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

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

high school dating

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

corporate e-mail

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

vertex

spanning tree

cycle

edge

clique

path

tree
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)
Maintain a two-dimensional $v \times v$ boolean array.

For each edge $v$–$w$ in graph: $adj[v][w] = adj[w][v] = true$. 

Adjacency matrix representation

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Two entries for each edge.
Adjacency-matrix graph representation: Java implementation

```java
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**: Represents the graph.
- **Create empty V-vertex graph**: initializes the graph.
- **Add edge v-w (no parallel edges)**: adds an edge between vertices v and w.
- **Iterator for v's neighbors**: returns an iterator for the neighbors of vertex v.
Adjacency matrix: iterator for vertex neighbors

```java
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)

- Each vertex has a list of its adjacent vertices.
- Two entries for each edge.
Adjacency-SET graph representation

Maintain vertex-indexed array of SETs
(take advantage of balanced-tree or hashing implementations)
Adjacency-SET graph representation: Java implementation

```java
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];
    }
}
```
Graph representations

Graphs are abstract mathematical objects, BUT

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

<table>
<thead>
<tr>
<th>representation</th>
<th>space</th>
<th>edge between v and w?</th>
<th>iterate over edges incident to v?</th>
</tr>
</thead>
<tbody>
<tr>
<td>list of edges</td>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>adjacency matrix</td>
<td>V²</td>
<td>1</td>
<td>V</td>
</tr>
<tr>
<td>adjacency list</td>
<td>E + V</td>
<td>degree(v)</td>
<td>degree(v)</td>
</tr>
<tr>
<td>adjacency SET</td>
<td>E + V</td>
<td>log (degree(v))</td>
<td>degree(v)*</td>
</tr>
</tbody>
</table>

* 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.
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 \( s \).

**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

```java
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.
public class DFSSearcher
{
    private boolean[] marked;

    public DFSSearcher(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];
    }
}
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.

recolored red blob to blue
Graph-processing challenge 2:

**Problem:** Is there a path from $s$ to $t$?

**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
Problem: Find a path from $s$ to $t$.
Assumptions: any path will do

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 $t$?

<table>
<thead>
<tr>
<th>method</th>
<th>preprocess time</th>
<th>query time</th>
<th>space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Union Find</td>
<td>$V + E \log^* V$</td>
<td>$\log^* V \dagger$</td>
<td>$V$</td>
</tr>
<tr>
<td>DFS</td>
<td>$E + V$</td>
<td>$1$</td>
<td>$E + V$</td>
</tr>
</tbody>
</table>

$\dagger$ 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 \( \text{pred}[v] \) (parent-link representation).

**Retrace path.** To find path between \( s \) and \( v \), follow \( \text{pred} \) back from \( v \).
public class DFSSearcher
{
    private int[] pred;
    public DFSSearcher(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 } 
}
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

```java
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.
    }
}
```
Breadth-first search (compute shortest-path distances)

```java
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

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

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>2</td>
</tr>
<tr>
<td>H</td>
<td>0</td>
</tr>
<tr>
<td>I</td>
<td>2</td>
</tr>
<tr>
<td>J</td>
<td>1</td>
</tr>
<tr>
<td>K</td>
<td>0</td>
</tr>
<tr>
<td>L</td>
<td>0</td>
</tr>
<tr>
<td>M</td>
<td>1</td>
</tr>
</tbody>
</table>

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.

---

<table>
<thead>
<tr>
<th>preprocess Time</th>
<th>query Time</th>
<th>extra Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E + V$</td>
<td>1</td>
<td>$V$</td>
</tr>
</tbody>
</table>
Depth-first search for connected components

```java
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];
    }
}
```
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
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.
Particle detection. Given grayscale image of particles, identify "blobs."
- Vertex: pixel.
- Edge: between two adjacent pixels with grayscale value $\geq 70$.
- Blob: connected component of 20-30 pixels.

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

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

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

**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
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

**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
Graph-processing challenge 9:

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

**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