Undirected Graphs

Graph Applications

<table>
<thead>
<tr>
<th>Graph</th>
<th>Vertices</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>communication</td>
<td>telephones, computers</td>
<td>fiber optic cables</td>
</tr>
<tr>
<td>circuits</td>
<td>gates, registers, processors</td>
<td>wires</td>
</tr>
<tr>
<td>mechanical</td>
<td>joints</td>
<td>rods, beams, springs</td>
</tr>
<tr>
<td>hydraulic</td>
<td>reservoirs, pumping stations</td>
<td>pipelines</td>
</tr>
<tr>
<td>financial</td>
<td>stocks, currency</td>
<td>transactions</td>
</tr>
<tr>
<td>transportation</td>
<td>street intersections, airports</td>
<td>highways, airway routes</td>
</tr>
<tr>
<td>social relationships</td>
<td>people, actors</td>
<td>friendships, movie casts</td>
</tr>
<tr>
<td>neural networks</td>
<td>neurons</td>
<td>synapses</td>
</tr>
<tr>
<td>protein networks</td>
<td>proteins</td>
<td>protein-protein interactions</td>
</tr>
<tr>
<td>chemical compounds</td>
<td>molecules</td>
<td>bonds</td>
</tr>
</tbody>
</table>

Why study graph algorithms?
- Interesting and broadly useful abstraction.
- Challenging branch of computer science and discrete math.
- Hundreds of graph algorithms known.
- Thousands of practical applications.
Some Graph Problems

**Path.** Is there a path between s to t?
**Shortest path.** What is the shortest path between two vertices?
**Longest path.** What is the longest path between two vertices?

**Cycle.** Is there a cycle in the graph?
**Euler tour.** Is there a cyclic path 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 graph?

**Planarity.** Can you draw the graph in the plane with no crossing edges?
**Isomorphism.** Do two adjacency matrices represent the same graph?

Graph Interface

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Method</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>Graph(int V)</td>
<td>create empty graph</td>
</tr>
<tr>
<td>void</td>
<td>Graph(int V, int E)</td>
<td>create random graph</td>
</tr>
<tr>
<td>void</td>
<td>insert(int v, int w)</td>
<td>add edge v-w</td>
</tr>
<tr>
<td>Iterable&lt;Integer&gt;</td>
<td>adj(int v)</td>
<td>return iterator over neighbors of v</td>
</tr>
<tr>
<td>int</td>
<td>V()</td>
<td>return number of vertices</td>
</tr>
<tr>
<td>String</td>
<td>toString()</td>
<td>return string representation</td>
</tr>
</tbody>
</table>

Graph Representation

**Vertex representation.**
- This lecture: use integers between 0 and V-1.
- Real world: convert between names and integers with symbol table.

**Other issues.** Parallel edges, self-loops.

Set of Edge Representation

**Set of edge representation.** Store list of edges.

Graph G = new Graph(V, E);
System.out.println(G);
for (int v = 0; v < G.V(); v++)
    for (int w : G.adj(v))
        // edge v-w

---

iterate through all edges (in each direction)
Adjacency Matrix Representation

Adjacency matrix representation.
- Two-dimensional $V \times V$ boolean array.
- Edge $v$-$w$ in graph: $\text{adj}[v][w] = \text{adj}[w][v] = \text{true}$.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Adjacency List Representation

Vertex indexed array of lists.
- Space proportional to number of edges.
- Two representations of each undirected edge.

Adjacency Matrix Iterator

```java
private class AdjIterator implements Iterator<Integer>, Iterable<Integer> {
    int v, w = 0;
    AdjIterator(int v) { this.v = v; }

    public boolean hasNext() {
        // does v have another neighbor w?
        while (w < v) {
            if (adj[v][w]) return true;
            w++;
        }
        return false;
    }

    public Integer next() {
        if (hasNext()) return w++;
        else return -1;
    }
}
```

Adjacency List Representation: Java Implementation

```java
public class Graph {
    private int V; // number of vertices
    private boolean[][] adj; // adjacency matrix

    // empty graph with V vertices
    public Graph(int V) {
        this.V = V;
        this.adj = new boolean[V][V];
    }

    // insert edge v-w if it doesn't already exist
    public void insert(int v, int w) {
        adj[v][w] = true;
        adj[w][v] = true;
    }

    // return iterator for neighbors of v
    public Iterable<Integer> adj(int v) {
        return new AdjIterator(v);
    }
}
```
Adjacency List Representation: Java Implementation

```java
public class Graph {
    private int V;                   // # vertices
    private Sequence<Integer>[] adj; // adjacency lists

    public Graph(int V) {
        this.V = V;
        adj = new Sequence<Integer>()[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Sequence<Integer>();
    }

    // insert v-w, parallel edges allowed
    public void insert(int v, int w) {
        adj[v].add(w);
        adj[w].add(v);
    }

    public Iterable<Integer> adj(int v) {
        return adj[v];
    }
}
```

Graph Representations

**Graphs are abstract mathematical objects.**
- ADT implementation requires specific representation.
- Efficiency depends on matching algorithms to representations.

<table>
<thead>
<tr>
<th>Representation</th>
<th>Space</th>
<th>Edge between v and w?</th>
<th>Enumerate edges incident to v?</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of edges</td>
<td>O(E)</td>
<td>O(E)</td>
<td>O(V)</td>
</tr>
<tr>
<td>Adjacency matrix</td>
<td>O(V^2)</td>
<td>O(1)</td>
<td>O(V)</td>
</tr>
<tr>
<td>Adjacency list</td>
<td>O(E + V)</td>
<td>O(degree(v))</td>
<td>O(degree(v))</td>
</tr>
</tbody>
</table>

**Graphs in practice.** [use adjacency list representation]
- Real world graphs are sparse.
- Bottleneck is iterating over edges incident to v.

Maze Exploration

**Maze graphs.**
- Vertex = intersections.
- Edge = passage.

**Goal.** Explore every passage in the maze.
Trémaux Maze Exploration

Trémaux maze exploration.
- Unroll a ball of string behind us.
- Mark each visited intersection by turning on a light.
- Mark each visited passage by opening a door.

History. Theseus entered labyrinth to kill the monstrous Minotaur; Ariadne held ball of string.

Maze Exploration

Depth First Search

Goal. Find all vertices connected to s.

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

Running time. $O(E)$ since each edge examined at most twice.
Typical client program.

- Create a Graph.
- Pass the Graph to a graph processing routine, e.g., DFSearcher.
- Query the graph processing routine for information.
- Design pattern: separate graph from graph algorithms.

```
public class DFSearcher {
    private boolean[] marked;
    public DFSearcher(Graph G, int s) {
        marked = new boolean[G.V()];
        dfs(G, s);
    }
    // depth first search from v
    private void dfs(Graph G, int v) {
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w]) dfs(G, w);
    }
    public boolean isReachable(int v) { return marked[v]; }
}
```

Reachability Application: Flood Fill

Flood fill. Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.

- Vertex: pixel.
- Edge: two neighboring lime pixels.
- Blob: all pixels reachable from chosen lime pixel.

Paths

Path. Is there a path from s to t? If so, find one.
Paths

Path. Is there a path from s to t? If so, find one.

<table>
<thead>
<tr>
<th>Method</th>
<th>Preprocess Time</th>
<th>Query Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Union Find</td>
<td>O(E log* V) †</td>
<td>O(1)</td>
<td>O(V)</td>
</tr>
<tr>
<td>DFS</td>
<td>O(E + V)</td>
<td>O(1)</td>
<td>O(V + E)</td>
</tr>
</tbody>
</table>

† amortized

UF advantage. Can intermix query and edge insertion.

DFS advantage. Can recover path itself in same running time.

Find Path

```java
public class DFSearcher {
    // initialize pred[v] to -1 for all v

    private void dfs(Graph G, int v) {
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w]) {
                pred[w] = v;
                dfs(G, w);
            }
    }

    // return path from s to v
    public Iterable<Integer> path(int v) {
        LinkedList<Integer> list = new LinkedList<Integer>();
        while (v != -1 && marked[v]) {
            list.addFirst(v);
            v = pred[v];
        }
        return list;
    }
}
```

DFS Summary

Enables direct solution of simple graph problems.
- Find path between s to t.
- Connected components.
- Euler tour.
- Cycle detection.
- Bipartiteness checking.

Basis for solving more difficulty graph problems.
- Biconnected components.
- Planarity testing.
**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.

**Breadth first search.**
- Initialize \( \text{dist}[v] = \infty, \text{dist}[s] = 0 \).
- When considering edge \( v \to w \):
  - if \( w \) is marked, then ignore
  - otherwise set \( \text{dist}[w] = \text{dist}[v] + 1 \) and add \( w \) to the queue

**Property.** BFS examines vertices in increasing distance from \( s \).

**BFS applications.**
- Facebook.
- Kevin Bacon numbers.
- Fewest number of hops in a communication network.
Connected Components

Depth-first search.
- To traverse a graph G:
  - initialize all vertices as unmarked
  - visit each unmarked vertex v
- To visit a vertex v:
  - mark v as visited
  - recursively visit all unmarked vertices w adjacent to v

Result.
- Preprocessing: O(V + E) time, O(V) extra space.
- Connectivity query: O(1) time.

Connected Components

Connectivity Queries

Def. Vertices v and w are connected if there is a path between them.
Property. Symmetric and transitive.

Goal. Preprocess graph to answer queries: is v connected to w?
Brute force. Run DFS from each vertex v: quadratic time and space.

Connected component. Maximal set of mutually connected vertices.

Connected Components

Connected Components

Depth-First Search: Connected Components

```java
public class CCFinder {
    private int components;
    private int[] cc;

    public CCFinder(Graph G) {
        this.cc = new int[G.V()];
        for (int v = 0; v < G.V(); v++) cc[v] = -1;
        for (int v = 0; v < G.V(); v++)
            if (cc[v] == -1) dfs(G, v); components++; }

    // depth first search from v
    private void dfs(Graph G, int v) {
        cc[v] = components;
        for (int w : G.adj(v))
            if (cc[w] == -1) dfs(G, w);
    }

    public int connected(int v, int w) { return cc[v] == cc[w]; }
}
```
**Connected Components**

**Goal.** Read in a 2D color image and find regions of connected pixels that have the same color.

**Efficient algorithm.**
- Connect each pixel to neighboring pixel if same color.
- Find connected components in resulting graph.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>6</th>
<th>6</th>
<th>8</th>
<th>9</th>
<th>9</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>11</td>
<td>9</td>
<td>9</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>4</td>
<td>8</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>2</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Connected Components Application: Image Processing**

**Goal.** Read in a 2D color image and find regions of connected pixels that have the same color.

**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.
Euler Tour

How to find an Euler tour. [assuming graph is Eulerian]

- Start at vertex v and follow unused edges until you return to v. (always possible since all vertices have even degree)
- Find additional cyclic paths using remaining edges and splice back into original cyclic path.

Euler Tour: Implementation

Q. How to efficiently keep track of unused edges?
A. Quick + dirty: delete edge from graph once you use it.

Q. How to efficiently find and splice additional cyclic paths?
A. Push each visited vertex onto a stack.
Two Related Problems

**Euler tour.** Is there a cyclic path that uses each edge exactly once?
*Linear time solution.* DFS.

**Hamilton tour.** Is there a cycle that uses each vertex exactly once?
*Polynomial time solution??* NP-complete.