4.1 **UNDIRECTED GRAPHS**

- introduction
- graph API
- depth-first search
- breadth-first search
- connected components
- challenges
4.1 Undirected Graphs

- introduction
- 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
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

Key: Employee (e-mail address)
At least one e-mail contact between employees

The analysis detected an anomaly: a new e-mail address for this person, who had been ‘philip.allen’ for 131 previous weeks.

Finding Patterns In Corporate Chatter

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

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>molecule</td>
<td>atom</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?
4.1 **Undirected Graphs**

- introduction
- graph API
- depth-first search
- breadth-first search
- connected components
- challenges
4.1 UNDIRECTED GRAPHS

- introduction
- graph API
- depth-first search
- breadth-first search
- connected components
- challenges
Graph representation

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

two drawings of the same graph

Caveat. Intuition can be misleading.
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

[Diagram of a graph with labeled vertices and edges showing self-loop and parallel edges]
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.

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**

The adjacency matrix for the graph is shown below, with two entries for each edge, indicating the presence or absence of a connection between vertices.
Adjacency-list graph representation

Maintain vertex-indexed array of lists.

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

Bag objects

```
representations of the same edge
```

```
0
0
6 → 2 → 1 → 5
```

```
5 → 4
```

```
5 → 6 → 3
```

```
3 → 4 → 0
```

```
0 → 4
```

```
8
```

```
7
```

```
11 → 10 → 12
```

```
9
```

```
9 → 12
```

```
9 → 12
```

```
11 → 9
```
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];
    }
}
```

- **Adjacency lists** (using `Bag` data type)
- Create empty graph with V vertices
- Add edge v-w (parallel edges and self-loops allowed)
- Iterator for vertices adjacent to 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
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
4.1 UNDIRECTED GRAPHS

- introduction
- graph API
- depth-first search
- breadth-first search
- connected components
- challenges
4.1 Undirected Graphs

- introduction
- graph API
- depth-first search
- breadth-first search
- connected components
- challenges
Maze exploration

Maze graph.

• Vertex = intersection.
• Edge = passage.

Goal. Explore every intersection in the maze.
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 instructed Theseus to use a ball of string to find his way back out.

Claude Shannon (with Theseus mouse)
Maze exploration
Maze exploration
**Depth-first search**

**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 challenge.** How to implement?
Design pattern for graph processing

**Design pattern.** Decouple graph data type from graph processing.
- Create a Graph object.
- Pass the Graph to a graph-processing routine.
- 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

To visit a vertex $v$:

- Mark vertex $v$ as visited.
- Recursively visit all unmarked vertices adjacent to $v$. 

![Graph G](image)
Depth-first search demo

To visit a vertex \( v \):

- Mark vertex \( v \) as visited.
- Recursively visit all unmarked vertices adjacent to \( v \).

<table>
<thead>
<tr>
<th>( v )</th>
<th>marked[]</th>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>–</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>11</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>12</td>
<td>F</td>
<td>–</td>
</tr>
</tbody>
</table>
**Depth-first search**

**Goal.** Find all vertices connected to \( s \) (and a corresponding 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 \( v \rightarrow w \) taken to visit \( w \) for first time
public class DepthFirstPaths
{
    private boolean[] marked;
    private int[] edgeTo;
    private int s;
    public DepthFirstPaths(Graph G, int s)
    {
        ...        if (!marked[w])
            {
                dfs(G, w);
                edgeTo[w] = v;
            }
    }
}

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;
        }
}

marked[v] = true if v connected to s
edgeTo[v] = previous vertex on path from s to v
initialize data structures
find vertices connected to s
recursive DFS does the work
**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).

**Pf.** [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 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()`

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

```
X
/   /
/  ____ /
/    |
X
/  _______________
/     |
/      |
/ _______
/   |
/   |
/  |
/  |
/  |
/  |
/  |
/  |
/  |
/  |
/  |
/  |

path
```
Depth-first search application: preparing for a date

**Preparing for a Date:**
- What situations might I prepare for?
  1) Medical emergency
  2) Dancing
  3) Food too expensive

**Okay, What Kinds of Emergencies Can Happen?**
- A) Snakebite
- B) Lightning strike
- C) Fall from chair

**Hmmm, Which Snakes Are Dangerous? Let's See...**
- A) Corn snake
- B) Garter snake
- C) Copperhead

**The research comparing snake venoms is scattered and inconsistent. I'll make a spreadsheet to organize it.**

**I'm here to pick you up. You're not dressed?**

**By the way, the Inland Taipan has the deadliest venom of any snake!**

**I really need to stop using depth-first searches.**

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.
4.1 Undirected Graphs

- introduction
- graph API
- depth-first search
- breadth-first search
- connected components
- challenges
4.1 UNDIRECTED GRAPHS

- introduction
- graph API
- depth-first search
- breadth-first search
- connected components
- challenges
Breadth-first search demo

Repeat until queue is empty:

- Remove vertex $v$ from queue.
- Add to queue all unmarked vertices adjacent to $v$ and mark them.
Breadth-first search demo

Repeat until queue is empty:

- Remove vertex $v$ from queue.
- Add to queue all unmarked vertices adjacent to $v$ and mark them.

<table>
<thead>
<tr>
<th>$v$</th>
<th>edgeTo[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

done
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 paths (fewest number of edges) from \( s \) to all other vertices in a 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 \( k + 1 \).

**Pf.** [running time] Each vertex connected to \( s \) is visited once.
Breadth-first search

```java
class BreadthFirstPaths {
    private boolean[] marked;
    private int[] edgeTo;
    
    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.

ARPANET, July 1977
Breadth-first search application: Kevin Bacon numbers

Kevin Bacon numbers.

http://oracleofbacon.org

Endless Games board game

SixDegrees iPhone App
Kevin Bacon graph

- Include one vertex for each performer and one 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: Erdős numbers

hand-drawing of part of the Erdős graph by Ron Graham
4.1 Undirected Graphs

- introduction
- graph API
- depth-first search
- breadth-first search
- connected components
- challenges
4.1 **Undirected Graphs**

- introduction
- 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 of the form *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.
Connected components

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

To visit a vertex $v$:
- Mark vertex $v$ as visited.
- Recursively visit all unmarked vertices adjacent to $v$. 

![Graph G](image)

<table>
<thead>
<tr>
<th>$v$</th>
<th>marked[]</th>
<th>id[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>11</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>12</td>
<td>F</td>
<td>–</td>
</tr>
</tbody>
</table>
To visit a vertex $v$:

- Mark vertex $v$ as visited.
- Recursively visit all unmarked vertices adjacent to $v$.

<table>
<thead>
<tr>
<th>$v$</th>
<th>marked[]</th>
<th>id[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>T</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>T</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>T</td>
<td>2</td>
</tr>
</tbody>
</table>
Finding connected components with DFS

```java
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) {
        id[v] = id of component containing v
        run DFS from one vertex in each component
    }
}
```

id[v] = id of component containing v
number of components
run DFS from one vertex in each component
see next slide
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

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.

**Particle tracking.** Track moving particles over time.
4.1 Undirected Graphs

- introduction
- graph API
- depth-first search
- breadth-first search
- connected components
- challenges
4.1 Undirected Graphs

- Introduction
- Graph API
- Depth-first search
- Breadth-first search
- Connected components
- Challenges
Graph-processing challenge 1

Problem. Is a graph bipartite?

How difficult?
- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

simple DFS-based solution (see textbook)
Bipartiteness application: is dating graph bipartite?
Graph-processing challenge 2

**Problem.** Find a cycle.

**How difficult?**
- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

simple DFS-based solution (see textbook)
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 (general) cycle that uses each edge exactly once?
Answer. A connected graph is Eulerian iff all vertices have even degree.
Graph-processing challenge 3

**Problem.** Find a (general) cycle that uses every edge exactly once.

**How difficult?**
- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

Eulerian tour (classic graph-processing problem)
Graph-processing challenge 4

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

**How difficult?**
- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
✓ Intractable.
- No one knows.
- Impossible.

Hamiltonian cycle (classical NP-complete problem)
Graph-processing challenge 5

**Problem.** Are two graphs identical except for vertex names?

**How difficult?**
- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

Graph isomorphism is longstanding open problem.
Graph-processing challenge 6

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

How difficult?
• Any programmer could do it.
• Typical diligent algorithms student could do it.
✓ • Hire an expert.
• Intractable.
• No one knows.
• Impossible.

linear-time DFS-based planarity algorithm discovered by Tarjan in 1970s (too complicated for most practitioners)
4.1 Undirected Graphs

- introduction
- graph API
- depth-first search
- breadth-first search
- connected components
- challenges
4.1 Undirected Graphs

- introduction
- graph API
- depth-first search
- breadth-first search
- connected components
- challenges