4.1 **Undirected Graphs**

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

Graph. Set of vertices connected pairwise by edges.

Why study graph algorithms?

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

Reference: Jeong et al, Nature Review | Genetics
The Internet as mapped by the Opte Project

http://en.wikipedia.org/wiki/Internet
Map of science clickstreams

http://www.plosone.org/article/info:doi/10.1371/journal.pone.0004803
10 million Facebook friends

"Visualizing Friendships" by Paul Butler
One week of Enron emails

Company leaders e-mail less frequently, leaving some communication to subordinates.

Finding Patterns In Corporate Chatter

The analysis detected an anomaly: a new e-mail address for this person, who had been "phillip.allen" for 131 previous weeks.
The evolution of FCC lobbying coalitions

“The Evolution of FCC Lobbying Coalitions” by Pierre de Vries in JoSS Visualization Symposium 2010
The Spread of Obesity in a Large Social Network Over 32 Years

by Christakis and Fowler in New England Journal of Medicine, 2007

Educational level; the ego's obesity status at the previous time point (t) and most pertinent, the alter's obesity status at times t and t+1.

We used generalized estimating equations to account for multiple observations of the same ego across examinations and across ego–alter pairs.

We assumed an independent working correlation structure for the clusters.

The use of a time-lagged dependent variable (lagged to the previous examination) eliminated serial correlation in the errors (evaluated with a Lagrange multiplier test) and also substantially controlled for the ego's genetic endowment and any intrinsic, stable predisposition to obesity. The use of a lagged independent variable for an alter's weight status controlled for homophily.

The key variable of interest was an alter's obesity at time t+1. A significant coefficient for this variable would suggest either that an alter's weight affected an ego's weight or that an ego and an alter experienced contemporaneous events affecting both their weights. We estimated these models in varied ego–alter pair types.

To evaluate the possibility that omitted variables or unobserved events might explain the associations, we examined how the type or direction of the social relationship between the ego and the alter affected the association between the ego's obesity and the alter's obesity. For example, if unobserved factors drove the association between the ego's obesity and the alter's obesity, then the directionality of friendship should not have been relevant.

We evaluated the role of a possible spread in smoking-cessation behavior as a contributor to the spread of obesity by adding variables for the smoking status of egos and alters at times t and t+1 to the foregoing models. We also analyzed the role of geographic distance between egos and alters by adding such a variable.

We calculated 95% confidence intervals by simulating the first difference in the alter's contemporaneous weight status.

Figure 1. Largest Connected Subcomponent of the Social Network in the Framingham Heart Study in the Year 2000.

Each circle (node) represents one person in the data set. There are 2200 persons in this subcomponent of the social network. Circles with red borders denote women, and circles with blue borders denote men. The size of each circle is proportional to the person's body-mass index. The interior color of the circles indicates the person's obesity status: yellow denotes an obese person (body-mass index, ≥30) and green denotes a nonobese person. The colors of the ties between the nodes indicate the relationship between them: purple denotes a friendship or marital tie and orange denotes a familial tie.
# Graph applications

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

**Path.** Sequence of vertices connected by edges.

**Cycle.** Path whose first and last vertices are the same.

Two vertices are *connected* if there is a path between them.
Some graph-processing problems

Path. Is there a path between $s$ and $t$?
Shortest path. What is the shortest path between $s$ and $t$?

Cycle. Is there a cycle in the graph?
Euler tour. Is there a cycle that uses each edge exactly once?
Hamilton tour. Is there a cycle that uses each vertex exactly once?

Connectivity. Is there a way to connect all of the vertices?
MST. What is the best way to connect all of the vertices?
Biconnectivity. Is there a vertex whose removal disconnects the graph?

Planarity. Can you draw the graph in the plane with no crossing edges?
Graph isomorphism. Do two adjacency lists represent the same graph?

Challenge. Which of these problems are easy? difficult? intractable?
› graph API
› depth-first search
› breadth-first search
› connected components
› challenges
Graph representation

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

**Caveat.** Intuition can be misleading.

two drawings of the same graph
Graph representation

Vertex representation.
• This lecture: use integers between 0 and $V - 1$.
• Applications: convert between names and integers with symbol table.

Anomalies.
• self-loop
• parallel edges

symbol table
Graph API

```java
public class Graph {

    Graph(int V) { /* create an empty graph with V vertices */ }
    Graph(In in) { /* create a graph from input stream */ }

    void addEdge(int v, int w) { /* add an edge v-w */ }

    Iterable<Integer> adj(int v) { /* vertices adjacent to v */ }

    int V() { /* number of vertices */ }
    int E() { /* number of edges */ }

    String toString() { /* string representation */ }

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

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

Graph input format.

tinyG.txt

% java Test tinyG.txt
0-6
0-2
0-1
0-5
1-0
2-0
3-5
3-4
...
12-11
12-9

In in = new In(args[0]);
Graph G = new Graph(in);
for (int v = 0; v < G.V(); v++)
   for (int w : G.adj(v))
      StdOut.println(v + "-" + w);
Typical graph-processing code

**compute the degree of** $v$

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

**compute maximum degree**

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

**compute average degree**

```java
public static double averageDegree(Graph G) {
    return 2.0 * G.E() / G.V();
}
```

**count self-loops**

```java
public static int numberOfSelfLoops(Graph G) {
    int count = 0;
    for (int v = 0; v < G.V(); v++)
        for (int w : G.adj(v))
            if (v == w) count++;
    return count/2;  // each edge counted twice
}
```
Set-of-edges graph representation

Maintain a list of the edges (linked list or array).
Maintain a two-dimensional $V$-by-$V$ boolean array; for each edge $v \rightarrow w$ in graph: $\text{adj}[v][w] = \text{adj}[w][v] = \text{true}$. 

Adjacency-matrix graph representation
Adjacency-list graph representation

Maintain vertex-indexed array of lists.
Adjacency-list graph representation: Java implementation

```java
public class Graph {
    private final int V;
    private Bag<Integer>[] adj;

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

    public void addEdge(int v, int w) {
        adj[v].add(w);
        adj[w].add(v);
    }

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

In practice. Use adjacency-lists representation.
- Algorithms based on iterating over vertices adjacent to $v$.
- Real-world graphs tend to be **sparse**.

---

Two graphs ($V = 50$)

- **sparse** ($E = 200$)
- **dense** ($E = 1000$)

huge number of vertices, small average vertex degree
**Graph representations**

**In practice.** Use adjacency-lists representation.

- **Algorithms based on iterating over vertices adjacent to** $v$.
- **Real-world graphs tend to be** sparse.

### Graph Representations

<table>
<thead>
<tr>
<th>Representation</th>
<th>Space</th>
<th>Add Edge</th>
<th>edge between v and w?</th>
<th>iterate over vertices adjacent to v?</th>
</tr>
</thead>
<tbody>
<tr>
<td>list of edges</td>
<td>$E$</td>
<td>1</td>
<td>$E$</td>
<td>$E$</td>
</tr>
<tr>
<td>adjacency matrix</td>
<td>$V^2$</td>
<td>$1 ,*$</td>
<td>$1$</td>
<td>$V$</td>
</tr>
<tr>
<td>adjacency lists</td>
<td>$E + V$</td>
<td>1</td>
<td>degree($v$)</td>
<td>degree($v$)</td>
</tr>
</tbody>
</table>

* disallows parallel edges

huge number of vertices, small average vertex degree
› graph API
› depth-first search
› breadth-first search
› connected components
› challenges
Maze exploration

Maze graphs.

- Vertex = intersection.
- Edge = passage.

**Goal.** Explore every intersection in the maze.
Trémaux maze exploration

Algorithm.

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

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

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

Claude Shannon (with Theseus mouse)
Maze exploration
Maze exploration
**Goal.** Systematically search through a graph.

**Idea.** Mimic maze exploration.

**DFS (to visit a vertex v)**

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

**Typical applications.**

- Find all vertices connected to a given source vertex.
- Find a path between two vertices.
**Design pattern**. Decouple graph data type from graph processing.

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

```java
public class Paths {
    Paths(Graph G, int s)  // find paths in G from source s
    boolean hasPathTo(int v)  // is there a path from s to v?
    Iterable<Integer> pathTo(int v)  // path from s to v; null if no such path
}
```

```java
Paths paths = new Paths(G, s);
for (int v = 0; v < G.V(); v++)
    if (paths.hasPathTo(v))
        StdOut.println(v);
```

print all vertices connected to s
Depth-first search demo
Depth-first search

**Goal.** Find all vertices connected to \( s \) (and a path).

**Idea.** Mimic maze exploration.

**Algorithm.**
- Use recursion (ball of string).
- Mark each visited vertex (and keep track of edge taken to visit it).
- Return (retrace steps) when no unvisited options.

**Data structures.**
- boolean[] marked to mark visited vertices.
- int[] edgeTo to keep tree of paths.
  
  \( (\text{edgeTo}[w] == v) \) means that edge \( v-w \) taken to visit \( w \) for first time
public class DepthFirstPaths {
    private boolean[] marked;
    private int[] edgeTo;
    private int s;
    public DepthFirstSearch(Graph G, int s) {
        ... 
        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);
                   edgeTo[w] = v;
                }
    }
}
Depth-first search properties

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

**Pf.**

- **Correctness:**
  - if $w$ marked, then $w$ connected to $s$ (why?)
  - if $w$ connected to $s$, then $w$ marked (if $w$ unmarked, then consider last edge on a path from $s$ to $w$ that goes from a marked vertex to an unmarked one)

- **Running time:** each vertex connected to $s$ is visited once.
**Depth-first search properties**

**Proposition.** After DFS, can find vertices connected to \( s \) in constant time and can find a path to \( s \) (if one exists) in time proportional to its length.

**Pf.** \( \text{edgeTo[]} \) is a parent-link representation of a tree rooted at \( s \).

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

public Iterable<Integer> pathTo(int v)
{
    if (!hasPathTo(v)) return null;
    Stack<Integer> path = new Stack<Integer>();
    for (int x = v; x != s; x = edgeTo[x])
       path.push(x);
    path.push(s);
    return path;
}
```

**Trace of pathTo() computation**

```
edgeTo[]
0  0
1  1  2
2  2  0
3  3  2
4  4  3
5  5  3
```

Diagram of the graph with edgeTo[] and pathTo() computation.
Depth-first search application: preparing for a date

xkcd
http://xkcd.com/761/
Depth-first search application: flood fill

**Challenge.** Flood fill (Photoshop magic wand).

**Assumptions.** Picture has millions to billions of pixels.

**Solution.** Build a grid graph.
- Vertex: pixel.
- Edge: between two adjacent gray pixels.
- Blob: all pixels connected to given pixel.
- graph API
- depth-first search
- breadth-first search
- connected components
- challenges
Breadth-first search demo
Breadth-first search

Depth-first search. Put unvisited vertices on a stack.
Breadth-first search. Put unvisited vertices on a queue.

Shortest path. Find path from $s$ to $t$ that uses fewest number of edges.

**BFS (from source vertex $s$)**

---

Put $s$ onto a FIFO queue, and mark $s$ as visited.
Repeat until the queue is empty:
- remove the least recently added vertex $v$
- add each of $v$'s unvisited neighbors to the queue, and mark them as visited.
---

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

Pf.

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

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

---

**Breadth-first search properties**

- **standard drawing**
- **dist = 0**
- **dist = 1**
- **dist = 2**
public class BreadthFirstPaths {
    private boolean[] marked;
    private boolean[] edgeTo[];
    private final int s;
    ...

    private void bfs(Graph G, int s) {
        Queue<Integer> q = new Queue<Integer>();
        q.enqueue(s);
        marked[s] = true;
        while (!q.isEmpty()) {
            int v = q.dequeue();
            for (int w : G.adj(v)) {
                if (!marked[w]) {
                    q.enqueue(w);
                    marked[w] = true;
                    edgeTo[w] = v;
                }
            }
        }
    }
}
Breadth-first search application: routing

Fewest number of hops in a communication network.
Breadth-first search application: Kevin Bacon numbers

Kevin Bacon numbers.

http://oracleofbacon.org

Endless Games board game

SixDegrees iPhone App
Kevin Bacon graph

- Include a vertex for each performer and for each movie.
- Connect a movie to all performers that appear in that movie.
- Compute shortest path from $s = \text{Kevin Bacon}$. 
Breadth-first search application: Erdös numbers

hand-drawing of part of the Erdös graph by Ron Graham
› graph API
› 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.

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

```java
public class CC {
    CC(Graph G) find connected components in G
    boolean connected(int v, int w) are v and w connected?
    int count() number of connected components
    int id(int v) component identifier for v
}
```

**Union-Find?** Not quite.
**Depth-first search.** Yes. [next few slides]
The relation "is connected to" is an equivalence relation:

- Reflexive: \( v \) is connected to \( v \).
- Symmetric: if \( v \) is connected to \( w \), then \( w \) is connected to \( v \).
- Transitive: if \( v \) connected to \( w \) and \( w \) connected to \( x \), then \( v \) connected to \( x \).

**Def.** A connected component is a maximal set of connected vertices.

**Remark.** Given connected components, can answer queries in constant time.
**Connected components**

**Def.** A connected component is a maximal set of connected vertices.
**Goal.** Partition vertices into 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.
Connected components demo
Finding connected components with DFS

```
public class CC {
    private boolean marked[];
    private int[] id;
    private int count;

    public CC(Graph G) {
        marked = new boolean[G.V()];
        id = new int[G.V()];
        for (int v = 0; v < G.V(); v++) {
            if (!marked[v]) {
                dfs(G, v);
                count++;
            }
        }
    }

    public int count() {
        return count;
    }

    public int id(int v) {
        return id[v];
    }

    private void dfs(Graph G, int v) {
        dfs(G, v);
        id[v] = id[component containing v];
    }
}
```
Finding connected components with DFS (continued)

```java
public int count()
{  return count;  }

public int id(int v)
{  return id[v];  }

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

- **number of components**
- **id of component containing v**
- **all vertices discovered in same call of dfs have same id**
Connected components application: study spread of STDs

Relationship graph at "Jefferson High"

Connected components application: particle detection

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
› depth-first search
› breadth-first search
› connected components
› challenges
Problem. Is a graph bipartite?

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 1

Problem. Is a graph bipartite?

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.

simple DFS-based solution (see textbook)
Relationship graph at "Jefferson High"

Graph-processing challenge 2

Problem. Find a cycle.

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 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.
Graph-processing challenge 4

**Problem.** Find a cycle that visits every vertex.

**Assumption.** Need to visit each vertex 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.
Graph-processing challenge 5

Problem. Are two graphs identical except for vertex names?

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?

**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.