4.1 Undirected Graphs

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

- introduction
- graph API
- depth-first search
- breadth-first search
- 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.
Social networks

Vertex = person; edge = social relationship.

“Visualizing Friendships” by Paul Butler
Protein-protein interaction network

Vertex = protein; edge = interaction.

Reference: Jeong et al, Nature Review | Genetics
The evolution of FCC lobbying coalitions

Vertex = company; edge = lobbying partner.

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

Vertex = IP address.
Edge = connection.

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

**Graph.** Set of vertices connected pairwise by edges.

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

**Def.** Two vertices are connected if there is a path between them.

**Cycle.** Path whose first and last vertices are the same.
### 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 path between every pair of 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>Are two graphs isomorphic?</td>
</tr>
</tbody>
</table>

**Challenge.** Which graph problems are easy? difficult? intractable?
4.1 **Undirected Graphs**

- introduction
- graph API
- depth-first search
- breadth-first search
- 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

**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.
Graph API

```java
public class Graph {
    // create an empty graph with V vertices
    Graph(int V) {
    }
    // create a graph from input stream
    Graph(In in) {
    }
    // add an edge v–w
    void addEdge(int v, int w) {
    }
    // vertices adjacent to v
    Iterable<Integer> adj(int v) {
    }
    // number of vertices
    int V() {
    }
    // number of edges
    int E() {
    }
    // 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;
    }
}
```
Graph representation: adjacency matrix

Maintain a two-dimensional $V$-by-$V$ boolean array; for each edge $v$–$w$ in graph: $\text{adj}[v][w] = \text{adj}[w][v] = \text{true}$. 
Which is order of growth of running time of the following code fragment if the graph uses the **adjacency-matrix** representation, where $V$ is the number of vertices and $E$ is the number of edges?

```java
for (int v = 0; v < G.V(); v++)
    for (int w : G.adj(v))
        StdOut.println(v + "-" + w);
```

A. $V$
B. $E + V$
C. $V^2$
D. $VE$
Graph representation: adjacency lists

Maintain vertex-indexed array of lists.
Which is order of growth of running time of the following code fragment if the graph uses the adjacency-lists representation, where $V$ is the number of vertices and $E$ is the number of edges?

A. $V$
B. $E + V$
C. $V^2$
D. $VE$

```java
for (int v = 0; v < G.V(); v++)
    for (int w : G.adj(v))
        StdOut.println(v + "-" + w);
print each edge twice
```
Graph representations

In practice. Use adjacency-lists representation.

- Algorithms based on iterating over vertices adjacent to $v$.
- Real-world graphs tend to be \textit{sparse} (not \textit{dense}).

\begin{itemize}
  \item proportional to $V$ edges
  \item proportional to $V^2$ edges
\end{itemize}

Two graphs ($V = 50$)

\begin{itemize}
  \item sparse ($E = 200$)
  \item dense ($E = 1000$)
\end{itemize}
Graph representations

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

- Algorithms based on iterating over vertices adjacent to $v$.
- Real-world graphs tend to be *sparse* (not *dense*).

<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
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
4.1 Undirected Graphs

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

Maze graph.

- Vertex = intersection.
- Edge = passage.

Goal. Explore every intersection in the maze.
Maze exploration: National Building Museum

Trémaux maze exploration

Algorithm.

- Unroll a ball of string behind you.
- Mark each newly discovered intersection and passage.
- Retrace steps when no unmarked options.
Trémaux maze exploration

Algorithm.

- Unroll a ball of string behind you.
- Mark each newly discovered intersection and passage.
- Retrace steps when no unmarked 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 Cretan Labyrinth (with Minotaur)
http://commons.wikimedia.org/wiki/File:Minotaurus.gif

Claude Shannon (with electromechanical mouse)
Maze exploration: easy
Maze exploration: medium
Maze exploration: challenge for the bored
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?
Depth-first search demo

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

**graph G**
To visit a vertex $v$:

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

vertices reachable from 0
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.

```
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 $v$.
- Recursively visit all unmarked vertices adjacent to $v$.

Data structures.
- Boolean array $marked[\cdot]$ to mark vertices.
- Integer array $edgeTo[\cdot]$ to keep track of paths.
  - ($edgeTo[w] == v$) means that edge $v$-$w$ taken to discover vertex $w$
- 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])
                {
                    edgeTo[w] = v;
                    dfs(G, w);
                }
    }
}
```

- `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 (plus time to initialize the \( \text{marked}[] \) 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.
Proposition. After DFS, can check if vertex $v$ is connected to $s$ in constant time and can find $v$–$s$ path (if one exists) in time proportional to its length.

Pf. `edgeTo[]` is parent-link representation of a tree rooted at vertex $s$.

```
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;
}
```
Problem. Implement flood fill (Photoshop magic wand).
**Non-recursive DFS**

**Challenge.** Implement DFS without recursion.
4.1 UNDIRECTED GRAPHS

- introduction
- graph API
- depth-first search
- breadth-first search
- challenges
Tree traversal. Many ways to explore every vertex in a binary tree.

- Inorder: A C E H M R S X
- Preorder: S E A C R H M X
- Postorder: C A M H R E X S
- Level-order: S E X A R C H M

Graph search. Many ways to explore every vertex in a graph.

- Preorder: vertices in order of calls to $\text{dfs}(G, v)$.
- Postorder: vertices in order of returns from $\text{dfs}(G, v)$.
- Level-order: vertices in increasing order of distance from $s$. 
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**
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

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 unmarked neighbors to the queue, and mark them.
Breadth-first search: Java implementation

```java
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 \( v \rightarrow w \)
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 $E + V$. 

![Graph G](image1)

![Distances](image2)
Breadth-first search application: routing

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

The Oracle of Bacon

Bernard Chazelle has a Bacon number of 3.

Guy and Madeline on a Park Bench (2009)

Anna Chazelle

La La Land (2016/I)

Ryan Gosling

Crazy, Stupid, Love. (2011)

Kevin Bacon

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 $s = \text{Kevin Bacon}$. 

Graph representation:

- **Kevin Bacon**
- **Dial M For Murder**
- **Caligola**
- **The Stepford Wives**
- **To Catch a Thief**
- **An American Haunting**
- **Cold Mountain**
- **The Eagle Has Landed**
- **The Woodsman**
- **The River Wild**
- **Titanic**
- **Apollo 13**
- **To Be or Not to Be**
- **Beverly Hills Cop**
- **Vernon Dobtcheff**
- **John Gielgud**
- **Glenn Close**
- **Streep Meryl**
- **Nicole Kidman**
- **John Belushi**
- **Kate Winslet**
- **Bill Paxton**
- **Steve Martin**
- **Mary Steenburgen**
- **Kevin Bacon**
- **Dial M For Murder**
- **Caligola**
- **The Stepford Wives**
- **To Catch a Thief**
- **Apollo 13**
- **To Be or Not to Be**
- **Beverly Hills Cop**
- **Vernon Dobtcheff**
- **John Gielgud**
- **Glenn Close**
- **Streep Meryl**
- **Nicole Kidman**
- **John Belushi**
- **Kate Winslet**
- **Bill Paxton**
- **Steve Martin**
- **Mary Steenburgen**
- **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
- challenges
Problem. Identify connected components.

How difficult?

A. Any programmer could do it.
B. Diligent algorithms student could do it.
C. Hire an expert.
D. Intractable.
E. No one knows.
Graph-processing challenge 1

Problem. Identify connected components.

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.
Graph-processing challenge 2

Problem. Is a graph bipartite?

How difficult?

A. Any programmer could do it.
B. Diligent algorithms student could do it.
C. Hire an expert.
D. Intractable.
E. No one knows.
Graph-processing challenge 3

Problem. Find a cycle in a graph (if one exists).

How difficult?

A. Any programmer could do it.
B. Diligent algorithms student could do it.
C. Hire an expert.
D. Intractable.
E. No one knows.
Graph-processing challenge 4

Problem. Is there a (general) cycle that uses every edge exactly once?

How difficult?

A. Any programmer could do it.
B. Diligent algorithms student could do it.
C. Hire an expert.
D. Intractable.
E. No one knows.
Problem. Is there a cycle that contains every vertex exactly once?

How difficult?

A. Any programmer could do it.

B. Diligent algorithms student could do it.

C. Hire an expert.

D. Intractable.

E. No one knows.
Graph-processing challenge 6

Problem. Are two graphs identical except for vertex names?

How difficult?

A. Any programmer could do it.
B. Diligent algorithms student could do it.
C. Hire an expert.
D. Intractable.
E. No one knows.
**Problem.** Can you draw a graph in the plane with no crossing edges?

<table>
<thead>
<tr>
<th>How difficult?</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A.</strong> Any programmer could do it.</td>
</tr>
<tr>
<td><strong>B.</strong> Diligent algorithms student could do it.</td>
</tr>
<tr>
<td><strong>C.</strong> Hire an expert.</td>
</tr>
<tr>
<td><strong>D.</strong> Intractable.</td>
</tr>
<tr>
<td><strong>E.</strong> No one knows</td>
</tr>
</tbody>
</table>

try it yourself at http://planarity.net
Graph traversal summary

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

<table>
<thead>
<tr>
<th>graph problem</th>
<th>BFS</th>
<th>DFS</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>s–t path</td>
<td>✔</td>
<td>✔</td>
<td>$E + V$</td>
</tr>
<tr>
<td>shortest s–t path</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 (odd cycle)</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>planarity</td>
<td></td>
<td>✔</td>
<td>$E + V$</td>
</tr>
<tr>
<td>graph isomorphism</td>
<td></td>
<td></td>
<td>$2^{c \ln^3 V}$</td>
</tr>
</tbody>
</table>