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
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

Finding Patterns In Corporate Chatter

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

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

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.
## 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?
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.

---

*Graph representation*

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

symbol table

self-loop

parallel edges
## 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
    }

    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);

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

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

E

read graph from input stream

print out each edge (twice)
# Typical graph-processing code

```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
    }
}
```

```java
// 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).
Adjacency-matrix graph representation

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

Maintain vertex-indexed array of lists.

Adjacency-lists representation (undirected graph)

Bag objects

representations of the same edge
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

huge number of vertices, small average vertex degree
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.
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

    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 $G$
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[v]</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 $v$–$w$ taken to visit $w$ for first time
Depth-first search

```java
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;
        }
    }
}
```

- `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.** `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()` computation

```
edgeTo[]

0
1 2
2 0
3 2
4 3
5 3
```
Depth-first search application: preparing for a date

[Diagram showing a comic of depth-first search application: preparing for a date]

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.
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.

**graph G**

tinyCG.txt

```text
V
  6
  8
  0 5
  2 4
  2 3
  1 2
  0 1
  3 4
  3 5
  0 2
E
```
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 $s$. 
Breadth-first search properties

**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
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 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 $s = Kevin Bacon$. 
Breadth-first search application: Erdös numbers

hand-drawing of part of the Erdös 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.

```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 $v$
        (between 0 and count() - 1)
    }
}
```

**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.

**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 \( v \).

---

**Connected components demo**

---

<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>

**done**
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 logic...
    }
}
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.
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.

A simple DFS-based solution (see textbook)
Bipartiteness application: is dating graph bipartite?

Image created by Mark Newman.
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. Yes iff connected and 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.

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)