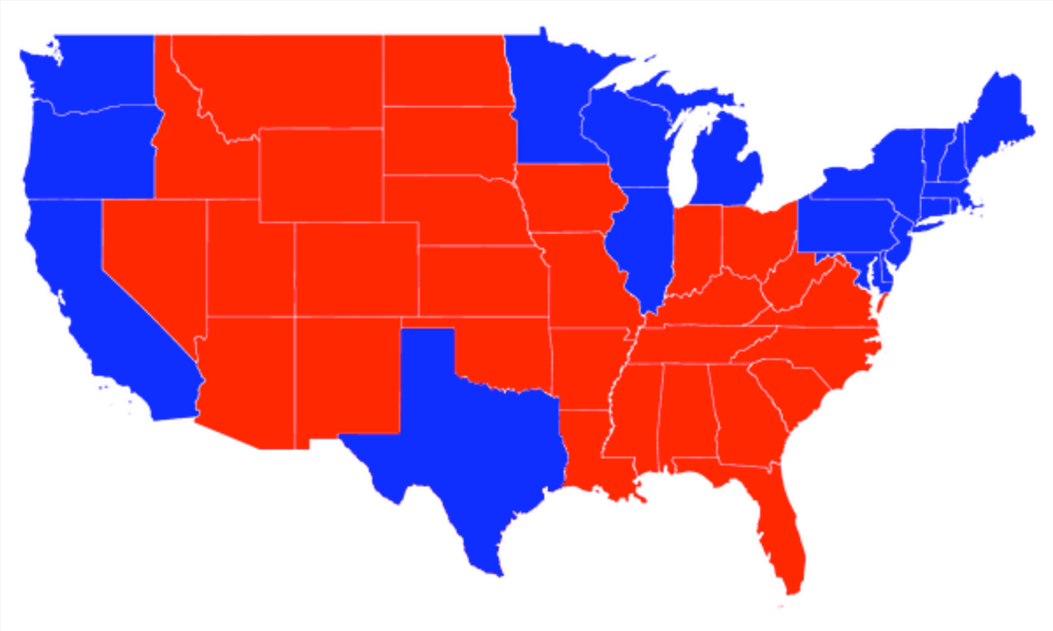
constant-time connectivity query

Connected Components

63 components

Connected components application: Image processing

Goal. Read in a 2D color image and find regions of connected pixels that have the same color.



Input: scanned image
Output: number of red and blue states

Connected components application: Image Processing

Goal. Read in a 2D color image and find regions of connected pixels that have the same color.

Efficient algorithm.

- Connect each pixel to neighboring pixel if same color.
- Find connected components in resulting graph.

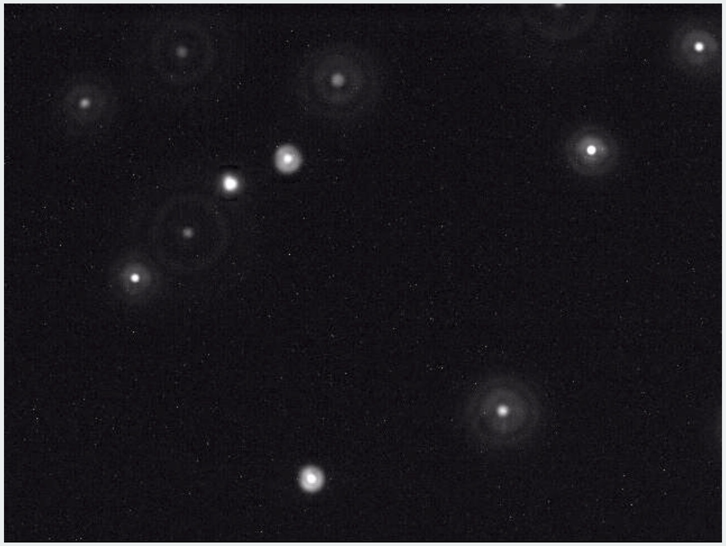
	1	1	1	1	1	6	6	8	9	9	11
			1	6	6	6	8	8	11	9	11
3			1	6	6	4	8	11	11	11	11
3			1	1	6	2	11	11	11	11	11
10	10	10	10	1	1	2	11	11	11	11	11
7	7	2	2	2	2	2	11	11	11	11	11
7	7				2	2	11	11	11	11	11

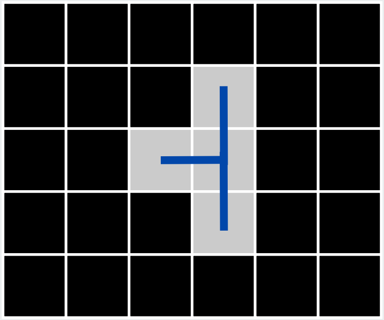
Connected components application: Particle detection

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





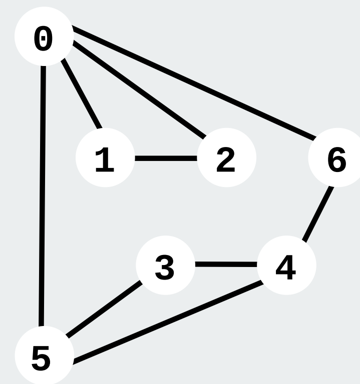
Particle tracking. Track moving particles over time.

- Graph API
- maze exploration
- depth-first search
- breadth-first search
- connected components
- **challenges**

Graph-processing challenge 4:

Problem: Find a path from s to t

Assumptions: any path will do




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

Which is faster, DFS or BFS?

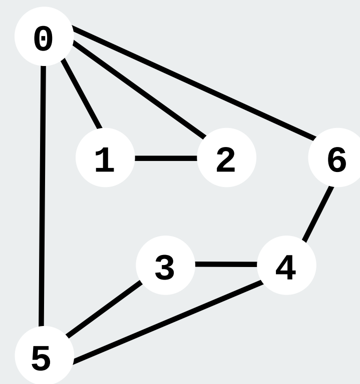
1) DFS
2) BFS
3) about the same
4) depends on the graph
5) depends on the graph representation

Graph-processing challenge 5:

Problem: Find a path from s to t

Assumptions: any path will do

randomized iterators



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

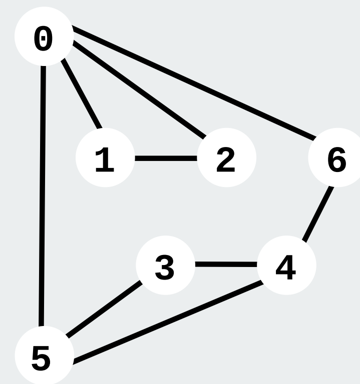
Which is faster, DFS or BFS?

1) DFS
2) BFS
3) about the same
4) depends on the graph
5) depends on the graph representation

Graph-processing challenge 6:

Problem: Find a path from s to t that uses every edge

Assumptions: need to use each edge exactly once




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

How difficult?

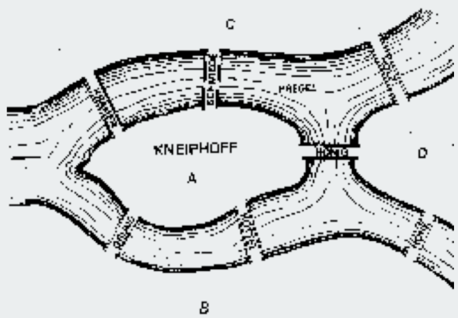
1) any CS126 student could do it
2) need to be a typical diligent CS226 student
3) hire an expert
4) intractable
5) no one knows

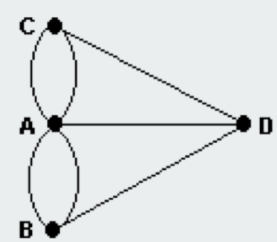
Bridges of Königsberg

earliest application of graph theory or topology

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..."



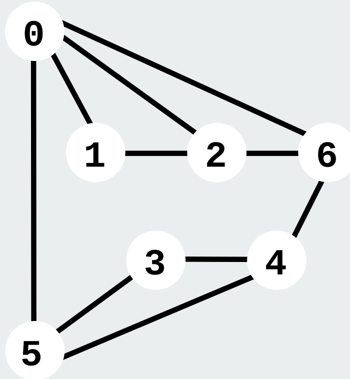


Euler tour. Is there a cyclic path that uses each edge exactly once?
Answer. Yes iff connected and all vertices have even degree.
Tricky DFS-based algorithm to find path (see Algs in Java).

Graph-processing challenge 7:

Problem: Find a path from s to t that visits every vertex

Assumptions: need to visit each vertex exactly once



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

How difficult?

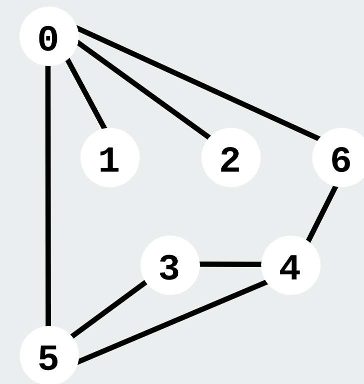
1) any CS126 student could do it
2) need to be a typical diligent CS226 student
3) hire an expert
4) intractable
5) no one knows

Graph-processing challenge 8:

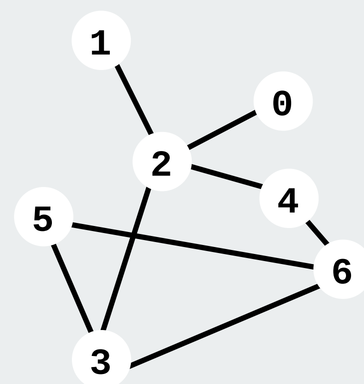
Problem: Are two graphs identical except for vertex names?

How difficult?

1) any CS126 student could do it
2) need to be a typical diligent CS226 student
3) hire an expert
4) intractable
5) no one knows



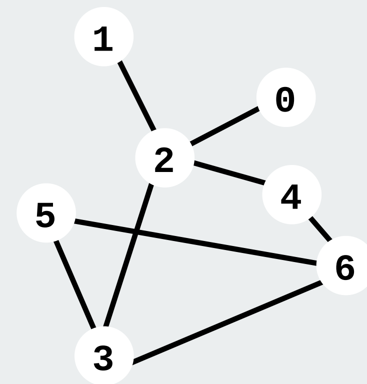
0-1
0-6
0-2
4-3
5-3
5-4
0-5
6-4



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

Graph-processing challenge 9:

Problem: Can you lay out a graph in the plane without crossing edges?




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

How difficult?

1) any CS126 student could do it
2) need to be a typical diligent CS226 student
3) hire an expert
4) intractable
5) no one knows