4.1 Undirected Graphs

- graph API
- maze exploration
- depth-first search
- breadth-first search
- connected components
- challenges

Graph. Set of vertices connected pairwise by edges.

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.

Protein interaction network

Reference: Jeong et al, Nature Review | Genetics

The Internet as mapped by the Opte Project

Map of science clickstreams

High-school dating

Kevin’s facebook friends (Princeton network)

One week of Enron emails

Reference: Bearman, Moody and Stovel, 2004
image by Mark Newman

### 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>

### Graph terminology

- **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 matrices represent the same graph?

- **Challenge.** Which of these problems are easy? difficult? intractable?
Graph representation

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

Issues. Parallel edges, self-loops.

Set of edges representation

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

Adjacency-matrix representation

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

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)  // return an iterator over the neighbors of v
int V()  // return number of vertices

In in = new In();
Graph G = new Graph(in);
for (int v = 0; v < G.V(); v++)
    for (int w : G.adj(v))
        /* process edge v-w */
```

Read graph from standard input:
```
% more tiny.txt
7
0 1
0 2
0 5
0 6
3 4
3 5
4 6
```

Adjacency-matrix representation

```java
0 1
0 2
0 5
0 6
3 4
5 5
4 6
7 8
9 10
9 11
9 12
```
Adjacency-matrix representation: Java implementation

```java
public class Graph {
    private final int V;
    private final boolean[][] adj;

    public Graph(int V) {
        this.V = V;
        adj = new boolean[V][V];
    }

    public void addEdge(int v, int w) {
        adj[v][w] = true;
        adj[w][v] = true;
    }

    public Iterable<Integer> adj(int v) {
        return new AdjIterator(v);
    }
}
```

Adjacency-list representation: Java implementation

```java
public class Graph {
    private final int V;
    private final 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-set (or adjacency-list) representation.
- Algorithms based on iterating over edges incident to v.
- Real-world graphs tend to be "sparse."

<table>
<thead>
<tr>
<th>representation</th>
<th>space</th>
<th>insert edge</th>
<th>edge between v and w?</th>
<th>iterate over edges incident to v?</th>
</tr>
</thead>
<tbody>
<tr>
<td>list of edges</td>
<td>E</td>
<td>E</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 list</td>
<td>E + V</td>
<td>1</td>
<td>degree(v)</td>
<td>degree(v)</td>
</tr>
<tr>
<td>adjacency set</td>
<td>E + V</td>
<td>log(degree(v))</td>
<td>log(degree(v))</td>
<td>degree(v)</td>
</tr>
</tbody>
</table>

* only if parallel edges allowed
Maze exploration

Maze graphs.
• Vertex = intersection.
• Edge = passage.

Goal. Explore every passage 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.

First use? Theseus entered labyrinth to kill the monstrous Minotaur; Ariadne held ball of string.

Claude Shannon (with Theseus mouse)
Maze exploration

‣ graph API
‣ maze exploration
‣ depth-first search
‣ breadth-first search
‣ connected components
‣ challenges

Goal. Systematically search through a graph.


Depth-first search

DFS (to visit a vertex s)
Mark s as visited.
Recursively visit all unmarked vertices v adjacent to s.

Challenge.
• Masks a complex recursive process.
• [stay tuned]

Typical applications.
• Find all vertices connected to a given s.
• Find a path from s to t.
Design pattern for graph processing

Design goal. Decouple graph data type from graph processing.

Typical client program.
- Create a Graph.
- Pass the Graph to a graph-processing routine, e.g., DFSearcher.
- Query the graph-processing routine for information.

Depth-first search (warmup)

Goal. Find all vertices connected to a given s.

Algorithm.
- Use recursion (ball of string).
- Mark each visited vertex
- Return (retrace steps) when no unvisited options.

Data structure
- boolean[] marked to mark visited vertices

Depth-first search (warmup) equivalent alternate version

public class DFSearcher
{
    private boolean[] marked;

    public DFSearcher(Graph G, int s)
    {
        marked = new boolean[G.V()];
        marked[s] = true;
        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);
    }

    public boolean isConnected(int v)
    {
        return marked[v];
    }
}

public class DFSearcher{   private boolean[] marked;
   public DFSearcher(Graph G, int s)   {      marked = new ... w);         {  marked[w] = true; dfs(G, w);  }   }
   public boolean isConnected(int v)   {  return marked[v];  }}
Flood fill

Photoshop "magic wand"

Graph-processing challenge 1

Problem. Flood fill.
Assumptions. Picture has millions to billions of pixels.

How difficult?
• Any COS 126 student could do it.
• Need to be a typical diligent COS 226 student.
• Hire an expert.
• Intractable.
• No one knows.
• Impossible.

Connectivity application: flood fill

Change color of entire blob of neighboring red pixels to blue.

Build a grid graph.
• Vertex: pixel.
• Edge: between two adjacent red pixels.
• Blob: all pixels connected to given pixel.

Connectivity application: flood fill

Change color of entire blob of neighboring red pixels to blue.

Build a grid graph.
• Vertex: pixel.
• Edge: between two adjacent red pixels.
• Blob: all pixels connected to given pixel.
Graph-processing challenge 2

**Problem.** Is there a path from \( s \) to \( t \)?

**How difficult?**
- Any COS 126 student could do it.
- Need to be a typical diligent COS 226 student.
- Hire an expert.
- Intractable.
- No one knows.

Graph-processing challenge 2A

**Problem.** Find a path from \( s \) to \( t \)?

**Assumption.** Any path will do.

**How difficult?**
- Any COS 126 student could do it.
- Need to be a typical diligent COS 226 student.
- Hire an expert.
- Intractable.
- No one knows.

Paths in graphs: union find vs. DFS

**Goal.** Is there a path from \( s \) to \( t \)?

If so, find one.
- Union-find: not much help (run DFS on connected subgraph).
- DFS: easy (see next slides).

**Union-find advantage.** Can intermix queries and edge insertions.

**DFS advantage.** Can recover path itself in time proportional to its length.

<table>
<thead>
<tr>
<th>method</th>
<th>preprocessing time</th>
<th>query time</th>
<th>space</th>
</tr>
</thead>
<tbody>
<tr>
<td>union-find</td>
<td>( V + E \log^* V )</td>
<td>( \log^* V \uparrow )</td>
<td>( V )</td>
</tr>
<tr>
<td>DFS</td>
<td>( E + V )</td>
<td>1</td>
<td>( E + V )</td>
</tr>
</tbody>
</table>

† amortized

Depth-first search (pathfinding)

**Goal.** Find paths to all vertices connected to a given \( s \).

**Idea.** Mimic maze exploration.

**Algorithm.**
- Use recursion (ball of string).
- Mark each visited vertex by keeping track of edge taken to visit it.
- Return (retrace steps) when no unvisited options.

**Data structure**
- `Integer[] edgeTo` instead of `boolean[] marked`
- `edgeTo[w] == null` means that \( w \) has not yet been visited
- `edgeTo[w] == v` means that edge \( v-w \) was taken to visit \( v \) the first time
public class PathfinderDFS {
    private Integer[] edgeTo;
    public PathfinderDFS(Graph G, int s) {
        edgeTo = new Integer[G.V()];
        edgeTo[s] = s;
        dfs(G, s);
    }
    private void dfs(Graph G, int v) {
        for (int w : G.adj(v))
            if (edgeTo[w] == null) {
                edgeTo[w] = v;
                dfs(G, w);
            }
    }
    public Iterable<Integer> pathTo(int v) {
        Stack<Integer> path = new Stack<Integer>();
        path.push(v);
        while (v != edgeTo[v]) {
            v = edgeTo[v];
            path.push(v);
        }
        return path;
    }
    // Stay tuned.
}

DFS pathfinding trace

(edgeTo[]) is a parent-link representation of a tree rooted at s

public Iterable<Integer> pathTo(int v) {
    Stack<Integer> path = new Stack<Integer>();
    path.push(v);
    while (v != edgeTo[v]) {
        v = edgeTo[v];
        path.push(v);
    }
    return path;
}
DFS summary

Enables direct solution of simple graph problems.
- Find path from s to t.
- Connected components (stay tuned).
- Euler tour (see book).
- Cycle detection (simple exercise).
- Bipartiteness checking (see book).

Basis for solving more difficult graph problems.
- Biconnected components (see book).
- Planarity testing (beyond scope).

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.

Property. BFS examines vertices in increasing distance from s.
BFS application

- Facebook.
- Kevin Bacon numbers.
- Fewest number of hops in a communication network.

Kevin Bacon graph

- Include vertex for each performer and movie.
- Connect movie to all performers that appear in movie.
- Compute shortest path from s = Kevin Bacon.

graph API
- maze exploration
- depth-first search
- breadth-first search
- connected components
- challenge
**Connectivity queries**

**Def.** Vertices v and w are connected if there is a path between them.

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

**Goal.** Preprocess graph to answer queries: is v connected to w? in constant time

![Graph illustration](image)

*Union-Find? Not quite.*

---

**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.*

<table>
<thead>
<tr>
<th>preprocess time</th>
<th>query time</th>
<th>extra space</th>
</tr>
</thead>
<tbody>
<tr>
<td>E + V</td>
<td>1</td>
<td>V</td>
</tr>
</tbody>
</table>

**Finding connected components with DFS**

```java
public class CCfinder {
    private Graph G;
    private Bag<Integer>[] connectedTo;
    private Bag<Integer> representatives;

    public CCfinder(Graph G) {
        this.G = G;
        representatives = new Bag<Integer>();
        connectedTo = (Bag<Integer>[])
                    new Bag[G.V()];
        for (int s = 0; s < G.V(); s++) {
            if (connected[s] == null) {
                Bag<Integer> bag = new Bag<Integer>();
                representatives.add(s);
                dfs(s, s, bag);
            }
        }
    }
}
```

**Finding connected components with DFS (data structures)**

![Data structures diagram](image)
Finding connected components with DFS (continued)

```java
private void dfs(int v, int s, Bag bag) {
    connectedTo[v] = bag;
    bag.add(v);
    for (int w : G.adj(v))
        if (connectedTo[w] == null)
            dfs(w, s, bag);
}
```

```java
public boolean connected(int v, int w) {
    return connectedTo[v] == connectedTo[w];
}
```

```java
public Iterable<Integer> representatives() {
    return representatives;
}
```

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

Tricky: object equality!

Connected components application: image processing

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

**Input.** Scanned image.

**Output.** Number of red and blue states.
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.

Graph-processing challenge 3

Problem. Find a cycle that uses every edge.
Assumption. Need to use each edge exactly once.

How difficult?
- Any COS 126 student could do it.
- Need to be a typical diligent COS 226 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 cyclic path that uses each edge exactly once?
Answer. Yes iff connected and all vertices have even degree.
To find path. DFS-based algorithm (see Algs in Java).
Graph-processing challenge 4

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

How difficult?
• Any COS 126 student could do it.
• Need to be a typical diligent COS 226 student.
• Hire an expert.
• Intractable.
• No one knows.
• Impossible.

Graph-processing challenge 5

Problem. Are two graphs identical except for vertex names?

How difficult?
• Any COS 126 student could do it.
• Need to be a typical diligent COS 226 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 COS 126 student could do it.
• Need to be a typical diligent COS 226 student.
• Hire an expert.
• Intractable.
• No one knows.
• Impossible.