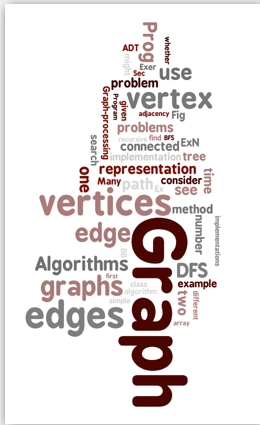
4.1 Undirected Graphs



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

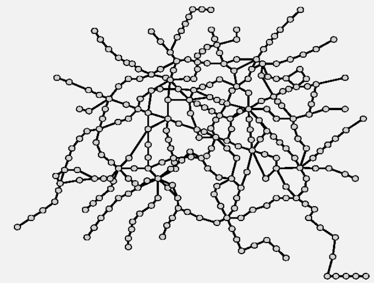
References: Algorithms in Java (Part 5), 3rd edition, Chapters 17 and 18

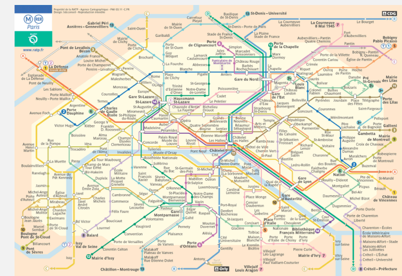
Undirected graphs

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.

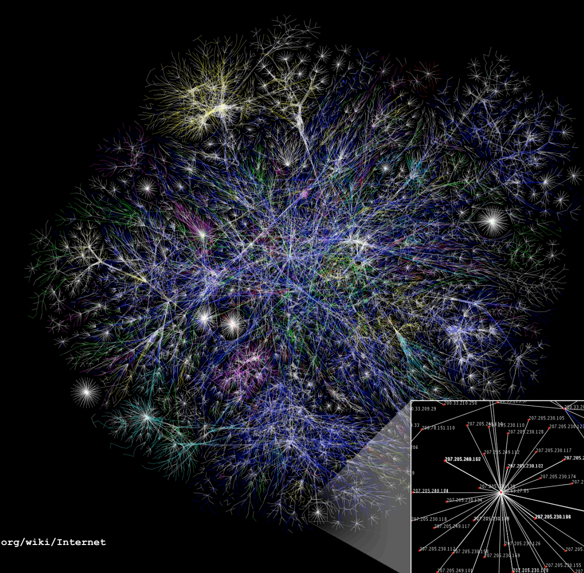




Protein interaction network

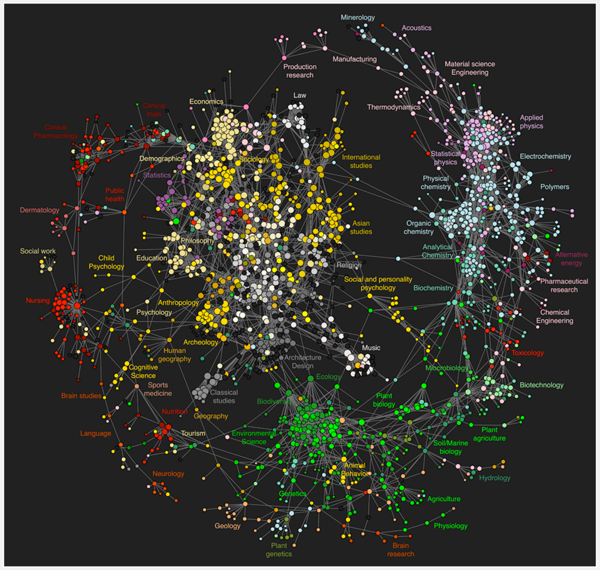
Reference: Jeong et al, Nature Review | Genetics

The Internet as mapped by the Opte Project



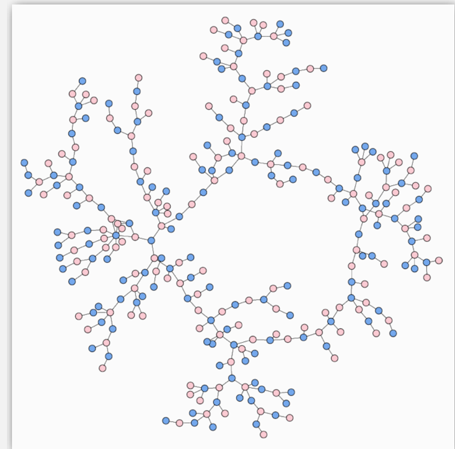
http://en.wikipedia.org/wiki/Internet

Map of science clickstreams



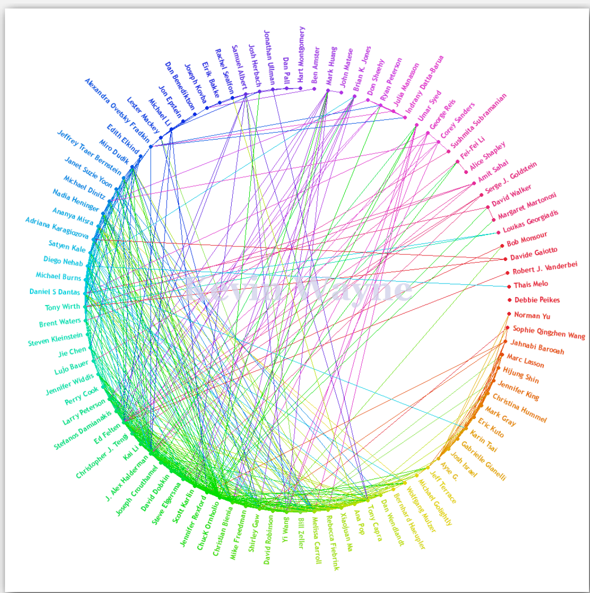
http://www.plosone.org/article/info:doi/10.1371/journal.pone.0004803

High-school dating

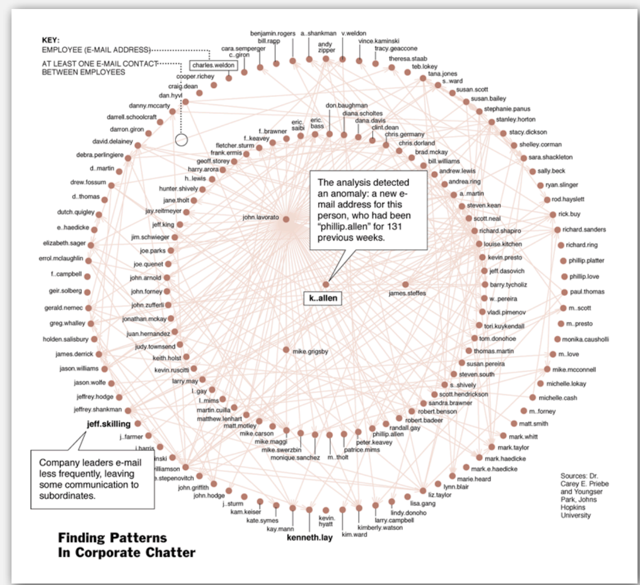


Reference: Bearman, Moody and Stovel, 2004
image by Mark Newman

Kevin's facebook friends (Princeton network)



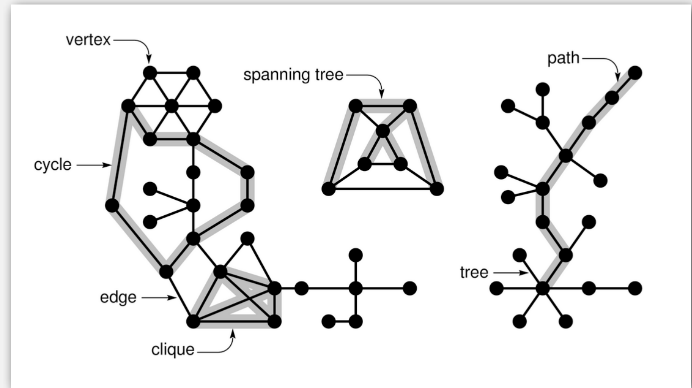
One week of Enron emails



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



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 matrices represent the same graph?

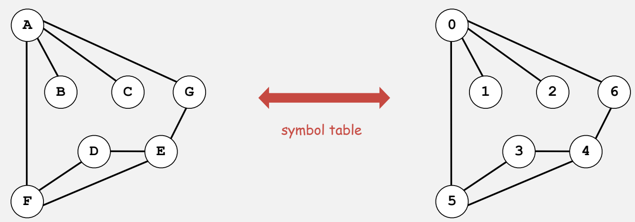
Challenge. Which of these problems are easy? difficult? intractable?

- **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.
- Applications: convert between names and integers with symbol table.



Issues. Parallel edges, self-loops.

Graph API

`public class Graph`		*graph data type*
	`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)`	*return an iterator over the neighbors of v*
`int`	`V()`	*return number of vertices*

```
In in = new In();
Graph G = new Graph(in);
```

← read graph from standard input

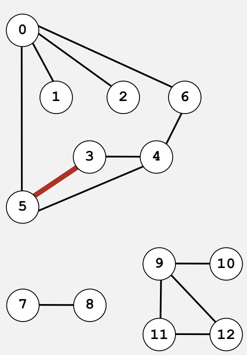
```
for (int v = 0; v < G.V(); v++)
   for (int w : G.adj(v))
      /* process edge v-w */
```

← processes both v-w and w-v

```
% more tiny.txt
7
0 1
0 2
0 5
0 6
3 4
3 5
4 6
```


Set of edges representation

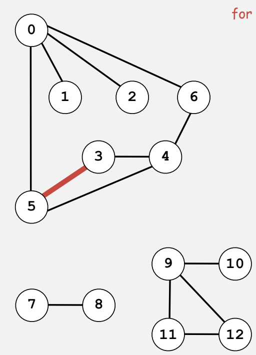
Maintain a list of the edges (linked list or array).




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


Adjacency-matrix 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	0	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	0	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 representation: Java implementation

```
public class Graph
{
   private final int V;
   private final 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

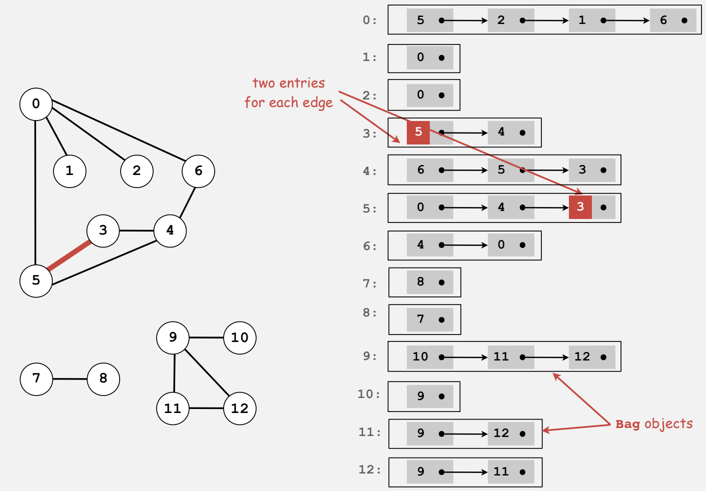
create empty graph with V vertices

add edge v-w (no parallel edges)

iterator for v's neighbors (code for `AdjIterator` omitted)

Adjacency-list representation

Maintain vertex-indexed array of lists (use **Bag** abstraction)



Adjacency-list representation: Java implementation

```
public class Graph
{
   private final int V;
   private final 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` ADT)

create empty graph with V vertices

add edge v-w (parallel edges allowed)

iterator for v's neighbors

Graph representations

In practice. Use adjacency-set (or adjacency-list) representation.

- Algorithms based on iterating over edges incident to v.
- Real-world graphs tend to be "sparse."

huge number of vertices, small average vertex degree

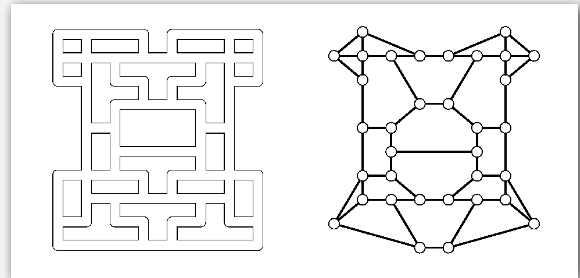
representation	space	insert edge	edge between v and w?	iterate over edges incident to v?
list of edges	E	E	E	E
adjacency matrix	V^2	1	1	V
adjacency list	E + V	1 *	degree(v)	degree(v)
adjacency set	E + V	log (degree(v))	log (degree(v))	degree(v)

used in IntroJava

* only if parallel edges allowed

Maze exploration

Maze graphs.
- Vertex = intersection.
- Edge = passage.

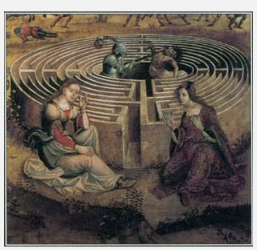


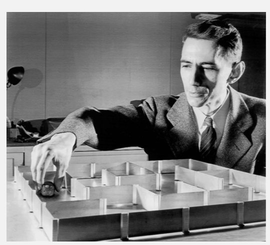
Goal. Explore every passage in the maze.

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.

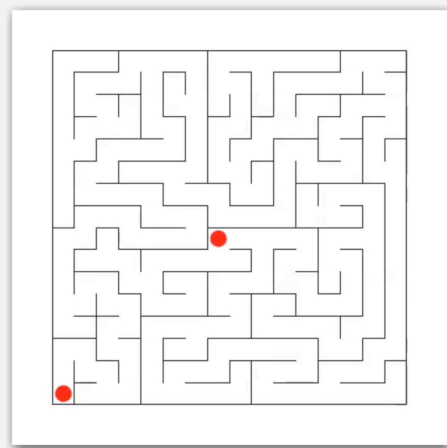
First use? Theseus entered labyrinth to kill the monstrous Minotaur; Ariadne held ball of string.



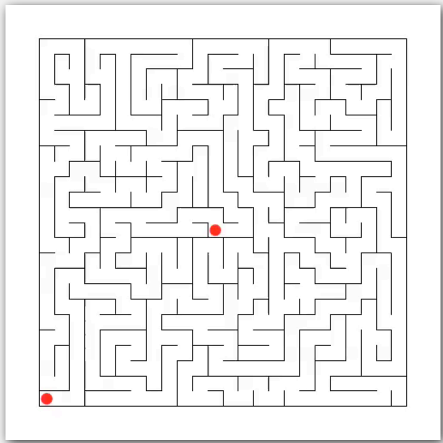


Claude Shannon (with Theseus mouse)

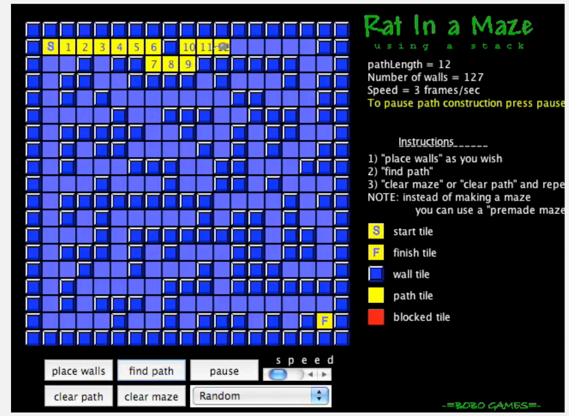
Maze exploration



Maze exploration



Rat in a maze



Depth-first search

Goal. Systematically search through a graph.

Idea. Mimic maze exploration.

DFS (to visit a vertex s)

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

Challenge.

- Masks a complex recursive process.
- [stay tuned]

Typical applications.

- Find all vertices connected to a given `s`.
- Find a path from `s` to `t`.

Design pattern for graph processing

Design goal. Decouple graph data type from graph processing.

```
// print all vertices connected to s
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))
      StdOut.println(v);
```

Typical client program.

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

Depth-first search (warmup)

Goal. Find all vertices connected to a given `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 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 isConnected(int v)
   {  return marked[v];  }
}
```

- `private boolean[] marked;` ← true if connected to s
- constructor ← constructor marks vertices connected to s
- `dfs` ← recursive DFS does the work
- `isConnected` ← client can ask whether any vertex is connected to s

Depth-first search (warmup) equivalent alternate version

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

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

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

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

- `private boolean[] marked;` ← true if connected to s
- constructor ← constructor marks vertices connected to s
- `dfs` ← recursive DFS does the work
- `isConnected` ← client can ask whether any vertex is connected to s

Flood fill

Photoshop "magic wand"

Graph-processing challenge 1

Problem. Flood fill.

Assumptions. Picture has millions to billions of pixels.

How difficult?

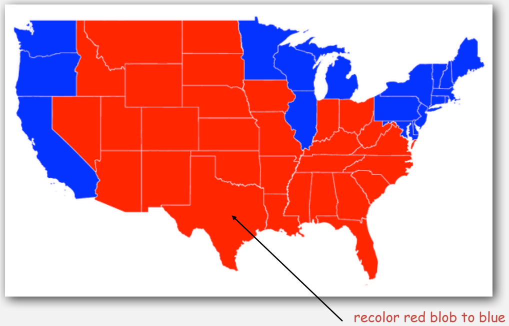
- Any COS 126 student could do it.
- Need to be a typical diligent COS 226 student.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

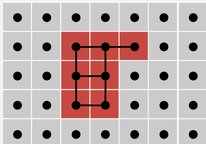
Connectivity application: flood fill

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.



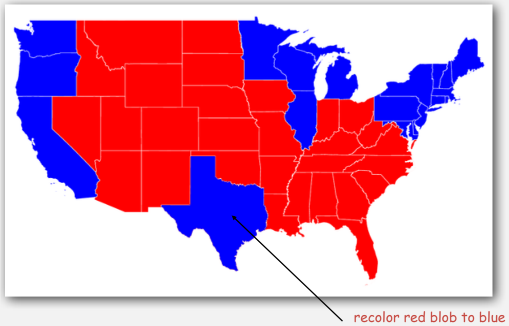


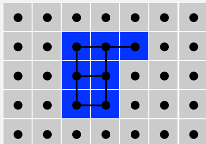
Connectivity application: flood fill

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

Build a grid graph.

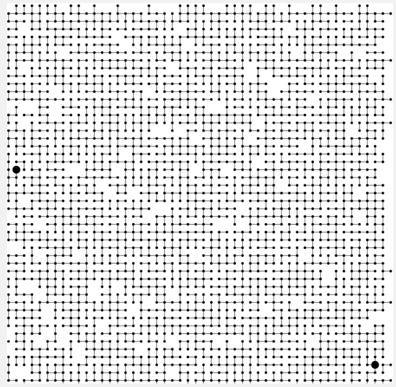
- Vertex: pixel.
- Edge: between two adjacent red pixels.
- Blob: all pixels connected to given pixel.





Graph-processing challenge 2

Problem. Is there a path from s to t ?



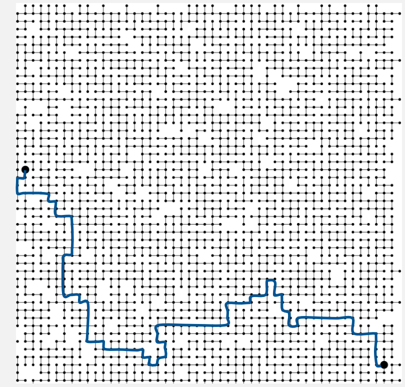
How difficult?

- Any COS 126 student could do it.
- Need to be a typical diligent COS 226 student.
- Hire an expert.
- Intractable.
- No one knows.

Graph-processing challenge 2A

Problem. Find a path from s to t ?

Assumption. Any path will do.



How difficult?

- Any COS 126 student could do it.
- Need to be a typical diligent COS 226 student.
- Hire an expert.
- Intractable.
- No one knows.

Paths in graphs: union find vs. DFS

Goal. Is there 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

† amortized

If so, find one.

- Union-find: not much help (run DFS on connected subgraph).
- DFS: easy (see next slides).

Union-find advantage. Can intermix queries and edge insertions.

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

Depth-first search (pathfinding)

Goal. Find paths to all vertices connected to a given `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 structure

- **`Integer[] edgeTo`** instead of **`boolean[] marked`**
- **`edgeTo[w] == null`** means that **`w`** has not yet been visited
- **`edgeTo[w] == v`** means that edge **`v-w`** was taken to visit **`v`** the first time

Depth-first-search (pathfinding)

```
public class PathfinderDFS
{
   private Integer[] edgeTo;

   public PathfinderDFS(Graph G, int s)
   {
      edgeTo = new Integer[G.V()];
      edgeTo[s] = s;
      dfs(G, s);
   }
   private void dfs(Graph G, int v)
   {
      for (int w : G.adj(v))
         if (edgeTo[w] == null)
         {
            edgeTo[w] = v;
            dfs(G, w);
         }
   }

   public Iterable<Integer> pathTo(int v)
   // Stay tuned.
}
```

replace **marked[]** with instance variable for parent-link representation of DFS tree

initialize it in the constructor with Integer, all values are initially null

not yet visited

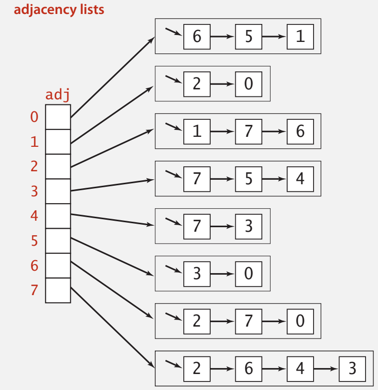
set parent link

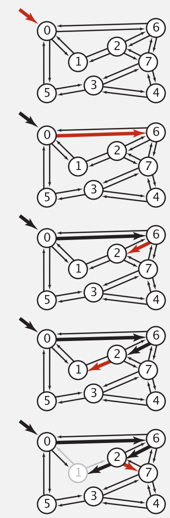
add method for client to iterate through path

DFS pathfinding trace

a way to truly understand maze exploration

adjacency lists

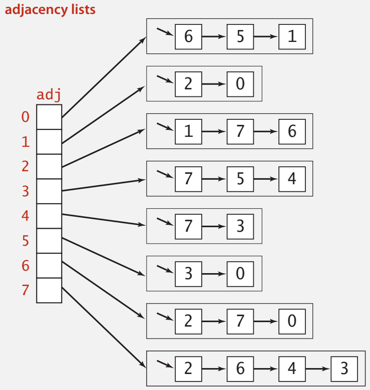


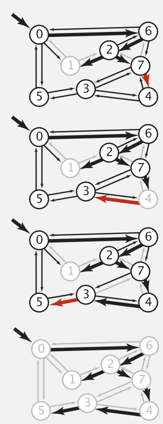


DFS pathfinding trace

a way to truly understand maze exploration

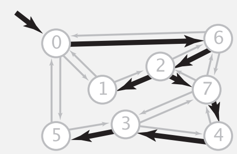
adjacency lists





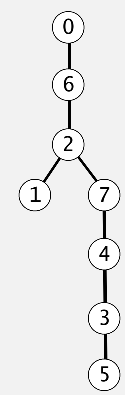
Depth-first-search (pathfinding iterator)

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



edgeTo[v]	0	2	6	4	7	3	0	2
v	0	1	2	3	4	5	6	7

```
public Iterable<Integer> pathTo(int v)
{
   Stack<Integer> path = new Stack<Integer>();
   path.push(v)
   while (v != edgeTo[v])
   {
      v = edgeTo[v];
      path.push(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).

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.*

Property. BFS examines vertices in increasing distance from s.

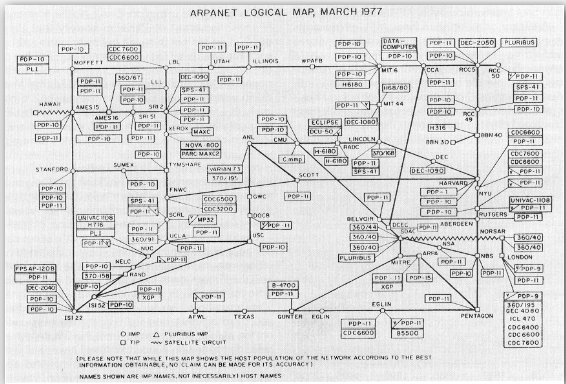
Breadth-first-search (pathfinding)

```
public class PathfinderBFS
{
   private Integer[] edgeTo;

   public PathfinderBFS(Graph G, int s)
   {
      edgeTo = new Integer[G.V()];
      edgeTo[s] = s;
      bfs(G, s);
   }
   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 (edgeTo[w] == null)
            {
               q.enqueue(w);
               edgeTo[w] = v;
            }
      }
   }
}
```

same setup as DFS

BFS application

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



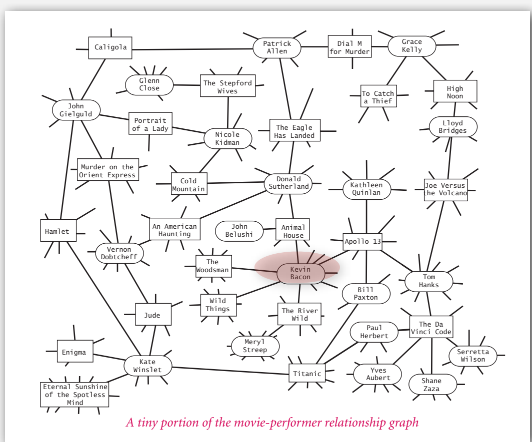
ARPANET

BFS application

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

Kevin Bacon graph

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

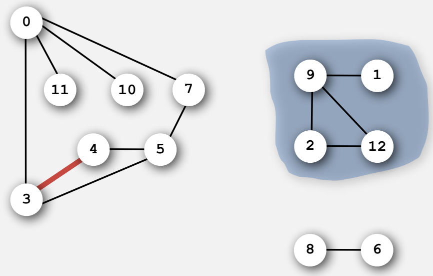


A tiny portion of the movie-performer relationship graph

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

Union-Find? Not quite.

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.

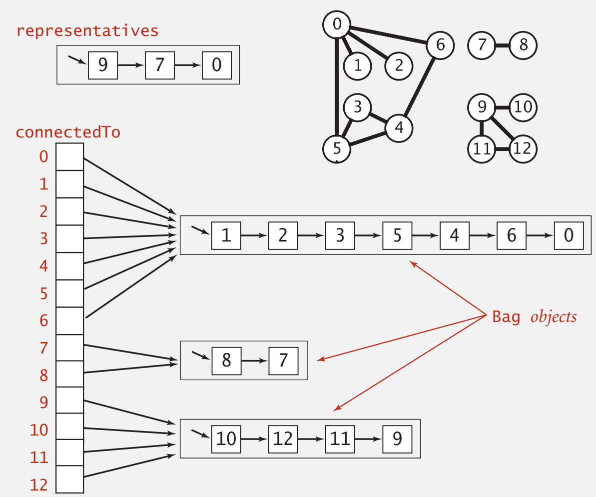
preprocess time	query time	extra space
E + V	1	V

Finding connected components with DFS

```
public class CCfinder
{
   private Graph G;
   private Bag<Integer>[] connectedTo;
   private Bag<Integer> representatives;

   public CCfinder(Graph G)
   {
      this.G = G;
      representatives = new Bag<Integer>();
      connectedTo = (Bag<Integer>[]) new Bag[G.V()];
      for (int s = 0; s < G.V(); s++)
         if (connected[s] == null)
         {
            Bag<Integer> bag = new Bag<Integer>();
            representatives.add(s);
            dfs(s, s, bag);
         }
   }
```

- vertices connected to each vertex (`connectedTo`)
- one vertex from each component (`representatives`)
- has s been visited? (`if (connected[s] == null)`)
- if not, create a bag for all vertices connected to s
- s represents them all (`representatives.add(s);`)
- call recursive method (`dfs(s, s, bag);`)

Finding connected components with DFS (data structures)

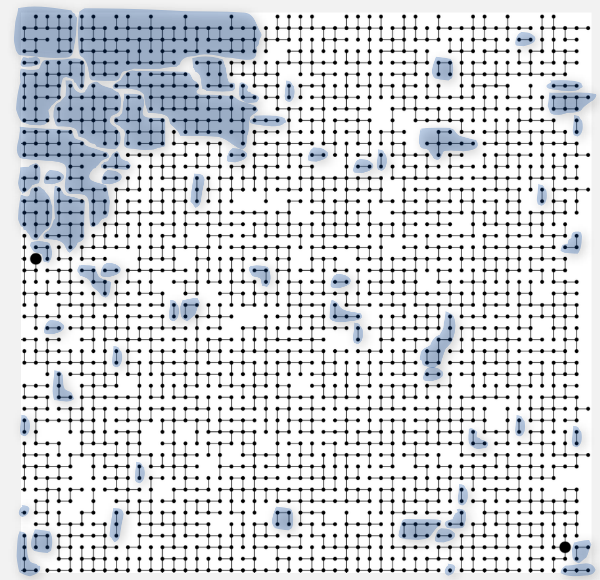


Finding connected components with DFS (continued)

```
private void dfs(int v, int s, Bag bag)
{
   connectedTo[v] = bag;
   bag.add(v);
   for (int w : G.adj(v))
      if (connectedTo[w] == null)
          dfs(w, s, bag);
}

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

public Iterable<Integer> representatives()
{  return representatives;  }

public Iterable<Integer> connectedTo(int v)
{  return connectedTo[v];  }
```

Tricky: object equality!

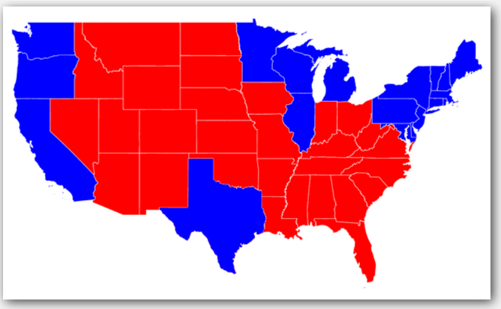
Connected components



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.

assuming contiguous 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.

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

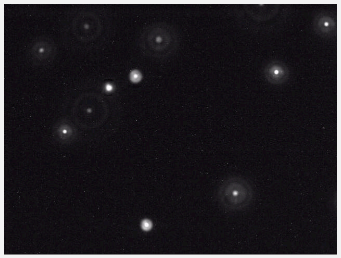
0	1	1	1	1	1	6	6	8	9	9	11
0	0	0	1	6	6	6	8	8	11	9	11
3	0	0	1	6	6	4	8	11	11	11	11
3	0	0	1	1	6	2	11	11	11	11	11
10	10	10	10	1	1	2	11	11	11	11	11
7	7	2	2	2	2	2	11	11	11	11	11
7	7	5	5	5	2	2	11	11	11	11	11

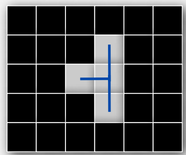
Connected components application: particle detection

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





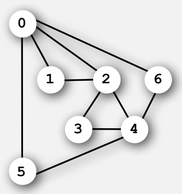
Particle tracking. Track moving particles over time.

Graph-processing challenge 3

Problem. Find a cycle that uses every edge.
Assumption. Need to use each edge exactly once.

How difficult?

- Any COS 126 student could do it.
- Need to be a typical diligent COS 226 student.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.



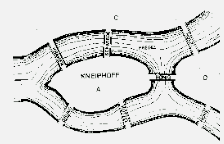
0-1-2-3-4-2-0-6-4-5-0

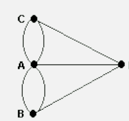
0-1
0-2
0-5
0-6
1-2
2-3
2-4
3-4
4-5
4-6

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."*

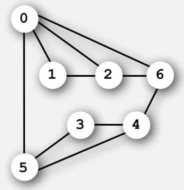




Euler tour. Is there a cyclic path that uses each edge exactly once?
Answer. Yes iff connected and all vertices have even degree.
To find path. DFS-based algorithm (see Algs in Java).

Graph-processing challenge 4

Problem. Find a cycle that visits every vertex.
Assumption. Need to visit each vertex exactly once.



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

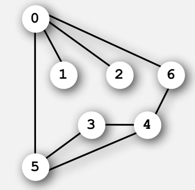
0-5-3-4-6-2-1-0

How difficult?

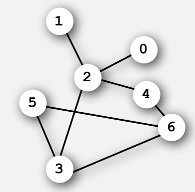
- Any COS 126 student could do it.
- Need to be a typical diligent COS 226 student.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

Graph-processing challenge 5

Problem. Are two graphs identical except for vertex names?



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



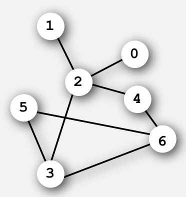
2-1
2-4
2-0
6-5
5-3
3-6
2-3
6-4

How difficult?

- Any COS 126 student could do it.
- Need to be a typical diligent COS 226 student.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

Graph-processing challenge 6

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



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

How difficult?

- Any COS 126 student could do it.
- Need to be a typical diligent COS 226 student.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.