4.1 Undirected Graphs

- graph API
- depth-first search
- breadth-first search
- connected components
- challenges
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.
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?
› graph API
› depth-first search
› breadth-first search
› connected components
› challenges
Graph representation

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

**Caveat.** Intuition can be misleading.

two drawings of the same graph
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

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
```

```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);
```

read graph from input stream
print out each edge (twice)
Graph API: sample client

Graph input format.

```
read graph from input stream
```

```
print out each edge (twice)
```

```
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
### 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
}
```

#### string representation of the graph's adjacency lists

```java
public String toString() {
    String s = V + " vertices, " + E + " edges
    for (int v = 0; v < V; v++)
        s += v + ": ";
        for (int w : this.adj(v))
            s += w + " "
        s += "\n";
    return s;
}
```
Possible Graph Representations:

- Set of edges
- Adjacency matrix
- Adjacency lists

On what basis to choose?

Let’s look at some example to gain perspective.
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
"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
# 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>
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 $v$–$w$ in graph: $\text{adj}[v][w] = \text{adj}[w][v] = \text{true}$. 

Adjacency-matrix graph representation 

two entries for each edge
Adjacency-list graph representation

Maintain vertex-indexed array of lists.

Adjacency-lists representation (undirected graph)

Bag objects

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

- Algorithms based on iterating over vertices adjacent to \( v \).
- Real-world graphs tend to be \textit{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 )</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
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];
    }
}
```
Main application. Adding items to a collection and iterating (when order doesn't matter).

```
public class Bag<Item> implements Iterable<Item>
Bag() create an empty bag
void add(Item x) insert a new item onto bag
int size() number of items in bag
Iterable<Item> iterator() iterator for all items in bag
```

Implementation. Stack (without pop) or queue (without dequeue).
› graph API
› depth-first search
› breadth-first search
› connected components
› challenges
Maze exploration

Maze graphs.
- 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
Warning: Don’t visit twice!

An' here I sit so patiently
Waiting to find out what price
You have to pay to get out of
Going through all these things twice.

Bob Dylan
“Stuck Inside Of Mobile With The Memphis Blues Again”
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. Decouple graph data type from graph processing.

- Create a `Graph` object.
- Pass the `Graph` to a graph-processing routine, e.g., `Paths`.
- 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
**Goal.** Find all vertices connected to $s$ (and a 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.
  - `(edgeTo[w] == v)` means that edge $v$-$w$ taken to visit $w$ for first time
public 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;
    }
}
**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)
- **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 a 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;
}
```
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

HMM. 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 LD_{50}, THE INLAND TAIPAN HAS THE DEADLIEST VENOM OF ANY SNAKE!

I REALLY NEED TO STOP USING DEPTH-FIRST SEARCHES.

http://xkcd.com/761/
› graph API
› depth-first search
› breadth-first search
› connected components
› challenges
Breadth-first search demo
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 \( s \))

---

<table>
<thead>
<tr>
<th>Put ( s ) onto a FIFO queue, and mark ( s ) as visited.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repeat until the queue is empty:</td>
</tr>
<tr>
<td>- remove the least recently added vertex ( v )</td>
</tr>
<tr>
<td>- add each of ( v )'s unvisited neighbors to the queue,</td>
</tr>
<tr>
<td>and mark them as visited.</td>
</tr>
</tbody>
</table>

---

Intuition. BFS examines vertices in increasing distance from \( s \).
Proposition. BFS computes shortest path (number of edges) from $s$ in a connected 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 $k + 1$.

- Running time: each vertex connected to $s$ is visited once.
public class BreadthFirstPaths
{
    private boolean[] marked;
    private boolean[] edgeTo[];
    private final int s;
    ...

    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 a vertex for each performer and 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: Erdös numbers

hand-drawing of part of the Erdös graph by Ron Graham
- 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: is $v$ connected to $w$? in constant time.

<table>
<thead>
<tr>
<th>public class CC</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC(Graph G)</td>
</tr>
<tr>
<td>boolean</td>
</tr>
<tr>
<td>connected(int v, int w)</td>
</tr>
<tr>
<td>find connected components in G</td>
</tr>
<tr>
<td>are v and w connected?</td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td>count()</td>
</tr>
<tr>
<td>number of connected components</td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td>id(int v)</td>
</tr>
<tr>
<td>component identifier for 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.
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
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`
- 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

Relationship graph at "Jefferson High"

- graph API
- depth-first search
- breadth-first search
- connected components
- challenges
Graph-processing challenge 1

Problem. Is a graph bipartite?

How difficult?

- Any Villanova CS student could do it.
- Need to be a typical diligent CSC 2053 student.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.
Problem. Find a cycle.

How difficult?
- Any Villanova CS student could do it.
- Need to be a typical diligent CSC 2053 student.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.
Problem. Find a cycle that uses every edge.
Assumption. Need to use each edge exactly once.

How difficult?
• Any Villanova CS student could do it.
• Need to be a typical diligent CSC 2053 student.
• Hire an expert.
• Intractable.
• No one knows.
• Impossible.
Bridges of Königsberg

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.
To find path. DFS-based algorithm (see textbook).
Graph-processing challenge 4

Problem. Find a cycle that visits every vertex.
Assumption. Need to visit each vertex exactly once.

How difficult?
• Any Villanova CS student could do it.
• Need to be a typical diligent CSC 2053 student.
• Hire an expert.
• Intractable.
• No one knows.
• Impossible.
Problem. Are two graphs identical except for vertex names?

How difficult?
• Any Villanova CS student could do it.
• Need to be a typical diligent CSC 2053 student.
• 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 Villanova CS student could do it.
- Need to be a typical diligent CSC 2053 student.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.