Undirected Graphs

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

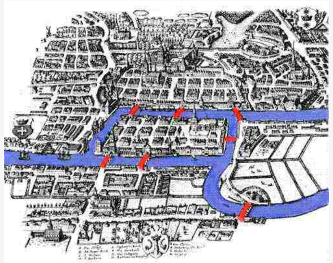
References:
Algorithms in Java, Chapters 17 and 18
`http://www.cs.princeton.edu/algs4/51undirected`

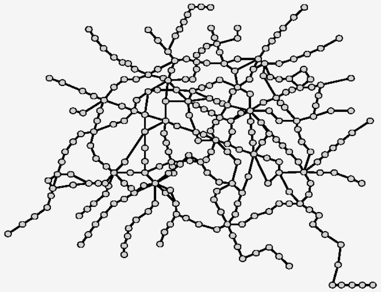
Undirected graphs

Graph. Set of vertices connected pairwise by edges.

Why study graph algorithms?

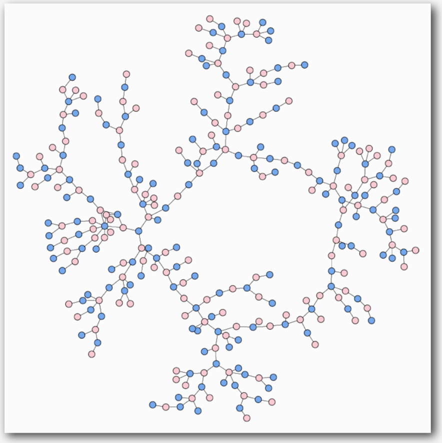
- Interesting and broadly useful abstraction.
- Challenging branch of computer science and discrete math.
- Hundreds of graph algorithms known.
- Thousands of practical applications.





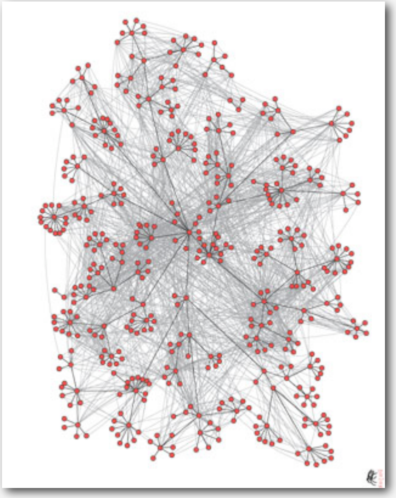
Social networks

high school dating



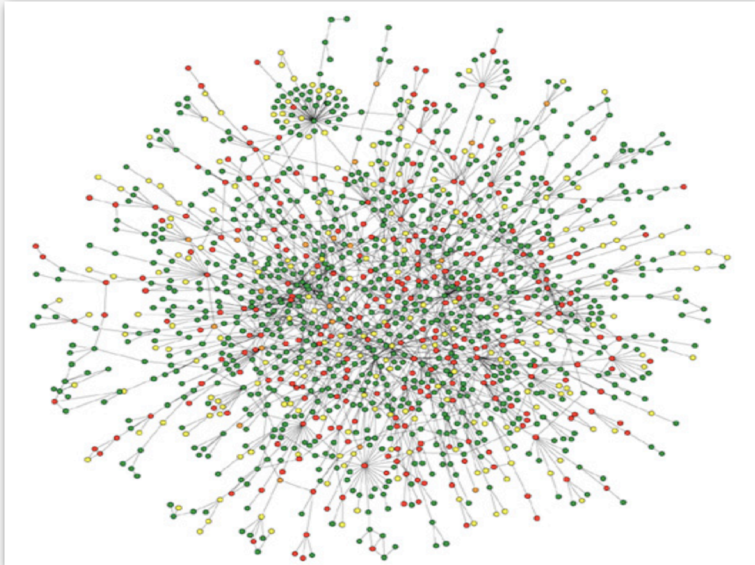
Reference: Bearman, Moody and Stovel, 2004
image by Mark Newman

corporate e-mail



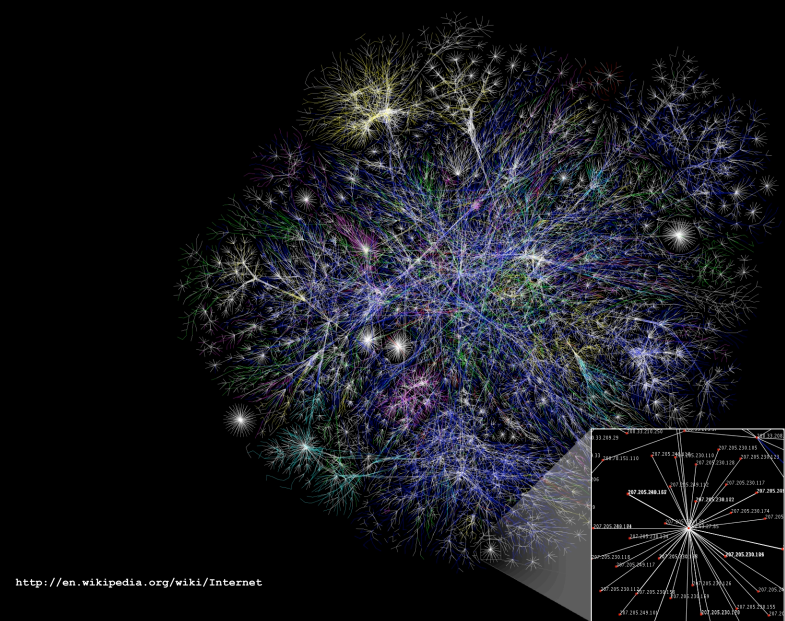
Reference: Adamic and Adar, 2004

Protein interaction network



Reference: Jeong et al, Nature Review | Genetics

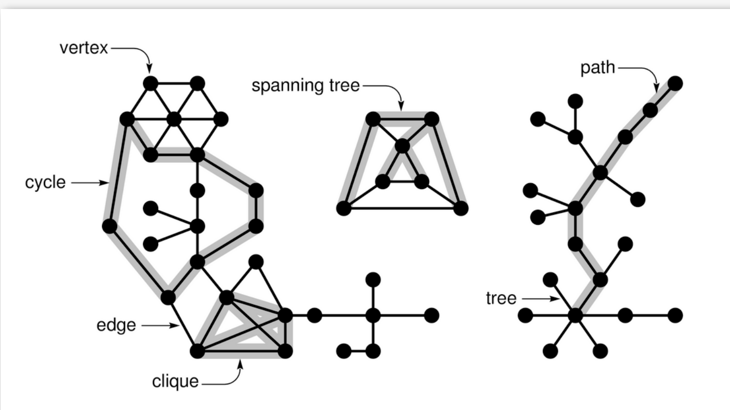
The Internet as mapped by the Opte Project



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?

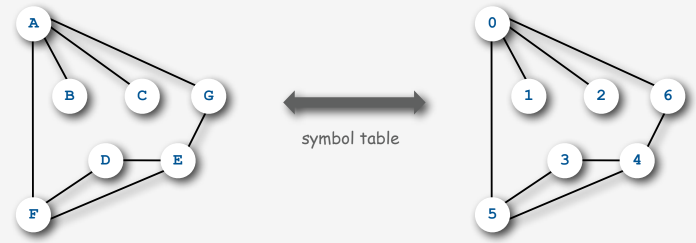
First challenge. Which of these problems are easy? difficult? intractable?

- 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.
- 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(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*
String	toString()	*return a string representation*

```
In in = new In();
Graph G = new Graph(in);
StdOut.println(G);
```

read graph from standard input

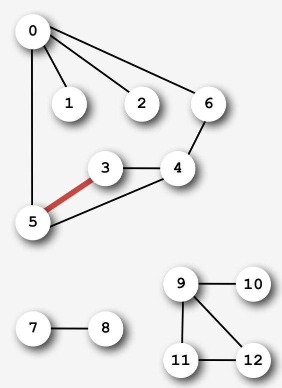
```
for (int v = 0; v < G.V(); v++)
   for (int w : G.adj(v))
      /* process edge v-w */
```

process 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

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

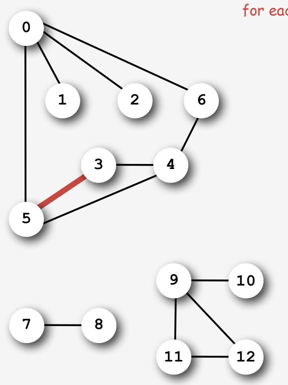



```
0 1
0 2
0 5
0 6
3 4
3 5
4 6
7 8
9 10
9 11
9 12
```

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

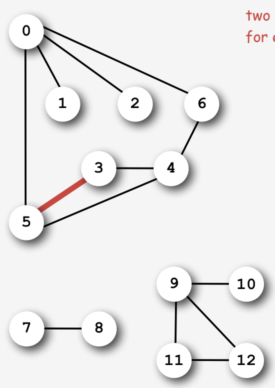
create empty graph with V vertices

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 (implementation omitted).

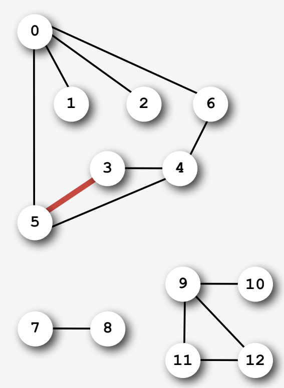


two entries for each edge

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

Adjacency-set graph representation

Maintain vertex-indexed array of sets.



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: { 1, 9 }

two entries for each edge

Adjacency-set representation: Java implementation

```
public class Graph
{
   private final int V;
   private final 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 graph with V vertices

add edge v-w (no parallel edges)

iterator for v's neighbors

Graph representations

Graphs are abstract mathematical objects, but:
- ADT implementation requires specific representation.
- Efficiency depends on matching algorithms to representations.

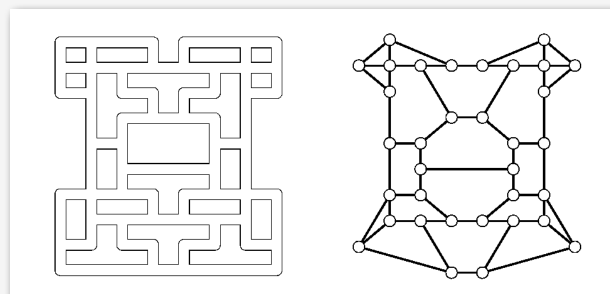
representation	space	edge between v and w?	iterate over edges incident to v?
list of edges	E	E	E
adjacency matrix	V^2	1	V
adjacency list	E + V	degree(v)	degree(v)
adjacency set	E + V	log (degree(v))	degree(v)

In practice. Use adjacency-set (or adjacency-list) representation.
- Algs all based on iterating over edges incident to v.
- Real-world graphs tend to be "sparse." huge number of vertices, small average vertex degree

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

Maze exploration

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



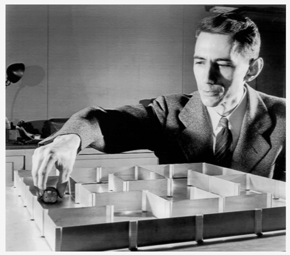
Goal. Explore every passage in the maze.

Trémaux maze exploration

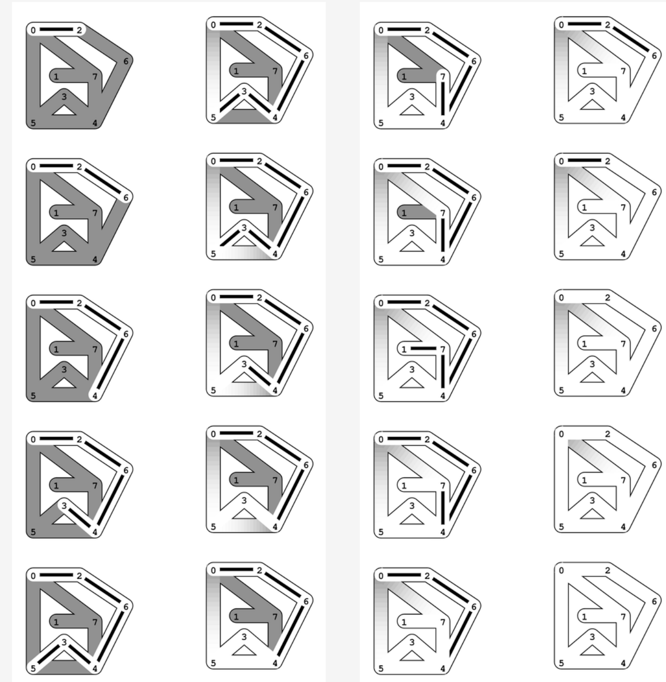
Algorithm.

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

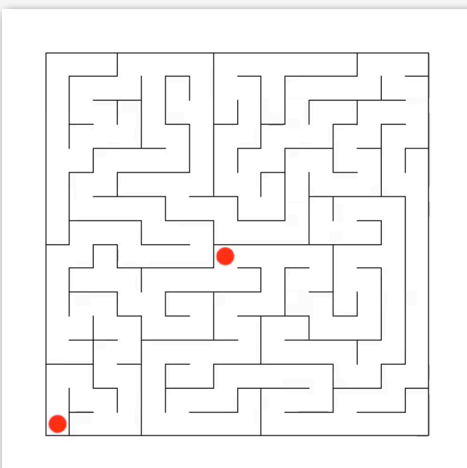
First use? Theseus entered labyrinth to kill the monstrous Minotaur; Ariadne held ball of string.



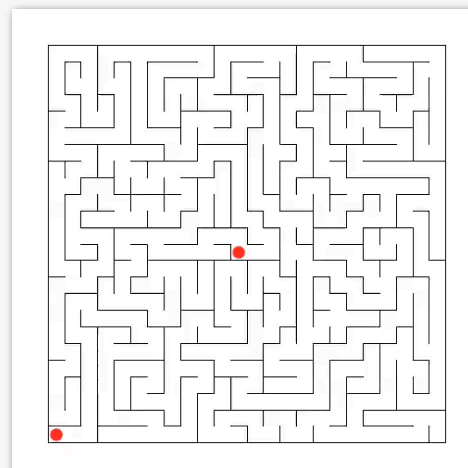
Claude Shannon (with Theseus mouse)



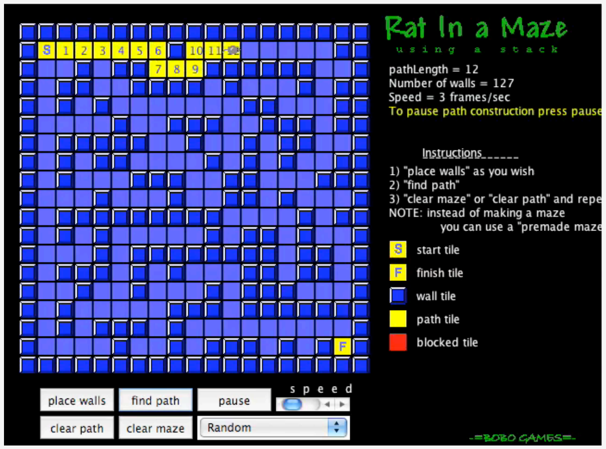
Maze exploration



Maze exploration



Rat in a maze



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

Depth-first search

Goal. Systematically search through a graph.

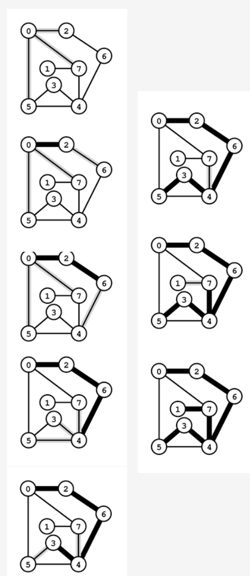
Idea. Mimic maze exploration.

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

Running time.

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

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



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 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))
         StdOut.println(v);
}
```

Design goal. 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 isConnected(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

Flood fill

Photoshop "magic wand"

Graph-processing challenge 1

Problem. Flood fill.
Assumptions. Picture has millions to billions of pixels.

How difficult?

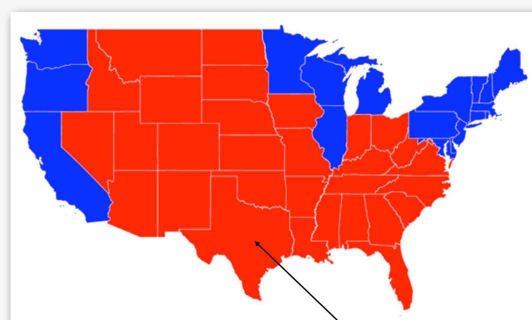
- Any COS 126 student could do it.
- Need to be a typical diligent COS 226 student.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

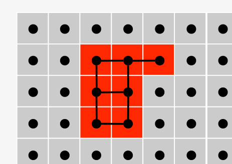
Connectivity application: flood fill

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.





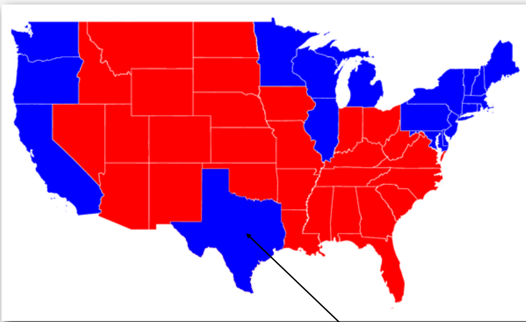
recolor red blob to blue

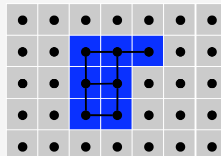
Connectivity application: flood fill

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.

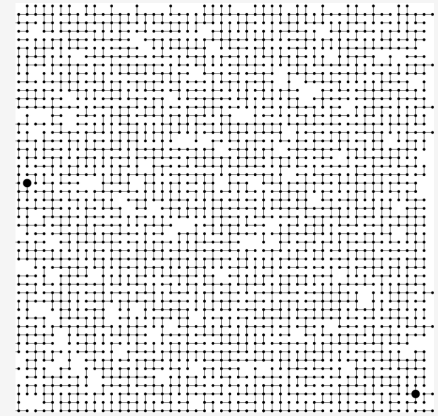




recolor red blob to blue

Graph-processing challenge 2

Problem. Is there a path from s to t ?



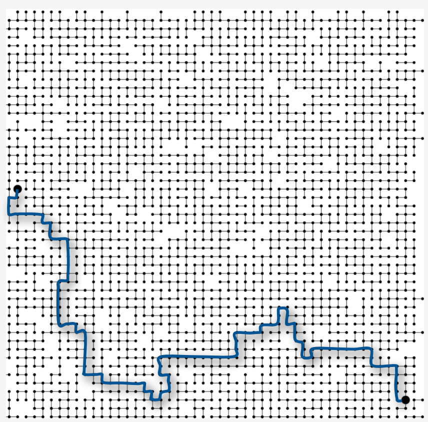
How difficult?

- 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 3

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

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 (stay tuned).

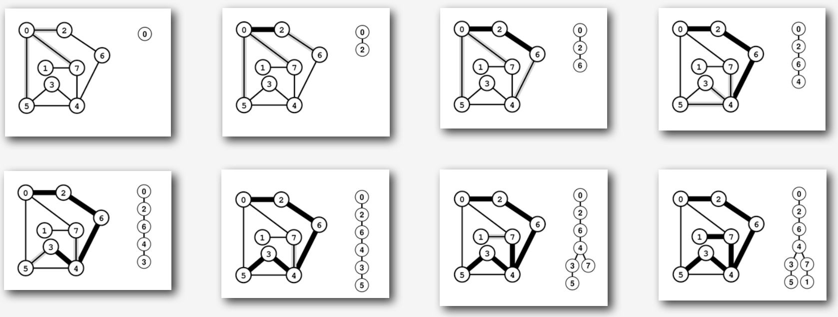
UF advantage. Can intermix queries and edge insertions.

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)
   {  /* see next slide */  }
}
```

add instance variable for parent-link representation of DFS tree

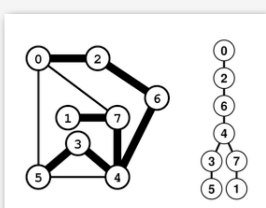
initialize it in the constructor

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
- challenge

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

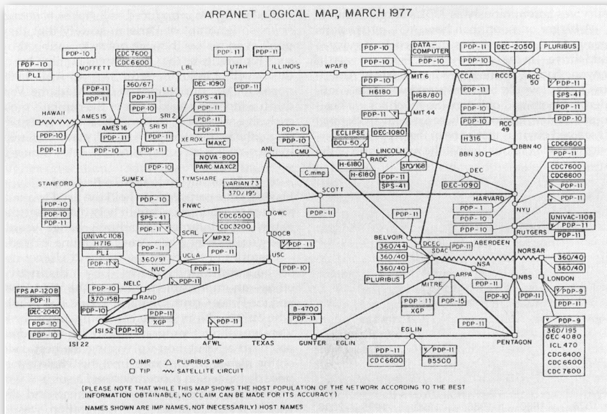
```
public class BFSearcher
{
   private int[] dist;                         // distances from s

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

      bfs(G, s);                               // compute distances
   }

   public int distance(int v)
   {  return dist[v];  }                       // answer client query

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

}
```


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;
         }
      }
   }
}
```

BFS application

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



ARPANET

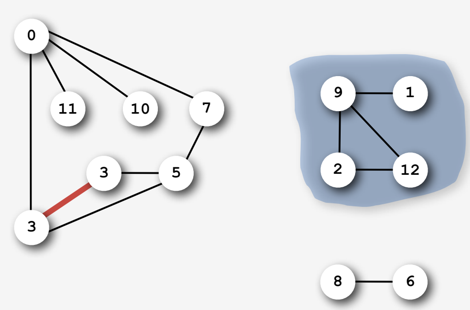
- graph API
- maze exploration
- depth-first search
- breadth-first search
- **connected components**
- challenge

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 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)
   {  /* see next slide */  }

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

}
```

component labels

constant-time connectivity query

Depth-first search for connected components

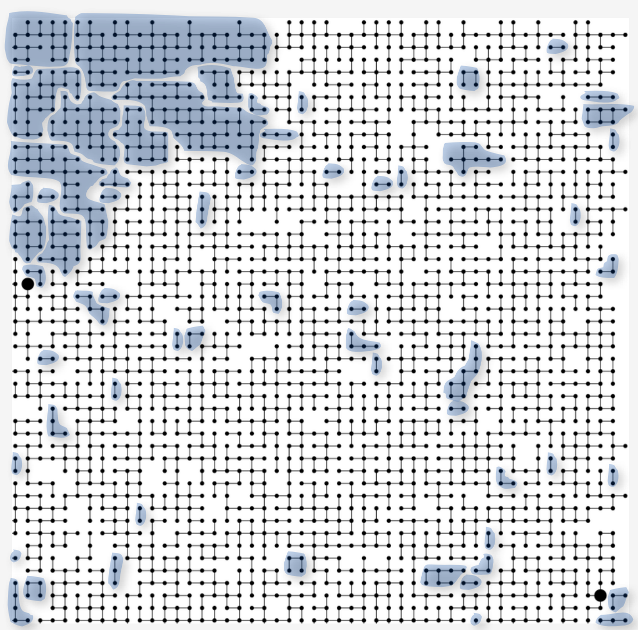
```
public CCFinder(Graph G)
{
   cc = new int[G.V()];
   for (int v = 0; v < G.V(); v++)
      cc[v] = UNMARKED;
   for (int v = 0; v < G.V(); v++)
      if (cc[v] == UNMARKED)
      {
         dfs(G, v);
         components++;
      }
}
```

DFS for each component

```
private void dfs(Graph G, int v)
{
   cc[v] = components;
   for (int w : G.adj(v))
      if (cc[w] == UNMARKED) dfs(G, w);
}
```

standard DFS

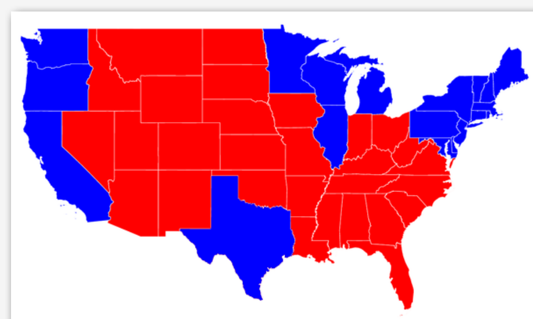
Connected components



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.

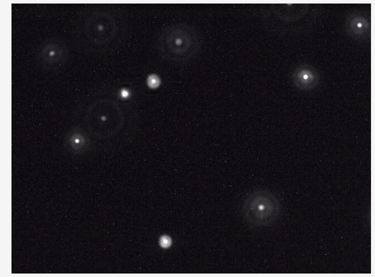
Efficient algorithm.
- Create grid graph.
- Connect each pixel to neighboring pixel if same color.
- Find connected components in resulting graph.

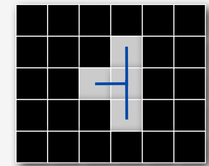
0	1	1	1	1	1	6	6	8	9	9	11
0	0	0	1	6	6	6	8	8	11	9	11
3	0	0	1	6	6	4	8	11	11	11	11
3	0	0	1	1	6	2	11	11	11	11	11
10	10	10	10	1	1	2	11	11	11	11	11
7	7	2	2	2	2	2	11	11	11	11	11
7	7	5	5	5	2	2	11	11	11	11	11

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



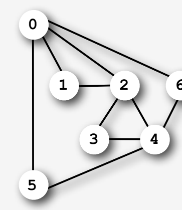


Particle tracking. Track moving particles over time.

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

Graph-processing challenge 4

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



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

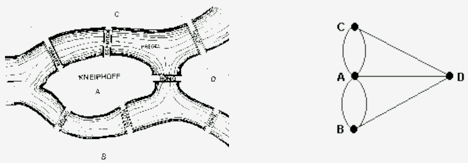
0-1
0-2
0-5
0-6
1-2
2-3
2-4
3-4
4-5
4-6

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.

Bridges of Königsberg

The Seven Bridges of Königsberg. [Leonhard Euler 1736]

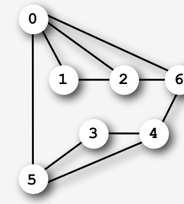
> *" ... in Königsberg in Prussia, there is an island A, called the Kneiphof; the river which surrounds it is divided into two branches ... and these branches are crossed by seven bridges. Concerning these bridges, it was asked whether anyone could arrange a route in such a way that he could cross each 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 5

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



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

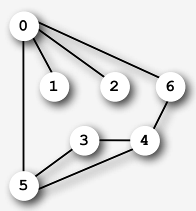
0-1
0-2
0-5
0-6
1-2
2-6
3-4
3-5
4-5
4-6

How difficult?

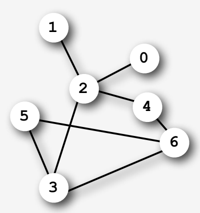
- Any COS 126 student could do it.
- Need to be a typical diligent COS 226 student.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

Graph-processing challenge 6

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



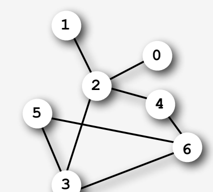
2-1
2-4
2-0
6-5
5-3
3-6
2-3
6-4

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 7

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.