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
Framingham heart study

The Spread of Obesity in a Large Social Network over 32 Years by Christakis and Fowler in New England Journal of Medicine, 2007

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.

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.

Figure 2. The Internet as mapped by the Opte Project.

http://en.wikipedia.org/wiki/Internet

The Internet as mapped by the Opte Project.
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?

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.
Graph API

<table>
<thead>
<tr>
<th>public class Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph(int V)</td>
</tr>
<tr>
<td>Graph(In in)</td>
</tr>
<tr>
<td>void addEdge(int v, int w)</td>
</tr>
<tr>
<td>Iterable&lt;Integer&gt; adj(int v)</td>
</tr>
<tr>
<td>int V()</td>
</tr>
<tr>
<td>int E()</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>public class Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph(int V)</td>
</tr>
<tr>
<td>Graph(In in)</td>
</tr>
<tr>
<td>void addEdge(int v, int w)</td>
</tr>
<tr>
<td>Iterable&lt;Integer&gt; adj(int v)</td>
</tr>
<tr>
<td>int V()</td>
</tr>
<tr>
<td>int E()</td>
</tr>
</tbody>
</table>

// 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 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);

Set-of-edges graph representation

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

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

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

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 ($V = 50$)

<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

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

huge number of vertices, small average vertex degree
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];
    }
}
```

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: easy

Maze exploration: medium

Maze exploration: challenge for the bored
**Depth-first search**

**Goal.** Systematically traverse a graph.

**Idea.** Mimic maze exploration.

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

**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.
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 $\text{marked}[]$ to mark visited vertices.
- Integer array $\text{edgeTo}[]$ to keep track of paths.
  $(\text{edgeTo}[w] == v)$ means that edge $v\rightarrow w$ taken to visit $w$ for first time
- Function-call stack for recursion.

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.

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[v] = v;
    }
}
```

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. $\text{edgeTo}[]$ 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;
}
```
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

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>

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 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: 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;
                }
            }
        }
    }
}
```
Breadth-first search application: routing

Fewest number of hops in a communication network.

ARPANET, July 1977

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: Kevin Bacon numbers

http://oracleofbacon.org

Endless Games board game

SixDegrees iPhone App

Breadth-first search application: Erdös numbers

hand-drawing of part of the Erdös graph by Ron Graham
4.1 UNDIRECTED GRAPHS

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 CC</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCGraph 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]

Connected components

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.

63 connected components
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.

For each unmarked vertex $v$, run DFS to identify all vertices discovered as part of the same component.

**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()
    public int id(int v)
    public boolean connected(int v, int w)
    private void dfs(Graph G, int v)
}
```

**Connected components demo**

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

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 ≥ 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

Particle tracking. Track moving particles over time.
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.

Bipartiteness application: is dating graph bipartite?

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.

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.

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-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.
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^{\sqrt{\log V}}</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^{\sqrt{V \log V}}</td>
</tr>
</tbody>
</table>