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

Map of science clickstreams

10 million Facebook friends

"Visualizing Friendships" by Paul Butler

One week of Enron emails

Finding Patterns In Corporate Chatter
The evolution of FCC lobbying coalitions

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>

Framingham heart study

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 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.
**Graph API**

```java
public class Graph
{
    public Graph(int V) {
        // create an empty graph with V vertices
    }
    public 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)`)
```java
StdOut.println(v + "-" + w);
```

**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
12-11
12-9
```

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)`)
```java
StdOut.println(v + "-" + w);
```

**Typical graph-processing code**

```java
public class Graph
{
    public Graph(int V) {
        // create an empty graph with V vertices
    }
    public 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).
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.

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];
    }
}
```

Graph representations

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

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 $v$
huge number of vertices, small average vertex degree
sparse (E = 200)
dense (E = 1000)
Two graphs (V = 50)
Graph representations

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

<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

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

Goal. Systematically search through a graph.

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.
- 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
        }
    }
}
```

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

Depth-first search demo

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

Goal. Find all vertices connected to s (and a corresponding path).

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
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);
  }
  return path;
}
```

Depth-first search application: preparing for a date

![XKCD comic](http://xkcd.com/761/)

**XKCD comic title:** I really need to stop using depth-first searches.
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.

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.

**Breadth-first search demo**

**4.1 Undirected Graphs**

- introduction
- graph API
- depth-first search
- breadth-first search
- connected components
- challenges
**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 application: routing**

Fewest number of hops in a communication network.

---

**ARPANET, July 1977**
Breadth-first search application: Kevin Bacon numbers

Kevin Bacon numbers.

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: 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]

Connected components

**Def.** A **connected component** is a maximal set of connected vertices.

**Goal.** Partition vertices into connected components.

**Remark.** Given connected components, can answer queries in constant time.

![Connected components diagram]

```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)
}
```

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 diagram]
To visit a vertex \( v \):  
- Mark vertex \( v \) as visited.  
- Recursively visit all unmarked vertices adjacent to \( v \).

```
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) {
        marked[v] = true;
        id[v] = count;
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                dfs(G, w);
            }
        }
    }
}
```

Finding connected components demo

Connected components demo

To visit a vertex \( v \):  
- Mark vertex \( v \) as visited.  
- Recursively visit all unmarked vertices adjacent to \( v \).
**Connected components application: study spread of STDs**


**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 detection example](image)

**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.

**4.1 Undirected Graphs**

- Introduction
- Graph API
- Depth-first search
- Breadth-first search
- Connected components
- Challenges
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.

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.

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.
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

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)