4.1 Undirected Graphs

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-protein interaction network

Reference: Jeong et al, Nature Review | Genetics

The Internet as mapped by the Opte Project

http://en.wikipedia.org/wiki/Internet
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>chemical compound</td>
<td>molecule</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.

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.

Caveat. Intuition can be misleading.

Two drawings of the same graph

Graph API

public class Graph

create an empty graph with V vertices
create a graph from input stream
add an edge v-w
vertices adjacent to v
number of vertices
number of edges
string representation

In in = new In(args[0]);
Graph G = new Graph(in);
for (int v = 0; v < G.V(); v++)
    for (int w : G.adj(w))
        StdOut.println(v + "-" + w);

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

Anomalies.
- self-loop
- parallel edges

Graph representation

Graph representation. Provides intuition about the structure of the graph.

Caveat. Intuition can be misleading.

Two drawings of the same graph

Graph API

public class Graph

create an empty graph with V vertices
create a graph from input stream
add an edge v-w
vertices adjacent to v
number of vertices
number of edges
string representation

In in = new In(args[0]);
Graph G = new Graph(in);
for (int v = 0; v < G.V(); v++)
    for (int w : G.adj(w))
        StdOut.println(v + "-" + w);
Set-of-edges graph representation

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

Adjacency-matrix graph representation

Maintain a two-dimensional $V \times V$ boolean array; for each edge $v \rightarrow w$ in graph: $adj[v][w] = adj[w][v] = true$.

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

Typical graph-processing code

```
public static int degree(G, int v)
    { int degree = 0;
        for (int w : G.adj(v)) degree++;
        return degree;
    }

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;
    }

public static int avgDegree(Graph G)
    { return 2 * G.E() / G.V();
    }

public static int maxDegree(Graph G)
    { int max = 0;
        for (int v = 0; v < G.V(); v++)
            max = degree(G, v);
        return max;
    }

public static int degree(Graph G, int v)
    { for (int w : G.adj(v)) degree++;
        return degree;
    }
```

Read graph from input stream.

Print out each edge (twice).

Compute the degree of $v$.
Compute maximum degree.
Compute average degree.
Count self-loops.

In $tinyG.txt$.
```
1 0 0
0 0 1
0 0 0
```

```
% java Test tinyG.txt
1
0
0
0
0
1
0
0
0
1
...
Adjacency-list graph representation

Maintain vertex-indexed array of lists. (use `Bag` abstraction)

```

<table>
<thead>
<tr>
<th>v</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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 <code>v</code> and <code>w</code></th>
<th>iterate over vertices adjacent to <code>v</code></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>$V$</td>
<td>$V$</td>
</tr>
<tr>
<td>adjacency lists</td>
<td>$E + V$</td>
<td>$1$</td>
<td>degree(<code>v</code>)</td>
<td>degree(<code>v</code>)</td>
</tr>
</tbody>
</table>

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 <code>v</code> and <code>w</code></th>
<th>iterate over vertices adjacent to <code>v</code></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>$V$</td>
<td>$V$</td>
</tr>
<tr>
<td>adjacency lists</td>
<td>$E + V$</td>
<td>$1$</td>
<td>degree(<code>v</code>)</td>
<td>degree(<code>v</code>)</td>
</tr>
</tbody>
</table>

* disallows parallel edges
Maze exploration

Maze graphs.
- 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.

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

Claude Shannon (with Theseus mouse)
Maze exploration

Goal. Systematically search through a graph.


Typical applications.

• Find all vertices connected to a given source vertex.
• Find a path between two vertices.

Depth-first search

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

Typical client program. [ahead]

• Create a graph.
• Pass the graph to a graph-processing routine, e.g., search.
• Query the graph-processing routine for information.

Design pattern for graph processing

Design pattern. Decouple graph data type from graph processing.

public class Search
{
  find vertices connected to s
  boolean marked(int v)
  is vertex v connected to s?
  int count()
  how many vertices connected to s?
}

Typical client program.
Search search = new Search(G, s);
for (int v = 0; v < G.V(); v++)
  if (search.marked(v))
    StdOut.println(v);

print all vertices connected to s
Depth-first search (warmup)

Goal. Find all vertices connected to $s$.

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)

public class DepthFirstSearch
{
    private boolean[] marked;
    public DepthFirstSearch(Graph G, int s)
    {
        marked = new boolean[G.V()];
        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 marked(int v)
    {  return marked[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 application: flood fill

Challenge. Flood fill (Photoshop magic wand).
Assumptions. Picture has millions to billions of pixels.

Q. How difficult?

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.

Goal. Does there exist a path from $s$ to $t$?

 Paths in graphs
Goal. Does there exist a path from \( s \) to \( t \)?

<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 )</td>
<td>( V )</td>
</tr>
<tr>
<td>DFS</td>
<td>( E + V )</td>
<td>1</td>
<td>( E + V )</td>
</tr>
</tbody>
</table>

Union-find. Can intermix queries and edge insertions.

Depth-first search. Constant time per query.

Pathfinding in graphs

Goal. Does there exist a path from \( s \) to \( t \)? If yes, find any such path.

```
public class Paths

    public 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
```

Union-find. Not much help.

Depth-first search. After linear-time preprocessing, can recover path itself in time proportional to its length.

```
easy modification (stay tuned)
```

Pathfinding in graphs

Goal. Does there exist a path from \( s \) to \( t \)? If yes, find any such path.

```
Depth-first search (pathfinding)

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

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 structures.
- boolean[] marked to mark visited vertices.
- int[] edgeTo to keep tree of paths.
- (edgeTo[v] == w) means that edge \( v-w \) was taken to visit \( w \) the first time
```
Depth-first search (pathfinding)

```java
public class DepthFirstPaths {
    private boolean[] marked;
    private int[] edgeTo;
    private final int s;

    public DepthFirstPaths(Graph G, int s) {
        marked = new boolean[G.V()];
        edgeTo = new int[G.V()];
        this.s = s;
        dfs(G, s);
    }

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

    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 (pathfinding iterator)

`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 summary

- Enables direct solution of simple graph problems.
- Does there exists a path between `s` and `t`?
- Find path between `s` and `t`.
- Connected components (stay tuned).
- Euler tour (see book).
- Cycle detection (see book).
- Bipartiteness checking (see book).

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

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

**Breadth-first search (pathfinding)**

```java
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 properties**

**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.
Breadth-first search application: routing

Fewest number of hops in a communication network.

![ARPANET, July 1977](image)

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 \( s = \) Kevin Bacon.

Breadth-first search application: Kevin Bacon numbers

Kevin Bacon numbers.

[Image of Kevin Bacon numbers](image)

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

movies.txt
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.

```
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
}
```

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.

```
63 connected components
```

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

```
public class CC {
    private boolean marked[];
    private int[] id;
    private int count;
    public CC(Graph G) {
        ...         count++;         }      }
    public int count()   public int id(int v)   private void dfs(Graph G, int v)
    {  return count;  }
    {  return id[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()   public int id(int v)
    {                      }
    private void dfs(Graph G, int v)
    {        }
```

**Finding connected components with DFS (trace)**

```
  ddfs(0)          2   T T T T T T T T T T   T
check 7
check 6
check 0
dfs(4)          0   T T T T
check 3
check 4
check 0
dfs(5)          0   T T T T
check 6
dfs(3)          0   T T T T
check 5
dfs(0)
check 0
check 1
dfs(12)        2   T T T T T T T T T T   T
check 11
dfs(1)           0   T
check 0
dfs(0)             0
```

Input format for Graph

```
tinyG.txt
0 1 2 3 4 5 6 7 8
 13
0 5
4 3
0 1
9 12
6 4
5 4
0 2
11 12
9 10
0 6
7 8
9 13
5 3
```

Input format for sameecalleofedfsehaveesameeid

```
alleverticesediscoveredeineideofecomponentecontainingev
```

Input format for numbereofeconnectedecomponents

```
231 248
233 240
235 238
9 10
13
```

Finding connected components with DFS (continued)

```
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 with DFS (trace)

```
count        marked[]        id[]
 0 1 2 3 4 5 6 7 8 9 10 11 12
0 1 2 3 4 5 6 7 8 9 10 11 12
```

Number of connected components

```
ddfs(0)
  0 T T T T T T T T T T
check 7
check 6
check 0
dfs(4)          0   T T T T
check 3
check 4
check 0
dfs(5)          0   T T T T
check 6
dfs(3)          0   T T T T
check 5
dfs(0)
check 0
check 1
dfs(12)        2   T T T T T T T T T T   T
check 11
dfs(1)           0   T
check 0
dfs(0)             0
```
Finding connected components with DFS (trace)

Connected components application: study spread of STDs

Particle detection. Given grayscale image of particles, identify "blobs."
- **Vertex:** pixel.
- **Edge:** between two adjacent pixels with grayscale value \( \geq 70. \)
- **Blob:** connected component of 20-30 pixels.

Particle tracking. Track moving particles over time.

Connected components application: particle detection
Graph-processing challenge 1

Problem. Is a graph bipartite?

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.

Relationship graph at “Jefferson High”


Graph-processing challenge 2

Problem. Find a cycle.

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 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 (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 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.