How do you find the pace of lectures so far? (answer only about yourself, not the class as a whole)
(64 responses)
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.
Border graph of 48 contiguous United States
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
The evolution of FCC lobbying coalitions

“The Evolution of FCC Lobbying Coalitions” by Pierre de Vries in JoSS Visualization Symposium 2010
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 change.
The Internet as mapped by the Opte Project

http://en.wikipedia.org/wiki/Internet
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>intersection</td>
<td>street</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</td>
<td>friendship</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

<table>
<thead>
<tr>
<th>problem</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s–t path</td>
<td>Is there a path between s and t ?</td>
</tr>
<tr>
<td>shortest s–t path</td>
<td>What is the shortest path between s and t ?</td>
</tr>
<tr>
<td>cycle</td>
<td>Is there a cycle in the graph ?</td>
</tr>
<tr>
<td>Euler cycle</td>
<td>Is there a cycle that uses each edge exactly once ?</td>
</tr>
<tr>
<td>Hamilton cycle</td>
<td>Is there a cycle that uses each vertex exactly once ?</td>
</tr>
<tr>
<td>connectivity</td>
<td>Is there a way to connect all of the vertices ?</td>
</tr>
<tr>
<td>biconnectivity</td>
<td>Is there a vertex whose removal disconnects the graph ?</td>
</tr>
<tr>
<td>planarity</td>
<td>Can the graph be drawn in the plane with no crossing edges ?</td>
</tr>
<tr>
<td>graph isomorphism</td>
<td>Do two adjacency lists represent the same graph ?</td>
</tr>
</tbody>
</table>

**Challenge.** Which graph problems are easy? difficult? intractable?
One of the first graph problems: Hamiltonian cycle

Sir William Rowan Hamilton: 1805 – 1865
Irish physicist, astronomer, and mathematician
TCD graduate

In 1857 he invented the icosian game: find a cycle along the edges of a dodecahedron such that every vertex is visited exactly once time, and the ending point is the same as the starting point.

The puzzle was distributed commercially as a pegboard with holes at the nodes of the dodecahedral graph and was subsequently marketed in Europe in many forms.
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
## Graph API

```java
public class Graph {

    // Constructor to create an empty graph with V vertices
    Graph(int V) {
        // Implementation
    }

    // Constructor to create a graph from input stream
    Graph(In in) {
        // Implementation
    }

    // Method to add an edge v-w
    void addEdge(int v, int w) {
        // Implementation
    }

    // Method to get the vertices adjacent to v
    Iterable<Integer> adj(int v) {
        // Implementation
    }

    // Method to get the number of vertices
    int V() {
        // Implementation
    }

    // Method to get the number of edges
    int E() {
        // Implementation
    }
}
```

- **Graph(\(\text{int } V\))**: create an empty graph with \(V\) vertices
- **Graph(\(\text{In } in\))**: create a graph from input stream
- **void addEdge(\(\text{int } v, \text{int } w\))**: add an edge \(v-w\)
- **Iterable<\(\text{Integer}\)> adj(\(\text{int } v\))**: vertices adjacent to \(v\)
- **int V()**: number of vertices
- **int E()**: number of edges

### Example
```java
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);
    }
}
```

- **In in = new In(args[0]);**: read graph from input stream
- **for (int v = 0; v < G.V(); v++)**: print out each edge (twice)
Graph API: sample client

Graph input format.

```
% java Test tinyG.txt
0-6
0-2
0-1
0-5
1-0
2-0
3-5
3-4

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);
```

read graph from input stream

print out each edge (twice)
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

// degree of vertex v in graph G
public static int degree(Graph G, int v)
{
    int degree = 0;
    for (int w : G.adj(v))
        degree++;
    return degree;
}
Set-of-edges graph representation

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

Q. How long to iterate over vertices adjacent to ν?
Adjacency-matrix graph representation

Maintain a two-dimensional $V$-by-$V$ boolean array; for each edge $v\rightarrow w$ in graph: $\text{adj}[v][w] = \text{adj}[w][v] = \text{true}$.

Q. How long to iterate over vertices adjacent to $v$?
Adjacency-list graph representation

Maintain vertex-indexed array of lists.

Q. How long to iterate over 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>$\text{degree}(v)$</td>
<td>$\text{degree}(v)$</td>
</tr>
</tbody>
</table>

* disallows parallel edges
Adjacency-list graph representation: Java implementation

```java
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
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.
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 instructed Theseus to use a ball of string to find his way back out.

The Labyrinth (with Minotaur)  Claude Shannon (with Theseus mouse)
Maze exploration: easy
Maze exploration: medium
Depth-first search

Goal. Systematically traverse a graph.

Typical applications.

- Find all vertices connected to a given source vertex.
- Find a path between two vertices.

Design challenge. How to implement?
To visit a vertex $v$:

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

**graph G**

**tinyG.txt**

```
V
13
13
0 5
4 3
0 1
9 12
6 4
5 4
0 2
11 12
9 10
0 6
7 8
9 11
5 3
```
To visit a vertex $v$:

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

### Depth-first search demo

Here is a sample depth-first search with a graph:

**Vertices reachable from 0:**
- 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

**Marked[]** and **edgeTo[]**:

<table>
<thead>
<tr>
<th></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>
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

    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: data structures

To visit a vertex $v$:

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

Data structures.

- Boolean array `marked[]` to mark visited vertices.
- Integer array `edgeTo[]` to keep track of paths.
  
  `(edgeTo[w] == v)` means that edge $v$-$w$ taken to visit $w$ for first time
- Function-call stack for recursion.
**Depth-first search: Java implementation**

```java
public class DepthFirstPaths {

    private boolean[] marked;
    private int[] edgeTo;
    private int s;

    public DepthFirstPaths(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 (plus time to initialize the `marked[]` array).

**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 check if vertex \( v \) is connected to \( s \) in constant time and can find \( v \rightarrow s \) path (if one exists) in time proportional to its length.

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

```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**

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 & 4 \\
1 & 2 & 0 & 3 & 2 \\
2 & 0 & 2 & 3 & 5 \\
3 & 2 & 3 & 5 & \\
4 & 3 & 3 & 5 & \\
5 & 5 & 5 & \\
\end{array}
\]
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 (implicitly).

- **Vertex:** pixel.
- **Edge:** between two adjacent gray pixels.
- **Blob:** all pixels connected to given pixel.
Depth-first search application: preparing for a date

http://xkcd.com/761/
4.1 **Undirected Graphs**

- introduction
- graph API
- depth-first search
- breadth-first search
- connected components
- challenges
Repeat until queue is empty:

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

Repeat until queue is empty:

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

**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.
Breadth-first search: Java implementation

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

    private void bfs(Graph G, int s) {
        Queue<Integer> q = new Queue<Integer>();
        q.enqueue(s);
        marked[s] = true;
        distTo[s] = 0;
        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;
                    distTo[w] = distTo[v] + 1;
                }
            }
        }
    }
}
```

- **Initialize FIFO queue of vertices to explore:**
- **Found new vertex w via edge v-w**
Breadth-first search properties

**Q.** In which order does BFS examine vertices?

**A.** Increasing distance (number of edges) from $s$.

- Queue always consists of $\geq 0$ vertices of distance $k$ from $s$, followed by $\geq 0$ vertices of distance $k+1$

**Proposition.** In any connected graph $G$, BFS computes shortest paths from $s$ to all other vertices in time proportional to $E + V$. 
Breadth-first search application: routing

Fewest number of hops in a communication network.

ARPANET, July 1977
Breadth-first search application: 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
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.

<table>
<thead>
<tr>
<th>public class <strong>CC</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>CC(Graph $G$)</td>
</tr>
<tr>
<td>boolean connected(int $v$, int $w$)</td>
</tr>
<tr>
<td>int count()</td>
</tr>
<tr>
<td>int id(int $v$)</td>
</tr>
</tbody>
</table>

**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.

---

**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 $v$.

```
<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 $v$.

\[
\begin{array}{c|c|c}
\text{v} & \text{marked[]} & \text{id}[] \\
\hline
0 & T & 0 \\
1 & T & 0 \\
2 & T & 0 \\
3 & T & 0 \\
4 & T & 0 \\
5 & T & 0 \\
6 & T & 0 \\
7 & T & 1 \\
8 & T & 1 \\
9 & T & 2 \\
10 & T & 2 \\
11 & T & 2 \\
12 & T & 2 \\
\end{array}
\]
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];
    }
    public boolean connected(int v, int w) {
        return id[v] == id[w];
    }
    private void dfs(Graph G, int v) {
        // DFS implementation
    }
}
```

- `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]; }

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

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
- v and w connected iff same id
- 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.
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 cycle. 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.

Euler cycle
(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.

Hamilton 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

0↔4, 1↔3, 2↔2, 3↔6, 4↔5, 5↔0, 6↔1
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)
Graph traversal summary

BFS and DFS enables efficient solution of many (but not all) graph problems.

<table>
<thead>
<tr>
<th>problem</th>
<th>BFS</th>
<th>DFS</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>path between s and t</td>
<td>✔</td>
<td>✔</td>
<td>$E + V$</td>
</tr>
<tr>
<td>shortest path between s and t</td>
<td>✔</td>
<td>✔</td>
<td>$E + V$</td>
</tr>
<tr>
<td>connected components</td>
<td>✔</td>
<td>✔</td>
<td>$E + V$</td>
</tr>
<tr>
<td>biconnected components</td>
<td></td>
<td>✔</td>
<td>$E + V$</td>
</tr>
<tr>
<td>cycle</td>
<td>✔</td>
<td>✔</td>
<td>$E + V$</td>
</tr>
<tr>
<td>Euler cycle</td>
<td></td>
<td>✔</td>
<td>$E + V$</td>
</tr>
<tr>
<td>Hamilton cycle</td>
<td></td>
<td></td>
<td>$2^{1.657V}$</td>
</tr>
<tr>
<td>bipartiteness</td>
<td>✔</td>
<td>✔</td>
<td>$E + V$</td>
</tr>
<tr>
<td>planarity</td>
<td></td>
<td>✔</td>
<td>$E + V$</td>
</tr>
<tr>
<td>graph isomorphism</td>
<td></td>
<td></td>
<td>$2^{c\sqrt{V \log V}}$</td>
</tr>
</tbody>
</table>