4.1 Undirected Graphs

- introduction
- graph API
- depth-first search
- breadth-first search
- connected components
- challenges

With added notes and slides by Betty O’Neil for cs310
Breadth-first search algorithm

Create a queue and put the source vertex in it
Repeat until queue is empty:
• Remove vertex \( v \) from queue.
• Add to queue all unmarked vertices adjacent to \( v \) and mark them.

**graph G**

Queue for bfs from 0:
[0]
[5 2 1] after dequeue 0, enqueue adj(0): 1,2,5 and mark
[5, 2] after dequeue 1, nothing unmarked to enqueue
[4, 3 ,5] after dequeue 2, enqueue 3, 4, all marked now
Breadth-first search algorithm

Repeat until queue is empty:
• Remove vertex $v$ from queue.
• Add to queue all unmarked vertices adjacent to $v$ and mark them.
• Track first visits with edgeTo

\[
\begin{array}{c|c|c}
\text{v} & \text{edgeTo[]} & \text{distTo[]} \\
\hline
0 & - & 0 \\
1 & 0 & 1 \\
2 & 0 & 1 \\
3 & 2 & 2 \\
4 & 2 & 2 \\
5 & 0 & 1 \\
\end{array}
\]

• $\text{distTo}[x] = \# \text{ edges on path from 0 to } x$
• $= \text{distTo[from-node]} + 1$
Breadth-first search

Repeat until queue is empty:
- Remove vertex $v$ from queue.
- Add to queue all unmarked vertices adjacent to $v$ and mark them.

**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.
Breadth-first search: Java implementation

```java
public class BreadthFirstPaths {
    private boolean[] marked;
    private int[] edgeTo;
    private int[] distTo;
    
    private void bfs(Graph G, int s) {
        Queue<Integer> q = new Queue<Integer>();
        q.enqueue(s);
        marked[s] = true;
        distTo[s] = 0;
        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;
                    distTo[w] = distTo[v] + 1;
                }
            }
        }
    }
}
```

initialize FIFO queue of vertices to explore

found new vertex w via edge v-w
Breadth-first search properties

Q. In which order does BFS examine vertices?
A. Increasing distance (number of edges) from $s$: $v$ itself, all distance-1 vertices, all distance-2 vertices, ....

queue always consists of $\geq 0$ vertices of distance $k$ from $s$, followed by $\geq 0$ vertices of distance $k+1$

Proposition. In any connected graph $G$, BFS computes shortest paths from $s$ to all other vertices in time proportional to $E + V$. 
Breadth-first search application: routing

Fewest number of hops in a communication network.
Breadth-first search application: Kevin Bacon numbers

http://oracleofbacon.org
Kevin Bacon graph(page 549)

• 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 \( s = \text{Kevin Bacon} \).
• Data in movies.txt in \text{algs4-data.zip}
Breadth-first search application: Erdös numbers (mine is 2!)

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. Provide processed graph info by setting up an API...

API on page 543:

```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]
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.

| \( v \) | \( \text{id}[\] | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 2 | 0 | 3 | 0 | 4 | 0 | 5 | 0 | 6 | 0 |
| 7 | 1 | 8 | 1 | 9 | 2 | 10 | 2 | 11 | 2 | 12 | 2 |

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

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

\begin{table}
\begin{tabular}{|c|c|c|c|c|}
\hline
$v$ & marked[] & id[] \\
\hline
0 & F & - \\
1 & F & - \\
2 & F & - \\
3 & F & - \\
4 & F & - \\
5 & F & - \\
6 & F & - \\
7 & F & - \\
8 & F & - \\
9 & F & - \\
10 & F & - \\
11 & F & - \\
12 & F & - \\
\hline
\end{tabular}
\end{table}

graph G
Connected components algorithm

To visit a vertex $v$: do dfs from $v$:

- Mark vertex $v$ as visited.
- Recursively visit all unmarked vertices adjacent to $v$.

**graph G**

```

<table>
<thead>
<tr>
<th>v</th>
<th>marked[]</th>
<th>id[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>T</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>T</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>T</td>
<td>2</td>
</tr>
</tbody>
</table>
```
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];
    }

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

    private void dfs(Graph G, int v) {
        // DFS logic here
    }
}
```

- `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];
}

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

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);
}
```
Connected components application: study spread of STDs

Relationship graph at "Jefferson High"

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.
4.1 Undirected Graphs

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

Problem. Is a graph bipartite?

Definition, page 521: vertices can be divided into two groups such that all edges connect a vertex in one group with a vertex in the other group, i.e., you can “color the graph” in two colors.

How difficult?
Any programmer could do it.

✓ Typical diligent algorithms student could do it.

Hire an expert.

Intractable.

No one knows.

Impossible.

simple DFS-based solution (see textbook page 547)

\{ 0, 3, 4 \}
Bipartiteness application: is dating graph bipartite?

Image created by Mark Newman.
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.

Idea: if there are no cycles, the graph is tree-like and the dfs just keeps finding new vertices, never revisiting marked vertices other than its own parent. If there’s a cycle, the dfs will revisit some other vertex.

simple DFS-based solution (see textbook page 547)
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.

[Graphs showing isomorphism]

Graph isomorphism is an 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)
Graph traversal summary

BFS and DFS enables efficient solution of many (but not all) graph problems.

<table>
<thead>
<tr>
<th>problem</th>
<th>BFS</th>
<th>DFS</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>path between s and t</td>
<td>✓</td>
<td>✓</td>
<td>$E + V$</td>
</tr>
<tr>
<td>shortest path between s and t</td>
<td>✓</td>
<td></td>
<td>$E + V$</td>
</tr>
<tr>
<td>connected components</td>
<td>✓</td>
<td>✓</td>
<td>$E + V$</td>
</tr>
<tr>
<td>biconnected components</td>
<td></td>
<td>✓</td>
<td>$E + V$</td>
</tr>
<tr>
<td>cycle</td>
<td>✓</td>
<td>✓</td>
<td>$E + V$</td>
</tr>
<tr>
<td>Euler cycle</td>
<td></td>
<td>✓</td>
<td>$E + V$</td>
</tr>
<tr>
<td>Hamilton cycle</td>
<td></td>
<td></td>
<td>$2^{1.657 V}$</td>
</tr>
<tr>
<td>bipartiteness</td>
<td>✓</td>
<td>✓</td>
<td>$E + V$</td>
</tr>
<tr>
<td>planarity</td>
<td></td>
<td>✓</td>
<td>$E + V$</td>
</tr>
<tr>
<td>graph isomorphism</td>
<td></td>
<td></td>
<td>$2^{c \sqrt{V \log V}}$</td>
</tr>
</tbody>
</table>