4.1 Undirected Graphs

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.

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.
- Track first visits with `edgeTo`

Breadth-first search: Java implementation

```java
public class BreadthFirstSearch {
    private int[] edgeTo;
    private int[] distTo;
    private boolean[] marked;

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

Breadth-first search properties

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

Proposition: In any connected graph `G`, BFS computes shortest paths from `v` to all other vertices in time proportional to `V + E`.

Graph G

Distance from `v` to...

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

Queue queue consists of all vertices of distance 2 from `v`, followed by 2 vertices of distance 1.
Breadth-first search application: routing

Fewest number of hops in a communication network.

MBTA subway system: subject of pa4


Breadth-first search application: Kevin Bacon numbers

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 = Kevin Bacon.
- Data in movies.txt in data-algs4.zip

Breadth-first search application: Erdős numbers (mine is 2!)

hand-drawing of part of the Erdős graph by Ron Stocker

4.1 Undirected Graphs

- connected components
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
    }
}
```

Union-Find? Not quite.

**Depth-First search.** Yes. [next few slides]

Connected components

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

**Initiate 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.

**Connected components**

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

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

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.
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) {
        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.
- **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.

4.1 **Undirected Graphs**

- Component:
  - Connected vertices.
  - Challenges:
    - Simple DFS-based solution (see textbook page 547)

- **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.

Graph-processing challenge 1

Simple DFS-based solution tree (see textbook page 547)
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.

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.

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 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</td>
<td>✓</td>
<td>✓</td>
<td>$E + V$</td>
</tr>
<tr>
<td>connected</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>bipartiteness</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>2 $O(MV)$</td>
<td>$E + V$</td>
<td></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>$O(V \log V)$</td>
</tr>
</tbody>
</table>